Programmation parallèle orientée objet et réutilisabilité appliquée à l'algèbre linéaire

Eric Noulard

Programmation parallèle orientée objet et réutilisabilité appliquée à l'algèbre linéaire

THÈSE

présentée et soutenue publiquement le 5 Décembre 2000

pour l'obtention du

Doctorat de l'université de Versailles St-Quentin-en-Yvelines

(spécialité informatique)

par

Eric Noulard

Composition du jury

Président :	William Jalby	Professeur, UVSQ, Versailles
Rapporteurs :	Yousef Saad	Professeur, University of Minnesota, Minneapolis
	Jean-Marc Jézequel	Professeur, Université de Rennes 1, Rennes
Examinateurs :	Denis Caromel	Professeur, Université de Nice, Sophia-Antipolis
	Nahid Emad	Maître de Conférences, UVSQ, Versailles
	Paul Feautrier	Professeur, UVSQ, Versailles

Remerciements

Je remercie **Nahid EMAD**, maître de conférences à l'UVSQ d'avoir accepté d'encadrer ma thèse ainsi que **Paul FEAUTRIER**, Professeur à l'UVSQ d'avoir accepté d'être mon directeur de thèse.

Je remercie **Laurence FLANDRIN** et **Jean-Yves CHATELIER** de la Société ADULIS d'avoir également encadré mon travail. Leurs encouragements, leurs remarques ainsi que leur écoute plusieurs fois renouvelés m'ont été très bénéfiques.

Je tiens à remercier **Jean-Marc JEZEQUEL**, Professeur à l'Université de Rennes 1, d'avoir accepté et pris le temps de rapporter cette thèse. Les remarques dont il m'a fait part, m'ont permis de connaître des références importantes et d'améliorer la qualité du rapport.

Je tiens à remercier **Yousef SAAD**, Professeur à l'Université du Minnesota, d'avoir accepté et pris le temps de rapporter cette thèse. Je le remercie doublement car il m'avait également accueilli, dans son équipe aux Etats-Unis, parmi les bagages d'un chercheur invité, avant le début de ma thèse.

Je remercie sincèrement **Denis CAROMEL**, à la fois pour avoir accepté de faire partie du jury, mais aussi pour la collaboration que nous avons eue autour du mécanisme *SharedOnRead* pour C++//. J'ai appris beaucoup de choses intérressantes lors de cette collaboration avec lui et **David SAGNOL**, que ce soit lors de ma visite à Sophia-Antipolis, ou par la suite, lors de nos contacts électroniques ou téléphoniques.

Je remercie donc tout autant **David SAGNOL**, avec qui se fût un plaisir de travailler. Il a du subir les bugs du second prototype de ma bibliothèque et répondait toujours présent à mes questions concernant C++//. Sans lui le mécanisme *SharedOnRead* n'aurait pas vu le jour.

Je remercie **William JALBY**, Professeur à l'UVSQ et Directeur du Laboratoire PRiSM d'avoir accepté de présider le jury. Je lui dois aussi un grand merci pour avoir participé, par son approbation, à l'obtention de ma dispense de DEA, car cela a grandement facilité mon travail et mon emploi du temps durant ma première année de thèse.

Je remercie la **Société ADULIS**, d'avoir accepté cette convention CIFRE qui m'a permis de faire cette thèse tout en vivant en milieu industriel. Je remercie par la même occasion, toutes les personnes d'ADULIS avec qui j'ai eu l'occasion de travailler, de discuter du travail et du reste et de boire de nombreux cafés, plus particulièrement **Agnès**, **Annick**, **Arnaud**, **Christine**, **Cyril**, **Florence**, **Jean-Marc** (L. et K.), **Jean-Yves**, **Laurence**, **Maude**, **Olivier** et **Sabine**.

Un merci spécial à **François** qui a toujours répondu présent à mes demandes et a notablement contribué à l'amélioration de mes conditions techniques de travail durant ma dernière année de thèse. Ses compétences techniques m'ont également permis d'apprendre beaucoup à son contact.

Je remercie le laboratoire **PRiSM** et plus particulièrement les membres de l'équipe du monastère du 4$^{\text{ième}}$, de l'accueil qu'ils m'ont accordé lors de mes passages plus ou moins réguliers au PRiSM. Un merci spécial à **Chantal DUCOIN** et **Isabelle MOUDENNER** qui ont été d'une efficacité redoutable concernant mes diverses démarches « secrétario-administratives ».

Je remercie **Ani** d'avoir utilisé ma bibliothèque pour son stage de DEA et de m'avoir

subi comme co-responsable de stage.

Je remercie l'**I**nstitut du **D**éveloppement et des **R**essources en **I**nformatiques **S**cientifique de m'avoir permis développer et tester ma bibliothèque sur le T3E.

Je remercie tous mes amis qui par leur sempiternelle question « *Euh au fait? Tu en es où de ta thèse* ? », ont certainement contribué à la faire avancer. Je remercie plus particulièrement Eric et Nicolas, qui m'ont fait grand plaisir en se libérant de leurs obligations pour assister à ma soutenance.

Finalement je remercie toute ma famille, et plus particulièrement pour l'aide logistique importante qu'ils ont déployée les quelques jours précédant ma soutenance.

Il y aurait pu y avoir des non-merci, mais je préfère laisser à Cyrano le soin de les servir.

Je ne remercie ni **Caroline**, ni **Louis**, pour cette thèse car je préfère le faire tous les jours de l'année, pour la joie et l'amour qu'ils me procurent quotidiennement et qui dépassent largement ce travail de thèse.

À Louis, Caroline
et tous ceux que j'aime et que j'aimerai.

Table des matières

Table des figures

Introduction

Les domaines d'application du calcul scientifique sont nombreux comme par exemple la chimie, l'électromagnétisme, les études énergétiques, la simulation automobile, la mécanique des fluides, ou bien encore la météorologie... Quel que soit le domaine les préoccupations principales sont souvent les mêmes : traiter des problèmes de taille de plus en plus importante et obtenir le plus rapidement possible le résultat en utilisant des moyens de calcul performants. Les spécialistes de ces domaines ont donc naturellement recours à la programmation parallèle. L'aspect le plus souvent négligé est l'ingénierie des codes de calculs parallèles, notamment leur réutilisabilité et leur perennité. Le cycle de vie des machines parallèles et de leurs moyens de programmation associés étant relativement courts, les applications sont développées puis redéveloppées sans cesse.

Dans le domaine de l'informatique, les langages et les paradigmes de programmation, notamment avec l'avènement de la programmation orientée-objet, arrivent à maturité. Cette maturité permet au programmeur de peu se soucier de l'architecture de la machine qu'il utilise et de se concentrer, avec un langage de haut niveau, sur son domaine d'application. De plus, les méthodologies orientées-objet permettent de construire des applications de façon incrémentale en s'appuyant sur et en réutilisant des programmes développés auparavant ou bien simultanément par d'autres personnes. La réutilisation de programmes est désormais une réalité industrielle, que ce soit dans le domaine des interfaces graphiques ou des noyaux de calculs spécialisés ou généralistes.

L'objectif de cette thèse est d'examiner comment les technologies orientées-objet peuvent apporter aux applications scientifiques tout ce qu'elles ont apporté dans la programmation des machines séquentielles : une meilleure réutilisabilité et perennité des codes, des démarches méthodologiques de conception et de réalisation claires, la possibilité de modulariser les développements... La contrainte supplémentaire et incontournable dans le domaine du calcul scientifique parallèle étant de ne pas sacrifier les performances.

Nous examinons donc dans un premier temps, au chapitre 1, les caractéristiques de la programmation parallèle puis celles de la programmation orientée-objet. Nous rappelons quels sont les moyens d'unification du parallélisme et de la programmation orientée-objet, au travers de différents langages à objets parallèles. Nous concluons ce chapitre en nous fixant les objectifs de travail de cette thèse en terme de développement et maintenance des applications parallèles.

Nous étudions, au chapitre 2, le domaine d'application que nous avons choisi, à savoir, la résolution de problèmes d'algèbre linéaire de grande taille. Nous verrons pourquoi ces problèmes nécessitent l'utilisation de machines parallèles et comment leur résolution informatique pourrait bénéficier d'une approche orientée-objet.

Les chapitres 3 et 4 examinent deux approches orientées-objet pour la réalisation d'une librairie parallèle orientée-objet offrant les briques de bases pour la résolution de nos problèmes. Chacune a posé des problèmes soit de performances soit de réutilisabilité que nous avons résolus.

Le chapitre 3 montre les avantages et les inconvénients de l'utilisation d'un modèle à objets actifs. Nous proposons notamment une optimisation de ce modèle permettant de surmonter un problème d'efficacité parallèle lié au modèle de base. Les résultats présentés dans ce chapitre ont fait l'objet d'une communication dans une conférence nationale et d'une communication dans une conférence internationale.

Au cours du chapitre 4, nous montrons, pas à pas, les limites, en termes de réutilisabilité, des fonctionnalités classiques des langages à objets, telles qu'elles sont mises en œvre dans le langage C++. Nous indiquons ensuite comment la généricité et de nouvelles approches telle que la programmation générative offrent des solutions satisfaisantes. Nous introduisons les notions de matrices avec formes et de formes de matrices qui sont les mécanismes permettant la bonne réutilisabilité de nos applications dans le domaine de l'algèbre linéaire. Les résultats présentés dans ce chapitre ont fait l'objet d'une communication dans une conférence internationale et d'une publication dans un journal international.

Nous proposons finalement au chapitre 5 l'intégration des contraintes de la programmation parallèle à une méthodologie de conception orientée-objet existante, en tirant les enseignements des chapitres précédents. Cette dernière étape constituera certainement le point de départ d'une méthodologie de conception permettant une programmation parallèle réutilisable et efficace.

Chapitre 1

Langages à objets et parallélisme

Sommaire

1.1 Introduction

Notre but principal est d'évaluer l'intérêt et la faisabilité d'une approche objet pour des applications de calcul scientifique parallèles. Cette étude nous amène à explorer conjointement deux domaines :

- la programmation orientée objet (POO),
- le parallélisme.

Nous précisons dans ce chapitre les principaux concepts de la POO et du parallélisme. Il ne s'agit en aucun cas d'un état de l'art de ces deux domaines mais plutôt d'une introduction nécessaire à la compréhension des interactions entre *l'approche objet*

et le *parallélisme*. Une couverture plus exhaustive de l'approche objet est présentée dans [112] et dans les références contenues dans ce livre. Pour une introduction générale sur le parallélisme et ses mécanismes nous référons aux ouvrages [45, 72, 46].

Au cours de ce chapitre nous souhaitons montrer l'intérêt d'une démarche orientée-objet pour la programmation d'applications parallèles. Il est communément admis qu'il est plus simple de programmer en C, C++ ou Ada qu'en Assembleur et l'avènement de la programmation orientée-objet avec Java et C++ va dans ce sens. La POO est plus simple, plus proche du langage courant car elle permet d'identifier les objets réels aux objets informatiques. Nous rappelons donc tout d'abord au paragraphe 1.2, ce qui caractérise cette différence entre le fonctionnement d'un ordinateur et la façon dont on le programme. Si cette distinction est d'abord apparue dans la programmation des machines séquentielles elle devrait également s'appliquer à la programmation des machines parallèles. Nous essaierons donc lors de notre présentation des concepts du parallélisme au paragraphe 1.3, de présenter de façon différenciée les modèles d'exécution et de programmation des machines parallèles. Nous rappellerons au paragraphe 1.4 ce qui caractérise le modèle de programmation orienté-objet et nous examinerons ensuite au paragraphe 1.5 des exemples de langages à objets parallèles qui associent POO et parallélisme. Finalement nous exposerons au paragraphe 1.6 la problématique de développement et de maintenance des applications parallèles de calculs scientifiques.

1.2 Modèle de programmation *vs* modèle d'exécution

Le modèle d'exécution d'un ordinateur est son mode de fonctionnement physique. Par exemple, le modèle d'exécution de nos PC de bureau est le modèle de *Von Neumann* [29, voir aussi modèle RAM] : la mémoire contient le programme et les données (d'entrée et de sortie), le processeur lit une instruction en mémoire, la décode et l'exécute de façon consécutive et l'opération est répétée. Le modèle de programmation est le mode de fonctionnement du langage utilisé par le programmeur. Pour ce même ordinateur ayant un modèle d'exécution de *Von Neumann*, les langages de programmation actuels offrent différents modèles de programmation. Cette distinction entre modèle d'exécution et modèle de programmation peut être résumée par l'équation (1.1) introduite par *L. Bougé* [22].

$$Calcul\ sur\ Ordinateur = Ordinateur + Programme$$

$$Calcul\ Abstrait = Modèle\ d'Exécution + Modèle\ de\ Programmation \tag{1.1}$$

On peut d'ailleurs remplacer la première équation par l'équation suivante :

$$Calcul = Support\ d'Exécution + Programme \tag{1.2}$$

Le programmeur écrit son programme dans un langage (`LPROG`) qui offre un *modèle de programmation* et il fera appel à un *traducteur* qui transcrira le programme dans le langage (`LEXEC`) du *support d'exécution* qui suit un *modèle d'exécution*, comme indiqué à la figure 1.1.

L'équation (1.2) signifie également que la différence entre modèle d'exécution et modèle de programmation reste vraie quel que soit le support d'exécution choisi : un ordinateur, un modèle théorique de machine, une machine virtuelle, une librairie,...

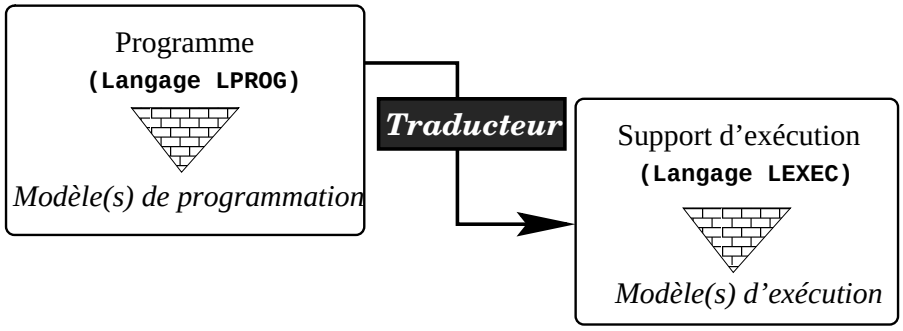

FIG. 1.1 – *Modèle de programmation / Modèle d'exécution*

Le *modèle de programmation* est le niveau d'abstraction fournit par le *traducteur* au programmeur d'applications. Plus le traducteur est « intelligent » plus le modèle de programmation s'éloigne du modèle d'exécution, et ainsi les fonctionnalités du langage d'exécution `LEXEC` se retrouveront moins directement (voire pas du tout) dans le langage de programmation `LPROG`. Une liste *non-exhaustive* de langages et de modèles de programmation associés est présentée dans le tableau 1.1. Certains langages, comme C++ ou O´CAML, supportent plusieurs modèles de programmation, dans ce cas nous ne notons que le(s) modèle(s) le(s) plus couramment utilisé(s) avec ces langages.

Pour les applications de calculs scientifiques, le modèle de programmation le plus employé actuellement est la programmation impérative structurée. Des langages comme Fortran 77 ou C supportent ce modèle de programmation. Les compilateurs de ces langages sont les traducteurs qui permettent l'exécution sur des machines séquentielles de type *Von Neumman*.

Cette séparation, entre modèle d'exécution et de programmation, montre une certaine maturité des modèles de programmation séquentiels qui facilite le travail du développeur d'applications. Il peut choisir le modèle de programmation le mieux adapté à son problème et laisser le soin au traducteur de faire son travail de transcription vers le modèle d'exécution de la machine. Le cas le plus courant de traducteur est celui du compilateur qui traduit le code d'un langage de programmation en du code machine, éventuellement celui d'une machine virtuelle si l'on prend l'exemple de Java[1].

1.2.1 Critères de choix d'un modèle de programmation

Lors du choix d'un modèle de programmation et du langage associé différents critères peuvent être pris en compte, parmi lesquels :

1. Cycle de vie du logiciel : tout ce qui concerne un programme de sa création jusqu'à son évolution en passant par la maintenance.
   (a) rapidité de développement : le modèle de programmation doit être simple et le `LPROG` associé fournir des fonctions de haut niveau qu'il est inutile de reprogrammer,

1. `http://java.sun.com/`

Langage LPROG	**Modèles de programmation**
Assembleur	programmation suivant le modèle d'exécution de la machine utilisée
C, Cobol, Fortran 77	programmation impérative et structurée
Ada, Fortran 90, Modula, Pascal	programmation impérative structurée et modulaire
C++, Eiffel, Java, O´CAML, Smalltalk	programmation orientée-objet
CAML, Lisp, ML, Haskell, Scheme	programmation fonctionnelle
Prolog	programmation déclarative
Intentional Programming [92]	programmation intentionnelle [47, §6.4.3]
AspectJ [11]	programmation par aspect [7]
C++ (avec template), Ada (generic), GJ [80]	programmation générique
C++ (avec template metaprogramme)	programmation générative [60] [47, Chap. 8]

TAB. 1.1 – *Langages et modèles de programmation associés*

(b) facilité de maintenance : une autre personne que le développeur initial du programme doit pouvoir effectuer des corrections facilement,

(c) possibilités d'évolution et de modification : le modèle et le langage de programmation doivent autoriser l'évolution du programme sans nécessiter une réécriture complète ou partielle lors d'une modification.

2. Consommation des ressources : les principales ressources utilisées par un programme sont le temps d'exécution, et les moyens de stockage (mémoire vive, disque etc...). Ces ressources peuvent coûter cher ou bien être limitées par la configuration disponible. Le LPROG ne doit pas augmenter la complexité en temps et en espace intrinsèque de l'application.

   (a) temps d'exécution : le traducteur du LPROG doit générer un code qui s'exécute le plus rapidement possible sur les architectures ciblées,

   (b) consommation mémoire : la machine disponible peut être limitée en mémoire ou l'application peut demander beaucoup de mémoire. Le modèle de programmation doit permettre de minimiser cette utilisation et le traducteur doit également générer un code qui soit économe en mémoire.

3. Portabilité : on doit pouvoir facilement compiler/exécuter le programme sur des machines différentes. Cela peut signifier qu'un traducteur du LPROG utilisé doit exister sur un grand nombre d'architectures. On peut remarquer ici l'intérêt d'un support d'exécution virtuel comme la machine virtuelle Java (JVM) qui assure une certaine portabilité en changeant le couple LPROG/LEXEC.

4. Réutilisabilité : le programme doit pouvoir être réutilisé (totalement ou partielle-

ment) facilement au sein d'une autre application. Cette caractéristique est liée à la fois au cycle de vie du logiciel et à la portabilité.

5. Méthodes et Outils : des méthodes et outils associés au modèle de programmation doivent exister et augmenter la productivité et la qualité des programmes développés. La disponibilité de méthodes et d'outils permet également de garantir une certaine qualité tout le long du cycle de vie du logiciel.
   (a) Méthode de conception : des méthodes sont associées à certains modèles de programmation par exemple comme les méthodes de conception orientée-objet (OOSE, OMT, Fusion, Demeter, ...)
   (b) Environnement de développement : des outils doivent permettre d'appliquer de façon pratique les méthodes de conception et de programmation. Ils peuvent regrouper des fonctionnalités aussi diverses que : la conception visuelle, l'édition de code, la compilation, le débogage, la génération de la documentation du programme, la visualisation des résultats...

Les critères énoncés ci-dessus sont des critères généraux et chaque domaine d'application aura ses propres contraintes. Il est naturel de choisir un `LPROG` de haut niveau qui fournisse les abstractions les plus proches de l'application. Si l'algorithme du programme s'exprime facilement de façon récursive, il sera certainement plus aisé de l'implanter avec un langage fonctionnel (voir tab. 1.1). Toutefois, le programmeur doit tenir compte de toute les contraintes liées à son domaine d'applications et s'attacher à trouver le meilleur compromis.

1.2.2 Calcul scientifique et efficacité

Une contrainte forte des applications de calculs scientifiques parallèles est l'efficacité. Concrètement, un programme de prévisions météorologiques ne peut mettre plus d'une journée à calculer le bulletin météorologique du lendemain. Si deux versions de ce programme sont écrites l'une en C++, en suivant un modèle de programmation objet et l'autre en Fortran, en suivant un modèle de programmation impérative structurée, des différences de performances sont à attendre. La qualité/l'intelligence du compilateur peut être la source de ce problème. En effet, les technologies de compilations ont considérablement évoluées et un bon compilateur Fortran 77 génère, *si le code est bien écrit*, un code machine aussi bon voire meilleur [207, 51] qu'un code machine optimisé manuellement. Ceci est vrai car les compilateurs de ces langages ont 10–20 ans d'expérience en matière d'optimisation de code pour les couples `LPROG`/`LEXEC` concernés. Les compilateurs de langages offrant un modèle de programmation plus récent tels que C++ ou Java n'ont pas toujours d'aussi bonnes performances. Mais des travaux [163, 204, 200] ou projets [95] visant à résoudre ces problèmes laissent présager de bonnes performances futures.

Si l'on prend le cas du C++, les techniques de programmation [203] qui rendent des applications écrites en C++ compétitives [202, 201] avec les mêmes applications écrites en Fortran sont compliquées et pas toujours supportées par les compilateurs actuels. De plus, ce sont des *techniques* de *programmation*, ce qui signifie que leur utilisation est à la charge du programmeur et non du compilateur. Ce dernier ne fournit qu'un support à l'implémentation de ces techniques. Nous pensons que ces problèmes d'optimisations

sont liés non seulement à « l'intelligence du compilateur » mais également au modèle de programmation lui-même qui n'est pas assez riche pour permettre au compilateur d'effectuer les optimisations que peut faire le programmeur. Les approches de programmation par aspect [7, 101] et plus généralement de programmation générative [60, 47] offrent des solutions à ce problème. Nous reviendrons plus en détail sur ces aspects au paragraphe 5.5 qui présente des démarches de conception et de programmation qui vont *au-delà* de l'orienté-objet.

Le choix d'un modèle de programmation pour des machines parallèles est une tâche encore plus difficile, car la séparation entre modèle d'exécution et de programmation n'est pas encore claire [22]. Nous nous efforcerons dans notre présentation des concepts du parallélisme de faire cette différence.

1.3 Les concepts du parallélisme

Nous appelons machine parallèle toute machine dont le modèle d'exécution est un des modèles d'exécution parallèle que nous présentons au paragraphe 1.3.1 *ou* que l'on programme au moyen d'un des modèles de programmation parallèle présentés au paragraphe 1.3.2. On peut d'ores et déjà noter que la définition d'une machine parallèle peut varier suivant le point de vue. Admettons qu'il existe un couple `LPROG`/`LEXEC` pour lequel `LPROG` supporte un modèle de programmation séquentiel et `LEXEC` supporte un modèle d'exécution parallèle. Le programmeur d'applications considérera la machine comme une machine séquentielle alors que celui qui programme le traducteur considérera la même machine comme une machine parallèle.

Nous présentons aux paragraphes 1.3.1 et 1.3.2 des modèles d'exécution et de programmation de machines parallèles qui illustrent la diversité et la complexité d'utilisation de telles machines. Nous verrons ensuite au paragraphe 1.3.3 des moyens de programmation parallèles supportant différents couples `LPROG`/`LEXEC`.

1.3.1 Les modèles d'exécution parallèle

Présenter des modèles d'exécution parallèle revient en quelque sorte à établir une classification des machines parallèles. De nombreuses classifications ont été proposées, les critères de classement dépendant des objectifs de l'étude : la compilation [64, §4], le coût théorique d'exécution d'un programme [29, voir PRAM] , l'architecture [69], la modélisation/simulation d'architecture [49]. Notre classification, inspirée de celle de Flynn [69], est volontairement simple et non exhaustive. Elle a uniquement pour but d'exposer les différences fondamentales de fonctionnement des machines parallèles actuelles [197, 194, 209] qui ont guidé le développement des différents modèles de programmation de ces machines.

La première distinction entre les différents modèles d'exécution parallèle porte sur le nombre de flots d'instructions : simple (SIMD) ou multiple (MIMD). Les machines citées

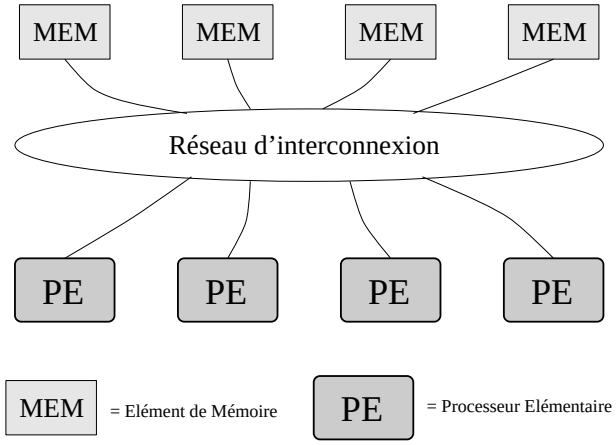

FIG. 1.2 – *Une machine parallèle à mémoire partagée*

en exemple sont décrites avec plus de détails dans [197][2].

DÉFINITION **1.1 (SIMD)** ***Single Instruction stream Multiple Data stream***
Une machine SIMD exécute chaque instruction d'un programme de façon identique sur chaque processeur mais sur des données qui sont locales au processeur. Les machines fonctionnant suivant ce modèle possèdent en général un grand nombre ($\geq$ 1024) de processeurs peu puissants.

Exemples : Aliena Quadrics, CPP Gamma II, MASPAR MP-x, TMC CM-2(00).

DÉFINITION **1.2 (MIMD)** ***Multiple Instruction stream Multiple Data stream***
Une machine MIMD peut exécuter des programmes différents sur des processeurs différents. Chacun des programmes possède ses propres données.

Exemples : Compaq/DEC GS Series, NEC SX-5 Series, SGI/Cray T90, SGI/Cray SV1, SUN E10000 (mémoire partagée)
IBM SP Series, SGI/Cray T3E, TMC CM-5, un NOW voir Déf. 1.6 (mémoire distribuée)
SGI Origin2000, Tera MTA (cc-NUMA voir Déf. 1.5)

Dans les définitions précédentes il convient de préciser comment les unités de traitement (processeur ou PE) accèdent aux données. C'est la distinction entre *mémoire distribuée* (ou DM pour **D**istributed **M**emory) et *mémoire partagée* (ou SM pour **S**hared **M**emory) qui est schématisée aux figures 1.3 et 1.2.

Dans une machine parallèle à mémoire partagée, tous les processeurs accèdent à la mémoire de façon identique. Comme les processeurs jouent généralement des rôles symétriques pour ce genre d'architectures, ces machines sont aussi qualifiées de SMP :

DÉFINITION **1.3 (SMP)** ***Symmetrical Multi-Processors*** *ou* ***Shared Memory multi-Processors systems***
Se dit d'une machine parallèle ayant plus de 2 processeurs qui partagent la même mémoire. Les processeurs jouent des rôles symétriques.

2. Ce rapport est mis à jour périodiquement

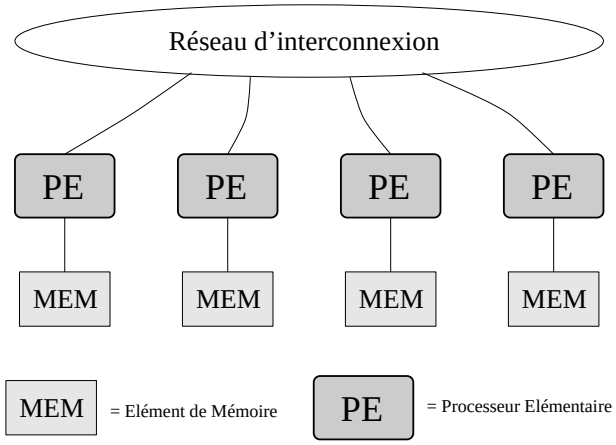

FIG. 1.3 – *Une machine parallèle à mémoire distribuée*

Exemples : Compaq/DEC GS Series, SGI/Cray T90, Sun E10000

Dans une machine à mémoire distribuée un processeur n'accède directement qu'à sa mémoire locale. Pour accéder au reste de la mémoire il doit dialoguer avec les autres processeurs, généralement par envoi de messages. Les machines à mémoire distribuée ayant généralement un plus grand nombre de processeurs que les machines SMP on les qualifie souvent de MPP :

DÉFINITION **1.4 (MPP)** ***Massively Parallel Processing systems*** *ou* ***Massively Parallel multi-Processors systems***
Se dit d'une machine ayant un nombre « important » de processeurs. Il y a encore quelques années on entendait par parallélisme massif plusieurs dizaines de milliers de processeurs. Actuellement le terme MPP concerne plutôt les machines parallèles à mémoire distribuée ayant quelques centaines ou quelques milliers de processeurs.

Exemples : Fujitsu AP3000, IBM SP Series, SGI/Cray T3E, CM-5.

Cette différence, en matière d'accès à la mémoire, dans les modèles d'exécution conduit à différents modèles de programmation que nous verrons au paragraphe 1.3.2. Cette relation montre le lien [trop] étroit qui existe entre les modèles d'exécution et de programmation parallèles. Toutefois, certaines architectures offrent des solutions matérielles ou logicielles pour gommer la différence entre les machines DM et SM, c'est le modèle mémoire NUMA :

DÉFINITION **1.5 (NUMA)** ***Non-Uniform Memory Access***
Certaines machines à mémoire [physiquement] distribuée offre un mécanisme de mémoire virtuellement partagée *qui permet un adressage global de la mémoire. L'adressage global est réalisé de façon matérielle ou logicielle. Ceci signifie que la machine, le système, le compilateur ou la librairie, à laquelle on fait appel génère lui-même les communications nécessaires aux accès à la mémoire [distribuée]. Ce fait se traduit généralement par un adressage de la mémoire pouvant être non-uniforme, i.e. la recherche d'une donnée en mémoire ne se fait pas en temps constant. On parle également de systèmes* ***DSM*** *[159, 58]*

*pour **D**istributed **S**hared **M**emory*[3]*. Lorsque le partage de la mémoire est assuré par un protocole de cohérence de cache [46] on qualifie ces machines de **cc-NUMA** pour **c**ache **c**oherent NUMA.*

Exemples : HP Exemplar V2500, KSR2, SGI Origin2000 (cc-NUMA), tout autre système DSM.

Même si la catégorie des machines NUMA-MIMD est certainement la plus prometteuse, car elle simplifie l'utilisation du parallélisme elle est largement réservée à des utilisateurs fortunés. Les machines parallèles ayant le meilleur rapport coût performance à l'heure actuelle sont probablement les réseaux de stations :

DÉFINITION **1.6 (NOW)** ***N**etwork **O**f **W**orkstations*

Se dit d'une machine DM-MIMD qui consiste en un réseau de machines reliées par un réseau de communication ([Fast|Gigabit]-ethernet, ATM , HiPPI , Myrinet, SCI, ...). Les différences principales entre ce type de machine et une machine MIMD classique est que le réseau d'interconnexion entre les unités de calculs peut être moins performant et que les unités de calcul (ici les machines) sont possiblement hétérogènes. Quand l'intégration réseau de ces machines fait l'objet d'une technique particulière on les qualifie également de cluster *[26].*

Exemples : Compaq/DEC GS Series Cluster, Fujitsu AP3000, Fujistu VPP700, SGI Power Challenge Array (SGI), un quelconque réseau homogène ou hétérogène de machines.

Ces technologies de clustering vont de la simple mise en réseau de machines (NOW) pour aller jusqu'à la construction de machines parallèles très performantes, des exemples de la réussite de ce type d'architecture sont les machines issues du projet américain ASCI [10]. Ce projet avait pour objectif de construire des machines parallèles dépassant le TeraFlops (10^{12} instructions virgule flottante par seconde). La plus puissante à l'heure actuelle [194] est l'option ASCI-*rouge* [4] fournie par Intel qui atteint 2121 GFlops réels pour 3154 GFlops théoriques. La prochaine machine planifiée par le projet ASCI, l'option *blanche* [5] qui devra être fournie par IBM vise les 10 TFlops théoriques. Des technologies de clustering moins onéreuses permettent désormais de construire des machines parallèles à moindre coût [26, 126, 117, 174, 172, 150]. Les NOW constitués de machines SMP sont les architectures qui semblent les plus prometteuses, *à l'heure actuelle*, car elles offrent une architecture mémoire hybride (DM entre les noeuds, SM au sein d'un noeud) pour un coût raisonnable et une certaine efficacité. Le problème qui reste le plus souvent non résolu avec ces machines de type Hybride-MIMD est la programmation. Elles offrent souvent peu d'outils de développement et les modèles de programmation sont souvent très proche du modèle d'exécution DM-MIMD. Nous examinerons donc dans le paragraphe suivant les modèles de programmation parallèle les plus couramment utilisés sur les types de support d'exécution parallèle que nous venons de présenter.

3. également **VSM** pour **V**irtually **S**hared **M**emory
4. `http://www.sandia.gov/ASCI/Red/`
5. `http://www.llnl.gov/asci/news/white-news.html`

1.3.2 Les modèles de programmation parallèle

Un modèle de programmation parallèle doit permettre au programmeur d'exprimer la sémantique parallèle du programme, c'est-à-dire :

1. la **concurrence** : le fait que des activités s'exécutent en parallèle,
2. la **synchronisation** : la coordinations des activités concurrentes,
3. la **distribution** (ou **répartition**) : c'est-à-dire la répartition des données et des calculs entrant en jeu dans les activités concurrentes,
4. les **communications** : les échanges de données entre les activités concurrentes.

Ces quatres aspects caractérisent le parallélisme. Ils sont liés entre eux, la concurrence nécessite des moyens de synchronisation, la distribution des données implique des communications, etc... On peut noter que la distribution [23] et les communications ne sont pas des aspects exclusivement liés au parallélisme. En effet, des supports d'exécution séquentiels comme des librairies de communications offrent ces aspects dans leur modèle de programmation associé. Nous qualifierons de modèles de programmation parallèle les modèles permettant d'exprimer *tous* les aspects. Suivant le modèle utilisé le programmeur aura la charge d'exprimer plus ou moins explicitement un ou plusieurs des quatres aspects au moyen du `LPROG` associé. Cette distinction amène à la définition de parallélisme[6] *explicite* ou *implicite* :

DÉFINITION **1.7 (parallélisme implicite/explicite)** *Un modèle de programmation parallèle sera qualifié de [modèle à] parallélisme explicite lorsque le programmeur a la maîtrise totale de tous les aspects du parallélisme (concurrence, synchronisation, distribution, communication) au travers de `LPROG`. C'est le cas par exemple lorsque l'on programme avec une librairie de passage de messages. Au contraire, lorsque le parallélisme est laissé au bon vouloir du traducteur, soit par le biais de construction syntaxique d'un langage parallèle, soit parce que le traducteur est aussi un paralléliseur le parallélisme sera qualifié d'implicite.*

Nous présentons dans un premier temps quelques modèles de programmation parallèle et dans un deuxième temps (§1.3.3) les moyens de programmation supportant ces modèles ou définissant de nouveaux modèles. Les deux principaux modèles de programmation parallèle utilisés pour le calcul scientifique héritent directement des modèles d'exécution MIMD et SIMD, ces modèles sont :

- le parallélisme de contrôle ou parallélisme de tâches
- le parallélisme de données

Le parallélisme de tâches

Les premiers exemples de parallélisme de tâches sont venus de la programmation système [87]. Quasiment tous les systèmes d'exploitation des ordinateurs modernes renferment un parallélisme de contrôle intrinsèque. Sous Unix ou Windows 9x/NT, l'Operating System (OS), peut aisément imprimer un document, compiler un programme pendant que l'utilisateur consulte son courrier électronique. Le temps de réponse dépendra bien

6. par la suite « parallélisme » sera synonyme de « modèle de programmation parallèle »

évidemment de la puissance de l'ordinateur utilisé. En fait, ce n'est pas l'OS qui accomplit les différents travaux, mais chacun des programmes concernés quant au système d'exploitation il s'occupe d'attribuer les ressources (processeur, mémoire ...) à chacun et de ce fait *contrôle* leur exécution en *parallèle*. C'est en ce sens que l'on peut considérer l'OS comme l'arbitre du contrôle de l'exécution parallèle des programmes utilisateurs, libre à lui de gérer les conflits : accès aux ressources critiques, équilibrage de la charge, équité... Le modèle de programmation parallèle à parallélisme de tâches est donc défini comme suit :

DÉFINITION **1.8 (Parallélisme de tâches / Task Parallelism)** *Un programme suit un modèle de programmation à parallélisme de tâches s'il comprend plusieurs processus qui s'exécutent de façon concurrente et qui communiquent pour réaliser l'action globale du programme. Chaque processus possède ses données privées.*

concurrence : ***explicite***
synchronisation : ***explicite non spécifiée***
distribution : ***explicite calculs et données***
communications : ***explicites non spécifiées***

Ce modèle est également appelé parallélisme de contrôle (Control Parallelism), car le contrôle de la concurrence, de la synchronisation et de la distribution des calculs sont explicites.

Si le programmeur décide d'utiliser un `LPROG` qui supporte le parallélisme de tâches, il devra décomposer lui-même, lors de la conception et de la programmation, son application en processus ou tâches élémentaires. Il devra ensuite répartir ces processus sur les unités de calcul dont il dispose (s'il y a plusieurs processeurs par exemple), gérer les communications entre ceux-ci, éventuellement résoudre les conflits des ressources qu'ils partagent, etc... On voit ici le lien direct qui existe entre le modèle d'exécution DM-MIMD (Déf. 1.2) et le modèle de programmation à parallélisme de tâches. Chaque tâche correspond à un flot d'exécution qui possède ses données privées. Les aspects du parallélisme qualifiés *d'explicites* ***non spécifiés*** sont ceux qui seront à la charge du programmeur (i.e. explicite) mais la réalisation exacte n'est pas *spécifiée* par le modèle. Si l'on prend l'exemple des communications, on ne sait pas si elles sont synchrones/asynchrones, bufferisées, unilatérales, de groupe...

On peut par exemple imaginer une application qui calculerait la dérivée d'une fonction f dans un domaine d'espace Ω par la méthode des différences finies centrales à trois points. Comme l'évaluation de cette fonction est coûteuse et que l'utilisateur dispose d'une machine à 2 processeurs son programme pourrait se diviser en 2 tâches qui calculeraient les valeurs de la fonction f' respectivement dans les domaines Ω_1 et Ω_2, avec $\Omega = \Omega_1 \cup \Omega_2$ et $\Gamma = \Omega_1 \cap \Omega_2$. Le problème étant que ces deux calculs ne sont pas complètement indépendants l'un de l'autre et que pour la partie Γ les deux processus doivent échanger des informations. La formulation classique de la méthode aux différences finies est donnée par (1.3).

$$\forall x_i \in \mathring{\Omega}_{1,2}, f'(x_i) \simeq \frac{f(x_{i+1}) - f(x_{i-1})}{x_{i+1} - x_{i-1}} \tag{1.3}$$

On peut remarquer que pour les points appartenant à Γ, la dérivée ne peut être évaluée sans la collaboration explicite des 2 processus.

Le parallélisme de tâches nécessite toujours la communication des processus du programme parallèle. Dans le cas contraire, il s'agirait simplement de plusieurs programmes et non pas d'un programme parallèle. Cela signifie que le parallélisme de tâches nécessite un moyen de communication entre processus afin de répartir et d'échanger au cours de l'exécution les données et résultats du calcul. Plusieurs méthodes/modèles de communications peuvent être disponibles suivant l'architecture mémoire (DM,SM,NUMA). Nous examinerons ce problème particulier lorsque nous aborderons les moyens de programmation des machines parallèles (§1.3.3).

Le parallélisme de données

Le deuxième modèle de programmation parallèle, le plus souvent opposé au premier que nous venons de présenter, est le parallélisme de données.

DÉFINITION **1.9 (Parallélisme de données / Data Parallelism)** *Un programme suit un modèle de programmation à parallélisme de données s'il comporte un seul processus qui agit sur un ensemble de données de façon concurrente.*

concurrence	:	***implicite***
synchronisation	:	***implicite***
distribution	:	***implicite des calculs et explicite [non spécifiée] des données***
communications	:	***implicites***

Ce modèle de programmation tire son nom du seul aspect explicite du modèle : la distribution des données. Les autres aspects du parallélisme sont dirigés par la distribution des données.

La définition sera plus facilement illustrée par un exemple d'algèbre linéaire : un « scaling diagonal » d'une matrice A. C'est une opération qui consiste à multiplier A par une matrice diagonale dont les éléments diagonaux sont les inverses des éléments diagonaux de A qui est également appelé un préconditionnement de Jacobi [168, p. 266]. On suppose bien sûr que les éléments diagonaux de A sont tous non nuls.

$$\text{DiagonalScaling}(A) = \begin{pmatrix} a_{11}^{-1} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & a_{nn}^{-1} \end{pmatrix} \cdot \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \tag{1.4}$$

Le résultat de l'opération (1.4) est la multiplication de chaque ligne de la matrice A par l'inverse de l'élément diagonal de cette ligne. On peut alors dire que sur chaque ligne on effectue une opération « Data Parallel », puisque en fait la même opération est exécutée sur chaque élément de cette ligne. Le programme 1.1, écrit en Fortran 90, effectue le scaling diagonal de A. Bien que le modèle soit à distribution explicite des données, nous n'avons pas décrit cette distribution dans le programme. La description de la distribution dépendra du moyen utilisé pour la parallélisation (cf. voir §1.3.3). L'important ici est de voir que, hormis la distribution, les autres aspects du parallélisme sont implicites et donc le programme parallèle diffèrera peu du programme séquentiel. De la même manière d'autres opérations d'algèbre linéaire s'implantent facilement si l'on utilise le modèle Data Parallel [62]. Une description plus complète du modèle de programmation Data Parallel, des concepts, outils et langages associés peut être trouvée dans [154].

Programme 1.1: *Scaling diagonal en Fortran 90*

```
1 Real, Dimension(n,n) :: A
2     Do i = 1, n
3         A(i,1:n) = A(i,1:n) / A(i,i)
4     End Do
```

Les deux modèles de programmation (Défs. 1.8 & 1.9) parallèle que nous venons de définir ont évidemment leurs avantages respectifs. La « scalabilité » et la simplicité du parallélisme de données ne sont pas des arguments qui peuvent effacer la plus grande généralité du parallélisme de contrôle. Dans bien des cas on souhaiterait utiliser les deux modèles conjointement, des discussions et des tentatives d'intégration le prouvent [38, 70, 71, 39, 206, 53]. Les deux modèles ont malgré tout le même défaut : ils sont trop proches des modèles d'exécution MIMD (Déf. 1.2) pour le parallélisme de tâches et SIMD (Déf. 1.1) pour le parallélisme de données. Le grain du parallélisme du programme semble lié au grain de l'architecture parallèle sur laquelle s'exécute le programme.

DÉFINITION **1.10 (Granularité / [Fine|Coarse] Parallelism)** *La granularité (ou le grain) d'un programme parallèle est la taille moyenne des entités du programme s'exécutant en parallèle. On parle de parallélisme de grain très fin au niveau des instructions du programme dans une machine SIMD ou bien par exemple dans un programme « data parallel » où chaque processeur virtuel prendrait en charge une composante unique d'un vecteur. On peut aller vers un parallélisme à gros grain où un processeur se charge d'exécuter une tâche relativement lourde du programme ou bien de traiter un volume de données assez important (toute une partie de tableau par exemple).*

Si une bonne séparation du modèle de programmation et du modèle d'exécution s'opère le grain de `LEXEC` et de `LPROG` seront décorélés [64]. Nous serons alors seulement concernés par le grain du programme, le grain de l'architecture concernera les concepteurs de traducteurs `LPROG` en `LEXEC` pour ces machines.

Le modèle de programmation à parallélisme de tâches bien que général est sous-spécifié. Par exemple, rien n'est dit sur la manière dont est structurée l'application (un seul programme monolithique ou plusieurs programmes coopérant), la manière dont sont faîtes les communications entre les processus (synchrone, asynchrone, globale, point-à-point, uni-latérale, multi-latérale)... Certains modèles de programmation peuvent être considérés comme des spécialisations du modèle à parallélisme de tâches, car ils précisent un ou plusieurs de ces points. C'est le cas du modèle SPMD (Déf. 1.11) qui simplifie la programmation, ou du modèle BSP (Déf. 1.13) qui modélise les communications afin de pouvoir modéliser les performances.

DÉFINITION **1.11 (SPMD)** ***S**ingle **P**rogram **M**ultiple **D**ata stream*
On dit qu'une application parallèle suit un modèle de programmation SPMD si elle est représentée par un unique programme, qui agit sur des données différentes. Lorsque cette application SPMD s'exécute sur plusieurs processeurs d'un support d'exécution parallèle, le programmeur a la responsabilité de tester sur quel processeur il se situe afin d'exécuter le traitement souhaité sur les données privées correspondantes.

concurrence	:	***explicite***
synchronisation	:	***explicite non spécifiée***
distribution	:	***explicite avec contraintes. Les données sont privées et liées à chaque copie du programme SPMD répliqué.***
communications	:	***explicites non spécifiées***

Le modèle SPMD est en fait une discipline de programmation qui vient compléter les modèles à parallélisme de données et parallélisme de tâches En ce sens, ce modèle peut être vu comme un descendant des deux modèles. C'est une simplification du modèle à parallélisme de tâches, où la même tâche est répliquée sur tous les processeurs mais agit différemment suivant son lieu d'exécution car chaque tâche possède ses données privées. C'est aussi un descendant du parallélisme de données car dans le cas le plus simple, l'unique programme agit de façon identique sur des données différentes qui sont locales à chaque instance du programme. L'intérêt du modèle SPMD par rapport au parallélisme de tâches est son rapprochement avec le parallélisme de données. En effet, si le programme SPMD le prévoit, il pourra facilement s'exécuter sur un nombre quelconque de processeurs, tout comme une application à parallélisme de données. Le modèle SPMD est couramment utilisé comme restriction du modèle de programmation par passage de messages.

DÉFINITION **1.12 (Modèle de programmation par passage de messages)**
Le paradigme de programmation par passage de messages (message-passing model) ou par processus communicants nécessite un certain nombre de primitives de communications. Dans ce modèle de programmation, un processus possède ses données privées et il ne peut échanger des données avec d'autre processus que par envoi de messages. Un processus doit donc pouvoir envoyer à un autre processus ou recevoir d'un autre processus un message de façon bloquante ou non-bloquante. Un processus doit aussi pouvoir se synchroniser avec un groupe d'autres processus.

Ces fonctionnalités peuvent être réalisées par quatres fonctions :

- `ISEND`*, envoi non-bloquant d'un message, le processus appelle la fonction et continue son exécution sans savoir si le message a effectivement été envoyé. Le système se charge de continuer l'envoi du message. Le processus pourra tester la complétion de la transmission du message avec la fonction* `TEST`*,*
- `IRECEIVE`*, réception non-bloquante d'un message, le processus appelle la fonction et continue son exécution sans savoir si le message a effectivement été reçu. Le système se charge de continuer la réception du message. Le processus pourra tester la réception effective du message avec la fonction suivante,*
- `TEST`*, test de complétion de l'envoi ou de la réception d'un message suite à une communication non-bloquante, cette fonction renvoie par exemple un booléen disant si la réception d'un message commencée par une fonction de réception non-bloquante est effectivement terminée.*
- `BARRIER`*, barrière de synchronisation, chacun des processus appelant la barrière attend que tous les autres l'aient atteinte.*

concurrence : ***explicite***
synchronisation : ***explicite spécifiée (`TEST, BARRIER`)***
distribution : ***explicite avec contraintes. Chaque processus communicant possède ses données privées.***
communications : ***explicites par envoi de messages (`ISEND, IRECEIVE`)***

Le modèle de programmation par passage de messages [28] correspond en fait au modèle d'exécution d'une machine MIMD à mémoire distribuée, c'est aussi pour cela qu'on peut considérer ce modèle de programmation comme un assembleur du parallélisme. Il permet d'exprimer tous les modèles de parallélismes connus. Il est couramment contraint par une discipline de programmation SPMD. On peut noter d'autres approches intéressantes contraignant le modèles à passage de messages dans un but de modélisation des performances. C'est le cas par exemple du modèle de programmation parallèle BSP [182, 25].

DÉFINITION **1.13 (BSP)** ***Bulk Synchronous Parallelism*** *ou* ***Bulk Synchronous Parallel model***
*Un programme parallèle suivant le modèle BSP est une suite séquentielle de méta-étapes (*supersteps*) qui occupent conceptuellement les p processeurs de la machine parallèle sur laquelle le programme BSP s'exécute. Une méta-étape se décompose en 3 phases ordonnées :*

1. *une phase de calculs locaux n'utilisant que les mémoires locales à chacun des processeurs,*
2. *une phase de communication, impliquant des transferts de données entre les (p) processeurs,*
3. *une phase de synchronisation globale (*barrier synchronisation*) qui attend que tous les tranferts de données soit terminés.*

concurrence : ***explicite***
synchronisation : ***explicite multi-latérale par barrière***
distribution : ***explicite avec contraintes. Chaque processus communicant possèdent ses données privées.***
communications : ***explicites asynchrones par envoi de messages***

L'intérêt principal du modèle BSP est sa simplicité, il offre également un modèle de calcul du coût en temps d'exécution d'un programme parallèle, une fois les trois paramètres du modèle calculés. Les trois paramètres sont :

s : la vitesse (mesurée) d'un processeur,
g : le coût de délivrance d'un message,
l : le coût d'une synchronisation globale.

Les paradigmes de programmation parallèle qui ont été introduits ne sont pas les seuls qui existent, et des approches très différentes existent comme par exemple LINDA [44, 109]. Mais nous ne nous intéresserons pas à ces autres solutions qui, à notre connaissance, sont peu adaptées au calcul scientifique. Une fois ces modèles de programmation parallèle définis il faut se préocupper de leur mise en œuvre. Nous présentons donc, dans le paragraphe suivant, les moyens de programmation parallèle qui supportent les différents modèles que nous venons de citer.

1.3.3 Les moyens de programmation parallèle

Les moyens de programmation parallèle sont des outils qui permettent la construction de programmes pouvant fonctionner sur un support d'exécution parallèle. Ces moyens de programmation sont constitués d'au moins 1 traducteur (le plus souvent un compilateur) qui supportent un couple `LPROG`/`LEXEC` (voir figure 1.1 page 5). Certains outils, notamment les librairies, définissent leur propre modèle de programmation parallèle qui étend, restreint ou précise ceux que nous avons présentés précédemment. Nous parlerons indifféremment de moyens de programmation parallèle et de moyens de parallélisation de code.

Suivant le caractère implicite ou explicite des différents aspects du parallélisme on qualifiera également d'explicite ou d'implicite les différents moyens de parallélisation. Nous verrons donc dans les paragraphes suivants des moyens de parallélisation ou de programmation parallèle, en commençant par les plus implicites pour aller vers les plus explicites.

Nous rappelons que notre étude concerne les applications *numériques* parallèles, nous ne prétendons donc pas donner ici *tous* les moyens de programmation parallèle.

Parallélisation automatique

La parallélisation automatique de code, comme son nom l'indique, consiste à écrire son programme dans un langage séquentiel classique, par exemple Fortran, et le compilateur paralléliseur (resp. vectoriseur) se chargera de générer le code parallèle (resp. vectoriel) du programme pour une machine cible. Dans cette démarche, le parallélisme est purement implicite puisque le programme initial n'a aucune sémantique parallèle.

`LPROG` :	un langage séquentiel tel que C ou Fortran
`LEXEC` :	celui du support d'exécution du traducteur : langage de la machine cible, langage parallèle ou bien l'interface d'une librairie parallèle
concurrence :	implicite
synchronisation :	implicite
distribution :	implicite
communications :	implicites

Les compilateurs paralléliseurs étaient au départ des compilateurs vectoriseurs [107] adaptés aux machines vectorielles à architecture pipelinée [85, 131]. Leur fonctionnement consiste principalement [67] à analyser des « nids de boucles », i.e. des boucles imbriquées, à rechercher les dépendances entre les indices des boucles et chercher, par ce biais, quelles sont les itérations des boucles pouvant être exécutées en parallèle. Suivant cette optique, on considère que le programme séquentiel définit un ordre *total* d'exécution des opérations du programme, la parallélisation consiste à trouver (grâce à l'analyse des dépendances) un ordre *partiel* [64] qui autorise donc l'exécution en parallèle de certaines opérations. Cette approche concerne plus particulièrement les machines à mémoire partagée car l'accès à la mémoire est considéré comme global, lors de la programmation de machines parallèles à mémoire distribuée on cherche aussi à minimiser les communications. En effet, le coût d'une communication entre différents processeurs (ne partageant pas la même mémoire) est de plusieurs ordre de grandeur supérieur au coût d'un calcul. Il est donc important

TAB. 1.2 – *Capacités actuelles des vectoriseurs/paralléliseurs automatiques*

Parallélisation	
Bonne	Mauvaise ou Impossible
nids de boucles dont les indices sont en dépendances *affines*.	les autres cas
codes pour des machines parallèles à mémoire *partagées*.	codes pour les machines à mémoire distribuée [a]
vectorisation de code	

[a] limitation pratique, il n'y a pas d'obstacle théorique à la parallélisation sur architecture distribuée

d'avoir un placement des données du programme qui minimise les échanges de données.

