N° d'ordre : 2994

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **CIROU Bertrand**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Expression d'algorithmes scientifiques et gestion performante de leur parallélisme avec PRFX pour les grappes de SMP

Soutenue le : 30/06/2005

Après avis des rapporteurs :

MÉHAUT Jean-François ..	Professeur
PAZAT Jean-Louis	Professeur

Devant la commission d'examen composée de :

COULAUD Olivier	Directeur de recherche	Examinateur
COUNILH Marie-Christine	Maître de Conférence	Co-directrice de thèse
MÉHAUT Jean-François ..	Professeur	Rapporteur
NAMYST Raymond	Professeur	Examinateur
PAZAT Jean-Louis	Professeur	Rapporteur
ROMAN Jean	Professeur	Directeur de thèse

– 2005 –

Expression d'algorithmes scientifiques et gestion performante de leur parallélisme avec PRFX pour les grappes de SMP

Résumé : Nous proposons un mode d'expression destiné à l'implémentation performante des algorithmes parallèles scientifiques, notamment ceux irréguliers, sur des machines à mémoires distribuées hétérogènes telles que les grappes de SMP. Ce mode d'expression parallèle est basé sur le langage C et des appels à la bibliothèque PRFX. Le découpage de l'algorithme en tâches est obligatoire mais les synchronisations et communications entre ces tâches sont implicites. L'identification des tâches et de leurs caractéristiques (identificateur de la fonction cible, paramètres, coût) doit être effectuée via l'interface de la bibliothèque PRFX. Cette interface doit également être utilisée pour allouer et libérer les données ainsi que pour déclarer les accès des tâches aux données. La structuration des données (référencement par pointeurs) est aisée car PRFX fournit un espace d'adressage global. D'autres fonctionnalités optionnelles de la bibliothèque PRFX permettent d'exprimer des optimisations génériques et lisibles. En effet, il est possible d'appliquer dans le cas des programmes ayant des propriétés prévisibles (i.e. structurés par un jeu de données d'entrée), des heuristiques d'ordonnancement globales paramétrées par des modèles de coût et guidées par le programmeur (e.g. spécification d'un placement initial).

Le support utilise une iso-mémoire assurant la validité des pointeurs en mémoire distribuée. Il adopte un fonctionnement de type inspection/exécution. L'inspecteur modélise l'exécution de l'algorithme parallèle par un DAG de tâches. Ce type de fonctionnement et ce modèle sont adaptés à l'exploitation performante des programmes irréguliers prévisibles. Les tâches et leurs accès aux données sont capturés dans ce DAG lors de l'inspection statique consistant en une pré-exécution partielle du programme. Les communications et synchronisations sont déduites de l'analyse des accès aux données faite par l'inspecteur. Ensuite, un ordonnanceur séquentiel applique l'heuristique d'ordonnancement intégrant les optimisations explicitées par le programmeur. Enfin, un exécuteur parallèle exécute le DAG de tâches en respectant l'ordonnancement des tâches et des communications. Il exploite la performance des communications unilatérales pour les communications inter-nœuds et tire profit de la mémoire partagée d'un nœud avec des *threads*.

Dans ce cadre, nous avons validé notre approche par l'implémentation d'algorithmes scientifiques (résolution de l'équation de Laplace, factorisation LU de matrices pleines et factorisation de matrices creuses par la méthode de Cholesky) et par des expérimentations en vraie grandeur sur des grappes de SMP IBM NH2 et p690+.

Discipline : Informatique

Mots-Clefs : iso-mémoire, synchronisations implicites, algorithmes irréguliers prévisibles, DAG de tâches, grappes de SMP, inspection / exécution, ordonnancement, *threads*, communications unilatérales.

LaBRI,
Université Bordeaux 1,
351 cours de la Libération,
33405 Talence Cedex (FRANCE).

Implementing scientific algorithms and managing their parallelism efficiently with PRFX onto SMP clusters

Abstract : We present a programing model aimed to efficiently implement parallel scientific algorithms, in particular irregular ones, onto distributed memory machines like clusters of SMP nodes. This programming model is based on the C language and calls to our PRFX library. The programmer must split its algorithm into tasks nervertheless synchronizations and communications between tasks are implicit. These tasks operate on data that are dynamically allocated in an iso-memory and may access sub-data through the declaration of their geometry with stencils. Data structures (with pointers) may be easily implemented thanks to the global address space provided. We also provide functionalities to take into account the static properties of irregular algorithms (with structuring data) like scheduling heuristics using cost functions and an initial task mapping.

Our parallel runtime uses an iso-memory keeping pointer validity through migration in the distributed memory. It adopts an inspection/execution scheme. Our inspector modelizes the parallel execution by a task DAG which is well suited to predicitble irregular algorithms. Tasks and their accesses are statically analyzed by the inspector. This allows to build data dependencies thanks to a partial pre-execution of the code, and to produce a task DAG with all necessary information for the static scheduler and the parallel executor. Then, a static sequential scheduler applies a heuristic taking into account the optimizations specified by the programmer. Finally, the parallel executor executes the tasks and the communications of the DAG with respect to this schedule. Our parallel executor uses POSIX threads with one-sided communications and works on shared and distributed memory machines.

We validate its performances by experimental results for a Laplace equation solver, a LU factorization of dense matrices and a sparse Cholesky factorization algorithm on SMP IBM® cluster with NH2 and p690+ nodes.

Discipline : Computer-Science

Keywords : iso-memory, implicit synchronisations, predictible irregular algorithms, task DAG, SMP cluster, inspector/executor, scheduling, threads, one-sided communications

LaBRI,
Université Bordeaux 1,
351 cours de la Libération,
33405 Talence Cedex (FRANCE).

Remerciements

Ces remerciements s'adressent à celles et ceux qui ont contribué, d'un point de vue logistique, affectif et recherche, à mes études de doctorat.

Pour la logistique, je remercie majoritairement l'université de Bordeaux 1 et le LaBRI pour ses infra-structures et ses personnels qui m'ont permis de travailler dans les meilleures conditions possibles.

Concernant le côté affectif, je souhaite remercier ma famille et les gars qui me connaissent. En particulier ceux qui tombent des avions et/ou font de la musique et/ou trinquent allègrement. Les filles ne sont pas en restes malgré un faible effectif, je pense tout particulièrement à ma Lady Emmanuelle. Enfin, mes recherches ont été investies par des contributeurs, notamment les membres de mon jury de thèse et plus fortement par mes encadrants.

Table des matières

Table des figures

Introduction

Contexte de travail

Le parallélisme est une branche de l'informatique qui traite de l'exploitation coordonnée de plusieurs ressources de calcul (processeurs), de communication (cartes et liens réseaux) et de stockage d'informations (mémoires). Les machines parallèles sont constituées de machines élémentaires interconnectées regroupant ces trois ressources. Cette interconnexion forme une grappe où chaque nœud représente une machine élémentaire.

Les grappes dont les nœuds sont des machines multi-processeurs de type SMP (Symmetric Multi Processor) offrent de bons ratios performances / prix. De fait, elles sont très répandues et se déclinent actuellement en versions allant de la grappe de stations de travail bi-processeurs à la grande grappe dédiée constituée de nœuds à 32 processeurs.

Leurs performances proviennent d'une technologie hétérogène, avec une mémoire partagée en intra-nœud et un réseau rapide en extra-nœud, permettant d'atténuer les écarts de puissance entre le processeur et les périphériques (réseau, mémoire).

Concernant les technologies réseaux, ces dernières sont sujettes à la contention lorsque, par exemple, tous les processeurs de deux nœuds différents tentent de communiquer en même temps.

La technologie de la mémoire partagée est moins sensible à la contention grâce à des interconnexions de type *crossbar*. Cependant, la mémoire partagée est organisée avec une hiérarchie de caches pour mieux alimenter le processeur. Les tailles différentes de ces caches provoquent des effets de seuil sur les coûts d'accès aux données. Ces écarts de coûts sont suffisamment importants pour que l'architecture des processeurs les exploite via une technologie multi-flots (*SMT Simultaneous Multi-Threading* [10]). Les *threads* permettent d'exploiter la mémoire partagée des nœuds et les optimisations matérielles du processeur.

Des applications parallèles performantes ont été implémentées sur ces machines, en particulier pour les besoins toujours grandissant de la communauté du calcul scientifique. Ces implémentations sont effectuées avec des modes d'expression parallèles dont les compilateurs et/ou supports conduisent à des exécutions parallèles. Selon que les modes d'expression sont de bas ou de haut niveau, tout ou partie de la gestion du parallélisme doit être explicitée par le programmeur. Cette gestion comporte notamment la cohérence, la synchronisation des accès aux données et des optimisations par exemple pour l'ordonnancement des calculs. La mise en œuvre de la cohérence n'est nécessaire qu'en présence d'une hiérarchie mémoire ou d'une mémoire distribuée. Elle

consiste à transférer les données d'une mémoire à une autre. La synchronisation des accès aux données consiste à garantir la validité de l'exécution du programme.

Les applications de calcul scientifique présentent un parallélisme couplé et structuré qui rend intéressant la mise en place d'approches exploitant ces caractéristiques. Dans le cas des algorithmes irréguliers [31] et encore plus sur des machines hétérogènes de type grappe de SMP, la performance est souvent liée au savoir-faire du programmeur vis à vis de la machine et de son algorithme, et à la qualité de l'ordonnancement global de l'ensemble des calculs et des communications. L'irrégularité repose en particulier sur des schémas d'accès irréguliers aux données ou sur des schémas de synchronisation irréguliers. Nous considérons qu'un schéma de synchronisations est irrégulier dès lors qu'il n'est pas répétitif, ou n'est pas du type 1 vers N, N vers 1 ou N vers N.

La plupart des algorithmes scientifiques, y compris ceux irréguliers, ont un déroulement et donc un parallélisme conditionné par un jeu de **données d'entrée** qualifiées de **structurantes**. Ces applications sont dites **prévisibles** lorsque leur flot de contrôle dépend uniquement de ce jeu de données d'entrée [28]. Si le flot de contrôle peut être prédit (calculé) avant l'exécution, alors cette information est exploitable par une inspection statique [45] ; cette approche suppose généralement que les machines sont dédiées et infaillibles. Sinon, seules des approches dynamiques plus générales (e.g. ordonnancement local à la volée ou fonctionnement *data-flow* avec vol de travail [24]) sont applicables.

L'équipe ALienor/ScAlAplix du LaBRI/INRIA Futurs à Bordeaux s'intéresse à la conception et à la mise en œuvre efficace sur de grandes grappes de SMP d'algorithmes parallèles irréguliers, en particulier sur des problèmes d'algèbre linéaire creuse [14, 36]. Cette thèse s'inscrit dans ce contexte.

Problématique

Dans le contexte précédemment posé, nous considérons le problème de la définition d'un mode d'expression pour l'implémentation *aisée* d'algorithmes parallèles irréguliers et qui autorise l'exploitation performante de leurs caractéristiques prévisibles sur des machines hétérogènes telles que les grappes de SMP.

Des langages séquentiels couplés à des bibliothèques parallèles (e.g ScaLAPack [19], PaStiX [36]) proposent des briques algorithmiques typiques de noyaux d'applications scientifiques. Par contre, lorsqu'un nouvel algorithme (notamment irrégulier) ou qu'un enchaînement de noyaux algorithmiques doit être parallélisé et de surcroît optimisé pour une grappe de SMP, alors, comme nous le verrons au chapitre 1, l'implémentation effective de cet algorithme n'est généralement possible et performante qu'avec un langage de bas niveau (langage séquentiel avec MPI [29]). De plus, les principales approches compilées telles que HPF [37, 65] (orienté données) ou OpenMP [49, 51] (orienté tâches implicites) ont montré leurs limites dans le cadre précédemment posé.

De cela, il ressort qu'une difficulté est d'adopter une répartition réaliste des rôles entre le support d'exécution, le compilateur et le programmeur. Cette répartition doit être telle que le

programmeur puisse exprimer des algorithmes notamment irréguliers, des optimisations génériques, et enfin avoir l'assurance que ces deux informations sont effectivement exploitées par un support d'exécution optimisé. Nous dégageons ci-après trois points importants de cette problématique.

Expression aisée et fine des algorithmes parallèles. Nous considérons que l'utilisation d'un langage parallèle est aisée lorsqu'il reprend des caractéristiques des langages séquentiels telles que la vision d'une mémoire unique pour les données, la conservation de la sémantique séquentielle du programme et l'absence de synchronisations ou de transferts (cohérence) de données explicites.

L'aisance d'utilisation d'un mode d'expression varie donc selon que la gestion parallélisme est automatiquement effectuée par un support (matériel ou logiciel) ou explicitée par le programmeur. De plus, l'implémentation d'une structure de données irrégulière est rendue aisée si le mode d'expression offre des possibilités de référencements dans un système commun de coordonnées (e.g. indices de tableaux) ou un espace d'adressage global.

La finesse d'expression repose sur la capacité de l'interface du mode d'expression à capturer suffisamment d'informations concernant le parallélisme de l'algorithme pour permettre la mise en œuvre d'une gestion de la cohérence comportant des transferts de données irrégulières et/ou la gestion de schémas de synchronisation irréguliers.

Comme nous le verrons au chapitre 1, l'utilisation des modes d'expression de bas niveau est ardue mais leur finesse est suffisante pour exprimer les schémas d'accès aux données ou les schémas de synchronisation des algorithmes parallèles irréguliers. En revanche, les langages de haut niveau sont d'utilisation aisée grâce à des primitives de construction du parallélisme ou de déclaration des données, mais ces dernières ne permettent pas d'exprimer finement les irrégularités souhaitées, et limitent souvent le programmeur dans son désir d'exprimer sa connaissance du problème.

Description lisible d'optimisations génériques. Nous appelons optimisations génériques pour un algorithme donné, des optimisations de stockage des structures de données, de distribution de données, ou d'ordonnancement de l'exécution exploitant le caractère prévisible des programmes et les caractéristiques hétérogènes de la machine. Cela suppose, par exemple, l'existence d'un modèle pour l'ordonnancement (e.g. DAG de tâches) et pour la configuration d'exécution sur cette machine (e.g. graphe de déploiement et d'interconnexion).

La généricité des optimisations, tout en étant lisibles, est un problème lié à la complexité des machines parallèles et à l'insuffisance des modes d'expression du parallélisme. Généralement, le programmeur injecte son savoir-faire "en dur" dans les modes d'expression de bas niveau et ceci spécifiquement pour un type de machine. Inversement, pour les langages de haut niveau, le savoir-faire du programmeur est "bridé" car la gestion du parallélisme est automatique, et de fait celui-ci ne dispose pas d'interface pour la contrôler. Les performances sont alors souvent décevantes ce qui induit une intervention non-standard du programmeur (structuration du code spécifique à un compilateur ou à une machine, utilisation directe du support d'exécution).

Exécution performante exploitant les grappes de SMP. L'exécution performante est l'étape finale d'un processus débuté lors de l'implémentation de l'algorithme. En supposant que la chaîne de traitements intervenant entre le support d'exécution et le programmeur est suffisamment optimisée, le problème est donc d'implémenter un support d'exécution performant pour les grappes de SMP.

La difficulté pour ce support est d'avoir le contrôle le plus fin possible des processeurs, de la mémoire et du réseau pour être capable de réaliser efficacement l'exécution de l'algorithme et des optimisations exprimées par le programmeur. Cela suppose l'utilisation d'un faible nombre de couches logicielles (problème de latence) tout en réalisant les services parfois complexes fournis par un langage de haut niveau. De fait, les choix de technologie pour ces couches logicielles doivent être en accord exact avec les modèles en amont (sous-jacents aux étapes de compilation, d'inspection et de génération de code) et avec ceux de la machine pour éviter des étapes de retranscription.

Un autre problème concerne l'implémentation conjointe de fonctionnalités pour les programmes ayant des parties prévisibles et imprévisibles. Cela suppose de conserver les informations produites à l'inspection ou à la compilation jusqu'à l'exécution afin de pouvoir opérer une inspection à la volée, des réordonnancements ou des migrations de tâches.

Objectifs et démarche

Face à l'inexistence d'un mode d'expression répondant de façon satisfaisante aux trois problèmes précédemment posés, nous souhaitons créer un nouveau mode d'expression parallèle nommé PRFX et implémenter l'environnement logiciel ad hoc pour exécuter des programmes PRFX. Notre objectif est d'offrir un mode d'expression et un support capable d'assurer l'exécution performante de ces programmes et notamment ceux irréguliers prévisibles, en faisant le moins de compromis possible sur l'aisance d'utilisation. Pour ce faire, nous utiliserons trois éléments majeurs qui contribueront à la mise en place d'une solution satisfaisante pour les trois problèmes cités.

Le premier élément est **une iso-mémoire au niveau du support** [4] (décrite à la section 3.1), ce qui par rapport à une SDSM (Software Distributed Shared Memory) autorise plus de performance tout en permettant au mode d'expression de proposer une vision unique de la mémoire et la possibilité de construire en mémoire distribuée des structures de données irrégulières avec un stockage optimisé.

Le deuxième élément est **une modélisation fine de l'exécution du programme par un DAG de tâches complet** (décrit à la section 2.3) et nous proposerons les outils pour le créer et le manipuler. Ce DAG sera utilisé tout au long de la chaîne de traitements du programme PRFX et y compris lors de l'exécution parallèle. De plus, avec le support iso-mémoire, nous pourrons également préempter et migrer les tâches du DAG (cf. les perspectives de ce travail au Chapitre 5).

Le troisième élément est l'adoption d'une approche de type **inspection statique / exécution** sur un code où le programmeur découpera explicitement son programme en tâches et décrira leurs accès aux données. Avec ces indications, l'inspecteur pourra modéliser finement les irrégularités et capturer le savoir-faire du programmeur ainsi que les propriétés statiques du programme. Cet

outil d'inspection déchargera le programmeur de la description des synchronisations et/ou des communications entre les tâches et conservera la sémantique séquentielle du programme lors de la modélisation de ce dernier par un DAG de tâches complet.

Nous proposerons de plus des **optimisations génériques** via l'application d'heuristiques d'ordonnancement [34, 35] pour les tâches et les communications de ce DAG. Ces heuristiques seront adaptées afin d'intégrer le savoir-faire du programmeur (placement initial et utilisation de fonctions de coût) et les caractéristiques de l'hétérogénéité de la machine.

Enfin, nous développerons un **exécuteur parallèle** proposant des **optimisations pour les grappes de SMP** et capable de respecter l'ordonnancement des tâches et des communications. Il fera usage de *threads* pour bénéficier d'un accès performant aux données à l'intérieur d'un nœud SMP et utilisera un modèle de **communications unilatérales** qui autorise intrinsèquement de meilleures performances en terme de latence et de bande passante qu'un modèle par passage de messages.

Nous re-écrirons avec PRFX des applications de calcul scientifique classiques. Ceci nous permettra d'affiner l'API de PRFX et de la tester pour l'expression d'applications irrégulières et de leurs optimisations. Nous évaluerons les performances absolues des exécutions de ces applications implémentées avec PRFX et nous les comparerons avec des exécutions de références obtenues avec des bibliothèques dédiées au calcul scientifique.

Pour effectuer des mesures et contrôler la qualité du DAG et de l'exécution, nous développerons des modules de production de traces et de fichiers pour la visualisation de ce DAG de tâches.

Le document est découpé en cinq chapitres et suivra le cheminement indiqué ci-après. Nous définissons plus en détail au chapitre 1 les notions d'algorithmes irréguliers et/ou prévisibles. Nous étudions des modes d'expression sélectionnés pour leur représentativité, notamment vis à vis de l'irrégularité des algorithmes et de l'hétérogénéité des machines. Nous expliquons pourquoi les caractéristiques prévisibles des machines et des algorithmes doivent être prises en compte au plus tôt par les modes d'expression pour obtenir de meilleures performances. Nous expliquons également pourquoi le modèle du DAG de tâches et une approche inspection/exécution s'imposent quand cette prise en compte est possible et en quoi l'existant ne répond seulement qu'en partie aux problèmes formulés.

De ces constats, nous proposons notre solution au chapitre 2. Ce chapitre est une description des modèles (programmation, DAG complet, machines, coûts, exécution) et du fonctionnement de l'environnement PRFX proposé dans cette thèse. Ce chapitre explique donc en quoi les choix faits, en matière de vision mémoire unique, de liberté des schémas d'accès aux données et d'expression du parallélisme basé sur des tâches aux synchronisations implicites, permettent une programmation aisée, lisible et fine des algorithmes parallèles. Les modalités d'exécution (déploiement, placement, ordonnancement) génériques peuvent être choisies par le programmeur et paramétrées par des fonctions de coût modélisant les caractéristiques de la machine parallèle et du programme. Nous terminons ce chapitre par deux exemples didactiques montrant l'implémentation d'algorithmes parallèles simples avec PRFX.

Le chapitre 3 décrit les structures de données sous-jacentes aux fonctionnalités proposées par

PRFX. Il décrit également l'implémentation des deux composants principaux liés au DAG de tâches complet que sont l'inspecteur et l'exécuteur. Ce chapitre fait état du support iso-mémoire de PRFX, il fait également ressortir les aptitudes du mode d'expression et du support à intégrer un maximum d'optimisations, de spécificités des algorithmes prévisibles et de propriétés de la machine. Cette prise en compte va de la génération du DAG de tâches jusqu'à son exécution ordonnancée.

Nous montrons comment ces choix d'implémentations influencent positivement les performances au chapitre 4. Il traite des expérimentations sur des exemples réguliers et irréguliers en les comparant à des exécutions de référence. La scalabilité, le rendement de la machine ainsi que les effets de caches et les recouvrements calcul/communication obtenus expérimentalement avec PRFX sont présentés. Les différences avec des exécutions théoriques que nous prévoyons y sont également expliquées.

Nous indiquons les perspectives envisagées pour PRFX au chapitre 5. Nous y expliquons notamment comment une certaine part d'imprévisibilité (i.e. programmes appelés **quasi-prévisibles**) peut être exprimée avec le support PRFX tout en ayant les avantages d'une exécution parallèle ordonnancée statiquement. Nous présentons aussi d'autres extensions notamment au modèle du DAG complet pour pouvoir, par exemple, faire de nouvelles optimisations ou intégrer des algorithmes effectuant des accès commutatifs aux données ou ayant un flot de contrôle imprévisible. Nous expliquons en quoi notre support est prêt pour faire de la régulation de charge dynamique par migration afin de pallier les approximations des modèles de coûts. Nous évoquons également quels obstacles sont à franchir pour passer des grappes de SMP à des "grilles" dédiées pour le calcul.

Enfin, au chapitre 6, nous concluons sur le travail réalisé et nous décrivons quelles perspectives nous semblent les plus prometteuses à court et à moyen terme.

Une annexe est fournie, elle contient les extraits de codes des différents modes d'expression cités ; ces extraits de codes correspondent à des algorithmes également implémentés avec PRFX et présentés dans cette thèse.

L'ensemble de ce travail a fait l'objet de trois publications [20, 21, 22].

Chapitre 1

Les modes d'expression du parallélisme et leurs supports

Nous utilisons les trois problèmes posés dans l'introduction pour comparer des modes d'expression du parallélisme quant à leurs aptitudes ou non à les satisfaire. Le fil directeur de la comparaison concernera l'implémentation d'algorithmes parallèles irréguliers avec des caractéristiques prévisibles et des données irrégulières, et leurs exécutions sur des grappes de SMP.

Dans ce chapitre, nous allons tout d'abord préciser quelques notions relatives aux algorithmes irréguliers et/ou prévisibles puis nous présenterons un panel de langages et de bibliothèques représentatifs de trois types d'approches pour la programmation des applications parallèles. D'abord, les approches de bas niveau où la gestion du parallélisme est totalement explicite puis celles de haut niveau pour lesquelles le compilateur joue un rôle important dans cette gestion. Ces deux premières approches sont souvent utilisées dans la communauté du parallélisme. Enfin, le troisième type d'approche correspond à celui choisi pour PRFX, avec une modélisation de l'exécution par un DAG de tâches. Ces approches sont soit statiques avec une construction du DAG préalable à l'exécution, soit dynamiques avec une construction à la volée.

Nous ferons à chaque fois un petit bilan indiquant les points qui nous paraissent importants à retenir pour l'approche PRFX.

1.1 Caractéristiques irrégulières et / ou prévisibles des algorithmes parallèles

L'irrégularité est une caractéristique structurelle du déroulement de l'algorithme alors que la prévisibilité est une caractéristique temporelle indiquant le moment ou cette structuration du déroulement est connue. Avant de définir plus précisément ces deux caractéristiques, voyons en quoi elles sont importantes. Lorsqu'un algorithme parallèle est irrégulier et prévisible, son exécution performante repose majoritairement sur la finesse de description de ses irrégularités et sur un ordonnancement global des calculs et des communications tenant compte de modèles de coûts. Le cas des algorithmes imprévisibles est moins problématique car ces derniers recèlent

moins d'informations exploitables si bien que des stratégies de type équilibrage de charge dynamique sont satisfaisantes (e.g. "vol de travail"). De même, les algorithmes réguliers ne sont pas problématiques car exploitables avec des fonctionnalités performantes dédiées (e.g. placements/distribution bloc, cyclique ou combinaisons bloc-cyclique).

Un algorithme est irrégulier s'il manipule une structure de données irrégulières (définies ci-après) ou s'il comporte des schémas de synchronisation irréguliers ou s'il nécessite des transferts de données irrégulières. Concernant les données, nous utiliserons au cours de notre étude les dénominations de **données logiques**, de **données parallèles** et de **données irrégulières**. Les **données logiques** correspondent à un niveau algorithmique et les **données parallèles** sont leurs implémentations avec un mode d'expression parallèle. Ces mêmes **données** sont dites **irrégulières** lorsqu'elles sont de conformation géométrique irrégulière ou constituées de sous-données disjointes (e.g. donnée de type matrice creuse).

Les algorithmes prévisibles ont un flot de contrôle qui peut être intégralement connu au moment du lancement du programme. Cette caractéristique est importante pour les optimisations de type ordonnancement et nous allons nous intéresser à leur impact au niveau de la chaîne de traitements et du support d'exécution.

La gestion de la synchronisation et de la cohérence d'un algorithme parallèle ne peut se faire qu'en fonction de la connaissance du déroulement du flot de contrôle. Cette connaissance peut être capturée à trois moments : à la compilation [37], au lancement via une inspection statique [45] ou à l'exécution via une gestion à la volée [24]. Ces trois moments correspondent à trois seuils croissants en terme d'instanciation de données structurantes. En référence aux outils permettant leur exploitation automatique, nous dirons par abus de langage que ces données structurantes sont respectivement des **données de niveau compilation**, **inspection** et **exécution**. Lors de la phase de compilation et lorsque le compilateur arrive à identifier du parallélisme, ce dernier est prévisible pour les portions de codes parallèles où les informations structurantes sont statiquement disponibles. Lors du lancement du programme, le jeu de données initial permet d'instancier davantage d'informations structurantes. Elles sont utilisables pour trouver du parallélisme supplémentaire via un inspecteur statique agissant avant l'exécution parallèle du programme. Au cours de l'exécution parallèle, les valeurs des dernières données structurantes jusque là inconnues, sont enfin instanciées. Elles sont utilisables pour trouver du parallélisme supplémentaire via un inspecteur dynamique s'exécutant de façon entrelacée avec l'application parallèle.

Le parallélisme des algorithmes imprévisibles n'est exploitable que par un mécanisme dynamique (présent tout le long de l'exécution parallèle). Ce système peut être mis en place automatiquement grâce à l'insertion par le compilateur d'un code d'inspection sur mesure ou être déclenché sur ordre du programmeur. Un inspecteur dynamique peut également traiter le parallélisme des algorithmes prévisibles, mais dans ce cas l'ordonnancement n'interviendra pas au plus tôt ; il sera plus local donc moins performant. En effet, lorsqu'un programme est prévisible, il est possible d'optimiser cette exécution en effectuant un ordonnancement statique global des calculs et des communications.

1.2 Modes d'expression explicites utilisant des bibliothèques

La programmation parallèle explicite nécessite l'usage de bibliothèques de bas niveau. Elle est dépourvue de sémantique séquentielle et charge le programmeur de toutes les tâches de gestion logicielle du parallélisme. Ces dernières peuvent être automatisées efficacement pour certaines classes d'algorithmes comme nous le verrons par la suite. Pour paralléliser un algorithme, le programmeur doit identifier les calculs et accès indépendants et en déduire les synchronisations garantissant la validité de l'algorithme. De plus, dans le cas d'une mémoire distribuée et sans support de type DSM, le programmeur doit implémenter une cohérence des données (e.g. avec des envois de messages). Cette obligation lui permet néanmoins d'exprimer finement des algorithmes irréguliers et d'exploiter les caractéristiques prévisibles des programmes et des machines.

1.2.1 Langages impératifs avec bibliothèque de communication MPI

MPI (Message Passing Interface) [29] est une interface standardisée d'envoi de messages utilisable au sein d'un groupe de processus. La configuration d'exécution la plus générale possible est de type multi-processus en mémoire distribuée. Cette interface est implémentée sous la forme d'une bibliothèque directement accessible depuis les langages C, C++ ou Fortran. Des compilateurs pour ces langages ainsi que des implémentations de MPI sont disponibles sur toutes les machines parallèles. Ceci confère une facilité de portage aux codes écrits avec MPI.

Cette bibliothèque propose des fonctionnalités de haut niveau (communications point à point, collectives, bloquantes, non bloquantes, topologies, communicateurs, envoi de données typées,. . .). Elle spécifie, depuis sa version 2.0, la possibilité de créer des processus dynamiquement et de faire des opérations mémoire à distance (lectures, écritures, écritures commutatives).

La standardisation et les communications collectives sont les principaux apports de MPI. Les communications collectives sont implémentées de façon optimisée en utilisant des fonctionnalités du matériel (implémentations constructeur). Ces implémentations optimisent le nombre de messages, les ordonnancent et les regroupent de sorte à factoriser les latences. Pour en bénéficier, la majorité des codes est donc exprimée selon un modèle SPMD où les phases de calcul alternent avec des phases d'échanges collectifs.

MPI est uniquement orientée vers les communications ; elle ne fournit donc pas d'interface pour déclarer des données logiques sur lesquelles opèrent ces communications. Leur implémentation en données parallèles est donc à la charge du programmeur. Pour cela, il utilise les possibilités du langage choisi afin de fixer leur visibilité et leur durée de vie. Bien qu'explicite, l'implémentation et la gestion en distribué d'une structure de données régulière sont relativement aisées. Par contre, lorsque cette structure de données est irrégulière (e.g. chaînage de données), cela entraîne des difficultés de programmation. En effet, la validité des pointeurs n'est pas assurée au travers de l'envoi de messages avec MPI. Le contenu des données parallèles transmises doit être, soit des scalaires, soit certaines entités MPI (communicateur, type) afin de garder une signification d'un processus à l'autre.

Les calculs et les accès aux données peuvent être explicités aussi finement que le langage le permet (calculs libres, appels à bibliothèque, chevauchement d'accès aux données, accès par pointeurs dans les données locales,...). La limite se situe au niveau de la difficulté d'implémentation de la cohérence et de la synchronisation induite par la vision locale de la mémoire. Théoriquement, cette gestion peut être explicitée finement par le programmeur (modulo les surcoûts du protocole MPI).

Les fonctionnalités de MPI permettent au programmeur d'implémenter finement la cohérence et la synchronisation de son programme. Ces deux informations sont fusionnées dans chaque appel à une primitive de communication MPI (émetteur(s), récepteur(s) et donnée typée à transférer).

L'envoi de données régulières est facilement réalisable avec des zones contiguës contenant des types MPI primitifs (`MPI_DOUBLE`, `MPI_INT`, `MPI_BYTE`,...). Si les données à transférer sont irrégulières, il faut qu'elles correspondent à des types dérivés MPI (`MPI_Datatype`). Ces types dérivés MPI permettent de décrire jusqu'à des collections de vecteurs de tailles différentes espacés par des sauts de tailles variables (avec `MPI_Type_indexed()`). Si la donnée est plus irrégulière alors le programmeur ne peut pas utiliser un type dérivé MPI, mais il peut par exemple utiliser un *buffer* construit avec `MPI_Pack()`.

Lorsqu'un schéma d'accès régulier concernant une même conformation de donnée (même `MPI_Datatype`) est présent dans un algorithme, le programmeur peut aisément implémenter la cohérence associée en utilisant les communications collectives adéquates. Dans les autres cas, cette implémentation est ardue car le programmeur doit écrire ses propres communications collectives.

L'implémentation des schémas de synchronisation réguliers est aisée grâce aux primitives de communications collectives de MPI. Leur domaine d'action est paramétrable par la création de topologies et de communicateurs MPI.

Les schémas de synchronisation irréguliers sont difficilement exprimables car ils nécessitent l'usage d'autant de *send* / *receive* que nécessaire. De plus, lorsque la cohérence et / ou la synchronisation d'un algorithme est imprévisible, le programmeur doit dans certains cas effectuer des communications préparatoires pour mettre en place les schémas de communication effectifs.

La finesse d'expression des schémas de synchronisation est possible grâce à une palette de comportements possibles des communications : appels bloquants ou non puis émissions synchrones ou asynchrones.

Le programmeur peut optimiser l'exécution de son programme MPI en implémentant un ordonnancement (ou placement) des calculs, des communications (hors communications internes aux communications collectives) ou des deux.

L'expression la plus simple d'un ordonnancement, lorsqu'il est invariant, consistera à l'écrire "en dur" en structurant le programme comme voulu. C'est ce qui se produit dans une utilisation classique de MPI. Chacun des processus déroule le même code de contrôle (SPMD) séquentiellement et cet ordre séquentiel fixe l'ordonnancement des calculs et des communications de chaque

processus. Cependant, cet ordonnancement n'est pas paramétrable.

Un ordonnancement plus générique nécessite de structurer le programme en plusieurs procédures (tâches) réceptionnant, traitant et émettant des données parallèles. Ces procédures peuvent être ensuite appelées dans l'ordre voulu par un exécuteur générique déroulant un vecteur de tâches. L'implémentation de cet exécuteur est un effort de programmation à fournir pour bénéficier d'un ordonnancement paramétrable. Ce type d'implémentation permet également d'adopter un ordonnancement statique pour exploiter les caractéristiques des programmes prévisibles.

Dans cette approche statique, une autre possibilité, complexe à implémenter, est d'ordonnancer à la fois les calculs et les communications. Dans ce cas, il faut découpler les calculs des communications : des procédures différentes doivent donc être identifiées pour les réceptions, les traitements et les émissions de messages.

L'usage de MPI peut également être optimisé en composant avec les détails de son implémentation. Ce point est crucial car la programmation au niveau d'un support d'exécution ne peut en être dénuée lors de la recherche de performances.

Par exemple, le programmeur doit éviter d'abuser des créations de communicateurs et de topologies MPI car ces fonctionnalités sont synchronisantes et coûteuses. En plus des problèmes de coûts prohibitifs de certaines fonctionnalités, les implémentations de la bibliothèque MPI respectent rarement la spécification en terme d'asynchronisme et ceci parce que le matériel ne le permet pas. Par exemple, certaines implémentations de MPI sont paramétrables avec une variable d'environnement contenant une taille de message (quelques Ko du *eager limit* sur MPI IBM et MPICH Myrinet) au-delà de laquelle les messages sont envoyés selon un protocole synchronisant de type rendez-vous. Le besoin d'asynchronisme n'est pas l'apanage exclusif des algorithmes aux synchronisations irrégulières, il est également présent dans les algorithmes réguliers. Cependant, l'exploitation de l'asynchronisme des cas irréguliers nécessite un mode de programmation MPMD car tous les processus ne recouvrent pas les communications asynchrones avec les mêmes calculs. Or, les communications collectives, principal apport de MPI, sont dédiées au SPMD. Dès lors, le coût de la latence introduite par l'implémentation de MPI n'est plus justifié face à une bibliothèque de bas niveau d'envoi de messages. En conséquence, l'utilisation performante de MPI dans des codes irréguliers est compliquée, spécifique à la machine et à l'application. Enfin, après cet effort d'optimisation, le code est peu lisible comme nous allons le voir.

Comme les implémentations de MPI ont des effets de seuil au niveau de leurs coûts d'envoi de messages, le programmeur optimise son programme en restant dans le meilleur mode de fonctionnement (choix de taille de messages arbitraires pour bénéficier de stratégies d'implémentation de MPI). Ces modifications permettent l'obtention de performances mais dégradent encore plus la lisibilité du code. Plus généralement, cette lisibilité est un point faible des codes utilisant la bibliothèque MPI. En effet, d'une part, la lisibilité s'amenuise pour les codes MPI performants car ils sont obligatoirement la fusion d'au moins trois codes décrivant : les calculs et données de l'algorithme, la gestion des synchronisations et celle de la cohérence. A ceci, viennent s'ajouter les optimisations telles que la prise en compte des effets de seuil (coûts des communications, coûts d'accès aux caches) et l'ordonnancement. Pour ces programmes, le changement

d'implémentation MPI rend souvent ces optimisations inopérantes.

D'autre part les messages ne sont pas contraints par les barrières modulaires du langage. Par exemple, le *send* correspondant à un *receive* peut se trouver n'importe où dans le code source (même problème que l'utilisation de variables globales en programmation séquentielle).

La hiérarchie mémoire des grappes de SMP est exploitée par des implémentations de MPI utilisant un segment de mémoire partagée pour les communications entre les processus d'un même nœud. Mais dans ce cas, les envois de messages en intra-nœud consomment de la bande passante mémoire. Cela dégrade les performances des processeurs et donc des calculs de l'application.

Le paramétrage du déploiement des processus utilisant MPI permet également une prise en compte de l'hétérogénéité de la machine. La distribution des données faite par le programmeur peut ainsi tenir compte non seulement de la localité par processeur mais aussi par nœud SMP. La taille des données peut également être choisie pour correspondre à la taille d'un niveau de cache de la hiérarchie mémoire d'un nœud SMP.