Une autre technique de parallélisation automatique consiste à choisir des placements automatisés des données [66], permettant un traitement parallèle des données réparties afin de minimiser automatiquement les échanges de données entre processeurs. Une fois ces calculs effectués la partie paralléliseur du compilateur va générer un code enrichit de directives de compilations et/ou d'appels à des librairies parallèles qui va être compilé par le compilateur adéquat. On parle alors de *compilateur source à source*, car en fait ces outils générent un code source compilable par un autre traducteur (compilateur) non pas un code exécutable sur une machine cible. Dans notre terminologie, utiliser un compilateur source à source revient à changer de couple modèle de programmation/d'exécution. Le compilateur paralléliseur est alors un traducteur dont le support d'exécution est la librairie ou le langage parallèle du code généré. Pour le concepteur du compilateur paralléliseur le modèle d'exécution est celui-de la machine cible et le modèle de programmation celui de la librairie ou du langage parallèle utilisé. L'utilisateur final d'un compilateur paralléliseur ne sera pas forcément conscient de cet « empilement » de traducteurs et pour lui `LPROG` sera un langage séquentiel et `LEXEC` celui de la machine cible. Nous verrons également au paragraphe 5.5 comment la compilation source à source se rapproche de la programmation générative.

Le problème de la parallélisation automatique est sa faible couverture des applications parallélisables. En effet, même si les techniques d'analyse de dépendances sont parfois très élaborées [210, 64, 65], elles ne permettent pas de paralléliser autre chose que des codes contenant des nids de boucles dont les indices sont en dépendances affines, *ce qui couvre déjà beaucoup de code de calculs scientifiques, mais pas les codes d'algèbre linéaire creux*. On peut d'ailleurs noter qu'un des problèmes majeur, *à notre avis*, est que le compilateur paralléliseur est un outil généraliste qui n'a aucune connaissance du domaine de l'application et son analyse ne peut donc se fier qu'à ce qui est exprimé par le code source.

Le tableau 1.2 résume les capacités actuelles des vectoriseurs/paralléliseurs automatiques. Les compilateurs paralléliseurs sont de bon moyens de programmation parallèle pour des programmes amenés à s'exécuter sur des machines à mémoires partagée. Il n'y apparemment aucun obstacle théorique à la réalisation de compilateurs paralléliseurs pour

des architectures distribuées [68] dès lors que des compilateurs implanterons les techniques de placements automatiques des données [66] associées aux autres techniques de parallélisation [210, 65]. Les compilateurs paralléliseurs ne sont pas des moyens autonomes de parallélisation pour des applications contenant des nids de boucles dont les indices ne sont pas en dépendance affine ou bien lorsque l'architecture cible est à mémoire distribuée, ils peuvent, en revanche, être d'une grande aide pour l'analyse de code et de bon outils d'aide à la parallélisation [61].

Langages parallèles

Nous venons de voir un moyen simple, car totalement implicite, de paralléliser un code. Si l'on fait un pas vers le parallélisme explicite nous arrivons vers ce que nous appelons les langages [de programmation] parallèles.

DÉFINITION **1.14 (Langage de programmation parallèle)** *Nous appelons* langage de programmation parallèle *tout langage de programmation qui permet de programmer une machine parallèle, c'est-à-dire d'exprimer la sémantique parallèle du programme avec les mots du langage.*

On peut noter 3 catégories de langages parallèles :

- ***Langage parallèle à extensions compatibles*** : est un langage dont la base est un langage de programmation séquentiel classique, comme C++ ou Fortran 90, auquel on a rajouté des extensions qui permettent d'exprimer le parallélisme. Ces extensions sont dites *compatibles* car un programme parallèle écrit avec les mots et la syntaxe de ce nouveau langage peut être compilé par un compilateur du langage de base en un programme séquentiel. Les langages de ce type *que nous connaissons* utilisent des directives de compilation (voir programme 1.2) mises sous forme de commentaires et qui sont donc ignorés par le compilateur séquentiel. Nous verrons au §5.5 que l'on peut également qualifier les langages parallèles à extensions compatibles de *langages parallèles à aspects*.

`LPROG` :	le langage de base + extensions parallèles
`LEXEC` :	celui du support d'exécution du traducteur : langage de la machine cible, ou bien l'interface d'une librairie parallèle
concurrence :	implicite
synchronisation :	semi-implicite ou implicite manuelle
distribution :	semi-implicite ou implicite manuelle
communications :	implicites

  Dans un langage parallèle à extensions compatibles comme HPF on qualifie la parallélisation « d'implicite manuelle » car le programmeur ne donne que des indications sur le placement des données ou la synchronisation par le biais de directives. Le compilateur se sert de ces indications pour générer le code relatif à ces aspects du parallélisme. Les informations fournies par un paralléliseur sont typiquement celles qui sont utiles pour écrire ces directives de compilation.
  Exemples : HPF [91] (voir aussi annexe A), OpenMP Fortran/C/C++ [141, 142], HPC++ [88, uniquement les directives de boucles]. .

Programme 1.2: *F90 + Directives = HPF*

```
1 !HPF$ INDEPENDENT, NEW J
2       Do  I=1, 200
3           Do J=1, 3
4               A(I) = A(I) + B(I,J)
5           End Do
6       End Do
```

– ***Langage parallèle à extensions incompatibles*** : est un langage dont la base est un langage de programmation séquentiel classique auquel on a rajouté des extensions qui permettent d'exprimer le parallélisme. Ces extensions sont dites *incompatibles* car un programme parallèle écrit avec ce langage n'est pas compilable par un compilateur du langage de base. Les langages de ce type offrent généralement de nouveaux mots clefs (voir programme 1.3) qui permettent d'exprimer la sémantique parallèle.

`LPROG` :	le langage de base + extensions parallèles
`LEXEC` :	celui du support d'exécution du traducteur : langage de la machine cible, ou bien l'interface d'une librairie parallèle
concurrence :	plus ou moins explicite suivant les mots du langage
synchronisation :	plus ou moins explicite suivant les mots du langage
distribution :	plus ou moins explicite suivant les mots du langage
communications :	plus ou moins explicite suivant les mots du langage

Beaucoup de traducteur de langages parallèles à extensions incompatibles sont en fait des compilateurs source à source qui traduisent le programme en un programme du langage de base agrémenté d'appels à une librairie parallèle écrite spécialement pour cette tâche. Exemples : CC++[72, Chap. 5], pC++ [21], Object-Oriented Fortran [160], Mentat [111]

– ***Nouveau langage parallèle*** : est un langage dont la syntaxe et la sémantique ont été entièrement construites pour les besoins de la cause.

`LPROG` :	le nouveau langage parallèle
`LEXEC` :	celui du support d'exécution du traducteur : langage de la machine cible, ou bien l'interface d'une librairie parallèle
concurrence :	plus ou moins explicite suivant les mots du langage
synchronisation :	plus ou moins explicite suivant les mots du langage
distribution :	plus ou moins explicite suivant les mots du langage
communications :	plus ou moins explicite suivant les mots du langage

Exemples : ABCL/R [2], Emerald [63], Obliq [135] ...

L'intérêt principal des langages parallèles à extensions compatibles est précisemment leur compatibilité car le programme parallèle peut être compilé par un compilateur séquentiel qui ignorera les instructions parallèles. Cette possibilité garantie la maintenabilité conjointe du code séquentiel et du code parallèle ce qui est un atout car une seule version du code peut servir aux deux implantations, nous verrons au paragraphe 1.6 que ceci est un des problèmes que nous souhaitons résoudre à l'aide notre approche objet.

Programme 1.3: *Exemple de collection en pC++*

```
1  Collection  NameOfCollectionType: Kernel {
2  private:   ...
3  protected: ...
4  public: ...
5  MethodOfElement: ...
6  }
7
8  Collection<E>  c; // declare un objet de
9                    // type collection de E
10
11 c.f()   // applique la MethodOfElement 'f'
12         // a chaque element de la collection
```

Les languages à extensions incompatibles sont souvent implantés au moyen de compilateurs sources à sources. Si l'on prend l'exemple de pC++ [153], le compilateur produit du code C++ compilable par un compilateur C++ standard, agrémenté d'une librairie. Cette technique est encore une fois un changement de couple modèle de programmation/exécution, le nouveau langage définit le nouveau modèle de programmation qu'il veut supporter et le traducteur s'appuit sur le modèle de programmation de la librairie qui l'accompagne.

Nous nous sommes peu intéressé aux nouveaux langages parallèles car ils ne constituaient pas une solution utilisable sous les contraintes de portabilité de nos applications. De plus, ils ont souvent été crées pour étudier un problème particulier comme la mobilité, la persistence, la réflexivité...

Librairies parallèles

Une dernière famille de moyens de parallélisation est celle des librairies parallèles. Avec cette dernière catégorie nous faisons un pas de plus vers le parallélisme explicite car les quatres aspects du parallélisme (concurrence, synchronisation, distribution et communication) sont exprimés *explicitement par le programmeur* par des appels aux fonctions de la librairie parallèle. Ceci implique que les librairies parallèles définissent plus ou moins implicitement un modèle de programmation associé qui correspond aux fonctionnalités offertes par la librairie. Les librairies parallèles sont généralement utilisables dans plusieurs langages de programmation (C, C++, Fortran) ce qui permet une certaine indépendance vis à vis du langage. Alors que les langages parallèles visent plutôt le programmeur d'applications, les librairies parallèles constituent généralement une meilleure solution pour le concepteur/programmeur de librairies ou bien le concepteur/programmeur de compilateur source à source. En effet, si une librairie s'appuie sur un langage parallèle pour son implantation un changement de langage parallèle sera plus difficile qu'un changement de librairie parallèle qui offrent bien souvent les mêmes fonctionnalités. Les librairies de programmation parallèles peuvent être considérées comme des « langages d'assemblage »

du parallélisme car elles permettent d'expliciter tous les aspects du parallélisme dans le programme, mais ne sont pas toujours simples à utiliser. Une dernière caractéristique des librairies parallèles est leur caractère **intrusif**, c'est-à-dire que le code séquentiel parallélisé est modifié par des appels à la librairie parallèle qui ne concerne que la version parallèle du programme. Si le programmeur initial du code ne connaît pas la librairie parallèle et ses fonctionnalités il peut être incapable de relire ou de comprendre le code parallélisé. Le parallélisme a fait **intrusion** dans le code séquentiel.

Nous classons les librairies parallèles en trois catégories :

- les librairies de processus légers ou « thread », qui s'utilisent généralement sur les machines parallèles à mémoire partagée.
- les librairies de passage de messages, qui s'utilisent généralement sur les machines parallèles à mémoire distribuée.
- les librairies objets, dont nous parlerons au §1.5 concernant les langages à objets car elles ont un statut à part :
  - elles sont rarement utilisables en dehors du langage dans lequel elles sont écrites,
  - elles sont généralement intimement liées avec un langage à objets pour lequel elles fournissent/définissent tout ou partie de leur modèle de programmation [88, 77],
  - elles font souvent partie d'approche mixte [27, 33](compilateur source à source + librairie) et ne sont par conséquent que rarement utilisées seules.

Nous ne présenterons ci-après que les deux premières catégories de librairies parallèles, et nous nous intéresserons plus en détail à la dernière au §1.5.

Librairies de processus légers. Le terme processus léger est généralement employé en opposition au processus classique (ou processus lourd) [127] qui est une unité d'exécution possédant son propre contexte d'exécution, sa pile, sa mémoire privée, ses descripteurs de fichiers ouverts... On prendra comme définition de processus ou processus lourd celle d'un processus Unix. La création d'un processus lourd est donc une opération coûteuse, de même le changement de contexte entre 2 processus lourd est également une opération coûteuse. Contrairement au processus lourd, un processus léger (ou thread) possède peu de ressources privées : une pile d'exécution et un compteur ordinal. Les autres ressources sont partagées avec le processus ayant crée le processus léger. La ressource partagée qui nous intéresse principalement est la mémoire. Les processus légers qui partagent une zone mémoire peuvent communiquer directement par des accès à cet espace d'adressage commun. Ces librairies de processus légers sont le principal support pour le modèle de programmation par processus légers ou programmation « multithread ».

DÉFINITION **1.15 (Modèle de programmation par processus légers)** *Le paradigme de programmation par processus légers (multithread programming model) est une spécialisation du modèle à parallélisme de tâches (Déf. 1.8) dans lequel la distribution des données et les communications ont été spécifiées comme se faisant en mémoire partagée.*

concurrence	:	***explicite***
synchronisation	:	***explicite par sémaphore d'exclusion mutuelle et thread_join***
distribution	:	***explicite pour les calculs (thread_create/thread_join) implicite pour les données puisqu'elles sont en mémoire partagée.***
communications	:	***implicite grâce à la mémoire partagée***

La majeure partie des données n'est plus privée à chaque tâche mais partagée par tous les threads issus d'un même processus lourd.

Même si certaines librairies [127, 12] proposent d'étendre la programmation multi-thread à des architectures distribuées, le domaine d'utilisation le plus répandu est la programmation des machines SMP. La portabilité des applications parallèles écrites est garantie soit par l'utilisation d'une interface standardisée comme les threads POSIX[7], ou bien par une librairie disponible sur un grand nombre de plateformes [3]. Certains langages comme Java supportent directement la notion de thread[8] dans le langage ou dans la librairie standardisée associée au langage. Si l'on utilise qu'un seul langage la portabilité est assurée par celle du langage lui-même.

Librairies de passage de messages. Les librairies de passage de messages implantent et supportent le modèle de programmation par passage de messages (Déf. 1.12). Toutefois chaque librairie offre des raffinements et des extensions au modèle. Les deux librairies de passage de messages les plus couramment utilisées sont PVM [192, 158] et MPI [118, 119]. La première est un standard de fait, car elle est utilisée par un grand nombre de personnes. La seconde est une proposition de standard issue de la réflexion d'un groupe de travail international le ***M**essage **P**assing **I**nterface **F**orum*[9] (MPIF). Nous ferons une courte introduction à la seconde bibliothèque à titre d'exemple, mais on retrouve essentiellement les mêmes fonctionnalités [78] dans l'ensemble de ces bibliothèques.

Les bibliothèques de passage de messages comme MPI, enrichissent donc le modèle de programmation par passage de messages en offrant un nombre bien plus important de fonctions que celles présentées à la définition 1.12. Même ci ces dernières pourraient être réalisées avec les seules précédentes. On distingue notamment les communications point-à-point mettant en jeu 2 processus réalisant des envois et réceptions qui se correspondent et les communications collectives qui mettent en jeu un groupe de processus. Dans les primitives précédentes, `BARRIER` est une communication collective puisque chaque processus voulant se synchroniser appelle la fonction. Les principales fonctions de communication globales ou collectives sont présentées à la figure 1.4.

Les fonctions de communications de base, sont bi-latérales (`SEND`/`RECEIVE`) ou multi-latérales (`BARRIER`), ce qui signifie que tous les processus souhaitant participer à la communication doivent appeler une fonction de la librairie. Par exemple, si le processus #0 veut envoyer une donnée de sa mémoire locale vers celle du processus #1, le processus #0 doit appeler `SEND` et le processus #1 doit appeler `RECEIVE`. Ce qui implique que #0 et #1

7. IEEE Std 1003.1c–1995 POSIX System API
8. Voir `http://java.sun.com/docs/index.html` → `java.lang.Thread`
9. Sur le web `http://www.mpi-forum.org/`

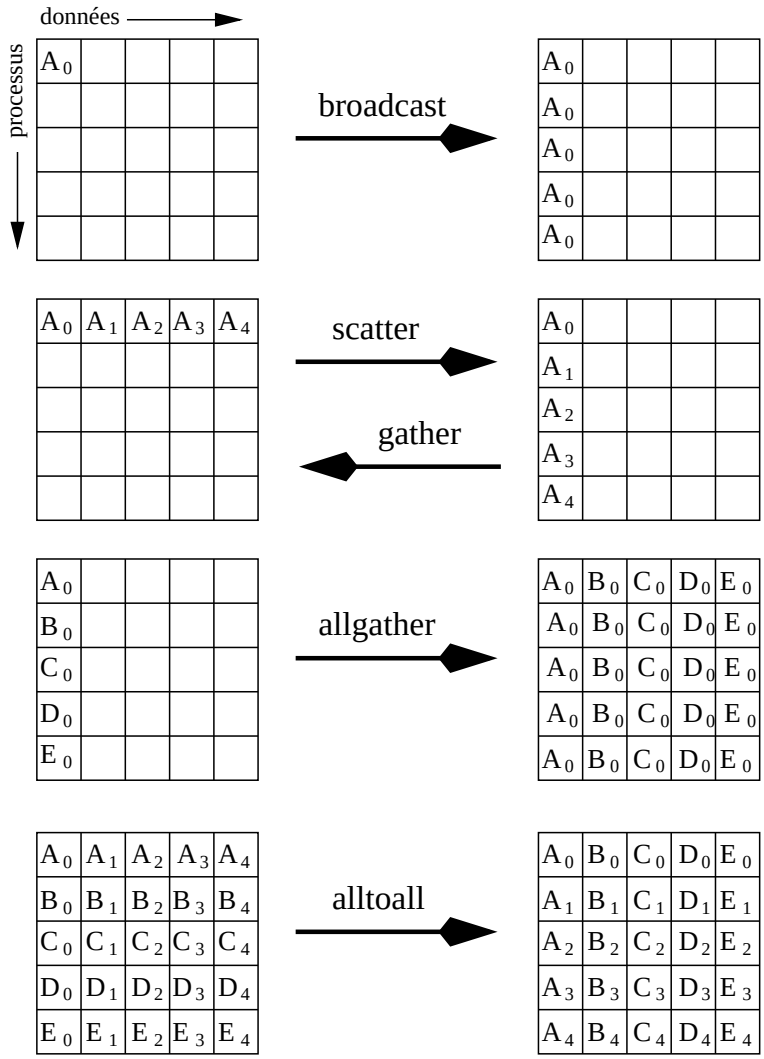

FIG. 1.4 – *Communications collectives dans un groupe de processsus*

doivent collaborer explicitement de façon *bi-latérale*. Le raisonnement est le même pour une communication globale impliquant une collaboration *multi-latérale*. Certaines librairies permettent des communications *uni-latérale* du type lecture ou écriture à distance. Dans ce dernier cas, un seul des processus appellera la librairie, dans notre exemple précédent le processus #0 appellera `PUT` **ou bien** le processus #1 appellera `GET`. Toutes ces fonctionnalités sont spécifiées dans le standard MPI, mais les communications uni-latérales ne sont présentes que dans MPI-2 [119] et pas dans MPI-1 [118]. Une présentation plus détaillée des fonctionnalités de MPI est donnée en Annexe B. Nous donnons ci-après un petit exemple d'utilisation de MPI avec Fortran 77.

Un produit scalaire distribué en MPI Afin d'illustrer l'utilisation de MPI nous prenons un exemple de programmation SPMD avec MPI : un produit scalaire distribué. Le code de l'application Fortran 77+MPI est donné par le programme 1.4. Les lignes 1–19 représentent l'initialisation, les lignes 20–53 servent à la distribution des données calculées par le processus #0, les lignes 54–59 effectuent le produits scalaire distribué. Au travers de cet exemple, le lecteur pourra voir l'aspect le plus génant de MPI : la complication du code du programmme. Cet aspect illustre le qualificatif « d'assembleur du parallélisme » parfois donné aux librairies de passage de messages, ceci est directement lié au caractère explicite et intrusif du modèle de programmation parallèle.

1.3.4 Autres aspects

Nous n'avons pas abordé tous les aspects du parallélisme car cela dépasse largement le cadre de notre étude. Les aspects que nous avons passés sous silence seront considérés comme pris en charge par le support d'exécution que nous utilisons ou bien par le traducteur qu'utilise notre application comme support d'exécution. C'est tout l'intérêt des changements de couples LPROG/LEXEC, on se place au niveau qui nous intéresse.

Par exemple, nous n'avons pas abordé l'équilibrage de charge dans les applications parallèles, c'est-à-dire l'équitable répartition des calculs sur les différentes unités de calcul. En effet, si l'équilibrage de charge d'une application parallèle est mauvais les processeurs sous-chargés vont attendre les processeurs surchargés et réduire ainsi l'efficacité parallèle du programme. Même si nos applications tests offrent la possibilité d'un équilibrage *statique* via une distribution initiale des données ou des calculs, ce n'est pas toujours le cas. Dans certaines applications dites irrégulières, on ne connaît pas à l'avance le volume des données à traiter ou bien les calculs à effectuer. C'est le cas, par exemple, des applications de Branch & Bound, qui explorent en parallèle des sous-arbres d'un arbre donné sans savoir à l'avance la profondeur du sous-arbre que l'on va explorer. Nous n'avons pas traité les problèmes d'équilibrage de charge car il dépassent le cadre de nos travaux. On peut toutefois, noter que certains supports d'exécution comme les librairies multithreadées spécialisées telles que PM^2 [127] ou Athapascan-0/1 [12, 13] proposent des solutions à ces problèmes. En effet, l'environnement de processus légers qu'elles mettent en œuvre permet de générer un grand nombre de tâches légères que l'ont peut répartir *dynamiquement* suivant la charge des unités de calcul. Pour la suite, nous considérerons donc que les problèmes d'équilibrage de charge sont du domaine du support d'exécution et ne nous concerne pas.

Programme 1.4: *Produit scalaire distribué en Fortran 77+MPI*

```
      PROGRAM DISTDOT
      IMPLICIT NONE
#include <mpif.h>
*     .. Variables Locales ..
      Integer     VecSize
      Parameter (VecSize=100)
      Real        SL, S, XL(VecSize), YL(VecSize)
      Integer     Ierror, MPI_COMM_PERSO, MyRank, PersoSize
      Integer     I, J, RequestX, RequestY, Status(MPI_STATUS_SIZE)
*     .. Subroutines Externes ..
      External    SDOT
      Real        SDOT
      Intrinsic   REAL
*     .. Initialisation de MPI
      Call MPI_Init(Ierror)
*     .. Duplication du communicateur universel
      Call MPI_Comm_Dup(MPI_COMM_WORLD,MPI_COMM_PERSO, Ierror)
      Call MPI_Comm_Rank(MPI_COMM_PERSO, MyRank, Ierror)
      Call MPI_Comm_Size(MPI_COMM_PERSO, PersoSize, Ierror)
*     .. Initialisation des vecteurs par le processus 0
*     .. et envoi aux autres processus
      If (MyRank.EQ.0) Then
         Do I = 1, PersoSize-1
            Do J = 1, VecSize
               XL(J) = REAL((I-1)*VecSize+J)
               YL(J) = REAL(J)
            End Do
*           .. On envoit les vecteurs aux autres processus
*           .. car seul le processus 0 peut faire des I/O
            Call MPI_ISEND(XL, VecSize,MPI_REAL, I, 0,
     +                     MPI_COMM_PERSO,RequestX, Ierror)
            Call MPI_ISEND(YL, VecSize,MPI_REAL, I, 0,
     +                     MPI_COMM_PERSO,RequestY, Ierror)
*  .. On doit attendre avant de reutiliser
*  .. les buffer d'envoi XL et YL
            Call MPI_WAIT(RequestX,Status,Ierror)
            Call MPI_WAIT(RequestY,Status,Ierror)
         End Do
         Do J = 1, VecSize
            XL(J) = REAL(J)
            YL(J) = REAL(-J)
         End Do
*  .. Les autres processus (pas le root) recoivent les donnees
      Else
         Call MPI_IRECV(XL, VecSize,MPI_REAL,0,0,MPI_COMM_PERSO,
     +                  RequestX, Ierror)
         Call MPI_IRECV(YL, VecSize,MPI_REAL,0,0,MPI_COMM_PERSO,
     +                  RequestY, Ierror)
*        .. On attend que les donnees soient effectivement recues
*        .. avant de faire le produit scalaire local
         Call MPI_WAIT(RequestX,Status,Ierror)
         Call MPI_WAIT(RequestY,Status,Ierror)
      End If
*  .. Produit scalaire local en faisant appel aux BLAS
      SL = SDOT(VecSize,XL,1,YL,1)
      Print *, 'Processeur ', MyRank, ' --> Somme locale:', SL
*  .. Reduction pour calculer la somme des produits scalaires locaux
      Call MPI_AllReduce(SL,S,1,MPI_REAL,MPI_SUM,MPI_COMM_PERSO, Ierror)
      Print *, 'Processeur ', MyRank, ' --> Somme Globale:', S
      Call MPI_Finalize(Ierror)
      END
```

Nous venons de faire une rapide revue des modèles d'exécution et de programmation parallèle ainsi que des moyens de programmation associés. Nous avons volontairement laissé de côté les approches orientées-objet car elles nécessitent la connaissance de concepts spécifiques. Nous rappelons ci-après ces concepts propres aux démarches orientées-objet et nous aborderons ensuite les modèles et moyens de programmation parallèle orientés-objet.

1.4 Les concepts orientés-objet

Les approches orientées-objet ont pour but de rapprocher la spécification, la conception et la réalisation d'applications informatiques d'un domaine, des notions, concepts ou objets qui le composent. Les objets peuvent être réels si le domaine concerne une réalité physique comme l'automobile : dans ce cas ils pourraient être une voiture, une roue, un volant, un moteur... Les objets peuvent être des abstractions d'un domaine sans réalité physique comme la géométrie, ces objets pourraient être un cercle, un polygone, une droite, un plan, un point... Ainsi les objets du domaine seront représentés par des objets informatiques, ce qui facilite la conception, l'écriture et la compréhension des programmes. Les méthodes de conception orientées-objet s'attachent donc à répondre à la question « **Sur quoi le système agit-il ?** » plutôt qu'à la question classique en conception fonctionnelle « **Que doit faire le système ?** » [112, §5.4 p116]. On cherche en premier lieu quels seront les objets du système et ensuite quelles seront les interactions entre ces objets. Les modèles de programmation orientés-objet et leurs outils associés sont donc centrés autour des objets et de leurs interactions.

1.4.1 Définitions

Une approche orientée-objet doit permettre la spécification, la conception et la réalisation des *objets* d'un système et des *relations* entre ces objets. C'est pourquoi nous qualifierons d'orienté-objet une méthode de conception ou un langage de programmation qui supporte toutes les notions qui caractérisent l'orienté-objet, soient :

1. l'abstraction,
2. l'encapsulation,
3. l'héritage,
4. le polymorphisme,
5. la généricité.

Les caractéristiques d'un langage orienté-objet sont discutables [128, 191, 190]. Celles énoncées précédemment correspondent à celles que l'on trouve dans les langages actuels comme C++, Eiffel ou Java[10]. Nous donnons ci-après des définitions de ces notions en expliquant comment elles se réalisent. On peut les retrouver dans différents ouvrages [112, 1, 140] avec des variantes ou définitions complémentaires.

DÉFINITION **1.16 (Abstraction)** *C'est la possibilité de définir des objets abstraits ou abstractions qui peuvent être par la suite déclinés ou instanciés sous leur forme concrètes*

10. la généricité devrait faire son apparition dans Java [79].

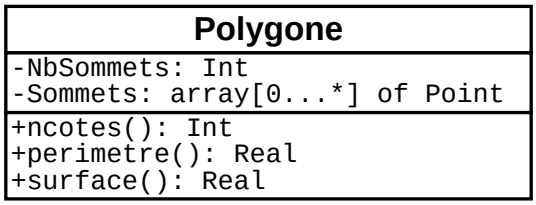

FIG. 1.5 – *Exemple d'encapsulation et d'abstraction : une classe Polygone*

pour être utilisés par le système. L'abstraction est le mécanisme qui sert à la classification des objets du système.

Dans un modèle de programmation orienté-objet le mécanisme d'abstraction est la **classe**. Elle regroupera à la fois ses données ou **attributs** et ses fonctionnalités ou **méthodes**. Les instances concrètes d'une classe seront les **objets** du programme. L'opération de création d'objet à partir de la classe est **l'instanciation**.

DÉFINITION **1.17 (Encapsulation)** *C'est la possibilité de séparer dans les objets du système les parties privées, qui ne seront accessibles que par l'objet lui-même, des parties publiques que d'autres objets pourront accéder. L'encapsulation permet de garantir une certaine modularité en isolant les responsabilités de chaque objet.*

Dans les modèles de programmation orientés-objet, l'encapsulation est généralement spécifiée au niveau de la classe qui définit une politique d'encapsulation pour toutes ses intances. On parlera des parties **publiques** ou **privées** d'une classe ou d'un objet. La partie privée renferme généralement les détails d'implantation qui ne font pas partie des spécifications ainsi que les variables d'état de l'objet. L'exemple d'une classe `Polygone` représentant l'abstraction polygone géométrique est donné en figure 1.5. Nous utilisons une notation graphique orientée-objet proche d'UML [134, 196, 195] où une classe est représentée par une boîte nommée contenant plusieurs compartiments. Le nom des entités privées est précédé d'un « - » comme pour `-NbSommets: Int` et le nom des entités publiques est précédé d'un « + ».

Les définitions 1.16 et 1.17 caractérisent ce qui est nécessaire à la réalisation des **T**ypes **A**bstraits de **D**onnées (TAD) [74] qui sont à la base de la conception et de la programmation orientée-objet [112, Chap. 6] . Des langages comme Fortran90/95 [108] ou Ada83 qui ne sont pas réellement orientés-objet, supportent ces deux notions. En revanche, les notions *d'héritage*, de *polymorphisme* et de *liaison dynamique* que nous explicitons ci-après constituent la pierre d'angle des langages orientés-objet que nous appellerons aussi *langages à objets* (LAO).

REMARQUE **1.1 (Langages à objets vs orientée-objet)** *On pourrait restreindre l'appellation langage à objets aux langages qui supportent* ***uniquement*** *un modèle de programmation orienté-objet, c'est-à-dire où tout est objet comme Eiffel, Smalltalk ou bien Java. Ceci exclurait d'office C++ et Ada95 qui sont bel et bien orientées-objet mais qui supportent d'autres modèles de programmation. Nous appellerons donc langages à objets, les langages orientés-objet que nous utiliserons dans un cadre strictement orienté-objet.*

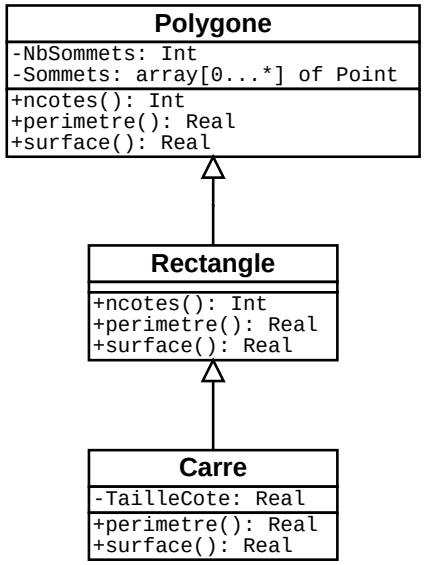

FIG. 1.6 – *Un arbre d'héritage pour des figures géométriques*

Un des objectifs majeurs des démarches orientées-objet est la réutilisation. L'héritage en est l'acteur principal.

DÉFINITION **1.18 (Héritage)** *C'est le mécanisme qui permet de réutiliser la définition d'une abstraction pour en définir une nouvelle qui hérite de la définition de la première. L'héritage peut être multiple si la nouvelle abstraction hérite de plusieurs abstractions. L'héritage est la définition même de la relation* ***Is-a*** *[112, Chap. 24]. Si une classe* `B` *hérite d'une classe* `A` *on peut alors considérer que tout objet de classe* `B` ***est un*** *objet de classe* `A`*.*

Dans les langages à objets, la relation d'héritage génère des hiérarchies de classes qui structurent et classifie les objets du système. On peut utiliser la fonctionnalité « héritage » du langage de programmation à d'autres fins que l'implantation de la relation **Is-a** comme le montre la taxinomie des utilisations de l'héritage présentée dans [81, Chap 3. §4.2]. Un exemple d'héritage par spécialisation est donné en figure 1.6 où la figure géométrique `Carre` hérite de `Rectangle` qui lui-même hérite de `Polygone`. La relation **Is-a** est représentée par une relation UML de *Generalization/Specialization* qui est symbolisée par une flèche évidée orientée de l'élément le plus spécifique (le fils) vers l'élément le plus général (le père).

Un autre objectif des LAOs est la facilité de programmation, de compréhension et de modification des application, le polymorphisme participe à cet objectif.

DÉFINITION **1.19 (Polymorphisme)** *C'est la capacité pour des objets ou des fonctions offrant la même interface, d'être manipulés de façon uniforme en répondant aux invocations de cette interface d'une manière qui leur est spécifique.*

Par exemple, un objet `Carre` *et un objet* `Cercle` *possédant tous deux la méthode* `perimetre(): Real` *dans leur interface pourraient être traités de façon polymorphe comme*

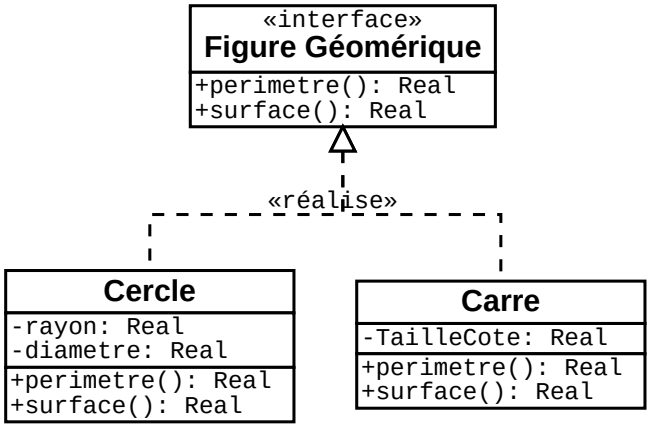

FIG. 1.7 – *Deux figures géométriques [potentiellement] polymorphes*

des objets géométriques sachant calculer leur périmètre. Toutefois, chaque objet calculera son périmètre d'une façon qui lui est propre.

Le polymorphisme est une caractéristique fonctionnelle, c'est pourquoi on qualifie également de polymorphes des fonctions qui acceptent des paramètres de types différents. C'est donc le traitement qui est effectué sur des objets qui est polymorphe et non pas les objets eux-mêmes. On qualifiera donc de polymorphe les types, classes ou références des objets qui peuvent potentiellement *être traités de façon polymorphe.*

Il existe plusieurs catégories de polymorphisme dépendant de leur réalisation [31]. En ce qui concerne les langages à objets, il s'agit le plus souvent de *polymorphisme universel par inclusion*, car les objets sont polymorphes grâce à la relation d'héritage qui les lient. Par exemple, les objets de classe `Polygone` et `Carre` de la figure 1.6 pourront être traités de façon polymorphe car `Carre` hérite de `Polygone`. L'interface de `Carre` inclut donc celle de `Polygone`. On peut toutefois noter que les types de la figure 1.7 sont potentiellement polymorphes sans toutefois être reliés par une relation d'héritage, ils sont en fait conformant au type « figure géométrique sachant calculer son périmètre et sa surface ». Le polymorphisme est intimement lié aux notions de conformance, de sous-typage et d'héritage. Ces relations ne sont pas toujours simples [31, 41] et leur réalisation dépend du language de programmation. Nous y reviendrons au paragraphe 4.3.

Dans les langages à objets, le polymorphisme par héritage amènent tout naturellement à la notion de *liaison dynamique* qui fait l'objet de la définition suivante.

DÉFINITION **1.20 (Liaison Dynamique ou Liaison Tardive)** *La liaison dynamique est le mécanisme qui permet à des objets traités de façon polymorphe de fournir dynamiquement leur comportement spécifique* ***à l'exécution*** *lors d'un appel à leur interface polymorphe. On peut dire que le mécanisme de liaison dynamique est réalisé lorsque le polymorphisme a (conceptuellement*[11]*) lieu pendant l'exécution du programme. Dans les langages à objets, la liaison dynamique est liée à l'héritage et au polymorphisme.*

11. l'implantation de la liaison tardive dépendra des capacités d'optimisation du compilateur du LAO choisi qui, à l'issue d'une analyse statique, lors de la compilation pourra, éventuellement, transformer cette liaison dynamique en une liaison statique.

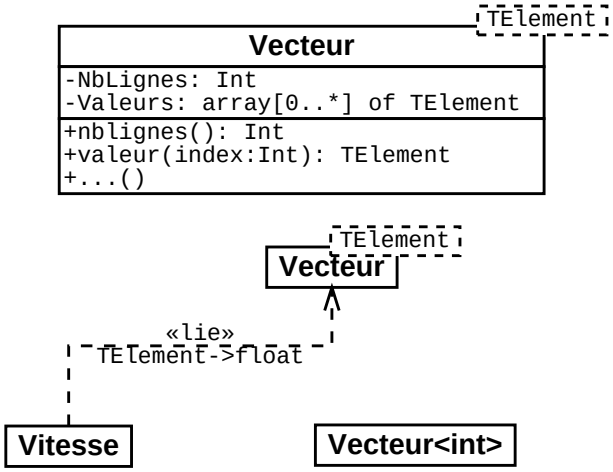

FIG. 1.8 – *Une classe vecteur générique*

L'exemple de programme 1.5 illustre le mécanisme de liaison dynamique issu de l'héritage entre les classes de la figure 1.6. Le tableau de `Polygones`, `TabPolygones` passé en paramètre de `somme_surface` contient diverses formes géométriques qui sont toutes considérées comme des `Polygones` au moment du stockage dans le tableau (lignes 30–33). Lors de l'appel de la méthode `Polygone::surface` à la ligne 17, la méthode spécifique de calcul de la surface de chaque objet sera dynamiquement appelée. En effet, même si les classes `Carre` et `Rectangle` héritent de `Polygone` ces dernières ont **redéfinie** leurs propres méthodes `Carre::surface` et `Rectangle::surface` en remplacement de `Polygone::surface`. Le mécanisme de liaison dynamique permettra d'appeler la « bonne » méthode même si un `Carre` est considéré comme un `Polygone`. Il ne faut pas confondre la notion de *redéfinition* et celle de *surcharge*. La première participe à un polymorphisme universel et permet la liaison dynamique alors que la seconde participe à un polymorphisme ad hoc et ne permet pas la liaison dynamique. Certains languages comme C++ supportent ces deux notions.

La notion suivante n'est pas communément admise comme une caractéristique des langages à objets. En effet, un LAO reconnu tel que Java[12] ne supportent pas la généricité au niveau du langage[13]. Pourtant elle trouve naturellement sa place au côtés du polymorphisme et de la liaison dynamique, car elle permet la réalisation du polymorphisme [universel] paramétrique [31] [140, Section 2 : Typing].

DÉFINITION **1.21 (Généricité)** *La généricité est la capacité de définition d'abstractions paramétrées par une ou plusieurs autres abstractions.*

L'exemple le plus courant d'utilisation de la généricité est la famille des types abstraits de données classiques comme les listes, les piles, les arbres ou autres conteneurs de données comme les tableaux, les vecteurs, les matrices, ... La figure 1.8 montre la représentation

12. La généricité devrait être introduite dans Java [79] en reprenant des fonctionnalités d'extension de Java comme GJ [80]

13. On remarquera que pour les LAOs à typage dynamique[112, page 612] tel que Smalltalk la question du support de la généricité ne se pose pas

Programme 1.5: *Liaison dynamique en C++*

```
/**
 * Calcule la somme des surfaces des N polygones
 * contenus dans le tableau DesPolygones.
 */
float somme_surface(Polygone* DesPolygones[], int n)
{
  int   i;
  float somme;

  somme = 0.0;
  /* Calcul de la somme des surfaces *
   * des polygones                   */
  for (i=0;i<n;++i)
    {
      // l'appel a la methode surface fera appel
      // au mecanisme de liaison tardive
      somme += DesPolygones[i]->surface();
    }
  return somme;
}

[...] // dans un programme principal

 Polygone* TabPolygones[4];
 float     surface_totale;

 /* On cree diverses formes geometriques qui sont  *
  * ensuite stockees dans le tableau des polygones *
  * EN TANT que Polygone                           */
 TabPolygones[0] = new Carre(...);     // creation d'un carre
 TabPolygones[1] = new Rectangle(...); // creation d'un rectangle
 TabPolygones[2] = new Polygone(...);  // creation d'un polygone
 TabPolygones[3] = new Carre(...);     // creation d'un carre

 /* Calul de la surface totale */
 surface_totale = somme_surface(TabPolygones,4);

[...]
```

d'une classe `Vecteur<TElement>` générique sous la forme d'une classe paramétrée UML. Les paramètres génériques se placent dans le cadre en traits pointillés apposé en haut à droite de celui de la classe. Avec l'introduction de la généricité il existe désormais 2 niveaux d'instanciation :

- instanciation **classe → objet**
  qui est l'opération qui permet de passer d'une abstraction à l'un de ses représentants,
- instanciation générique **classe générique → classe**
  qui est l'opération qui permet de passer d'une famille d'abstraction à une abstraction.

Ainsi la classe `Vecteur<int>` réprésentant un vecteur d'entiers, est le résultat de l'instanciation générique de la famille de la classe générique `Vecteur<TElement>` où le paramètre générique formel `TElement` a été remplacé par le paramètre effectif `int`. La classe `Vitesse` est une autre instanciation générique de la classe `Vecteur<TElement>`, notée différemment, avec une contrainte stéréotypée UML `«lie»` qui indique la liaison des paramètres génériques, ici `TElement->float`, donc `Vitesse` est un vecteur de réels. Il faut noter qu'une classe générique peut avoir plusieurs paramètres génériques. On peut donc remarquer que les deux types d'instanciations ont une grande similitude avec un appel d'une fonction prenant des paramètres. Dans le cas de l'instanciation classe → objet c'est un appel à la fonction « constructeur » de la classe, dans le cas de l'instanciation classe générique → classe c'est la classe générique elle-même qui est la fonction dont les paramètres formels sont les paramètres génériques. Les possibilités offertes par la généricité, notamment la spécification des paramètres formel, dépend du langage de programmation. Par exemple, Eiffel autorise la généricité *contrainte*, c'est-à-dire que le paramètre formel est spécifié comme devant être un descendant d'une classe donnée. C++ en revanche ne permet pas de contraindre les paramètres génériques. Discuter les avantages et les inconvénients de la généricité contrainte dépasse le cadre de notre présentation. Il nous faut tout de même noter que lorsque l'on définit une classe ou une fonction générique ceci implique implicitement que les paramètres génériques satisfont certaines contraintes. Par exemple, supposons que la classe `Vecteur<TElement>` possède une méthode `Vecteur<TElement>::max(): TElement` qui renvoie l'élément maximum du vecteur. Ceci implique implicitement que les composantes du vecteur du type `TElement` soient ordonnées et comparables aux moyen d'une opération de comparaison pour ce type `TElement`.

Même s'il a été montré [112, Appendix B : Genericity versus inheritance] que la généricité pouvait être « émulée » au moyen de l'héritage et du polymorphisme, il n'en reste pas moins que la généricité possède un net avantage dans certains cas d'utilisations [177]. Nous verrons plus en détail les limitations du couple héritage/polymorphisme ainsi que les avantages offerts par la généricité dans ces cas aux paragraphes 4.3 et 4.4 qui constituent une des contributions de cette thèse.

Toutefois, on peut d'ores et déja noter que d'une façon générale lorsqu'un langage de programmation ne supporte pas une fonctionnalité souhaitable de façon récurrente, un motif de conception[14] [76] fini par être identifié. On peut penser que, lorsque ce motif est utilisé couramment, la fonctionnalité qu'il remplace est une bonne candidate comme

14. en anglais : *design pattern*

nouvelle fonctionnalité du langage lui-même [17]. Ce sera peut-être le cas pour la généricité dans le langage Java [79].

Nous venons d'examiner les notions qui caractérisent les modèles de programmation orientée-objet mais la force de l'approche objet ne se situe pas uniquement dans le modèle de programmation associé. En effet, le principal intérêt d'une approche objet est qu'elle offre un grand nombre d'outils et de méthodes qui vont de la conception à la maintenance en passant par la réalisation.

1.4.2 Conception, réalisation, maintenance orientées-objet

Le cycle de vie du logiciel peut être résumé par les étapes suivantes :

1. conception,
2. réalisation,
3. maintenance (corrective et évolutive).

Ces étapes grossières, présentées séquentiellement sont liées entre elles et différents aller/retour sont possibles. Une étape de maintenance corrective entraîne une reprise de la réalisation, une étape de la réalisation peut entraîner une re-conception mais une étape de maintenance évolutive peut également entraîner une re-conception... Les concepts objets facilitent grandement toutes les étapes du cycle de vie du logiciel. Il n'est d'ailleurs pas étonnant que l'approche objet vise à simplifier la conception, la réalisation et la maintenance de systèmes complexes [112].

Le succès de l'orienté-objet a provoqué le développement de nombreux outils ou méthodes qui supportent les démarches objets à tous les niveaux :

- conception
  - méthodes de conception : Booch, HOOD, Objectory Process, OMT, OOSE ...
  - languages de conception graphique indépendant du langage d'implantation : UML
  - languages de description d'interface générique : IDL
- réalisation
  - langages de programmation : Ada 95, C++, Eiffel, Java, Smalltalk ...
  - environnements de développement intégrés
  - bibliothèques de composants réutilisables
- maintenance
  - outils de documentation intégré
  - schémas objets

Cette énumération plus qu'incomplète ne suffit pas à montrer qu'il existe une pléthore d'outils et de méthodes issues des concepts objets. Nous référons au portail Web CETUS [37] qui contient un très grand nombre de liens sur tous les sujets concernant l'orienté-objet.

Il faut donc retenir qu'au delà des concepts, les approches orientées-objet sont aussi très attractives car de nombreux outils et méthodes les supportent. C'est ce genre d'infrastructure que le programmeur d'application parallèles aimerait avoir à sa disposition.

Pour que cela puisse se produire il faut d'abord réussir le mariage des concepts du parallélisme présentés au paragraphe 1.3 avec les concepts orientés-objet que nous venons d'aborder. Le principal objectif étant la définition d'un *modèle de programmation parallèle orienté-objet*. C'est le but poursuivi par les langages à objets parallèles que nous présentons ci-après.

1.5 Les langages à objets parallèles

Le but principal des LAOs parallèles est d'apporter les atouts de la programmation orientée-objet à la programmation parallèle et de lui offrir de ce fait un modèle de programmation de haut niveau. Cet objectif est clairement souhaitable face aux modèles de programmation parallèles actuels qui sont, comme nous l'avons dit (§1.3.2) trop proches des modèles d'exécution des machines parallèles.

1.5.1 Classifications

Les concepts à assembler sont donc les aspects du parallélisme et les concepts objets :

Aspects du parallélisme	Concepts objets
concurrence	abstraction et encapsulation (classes, objets)
synchronisation	héritage
distribution	polymorphisme
communication	généricité

Les choix qui sont fait concernant la réunion de ces aspects et concepts amènent à des classifications des langages à objets parallèles. Nous retiendrons trois classifications possibles :

1. Classification par les moyens d'introduction du parallélisme dans les LAOs [131]
   Cette classification suit l'axe de présentation des moyens de parallélisation présentés au §1.3.3 déjà utilisés pour introduire le parallélisme dans les langages de programmation suivant un modèle de programmation procédural. C'est un moyen simple de voir comment le parallélisme a été introduit dans les LAOs :
   (a) *Parallélisation automatique*
   Les solutions de parallélisation automatique que nous connaissons n'ont rien de spécial concernant les LAOs. Ce sont les mêmes moyens que pour les langages procéduraux. La classe des langages parallèles à objets qui utilisent des directives de compilation se situe toutefois à la frontière entre cette première catégorie et la suivante.
   (b) *Langages parallèles*
   Il existe les 3 mêmes sous-catégories :
      i. *langage à objets parallèle à extensions compatibles*
      ii. *langage à objets parallèle à extensions incompatibles*
      iii. *nouveaux langage à objets parallèle*

(c) *Librairies parallèles*

Ici la différence principale avec les librairies du même style mais non orientées-objet est que les librairies parallèles orientées-objet sont généralement des librairies de classes. Celles-ci essayent donc de tirer au maximum partie des concepts objets, d'abstraction, d'encapsulation, d'héritage et de polymorphisme. Ces librairies de classes offrent les aspects du parallélisme sous la forme de composants objets qui servent de briques de base à la construction d'objets parallèles et distribués. Il est à noter que les concepteurs de librairies parallèles orientées-objet les construisent généralement en utilisant des librairies parallèles non orientées-objet. Les librairies parallèles telles que celles présentées au §1.3.3 sont donc également un bon moyen de parallélisation si elles sont utilisables dans un LAO.

2. Classification de *Papathomas* [148]

Contrairement à la première celle-ci est guidée par le modèle [de programmation] objet. On regarde donc comment les aspects du parallélisme font leur apparition dans les objets. La classification est résumée par l'auteur [148, §6.1] de la façon suivante :

(a) *Modèles objets*

Ils spécifient comment les notions d'objet et de concurrence sont unifiées.

i. *l'approche orthogonale*, les objets sont indépendants de la concurrence et sont seulement des types de données abstraits partagés par des processus parallèles.

ii. *l'approche homogène*, un objet≡un processus, c'est-à-dire que tous les objets sont dit actifs. Chaque objet est une entité d'exécution autonome.

iii. *l'approche hétérogène*, c'est le mélange des deux approches précédentes. Les objets actifs et les objets passifs cohabitent.

(b) *Activité des objets*

Elle spécifie si la concurrence se situe entre les objets seulement ou également à l'intérieur de l'objet.

i. *objets séquentiels*, un objet n'est associé qu'à un seul processus au maximum.

ii. *objets quasi-parallèles*, un objet peut être associé à plusieurs processus mais seul un des processus associés à l'objet peut être actif à un instant donné.

iii. *objets parallèles ou multi-actifs*, un objet est associé à plusieurs processus qui peuvent s'exécuter simultanément.

L'auteur ajoute que la création des processus associés aux objets peut-être implicite ou explicite.

(c) *Interactions entre objets*

La concurrence se fait par le biais des processus associés aux objets. Ces processus exécutent le code des méthodes des objets. En suivant la terminologie de Smalltalk, l'appel de méthode est aussi appelé *envoi de message*. Les objets actifs ont donc plusieurs options pour traiter ces messages de façon concurrente.

i. *envoi de messages simple*, l'appelant appelle une méthode d'un objet et

n'attend pas de retour. Un message doit être envoyé séparemment par l'appelé pour répondre à l'appelant.

ii. *envoi de messages de type requête/réponse*, appel de méthode de type RPC ou par proxy.

iii. *acceptation explicite des messages*, l'appelé choisit explicitement de traiter les messages reçus.

iv. *conditions d'activations*, ces conditions déterminent si l'appelé est dans un état lui permettant de traiter le message.

v. *envoi de message réifié (réfléxivité)*, le message est réifié et traité par le méta-objet associé à l'objet ayant reçu le message.

3. Classification de *Briot, Guerraoui et Löhr* [23]
   C'est certainement la classification la plus complète car elle vise à classer d'une façon générale les méthodes pour marier concurrence, distribution et programmation orientée-objet. Elle est proche de la première classification car elle est orientée suivant l'axe des moyens de réalisation d'applications parallèles orientées-objet et distribuées. Les auteurs distinguent donc trois approches,

   (a) *Approche par librairie*
   Cette catégorie est la même que celle présentée dans la première classification (1c).

   (b) *Approche par intégration*
   Cette approche correspond aux modèles objets homogènes et hétérogènes de la classification 2 de *Papathomas*. En effet comme le notent les auteurs, l'approche par intégration tente de fournir un modèle de programmation orienté-objet parallèle et distribué, la plupart du temps sous la forme d'un langage parallèle (cf. 1b). Les dimensions d'intégration présentées par les auteurs :
   - concurrence des objets
   - synchronisation des objets
   - distribution des objets

   rappellent les branches 2b et 2c du modèle de *Papathomas* et correspondent exactement au choix d'intégration des 3 aspects du parallélisme que sont la concurrence, la synchronisation et la distribution. Une approche intégrée est donc un moyen de plus haut niveau qu'une approche par librairie ou même, d'une certaine façon, qu'une approche réflexive. Ces dernières approches sont d'ailleurs souvent utilisées pour mettre en œuvre une approche par intégration. Dans notre terminologie du parallélisme (Déf. 1.7 page 12) les approches par intégration visent un parallélisme plus implicite puisque unifié avec (voir caché par) les concepts objets.

   (c) *Approche refléxive*
   La réfléxivité est la capacité d'un système à se décrire lui-même. Dans les langages de programmation réfléxifs il existe deux niveaux [152] :
   - le niveau de base, qui permet de programmer l'application au moyen des concepts du modèle de programmation

– le niveau *méta* dont les éléments décrivent le modèle des éléments du niveau de base lui-même. En modifiant les éléments du niveau méta, on modifie globalement le modèle de programmation du niveau de base. Programmer au niveau méta s'appelle *méta-programmation*.

Le protocole de communication entre les niveaux de base et méta dans un LAO est appelé *protocole méta-objet* (MOP[15]) [100]. Les fonctionnalités du protocole méta-objet permettent d'introduire les aspects du parallélisme tels que la distribution ou la synchronisation au niveau méta.

Discussion

La richesse des critères de classification des LAOs nous amènent à penser que cette approche de classement hiérarchique est inadaptée. De plus, comme le remarquent *Briot, Guerraoui et Löhr* [23, §5.4], les systèmes à objets parallèles et distribués et les LAOs parallèles tels que ceux que nous présenterons ci-après utilisent plus ou moins tous les moyens présentés dans les classifications précédentes sans pouvoir rentrer dans une case bien déterminée. Nous pensons, contrairement à [5], qu'une approche par intégration qui permettrait de définir un *unique* modèle de programmation parallèle orientée-objet ne peut pas satisfaire toutes les contraintes de la POO parallèle. En revanche, il nous semble qu'un modèle objet parallèle configurable ou modifiable est l'approche la plus générale et la plus intéressante. C'est le principe même de la programmation ouverte [137] et c'est ce qui fait d'après nous le succès des approches réfléxives ou de programmation par aspects [7]. C'est aussi la différence entre la conception/programmation avec réutilisation (conception/programmation OO) et la conception/programmation ayant pour but de réutiliser (conception de domaine/programmation générative) [47, §3.9 page 59]. Nous reviendrons à la notion de programmation générative au paragraphe 5.5 une fois que nous aurons vues certaines limitations de la programmation orientée-objet (§4.3).