Enfin, du point de vue modèle, le passage de messages implique l'utilisation de deux processeurs (*send*, *receive*) alors qu'un seul est nécessaire avec un mode de communication unilatéral. MPI-2 propose des fonctionnalités d'écriture et de lecture dans des fenêtres mémoire distantes selon deux modes. Le premier est synchrone et le processus cible actif : la communication n'est donc pas unilatérale. Le second est asynchrone et le processus cible potentiellement inactif : la communication est donc unilatérale. Par contre, le processus cible n'a aucun moyen de savoir si un message est arrivé. En effet dans ce second cas, les seuls appels MPI en rapport avec les fenêtres mémoire définies sont `MPI_WinLock()` et `MPI_WinUnlock()`. MPI-2 définit des fonctionnalités pour les écritures et lectures distantes inadaptées à l'exploitation de l'asynchronisme d'un programme. Cette interface est dédiée à un fonctionnement SPMD par phase synchronisante (`epoch` MPI). A la fin de chaque phase, chacun des processus dispose de la donnée cohérente de la phase précédente.

Nous retiendrons que l'écriture des codes MPI est fastidieuse mais autorise une expression fine des algorithmes irréguliers, et que la lisibilité et la généricité des optimisations sont difficilement possibles. Ces défauts sont compensés par de bonnes performances qui peuvent être améliorées pour les grappes de SMP grâce à l'existence d'implémentations de MPI exploitant la mémoire partagée (i.e. communications via des segments de mémoire partagée).

1.2.2 Langages impératifs avec bibliothèque MPI et *threads* POSIX

L'API des *threads* POSIX [39] est standard et une implémentation est fournie avec la majorité des systèmes d'exploitation modernes. Cette API permet la description de la gestion des *threads* et de leurs synchronisations de façon portable.

L'ajout de *threads* dans un code MPI nécessite la ré-écriture du programme et l'utilisation au minimum d'une implémentation *thread safe* de MPI (réentrante dans le cas idéal). Dans ce

cas, l'exécution se déroule dans un cadre multi-processus multi-*threads* dont le déploiement est implémenté par le programmeur.

Les données conservent les mêmes caractéristiques que dans le cas MPI "pur" (cf. section précédente). Dans le cas d'un déploiement d'un processus par nœud SMP et d'autant de *threads* que de processeurs, la granularité des données par processus est plus importante que dans le cas d'une exécution seulement multi-processus. La vision des données change car une partie des données devient directement accessible via la technologie performante de la mémoire partagée alors que l'autre conserve son caractère distribué. L'implémentation de la cohérence peut donc concerner des conformations de données de taille plus importante et être plus aisée qu'en MPI pur.

Par contre, la synchronisation est plus complexe à implémenter même pour des schémas de synchronisation réguliers. En effet, une bibliothèque supplémentaire (e.g. Pthread) doit être manipulée par le programmeur pour effectuer les synchronisations intra-processus car MPI n'identifie que les processus. Le programmeur ne peut donc pas identifier un *thread* particulier comme destinataire d'un message. Cela force le programmeur à réécrire la partie synchronisation intra-processus lors des communications collectives car seul un *thread* par processus y participe. Néanmoins, les grappes de SMP peuvent être exploitées au mieux de leurs capacités dans ce mode de programmation hybride (MPI+Pthreads).

Les *threads* permettent en théorie un recouvrement des communications sans effort particulier du programmeur. En effet lorsqu'un *thread* est en attente d'une communication, c'est l'ordonnanceur système et non le programmeur qui a la charge de trouver un *thread* prêt à s'exécuter pour occuper le processeur. Les implémentations d'algorithmes irréguliers peuvent être plus lisibles car elles n'ont plus besoin de contenir des calculs totalement étrangers au contexte d'une procédure et dont la présence n'était justifiée que pour bénéficier d'un hypothétique recouvrement. Le recouvrement sur certains réseaux actuels (Myrinet, IBM *Colony*) n'est pas intéressant à exploiter car les implémentations des bibliothèques MPI ne sont pas suffisamment asynchrones. A notre connaissance, l'implémentation de MPI sur ELAN quadrics de Compaq est réellement asynchrone.

Ces facilités liées aux fonctionnalités dynamiques proposées par le système s'opposent au respect d'un ordonnancement statique pour exploiter les programmes prévisibles. Elles doivent être abandonnées pour la solution plus complexe déjà décrite dans le cas MPI "pur", avec des *threads* exécuteurs de tâches affectés à des processeurs exclusifs. Si ce n'est le cas, l'exécution peut être perturbée et rendre l'ordonnancement caduc (quantum de temps perdus, effet pipeline ou de cache supprimé par la préemption). Cela se produit même pour des exécutions dédiées car d'autres *threads* sont créés de façon indépendante par des bibliothèques utilisées (e.g. MPI) et parce que le système peut faire migrer un *thread* vers un autre processeur du nœud SMP.

L'usage conjoint des deux bibliothèques pour les synchronisations complexifie l'écriture des algorithmes réguliers et irréguliers. Les optimisations sont encore plus spécifiques pour tenir

compte des implémentations de ces deux bibliothèques. Cependant, les *threads* permettent d'améliorer les performances sur les grappes de SMP.

1.2.3 Langage C avec bibliothèque de RPC PM^2

PM^2 [4, 47] est une bibliothèque proposant un support multi-processus et multi-*threads* pour effectuer des RPC. Elle utilise la bibliothèque de *threads* Marcel et la bibliothèque de communication Madeleine qui est multi-threadée. Elle a été initiée en 1994 et son développement est actuellement poursuivi au LaBRI (Bordeaux). La configuration d'exécution typique est un ensemble de processus sur un cluster de PC Linux.

Les *threads* PM^2 sont des *threads* utilisateurs de la bibliothèque Marcel associés à des *threads* POSIX et également compatibles avec l'API POSIX. Ces *threads* sont gérés dynamiquement. Un *thread* de service est utilisé par la bibliothèque de communication Madeleine dans chaque processus pour gérer les RPC (empaquetage, dépaquetage explicite des données passées en arguments). Un RPC provoque l'exécution d'une fonction cible du programmeur dans le *thread* de service à distance. Dans cette fonction, le programmeur effectue les allocations et le désempaquetage des données, puis il crée un *thread* PM^2 pour poursuivre l'exécution de la fonction cible du RPC ; ainsi il ne monopolise pas le *thread* de service.

Le mode de programmation est de type MPMD, donc PM^2 ne propose pas de fonctionnalités pour des échanges collectifs. Toutes les synchronisations et communications sont descriptibles finement mais par le seul usage des RPC ; ceci rend le code difficile à écrire et à lire. Le code des RPC est entièrement libre mais le programmeur doit protéger les accès s'ils sont faits en dehors des arguments (e.g. accès à des données locales persistantes). Cette liberté permet d'exprimer finement tous les algorithmes quelle que soit leur irrégularité.

Des données spéciales (iso-données) peuvent être créées par le programmeur. Elles sont créées dans la zone d'iso-allocation du processus local. Chaque processus a sa propre zone d'iso-allocation dans un espace d'adressage commun à tous les processus. L'iso-espace d'adressage permet d'implémenter des données parallèles irrégulières car les pointeurs restent valides de l'espace mémoire d'un processus à un autre. Ces données ne sont pas manipulables finement car elles doivent être structurées par blocs de la taille d'une page mémoire (4 Ko) et explicitement cédées entre *threads*.

Les grappes de SMP peuvent être exploitées grâce au support multi-*thread* Marcel en intra-nœud et à la couche de communication Madeleine en extra-nœud.

L'ordonnancement de ces *threads* est géré dynamiquement à la fois par le système et par l'ordonnanceur Marcel. Cet ordonnancement des *threads* peut être effectué globalement grâce aux propriétés de migration préemptive des *threads* Marcel. L'ordonnanceur Marcel est interchangeable avec un autre écrit par le programmeur, mais n'est pas adapté à l'application d'un ordonnancement statique de tâches (en supposant qu'un *thread* est créé par tâche).

PM^2 étant un support d'exécution, il est difficile de discuter du mode d'expression qu'il propose, car par nature, il doit être utilisé par des modes d'expression de plus haut niveau (e.g. HPF [12]).

Le niveau d'écriture est fastidieux, mais la programmation MPMD et l'iso-mémoire rendent PM2 plus adaptées que MPI à la description fine des synchronisations des algorithmes irréguliers. Les optimisations concernant les données sont peu lisibles et un ordonnancement dédié est difficile à concilier avec le fonctionnement de PM2. L'exécution sur les grappes de SMP est efficace et adaptative grâce à la possibilité de migration de *threads*.

1.2.4 Langages impératifs en mémoire partagée multi-processus et/ou multi-*threads*

Une SM (Shared Memory) ou une DSM (Distributed Shared Memory) fournissent respectivement un espace mémoire partagé exploitable par des *threads* au sein d'un système à image unique ou par des processus répartis sur plusieurs systèmes/machines (masquage du réseau). La mémoire partagée permet d'implémenter directement les données logiques avec les données du langage séquentiel utilisé (C, C++, Fortran,...).

Le programmeur explicite les synchronisations (e.g. via la bibliothèque Pthread), par contre la cohérence est gérée par un support matériel ou logiciel (SDSM). Dans le cas des SDSM (CASHMERe-2L [63], MGS [68] du MIT, Shasta fine grain [59], TreadMarks [2], SCASH [60]), plusieurs processus sont utilisés alors qu'un seul avec des *threads* peut suffire lors de l'utilisation d'une SM ou DSM à cohérence matérielle. Les synchronisations sont finement exprimables mais leur programmation devient complexe pour les schémas de synchronisations irréguliers. Le contenu du code du programme n'est pas contraint et la vision de la mémoire est unique.

Le parcours de tout ou partie de données parallèles est exprimable ; cependant, pour que les accès soient performants, le programmeur doit adapter le stockage des données au fonctionnement de la cohérence du support. Ces accès doivent être alignés et dimensionnés selon la granularité utilisée par la cohérence mémoire. Ceci autorise l'implémentation raisonnablement aisée des algorithmes irréguliers et de structure de données irrégulières [48].

La cohérence d'une SM fonctionne par invalidations et défauts de lignes de caches (environ 128 octets). Dans le cas d'une DSM, la granularité passe à la taille d'une page mémoire (de 4 Ko jusqu'à 16 Mo). Ceci peut provoquer une synchronisation pour les écritures mémoires en parallèle dès l'instant où elles concernent une même page mémoire (*false sharing*). Ce surcoût intervient statistiquement d'autant plus fréquemment que la granularité des tailles de pages est importante et que les accès concurrents sont localisés sur un nombre réduit de pages. Dans le cas des SDSM, ces synchronisations sont coûteuses et n'autorisent donc pas une bonne scalabilité des applications qui ne prennent pas en compte ce phénomène. Ces coûts sont liés à l'utilisation des fonctionnalités du système d'exploitation (appels systèmes) et aux temps de transferts de pages. Le mécanisme sous-jacent s'appuie sur les informations récupérées auprès des processeurs par le système sur les pages accédées. Il consiste en l'invalidation d'une ou de plusieurs pages dans la table des pages du processeur anciennement propriétaire, en leur transfert et en leur activation chez le nouvel acquéreur. Il existe de nombreuses stratégies ne transmettant que les différences ou en relâchant les contraintes [44, 41].

Les transferts de données générées par cohérence des SM et DSM s'effectuent dans l'urgence en interrompant le flot d'exécution du processeur. Ceci est nécessaire pour garantir la validité de

l'exécution du programme, mais empêche de connaître les futurs accès et les fautes de pages potentielles associées. Le programmeur cherchant la performance doit être très vigilant car la contention mémoire / réseau est le principal problème de ce mécanisme.

Le fonctionnement des SM ou des DSM altère les modèles de coûts car il interrompt les calculs de façon inopinée. Ces altérations sont importantes dans le cas des SDSM (e.g. nouvelle page mémoire accédée). Pour ces raisons, une SDSM n'autorise ni une bonne prise en compte des aspects prévisibles du programme ni un ordonnancement de qualité des calculs et des communications.

Les grappes de SMP peuvent être exploitées de façon performante quand la cohérence est assurée par un matériel performant comme sur les machines SGI Origin (type 3400-3800). Ces machines à mémoire distribuée CC-NUMA (Cache Coherent Non Uniform Memory Access) ont des coûts d'accès mémoire variables selon l'éloignement de la page mémoire accédée.

La DSM masque l'existence du réseau, elle ne transparaît donc à l'utilisateur que par des temps d'accès plus importants entre mémoire locale et mémoire distante.

Le programmeur peut prendre en compte la hiérarchie mémoire interne à un nœud SMP en utilisant une distribution de données paramétrée par la taille de la mémoire locale.

La vision mémoire unique facilite l'usage de ce mode d'expression mais pour obtenir des performances le programmeur doit faire des concessions sur la finesse et l'aisance d'expression pour adapter l'algorithme au grain de fonctionnement du support matériel ou logiciel fournissant la mémoire partagée. De même, la lisibilité et la généricité des optimisations dépendent du comportement modélisable ou non de ce support. Les grappes de SMP ayant une cohérence mémoire matérielle peuvent être exploitées efficacement.

1.3 Modes d'expression partiellement explicites compilés

Nous avons observé quatre éléments importants dans l'utilisation des modes d'expression explicites présentés précédemment. Le premier concerne la programmation MPMD par tâche qui est mieux adaptée que le modèle SPMD pour l'ordonnancement des exécutions et la description des algorithmes irréguliers. Le deuxième est la simplicité de programmation dès que la vision de la mémoire est unique ou que la description de la cohérence n'est plus nécessaire et la difficulté de programmation lorsque le programmeur doit expliciter les synchronisations. Le troisième met en évidence les performances obtenues avec des stockages en mémoire physique précis. Le quatrième correspond à la prise en compte de l'hétérogénéité des SMP par l'utilisation de *threads*. Les langages parallèles partiellement explicites tentent de reprendre ces éléments et d'en automatiser l'utilisation.

1.3.1 Langage OpenMP : Open Multi Processing

OpenMP [51] est un langage à parallélisme de tâche de type *fork-join* pour architectures à mémoire partagée. Il a été standardisé en 1997 et plusieurs compilateurs éprouvés sont disponibles

chez les constructeurs et dans le domaine public (Adaptor [64], OdinMP [15], Omni [50]). Son existence est une volonté des constructeurs de machines SMP pour permettre aux utilisateurs, notamment scientifiques, de les programmer plus facilement en annotant des codes séquentiels C, C++ et FORTRAN existants.

Pour le programmeur, le mode de programmation est de plus haut niveau que la programmation multi-*threads* précédemment étudiée. En effet, toute la gestion des *threads* est confiée au support qui lui-même utilise les fonctionnalités du système. Ce choix de simplification pour le programmeur conduit à le priver de fonctionnalités notamment de synchronisations fines proposées par l'API Pthread.

Un programme OpenMP est décrit par l'insertion de directives dans un code séquentiel sous la forme de `#pragma` en C, C++ ou de commentaires en Fortran. Ces directives n'ont que la certification du programmeur.

La déclaration du parallélisme apparaît dans le code sous la forme de sections parallèles ou d'itérations de boucles parallèles (parallélisme de boucle). Cette parallélisation peut être effectuée de façon incrémentale.

Une exécution d'un programme OpenMP comporte un seul processus (*thread* maître) et une multitude de *threads* avec lesquels le programmeur interagit via l'interface OpenMP. En effet, les sections et itérations parallèles sont exécutées par le nombre de *threads* indiqué par le programmeur. Par contre, la relation entre *threads* et processeurs physiques n'est pas spécifiable via OpenMP.

La vision mémoire est unique et le programmeur peut contrôler le stockage de ses données parallèles (via l'allocation dynamique en C, C++ et FORTRAN) en mémoire. L'implémentation des données logiques même irrégulières est donc aisée.

Les données parallèles sont par défaut toutes les données statiques (C, C++, Fortran), toutes les allocations dynamiques et toutes les variables déclarées avec la clause `shared(varname)`. Le programmeur définit à loisir leur visibilité et leur durée de vie dans le langage utilisé. Le contenu est libre (scalaires, pointeurs, descripteurs de fichiers, identificateur système ou de bibliothèques *thread safe*,...) grâce à l'unicité du processus. Des variables peuvent être privatisées avec la clause `private(varname)`. Cette clause peut être aussi utilisée pour privatiser des tableaux entiers s'ils sont alloués statiquement.

Le parallélisme de boucle d'OpenMP suppose que l'espace des itérations d'une boucle parallèle doit pouvoir être réparti entre les *threads* par le support OpenMP à l'exécution. Cela entraîne l'impossibilité de faire un effet de bord sur le pas ou la borne de fin et donc de faire un `break`. Hormis ces contraintes, le contenu des boucles est aussi libre (calculs, appels bibliothèque, accès,...) que le permet le langage d'origine.

La génération automatique des synchronisations est impossible par compilation dans le cas général. Ces synchronisations sont donc explicites et à la charge du programmeur. Elles interviennent par exemple en fin de région parallèle, sur des verrous, des sections critiques, des accès à des variables atomiques, lors de barrières ou de `flush`. La directive `flush` n'est pas d'une

utilisation très simple. Elle permet de régler les problèmes dus à la mise en registres des variables par les compilateurs. Cette directive génère un point de synchronisation, au sortir duquel les variables spécifiées sont cohérentes en mémoire.

Les directives de parallélisme des sections et boucles parallèles permettent de décrire aisément des synchronisations régulières de type *fork-join*. Elles peuvent être étayées pour décrire certaines synchronisations irrégulières par les directives `master`, `critical`, `atomic` et `nowait`. Par exemple, la résolution des conflits d'accès à des données partagées est facilement exprimable avec `critical`. Lorsque le conflit porte sur une seule affectation la directive `atomic` est utilisable. Les synchronisations irrégulières des programmes asynchrones peuvent être en partie exploitées par la directive `nowait`. Les possibilités de parallélisme de tâche et de synchronisation à plusieurs niveaux sont également exprimables mais encore mal supportées [6]. Le grain des tâches est paramétrable par le découpage des itérations des boucles en tronçons (*chunk*) de tailles spécifiables avec l'option (`schedule(type,taille)`). Lors d'une exécution, l'ordonnancement des calculs dépend de celui des *threads* par le système et de la répartition des tronçons d'itérations des boucles. La répartition des itérations parallèles est faite à l'exécution avec différentes variantes possibles. Ces variantes sont spécifiables à l'aide des clauses `schedule(guided,taille)`,`schedule(dynamic,taille)` et `schedule(static,taille)`. Ainsi, l'utilisateur choisit le mode de répartition des tâches. Leur coût n'est pas communicable au compilateur OpenMP mais les modes `dynamic` et `guided` gèrent les coûts irréguliers et/ou imprévisibles avec une stratégie générique de vol de tronçons d'itérations. Ce mode de fonctionnement ne permet pas d'exprimer de façon lisible des heuristiques d'ordonnancement global exploitant les caractéristiques prévisibles des algorithmes.

De plus, lorsque l'algorithme contient des synchronisations irrégulières, ces directives ne sont pas assez fines. Dans ce cas, le programmeur doit procéder à une implémentation ardue effectuant le découpage de l'espace d'itération des boucles et utilisant ses propres moyens de synchronisation (usage anormal du langage).

Cette programmation explicite est aussi utilisée dans les cas réguliers pour améliorer les performances. En effet, la technologie des compilateurs OpenMP et les performances des supports utilisant les bibliothèques de *threads* ne permettent pas d'obtenir une bonne scalabilité des applications suffisamment couplées ou irrégulières au-delà d'une dizaine de processeurs sur les machines actuelles. Les programmeurs s'inspirent donc de codes MPI en adoptant une programmation de type SPMD de plus bas niveau. L'implémentation de l'algorithme de Jacobi présentée en Annexe A.1.1 utilise ce style de programmation. L'identification des tâches est faite par le programmeur au lieu du compilateur en écrivant un code calculant les bornes des boucles pour chaque *thread*. Dans ce code, il n'y a donc aucune boucle précédée d'une directive parallèle OpenMP. Dans ce cas, le compilateur OpenMP est utilisé comme un simple compilateur C, C++ ou Fortran ; toute la gestion du parallélisme est décrite par le programmeur dans un code peu lisible.

La distribution des données n'est pas spécifiable dans le langage OpenMP : les optimisations de stockages des données améliorant la localité des accès sont difficilement exprimables et donc

peu lisibles. En effet, une distribution peut être indirectement effectuée par des experts en utilisant les découpages des espaces d'itérations ou des sections parallèles, et en connaissant les stratégies systèmes pour les pages mémoires et l'affectation des *threads* sur les processeurs. Lorsque les accès n'ont pas les mêmes alignements d'une boucle à une autre, le programmeur doit retravailler son code pour conserver la localité des accès malgré des bornes de boucles disparates. Il en résulte une écriture et une lecture du code plus ardues.

Une grappe de SMP dont la mémoire est physiquement distribuée, peut être exploitée si des systèmes à image unique comme IRIX [61] ou MOSIX [46] sont utilisés. Ils garantissent la cohérence des accès mémoire par les processeurs au moyen d'une technologie logicielle (SDSM) ou matérielle (DSM). L'obtention de performances [16] sur ces machines requiert une adaptation du code, l'utilisation de directives propriétaires [62] et d'outils externes de découpage et de placement de la mémoire du processus (e.g. `dplace` sur SGI Origin). Le fonctionnement par page de ces machines entraîne une diminution des performances ou la nécessité d'adapter le stockage des données parallèle pour les DSM.

Les versions distribuées des supports OpenMP fonctionnant avec plusieurs processus et une SDSM sur une grappe de SMP (Omni SCASH [49] avec l'environnement SCORE [60]), sont expérimentales et ne respectent pas la norme à savoir l'unicité du processus avec *N threads*.

Le code libéré des annotations OpenMP peut ne plus être valide pour une compilation et une exécution séquentielle, ce qui est dommageable pour le développement et le déboguage de programmes OpenMP.

La vision mémoire unique facilite la programmation mais le langage OpenMP n'est pas adapté à la description fine d'algorithmes irréguliers. Les optimisations concernant les données sont dépendantes du support assurant la cohérence mémoire et sont donc peu lisibles. Les optimisations via les directives de tronçonnage et d'ordonnancement des espaces d'itérations sont lisibles mais insuffisantes ; elles ne permettent pas d'exploiter les caractéristiques prévisibles des programmes. Les grappes de SMP à cohérence mémoire matérielle sont exploitables efficacement avec des directives constructeurs.

1.3.2 Langage HPF : High Performance Fortran

HPF [37] est un langage à parallélisme de donnée sur les tableaux. Il date des années 1990 et exploite les machines à mémoire distribuée ou partagée. Il étend le langage séquentiel Fortran 90 et est doté de plusieurs compilateurs éprouvés d'origine privée (IBM, Intel, DEC, NEC, PGI) ou publique (Adaptor[65], Vienna Fortran [17]). Ce langage propose un mode de programmation à base de boucles de calculs parallèles [27, 69] opérant sur des tableaux distribués sur des processeurs virtuels. Ces derniers sont une abstraction de la machine parallèle.

Le compilateur se charge de grouper les calculs locaux à chaque processeur et de générer un code SPMD assurant la cohérence pour les données accédées.

De son côté, le programmeur doit expliciter la synchronisation en insérant des directives pour relâcher les contraintes de synchronisation implicitement induites par le déroulement linéaire du code séquentiel. Ce relâchement de l'ordre séquentiel peut être décrit avec deux directives prédéfinies : `FORALL` et `INDEPENDENT`. L'indépendance des itérations de boucles (absence de synchronisation), est indiquée par la directive `INDEPENDENT`. La directive `FORALL`, plus restrictive, décrit une exécution en parallèle mais synchrone des calculs du corps d'une boucle. Ces directives interdisent les tests de sortie dans les corps de boucles qu'elles englobent car ils rendraient la synchronisation imprévisible (indéfinie). Avec cette contrainte, la présence de ces directives, et l'utilisation par défaut de la règle des écritures locales (i.e. chaque processeur execute seulement les instructions modifiant ses propres données), le compilateur est capable de générer un exécutable parallèle. Cette stratégie constitue la base des compilateurs *data-parallel*. D'autres langages à parallélisme de données tel que PANDORE [3, 43] en faisaient usage. De plus, le compilateur PANDORE optimisait la parallélisation des boucles avec une méthode utilisant les polyèdres convexes [40]. Dans le cas de HPF, cette règle des écriture locales peut être changée au profit d'une autre règle de localité des opérations, spécifiée avec la directive `ON HOME` [14].

Les données logiques sont implémentables avec des tableaux. Ainsi, pour les nombreux algorithmes scientifiques utilisant des matrices, le mode d'expression est aisé.

Dans l'exécutable produit après passage du compilateur HPF, les calculs sont regroupés selon la distribution des données. La règle des écritures locales est utilisée pour ne pas créer d'ambiguïté lors de la compilation des communications. A un instant donné, la distribution d'un tableau est unique et le membre gauche (en écriture) d'une affectation correspond à un unique indexage dans un tableau donc à un unique processeur propriétaire.

L'exécution d'un programme HPF nécessite un support d'exécution associé au compilateur qui implémente la machine parallèle abstraite sur la machine réelle. Cette implémentation utilise généralement MPI pour des raisons de portabilité et permet d'exploiter les machines à mémoire distribuée.

Le langage HPF donne les moyens de décrire un parallélisme de données par l'insertion de directives dans un code FORTRAN 90 existant. Ces insertions précèdent les boucles parallèles itérant sur des tableaux. Ces derniers doivent obligatoirement avoir été préalablement distribués via une directive pour que le compilateur puisse travailler. L'insertion de ces deux types de directives sont suffisantes. Cette simplicité a permis de paralléliser sommairement mais facilement, de nombreux codes scientifiques FORTRAN.

Toutefois, la mise au point par un programmeur novice peut se révéler délicate car les directives ne peuvent pas toutes être vérifiées par les compilateurs HPF. En effet, il est du ressort du programmeur de garantir l'indépendance des accès. L'optimisation d'un code HPF est un autre point délicat. Elle ne peut être faite que par un expert. Elle nécessite non seulement la connaissance des caractéristiques de l'algorithme, de la machine mais aussi de l'implémentation du compilateur et du support d'exécution (e.g. l'implémentation de MPI sous-jacente).

Seules les données logiques de type tableaux *n*-aires sont exprimables aisément en données parallèles. Ces tableaux sont ceux du langage Fortran 90 mais dans le cadre de HPF, leur contenu est limité à des valeurs scalaires car d'autres informations seraient incohérentes si elles étaient

échangées entre les processus. Les tableaux peuvent être déclarés allouables dynamiquement (`ALLOCATABLE`) ou statiquement mais le langage ne permet pas de capturer tout le savoir-faire du programmeur en matière de stockage en distribué. La performance dépend de la capacité d'optimisation du compilateur. Par exemple, les tableaux alloués dynamiquement sont pénalisants pour les performances [25] car le compilateur doit générer un code paramétré par cette taille dynamique.

La distribution des données conditionne le grain d'accès (découpage), l'équilibrage de charge (placement) et indirectement les communications. Une seule source d'information est donc utilisée pour paramétrer trois aspects importants et différents de l'application. Elle doit être obligatoirement choisie par l'utilisateur parmi celles prédéfinies (`BLOCK`, `CYCLIC`) ou celles quelconques (`MAP`). Cette dernière directive permet de spécifier des distributions irrégulières et donc d'équilibrer la charge de programmes aux coûts de calcul hétérogènes et prévisibles. Si cette distribution n'est connue qu'à l'exécution, le compilateur manquera d'information pour générer un code performant. Généralement, seules les distributions et découpages réguliers de grain suffisant sont implémentés de façon performante par les compilateurs et les supports. Ceci limite la classe de programmes réellement exploitables.

Le programmeur a à sa disposition une vision unique des tableaux du langage. Les accès aux valeurs se font directement par indices $(i_0,\ldots,i_{n-1})$ pour un tableau de dimension n.

Les calculs et les accès effectués dans les boucles parallèles doivent être prévisibles (fonctions affines de données de niveau compilation) pour que le compilateur HPF puisse générer automatiquement la cohérence associée. Tous les calculs des boucles parallèles doivent uniquement faire des accès indicés dans des tableaux comme dans l'exemple du listing 1.1.

Listing 1.1 – boucle parallèle HPF.

```
1 ! declaration d'un tableau TAB de flottants bi-dimensionnel N x N
2       Real TAB(0:N-1,0:N-1)
3 ! distribution par block
4 !HPF$ DISTRIBUTE(BLOCK,BLOCK)::TAB
5 ! relâchement de la synchronisation séquentielle
6 !HPF$ DO INDEPENDENT
7       do i = 0, (N-5)/2
8         TAB(2*i+3, i) = TAB_1(i + 4) + ... + TAB_K(3*i, 2*i+1, 4*i)
9       enddo
```

Si un appel de procédure intervient dans une boucle parallèle, elle doit être sans effet de bord (`PURE`) ou avoir des accès analysables par compilation (`EXTRINSIC` ou `INTRINSIC`).

Les fonctions `INTRINSIC` sont celles prédéfinies par le langage Fortran 90 (`dot_product()`, `sum()`, `matmul()`) et sont utilisables directement. Par contre, pour utiliser, par exemple, une routine de la bibliothèque BLAS, sa description `EXTRINSIC` doit être faite.

Si les accès sont irréguliers, le compilateur HPF produira une cohérence peu performante car elle concernera des transferts de données irrégulières. Toutefois, si l'irrégularité de ces accès est descriptible avec des données de niveau exécution telles que des tableaux d'indirection

(INDIRECT) alors la cohérence est compilable de façon optimisée. Les motifs d'accès s'intersectant peuvent également être exprimés et traités de façon performante grâce à des directives sur les tableaux (alignement TEMPLATE, extension HPF2 : SHADOW et non standard : HALO [13]). Il faut noter que selon la spécification, les SHADOW ne sont pas utilisables dans des codes génériques où la distribution des données est seulement connue à l'exécution.

La migration de données est ponctuelle et s'effectue lors d'une redistribution. Elle permet d'améliorer les performances des programmes dont les schémas d'accès changent en cours d'exécution. La redistribution, avec possibilité de changement de grain, est coûteuse et doit être utilisée avec précaution [17](directives DYNAMIC, REALIGN et REDISTRIBUTE).

Le mode d'expression proposé par HPF est dédié aux algorithmes réguliers et structurés par des données de niveau compilation. Néanmoins une part d'imprévisible est exprimable grâce à l'utilisation d'un système d'inspection / exécution au niveau du support. Si les accès dépendent de données de niveau inspection ou exécution, le compilateur ou le programmeur en dernier recours, peut utiliser l'interface de la bibliothèque d'inspection / exécution [5] du support en tentant d'amortir le coût de ce mécanisme. Lorsque le programmeur insère des appels à cette bibliothèque interne, cela constitue un usage anormal du langage HPF. Le code A.27, correspondant à l'algorithme de Jacobi en annexe A.2.1, montre un exemple d'utilisation de l'inspecteur avec le support Adaptor.

D'autre part, HPF permet l'utilisation d'opérations globales de réduction (clause REDUCTION) ou de macro opérations (max, min, égalité) sur les tableaux distribués. Ceci permet de simplifier l'expression du code et de bénéficier d'optimisations spécifiques à la synchronisation associée. Ces opérations globales ne centralisent pas la donnée sur un seul processeur virtuel HPF, elles opèrent en distribué.

HPF 2 permet de définir des *task region* mais sans possibilité de synchronisation entre elles. Cette nouvelle fonctionnalité est a priori destinée aux programmeurs d'applications de type *divide and conquer* (exécution arborescente),.

L'expression de boucles avec des dépendances partielles inter-itérations n'est pas possible dans le langage HPF. Ceci empêche l'expression fine des algorithmes aux synchronisations irrégulières. Pour permettre ces synchronisations, des travaux proposent des extensions [14].

Dans le langage HPF, l'ordonnancement des calculs et des communications n'est pas paramétrable. Les calculs et les communications suivent l'ordre dans lequel ils apparaissent dans l'exécutable généré. Or, l'exploitation performante du parallélisme des programmes passe par un ordonnancement spécifique.

L'hétérogénéité d'une grappe de machine SMP peut être prise en compte par le support ou le programmeur mais non par le compilateur. En effet, le compilateur suppose un modèle de machine homogène de mono-processeurs. Le programmeur peut jouer sur la distribution et le découpage des données en contrôlant le déploiement de MPI et l'affectation des processeurs virtuels aux processus MPI (non standardisé au niveau du mode d'expression HPF). Toutefois le programmeur n'a pas de retour d'informations précises relatives aux performances. Par exemple,

il ne peut évaluer le temps passé dans chaque calcul d'un bloc de tableau. L'hétérogénéité apparaît donc uniquement sur des fluctuations de durée de temps total des phases parallèles qui sont influencées par beaucoup d'autres paramètres. Dans ce contexte, les optimisations sont empiriques.

Dans tous les cas, la bibliothèque MPI utilisée peut faire bénéficier le support HPF de communications via la mémoire partagée du nœud SMP.

Une autre solution est l'intégration de *threads* au niveau du support (e.g. Adaptor [9]) sur lesquels sont répartis les processeurs virtuels. Dans ces travaux, le programmeur peut signifier l'utilisation de deux niveaux hiérarchiques (nœud et processeur) au compilateur.

HPF offre un niveau d'expression aisé mais seul l'usage d'extensions (directives) HPF2 d'un support adoptant une approche inspection / exécution permettent d'exploiter certains algorithmes irréguliers. HPF ne permet de décrire ni des d'optimisations de stockage, ni des ordonnancements exploitant les caractéristiques des programmes prévisibles. Les supports multi-*threads* permettent d'exploiter la mémoire partagée des grappes de SMP.

1.4 Modes d'expressions basés sur un modèle de DAG de tâches

Les langages de haut niveau, tels OpenMP et HPF, que nous venons de voir ne sont pas adaptés à la prise en compte des programmes aux synchronisations ou aux structures de données irrégulières. En effet, ces informations font partie du langage et ces derniers ne proposent qu'un nombre limité de constructeurs de base. Ce choix pousse les programmeurs à demander toujours plus de directives et d'extensions difficilement implémentables. La cohérence des programmes ne pose, quant à elle, pas de problème au niveau expression car elle est implicite avec une gestion par un support matériel ou logiciel (OpenMP), ou générée par compilation (HPF).

Les modes d'expression basés sur un modèle de DAG de tâches que nous allons voir adoptent une approche inverse avec une cohérence explicite et des synchronisations implicites. En effet, le programmeur décrit des tâches et leurs accès aux données qui sont ensuite analysés (par inspection) pour en déduire automatiquement le schéma de synchronisation (ou communication) correspondant. L'expression de l'algorithme est donc plus verbeuse que celle des langages OpenMP et HPF mais reste de haut niveau.

Le DAG de tâches est utilisé par ces modes d'expression pour modéliser finement les informations relatives à la cohérence et à la synchronisation du programme. De plus, ce modèle permet de contrôler et donc d'optimiser l'exécution à l'aide d'un ordonnancement.

Ainsi, grâce aux possibilités du support, ces modes d'expression ne contraignent pas le programmeur à implémenter un algorithme avec des squelettes (types de données, schémas de communication ou de synchronisation) prédéfinis. Par contre, la précision des informations stockées dans le DAG de tâches peut contraindre l'expression. C'est le cas pour RAPID et ATHAPASCAN que nous décrirons par la suite.

La difficulté dans la définition de ces modes d'expression utilisant un DAG est de leur donner la capacité à capturer le plus finement possible un maximum d'informations sur le parallélisme intrinsèque à un algorithme tout en ménageant l'aisance d'utilisation par le programmeur. Les synchronisations étant implicites, leur génération automatique nécessite la mise en œuvre d'un

inspecteur statique si elles sont prévisibles ou sinon d'un inspecteur dynamique. Bien souvent, l'inspecteur ne peut opérer seul, si bien que l'interface d'aide à l'inspection et le mode d'expression sont liés. L'inspection travaille sur des informations (données structurantes, annotations utilisateurs) présentes dans le code pour générer automatiquement les précédences entre les tâches. Ces dépendances peuvent juste modéliser la synchronisation nécessaire au bon fonctionnement du programme en mémoire partagée ou également décrire les transferts de données (cohérence) quand une mémoire distribuée est utilisée.

1.4.1 DAG de tâches statique

Dans le cas des programmes prévisibles, la connaissance du DAG de tâches intervient dans le pire des cas au début de l'exécution. Cette étape de découverte du DAG est antérieure à l'exécution parallèle du programme. Cela permet d'appliquer statiquement certaines heuristiques d'ordonnancement de la littérature sur les DAG (ordonnancement global et paramétré par des fonctions de coût des communications et des tâches [67])

1.4.1.1 RAPID

RAPID [30] est une bibliothèque rudimentaire datant de 1996 permettant d'exploiter un parallélisme de tâche en utilisant le langage C. Cette bibliothèque est composée d'un inspecteur statique, d'un ordonnanceur statique et d'un exécuteur parallèle. Elle est issue des recherches de Tao Yang et Cong Fu à l'université de Californie où elle fonctionnait sur les machines à mémoire distribuée (CRAY T3E et MEIKO CS-2).

Le code source d'un programme RAPID est divisé en deux parties : le code de la procédure de la tâche maître (description du flot de contrôle et des données), et les codes des procédures exécutés par les tâches filles (description des calculs). Par conséquent la création de tâches imbriquées est impossible.

Le programmeur doit utiliser les données parallèles proposées par RAPID sous forme d'objet ainsi que les primitives de RAPID pour exprimer le contrôle.