1.5.2 Quelques exemples de LAOs parallèles

Pour les raisons évoquées précédemment, nous n'étudierons pas un représentant de chaque choix issu des classifications précédentes. Nous présenterons donc ci-après quelques exemples qui nous semblent représentatifs et référerons à la littérature [23, 148, 75, 131] pour une couverture plus exhaustive. Nos exemples sont dans la grande majorité des LAOs parallèles dérivés de C++, non pas que ceux dérivant d'autres LAOs ne soient pas autant représentatifs des LAOs parallèles mais le choix de C++ correspond à des critères de disponibilité et d'applicabilité industrielle.

15. **M**eta**O**bject **P**rotocol

HPC++

Références	:	[77, 97, 19]
Caractéristiques	:	directives de compilation, STL parallèle, librairie orientée-objet parallèle, pointeurs généralisés, CORBA IDL, programmation multithread.
URL(s)	:	`http://www.extreme.indiana.edu/hpc++/`

HPC++ est un effort de normalisation américain visant la définition d'un langage à objets parallèle dérivant de C++. Le groupe de réflexion a débuté en 1994 et se nomme *The HPC++ working group*. HPC++ est défini en 2 niveaux : le premier, (HPC++ Level 1) consiste en la spécification d'une librairie et d'outils ne nécessitant pas d'extension au langage C++, le second niveau (HPC++ Level 2) fournit des extensions de langages ainsi que le support d'exécution pour le niveau 1.

Le niveau 1 contient :

- la définition de directives de compilation,
  Celles-ci permettent de spécifier le parallélisme des boucles de la même manière qu'avec OpenMP [144] ou HPF [90] (voir l'annexe A). Un exemple de parallélisation d'un produit matrice vecteur tiré de [77] est donné par le programme 1.6.
- PSTL [97] : une version parallèle de la STL[16].
- une librairie de classes encapsulant les aspects du parallélisme
  - classe de processus légers (`HPCxx_Thread`),
  - classes encapsulant des primitives de synchronisation (`HPCxx_Sync`, `HPCxx_SyncQ`, `HPCxx_CSem`, `HPCxx_Barrier` ...),
  - classe de support pour des opérations collectives (`HPCxx_Group`, `HPCxx_Reduct`, `HPCxx_Barrier`),
  - support pour la programmation distribuée :
    pointeurs globaux (`HPCxx_GlobalPtr<T>`), appel de méthode à distance, ...

Il est intéressant de noter que le langage HPC++ permet, via la version parallèle de la STL (PSTL), le parallélisme de données. Une classe de tableaux distribués `Array` est également proposée afin de compléter PSTL, qui n'offre que des conteneurs monodimensionnels. Des classes, permettant la distribution des tableaux, accompagnent la classe `Array`, comme on peut le voir sur le programme 1.7. Les fonctionnalités de distribution en HPC++ sont directement inspirées de celles offertes par HPF.

HPC++ vise aussi bien les modèles d'exécution à mémoire partagée, en offrant l'abstraction `HPCxx_Thread`, que les modèles à mémoire distribuée, en offrant la notion de pointeur généralisé ou *pointeur global*. Un pointeur global peut être vu comme une référence vers un objet pouvant être dans un espace d'adressage différent de l'objet possédant la référence. On peut voir un exemple simple de ce qu'autorise ces `HPCxx_GlobalPtr` au programme 1.8.

L'utilisation des pointeurs globaux, pour invoquer une méthode d'un objet à distance, n'est pas aussi « naturelle » qu'on le souhaiterait, car on doit passer par l'interface

16. **S**tandard **T**emplate **L**ibrary : librairie de conteneurs génériques faisant partie de la librairie standard du langage C++ [186]

Programme 1.6: *Directives HPC++*

```
1  void Matvec(double **A, int n, double *X, double *Y)
2  {
3    double tmp;
4    // les iterations en i sont independantes si l'on
5    // utilise une copie privee de la variable tmp
6    #pragma HPC_INDEPENDENT, PRIVATE tmp
7    for (int i=0; i < n; i++)
8      {
9        tmp = 0;
10       // les iterations en j sont independantes
11       #pragma HPC_INDEPENDENT
12       for (int j=0; j < n; j++)
13         {
14           // la variable tmp est la cible d'une reduction
15           #pragma HPC_REDUCE
16           tmp += A[i][j]*X[j]
17         }
18       y[i] = tmp;
19     }
20 }
```

Programme 1.7: *Distribution de tableaux en HPC++*

```
1  // Objet definissant la correspondance entre
2  // les processeurs physiques et une grille
3  // virtuelle P, definition par defaut
4  Processor_Map P();
5
6  // On cree un modele de distribution D pour un
7  // tableau bidimensionnel, les lignes sont distribuees
8  // par bloc et les colonnes de facon entiere
9  Distribution D(BLOCK,WHOLE,P);
10
11 // On cree un tableau de reels double precision A
12 // ayant la distribution D
13 Array<double> A(100,100,D);
```

Programme 1.8: *Pointeurs Globaux en HPC++*

```
1 // declaration d'un pointeur global sur un float
2 HPCxx_GlobalPtr<float> p;
3 // On associe p a un objet distant
4 // [...]
5 float  t = *p + 2;// lecture distante + addition locale
6       *p = 3.14;  // ecriture distante
```

`hpcxx_invoke` en ayant, au préalable, enregistré (`hpcxx_register`) les méthodes de la classe. L'utilisation d'un language de définition d'interface (IDL[17]) à la CORBA, permet la génération automatique d'un objet proxy (cas particulier d'un pointeur global) à partir de la définition de l'interface des objets distribués. Un autre avantage de l'utilisation d'un IDL est la génération des fonctions/méthodes (`hpcxx_pack`/`hpcxx_unpack`) nécessaires à la dé/sérialisation des objets entre deux contextes d'exécution.

Remarque **1.2 (Sérialisation)** *Dans les LAOs parallèles, lorsque l'on dispose de la notion d'objet distribué, il est nécessaire de pouvoir* ***sérialiser*** *un objet. Par exemple, lorsque qu'on invoque la méthode* `f` *d'un objet* `DA` *prenant en paramètre un autre objet* `B`*, lors de l'appel* `DA->f(B)`*, ce paramètre doit être transmis dans le contexte d'exécution de* `DA`*. On appelle l'opération de transmission de l'objet la* ***sérialisation****, la mise à plat ou bien en anglais* marshalling*. Grossièrement, cela signifie que l'on doit prendre un à un les champs de données de l'objet et les envoyer, par exemple à l'aide d'une librairie de communication, dans le contexte attendu. L'opération inverse appelée désérialisation, reconstruction ou en anglais* unmarshalling*, consiste à recevoir les données et à les stocker dans un objet du même type préalablement reconstruit.*

La sérialisation automatique nécessite généralement un support de la part du langage (Java) ou bien l'utilisation d'un précompilateur (compilateur IDL ou précompilateur C++ spécifique du LAO parallèle utilisé). En effet, l'objet pour pouvoir se sérialiser lui-même doit avoir des capacités d'introspection[18], c'est-à-dire s'interroger lui-même pour connaître la structure des données de son modèle (la classe). Une solution intermédiaire est d'exiger que le programmeur définisse des classes conformes à une certaine interface (`hpcxx_pack`/`hpcxx_unpack` dans le cas d'HPC++). Cette interface sera automatiquement appelée lorsque cela est nécessaire. Nous reviendrons aux problèmes liés à la sérialisation au §3.5.

HPC++ fournit des outils qui sont principalement au niveau d'une librairie parallèle orientée-objet. Cette librairie propose une encapsulation objet des différents aspects du parallélisme. Les modèles de programmation parallèles supportés sont le modèle SPMD et le modèle à mémoire partagée multithreadé, ces modèles sont encore dans leur grande majorité à parallélisme explicite, comme le montre les quelques exemples de programmes

17. **I**nterface **D**efinition **L**anguage

18. c'est un cas particulier de la réfléxivité qui se situe plutôt à l'exécution [152, *§3.4 The Time of Reflection*]

Programme 1.9: *Distribution de données en pC++*

```
// Les processeurs
Processors P;
// Definition du patron de distribution
Template    myTemplate(7,7,&P,BLOCK,WHOLE);
// Specification de l'alignement
Align       myAlign(5,5,"[ALIGN(dummy[i][j],myTemplate[i][j])]");
// Creation d'un objet de type collection
// dont la distribution est aligne suivant l'objet myAlign
// sur le patron de distribution myTemplate
DistributedMatrix<Complex> A(&myTemplate, &myAlign);
```

précédents. L'encapsulation objet apporte la modularité et la facilité d'utilisation d'une interface souvent plus naturelle et plus riche que celle des librairies parallèles comme MPI (voir §1.3.3 et annexe B).

pC++

Références	:	[21, 153]
Caractéristiques	:	Parallélisme de donnée, agrégat distribué, extension de language, programmation SPMD.
URL(s)	:	`http://www.extreme.indiana.edu/sage/index.html`

pC++ est en quelque sorte un prédécesseur d'HPC++. pC++ introduit le parallélisme dans C++ par une extension de langage et la définition d'une classe parallèle appelée `collection` (cf programme 1.3 page 22). La classe `collection` est du type agrégat distribué, c'est un objet réparti dont certaines méthodes peuvent être appliquées en parallèle à toutes les parties de l'agrégat. Un agrégat distribué est un tableau que l'on peut distribuer sur motif de distribution appelée `Template`[19] (voir HPF à l'annexe A) qui est lui-même un objet. En s'inspirant des directives de distribution des données de HPF, les concepteurs de pC++ offrent un ensemble d'objets et de constructeurs très bien adaptés à la distribution (programme 1.9) et à l'alignement des objets du type `collection`.

La définition des classes de type `collection` nécessite l'introduction de nouveaux mots réservés du langage, notamment `MethodOfElement` et `ElementType`. Le mot clé `ElementType` permet d'utiliser un type générique pour l'écriture des `collection`s à la manière des templates C++. Le champ `MethodOfElement` permet de spécifier les méthodes qui s'appliqueront aux différentes parties des classes `collection`s, comme le montre le programme 1.3 page 22.

pC++ n'est pas un exemple récent mais il illustre bien l'effort des LAOs parallèles cherchant à supporter au mieux le modèle de programmation à parallélisme de données. Les principaux reproches que l'on aurait pu faire à pC++ est le choix des extensions de

19. cette classe d'objet pC++ n'a rien à voir avec les `template` C++ qui sont le support à la généricité en C++

langages et le non-support du parallélisme de tâches. Les extensions de langages empêchent définitivement la compilation d'un programme pC++ par un compilateur C++ standard sans le précompilateur pC++. De plus, nous avons déjà vu (§1.3.2) que le parallélisme de données seul ne pouvait suffir.

EPEE

Références	:	[96, 83, 147]
Caractéristiques	:	Parallélisme de données, agrégat distribué, basé sur le langage Eiffel, programmation SPMD, librairies de classes.
URL(s)	:	`http://www.irisa.fr/pampa/EPEE/epee.html`

EPEE[20] regroupe une librairie et tous les outils nécessaires à l'exécution parallèle d'Eiffel. Aucune modification du langage Eiffel n'est effectué. Le parallélisme est introduit par héritage multiple (fig. 1.9) de classes fournies dans une librairie parallèle. La librairie EPEE encapsule dans des classes les aspects du parallélisme offrant ainsi une interface séquentielle au programmeur. Bien qu'Eiffel ne soit pas un langage très utilisé dans le domaine du calcul numérique les auteurs ont testé leur approche sur la réalisation d'une librairie parallèle pour l'algèbre linéaire en Eiffel : Paladin [83]. Les objectifs poursuivis par *F. Guidec* pour la conception de Paladin sont proches des notres pour la conception de LAKe (voir chapitres 3 et 4) : encapsulation du parallélisme dans une librairie de classe, application d'algèbre linéaire, implantation efficace. La principale différence est que la réalisation de notre librairie de classes de matrices et vecteurs distribuées n'est pour nous qu'un moyen d'accéder à la réutilisabilité séquentielle/parallèle des codes et algorithmes impliquants des matrices creuses dans les méthodes itératives. Cela nous a également permis de proposer une méthodologie (Chap. 5) pour la conception d'applications parallèles réutilisables. On peut noter que le choix de la programmation SPMD, comme modèle de base rend simple la mise en place d'agrégats distribués. En effet, le programme initial étant exécuté sur chaque processeur, le constructeur d'un objet agrégat distribué crée un objet local à chaque processeur capable de contenir les parties locales de l'agrégat. A la manière d'un programme SPMD en MPI, le constructeur de l'objet distribué peut tester sur quel processeur il se trouve et créer la partie locale de l'objet en conséquence. L'encapsulation de la distribution et des communications dans des objets distribués facilite ensuite l'utilisation des objets parallèles. *J.L. Pacherie* a poursuivi les efforts relatifs à EPEE en étudiant des systèmes de motifs pour l'expression et l'évaluation des opérations sur des collections distribuées [147]. Les formalismes introduit par *J.L. Pacherie* (morphismes de monoïdes) permettent la parallélisation des traitements sur les collections en partant du principe que toutes les collections sont des énumérations. A la manière de la STL [186], l'introduction des itérateurs sur des collections permet le découplage entre les algorithmes (à paralléliser) et les structures de données initiales (les collections). La difficulté d'utilisation du formalisme et des motifs introduit tient au fait que pour implanter une opération du type « produit matrice vecteur » il faut retrouver quel ensemble de motifs et de morphismes la réalisent. Même si cette approche est certai-

20. **E**nvironnement **P**arallèle d'**E**xécution d'**E**iffel

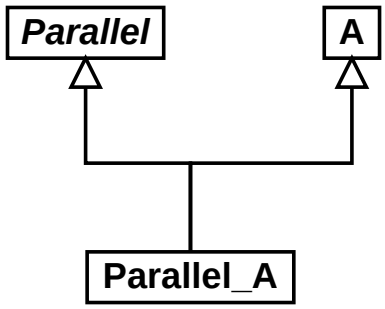

FIG. 1.9 – *Parallélisme et héritage multiple*

nement très bonne pour générer du code SPMD à partir d'un code séquentiel une fois les motifs et morphismes identifiés, elle nous a apparu comme peu naturelle et trop loin de nos objectifs pour la mettre en œuvre dans notre librairie.

EPEE présente l'intérêt de montrer la faisabilité de supporter un modèle de programmation SPMD avec une librairie objet parallèle. C'est en quelque sorte une version « librairie » des fonctionnalités offertes par pC++.

C++//

Références	:	[33, 193]
Caractéristiques	:	parallélisme de contrôle via héritage multiple, modèle mixte objets actifs/passifs, programmation MIMD, synchronisation par future, MOP, proxy.
URL(s)	:	`http://www.inria.fr/oasis/c++ll/`

C++// est à la fois un langage parallèle étendant C++ sans en changer la syntaxe et un système de précompilation. Le système suit les recommandations de la proposition d'architecture *Europa C++* [193] à savoir un modèle à 2 niveaux :

- Level 0
  C'est la couche juste au dessus du modèle d'exécution de la machine cible. Cette couche définit un modèle de proxy qui est un méta-objet permettant la réification des appels aux méthodes des objets parallèles. Les proxy sont générés par le système de précompilation.
- Level 1
  C'est le niveau du modèle de programmation qui fournit les abstractions nécessaires aux modèles (Objets Actifs, SPMD, BSP, ...)

En C++// le parallélisme est introduit par héritage multiple (voir programme 1.10), la librairie de classes parallèles étant fournie avec le système de compilation. Une classe `A` peut devenir parallèle en héritant de la classe `Process`, fournie par le système de compilation de C++//. Par défaut la classe `Process` possède une méthode `Live` qui est exécutée à sa création. Cette méthode est en fait une boucle infinie qui attend des appels aux méthodes de l'objet auquel elle appartient (dans l'exemple les méthodes publiques de

Programme 1.10: *Parallélisme et héritage multiple en C++//*

```
class Parallel_A : public A, public Process
{
  virtual void Live() {
    // corps de Process
  }
};

// [...]
Parallel_A *p;
p = new Parallel_A(...)
```

Programme 1.11: *Synchronisation en C++//*

```
v = p->f(param); //  appel asynchrone
// [...]

v->action(); /* attente implicite si v  *
                * est attendu Awaited     */
// [...]
// teste (explicitement) le statut de v
if (Awaited(v)) {
// [...]
}
// [...]
// attente explicite si v est attendu
Wait(v);
// [...]
```

`Parallel_A`). `Live` traite ces appels suivant la politique qu'on lui aura défini ; par défaut si l'on ne ré-écrit pas la méthode virtuelle `Live`, la politique est FIFO[21].

C++// contient des objets actifs et passifs, les objets actifs communiquent via des appels asynchrones à leurs méthodes. On peut voir un schéma des processus et objets durant une exécution à la figure 1.10.

La synchronisation est simplifiée par un système d'attente par nécessité : une variable mise à jour de façon asynchrone ne pourra être utilisée que lorsque cette mise à jour aura effectivement été exécutée. Une variable peut ne pas être « à jour »[22] si elle est le résultat d'un appel asynchrone à une méthode d'un objet actif. Des fonctions `Awaited` et `Wait`, prenant en paramètre la variable potentiellement attendue, permettent une synchronisation explicite, voir le programme 1.11.

21. First In First Out, où la première méthode appelée sera la première exécutée
22. on appelle aussi ces variables des « *future* »

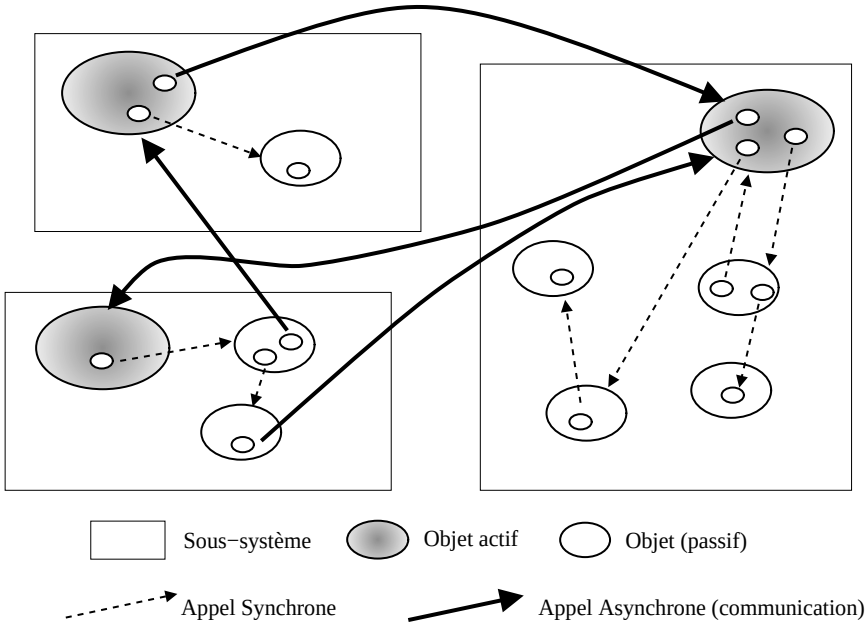

FIG. 1.10 – *Objets actifs/passifs en C++//*

L'intérêt principal de C++//, par rapport aux exemples précédents est que la parallélisation d'un code est non-intrusive. Ceci signifie que la parallélisation d'un code séquentiel ne modifie pas le code séquentiel lui-même comme le ferait l'utilisation d'une librairie parallèle comme MPI (voir programme 1.4) ou même HPC++lib. Dans l'exemple du programme 1.10, on dérive la classe `Parallel_A` *sans modifier* la classe `A`. Le processus de compilation utilisé par C++// rentre dans le cadre du schéma présenté à la figure 1.11. Un processus de compilation de ce type permet une parallélisation non-intrusive d'un code séquentiel. En effet, les aspects du parallélisme sont encapsulés dans la librairie parallèle associée au langage parallèle. Les classes séquentielles sont parallélisées par héritage multiple et le code nécessaire à la mise en œuvre du MOP est généré par le précompilateur du LAO parallèle. De cette façon, les classes séquentielles peuvent évoluer quasiment indépendamment de leurs homologues parallèles. « Re-paralléliser » des classes séquentielles ayant évolué ne nécessite qu'une recompilation. Comme il est noté dans [23], nous constatons que des approches mixtes mélangeant librairie parallèle, approche intégrative parallèle et réfléxivité offrent des avantages par rapport à des solutions non mixtes.

C++// est un représentant des langages d'acteurs avec un modèle objet hétérogène où les objets actifs et passifs coexistent.

Nous verrons plus en détails les problèmes de performances liés au modèle à objets actifs/passifs et à la sémantique du passage de paramètres qui l'accompagne au paragraphe 3.5 qui constitue l'une des contributions de cette thèse. Ce travail a fait l'objet d'une collaboration avec *Denis Caromel* et *David Sagnol* [34, 35].

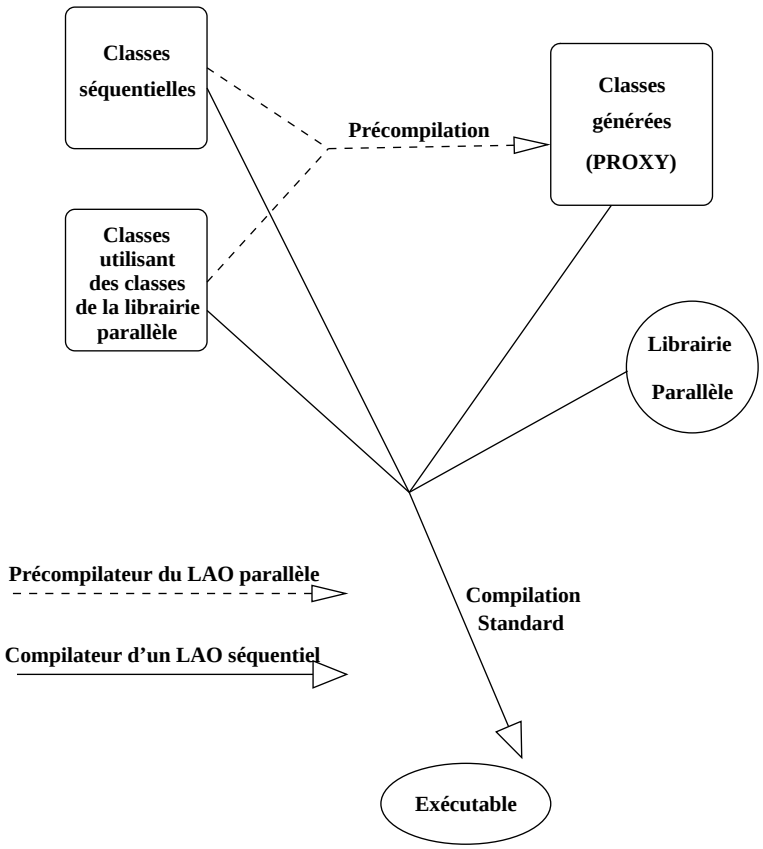

FIG. 1.11 – *Processus de compilation parallèle non-intrusif*

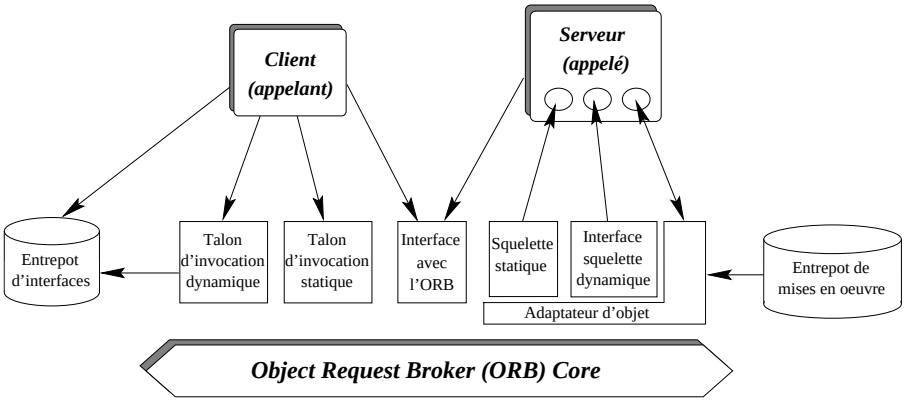

FIG. 1.12 – *ORB CORBA 2.x*

CORBA

Références	:	[132, 139, 146, 50]
Caractéristiques	:	Middleware, Architecture 3-tiers, Bus Logiciel, Invocation de méthode à distance, IDL.
URL(s)	:	`http://www.omg.org/`

CORBA (**C**ommon **O**bject **R**equest **B**roker **A**rchitecture) n'est pas un LAO parallèle mais la définition d'une architecture logicielle orientée-objet permettant l'inter-opérabilité entre objets dans un système réparti. Le coeur de CORBA est la définition d'un bus logiciel ou ORB (**O**bject **R**equest **B**roker), permettant le dialogue (par exemple invocation de méthode à distance) entre des objets répartis sur des systèmes distants, éventuellement hétérogènes. La normalisation en est à sa version 2.3 (3.0 en cours). Ce travail de normalisation est réalisé par l'OMG (***O**bject **M**anagement **G**roup*) qui regroupe actuellement plus de 500 acteurs du monde objet (vendeurs, développeurs et utilisateurs) [146, 50].

CORBA est un standard industriel dont les objectifs premiers sont différents de ceux des LAOs parallèles que nous venons de présenter brièvement. CORBA a pour objectif premier de faciliter la construction d'applications distribuées à grande échelle en privilégiant l'interopérabilité et l'hétérogéneité. Clairement la performance des applications n'est pas un des objectifs premiers de CORBA même si la volonté d'introduire des fonctionnalités de haute performance dans CORBA apparaît aussi bien au travers d'applications [157, 18, 98, 116] qu'au sein du processus de normalisation de CORBA [145, 133]. Malgré ces différences initiales, il s'avère que de nombreuses similitudes existent entre les concepts utilisés dans CORBA et dans de nombreux LAOs parallèles. Si l'on compare l'architecture de CORBA 2.x de la figure 1.12 avec celle du modèle à objets actifs/passifs de C++// de la figure 1.10 on voit que les objets actifs de C++// pourraient être des objets serveur ou client CORBA suivant si ils sont appelés ou appelant. L'architecture CORBA est donc un modèle à objets actifs, avec appel de méthode à distance. Les talons de CORBA sont les proxy de C++// ou les « global pointers » d'HPC++, les squelettes sont inutiles en C++// car aucune hétérogénéité n'est prévue.

Inversement, l'interopérabilité de CORBA passe par l'utilisation d'un IDL qui permet

à partir d'un code source unique de générer les talons et squelettes pour les langages cibles du client et du serveur. Une approche similaire dans l'utilisation d'un IDL pour l'interopérabilité des codes scientifiques parallèles écrit en C/C++ et Fortran a été étudié [183]. Cela permet également de générer les procédures de sérialisation et de désérialisation des objets CORBA passés en paramètre d'un appel à une méthode d'un objet serveur. On retrouve cette idée dans l'IDL utilisé par HPC++ dans le même but. En C++// c'est le précompilateur qui inspectera le code C++ directement afin d'extraire les informations qui seraient contenues dans la définition d'une interface IDL ; une fois cette étape effectuée l'objectif est le même : générer le code d'un proxy. On peut donc faire un parallèle entre le schéma de précompilation de la figure 1.11 et l'utilisation d'un compilateur IDL.

On retrouve les mêmes problématiques dans les applications CORBA que dans des applications de calcul scientifique parallèle. CORBA étant plus approprié pour du parallélisme à gros grain et les autres approches de LAOs parallèles pour le parallélisme à grain fin. Il manque à CORBA la notion d'objet parallèle et des extensions de l'IDL CORBA ont été étudiées en ce sens [156, 183]. CORBA inspire et inspirera les LAOs parallèles visant le calcul à haute performance et inversement [145, 133]. Il est donc probable que CORBA joue un rôle grandissant dans le domaine du calcul scientifique parallèle et c'est pourquoi nous devions le présenter brièvement même si nous n'avons pas utilisé CORBA pour réaliser les applications tests de notre étude.

Nous avons examiné caractéristiques du parallélisme et de la programmation à objet, nous avons ensuite constaté les nombreuses solutions pour la définition d'un modèle de programmation parallèle à objet. Le but principal de notre étude étant de savoir comment les approches orientées-objet peuvent aider à une meilleure réutilisabilité des codes parallèles, nous examinerons dans le paragraphe suivant les problèmes posés par le développement et la maintenance de ces codes.

1.6 Développement et maintenance des applications parallèles

Un problème crucial dans le cycle de vie d'un code parallèle est sa maintenance car le lien trop étroit qui jusqu'à présent liait les modèles d'exécution et de programmation parallèles complique notablement le support d'un code devant suivre l'évolution des machines [197]. Les critères à observer pour le choix d'un modèle de programmation parallèle sont les mêmes que pour un modèle séquentiel (voir §1.2.1) mais certains sont plus fortement liés à l'introduction du parallélisme.

REMARQUE **1.3** *Dans un contexte industriel on peut dénombrer trois profils de personnes concernées par le cycle de vie d'un logiciel :*

1. ***le développeur spécialiste du domaine d'application***
   Une personne qui connaît bien le domaine d'application (neutronique, chimie moléculaire, électromagnétisme ...) conçoit et ajoute des fonctionnalités au code. Le nombre de développeurs spécialistes d'un code peut varier pendant la durée de vie du code.
2. ***le responsable de la maintenance***

Une personne qui connaît bien le code et le domaine d'application et qui est capable d'apporter des corrections (bug) ou de faire l'intégration de nouvelles fonctionnalités avec des développeurs. Pour un même code, le responsable de la maintenance peut changer dans le temps et ce n'est pas forcément un développeur du code qu'il maintient.

3. ***le spécialiste du parallélisme***
   Une personne qui aura parallélisé le code ou qui aura créé une [nouvelle] version parallèle du code en collaboration avec les développeurs du code séquentiel. Il n'est pas forcément un spécialiste du domaine d'application et n'est pas nécessairement responsable de la maintenance du code [parallèle].

Idéalement ces 3 types de personnes doivent pouvoir collaborer sans acquérir toutes les compétences : connaissance du domaine d'application, connaissance du/des langage(s) informatique(s), connaissance du parallélisme.

Lors du développement d'un code parallèle ou de la parallélisation d'un code séquentiel les choix d'implantation et plus particulièrement de parallélisation guideront les critères de qualité parallèle du code :

1. *efficacité parallèle* : plus le code est exécuté sur un grand nombre de processeurs plus la résolution du problème doit être rapide. C'est souvent le critère premier pour les applications de calcul scientifique car l'implantation parallèle d'un code est généralement faite dans le but de résoudre plus rapidement un problème[23].
2. *portabilité* : c'est la capacité du code parallèle à changer de support d'exécution parallèle (librairie parallèle, langage parallèle...). C'est un critère important pour la durée de vie du code vue l'évolution rapide des machines parallèles [197].
3. *maintenabilité (corrective et évolutive)* : il faut que le code parallélisé soit maintenable et donc compréhensible par le responsable de la maintenance du code. Le problème principal est que le responsable du code n'est pas forcément un spécialiste du parallélisme. Ceci implique que le code parallèle (ou parallélisé) doît pouvoir être modifié, corrigé, étendu par un spécialiste du domaine d'application qui ne maîtrise pas forcément complètement les aspects parallèles du code. La modularité et l'encapsulation inhérentes à une approche objet devraient permettre d'isoler les aspects parallèles du code qui seront maintenus par un spécialiste du parallélisme.
4. *réutilisabilité*
   (a) *réutilisabilité parallèle* : lorsque l'on a construit une application parallèle on doit pouvoir réutiliser les modules qui la composent dans d'autres applications comme c'est le cas pour des applications séquentielles orientées-objet. Le mécanisme principal de réutilisation dans les approches objets est l'héritage, malheureusement l'héritage de certains aspects du parallélisme comme la synchronisation est difficile [23, §3.6.1].
   (b) *réutilisabilité séquentielle/parallèle* : lorsque l'on a parallélisé un code séquentiel et que, dans le même temps, l'application séquentielle a évolué il est souhaitable que la parallélisation des nouvelles parties séquentielles puisse s'appuyer sur la

23. la taille du problème peut également être telle que sa résolution sur machine séquentielle est impossible, mais même dans ce cas la performance est primordiale

parallélisation de l'existant. Ce critère est lié à la maintenance. Les versions parallèles et séquentielles d'un code doivent pouvoir évoluer de conserve.

(c) *polymorphisme séquentiel/parallèle* : lorsqu'un objet parallèle est polymorphe avec un objet séquentiel les objets parallèles doivent pouvoir être utilisés de façon polymorphe et transparente pour des clients des objets séquentiels. C'est une des contributions de cette thèse que de donner des solutions efficaces à ce problème (voir §4.3)

1.6.1 Choix de parallélisation

Les choix de parallélisation (modèle et moyen de programmation parallèle) actuels permettent de privilégier un ou plusieurs critères de qualité parallèle du code parallélisé. La démarche que nous proposons au chapitre 5 contribuera, nous l'espérons, à mieux satisfaire les critères de qualité que nous nous sommes fixés. Pour une maintenabilité maximale on choisira un moyen de parallélisation à parallélisme implicite comme la parallélisation automatique, si la performance prime sur les autres critères on choisira certainement un moyen de parallélisation non portable et explicite pour le support d'exécution visé. Au final, le problème sera toujours de trouver le meilleur compromis.

Dans le cadre de cette thèse, nos critères primordiaux sont :

- la réutilisabilité
- la maintenance
- la performance
- la portabilité

tous ayant à peu près la même priorité dans le but de faciliter la maintenance des codes parallèles. La réutilisabilité et la maintenance seront issues de l'utilisation d'une approche et d'un langage de programmation orientée-objet. La portabilité nous oblige à choisir un moyen de parallélisation standard et utilisable avec un LAO disponible sur une majorité d'architecture parallèle. Ces contraintes nous ont amené à choisir le langage C++[24] comme langage de programmation. Concernant le moyen de parallélisation, les critères de portabilité et de performance nous ont guidés vers 2 solutions :

- un LAO parallèle dérivé de C++ : **C++//**
- une librairie parallèle utilisable avec C++ : **MPI + C++**

Nous avons essayé les deux approches en visant à satisfaire au maximum la réutilisabilité et la maintenance puisque c'est notre objectif premier dans l'utilisation d'une approche objet pour les applications numériques parallèles. C'est-à-dire que nous avons mis l'accent sur la réutilisabilité séquentielle/parallèle au service de la maintenance des codes parallèles.

1.7 Conclusion

La réutilisabilité des codes n'est pas seulement le résultat du modèle de programmation. Le mécanisme d'héritage est le support à la réutilisation mais la méthode de

24. Java aurait peut-être été un choix possible si nos travaux avaient démarré plus tard

conception doit prévoir et permettre la bonne mise en œuvre de ce mécanisme. Nous proposerons donc au chapitre 5 une méthode de développement objet intégrant le parallélisme et rendant possible la réutilisation séquentielle/parallèle. Avant cela nous aurons examiné aux chapitres 3 et 4 deux solutions de réalisation de notre librairie d'algèbre linéaire qui soulèveront chacunes des problèmes que nous résoudrons. Mais avant de commencer la conception de notre librairie d'algèbre linéaire nous présentons dans le chapitre suivant l'analyse des méthodes qu'elle devra implanter.

Chapitre 2

Méthodes itératives d'algèbre linéaire

Sommaire

2.1 Introduction

L'objectif principal de cette thèse est d'évaluer les avantages d'une approche orientée-objet pour les applications de calcul scientifique parallèle. Notre domaine d'application est plus particulièrement le domaine des méthodes itératives de résolution des problèmes d'algèbre linéaire creux. Notre application test est une librairie parallèle d'algèbre linéaire que nous appellerons LAKe pour **L**inear **A**lgebra **Ke**rnel. Ce chapitre fixe les objectifs de notre application (§2.2), et présente les caractéristiques des méthodes itératives d'algèbre linéaire (§2.5) à partir d'un de leur représentant (§2.4).

2.2 Objectifs de LAKe

Les objectifs de LAKe sont multiples mais l'objectif principal est d'obtenir la meilleure réutilisabilité séquentielle/parallèle qui soit *sans sacrifier* les performances séquentielles ou parallèles.

Les problèmes qui nous concernent en algèbre linéaire sont de deux types :

1. La résolution de systèmes linéaires
   Ce problème consiste à trouver la solution x de l'équation (2.1).

$$Ax = b \tag{2.1}$$

   où

$$A \in \mathbb{R}^{n \times n} (\text{ou } \mathbb{C}^{n \times n})$$
$$x \in \mathbb{R}^{n} (\text{ou } \mathbb{C}^{n})$$
$$b \in \mathbb{R}^{n} (\text{ou } \mathbb{C}^{n})$$

2. Le problème de valeurs propres de matrices
   Ce problème consiste à trouver un certain nombre de couples (λ_i, u_i) solutions de l'équation (2.2).

$$Au = \lambda u \tag{2.2}$$

   où

$$A \in \mathbb{R}^{n \times n} (\text{ou } \mathbb{C}^{n \times n})$$
$$u \in \mathbb{R}^{n} (\text{ou } \mathbb{C}^{n})$$
$$\lambda \in \mathbb{C}$$

De nombreuses méthodes de résolution de ces problèmes existent, dont on peut dériver deux sous-ensembles : les méthodes directes de résolution de systèmes linéaires et les méthodes itératives. Nous nous intéresserons aux méthodes itératives car elles sont bien adaptées aux problèmes creux et de très grande taille que nous souhaitons résoudre. En effet, dans notre cas la matrice A des équations (2.1) et (2.2) est de grande taille (n grand) et très creuse (Nombre$(a_{ij} \neq 0)$ petit devant n^2). Dans la suite de ce chapitre nous considérerons que la matrice A est complexe, sauf indication explicite.

Parmi les méthodes itératives, une classe incontournable est celle des méthodes basées sur les projections sur les sous-espaces de Krylov [168, 165]. Le domaine d'application de LAKe est l'implantation de ces méthodes. LAKe devra constituer les briques de base pour l'implantations de ces méthodes aussi bien en parallèle qu'en séquentiel. Comme nous le verrons par la suite les différentes méthodes de Krylov ont de nombreuses similitudes qui devront se traduire par autant de réutilisation de code dans LAKe. Nous verrons également que dans ces méthodes le parallélisme est localisé de telle façon, que l'implantation parallèle ou séquentielle de ces méthodes devrait idéalement être identique.

Nous présentons ci-après brièvement ce que sont les méthodes de projections sur les sous-espaces de Krylov (§2.3) puis nous examinons plus en détail la méthode d'Arnoldi (§2.4).

2.3 Méthodes de projection et sous espaces de Krylov

Une méthode de projection consiste à réduire la taille d'un problème initial en le projetant sur un espace $\mathcal{K}$ orthogonalement à un espace $\mathcal{L}$. On résout ensuite le problème projeté et on replonge la solution trouvée dans l'espace de départ en la considérant comme solution approchée du problème initial. La résolution du système linéaire devient : *trouver $\tilde{x}$ solution de (2.3).*

$$\begin{cases} \tilde{x} \in \mathcal{K} \\ \rho = b - A\tilde{x} \perp \mathcal{L} \end{cases} \tag{2.3}$$

La résolution du problème de valeurs propres devient : *trouver les couples $(\tilde{\lambda}_i,\tilde{u}_i)$ solutions de (2.4).*

$$\begin{cases} \tilde{\lambda}_i \in \mathbb{C},\tilde{u}_i \in \mathcal{K} \\ (A - \tilde{\lambda}_i I)\tilde{u}_i \perp \mathcal{L} \end{cases} \tag{2.4}$$

Lorsque $\mathcal{K} \neq \mathcal{L}$ on dit que la projection est oblique et les conditions d'orthogonalités sont appelées conditions de Petrov-Galerkin, si $\mathcal{K} = \mathcal{L}$ on parle de projection orthogonale et de conditions de Galerkin [168, p. 124].

Les choix des espaces $\mathcal{K}$ et $\mathcal{L}$ sont divers et variés, dans leur construction ou leur dimension[25], les cas qui nous intéressent sont ceux où $\mathcal{K}$ est un sous-espace de Krylov. Le sous-espace de Krylov d'ordre m généré par A et v est l'espace engendré par les vecteurs $v,Av,A^2v,\ldots,A^{m-1}v$, on le note $\mathcal{K}_m(A,v)$ ou simplement $\mathcal{K}_m$ s'il n'y a pas ambiguïté.

$$\mathcal{K}_m = \mathcal{K}_m(A,v) = \text{lin}\left\{v,Av,A^2v,\ldots,A^{m-1}v\right\} \tag{2.5}$$

Lorsque l'on utilise les sous-espaces de Krylov comme espaces de projection, il est nécessaire de construire une base pour chaque sous-espace $\mathcal{K}_m$ utilisé, même si on ne conserve pas tous les vecteurs de base[26]. On notera généralement $\{v_i\}_{1\leq i\leq m}$ la base de l'espace $\mathcal{K}$ sur lequel on projette et $\{w_i\}_{1\leq i\leq m}$ la base de l'espace $\mathcal{L}$ servant dans la condition de Petrov-Galerkin. On notera V_m et W_m les matrices dont les colonnes sont les vecteurs des bases respectives, comme indiqué pour V_m à l'équation (2.6).

$$V_m = [v_1\ v_2\ \cdots\ v_m] \tag{2.6}$$

Si on utilise une projection orthogonale on aura $W_m = V_m$.

Nous allons maintenant rappeler la méthode de base à laquelle nous nous intéresserons plus particulièrement, la méthode d'Arnoldi.

2.4 La méthode d'Arnoldi

La méthode d'Arnoldi a été proposée en 1951 par *W. Arnoldi* [9] pour réduire une matrice A, non hermitienne, en une matrice H de structure plus simple dite de Hessenberg

25. $\mathcal{K}$ et $\mathcal{L}$ sont le plus souvent de même dimension m, mais m varie.
26. voir par exemple CG [86]

(Déf. 2.1). La matrice H est construite itérativement, et Arnoldi suggéra que les valeurs propres des matrices intermédiaires pouvaient être de bonnes approximations de certaines valeurs propres de A.

DÉFINITION **2.1 (Matrice de Hessenberg)** *$H = (h_{ij})_{1\leq i\leq n, 1\leq j\leq n}$ est une matrice de Hessenberg supérieure (resp. inférieure) si la condition (2.7) (resp. (2.8)) est vérifiée*

$$\forall i > j+1, h_{ij} = 0 \tag{2.7}$$

$$\forall i+1 < j, h_{ij} = 0 \tag{2.8}$$

Dans la suite, si l'on ne précise pas, on appellera matrice de Hessenberg une matrice de Hessenberg supérieure.

2.4.1 Processus d'Arnoldi

Le processus d'Arnoldi consiste en fait à orthogonaliser les vecteurs de la base naturelle de Krylov $v, Av, A^2v, \ldots$ par le procédé de Gram-Schmidt. Nous présentons l'algorithme 2.1 qui en résulte en ayant toutefois remplacé le procédé de Gram-Schmidt classique (CGS) par sa version modifié (Modified GS = MGS), plus stable numériquement.

ALGORITHME 2.1 – **Processus d'Arnoldi-MGS**

1) Initialisation :

Choisir un vecteur initial v_1 de norme 1.

Choisir la taille m du sous-espace.

2) Itération :

Pour $j = 1,2,\ldots,m$

(a) $w = Av_j$

(b) *Pour $i = 1,\ldots,j$*

$h_{ij} = (w, v_i)$

$w = w - h_{ij}v_i$

Fin Pour i

(c) $h_{j+1,j} = \|w\|_2$

Si $(h_{j+1,j} = 0)$

STOP

Sinon

(d) $v_{j+1} = w/h_{j+1,j}$

Fin Si

Fin Pour j

Lorsque l'algorithme 2.1 se termine, on a formé une matrice de Hessenberg H_m qui vérifie la relation d'Arnoldi (2.9), dans laquelle e_m est la dernière colonne de la matrice identité I_m de dimension m.

$$AV_m = V_m H_m + h_{m,m+1} v_{m+1} e_m^H \quad (2.9)$$

$$V_m^H A V_m = H_m \quad (2.10)$$

Une fois la matrice H_m obtenue on peut alors :

- chercher les valeurs propres $\tilde{\lambda}_i$ et vecteurs propres z_i de H_m. Les $\tilde{\lambda}_i$ obtenues et les vecteurs $\tilde{u}_i = V_m z_i$ sont les paires (valeurs, vecteurs) propres approchées[27] de A par *la méthode itérative d'Arnoldi pour la recherche des éléments propres*. La méthode itérative d'Arnoldi pour la recherche des éléments propres, consiste à redémarrer[28] la méthode avec un nouveau vecteur de départ v, choisi judicieusement [165, 184, 115]. La méthode est décrite par l'algorithme 2.2
- chercher une solution approchée, $\tilde{x}$ du système linéaire (2.1) de la forme $\tilde{x} = V_m y$, avec $y \in \mathbb{C}^m$. Pour résoudre le problème il faut imposer une condition d'orthogonalité comme nous l'avons présentée au paragraphe 2.3. Si l'on choisit $\mathcal{L} = \mathcal{K}_m$ on obtient une méthode appelée FOM (**F**ull **O**rthogonalization **M**ethod [168]), si l'on choisit $\mathcal{L} = A\mathcal{K}_m$ on obtient la méthode appelée GMRES(m) (**G**eneral **M**inimum **RES**idual, [169]). Le processus d'Arnoldi a donc été ré-utilisé dans GMRES et ses dérivés [166, 168, 105] qui constituent la famille de méthodes itératives des plus utilisée pour la résolution de systèmes linéaires non symétriques.

ALGORITHME 2.2 – **Méthode Itérative d'Arnoldi**

1) Initialisation :

(a) Choisir le nombre r de paires de Ritz souhaitées

(b) Choisir un vecteur initial v_1 de norme 1.

(c) Choisir la taille m du sous-espace de Krylov $\mathcal{K}_m(A,v_1)$.

2) Projection :

(a) Réduire la matrice A sous sa forme de Hessenberg

$H_m = V_m^H A V_m$, par l'algorithme 2.1 (Processus d'Arnoldi)

3) Calcul des éléments propres :

(a) Calculer les m valeurs propres λ_i^m de H_m

ainsi que les m vecteurs propres z_i^m associés, $1 \leq i \leq m$

$H_m Z_m = Z_m \Lambda_m$, avec $\Lambda_m = diag(\lambda_i^m), 1 \leq i \leq m$

(b) Calculer les r vecteurs de Ritz $\tilde{u}_i$ de A, $\tilde{u}_i, 1 \leq i \leq r$ à partir de ceux de H_m,

Pour $r = m$ on a : $\tilde{U}_m = V_m Z_m$

27. on les appelle également valeurs et vecteurs de Ritz de A
28. « restart » en Anglais

Les valeurs de Ritz $\tilde{\lambda}_i$ *de* A *sont les valeurs propres* λ_i^m *de* H_m

4) Calcul des critères d'arrêt :

(a) Calcul des résidus pour les r *paires de Ritz souhaitées,*

$\varepsilon_i = \frac{\|(A-\tilde{\lambda}_i I)\tilde{u}_i\|}{\|A\|}$, $1 \leq i \leq r$

$\varepsilon = f(\varepsilon_1, \ldots, \varepsilon_r)$

(b) Si $(\varepsilon \leq tol_\varepsilon)$ *Alors*

STOP

Sinon 5)

5) Redémarrage

(a) Choisir une méthode de redémarrage

(cette étape produit implicitement ou explicitement un nouveau vecteur v_1*)*

(b) Redémarrer l'algorithme en 1.c)

Il peut être intéressant de travailler sur des blocs de lignes ou de colonnes de la matrice, soit pour des raisons de performances (utilisation de mémoire cache) ou bien parce qu'on résout un système linéaire comportant plusieurs second membres, i.e. $b \in \mathbb{R}^{n \times s}$ avec $s > 1$ contrairement à ce qu'on avait supposé précédemment aux équations (2.1) et (2.2). Une nouvelle version de l'algorithme 2.1 permet de travailler sur des blocs de lignes ou de colonnes des matrices utilisées. Ce nouvel algorithme fait partie de la famille des méthodes *par bloc*. Les méthodes que nous avons décrites jusqu'alors sont appelées méthodes *par point* en opposition à celles que nous allons présenter désormais.

2.4.2 Processus d'Arnoldi par bloc

Les algorithmes par bloc utilisant les sous-espaces de Krylov consistent à contruire les espaces $\mathcal{K}_m$ plus rapidement en construisant à chaque itération, un bloc de s vecteurs indépendants de l'espace $\mathcal{K}_m$. Pour ce faire, on construit l'espace de Krylov par bloc $\mathcal{K}_m(A,V)$, où V est désormais une matrice orthonormale de dimension $n \times s$. On doit toutefois restreindre le nombre d'itérations m de l'algorithme, car on travaille par bloc de largeur s :

$$ms \leq n \tag{2.11}$$

En réalité, l'inégalité (2.11) est toujours vérifiée car on ne mène quasiment jamais les itérations jusqu'au bout, pour des raisons d'occupation mémoire ou de perte d'orthogonalité. On utilise donc des versions redémarrées des algorithmes itératifs, comme dans l'étape 5) de l'algorithme 2.2. Au bout d'un petit ($ms \ll n$) nombre d'itérations, la méthode est redémarrée en prenant comme vecteur initial un vecteur calculé grâce aux approximations précédentes. Dans la suite, les matrices « classiques » seront notées comme précédemment (ex : M) et leurs homologues « par bloc » seront notées M, ainsi les blocs d'une matrice par bloc V pourront être notés V_i. La seule exception à cette règle est la suivante $\mathsf{A} = A$, mais lorsque l'on utilise la notation A on sous-entend que l'on effectue des opérations par bloc avec la matrice A (par exemple pour un produit matrice vecteur).

L'algorithme 2.3 décrivant le processus d'Arnoldi par bloc se déduit simplement de l'algorithme 2.1 décrivant le processus d'Arnoldi par point. Dans l'algorithme 2.3 l'étape *2.c* notée

$$[V_{j+1}, H_{j+1,j}] = \mathtt{QR}(W)$$

signifie que V_{j+1} et $H_{j+1,j}$ sont respectivement les facteurs `Q` et `R` de la factorisation `QR` [82, 187] de W, V_{j+1} est donc une matrice orthonormale et $H_{j+1,j}$ une matrice triangulaire supérieure, telles que $W = V_{j+1}H_{j+1,j}$.

ALGORITHME 2.3 – **Processus d'Arnoldi-MGS par bloc**

1) Initialisation :

Choisir une matrice orthonormale initiale V_1 de taille $n \times s$.

Choisir m la taille (par bloc) du sous-espace.

2) Itération :

Pour $j = 1,2,\ldots,m$

(a) $W = \mathsf{A}V_j$

(b) *Pour $i = 1,\ldots,j$*

$H_{ij} = V_i^H W$

$W = W - V_i H_{ij}$

Fin Pour i

(c) $[V_{j+1}, H_{j+1,j}] = \mathtt{QR}(W)$

Fin Pour j

Lorsque l'algorithme 2.3 se termine, on obtient une matrice de Hessenberg par bloc H_m qui vérifie la relation d'Arnoldi par bloc (2.12), correspondant à la relation par point (2.9).

$$\mathsf{A}\mathsf{V}_m = \mathsf{V}_m\mathsf{H}_m + V_{m+1}H_{m,m+1}E_m^H \tag{2.12}$$

$$\mathsf{V}_m^H\mathsf{A}\mathsf{V}_m = \mathsf{H}_m \tag{2.13}$$

Dans la relation précédente (2.12) chaque terme a une structure bloc. Cette structure par bloc est illustrée à la figure 2.1. Il faut, par exemple, noter que H_m a désormais une structure d'Hessenberg par bloc, comme indiqué à l'équation (2.15). Les blocs $H_{j+1,j}$, de la diagonale (par bloc) inférieure de H_m, sont des matrices triangulaires supérieures car elles sont issues de la factorisation $\mathtt{QR}(W)$ indiquée à l'étape *2.c* de l'algorithme 2.3.

$$\mathsf{V}_m = [V_1\ V_2\ \ldots\ V_m], \text{ avec}, V_i \in \mathbb{C}^{n\times s} \tag{2.14}$$

$$\mathsf{H}_m = [H_{ij}]_{1\le i,j\le m}, H_{ij} = 0 \text{ si } i > j+1 \text{ avec}, H_{ij} \in \mathbb{C}^{s\times s} \tag{2.15}$$

$$E_m = \text{matrice des } s \text{ dernières colonnes de } I_{ms} \tag{2.16}$$

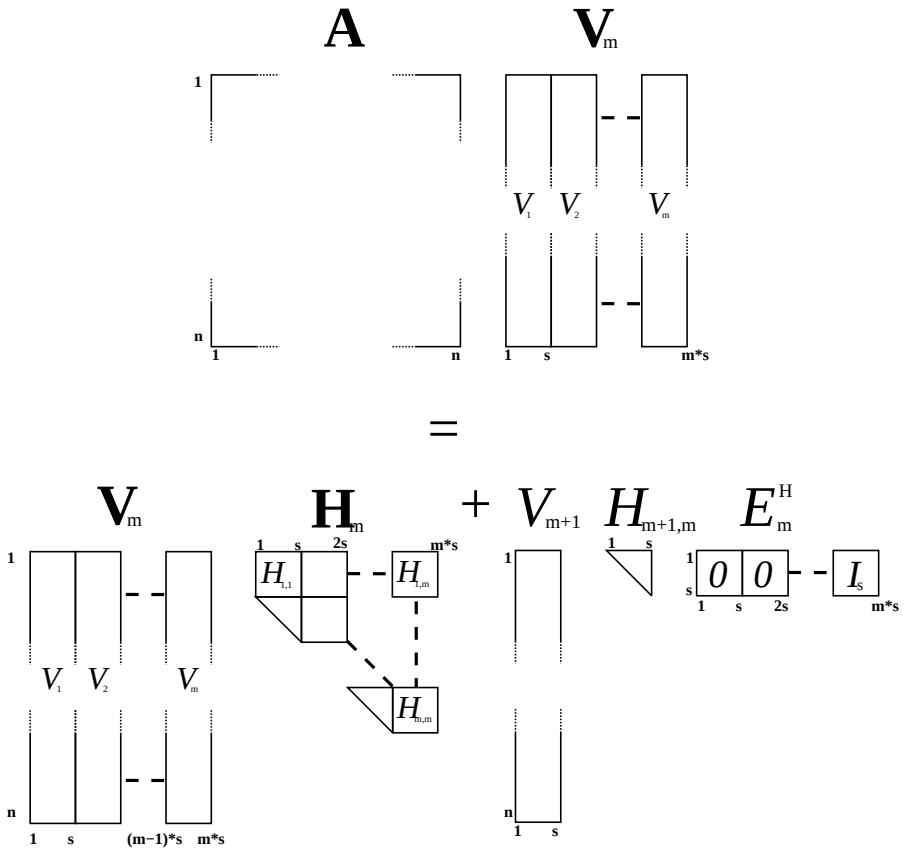

FIG. 2.1 – *Structure de la réduction d'Arnoldi par bloc*

Une fois la matrice par bloc H_m obtenue, nous souhaitons, comme précédemment, résoudre un système linéaire ou un problème de valeurs propres. Il faudra donc :

- soit retransformer la matrice de Hessenberg par bloc H_m en une matrice de Hessenberg afin de se ramener au cas précédent
- soit utiliser des algorithmes adaptés à la résolution de systèmes linéaires ou de valeurs propres pour des matrices de Hessenberg par bloc

La méthode itérative d'Arnoldi par bloc est décrite par l'algorithme 2.4.