L'inspection consiste en l'exécution de la tâche maître. Cette dernière n'est exécutée qu'à cette occasion. Seules les tâches filles seront exécutées pendant la phase parallèle. L'inspection étant statique, la tâche maître ne doit travailler que sur des données structurantes sans faire d'effets de bord sur les données parallèles car ces dernières ne sont allouées qu'après l'inspection. Ce fonctionnement statique de l'inspecteur RAPID ne permet d'exploiter que des applications parallèles prévisibles.

L'inspecteur intercepte les déclarations concernant les tâches et les données qu'il enregistre dans un DAG généré lors de cette phase. Les dépendances du DAG sont uniquement déduites des déclarations d'accès aux objets RAPID (lecture, écriture, écriture commutative) pour chaque création de tâche. Ces dépendances garantissent l'équivalence avec une sémantique séquentielle.

Les paramètres des procédures des tâches ne sont pas utilisables par l'inspecteur pour générer des dépendances car ils ne sont associés à aucune donnée RAPID.

Avec le mode d'expression proposé par RAPID, le programmeur est déchargé des synchronisations et de la cohérence mémoire de son programme grâce au mécanisme d'inspection / exécution. Ce mode d'expression reprend une propriété du mode séquentiel qui est l'absence de synchronisations. En revanche, le programmeur doit explicitement découper son programme en tâches et déclarer les accès et les créations de données parallèles.

La création de tâches est signifiée à l'inspecteur par `task_begin()`, en spécifiant son nom avec une chaîne de caractères, un coût et des paramètres (cf. code 1.2). Ces paramètres sont des indices et sont au maximum trois. Ils servent comme nous le verrons par la suite à accéder aux données parallèles. La chaîne de caractères doit être associée à un pointeur de fonction avec `task_func()`. Le coût correspond à la durée de la tâche. Dans l'exemple ci-dessous la procédure associée à la tâche est `"factor_kk"`, sa durée est estimée par l'appel à la fonction du programmeur `TaskCost()` et elle prend deux paramètres `1` et `k`. Ensuite, les accès aux données sont déclarés (`task_read()`, `task_write()`) puis l'appel de tâche est finalisé par `task_end()`.

Listing 1.2 – interfaces tâches / données RAPID.

```
1  // [...]
2  task_func ("factor_kk", factor_kk);
3
4  for( k = 0; k < n; k++)
5  {
6    //[...]
7    task_begin ("factor_kk", TaskCost (k, k, k), 1, k);
8    task_read ("blk", 2, k, k);
9    task_write ("blk", 2, k, k);
10   task_end ();
11   //[...]
12 }
```

Les données parallèles sont au mieux des tableaux tri-dimensionnels de références à des objets unitaires RAPID ou les objets eux-mêmes. Ces objets sont enregistrés dans ces tableaux au moyen de l'appel à `object("tab name", sz, dim, coor1, coor2, coor3)`. Cet appel stocke dans la case [`coor1`, `coor2`, `coor3`] du tableau `tab name` une référence vers une future zone mémoire contiguë de taille `sz`. L'utilisation de l'espace d'indexage partagé tridimensionnel fourni pour chaque donnée parallèle RAPID est restreinte au maître. Les données parallèles ne peuvent être accédées que de façon restreinte (données unitaires sans possibilités de référencements) par les tâches filles ce qui éloigne ce modèle mémoire des facilités d'expression permises par une mémoire partagée. En effet, d'une part, un accesseur RAPID doit être utilisé : `void *get_dataaddr(char*tab_str, int dim, int coor_1, int coor_2,int coor_3)` pour obtenir le pointeur sur une donnée. D'autre part, le programmeur ne peut pas agencer ses données de sorte à bénéficier d'accès continus ou à pouvoir faire des changements de grain d'accès, car chaque donnée unitaire est allouée en interne par RAPID à des adresses potentiellement disparates. Dans les tâches filles, le programmeur a une vision des données bornée par chacun des objets qu'elles réclament avec `get_dataaddr()`. Il est impossible d'exprimer simplement des parcours de structure de données avec par exemple un grain d'accès variable.

Une autre différence avec la mémoire partagée concerne le contenu des données qui est limité aux données scalaires. De plus, RAPID ne fournit ni les interfaces ni les implémentations pour exploiter des chevauchements et des intersections entre des accès aux données parallèles.

La distribution des tâches et donc des données parallèles, est faite par l'appel à la fonction `set_cluster()` non documentée.

L'inspecteur RAPID ne travaille qu'avec des identificateurs (abstraction) car les données du niveau expression sont unitaires. Ceci simplifie l'inspection, car les données portées par les dépendances sont toujours des objets unitaires RAPID (données élémentaires régulières). Par contre, cette simplification empêche la description directe d'algorithmes aux dépendances portant des sous-données (forme d'irrégularité). L'inspecteur est néanmoins capable de capturer les schémas de synchronisations irréguliers.

D'autres contraintes d'expression liées aux accès par objet RAPID unitaire existent. En effet, dès que des regroupements, fragmentations de données ou chevauchements d'accès sont à décrire, de nouvelles données doivent être créées ainsi que des tâches remplissant ces nouvelles données à partir des anciennes. Cet inconvénient apparaît dans l'exemple de l'algorithme de factorisation de matrice creuse par la méthode de Cholesky écrit par les auteurs de RAPID (cf. Annexe A.3.1). La version de l'algorithme est à grain fin (2D) pour que les données parallèles accédées par les tâches restent unitaires. Dans ce code, l'affichage de la matrice résultat (qui nécessite un regroupement de données) est effectué en utilisant des appels MPI plutôt que le mode d'expression de RAPID. En effet, l'affichage de la matrice intervient une fois l'exécution du DAG terminée. Le programmeur est alors dans un mode classique SPMD. L'utilisation de ce mode de programmation SPMD, bien qu'incommode, est un moindre mal pour le programmeur.

Cet exemple de factorisation de Cholesky 2D de matrices creuses est la principale raison d'être de RAPID. Les performances obtenues par les auteurs sont bonnes pour cette version de l'algorithme prévisible où seules les synchronisations sont irrégulières.

Le DAG de tâches généré par l'inspection statique est ensuite ordonnancé séquentiellement par le logiciel PYRROS [32] et enfin utilisé par les processus exécuteurs de RAPID.

L'ordonnancement des tâches est calculé par PYRROS en utilisant l'algorithme DSC [33]. Un fichier est donc généré par le support RAPID pour PYRROS à partir du DAG de tâches interne de RAPID. Ce fichier ne contient que les informations nécessaires à l'ordonnancement d'un DAG, à savoir la structure et les coûts associés aux nœuds et aux arcs. Ce fichier ne peut donc pas être réutilisé pour amortir le coût de l'inspection. Par contre, le fichier produit par PYRROS contenant les vecteurs d'ordonnancement est réutilisable. Les modèles de coût utilisés sont simples car ils correspondent à ceux de PYRROS. Les coûts des tâches et des communications (taille des données RAPID) sont spécifiables dans le code de la tâche maître et pris en compte statiquement par l'ordonnanceur PYRROS. Le coût d'une tâche est défini dans le code 1.2 par le deuxième paramètre de `task_begin()`. Le coût d'une communication est calculé par RAPID à partir de la taille des données déclarées. L'ordonnancement des communications est inexistant. Il n'y a pas d'identification des tâches de gestion internes à RAPID et de fait celles-ci ne sont pas prises en compte lors de l'ordonnancement.

L'exécution parallèle se déroule au sein d'un nombre paramétrable de processus du support RAPID. Ce support fait bénéficier le programme de communications unilatérales performantes (ShMem). L'exécuteur utilise les numéros de tâches stockés dans le vecteur d'ordonnancement pour lancer une à une les procédures associées avec leurs paramètres propres stockés dans le DAG. Ces procédures s'exécutent de façon atomique. L'intégralité des données parallèles n'est effectivement allouée par le support RAPID que juste après l'inspection mais avant l'exécution parallèle avec `shmalloc()` sur CRAY. Cette allocation spéciale est impérative, car seules les zones mémoire retournées par cette fonction supportent les communications unilatérales (*put* / *get*). Elle est effectuée par tous les processus RAPID et n'est jamais libérée (durée de vie infinie). L'allocateur `shmalloc()` est un iso-allocateur, mais cette fonctionnalité n'est pas exploitée par RAPID pour simplifier le mode d'expression.

Enfin, le code séquentiel, libéré des appels à RAPID, ne peut plus être exécuté car les données font partie intégrante de RAPID. La mise au point des programmes en séquentiel nécessite la réécriture des créations des données parallèles du programme.

La vision des données unique offerte par RAPID permet d'exprimer aisément des programmes irréguliers mais la finesse de description des données est insuffisante. Les optimisations de stockage ne sont pas spécifiables. Par contre, les optimisations d'ordonnancements sont génériques et permettent de prendre en compte les caractéristiques prévisibles des algorithmes et des machines. Le support est performant grâce à l'usage des communications unilatérales, mais il est dépourvu de *threads* permettant de bénéficier des performances de la mémoire partagée des grappes de SMP.

1.4.2 DAG de tâches dynamique

Un modèle de DAG de tâches dynamique suppose sa prise en compte à la volée au fur et à mesure de l'exécution du programme parallèle. Ainsi, le contexte d'exécution (indicateurs de charge ou de performances) peut être utilisé pour répartir les tâches de façon adaptative.

Le DAG de tâches lui-même peut également être construit dynamiquement par un inspecteur. Dans ce cas, les programmes imprévisibles peuvent être modélisés au fur et à mesure de l'instanciation des valeurs structurantes obtenues en cours d'exécution. L'inspection à la volée en parallèle génère un surcoût de contrôle interne pour sa gestion en distribué (construction, accès mémoire et communications) lors de l'exécution parallèle. Ces tâches de gestion peuvent être intercalées à des moments d'inactivité du processeur ou du réseau pour perturber le moins possible l'exécution. Le futur de l'exécution n'est connu qu'au fur et à mesure de l'exécution de la partie contrôle du code utilisateur.

1.4.2.1 OVM

OVM [11] (Out-of-order Virtual Machine) est une bibliothèque proposant une interface client-serveur pour télécommander un programme MPI avec des RPC non bloquants. La configuration

d'exécution est un unique client séquentiel avec un serveur frontal sur lequel s'exécute un répartiteur également séquentiel et un ensemble de machines de calcul (cluster Linux). Un support MPI multi-processus s'exécute sur les machines de calcul sous les ordres du répartiteur avec lequel le client interagit.

Le code du client est un code MPI où les appels à cette bibliothèque et les boucles sont instrumentés par des requêtes sous forme de commentaires. Les calculs doivent être encapsulés dans des tâches appelables par le client à l'aide de RPC. Ces tâches ne peuvent faire que du calcul (interdiction de faire des requêtes OVM imbriquées). Ces requêtes sont expansées en appels de bibliothèque OVM par un *parser* PERL. Le code généré peut ensuite être compilé.

Les requêtes OVM permettent les créations, destructions et mouvements (*put* / *get*) de données entre les machines de calcul et le client. L'opération *get* est bloquante pour le client. Les caractéristiques des données parallèles implémentables avec OVM sont identiques à celles des données parallèles MPI en terme d'irrégularité et de continuité d'accès. Le contenu des données est limité à des variables scalaires (usage de la structure de données côté serveur impossible). Ces données sont passées en paramètres aux RPC avec la possibilité de les rendre persistantes. Les données de taille importante peuvent donc être localement sur le serveur ciblé par le RPC et non du côté du client (RPC découplés).

D'autres requêtes (RPC) correspondent directement à des communications collectives MPI (`gather`, `scatter`, `alltoall`, `reduce`, `broadcast`) à effectuer entre le client et les serveurs. Ces schémas de synchronisations permettent avec `put/get` d'avoir les mêmes possibilités d'expression de cohérence et de synchronisation que MPI. Le répartiteur intercepte toutes les informations relatives aux données et aux communications déclarées par le client pour produire un DAG de tâches à la volée. Ce DAG est une modélisation de la fenêtre d'exécution parallèle requise par le client. Cette fenêtre est plus ou moins vaste en fonction des opérations bloquantes côté client qui stoppent l'alimentation du répartiteur en requêtes à analyser.

OVM fournit un support pour identifier les tâches (RPC) et les communications dans le but de pouvoir les ordonnancer. La génération des dépendances de données par le répartiteur est directe car le client lui fournit toutes les informations dans son code via les appels MPI instrumentés.

L'absence de *threads* compromet la prise en compte des SMP. L'usage d'une bibliothèque MPI optimisée pour les SMP est un moyen de mieux les exploiter.

OVM est un mode d'expression d'usage ardu car la cohérence des données est explicitée via des RPC sur des fonctions collectives MPI (i.e. synchronisations explicites). Les optimisations effectuées avec OVM ont les mêmes caractéristiques que celles indiquées pour MPI sauf qu'un ordonnancement à la volée est fourni. Ce dernier ne permet pas d'exploiter les caractéristiques prévisibles des programmes. Les grappes de SMP sont exploitables mais les performances pourraient être limitées en scalabilité par le répartiteur centralisé.

1.4.2.2 ATHAPASCAN 1.8

ATHAPASCAN [26](Asynchronous Task Handling) 1.8 est une bibliothèque inspecteur / exécuteur à la volée datant de 1998 pour des programmes écrits en C++. La version 2.0 s'oriente vers une plate-forme pour la "grille" mais nous ne la décrirons pas car elle sort du cadre de cet

état de l'art. La version 1.8 de cette bibliothèque est issue des recherches de Gerson Cavalhueiro, Mathias Doreille, François Galilée, Thierry Gautier et Jean-Louis Roch à l'Institut d'Informatique de Grenoble où elle fonctionne principalement sur des clusters de PC Linux (mono ou bi-processeurs).

Le support ATHAPASCAN est capable d'exploiter une mémoire distribuée. L'API ATHAPASCAN permet d'exprimer le parallélisme d'un programme par son découpage explicite en tâches. La synchronisation des tâches et la cohérence des accès aux données de ces dernières sont automatiquement modélisées par le DAG dynamiquement généré par une inspection à la volée. La classe d'algorithmes traitée comprend ceux imprévisibles grâce à ce mécanisme d'inspection à la volée.

Les données parallèles doivent être implémentées avec des objets unitaires communicables proposés par ATHAPASCAN. Ces données parallèles sont des instances de classes C++ `Shared`. Les données de l'utilisateur sont portées par le champ `data` de ces classes. Elles ne peuvent contenir que des scalaires. L'utilisateur doit implémenter ces classes et peut définir la visibilité et la durée de vie de leurs instances. L'implémentation d'une classe `Shared` nécessite en particulier d'écrire le code d'empaquetage et de dépaquetage de l'objet communicable. Ceci s'explique par l'utilisation de communications point à point par le support. La vision des données au niveau expression s'arrête aux frontières de chaque objet. Cette vision peut être augmentée en reconstruisant la structure de données perdue du fait de la communication. Cette vision des données est limitée à l'intérieur d'un objet car l'allocation n'est pas maîtrisée par l'utilisateur. En effet, ATHAPASCAN a une approche donnée trop restrictive et abstraite qui oblige le programmeur à utiliser la méthode `access()` lors de chaque accès à une donnée parallèle. Cet accesseur doit être utilisé car de multiples versions (copies) d'une donnée peuvent exister en mémoire et de fait, l'usage des pointeurs est impossible car ils peuvent référencer des objets invalides. Cette limitation vient de l'absence d'un support mémoire adéquat telle par exemple une mémoire partagée. Néanmoins, le compilateur C++ permet de vérifier le bon usage des types et des accès des objets `Shared` ATHAPASCAN.

Une création de tâche est signifiée par le programmeur à l'inspecteur au moyen de l'appel de méthode `Fork` opérant sur une méthode de classe `Shared` prenant en paramètre un ou des objets `Shared` ATHAPASCAN. Les appels de tâches imbriqués et récursifs sont autorisés. Chaque tâche ne peut accéder qu'aux données qu'elle crée et aux données parallèles qu'elle reçoit en argument.

Le modèle mémoire d'ATHAPASCAN est selon leurs auteurs, celui d'une MVP (Mémoire Virtuellement Partagée) de niveau objet. Trois principales fonctionnalités font défaut à l'implémentation de ce modèle dans le support ATHAPSACAN par rapport à une mémoire partagée.

Premièrement, la manipulation de structures de données est difficilement réalisable en parallèle. En effet, aucun moyen de repérage ou de référencement global (pointeurs) des données n'est possible. Quand une structuration des données est utilisée, cette utilisation doit être cantonnée localement à l'intérieur d'une même tâche car elle n'est pas communicable et elle perdrait toute sa signification (perte de validité des pointeurs).

Deuxièmement, concernant l'expression directe des changements de grain d'accès (regroupements ou fragmentations de données). Ils nécessitent la création d'un ou plusieurs nouveaux objets et d'autant de tâches faisant des recopies de parties d'objets que le programmeur doit lui-même déterminer. Dans le cas de l'exemple de la factorisation LU de matrices ayant un découpage par blocs bi-dimensionnel (cf. Annexe A.4.2), la matrice semble être vérifiable en écrivant un code `_regroup` (lignes 230 à 252 du code A.29) puis en affichant la matrice sur la sortie standard. Comme dans le cas de RAPID, la recentralisation de la matrice est problématique. Cette méthode est utilisée car un code utilisant des tâches ATHAPASCAN est a priori très difficile à écrire. En effet, cette centralisation est impossible à faire directement (une seule tâche) car le nombre d'objets blocs est supérieur au nombre de paramètres d'une tâche. Le grain fixe des données parallèles d'ATHAPASCAN (idem pour RAPID) génère un *false sharing* et un surcoût en volume de communication lorsque les accès concernent des sous-données (accès partiels à la donnée unitaire). En effet, à chaque fois, la donnée est envoyée dans son intégralité. Par contre, le *false sharing* peut être amoindri sans pour autant être supprimé totalement (il reste des synchronisations qui n'ont pas lieu d'être) comme ce serait le cas si les accès partiels étaient gérés. Pour cela, il faut faire un usage détourné des accès commutatifs pour RAPID ou d'opérateurs commutatifs ATHAPASCAN. Mais l'accès est plus que commutatif : il est disjoint et cette solution imparfaite ne contraint pas l'ordre des opérations mais introduit autant de fausses dépendances qu'il y a d'accès commutatifs et donc autant de transferts de données dont la taille est surévaluée.

Enfin, troisièmement concernant les accès se chevauchant (e.g. accès de voisinage). Les données parallèles unitaires n'offrent pas la possibilité d'exprimer le parallélisme induit par de tels accès.

Lorsque ces trois fonctionnalités sont requises pour exprimer un algorithme, le mode d'expression d'ATHAPASCAN requiert l'écriture d'un code ardu et peu lisible. Par contre, l'utilisation du mode d'expression d'ATHAPASCAN est aisée lorsque l'algorithme à implémenter n'utilise pas de structure de données irrégulières et lorsque ses schémas d'accès mémoire sont réguliers.

L'expression d'un algorithme avec la bibliothèque ATHAPASCAN peut nécessiter un volume de code annexe important. L'écriture de l'algorithme d'une factorisation LU de matrices (code A.31 de l'annexe A.4.2) nécessite l'implémentation du module annexe `matrix` (contenant les opérations matricielles standards) spécifiquement pour ATHAPASCAN (codes de l'Annexe A.29 et A.30 de l'Annexe A.4.1). La difficulté réside dans l'implémentation des classes héritant de la classe `Shared` de la bibliothèque ATHAPASCAN. Une fois ce travail fait, l'algorithme de la factorisation LU s'exprime comme une version par blocs en séquentiel (cf. code A.31 présenté en Annexe A.4.2 lignes 49 à 63). La façon dont la distribution de la matrice est exprimée par la méthode `two_dim_partitionning_any_mode()` est compliquée (cf. lignes 159-180 du code A.31 de l'Annexe A.4.2).

Le développement est également difficile car le code séquentiel libéré des appels à ATHAPASCAN n'est plus compilable à cause des données qui sont des objets ATHAPASCAN. Ceci complique la mise au point du programme en séquentiel puis la transformation en un programme

ATHAPASCAN.

Le programme est modélisé au fur et à mesure de l'exécution par un DAG de tâches construit à la volée par un inspecteur. Ce DAG de tâches généré n'est pas récupérable par le programmeur, il ne peut donc amortir le coût de la construction du DAG pour un algorithme prévisible.

Les tâches s'exécutent de façon atomique (modèle du DAG) et nous avons vu qu'elles ne pouvaient accéder qu'à leurs arguments. Ceci permet à l'inspecteur de générer les dépendances entre ces tâches par simple connaissance de leurs arguments. Néanmoins, le programmeur doit déclarer les modes d'accès aux arguments de chaque méthode de tâche.

Un mode `post-poned` d'accès à une donnée informe l'inspecteur d'une absence d'utilisation immédiate mais d'une utilisation future potentielle. Sans ce nouvel accès et la condition que nous allons énoncer, le pouvoir d'expression du parallélisme serait moindre. Cette condition stipule que chaque tâche doit avoir reçu en paramètre, au minimum, l'intégralité des données accédées (hors création) par toutes les tâches de son futur sous-arbre des appels (elle compris). Cette condition est utile lorsque les tâches sont rajoutées dynamiquement dans le DAG au cours de l'exécution. Elle permet à l'inspecteur de savoir très tôt si deux tâches sont indépendantes ou non.

Toutefois, cela oblige le programmeur à prévoir avant chaque `Fork` (création de tâche) de passer en paramètre suffisamment de données `post-poned` pour garder un fonctionnement valide du support. Ceci nuit au parallélisme lorsque cette prévision n'est pas possible de façon exacte. En effet, dans ce cas il faut surestimer ces objets accédés avec une granularité par objet et attendre la terminaison du sous arbre pour se rendre compte que la ou les données post-poned dont dépendent d'autres tâches en attente, n'ont finalement pas été utilisées.

L'inspection est dynamique mais la génération du parallélisme nécessite donc de connaître statiquement au moment de la création d'une tâche, tous les futurs accès (hors création de données) du sous arbre des appels engendré par cette dernière.

Le coût des tâches est spécifiable et le coût des communications est calculé par la plate-forme d'après la taille des objets. Ces deux coûts sont pris en compte dynamiquement à l'exécution par l'ordonnanceur qui applique un algorithme de vol de travail [57]. Cet ordonnanceur est interchangeable avec un autre écrit par l'utilisateur. L'ordonnanceur à la volée n'a qu'une vision locale du DAG de tâches. Ce manque d'information peut être pallié par une implémentation dédiée de l'ordonnanceur à la volée dans laquelle le programmeur injecte une stratégie statique spécifique à son algorithme.

La migration des données en continu est possible et est utilisée par la politique de vol de travail de l'ordonnanceur à la volée. Cette stratégie peut générer potentiellement une redistribution continuelle des tâches anti-productive.

Le placement des tâches par l'utilisateur est optionnel. Lorsqu'il est spécifié, il est respecté par l'ordonnanceur. Du fait de l'implémentation dynamique du support d'exécution, le programmeur ne peut utiliser des algorithmes d'ordonnancement classiques de DAG. Il doit les écrire spécifiquement pour chacun de ses algorithmes.

La hiérarchie mémoire des SMP n'est qu'en partie exploitée par les *threads* du support. En effet, la mémoire partagée n'est pas pleinement exploitée car les données parallèles accédées par une tâche sont recopiées dans sa pile.

Le support utilise les communications point à point MPI sans en exploiter les communications collectives. De plus, ce modèle de communication ne correspond pas à celles unilatérales modélisées par le DAG de tâches.

Les performances sont faibles pour les applications prévisibles irrégulières [26](e.g. algorithme de Cholesky pour la factorisation de matrices creuses). ATHAPASCAN est dédié aux applications récursives imprévisibles de type "branch and bound". Ce type d'application produit de façon peu prévisible des tâches indépendantes ne nécessitant et ne générant pas de communications autres que celles venant de la tâche qui a réalisé le `Fork`.

La vision des données unique offerte par ATHAPASCAN permet d'exprimer aisément des programmes irréguliers mais la finesse de description des données est insuffisante. Les optimisations de stockage ne sont pas spécifiables. Les optimisations concernant les ordonnancements exploitant les caractéristiques prévisibles des algorithmes et des machines sont lisibles mais ne sont pas implémentables de façon générique. Le support utilise les *threads* permettant de bénéficier des performances de la mémoire partagée des grappes de SMP

Nous retiendrons le fonctionnement avec un code intégrant la description du DAG de tâches et servant également à l'exécution, ainsi que l'analyse de paramètres des tâches pour construire les dépendances.

1.4.2.3 Jade

Jade [54] est un langage datant de 1993 étendant le langage C et porté sur des machines parallèles à mémoire partagée (DASH Stanford Machine, SGI 4D/340) et à mémoire distribuée (réseau de stations de travail et Intel iPSC/860).

Jade adopte une approche compilée. Le langage C est étendu avec des mots clés pour identifier des portions de code comme des tâches, des données, des accès aux données ou des phases d'exécution.

Jade exploite un parallélisme de tâche explicitement décrit avec une vision des données partagée.

La tâche initiale du programme est la fonction `int start(int argc, char*argv[])` équivalente à la fonction `int main (int argc, char*argv[])` du langage C. Le programme démarre par l'exécution séquentielle de cette tâche.

Le mot clé `withonly` identifie une tâche par une portion de code entre accolades, déclare ses accès aux données parallèles et ses paramètres. Le mot clé `with` permet de changer les modes d'accès aux données parallèles déclarées précédemment en cours d'exécution de cette tâche.

Jade a été décrit par ses auteurs comme un langage permettant d'exprimer le parallélisme facilement. Les extraits de code présentés dans les diverses publications par exemple ceux concernant l'algorithme de factorisation de matrices creuses par la méthode de Cholesky ne correspondent pas au code compliqué de cet exemple fourni dans la distribution de Jade. Dans la thèse de Martin C. Rinard, le noyau de ce code prend 25 lignes alors que dans la réalité il fait 110 lignes et est

extrêmement confus (cf. code source A.32 en Annexe A.5.1). Cette incohérence est liée à l'élision volontaire par l'auteur, de toute la partie structure de donnée irrégulière de cet algorithme. Or, les données sont un point crucial dans un mode d'expression parallèle, nous ne pouvons donc pas discuter du mode d'expression de Jade.

Enfin, les performances indiquées par les auteurs sont décevantes [55, 56] pour les applications irrégulières (factorisation de Cholesky pour les matrices creuses) sur les machines à mémoire distribuée de l'époque.

Nous retiendrons ici la fonctionnalité permettant de travailler avec des sous-données.

1.5 Conclusion

Dans les modes d'expression visités ici, seuls ceux proposant une mémoire partagée permettent une programmation aisée et des stockages de données optimisés. La mémoire partagée fait défaut aux approches par DAG existantes car leur mode d'expression et leur support ont été définis avec une interface ne permettant pas de manipuler séparément l'allocation et la déclaration d'accès des données parallèles.

L'expression des synchronisations est explicite dans tous les modes d'expression n'utilisant pas le modèle du DAG.

Enfin, aucune des approches étudiées ne propose de fonctionnalités permettant l'ordonnancement des communications qui sont pourtant le principal goulet d'étranglement notamment pour les grappes de SMP.

Mises à part les approches RPC (e.g. PM2), les approches totalement explicites sont prévues pour la programmation d'algorithmes parallèles réguliers avec une partie du code contrôlant totalement le support d'exécution. Ce code permet d'avoir la finesse mais la programmation est fastidieuse notamment celle des algorithmes irréguliers. De fait, à ce niveau de programmation, les optimisations du programmeur concernent l'usage du support d'exécution. Elles sont peu génériques et peu lisibles, mais permettent quand même au programmeur d'injecter son savoir-faire pour exploiter les caractéristiques prévisibles des algorithmes et des machines.

Les approches compilées, sans mise en place de mécanisme inspection/exécution, sont prévues pour la parallélisation aisée d'algorithmes séquentiels réguliers et structurés par des données de niveau compilation avec une partie du code contenant des directives pour le compilateur. Ceci ne permet ni la facilité ni la finesse d'expression nécessaire aux algorithmes irréguliers. De fait, en l'absence d'interfaces, les optimisations du programmeur concernent majoritairement l'usage du compilateur ce qui les rend peu génériques et peu lisibles. Les approches compilées étudiées n'exploitent les caractéristiques des algorithmes structurés par des données de niveau inspection et exécution que par la mise en place d'un système d'inspection / exécution dynamique.

Le principal défaut des langages parallèles de haut niveau a été, lors de leur définition, de

confier un rôle trop important au compilateur ainsi qu'au support relativement à celui du programmeur. Dans l'état actuel des technologies des compilateurs parallèles, les modes d'expression parallèles ne peuvent ni se contenter de codes séquentiels quelconques ni d'abstraire trop la machine et le programme. Le fait de supprimer l'accès et le contrôle direct des données et l'accès aux processeurs empêche le programmeur d'appliquer un savoir-faire permettant d'exploiter les caractéristiques prévisibles des machines et des algorithmes.

Les approches partiellement explicites utilisant les DAG sont prévues pour la programmation aisée d'algorithmes parallèles réguliers et irréguliers découpés en tâches et structurés par des données de niveau inspection ou exécution avec une partie du code informant le support d'exécution. Ceci permet la facilité et la finesse d'expression nécessaires aux algorithmes irréguliers. De fait, les optimisations du programmeur concernent l'usage du DAG. Elles sont génériques et lisibles et permettent d'injecter son savoir-faire pour exploiter les caractéristiques prévisibles des algorithmes et des machines.

Néanmoins, les approches dynamiques avec un DAG n'exploitent pas pleinement les caractéristiques des algorithmes structurés par des données de niveau compilation et inspection car elles sont découvertes dynamiquement alors que cela aurait été possible de façon statique.

Chapitre 2

Mise en œuvre et schéma d'exécution de programmes parallèles avec PRFX

2.1 Introduction

Suite aux conclusions du chapitre 1, nous proposons un nouvel environnement défini de sorte à satisfaire les trois problèmes que nous avons identifiés dans l'introduction. Il permet au programmeur d'exprimer finement et aisément son algorithme même s'il est irrégulier. Cette nouvelle solution permet également au programmeur d'exprimer des optimisations lisibles et génériques de stockage des données (description de placements et de stockages contigus autorisant des types d'accès variables) et d'ordonnancement global des calculs et des communications exploitant les caractéristiques prévisibles des algorithmes et des machines. Notre solution inclut un support réalisant ces optimisations, paramétrables par le programmeur, auxquelles s'ajoutent celles internes au support pour les grappes de SMP.

PRFX permet d'adapter le déploiement (processus/*threads*) de l'exécution parallèle à la hiérarchie nœud/processeurs des grappes de SMP. Cette spécification peut être automatiquement générée ou être à l'initiative du programmeur. Les informations relatives à ce déploiement sont disponibles au niveau du mode d'expression. Le programmeur peut ainsi exploiter par exemple la hiérarchie mémoire des grappes de SMP.

Le mode d'expression est d'utilisation aisée grâce à la vision unique de la mémoire fournie au programmeur, au respect de la sémantique séquentielle et à la génération implicite des synchronisations. Par contre, la génération de la cohérence mémoire doit être partiellement explicitée par le programmeur.

Le programmeur a la même liberté de structuration du code en modules et de manipulation des données qu'en séquentiel (usage de bibliothèques de calcul, de pointeurs, d'allocations mémoire dynamiques, de type d'accès variable, de conditionnelles et de corps de boucles au contenu libre). Par contre, le programmeur doit décrire un algorithme parallèle adoptant une décomposition des calculs en tâches. Ces tâches induisent naturellement une modélisation de l'exécution du programme par un DAG de tâches qui sera généré automatiquement. Néanmoins, notre support

d'exécution requiert l'utilisation de données structurantes de niveau compilation ou inspection pour les parties du code (paramètres de boucles ou de conditionnelles) englobant des créations de tâches.

Une fois l'algorithme implémenté et compilé, arrive la phase d'exécution qui est optimisée. Une première optimisation consiste en l'exploitation des caractéristiques prévisibles via une exécution en deux phases : celle préparatoire et celle de l'exécution parallèle elle-même. Ce schéma permet de déplacer (voire de factoriser en cas de ré-exécution) une partie des coûts de gestion du parallélisme. Cette gestion est réalisée avec des tâches créées lors de la phase préparatoire (certaines pourraient être créées avant si nous disposions d'un compilateur capable de faire ce travail). Ces tâches de gestion correspondent à des fonctionnalités du support, leur coût est prévisible et est intégré à la préparation de l'exécution avec notre ordonnanceur statique. Ces tâches s'occupent de la gestion de la cohérence des données et des synchronisations de l'algorithme parallèle.

Ensuite, le support d'exécution entre en jeu. Il est capable de retranscrire sur la machine ce que le programmeur a indiqué au niveau du mode d'expression et d'appliquer des optimisations internes toujours en fonction du déploiement et des caractéristiques (via des modèles de coûts) des couches basses logicielles et de celles de la machine. Il utilise diverses optimisations pour les *threads* qu'il manipule (priorités différentes, *threads* utilisateurs/systèmes, *thread* en attente active par scrutation ou cession du quantum de temps) et également pour les communications unilatérales (communications non-bloquantes et communications collectives ou agrégées).

Enfin, le programmeur a un retour (e.g. sous forme de traces, d'indicateurs de performances) pour contrôler la réalisation effective des choix qu'il a exprimés concernant son algorithme parallèle.

Dans ce chapitre, nous allons donc expliquer nos choix pour la conception de PRFX en partant du besoin d'ordonnancement pour mettre en lumière l'organisation de l'API de PRFX et de son support. Après cette description des choix, nous décrirons le modèle mémoire avec ses possibilités en termes de structuration et de gestion de données, le modèle de programmation par tâche avec ses facilités pour l'expression de programmes irréguliers ou pour la mise en place d'un d'inspecteur et d'un ordonnanceur statiques, et enfin le modèle d'exécution parallèle avec ses optimisations pour les grappes de SMP. Pour les modèles mémoire et de programmation nous indiquerons les interfaces de PRFX correspondantes. Ces interfaces seront illustrées par des exemples simples mettant en jeu des données, des tâches et leurs coûts, des distributions de données, des heuristiques d'ordonnancement. Concernant l'inspecteur, nous décrirons le procédé utilisé pour obtenir les informations requises afin de produire un DAG de tâches complet (présenté à la section 2.3). Nous décrirons également comment le DAG complet est traité par l'ordonnanceur et l'exécuteur.

Nous terminerons ce chapitre par une méthodologie pour écrire un programme parallèle PRFX avec deux exemples d'implémentations d'algorithmes simples. Ceux-ci démontreront la facilité d'écriture et la lisibilité des codes utilisant notre mode d'expression. Ces exemples présenteront plutôt un faible niveau d'irrégularité (accès de voisinage pour l'un et *broadcasts* vers un nombre décroissant de destinataires pour l'autre). Les implémentations proposées auront la propriété d'être prévisibles et donc exploitables par le support PRFX.

2.2 PRFX : choix et concepts

Nous avons choisi une approche bibliothèque basée sur le langage séquentiel C. Ce langage est très portable et dispose de nombreux outils de développement annexes fiables. De plus, les compilateurs de ce langage sont capables d'exploiter spécifiquement les caractéristiques des architectures matérielles, ce qui assure les performances. Ce langage est facilement accessible, mais pour utiliser notre mode d'expression, le programmeur doit acquérir des compétences identiques à celles requises en programmation séquentielle pour l'optimisation des effets de cache mémoire.

Pour le mode d'expression, nous avons choisi un niveau d'expression où le programmeur conserve un rôle décisif. Nous lui demandons d'exprimer son algorithme avec seulement deux informations qui nous suffiront à modéliser finement le parallélisme. La première relève purement de l'expression alors que la deuxième est liée au fonctionnement de notre support. Nous verrons quelles sont ces informations ci-après. Nous lui demandons de choisir éventuellement des optimisations ou paramétrages d'exécution à utiliser de façon externe à son algorithme. Ces informations d'optimisation sont clairement communiquées par une API (*Application Programming Interface*).

Les choix faits pour notre mode d'expression sont principalement conditionnés par les besoins qu'ont les programmeurs d'algorithmes scientifiques d'exprimer toutes les propriétés de ces derniers le plus simplement possible. Les choix faits pour implémenter le support de notre mode d'expression sont, quant à eux, principalement conditionnés par un autre besoin des utilisateurs d'exécuter ces algorithmes sur des grappes de nœuds SMP de façon performante.

Les algorithmes scientifiques peuvent être réguliers ou irréguliers. Or nous nous devons de fournir les moyens d'exprimer tous les algorithmes scientifiques, et nous savons que la description par tâches permet une description élémentaire et donc fine du parallélisme qu'il soit régulier ou irrégulier. Nous choisissons donc ce modèle de programmation par tâches et le découpage du programme en tâches est la première information demandée au programmeur. Ces tâches sont lancées avec un mécanisme de type RPC. De plus, pour que le mode d'expression soit aisé, nous maintenons le plus grand nombre possible de caractéristiques de l'expression séquentielle. Nous y parvenons en conservant tout d'abord l'aspect procédural et la sémantique parallèle équivalente à celle séquentielle. Ensuite, nous choisissons un espace d'adressage unique qui va permettre d'exploiter les possibilités du langage C, notamment d'accéder de façon continue aux données parallèles, de gérer (allocations dynamiques) et de structurer ces dernières avec des pointeurs. Ceci permet également d'implémenter directement en données parallèles des structures de données logiques ou issues du monde séquentiel.

Enfin, nous faisons en sorte que notre mode d'expression ne nécessite que la description par le programmeur des synchronisations de contrôle existant entre les tâches (arbre des appels). Ceci permet d'alléger sa charge de travail et de rendre le code source plus lisible. Nous supposons que les synchronisations restantes peuvent être déduites automatiquement à partir de cette base par des outils de compilation ou d'inspection. Dans notre cas, comme nous le verrons ci-après, nous avons choisi un inspecteur statique. Cet inspecteur sera implémentable grâce à une deuxième information (relative aux données accédées par les tâches) que nous demanderons au programmeur, et grâce à la restriction à la classe de programmes prévisibles.