ALGORITHME 2.4 – **Méthode Itérative d'Arnoldi par bloc**

1) Initialisation :

(a) Choisir la taille initiale des blocs s

(b) Choisir le nombre r de paires de Ritz souhaitées

(c) Choisir une matrice orthonormale initiale V_1 de taille $n \times s$.

(d) Choisir la taille m du sous-espace de Krylov par bloc $\mathcal{K}_m(A,V_1)$

2) Projection :

(a) Réduire la matrice A sous sa forme de Hessenberg par bloc

$\mathsf{H}_m = \mathsf{V}_m^H \mathsf{A}\mathsf{V}_m$, *par l'algorithme 2.3 (Processus d'Arnoldi par bloc)*

(b) *Déterminer la matrice de Hessenberg* H_{ms} *associée à* H_m,

$H_{ms} = Q_{ms}^H \mathsf{H}_m Q_{ms}$

3) Calcul des éléments propres :

(a) *Calculer les* $m \cdot s$ *valeurs propres* λ_i^{ms} *de* H_{ms}

ainsi que les $m \cdot s$ *vecteurs propres* z_i^{ms} *associés,* $1 \leq i \leq m \cdot s$

$H_{ms} Z_{ms} = Z_{ms} \Lambda_{ms}$, *avec* $\Lambda_{ms} = diag(\lambda_i^{ms})$,$1 \leq i \leq m \cdot s$

(b) *Calculer les* r *vecteurs de Ritz de* A, $\tilde{u}_i$,$1 \leq i \leq r$, *à partir de ceux de* H_{ms},

Pour $r = ms$ *on a :* $\widetilde{U}_{ms} = \mathsf{V}_m Q_{ms} Z_{ms}$

Les valeurs de Ritz $\tilde{\lambda}_i$ *de* A *sont les valeurs propres* λ_i^{ms} *de* H_{ms}

4) Calcul des critères d'arrêt :

(a) $\varepsilon_i = \frac{\|(A - \tilde{\lambda}_i I)\tilde{u}_i\|}{\|A\|}$, $1 \leq i \leq r$

$\varepsilon = f(\varepsilon_1, \ldots, \varepsilon_r)$

(b) *Si* $(\varepsilon \leq tol_\varepsilon)$ *Alors*

STOP

Sinon 5)

5) Redémarrage

(a) *Choisir une méthode de redémarrage*

(on construit implicitement ou explicitement une nouvelle matrice V_1*)*

(b) *Redémarrer l'algorithme en 1.d)*

2.4.3 Avantages des méthodes par bloc

Les méthodes itératives par bloc [205, 138, 73, 8] [180, 181, 179] sont, à l'heure actuelle, beaucoup moins utilisées que leurs homologues par point (voir références incluses dans [168, 104] et [82])[29]. Les méthodes par bloc ont malgré tout certains avantages sur les méthodes par point.

Détection des valeurs propres multiples

La convergence des méthodes par bloc peut être meilleure que celle des méthodes par point car la structure par bloc permet de trouver des valeurs propres de multiplicité inférieure ou égale à la taille des blocs [205, page 59] sans technique de déflation. Afin d'illustrer ce fait nous montrons ci-après pourquoi la méthode itérative d'Arnoldi par point ne peut pas détecter des valeurs propres multiples sans déflation.

29. Les algorithmes par bloc pour les méthodes directes d'algèbre linéaire sont en revanche très utilisés [6] et bien documentés [57, 82]

Si l'on prend la relation d'Arnoldi (2.9) pour $m = k$, et que l'on suppose qu'aucun problème de déficience de rang n'est survenu (voir §2.4.6), nous avons construit une matrice de Hessenberg supérieure irréductible non-singulière H_k de taille $k \times k$. C'est-à-dire que H_k est une matrice non-singulière dont la première sous-diagonale est constituée d'éléments non-nuls (strictement positifs car ce sont des normes de vecteurs). Donc quel que soit λ la matrice $H_k - \lambda I_k$ est une matrice de Hessenberg supérieure de rang $\geq k-1$ car les $k-1$ premières colonnes[30] de $H_k - \lambda I_k$ sont linéairement indépendantes. De plus, cette matrice est singulière si et seulement si λ est une valeur propre de H_k. Ce qui signifie, comme il est spécifié dans [82, Théoreme 7.4.4], que si λ est une valeur propre de H_k sa multiplicité géométrique est 1, ou de façon équivalente que le sous-espace propre associé à λ est de dimension 1. Ceci implique que la méthode itérative d'Arnoldi par point ne peut pas trouver de sous-espace propre de dimension supérieure à 1 (voir remarque 2.1).

Le même raisonnement à partir d'une matrice de Hessenberg par bloc H_k issue d'un processus d'Arnoldi par bloc nous amène à la conclusion que la matrice par bloc de taille s, H_k peut avoir des valeurs propres de multiplicité géométrique jusqu'à s.

REMARQUE **2.1 (Matrice de Hessenberg deffective)** *Il n'est pas exclu que la multiplicité algébrique de λ soit supérieure à 1 et H_k est alors une matrice défective. En effet, si l'on prend l'exemple de la matrice H suivante :*

$$H = \begin{pmatrix} 8 & 0 & 0 & 0 \\ 1 & 8 & 0 & 0 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

H est une matrice de Hessenberg supérieure irréductible ayant pour valeurs propres :

$$\lambda_1 = \lambda_2 = 8 < \lambda_3 = 1 + \sqrt{2} < \lambda_4 = 1 - \sqrt{2}$$

$\lambda_1 = \lambda_2 = 8$ est une valeur propre double de multiplicité algébrique égale à 2 et de multiplicité géométrique égale à 1. Le sous-espace propre associé à cette valeur propre est donc de dimension 1 et il est engendré par le vecteur u_1 les autres vecteurs propres u_3 et u_4 associés respectivement à λ_3 et λ_3 sont donnés à titre indicatif.

$$u_1 = \begin{pmatrix} 0 \\ 47 \\ 7 \\ 1 \end{pmatrix}, u_3 = \begin{pmatrix} 0 \\ 0 \\ 2 \\ \sqrt{2} \end{pmatrix}, u_4 = \begin{pmatrix} 0 \\ 0 \\ -2 \\ \sqrt{2} \end{pmatrix}$$

H est donc une matrice de Hessenberg supérieure irréductible et défective. *En réalité, il est très peu probable que la matrice de Hessenberg issue d'un processus d'Arnoldi soit défective. On remarquera que c'est tout de même possible si l'on applique l'algorithme 2.1 avec $A = H$ et $v_1 = e_1$ qui produit la matrice H elle-même.*

REMARQUE **2.2 (Matrice défective et cas symétrique)** *On notera que dans le cas symétrique la matrice H_k du processus d'Arnoldi devient la matrice T_k du processus de*

30. par exemple, ce n'est pas le seul choix possible

Lanczos par point [131] et que si cette matrice est irréductible elle ne peut avoir que des valeurs propres simples[31] *et ne peut en aucun cas être défective. L'algorithme de Lanczos par bloc permet également de détecter les valeurs propres multiples [151, page 316].*

Résolution des systèmes linéaires à plusieurs second membres

Les méthodes par bloc sont mieux adaptées à la résolution de systèmes linéaires à plusieurs second membres. Les problèmes de systèmes linéaires à plusieurs second membres se posent généralement de 2 façon distinctes :

– second membres multiples et simultanés,

$$AX = B \tag{2.17}$$

– second membres multiples en séquence,

$$Ax_i = b_i, \text{ pour } 1 \leq i \leq s \tag{2.18}$$

Dans les équations (2.17) et (2.18) on a :

$$A \in \mathbb{R}^{n \times n} (\text{ou } \mathbb{C}^{n \times n})$$

$$X \in \mathbb{R}^{n \times s} (\text{ou } \mathbb{C}^{n \times s})$$

$$B \in \mathbb{R}^{n \times s} (\text{ou } \mathbb{C}^{n \times s})$$

et

$$B = [b_1 \cdots b_s], \text{ avec } b_i \in \mathbb{R}^n \text{ pour } 1 \leq i \leq s$$

$$X = [x_1 \cdots x_s], \text{ avec } x_i \in \mathbb{R}^n \text{ pour } 1 \leq i \leq s$$

Les techniques de résolution « par bloc » pour les 2 types de problèmes (équations (2.17) et (2.18)) sont différentes [181] mais la résolution d'un système linéaire à s second membres (simultanés ou en séquence) par une méthode par bloc peut être plus rapide [73, 138, 181] que s résolutions *indépendantes* par une méthode par point.

Efficacité

Les méthodes par bloc sont mieux adaptées au calcul parallèle et d'une façon générale aux machines dont l'architecture mémoire est hiérarchisée [57, page 11] car elles sont plus riches en opérations BLAS de niveau 3 dont le ratio entre les références mémoire et les opérations en virgules flottante est plus avantageux [57, page 77].

Nous décrivons ci-après un peu plus en détail les étapes de la méthode itérative d'Arnoldi par bloc constituant l'algorithme 2.4. C'est-à-dire l'estimation de la norme des résidus pour le calcul des critères d'arrêt (§2.4.4), le redémarrage (§2.4.5).

31. multiplicité géométrique=multiplicité algébrique=1

2.4.4 Estimation des résidus

À l'étape 4.a de l'algorithme 2.4, il est nécessaire de calculer la norme des résidus correspondant aux paires de Ritz désirées. Les vecteurs de Ritz $\tilde{u}_i$ de A étant définis par l'équation (2.19),

$$\tilde{U}_{ms} = [\tilde{u}_1 \cdots \tilde{u}_{ms}] = \mathsf{V}_m Q_{ms} Z_{ms} \tag{2.19}$$

si l'on définit les vecteurs propres Y_{ms} de H_m par l'équation (2.20),

$$Y_{ms} = Q_{ms} Z_{ms} \tag{2.20}$$

on a $\tilde{U}_{ms} = \mathsf{V}_m Y_{ms}$ et :

$$\begin{aligned}
R_{ms} &= A\tilde{U}_{ms} - \tilde{U}_{ms}\tilde{\Lambda}_{ms} \\
\textit{[par définition]} &= A\mathsf{V}_m Y_{ms} - \mathsf{V}_m Y_{ms}\tilde{\Lambda}_{ms} \\
\textit{[d'après l'équation (2.12)]} &= (\mathsf{V}_m\mathsf{H}_m + V_{m+1}H_{m+1,m}E_m^H)Y_{ms} - \mathsf{V}_m Y_{ms}\tilde{\Lambda}_{ms} \\
&= \mathsf{V}_m(\underbrace{\mathsf{H}_m Y_{ms} - Y_{ms}\tilde{\Lambda}_{ms}}_{=0}) + V_{m+1}H_{m+1,m}E_m^H Y_{ms} \\
&= V_{m+1}H_{m+1,m}E_m^H \underbrace{Y_{ms}}_{\mathsf{V}_m^H\tilde{U}_{ms}}
\end{aligned}$$

On obtient finalement l'équation (2.21) qui donne un moyen simple d'évaluer la norme des résidus, on ne calcule que les colonnes de $V_{m+1}H_{m+1,m}E_m^H Y_{ms}$ qui correspondent aux vecteurs de Ritz choisis :

$$A\tilde{U}_{ms} - \tilde{U}_{ms}\tilde{\Lambda}_{ms} = V_{m+1}H_{m+1,m}E_m^H Y_{ms} \tag{2.21}$$

En remarquant que $\tilde{U}_{ms} = \mathsf{V}_m Y_{ms}$ et $E_m^H\mathsf{V}_m = V_m^H$ on peut ré-écrire l'équation (2.21) en fonction de $\tilde{U}_{ms}$ et passer le résidu dans la partie gauche de l'équation, ce qui donne l'équation (2.22).

$$(A - V_{m+1}H_{m+1,m}V_m^H)\tilde{U}_{ms} = \tilde{U}_{ms}\tilde{\Lambda}_{ms} \tag{2.22}$$

Cette équation montre que $\{\tilde{U}_{ms},\tilde{\Lambda}_{ms}\}$ sont les paires propres exactes d'une matrice perturbée $\tilde{A} = A + \Delta A = A - V_{m+1}H_{m+1,m}V_m^H$. C'est la norme de $H_{m+1,m}$ qui guide la norme de la perturbation.

Une fois la norme des résidus calculée, on peut évaluer les critères d'arrêt et s'ils ne sont pas satisfaits redémarrer la méthode, étape 5) des algorithmes 2.2 et 2.4.

2.4.5 Redémarrage

Le coût de stockage, dominant [32] en mémoire, de la méthode d'Arnoldi par bloc croît proportionnellement à $ns(m+1)$, cf. tableau 2.1. La réduction d'Arnoldi (2.12) ne peut donc pas être calculée pour ms trop grand, et doît être redémarrée. Le redémarrage

32. on suppose toujours que $n \gg ms$

	Processus d'Arnoldi-MGS par bloc
V_m	$ns(m+1)$
W	ns
H_m	$ms^2(m+1)$

TAB. 2.1 – *coûts de stockage du processus d'Arnoldi par bloc*

consiste en fait à calculer une nouvelle réduction d'Arnoldi à partir d'une nouvelle matrice V_1 [33].

La nouvelle matrice de départ est calculée à partir des informations accumulées dans la réduction d'Arnoldi précédente de telle façon que l'algorithme itératif 2.4 puisse converger. Aucune preuve de convergence des algorithmes itératifs redemarrés de type Arnoldi (Méthode itérative d'Arnoldi [par bloc], GMRES, ...) n'est connu actuellement.

Le redémarrage peut être calculé principalement de deux façon différentes :

- **Redémarrage explicite** [165, 170, 59]
  On calcule *explicitement* une nouvelle matrice de départ V_1 et on re-construit une réduction d'Arnoldi (2.12) depuis le départ en appliquant l'algorithme 2.3.
- **Redémarrage implicite** [184, 115, 106]
  Une nouvelle matrice de départ est calculée *implicitement* en « compactant » la réduction d'Arnoldi précédente produisant ainsi, en même temps, le début de la prochaine réduction d'Arnoldi. Pour mieux comprendre le mécanisme du redémarrage implicite il faut remarquer qu'une réduction d'Arnoldi AR(A,V_1,m), par bloc d'ordre k (comme celle produite par l'algorithme 2.3 pour m=k) est caractérisée par les équations suivantes :

$$\mathsf{A}\mathsf{V}_k = \mathsf{V}_k\mathsf{H}_k + WE_k^H \tag{2.23}$$
$$\mathsf{V}_k^H W = 0 \tag{2.24}$$
$$\mathsf{V}_k = [V_1\ V_2\ \ldots\ V_k], \text{ avec,} V_i \in \mathbb{C}^{n\times s} \tag{2.25}$$
$$\mathsf{H}_k = [H_{ij}]_{1\le i,j\le k}, H_{ij} = 0 \text{ si } i > j+1 \text{ avec,} H_{ij} \in \mathbb{C}^{s\times s} \tag{2.26}$$
$$E_k = \text{matrice des } s \text{ dernières colonnes de } I_{ks} \tag{2.27}$$

  Le redémarrage implicite produit une réduction d'Arnoldi d'ordre $k = (m-p)$, avec $p > 0$, à partir d'une réduction d'Arnoldi d'ordre m *sans reconstruire* la réduction d'Arnoldi :

$$\text{AR}(A,V_1^{new},m-p) = \textbf{ImplicitRestart}(p,\text{AR}(A,V_1,m))$$

  La nouvelle réduction d'Arnoldi, AR(A,V_1^{new},k) *n'est pas construite* en partant du nouveau vecteur V_1^{new} et en appliquant le processus d'Arnoldi. En revanche, une fois obtenue cette nouvelle réduction d'Arnoldi d'ordre k, on peut l'étendre à l'ordre $m = k+p$ en *continuant* le processus d'Arnoldi directement à l'étape 2.c) de l'algorithme 2.3, en ayant fixé $j = k$.

33. ou un nouveau vecteur v_1 dans le cas d'Arnoldi par point

2.4.6 Taille de bloc variable

La méthode itérative d'Arnoldi *par bloc* n'est pas très utilisée [106, 170, 165] et de la même façon que son homologue symétrique (Lanczos par bloc [131]) [73, 14] un problème de déficience du rang peut survenir dans le processus de Gram-Schmidt (étape 2.c de l'algorithme 2.3). Comme nous souhaitons exploiter le parallélisme potentiel des opérations par bloc nous voulons éviter d'utiliser l'implantation de Ruhe [164] des méthodes par bloc comme cela est fait par exemple dans [73]

À l'étape **2.c** de l'algorithme 2.3, la factorisation QR peut ne pas être de rang plein car W peut également ne pas être de rang plein. Ceci implique qu'à cette étape $H_{j+1,j}$ n'est pas une matrice de rang plein et qu'il existe une indétermination concernant la matrice V_{j+1}. Dans le cas d'Arnoldi par point (algorithme 2.1), c'est un cas *d'échec heureux*[34] car cela signifie qu'un sous-espace invariant a été trouvé. En effet, si on prend la relation d'Arnoldi par point (2.9) pour $m = j$, on voit que si $h_{j+1,j} = 0$ les valeurs propres de H_j sont des valeurs propres exactes de A.

Dans le cas d'Arnoldi par bloc, l'équation (2.12) nous montre que nous ne serons pas en mesure de continuer le processus si l'on ne résout pas le problème du rang. Le problème vient du fait que comme $H_{j+1,j}$ est singulière, supposons de rang $s - p$, il y a p zeros sur la diagonale de $H_{j+1,j}$. On peut donc obtenir une matrice V_{j+1} singulière ou qui n'est tout simplement pas orthogonale aux V_k précédentes ($1 \leq k \leq j$). Plus précisément, on peut trouver une matrice V_{j+1} telle que $V_{j+1}H_{j+1,j} = W$ et $V_{j+1}^T V_{j+1} = I_s$ mais telle que $V_{j+1}^T \mathsf{V}_k \neq 0$. Il faut donc noter que les seuls choix valides du couple (V_{j+1},$H_{j+1,j}$) doivent vérifier les équations (2.28) et (2.29).

$$[V_{j+1},H_{j+1,j}] = \mathtt{QR}(W), \text{ avec } V_{j+1}^T V_{j+1} = I_s, \text{ et } W = V_{j+1}H_{j+1,j} \tag{2.28}$$

$$V_{j+1}^T \mathsf{V}_j = 0 \tag{2.29}$$

Une solution proposée par *Lehoucq* et *Maschhoff* dans [106] consiste à remplir les colonnes indéterminées de V_{j+1} avec des vecteurs générés aléatoirement et orthogonalisés par rapport aux précédentes colonnes de V_{j+1} et V_j. Les éléments diagonaux de $H_{j+1,j}$ correspondants sont mis à zéro. Leur solution vérifie les équations (2.28) et (2.29) et est facilitée par le fait qu'ils calculent la factorisation QR de W avec un processus de Gram-Schmidt qui produit les colonnes une à une. Nous souhaitons pouvoir faire la factorisation QR avec des réflecteurs orthogonaux de Householder[35] nous proposons donc une autre solution qui satisfait également les équations (2.28) et (2.29). Il est possible de réduire la taille des blocs lorsqu'un problème de rang se pose. Supposons que nous ayons effectué une factorisation QR de W avec des réflecteurs de Householder et un pivotage des colonnes[36] :

$$W\Pi = \mathtt{QR}_{piv}(W) = QR \tag{2.30}$$

Dans l'équation précédente, Π est la matrice de permutation issue de la stratégie de pivotage des colonnes [187, Chap. 5 – §2.1]. Si $W \in \mathbb{R}^{n \times s}$ avec $n \gg s$ et W de rang $s - p$

34. « lucky breakdown », cf. [165]
35. Comme cela est fait dans LAPACK
36. voir procédures **xGEQPF** de LAPACK v2.0, ou xGEQP3 de LAPACK v3.0

pour $0 \leq p \leq s$, alors 3 cas de figures sont à envisager :

1. $p = 0$, rien à faire la factorisation QR est de rang plein.
2. $p = s$, c'est un échec heureux par bloc, car comme dans le cas par point nous avons trouvé un sous-espace invariant.
3. $1 < p < s$, on doit résoudre le problème de déficience du rang.

Dans le dernier cas, on obtient un facteur $R = H_{j+1,j}\Pi$ qui n'est pas de rang plein. Une factorisation QR qui n'est pas de rang plein est illustrée à la figure 2.2, sur laquelle les zones hachurées des matrices $H_{j+1,j}$ et V_{j+1} seront « coupées » car identiquement nulles (zone hachurée de $H_{j+1,j}\Pi$) ou ne satisfaisant pas [forcément] l'équation (2.29) (zone hachurée de V_{j+1}). Cette disposition est le résultat de la factorisation QR avec pivotage des colonnes

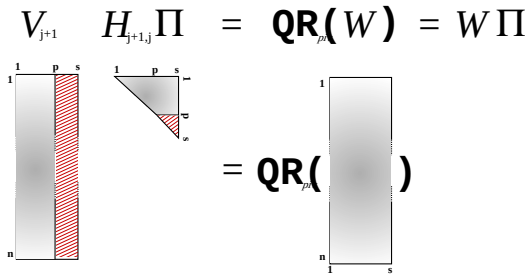

FIG. 2.2 – *Factorisation QR de rang p sur s*

qui contruit un facteur R dont les éléments diagonaux sont rangés par ordre décroissant (en valeur absolue). On pourrait obtenir la même propriété avec un processus de Gram-Schmidt avec pivotage de colonnes, qui mettraient les colonnes linéairement dépendantes à la fin à chaque fois qu'un élément diagonal de R est nul. Une fois cette factorisation obtenue il suffit de supprimer les p dernières lignes de $H_{j+1,j}\Pi$ ainsi que les p dernières colonnes correspondantes de V_j. On obtient ainsi les nouveaux $\tilde{H}_{j+1,j}$ et $\tilde{V}_j$ qui vérifient les équations (2.28) et (2.29). On peut noter que $H_{j+1,j} = R\Pi^{-1}$ et que par conséquents $\tilde{H}_{j+1,j}$ n'est plus une matrice triangulaire supérieure, mais ce n'est pas génant car la matrice H sera retransformée en une matrice de Hessenberg afin de chercher ses valeurs propres. Une fois cette réduction de la taille des blocs effectuée on peut continuer le processus d'Arnoldi par bloc en utilisant $\tilde{V}_j$ à l'étape **2.a** de l'algorithme 2.3.

2.5 Caractéristiques des méthodes itératives d'algèbre linéaire

Nous venons d'examiner, les caractéristiques d'un représentant particulier, la méthode d'Arnoldi pour le problème de valeurs propres, des méthodes itératives utilisant les sous-espaces de Krylov. Ce n'est pas l'objectif de cette thèse de faire une présentation exhaustive des méthodes de Krylov et nous renvoyons le lecteur intéressé aux références citées précédemment. Toutefois notre objectif est de fournir les briques de bases, qui soient les plus ré-utilisables possible, lors de l'implantation parallèle ou séquentielle des méthodes

de Krylov. Nous postulons que la méthode d'Arnoldi [par bloc] est un bon représentant de la famille des méthodes de Krylov car elle contient toutes les opérations élementaires utilisées par les méthodes de Krylov. Spécifier et concevoir LAKe à partir des besoins exprimés par l'implantation parallèle et séquentielle de cette méthode permettra d'implanter facilement d'autres méthodes de Krylov, comme la méthode Multi-ERAM [175].

2.5.1 Les opérations élémentaires

Nous présentons ci-après les opérations dîtes élémentaires nécessaires à l'implantation de méthodes de Krylov par bloc. Dans les formules ci-après $\mathbb{K} = \mathbb{R}$ ou $\mathbb{C}$ et $n,s,m,p \in \mathbb{N}$. De plus, si l'on choisit $s = 1$ on obtient les opérations nécessaires à l'implantation des méthodes de Krylov par point.

1. Opérations algébriques de base
   (a) SAXPY[37], mise à jour de matrices pleines : $Y = \alpha X + \beta Y$ avec $Y,X \in \mathbb{K}^{n \times s}$ and $\alpha,\beta \in \mathbb{K}$.
   (b) GAXPY[38], produits de matrices pleines : $Y = \alpha A \cdot X + \beta Y$ avec $Y \in \mathbb{K}^{n \times s}$, $A \in \mathbb{K}^{n \times s}$, $X \in \mathbb{K}^{s \times s}$ and $\alpha,\beta \in \mathbb{K}$.
   (c) GTAXPY[39], produits par la transposée de matrices pleines : $Y = \alpha A^H \cdot X + \beta Y$ avec $Y \in \mathbb{K}^{s \times s}$, $A \in \mathbb{K}^{n \times s}$, $X \in \mathbb{K}^{n \times s}$ and $\alpha,\beta \in \mathbb{K}$.
   (d) Produits matrice creuse/matrice pleine : $Y = \alpha A \cdot X$ avec $X,Y \in \mathbb{K}^{n \times s}$ et $A \in \mathbb{K}^{n \times n}$, A est **creuse**. A peut n'être disponible que sous la forme d'un opérateur effectuant des produits matrice/vecteur ou matrice/matrice.
   (e) Scaling diagonal : $Y = D \cdot Y$ ou $Y = Y \cdot D$ avec $Y \in \mathbb{K}^{n \times s}$ et $D = \mathrm{diag}(d_i)_{1 \leq i \leq s}$.
2. Indexation par bloc et par point
   (a) Indexation par bloc :
      - $Y = A(i_1 : i_2, j_1 : j_2)$ [*lecture*]
      - $A(i_1 : i_2, j_1 : j_2) = Y$ [*écriture*]

      avec $A \in \mathbb{K}^{m \times n}$ et $Y \in \mathbb{K}^{(i_2 - i_1 + 1) \times (j_2 - j_1 + 1)}$.
   (b) Indexation par point :
      - $\alpha = A(i,j)$ [*lecture*]
      - $A(i,j) = \alpha$ [*écriture*]

      avec $A \in \mathbb{K}^{m \times n}$ et $\alpha \in \mathbb{K}$.

   Les opérations d'écriture ne sont a priori utilisées que sur les matrices pleines.
3. Allocation mémoire
   (a) Allouer, désallouer une matrice $A \in \mathbb{K}^{m \times n}$.
4. Opérations de conversion
   La plupart des librairies ou langages de programmation ont une façon particulière de stocker des tableaux en mémoire (par exemple le langage C stocke les éléments

37. **S**calar **A** **X** **P**lus **Y**
38. **G**eneral **A** **X** **P**lus **Y**
39. **G**eneral **T**ranspose **A** **X** **P**lus **Y**

ligne par ligne et le Fortran colonne par colonne), ceci signifie que pour appeler une fonctionnalité d'une librairie (par exemple LAPACK) écrite dans un langage (par exemple Fortran 77) à partir d'une autre librairie (LAKe) écrite dans un autre langage (par exemple C++) il faut que la librairie appelante fournisse des fonctions de conversion.

   (a) Conversion vers LibLang : $Y_{\text{LIBLANG}} = \text{LIBLANG}((Y)$.
   (b) Vue complète : $Y_{whole} = \text{whole}(Y)$. Cette opération fournie une représentation contiguë en mémoire de la matrice Y stockée colonnes par colonnes. Cela doit permettre de passer la matrice Y_{whole} à un sous-programme externe à LAKe (typiquement un appel à LAPACK). Cette opérations est la fonction de conversion LapackF77

5. Opérations algébriques de haut niveau
   (a) Factorisation QR d'une matrice pleine : $[Q,R] = \texttt{QR}(W)$, avec $W,Q \in \mathbb{K}^{n \times s}, n \geq s$, $R \in \mathbb{K}^{s \times s}$ où Q est une matrice orthonormale et R une matrice triangulaire supérieure.
   (b) Calcul des éléments propres d'une matrice pleine : $[\Lambda,U] = \texttt{Eigenpair}(A)$ tels que $A \cdot U = U \cdot \Lambda$, avec $A \in \mathbb{K}^{n \times n}$, $U,\Lambda \in \mathbb{C}^{n \times n}$. $\Lambda = \text{diag}(\lambda_i)_{1 \leq i \leq n}$ si A est diagonalisable, Λ est une matrice de Jordan [188, Chap. 1 – §1.6] [208] sinon.

Les opérations algébriques élémentaires sur les matrices pleines correspondent en fait à un sous-ensemble des opérations BLAS[40]. On peut noter que dans le cas des opérations nécessaires aux méthodes par points les opérations GAXPY se réduisent à des SAXPY et les opérations GTAXPY à des produits scalaires.

Les opérations d'indexations par point et par bloc de matrices peuvent être directement fournies par le language de programmation (Fortran 95 par exemple) mais ce n'est pas le cas de tous les languages (C, C++, Java) c'est pourquoi il ne faut pas oublier de les spécifier. De plus, nous verrons que les opérations d'indexation ne sont pas traitées de la même façon si il s'agit d'une indexation en vue d'une *lecture* ou d'une *écriture* du/des élément/s.

Les opérations algébriques de haut niveaux correspondent à des algorithmes dont l'implantation ne relève pas [forcément] des objectifs de LAKe. On peut considérer que ces opérations de haut niveau sont disponibles dans une librairie externe à laquelle LAKe fera appel, LAPACK par exemple [6, 103]. L'appel à ces opérations de haut niveau peut nécessiter un appel à une opération de conversion 4b. D'autres opérations algébriques de haut niveau peuvent être nécessaires pour l'implantation des méthodes Krylov : décomposition SVD (méthode de Lanczos par bloc pour les systèmes non-hermitiens), rotation de Givens ou réflecteurs de Householder (GMRES)... Ces opérations, mêmes si elle ne figurent pas dans cette liste, seront tout de même prises en compte dans la conception de LAKe car elles utiliseront les mêmes mécanismes que celles utilisées pour la méthode itérative d'Arnoldi.

40. **B**asic **L**inear **A**lgebra **S**ubprograms, cf. [57, §5.1 p. 71] et [20]

Méthode itérative d'Arnoldi par bloc *(complexité d'une itération)*		
Opération	*Complexité en Flop*	*Algo – étape*
$W = \mathsf{A}V_j$	$2nnz \times ms$	Algo. 2.3 – 2.a
$H_{ij} = V_i^H W$	$(\frac{m(m+1)}{2}) \times (2n-1)s^2$	Algo. 2.3 – 2.b.i
$W = W - V_i H_{ij}$	$(\frac{m(m+1)}{2}) \times (ns + 2ns^2)$	Algo. 2.3 – 2.b.ii
$[V_{j+1}, H_{j+1,j}] = \mathtt{QR}(W)$	$ns(s+1) - s(s^2-1)/3$ [187, Chap. 4 – §1.2]	Algo. 2.3 – 2.c
Sous-Total Processus d'Arnoldi :	$2nnz \times ms$ + $n\left(ms\frac{m+1}{2}(4s+1) + s(s+1)\right)$ + $\mathcal{O}\left((ms)^2\right)$	
$H_{ms} = Q_{ms}^H \mathsf{H}_{ms} Q_{ms}$	$\mathcal{O}\left((ms)^2\right)$ voir [82]	Algo. 2.4 – 2.b
$H_{ms} Z_{ms} = Z_{ms} \Lambda_{ms}$	$\mathcal{O}\left((ms)^3\right)$ voir [82]	Algo. 2.4 – 3.a
$\tilde{U}_{ms}(1:r) = \mathsf{V}_{ms} Q_{ms} Z_{ms}$	$2nr(ms)^2 + \mathcal{O}\left((ms)^3\right)$	Algo. 2.4 – 3.b
$V_{m+1} H_{m+1,m} E_m^H Y_{ms}$	$2ns^2 + 2s^3$	Algo. 2.4 – 4.a
Total Méthode itérative d'Arnoldi :	$2nnz \times ms$ + $n\left(2(r+1)(ms)^2 + \frac{(4s+1)}{2}ms + 3s^2 + s\right)$ + $\mathcal{O}\left((ms)^3\right)$	

TAB. 2.2 – *Complexité d'une itération de la méthode itérative d'Arnoldi par bloc*

2.5.2 Critères d'arrêt et redémarrages

L'évaluation des critères d'arrêts ainsi que le redémarrage sont spécifiques à chaque méthode itérative, de plus il existe un grand nombre de variantes et de choix pour ces deux opérations. La conception des ces deux caractéristiques des méthodes de Krylov sera donc extrêmement paramétrable, car on ne peut prévoir toutes les manières de redémarrer ou d'évaluer un critère d'arrêt.

2.5.3 Complexité

Avant d'aborder le problème de la parallélisation des méthodes itératives (§2.6) il convient d'examiner la complexité des algorithmes des processus d'Arnoldi par point et par bloc afin de voir quelles sont les opérations parallélisables. On note nnz le nombre d'éléments non-nuls de la matrice creuse $A \in K^{n \times n}$,$nnz \ll n^2$.

La complexité des opérations utilisées dans la méthodes itérative d'Arnoldi par bloc est présentée dans le tableau 2.2. Pour des raisons de coûts de stockages déjà évoqués on suppose que $n \gg ms$, les opérations les plus coûteuses sont donc les opérations de complexité proportionnelle à n ou nnz, c'est-à-dire les opérations 1a, 1b, 1c et 1d du paragraphe 2.5.1.

2.6 Parallélisme et Méthodes Itératives

Nous avons fait l'hypothèse que la matrice A de notre problème de projection est très grande et très creuse ($nnz \ll n^2$), ce qui justifie l'utilisation des méthodes que nous venons de présenter. Leur parallélisation s'impose, soit parce que l'on cherche à minimiser le temps de calcul, soit parce que l'opérateur matriciel A ne tient pas dans la mémoire d'une machine monoprocesseur. Dans le premier cas, on peut avoir recours à une machine parallèle à mémoire partagée. Dans le deuxième, il faut implanter la méthode sur une machine parallèle à mémoire distribuée (voir paragraphe 1.3, chap. 1).

2.6.1 Modèle de programmation de LAKe

Un des objectifs majeur de LAKe est la réutilisation séquentielle/parallèle ce qui signifie que :

1. le modèle de programmation offert par LAKe doit être indépendant du modèle d'exécution utilisé pour implanter LAKe.
2. les composants de haut niveau de LAKe doivent avoir la même implantation en parallèle et en séquentielle.

L'architecture de LAKe qui découle de ces objectifs est présentée à la figure 2.3.

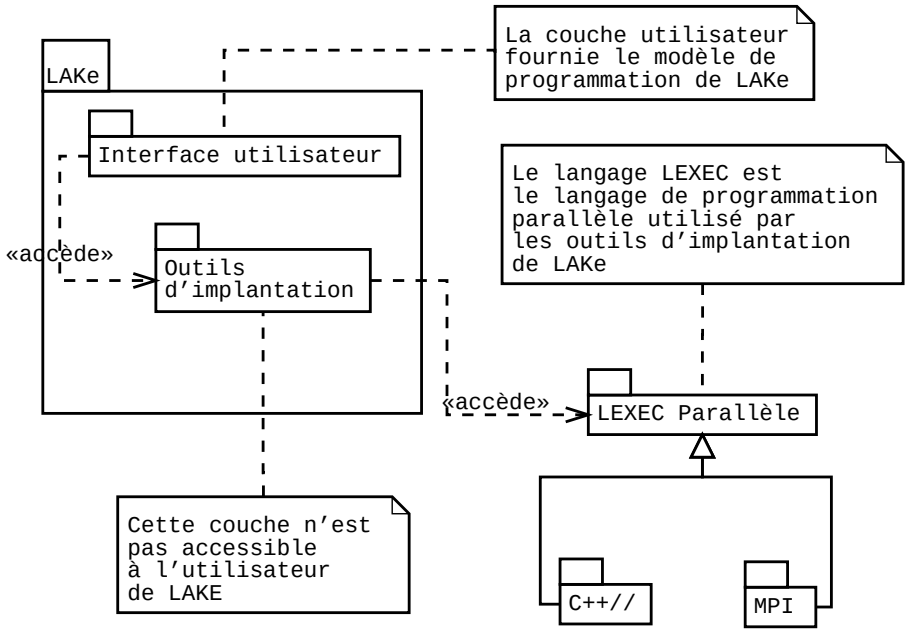

FIG. 2.3 – *Modèle de programmation*

Les objectifs de LAKe pourront être atteints si durant la conception on tient compte des différents modèles d'exécution utilisés par les outils d'implantation de LAKe. Nous considérerons que le modèle d'exécution que nous utilisons est un modèle à mémoire distribuée, car c'est le plus contraignant. Nous spécialiserons les outils d'implantation de LAKe suivant le langage d'exécution parallèle utilisé. Nous verrons dans le chapitre 3, une

spécialisation pour une implantation avec C++// et au chapitre 4 une spécialisation pour une implantation avec MPI. Il est important de garder cette contrainte à l'esprit durant la conception de LAKe afin que les utilisateurs de la librairie n'aient plus à se soucier de cette contrainte puisqu'elle sera gérée par LAKe. C'est un des objectifs de notre travail, masquer le modèle d'exécution utilisé afin de bien séparer le modèle d'exécution utilisé par la librairie du modèle de programmation qu'elle offre.

Dans le paragraphe suivant nous examinons les possibilités de parallélisation des méthodes de Krylov sur un modèle d'exécution à mémoire distribuée.

2.6.2 Distribution des calculs et/ou des données

La stratégie de parallélisation des méthodes de Krylov est relativement simple : paralléliser les opérations les plus coûteuses, c'est-à-dire celle qui sont d'ordre $n(ms)^i$,$i = 2,1,0$. Cette stratégie revient à effectuer des opérations parallèles sur les matrices dont la taille est proportionnelle à n (A, V_m, W, $\widetilde{U}_{ms}$) et à effectuer des traitements séquentiels sur le reste.

Etant donné que l'objectif est de traiter un gros volume de données les modèles de programmation parallèle qui semblent les mieux adaptés sont ceux issus du parallélisme de données (Def. 1.9), notamment la programmation SPMD (Def. 1.11). LAKe offrira donc une interface essentiellement SPMD, ce qui ne signifie pas que le modèle d'exécution utilisé par la couche d'implantation de LAKe sera exclusivement un modèle data-parallel. Notre problème de parallélisation des opérations matricielles sur des matrices de grande taille se découpe donc en 3 étapes :

1. partitionnement des matrices
2. affectation des données partitionnées aux processus
3. collaboration des différents processus pour la réalisation des opérations

Ce découpage pour la gestion parallèle des matrices nous permet notamment de gérer les distributions classiques de matrices comme les distributions cycliques par bloc [40].

Partitionnement des matrices

Concernant le partitionnement des matrices $M \in \mathbb{K}^{m \times n}$ nous nous réduirons au cas des partitionnement cartésiens de $[1..m] \times [1..n]$, des exemples de partitionnements sont présentés à la figure 2.4. Une fois ces partitionnements effectués on *distribue* les partitions sur le nombre de processus choisis. Les processus représentent les processeurs du modèle d'exécution parallèle utilisé par LAKe.

Distribution des partitions

Chaque processus est responsable des données qui lui sont affectées et il est le seul à pouvoir les modifier[41]. Des exemples de distributions de partitions sont présentées à la figure 2.5. On peut remarquer une distribution particulière intitulée #a (*all*) qui correspond à l'affectation simultanée de l'ensemble des bloc d'une partition à tous les processus,

41. c'est la règle des calculs locaux ou « *owner computes rule* »

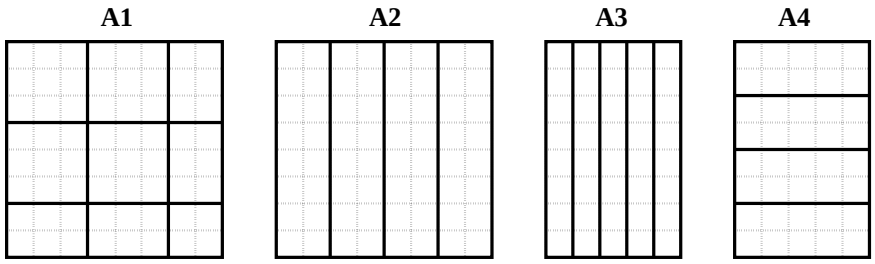

FIG. 2.4 – *Exemples de partitionnement de matrices*

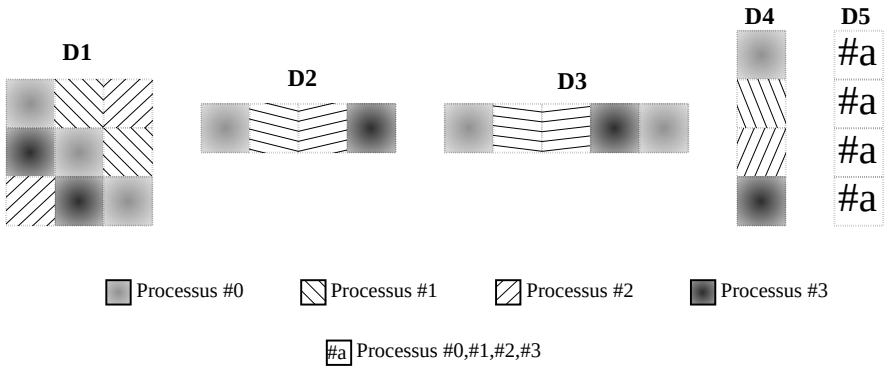

FIG. 2.5 – *Exemples de distribution de partitions sur 4 processus*

autrement dit chaque processus sera responsable d'une copie des éléments affectés indépendamment des autres processus. La cohérence entre les copies dans le temps pourra être différentes suivant le modèle d'exécution utilisé (mémoire distribuée ou partagée). En revanche, les opérations algébriques mettant en jeu des données répliquées, c'est-à-dire distribuées sur #a, sont responsables de la cohérence (ou de la non-cohérence) des données répliquées à la fin de l'opération. Des exemples de matrices distribuées qui résultent des partitions et distributions précédentes sont présentées à la figure 2.6.

La dernière étape est la réalisation des opérations décrites au paragraphe 2.5.1, par les processus qui se sont vus affectés les données impliquées dans l'opération.

2.6.3 Implantation des opérations élémentaires distribuées

Les matrices dont la taille est proportionnelle à n des algorithmes 2.3 et 2.4 seront distribuées et les autres matrices seront répliquées. Ceci implique que lors de l'implantation des opérations élémentaires décrites au paragraphe 2.5.1, il faudra prévoir le fait que certaines matrices utilisées dans ces opérations seront potentiellement distribuées.

La topologie des distributions des matrices devra être prise en compte dans l'implantation des opérations algébriques avec ces matrices. En effet, dès lors qu'une opération implique des éléments d'une matrice (ou plusieurs) matrice(s) qui ne sont pas sous la responsabilité du même processeur, des communications doivent être générées pour com-

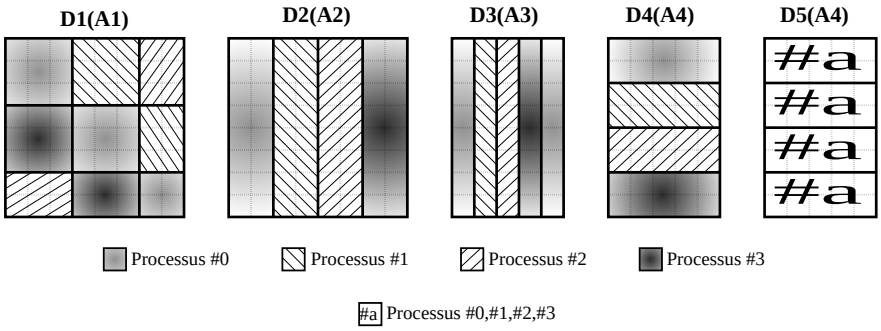

FIG. 2.6 – *Exemples de distribution de matrices sur 4 processus*

muniquer les valeurs au(x) processeur(s) impliqué(s) dans l'exécution de l'opération. Le traitement de ces communications dépendra du modèle d'exécution utilisé par la couche d'implantation de LAKe, nous verrons deux exemples d'implantation différents aux chapitres 3 et 4.

2.7 Conclusion

Nous venons d'étudier un peu plus en détail notre domaine d'application, les méthodes de projections sur les espaces de Krylov. Nous avons examiné plus particulièrement un membre représentatif de la famille des méthodes de Krylov : la méthode itérative d'Arnoldi. Nous avons également proposé une alternative pour la gestion dynamique de la taille des blocs dans la méthode d'Arnoldi par bloc. Nous avons conclu sur la nécessité de paralléliser les opérations sur les matrices de grandes tailles. Nous avons désormais les éléments suffisants pour concevoir une librairie séquentielle et parallèle orientée-objet permettant l'implantation des méthodes de Krylov.

Nous examinerons dans les deux chapitres suivant, deux stratégies de parallélisation exploitant deux moyens de parallélisation différents qui seront autant de modèles d'exécution pour LAKe. Nous verrons les problèmes posés par les modèles de programmation orientés-objet parallèles offerts ainsi que les solutions que nous apportons.

Chapitre 3

Active-LAKe : utilisation d'un modèle de programmation à objets actifs

Sommaire

3.1 Introduction

Nous avons vu au chapitre précédent les spécifications de notre librairie au travers de l'étude de la méthode itérative d'Arnoldi par bloc. Nous étudierons dans ce chapitre l'utilisation du modèle de programmation parallèle orienté-objet à base d'acteur, offert par C++//[27, 54] pour l'implantation de notre librairie : Active-LAKe. Nous avons déjà présenté le modèle à objets actifs hétérogènes de C++// au §1.5.2, nous ne présenterons ci-après plus en détail que les parties qui nous concernent et nous en référons à [54, 171] pour une présentation plus exhaustive du modèle. Nous verrons en particulier le problème de performance lié à la sémantique de non-partage des objets passifs au §3.4, et nous verrons l'extension du modèle que nous avons proposé au §3.5. Ce travail est le fruit d'une

collaboration avec *David Sagnol* et *Denis Caromel* qui a donné lieu à une communication dans un colloque national [34] et une communication dans un colloque international [35]. La contribution de l'auteur de cette thèse dans cette collaboration est :

- la mise en évidence du problème de performance par la réalisation des versions séquentielle et MPI des opérations élémentaires d'algèbre linéaires
- la participation aux spécifications du mécanisme SOR répondant aux besoins et respectant la sémantique de C++//.

3.2 Le modèle d'acteur de C++//

Le modèle de programmation parallèle proposé par C++// [54] est issu des modèles d'acteurs [4, 5]. Dans un modèle à base d'acteur les aspects du parallélisme ont été fusionné aux notions orientées-objet de la façon suivante :

- concurrence + objet = objet actif = acteur
- communication = appel de méthode
- distribution = répartition des objets actifs

La synchronisation, elle, dépend du modèle d'acteur et est réalisée, en C++//, par la notion de *future* (voir programme 1.11 page 46 ou [54]). Dans un modèle d'acteur on aura autant d'activités parallèles qu'il existe d'acteurs dans le système[42]. Paralléliser une application à l'aide d'un modèle à base d'acteurs revient à décider quels seront les acteurs du système, ce qui amène à une méthologie de conception assez simple.

3.2.1 Méthodologie de conception

La méthodologie suggérée par *Caromel* [32], puis par *Caromel, Belloncle et Roudier* [33] consiste en les étapes présentées à la figure 3.1. Cette méthodologie est simple et applicable de façon systématique. L'étape clef est l'étape 2 qui consiste à identifier les objets actifs. Voici une liste non-exhaustive de critères indiquant qu'un objet doit être actif :

- l'objet représente une tâche ou un ensemble autonome de tâches du système
- l'objet ou les données qu'il contient doivent être distribuées
- l'objet doit être partagé par plusieurs objet

Nous verrons au paragraphe 3.4, que le dernier critère pose des problèmes de performance.

3.2.2 Quels mécanismes de réutilisation ?

Notre objectif premier dans la conception de LAKe est d'obtenir la meilleure réutilisabilité possible. C++// présente des atouts indéniables en matière de réutilisation :

1. Langage à objets parallèle à extensions compatibles, mis en œuvre avec un préprocesseur et une librairie parallèle :

42. on suppose que les acteurs ne sont pas multi-actifs

1. Conception et programmation séquentielle
   1.1. Identification des objets
   1.2. Interface et topologie
   1.3. Implantation séquentielle
2. Identification des objets actifs
   2.1. Activités initiales
   2.2. Objets partagés
3. Programmation des objets actifs (processus)
   3.1. Définition de chaque classe « active »
   3.2. Définition de l'activité (méthode `Live`)
   3.3. Redéfinition des méthodes nécessaires
4. Adaptation aux contraintes retour au 2.
   4.1. Raffiner la topologie
   4.2. Définir de nouveaux processus

FIG. 3.1 – *Méthode de programmation parallèle avec C++// (traduction de [33, §3.])*

C++// n'apporte aucune extension de langage à C++, et le pré-processeur génère du code C++ compilable avec un compilateur C++ standard (voir fig. 1.11 page 48).

2. Séparation entre code parallèle et code séquentiel :
   La parallélisation est non-intrusive, c'est-à-dire que les classes parallélisées sont dérivées de leur homologues séquentielles sans que le code des classes séquentielles soit modifié.
3. Utilisation des concepts objets pour la parallélisation :
   La parallélisation utilise l'héritage et des objets parallèles et séquentiels peuvent être polymorphes. Concrètement, des objets séquentiels peuvent utiliser des objets parallèles de façon polymorphe.
4. Communications implicites :
   Les communications entre objets parallèles sont implicites et unifiées avec l'appel de méthode. Le code de gestion des communications est automatiquement généré par le précompilateur.

Les mécanismes de réutilisation offerts par C++// s'appuie sur 2 choses :

- L'héritage multiple :
  C'est par ce biais que l'on transforme un objet passif issu de la conception séquentielle en un objet actif. La séparation entre le code séquentiel et parallèle est donc bonne.
- Un MOP [43] mis en oeuvre par un pré-compilateur :

43. **M**eta-**O**bject **P**rotocol voir §1.5.1–*Approche Réflexive*, et [152]

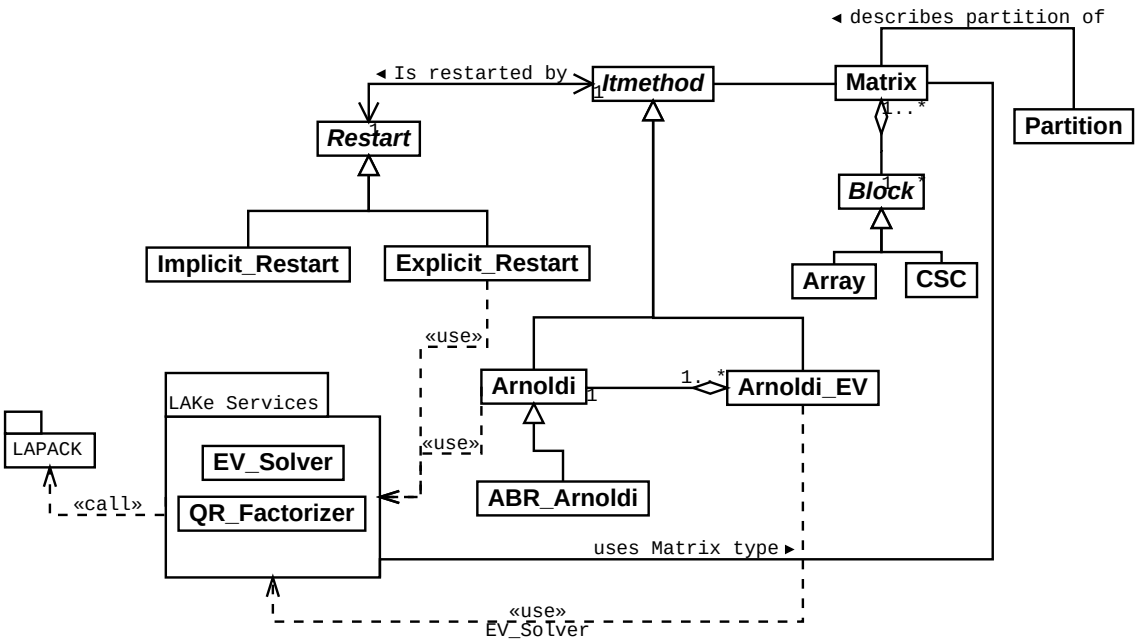

FIG. 3.2 – *Architecture objet de LAKe*

Le MOP de C++// est le moyen de mettre en œuvre les communications implicites et asynchrones entre objets actifs. Les paramètres passés aux méthodes des objets actifs sont automatiquement sérialisés grâce au MO.