En ce qui concerne les performances, le code peut être optimisé de façon externe via les options du compilateur C et également par l'usage de bibliothèques de calculs ou autres. Ces bibliothèques ont l'obligation d'être *thread safe* ; elles sont autorisées à créer leurs propres *threads* (e.g. IBM ESSLSMP).

D'autre part, nous souhaitons proposer des exécutions performantes de ces algorithmes sur des grappes de SMP. Or, ces machines sont complexes et l'utilisation de leurs moyens de communication est le principal goulet d'étranglement pour les performances. Or, l'usage des *threads* permet de bénéficier de la performance de la mémoire partagée entre les processeurs d'un même nœud. De plus, les communications unilatérales sont les plus performantes actuellement car elles sont sur certains matériels de plus bas niveau que les communications classiques par passage de messages, et parce que l'absence d'interlocuteur économise des ressources. Enfin, on sait que la performance de ces algorithmes est intrinsèquement liée à l'ordonnancement de leurs calculs et de leurs communications, et au placement de leurs données. Les heuristiques d'ordonnancement classiques supposent un nombre de processeurs constant. Nous nous devons donc d'ordonnancer les tâches et les communications dans un contexte d'exécution avec une configuration statique de processus "*threadés*" utilisant des communications unilatérales.

Le choix de forcer le programmeur à découper lui-même l'algorithme en tâches va se révéler utile. Ces tâches étant identifiées par le programmeur, le découpage de l'algorithme peut être optimal car intelligent et adapté au problème et / ou à la machine. Ensuite, il suffit d'ajouter une propriété d'atomicité des tâches (absence de synchronisations bloquantes) et une notion de coût d'exécution pour pouvoir appliquer des heuristiques d'ordonnancement décrites dans la littérature. L'ordonnancement des tâches est un moyen d'action externe et générique sur les performances.

Les coûts des tâches et des communications doivent être paramétrés par un modèle de coût du support qui dépend de la machine utilisée. Nous choisissons de faire spécifier le coût des tâches par le programmeur (cf. partie gauche de la figure 1). Le coût des communications est automatiquement calculé par l'ordonnanceur d'après les informations liées à la cohérence des données fournies par l'inspecteur. Ces coûts de communications nécessitent la connaissance par l'ordonnanceur du déploiement décrivant combien de processus (et combien de *threads* par processus) participeront à l'exécution parallèle, sur quelle machine (e.g. un nœud SMP) chaque processus sera exécuté et sur quel processeur chaque *thread* évoluera. Nous choisissons de paramétrer l'ordonnancement et donc l'exécution par un fichier contenant toutes ces informations (cf. haut de

la figure 1). Ce dernier peut être généré automatiquement en fonction du problème, mais son écriture est à la charge du programmeur. Dans notre implémentation, l'algorithme d'ordonnancement est choisi par le programmeur. L'ordonnanceur produit des vecteurs de tâches. Chacun de ces vecteurs est exécuté par un *thread* identifié par un numéro. Ces *threads* sont appelés ***threads* exécuteurs**. Ces vecteurs de tâches peuvent être stockés dans un fichier pour être éventuellement réutilisés et ainsi amortir le coût de l'ordonnancement (cf. boîte ordonnanceur au centre de la figure 1). Ces vecteurs de tâches contiennent aussi bien des tâches de l'utilisateur (e.g. tâches de calcul) que des tâches de contrôle internes à PRFX. Nous faisons remarquer que le placement, l'ordonnancement, les fonctions de coûts et le déploiement qui sont ici paramétrables, sont des moyens de prendre en compte l'hétérogénéité des coûts intervenant dans une grappe de SMP.

Il nous reste maintenant à choisir un ordonnancement soit à la volée soit statique. Le choix d'un ordonnancement statique est en fait plus vaste car il peut être réalisé par phase ou par ré-ordonnancement dynamique ponctuel. Nous éliminons l'ordonnancement à la volée car il ne permet pas une prise en compte globale du problème sauf si le programmeur écrit autant d'ordonnanceurs qu'il a de programmes prévisibles différents à ordonnancer selon ses desiderata. Avec des heuristiques classiques, le mode à la volée peut se révéler peu performant, notamment pour les programmes irréguliers prévisibles. Cette approche est celle choisie dans le support ATHAPASCAN (partiellement car les communications ne sont pas ordonnancées).

Comme nous souhaitons permettre l'obtention de performances, nous optons pour un ordonnancement statique qui pourra prendre en compte les programmes prévisibles quelle que soit leur irrégularité. Avec ce type d'ordonnancement, un fonctionnement par phase est certes intéressant mais moins direct à implémenter dans un premier temps ; nous le laissons donc de côté sans nous l'interdire car ce n'est rien de moins qu'une succession d'ordonnancements statiques. Le ré-ordonnancement dynamique ponctuel est conceptuellement intéressant car plus les durées d'exécution sont importantes, plus les approximations, faites sur les modèles utilisés par l'ordonnancement statique initial, éloignent les prévisions théoriques des observations pratiques. Nous le laissons également comme perspective, mais nous le prévoyons en autorisant la migration préemptive des tâches de calcul grâce au choix de la technologie *iso-mémoire*. La volonté de faire un ordonnancement statique initial va nous obliger à restreindre la classe de programmes supportés. Notre support sera donc limitatif par rapport à la classe de programmes parallèles génériques exprimables via notre mode d'expression. En effet, pour ordonnancer l'intégralité d'un programme, la majorité des heuristiques d'ordonnancement de la littérature suppose l'existence de l'intégralité d'un DAG de tâches étiqueté avec des coûts pour les tâches et les communications. Le fait de vouloir ordonnancer, en bénéficiant de ces heuristiques, nous conduit donc à choisir une modélisation du programme par un DAG de tâches. A ce propos, nous faisons remarquer que les besoins pour réaliser les communications décrites dans le DAG rejoignent exactement les services fournis par les modèles de communications unilatérales et les données parallèles en iso-mémoire. Avec cet ensemble de modèles, nous pourrons donc implémenter notre support de façon cohérente. Notre modélisation du programme sera plus riche puisqu'elle consistera en un DAG complet. Cette modélisation autorise également l'application d'optimisations externes et paramétrables telles que la détection de schémas de communications collectives, la duplication de tâches ou d'autres transformations comme le changement de grain par fusion des tâches, la limitation de la mémoire consommée par les tâches. Pour notre part, nous détectons les envois de

données multiples à un même destinataire (messages agrégés) et les envois de la même donnée à de multiples destinataires (*broadcast*).

L'écriture par le programmeur de la description du DAG complet modélisant l'exécution du programme serait une tâche fastidieuse notamment pour les algorithmes irréguliers que nous souhaitons traiter. Dans certains cas, des formules analytiques existent et permettent de générer ces DAG, mais nous préférons fournir un outil générique au programmeur permettant leur génération automatique. Nous utilisons donc un inspecteur statique qui intervient dans une phase de pré-exécution du programme où les données de niveau inspection sont disponibles. Un autre élément décisif concerne la prise en compte des programmes imprévisibles en sus des programmes prévisibles. Nous verrons à la section 5.2 comment étendre la classe des programmes traités à celle des programmes quasi-prévisibles sans faire toutes les concessions de performances induites par le support de la classe des programmes imprévisibles. Un autre atout de l'inspection statique est de ne pas perturber l'exécution parallèle comme le feraient des tâches d'inspection à la volée. La phase d'exécution ne contient que du code utilisateur et les tâches internes nécessaires à la gestion du contrôle et des communications ce qui rend les modèles de coûts plus précis (effets de cache conservés) et donc l'exécution potentiellement plus performante.

Cette classe des programmes prévisibles est néanmoins relativement vaste dans le cas des programmes scientifiques et donc le choix de faire une inspection suivie d'un ordonnancement statique est raisonnable.

L'inspection (cf. centre de la figure 1) va générer toutes les synchronisations (hors contrôle déjà décrit par le programmeur via le mode d'expression) et également la cohérence mémoire. Dans tous les cas de déploiement, mono-processus ou multi-processus, l'inspecteur travaille en faisant des intersections sur les accès mémoire. Le choix d'un espace d'adressage unique se révèle ici salvateur car toutes les intersections d'accès pourront se faire directement. L'inspecteur en tant qu'outil pour l'ordonnanceur se révèle donc aussi un allié pour la simplicité du mode d'expression et la lisibilité, car le code n'est pas entrelacé de calculs et de communications ne respectant pas les barrières modulaires du langage. De plus, le modèle de tâche est une fois de plus utile pour faire fonctionner l'inspecteur à un niveau macroscopique par tâche et non pas au niveau instruction. Pour cela nous avons besoin d'édicter des contraintes sur les accès aux données réalisées par les tâches qui sont dans notre cas associées à la fonction cible d'un RPC. Ces contraintes concernent le confinement des accès qui est semblable à celui intervenant dans une programmation par RPC (accès limités aux arguments de la fonction appelée).

L'inspection du code source du programme est impossible dans le cas d'un usage non contraint du langage C, notamment à cause des pointeurs. Nous avons donc opté pour une méthode simple qui consiste en une pré-exécution légère du programme mais suffisante pour le modéliser avec un DAG complet. Lors de cette pré-exécution légère, il faut au moins que toutes les tâches "non feuilles" (i.e. les tâches du contrôle de l'algorithme) de l'arbre des appels soient exécutées. La discrimination entre les tâches utiles ou non lors de la pré-exécution est effectuée par le programmeur car l'inspecteur ne peut avoir la compétence du programmeur en ce domaine.

L'inspection a un coût qui est faible si l'on suppose que la majorité des calculs se trouve dans

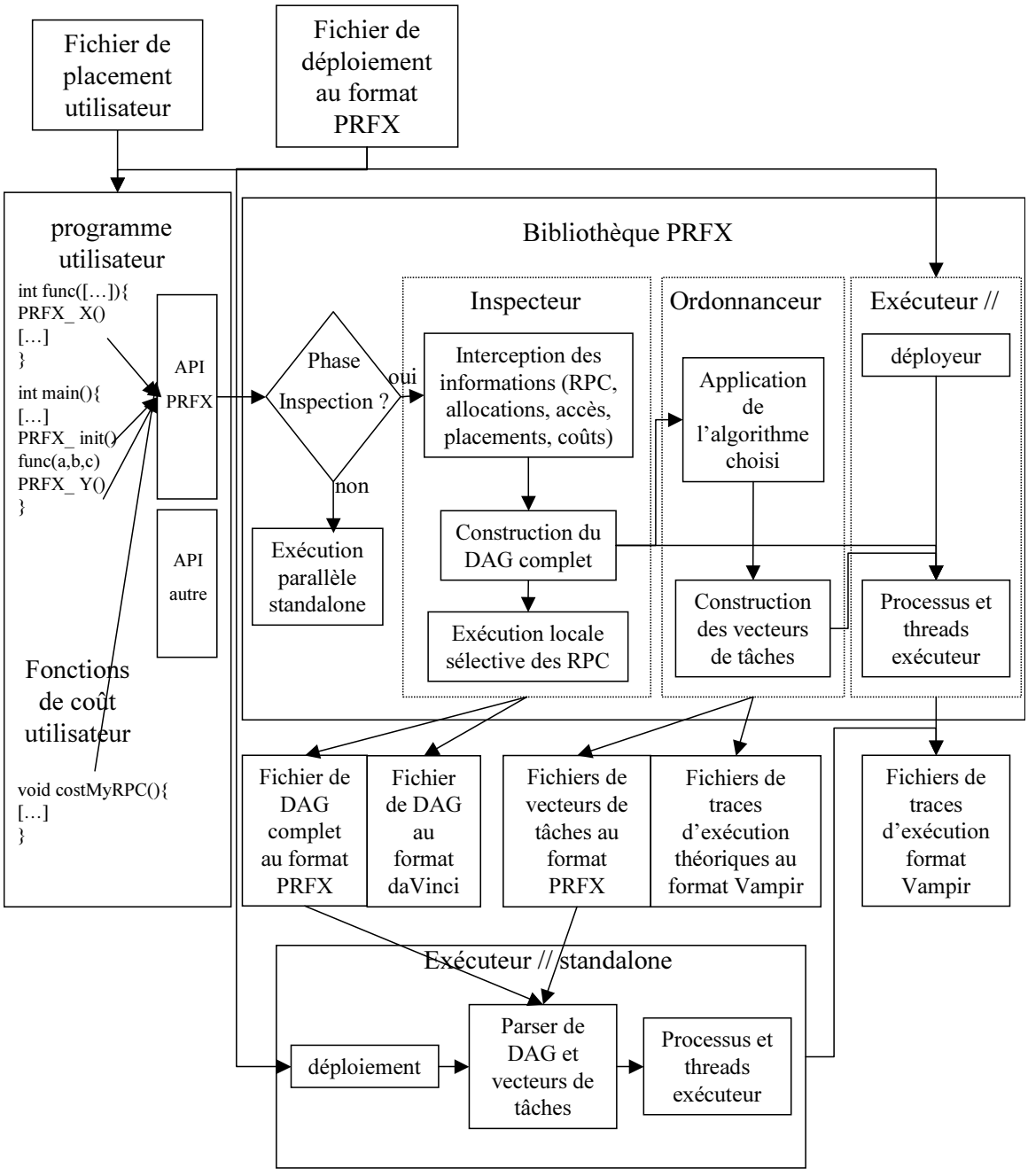

FIG. 1 – Interaction du programme utilisateur avec la bibliothèque PRFX.

les feuilles de l'arbre des appels. De plus, ce coût peut être amorti, dès lors que le même DAG est réutilisable pour plusieurs exécutions (e.g. programmes prévisibles dont seul le contenu des données parallèles change). Ces deux suppositions sont satisfaites par de nombreux algorithmes scientifiques notamment en algèbre linéaire (e.g. algorithme de factorisation de Cholesky de matrices pleines). Dans PRFX, le travail de l'inspecteur et de l'ordonnanceur est réutilisable grâce au stockage du DAG complet et des vecteurs d'ordonnancement dans des fichiers. Ces fichiers sont ensuite lus par un exécuteur à part (cf. bas de la figure 1).

Maintenant que nous avons choisi d'inspecter le programme par une pré-exécution légère, l'inspecteur doit récolter les informations essentielles lui permettant de construire automatiquement le DAG complet dont l'ordonnanceur puis l'exécuteur auront besoin. Nous choisissons de faire communiquer ces informations explicitement par le programmeur car cette méthode est cadrée et fiable. Pour ce faire, nous définissons une interface (cf. API PRFX à gauche de la figure 1). Ces informations concernent entre autres, le placement des tâches (provenant par exemple d'un fichier cf. en haut à gauche de la figure 1), l'enregistrement des fonctions cibles de tâches (RPC), la déclaration des créations de tâches et des créations / destructions de données parallèles, la description de la géométrie des accès aux données parallèles (via les *stencils* que nous décrirons à la section 2.3.2.2) et de leurs modes d'accès. Les informations relatives aux données constituent le deuxième type d'information que le programmeur doit fournir.

En principe, le fait de faire calculer les dépendances par un inspecteur ne nuit théoriquement pas à la finesse d'expression. En effet, la granularité des tâches et des accès analysables et discernables par l'inspecteur est suffisante pour les algorithmes scientifiques.

L'exécuteur (cf. droite ou bas de la figure 1) parallèle se déploie au sein de processus et de *threads* exécuteurs dont le nombre et la localisation sur les processeurs et les machines sont indiqués par l'utilisateur dans un fichier. L'exécuteur réalise l'exécution théoriquement prévue en respectant l'ordonnancement des calculs et des communications précédemment calculé. L'exécution d'un programme utilisant notre support PRFX nécessite une préparation des structures de données utilisées par PRFX. Cette initialisation intervient lors de l'appel à `PRFX_init()` par chaque processus PRFX. Cette initialisation peut provenir de l'inspection, auquel cas elle est directement intégrable au niveau de l'exécuteur. Sinon, elle peut provenir de fichiers nécessitant l'action d'un *parser* dans le cas de l'exécuteur parallèle *standalone*.

Nous avons choisi d'utiliser peu de couches logicielles pour conserver un fonctionnement du support modélisable. Ce faible nombre de couches logicielles assure également la pérennité de l'ensemble et une exposition moindre aux défauts de fonctionnement latents des couches subordonnées.

Les choix que nous avons faits concernant les couches logicielles, servant à bâtir notre support PRFX concernent tout d'abord le système d'exploitation. Ce dernier est AIX (IBM), il a fait ses preuves dans le monde du parallélisme hautes performances. Bien que ce système soit propriétaire et que son fonctionnement et son paramétrage soient uniquement décrits par une documentation, il a l'avantage de supporter les conditions de production drastiques et intenses sur des grandes grappes de SMP.

Nous tirons parti des bénéfices de la mémoire partagée à l'intérieur d'un nœud SMP en utilisant la bibliothèque de *threads* POSIX. Nous exploitons la performance des communications unilatérales avec la bibliothèque propriétaire IBM LAPI [38] qui est dédiée au matériel de communication IBM (*switch Colony* et surtout *Federation*).

Enfin, pour avoir un retour d'information et pour pouvoir vérifier le bon fonctionnement de PRFX, nous proposons des modules annexes permettant d'utiliser des outils logiciels. Dans la figure 1, ces deux modules annexes apparaissent en bas. Le premier sert à la génération des traces d'exécution théoriques prévues et des traces réelles. Ces traces sont exploitables par le logiciel de visualisation de traces VAMPIR (Pallas [52]). Le deuxième module permet de produire un fichier représentant le DAG étiqueté avec quelques informations pour le logiciel de visualisation de graphes DaVinci (B-novative [7]). Ce logiciel est utilisé par de nombreuses universités et sociétés commerciales en particulier par 'Intel Hillsboro Oregon / USA", " Oxford Parallel Applications Centre" et NASA's Jet Propulsion Laboratory. Ces modules annexes servent d'aide au développement et à la compréhension du déroulement de l'exécution.

2.3 Le mode d'expression du parallélisme

Un programme parallèle PRFX s'écrit avec un code séquentiel en langage C optimisé comportant des appels à la bibliothèque PRFX. L'optimisation requise ici est celle effectuée lorsque l'on désire exploiter les effets de cache en séquentiel. Pour cela, il faut découper le programme en tâches élémentaires opérant sur des volumes de données rentrant dans le cache du processeur. Dans notre cas, seul le découpage importe car il est l'élément permettant l'extraction du parallélisme. Le fait que les données rentrent dans les caches est un plus. L'expression du parallélisme passe donc par l'identification de tâches, librement par le programmeur (choix du début et de la fin du code de chaque tâche), sans que ce dernier n'ait à se soucier de la gestion des synchronisations nécessaires pour garantir la même sémantique qu'en séquentiel. Cet ordre est naturel pour le programmeur car il correspond à celui exprimé dans le code source.

D'un point de vue algorithmique, l'expression du parallélisme passe d'abord par la définition des données parallèles, leur gestion dynamique (allocation, libération) et leur chaînage (e.g. avec des pointeurs) en iso-mémoire. Grâce à cette iso-mémoire et aux accès continus qu'elle autorise, le programmeur peut faire varier simplement et à loisir le grain des accès au cours de l'algorithme.

Cette expression du parallélisme est indirecte et donc potentiellement infructueuse. Elle est indirecte car à aucun moment le programmeur ne dit explicitement que deux tâches ou plus sont indépendantes. Ce choix évite d'avoir à fournir une interface pour exprimer les schémas de synchronisation rencontrés, par exemple, dans les algorithmes irréguliers. L'absence ou la présence de parallélisme entre ces tâches est directement et automatiquement détectée par des calculs d'intersections sur les accès aux données décrits par des plages d'iso-adresses. Si deux tâches accèdent à des données s'intersectant, et que l'une d'entre elles effectue un accès exclusif (e.g. écriture), elles sont automatiquement synchronisées selon l'ordre d'exécution séquentielle du programme.

Une autre partie de l'expression est purement informative et est à destination de l'inspecteur pour l'aider à générer le DAG complet modélisant le programme utilisateur. Nous considérons que le **DAG complet** est un DAG dont les nœuds modélisent les tâches de calcul (ensemble d'opérations) et les arcs, les dépendances de données (synchronisation, communication) entre les tâches. Chacune des tâches est étiquetée par son coût, la liste des tâches qu'elle engendre, ses accès aux données parallèles, ses allocations et libérations de données parallèles. Chacune des dépendances de données satisfait tout ou partie des accès d'une tâche. L'ensemble des dépendances de données entrantes sur une tâche satisfait la totalité des accès de cette dernière. Une dépendance de donnée porte une information structurelle de synchronisation des tâches (tâche émettrice / réceptrice) et une information sur la conformation de la donnée parallèle et de sa localisation permettant son transfert pour assurer sa cohérence. Le DAG complet est une modélisation valide de l'exécution des algorithmes parallèles, sauf lorsqu'il y a un indéterminisme d'accès (e.g. accès commutatifs).

L'inspecteur stocke donc dans le DAG complet des informations telles que les iso-allocations et libérations des données parallèles, les modes (lecture, écriture,...), la géométrie (*stencils*) des accès associés aux arguments des RPC ainsi que des informations pour l'ordonnanceur : la phase et le placement conseillés pour chaque tâche par le programmeur.

Il est de bon aloi que le programmeur ne décrive pas tous les accès de toutes les fonctions C de son programme, mais se restreigne à un niveau de granularité et à un nombre de fonctions (RPC) pour lesquels PRFX saura gérer efficacement le parallélisme. Par contre, pour conserver un fonctionnement valide de son programme, il doit encapsuler, dans des fonctions (RPC) englobantes, les parties séquentielles de son programme et en décrire les accès. En effet, durant toute la vie du programme utilisateur, il ne doit y avoir que du code exécuté par des RPC via PRFX même si ce code est séquentiel.

Il faut noter qu'en aucun cas un programme séquentiel existant ne peut être directement parallélisé par l'ajout d'appels à la bibliothèque PRFX. Seule l'implémentation parallèle d'un algorithme parallèle est facilitée par le mode d'expression de PRFX.

2.3.1 Modèle iso-mémoire

Contrairement aux fonctionnalités déclaratives des sections suivantes qui sont un investissement du programmeur pour exprimer son algorithme parallèle et le faire analyser par l'inspecteur, les fonctionnalités de l'iso-mémoire autorisent la migration de *thread* [4] et simplifient la structuration des données et de leurs accès. Le choix de la mise en valeur de l'iso-mémoire depuis l'expression des accès des algorithmes jusqu'à l'implémentation des communications unilatérales est donc un élément clé pour la facilité d'expression et l'obtention de performances.

Sans la technologie logicielle de l'iso-mémoire, le support PRFX devrait se restreindre à des machines à mémoire partagée, ou utiliser une mémoire partagée logicielle (SDSM), ou utiliser directement la mémoire distribuée mais avec des contraintes d'expression. Les grappes de SMP (sauf celles offrant une CC-NUMA de type SGI Origin) n'offrent par défaut pas de cohérence mémoire entre les nœuds. Se limiter à des machines parallèles proposant une cohérence matérielle

est donc trop restrictif. La solution logicielle avec une SDSM n'est pas une approche adaptée car son fonctionnement est intrinsèquement coûteux et difficilement modélisable pour être pris en compte par une heuristique d'ordonnancement. Intéressons-nous aux problèmes liés à une solution logicielle utilisant directement (i.e. sans passer par une SDSM) la mémoire distribuée. Dans ce cas, pour continuer à fournir au programmeur un service de type mémoire partagée, le support devrait parcourir les données du programmeur à la recherche de pointeurs pour les modifier afin de conserver la structure des données lors des transmissions entre processus. Cette modification des pointeurs est impossible à faire car pour une zone mémoire donnée, le langage C ne permet pas de les discriminer. Pour résoudre ce problème, on peut obliger le programmeur à enregistrer les emplacements des pointeurs, mais les pointeurs contenus dans les registres et les optimisations des compilateurs limitent cette méthode. Un autre moyen consiste à interdire au programmeur de structurer ses données avec des pointeurs (cas des objets communicables RAPID et ATHAPASCAN pour lesquels la vision des données parallèles est réduite).

Nous avons décidé que ces limitations relevant de l'expression sont plus handicapantes que les limitations matérielles de l'iso-mémoire que nous allons détailler ci-après. Nous rappelons qu'en particulier au niveau expression, l'iso-mémoire permet la structuration des données avec des pointeurs et une vision des données illimitée. Notre solution mise en œuvre dans PRFX consiste donc en l'utilisation d'une iso-mémoire et d'un iso-allocateur (équivalent à malloc()) associé.

Cette technologie est bornée en scalabilité mémoire car la quantité de mémoire disponible pour le déroulement de l'exécution parallèle est fixe (portion de l'espace d'adressage). Ainsi, pour des espaces d'adressages 32 bits (4 Go), il suffit d'utiliser deux machines ayant chacune 2 Go de mémoire pour atteindre la limite de scalabilité mémoire de cette technologie. Le fait d'ajouter des machines à cette configuration ne permettra pas de traiter des problèmes plus grands. Pour repousser cette limitation, il faut utiliser une architecture matérielle 64 bits. Alors, dans ce cas, si l'on considère toujours des machines possédant 2 Go de mémoire, la limite de scalabilité se situe à 8 milliards de machines. Ceci est une valeur théorique, car les systèmes d'exploitation 64 bits ne sont pas toujours capables de fournir la totalité de l'espace d'adressage aux processus. Cette limite de scalabilité mémoire est la même que celle rencontrée dans un paradigme utilisant la mémoire partagée. Dorénavant, nous ne nous intéresserons donc qu'à l'iso-mémoire 64 bits.

L'iso-mémoire n'est pas une mémoire partagée distribuée mais un espace d'adressage commun car aucune cohérence n'existe par défaut. La combinaison du support PRFX (qui assure entre autres la cohérence des données du programme) et de l'iso-mémoire fournit au programmeur une mémoire partagée en distribué sans son mode de fonctionnement coûteux. Nous verrons que cela est possible grâce à la génération des informations relatives à la cohérence des données en iso-mémoire par le système d'inspection et leur utilisation par l'exécuteur. Utiliser un espace d'adressage commun n'est pas coûteux, cela revient à utiliser le même système de coordonnées dans l'espace. En effet, après une concertation initiale, les informations extérieures (coordonnées, adresses, etc...) sont intégrables par chacun de façon autonome. Dans le cas des ordinateurs, l'étape initiale est déjà effectuée grâce au concept identique de mémoire virtuelle par processus 64 bits qui s'étale sur le même espace d'adresses de 0 à 2^{64}-1.

Maintenant que nous avons une iso-mémoire, il nous faut y stocker des données pour pouvoir

effectuer des calculs. Pour cela nous avons besoin d'un allocateur dynamique d'iso-mémoire : un iso-allocateur. Cet iso-allocateur permet à un processus de réserver un intervalle d'adresses en iso-mémoire (iso-allocation). De fait, cet intervalle ne doit pas alors pouvoir être réservé par les autres processus. A priori, ceci nécessite des synchronisations avec tous les processus à chaque iso-allocation ou iso-libération. Ces synchronisations peuvent être évitées en distribuant initialement l'iso-mémoire entre les iso-allocateurs. Chaque iso-allocateur opère sur un espace d'iso-allocation privé. Mais cette stratégie entraîne une deuxième limite de scalabilité à l'iso-mémoire. En effet, l'espace d'iso-allocation privé par processus est une partition de l'iso-mémoire qui dépend de sa distribution. Par exemple, si l'iso-mémoire est distribuée entre les p processus de façon équitable, alors la quantité de mémoire adressable disponible pour un programme faisant ses allocations de façon centralisée est $\frac{2^{64}}{p}$ octets. Le numérateur est suffisamment grand pour que cette limitation soit raisonnable même si une machine avec des milliers de nœuds est utilisée. Ceci conforte le choix de cette méthode qui ne nécessite aucune communication lors d'une iso-allocation. La répartition de l'iso-mémoire est spécifiable dans le fichier de déploiement que nous verrons à la section 2.6.3. Nous détaillerons l'implémentation de cette méthode au Chapitre 3. Pour éviter ce problème de scalabilité des iso-allocations, il faut mettre en place un iso-allocateur capable d'emprunter aux autres iso-allocateurs leur propre espace d'iso-allocation [4]. Ces négociations sont bien entendu synchronisantes.

Avec l'iso-allocation, les données de l'algorithme peuvent donc être structurées avec des pointeurs. Ces pointeurs et structures de données ne nécessitent pas de déclarations spéciales (typage) ni ne subissent de transformations. Ils sont toujours valides même après migration depuis un processus vers un autre (et entre processeurs compatibles). Les structures de données peuvent chaîner des données iso-allouées par des processus différents.

Lorsque la pile d'exécution est également iso-allouée, cela autorise la migration préemptive du flot d'exécution (contexte du *thread*) si les processeurs et la bibliothèque de *threads* de la source et de la destination sont semblables.

La garantie d'unicité de l'emplacement d'une donnée parallèle en iso-mémoire fournie par l'iso-allocateur permet d'envoyer les données à distance avec des communications unilatérales (`Put()`). Ces communications ont un modèle de fonctionnement intrinsèquement moins synchronisant que celui du passage de messages (nécessité d'un interlocuteur). Elles ne nécessitent pas la préparation de zones de réception (possibilité de zéro copie sans entente préalable) car comme nous l'avons vu, les zones mémoire obtenues par iso-allocation sont initialement disponibles dans tous les iso espaces d'adressage de tous les processus.

2.3.2 Modèle de données

Au cours du déroulement d'un programme PRFX, les données utilisées sont classées en deux catégories distinctes. Les premières sont les données parallèles ayant une durée de vie potentiellement longue, résistant à des transferts inter RPC. Le programmeur les gère, les structure et les accède comme des données en mémoire partagée grâce à la technologie iso-mémoire employée. Les secondes dites "privées" sont celles allouées classiquement par les programmes séquentiels

et sont destinées à un usage temporaire et local aux RPC (`malloc()` ou équivalent, données de pile : variables, tableaux, structures, ...).

2.3.2.1 Description des données parallèles

Dans un programme PRFX, les données parallèles sont supposées contenir les valeurs initiales et finales (résultat) d'un traitement parallèle. Toutefois, dans le cas de tâches stockant leurs résultats dans des fichiers, ces résultats peuvent provenir de données temporaires.

Les données parallèles doivent être allouées par le programmeur via l'iso-allocateur de PRFX. Nous rappelons que cela confère à chaque donnée et sous-donnée parallèle un ensemble d'espaces d'adresses invariant lors de la migration de ces dernières d'un processus à l'autre (iso-adresses). Ainsi, quel que soit le placement ou la migration des données, les calculs associés accèdent aux bons emplacements mémoire. En effet, les données peuvent avoir été découpées, dispersées ou intersectées, elles s'ancrent toujours aux mêmes adresses mémoire.

Contraintes de contenu : Les données dédiées au traitement parallèle ne doivent en aucun cas contenir des informations spécifiques à la machine locale, au système d'exploitation ou à des bibliothèques. Par exemple les descripteurs de fichiers ne peuvent supporter la traversée d'une tâche à une autre car la destination peut être dans un processus différent. PRFX ne gère pas ce type de service car les systèmes d'exploitation UNIX classiques n'assurent pas la validité d'un descripteur de fichier après sa migration vers un autre processus. En langage C, il est impossible par exemple d'identifier un descripteur de fichier dans une zone mémoire. Aucun contrôle n'est donc fait par PRFX sur le contenu des données. L'utilisation et le transit d'informations à caractère local au système d'exploitation sont donc aux risques et périls du programmeur. Deux types d'informations restent, de façon certaine, corrects après transfert entre deux tâches :

- les données scalaires ;
- les références à des données parallèles (ici en iso-mémoire).

Les références de données parallèles, comme nous le verrons dans les prototypes de fonctions PRFX, sont des pointeurs de pointeurs pour des raisons d'implémentation.

2.3.2.2 Opérations sur les données parallèles

Allocation dynamique : La mise à disposition d'un allocateur dynamique par PRFX permet d'implémenter librement, comme en séquentiel, les données logiques notamment irrégulières. Ces allocations servent également à informer l'inspecteur des créations de données parallèles. L'allocation d'une donnée parallèle s'effectue uniquement via l'appel à `void PRFX_isomalloc( void** data, ulong size)` (cf. listing 2.3) et permet de réserver un espace d'adresses contigu de taille `size` au niveau global ainsi que de réserver localement la mémoire associée à ce même espace d'adresses.

L'appel à la fonction `PRFX_isomalloc()` fournit un nouvel emplacement en iso-mémoire pour la donnée parallèle référencée par `data`. Il déclare également automatiquement un *stencil*

1D (décrit ci-après) associé à la référence `data` de la donnée parallèle car l'allocation d'une donnée parallèle suppose qu'elle sera utilisée en tant que telle.

Listing 2.3 – Exemple d'allocation et d'initialisation d'une matrice de dimension $N \times N$.

```
1  int i,j,N = 8000;
2  double (*matrix)[N];
3
4  PRFX_isomalloc(&matrix, N*N*sizeof(double));
5  for(i = 0; i < N; i++)
6    for(j = 0; j < N; j++)
7      matrix[i][j] = i * j;
```

Stencils : Un *stencil* est un descripteur de la forme géométrique d'une donnée parallèle en mémoire indépendamment de sa localisation. Il est l'un des éléments permettant la déclaration d'un accès à une donnée parallèle par une tâche. La déclaration des accès faits par les tâches (aux données parallèles) est obligatoire. Elle sert à renseigner l'inspecteur pour qu'il puisse faire des intersections entre les données accédées par les tâches. Dans le cas de notre implémentation, ceci implique donc que les *stencils* doivent pouvoir être calculés lors de l'inspection.

Cette déclaration d'accès à une donnée parallèle se fait en trois étapes dont les deux premières peuvent commuter. La première étape est un appel à une fonction du type :
`PRFX_stencilDeclXXXX(void**data,...)` (voir pages suivantes) qui définit le *stencil* de la donnée parallèle. Il effectue l'association d'un *stencil* avec la référence `data` à la donnée parallèle (cf. listing 2.4 ligne 7). La deuxième étape correspond à l'affectation libre de `data` avec une référence dans une donnée parallèle (cf. listing 2.4 ligne 12). Enfin, la troisième étape correspond à la création d'une tâche avec `data` en paramètre via l'appel à `PRFX_call()` décrit à la section 2.3.3 (cf. listing 2.4 ligne 13). Aussi, pour chaque référence à une donnée parallèle passée en paramètre de cet appel, le support PRFX connaît donc l'adresse de base et le *stencil* correspondant.

La bibliothèque PRFX propose plusieurs fonctions du type `PRFX_stencilDeclXXXX(void **data,...)`. Ces fonctions associent un *stencil* à la référence `data` de la donnée parallèle passée en paramètre. Une fois cette déclaration faite, la valeur de la référence `data`, et donc la donnée parallèle, peut être déplacée en mémoire au gré du programmeur sans nécessiter de redéclaration. Ce déplacement est libre mais le *stencil* ne doit pas déborder de la donnée parallèle (initialement iso-allouée) à laquelle il s'applique. Nous expliquerons pourquoi à la section 2.4. Par contre, les sous-données identifiées par des *stencils* à l'intérieur d'une même donnée parallèle peuvent se recouvrir mutuellement. La déclaration des accès n'est pas vérifiée (e.g. accès valide dans une donnée parallèle allouée par un unique iso-malloc).

`void *PRFX_stencilDecl1D (void**data, ulong sz)`
Cet appel permet de décrire une zone contiguë référencée par `data` qui s'étend sur `sz` octets. En langage C, les tableaux sont stockés de façon contiguë quel que soit leur nombre de dimensions. Cet appel permet donc de décrire tous les accès dès lors qu'ils concernent l'intégralité des données présentes dans les dernières dimensions du tableau (e.g. de d_i à d_n pour un tableau `int tab[`d_1`][...][`d_i`][...][`d_n`]`). C'est le cas dans l'exemple du listing 2.4 où chaque bloc est

bi-dimensionnel, mais nous exploitons le fait qu'en langage C les lignes d'un tableau sont stockées de façon contiguë en mémoire.

Listing 2.4 – Utilisation d'un *stencil* de forme 1D d'une taille de 100 `float` pour déclarer les accès aux blocs (10×10) dans une matrice de 7×7 blocs.

```
1  float (*matrix)[7][10][10];
2  float (*blk)[10];
3
4  int blk_sz = 10*10*sizeof(float);
5
6  PRFX_isomalloc(&matrix, 7*7*blk_sz));
7  PRFX_stencilDecl1D(&blk, blk_sz);
8
9  for(i = 0; i < 7; i++)
10   for(j = 0; j < 7; j++)
11     {
12       blk = matrix[i][j];
13       PRFX_call(i*7+j,0.0,func, &blk);
14     }
```

`void *PRFX_stencilDecl2D (void**data, ulong dim_sz, ulong height, ulong width_sz)`

Cet appel permet de décrire une zone mémoire rectangulaire "striée". Cette zone mémoire est constituée de `height` sous-zones mémoires de `width_sz` octets espacées de `dim_sz` octets. Ce *stencil* peut être utilisé pour extraire par exemple un sous-tableau dans un tableau bi-dimensionnel, une colonne de blocs dans une matrice de blocs ou une bande dans une matrice.

Listing 2.5 – Utilisation d'un *stencil* de forme 2D (10 × 10 et un *stride* 80) pour déclarer les accès aux blocs dans une matrice de 80×80 `float`.

```
1  float (*matrix)[80];
2  float (*blk)[80];
3  int stride_off = 80*sizeof(float);
4  int line_sz = 10*sizeof(float);
5  int nb_line = 10;
6
7  PRFX_isomalloc(&matrix, 80*80*sizeof(float));
8  PRFX_stencilDecl2D(&blk, stride_off, nb_line, line_sz);
9
10 for(i = 0; i < 8; i++)
11   for(j = 0; j < 8; j++)
12     {
13       blk = &matrix[10*i][10*j];
14       PRFX_call(i*8+j,0.0,func, &blk);
15     }
```

Dans l'exemple du listing 2.5, 64 RPC sont lancés sur 64 blocs indépendants de la matrice. Les intersections entre ces accès étant manifestement vides, le DAG de tâches sera constitué d'un père et de 64 fils. Ces 65 tâches peuvent s'exécuter en parallèle modulo le nombre de processeurs utilisés et le temps que met le support à satisfaire les dépendances du père vers ses fils.

`void PRFX_stencilDeclUEDiag (void**data, ulong dim_sz, ulong height, ulong width_sz, ulong inc_sz)`

La géométrie de la zone mémoire décrite est constituée de `height` sous-zones mémoire espacées de `dim_sz` octets. La taille de ces zones mémoire varie de `width_sz` à 0 octet(s) en décroissant par la gauche de `inc_sz` octets. Ce *stencil* peut être utilisé pour extraire une partie triangulaire supérieure dans une matrice stockée dans un tableau bi-dimensionnel.

`void PRFX_stencilDeclLEDiag (void**data, ulong dim_sz, ulong height,`

`ulong width_sz, ulong inc_sz)`

La géométrie de la zone mémoire décrite est constituée de `height` sous-zones mémoire espacées de `dim_sz` octets. La taille de ces zones mémoire varie de 0 à `width_sz` octets en décroissant par la droite de `inc_sz` octets. Ce *stencil* peut être utilisé pour extraire une partie triangulaire inférieure dans une matrice stockée dans un tableau bi-dimensionnel.

Les *stencils* présentés ici permettent de couvrir la majorité des accès des algorithmes d'algèbre linéaire. Cette liste de *stencils* peut être enrichie ; pour cela il suffit d'insérer le code correspondant à la description du nouveau *stencil* à la suite de ceux existants de la plate-forme. L'ajout d'un nouveau *stencil* est néanmoins simple à faire car le code d'une fonction de déclaration de *stencils* comporte uniquement l'enregistrement d'un motif sous forme d'une collection d'intervalles mémoire relativement à une base. Cette description est de très bas niveau (adresses et *offset* mémoire) rendant PRFX potentiellement utilisable par des langages de plus haut niveau.

Les *stencils* sont une évolution par rapport aux premières versions de PRFX où nous avions choisi de faire des extractions de sous-données dans des données parallèles. Ces extractions contenaient toutes les informations relatives à l'emplacement en mémoire de la sous-donnée. Par la suite, nous avons décidé de scinder ces informations entre la forme géométrique de l'accès (*stencil*) et son adresse de base permettant de retrouver les adresses mémoire atteintes.

Nous avons choisi de faire déclarer (au programmeur) les *stencils* de façon indépendante des RPC pour une plus grande liberté de programmation. Nous aurions pu inclure la déclaration des *stencils* avec la déclaration des modes d'accès aux arguments (cf. section 2.3.3.1). Toutefois, cela aurait provoqué un bloc figé d'informations pour un RPC donné. De plus, cela supposerait des redéclarations inutiles car automatique des *stencils* à chaque lancement de RPC. En laissant le programmeur gérer les *stencils*, il peut factoriser leur coût de création et en optimiser l'utilisation.

Les *stencils* mettent en jeu des structures internes à l'inspecteur. Des mises à jour de ces structures sont faites lorsque des références à des données parallèles associées aux *stencils* sont passées en paramètre d'un RPC. Ces mises à jour peuvent être coûteuses dans le cas où l'accès ainsi instancié est nouveau et s'intersecte de façon complexe avec les accès précédents.

Combinaison de stencils : A partir des *stencils* proposés, d'autres *stencils* plus complexes peuvent être obtenus en les combinant. Cette opération s'effectue avec la fonction :
`void PRFX_stencilDeclMerge (void**data, int n, ulong offset_1, void** data_1, ulong offset_2, void**data_2,..., ulong offset_n, void**data_n).`
Les `n` références `data_i` de données parallèles sont supposées avoir fait l'objet d'une déclaration d'accès. Le résultat est l'enregistrement de la référence `data` de donnée parallèle en tant que fusion des *stencils*. Dorénavant, cette référence correspondra au regroupement de l'ensemble des *stencils* associés aux `data_i` respectivement décalés d'un `offset_i`.

Nous rappelons que lors de l'application ou de la construction d'un *stencil*, et l'erreur est plus tentante avec un *stencil* **combiné**, celui-ci doit être inclus dans une zone mémoire précédemment allouée par un unique appel à `PRFX_isomalloc()`.

La combinaison n'effectue aucune copie des données et sa complexité dépend de la complexité des *stencils* à traiter. En fait, plus les formes de *stencils* à combiner sont morcelées de façon anarchique, plus les tests nécessaires à la mise à jour des structures internes de PRFX seront nombreux.

Cette possibilité de regrouper *n* données parallèles sous un seul argument est utile dès que le programmeur veut décrire des accès morcelés. Ainsi, le programmeur va pouvoir conserver un faible nombre d'arguments pour ses RPC tout en leur fournissant de nombreuses sous données parallèles .

Listing 2.6 – Exemple d'accès équivalents avec et sans utilisation de *stencils* combinés.

```
1  /* Without merging */                          /* With merging */
2  void RPC_func3(cell*list, cell*unused1,cell*unused2)  void RPC_func1(cell*list)
3  {                                              {
4    cell *c = list;                                cell *c = list;
5    while (c != NULL)                              while (c != NULL)
6      {                                              {
7        compute(c);                                    compute(c);
8        c = c->next;                                   c = c->next;
9      }                                              }
10 }                                              }
11
12
13
14 [...]                                          [...]
15                                                PRFX_isomalloc(&tab_cell, 6*sizeof(cell));
16 PRFX_isomalloc(&c0, sizeof(cell));             c0 = &tab_cell[0];
17 PRFX_isomalloc(&c2, sizeof(cell));             c2 = &tab_cell[2];
18 PRFX_isomalloc(&c4, sizeof(cell));             c4 = &tab_cell[4];
19 c0->next = c2;                                 c0->next = c2;
20 c2->next = c4;                                 c2->next = c4;
21 c4->next = NULL;                               c4->next = NULL;
22                                                PRFX_stencilDecl1D(&list,0);
23                                                PRFX_stencilDecl1D(&c,0);
24                                                c = c0;
25                                                while (c != NULL)
26                                                  {
27                                                    offset = (ulong)c - (ulong)&tab_cell[0];
28
29                                                    PRFX_stencilMerge(&list, 2, 0,&list, offset,&c);
30                                                    c = c->next
31                                                  }
32                                                list = c0;
33 PRFX_call(0,0,RPC_func3, &c0, &c2, &c4);       PRFX_call(0,0,RPC_func1, &list);
34 [...]                                          [...]
```

Nous avons choisi pour cet exemple du listing 2.6 de chaîner les cellules paires dans l'ordre `c0`, `c2` et `c4` mais ce chaînage peut être quelconque. Dans le code de gauche, chaque cellule est allouée et nommée séparément ; il faut ensuite toutes les passer en paramètre de l'appel à `RPC_func3()` pour que ce dernier puisse disposer des cellules constituant la liste. Si nous devions passer une liste de *n* cellules, il nous faudrait écrire une fonction cible que nous nommerions `RPC_funcn()` pour être cohérent, et à laquelle il faudrait passer *n* références à des données parallèles de type "cellule", ce qui n'est pas très commode.

Le code de droite est plus générique et la taille de la liste n'est pas bornée par le nombre d'arguments maximum autorisé pour un RPC. Le chaînage des cellules est effectué dans une donnée parallèle (tableau `tab_cell`) constituée à l'aide d'une seule iso-allocation. Ensuite, le *stencil* de la liste chaînée est construit en la parcourant et en fusionnant successivement tous les *stencils* de ses cellules. Pour chacun de ces *stencils*, le décalage (*offset*) se compte par rapport à la base du tableau de cellules. Il doit être calculé pour chaque cellule sinon la fusion successive des *stencils* ne décrirait que la forme d'une seule cellule. L'appel final est simplifié car il suffit de passer la référence à la cellule tête de liste en paramètre à `RPC_func1()` pour que toutes les cellules lui soient transmises.

Libération : La libération d'une donnée parallèle nécessite un droit d'accès en écriture sur cette donnée (la déclaration des modes d'accès est décrite à la section 2.3.3.2). La libération d'une donnée parallèle référencée par `data` obtenue avec `PRFX_isomalloc(data,sz)` s'effectue via l'appel à `PRFX_free(data)`. La libération ne supprime pas l'association créée lors de l'allocation entre la donnée parallèle référencée par `data` et son *stencil* 1D. Cette référence peut être réutilisée avec sa forme géométrique associée pour référencer une autre donnée parallèle.

2.3.2.3 Données (non parallèles) privées et globales

Toutes les données, hormis celles parallèles ou globales, sont privées aux tâches. Elles sont composées des données allouées dynamiquement (e.g. avec `malloc`) et parfois automatiquement dans le tas ou dans la pile. Ces données sont temporaires et doivent naturellement être créées et libérées au cours de l'exécution d'une seule et même tâche. En effet, si une première tâche alloue une donnée locale et si une seconde la libère, il se peut que ces événements interviennent dans le contexte d'exécution de deux processus différents.

Les données globales (dans le sens du langage C) se présentent dans le code source par des déclarations globales. Leur portée syntaxique est globale et ces données sont présentes dans le binaire du programme et répliquées aux mêmes adresses dans chaque processus PRFX à la manière d'une iso-allocation (garanti par la spécification UNIX du *loader*). Mais ces données ne sont pas des données parallèles, car elles ne sont pas déclarées en tant que telles. La cohérence et les synchronisations associées ne sont donc pas gérées par PRFX. Elles pourraient l'être si l'on intégrait au compilateur C un appel vers PRFX pour déclarer ces iso-allocations statiques dans le segment de données également statiques. Mais ceci pose des problèmes d'implémentation car certaines architectures ne peuvent effectuer des accès directs aux mémoires distantes que sur des segments mémoire spéciaux.

Les données déclarées `static` dans le corps des fonctions ne sont pas supportées car elles impliquent une zone mémoire persistante non gérée par PRFX. Néanmoins, le programmeur peut les utiliser en protégeant leurs accès avec des verrous, de même pour les données spécifiques des *threads* POSIX. Ces actions constituent un usage anormal du mode d'expression PRFX. Elles n'auront qu'une portée locale au processus. De plus, elles dépendront du placement des tâches en relation avec ces variables, le programmeur doit donc maîtriser tous ces paramètres pour ne pas provoquer d'erreurs.

2.3.3 Modèle de programmation MPMD (RPC / tâche)

Le lancement (en parallèle) des processus du programme utilisateur n'est pas contrôlé par PRFX mais par l'environnement du système utilisé. Le programme démarre par une exécution SPMD (un *thread* principal par processus) de la fonction `main()` jusqu'à ce que `PRFX_init()` soit rencontré. A cet instant, les *threads* exécuteurs PRFX sont créés conformément au déploiement. La portion de code (tâche `main()`) encadrée par `PRFX_init()` et `PRFX_terminate()` est traitée comme une fonction cible de RPC, elle est exécutée par un seul *thread* exécuteur (par défaut le *thread* 0 du processus 0). La tâche `main()` crée des tâches qui à leur tour peuvent

en créer d'autres. Ces tâches peuvent être disparates ainsi que les données qu'elles traitent, le déroulement du programme est donc de façon générale MPMD.

A l'origine d'un RPC se trouve un seul et unique père, l'**initiateur**. Un RPC est un appel à une procédure à distance avec sa liste d'arguments. Une tâche représente l'exécution séquentielle d'une fonction cible de RPC jusqu'à son retour d'appel. Les fonctions cibles des RPC (tâches du DAG) sont exécutées de façon atomique du point de vue des *threads* exécuteurs. Ceux-ci exercent un contrôle de sorte que l'ordonnancement, la synchronisation et la cohérence des données des tâches soient respectés.

Le modèle que nous utilisons diffère de celui des RPC "classiques" car il est simplifié grâce à la génération automatique des dépendances (synchronisation des tâches plus cohérence des données). Le programmeur est déchargé de cette tâche et ne dispose pas de primitive de synchronisation autre que l'appel de création de tâche `PRFX_call()`. Nous avons choisi de rendre la procédure de création de tâche effectivement non bloquante en déléguant la communication à une tâche interne asynchrone et ordonnancée. Ainsi, nous avons la garantie, nécessaire à l'ordonnancement, que le code de la tâche utilisateur ne sera pas bloqué par une synchronisation. En effet, aucun appel à la bibliothèque PRFX ne provoque d'attente.

Les envois de données requis pour satisfaire les précédences d'une tâche peuvent provenir d'autres tâches que celles de l'initiateur. L'envoi des données est dit découplé de la partie contrôle des RPC. Alors, deux types de transfert de données existent :

– transfert des données à partir de la tâche initiatrice du RPC ;
– transferts découplés des données à partir de tâches parentes.

D'un point de vue implémentation, ces transferts sont gérés par des tâches internes de communication spécialisées. Ces dernières effectuent un ou plusieurs appels à la bibliothèque de communication unilatérale lorsque la destination est distante et une simple synchronisation lorsqu'elle est locale au processus.

L'activation des tâches associées aux deux types de transfert survient à des moments différents pour satisfaire une règle liée à la notion de tâches propriétaires d'une donnée. Nous verrons cette notion ci-après (section 2.3.3.3).

Les RPC peuvent eux-mêmes lancer des RPC et appeler des fonctions classiques faisant d'autres RPC. La récursion est également possible dans la mesure où la condition d'arrêt ne dépend au pire que de données structurantes de niveau inspection. Si cette restriction n'est pas respectée, l'inspection sera invalide (récursion s'arrêtant trop tôt, trop tard ou jamais lors de la pré-exécution).

Ce modèle a l'avantage de ne pas mettre en œuvre de technologies lourdes au niveau des communications. Le niveau d'implémentation est bas, ce qui garantit sa pérennité et autorise l'obtention de performances.

2.3.3.1 Description des RPC / tâches

Dans le cadre de PRFX, un RPC (ou appel de tâche) nécessite des paramètres de configuration optionnels (distribution, phase), une fonction cible et un ensemble d'arguments. La fonction cible doit avoir été préalablement enregistrée en tant que RPC avec :

`PRFX_declRPCn(void (*func)(), int inspec, int mode_arg1, ..., int mode_argn, PRFXtskcost (*costfunc)(arg1,..., argn))` où *n* est à remplacer par le nombre d'arguments. Le premier argument de `PRFX_declRPCn()` est le nom de la fonction RPC. Le deuxième argument indique si son exécution est évitable lors de l'inspection. Les `n` arguments suivants sont les modes d'accès des `n` paramètres du RPC. Nous en détaillerons les différentes possibilités à la section 2.3.3.2. Le dernier argument correspond à la fonction de coût associée à ce RPC (cf. section 2.5.1). La valeur de *n* est bornée dans le cas de notre implémentation par la valeur 32.

Nous avons choisi d'associer les modes d'accès des arguments à la déclaration des RPC, car les modes d'accès sont supposés stables pour des conformations variables des arguments. Dans le cas où le programmeur souhaite utiliser des modes d'accès différents pour une même fonction cible, il lui faut faire autant de déclarations qu'il y a de séquences de modes d'accès différents avec des noms de fonctions cibles également différents (ces fonctions cibles ont le même corps).

La déclaration de tous les RPC doit être faite une seule fois avant l'initialisation de PRFX (`PRFX_init()`). Cette déclaration est obligatoire pour chaque fonction cible de RPC, et spécialement la distinction dans les paramètres entre les variables et les données parallèles. En effet, les données parallèles doivent être passées en paramètres des RPC avec un niveau d'indirection supplémentaire par rapport au prototype de la fonction cible. Donc, si la distinction est mal faite, le niveau d'indirection supposé n'en sera pas un et cela constitue une erreur de programmation.

La déclaration des RPC et leur utilisation ne sont pas limitées à des fonctions écrites par le programmeur. Ce dernier peut également déclarer des RPC pour des fonctions de bibliothèques *thread-safe* telles que les bibliothèques C, BLAS, etc...

Listing 2.7 – Exemple de déclarations des modes d'accès d'une fonction cible de RPC.

```
1  PRFXtskcost RPC_funcNameCost(int a, ulong b, double *data1, double *data2)
2  {
3    PRFXtskcost tsk_cost;
4    tsk_cost.nbflops = [...];
5    tsk_cost.peakfactor = [...];
6    return tsk_cost
7  }
8
9  void RPC_funcName(int a, ulong b, double *data1, double *data2)
10 {
11 [...]
12 }
13
14 int main(int argc, char*argv[])
15 {
16   [...]
17   PRFX_declRPC4(RPC_funcName,1,IMM,IMM,RW,RO,RPC_funcNameCost);
18
19   PRFX_init(NULL);
20   PRFX_isoMalloc(&tab1,count1*sizeof(double));
21   PRFX_isoMalloc(&tab2,count2*sizeof(double));
22
23   PRFX_call(-1,0.0,count1,count2,&tab1,&tab2);
24   [...]
25
26   PRFX_terminate();
27 }
```

Le listing 2.7 montre un exemple de déclaration du RPC `RPC_funcName()` avec `PRFX_declRPC4()`. Ce RPC est déclaré à la ligne 17 comme utile à l'inspection (valeur 1), ayant quatre arguments accédés selon les modes indiqués et une fonction de coût.

Un RPC est initié par l'intermédiaire de l'appel à `PRFX_call()` dont le prototype est de la forme :
`void PRFX_call(int num, double phase, void (*func)(), void*arg1,...).`
Cette fonction initie un RPC optionnellement placé dans l'espace et dans le temps avec les paramètres `num` et `phase` destinés à l'ordonnanceur et que nous décrivons ci-après. La fonction cible du RPC est `func` et ses arguments sont les `arg`*i*. Les arguments sont les seuls messages reçus par un RPC. Ils sont soit des valeurs scalaires de taille fixe, soit des références à des données parallèles auxquelles un *stencil* a été associé. Les variables scalaires passées par copie en arguments des RPC seront appelées **données immédiates**. Elles doivent être stockées dans un mot mémoire de la taille d'un pointeur (`sizeof(void*)`) pour être conformes au prototype d'un RPC défini par l'interface de PRFX. Leur existence facilite l'utilisation des appels de tâches PRFX. Elles ne provoquent la génération d'aucune dépendance supplémentaire entre les RPC. Ce sont juste des valeurs recopiées dans la pile de la fonction cible avant son lancement.

Le code de la fonction cible du RPC doit respecter un certain nombre de contraintes pour que l'inspection soit valide. Il ne peut contenir que des appels de fonctions (e.g. bibliothèques) *thread-safe*. Il ne peut effectuer des accès mémoire autres que ceux respectant les modes d'accès en cours (lié à la notion de propriété que nous verrons à la section 2.3.3.3).

Seuls les accès sur les données suivantes sont valides à l'intérieur d'une tâche :
- les données parallèles correspondant strictement aux zones de données des arguments tels que leurs accès ont été déclarés via les *stencils* ;
- les données (parallèles ou locales) iso-allouées par la fonction ;
- les données statiques globales (en lecture seule uniquement) ;
- les données de pile.

La tâche sera exécutée à une date et par un processeur fixés par l'ordonnancement optionnellement choisi par le programmeur. Le placement des tâches peut être indiqué via la valeur de numérotation `num` (spécifiée en premier paramètre de l'appel `PRFX_call()`) et un choix de type de distribution pour cette numérotation sur les *threads* exécuteurs. Ce placement peut aussi être laissé à la liberté de l'ordonnanceur (`num = -1`). La fonction de distribution associe, via la numérotation des RPC par l'utilisateur, un numéro de *thread* exécuteur à chaque tâche. La numérotation de l'utilisateur est supposée être comprise entre 0 à $n-1$. Plusieurs tâches peuvent porter le même numéro et des numéros peuvent être inutilisés. Lorsqu'elle n'est pas spécifiée, une fonction par défaut est utilisée. Cette dernière effectue une distribution cyclique des tâches sur les *threads* exécuteurs. Lorsqu'une fonction de distribution est positionnée dans une tâche, elle reste effective jusqu'à ce que la tâche termine ou que la fonction soit repositionnée.

Quelques types de distribution sont proposés et spécifiables avec la fonction :
`void PRFX_thrDistributionSetDefault(void *distrib_code, int param1, int param2, int param3, int param4, int param5).`
Ces distributions utilisent le numéro `num`, les paramètres `param`*i* ainsi que le déploiement *threads* / processus. Le paramètre `distrib_code` peut correspondre aux identificateurs suivants pour obtenir les distributions cycliques ou blocs classiques :

1. `PRFX_DISTRIB_BLOCK1D`

2. PRFX_DISTRIB_CYCLIC1D
3. PRFX_DISTRIB_CYCLIC2D_COLUMN
4. PRFX_DISTRIB_CYCLIC2D_LINE
5. PRFX_DISTRIB_CYCLIC2D_LINE_COLUMN
6. PRFX_DISTRIB_CYCLIC2D_LINE_COLUMN_SMP

La numérotation est interprétée par la fonction de distribution comme un vecteur de dimension param1 pour les cas 1D et comme une grille de largeur param1 et de hauteur n_L/param1 pour les cas 2D. La fonction de distribution utilise respectivement un vecteur ou une grille de *threads* selon qu'elle doit paver un vecteur ou une grille associée à la numérotation utilisateur. Ce pavage permet d'associer un numéro de *thread* exécuteur à chaque numéro utilisateur. La numérotation des *threads* exécuteurs est fixée par PRFX, elle va de de 0 à n_{thr}.

La fonction de distribution BLOCK1D effectue le pavage avec un vecteur V de numéros de *threads* exécuteurs de dimension n_L tel que $V(i) = i \; div$ param2 avec $i \in [0,n_L[$.

La fonction de distribution CYCLIC1D effectue le pavage avec un vecteur V de numéros de *threads* exécuteurs de dimension param2 tel que $V(i) = i$ avec $i \in [0,$param2 $[$.

La fonction de distribution CYCLIC2D LINE effectue le pavage avec une grille G de numéros de *threads* exécuteurs de dimensions param2 $\times$ param1 tel que $G(i,j) = i$ avec $i \in [0,$param2 $[$ et $j \in [0,$param1 $[$.

La fonction de distribution CYCLIC2D COLUMN effectue le pavage avec une grille G de numéros de *threads* exécuteurs de dimensions param1 $\times$ param2 tel que $G(i,j) = j$ avec $i \in [0,$param1 $[$ et $j \in [0,$param2 $[$.

La fonction de distribution CYCLIC2D LINE COLUMN effectue le pavage avec une grille G de numéros de *threads* exécuteurs de dimensions param2 $\times$ param3 tel que $G(i,j) = i$param3$+j$ avec $i \in [0,$param2 $[$ et $j \in [0,$param3 $[$.

Le suffixe _SMP correspond à une distribution sur une grille de processus de taille $P \times Q$ composée d'une sous-grille de *threads* exécuteurs de taille $R \times S$. Ici, P,Q,R et S correspondent respectivement à param2, param3, param4 et param5. Cette fonction de distribution permet de prendre en compte la hiérarchie des SMP. Elle calcule le numéro du *thread* exécuteur destination (thr_num) comme indiqué ci-dessous. Les variables p_l et p_c sont respectivement les numéros de ligne et de colonne du processus cible dans la grille de processus, l et c sont, quant à eux, les numéros de ligne et de colonne dans la grille de *threads* locale au processus, et nb_local_thr est le nombre (supposé homogène) de *threads* par processus.

```
p_l = ((num / param1) / S) mod Q;
p_c = ((num mod param1) / R) mod P;
l = (num/param1) mod S;
c = (num mod param1) mod R;
thr_num = (p_l*P + p_c)*(nb_local_thr)+ l * R + c;
```

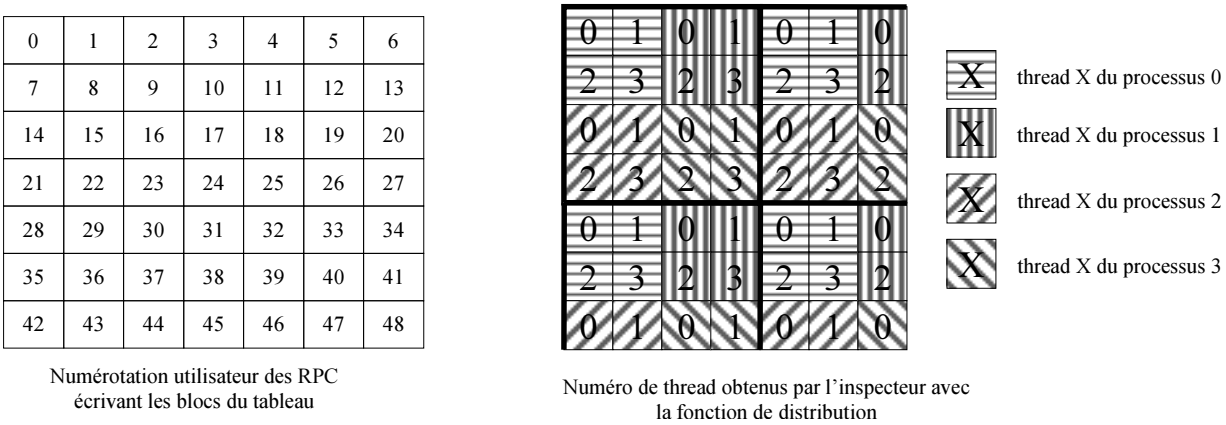

FIG. 2 – Distribution de type `PRFX_DISTRIB_CYCLIC2D_LINE_COLUMN_SMP` d'une matrice de 7×7 blocs avec hiérarchie de placement cyclique sur une grille de processus 2×2 puis sur une grille de *threads* 2×2.

La figure 2 présente un cas de prise en compte des SMP en adoptant une distribution cyclique à deux niveaux. La distribution de la donnée parallèle, ici un tableau de blocs 2D, est indirectement obtenue par le placement des RPC initialisant chacun des blocs (listing 2.4). La numérotation de ces RPC est `i*7+j` et ainsi chaque bloc a un numéro unique. La distribution des RPC est réalisée en faisant précéder le code du listing 2.4 de l'appel :
`PRFX_thrDistributionSetDefault( PRFX_DISTRIB_CYCLIC2D_LINE_COLUMN_SMP, 7, 2, 2, 2, 2).`
La distribution sera effectuée sur une grille carrée 2×2 de processus eux-mêmes constitués d'une grille 2×2 de *threads*. La distribution choisie n'a pas obligation d'être en adéquation avec la hiérarchie du déploiement utilisé. Dans le cas de notre exemple, l'adéquation existe si le déploiement est sur quatre nœuds quadri-processeurs avec un processus par nœud et un *thread* par processeur. Ce type de distribution convient bien par exemple aux algorithmes de type résolution de systèmes linéaires car il favorise la localité des accès et équilibre la charge des processeurs.

Le paramètre `phase` est destinée à l'ordonnanceur que nous verrons à la section 2.5.

2.3.3.2 Mode d'accès des RPC aux données parallèles

La description du motif d'accès n'est pas suffisante à notre inspecteur de programmes parallèles pour générer un DAG de tâches complet modélisant le programme de l'utilisateur. En effet, il faut également une connaissance des modes d'accès aux données parallèles car une partie du parallélisme résulte des accès disjoints, mais une autre partie tout aussi importante vient des accès en lecture seule par plusieurs tâches sur des données s'intersectant. De base, le langage C ne fournit pas les moyens de connaître ces modes d'accès. Pour garder tout le potentiel des optimisations du compilateur, nous n'instrumentons pas chaque affectation, nous préférons demander au programmeur de les spécifier à un niveau macroscopique par tâche et de s'y conformer.

Les différents modes d'accès possibles pour un argument de RPC sont donc :

– argument de type donnée parallèle
    1. RO (Read Only) : la donnée parallèle peut être lue uniquement ;
    2. RW (Read Write) : la donnée parallèle peut être lue et / ou écrite ;
    3. CRW (Commutative Read Write) : la donnée parallèle peut être lue et / ou écrite de façon commutative ;
    4. FW (Full Write) : la donnée parallèle doit être modifiée en totalité par la tâche et elle ne doit effectuer aucun accès en lecture préalablement à l'écriture ;
    5. VR (virtual) : la donnée parallèle n'est pas modifiable mais elle peut être découpée en sous-données.
– argument de type donnée immédiate
    1. (IMM) : valeur scalaire devant être stockée dans une donnée de type `void*` (de taille 4 ou 8 octets selon le mode d'exécution 32 ou 64 bits).

Les deux premiers modes d'accès classiques que sont la lecture seule (RO) et la lecture écriture (RW), suffisent à la définition d'une structure de DAG modélisant tout le parallélisme d'un algorithme prévisible. Nous fournissons un mode d'accès commutatifs (CRW) pour décrire un indéterminisme d'exécution identifié. Ces accès commutatifs sont utiles pour décrire des réductions. La gestion des accès commutatifs est problématique car elle nécessite de laisser le DAG dans un état inachevé. Ce n'est qu'à l'exécution que l'accès commutatif opèrera selon un ordre non-déterministe dépendant de la date effective de démarrage des tâches en concurrence. Pour l'instant, notre support au mode d'expression PRFX n'implémente pas la gestion optimisée des accès commutatifs. Ces accès sont traités comme des accès RW. Nous reviendrons sur ce problèmes dans les perspectives de cette thèse (cf. section 5.2.2).

Les deux derniers modes d'accès (FW et VR) aux données parallèles permettent d'améliorer les performances de l'exécution en optimisant le DAG. Pour le mode FW, l'optimisation vient du fait que les dépendances de données satisfaisant ce type d'accès se réduisent à une synchronisation vide de contenu. En effet, le contenu antérieur de la donnée est inutile car elle sera intégralement écrite avant même d'être lue. Ce mode permet de prendre en compte le cas où l'on souhaite réutiliser des zones mémoire pour faire l'économie de leur libération et de leur réallocation. Il peut aussi servir comme mode d'accès optimisé pour un RPC libérant une donnée parallèle.

Nous proposons également le concept de l'accès virtuel (mode VR). Il permet d'optimiser les communications en n'envoyant que la localisation en mémoire et le *stencil* de la donnée parallèle. Cela permet par exemple d'appliquer des *stencils* sur des données de grandes tailles et de distribuer les calculs via des RPC sans avoir le contenu de la donnée présent localement. Ainsi, les coûts de communications sont uniquement dépendants de la taille de la structure de donnée du *stencil* et non de la donnée elle-même.

Le mode d'accès VR permet par exemple de décrire des programmes parallèles de type maître-esclave où le maître décide des modalités du partage du travail et les tâches esclaves traitent le problème avec des données persistantes. Un exemple de ce type de fonctionnement est présenté dans le listing 2.8. Dans ce code, un RPC maître (`RPC_spreadBlkCol()`) et un RPC esclave (`RPC_computeBlkCol()`) sont déclarés juste après l'initialisation de PRFX. La

matrice de travail `A` est iso-allouée et remplie séquentiellement (ligne 15) avant l'appel au maître par la tâche `main()`. Le maître reçoit cette matrice en mode virtuel (VR), il peut donc la découper (ici en `nb_cblk` blocs colonnes striés). Chacun de ces blocs colonnes est traité via l'appel `PRFX_call()` à la fonction de tâche `RPC_computeBlkCol()`. Un traitement spécial est effectué pour le dernier bloc dans le cas où la dimension de la matrice n'est pas un multiple de `nb_cblk`.

Listing 2.8 – Exemple d'utilisation de l'accès virtuel .

```
1  [...]
2  typedef double elem_t;
3  void RPC_spreadBlkCol(ulong matrix_dim, double A[matrix_dim][matrix_dim], ulong nb_cblk);
4
5  int main(int argc, char* argv[])
6  {
7    ulong  nb_cblk = atoi(argv[1]), matrix_dim = atoi(argv[2]);
8    elem_t (*A)[matrix_dim];  // ptr on sqr matrix
9
10   PRFX_init(NULL);
11   PRFX_declRPC3(RPC_spreadBlkCol,0, IMM, VR, IMM, NULL);
12   PRFX_declRPC4(RPC_computeBlkCol,1, IMM, IMM, IMM, RW, NULL);
13
14   matrix = PRFX_isomalloc(&A,matrix_dim*matrix_dim*sizeof(elem_t));
15   fill(matrix_dim, A);
16   PRFX_call(0,0.0,RPC_spreadBlkCol,matrix_dim, A, nb_cblk);
17
18   PRFX_terminate();
19 }
20
21
22 void RPC_spreadBlkCol(ulong matrix_dim, double A[matrix_dim][matrix_dim], ulong nb_cblk)
23 {
24   ulong c;
25   ulong blk_ldim = matrix_dim, blk_cdim = matrix_dim / nb_cblk;
26   ulong last_blk_cdim = matrix_dim % blk_cdim;
27
28   for(c = 0; c < nb_c_blk-1; c++)
29     {
30       elem_t (*blk_col)[blk_cdim];
31       PRFX_stencil2D(&blk_col, &A[0][c*blk_width], matrix_dim, matrix_dim, blk_cdim);
32       PRFX_call( c,0.0, RPC_computeBlkCol, matrix_dim, blk_ldim, blk_cdim, &blk_col);
33     }
34
35   if (last_blk_cdim != 0 )
36     {
37       elem_t (*last_blk_col)[last_blk_cdim];
38       PRFX_stencil2D(&blk_col, &A[0][(nb_cblk-1)*blk_width], matrix_dim, matrix_dim, last_blk_cdim);
39       PRFX_call( c,0.0, RPC_computeBlkCol, matrix_dim, blk_ldim, blk_cdim, &last_blk_col);
40     }
41 }
```

2.3.3.3 Règles d'usage et notion de tâche propriétaire de données parallèles

Des règles ou du bon sens doivent être observés lors de l'usage du mode d'expression PRFX. Par exemple, un *stencil* ne doit pas déborder d'une iso-allocation ou chevaucher plusieurs données parallèles iso-allouées séparément. Le premier cas est invalide de façon évidente, pour le deuxième cas, cette interdiction existe parce que l'inspecteur n'est pas capable de gérer cette complexité d'accès. En effet, dans le cas où un *stencil* chevauche deux données parallèles différentes, il n'y a aucune garantie que les décalages relatifs (*offset*) entre les deux pointeurs iso-alloués lors de l'inspection et entre ceux iso-alloués lors de l'exécution soient identiques. Lors de la déclaration d'un *stencil*, l'inspecteur enregistre seulement la forme de la donnée et son décalage relatif à une adresse obtenue grâce à un numéro d'argument de RPC ou à un numéro d'iso-allocation (gestion interne). Ni le contenu de la donnée parallèle ni même sa position absolue (adresse) ne sont enregistrés car ces informations seront dynamiquement instanciées lors

de l'exécution parallèle. En effet, les fonctions de type `PRFX_stencilDecl...()` enregistrent juste un nouveau motif d'accès de plus faible envergure qui pourra être appliqué dans une donnée parallèle existante ou prochainement créée. Lors de la déclaration des *stencils*, la référence à la donnée parallèle n'a pas besoin d'être initialisée. De fait, les *stencils* ne correspondent pas à l'état de la donnée parallèle (éventuellement) référencée à l'instant où l'appel à `PRFX_stencil()` est fait.

Notons que le contenu des données parallèles reçues par une tâche n'a potentiellement pas la même valeur qu'au moment où le RPC correspondant a été initié (envoi des données découplé du contrôle). Considérons l'exemple d'un code où n RPC opérant sur la même donnée en écriture sont issus du même initiateur. Comme l'initiation des RPC est non bloquante, la satisfaction des dépendances de contrôle peut intervenir très rapidement (modulo l'ordonnancement des communications), par contre l'exécution des fonctions cibles est conditionnée par les dépendances de données. Ces dernières induisent une séquentialisation de ces n RPC car la même donnée est successivement accédée en écriture. Les versions des données traitées par les différents RPC en jeu seront donc potentiellement toutes différentes. Ces versions des données sont les mêmes que celles qui auraient été obtenues lors d'une exécution séquentielle de ce code (transformation des RPC en appels de fonction).

Dans le DAG, les données parallèles et immédiates transitent d'un nœud (tâche) à l'autre via les arcs. Nous introduisons donc la notion de tâche propriétaire d'une donnée et nous raffinons cette notion de propriété avec les modes d'accès. Nous utilisons donc trois degrés croissants pour la notion de tâche propriétaire : **propriétaire virtuel** (VR), **en lecture** (RO) ou **en écriture** (RW, CRW et FW).

Une tâche utilisateur peut devenir propriétaire d'une donnée parallèle à deux moments distincts : avant l'exécution du RPC, lorsque la donnée parallèle (correspondant à un argument du RPC) arrive à destination, ou bien en cours d'exécution de ce RPC, lorsque la donnée parallèle est allouée par `PRFX_isomalloc()`. Lors du passage par argument, la donnée parallèle est possédée selon le mode d'accès lié à la déclaration du RPC. Lors de l'allocation, la donnée parallèle est possédée avec le plus haut degré de propriété (i.e écriture)

La perte de propriété ou l'affaiblissement du degré de propriété, peuvent se produire lorsqu'un RPC se termine ou lorsqu'un RPC initie un autre RPC. Mais une tâche est toujours au minimum propriétaire virtuelle. Les données parallèles sont successivement détenues par des tâches avec des degrés de propriété fonction de leurs modes d'accès à leurs arguments. Le programmeur doit respecter les règles d'affaiblissement du degré de propriété suivantes :

- après une cession d'une donnée en lecture seule (RO) l'initiateur voit son degré de propriété réduit au minimum entre la lecture seule et son degré courant sur cette donnée ;
- après une cession d'une donnée en écriture (CRW, RW, FW), l'initiateur voit son degré de propriété sur cette donnée devenir virtuel ;
- après une cession d'une donnée en virtuel (VR), l'initiateur conserve le même degré de propriété courant sur cette donnée.