3.3 Conception objet de LAKe

Nous avons adopté la démarche de la figure 3.1 pour la conception de notre librairie.

3.3.1 Conception séquentielle : architecture objet de LAKe

L'architecture objet séquentielle de LAKe est issue des spécifications que nous avons présentées au chapitre précédent. Cette architecture est présentée à la figure 3.2, sous la forme d'un diagramme de classes UML. La classe `Itmethod` représente une méthode itérative quelconque dont dériveront toutes les méthodes itératives de LAKe. Les classes `Arnoldi` et `Arnoldi_EV` représente respectivement un processus d'Arnoldi (par bloc) et une méthode itérative d'Arnoldi (par bloc) implantant les algorithmes 2.3 et 2.4 du chapitre 2. Les classes dérivant de `Restart` représentent des stratégies de redémarrage associées à une méthode itérative. Les classes `QR_Factorizer` et `EV_Solver` implantent respectivement les services nécessaires à des factorisations QR et à la recherche des éléments propres d'une matrice pleine. Ces classes utilisent la librairie fortran LAPACK. La classe `Matrix` est la classe qui implante les matrices partitionées par bloc ainsi que les opérations élémentaires SAXPY, GAPXY, GTAXPY décrites au paragraphe §2.5.1. Le diagramme de classe de la figure 3.3 détaille un peu plus la structure d'une matrice par

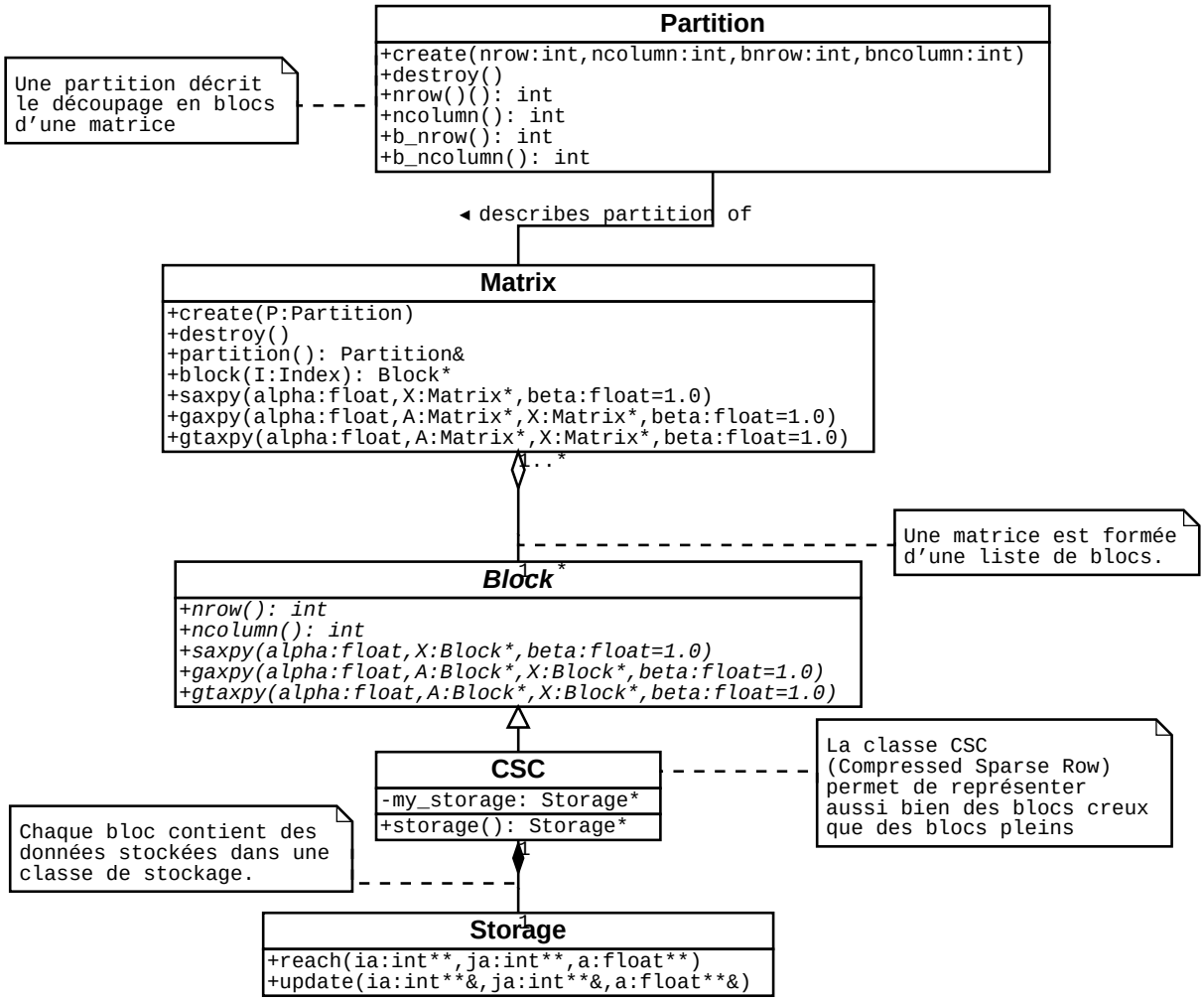

FIG. 3.3 – *Structure de la classe* ***Matrix***

bloc. Une matrice (objet de la classe `Matrix`) est constituée d'une liste de blocs (objets d'une classe dérivant de la classe `Block`). Ces blocs implantent à leur tour les opérations SAXPY, GAPXY et GTAXPY. Chaque objet instance d'une classe dérivant de la classe abstraite `Block` stocke les données (éléments du bloc de la matrice) dans une classe de stockage[44]. Les opérations implantées dans la classe `Matrix` le sont en utilisant celles des classes dérivant de `Block` comme le montre le programme 3.1. Un diagramme d'objets présentant une matrice `A` partitionnée en 4 blocs est présenté à la figure 3.4. Chaque bloc est une instance de `CSC` qui est une classe qui implante un format de stockage creux couramment utilisé [167], le format *Compressed Sparse Column*. On pourrait changer aisément le format de stockage des blocs de la matrice en dérivant une nouvelle sous-classe de `Block`.

44. On notera ici qu'en UML la relation d'aggrégation est notée par un losange évidé et la relation de composition par un losange plein.

Programme 3.1: *GAXPY par bloc*

```
1  void
2  Matrix::gaxpy(float alpha, Matrix* A,
3                Matrix* X, float beta=1.0) {
4  [...]
5  /* Get number of block    *
6   * of column of matrix A */
7  bkmax = (A->partition()).b_ncolumn();
8  [...]
9  /* i,j,k Block Matrix/Matrix multiply */
10 for (I.i()=1;I.i()<=my_partition.b_nrow();(I.i())++)
11  for (I.j()=1;I.j()<=my_partition.b_ncolumn();(I.j())++)
12   for (bk=1;bk<=bkmax;bk++)
13    {
14     block(I)->gaxpy(alpha,
15                     A->block(Index(I.i(),bk)),
16                     X->block(Index(bk,I.j()))
17                     1.0);
18    }
19 [...]
20 }
```

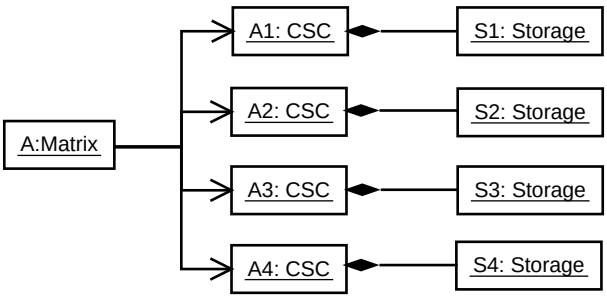

FIG. 3.4 – *Diagramme d'objets UML représentant Matrice* `A` *partitionnée en 4 blocs*

3.3.2 Conception parallèle : architecture objet d'Active-LAKe

Nos objectifs de parallélisation sont les suivants :

1. distribuer les blocs des matrices afin de paralléliser les opérations matricielles élémentaires (SAXPY,GAXPY,GTAXPY).
2. maximiser la réutilisation séquentielle/parallèle ; idéalement *toutes les classes dérivant de la hiérarchie méthodes itératives* (celles qui dérivent d'`Itmethod`) doivent rester inchangées dans la version parallèle de LAKe.
3. l'efficacité de l'implantation parallèle doît être bonne en terme de gain de performance (speed-up)

La parallélisation de LAKe avec C++// suivant ces critères est simple, les blocs des matrices doivent être distribués, ils deviendront donc des objets actifs. Les nouvelles classes nécessaires à la parallélisation sont les suivantes :

1. `CSC_ll`
   Classe active dérivant de `Process` et `CSC`, qui transformera les appels aux méthodes de CSC en appels asynchrones et permettra la distribution de ses instances.
2. `Matrix_ll`
   Classe passive dérivant de `Matrix` qui modifiera uniquement les constructeurs de la classe afin de créer des blocs distribués en fonction de la `Distribution` spécifiée.
3. `Distribution`
   Classe permettant la description de la distribution d'une partition.

Ces changements sont schématisés par le diagramme de classes de la figure 3.5. Cette

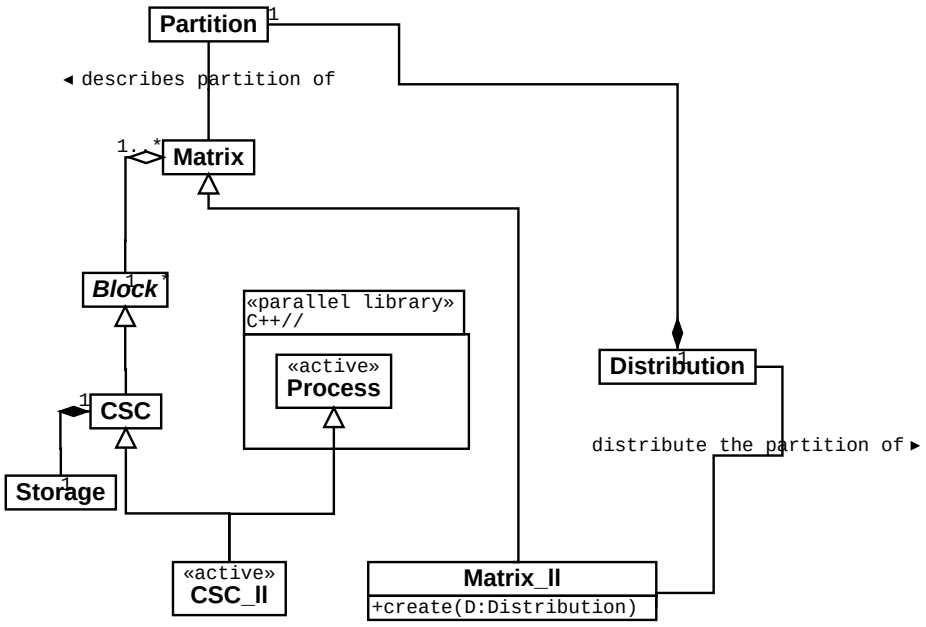

FIG. 3.5 – *Matrices parallèles avec C++//*

parallélisation avec C++// atteint une très bonne réutilisabilité de code car tous les

clients de la classe séquentielle `Matrix` peuvent désormais utiliser de façon polymorphe des objets de la classe `Matrix_ll`. Cela signifie entre autre que les méthodes itératives peuvent utiliser des matrices parallèles de façon polymorphe. Un exemple d'utilisation polymorphe de matrices distribuées est donné par le programme 3.2.

Programme 3.2: *Utilisation polymorphe de matrices distribuées*

```
1  /* An Arnoldi  constructor *
2   * expecting Matrix object */
3  Arnoldi::Arnoldi(Matrix* A, Matrix* X, Matrix* Y)
4  {
5      A_ = A;
6      X_ = X;
7      Y_ = Y;
8  }
9
10 Arnoldi::do_axpy()
11 {
12     /* Here we do call                          *
13      * Matrix::gaxpy(float,Matrix*,Matrix*,float) */
14     Y_->gaxpy(1.0,A_,X_);
15 }
16
17 void main()
18 {
19     Matrix_ll A_ll;
20     Matrix_ll X_ll;
21     Matrix_ll Y_ll;
22     Distribution DA,DX,DY;
23
24     Arnoldi* an_Arnoldi;
25
26     [...]
27     /* Create the distributed matrices */
28     A_ll.create(DA);
29     X_ll.create(DX);
30     Y_ll.create(DY);
31     [...]
32     /* Pass parallel matrices polymorphically *
33      * to the Arnoldi constructor.            */
34     an_Arnoldi = new Arnoldi(A_ll, X_ll, Y_ll);
35     /* This call will perform a parallel operation */
36     an_Arnoldi->do_axpy();
37     [...]
38 }
```

La réutilisation de code est maximale car l'implantation des méthodes itératives est inchangée. De même, quasiment tout le code séquentiel des classes `CSC` et `Matrix` est réutilisé sans modification. Tous nos critères auraient été satisfait si la performance avait été comparable avec l'implantation que nous avions faîte en utilisant MPI, mais ce n'est malheureusement pas le cas.

3.4 Le modèle à objets actifs et les recopies

Le problème de performance vient du modèle de partage des objets en C++//, et dans les modèles d'acteurs d'une façon générale [5] : *aucun objet passif ne peut être partagés par plusieurs objets actifs*. La raison est simple et tout à fait fondée, un objet actif définit un sous-système [171, §4.1], chaque sous-système peut s'exécuter sur des noeuds de calcul différents ou plus exactement dans des espaces d'adressage disjoints. De ce fait, si un objet passif pouvait être partagé par plusieurs sous-systèmes, lorsque ceux-ci s'exécutent sur le même noeud par exemple, la sémantique du programme changerait suivant le mapping des sous-systèmes sur les noeuds de calcul. La conséquence de cette politique de non-partage est que dès qu'un objet passif est passé en paramètre d'un appel de méthode ou qu'il est le résultat d'un retour de fonction entre 2 sous-systèmes, cet objet passif est copié en profondeur[45]. C'est exactement ce qu'il se passe aux lignes 4 et 5 du programme 3.3.

Programme 3.3: *GAXPY CSC*

```
1  CSC::gaxpy(float alpha, CSC* A,
2             CSC* X, float beta=1.0) {
3    /* Get storage reference */
4    Storage* SA = A->storage();
5    Storage* SX = X->storage();
6    Storage* SY = storage();
7
8    SA->reach(&tia, &tja, &ta);
9    SX->reach(&tix, &tjx, &tx);
10   SY->update(&tiy, &tjy, &ty);
11   [...]
12   for (j=1;j<=ncol_;j++)
13     for (i=1;i<=nrow_;i++)
14       for (k=1;k<=ncola;k++)
15         /* Y(i,j) += A(i,k)*X(k,j) */
16         ty[(j-1)*nrow_+i] += alpha * ta[(k-1)*nrowa+i]
17                                    * tx[(j-1)*ncola+k];
18   [...]
19 }
```

45. en anglais : *deep copy*, les objets ou références contenus dans l'objet sont eux-mêmes copiés et ceci de façon récursive.

L'objet actif courant, `Y` instance de la classe `CSC_ll`, exécute le code de la méthode `CSC::gaxpy` héritée de la classe séquentielle `CSC`. Les paramètres effectifs `A` et `X` sont des objets actifs instances de la classe `CSC_ll` qui sont traités de façon polymorphe comme des objets de la classe `CSC`. Lors de l'appel à la méthode `CSC::storage()` – Prog. 3.3 ligne 4 – l'objet actif `Y` récupère en retour *une copie profonde* de l'objet passif de la classe `Storage` appartenant à l'objet actif `A`. Il en est de même pour `X`. La sémantique du calcul est respectée car `Y` ne fait que lire les objets `SA` et `SX`. Le problème de dégradation de performance vient du fait que si les objets actifs `Y`, `A` et `X` sont situés sur le même noeud de calcul et possiblement dans le même espace d'adressage[46] la copie est inutile. Une parallélisation explicite en MPI évite *explicitement* cette copie, puisqu'on ne communique que les objets non locaux entre eux.

Vu ce problème de performance, nous avons spécifié conjointement avec les auteurs de C++//, *David Sagnol* et *Denis Caromel*, un mécanisme de partage en lecture semi-automatique qui évite ces copies lorsque le mapping des objets actifs le permet. Ce mécanisme ne change en rien la sémantique de C++// quel que soit le mapping des objets concernés, il s'agit juste d'une optimisation.

REMARQUE **3.1 (copy on write)** *La technique du partage d'objet en lecture est connue et a déjà été utilisée dans les systèmes d'exploitation sous le nom de* copy on write *[114, page 24] ou les systèmes implémentant une mémoire partagée distribuée [15, 58]. Même si nous poursuivons le même objectif de base, à savoir minimiser les copies inutiles, nous ne cherchons pas à introduire tous les mécanismes nécessaires à la cohérence des différentes copies dans des systèmes DSM [58] (voir aussi Déf. 1.5 page 11).*

Cette extension du modèle de C++// a été réalisée par *David Sagnol* [171, §4.5], suivant les spécifications que nous avons élaborées conjointement. Nous présentons ci-après le mécanisme §3.5 et les résultats de son application à Active-LAKe §3.6.

3.5 Le mécanisme SOR - *SharedOnRead*

Le mécanisme de partage en lecture est réalisé automatiquement par tous les objets d'une classe dérivant de *SharedOnRead*. La stratégie de partage est illustrée par la figure 3.6. Lorsque deux sous-systèmes communiquent, via un appel de méthode, les objets passifs passés en paramètres sont partagés si :

- les 2 objets actifs communicants sont situés dans le même espace d'adressage
- les objets passifs passés en paramètre ont fait l'objet d'une optimisation `SharedOnRead`
- l'accès aux objets passifs est un accès *en lecture*

3.5.1 Spécifications de la classe `SharedOnRead`

L'algorithme de gestion des objets `SharedOnRead` que nous avons spécifié, également présenté dans [171, §4.5.1] est le suivant :

1. lorsqu'un objet `SharedOnRead` est utilisé comme paramètre d'une communication

46. C++// autorise l'implantation des objets actifs résidants sur une même machine sous forme de threads cohabitant au sein d'un même processus lourd [171, §4.2].

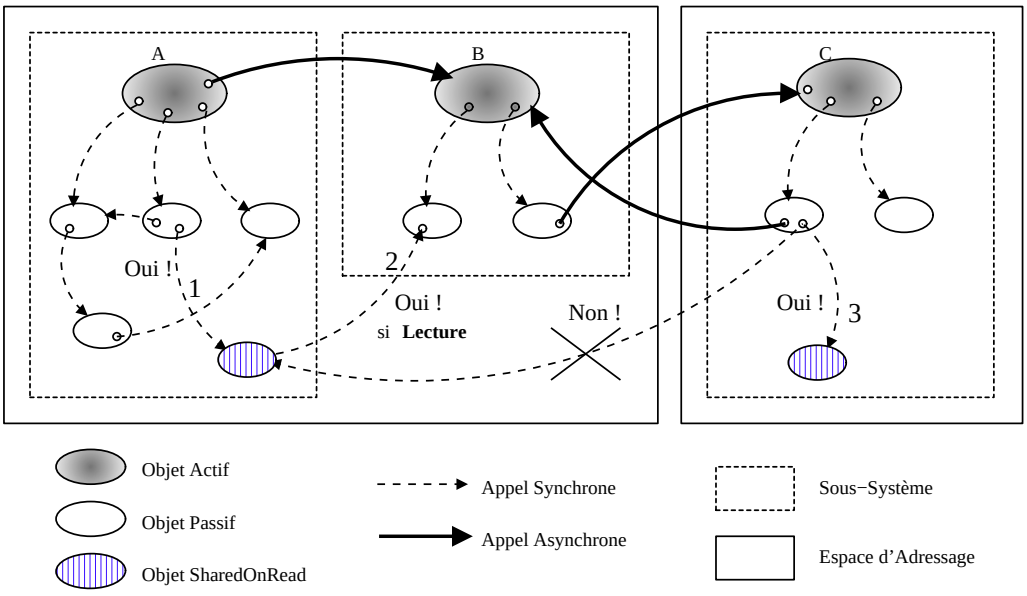

FIG. 3.6 – *Mécanisme SharedOnRead en C++//*

entre deux sous-systèmes situés dans le même espace d'adressage, l'objet original n'est pas copié mais un compteur de référence est incrémenté

2. un accès en lecture à un objet `SharedOnRead` est librement exécuté par tous les sous-systèmes participant au partage
3. un accès en écriture provoque la copie immédiate de l'objet `SharedOnRead` avant que le sous-système ayant demandé l'accès n'ait pu modifier l'objet. Le compteur de référence du nouvel objet est positionné à 1 et l'accès en écriture peut se poursuivre. Le compteur de référence de l'objet initial est décrémenté.
4. une opération de relâche est appelée pour signaler la fin d'un accès en lecture ou en écriture, le compteur de référence est alors décrémenté et l'objet désalloué si le compteur atteint 0.
5. Un objet `SharedOnRead` qui est une donnée membre d'un autre objet ne devrait jamais être détruit par le système sans l'initiative du possesseur. Cela impose la contrainte suivante pour l'utilisation d'un `SharedOnRead` par son possesseur :
   - le possesseur d'un `SharedOnRead` qui accède à l'objet en écriture ne doit *jamais* appeler l'opération relâche sur cet objet sous peine de le détruire.

Les spécifications montrent bien le caractère *semi*-automatique du mécanisme, notamment dans la responsabilité qu'à le programmeur de vérifier la bonne utilisation du `SharedOnRead` par son possesseur. Cette spécification volontairement restrictive pourrait être relachée, mais probablement au détriment de l'efficacité du mécanisme qui guidait nos spécifications.

3.5.2 Réalisation

La classe `SharedOnRead` vient enrichir la librairie de classes de C++// au niveau 1 défini par Europa [193]. Le diagramme de classe de la figure 3.7 précise la place de la classe `SharedOnRead` dans C++// ainsi que son interface. Les méthodes membres de la classe `SharedOnRead` ont la sémantique suivante :

- `read_access(EC_Member_Fct)`
  Cette méthode est utilisée pour spécifier qu'une méthode membre d'une classe dérivant de `SharedOnRead` accède en lecture aux données de l'objet `SharedOnRead`.
- `write_access(EC_Member_Fct)`
  Idem précédemment mais spécifie un accès en écriture. Une méthode membre qui modifie l'état d'un objet *doit* être déclarée comme accédant en écriture à l'objet.
- `access_declaration()`
  Cette méthode abstraite doit être définie dans les descendants de `SharedOnRead` afin de préciser les politiques d'accès des méthodes membres. La politique d'accès par défaut, si elle n'est pas précisée ici est l'accès en écriture.
- `forget()`
  Signifie à l'objet `SharedOnRead` qu'il n'est plus référencé par l'appelant.

L'implantation de la classe `SharedOnRead` dans C++// utilise les mécanismes de réflexivité offert par le niveau 0 d'Europa. C'est pourquoi, comme le spécifie la figure 3.7 la classe `SharedOnRead` hérite de `EC_Reflect`, de ce fait les appels aux méthodes d'un objet

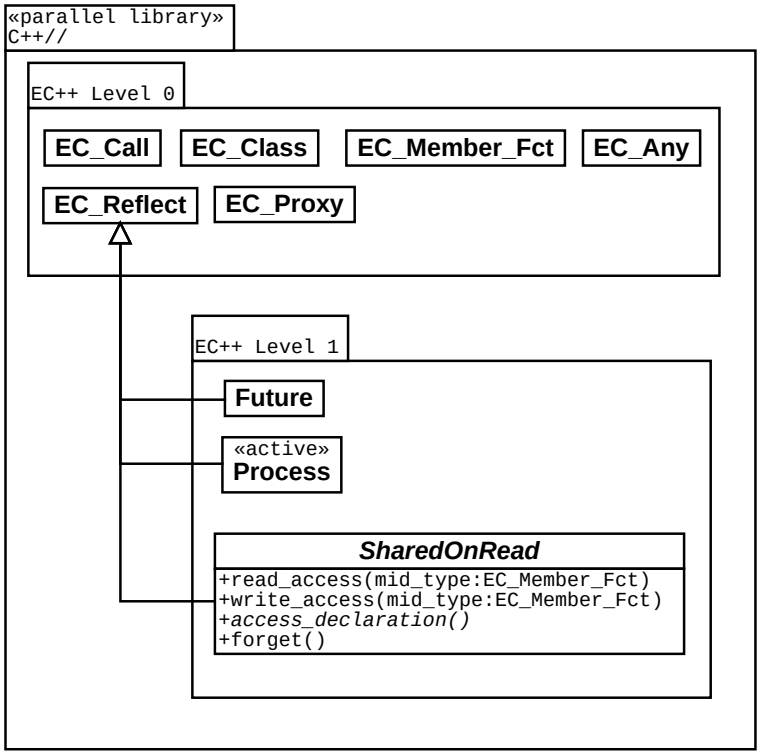

FIG. 3.7 – *Classe* `SharedOnRead` *de C++//*

`SharedOnRead` sont réifiés et traité par un proxy qui gère : le compteur de référence, les copies, la désallocation automatique... On voit ici tout l'intérêt et la puissance d'un modèle réflexif qui permet d'implanter ces mécanismes sans modifier les classes qui veulent en tirer partie.

3.6 Application et résultats

3.6.1 Implantation dans Active-LAKe

L'optimisation *SharedOnRead* de C++// a été introduite dans Active-LAKe en transformant les classes de stockage utilisées par les objets de la classe `CSC_ll` en des objets implantant l'optimisation *SharedOnRead*. Ce changement est spécifié par le diagramme UML de la figure 3.8. Le code de la classe `SOR_Storage` est donné ci-après. Dans cet extrait de code, les méthodes `write_access` et `read_access` sont héritées de la classe `SharedOnRead` et sont utilisées pour spécifier la politique d'accès des méthodes de la classe héritère de `SharedOnRead`. En réalité, seule les spécifications d'accès en lecture sont nécessaires car la politique par défaut est de supposer un accès en lecture. Grâce à cette spécification le mécanisme de partage en lecture peut être mis en œuvre car on sait (vus que les appels aux méthodes d'une classe dérivant de `SharedOnRead` sont réifiés) si l'appel à une méthode sur un objet risque de modifier l'objet.

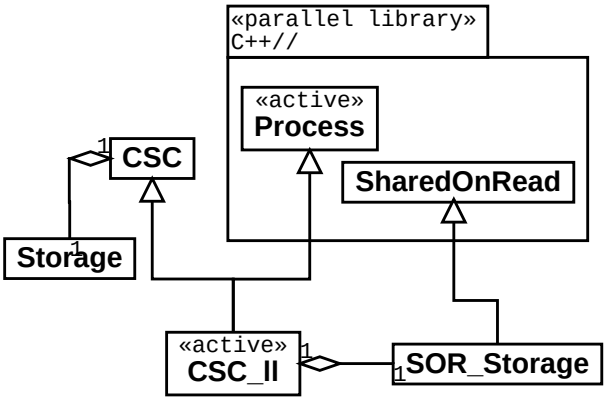

FIG. 3.8 – *Classe* `CSC_ll` *avec stockage* SharedOnRead

Programme 3.4: *La classe* `SOR_Storage`

```
1  class SOR_Storage: public SharedOnRead,
2                     public Storage {
3  public:
4    SOR_Storage();
5
6    virtual void access_declaration() {
7      write_access(mid(update)); // optionel
8      read_access(mid(reach));
9    }
10 };
```

3.6.2 Performance

Des expérimentations numériques ont été effectuées pour les différentes opérations spécifiées au paragraphe 2.5.1. La taille des matrices impliquées dans les calculs est rappelée dans le titre de chaque figure présentant les résultats. Ces tests ont été réalisés sur 4 stations SUN Ultra 1 avec 128 Mo de mémoire tournant sous Solaris, les machines étaient reliées par un réseau ethernet à 10Mb. La version MPI a été développée et exécutée sur ce même réseau de machines avec LAM 6.1 [102]. La figure 3.9 montre le gain obtenu par l'optimisation *SharedOnRead* pour une opération de type SAXPY. Les performances de la version C++// avec `SharedOnRead` sont comparables à celles obtenues avec MPI. On remarquera que la version C++// standard devient de plus en plus coûteuse car le surcoût est proportionnel à la taille des données copiées qui augmente avec le nombre de colonnes. La figure 3.10 montre l'accélération obtenue pour la même opération. Une scalabilité quasi-parfaite est obtenue pour les versions MPI et C++// car cette opération se fait sans aucune communication. La figure 3.11 présente les résultats d'une opération de type GTAXPY entre matrices pleines. La version C++// présente un surcoût par rapport à la version MPI venant du fait que l'opération de réduction nécessaire au calcul est faîte

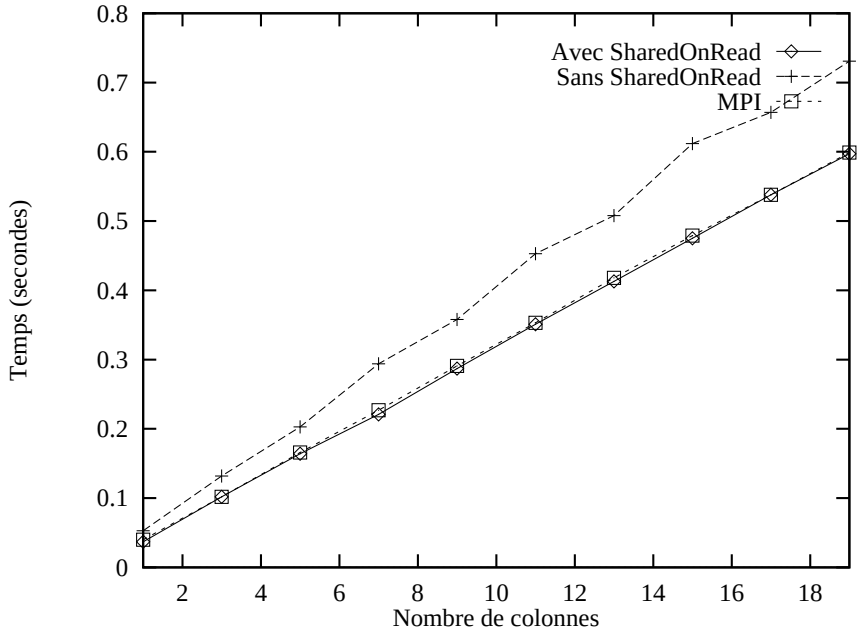

FIG. 3.9 – *SAXPY matrice pleine* $90449 \times j$, $1 \leq j \leq 19$

séquentiellement car elle est réalisée par le code séquentiel (voir ligne 16 programme 3.3). Dans le cas de MPI la réduction est faîte par un algorithme parallèle en $\mathcal{O}(\log_2 nproc)$ mais l'implantation avec MPI de l'opération GTAXPY a nécessité la ré-écriture de cette méthode afin qu'elle contienne les communications explicites inhérentes à MPI. La figure 3.12 présente les performances d'une opération GAXPY utilisant une matrice creuse de taille 21200×21200 comportant 1488768 élément non nuls. Pour cette opération les réductions de la version C++// ont dû être ré-écrites afin d'implanter une version parallèle compétitive avec la version MPI. Dans cette expérience nous avons également testée une deuxième architecture de machine qui est plus hétérogène. 2 des quatres stations Ultra 1 ont été remplacées par des Ultra 2 ayant 192 Mo de mémoire (au lieu de 128 Mo pour les Ultra 1). Avec cette architecture hétérogène, l'implantation avec C++// tire partie des communications asynchrones et de l'attente par nécessité automatiquement fournies par le langage ce qui permet aux machines rapides de commencer l'opération de réduction alors que les machines plus lentes n'ont pas terminé leur calcul. L'implantation utilisant MPI ne tire pas partie des machines les plus rapide car les opérations de réductions constituent une barrière de synchronisation forçant les plus rapides à attendre les plus lentes.

3.7 Conclusion

Nous avons expérimenté l'utilisation d'un modèle de programmation parallèle orientée-objet à objets actifs au travers du système de compilation et d'exécution C++//[27]. Nous avons vu que ce modèle permettait une très bonne ré-utilisabilité séquentielle mais que la sémantique de partage des objets passifs devait être améliorée sous peine de performances inacceptables pour un bon passage à l'echelle. Nous avons spécifié un mécanisme

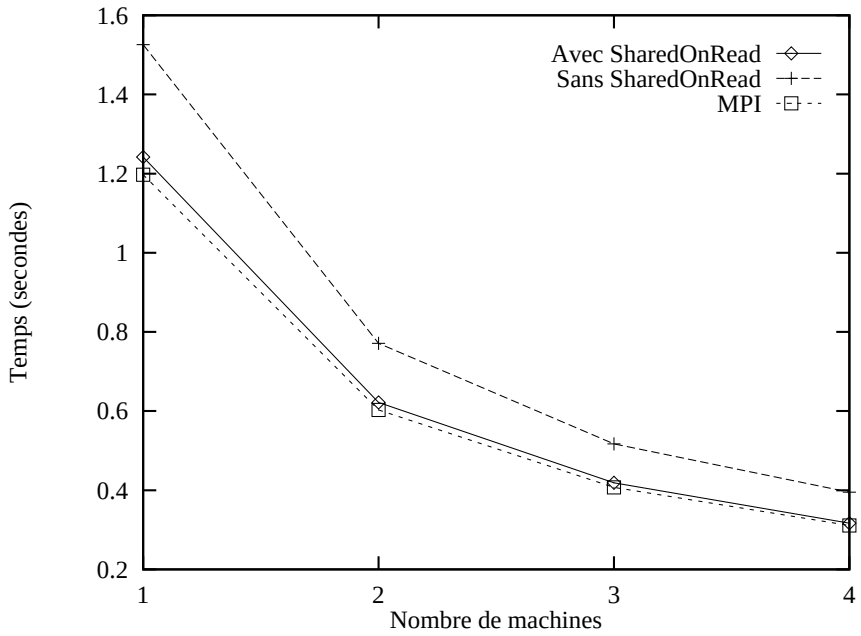

FIG. 3.10 – *Accélération SAXPY matrice pleine* 90449×10

de partage en lecture nommé *SharedOnRead* qui autorise le partage contrôlé des objets passifs entre les sous-systèmes d'une application à base d'objets actifs. Cette optimisation a été implantée dans C++// par les auteurs du système et les tests effectués sur la base des opérations implantées dans notre librairie d'algèbre linéaire LAKe. Les tests ont montré que l'optimisation *SharedOnRead* rendait l'implantation C++// de notre librairie, Active-LAKe, compétitive avec une implantation en MPI.

Avec cette optimisation la ré-utilisablité est un peu moins bonne car il a fallu réécrire plus de code, notamment les opérations de réduction mais elle est néanmoins bien meilleure qu'avec MPI puisqu'elle est non intrusive. Notre expérience montre qu'un modèle à objets actifs, naturellement plus proche d'un modèle à parallélisme de tâches peut être utilisé pour implanter des opérations essentiellement data-parallel de façon efficace. Bien qu'ayant été implantée dans C++// notre approche pourrait être implantée dans n'importe quel langage à objets parallèle utilisant des objets actifs. Nous avons également vu l'intérêt de la réfléxivité pour l'implantation de tels mécanismes.

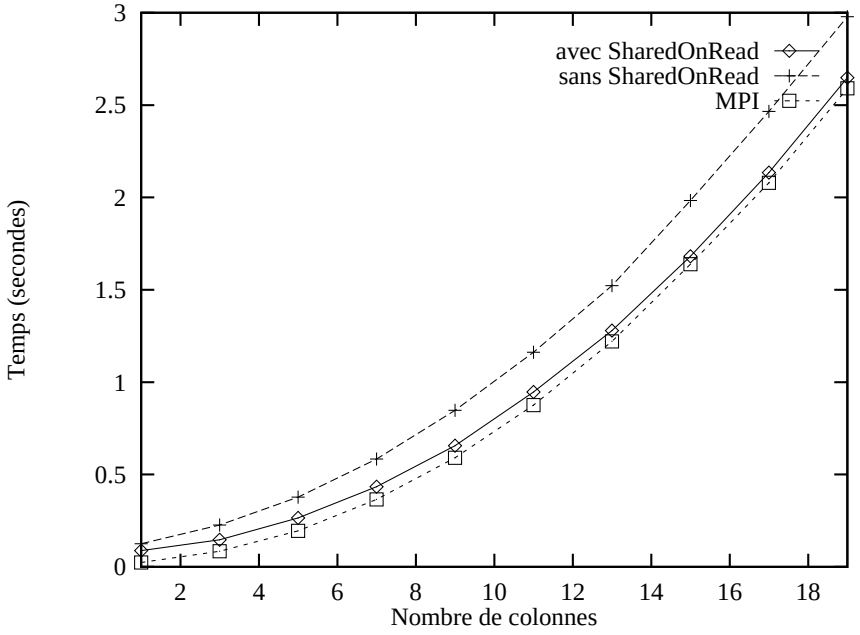

FIG. 3.11 – *GTAXPY matrice pleine* $90449 \times j,\ 1 \leq j \leq 19$

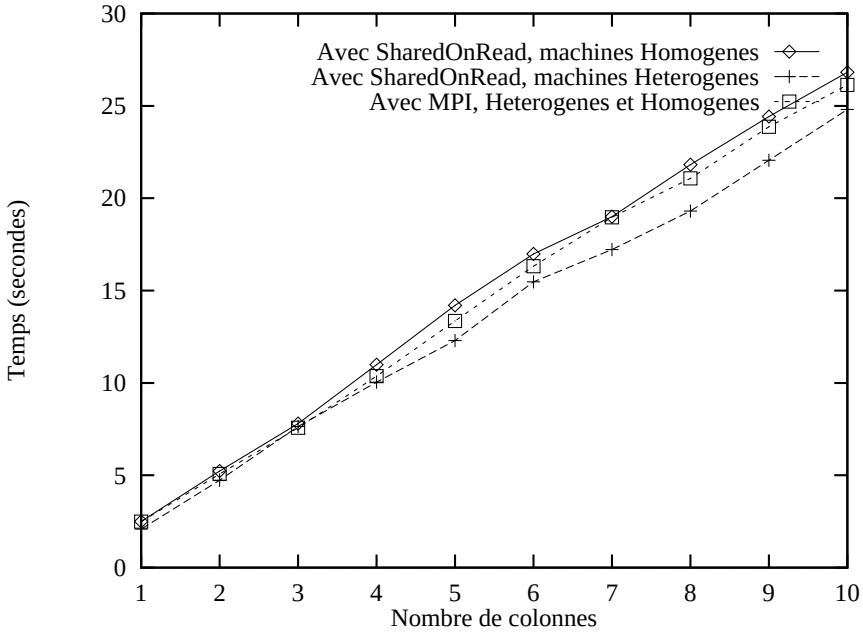

FIG. 3.12 – *GAXPY creux/plein* $21200 \times 21200 * 21200 \times j,\ 1 \leq j \leq 10$

Chapitre 4

Gene-LAKe : vers plus de réutilisabilité

Sommaire

4.1 Introduction

Nous avons vu précédemment quelle ré-utilisabilité et quelles performances un modèle de programmation à objets actifs pouvait fournir à nos applications. Cette précédente étude a été faîte dès le début du développement de LAKe et s'est donc concentrée sur les opérations matricielles élémentaires. Parallèlement au développement d'Active-LAKe nous avons développé une version de LAKe utilisant MPI §4.2 et mettant en œuvre le principe d'encapsulation pour isoler le parallélisme. Durant cette étude, d'autres problèmes sont apparus dont certains, comme le traitement polymorphe des objets parallèles

(§4.3), n'existent pas dans Active-LAKe et d'autres, comme l'allocation d'objets distribués (§4.3.3), seraient également apparus si notre développement avait atteint ce stade. Ce chapitre présente donc l'étude d'un modèle de programmation parallèle alternatif à un modèle à objet actifs pour l'implantation de LAKe. Nous montrons les limites du polymorphisme pour l'encapsulation du parallélisme dans notre librairie au §4.3, nous expliquons ensuite pourquoi la généricité est un élément essentiel de réutilisabilité en introduisant la notion de forme de matrice au §4.4. Les résultats présentés dans ce chapitre ont fait l'objet d'une communication dans un colloque international [129] et d'un article publié dans un journal international [130].

4.2 Une implémentation de LAKe avec MPI

Une contrainte forte pour atteindre nos objectifs de ré-utilisabilité est également la portabilité qui passe par la disponibilité des outils de développement (langages, compilateurs) et de déploiement (support d'exécution) sur les architectures cibles. MPI [118, 119, 120] s'imposant comme un standard de programmation parallèle pour les application numériques, à la fois disponible sur quasiment toutes les architectures parallèles et ayant prouvé son efficacité, il était naturel d'étudier une réalisation de notre librairie s'appuyant sur MPI.

4.2.1 Les avantages et les inconvénients

MPI présente un certain nombre d'atouts pour satisfaire nos objectifs de portabilité et d'efficacité :

- standard bien établi [118] et évoluant [119],
- librairie parallèle multi-langages [118] (C, Fortran 77), [119] (C++, Fortran 90),
- disponibilité sur un grand nombre d'architectures parallèles (implantation multiple[47] du standard par les constructeurs de machines parallèles ou par des tiers),
- modèle de programmation multi-MIMD très général.

Ces atouts garantissent qu'en utilisant MPI pour la version parallèle de notre librairie, celle-ci sera portable et qu'aucune limitation du modèle de programmation parallèle ne risque de nous gêner. En revanche, le fait que MPI soit une librairie parallèle proposant un modèle de programmation explicite (*passage de messages*, voir Déf. 1.12), est indicateur de frein à la réutilisabilité. Parmi les désavantages prévisibles de MPI, en terme de réutilisabilité, on peut citer :

- la parallélisation explicite → *réutilisation parallèle plus difficile*,
- la parallélisation intrusive → *problème de réutilisation séquentielle/parallèle*,
- la très faible intégration des concepts objets dans la librairie → *réutilisation de concepts de haut niveau moins aisée*.

47. `http://www.mpi.nd.edu/MPI/`

4.2.2 Une approche objet avec MPI

La conception d'une approche objet avec une librairie de passage de messages comme MPI est relativement simple et peut se décomposer en 2 étapes :

1. création d'une surcouche orientée-objet minimale à la librairie de passage de message. Cette surcouche se contente de définir les classes correspondantes aux objets et concepts utilisés dans les librairies de passages de messages :
   - Messages,
   - Port de communication,
   - Groupe de ports de communication (Communicateurs MPI).

   Les méthodes de ces classes implanteront les fonctions de communications qui les concernent. Cette couche doît être aussi légère que possible en terme de surcoût par rapport aux performances initiales de la librairie de passage de messages utilisée.
2. encapsulation du parallélisme dans les classes de la librairie cible devant être parallèles ou distribuées. L'implantation de ces classes parallèles utilisera la surcouche objet minimale précédente.

Cette approche objet, en 2 étapes, permet d'une part, de ne pas introduire une programmation hybride objet/procédurale et d'autre part de cantonner les aspects parallèles de la librairie uniquement là où ils doivent apparaître. De cette manière, les classes parallèles dériveront de leurs homologues séquentielles et les classes ne nécessitant pas *explicitement* un traitement parallèle pourront utiliser des classes parallèles de façon polymorphe.

Concernant la première étape, après avoir fait notre propre analyse des besoins, nous avons examiné deux librairies : Object-Oriented MPI [185] et Para++ [149]. La première correspondait quasiment à ce que nous aurions réalisé si nous l'avions fait. Nous avons donc décidé de l'utiliser. OOMPI est une mince [162] surcouche orientée-objet à MPI, qui s'appuie uniquement sur une implantation de MPI-1.1 et un compilateur C++ standard. Grâce à cela, nous avons pu la compiler et l'utiliser aussi bien sur un réseau de stations utilisant LAM [102] comme implantation de MPI, que sur un CRAY T3E en utilisant la librairie MPI fournie par le constructeur. Une autre approche un peu plus « minimaliste » aurait consistée à utiliser l'interface C++ pour MPI-1.2 définie par MPI-2 [119, Chap. 10 et Annexe B]. Celle-ci n'était, d'une part, pas implantée dans toutes les réalisations de MPI que nous utilisions et ne constituait pas, d'autre part, une solution aussi « proche » que celle que nous aurions implantée.

La seconde étape d'encapsulation du parallélisme et d'utilisation polymorphe des objets parallèles est décrite dans le paragraphe suivant. Cette étape, même si elle ne constitue pas en tant que telle une contribution de cette thèse, nous a permis d'exhiber les différences, en terme de réutilisabilité, entre un LAO parallèle comme C++// et une parallélisation explicite avec MPI suivant une démarche objet.

4.2.3 Encapsulation du parallélisme et polymorphisme

Le principe orienté-objet de l'encapsulation (voir Déf. 1.17 page 29) appliqué au parallélisme implanté avec MPI/OOMPI est simple. Les appels explicites à OOMPI n'apparaîtront que dans les classes nécessitant un traitement parallèle explicite. En reprenant le

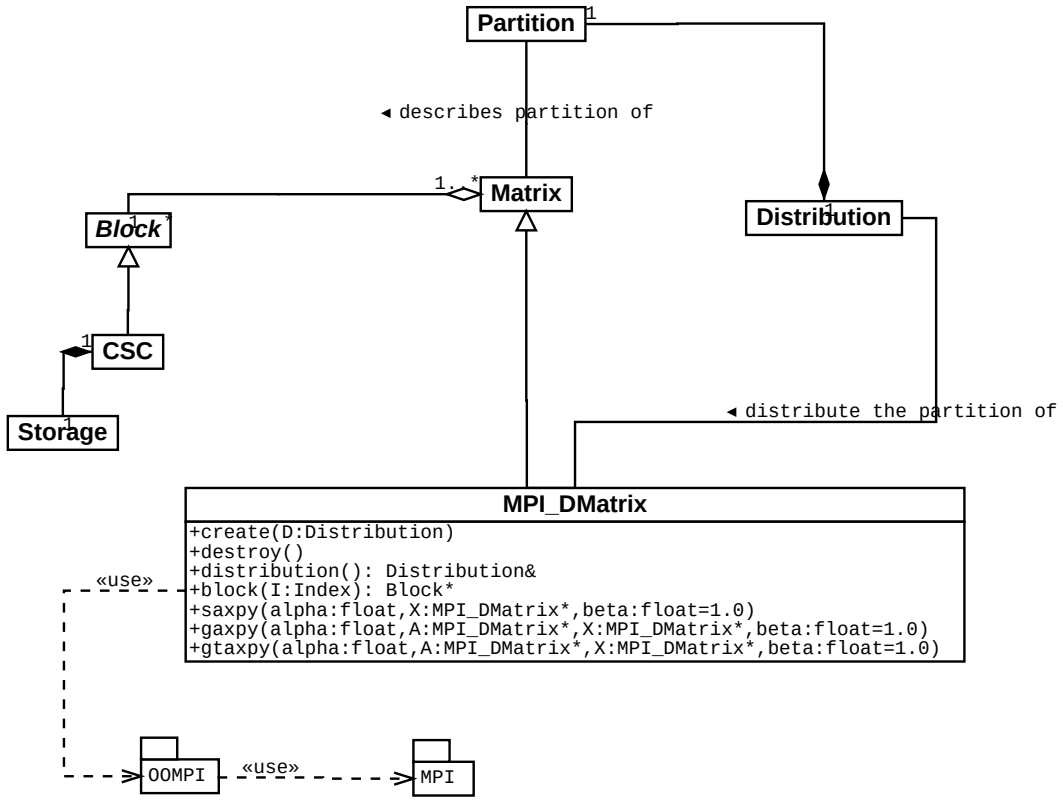

FIG. 4.1 – *Encapsulation du parallélisme*

diagramme de classes de LAKe séquentiel, présenté au chapitre précédent en figure 3.2, et le raisonnement de parallélisation de ce même chapitre, il en découle que seules les classes implantant les matrices doivent être parallélisées. On appelle la classe réprésentant des matrices parallèles implantées avec MPI, `MPI_DMatrix`. Cette classe dérive de la classe séquentielle `Matrix` et redéfinie toutes les opérations matricielles afin qu'elles aient une sémantique parallèle, comme illustrée à la figure 4.1. De la même façon que pour Active-LAKe, les classes clientes de `Matrix` utiliseront de façon polymorphe les objets instances de `MPI_DMatrix`. Cette utilisation polymorphe est illustrée par le diagramme de classes UML de la figure 4.2, pour les classes `Arnoldi`, `Arnoldi_EV` et toutes les méthodes itératives.

Moins de réutilisabilité séquentielle/parallèle

La première différence à noter avec Active-LAKe est le niveau auquel intervient la parallélisation. Avec un modèle de programmation par passage de messages comme MPI, il est nécessaire que les classes parallèles aient la connaissance explicite de la distribution des données qu'elles manipulent. Dans le modèle de programmation SPMD que nous avons adopté, cela se traduit par des appels à des méthodes de OOMPI indiquant sur quel processeur s'exécute le code des méthodes de `MPI_DMatrix`. Ainsi, on note que même si ce sont les objets de la classe `CSC`, contenus dans un objet de classe `MPI_DMatrix`, qui sont

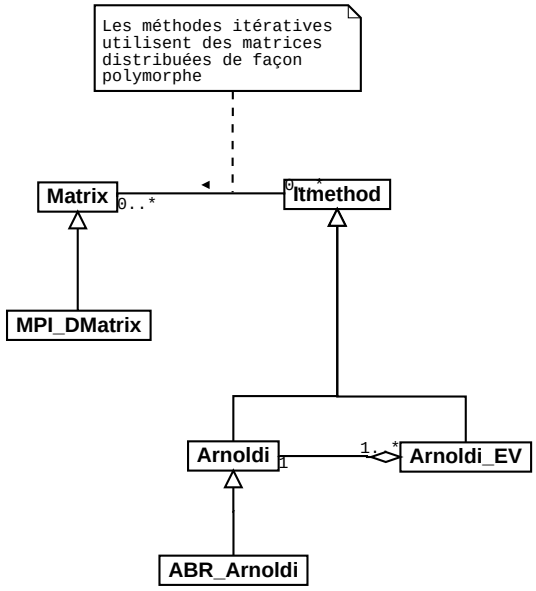

FIG. 4.2 – *Utilisation polymorphe de* `MPI_DMatrix`

distribués, c'est la classe les « possédant » (c'est-à-dire `MPI_DMatrix`) qui gère explicitement leur distribution. Ceci implique une baisse de réutilisabilité séquentielle/parallèle si l'on compare `Matrix_ll` et `MPI_DMatrix`. En effet, lorsque `Matrix_ll` n'implante que les constructeurs parallèles nécessaires à la distribution initiale des blocs parallèles (`CSC_ll`) et réutilise *complètement* les autres méthodes héritées de `Matrix`, la classe `MPI_DMatrix`, ré-implante quasiment toutes les méthodes de `Matrix`. En effet, en tant que modèle de programmation à parallélisme explicite, le modèle par passsage de messages implique des communications explicites. Ces communications, gérées automatiquement par la couche réfléxive de C++// générée à la précompilation, le sont explicitement dans les méthodes de `MPI_DMatrix`. On peut également remarquer qu'à l'inverse, la parallélisation avec MPI ne nécessite l'écriture que d'une seule classe parallèle : `MPI_DMatrix` contre deux classes (`Matrix_ll` et `CSC_ll`) pour C++//. La quantité de code à ré-écrire reste en faveur de C++//.

4.3 Polymorphisme, généricité et parallélisme

L'utilisation du polymorphisme comme instrument de réutilisation séquentielle/parallèle pose un problème pour les méthodes prenant comme argument des variables de la même classe que celle à laquelle appartient la méthode. C'est le cas pour `Matrix::axpy(alpha: float, A: Matrix*, X: Matrix*, beta: float)`, défini dans la classe `Matrix`, qui prend comme arguments (`A` et `X`), deux objets de la classe `Matrix`. Nous expliciteront le problème au §4.3.1 et proposerons une solution au §4.3.2 sous la forme d'un nouveau motif de conception. Nous verrons ensuite au §4.3.3 pourquoi la création de matrices distribuées fait du polymorphisme un mécanisme insuffisant pour réaliser une version parallèle de notre librairie satisfaisant nos critères de réutilisation. Nous montrerons que la généricité

est un très bon moyen de solutionner les deux problèmes précédents et nous détaillerons cette solution au §4.4 en introduisant la notion de matrices avec forme qui est une des contributions de cette thèse.

4.3.1 Contravariance

Le problème de la contravariance apparaît dans les langages à objets lorsque l'on pense que la relation d'héritage définit également une relation de sous-typage. La définition 4.1 rappelle la notion de sous-type.

DÉFINITION **4.1 (Sous-type)** *Un type T' est un* sous-type *du type T, que l'on note $T' \leq T$, si et seulement si toute fonction prenant comme argument un objet de type T accepte à la place un argument de type T'.*

La contravariance apparaît lorsque l'on applique la définition précédente au sous-typage de fonctions. La définition 4.2, donne la règle de sous-typage appliquée aux fonctions et explique la notion de contravariance. La contravariance n'est pas une notion « naturelle », des exemples détaillés et des références théoriques sont donnés dans [84, 30, 31].

DÉFINITION **4.2 (Contravariance)** *Soit $TA \rightarrow TR$ le type d'une fonction prenant comme argument un objet de type TA et retournant une valeur de type TR. La règle de sous-typage pour les fonctions est la suivante : $TA' \rightarrow TR' \leq TA \rightarrow TR$ ssi $TR' \leq TR$ et $TA \leq TA'$. On dit que le type de retour de la fonction est* covariant *car il varie dans le même sens que le type des fonctions, mais que le type des arguments est* contravariant *car la relation de sous-typage est inversée.*

Afin d'illustrer ce problème dans LAKe nous notons que l'exemple du programme 3.2 qui fonctionnait avec Active-LAKe, présente, dans sa version MPI encapsulée, un problème de contravariance à la ligne 15 du programme 4.1.

Programme 4.1: *Mauvais aiguillage à cause d'une redéfinition covariante de méthode*

```
1  /* An Arnoldi   constructor *
2   * expecting Matrix object */
3  Arnoldi::Arnoldi(Matrix* A, Matrix* X, Matrix* Y)
4  {
5     A_ = A;    X_ = X;    Y_ = Y;
6  }
7
8  Arnoldi::do_axpy()
9  {
10    /* Here we do call                                  *
11     * Matrix::gaxpy(float,Matrix*,Matrix*,float)      *
12     * instead of the expected                         *
13     * MPI_DMatrix::gaxpy(float,MPI_DMatrix*,          *
14     *                    MPI_DMatrix*,float)          */
15    Y_->gaxpy(1.0,A_,X_);
16 }
17
```

```
18 void main()
19 {
20   MPI_DMatrix  A_mpi, X_mpi, Y_mpi;
21   Distribution DA,    DX,    DY;
22
23   Arnoldi* an_Arnoldi;
24
25   [...]
26   /* Create the distributed matrices */
27   A_mpi.create(DA);
28   X_mpi.create(DX);
29   Y_mpi.create(DY);
30   [...]
31   /* Pass parallel matrices polymorphically *
32    * to the Arnoldi constructor.            */
33   an_Arnoldi = new Arnoldi(A_mpi, X_mpi, Y_mpi);
34   /* This call  will fail and raise *
35    * a bad dispatch fatal error     */
36   an_Arnoldi->do_axpy();
37   [...]
38 }
```

En effet, lors de l'appel à la méthode `gaxpy` les variables `Y_`, `A_` et `X_` sont du type `Matrix*`, même si leur type dynamique est `MPI_DMatrix*`, puisqu'ils sont traités de façon polymorphe. C'est-à-dire que le constructeur de l'objet `Arnoldi` à la ligne 33 a accepté les arguments en les considérant comme des sous-types de `Matrix*`. Le problème est la règle de sous-typage des fonctions (Déf. 4.2), qui indique que la méthode

`MPI_DMatrix::gaxpy(float,MPI_DMatrix*,MPI_DMatrix*,float)`

n'est pas un sous-type de

`Matrix::gaxpy(float,Matrix*,Matrix*,float)`

car les arguments en entrée doivent être contravariants. La méthode `MPI_DMatrix::gaxpy` que nous avons spécifiée, n'est donc pas une rédéfinition de `Matrix::gaxpy` mais une surcharge, qui ne permet pas le polymorphisme dynamique via une liaison tardive (cf. §1.4.1 Déf. 1.20). La seule redéfinition valable dans ce cas est une redéfinition *invariante* soit

`MPI_DMatrix::gaxpy(float,Matrix*,Matrix*,float)`.

Il s'ensuit donc que le type effectif des arguments doît être vérifié, on appellera cela l'aiguillage : simple, double ou multiple[48]. Le problème de l'aiguillage multiple, que l'on trouve dans les méthodes dîtes binaires [24], est un problème classique des langages à objets et a été résolu de différentes façons dans le passé. Les solutions ne sont généralement pas intégrées directement dans les LAOs car elles sont coûteuses (en temps d'exécution).

REMARQUE **4.1 (Polymorphisme Multiple)** *L'aiguillage multiple est également appelé* polymorphisme multiple *(English : multiple polymorphism [140]). On parle également*

48. en anglais « single, double and multiple dispatch »

de méthodes multiples *(English : multimethods). La dénomination multiple, concernant le polymorphisme, vient du fait que le polymorphisme « classique » correspond en fait à l'aiguillage simple (English : single-dispatch) qui est fourni par tous les langages à objets, par le biais de la redéfinition de méthodes et du mécanisme de liaison dynamique.*

Ce même problème a été noté et résolu dans le même contexte d'algèbre linéaire par *F. Guidec* [83, pages 96–99 et §4.4 pages 130–143] pour la librairie PALADIN. La solution de l'auteur consiste à contrôler dynamiquement et explicitement le type des arguments en utilisant les fonctionnalités de contrôle dynamique de type d'Eiffel.

Nous proposons dans le paragraphe suivant une solution améliorée sous la forme d'un motif de conception. Celui-ci a deux avantages par rapport à PALADIN, l'aiguillage d'un argument n'est réalisé que lorsque celui change entre deux appels successifs, un seul aiguillage peut-être fait pour des appels à plusieurs fonctions utilisant le(s) même(s) argument(s).

REMARQUE **4.2 (Erreur de conception ?)** *On pourrait croire que le problème précédent vient d'une erreur de conception de la classe* `Matrix`*. Il suffirait de fournir une interface de bas niveau à cette classe, comme une indexation par point $A(i,j)$, pour réutiliser la méthode* `Matrix::gaxpy` *dans* `MPI_DMatrix`*. En effet, l'indexation par point ne souffre pas du problème de covariance/contravariance. Cette réalisation serait extrêmement inefficace en environnement parallèle car elle générerait un message pour chaque accès à un élément.*

REMARQUE **4.3 (Type ancré)** *Certains langages comme Eiffel offrent une autre solution au problème de contravariance dans les méthodes binaires : les types ancrés. Une variable est de type ancré, si son type est spécifié comme étant* le-même *que celui d'une autre variable, fonction ou type qui peut être en cours de définition*[49]*. Ce mécanisme de type ancré permet la (re)définition covariante de méthodes d'un objet. Nous n'avons pas considéré ce mécanisme comme une solution car :*

1. *notre langage cible, C++, ne comporte pas ce mécanisme et n'accepte que la redéfinition* invariante *des méthodes de classe,*
2. *les types ancrés permettent de « briser » le système de type du langage [112, pp. 621–642].*

4.3.2 Motif de conception « Service »

La définition même d'un motif de conception, sous-entend que cette conception a été vue et revue de telle façon qu'elle constitue un *motif* récurrent en matière de conception indépendamment de son implantation. Le qualificatif de motif utilisé par la suite, est donc un peu usurpé, il se justifie par le fait que notre motif *Service* peut être vu comme une spécialisation du motif *Strategy* [76] dans lequel les paramètres nécessaires à l'invocations des algorithmes peuvent être sélectionnés un à un. Le motif de conception « Service », que nous appellerons *motif Service* ou simplement *Service*, est la réification d'une méthode multiple. Dans notre cas la méthode `Matrix::gaxpy` devient le *Service* `AXPY_Operator`.

49. voir « *like Current* » en Eiffel [112, pages 601–]

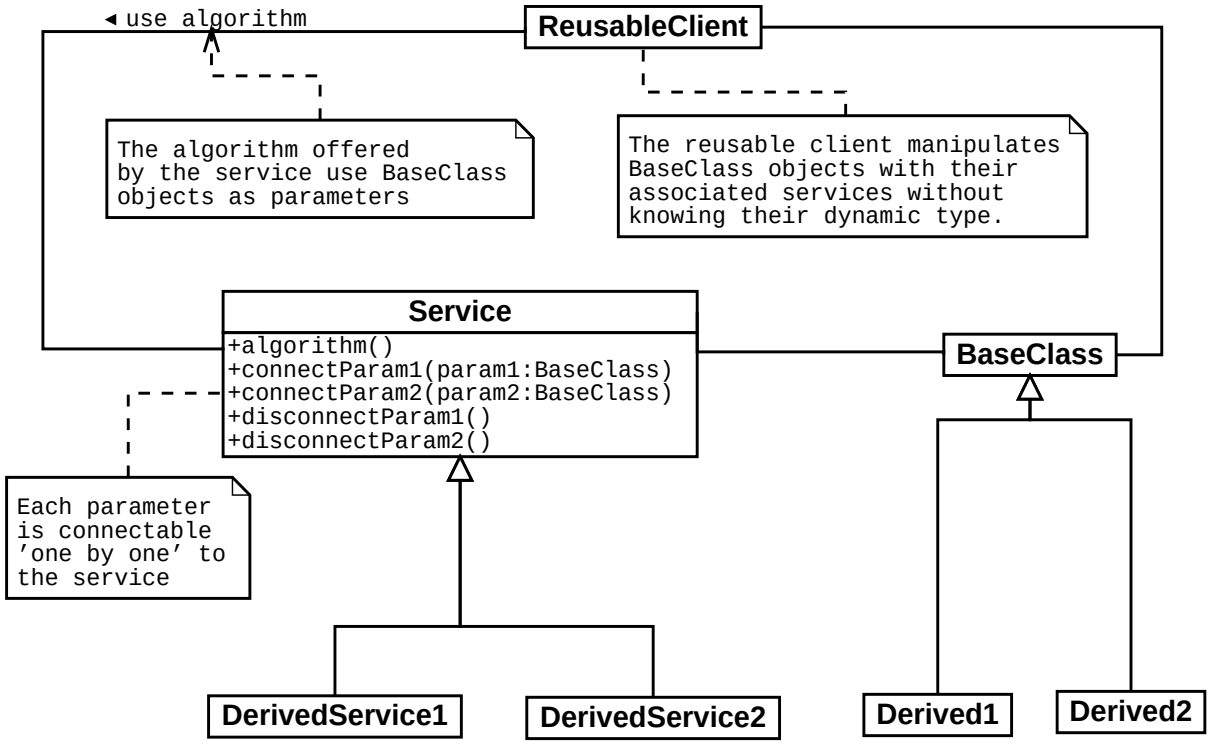

FIG. 4.3 – *Description du motif* Service *en UML*

Le motif *Service* est inspiré du motif *Visiteur* [76, Visitor Pattern], qui permet d'implanter un aiguillage double dans un langage fournissant l'aiguillage simple. Le motif *Service* peut d'ailleurs être implanté aussi bien à l'aide du motif *Visiteur* qu'avec les fonctionnalités de contrôle dynamique de type du langage cible[50]. Un *Service* représente un ensemble d'opérations, qui connectent (ou inscrivent) leurs arguments un par un. Le *Service* possède un état interne qui spécifie quelles opérations peuvent être invoquées sur le *Service*. Chaque opération fournie par le *Service*, vérifiera cet état interne et signalera une erreur si le *Service* est dans un état ne permettant pas de servir la requête. Ceci signifie que, lors de l'exécution, une méthode du *Service* ne peut être invoquée que si l'état de l'objet le permet. Une description du motif *Service* en UML est donnée à la figure 4.3. Un exemple de réalisation d'un *Service*, `AXPY_Operator`, et son utilisation sont présentées dans le programme 4.2.

Programme 4.2: *Exemple de* Service *AXPY*

```
1 // Declaration
2 Matrix           *A, *B, *C;// pointer to matrix objects
3 AXPY_Operator*  AXPY;       // a pointer to an AXPY service
4
5 // polymorphically assign a distributed AXPY service
6 AXPY = new Distributed_AXPY_Operator();
```

50. en C++ le RTTI (**R**un **T**ime **T**ype **I**dentification).

```
7
8  // Configure default service parameter
9  AXPY->alpha() = 1.0;
10 AXPY->beta()  = 0.0;
11
12 // polymorphic assignment
13 A = new MPI_DMatrix();
14 B = new MPI_DMatrix();
15 C = new MPI_DMatrix();
16 [...]
17 // Compute A = B * C
18 AXPY->connect(A,"Y"); // register A as "Y"
19 AXPY->connect(B,"A"); // register B as "A"
20 AXPY->connect(C,"X"); // register C as "X"
21 AXPY.gaxpy();         // compute A = B * C
22 // Compute C = B^{T} * A
23 AXPY->disconnect("Y"); AXPY->connect(C,"Y");
24 AXPY->disconnect("X"); AXPY->connect(A,"X");
25 AXPY->taxpy();        // compute C = B^{T} * A
26 AXPY->disconnect();
```

Dans cet exemple, on voit que désormais, les méthodes itératives devront utiliser non seulement des matrices mais des services de multiplication de matrices, c'est-à-dire des objets dont la classe dérive de `AXPY_Operator`. Les objets `Matrix` et `AXPY_Operator` pourront être traités de façon polymorphe par les méthodes itératives sans poser de problème de contravariance, que les types dynamiques des services ou des matrices correspondent à des implantations séquentielles ou parallèles.

Un avantage du motif *Service* (comparé à la solution adoptée dans PALADIN) est que les opérations logiquement liées entre elles peuvent être regroupées au sein d'un même service offrant une solution complète à un problème particulier. Dans l'exemple du service `AXPY_Operator`, toutes les opérations de multiplication matricielles sont offertes par le service (SAXPY, GAXPY, TAXPY). Les relations entres les différentes classes concernées par le service `AXPY_Operator` sont résumées par le diagramme de classes de la figure 4.4. Le seul inconvénient est la syntaxe non naturelle (`connect`/`disconnect`) qui peut être améliorée, si l'on autorise les connexions/deconnexions automatiques. Si, par exemple, le paramètre « `A` » du service `AXPY_Operator` ne change pas, l'appel suivant à `AXPY_Operator::gaxpy(X: Matrix*, Y: Matrix*)` provoquera la connexion automatique de `X` et `Y`, l'appel à `AXPY_Operator::gaxpy()` et enfin la deconnexion de `X` et `Y`.

Comme nous l'avons déjà signalé, le motif *Service* peut être implanté avec n'importe quel langage à objets, mais il faut noter que le même esprit de service est utilisé dans le motif *Reverse Communication* essentiellement utilisé en Fortran 77 [55]. Ce n'est donc pas surprenant que l'objectif du motif *Reverse Communication* soit de libérer l'implantation des méthodes itératives de leur dépendance vis-à-vis des opérations matrices/vecteurs.

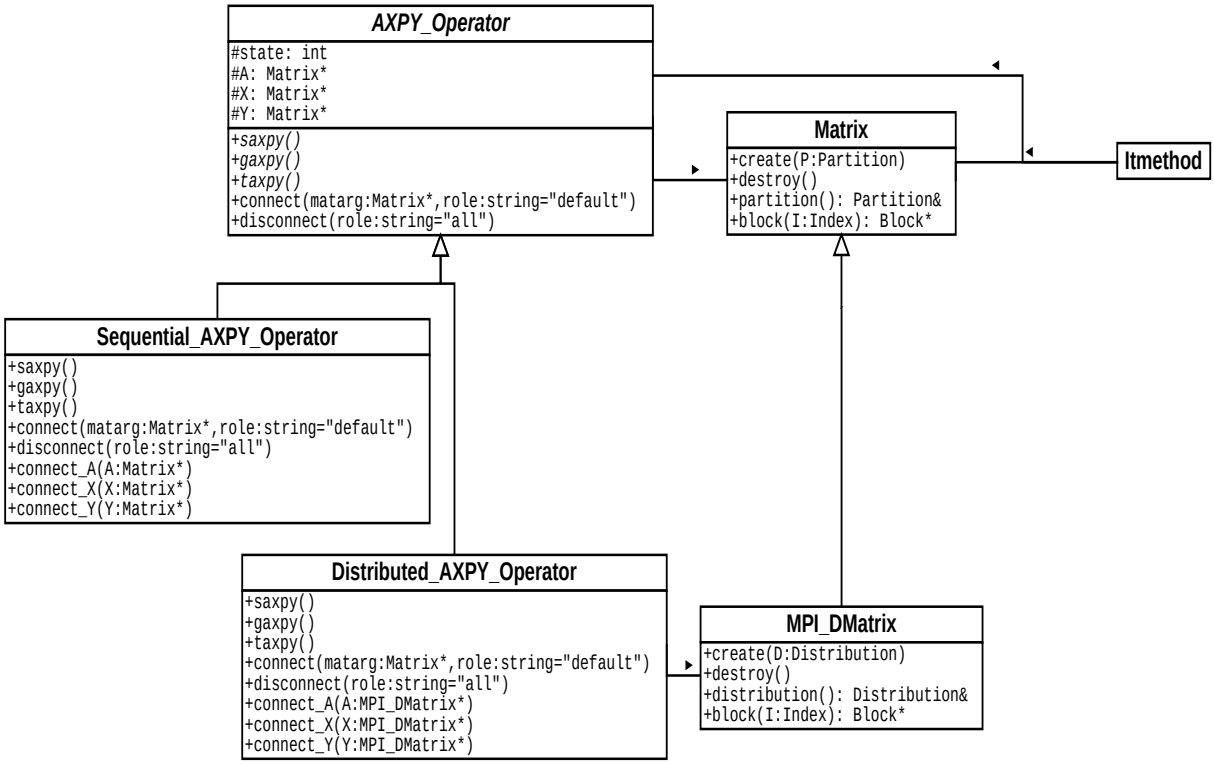

FIG. 4.4 – *Service* `AXPY_Operator` *et classes afférentes*

Autres utilisations

Un autre avantage du motif *Service* est qu'il peut servir à implanter de façon claire et efficace n'importe quel service, qu'il soit séquentiel ou parallèle. La clarté est apportée par le regroupement, au sein d'un même objet, d'opérations logiquement liées entre elles. L'efficacité découle des fonctionnalités de connexion/deconnexion des objets concernés qui permettent de gérer les ressources (mémoire par exemple) nécessaire au service dans le service lui-même. L'exemple du programme 4.3 suivant montre l'utilisation d'un service de factorisation QR de matrice.

Programme 4.3: *Exemple de* Service *de factorisation QR*

```
1  // Declaration
2  Matrix          *R1, *X; // pointer to matrix objects
3  Matrix          *R2, *Y; // pointer to matrix objects
4  QR_Factorizer *QRF;     // a pointer to a QR factorizer service
5  int              rank;
6
7  // create a sequential QR_Factorizer
8  QRF = new Sequential_QR_Factorizer;
9  [...]
10
11 // Compute QR factorization of X
```

```
12 QRF->connect(X);       // register X as the default argument
13 QRF->factorize();
14 QRF->get_R(R1);        // get the R factor of QR factorize
15 QRF->compute_Q();        // compute Q factor which overrides X
16 QRF->disconnect();
17
18 // Compute QR factorization of Y with column pivoting
19 QRF->connect(Y);
20 QRF->enable_column_pivoting();
21 QRF->factorize(); // factorize with column pivoting
22 QRF->get_R(R2)
23 rank = QRF->rank(); // get the rank of QR factorization
24 QRF->compute_Q()
25 QRF->disconnect();
26 [...]
```

Le service fournit toutes les opérations nécessaires à la factorisation QR d'une matrice pleine. Dans l'extrait de programme 4.3 les lignes 12–16 montrent la factorisation QR de la matrice X. Le facteur R est stocké dans $R1$ et le facteur Q remplace X au moment de l'appel à `QR_Factorizer::compute_Q()`. Les lignes 19–25 montrent l'utilisation du même objet service pour calculer une factorisation QR avec pivotage des colonnes (ligne 20), ce qui permet de connaître facilement le rang (ligne 23) de la matrice Y (ou $R2$). L'efficacité de ce service réside dans le fait que la mémoire nécessaire à ces opérations est gérée par le service. De ce fait, lors de la première factorisation une certaine quantité de mémoire de travail est allouée, ce même espace de travail est réutilisé sans réallocation pour la deuxième factorisation, si les dimensions des matrices le permettent. La mémoire utilisée par le service est automatiquement libérée lorsque le service lui-même est détruit. Dans l'implantation actuelle de LAKe, deux services de ce type existent :

- `QR_Factorizer`
  Un service de factorisation QR de matrices pleines,
- `EV_Solver`
  Un service de résolution de problèmes aux valeurs propres pour matrices pleines.

Ces deux services sont en fait implantés en utilisant la librairie LAPACK [103] (version Fortran). L'intérêt du motif *Service* est mieux illustré par la liste des tâches effectuées une fois pour toute par ces services :

1. conversion inter-langages C/C++ <->Fortran,
2. regroupement logique de toute les subroutines LAPACK concernées par le service,
   - `QR_Factorizer`
     `xGEQRF`, `xGEQPF`, `xCOPY`, `xLACPY`, `xORGQR`,
   - `EV_Solver`
     `xGEHRD`, `xORGHR`, `xHSEQR`, `xTREVC`, (`xHSEIN`), `xGEBAL`, `xGEBAK`.
3. gestion de la mémoire temporaire pour les appels aux subroutines Fortran,
4. configuration conviviale des opérations offertes par le service,

`QR_Factorizer::enable_column_pivoting()` permet par exemple d'utiliser simplement une factorisation QR avec pivotage des colonnes. Les différents appels (`xGEQPF` au lieu de `xGEQRF`) et le réordonnancement éventuel des colonnes de R sont automatiquements gérés par le service,

5. vérification de la cohérence des appels,

   - On ne peut pas demander un facteur R ou le rang d'une factorisation QR, avant d'avoir appelé `QR_Factorizer::factorize()`,
   - On ne peut plus obtenir le facteur R avec `QR_Factorizer::get_R(Matrix*)` une fois que l'on a écrasée la factorisation en appelant `QR_Factorizer::compute_Q()`
   - ...

Le motif *Service* permet donc de mettre en œuvre les principes orientés-objet d'encapsulation et de regroupement données/fonctionnalités, tout en préservant les performances, que ce soit pour un modèle de programmation séquentiel ou parallèle. On peut noter sur ce point, que vue l'utilisation initiale du motif *Service* pour effectuer de l'aiguillage dynamique, il est aisé de dériver un service parallèle à partir d'un service séquentiel sans que les objets clients de ces services soient modifiés[51]. Les spécifications du motif *Service* sont données ci-après.

Spécifications du motif *Service*

Les participants au motif *Service* sont :

- une classe *Service*, qui implante les fonctionnalités du motif *Service*,
- une classe de base *Object*, qui n'a d'autre contrainte que de fournir des informations sur les types dynamiques de ses descendants[52],
- autant de classes dérivant d'`Object` que le service doit accepter.

Les méthodes membres d'un service sont les suivantes :

- une méthode redéfinissable `connect(Object* O, string role)` qui se chargera de faire l'aiguillage dynamique du paramètre dont le nom est spécifié par `role` et appellera la méthode de connexion spécialisée pour le type dynamique trouvé. Cette méthode doit être redéfinie par les descendants d'un service, pour gérer les nouveaux types dynamiques gérés par le descendant,
- une méthode `connect_PARM_NAME(DType* O)` pour chaque nom de paramètre nécessaire au service concerné et pour chaque type de ce paramètre accepté par le service. Un appel à cette méthode connectera l'objet concerné au service suivant son rôle et mettra à jour l'état du service,
- une méthode redéfinissable `disconnect(string role)` qui déconnectera le(s) paramètre(s) concerné(s) et mettra à jour l'état du service,

51. A condition que les clients aient prévu d'utiliser ces services de façon polymorphe
52. En C++ ce là signifie que la classe `Object` possède au moins une méthode virtuelle (non pure)

- une méthode `do_task(...)` pour chaque fonctionnalité offerte par le service. Un service peut avoir plusieurs méthodes de ce type. Le service doît vérifier son état avant de servir une requête de ce type. A titre d'exemple le service `QR_Factorizer` offre les méthodes suivantes (liste partielle) :
  - `void factorize()` : calcule la factorisation QR sous une forme compressée,
    pré-condition : la matrice à factoriser doit être connectée,
    post-condition : la matrice connectée au service contient désormais sa factorisation QR sous une forme compressée,
  - `void get_R(Matrix*)` : récupère le facteur R de la factorisation,
    pré-condition : la factorisation doit avoir été calculée par un appel à `factorize()` et le facteur Q ne doit pas avoir été formé par un appel à `void compute_Q()`,
    post-condition : le paramètre contient le facteur R,
  - `int rank()` : renvoie le rang de la factorisation QR,
    pré-condition : la factorisation doit avoir été calculée par un appel à `factorize()` et le facteur Q ne doit pas avoir été formé par un appel à `void compute_Q()`,
    post-condition : -
  - `void compute_Q()` : forme le facteur Q explicitement,
    pré-condition : la factorisation doit avoir été calculée par un appel à `factorize()`,
    post-condition : la matrice connectée est remplacée par le facteur Q de la factorisation QR.

Les spécifications d'un service doivent aussi préciser les modes d'utilisations du service : ordre d'appel des fonctionnalités, paramètres optionels, options de fonctionnement... La bonne réalisation d'un service doît appliquer les principes de la programmation par contrat, [112, Chap. 11 *Design by Contract*] qui seront mis en œuvre à l'aide de l'état de l'objet.

L'introduction du motif *Service* demande des modifications de l'architecture séquentielle (et parallèle) (voir figure 3.2) de LAKe. Ces changements sont résumés par la figure 4.5. Nous n'avons pas présenté sur cette figure les classes parallèles afin de ne pas alourdir le schéma. La règle est simple : chaque opération parallèle sera construite en dérivant une version parallèle du service séquentiel spécifiant cette opération. Le service parallèle utilisera lui les versions parallèles des objets concernés. Ainsi, comme il a été précédemment présenté à la figure 4.4, le service parallèle `Distributed_AXPY_Operator` est dérivé du service abstrait `AXPY_Operator`. Le premier utilise donc des objets parallèles de la classe `MPI_DMatrix` dérivant de leur homologue séquentiel.

L'utilisation du motif *Service* résout le problème de contravariance posé par C++ et permet ainsi de faire un pas de plus vers la réutilisabilité séquentielle/parallèle, car les méthodes itératives peuvent désormais utiliser de façon polymorphe des matrices séquentielles ou parallèles.

REMARQUE **4.4 (Contravariance vs covariance)** *Le choix d'autoriser ou non la redéfinition covariante de méthodes dans les langages a été source de débat. Ce « faux » problème a été identifié comme tel, par Castagna [36] comme caractérisant deux mécanismes différents, le* sous-typage *et la* spécialisation. *La conclusion de l'étude est que les deux mécanismes sont nécessaires et complémentaires. De plus, il est également noté que leurs différences apparaissent clairement lors de l'utilisation de méthodes à aiguillage*

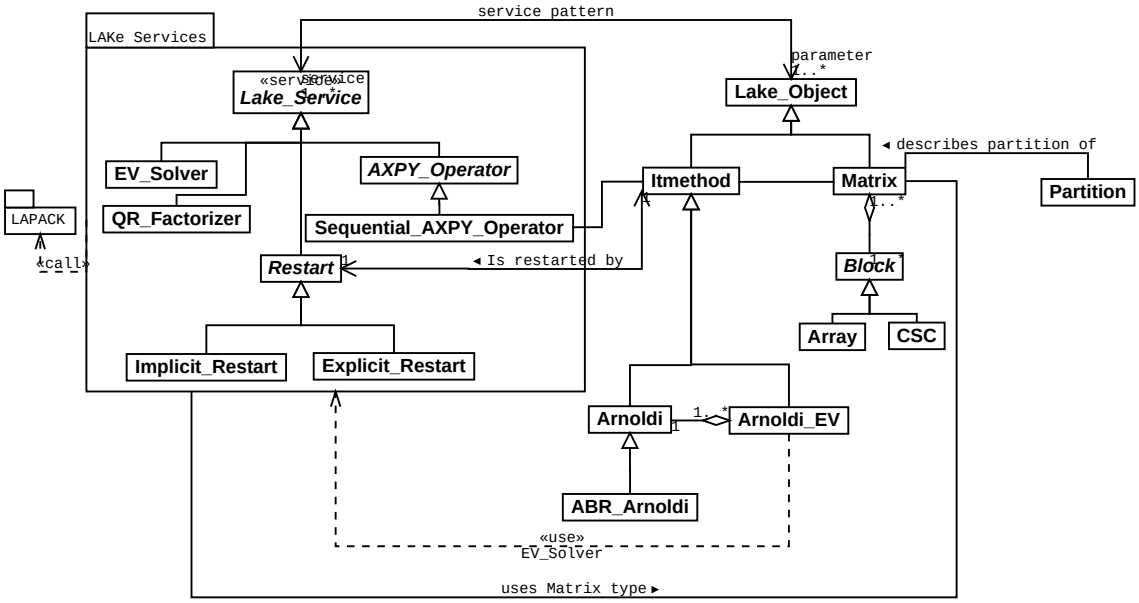

FIG. 4.5 – *Architecture objet de LAKe intégrant la notion de* service

multiple.

Malheureusement, il reste encore un obstacle à la réutilisation complète du code des méthodes itératives séquentielles pour leur implantation parallèle. Les opérations qui posent problèmes sont les créations/destructions de matrices distribuées par les méthodes itératives (voir §2.5.1 page 69 – Allocation mémoire). Nous expliquerons les raisons de ce problème dans le paragraphe suivant, nous donnerons plusieurs solutions à ce problème et approfondirons la solution utilisant la généricité au paragraphe 4.4, en introduisant la notion de forme de matrice qui est une contribution de cette thèse.

4.3.3 Généricité et distribution

Certaines classes (qui peuvent être des services) de LAKe doivent être capable d'allouer des matrices dont les dimensions et la distribution dépendent de paramètres passés au constructeur ou à des méthodes de ces classes. Par exemple la classe `Arnoldi`, dont la spécification partielle est donnée à la figure 4.6, doit allouer la matrice de Hessenberg par bloc H_m et la matrice de la base de Krylov V_m. Ces matrices contruites itérativement par le processus d'Arnoldi par bloc (cf. Algo. 2.3, Chap. 2), sont représentées dans la classe `Arnoldi` par des membres protégés `Arnoldi::Hm_` et `Arndoli::Vm_`. Les dimensions de $\mathsf{H}_m \in \mathbb{R}^{(m+1)s \times ms}$ et $\mathsf{V}_m \in \mathbb{R}^{n \times (m+1)s}$ se déduisent de deux paramètres fournis à l'algorithme `Arnoldi` (Alg. 2.3 page 60) :

1. la taille du sous-espace de Krylov par bloc m,
2. les dimensions du vecteur initial $V_1 \in \mathbb{R}^{n \times s}$.

Dans un contexte séquentiel, ce n'est pas un problème car la classe `Matrix` utilisée par `Arnoldi` doit avoir une méthode permettant de créer n'importe quelle matrice rectangu-

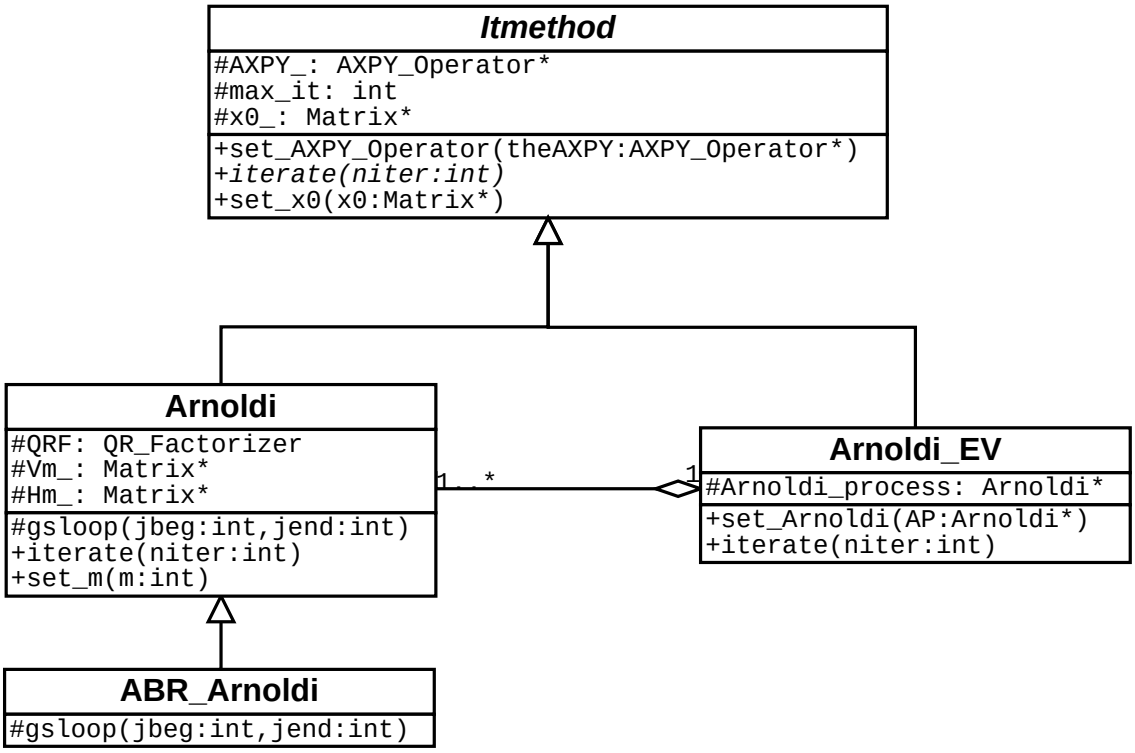

FIG. 4.6 – *Quelques méthodes itératives de LAKe*

laire de taille $m \times n$. Le problème dans une implantation parallèle est que la matrice V_1 est distribuée ce qui implique que V_m doit être distribuée. En revanche, H_m ne l'est généralement pas (cf. §2.6.2). Malheureusement la classe `Arnoldi` n'a aucun moyen d'allouer des matrices distribuées[53], car elle doit ignorer si elle manipule des matrices séquentielles ou parallèles.

Il existe une solution simple à ce problème, faire de tous les objets [potentiellement] distribués des paramètres de la classe `Arnoldi`. Suivant cette optique, on doit également ajouter à la liste des paramètres la variable temporaire W de l'algorithme 2.3, qui doit être distribuée, car elle est impliquée dans des opérations avec des matrices distribuées (A et V_i notamment). Finalement, la création d'un objet instance d'`Arnoldi` nécessite que *toutes* les matrices qu'il utilise soient passées en paramètres. Nous sommes retournés quelques deux décennies en arrière, car notre classe a la même structure qu'un sous-programme Fortran 77, dont les paramètres sont usuellement les variables d'entrée et de sortie du sous-programme, ainsi que les variables représentant des espaces de travail du sous-programme[54].

REMARQUE **4.5 (Classes ou fonctions)** *À ce stade, il pourrait sembler plus judicieux de faire de la classe* `Arnoldi` *une fonction comme celà est fait dans des bibliothèques comme IML++ [56] ou ITL [178, 94]. Nous sommes convaincus que ce n'est pas un bon choix, car nous souhaitons réutiliser le code implantant la méthode d'Arnoldi. En effet,*

53. il faut noter ici qu'une copie simple d'un paramètre distribué tel que V_1 ne suffit pas
54. voir par exemple les interfaces Fortran 77 des procédures disponibles dans LAPACK

si `Arnoldi` *était une fonction, chaque client [classe/fonction] utilisant* `Arnoldi` *serait responsable de l'allocation de* H, V *et* W *avant l'appel à la fonction* `Arnoldi`*. Un client comme* `Arnoldi_EV` *prenant déjà en paramètre* V_1 *et* m *exigerait également qu'on lui spécifie* H, V *et* W *dans l'unique objectif de passer ces paramètre à* `Arnoldi`*. Au bout du compte, chaque méthode itérative serait une fonction, qui nécessiterait chacune leur lot de matrices pré-allouées, y compris des tableaux de travail comme* W*. Les tableaux de travail doivent nécessairement être passés en paramètre car les méthodes itératives ne « savent » pas si ces tableaux sont séquentiels ou parallèles. Cette approche brise l'encapsulation, car les classes/fonctions clientes d'*`Arnoldi` *doivent fournir à* `Arnoldi` *ses données privées et son espace de travail. Cette démarche va également à l'encontre d'une bonne réutilisation car aucun client d'*`Arnoldi` *ne peut utiliser de façon polymorphe une spécialisation [redéfinition] de la fonction* `Arnoldi`*.*

Une autre solution au problème d'allocation est d'utiliser le motif *Service* ou encore le motif *Abstract Factory* [76, pages 87–95] et/ou le motif *Factory Method* [76, pages 107–116] pour concevoir un service `Matrix_Allocator` d'allocation de matrices et de passer celui-ci en paramètre d'`Arnoldi`. Cette solution rendrait le code d'`Arnoldi` obscur, ce qui est justement ce que l'on veut éviter grâce à l'approche objet. La notion qui résout tous nos problèmes techniques est la généricité (voir Déf. 1.21), car elle permet la mise en œuvre d'un polymorphisme paramétrique [31], et c'est d'ailleurs pour cette raison que nous avons inclue la généricité comme une notion caractéristique des approches orientées-objet (voir §1.4).

La solution au problème de l'allocation de matrices distribuées est de paramétrer les classes de matrices avec un type opaque `TShape`, qui renferme les informations concernant la *forme* des matrices. Ce sont les notions de *formes* de matrice et de matrices avec *forme* que nous présenterons dans le paragraphe suivant et qui constituent une contribution de cette thèse.

4.4 Matrices avec forme et formes de matrice

Dans ce paragraphe nous expliquerons les notions de *Matrices avec forme* (§4.4.2) et de *Formes de matrices* (§4.4.1). Nous montrerons, par des exemples, de quelle façon les problèmes de contravariance et d'allocation de matrices distribuées sont résolus grâce à ces notions.

DÉFINITION **4.3 (Matrice avec forme)** *Une matrice avec forme est un type de matrice générique noté* `Matrix<TShape>`*, pour lequel le paramètre générique formel* `TShape` *doît être conforme aux spécifications d'une forme de matrice. Le seul argument nécessaire à la création d'un objet « matrice avec forme » est une forme de matrice de type* `TShape`*. Une matrice avec forme doît être capable de fournir le type de sa forme et un objet de ce type représentant sa forme actuelle.*

Avant d'aller plus avant dans la présentation des matrices avec forme, nous présentons ci-après les spécifications des formes de matrices.

4.4.1 Formes de matrices

Une *Forme de matrice* est un type qui fournit un ensemble minimal d'opérations permettant des calculs structurels sur des matrices, c'est-à-dire que chaque opération, applicable à une matrice, l'est aussi à ses formes [de matrice] associées. Une *Forme de matrice* est en quelque sorte la réification du type de donnée abstrait matrice tel qu'il est spécifié par les opérations élémentaires du paragraphe §2.5.1.

Les opérations sur les *Formes de matrice* se divisent en 3 catégories :

1. Opérations création/destruction
2. Opérations logiques
3. Opérations algébriques

Les fonctions de chacunes de ces catégories sont présentées par les tableaux 4.1, 4.2, 4.3.

Puisque nous avons spécifié ce qu'était une forme de matrice, nous pouvons désormais présenter la conception et l'utilisation des matrices avec formes.

4.4.2 Matrices avec formes

D'après la définition 4.3, nous savons qu'une matrice avec forme possède un paramètre générique formel conforme à une forme de matrice (voir figure 4.7).

Les conditions pour qu'une matrice avec forme soit conforme au type `Matrix<TShape>` sont les suivantes :

1. la seule information nécessaire à la création (allocation) d'une matrice `Matrix<TShape>` est un objet de type `TShape`,
2. une `Matrix<TShape>` doit pouvoir fournir sa forme actuelle, c'est-à-dire un objet de type `TShape`,
3. une `Matrix<TShape>` doit pouvoir fournir le *type* de sa forme, noté `Matrix<TShape>::shape_type`.

Il faut noter que même si l'on note une matrice avec forme `Matrix<TShape>`, cela peut en fait correspondre à n'importe quel type de matrice utilisateur qui est conforme à `Matrix<TShape>`.

L'extrait de programme 4.4, tiré de l'implantation de la classe `Arnoldi`, montre l'utilisation des formes de matrices pour allouer H, V et W. Aux lignes 3–15, on crée la matrice W dont la forme est le produit de la forme l'opérateur matricielle (qui représente la matrice A du système) par la forme de x_0. Aux lignes 17–19, on définit V dont la forme est celle de x_0 étendue $m+1$ fois suivant les colonnes. Finalement, aux lignes 20–27, on crée la matrice de Hessenberg par bloc H dont la forme est celle de $x_0^T x_0$ étendue m fois suivant les colonnes et $m+1$ fois suivant les lignes. Aux lignes 3, 17 et 20, on utilise le type de forme `TMatrix::shape_type` fourni par le type de matrice `TMatrix` utilisé par `Arnoldi`. Les lignes 5–12 montrent l'utilisation des formes de matrices pour vérifier une garde : la

TAB. 4.1 – *Opérations de création/destruction sur une* Forme de Matrice

create()	$: \Box \rightarrow TShape$
Création par défaut. Cette fonction crée une forme de matrice invalide. Après cette création une vérification de validité indiquerait que la forme est invalide. Post-condition : la forme créée est invalide.	
destroy(S)	$: TShape \rightarrow TShape$
Détruit une forme de matrice. Cette fonction détruit la forme et la rend invalide. Post-condition : la forme S est invalide.	
create(m,n)	$: int,int \rightarrow TShape$
Crée la forme par défault d'une matrice $m \times n$.	
create(S)	$: TShape \rightarrow TShape$
Crée une copie de **S**.	
row_shape(S)	$: TShape \rightarrow TShape$
Crée une forme mono-ligne ayant le même nombre de colonnes que **S**. Si **S** représente une matrice $m \times n$ alors la forme résultante représentera une matrice $1 \times n$. Post-conditions : le nombre de lignes de la forme créée est 1 et le nombre de colonnes, le même que celui de S.	
column_shape(S)	$: TShape \rightarrow TShape$
Crée une forme mono-colonne ayant le même nombre de lignes que **S**. Si **S** représente une matrice $m \times n$ alors la forme résulante représentera une matrice $m \times 1$. Post-conditions : le nombre de colonnes de la forme créée est 1 et le nombre de lignes, le même que celui de S.	
sub_shape(S,I1,I2)	$: TShape,Index,Index \rightarrow TShape$
Crée une forme qui est une sous-forme de **S** allant de l'indice **I1** à l'indice **I2**. Le type des indices est non spécifié. Un indice **I** pourra être un couple (i,j) si l'on veut définir une sous-forme contiguë, un triplet (i,j,s) si l'on veut définir une sous-forme non contiguë ou n'importe quel autre type d'indice défini par l'utilisateur (par exemple une indexation par bloc).	
expand_shape(S,m,n)	$: TShape,int,int \rightarrow TShape$
Crée une forme qui est identique à **S**, mais avec m fois plus de lignes et n fois plus de colonnes. Si **S** représente une matrice $p \times q$ alors la forme résultante représentera une matrice $m \cdot p \times n \cdot q$.	

TAB. 4.2 – *Opérations logiques sur une* Forme de Matrice

!S	: $TShape \rightarrow boolean$
Test d'invalidité. La fonction teste si la forme est valide et renvoie VRAI si la forme est invalide. Une forme peut être invalide, si une précédente opération a produit une forme invalide.	
S1==S2	: $TShape, TShape \rightarrow boolean$
Test d'égalité stricte. La fonction renvoie VRAI si **S1** est strictement égale à **S2**.	
S1_is_assignable_to_S2(S1,S2)	: $TShape, TShape \rightarrow boolean$
La fonction vérifie si l'on peut affecter une forme **S1** à une forme **S2** et renvoie VRAI si c'est possible, FAUX sinon. *Il est possible que cette fonction soit la même que l'égalité stricte mais ce n'est pas obligatoire.* En effet dans le cas de formes de matrices parallèles on peut parfois affecter **S1** à **S2** même si **S1** n'est pas strictement égale à **S2**, par exemple si **S1** est répliquée et **S2** ne l'est pas.	

TAB. 4.3 – *Opérations algébriques sur une* Forme de Matrice

nrow(S)	: $TShape \rightarrow int$
Nombre de lignes de la forme. Cela représente le nombre de lignes de la matrice.	
ncolumn(S)	: $TShape \rightarrow int$
Nombre de colonnes de la forme. Cela représente le nombre de colonnes de la matrice.	
transpose(S)	: $TShape \rightarrow TShape$
Transposition matricielle. Cette fonction renvoie une forme de matrice, correspondant à la forme de la transposée de la matrice représentée par la forme **S**. Le résultat peut être une forme invalide si le type de forme de **S** ne peut être transposé.	
S1+S2	: $TShape, TShape \rightarrow TShape$
Addition matricielle. Cette fonction renvoie la forme correspondant à l'addition de deux matrices dont les formes sont **S1** et **S2**. Le résultat peut être une forme invalide si l'addition des deux matrices est impossible.	
S1*S2	: $TShape, TShape \rightarrow TShape$
Produit matriciel. Cette fonction renvoie la forme correspondant au produit des matrices dont les formes sont **S1** et **S2**. Le résultat peut être une forme invalide si le produit des deux matrices est impossible.	

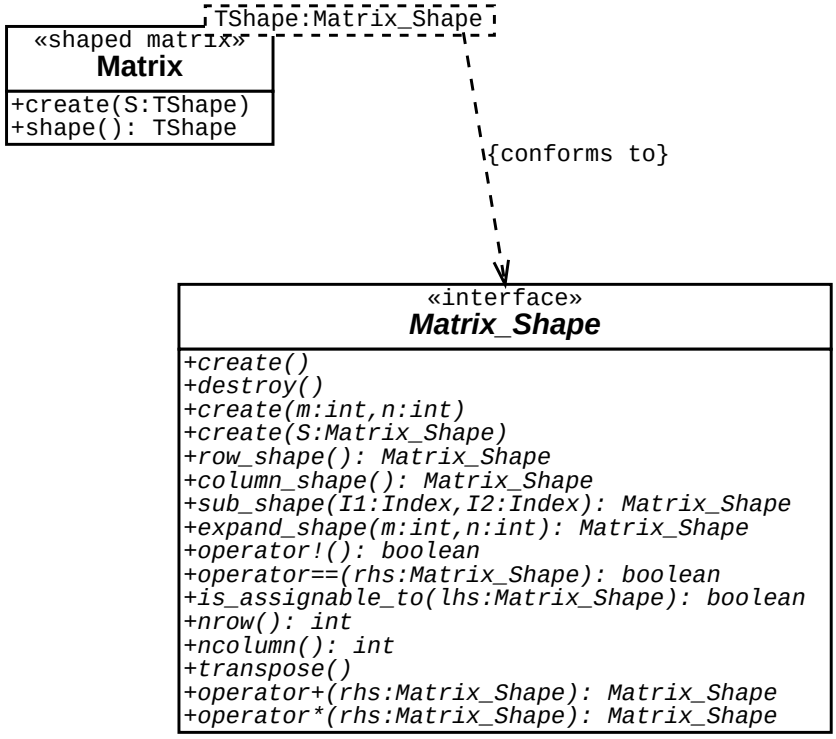

FIG. 4.7 – *Spécification [partielle] d'une matrice avec forme*

validité du produit est vérifiée sur les formes de matrices avant que le produit effectif ne soit calculé.

Programme 4.4: *Utilisation des formes de matrices pour allouer des matrices*

```
1  // TMatrix is the type of Shaped Matrix used by Arnoldi
2  [...]
3  TMatrix::shape_type     SW;
4  SW = SMatrix_Operator.shape()*x0.shape();
5  #ifdef _LAKE_CHECK
6    if (!SW)
7      {
8        error("Invalid Sparse Matrix Vector Multiply");
9        // print invalid_info string of SW
10       error(SW.invalid_info());
11     }
12 #endif
13 // allocate W whose shape is the product of
14 // SMatmul and x0 shapes.
15 W.create(SW);
16
17 TMatrix::shape_type    Sx0;
18 Sx0 = x0.shape();
```

```
19 bV.create(Sx0.expand_shape(1,max_it()+1));
20 TMatrix::shape_type    Sx0t;
21 Sx0t = Sx0;
22 Sx0t.transpose();
23 // allocate bH whose shape is the shape of
24 // x0^{T} times x0 expanded max_it()+1 times
25 // along the rows and max_it() times along
26 // the columns.
27 bH.create((Sx0t*Sx0).expand_shape(max_it()+1,max_it()));
```

On pourrait se demander pourquoi ne pas avoir doté les matrices elles-mêmes des fonctionnalités offertes par les formes de matrice. Si nous avions donné à l'opérateur `Matrix& Matrix::operator*(Matrix&)` la sémantique de la multiplication matricielle, nous aurions eu à gérer les objets temporaires générés par cet opérateur, ce qui peut être relativement compliqué (au moins en C++ [203, 199]) et qui ne fait pas partie de nos objectifs. Si nous avions adopté cette optique, nous aurions dû définir des opérateurs algébriques n'ayant pas le sens usuel. Comme les formes de matrices sont de petits objets nous pouvons accepter les temporaires générés par les opérateurs, de plus une optimisation spécifique les concernant peut être rajoutée par la suite [203, 155].

L'exemple de programme 4.4 montre comment un client d'une classe de matrice avec forme peut allouer des matrices dont la forme peut être calculée à partir d'autres matrices passées en argument à ce client. Dans notre exemple, le client est la classe `Arnoldi` à laquelle on a fourni la matrice x_0 et la taille m du sous-espace de Krylov par bloc (m est la valeur renvoyée par `Arnoldi::max_it()`). À présent, l'allocation des matrices [éventuellement] distribuées dans `Arnoldi` est réalisée grâce aux faits que :

1. `x0` est une matrice distribuée,
2. les règles algébriques de calcul sur les formes de matrices dictent comment créer des formes de matrices [éventuellement] distribuées à partir de celle de `x0`.

Le mécanisme de forme de matrice ne dispense pas l'utilisateur de distribuer lui-même initialement les matrices du programme principal (comme `x0` dans notre cas). Pour créer une matrice distribuée `x0` dans le programme principal l'utilisateur pourrait avoir écrit quelque chose de semblable à l'extrait de programme 4.5.

Programme 4.5: *Distribution de matrices parallèles avec formes*

```
1  int
2  main(int argc, char* argv[])
3  {
4  // initialize LAKe
5  LAKELL.Initialize(argc,argv);
6  // retrieve the number of processors
7  const int n_processors = Lakell::nprocessors();
8  [...]
9  // declare a distributed matrix whose shape is a Distribution
10 Matrix<Distribution>   x0;
11 Distribution          Dx0; // a Distribution object
```

```
12 int n_rows     = 10000;
13 int n_columns  = 5;
14
15 // create a row cyclic distribution for a
16 // matrix with n_rows and n_columns
17 Dx0.create_row_cyclic(n_rows,n_columns,n_processors);
18 // allocate the distributed matrix x0
19 x0.create(Dx0);
20
21 [...]
22 // pass x0 as parameter to Arnoldi ...
23 [...]
24 }
```

La méthode utilisée pour créer des formes de matrices distribuées ne fait *pas* partie des spécifications standard des formes de matrices. Chaque concepteur de forme de matrice parallèle fournira à l'utilisateur ses propres fonctions de création de formes parallèles en plus des opérations standard (voir tableaux 4.1,4.2,4.3). De cette façon, seul le programme principal devra être changé pour passer d'une classe de matrice parallèle avec forme à une autre. Les concepteurs de forme de matrice ne sont donc pas obligés d'implanter toutes les distributions possibles mais seulement celles qui sont utiles à leurs applications. La seule contrainte pour une classe de forme de matrice est d'implanter les opérations standard sur les formes.

4.4.3 La généricité résout le problème de contravariance

Nous venons de voir comment le mécanisme de matrices avec forme résout le problème de l'allocation distribuée de matrices. Il n'est peut être pas évident que la généricité soit réellement nécessaire pour mettre en œuvre ce mécanisme. Nous aurions pu définir une classe matrice avec forme, `Shaped_Matrix` qui utilise de façon polymorphe la classe abstraite `Matrix_Shape`, dont on aurait dérivé deux classes concrètes implantant des formes de matrices séquentielle et parallèle. Malheureusement, le problème de la contravariance serait encore présent dans la classe `Arnoldi` comme nous l'avons montré dans l'extrait de programme 4.1 du paragraphe 4.3.1.

Ce problème de contravariance est également résolu par l'utilisation de la généricité. En effet, il suffit de faire de `Matrix` un paramètre générique de la classe `Arnoldi`, qui devient `Arnoldi<TMatrix>` pour laquelle `TMatrix` est une matrice avec forme. À présent, lorsque l'on compile la classe `Arnoldi<TMatrix>` avec pour paramètre générique `TMatrix`, une matrice avec forme séquentielle ou parallèle, l'instanciation générique permet d'appeler la « bonne » méthode `TMatrix::axpy(float,TMatrix*,TMatrix*,float)`. Le problème de contravariance est résolu par l'utilisation de la généricité pour toutes les classes dérivant de `Itmethod<TMatrix>`. La résolution du problème de contravariance par la généricité est un résultat théorique connu issu des travaux de Castagna [36], et sont à prendre ici comme leur mise en œuvre pratique en C++.