L'application de ces règles assure l'unicité du propriétaire en écriture au cours de la propagation d'une donnée parallèle dans le DAG. En fait, au cours de cette propagation, plusieurs

propriétaires en écriture se succèderont, mais à un endroit donné du DAG, une seule et unique tâche possède la donnée parallèle en écriture (la version en cours). Par contre, plusieurs tâches peuvent être simultanément propriétaires en lecture ou en virtuel d'une donnée parallèle.

Nous avons mis en place ces règles de programmation pour qu'une fonction cible puisse potentiellement travailler immédiatement sur ses arguments après l'initialisation du RPC (cas où les données sont déjà présentes sur le lieu d'exécution de la fonction cible). Les deux RPC père et fils sont alors en concurrence, il en résulte, une interdiction ou des contraintes pour l'initiateur s'il veut continuer d'utiliser une donnée parallèle venant d'être cédée en paramètre d'un RPC.

De base, le langage C ne permet pas de vérifier que les règles liées aux degrés d'accès aux données parallèles sont bien respectées. Il est donc à la charge du programmeur d'être attentif aux affaiblissements des degrés de propriété suite au passage d'une donnée parallèle en argument d'un RPC fils (danger comparable aux accès après avoir effectué un `free()`).

Les changements de mode d'accès au travers des appels ne sont pas contraints comme dans ATHAPASCAN où il doit y avoir une décroissance des degrés des accès pour les objets pris en paramètres par une tâche fille. Par exemple dans notre cas, une tâche possédant une donnée avec degré faible comme le RO peut générer des RPC accédant à cette donnée en RW. En effet, le degré RO est supérieur au degré VR et comme nous l'avons vu dans le code 2.8, ce mode VR permet de lancer des RPC sans restriction.

2.3.3.4 Exemple d'erreurs d'accès

Listing 2.9 – Exemple faux et corrigé d'un accès post RPC.

```
1  // INCORRECT                              // CORRECT
2  ...                                       void RPC_square1D(int dim, int*v)
3   int *data;                               {
4   int n = 10, sz = n *sizeof(int)            for (i = 0; i < dim; i++)
5   PRFX_isomalloc(&data, sz);                   v[i] = v[i] * v[i];
6   PRFX_call(0,0.0,RPC_fill,n,&data);       }
7   for (i = 0; i < n; i++)                  ...
8     data[i] = data[i] * data[i];           int *data;
9  ...                                       int n = 10, sz = n * sizeof(int)
10                                           PRFX_isomalloc(&data, sz);
11                                           PRFX_call(0,0.0,RPC_fill,n,&data);
12                                           PRFX_call(0,0.0,RPC_square1D,n,&data)
13                                           ...
```

L'exemple du listing 2.9 suppose que la fonction `RPC_fill()` accède en écriture aux `n` éléments de `data`. Dans le code incorrect, aussi bien la lecture de `data` que son écriture peuvent générer des erreurs de fonctionnement car la boucle `for` travaille en parallèle sur la même donnée que `RPC_fill()`. Cette incohérence peut être résolue par l'utilisation d'un second RPC, `RPC_square1D()`, reprenant le code de la boucle `for`. L'avantage, ici, est que la synchronisation et la cohérence entre les deux tâches sont garanties par le mode d'expression, si bien qu'une dépendance de donnée portant le tableau `data` existera entre `RPC_fill()` et `RPC_square1D`. Du point de vue des degrés de propriété, dans le code correct, la tâche devient propriétaire en écriture de `data` à la ligne 10 suite à l'iso-allocation. A la ligne 11, elle n'est plus que propriétaire virtuel et la tâche créée via `RPC_fill()` deviendra propriétaire en écriture lorsque ses dépendances seront satisfaites. Enfin (à la ligne 12) la tâche créée via `RPC_square1D()` deviendra à son tour propriétaire en écriture lorsque ses dépendances seront satisfaites.

Autre exemple plus sibyllin correspondant à une mise au carré de tous les éléments du tableau `data`, entrelacée avec un traitement `RPC_func()` sur ce dernier :

Listing 2.10 – Exemple faux et corrigé d'un accès post RPC.

```
1  // INCORRECT                                  // CORRECT
2  int*data;                                     void RPC_squareElem(int *v,int idx)
3  int n = 10, sz = n *sizeof(int);              {
4  PRFX_isomalloc(&data,sz);                       v[idx] = v[idx] * v[idx]
5  for(i = 0; i < n; i++)                        }
6   {                                            ...
7      data[i] = data[i] * data[i];              int *data;
8                                                int sz = n *sizeof(int);
9      PRFX_call(RPC_func,n,&data);                PRFX_isomalloc(&data,sz);
10  }                                            for(i = 0; i < n; i++)
11 ...                                          {
12                                                  PRFX_call(RPC_squareElem,data,i)
13                                                  PRFX_call(RPC_func,n,data);
14                                               }
```

Le listing 2.10 suppose que la fonction `RPC_func()` accède en écriture aux `n` éléments de `data`. Dans le code incorrect, des accès à `data` interviendront après des appels à `RPC_func()` (le dépliage de la boucle permet de le caractériser). Pour les besoins de l'exemple, le code correct est simple mais non optimal. En effet, l'intégralité du tableau `data` est transmise à `RPC_squareElem()` alors que ce dernier ne modifie qu'un élément. Pour remédier à ce problème, il faut créer et utiliser un *stencil* décrivant un seul élément du tableau `data`.

2.4 Inspecteur de programmes prévisibles

L'inspecteur (représenté au centre de la figure 1) fait partie du mécanisme d'inspection / exécution utilisé par le support PRFX. Ce moyen de découverte et de modélisation du parallélisme prévisible, combiné à l'espace d'adressage unique de l'iso-mémoire et au langage C, permet en amont de proposer un mode d'expression parallèle lisible et puissant. La lisibilité est obtenue via la grande liberté d'accès aux données (continuité d'accès, structuration par pointeur, etc...). La puissance d'expression (en particulier des algorithmes irréguliers) est obtenue grâce au modèle du DAG de tâche utilisé de façon sous-jacente.

Nous avons vu que l'inspection statique était nécessaire en raison du choix d'un ordonnancement statique pour avoir des performances. Nous avons opté pour l'inspection statique plutôt que la compilation pour nous autoriser à exploiter des programmes plus génériques mais prévisibles prenant en entrée un jeu de données (fichiers, paramètres) potentiellement structurant de niveau inspection.

Comme nous l'avons vu à la section 2.3 l'interface PRFX permet d'exprimer des propriétés algorithmiques mais également d'informer l'inspecteur sur les allocations et libérations des données parallèles, leur géométrie et leur mode des accès ainsi que sur les tâches du programme, leur placement et leur phase. Les données parallèles jouent un rôle primordial dans la découverte du parallélisme par l'inspecteur dans un programme PRFX. Les informations relatives aux données sont pertinentes car elles proviennent directement du programmeur. Comme nous l'avons vu, notre inspecteur ne fait donc d'analyse ni du code source ni même du code compilé. Il procède par une pré-exécution légère lui permettant de connaître les informations fournies par le programmeur de "façon claire et directe".

Les algorithmes n'accèdent pas toujours à leurs données de façon unitaire et sans intersection. Avec notre inspecteur, ces intersections entre les accès des tâches sont calculées automatiquement et au plus juste. En effet, d'une part l'iso-mémoire permet un référencement global pour gérer ces intersections simplement en interne par l'inspecteur. Ce même référencement est ensuite utilisé par l'exécuteur PRFX. D'autre part, une grille ayant un pas de 4 octets et alignée sur l'adresse `0x0` est utilisée pour enregistrer les accès. Ce niveau de résolution permet de décrire de façon fine la majorité des accès des algorithmes scientifiques prévisibles où l'usage de données de type flottant de 4 ou 8 octets est prédominant. Le résultat de cette pré-exécution est une modélisation fine de ces programmes par un DAG complet. Le résolveur de dépendances et la structure de données associée interne à PRFX sont le cœur de l'inspecteur. Dans notre cas, nous verrons que nous avons choisi de générer un DAG complet qui n'est pas optimal. Cela nous permet d'abaisser la complexité de l'algorithme de résolution des dépendances.

Pour que le résolveur fonctionne, le programmeur est supposé écrire un code dont l'exécution séquentielle est valide. L'inspecteur part de l'ordre séquentiel total entre les appels de tâches qu'il relâche en ordre partiel grâce à la connaissance des tâches et de leurs accès. La transformation de l'ordre total en ordre partiel intervient dès l'existence de parallélisme brut entre les tâches. Ce parallélisme est modélisé par un DAG de tâches complet qui est réutilisé à l'exécution. Le parallélisme est brut, car l'inspecteur n'augmente pas ce parallélisme par des artifices d'exécution. Cela serait possible en créant des copies de données parallèles rendant les accès indépendants ou en dupliquant les tâches. Un autre moyen d'action serait de pouvoir adapter le grain de l'application en décomposant (resp. fusionnant) les tâches automatiquement en des tâches plus petites (resp. grandes).

2.4.1 Perfection du DAG de tâches complet

Le modèle du DAG de tâches complet est utilisé par l'inspecteur pour décrire le parallélisme d'un programme PRFX prévisible. Ainsi, nous allons pouvoir proposer des optimisations opérant sur cette modélisation prévisionnelle du programme. Ces optimisations interviendront avant et pendant l'exécution parallèle.

Pour un ensemble de tâches ordonnées dont leurs accès sont connus, il peut exister plusieurs façons de tisser les dépendances de données. Elles conservent la validité du programme et conduisent donc toutes à ce que ce dernier produise le même résultat. Lorsque les dépendances sont décrites à la main, le temps investi dans ce travail fait partie de la programmation. Il peut donc être aussi long que nécessaire pour satisfaire les contraintes que le programmeur s'est fixées. Un DAG complet spécifié par le programmeur ou généré par un outil dédié à une classe de programmes est supposé tendre vers la perfection (DAG idéal). Par contre, lorsque ces dépendances sont générées automatiquement de façon générique pour tout type d'algorithmes prévisibles comme dans notre cas, il n'est pas possible de maintenir la même qualité. Qui plus est, l'algorithme de résolution des dépendances doit être de complexité raisonnable car il fait partie de l'exécution (en fait de la pré-exécution). Notre algorithme de résolution des dépendances est donc générique et de faible complexité. La seule perte de qualité par rapport au DAG parfait concerne les anti-dépendances. Nous verrons par la suite pourquoi certaines anti-dépendances sont inutilement générées. Cet accroc ne change pas l'ordre partiel entre les tâches par rapport à

un DAG parfait, le parallélisme reste donc finement exprimé. De plus, une anti-dépendance ne portant pas de donnée à transférer, cet accroc est peu coûteux en intra-processus (modification d'une variable de synchronisation en mémoire partagée) mais plus pénalisant en extra-processus (communication via appel bibliothèque).

2.4.1.1 Indépendance relative au déploiement et au matériel

Si le programme n'est pas paramétré par des aspects matériels (e.g. hiérarchie mémoire), alors le DAG généré sera indépendant de la configuration d'exécution. La phase d'ordonnancement assurera une prise en compte des caractéristiques de la configuration d'exécution et cette prise en compte se poursuivra au travers de l'exécuteur parallèle par une gestion dédiée à la machine.

Une possibilité parfois utilisée dans les algorithmes parallèles est leur paramétrage par la taille de découpage optimal (dépendant de la taille du cache processeur) des données parallèles. Ainsi, quel que soit le nombre de machines et de processeurs (homogènes) le DAG de tâches est identique et, une fois ordonnancé, les temps d'exécution de chaque tâche pris séparément reflètent un bon rendement. Si l'ordonnancement est bon (prise en compte des coûts de communications), alors le rendement global le sera également.

Enfin, d'autres algorithmes prennent en compte le déploiement et / ou des coûts associés au matériel. Ce paramétrage va, au final, adapter le DAG pour obtenir de meilleures performances dans la configuration de machine utilisée. Cette adaptation peut se faire via un grossissement (resp. affinement) du grain qui, pour un problème de taille donnée, provoquera la génération d'un DAG avec moins (resp. plus) de tâches.

2.4.2 Génération du DAG de tâches complet

La génération du DAG de tâches complet est effectuée lors de la pré-exécution séquentielle des tâches marquées par le programmeur. Ces dernières comprennent obligatoirement les tâches utilisateurs effectuant le contrôle (arbre des appels de RPC de l'application). Résoudre une dépendance de donnée revient à retrouver la dernière version, selon l'ordre séquentiel, des morceaux de la donnée recherchée. L'utilisation de cet ordre séquentiel nous permet de garantir la même sémantique que l'exécution séquentielle. La notion de dernière version d'une donnée est relative à la position dans le DAG de la tâche qui la requiert. Cette dernière version de la donnée est possédée en écriture par un nombre n restreint de tâches (i.e. borné par le nombre suffisant de sous-données pour paver la donnée en question). En revanche, dans un cas extrême toutes les tâches du DAG peuvent posséder cette même version de donnée en lecture. Si l'accès à résoudre est en écriture, alors tant que les n tâches en écriture n'ont pas été rencontrées, les tâches ayant accédé en lecture à tout ou partie de cette donnée doivent être synchronisées avec la tâche courante par une anti-dépendance. En fait, la condition est plus fine que cela car lorsque la donnée est morcelée, la résolution progresse avec une donnée courante recherchée dont la taille s'amenuise à chaque dernière version de morceau de donnée trouvée. Les informations relatives à la synchronisation et à la cohérence sont donc générées en tenant compte de cette décroissance de la donnée en cours de traitement.

TAB. 1 – Description des différentes dépendances possibles. A :anti-dépendance, D :dépendance de données, ? : dépendance non orientée.

→	RO	RW	FW	CRW
RO		A	A	A
RW	D	D	A	D
FW	D	D	A	D
CRW	D	D	A	?

A présent, intéressons-nous aux anti-dépendance inutiles. Elles surviennent si un chemin vient à exister entre les deux tâches présentes à ses extrémités. Nous allons tout d'abord voir une méthode permettant l'obtention d'un DAG sans anti-dépendances inutiles. Ensuite, nous abaisserons la complexité de cette méthode avec un compromis sur la perfection du DAG généré. Ce compromis ne nuit pas à la construction des dépendances de données (nécessitant un transfert) mais uniquement à la construction des anti-dépendances. La table 1 répertorie les différentes dépendances possibles entre une tâche effectuant initialement l'accès indiqué en première colonne et une tâche effectuant ensuite un accès indiqué par la première ligne.

2.4.2.1 Méthode de résolution exacte

Cette méthode consiste à générer toutes les dépendances de la tâche en cours puis à tester pour chaque anti-dépendance générée s'il existe un autre chemin entre la tâche en amont et la tâche courante. Si tel est le cas alors l'anti-dépendance double un chemin existant et peut de fait être supprimée, sinon elle doit être conservée. Cette méthode est coûteuse car il faut obtenir ou maintenir l'information sur l'existence d'un chemin entre les tâches du DAG.

2.4.2.2 Méthode de résolution imparfaite

Cette méthode de résolution est imparfaite car elle génère plus d'anti-dépendances que nécessaire. En revanche, les dépendances de données sont en nombre minimal. Pour ces dépendances, nous avons choisi de toujours les faire venir des tâches ayant accédé en écriture aux zones mémoire recherchées. Ces tâches sont les plus anciennes détentrices de la version de la donnée recherchée. D'autres méthodes sont possibles, car cette version de la donnée est parfois possédée par plusieurs autres tâches (plus récentes) effectuant des accès en lecture seule. Notre stratégie couplée avec la détection de *broadcast* assure la date de disponibilité la plus tôt possible et ce, sans redondance des envois car ces derniers sont gérés par un seul *broadcast*. Ainsi, l'ordonnanceur a une latitude maximale pour faire des optimisations en ordonnançant cette communication dans la fenêtre temporelle ici maximisée.

La méthode de résolution que nous avons choisie ne vérifie pas l'existence d'un chemin comme mentionné à la section 2.4.2.1. Elle applique juste les optimisations permises par les différents modes d'accès. Les anti-dépendances inutiles ne sont évitées que lorsqu'elles proviennent d'une tâche avec laquelle la tâche courante est déjà en relation.

Cette méthode se sert de l'interception des créations de tâches comme montré dans le code 2.11. Au cours du déroulement de la boucle `pour` de la ligne 4 à la ligne 12, les arguments sont traités séparément. Les accès sont enregistrés puis à la ligne 11 le résolveur de dépendances est appelé en prenant soin de soustraire aux futurs arguments à résoudre les zones mémoire des arguments déjà résolus (ligne 9).

Listing 2.11 – Interception du `PRFX_call` et appel au résolveur de dépendances.

```
1  Comportement de la création d'une tâche "tsk" lors de la pré-exécution
2
3       descripteur de donnée "donnee" <- descripteur de donnée vide
4       pour tous les arguments "arg" de "tsk" faire
5       {
6         descripteur de donnée "dscr" <- descripteur de la donnée "arg"
7         mode d'accès "mode" <- mode d'accès à "arg" par "tsk"
8         enregistrer que "tsk" accède à "dscr" selon  "mode"
9         soustraire "donnee" à "dscr"
10        ajouter "dscr" à "donnee"
11        résoudre les dépendances pour l'accès à "dscr" selon "mode" par "tsk"
12      }
```

Résolution des dépendances : Pour générer les dépendances de données d'un RPC, l'inspecteur recherche les *stencils* associés aux références de données parallèles passées en paramètre de ce RPC. Avec l'adresse de base donnée par la référence et la connaissance des modes d'accès de chaque paramètre du RPC (déclarés avec `PRFX_declRPC()`), il obtient les zones mémoire effectivement atteintes ainsi que leur mode d'accès. Comme nous l'avons vu, ces différents accès sont minimisés (lignes 9-10 du listing 2.11) avant d'être résolus. Cette minimisation évite de résoudre deux fois les parties communes lorsque les arguments d'un même RPC s'intersectent entre eux. La ligne 11 du listing 2.11 fait appel au résolveur de dépendance dont le pseudo-code est contenu dans le listing 2.12.

Il est à noter que si les zones référencées par un tableau de références ne sont pas décrites par le programmeur comme accédées, seul le contenu du tableau est transmis au RPC destinataire. L'environnement PRFX ne parcourt pas la donnée à la recherche de références pour sérialiser le graphe de données comme cela serait possible si nous utilisions un langage comme JAVA. Quand bien même cette opération serait effectuée, du fait de la simplicité du langage C, l'inspecteur ne serait pas capable de déterminer la conformation de la zone référencée et encore moins d'identifier récursivement des références dans ces zones successives. La description de structures de données complexes passerait par l'ajout d'informations sur les emplacements des chaînages et de leurs valeurs terminales (sorte de sérialisation guidée) dont nous ferons état au chapitre 5.

Listing 2.12 – Pseudo code de l'algorithme de résolution imparfaite d'un argument de tâche .

```
1  Algorithme de résolution des dépendances relatives à un accès à une donnée "a" par une tâche "tsk"
2
3  mode d'accès "mode" <- mode d'accès de "tsk" à "a"
4  descripteur de donnée "dscr" <- descripteur de "a"
5  liste de tâches "L" <- liste des tâches en relation avec "a"
6  tâche "ante" <- 1er élément de la liste "L"
7  mode d'accès "ante mode" <- mode d'accès de "ante" à la donnée en relation avec "a"
8  descripteur de donnée "ante dscr" <- descripteur de la donnée de "ante" en relation avec "a"
9
10 Tant que "dscr" non vide et "L" non vide faire
11 {
12      si "mode" est en écriture ou que "ante mode" est en écritrure faire
13         {
14                 descripteur de donnée "inter" <- intersection de "ante dscr" avec "dscr"
15                 si "inter" est non vide faire
16                   {
```

```
17                      si "ante mode" est en écriture faire
18                         soustraire "inter" à "dscr"
19
20                      si "mode" est en écriture totale (FW) ou que
21                         ( "ante mode" est en leture seule et que "mode" est en écriture)
22                        ajouter une anti-dépendance de "ante" vers "tsk"
23                      sinon si "mode" est en lecture seule (RO)
24                        ajouter un broadcast de donnée de "ante" vers
25                         "tsk" portant sur "inter"
26                      sinon
27                        ajouter une dépendance de donnée agrégée de
28                         "ante" vers "tsk" portant sur "inter"
29                  }
30          }
31      "ante"      <- élément suivant de la liste
32      "ante mode" <- mode d'accès de "ante" à la donnée en relation avec "a"
33  }
34  fin tant que
```

Aux lignes 3 et 4 du listing 2.12, les informations sur l'accès mémoire à résoudre ainsi que le premier élément de la liste des tâches en relation avec cet accès (i.e. faisant le même accès ou accédant à une sous-partie de la donnée) sont récupérés. Cette liste des tâches a deux types de contenus possibles suivant que l'accès à résoudre est de type lecture ou écriture. Pour un accès en écriture, cette liste contient toutes les tâches en relation quel que soit leur mode d'accès. Pour un accès en lecture, seules les tâches en relation et faisant un accès en écriture sont contenues dans cette liste car il n'y a pas de dépendance entre des accès en lecture. Ainsi, dans le cas de la résolution d'un accès en lecture d'une donnée (accès supposés en nombre majoritaire dans un programme parallèle), le nombre d'accès énuméré est égal au nombre d'accès en écriture sur les morceaux de la donnée.

A la ligne 10, démarre une boucle au cours de laquelle la zone mémoire à résoudre va s'amenuiser au fur et à mesure de la construction des dépendances satisfaisant des morceaux de la donnée. A la fin de cette boucle, des erreurs de programmation sont détectables lorsque la résolution n'est pas terminée et qu'il n'y a plus de tâches en relation dans la liste. Ce cas est rencontré par exemple lorsque le programmeur fait une erreur en déclarant des accès en dehors des zones qu'il a iso-allouées.

A la ligne 12, l'algorithme ne concerne plus que les accès en écriture (i.e. RW, CRW et FW), car sur un couple d'accès ordonnés dans le temps, il faut et il suffit que l'un des deux soit en écriture pour qu'une dépendance existe (rappel : le mode CRW n'est pas implémenté).

A la ligne 14, nous sommes obligés d'intersecter les descripteurs des accès pour savoir s'ils ont des parties de données parallèles communes. Ceci est utile car la liste des accès antérieurs que nous parcourons a été construite pour l'accès de l'argument initial. Or, la zone mémoire courante à résoudre s'amenuise au cours du déroulement de la boucle, elle peut donc ne plus avoir de relation avec des tâches de cette liste.

De la ligne 17 jusqu'à la ligne 26, l'algorithme traite le cas où cette intersection n'est pas vide. Si l'accès antérieur est en écriture, l'intersection trouvée est résolue et elle peut donc être soustraite à l'accès en cours de résolution.

2.4.2.3 Génération des communications

Les communications sont calculées par intersection des descripteurs des données parallèles en jeu. Lors de la pré-exécution, la description d'une communication est indépendante de la localisation physique de la donnée à transférer. La communication est décrite et stockée sous

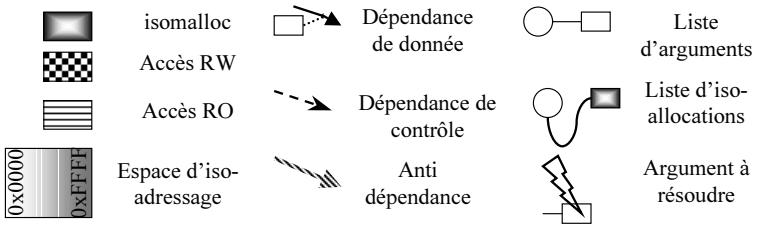

FIG. 3 – Légende pour les schémas de résolution de dépendances.

forme de *stencils* et d'informations décrivant comment appliquer le *stencil* relativement à une adresse de base, lors de l'exécution. Cette adresse est retrouvée grâce à un numéro d'argument ou un numéro d'iso-allocation. Si besoin est, un *offset* peut être rajouté à l'adresse de base servant à l'application du *stencil* dans la donnée parallèle.

L'enregistrement des dépendances se fait sous forme de tâches de gestion internes à PRFX. Ces tâches peuvent être ordonnancées et placées comme les tâches utilisateur mais avec la contrainte d'être dans le même processus que la tâche utilisateur à laquelle elles sont rattachées.

Aussi, une tâche interne de contrôle `cotsk` est créée pour traiter chaque `PRFX_call()`. Une tâche `Factocommtsk` est créée pour les données à envoyer afin d'être accédées en écriture ; ce type de tâche est celui par défaut. Toutes les données à envoyer provenant de la même tâche et allant vers une même tâche destination, sont agrégées et effectuées par une seule tâche `Factocommtsk`. Le fait de répertorier ces messages agrégés permet juste d'obtenir des gains liés au matériel en factorisant la latence des communications.

En revanche, il existe un problème de performances plus critique dans le cas où un message est diffusé en sortie d'une tâche et dans le cas où cette diffusion n'est pas détectée. En effet, le message est alors envoyé, autant de fois qu'il y a de tâches cibles. La redondance existe dès que le nombre de processus cible est inférieur au nombre de tâches ciblées. Enfin, si les messages diffusés ne sont pas répertoriés, les fonctionnalités de diffusion optimisées proposées par les bibliothèque de communication ne peuvent être utilisées.

Une tâche `Multicommtsk` est créée pour chaque diffusion (*broadcast*) de données parallèles en sortie de tâche utilisateur. Ce type de tâche est recherché et maintenu uniquement pour des données parallèles accédées en lecture seule. Il ne peut d'ailleurs pas y avoir de *broadcast* vers des destinations en écriture. En effet, est-il besoin de préciser que la diffusion concerne la même donnée parallèle (en iso-mémoire), et si deux destinations t1 et t2 écrivent la même donnée provenant de t0 alors t2 dépend de t1 et non de t0.

En plus de ces tâches internes de gestion des communications, l'inspecteur génère deux autres types de tâches internes de contrôle : les `pretsk` et les `posttsk`. Elles ont les même contraintes de placement que leurs tâches homologues de communication. Elles sont ajoutées pour chaque tâche utilisateur dont elles précèdent et suivent l'exécution comme leur nom l'indique. Ces tâches internes sont prises en compte par l'ordonnanceur et mises en œuvre par l'exécuteur.

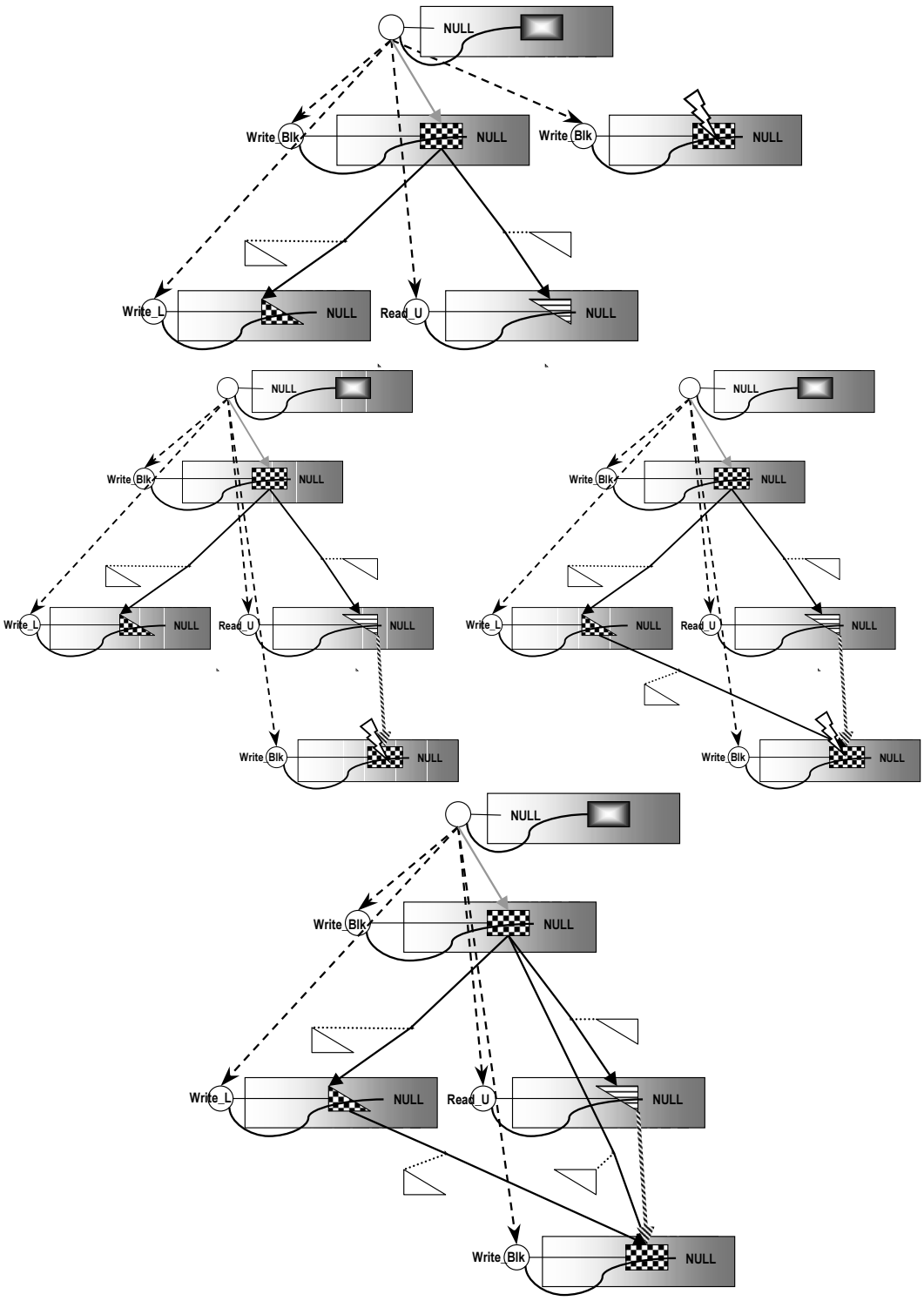

FIG. 4 – Résolution du premier argument du RPC (ligne 20, code 2.13) sur `Write_Blk()`.

2.4.2.4 Exemple de résolution de dépendances

Le code 2.13 présente la simplicité d'expression d'un programme parallèle mettant en jeu des intersections de données grâce à la résolution automatique des dépendances. Dans ce code, le programmeur exprime les accès (bloc plein, triangulaire supérieur et inférieur) qu'il souhaite effectuer sur un bloc de données. Pour ce faire, il déclare les RPC et leurs modes d'accès, puis il iso-alloue le bloc et déclare les *stencils* triangulaires dont il aura besoin. Enfin, il écrit son algorithme très simplement en lançant les quatre RPC qui correspondent respectivement à l'écriture du bloc, à la lecture et à l'écriture de la partie triangulaire inférieure du bloc, à la lecture de la partie triangulaire supérieure du bloc et enfin à la lecture et à l'écriture du bloc.

Listing 2.13 – Code avec des accès irréguliers servant d'exemple à la résolution de dépendances.

```
1  int main(int argc, char*argv[])
2  {
3    [...]
4      elem_t *L,*U,*blk;
5
6    PRFX_init();
7    PRFX_decl2(Write_Blk,0, PRFX_IMM, PRFX_RW,NULL);
8    PRFX_decl2(Write_L,0, PRFX_IMM, PRFX_RW,NULL);
9    PRFX_decl2(Read_U,0, PRFX_IMM, PRFX_RO,NULL);
10
11   PRFX_isomalloc(&blk, dim*dim*sizeof(elem_t));
12   PRFX_stencilDeclLEDiag( &L, dim*sizeof(elem_t), dim, dim*sizeof(elem_t), sizeof(elem_t));
13   PRFX_stencilDeclUEDiag(&U, dim*sizeof(elem_t), dim, dim*sizeof(elem_t), sizeof(elem_t));
14   L = blk;
15   U = blk;
16
17   PRFX_call(-1,0.0, Write_Blk, dim, &blk);
18   PRFX_call(-1,0.0, Write_L,   dim, &L);
19   PRFX_call(-1,0.0, Read_U,    dim, &U);
20   PRFX_call(-1,0.0, Write_Blk, dim, &blk);
21
22   PRFX_terminate();
23   return 0;
24 }
```

La Figure 4 présente le déroulement de l'algorithme de résolution automatique des dépendances pour le cas du premier argument du dernier RPC : `Write_Blk()` (ligne 20 du code 2.13).

Pour comprendre le schéma de la figure 4, nous explicitons d'abord sa légende indiquée par la figure 3. Cette légende comporte en première colonne les créations et les accès aux données dans l'espace d'iso-adressage. La deuxième colonne répertorie la dépendance de contrôle et les différentes dépendances de données. La dépendance de donnée porte la donnée à transférer. Les dépendances de contrôle correspondent aux créations de tâches (envoi des références et données immédiates passées en paramètre des RPC). Les anti-dépendances correspondent à une synchronisation (message vide) d'une tâche accèdant à une donnée en lecture vers une tâche accèdant à la même donnée, mais en écriture. Enfin, la troisième colonne permet de décrire les données attachées aux tâches comme par exemple leur liste d'arguments et celle de leurs allocations. Pour suivre le déroulement de l'inspection sur les schémas, l'argument en cours de résolution est affublé du symbole de l'éclair.

Dans la figure 4, le premier schéma représente l'état du DAG lorsque l'inspecteur s'apprête à résoudre l'argument de la tâche `Write_Blk`. La tâche correspondant à ce RPC est positionnée à un niveau de profondeur en dessous de l'initiateur car la seule dépendance résolue, ou plutôt connue à cet instant, est celle de contrôle. Lors de la résolution de l'accès en écriture du bloc rectangulaire, le résolveur va parcourir la liste des tâches ayant fait des accès en relation avec ce

bloc dans l'ordre anti-séquentiel. Il s'intéressera donc successivement à la tâche `Read_U` puis à la tâche `Write_L` et enfin à la tâche `Write_Blk`.

Dans le schéma central à gauche, le premier accès en relation, effectué par `Read_U()`, est un bloc triangulaire supérieur en lecture seule : une anti-dépendance est alors construite. Ensuite, dans le schéma central droit, l'accès en relation, effectué par `Write_L()` est en écriture : une dépendance qui permettra de retrouver la partie triangulaire inférieure à l'exécution est construite. Dorénavant, suite à cette résolution sur un morceau de la donnée, l'accès à résoudre ne concerne plus que la partie triangulaire supérieure du bloc initialement recherché. Enfin, dans le schéma du bas, l'accès antérieur concerne tout le bloc. L'intersection avec l'accès courant à résoudre donne la partie triangulaire supérieure manquante. La dépendance satisfaisant cet accès est construite. Comme l'accès antérieur est en écriture, la partie en relation (intersection) est soustraite si bien que l'accès à résoudre est désormais vide. Ceci arrête le résolveur pour cet argument et également pour cette tâche car elle n'avait qu'un argument.

2.4.2.5 Exemple de limitation de la méthode imparfaite

Pour le code 2.14, deux DAG différents pourraient être générés par un résolveur automatique selon qu'il teste ou non l'existence d'un chemin lors de la construction des anti-dépendances.

Le schéma de gauche de la figure 5 illustre l'état du DAG après la résolution du deuxième argument du RPC `Write_blk2()`. Cette résolution provoque la génération de deux anti-dépendances qui vont s'avérer inutiles. En effet, le schéma de droite de la figure 5 présentant l'état final montre l'existence d'un chemin court-circuité par chacune des deux anti-dépendances. Cette situation se produit parce que la résolution suivante, concernant le premier argument du RPC `Write_Blk2()`, rajoute un arc qui crée deux chemins depuis ces deux tâches d'où partent les anti-dépendances générées lors de l'étape précédente. Par contre, (voir la figure 6), en testant l'existence d'un chemin, cela permettrait de produire un DAG sans anti-dépendances inutiles.

Listing 2.14 – Code pour lequel le risque d'anti-dépendances inutiles existe .

```
1  int main(int argc, char*argv[])
2  {
3    [...]
4    PRFX_init();
5    PRFX_decl5(Write_Blk3,0, PRFX_IMM,PRFX_IMM, PRFX_RW, PRFX_RW, PRFX_RW);
6    PRFX_decl4(Write_Blk2,0, PRFX_IMM,PRFX_IMM, PRFX_RW, PRFX_RW);
7    PRFX_decl4(Read_Blk,0,   PRFX_IMM,PRFX_IMM, PRFX_RO, PRFX_RO);
8
9    PRFX_isomalloc(&blk1, dim*dim*sizeof(elem_t)/2);
10   PRFX_isomalloc(&blk2, dim*dim*sizeof(elem_t));
11   PRFX_isomalloc(&blk3, dim*dim*sizeof(elem_t));
12
13   PRFX_call(-1,0.0, Write_Blk3, dim/2, dim, &blk1, &blk2, &blk3);
14   PRFX_call(-1,0.0, Read_Blk,   dim/2, dim, &blk1, &blk3);
15   PRFX_call(-1,0.0, Read_Blk,   dim/2, dim, &blk1, &blk3);
16   PRFX_call(-1,0.0, Write_Blk2, dim/2, dim, &blk1, &blk2);
17   PRFX_call(-1,0.0, Write_Blk2, dim,   dim, &blk2, &blk3);
18
19   PRFX_terminate();
20   return 0;
21 }
```

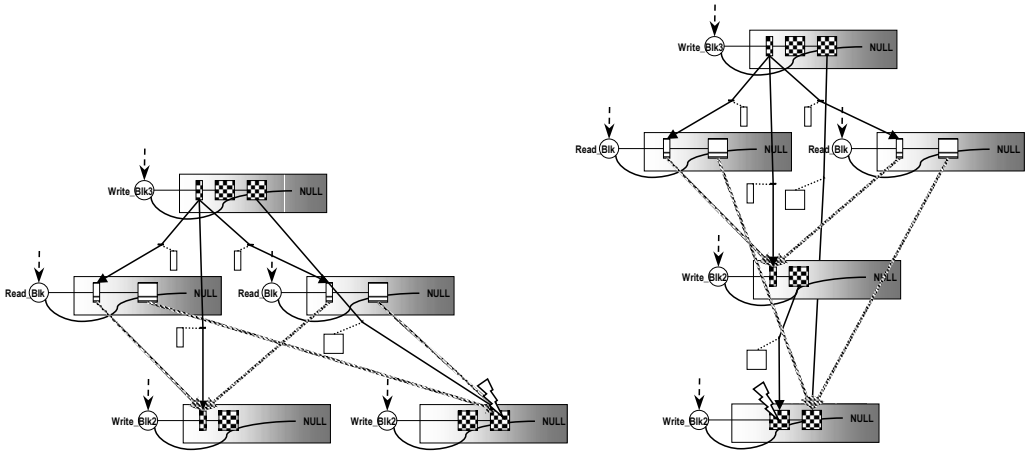

FIG. 5 – DAG avec deux anti-dépendances inutiles lors de la résolution des arguments du dernier RPC `Write_Blk2()` (code 2.14).