Nous avons fait des classes de matrices avec forme des classes génériques du type

`Matrix<TShape>`, car nous souhaitions réutiliser le code de cette classe, afin qu'une matrice séquentielle `Sequential_Matrix` soit en fait `Matrix<Sequential_Shape>` et que qu'une matrice parallèle soit en fait `Matrix<Parallel_Shape>`. C'est ce qui est réalisé dans la dernière version de LAKe que nous appellerons Gene-LAKe. On doit toutefois noter que, dans LAKe, la classe `Parallel_Matrix` n'est pas strictement égale à `Matrix<Parallel_Shape>` mais dérive de cette dernière. C'est un point important, car nous devons connaître le type exact de la forme de matrice parallèle, afin d'implanter `Matrix<TShape>::create(TShape S)` et toutes les autres opérations élémentaires parallèles (cf §2.5.1). Donc, en réalité, `Parallel_Matrix` redéfinit toutes les méthodes héritées de `Matrix<Parallel_Shape>` qui nécessitent l'accès à l'interface de `Parallel_Shape` ne faisant pas partie de `Matrix_Shape`.

REMARQUE **4.6 (réutilisabilité séquentielle/parallèle ?)** *On peut douter du fait que notre objectif de réutilisabilité séquentielle/parallèle ait réellement été atteint, vu qu'il est nécessaire de redéfinir des méthodes de* ***`Matrix<Parallel_Shape>`****. En réalité, il n'en est rien, notre objectif de réutilisation séquentielle/parallèle se situant au niveau des méthodes itératives. En revanche, nous doutons du fait qu'il soit même possible de réutiliser le code séquentiel de bon nombre d'opérations algébrique matricielles,* tout en ayant des performances parallèles satisfaisantes. *Si l'on prend le cas du produit matrice/matrice, il suffit de consulter la littérature, pour se convaincre qu'il est absolument nécessaire de ré-écrire une implantation parallèle pour obtenir de bonnes performances parallèles. On peut également noter que l'écriture d'algorithmes spécialisés est déjà nécessaire pour des matrices séquentielles de formes particulières (creuse, triangulaire, bande, ...) et c'est ce qui est fait dans MTL [121].*

4.4.4 Les avantages des matrices avec formes

Nous devons signaler un certain nombre d'avantages des matrices avec forme, qui font que les classes clientes [utilisant] des matrices avec forme sont réellement indépendantes de leur implantation.

- Un client d'une matrice avec forme peut allouer des matrices temporaires sans savoir comment elles sont allouées, y compris si elles sont distribuées. Le modèle de stockage mémoire est donc masqué,
- Les opérations sur les formes de matrices fixent les règles pour distribuer le résultat d'une opération distribuée avec des matrices.

  Par exemple, le résultat du produit d'une matrice distribuée par colonne par une matrice distribuée par ligne doit être une matrice répliquée. Ces règles peuvent être changées dans la classe de forme de matrice elle-même et influencer la distribution de toutes les matrices, **y compris les variables temporaires** utilisées par les méthodes itératives. Si les régles de distribution changent, il n'est même pas nécessaire de changer ne serait-ce qu'une ligne dans le code des méthodes itératives. Les règles régissant les opérations algébriques sur les formes de matrices [éventuellement distribuées] et les matrices associées, sont sous la responsabilité du concepteur de ces classes. Ainsi, deux classes de matrices avec forme distribuées pourront produire des distributions différentes pour les mêmes matrices, sans que l'utilisateur des matrices

ne s'en soucie. Il est à prévoir que les règles de distributions des matrices guident les performances de l'applications. Ainsi, spécifier que l'on veut créer une matrice W, dont la forme est le produit de A par V, indique que W doit être distribuée efficacement pour être le résultat du produit $A \cdot V$,

– Les formes de matrices fournissent un cadre unificateur pour la vérification des propriétés structurelles sur les opérations matricielles.
Une méthode comme `Matrix<TShape>::axpy(...)` vérifie habituellement que ses arguments matriciels sont conformants avant d'effectuer le produit. Ces méthodes utilisent désormais les opérateurs logiques sur les formes de matrices des arguments. Avant de faire $Y = A \cdot X$ on doit avoir :
`(A.shape()*X.shape()).is_assignable_to(Y.shape())`.
Avant d'additionner deux matrices [potentiellement] distribuées A et B on peut vérifier que `!(A.shape()+B.shape())`. Si A et B ne sont pas distribuées de façon compatible[55], la somme des formes de A et B sera invalide.

4.4.5 Polymorphisme universel paramétrique ou par inclusion

En C++, la généricité peut être vue comme du polymorphisme *à la compilation*. L'héritage et la liaison dynamique serait donc le support du polymorphisme à l'exécution et la conformance et la généricité le support du polymorphisme à la compilation. C'est le point de vue de la réalisation qui en est faite dans le langage **C++**. En réalité, si l'on replace ces notions dans la classification de *Cardelli et Wegner* [31], le polymorphisme à l'exécution est une réalisation du *polymorphisme universel par inclusion* et la généricité une réalisation du *polymorphisme universel paramétrique*.

L'équivalence entre la généricité et l'héritage [plus la liaison dynamique] a déjà été étudiée [112, 177]. La conclusion de ces études est que l'héritage est un mécanisme général qui permet d'émuler la généricité[56], mais l'inverse n'est pas vrai. La généricité ne permet pas d'émuler l'héritage, mais elle peut, en revanche, être un très bon outil pour gérer les classes référençant leur propre type[177][57].

Nous pensons que ces deux types de polymorphismes sont utiles et complémentaires. Nous proposons un moyen simple pour choisir lequel devra être mis en œuvre. C'est la notion d'association *statique* ou *dynamique*.

DÉFINITION **4.4 (Association dynamique/statique)** *Une classe `A` est un client d'une classe `B`, si `A` utilise au moins un objet de classe `B`. `A` et `B` sont donc reliés par une association au sens UML [122, pages 41,123]. L'association est dîte* dynamique, *si un objet de classe `A` peut potentiellement utiliser, lors de l'exécution, des objets de classes différentes [dérivant toutes de `B`]. L'association est dite* statique *si la classe des objets utilisés par les objets de classe `A` ne change pas durant l'exécution.*

55. ce qui est défini par l'implantation
56. jusqu'à présent, c'est de cette façon que Java supporte les collections *génériques*, ce qui est susceptible de changer [79]
57. en anglais : self-referencing

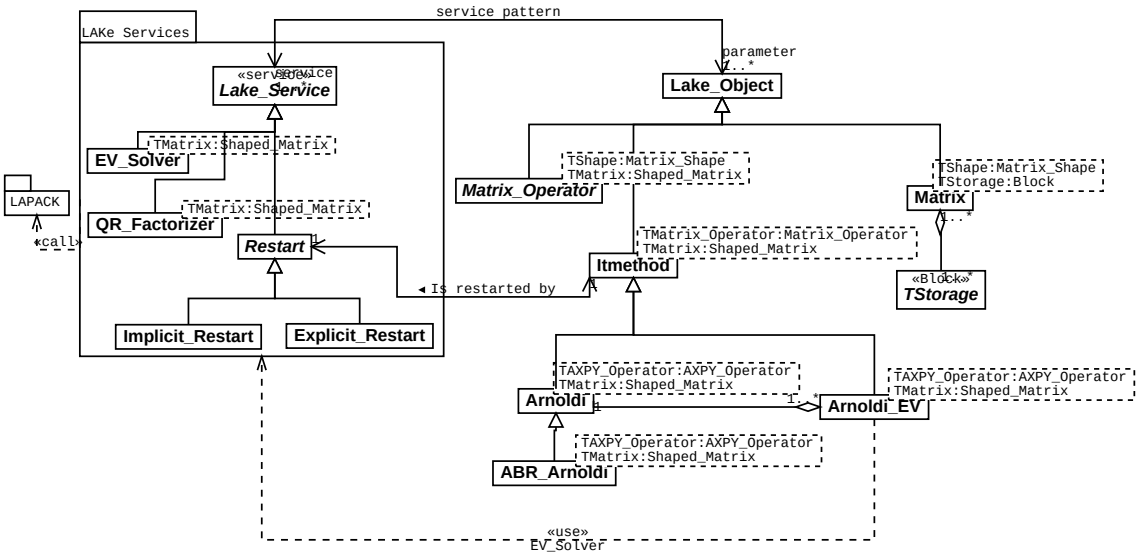

FIG. 4.8 – *Architecture objet de Gene-LAKe*

À partir de cette définition, on peut définir la règle du choix de la généricité :

REMARQUE **4.7 (Règle du choix de la généricité)** *Si une classe `A` est liée par une association statique à une classe `B`, on peut faire de `B` un paramètre générique de `A`. Si `A` est liée par une association dynamique, on doit utiliser le polymorphisme et la liaison dynamique dans la réalisation.*

D'un point de vue implantation, l'architecture du motif de *Service* (§4.3.2) est faite pour aider à l'implantation des associations dynamiques, dans lesquelles `B` est un service pour `A`. Il est bon de noter que l'apparente dichotomie entre `Lake_Object` et `Lake_Service` est artificielle, car une classe peut tout à fait être à la fois `Service` et `Objet`, tout dépend de son rôle dans l'association.

En ce qui concerne la généricité, l'implantation est immédiate si le langage cible la supporte directement[58]. Pour un langage cible ne supportant pas la généricité, un pré-processeur ou un compilateur source à source devrait permettre de la mettre en œuvre.

Appliquer la règle du choix générique à l'architecture de LAKe de la figure 4.5, conduit aux classes génériques suivantes :

- `Matrix<TShape,TStorage>`
- `QR_Factorizer<TMatrix>`
- `EV_Solver<TMatrix>`
- `Itmethod<TMatrix_Operator,TMatrix>`
- `Arnoldi<TMatrix_Operator,TMatrix>`
- `ABR_Arnoldi<TMatrix_Operator,TMatrix>`
- `Arnoldi_EV<TMatrix_Operator,TMatrix>`
- `TMatrix_Operator<TShape,TMatrix>`

Ces changements sont présentés à la figure 4.8.

58. ce qui est le cas avec C++

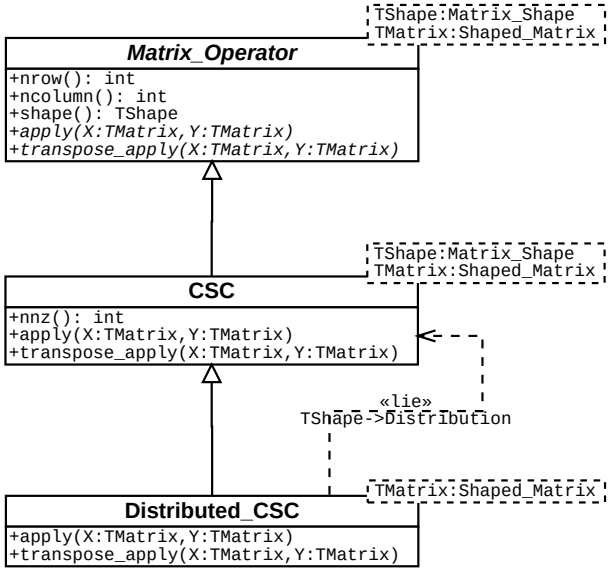

FIG. 4.9 – *Les opérateurs matriciels de Gene-LAKe*

4.5 Opérations basiques et algorithmes

Il faut noter, que l'introduction de la généricité a fait disparaitre le service de multiplication matricielle `AXPY_Operator`, au profit d'une nouvelle classe abstraite nommée `Matrix_Operator`. Cette classe représente un opérateur matriciel capable d'effectuer des opérations $Y = A \cdot X$, où A est l'opérateur matriciel en question. À la différence du service `AXPY_Operator`, l'opérateur matriciel possède comme argument implicite la matrice A, qui est représentée par l'objet lui-même. `Matrix_Operator` n'est pas un service permettant de solutionner le problème de contravariance, mais une classe issue de la conception du domaine des méthodes itératives de l'algèbre linéaire. Cette classe correspond, donc, à la réalisation d'opérateurs matriciels [creux] tels qu'ils sont utilisés dans la résolution de problèmes d'algèbre linéaire par des méthodes itératives. Dans Gene-LAKe nous avons dérivé deux classes concrètes de `Matrix_Operator` présentées à la figure 4.9. L'opérateur matriciel possède une forme de matrice, car il représente une matrice, mais il n'a *a priori* pas les fonctionnalités d'indexation que pourrait avoir des objets matrices.

Méthodes membres ou fonctions génériques ?

L'introduction de la généricité, même si elle résout le problème de la contravariance, amène toujours à se demander si une méthode d'une classe comme,

```
Matrix<TShape>::gaxpy(float&,TMatrix<TShape>&,TMatrix<TShape>&,float&)
```

ne serait pas plus judicieusement implantée comme une fonction générique hors classe,

```
template <class TOperator, class TMatrix, class TElem>
gaxpy(TElem& alpha, TOperator& A, TMatrix& X, TMatrix& Y, TElem& beta)
```

Les principes orientés-objet semblent indiquer, qu'un objet d'une classe ne doit toujours être modifié que par les méthodes de la classe à laquelle il appartient. Cela permet, entre autre, de préserver l'intégrité des objets et d'augmenter l'encapsulation en laissant entrevoir une meilleure réutilisabilité. En réalité, nous pensons que pour des fonctions purement algorithmiques comme les opérations algébriques sur les matrices, le choix de méthodes membres, plutôt que de fonctions génériques hors classes, est loin d'être tranché pour les raisons suivantes :

- on peut écrire une seule fonction générique capable de s'appliquer à plusieurs types de matrice, offrant des interfaces conformantes (par exemple implanter l'opération $Y = A * X$ à partir de classes de matrices offrant un opérateur d'indexation). On écrit donc le code générique qu'une seule fois, au lieu de l'écrire dans chaque classe,
- les méthodes binaires (ou plus) posent le problème de la contravariance qui ne se pose pas dans le cas de fonctions génériques hors classe,
- les méthodes binaires (ou plus) posent des problèmes de performances, lorsqu'elles sont réalisées par des appels successifs à des opérateurs binaires, qui génèrent une variable temporaire par appel. En C++, la technique des « Expression template » [199] permet de solutionner ce problème, mais cette technique n'est pas simple à mettre en oeuvre [203].

L'exemple de la STL C++ [186, 123, 189] illustre tout à fait notre propos, car la grande majorité des algorithmes sont implantés comme des fonctions génériques agissant sur des structures de données, toutes capables de fournir diverses sortes d'itérateurs sur leur contenu. Ce sont ces itérateurs qui font le lien entre les structures de données et les algorithmes. La librairie MTL [121] utilise le même principe. Dans Gene-LAKe, nous avons donc également fait le choix d'implanter les opérations algébriques matricielles sous la forme de fonctions génériques, mais sans que les matrices soient tenues de fournir des itérateurs généralisés à la manière des extensions parallèles de la STL [97, 161]. En effet, le mécanisme de surcharge [125] de fonctions `template` C++ permet de redéfinir une fonction générique. Ce mécanisme permet notamment d'écrire une fonction générique de base, qui sera ensuite spécialisée pour certaines valeurs des types des paramètres génériques. Nous utilisons la techniques des « traits class » [124] et les formes de matrices, afin de spécialiser les différentes fonctions génériques pour nos différents types de matrices (essentiellement les matrices séquentielles et les matrices parallèles). On peut remarquer, que le principe de la surcharge de fonctions génériques réalise une mécanisme similaire à celui la redéfinition d'une méthode, dans une classe héritière d'une classe de base.

4.6 Expérimentations numériques

Nous avons implanté la librairie Gene-LAKe en C++ et utilisé MPI au travers de OOMPI pour les classes parallèles. Nous avons utilisé la méthode d'Arnoldi par bloc, afin de trouver les 10 valeurs propres de plus grand module des matrices concernées. Les paramètres de l'algorithme 2.4 sont les suivant : $r = 10$, $s = 4$, $m = 15$. Les itérations sont stoppées dès que le résidu associé aux paires de Ritz est inférieur à 10^{-6}. La première ma-

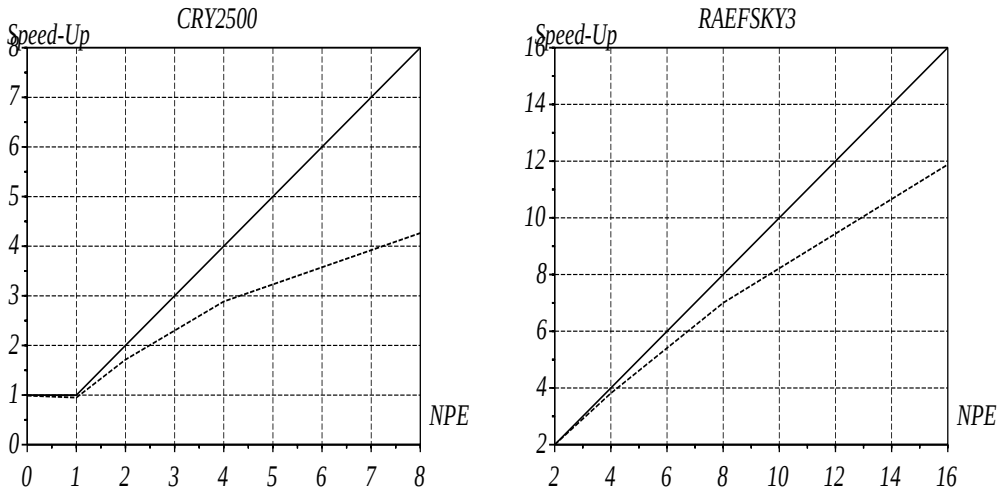

FIG. 4.10 – *Accélération*

trice (CRY2500) est issue du Matrix Market[59] (Sous-ensemble CRYSTAL de la collection NEP). Elle a 2500 lignes et 12349 éléments non-nuls. La seconde matrice (RAEFSKY3) a 21200 lignes et 1488768 éléments non-nuls. Les expérimentations ont été exécutées sur le CRAY T3E de l'IDRIS[60].

Les courbes d'accélération sont présentées à la figure 4.10. La courbe en trait plein correspond à l'accélération théorique et la courbe pointillée à l'accélération mesurée. L'accélération correspondant à la matrice RAEFSKY3 est calculée par rapport au temps d'exécution sur deux processeurs car le code ne pouvait s'exécuter sur un seul, vu la taille du problème. Concernant la matrice CRY2500 le nombre de processeurs $NPE = 0$ correspond au temps d'exécution du code séquentiel et $NPE \geq 1$ au temps d'exécution du code parallèle sur le nombre de processeurs indiqué. Les accélérations sont satisfaisantes tant que le nombre de processeurs est raisonnablement petit par rapport à la taille du problème. Il faut noter que le code séquentiel utilisé pour la matrice CRY2500 et le code parallèle utilisé pour CRY2500 et RAEFSKY3 sont dérivés du *même* code générique. Le choix se fait uniquement au moment de l'instanciation générique du code [de méthodes itératives], qui utilise soit des classes de matrices séquentielles, soit des classes de matrices parallèles. Cela signifie, entre autre, que pour un problème qui tient en mémoire d'une station de travail, il n'est pas nécessaire d'exécuter la version parallèle du code sur un processeur : on instancie la version séquentielle. Une comparaison grossière, avec un code Fortran 77 implantant la méthode de façon non générique, montre que le code Fortran est à peu près 2 fois plus rapide que le code C++ générique. Cette comparaison n'est pas équitable, car la version Fortran 77 est loin d'avoir toutes les fonctionnalités de la version C++.

59. `http://gams.cam.nist.gov/MatrixMarket/`
60. Institut du Développement et des Ressources en Informatique Scientifique, CNRS, Orsay, France

4.6.1 Travaux connexes

REMARQUE **4.8 (Autres librairies génériques)** *IML++ (Iterative Method Library) [56] et MTL/ITL (Matrix Template Library/Iterative Template Library) [178] fournissent toutes deux des méthodes itératives génériques. LAKe résout des problèmes qui ne sont pas abordés dans ces librairies :*

1. *Elles n'ont pas été utilisées avec des classes de matrices distribuées. Une extension parallèle de MTL appelée PMTL est en cours de dévelopement, mais à la date d'aujourd'hui aucun résultat n'est disponible. De plus, il n'est pas précisé que les composants d'ITL pourront utiliser PMTL sans modification,*
2. *Les méthodes itératives sont implantées comme des fonctions génériques et non par des classes. Cela signifie, que la réutilisation polymorphe d'algorithmes, comme un processus d'Arnoldi, n'est pas un des objectifs de ces librairies,*
3. *les fonctions qui implantent les méthodes itératives ne peuvent pas traiter l'allocation d'une variable distribuée de façon opaque.*

Nos travaux, même s'ils sont guidés par une implantation dans un langage supportant directement les concepts orientés-objet, ne sont pas sans liens avec Fortran. Il est tout d'abord possible d'envisager une démarche orientée-objet en Fortran [52]. Nous avons également déjà signalé la similitude des objectifs entre le motif de *Service* et le motif de *Reverse Communication* à la Fortran 77 [55]. On peut également voir une similitude entre les directives HPF [90], comme `DISTRIBUTE`, `TEMPLATE` ou `ALIGN`, qui indiquent au compilateur la manière de distribuer les données. Les formes de matrice remplissent un rôle similaire lors de l'exécution. On pourrait alors imaginer un compilateur paralléliseur qui sache exploiter les informations fournies par les formes de matrices, au moment de la compilation.

4.7 Conclusion

Gene-LAKe a atteint les objectifs que nous nous étions fixé en terme de réutilisabilitté séquentielle/parallèle, car le même code source d'une méthode itérative permet d'instancier, à la compilation, une version séquentielle ou parallèle. De cette façon, la hiérarchie de classes des méthodes itératives est strictement la même pour les versions séquentielles et parallèles. Nous avons présenté les limites du polymorphisme et de la liaison dynamique pour nos problèmes et avons ainsi argumenté en faveur de l'utilisation de la généricité. Nous avons présenté, à cette occasion, un nouveau motif de conception permettant le polymorphisme multiple. Un nouveau concept de *Matrice avec forme* a été présenté, permettant l'implantation générique efficace de méthodes itératives séquentielles ou parallèles. Les expérimentations numériques sur machines parallèle montrent le bien-fondé de notre démarche.

Chapitre 5

Intégration du parallélisme dans une méthodologie de développement objet

Sommaire

5.1 Introduction

La seule utilisation d'un langage de programmation orienté-objet ne suffit pas à satisfaire les objectifs de réutilisabilité et de maintenabilité que nous nous étions fixés et les deux études, que nous venons de présenter aux chapitres 3 et 4, le prouvent. Que ce soit pour la programmation séquentielle ou parallèle, l'utilisation d'une méthode de conception elle-même orientée-objet est nécessaire. Un grand nombre de méthodes de conceptions orientées-objet sont à notre disposition [37, OOAD Methods], malheureusement celles-ci n'intègrent pas dès le départ la conception d'applications parallèles. De plus, la grande majorité de ces méthodes privilégient une conception *avec réutilisation*, plutôt qu'une

conception *ayant pour objectif de réutiliser*. Nous expliciterons cette distinction au paragraphe 5.2. En partant d'une méthodologie orientée-objet, dans laquelle la conception est réalisée avec comme objectif de réutiliser [47, Chap. 9], nous proposons une extension de la méthode intégrant les contraintes liées aux aspects du parallélisme. Cette démarche orientée-objet, qui est une des contributions de cette thèse, met en évidence les problèmes de réutilisation et de maintenance des codes parallèles et propose des solutions adaptées.

Nous explicitons tout d'abord au §5.2 la différence entre une conception *avec réutilisation* et une conception *visant la réutilisabilité*, puis, nous rappelons les principales étapes de la méthode DEMRAL de Czarnecki [48] au §5.3 qui met en œuvre ces principes dans un cadre séquentiel. Nous proposons, au paragraphe 5.4, l'intégration des spécificités du parallélisme à cette méthode. Nous nous basons, pour ce faire, sur l'expérience des problèmes de réutilisation posés par les moyens et modèles de programmation parallèle orientés-objet que nous avons étudiés précédemment aux paragraphes 3.5 et 4.3.

5.2 Concevoir *avec réutilisation* ou *pour réutiliser*

Les méthodes de conceptions orientées-objet ont longtemps considéré que la réutilisabilité était intrinsèque à l'approche objet [47, Chap. 4]. Les objets et les classes étaient donc réutilisables « par nature », notamment par le biais des mécanismes orientés-objet tels que l'héritage, la liaison dynamique et le polymorphisme. La conséquence de cette idée est, que les méthodes de conception orientées-objet permettent uniquement de concevoir des applications *avec réutilisation* des composants objets au sein d'une application, c'est-à-dire que les composants ne sont réutilisables qu'au sein du système pour lequel ils ont été conçus. Nous avons vu, par exemple, au §4.3 qu'une classe `Matrix` conçue pour une application séquentielle est très peu réutilisable pour l'implantation parallèle d'une classe matrice distribuée. C'est surtout le cas lorsque l'on souhaite utiliser ces deux classes de façon polymorphe. Les mécanismes d'héritage et de liaison dynamique ne suffisent pas à obtenir la réutilisation séquentielle/parallèle souhaitée. Le principe d'encapsulation ou de l'ouverture/fermée [112, Open-Closed principle – p. 57], dans lequel un composant est ouvert à des extensions, mais suffisament spécifié pour que son interface soit fermée, ne permet pas la réutilisation séquentielle/parallèle. Cette réutilisation est difficile car elle nécessite :

1. que l'implantation parallèle soit prévue dès la conception de la classe séquentielle, ce que ne recommandent pas les méthodes de conception orientées-objet classiques,
2. que l'implantation soit suffisamment « ouverte » pour que les modifications nécessaires à l'implantation parallèle soit aisément réalisable.

Le premier point est un problème de méthode de conception. Les méthodes d'ingénierie de domaine [47, Chap. 3] [42, 43] [176, 136], ayant pour objectif de concevoir des composants pour un ensemble de systèmes, permettent de prévoir des points de variations qui résolvent le premier écueil. Ces méthodes préconisent des conceptions *ayant pour but de réutiliser* des composants d'un domaine d'application et dépassent donc le cadre de la réutilisation intra-application des approches uniquement orientées-objet.

Le second point est un problème d'implantation. La notion d'implantation ouverte [137] et son successeur, la programmation par aspect [7], offrent des techniques d'implan-

tation de ces points de variations. Les principes de la programmation par aspects sont rappelés au §5.5. On peut les considérer comme des extensions naturelles de la programmation orientée-objet.

Nous rappelons, dans le paragraphe suivant, les étapes de la méthode d'ingénierie de domaine DEMRAL, que nous complèterons dans le paragraphe 5.4, afin d'obtenir une méthode d'ingénierie de domaine intégrant le parallélisme .

5.3 Les étapes de la méthode DEMRAL

La méthode DEMRAL (**D**omain **E**ngineering **M**ethod for **R**eusable **A**lgorithmic **L**ibrairies) [47, Chap. 9] [48] est une méthode d'ingénierie de domaine, spécialisée pour les librairies algorithmiques. Elle est particulièrement intéressante, car elle vise à une conception ayant pour objectif de réutiliser. De plus, elle a été appliquée à une librairie matricielle [47, GMCL – Chap. 10] proche de notre domaine d'étude. Nous rappelons, ci-après, brièvement les différentes étapes de DEMRAL, en insistant sur les étapes spécifiques à une méthode ayant pour objectif de réutiliser. La description complète de la méthode, ainsi qu'une étude de cas illustrant son application, sont présentées dans [47, Chap. 9 et 10] ou [48, Chap. 5 et 14]. Une vue d'ensemble des étapes de la méthode est donnée à la figure 5.1. Avant d'en détailler les différentes étapes, il faut noter que, bien que présentées séquentiellement, des phases apparaissant en premier peuvent être recommencées ou plutôt complétées, après qu'une phase ultérieure ait été abordée.

5.3.1 Analyse de domaine

Cette activité doit permettre de borner clairement le domaine d'application. De cette façon, tous les composants développés auront une réutilisabilité maximale au sein du domaine.

Définition du domaine

La définition du domaine est le recensement objectif de ce qui constitue le domaine. Les objectifs du domaine et les participants sont identifiés, des exemples de systèmes et d'application du domaine sont examinés. À partir de là, les caractéristiques du domaine sont identifiées et les relations avec d'autres domaines sont exhibées.

Modélisation du domaine

La modélisation du domaine est une phase des plus importantes pour atteindre une bonne réutilisabilité, car les concepts clefs du domaine y sont identifiés et modélisés.

Les concepts clefs de la méthodes DEMRAL sont principalement les **T**ypes **A**bstraits de **D**onnées (TAD) [74] et les algorithmes. Dans notre cas, les TAD seront les objets mathématiques de l'algèbre linéaire : matrices (creuses, pleines, symétriques, par bloc...) et vecteurs. Les familles d'algorithmes sont le sous-ensemble des algorithmes d'algèbre linéaire qui nous intéresse : les méthodes itératives.

1. Analyse de domaine
   - 1.1. Définition du domaine
     - 1.1.1. Analyse des objectifs et des participants
     - 1.1.2. Choix de l'étendue du domaine et analyse du contexte
       - 1.1.2.1. Analyse de l'étendue applicative du domaine et des systèmes existants
       - 1.1.2.2. Identification des caractéristiques du domaine
       - 1.1.2.3. Identification des relations avec d'autres domaines
   - 1.2. Modélisation du domaine
     - 1.2.1. Identification des concepts clefs
     - 1.2.2. Modélisation des caractéristiques des concepts clefs
       identification des points communs, des points de variations et des dépendances et interactions entre les caractéristiques.
2. Conception de domaine
   - 2.1. Identification de l'architecture globale d'implantation
   - 2.2. Identification et spécification des langages de domaine spécifiques (DSL)
3. Réalisation de domaine
   - 3.1. Implantation des langages de domaine spécifiques (DSL)
   - 3.2. Réalisation des composants d'implantation

FIG. 5.1 – *Les étapes de la méthode DEMRAL (traduction de [47, Fig. 133, p. 273])*

DEMRAL note dès cette étape que, même si la définition d'un TAD contient les opérations admissibles sur ce TAD, il convient de n'inclure dans la définition du TAD que les opérations *basiques*. Les opérations impliquant plusieurs TAD (produit matrice/vecteur par exemple) ou jugées *complexes* seront analysées et modélisées comme des algorithmes ne faisant pas partie du TAD.

Les TAD sont le premier pas vers l'orienté-objet (voir §1.4.1 page 28) et se traduisent généralement aisément en des classes les représentant. Un choix important concernant les opérations basiques des TAD doit être fait : *Font-elles partie ou non de l'interface de la classe représentant le TAD?* Nous avons vu aux paragraphes 3.5 et 4.5 que ce choix n'est pas seulement un choix d'implantation, mais aussi un choix de conception important qui est guidé par le modèle de programmation.

Une fois les concepts du domaine identifiés, il convient de modéliser les caractéristiques de ces concepts. Une originalité de DEMRAL est de suggérer l'utilisation *d'ensembles caractéristiques de départ* [47, "Feature starter set", pp. 109 et 276–279], qui permettent de trouver itérativement les caractéristiques des concepts d'un domaine. Ces ensembles de départ sont spécifiques au domaine, et nous complètons au paragraphe §5.4 ceux de DEMRAL [47, §9.3.2.2.1 et §9.3.2.2.2] par ceux que nous considérons essentiels pour une application parallèle de calcul scientifique. Il est à noter qu'à l'inverse d'autres méthodes de conception orientées-objet, DEMRAL, intègre également les caractéristiques d'implantation dans la phase d'analyse initiale. Concrètement, la gestion de la mémoire, les optimisations d'algorithmes liés à une spécialisation d'un TAD (matrice *creuse* par exemple), des variations d'implantations (liaison dynamique ou statique)...font partie des ensembles caractéristiques de départ.

5.3.2 Conception de domaine

L'objectif de cette phase est le découpage de l'architecture globale en modules regroupant un ou plusieurs des concepts identifiés précédemment. Les langages spécifiques de domaine (DSL[61]) sont ensuite spécifiés. Ce sont eux qui constitueront l'interface utilisateur de la librairie.

Langages spécifiques de domaine

Un DSL est un langage [de programmation] spécialisé capable d'exprimer synthétiquement les problèmes d'un domaine [198]. SQL est un DSL dans le domaine des bases de données, Make est un DSL dans le domaine de la gestion des dépendances de compilation, ...Suivant le même schéma qu'au chapitre 1 (page 4) un DSL offre un modèle de programmation, un traducteur lui est associé, ce dernier le transcrit dans un langage d'exécution respectant un modèle d'exécution (voir figure 1.1 page 5). La différence avec un langage de programmation général (ou générique) est qu'un DSL permet d'exprimer spécifiquement les programmes d'un domaine, mais n'est d'aucune aide dans le cas général.

Un DSL est donc un langage de programmation offrant un modèle de programmation adapté à un domaine d'application. DEMRAL prévoit donc les étapes suivantes :

1. Identification des DSLs utilisateurs

61. **D**omain **S**pecific **L**anguages

2. Identification des interactions entre les DSLs
3. Spécification des DSLs

Il est à noter que les DSLs sont par nature dépendants du domaine, et c'est aussi pour cela qu'ils sont très utiles. Dans notre cas, on peut considérer que les formes de matrices (§4.4) constituent un DSL important pour le domaine de l'algèbre linéaire.

5.3.3 Réalisation de domaine

On implante tous les objets modélisés et spécifiés dans les étapes précédentes :

- DSL utilisateurs
- TAD
- algorithmes

La réalisation de ces composants nécessite des fonctionnalités d'implantation dans le langage cible (héritage, polymorphisme, liaison tardive, contrôle de type dynamique, généricité). Si des fonctionnalités ne sont pas directement disponibles dans le langage cible, la réalisation nécessitera l'introduction de motifs d'implantation [76, 17]. L'implantation des DSLs peut-être plus ou moins complexe, suivant si le DSL en question nécessite la réalisation du traducteur associé ou non. Dans notre cas, nous n'avons utilisé que les fonctionnalités du langage C++.

5.4 Intégration des éléments du parallélisme

Nous ne connaissons pas beaucoup de méthodes de conceptions orientées-objet intégrant le parallélisme, et cela semble venir du fait que les applications sont d'abord conçues séquentiellement et ensuite parallélisées. Suivant cette optique, c'est le modèle de programmation parallèle qui guide la parallélisation et donc la conception de l'application parallèle. On peut prendre l'exemple de méthode suggérée par *Caromel* [32], puis par *Caromel, Belloncle et Roudier* [33] dont les étapes principales ont déjà été présentées à la figure 3.1 page 78 lors de la présentation d'Active-LAKe au chapitre 3. Nous ne connaissions pas les travaux de Czarnecki [47, 48] lorsque nous avons conçu et réalisé Active-LAKe et Gene-LAKe il s'avère que les techniques et les idées que nous avons utilisées sont quasiment les mêmes que certaines[62] de celle présentées dans les travaux de Czarnecki.

5.4.1 Les idées communes issues de LAKe et DEMRAL

Analyse de domaine

DEMRAL, en tant que méthode d'ingénierie de domaine, prévoit une phase d'analyse et de conception du domaine qui nécessite une bonne connaissance du domaine concerné

62. les travaux de Czarnecki ont des objectifs différents des nôtres (pas d'implantation parallèle par exemple) et ont un spectre beaucoup plus large que nos travaux en terme de méthodologie de conception

et pas seulement de l'application visée. Dans le processus de conception de LAKe, une définition du domaine a été menée dont une partie a été présentée au chapitre 2.

La modélisation du domaine de LAKe s'est malheureusement faite au travers des conceptions successives de LAKe (Active-LAKe, Gene-LAKe). Nous serions, probablement, arrivés à une solution complète satisfaisant nos objectifs de réutilisabilité plus rapidement si nous avions réalisé dès le départ l'étape de modélisation des concepts clefs préconisée par DEMRAL.

En effet, cette étape, que nous avons faite a posteriori, une fois confronté aux problèmes de contravariance et de l'allocation de matrice distribuée, nous aurait amenés plus ou moins directement à la conception des formes de matrices.

En revanche, nous avions tout comme il est préconisé dans DEMRAL, pris en compte dès le départ l'aspect distribué mais sans en prévoir tous les impacts (allocation de matrices distribuées) car nos *ensembles de caractéristiques de départ* étaient trop pauvres.

Conception de domaine

Les formes de matrices et les matrices avec formes sont un DSL au sens de DEMRAL. Contrairement à DEMRAL, nous n'avons pas cherché à identifier tous les DSLs qui auraient été utiles pour notre domaine. Le fait que nous soyons arrivés à la conception d'un DSL montre le bien-fondé de la démarche.

5.4.2 Intégrer les contraintes liées au Parallélisme

Le manque principal de DEMRAL, pour la conception d'applications parallèles, est d'intégrer directement dans les ensembles de caractéristiques de départ les contraintes du parallélisme. Que ce soit pour les algorithmes, ou pour les TAD, le parallélisme introduit des contraintes suivant chacun des aspects qui le compose :

1. Concurrence

   Les algorithmes qui auront des réalisations parallèles doivent être analysés afin d'indentifier quels sont les points de variations entre leur version séquentielle et les différentes versions parallèles. Concernant les algorithmes il faut donc examiner si il existe des variantes de réalisation parallèle.

   (a) Pour les algorithmes « simples » comme le produit matrice/vecteur il est raisonnable de prévoir des spécialisations parallèles explicites qui permettent d'exploiter au mieux les performances d'un support d'exécution parallèle spécifique.

   (b) Pour les algorithmes plus complexes, comme une méthode itérative, il convient de se demander si il est possible et raisonnable de pouvoir générer les versions séquentielle et parallèle avec des techniques de programmation générative.

   Concernant les TAD, il faut examiner comment seront [éventuellement] distribuées les structures de données qui les composent, suivant le modèle mémoire (partagée, distribuée ou NUMA). Il faudra ensuite prendre grand soin des opérations de création, destruction et restructuration (ajout ou obtention d'un élément de la structure de donnée ou construction d'une nouvelle structure par copie et restructuration d'une autre) de ces TAD.

2. Synchronisation
   L'aspect synchronisation peut être plus fortement lié aux données (synchronisation par les données comme dans le modèle de programmation à parallélisme de données) ou bien aux algorithmes (modèle de programmation à parallélisme de tâches). Il faut identifier quels types de synchronisations sont nécessaires, et, soit les réaliser explicitement dans les spécialisations explicites des algorithmes parallèles (c'est le cas de LAKe), ou bien étudier la conception d'un DSL adapté à ces synchronisations (c'est le cas des directives de compilation OpenMP [141]). Dans tous les cas, il faut être conscient du fait que la synchronisation est une caractéristique qui s'hérite mal [113, 110] et pose des problèmes de réutilisation et de redéfinition. Il est donc souvent plus raisonnable de ne pas inclure à la réalisation d'un TAD ou un algorithme sa synchronisation, mais plutôt de concevoir le TAD ou l'algorithme afin que l'on puisse y insérer la synchronisation voulue, par spécialisation ou paramétrisation.
3. Distribution
   La distribution est un autre aspect du parallélisme, donc un point de variation des programmes parallèles. La connaissance de la distribution des algorithmes ou des structures de données utilisées pour implanter les TAD est importante, pour une implantation efficace offrant de bonnes performances parallèles. La distribution induit des communications qui sont, dans notre cas, le critère à minimiser car elles coûtent beaucoup plus cher que les calculs effectués par les algorithmes. En d'autres termes, si les calculs effectués par un algorithme parallèle particulier nécessite beaucoup de communications, du fait de la distributions des données ou des calculs, l'utilisation de machines parallèles ne sera peut-être pas rentable. Dans nos applications, ce sont essentiellement les données contenues dans les TAD (matrices et vecteurs) qui sont distribuées. Les algorithmes d'algèbres linéaire doivent pouvoir vérifier que les matrices qu'ils utilisent ont des distributions compatibles avec une bonne exécution parallèle. C'est le rôle des formes de matrice de spécifier (implicitement) dans quelles opérations seront utilisées les matrices. La conception et la spécification d'un DSL pour la distribution nous parait donc essentiel. Une étude des types de distributions des TAD et des algorithmes doit être menée afin de prévoir ces différents cas comme des points de variations des TAD et des algorithmes. Le modèle mémoire est un critère à prendre en compte dans l'étude des distributions.
4. Communication
   Les communications sont le résultat de la distribution des données et des calculs qui impliquent que les tâches concurrentes (ou une tâche agissant sur des données distribuées) doivent échanger des données. Le modèle mémoire est un critère à prendre en compte dans l'étude des communications engendrée par les distributions, car les contraintes de performances seront différentes pour des modèles à mémoire partagée, distribuée ou hiérarchique (NUMA). Dans tous les cas, on cherche à préserver la meilleure localité des données, par rapport aux calculs, afin de minimiser les communications. Les synchronisations engendrent également des communications qui sont là pour garantir la cohérence des calculs. Les communications générées par des points de synchronisation sont à étudier avec les contraintes de synchronisation des algorithmes concernés.

Ces caractéristiques de départ doivent *toutes* être prises en compte dans la conception de la librairie et conduiront certainement à la conception d'un ou plusieurs DSL utilisateurs, permettant de masquer au maximum les aspects du parallélisme à l'utilisateur de la librairie, tout en laissant un bon contrôle du parallélisme au programmeur du ou des DSL concernés.

5.5 L'orienté-objet et après...

Un certains nombre de modèles de programmation émanent du modèle orientée-objet, ces « extensions » ou compléments ont pour but de pallier les manques de réutilisabilité et d'extensibilité du modèle objet. Nous traçons ici une brève revue des différentes approches et concluons sur la position des formes de matrices dans ce contexte.

5.5.1 Programmation/implantation ouverte

Les principes fédérateurs de la programmation ouverte (ou **O**pen **I**mplementation (OI)) [137] ont été identifiés par les chercheurs de Xerox PARC[63]. L'idée est de passer de la notion de logiciel « boîte noire » [99] ou l'encapsulation est maximum à la notion de logiciel à implantation ouverte ou l'utilisateur du module a un certain contrôle sur les choix d'implantation de ce module. L'utilisateur du module est capable de le configurer ou bien de lui fournir une partie de son implantation. Une boîte noire offre, à l'utilisateur, la possibilité d'utiliser son interface en ne lui donnant *aucun* accès à la manière dont elle est implantée. Les principes d'implantation ouverte indique qu'un module, implanté de façon ouverte, doit permettre à l'utilisateur de paramétrer sa propre implantation.

La conception des matrices avec formes suit les principes d'implantation ouverte en laissant le choix à l'utilisateurs des matrices, d'une part de leur spécifier une forme et d'autre part d'utiliser ces formes pour influencer la création et les autres opérations sur ces matrices.

Un module à implantation ouverte possède donc deux interfaces, la première est une interface d'utilisation comme celle de la boîte noire et l'autre est une méta-interface qui sert à la configuration du module. Le paramètre générique `TShape: Matrix_Shape` d'une matrice avec forme est sa méta-interface.

5.5.2 Programmation par aspects

La programmation par aspect (**A**spect-**O**riented **P**rogramming (AOP)) est l'héritière directe de la programmation ouverte. Les travaux réalisés sur la programmation par aspect [101, 93] sont issus de la même équipe de recherche du Xerox PARC [7].

L'idée principale de la programmation par aspect est que certaines caractéristiques d'un programme sont tranverses à sa conception. Plus simplement, si l'on conçoit un programme ou une librairie de façon modulaire (comme le préconise les démarches orientées-objet), l'introduction d'un aspect dans ce programme ou cette librairie apparaitra dans

63. Xerox **P**alo **A**lto **R**esearch **C**enter, `http://www.parc.xerox.com/parc-go.html`

l'implantation de quasiment tous les modules. Des exemples, désormais classiques, d'aspects sont :

- la persistance
- la distribution
- la synchronisation
- la stratégie de gestion d'erreur
- ...

C'est pour cette raison que nous avons qualifié d'aspects les 4 caractéristiques du parallélisme. Ce sont effectivement des aspects, car ils se retrouvent à tous les niveaux de la programmation d'un programme parallèle. C'est aussi la raison pour laquelle la plupart des moyens de programmations parallèles sont intrusifs.

Les moyens de programmation par aspect se décomposent alors de la façon suivante :

- un langage de base :
  un langage de programmation classique comme C++, Fortran ou n'importe quel autre langage, qui sert à la programmation algorithmique de l'application.
- un langage d'aspects :
  qui permet de décrire par dessus le langage de base, les aspects du programme pour lesquels il a été conçu.
- un tisseur d'aspects[64] :
  qui aura pour tâche de prendre en entrée deux programmes l'un réalisé au moyen du langage de base et l'autre au moyen du langage d'aspect et rendre en sortie un programme écrit dans un langage cible (qui peut être le langage de base) qui aura intégré les aspects, décrit dans le langage d'aspect, au programme de base.
  Dans le cadre de notre présentation du paragraphe 1.2, le tisseur d'aspect est un traducteur (voir figure 1.1 page 5) spécial capable de fusionner deux modèles programmation différents en produisant du code conforme au modèle d'exécution choisi.

Des exemples de langages parallèles à aspect sont les langages parallèles à extensions compatibles, que nous avons déjà cités, OpenMP [141] et HPF [91], voir aussi annexe A. Dans ces cas précédents, le langage de base est C/C++ ou Fortran et le langage d'aspect est constitué de l'ensemble des directives de compilations spécifiées par ces langages [90, 144, 143]. Le tisseur d'aspects est le compilateur répondant aux spécifications en questions. On peut noter que, dans ce cas, le tisseur d'aspect et le compilateur final sont intégrés de telle façon que le compilateur Fortran/C/C++ inclut le tisseur d'aspects et génère directement le code exécutable à partir des deux programmes.

La réalisation du tisseur d'aspect est certainement la partie la plus difficile du travail, car, le but est que le tisseur génère un programme *efficace* intégrant correctement les aspects. Ceci demande que le tisseur puisse faire le lien entre le langage d'aspect et le langage de base.

REMARQUE **5.1 (Programmation par Aspect et réflexivité)** *La programmation par aspect est intimement liée aux approches réflexives qui définissent des protocoles méta-objet (voir §1.5 (Classifications) et [101, §8.2]). La réflexivite fournit un moyen d'introspection puissant dans le langage de base et la définition du protocole méta-objet (MOP)*

64. ou tresseur d'aspect issu de l'anglais : *Aspect weaver*

est la méta-interface permettant d'introduire les aspects dans le langage de base. A titre d'exemple, on peut voir comment C++// (voir [33, 27] et chapitre 4) permet la gestion implicite des aspects parallèles, synchronisation et distribution.

5.5.3 Programmation générative

La programmation générative [60, 48] est le terme général pour spécifier que l'on écrit à la fois des programmes et des méta-programmes. C'est-à-dire que les méta-programmes *génèrent* des programmes issus de l'exécution du méta-programme prenant en paramètres des données, dont certaines sont des programmes ou des méta-programmes. La programmation générative présente donc un cadre unificateur pour la programmation par aspect, la réflexivité, la méta-programmation. L'intérêt de la programmation générative telle qu'elle est présentée dans [48] est qu'elle comporte à la fois des techniques d'implantation et des exemples de démarches méthodologiques qui peuvent manquer dans les approches précédentes.

5.5.4 Formes de matrices : un langage d'aspects ?

On peut remarquer une grande similitude entre les formes de matrices et le langage d'aspect constitué par les directives HPF de distributions. Nous avons dit que les formes de matrices était un langage de domaine spécialisé (DSL), on aurait tout autant pu le qualifier de langage d'aspect. À la différence près, que nous n'avons pas implanté de tisseur d'aspects, mais que le tressage s'effectue quasiment manuellement par spécialisation des fonctions génériques prenant comme argument des matrices avec forme. Le reste du tressage s'effectue par instanciation générique. Notre outil de tressage est donc la partie du compilateur C++ qui gère les génériques.

Notre but n'était pas de développer un langage d'aspect général permettant de paralléliser n'importe quelle application (comme c'est le cas d'HPF) mais d'offrir un DSL parlant pour le domaine d'application. Le résultat est que le programmeur du domaine spécifie des contraintes *de domaine*, « la matrice A aura la forme du produit de C par X ». Ceci a été manuellement traduit en, « les matrices A, C et X doivent être distribuées de manières à ce que l'opération parallèle $A = C \cdot X$ soit efficace. ». Nous pensons que les formes de matrices donnent plus d'informations utiles [65] à la fois au programmeur du domaine et au programmeur de la librairie parallèle, en laissant plus de liberté au programmeur parallèle pour implanter les opérations parallèles sur les matrices.

On peut noter que le problème de la redistribution peut s'exprimer également avec les formes de matrices si l'on inclut à l'interface des matrices avec forme la méthode `Matrix<TShape>::reshape(TShape S)` qui indique que l'objet concerné doit changer de forme :

Programme 5.1: *Exemple hypothétique de changement dynamique de forme*

```
1 // create a matrix Y which will be
2 // used as the result of a product of A*X
```

65. que des directives HPF

```
3 Y.create(A.shape()*X.shape());
4
5 [...]
6 // indicates that Y will be used in operation
7 // of kind W+Z
8 Y.reshape(W.shape()+Z.shape());
```

5.6 Conclusion

Nous avons présenté une méthodologie de conception ayant pour but de réutiliser et nous avons indiqué comment enrichir cette démarche afin d'intégrer les aspects du parallélisme. Nous avons indiqué des approches complémentaires aux approches objets qui nous semblent être incontournables pour arriver à une bonne réutilisabilité des applications parallèles. Nous avons montré les liens qui existaient entre nos approcbes pour la conception et la réalisation de LAKe ainsi que la contribution que constitue les formes de matrices dans ce cadre. Nous pensons que la démarche exposée dans ce chapitre constitue une ébauche de méthode pour la conception d'applications parallèles réutilisables.

Conclusion

Nous avons présenté dans cette thèse les possibilités d'utilisation de conception et de programmation orientée-objet parallèle. L'accent a été mis sur les possibilités de réutilisation séquentielle/parallèle, qu'apporte une approche objet. Nous avons ainsi montré qu'il était possible d'atteindre un bon niveau de réutilisabilité sans sacrifier les performances.

Plusieurs moyens de mise en œuvre, ont été étudié et notamment un modèle à objet actifs et la simple utilisation de MPI et d'un langage à objets séquentiel.

Nous avons notamment conçu et expérimenté un mécanisme de partage en lecture permettant aux langages à objets parallèles qui offrent un modèle à objets actifs d'être efficaces pour l'implantation d'une librairie d'algèbre linéaire. Cette étude a montré tout l'intérêt des approches réflexives pour l'implantation de ce type de mécanisme et surtout que le modèle de programmation doît être suffisamment ouvert et flexible pour permettre au programmeur de spécifier une optimisation qui est du domaine de l'implantation.

Nous avons ensuite montré de façon progressive l'insuffisance des caractéristiques classiques des LAOs que sont le polymorphisme et la liaison dynamique dans l'implantation parallèle efficace et réutilisable de méthodes itératives d'algèbre linéaire. Nous avons montré, comment une approche de programmation générative, utilisant la généricité pouvait permettre d'atteindre une réutilisabilité séquentielle/parallèle optimale. Dans la version finale de notre librairie, le code des méthodes itératives est identique pour leur implantation séquentielle et parallèle. Ce résultat est obtenu grâce au concept de matrice avec forme et de formes de matrices que nous avons introduit. Le concept de matrices avec forme illustre le fait qu'un spécialiste d'un domaine peut développer son application séquentielle en utilisant un langage spécifique de domaine spécifié par lui et un spécialiste du parallélisme puis dériver son code parallèle par une simple recompilation.

Fort de ces deux expériences, nous avons ensuite proposé des extensions à une méthodologie d'ingénierie de domaine permettant d'intégrer dès l'analyse les caractéristiques du parallélisme. Cette ébauche de méthodologie de conception veut être le point de départ d'une méthode permettant une programmation parallèle réutilisable.

Nous pensons que les approches de programmation générative telle que la programmation par aspect, s'appuyant sur des techniques de méta-programmation sont une clef de la réutilisabilité des applications séquentielles et parallèles. Nous aimerions poursuivre d'une façon plus générale l'étude de ces concepts dans des domaines d'applications différents afin de montrer leur puissance et leur utilisabilité dans un projet industriel. Nous aimerions également participer au développement d'outils permettant de mettre en oeuvre ces techniques qui restent pour l'instant réservées à des utilisateurs avertis.

Annexe A

High Performance Fortran (HPF)

A.1 Introduction

Fortran, en tant que langage dédié au calcul numérique, devait s'ouvrir également à la programmation parallèle . C'est à cet effet que depuis 1992, un groupe de travail[66] regroupant industriels, chercheurs, laboratoires de recherche et universités a étudié la définition d'une extension de Fortran 90 appelée HPF (**H**igh **P**erformance **F**ortran). Nous ne donnons ici qu'une courte introduction aux fonctionnalités d'HPF nous référons à [154] au site Web[67] [91] du HPF Forum pour de plus amples renseignements.

A.2 Directives de compilations

Les directives de compilations sont l'outil de base de l'introduction du parallélisme dans HPF. En effet, un programme HPF doit pouvoir être compilé par un compilateur Fortran 90 standard. Ceci signifie en fait que :

$$\textbf{HPF} = \textbf{F90} + \textbf{Commentaires} \tag{A.1}$$

Comme nous pouvons le voir sur le listing A.2 plusieurs directives de compilations ont été introduites. On peut considérer deux groupes de directives : celles liées à la distribution de données et celles décrivant les boucles parallèles.

Nous ne détaillerons pas toutes les directives de compilation d'HPF ou toutes les options des directives que nous examinerons, mais nous montrerons les principales fonctionnalités apportées par les directives HPF. Pour une description complète du langage, les spécifications du langage sont décrites dans [89, 90].