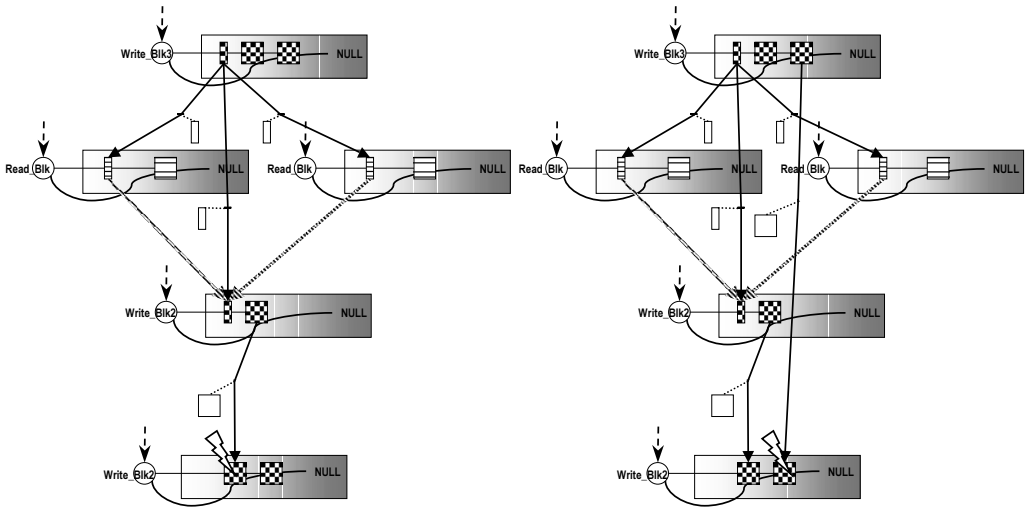

FIG. 6 – DAG sans anti-dépendances inutiles lors de la résolution des arguments du dernier RPC `Write_Blk2()` (code 2.14).

2.4.2.6 Coût de l'inspection

Le coût par paramètre de tâche est celui d'une recherche pour retrouver où insérer l'accès dans l'arbre de l'historique des accès aux données, arbre dont nous parlerons au chapitre Implémentation, et celui d'une insertion dans l'arbre si l'accès n'existe pas. Ensuite la tâche est mise en tête de liste pour cet accès.

Une fois l'accès trouvé, il faut parcourir l'historique des tâches ayant des intersections avec cet accès dans l'ordre anti-séquentiel et ajouter les dépendances (ou anti-dépendances) jusqu'à ce que l'accès soit totalement satisfait. Ce coût dépend donc du nombre de tâches en RO sur tout ou partie de la donnée et du nombre de tâches en RW nécessaires à la résolution de toutes les parties de la donnée. Dans le pire des cas, seule la tâche `main()` initiale a accédé à la donnée en écriture et toutes les autres ont effectué un accès en lecture.

2.4.2.7 Cas d'une cohérence uniquement par SM ou DSM

Supposons à présent que l'utilisateur laisse la gestion du contrôle de flot et de la synchronisation à PRFX mais confie la gestion de la cohérence des données à une DSM ou à une SM. Dans ce cas, le fonctionnement du mécanisme inspection / exécution resterait le même. Par contre, une différence apparaîtrait au niveau des communications engendrées par la gestion de la cohérence par cette SM ou DSM. Cette différence existera car les DSM ou SM n'ont pas la connaissance comme PRFX des futurs accès mémoire du programme. Il se produira donc des fautes de pages ou de cache inopinées au fur et à mesure que les tâches cibles s'exécuteront. Il en résultera une exécution hachée ou ralentie et un accaparement de plus de données que nécessaire du fait de la granularité par taille des pages (de 4 ko à 16 Mo) ou de cache dans une moindre mesure. Ceci ne correspond pas au modèle où les tâches s'exécutent de façon atomique et où les données sont communiquées avant le démarrage de la tâche. Ce phénomène à l'exécution se produit car les accès des tâches provoquent des allers-retours des pages ou des lignes de cache accédées entre les processeurs ou nœuds y accédant. Ce phénomène se produit à l'intérieur d'un nœud SMP, où le programmeur peut choisir de confier la cohérence au matériel plutôt qu'à PRFX en utilisant des *threads*. Dans ce cas, une partie du travail de l'inspecteur, qui consiste à calculer les intersections de données pour que l'exécuteur puisse en assurer la cohérence, ne sert à rien. En effet, pour la partie du DAG exécutée en intra-nœud, les tâches de gestion des précédences sont lancées mais se terminent en effectuant juste la synchronisation dès qu'elles détectent que la communication concerne une tâche intra-processus.

Dans le cas de programmes prévisibles, le DAG d'exécution n'est qu'une succession de traitements et de transferts de données connus à l'avance. Dans ce cas l'environnement PRFX devrait pouvoir atteindre les mêmes performances et la même scalabilité qu'un programme multi-*threads* utilisant le passage de messages. En effet, notre gestion de la cohérence, contrairement à une DSM ou SM classique, n'est pas bridée par l'utilisation du réseau. Ceci est possible car sa gestion est décrite dans le DAG sous forme de communications unilatérales, si bien que le surcoût de gestion sera approximativement le même que celui d'un modèle de type passage de messages. Les possibilités d'ordonnancement générique brident un peu l'exécution. Nous ne pouvons exploiter l'enchaînement séquentiel des instructions du code compilé contrairement à

une implémentation (e.g. avec MPI) statique et à placement fixe du programme utilisateur. Nous avons un intermède entre chaque tâche, qui est plus coûteux que le passage d'une instruction à la suivante.

Nous ne pouvons pas non plus compacter le DAG comme le ferait un programmeur d'applications parallèles en exprimant de façon factorisée la synchronisation et la cohérence directement dans le code de son programme.

2.5 Ordonnanceur statique de DAG de tâches

L'ordonnanceur (situé à droite de la figure 1) traite le DAG de tâches avec l'heuristique potentiellement choisie et paramétrée par l'utilisateur. L'ordonnanceur utilise une heuristique car le problème qu'il doit résoudre est NP-complet ([66]). Les heuristiques proposées sont au nombre de deux : celle dites du "chemin critique" et celle du "démarrage au plus tôt des tâches". D'autres heuristiques existent ([23],[34], [35], [42], [58], [67]) et peuvent être rajoutées pour exploiter les nombreuses informations du DAG complet.

Les heuristiques proposées utilisent une technique classique basée sur une liste de tâches. Leur fonctionnement est divisé en deux phases. Lors de la première, les tâches (ici utilisateur) du DAG sont ordonnées dans une liste selon une priorité donnée (dépendante de l'heuristique) en respectant leurs contraintes de précédences. Lors de la seconde phase, les tâches sont considérées une par une dans l'ordre de cette liste et à chaque fois assignées au processeur (*thread* exécuteur dans notre cas) qui minimise sa date de terminaison. Leur complexité est de l'ordre de $O(|V|log|V|+|P|(|V|+|E|))$ avec V l'ensemble des tâches, E l'ensemble des communications et P l'ensemble des processeurs. De plus comme nous ordonnançons les communications, nous avons un surcoût de l'ordre de $O(|E|log|E|)$. Si l'on suppose que $P << |V|$ alors cette dernière complexité est la borne haute des algorithmes d'ordonnancement proposés. Nous offrons la possibilité de paramétrer ces heuristiques pour les forcer à respecter les phases d'exécutions spécifiées en paramètre des `PRFX_call()` par le programmeur.

Les heuristiques sont libres de ne pas respecter la distribution des tâches spécifiée par le programmeur. En effet, cela ne nuit en rien à la validité de l'exécution. Elles peuvent donc rechercher un meilleur placement des tâches, parmi un nombre de *threads* candidats plus ou moins restreint. Cette recherche se fait relativement au placement initial de la tâche avec la restriction à :

- l'unique *thread* conseillé par le placement utilisateur ;
- aux *threads* évoluant sur la même carte de processeurs ;
- aux *threads* d'un même processus ;
- à l'ensemble des *threads* du déploiement.

En l'absence d'heuristique précisée, l'heuristique dite du "chemin critique" est utilisée, avec la contrainte d'un unique *thread* candidat mais sans contrainte du respect des phases.

L'ordonnancement doit traiter les tâches en tenant compte des spécificités et des contraintes de placement des tâches internes de PRFX relativement aux tâches utilisateurs auxquelles elles sont attachées.

Dans le cas de PRFX, un DAG complet est fourni à l'ordonnanceur. Les schémas de communication (messages agrégés, *broadcast*) de ce DAG sont identifiés et les modèles de coûts (tâches et communications) sont fournis (cf. droite de la figure1). En sortie, l'ordonnanceur travaille pour l'exécuteur parallèle PRFX. Ce dernier est capable de réaliser n'importe quel ordonnancement d'un DAG de tâches complet grâce à la migration des données. De plus, il propose des fonctionnalités optimisées pour les envois de données diffusées et agrégées. L'algorithme d'ordonnancement n'a donc pas à être bridé pour répondre à des manques de fonctionnalités au niveau de l'exécuteur. Il doit fournir à ce dernier un fichier de vecteurs d'ordonnancement et éventuellement un déploiement amélioré (e.g. plus restreint si les communications sont coûteuses).

Toutes les informations (placement initial, phase, accès aux données, fonctions exécutées par les tâches, *broadcast* et messages agrégés) en entrée et fonctionnalités en sortie (*threads* exécuteurs, tâches atomiques et migration) recouvrent, voire dépassent, les besoins des heuristiques d'ordonnancement présents dans la littérature. L'application de ces dernières va permettre d'agir de façon externe et générique sur l'exécution du programme pour que ses performances soient meilleures sur l'architecture visée. Les heuristiques d'ordonnancement de la littérature que nous souhaitons utiliser via une bibliothèque construisent et utilisent un diagramme de Gantt pour modéliser l'occupation des processeurs dans le temps. Le modèle d'exécution de ces heuristiques suppose que, lorsqu'une tâche débute son exécution, elle n'est plus jamais interrompue jusqu'à sa terminaison. Cette supposition permet de simplifier les algorithmes. Elle permet accessoirement aux modèles de coûts des tâches utilisés d'être plus fiables. En effet, dans le cas où les processeurs entrelacent l'exécution de plusieurs tâches (avec des *threads*), le coût de chaque tâche est influencé par les modifications du cache intervenues lors de la préemption du processeur par la tâche concurrente.

L'algorithme d'ordonnancement est choisi par l'appel à `PRFX_setStaticScheduler(int heuristique, int distance, int phase)`. Cet appel prend en paramètre le choix de l'heuristique, celui de la distance de remise en question du placement utilisateur et celui du respect des phases utilisateur.

La phase est utilisable pour forcer l'ordonnanceur à séparer l'exécution du programme en phases définies par le programmeur. Ce dernier peut par exemple utiliser trois valeurs croissantes, tout d'abord pour les RPC initialisant les données parallèles, puis pour ceux effectuant les calculs et enfin ceux stockant les résultats du programme. Cette séparation est utile pour déboguer le programme en évitant un mélange des tâches. Cela permet également de faire des mesures de performances sur des parties du programme. En effet, si la phase n'est pas utilisée, l'ordre d'exécution des tâches obéit uniquement à des contraintes de coûts et de précédences. Ceci conduit à un mélange entre toutes les tâches de même niveau du programme car elles sont par exemple toutes prêtes en terme de date de démarrage. L'ordonnanceur respecte l'ordre des phases uniquement par CPU pris séparément. Ainsi, pour un processeur donné, les valeurs des phases des tâches qu'il exécute sont croissantes. La définition des phases ne permet pas d'effectuer une barrière de synchronisation entre les tâches.

L'ordre partiel sur les RPC imposé par la phase doit être compatible avec l'ordre partiel dû aux précédences entre les tâches. Il est à la charge du programmeur de s'en assurer.

2.5.1 Modèle de communications et d'exécution

Un modèle de coût des communications à deux niveaux (intra et extra-nœud SMP) et le temps pris par l'exécuteur entre deux lancements de tâches successifs sont mis à disposition de l'ordonnanceur par PRFX. L'ordonnanceur accède à ces informations avec une fonctionnalité prenant en entrée la taille du message à envoyer et le couple source-destination, et retournant le temps pris par cette opération. La topologie de la machine parallèle est supposée être en étoile, elle n'est donc pas indiquée. La hiérarchie *thread* / processus de l'exécution et mémoire / réseau du matériel peut être récupérée par l'ordonnanceur via les informations du déploiement fournis par PRFX.

Les coûts des tâches sont fournis par le programmeur par l'intermédiaire de fonctions ayant le même prototype que la fonction cible du RPC mais renvoyant une structure de type `PRFXtaskcost`. Cette dernière contient le nombre d'instructions et le pourcentage de puissance de crête atteint lorsqu'elle est appelée avec les vrais paramètres de la tâche.

Ces modèles sont pertinents si un régime de fonctionnement dédié des grappes de SMP est observé.

L'ordonnancement doit également tenir compte des caractéristiques de la bibliothèque de communication qui est une source potentielle d'écart par rapport aux modèles théoriques. Pour cela, il peut confiner sur un ou des *threads* exécuteurs les appels vers cette bibliothèque si leur comportement n'est pas fiable (implémentation de la couche communication non conforme à la spécification). L'ordonnancement peut également tenir compte des possibilités de communications simultanées en répartissant les tâches de communication sur plusieurs processeurs.

L'ordonnanceur peut tenir compte des accès aux données enregistrés dans le DAG complet pour calculer une distribution des données. Il peut également favoriser les effets de cache en plaçant les tâches utilisant les mêmes données sur des processeurs ayant un cache mémoire commun.

Il serait également possible d'éviter de saturer les liens réseaux ou bus mémoires en répartissant dans le temps les communications ou les tâches utilisateur consommant beaucoup de bande passante mémoire.

2.5.2 Heuristiques à liste de tâches proposées

Les deux heuristiques du "chemin critique" et de "la date de démarrage au plus tôt des tâches" sont légèrement modifiées pour intégrer les deux paramétrages concernant la phase et la remise en cause du placement à distance variable.

Lors de l'application de l'algorithme de l'heuristique, les tâches de l'utilisateur sont triées en tenant compte d'abord des phases puis des dépendances. Un *thread* exécuteur de destination leur est assigné selon les conseils de distribution du programmeur. Cette liste des tâches est ensuite parcourue avec un traitement glouton. Lors de ce parcours, l'assignation initiale peut être remise en cause si le programmeur le choisit ; dans ce cas l'ordonnanceur cherche un meilleur *thread* exécuteur à la distance voulue.

De l'ordonnancement des tâches nous déduisons par un simple tri l'ordonnancement des communications. Les tâches internes de gestion des communications sont triées en fonction des

dates théoriques de terminaison et de démarrage des tâches utilisateurs sources et destinations. Ainsi, avec cet ordre, une communication satisfait à chaque fois la tâche en ayant le besoin le plus urgent.

Dans le cas de l'heuristique utilisant l'information relative au chemin critique du DAG de tâches, à chaque tâche est assignée la somme des temps (calcul plus communication) n'écessaire à l'exécution du plus court chemin atteignant une tâche terminale. Le chemin critique est évalué dans le cas d'une exécution idéale (nombre infini de processeurs, absence de contention réseau / mémoire). Cette somme est utilisée pour trier les tâches dans un ordre décroissant d'importance.

L'heuristique utilisant la date de démarrage au plus tôt des tâches du DAG, calcule quant à elle cette date pour chaque tâche en supposant également une machine idéale. Cet ordre total sur les dates des tâches est ensuite utilisé pour construire les vecteurs d'ordonnancement de chaque *thread* exécuteur.

2.5.3 Calcul et amortissement de l'ordonnancement

L'ordonnancement permet d'équilibrer la charge des processeurs pour en maximiser l'utilisation (date de fin quasi identique pour tous) et ainsi tenter de faire mieux qu'une exécution aléatoire (coût d'ordonnancement nul).

Calculer un ordonnancement n'est pas toujours rentable par rapport à une version aléatoire. Pour un algorithme donné, plus le nombre de processeurs est réduit, plus le nombre de tâches prêtes à un instant t est important. Lorsque la somme des durées d'exécution des tâches prêtes à chaque instant est importante, le calcul d'un ordonnancement n'apportera pas de bénéfices car n'importe quel ordre produira un résultat satisfaisant (processeurs occupés constamment).

Lorsque l'application est suffisamment couplée (algorithmes scientifiques), que le nombre de processeurs est grand et que la machine présente une hétérogénéité (grappes de SMP), l'ordonnancement est rentable. Il permet de minimiser les temps d'inactivité et de recouvrir les calculs par des communications. Ce recouvrement est possible, soit en assignant la gestion des communications à un processeur sacrifié dans le cas où la bibliothèque de communication n'offre que des primitives bloquantes, soit en les entrelaçant avec des calculs en tenant compte des durées à recouvrir.

Le coût de l'ordonnancement doit rester en dessous de la complexité de l'algorithme. Par exemple, supposons qu'un algorithme parallèle traite t données de taille k. Supposons également que le nombre de tâches soit linéaire par rapport au nombre de données (i.e. de l'ordre de $O(\lambda t)$) et que chaque tâche traite une sous-donnée de taille k avec un algorithme d'une complexité de $O(k^2)$. Alors la complexité totale de l'algorithme parallèle est de $O(\lambda t k^2)$. La complexité des algorithmes d'ordonnancement que nous proposons est bornée par $O(|E|log|E|)$ le nombre de tâches du DAG. Si l'on suppose que le DAG de tâche est raisonnablement couplé avec un degré moyen $\Delta << |E|$ alors cette complexité s'abaisse à $O(\Delta|V|log[V|)$. Dans le cas de notre problème traitant t données, l'ordonnancement aura une complexité de $O(\Delta\lambda t log(\lambda t))$. Cette dernière est raisonnable par rapport à celle de l'algorithme car le coût d'ordonnancement par tâche est de

l'ordre de $O(\Delta log(\lambda t))$ alors que le coût d'exécution de chaque tâche est de l'ordre de $O(k^2)$. Si la complexité de l'ordonnancement est équivalente ou supérieure à celle de l'algorithme, il est nécessaire d'effectuer un amortissement sur plusieurs exécutions lorsque l'algorithme le permet.

2.6 Exécuteur parallèle

La réalisation d'un ordonnancement n'est effective que lorsque le support d'exécution est capable de reproduire le scénario du diagramme de Gantt théorique prévu par l'ordonnanceur. L'exécuteur n'est pas un exécutable à part, il fait partie de la bibliothèque PRFX. Il est donc inclus (via l'édition des liens) dans tous les exécutables parallèles compilés avec la bibliothèque PRFX. Cet exécuteur peut intervenir à la suite de l'inspection (droite de la figure 1) ou être utilisé à part comme indiqué en bas de la figure 1. Dans les deux cas, l'exécuteur utilise le DAG complet du programme à exécuter et les vecteurs d'ordonnancement, la variation se situe au niveau du format d'entrée. Dans le premier cas, les données sont directement accessibles en mémoire alors que dans le second cas, elles proviennent de fichiers.

Nous avons choisi d'utiliser les *threads* pour exécuter les tâches. Toutefois, nous ne créons pas un *thread* pour chaque nouvelle tâche à exécuter. Ainsi, nous factorisons la latence de mise en place d'une tâche. Le fait d'avoir un nombre fixe de *threads* permet également au programmeur de mieux comprendre le déroulement de l'algorithme et donc d'intervenir plus facilement (e.g. sur le déploiement de la plate-forme ou sur le placement des tâches).

L'exécuteur est un code qui est exécuté en parallèle par tous les *threads* exécuteurs PRFX. Ce code déroule une liste de tâches en faisant soit de l'attente active paramétrable, soit en passant la main à un autre *thread* (*time slicing*).

L'exécuteur *standalone* prend en paramètre plusieurs fichiers d'entrée nécessaires à son fonctionnement : le DAG de tâches complet, les vecteurs d'ordonnancement et le déploiement. Le fichier exécutable contient le code des tâches à exécuter, mais comme nous l'avons vu ce n'est pas réellement un fichier d'entrée car la bibliothèque PRFX est lié avec ce binaire.

2.6.1 Stockage du DAG de tâches complet

Listing 2.15 – Fichier de DAG au format PRFX.

```
1  nbtsk 108306 nbfactocntr 17468 nbarg 19306 nbdscr 789 nbfunc 3
2  #-------------------------------------------------------------
3  dscr 0 nbrng 1 lowestoff 0 highestoff 8
4  base 0 count 8
5
6  dscr 1 nbrng 1 lowestoff 0 highestoff 1040
7  base 0 count 16
8  base 1024 count 16
9
10 [...]
11
12 dscr 105 nbrng 1 lowestoff 0 highestoff 328680
13 base 0 count 328680
14 #-------------------------------------------------------------
15 pretsk 0 oftsk 1
16 usrtsk 1 nbdep 0 nbco 19305 func 11 isfuzzconcerned 0 isargdscrused 0 argaccessnum 0
17 cotsk 2 calltsk 192
18 cotsk 3 calltsk 193
19 [..]
20 cotsk 10 calltsk 200
21 posttsk 2 oftsk 1
```

```
22
23  Multicommtsk 3 oftsk 1 taskdest 192 nbdscr 2
24  Mdscr 80 arg 4 offset 0
25  Mdscr 62 iso 4 offset 0
26
27
28  Factocommtsk 4 oftsk 1 nbdest 2 cntr 15812
29  Fdscr 7 arg 3 offset 0
30  task 193
31  task 194
```

Le fichier permettant d'archiver et de réutiliser un DAG de tâches complet est composé d'un en-tête et de deux parties comme dans l'exemple du listing 2.15. L'en-tête permet de connaître de la gauche vers la droite, le nombre total de tâches (internes plus utilisateur), le nombre de *broadcast*, le nombre d'arguments et le nombre de fonctions RPC de l'utilisateur.

La première partie, ici de la ligne 1 à la ligne 13, contient tous les *stencils* de données. Chaque *stencil* a un en-tête avec un numéro, le nombre d'intervalles qu'il contient et deux valeurs le bornant (boîte englobante). Après, suivent les intervalles d'adresses contenus dans le *stencil*.

La deuxième partie, de la ligne 15 à la fin de l'exemple, contient la description des tâches. Dans cette description, on retrouve les deux grands types de tâches, à savoir les tâches utilisateurs (`usrtsk`) et les tâches de gestion internes à PRFX (`pretsk`, `cotsk`, `posttsk`, `Multicommtsk` et `Factocommtsk`). Les tâches de type `usrtsk` sont répertoriées comme par exemple à la ligne 16 par un numéro et des informations sur leur nombre de précédences, le nombre de `PRFX_call()` qu'elles effectuent, l'identificateur de la fonction cible, l'usage des *stencils* des arguments et l'identificateur de la zone de stockage des arguments.

Parmi les tâches de gestion, les tâches `pretsk` et `posttsk` sont automatiquement présentes pour chaque tâche utilisateur dont elles encadrent l'exécution. Les tâches `cotsk` correspondent aux `PRFX_call()` effectuées par la tâche utilisateur associée, elles sont décrites par leur propre numéro et le numéro de la tâche cible à appeler. Les `Factocommtsk` correspondent à des agrégations de données avant leur envois vers un destinataire. Leur description se fait en plusieurs lignes, la première contient leur numéro, le numéro de la tâche à laquelle elles sont rattachées, la tâche destination et le nombre de messages qui lui sont destinés. Vient ensuite la liste des *stencils* avec leur protocole d'application. Ce dernier indique dans quelle donnée parallèle le *stencil* doit être appliqué (adresse de base de la donnée), soit par un numéro d'iso-allocation, soit par un numéro d'argument.

Les tâches `Multicommtsk` correspondent à des diffusions de données (*Broadcast*). Leur description est également sur plusieurs lignes. La première décrit le numéro de la tâche à laquelle elle est rattachée, le nombre de tâches destination et le numéro de compteur commun à utiliser pour la synchronisation. La deuxième ligne contient le *stencil* de la donnée à diffuser avec son numéro, sa base d'application et l'*offset*. Ensuite, il y a autant de lignes que de tâches destination.

2.6.2 Stockage des vecteurs d'ordonnancement

Les fichiers de vecteur d'ordonnancement utilisent la même numérotation des tâches que celle présente dans le fichier du DAG de tâches. Ils contiennent pour chaque *thread*, les tâches utilisateurs, de communication et de gestion du parallélisme à exécuter.

Listing 2.16 – Fichier des vecteurs d'ordonnancement.

```
#-------------------------------
thr 0 nbtsk 3
tsk 0
tsk 1
tsk 8
#-------------------------------
thr 1 nbtsk 3
tsk 2
tsk 3
tsk 4
[...]
```

2.6.3 Description du déploiement

Le déploiement de PRFX nécessite deux fichiers. Le premier fichier dépend de l'environnement système utilisé (e.g. fichier de *job* loadleveler (AIX), `lsf` ou fichier `poe` (AIX), `mpirun` avec `dplace` sous IRIX) qui va permettre de distribuer les processus sur la machine ou la grappe de machines. Nous ne détaillerons pas ces fichiers, la documentation associée est disponible chez les constructeurs ou dans les documentations MPI.

Le déploiement en termes de processus étant fait, le second fichier au format PRFX sert à paramétrer la répartition mémoire, les processus, les *threads* et leur placement sur les processeurs des différents nœuds comme indiqué dans le listing 2.17. Chaque processus est assignable à un numéro de machine. Ce numéro est le même que celui attribué par l'environnement système. Chaque processus se voit confier un pourcentage de l'espace d'iso-adressage conformément à notre choix fait à la section 2.3.1. Chaque processus peut contenir un nombre de *threads* différent. Chacun de ces *threads* est associé à un CPU physique ou laissé libre. Les *threads* sont répartis en deux catégories : ceux exécutant des opérations bloquantes et ceux exécutant des opérations non bloquantes.

Les opérations bloquantes pourront être potentiellement recouvertes grâce à un ordonnancement efficace des *threads* de la part du système d'exploitation. Ceci va à l'encontre du modèle d'exécution atomique des tâches, mais permet sur ces types de tâches au comportement imprévisible d'améliorer le rendement des processeurs. Comme les *threads* sont confinables dans un ensemble précis de CPU de la machine, le désordre occasionné n'interfère pas ou peu avec les calculs non bloquants car ils sont exécutés sur un pool de processeurs différents.

Listing 2.17 – déploiement sur deux nœuds IBM NH2 16-ways.

```
Machine 0 nb CPU 16 memory 16000000000
Machine 1 nb CPU 16 memory 16000000000

Process 0 on Machine 0 isoMem 60.0 %
        #-------------------------
MainThr           0 on CPU 0 stack -1
NonBlockingKThr   1 on CPU 1 stack 2000000
NonBlockingKThr   2 on CPU 2 stack 2000000
NonBlockingKThr   3 on CPU 3 stack 2000000
NonBlockingKThr   4 on CPU 4 stack 2000000
NonBlockingKThr   5 on CPU 5 stack 2000000
NonBlockingKThr   6 on CPU 6 stack 2000000
NonBlockingKThr   7 on CPU 7 stack 2000000
NonBlockingKThr   8 on CPU 8 stack 2000000
NonBlockingKThr   9 on CPU 9 stack 2000000
NonBlockingKThr  10 on CPU 10 stack 2000000
NonBlockingKThr  11 on CPU 11 stack 2000000
NonBlockingKThr  12 on CPU 12 stack 2000000
NonBlockingKThr  13 on CPU 13 stack 2000000
NonBlockingKThr  14 on CPU 14 stack 2000000
BlockingKThr     15 on CPU 15 stack 2000000
```

```
LAPIMainthr      16 on CPU 15 stack -1
LAPIAuxthr       17 on CPU 15 stack -1
PoeAuxthr        18 on CPU 15 stack -1

Process 1 on Machine 1 isoMem 40 %
        #-------------------------
MainThr          0 on CPU 0 stack -1
NonBlockingKThr  1 on CPU 1 stack 2000000
NonBlockingKThr  2 on CPU 2 stack 2000000
NonBlockingKThr  3 on CPU 3 stack 2000000
NonBlockingKThr  4 on CPU 4 stack 2000000
NonBlockingKThr  5 on CPU 5 stack 2000000
NonBlockingKThr  6 on CPU 6 stack 2000000
NonBlockingKThr  7 on CPU 7 stack 2000000
NonBlockingKThr  8 on CPU 8 stack 2000000
NonBlockingKThr  9 on CPU 9 stack 2000000
NonBlockingKThr  10 on CPU 10 stack 2000000
NonBlockingKThr  11 on CPU 11 stack 2000000
NonBlockingKThr  12 on CPU 12 stack 2000000
NonBlockingKThr  13 on CPU 13 stack 2000000
NonBlockingKThr  14 on CPU 14 stack 2000000
BlockingKThr     15 on CPU 15 stack 2000000
LAPIMainthr      16 on CPU 15 stack -1
LAPIAuxthr       17 on CPU 15 stack -1
PoeAuxthr        18 on CPU 15 stack -1
```

Les *threads* exécuteurs sont répartis selon la volonté du programmeur décrite dans le fichier de déploiement entre les *threads* non bloquants et les *threads* bloquants. Par exemple, les appels BLAS ou d'autres routines de calcul gagneront à être effectués par les *threads* non bloquants. En effet, ces routines ont un comportement stable même si des variations de performances sont possibles en fonction de l'alignement et de la taille de la donnée pour une même machine.

Les tâches de gestion des communications sont par contre plus difficilement modélisables car elles accèdent à une couche de communication soumise à des phénomènes asynchrones (disponibilité). Ceci est dû à un ensemble de contraintes relevant aussi bien du matériel que des couches logicielles réseau que l'on n'observe pas dans le cas des calculs processeur "purs". Des stratégies obscures sous-jacentes sont employées, des phénomènes de seuil et de saturation interviennent sur les réceptions / émissions. C'est pourquoi, pour avoir des performances, le programmeur doit paramétrer l'ordonnanceur afin d'affecter les tâches de gestion aux *threads* bloquants, avec la possibilité d'utiliser, par exemple, autant de *threads* que de cartes de communications exploitables. Cela permet en outre de conserver la continuité d'utilisation des caches pour les *threads* exécuteurs faisant uniquement du calcul.

2.7 Contrôle de l'exécution

Nous mettons tout en œuvre afin de rendre efficace le contrôle de l'exécution. La distribution du contrôle permet d'éviter des phénomènes de séquentialisation inhérents à la conservation d'un contrôle centralisé. Cette séquentialisation n'est pas visible sur des petites configurations d'exécution, mais apparaît de fait, dès que les temps de réponse du contrôleur deviennent critiques. C'est le cas sur de grandes configurations d'exécution où un contrôleur centralisé a des risques d'être souvent saturé. La distribution du contrôle est à la fois de la responsabilité du programmeur et de celle du support.

Listing 2.18 – distribution du contrôle.

```
1 //Contrôle centralisé                    // Contrôle distribué
2                                          void RPC_aux(int n, void(*func)(), void*arg1,void*arg2)
3                                          {
4                                            for ( j = 0; j < n; j++)
```

```
5                                                    PRFX_call(-1,0.0,func,&arg1,&arg2)
6                                                  }
7   [...]                                          [...]
8   int n = 10;                                    int n = 10;
9   for ( i = 0; i < n; i++)                       for (i =0; i < n; i++)
10  {                                              {
11    for ( j1 = 0; j1 < n; j1++)                    PRFX_call(-1,0.0,RPC_aux,n,RPC_func1,&arg1,&arg2);
12      PRFX_call(-1,0.0,RPC_func1,&arg1,&arg2);     PRFC_call(-1,0.0,RPC_aux,2*n,RPC_func,&arg1,&arg2);
13    for ( j2 = 0; j2 < 2*n; j2++)                }
14      PRFX_call(-1,0.0,RPC_func2,&arg1,&arg2);
15  }
```

En ce qui concerne le programmeur, lorsqu'il implémente des algorithmes directement à partir de leur spécification séquentielle, il a à tendance à conserver un contrôle de son algorithme (contrôle utilisateur) centralisé. Pour gagner en performances, il doit créer des tâches supplémentaires dans lesquelles il délègue ce contrôle. Dans l'exemple du listing 2.18, le code de gauche a un contrôle centralisé, alors que dans le code de droite, le contrôle est réparti entre deux tâches. Dès lors, dans le second cas, le lancement des RPC des deux boucles peut s'effectuer en parallèle. De plus, le code proposé ici exploite un moyen générique de distribution du contrôle en passant un pointeur de fonction au RPC auxiliaire `RPC_aux()`. La sémantique du code n'est pas changée, mais le contrôle n'est maintenant plus séquentiel. Cette étape de distribution du contrôle pourrait être automatisée par un compilateur ou par PRFX lui-même.

En ce qui concerne le support, nous proposons une gestion décentralisée des opérations de contrôle entre les tâches utilisateurs. Nous n'avons pas besoin de gérer de contrôle lié à la création ou à l'ajout de tâches dans la structure du DAG complet car il est connu statiquement. Le DAG de tâches complet décrivant toutes les opérations de contrôle liées aux tâches utilisateurs est répliqué dans tous les processus PRFX. Nous n'avons pas besoin d'en maintenir une version cohérente car nous ne résolvons pas de dépendances à la volée. Si tel était le cas, nous devrions modifier sa structure en conservant sa cohérence en distribué.

D'un point de vue modèle, nous utilisons les communications unilatérales (resp. les synchronisations en mémoire partagée) à l'extérieur (resp. l'intérieur) d'un processus PRFX. De fait toutes deux économisent la partie contrôle de l'interlocuteur.

De plus, chaque *thread* a son propre vecteur d'ordonnancement ce qui permet à l'ordonnanceur de distribuer le contrôle associé aux dépendances (tâches internes de gestion du parallélisme) sur plusieurs *threads* exécuteurs. Pour chacune des tâches qu'un *thread* exécuteur doit traiter, ce dernier attend et contrôle régulièrement le nombre de dépendances satisfaites. Il prépare le contexte de tâche et fait l'appel à la fonction cible de tâche avec les arguments. A la terminaison de la tâche, il libère les données internes temporairement allouées et passe à la tâche suivante.

Enfin, les modèles théoriques d'ordonnancement et les mesures pratiques faites pour la génération des traces permettraient d'évaluer la qualité de service en distribué. Chaque *thread* pourrait contrôler son état d'avancement connaissant les dates d'exécution théoriques des tâches. En mesurant la qualité de service à la volée, la décision de ré-ordonnancer peut être prise. Lors de ce ré-ordonnancement, les différences constatées sur les modèles de coûts permettent d'affiner ces derniers et de les réinjecter dans l'algorithme d'ordonnancement.

2.8 Exemples de mises en œuvre d'algorithmes parallèles avec PRFX

L'implémentation d'un algorithme est plus ou moins longue et complexe en fonction d'une part de la complexité intrinsèque de ce dernier et d'autre part de sa formulation. Il peut avoir déjà été implémenté sous une forme parallèle auquel cas sa formulation avec le mode d'expression PRFX est facilitée. Par contre, si la version de l'algorithme de départ est séquentielle, le programmeur doit tout d'abord fournir un effort algorithmique de découpage de son application ou de son problème.

2.8.1 Ecriture séquentielle à finalité parallèle

Le code privé des appels aux fonctionnalités de la bibliothèque PRFX est séquentiel, compilable et valide. Mais tout code séquentiel compilable et valide ne supporte pas obligatoirement l'ajout d'appels à la bibliothèque PRFX. Le code doit donc être écrit dans l'optique d'une exécution en parallèle en respectant les règles du mode d'expression PRFX. De plus, la parallélisation d'un code séquentiel existant ne peut être faite de façon incrémentale. Cette parallélisation est opérée en intervenant sur les données et les appels de fonctions du code séquentiel.

2.8.1.1 Identification des données parallèles

Une fois le découpage de l'application réalisée, la première chose à faire est de rechercher les données dont l'usage est localisé et de les discriminer des futures données parallèles qui sont traitées dans divers endroits du code. Si des appels aux fonctions d'allocation et de libération étaient utilisés pour les données parallèles alors, ils doivent être remplacés par ceux de la bibliothèque PRFX : `PRFX_isomalloc()` et `PRFX_isofree()`. Si ces futures données parallèles étaient stockées automatiquement dans la pile ou allouées statiquement, alors elles doivent être allouées avec `PRFX_isomalloc()` et libérées dès que possible avec `PRFX_isofree()`. Nous rappelons que la libération nécessite d'être propriétaire en écriture ; dans le cas contraire, l'appel de fonction de libération doit être transformé en appel de tâche (e.g. `PRFX_call(-1,0.0, PRFX_isofree, &ptr)`).