`DISTRIBUTE` Permet de distribuer un tableau déclaré préalablement. Les distributions autorisées sont régulières par bloc, éventuellement cycliques. On peut voir un exemple en figure A.1. Syntaxe :

`!HPF$ DISTRIBUTE T(`*order*`)`,

66. The High Performance Fortran Forum (HPFF)
67. `http://dacnet.rice.edu/Depts/CRPC/HPFF/index.cfm`

Programme A.2: *Directive de compilation HPF*

```
1  !HPF$ TEMPLATE, DIMENSION(400,400) :: T
2        Real, Dimension(400,400)    :: A, B
3        Real, Dimension(400)        :: X, Y
4  !HPF$ DISTRIBUTE T(BLOCK, CYCLIC)
5  !HPF$ ALIGN with T :: A, B
6  !HPF$ ALIGN (I) with T(I,*) :: X(I)
7  !HPF$ ALIGN (J) with T(*,J) :: Y(J)
8        A = A - 2*B
9        Do I = 1, 400
10           Y(I) = Sum(A(I,:)*X)
11       End Do
12       Y = Y - 2*X
```

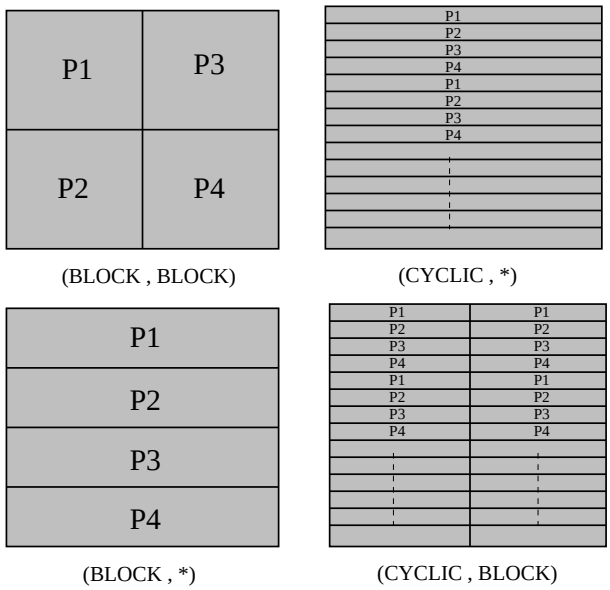

FIG. A.1 – *Distribution en HPF sur 4 processeurs*

Programme A.3: *F90 + Directives = HPF*

```
1       Integer, Parameter              :: NPROCS
2                       = Number_of_processors()
3       Real, Dimension(NPROCS,100)     :: Table
4       Real, Dimension(NPROCS)         :: X
5       Integer Dimension(NPROCS)       :: Indirect
6 !HPF$ DISTRIBUTE(CYCLIC)      :: Indirect, X
7 !HPF$ DISTRIBUTE(CYCLIC,*)    :: Table
8       Forall(P = 1:NPROCS)
9          X(P) = Table(P,Indirect(P))
```

où *order* peut être `*, BLOCK ou CYCLIC`.

`ALIGN` Permet d'aligner un tableau sur un autre tableau déjà distribué. On peut aussi aligner un tableau d'ordre inférieur sur les dimensions nécessaires d'un tableau d'ordre supérieur. On peut, par exemple, aligner un vecteur sur les colonnes d'une matrice. Syntaxe :

```
!HPF$ ALIGN (I) with A(I,*) :: V
```

`TEMPLATE` Permet de définir un tableau d'alignement virtuel, le TEMPLATE HPF, est un tableau d'indices qui ne contient aucune donnée. On peut l'utiliser ensuite dans une directive `ALIGN` ou `REALIGN`.

```
!HPF$ TEMPLATE, DIMENSION(400,400) :: T
```

Il ne faut pas confondre les `template` C++, qui sont le support de la généricité en C++ et les `TEMPLATE` HPF que nous venons de décrire.

`DYNAMIC, REDISTRIBUTE, REALIGN` Les tableaux ayant été déclarés `DYNAMIC` au moyen de la directive `!HPF$ DYNAMIC :: A,B` peuvent être RE-distribués ou RE-alignés avec les directives `REDISTRIBUTE` et `REALIGN` de la même façon qu'avec `DISTRIBUTE` et `ALIGN`.

`INDEPENDENT` La directive `!HPF$ INDEPENDENT` indique au compilateur que les itérations de la boucle qui suit sont indépendantes et qu'il peut les exécuter dans n'importe quel ordre. Un exemple a déjà été donné sur le listing 1.2 page 21.

`FORALL` Une extension de langage a été introduite dans HPF c'est la construction `FORALL`, qui permet l'écriture simple d'une boucle avec un adressage indirect, où chacun des processeurs possède la partie locale du tableau d'indirection. On peut voir un exemple tiré de [173] à la figure A.3. Fortran 95 inclu désormais la construction `FORALL` (voir [90, page 11, lignes 24–33]), ce qui fait que l'identité **HPF=F90+Commentaires**, partiellement fausse à cause de la construction FORALL, ne le sera plus lorsque Fortran 95 aura complètement remplacé Fortran 90.

Annexe B

The Message Passing Interface (MPI) : Une introduction

B.1 Les fonctions principales de MPI

MPI est une librairie de communication supportant le modèle de programmation parallèles par passage de messages (voir Déf. 1.12).

B.1.1 Les fonctions intrinsèques à la librairie

- MPI_INIT()
  Fonction d'initialisation des processus utilisant MPI. Ce doit être le premier appel à MPI d'un programme l'utilisant.
- MPI_FINALIZE()
  Le dernier appel à MPI.

Lors de l'appel de chaque processus à MPI_INIT(), MPI crée un communicateur global nommé MPI_COMM_WORLD qui pourra être utilisé dans les fonctions de communication. Nous expliquerons la notion de communicateur après la présentation des fonctions de communications de base de MPI, pour ces fonctions on peut, dans un premier temps, remplacer, à chaque fois que nécessaire, l'argument communicateur comm par MPI_COMM_WORLD.

B.1.2 Les fonctions de communications point-à-point

MPI offre une multitude de fonctions de communication point-à-point qui se différencient par les *modes* de communication : non-bloquant, bloquant, bufferisé, synchrone, asynchrone. Nous renvoyons au standard [118] pour la description complète de tous ces modes de communication.

- MPI_SEND(buf, count, datatype, dest, tag, comm)
  Envoi synchrone d'un message.
  - buf adresse du buffer d'émission

  - count nombre d'éléments contenus dans le buffer d'émission
  - datatype type de chaque élément du buffer d'émission
  - dest numéro du processus destinataire
  - tag étiquette associée au message
  - comm communicateur MPI
- MPI_RECV(buf, count, datatype, source, tag, comm, status)
  Réception synchrone d'un message. Les arguments sont les mêmes que pour MPI_SEND en remplaçant dest par source. L'argument supplémentaire status donne des informations supplémentaires sur la réception du message, après réception.
- MPI_ISEND(buf, count, datatype, dest, tag, comm, request)
  Envoi asynchrone d'un message. Les arguments sont les mêmes que pour MPI_SEND en rajoutant request, ce dernier argument permet de tester la complétion de la communication. C'est la version non-bloquante de MPI_SEND.
- MPI_IRECV(buf,count, datatype, dest, tag, comm, request)
  Réception asynchrone d'un message. Idem que pour MPI_ISEND.
- MPI_WAIT(request, status)
  Attente de complétion d'une communication non-bloquante. MPI_WAIT se termine lorsque la requête request, correspondant à une communication non-bloquante (MPI_ISEND MPI_IRECV), est terminée. La variable status permet de connaître des informations supplémentaires sur la communication (voir [118]).
- MPI_TEST(request, flag, status)
  Teste la complétion d'une communication non-bloquante. MPI_TEST a la même fonction que MPI_WAIT, sauf que MPI_TEST met à jour le booléen flag afin de savoir si la communication est terminée ou non et ne bloque pas l'exécution du processus.

B.1.3 Les fonctions de communications collectives

- MPI_BARRIER(comm)
  Barrière de synchronisation. L'appel à MPI_BARRIER bloque le processus appelant jusqu'à ce que tous les processus du communicateur comm aient appelés MPI_BARRIER. Tous les processus du communicateur comm *doivent* appeler MPI_BARRIER.
- MPI_BCAST(buffer, count, datatype, root, comm)
  Broadcast ou diffusion d'un message d'un membre (root) du groupe vers tous les autres.
  - buffer le buffer d'envoi pour le processus root et de réception pour les autres membres de comm.
  - count nombre d'élément du buffer buffer
  - datatype type de chaque élément du buffer
  - root le processus source de la diffusion
  - comm le communicateur

- MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
  Regroupement de données de chaque processus d'un groupe vers un membre particulier (root) du groupe. Chaque processus (y compris root) envoit un message au processus root.
  - sendbuf le buffer d'envoi pour tous les processus membres de comm.
  - sendcount nombre d'éléments du buffer sendbuf
  - sendtype type de chaque élément du buffer sendbuf
  - recvbuf le buffer de réception pour le processus root
  - recvcount nombre d'élément du buffer recvbuf
  - recvtype type de chaque élément du buffer recvbuf
  - root le processus récepteur du regroupement
  - comm le communicateur
- MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
  Diffusion personalisée, un membre d'un groupe distribue des messages différents à chacun des autres membres du groupe. Mêmes paramètres que MPI_GATHER.
- MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)
  Regroupement de données de chaque processus d'un groupe vers tous les membres d'un groupe. Mêmes paramètres que MPI_GATHER.
- MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)
  Diffusion général chaque processus d'un groupe envoie un message à chacun des autres membres du groupe. Mêmes paramètres que MPI_GATHER.
- MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)
  Réduction globale personalisée. Chaque processus envoie une donnée à un processus root, ces données sont combinées par MPI_REDUCE grâce à l'opérateur op pour fournir un résultat unique dans le buffer de réception du processus root, recvbuf.
  - sendbuf le buffer d'envoi pour tous les processus membres de comm.
  - recvbuf le buffer de réception pour le processus root
  - count le nombre d'élément du buffer sendbuf
  - datatype type de chaque élément du buffer sendbuf
  - op opération de combinaison des éléments de type datatype cette opération doit être associative et éventuellement commutative. Pour les types de base fournit par MPI des opérateurs de base tel (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LOR ...). L'utilisateur peut définir de nouvelles fonctions.
  - root le processus récepteur du regroupement
  - comm le communicateur
- MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)
  Réduction globale générale, idem que MPI_REDUCE mais ici chaque processus reçoit le résultat de la réduction.

B.2 Autres fonctionnalités de MPI

MPI offre de nombreuses autres possibilités qui en font une librairie portable et puissante. Une partie de la portabilité[68] par exemple, est assuré par la définition des types de données MPI.

Un certains nombres de types de base sont disponibles (MPI_INT, MPI_LONG, MPI_REAL, MPI_FLOAT, MPI_LOGICAL, MPI_BYTE ...) mais on peut également définir des types de données utilisateurs (ou types dérivés) grâce aux fonctions MPI_TYPE_STRUCT, MPI_TYPE_VECTOR ... Le type crée doit être ensuite appliqué avec MPI_TYPE_COMMIT avant d'être utilisé en tant que datatype dans les primitives de communications que nous venons de voir. MPI permets grâce aux types de base et aux types utilisateurs de transmettre n'importe quelle sorte de données dans un environnement *hétérogène*.

Les communicateurs sont le moyen dans MPI de définir des groupes processus qui auront des communications privilégiées. A l'initialisation MPI_INIT(), MPI crée un communicateur universel MPI_COMM_WORLD qui permet à tous les processus de communiquer. L'utilisateur peut ensuite créer d'autres communicateurs qui soient des sous-ensembles (au sens large, i.e. éventuellement un synonyme) du communicateur universel. Au sein de ces groupes les processus pourront effectuer des réductions, des opérations de communication globale sans risquer les conflits qu'entrainerait l'utilisation d'un unique espace de communication. Ainsi 2 processus communicant pour accomplir 2 opérations différentes pourront utiliser 2 communicateurs différents pour ne pas confondre les messages concernant ces 2 opérations. Le fait que l'ont puisse créer des synonymes de communicateurs, par exemple du communicateur universel, permet à un utilisateur de faire appel à une librairie externe (par exemple PETSc [16]) utilisant MPI tout en utilisant MPI lui-même sans risquer de perturber le fonctionnement de la librairie externe utilisant déjà MPI.

MPI permet également de définir des topologies virtuelles de processus via 2 fonctions MPI_CART_CREATE et MPI_GRAPH_CREATE. Ces fonctions créent un nouveau communicateur à partir d'un ancien communicateur qui aura une structure associé. Les topologies ainsi créees peuvent permettrent de spécifier des communications avec des voisins, ceci permet de structurer les communications. La correspondance entre topologies virtuelles MPI et les topologies physiques des machines [131] n'est pas spécifiée par la norme, mais des implantations de MPI pourraient exploiter cette correspondance afin de rendre les communications plus efficaces.

MPI-1 définie l'interface avec les langages C et Fortran 77, et MPI-2 y a ajouté la définition des interfaces avec C++ et quelques extensions pour Fortran 90. La plupart des implantations actuelles de MPI sont basées sur la version 1.1 [118] du standard qui manque encore de nombreuses fonctionalités (gestion dynamique de processus, entrées-sorties parallèles, interface pour C++, support explicite de threads ...).

68. portabilité du code écrit avec MPI car ceci n'assure pas la portabilité des différentes implantations de MPI

B.3 MPI-2, nouvelles fonctionnalités

La version 2.0 du standard, MPI-2 [119] a été publiée par le forum MPI en Juillet 1997, elle précise et ajoute de nombreuses fonctionnalités à MPI notamment : la création dynamique de processus, les communications unilatérales, les entrées/sorties parallèles ... Le document [119] décrivant MPI-2 contient également la norme MPI-1.2 qui est constituée de précisions et corrections mineures de MPI-1.1.

Bibliographie

[1] M. Abadi and L. Cardelli, *A Theory of Objects*, Monographs In Computer Science, Springer Verlag, 1996. http://www.luca.demon.co.uk/TheoryOfObjects.html.

[2] *The ABCL family of languages.* http://web.yl.is.s.u-tokyo.ac.jp/pl/abcl.html.

[3] *ACE home page*, Fév. 2000. http://siesta.cs.wustl.edu/~schmidt/ACE.html.

[4] G. Agha, *Actors : A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

[5] G. A. Agha and W. Kim, *Actors : a unifying model for parallel and distributed computing*, Tech. Rep. RR-95-02, Open System Laboratory, 199 ? http://yangtze.cs.uiuc.edu/~agha/.

[6] E. Anderson, Z. Bai, C. Bishof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK User's Guide*, SIAM, 1992. http://www.netlib.org/lapack/lug/lapack_lug.html.

[7] *Aspect-Oriented Programming (AOP) home page*, Oct. 1999. http://www.parc.xerox.com/csl/projects/aop/.

[8] M. Arioli, I. S. Duff, J. Noailles, and D. Ruiz, *A block projection method for sparse matrices*, SIAM J. Matrix Anal. and Appli., 13 (1992), pp. 47–70.

[9] W. E. Arnoldi, *The principle of minimized iteration in the solution of the matrix eigenvalue problem*, Quart. Appl. Math., 9 (1951), pp. 17–29.

[10] *Accelerated Strategic Computing Initiative (ASCI) project.* http://www.llnl.gov/asci/.

[11] *AspectJ home page*, Oct. 1999. http://aspectj.org/.

[12] *Athapascan-0 home page*, Fév. 2000. http://www-apache.imag.fr/software/ath0/.

[13] *Athapascan-1 home page*, Fév. 2000. http://www-apache.imag.fr/software/ath1/.

[14] Z. Bai, D. Day, and Q. Ye, *ABLE : an adaptative block lanczos method for non-hermitian eigenvalue problem*, Research Report 95-04, University of Kentucky, Dept. of Mathematics, May 1995. Revised February 1996.

[15] H. E. Bal, M. F. Kaashoek, A. S. Tanenbaum, and J. Jansen, *Replication techniques for speeding up parallel applications on distributed systems*, Concurrency Practice & Experience, 4 (1992), pp. 337–355. http://www.cs.vu.nl/vakgroepen/cs/orca_papers.html.

[16] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *Efficient management of parallelism in object oriented numerical software libra-*

ries, in Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhauser Press, 1997, pp. 163–202. ftp ://info.mcs.anl.gov/pub/petsc/scitools96.ps.gz.

[17] G. Baumgartner, K. Läufer, and V. F. Russo, *On the interaction of object-oriented design patterns and programming languages*, Tech. Rep. CSD-TR-96-020, Purdue University, CS Dept., Feb. 1996. http ://www.cis.ohio-state.edu/~gb/.

[18] P. Beaugendre, T. Priol, G. Alléon, and D. Delavaux, *A client/server approach for HPC applications within a networking environment*, in Proceedings of HPCN'98, Apr. 1998.

[19] P. Beckman, D. Gannon, and E. Johnson, *Portable parallel programming in HPC++*, 1996. http ://www.extreme.indiana.edu/hpc++/.

[20] *BLAS technical forum*, July 2000. http ://www.netlib.org/cgi-bin/checkout/blast/blast.pl.

[21] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. X. Yang, *Distributed pC++: Basic ideas for an object parallel language*, tech. rep., IRISA - Université de Rennes, Jan. 1996. http ://www.extreme.indiana.edu/sage/index.html.

[22] L. Bougé, *The data parallel programming model: A semantic perspective*, in Perrin and Darte [154], pp. 4–26. School on Data Parallelism, Les Ménuires, France, March 25–28.

[23] J.-P. Briot, R. Guerraoui, and K.-P. Lohr, *Concurrency and distribution in object-oriented program*, ACM Computing surveys, 30 (1998), pp. 291–329. http ://lsewww.epfl.ch/~rachid/papers/cs.ps.gz.

[24] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Objects Group, G. T. Leavens, and B. Pierce, *On binary methods*, Theory and Practice of Object Systems, 1 (1995), pp. 221–242. http ://www.luca.demon.co.uk/.

[25] *BSP worldwide home page*, Oct. 1999. http ://www.bsp-worldwide.org/.

[26] R. Buyya, ed., *High Performance Cluster Computing*, vol. 1 – Architectures and Systems, Prentice Hall, 1999. http ://www.dgs.monash.edu.au/~rajkumar/cluster/index.html.

[27] *C++// home page*, Fév. 2000. http ://www.inria.fr/oasis/c++ll/.

[28] C. Calvin and L. Colombet, *Introduction au modèle de programmation par processus communicants : deux exemples PVM et MPI*, Rapport Projet Apache Numéro 12, INRIA, July 1994.

[29] D. K. G. Campbell, *A survey of models of parallel computation*, Tech. Rep. YCS-97-278, University of York, 1997. ftp ://ftp.cs.york.ac.uk/reports/YCS-97-278.ps.Z.

[30] L. Cardelli, *A semantics of multiple inheritance*, Information and Computation, 76 (1988), pp. 138–164. http ://www.luca.demon.co.uk/.

[31] L. Cardelli and P. Wegner, *On understanding types, data abstraction and polymorphism*, Computing Survey, 17 (1985), pp. 471–522. http ://www.luca.demon.co.uk/.

[32] D. Caromel, *Toward a method of object-oriented concurrent programming*, Communication of the ACM, 36 (1993), pp. 90–102. http://www-sop.inria.fr/sloop/personnel/Denis.Caromel/ps/toward.ps.

[33] D. Caromel, F. Belloncle, and Y. Roudier, *The C++// language*, tech. rep., I3S - CNRS - University of Nice, 1996. http://www.inria.fr/oasis/c++ll/.

[34] D. Caromel, E. Noulard, and D. Sagnol, *Partage en lecture pour la programmation à objets parallèle et distribuée*, in RENPAR'11, June 1999. http://www.irisa.fr/renpar11/.

[35] ——, *SharedOnRead Optimization in Parallel Object-Oriented Programming*, in ISCOPE'99, no. 1732 in Lecture Notes in Computer Science, Dec. 1999. http://www.acl.lanl.gov/iscope99/.

[36] G. Castagna, *Covariance and contravariance: Conflict without a cause*, ACM Transactions on Programming Languages and Systems, 17 (1995), pp. 431–447. http://www.ens.fr/~castagna/.

[37] *Cetus links*, Fév. 2000. http://www.cetus-links.org/.

[38] M. Chandy, I. Foster, K. Kennedy, C. Koelbel, and C.-W. Tseng, *Integrated support for task and data parallelism*, International Journal of Supercomputer Applications, 8 (1994), pp. 80–98. http://www-fp.mcs.anl.gov/~foster/papers.html.

[39] B. Chapman, P. Mehrotra, J. V. Rosendale, and H. Zima, *A software architecture for multidisciplinary applications: Integrating task and data parallelism*, Technical Report TR-94-18, Institute for Computer Applications in Science and Engineering, 1994. ftp://ftp.icase.edu/pub/techreports/94/94-18.ps.Z.

[40] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. Whaley, *A proposal for a set of parallel basic linear algebra subprograms*, LAPACK Working Note 100 UT, CS-95-292, May 1995. http://www.netlib.org/lapack/lawns.

[41] W. R. Cook, W. L. Hill, and P. S. Canning, *Inheritance is not subtyping*, in Proceedings of Principles of Programming Languages, ACM, 1990, pp. 125–135.

[42] J. O. Coplien, *Multi-Paradigm Design for C++*, Addison-Wesley, 1999. http://www1.bell-labs.com/user/cope/bibliography.html.

[43] ——, *Multi-Paradigm Design*, PhD thesis, Dept. of Infromatica, Faculteit Wetenschappen, Vrije Universiteit Brussel, July 2000. http://www1.bell-labs.com/user/cope/bibliography.html.

[44] A. Corbel and F. Fleter, *LINDA: un modèle de programmation parallèle*, Calculateurs Parallèles, 7 (1995), pp. 159–172.

[45] M. Cosnard and D. Trystram, *Algorithmes et architectures parallèles*, Inter-Editions, Dec. 1993.

[46] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, Aug. 1998. http://www.cs.berkeley.edu/~culler/book.alpha/index.html.

[47] K. Czarnecki, *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component*

Models, PhD thesis, Dept. of Computer Science and Automaton, Technical University Ilmenau, Oct. 1998. http://www.prakinf.tu-ilmenau.de/~czarn/.

[48] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, May 2000. http://www.generative-programming.org/.

[49] G. Damm, *Conception et Réalisation d'une Bibliothèque de Modèles d'Architectures Parallèles*, PhD thesis, Université Pierre et Marie Curie (Paris VI), Dec. 1996.

[50] J. Daniel, *Au cœur de CORBA avec Java*, Vuibert Informatique, Apr. 2000.

[51] M. J. Daydé and I. S. Duff, *Blocked implementation of level 3 BLAS for RISC processors*, Tech. Rep. RT/APO/96/1, LIMA-ENSEEIHT, 1996. ftp://ftp.enseeiht.fr/pub/numerique/BLAS/RISC.

[52] V. K. Decyk, C. D. Norton, and B. K. Szymanski, *High performance object-oriented programming in Fortran 90*, Oct. 1999. http://www.cs.rpi.edu/ szymansk/oof90.html.

[53] E. Deelman, W. K. Kaplow, B. K. Szymanski, P. Tannenbaum, and L. Ziantz, *Languages, Compilers and Run-Time Systems for Scalable Computers*, Kluwer Academic Publishers, 1996, ch. 13, Integrating Data and Task Parallelism in Scientific Programs, pp. 169–184.

[54] F. B. Denis Caromel and Y. Roudier, *Parallel Programming with C++*, MIT Press, 1996, ch. The C++// system. http://www-sop.inria.fr/oasis/c++ll/c++ll.pdf.

[55] J. Dongarra, V. Eijkhout, and A. Kalhan, *Reverse communication interface for linear algebra templates for iteratives methods*, Lapack Working Note 99, Oak Ridge National Laboratory, May 1995. http://www.netlib.org/lapack/lawns.

[56] J. Dongarra, A. Lumsdaine, R. Pozo, and K. A. Remington, *Iterative Methods Library (IML) Reference Guide*, NIST, April 1996. http://math.nist.gov/iml++/.

[57] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, *Numerical Linear Algebra for High-Performance Computers*, SIAM, 1998.

[58] *The distributed shared memory (DSM) home pages*, Oct. 1999. http://www.cs.umd.edu/~keleher/dsm.html.

[59] G. Edjlali, S. Petiton, and N. Emad, *Interleaved parallel hybrid arnoldi method for a parallel machine and a network of workstations*, in Conference on Information, Systems, Analysis and Synthesis (ISAS'96), Orlando, USA, July 1996.

[60] U. Eisenecker, *Generative programming references*, Oct. 1999. http://home.t-online.de/home/Ulrich.Eisenecker/gpref.htm.

[61] N. Emad, *Détection de parallélisme à l'aide de paralléliseurs automatiques*, Tech. Rep. MASI 92.54, MASI, Institut Blaise Pascal, 1992.

[62] ——, *Data parallel Lanczos and Padé-Rayleigh-Ritz methods on the CM5*, in Science on the Connection Machine System, H. S. Jean-Michel Alimi, Arturo Serna, ed., 1995, pp. 29–45.

[63] *The Emerald distributed programming language*. http://www.cs.ubc.ca/nest/dsg/emerald.html.

[64] P. Feautrier, *Compiling for massively parallel architecture : a perspective*, Microprocessing and Microprogramming, 41 (1995), pp. 425–439. http ://www.prism.uvsq.fr/rapports/1994/abstract_1994_30.html.

[65] ——, *Automatic parallelization in the polytope model*, in Perrin and Darte [154], pp. 79–103. Rapport PRiSM N°1996/8. http ://www.prism.uvsq.fr/rapports/1996/abstract _1996_8.html.

[66] ——, *Distribution automatique des données et des calculs*, Techniques et Sciences Informatiques, 15 (1996), pp. 529–557. http ://www.prism.uvsq.fr/rapports/1995/abstract_1995_17.html.

[67] ——, *Parallélisation automatique*, 1997. Cours de DEA MISI, Université de Versailles – St-Quentin-en-Yvelines.

[68] ——, *Private communication*, Oct. 1999.

[69] M. J. Flynn, *Some computer organization and their effectiveness*, IEEE Trans. on Computers, C-21 (1972), pp. 948–960.

[70] I. Foster, *Task parallelism and high performance languages*, in Perrin and Darte [154], pp. 179–196. School on Data Parallelism, Les Ménuires, France, March 25–28. http ://www-fp.mcs.anl.gov/~foster/papers.html.

[71] I. Foster, D. R. Kohr, Jr., R. Krishnaiyer, and A. Choudhary, *Double standards : Bringing task parallelism to HPF via the message passing interface*, in Supercomputing '96 Conference Proceedings : November 17–22, Pittsburgh, PA, 1996. http ://www.supercomp.org/sc96/proceedings/SC96PROC/FOSTER2/INDEX.HTM.

[72] I. T. Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995. http ://www.mcs.anl.gov/dbpp.

[73] R. W. Freund and M. Malhotra, *A block-QMR algorithm for non-hermitian linear systems with multiple right-hand sides*, technical report sccm-96-01, Stanford University, Computer Sciences Dept., 1996. Revised version. http ://cm.bell-labs.com/who/freund/.

[74] C. Froidevaux, M.-C. Gaudel, and M. Soria, *Types de Données et Algorithmes*, McGraw-Hill, 1992.

[75] N. Furmento, Y. Roudier, and G. Siegel, *Parallélisme et distribution en C++, une revue des langages existants*, Rapport de Recherche RR-95-02, I3S, 1995. http ://www-sop.inria.fr/sloop/Nathalie.Furmento.

[76] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, 1995.

[77] D. Gannon, P. Beckman, E. J. T. Green, and M. Levine, *HPC++ and the HPC++lib toolkit*, Fév. 2000. http ://www.extreme.indiana.edu/hpc++/.

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

[79] *Add generic types to the java programming language*, Nov. 1999. Java Specification Request #000014. http ://java.sun.com/aboutJava/communityprocess/jsr/jsr_014_gener.html.

[80] *Generic Java (GJ) home page*, Nov. 1999. http ://www.cs.bell-labs.com/who/wadler/pizza/gj/index.html.

[81] X. Girod, *Conception par objets. MECANO : une méthode et un environnement de construction d'application par objet.*, PhD thesis, Université Joseph Fourier, Grenoble 1, June 1991.

[82] G. H. Golub and C. F. V. Loan, *Matrix Computation*, The John Hopkins University Press, 2nd ed., 1989.

[83] F. Guidec, *Un cadre conceptuel pour la programmation par objets des architectures parallèles distribuées : application à l'albgèbre linéaire*, PhD thesis, Université de Rennes 1, Rennes, France, June 1995. Voir aussi thèses éditées par l'IRISA. `http://www.irisa.fr`.

[84] W. Harris, *Contravariance for the rest of us*, Tech. Rep. HPL-90-121, Hewlett-Packard Software and Systems Laboratory, Aug. 1990.

[85] J. L. Hennessy and D. A. Patterson, *Computer Architecture : A Quantitative Approach*, Morgan Kaufmann, 2nd ed., Jan. 1996.

[86] M. R. Hestenes and E. L. Stiefel, *Methods of conjugate gradients for solving linear systems*, Journal of Research of the National Bureau of Standards, 49 (1952), pp. 409–436. section B.

[87] C. A. R. Hoare, *Communicating sequential processes*, Communication of the ACM, 21 (1978), pp. 666–677.

[88] *HPC++ home page*, Fév. 2000. `http://www.extreme.indiana.edu/hpc++/`.

[89] HPF Forum, *High Performance Fortran language specification v1.1*, tech. rep., Rice University, Houston, Texas, Nov. 1994. `ftp://softlib.rice.edu/pub/HPF/hpf-v11.ps.gz`.

[90] ——, *High Performance Fortran language specification v2.0*, tech. rep., Rice University, Houston, Texas, Jan. 1997. `ftp://softlib.rice.edu/pub/HPF/hpf-v20-final.ps.gz`.

[91] *HPF Forum*, Oct. 1999. `http://dacnet.rice.edu/Depts/CRPC/HPFF/index.cfm`.

[92] *Intentional Programming (IP) home page*, Oct. 1999. `http://www.research.microsoft.com/research/ip/`.

[93] J. Irwin, J.-M. Loingtier, J. R. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman, *Aspect-oriented programming of sparse matrix code*, in Proceedings of International Scientific Computing in Object-Oriented Parallel Environments, Lecture Notes in Computer Science, Springer Verlag, Dec. 1997. `http://www.parc.xerox.com/spl/groups/eca/pubs/papers/Irwin-ISCOPE97/`.

[94] *Iterative template library (ITL) home page*, Aug. 2000. `http://www.lsc.nd.edu/research/itl/`.

[95] *Java Grande Forum home page*, Oct. 1999. `http://www.javagrande.org/`.

[96] J.-M. Jézéquel, *Programmer les machines parallèles avec l'EPEE*, Calculateurs Parallèles, (1994), pp. 31–50. Num. Spécial « Les Langages à Objets ».

[97] E. Johnson, P. Beckman, and D. Gannon, *HPC++ : An experiment with the parallel standard template library*, Tech. Rep. TR-96-51, Indiana University – Department of Computer Science, Dec. 1996. `http://www.extreme.indiana.edu/hpc++/`.

[98] K. Keahey and D. Gannon, *PARDIS: CORBA-based architecture for application-level parallel distributed computation*, in Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computation, Aug. 1997.

[99] G. Kiczales, *Beyond the black box : Open implementation*, IEEE Software, (1996). http://www.parc.xerox.com/spl/projects/oi/.

[100] G. Kiczales, J. des Rivieres, and D. G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

[101] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*, in Proceedings of European Conference on Object-Oriented Programming, no. 1241 in Lecture Notes in Computer Science, Springer Verlag, June 1997. http://www.parc.xerox.com/spl/groups/eca/pubs/papers/Kiczales-ECOOP97/.

[102] *LAM home page*, July 2000. Local Area Multicomputer. http://www.mpi.nd.edu/lam/.

[103] *LAPACK repository at netlib*, July 2000. http://www.netlib.org/lapack/index.html.

[104] C. Le Calvez, *Accélération de méthodes de Krylov pour la résolution de systèmes linéaires creux sur machines parallèles*, PhD thesis, Université de Lille 1, Villeneuve-d'Ascq, France, Dec. 1998. http://www-anp.lip6.fr/~lecalvez/publi_lecalvez.html.

[105] C. Le Calvez and B. Molina, *Implicitly restarted and deflated GMRES*, Numerical Algorithms, 21 (1999), pp. 262–285. voir aussi Tech. Rep. AS-184. http://www-anp.lip6.fr/~lecalvez/as-184.html.

[106] R. B. Lehoucq and K. Maschoff, *Implementation of an implicitly restarted block Arnoldi method*, tech. rep., Argonne National Laboratory, 199x. To be published in "Templates for eigenvalue problems, SIAM".

[107] D. Levine, D. Callahan, and J. Dongarra, *A comparative study of automatic vectorizing compilers*, Parallel Computing, 17 (1991), pp. 1223–1244. http://www.netlib.org/tennessee/vector.ps.

[108] P. Lignelet, *Structures de Données en Fortran90/95*, Masson, 1996.

[109] *Linda group home page*, Oct. 1999. http://www.cs.yale.edu/Linda/linda.html.

[110] S. F. lund, *Inheritance of synchronization constraints in concurrent object-oriented programming languages*, in Proceedings of European Conference on Object-Oriented Programming, no. 615 in Lecture Notes in Computer Science, Springer Verlag, June 1992. http://www.hpl.hp.com/personal/Svend_Frolund/docs/ecoop92.ps.

[111] *Mentat home page*, Oct. 1999. http://www.cs.virginia.edu/~mentat/.

[112] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, 2nd ed., 1997. http://www.eiffel.com/doc/oosc/page.html.

[113] S. E. Mitchell and A. J. Wellings, *Synchronisation, concurrent object-oriented programming and the inheritance anomaly*, Tech. Rep. YCS-94-234, University of York, Dpt. of Computer Science, June 1994. http://hypatia.dcs.qmw.ac.uk/data/uk/cs.york.ac.uk/YCS-94-234.ps.Z.

[114] J.-M. Moreno, *UNIX Administration*, Ediscience International, 2nd ed., 1998.

[115] R. B. Morgan, *On restarting the Arnoldi method for large nonsymmetric eigenvalue problems*, Math. Comput., 65 (1996), pp. 1213–1230.

[116] G. Z. Mou, *BORG: A CORBA-based high-performance programming system*, Tech. Rep. orbos/99-07-20, Applied Physics Laboratory, John Hopkins University, 1999. Response to the Aggredgated Computing RFI. http://www.omg.org/cgi-bin/doc?orbos/99-07-20.

[117] *The MPC project*, Oct. 1999. http://mpc.lip6.fr/.

[118] MPI Forum, *MPI: A message passing interface standard*, tech. rep., University of Tennessee, 1994. http://www.mpi-forum.org/docs/docs.html.

[119] ——, *MPI-2: Extensions to the message passing interface*, tech. rep., Message Passing Interface Forum, 1997. http://www.mpi-forum.org/docs/docs.html.

[120] *MPI Forum home page*, Oct. 1999. http://www.mpi-forum.org/.

[121] *Matrix template library (MTL) home page*, Aug. 2000. http://www.lsc.nd.edu/research/mtl/.

[122] P.-A. Muller and N. Gaertner, *Modélisation objet avec UML*, Eyrolles, 2nd ed., Mar. 2000.

[123] D. R. Musser and A. A. Stepanov, *Algorithm-oriented generic libraries*, Software–Practice & Experience, 24 (1994), pp. 623–642. http://www.cs.rpi.edu/projects/STL/htdocs/node54.html#stlrefs.

[124] N. Myers, *Traits: a new and useful template technique*, C++ Report, (1995).

[125] ——, *Gnarly new C++ language features (that you can finally use)*, 1997. http://www.cantrip.org/gnarly.html.

[126] *Myrinet*, Oct. 1999. http://www.myri.com/myrinet/overview/index.html.

[127] R. Namyst, *PM^2 : un environnement pour une conception portable et une exécution efficace des applications paralèles irrégulière*, PhD thesis, LIFL, Université de Lille I, Dec. 1996. http://www.ens-lyon.fr/~rnamyst/these/main.ps.gz.

[128] C. D. Norton, *Object-Oriented Programming Paradigms in Scientific Computing*, PhD thesis, Rensselaer Polytechnic Institute Troy, New York, Aug. 1996. ftp://ftp.cs.rpi.edu/pub/nortonc.

[129] E. Noulard and N. Emad, *Object oriented design for reusable parallel linear algebra software*, in Euro-Par'99, Parallel Processing, P. Amestoy, P. Berger, M. Daydé, I. Duff, V. Frayssé, L. Giraud, and D. Ruiz, eds., no. 1685 in Lecture Notes in Computer Science, Springer Verlag, Aug. 1999, pp. 1385–1392. Voir aussi Rapport Technique PRiSM 1999/004. http://www.enseeiht.fr/europar99.

[130] ——, *A key for reusable parallel linear algebra software*, Parallel Computing, (2000). accepted for publication.

[131] E. Noulard, N. Emad, and L. Flandrin, *Calcul numérique parallèle et technologies objet*, Tech. Rep. Rapport PRISM 1998/003, ADULIS/PRiSM, July 1997. Révision du 30/01/98. http://www.prism.uvsq.fr/rapports/1998/abstract_1998_3.html.

[132] Object Management Group, *The common object request broker: Architecture and specification*, formal 99-10-07, OMG, Oct. 1999. Revision 2.3.1. http://www.omg.org/corba/corbaiiop.html.

[133] ——, *Data parallel application support for CORBA*, RFP 00-03-17, OMG, 2000. http://www.omg.org/cgi-bin/doc?orbos/00-03-17.

[134] ——, *OMG Unified Modeling Language specification*, Tech. Rep. formal/00-03-01, Object Management Group, Mar. 2000. version 1.3. http://www.omg.org/cgi-bin/doc?formal/00-03-01.

[135] *Obliq home page*. http://www.luca.demon.co.uk/Obliq/Obliq.html.

[136] *ODM Overview*, June 2000. http://www.synquiry.com/Company/st-ODM.html.

[137] *Open Implementation (OI) home page*, Oct. 1999. http://www.parc.xerox.com/spl/projects/oi/.

[138] D. P. O'Leary, *The block conjugate gradient algorithm and related methods*, Linear Algebra and Its Applications, 29 (1980), pp. 293–322. http://www.cs.umd.edu/ oleary/.

[139] *Object management group (OMG) home page*. http://www.omg.org/.

[140] *Object-oriented frequently asked questions*, Fév. 2000. http://www.cyberdyne-object-sys.com/oofaq2/.

[141] *OpenMP home page*, Oct. 1999. http://www.openmp.org.

[142] OpenMP ARB, *OpenMP: A proposed industry standard API for shared memory programming*. A White Paper, Oct. 1997. http://www.openmp.org.

[143] ——, *OpenMP Fortran application program interface version 1.0*, tech. rep., OpenMP ARB, Oct. 1997. http://www.openmp.org/.

[144] ——, *OpenMP C and C++ application program interface version 1.0*, Tech. Rep. 004 2229 001, OpenMP ARB, Oct. 1998. http://www.openmp.org/.

[145] ORBOS Platform Task Force, *Supporting aggregate computing in CORBA*, RFI 99-01-04, OMG, 1999. http://www.omg.org/cgi-bin/doc?orbos/99-01-04.

[146] R. Orfali, D. Harkey, and J. Edwards, *The Essential Distributed Objects Survival Guide*, John Wiley & Sons Inc., 1996.

[147] J.-L. Pacherie, *Système de Motifs pour l'Expression et la Parallélisation des Traitements des Collections dans un Contexte de Génie Logiciel*, PhD thesis, Université de Rennes 1, Rennes, France, Dec. 1997. Num. Ordre 1889. http://www.irisa.fr.

[148] M. Papathomas, *Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming*, PhD thesis, Faculté des Sciences, Université de Genève, Jan. 1992. Thèse No 2522. http://cuiwww.unige.ch/OSG/publications/OO-articles.

[149] *Para++ home page*. http://www.loria.fr/projets/para++/.

[150] *Paralline home page*, Oct. 1999. http://www.paralline.com.

[151] B. N. Parlett, *The Symmetric Eigenvalue Problem*, Classics In Applied Mathematics, SIAM Edition, 1998.

[152] R. Pawlak, *Metaobject protocols for distributed programming*, rapport de DEA, Laboratoire CNAM-CEDRIC, Sept. 1998. http://cedric.cnam.fr/personne/pawlak.

[153] *pC++/Sage++ home page*, Oct. 1999. http://www.extreme.indiana.edu/sage/index.html.

[154] G.-R. Perrin and A. Darte, eds., *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, no. 1132 in Lecture Notes in Computer Science, Springer Verlag, June 1996. School on Data Parallelism, Les Ménuires, France, March 25–28.

[155] *PETE home page*, Oct. 1999. PETE, a Portable Expression Template Engine. http://www.acl.lanl.gov/pete.

[156] T. P. Pierryck Beaugendre and C. René, *Cobra: a CORBA-compliant programming environment for high-performance computing*, Publication Interne PI 1141, IRISA, 1997. http://www.irisa.fr/EXTERNE/bibli/pi/1141/1141.html.

[157] T. Priol and C. René, *Cobra: a CORBA-compliant programming environment for high-performance computing*, in Proceedings of Euro-Par'98, Sept. 1998, pp. 1114–1122. http://www.irisa.fr/EuroTools/Documents/Documents.texte.html.

[158] *PVM home page*, Oct. 1999. http://www.epm.ornl.gov/pvm/.

[159] S. Raina, *Virtual shared memory: A survey of techniques and systems*, Tech. Rep. CS-TR-92-36, University of Bristol, Dec. 1992. http://www.cs.bris.ac.uk/Tools/Reports/Abstracts/1992-raina.html.

[160] D. Reese, E. Luke, G. Henley, N. Doss, S. Korlakunta, and G. Smith, *Object-oriented Fortran*, Tech. Rep. MSSU-EIRS-ERC-94-1, Mississipi State University, Aug. 1994.

[161] D. Remy and F. Delaplace, *Eléments de conception d'une stl data-parallèle pour le traitement des matrices creuses*, in RENPAR'11, June 1999. http://www.irisa.fr/renpar11/.

[162] P. W. Rijks, J. M. Squyres, and A. Lumsdaine, *Performance benchmarking of object-oriented MPI (OOMPI) version 1.0.2g*, Tech. Rep. TR-99-14, University of Notre Dame – Department of Computer Science, 1999.

[163] A. D. Robison, *C++ gets faster for scientific computing*, Computer in Physics, 10 (1996), pp. 458–462. http://www.kai.com/publications/comp_phys/.

[164] A. Ruhe, *Implementation aspects of band Lanczos algorithms for computation of eigenvalues of large sparse symmetric matrices*, Mathematics of Computations, 33 (1979), pp. 680–687.

[165] Y. Saad, *Numerical Methods For Large Eigenvalue Problems*, Algorithms and Architectures for Advanced Scientific Computing, Manchester University Press, 1992. ftp://ftp.cs.umn.edu/dept/users/saad/reports/FILES/EIGBOOK.tar.gz.

[166] ——, *A flexible inner-outer preconditioned GMRES algorithm*, SIAM Journal on Scientific and Statistical Computing, 4 (1993). Également Tech. Rep. umsi-92-279. ftp://ftp.cs.umn.edu/dept/users/saad/reports/FILES/umsi-91-279.ps.gz.

[167] ——, *SPARSKIT: A basic tool kit for sparse matrix computations*, version 2 ed., June 1994. http://www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html.

[168] ——, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996. ftp://ftp.cs.umn.edu/dept/users/saad/reports/FILES/ITBOOK.tar.gz.

[169] Y. Saad and M. Schultz, *GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing, (1986), pp. 856–869.

[170] M. Sadkane, *A block Arnoldi-Chebyshev method for computing the leading eigenpair of large sparse unsymmetric matrices*, Num. Math, 64 (1993), pp. 181–193.

[171] D. Sagnol, *Conception et Optimisation de programmes à objets parallèles, répartis et multi-threadés*, PhD thesis, Univsersité de Nice – Sophia Antipolis, June 2000.

[172] *SCALI home page*, Oct. 1999. `http://www.scali.com/`.

[173] R. S. Schreiber, *An introduction to HPF*, in Perrin and Darte [154], pp. 27–44. School on Data Parallelism, Les Ménuires, France, March 25–28.

[174] *SCIzzL home page*, Oct. 1999. `http://www.SCIzzL.com/`.

[175] A. Sedrakian, *Etude de la méthode multi-ERAM*, rapport de stage de dea, Laboratoire PRiSM, Sept. 2000. En préparation.

[176] SEI at Carnegie Mellon, *Domain engineering: A model-based approach*, June 2000. `http://www.sei.cmu.edu/domain-engineering/domain_engineering.html`.

[177] E. Seidewitz, *Genericity versus inheritance reconsidered: self-reference using generics*, in OOPSLA'94, 1994, pp. 153–163.

[178] J. G. Siek, A. Lumsdaine, and L. Q. Lee, *Generic programming for high performance numerical linear algebra*, in SIAM Workshop on Interoperable OO Sci. Computing, 1998. `http://www.lsc.nd.edu/research/mtl/publications.php3`.

[179] V. Simoncini and E. Gallopoulos, *Convergence properties of block GMRES and matrix polynomials*, Linear Algebra and Its Applications, (1994). Replace CSRD Report No 1470. `http://www.csrd.uiuc.edu/tech_reports.html`.

[180] ——, *An iterative method for nonsymmetric systems with multiple right-hand sides*, SIAM J. Sci. Comput., 16 (1995), pp. 917–933. Replace CSRD Report No 1242. `http://www.csrd.uiuc.edu/tech_reports.html`.

[181] ——, *A hybrid block GMRES method for nonsymmetric systems with multiple right-hand sides*, J. Comput. Appl. Math., 66 (1996). Replace CSRD Report No 1378. `http://www.csrd.uiuc.edu/tech_reports.html`.

[182] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl, *Questions and answers about BSP*, tech. rep., Oxford University Computing Laboratory, 1996. `http://www.cs.queensu.ca/home/skill/QandA.ps`.

[183] B. Smolinski, S. Kohn, N. Elliott, and N. Dykman, *Language interoperability for high-performance parallel scientific component*, in ISCOPE'99, Lecture Notes in Computer Science, Dec. 1999. `http://www.acl.lanl.gov/iscope99/`.

[184] D. C. Sorensen, *Implicit application of polynomial filters in a k-step Arnoldi method*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 357–385.

[185] J. M. Squyres, B. C. McCandless, and A. Lumsdaine, *Object Oriented MPI (OOMPI): A C++ Class Library for MPI*, Dept. of Computer Science and Engineering, University of Notre Dame, version 1.0.2f ed., June 1998. `http://www.lsc.nd.edu/research/oompi/`.

[186] A. Stepanov and M. Lee, *The standard template library*, Tech. Rep. HPL-95-11, Hewlett-Packard Laboratories, Nov. 1995. `http://www.hpl.hp.com/techreports/95/HPL-95-11.html`.

[187] G. W. Stewart, *Matrix Algorithms I: Basic Decompositions*, SIAM, 1998. `http://www.cs.umd.edu/~stewart/`.

[188] ——, *Matrix Algorithms II: Eigensystems*, SIAM, 1999, Preliminiray draft online version ftp://thales.cs.umd.edu/pub/survey. http://www.cs.umd.edu/~stewart/.

[189] *STLport home page*. http://www.STLport.org/.

[190] B. Stroustrup, *What is "Object-Oriented Programming"*, in 1st European Software Festival, 1991. revised, version. http://www.research.att.com/~bs/.

[191] ——, *Why C++ is not just an Object-Oriented Programming Language*, OOPS Messenger, 6 (1995), pp. 1–13. http://www.research.att.com/~bs/.

[192] V. S. Sunderam, *PVM: A framework for parallel distributed computing*, Concurrency: Practice and Experience, 2 (1990), pp. 315–339. http://www.netlib.org/ncwn/pvmsystem.ps.

[193] A. S. The Europa Working Group on Parallel C++, *Europa parallel C++, version 2.1*, tech. rep., The Europa Working Group on Parallel C++, July 1997.

[194] *Top 500 supercomputers sites*. http://www.top500.org/.

[195] *UML resources at rational software*, Mar. 2000. http://www.rational.com/uml/resources/index.jtmpl.

[196] *Infos en français sur UML*, Mar. 2000. http://www.essaim.univ-mulhouse.fr/uml/.

[197] A. J. van der Steen and J. J. Dongarra, *Overview of recent supercomputers*, tech. rep., NCF Utrecht University, Feb. 2000. Ce rapport est mis à jour périodiquement. http://www.phys.uu.nl/~steen/web00/overview00.html.

[198] A. van Deursen, P. Klint, and J. Visser, *Domain-specific languages – an annotated bibliography*, tech. rep., Centrum voor Wiskunde en Informatica, 2000. submitted to ACM SIGPLAN Notices. http://www.cwi.nl/~arie/papers/dslbib.pdf.

[199] T. Veldhuizen, *Expression templates*, C++ Report, 7 (1995), pp. 26–31.

[200] T. L. Veldhuizen, *Scientific computing: C++ versus Fortran*, Dr Dobbs Journal, (1997). http://extreme.indiana.edu/~tveldhui/papers/DrDobbs2/drdobbs2.html.

[201] ——, *Arrays in Blitz++*, in Proceedings of ISCOPE'97, Lecture Notes in Computer Science, 1998. http://oonumerics.org/blitz/.

[202] ——, *Blitz++*, Oct. 1999. http://oonumerics.org/blitz/.

[203] ——, *Techniques for scientific C++*, Aug. 1999. http://www.extreme.indiana.edu/~tveldhui/papers/techniques/.

[204] T. L. Veldhuizen and M. E. Jernigan, *Will C++ be faster than Fortran?*, in Proceedings of ISCOPE'97, no. 1343 in Lecture Notes in Computer Science, Springer Verlag, 1997. http://extreme.indiana.edu/~tveldhui/papers.

[205] B. Vital, *Étude de quelques méthodes de résolution de problèmes linéaires de grande taille sur multiprocesseur*, PhD thesis, Université de Rennes 1, Rennes, France, Nov. 1990.

[206] E. A. West and A. S. Grimshaw, *Braid: Integrating task and data parallelism*, tech. rep., Department of Computer Science, University of Virginia, 1994. ftp://ftp.cs.virginia.edu/pub/techreports/CS-94-45.ps.Z.

[207] R. Whaley and J. Dongarra, *Automatically Tuned Linear Algebra Software (ATLAS)*, in Proceedings of Supercomputing 98, Oct. 1998. http://www.netlib.org/atlas/.

[208] J. H. Wilkinson, ed., *The Algebraic Eigenvalue Problem*, Monographs on Numerical Analysis, Oxford University Press, 1965, 1992 reprint.

[209] *World's most powerful computing sites*, Oct. 1999. `http://www.gapcon.com/listg.html`.

[210] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*, ACM Press, 1990.

Résumé

Programmation parallèle orientée objet et réutilisabilité appliquée à l'algèbre linéaire

L'objectif de cette thèse est d'examiner comment les technologies orientées-objet peuvent apporter aux applications scientifiques tout ce qu'elles ont apporté dans la programmation des machines séquentielles : une meilleure réutilisabilité et pérennité des codes, des démarches méthodologiques de conception et de réalisation claires... La contrainte du calcul scientifique parallèle de ne pas sacrifier les performances devant être respectée.

Après une revue des moyens de programmation parallèle et des concepts objets, la conception et la réalisation d'une bibliothèque parallèle d'algèbre linéaire orientée-objet sont présentées. Nous étudions deux moyens de programmation parallèle, le premier, C++//, est un LAO parallèle à objets actifs dérivé de C++, le second est l'utilisation de MPI au travers d'une surcouche objet minimale. Ces deux approches objets posent des problèmes soit de performances soit de réutilisabilité séquentielle/parallèle qui sont présentés et résolus.

Nous proposons notamment un mécanisme simple de partage en lecture pour les modèles à objets actifs, en montrant son utilité en terme de performances de nos applications. Suite à la seconde approche nous définissons les notions de formes de matrices et de matrices avec forme qui permettent d'atteindre nos objectifs de réutilisabilité séquentielle/parallèle.

Au final, la conception et la réalisation permettent d'instancier, à partir du même code [séquentiel] d'algèbre linéaire, une version séquentielle et parallèle offrant des performances satisfaisantes.

Mots-clés: POO Parallèle, Parallélisme, Algèbre Linéaire (Méthodes de Krylov), Méthodologie de Conception, Patrons de Conception, Réutilisabilité, MPI, C++.

Abstract

Parallel Object-Oriented Programming & Reuse applied to Numerical Linear Algebra

The primary thesis topic is to study how object-oriented technologies may bring to scientific applications all the benefits they bring to traditionnal sequential application like: better reuse and longer software life, design methodology, cleaner realization... The main scientific computing constraint, which is performance is kept in mind.

After a survey of parallel and object-oriented programming concepts, the design and realization of a parallel linear algebra library is presented. We study two different object-oriented approaches. The first use C++// language, a concurrent object-oriented language derived from C++, which implements an active object model. The second is the use of MPI through a minimal object-oriented layer realized in C++. Both of them, raise problems, regarding performance and/or sequential/parallel reuse. We solve these problems.

We propose a shared on read mecanism applicable to any active object model, showing its efficiency for our applications. The second approach leads us to the new concepts of matrix shape and shaped matrices which enable us to reach our sequential/parallel reuse objectives.

In the end, the implemented design enable us to instantiate both a sequential and parallel version of our linear algebra algorithms from the very same piece of code. The resulting parallel and sequential applications both exhibit satisfying performances.

Keywords: Parallel OOP, Parallelism, Linear Algebra (Krylov Methods), Design Method, Design Pattern, Reuse, MPI, C++.