2.8.1.2 Identification des tâches

L'identification des tâches est un travail difficile précédant l'écriture de l'application. Il peut déjà avoir été fait du point de vue algorithmique pour une autre implémentation parallèle ou pour une implémentation séquentielle optimisée utilisant des tailles de données entrant dans les caches du processeur.

Une difficulté de programmation est liée au fait que les fonctions cibles de RPC ne peuvent pas retourner de valeur (prototype : `void taskFuncName()`). En effet, une valeur de retour supposerait une opération bloquante par la tâche initiatrice (ce qui est contraire au modèle). Pour traiter une valeur de retour d'un premier RPC, la tâche initiatrice doit par exemple lancer un deuxième RPC juste après le premier qui traite la valeur de retour. Cette valeur transite entre

les deux RPC, c'est donc une donnée parallèle. De fait, elle doit être iso-allouée puis passée par référence en paramètre de ces deux RPC.

2.8.1.3 Distribution des données parallèles

La distribution des données consiste classiquement en deux étapes : le découpage puis le placement des sous-données issues de ce découpage sur des processus (e.g. MPI) ou des processeurs virtuels (e.g. HPF). Avec PRFX, le placement des données est indirectement conditionné par celui des tâches. De plus, les données ne sont pas découpées mais délimitées par le programmeur avec des contours (*stencils*) se chevauchant potentiellement.

Le programmeur peut optionnellement conseiller la localisation du *thread* où seront exécutées les tâches. Dans le cas où les données sont disjointes et où chaque tâche écrit au plus une seule donnée, la distribution des tâches est aussi celle des données. Dans tous les autres cas (chevauchement ou plusieurs données en écriture pour une même tâche), l'utilisateur peut décider d'une destination selon ses critères ou laisser faire l'algorithme de placement / ordonnancement. Dans ce dernier cas, la distribution des données varie en cours d'exécution selon les choix faits par l'ordonnanceur.

2.8.2 Exemples d'utilisation du mode d'expression PRFX

Les exemples qui vont suivre sont des cas réels de noyaux de calcul permettant la résolution de problèmes scientifiques. Ils ont été choisis car ils sont bien connus et donc implémentés dans beaucoup de modes d'expression. Ainsi, nous allons pouvoir mettre en valeur la facilité d'expression de PRFX et apporter des points de comparaison avec les modes d'expression présentés précédemment. Le premier algorithme abordé est celui de Jacobi ; sa version parallèle est facile à implémenter, notamment avec PRFX, car les accès de voisinage s'expriment comme en séquentiel. Le second algorithme présenté factorise les matrices pleines symétriques définies positives avec la méthode de Cholesky[1]. La matrice est découpée en blocs carrés de taille uniforme.

2.8.2.1 Algorithme de Jacobi (maillage du plan)

L'algorithme de Jacobi permet de résoudre de façon itérative le problème de Laplace. D'autres algorithmes itératifs existent aussi : SOR, Gauss-Seidel, Multigrid mais ils sont moins faciles à paralléliser. Une version directe (Gauss) existe également mais présente un problème de précision dû à des pivots numériquement faibles dès que la taille du maillage est grande.

La version continue de ce problème se pose dans de nombreux phénomènes physiques (chaleur, électrostatique, mécanique des fluides, ...).

[1] 15 oct. 1875- 31 août 1918. Né à Montguyon en Charente-Inférieure (dénomination de l'époque). Commandant, directeur technique du service géographique de l'armée française. "Officier travailleur, ingénieux, réfléchi, [...], devra toutefois se méfier comme chef de service de quelque tendance à l'originalité et au paradoxe ..." (Lt Col. Hergault, chef d'Etat-Major de la 7ème Armée, 24 oct. 1916). Cf. C. Brezinski : André Louis Cholesky, rapport ANO 347, U.S.T.Lille I, 1995.

Ces phénomènes physiques sont régis par l'équation de Laplace :

$$\Delta u = 0 \qquad \text{avec } \Delta \text{ l'opérateur laplacien.}$$

Dans le cas bi-dimensionnel, u est une fonction scalaire des variables d'espace x et y et

$$\Delta u = \frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2}.$$

Cependant, l'équation de Laplace n'a pas toujours de solution analytique simple, d'où la nécessité de discrétiser le problème. Nous ne nous intéresserons qu'aux maillages cartésiens du plan. Les conditions aux limites du maillage sont choisies constantes. Dans ce cas, l'équation de Laplace se réduit au système d'équations suivant :

$$\begin{aligned} u_{x,y} &= Cte && \text{si } (x,y) \in \text{au bord du maillage} \\ u_{x-1,y} + u_{x+1,y} + u_{x,y-1} + u_{x,y+1} - u_{x,y} &= 0 && \text{sinon} \end{aligned}$$

Le résultat est approximé en arrêtant le calcul lorsque :

$$\max_{x,y \in \text{maillage}} \left\{ \left| \frac{u_{x-1,y} + u_{x+1,y} + u_{x,y-1} + u_{x,y+1}}{4} - u_{x,y} \right| \right\} \leqslant \varepsilon$$

Listing 2.19 – Implémentation séquentielle de l'algorithme de Jacobi sur un maillage du plan - version par lignes.

```
1  #include <sys/types.h>
2  #include <stdio.h>
3  typedef double elem_t;
4  void fillMeshValue(ulong xdim, ulong ydim, ulong base_x, ulong base_y, elem_t mesh[ydim][xdim])
5  {
6    ulong x,y;
7      for ( y = 0; y < ydim; y++)
8        for ( x = 0; x < xdim; x++)
9          mesh[y][x] = getMeshVal(base_x+x, base_y+y);
10 }
11
12 void storeMeshToFile(char*filename, ulong xdim, ulong ydim, elem_t mesh[ydim][xdim], float delta_max)
13 {
14   ulong x,y;
15   FILE*fd = fopen(filename,"+w");
16
17   fprintf(fd,"%lu %lu %f",xdim,ydim,delta_max);
18   for ( y = 0; y < ydim; y++)
19     for ( x = 0; x < xdim; x++)
20       fprintf(fd,"%g",mesh[y][x]); //[...]
21
22   fclose(fd);
23 }
24 void computeDeltaMax(float*delta_max_ptr, ulong xdim, ulong ydim, elem_t dest_mesh[ydim][xdim],elem_t src_mesh[ydim][xdim])
25 {
26   ulong x,y;
27   elem_t abs;
28   float max = 0.0;
29
30   for ( y = 0; y < ydim; y++)
31     for ( x = 0; x < xdim; x++)
32       {
33         abs = fabs(dest_mesh[y][x] - src_mesh[y][x]);
34         if (abs > max)
35           max = abs;
36       }
37   *delta_max_ptr = max;
38 }
39
40
41 void averageMesh(ulong xdim, ulong ydim, elem_t dest_mesh[ydim][xdim],elem_t src_mesh[ydim][xdim])
42 {
```

```
43    ulong x,y;
44
45    for ( y = 1; y < ydim-1; y++)
46      for ( x = 1; x < xdim-1; x++)
47        {
48          dest_mesh[y][x] = (src_mesh[y-1][x] + src_mesh[y+1][x] + src_mesh[y][x+1] + src_mesh[y][x-1])/ 4;
49        }
50  }
51
52  void solveDiscreteLaplace(ulong mesh_xdim, ulong mesh_ydim, elem_t mesh[mesh_ydim][mesh_xdim],
53                            float* delta_max_ptr, ulong nb_loop)
54  {
55    ulong i;
56    elem_t max;
57    ulong mesh_sz = mesh_xdim * mesh_ydim * sizeof(elem_t);
58    elem_t (*aux_mesh)[mesh_xdim] = malloc (mesh_sz);
59
60    for (i = 0; i < nb_loop; i++)
61      {
62        averageMesh(mesh_xdim, mesh_ydim, aux_mesh, mesh);
63        memcopy(mesh, aux_mesh, mesh_sz);
64      }
65    computeDeltaMax(delta_max_ptr,mesh_xdim, mesh_ydim, aux_mesh, mesh);
66    free(aux_mesh);
67  }
68
69  int main(int argc, char*argv[])
70  {
71    ulong mesh_xdim    = atol(argv[1]);
72    ulong mesh_ydim    = atol(argv[2]);
73    ulong nb_loop      = atol(argv[3]);
74    char *filename     =      argv[4];
75    float delta_max;
76    ulong mesh_sz = mesh_xdim * mesh_ydim * sizeof(elem_t);
77    elem_t (*mesh)[mesh_xdim] = malloc (mesh_sz);
78
79    fillMeshValue(mesh_xdim, mesh_ydim, 0, 0, mesh);
80    solveDiscreteLaplace(mesh_xdim, mesh_ydim,mesh, &delta_max,nb_loop);
81    storeMeshToFile(filename, mesh_xdim, mesh_ydim, mesh, delta_max);
82    free(mesh);
83  }
```

L'algorithme de Jacobi nécessite deux tableaux (un principal et un auxiliaire) pour fonctionner. Ce doublement de la consommation mémoire est intrinsèque à cet algorithme et n'est donc pas lié à l'usage de PRFX.

Le tableau principal de l'étape $i+1$ est obtenu à partir du tableau principal de l'étape i via un tableau auxiliaire (lignes 48 et 49). Le calcul s'arrête au bout de n itérations et la variation Δ des valeurs du tableau principal entre la $n-1$ ième et la n ième itération est calculée à titre informatif. Le calcul de la nouvelle valeur d'un nœud du maillage utilise les quatre valeurs voisines. Cette valeur est écrite dans le tableau auxiliaire et une fois ce dernier rempli, il est recopié dans le tableau principal. Le fait de fixer le nombre d'étapes avec une donnée de niveau inspection (ligne 66) est une modification de l'algorithme par rapport à sa version classique où le calcul s'arrête dès que le niveau de convergence souhaitée est atteint. Cette modification est utile pour l'implémentation avec PRFX, elle permet à l'inspecteur de générer le DAG de tâches complet. Cette modification n'a pas besoin d'être faite avec un langage comme HPF (cf. annexe A.2.1), ni avec le langage OpenMP. Dans le cas d'OpenMP, le programme peut être reformulé dans une version SPMD (cf. annexe A.1.1) pour augmenter les performances. Au-delà de cette modification de l'algorithme, nous avons surtout choisi cet exemple pour montrer qu'un code séquentiel accédant à des données s'intersectant peut être réutilisé et parallélisé avec PRFX en faisant quelques modifications. Ces intersections sont difficilement exprimables avec des modes d'expression comme RAPID ou ATHAPASCAN car ils n'offrent pas une continuité des accès à leurs données parallèles aussi vaste que celle de PRFX.

Pour montrer la démarche de parallélisation avec PRFX, nous partons du code séquentiel

2.19. Ce code contient toutes les fonctions utiles à l'algorithme de Jacobi :

- `fillMeshValue()` qui initialise le tableau principal représentant le maillage ;
- `solveDiscreteLaplace()` qui procède aux itérations (étapes) de calcul sur le maillage ;
- `averageMesh()` qui calcule les valeurs du maillage auxiliaire ;
- `computeDeltaMax()` qui calcule la variation maximale Δ entre les valeurs des éléments des deux maillages passés en paramètre ;
- `storeMeshTofile()` qui écrit le maillage dans un fichier.

Les versions, séquentielle (code 2.20) et parallèle (code 2.21) par blocs de lignes, réutilisent les fonctions du code séquentiel 2.19. Ceci est possible car le stockage du maillage est par lignes, par conséquent un bloc de lignes peut être vu comme un maillage. Pour ces codes 2.20 et 2.21, nous avons besoin d'une nouvelle fonction `getTabMax()` afin de déduire la variation maximale sur le maillage à partir des variations locales de chaque bloc de ce dernier.

Le découpage en N blocs est une optimisation relativement facile à mettre en place. Elle apporte un faible effet de cache en séquentiel car le nombre d'opérations est linéaire par rapport au nombre d'accès. Nous écrivons néanmoins ce code en séquentiel suivant cette optique car il est une étape intermédiaire utile à l'obtention d'un code parallèle utilisant PRFX. Dans les deux versions, parallèle et séquentielle par bloc, nous utilisons des pointeurs pour accéder aux différents blocs de lignes comme présenté dans la figure 7. Le maillage de gauche est le maillage auxiliaire, il est atteint par les pointeurs `mesh_blk[i]` référençant le ième bloc avec un décalage d'une ligne par rapport au maillage principal (rappel : la première ligne correspond au bord qui est constant). Le maillage principal est à droite sur la figure, ses blocs sont accédés via les pointeurs `mesh_blk_plus[i]` qui pointent sur le bloc augmenté de 2 lignes (une avant et une après le bloc).

Listing 2.20 – Implémentation séquentielle de l'algorithme de Jacobi sur un maillage du plan - version par blocs de lignes.

```
1  void fillMeshValue(ulong xdim, ulong ydim, ulong base_x, ulong base_y, elem_t mesh[ydim][xdim]);
2
3  void storeMeshToFile(char*filename, ulong xdim, ulong ydim, elem_t mesh[ydim][xdim], float delta_max);
4
5  void averageMesh(ulong xdim, ulong ydim, elem_t dest_mesh[ydim][xdim],elem_t src_mesh[ydim][xdim]);
6  void computeDeltaMax(float*delta_max, ulong xdim, ulong ydim, elem_t dest_mesh[ydim][xdim],elem_t src_mesh[ydim][xdim]);
7  void getTabMax(float*res, ulong n, float*tab)
8  {
9    ulong i;
10   float max = tab[0];
11   for( i = 1; i < n ; i++)
12     {
13       if (tab[i] > max)
14         max = tab[i];
15     }
16   *res = max;
17 }
18
19 void solveDiscreteLaplaceBlk1D(ulong xdim, ulong ydim, ulong nb_blk, elem_t mesh_blk_plus[nb_blk][ydim][xdim],
20                                float *delta_max_ptr, ulong nb_loop){
21   ulong i,j;
22   ulong blk_sz = xdim * ydim * sizeof(elem_t)
23   ulong mesh_sz = nb_blk * blk_sz;
24   elem_t (*aux_mesh_blk_plus)[ydim][xdim];
25   elem_t *tab_max;
26   elem_t *blk_src;
27   elem_t *blk_dest;
28   elem_t *blk_src_plus;
29   elem_t (*aux_mesh_blk)[ydim][xdim];
30   elem_t (*mesh_blk)[ydim][xdim];
```

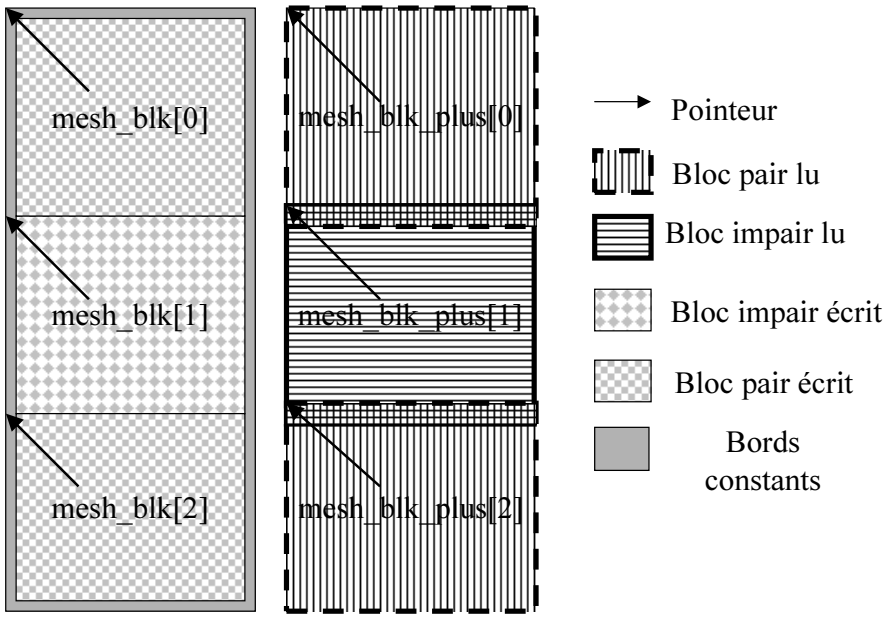

FIG. 7 – Affectation des pointeurs utilisés dans l'implémentation de l'algorithme de Jacobi pour un maillage découpé en trois blocs de lignes.

```
31
32      aux_mesh_blk_plus = malloc (mesh_sz + 2*line_sz);
33      aux_mesh_blk = &aux_mesh_blk_plus[0][1][0];
34      mesh_blk = &mesh_blk_plus[0][1][0];
35
36      for ( i = 0; i <  nb_loop; i++)
37        {
38          for ( j = 0; j < nb_blk; j++)
39            {
40              averageMesh(xdim, ydim, aux_mesh_blk[j], mesh_blk_plus[j]);
41              memcpy(mesh_blk[j], aux_mesh_blk[j], blk_sz);
42            }
43        }
44      tab_max = malloc(nb_blk * sizeof(elem_t));
45      for ( j = 0; j < nb_blk; j++)
46        {
47          computeDeltaMax(&tab_max[j], xdim, ydim, aux_mesh_blk[j], mesh_blk[j]);
48        }
49      getTabMax(delta_max_ptr, nb_blk, tab_max);
50      free(aux_mesh_blk);
51      free(tab_max);
52    }
53
54    int main(int argc, char*argv[])
55    {
56      ulong xdim = atol(argv[1]); // width of one mesh
57      ulong ydim = atol(argv[2]); // height of one blk
58      ulong nb_blk = atol(argv[3]);
59      char *filename = argv[4];
60      uint32_t nb_loop = atoi(argv[5]);
61      float delta_max;
62      ulong line_sz = xdim * sizeof(elem_t);
63      ulong blk_sz = ydim * line_sz;
64      ulong mesh_sz = nb_blk *  xdim * ydim * sizeof(elem_t);
65      elem_t (*mesh_blk_plus)[ydim][xdim];
66      elem_t (*mesh_blk)[ydim][xdim];
67
68      mesh_blk_plus = malloc (mesh_sz+2*line_sz);
69      mesh_blk = &mesh_blk_plus[0][1][0];
70
71
72      fillMeshValue(xdim, 1, 0, 0, mesh_blk_plus[0]);
73      fillMeshValue(xdim, 1, 0, ydim*(nb_blk), mesh_blk[nb_blk]);
74
```

```
75    for ( i = 0; i < nb_blk; i++)
76      fillMeshValue(xdim, ydim,0,ydim*i+1, mesh_blk[i]);
77
78    solveDiscreteLaplaceBlk1D(xdim, ydim, nb_blk, mesh_blk_plus, &delta_max, nb_loop);
79    storeMeshToFile(filename, xdim,ydim*nb_blk+2, mesh_blk_plus, delta_max);
80    free(mesh_blk_plus);
81  }
```

Pour obtenir le code PRFX 2.21, nous devons modifier la version séquentielle par blocs (code 2.20). Nous allons tout d'abord choisir les appels de fonctions générant potentiellement du parallélisme et les modifier pour que l'inspecteur PRFX puisse analyser leurs paramètres. Ces fonctions ne doivent utiliser que du code ou des appels à d'autres bibliothèques compatibles avec la bibliothèque PRFX. Ici, tous les corps de fonctions de la version séquentielle sont conformes aux contraintes fixées par le mode d'expression PRFX.

Nous choisissons de faire exploiter par PRFX le parallélisme présent lors de l'initialisation des blocs, lors des calculs sur les blocs et lors des copies de blocs auxiliaires vers les blocs du maillage principal. Les fonctions concernées de la version séquentielle (code 2.20) sont donc `fillMeshValue()`, `averageMesh()` et `memcpy()`. Les appels vers ces fonctions sont donc transformés en appels de tâches (`PRFX_call()`).

En regardant le corps de la fonction `solveDiscreteLaplaceBlk1D()` du code séquentiel, on constate que les fonctions `computeDeltaMax()`, `getTabMax()` et `free()` utilisent des données passées en paramètres des tâches précédemment identifiées. Pour conserver un fonctionnement valide (lié à l'affaiblissement du degré de propriété), ces appels de fonctions doivent également être transformés en appels de tâches PRFX.

Dès que sont identifiés tous les appels de fonctions devant être transformés en appels de tâches, il faut déclarer les modes d'accès des fonctions cibles. Ces déclarations doivent intervenir avant l'initialisation de PRFX, elles sont donc regroupées au début de la fonction `main()`.

Listing 2.21 – Implémentation parallèle avec PRFX par blocs de lignes de l'algorithme de Jacobi sur un maillage du plan.

```
1   #include "prfx.h"
2   void fillMeshValue(ulong xdim, ulong ydim, ulong base_x, ulong base_y, elem_t mesh[ydim][xdim]);
3
4   void storeMeshToFile(char*filename, ulong xdim, ulong ydim, elem_t mesh[ydim][xdim], float delta_max);
5
6   void averageMesh(ulong xdim, ulong ydim, elem_t dest_mesh[ydim][xdim],elem_t src_mesh[ydim][xdim]);
7   void computeDeltaMax(float*delta_max, ulong xdim, ulong ydim, elem_t dest_mesh[ydim][xdim],elem_t src_mesh[ydim][xdim]);
8   void getTabMax(float*res, ulong n, float*tab);
9
10
11  void solveDiscreteLaplaceBlk1D(ulong xdim, ulong ydim, ulong nb_blk, elem_t mesh_blk_plus[nb_blk][ydim][xdim],
12                                 elem_t **delta_max_ptr_ptr, ulong nb_loop)
13  {
14    ulong i,j;
15    ulong line_sz = xdim * sizeof(elem_t);
16    ulong blk_sz = ydim * line_sz;
17    ulong mesh_sz = nb_blk * blk_sz;
18    elem_t (*aux_mesh_blk_plus)[ydim][xdim];
19    elem_t *tab_max;
20    elem_t *blk_src;
21    elem_t *blk_dest;
22    elem_t *blk_src_plus;
23    elem_t (*aux_mesh_blk)[ydim][xdim];
24    elem_t (*mesh_blk)[ydim][xdim];
25
26    PRFX_isomalloc (&aux_mesh_blk_plus, mesh_sz + 2*line_sz);
27
28    aux_mesh_blk = &aux_mesh_blk_plus[0][1][0];
29    mesh_blk = &mesh_blk_plus[0][1][0];
30
31    PRFX_stencilDecl1D(&blk_src, blk_sz);
```

```
32    PRFX_stencilDecl1D(&blk_dest, blk_sz);
33    PRFX_stencilDecl1D(&blk_src_plus, blk_sz + 2 * line_sz);
34
35    for (i = 0; i < nb_loop; i++)
36      {
37        for ( j = 0; j < nb_blk; j++)
38          {
39            blk_dest = aux_mesh_blk[j];
40            blk_src_plus =  mesh_blk_plus[j];
41            max_ptr = &tab_max[j];
42            PRFX_call(i,1.0, averageMesh, &stop, xdim, ydim, &blk_dest, &blk_src_plus);
43
44            blk_src = aux_mesh_blk[j];
45            blk_dest = mesh_blk[j];
46            PRFX_call(i,1.0, memcpy, &blk_dest, &blk_src, blk_sz);
47          }
48      }
49
50    PRFX_isomalloc (&tab_max, nb_blk * sizeof(elem_t));
51    PRFX_stencilDecl1D(&max_ptr,sizeof(elem_t));
52
53    for ( j = 0; j < nb_blk; j++)
54      {
55        blk_dest = aux_mesh_blk[j];
56        blk_src = mesh_blk[j];
57        max_ptr = &tab_max[j];
58        PRFX_call(j,1.0,computeDeltaMax, &max_ptr, xdim, ydim, &blk_dest, &blk_src);
59      }
60    PRFX_call(0,1.0,getTabMax, delta_max_ptr_ptr, nb_blk, &tab_max);
61
62    PRFX_call(0,2.0,PRFX_isofree,&aux_mesh_blk_plus);
63    PRFX_call(0,2.0,PRFX_isofree,&tab_max);
64  }
65
66
67
68  int main(int argc, char*argv[])
69  {
70    ulong xdim = atol(argv[1]); // width of one blk
71    ulong ydim = atol(argv[2]); // height of one blk
72    ulong nb_blk = atol(argv[3]);
73    uint32_t nb_loop = atoi(argv[5]);
74    float*delta_max_ptr;
75    char *filename;
76    ulong line_sz = xdim * sizeof(elem_t);
77    ulong blk_sz = ydim * line_sz;
78    ulong mesh_sz = nb_blk *  blk_sz;
79    elem_t (*mesh_blk_plus)[ydim][xdim];
80    elem_t (*mesh_blk)[ydim][xdim];
81
82
83    PRFX_declRPC5(fillMeshValue,1,IMM, IMM, IMM, IMM, FW);
84    PRFX_declRPC5(averageMesh,1,IMM,IMM,FW,RO,FW);
85    PRFX_declRPC3(memcpy,1,FW,RO,IMM);
86    PRFX_declRPC5(computeDeltaMax,1,FW,IMM,IMM,RO,RO);
87    PRFX_declRPC3(getTabMax,1,FW,IMM,RO);
88    PRFX_declRPC5(storeMeshToFile,1, RO, IMM, IMM, RO, RO);
89    PRFX_declRPC1(PRFX_isofree,0,FW);
90
91    PRFX_init();
92
93    PRFX_thrDistributionSetDefault(PRFX_DISTRIB_BLOCK1D, nb_blk,-1,-1,-1,-1);
94
95    PRFX_isomalloc (&mesh_blk_plus, mesh_sz + 2*line_sz);
96    mesh_blk = &mesh_blk_plus[0][1][0];
97
98    PRFX_stencilDecl1D(&line_dest, line_sz);
99    line_dest = mesh_blk_plus[0];
100   PRFX_call(0,0.0,fillMeshValue, xdim, 1, 0, 0, &line_dest);
101   line_dest = mesh_blk[nb_blk];
102   PRFX_call(nb_blk-1,0.0,fillMeshValue, xdim, 1, 0, ydim*(nb_blk), &line_dest);
103
104   PRFX_stencilDecl1D(&blk_dest, blk_sz);
105   for ( i = 0; i < nb_blk; i++)
106     {
107       blk_dest = mesh_blk[i];
108       PRFX_call(i,0.0,fillMeshValue xdim, ydim,0,ydim*i+1, &blk_dest);
109     }
110
111   PRFX_isomalloc (&delta_max_ptr, sizeof(float));
112
113   solveDiscreteLaplaceBlk1D(xdim, ydim, nb_blk, mesh_blk_plus, &delta_max_ptr, nb_loop);
114
115   PRFX_isomalloc (&filename, strlen(argv(4));
116   strncpy(filename, argv[4], strlen(argv(4)+1);
117   PRFX_call(0,2.0, storeMeshToFile, &filename, xdim, ydim*nb_blk+2, mesh_blk_plus, &delta_max_ptr);
118
119   PRFX_call(0,2.0, PRFX_isofree, &mesh_blk_plus);
```

```
120
121     PRFX_terminate();
122 }
```

Nous rappelons que les données utilisées en paramètres des RPC doivent être allouées avec l'allocateur en iso-mémoire. Les données scalaires passées en paramètres de RPC doivent, quant-à-elles, être transtypées en `(void*)`. Le tableau stockant les blocs du maillage auxiliaire et le tableau contenant les Δ_i maxima locaux sont tous deux iso-alloués car ils sont passés en paramètres des tâches. Les sous-données du maillage (ici blocs et blocs augmentés) vont pouvoir être identifiées avec les trois *stencils* de la ligne 31 à la ligne 33. Le *stencil* suivant de la ligne 51 sert à identifier un élément du tableau des Δ_i.

A la fin de la fonction `solveDiscreteLaplaceBlk1D()`, des tâches sont créées (fonction cible `PRFX_isofree()` pour libérer le maillage auxiliaire et le tableau des Δ_i locaux. Ces libérations ne peuvent être faites directement dans `solveDiscreteLaplaceBlk1D()` car la tâche associée n'est plus propriétaire en écriture des données parallèles à libérer.

Intéressons-nous à présent au `main()` dans lequel la tâche initiale est délimitée par les deux appels `PRFX_init()` et `PRFX_terminate()`. A la ligne 94, nous choisissons une distribution par blocs des RPC sur les *threads* exécuteurs. Cette distribution étant faite dans la tâche initiale, toutes les tâches filles en hériteront. Cette distribution des RPC va pouvoir correspondre à la distribution des blocs du maillage. Pour ce faire, nous numérotons les RPC avec le numéro du bloc que chacun d'eux traite en écriture.

Les *stencils* utilisés par l'initialisation des blocs sont ensuite déclarés aux lignes 98 et 104. Ici, cette déclaration est concise car seuls deux motifs géométriques simples interviennent : celui des premières et dernières lignes, et celui des blocs internes du maillage. Les appels de fonctions d'initialisation des blocs `fillMeshValue()` sont transformés en créations de tâche (lignes 100, 102 et 108). L'appel à `solveDiscreteLaplaceBlk1D()` est conservé tel quel pour montrer que tous les appels de fonction n'ont pas obligation d'être transformés en tâches. Cela aurait également pu être fait, auquel cas nous aurions dû déclarer la fonction cible de cet appel avec des accès en virtuel (VR) sur le maillage principal. Ensuite, le nom du fichier doit être recopié dans une donnée parallèle pour pouvoir être transmis à la tâche exécutant la fonction `storeMeshToFile()`.

L'écriture du fichier résultat n'est pas parallélisée car cela compliquerait le code. Nous aurions pu le paralléliser en adoptant un format de fichier binaire où des tâches (de façon analogue au remplissage du maillage) écrivent chacune un bloc de lignes du maillage à un endroit précis dans le fichier. L'ouverture, l'écriture et la fermeture du fichier doivent être faites par chacune de ces tâches. En ce qui concerne le code présenté, et pour garder un fonctionnement valide, l'appel à la fonction `storeMeshToFile()` ligne 117 qui écrit séquentiellement tous les blocs du maillage, est transformé en appel de tâche. En effet, à cet endroit du code, la tâche `main()` n'est plus que propriétaire virtuelle (VR) du maillage principal. Pour la même raison, la libération du maillage principal est effectuée par une tâche autre que la tâche `main()`.

L'ordonnancement du DAG de tâches de l'algorithme de Jacobi est de faible complexité ($O(Nlog(N))$) car le nombre de tâches est linéaire par rapport au nombre de blocs ($O(N)$) et le

nombre de précédences pour une tâche est une constante valant ici 5. La complexité de l'algorithme de Jacobi est de l'ordre de $O(Nhw)$ avec h et w respectivement la hauteur et la largeur d'un bloc du maillage.

Si nous disposions d'un support dynamique, nous pourrions implémenter une version respectant l'algorithme original de Jacobi qui s'arrête dès que la convergence est suffisante (cf. section 5.3). Cette version pourrait par exemple être récursive avec une récursion portant sur la tâche `solveDiscreteLaplaceBlk1D()`. Le corps et l'en-tête de cette fonction doivent être changés pour être appelables récursivement. En particulier, le maillage auxiliaire doit être passé en paramètre ainsi que la valeur `epsilon`. Cette valeur est utilisée par la fonction `isBelowEpsilon()` pour arrêter la récursion lorsque les Δ_i sont tous inférieurs à `epsilon`.

Listing 2.22 – Ecriture récursive avec PRFX pour l'algorithme de Jacobi sur un maillage du plan.

```
1  int isBelowEpsilon(ulong nb_val, elem_t tab[nb_val], elem_t epsilon){
2    ulong i;
3
4    for ( i = 0; i < nb_blk; i++)
5      if ( tab[i] > epsilon)
6        break;
7
8    return i == nb_blk;
9  }
10
11 void solveDiscreteLaplaceBlk1D(ulong xdim, ulong ydim, ulong nb_blk, elem_t mesh_blk_plus[nb_blk][ydim][xdim],
12                                 void* epsilon_, ulong nb_loop_max, elem_t*tab_max,
13                                 elem_t aux_mesh_blk_plus[nb_blk][ydim][xdim])
14 {
15   ulong i,j;
16   ulong line_sz = xdim * sizeof(elem_t)
17   ulong blk_sz =  ydim * line_sz;
18   elem_t *blk_src;
19   elem_t *blk_dest;
20   elem_t *blk_src_plus;
21   elem_t (*aux_mesh_blk)[ydim][xdim];
22   elem_t (*mesh_blk)[ydim][xdim];
23   float epsilon = *(float*)&epsilon_;
24   aux_mesh_blk = &aux_mesh_blk_plus[0][1][0];
25   mesh_blk = &mesh_blk_plus[0][1][0];
26
27   PRFX_stencilDecl1D(&blk_src, blk_sz);
28   PRFX_stencilDecl1D(&blk_dest, blk_sz);
29   PRFX_stencilDecl1D(&blk_src_plus, blk_sz + 2 * line_sz);
30   PRFX_stencilDecl1D(&max_ptr,sizeof(elem_t));
31
32   if ( isBelowEpsilon(nb_blk, tab_max, epsilon))
33     {
34       return;
35     }
36
37   for ( j = 0; j < nb_blk; j++)
38     {
39       blk_dest = aux_mesh_blk[j];
40       blk_src_plus =  mesh_blk_plus[j];
41       max_ptr = &tab_max[j];
42       PRFX_call(i,1.0, averageMesh, xdim, ydim, &blk_dest, &blk_src_plus , &max_ptr);
43
44       blk_dest = mesh_blk[j];
45       PRFX_call(i,1.0, memcpy, &blk_dest, &blk_src, blk_sz);
46     }
47
48   if ( nb_loop_max-- > 0 )
49     {
50       PRFX_call(0,1.0,solveDiscreteLaplaceBlk1D,xdim, ydim, nb_blk, &mesh_blk_plus,epsilon_, nb_loop_max,
51                 tab_max, aux_mesh_blk_plus);
52     }
53 }
```

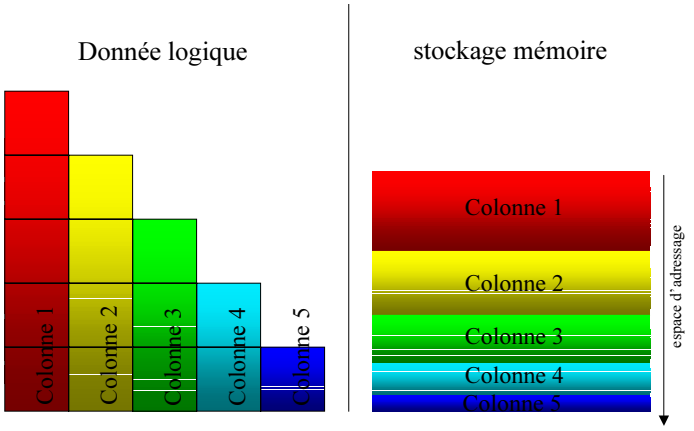

FIG. 8 – Vision logique de la matrice pleine et son stockage en mémoire.

2.8.2.2 Factorisation de matrices pleines par la méthode de Cholesky

La factorisation de Cholesky est utilisée pour la résolution de problèmes dans le domaine de l'électromagnétisme. La matrice à factoriser A, de taille $n \times n$, est supposée symétrique et définie positive. Le stockage en mémoire peut tirer parti de cette propriété en ne stockant par exemple que la matrice triangulaire inférieure (cf. figure 8). Le stockage est ici par colonne de blocs où chaque bloc de coefficients est stocké par lignes.

Le nombre d'opérations pour une matrice pleine de dimension $n \times n$ est de l'ordre de $n^3/3 + n^2 + 5/3n$.

Algorithme 1 Algorithme de factorisation de matrices denses par la méthode de Cholesky.

Entrée Une matrice A de $N \times N$ blocs symétrique définie positive

Sortie La matrice A dont la partie triangulaire inférieure modifiée : L vérifie $A = LL^t$

1: **pour** k de 1 à N **faire**
2: factoriser A_{kk} en $L_{kk}U_{kk}$, L_{kk} et U_{kk} écrasent A_{kk}
3: **pour** i de $k+1$ à N **faire**
4: résoudre $L_{kk}L_{ik}^t = A_{ik}^t$ où L_{ik} écrase A_{ik}
5: **fin pour**
6: **pour** i de $k+1$ à N **faire**
7: **pour** j de $k+1$ à i **faire**
8: $A_{ij} \leftarrow A_{ij} - L_{ik}L_{jk}^t$
9: **fin pour**
10: **fin pour**
11: **fin pour**

L'algorithme de Cholesky opère sur la matrice en place ; à la fin de son déroulement, la partie inférieure de la matrice initiale A contient le facteur L tel que $A = LL^t$. Dans sa version par blocs carrés, cet algorithme se déroule en autant d'étapes qu'il y a de blocs sur la diagonale de la

matrice. A chaque étape k, l'algorithme opère sur la sous-matrice de blocs entre k et N (nombre de blocs diagonaux). Le volume de données traitées à chaque étape k est donc de $(N-k+1)(N-k+2)/2$ blocs. Lors d'une étape, le bloc diagonal `Akk` est factorisé, ensuite les systèmes correspondant aux blocs de la même colonne que `Akk` sont résolus et enfin les contributions modifiant toute la sous-matrice à droite sont calculées. Le découpage par blocs améliore les effets de cache en séquentiel car chaque calcul est réalisé par un opérateur BLAS3 qui effectue $O(n^3)$ opérations pour $O(n^2)$ mouvements de données. Nous écrivons donc la version optimisée séquentielle par blocs (code 2.23) ce qui demande un peu de savoir-faire et de connaissances de la bibliothèque BLAS ([8]). Nous avons nommé dans ce code séquentiel les fonctions en les préfixant avec `RPC_`. Ceci nous permet de raccourcir le code PRFX (code 2.24) en omettant les parties qui restent identiques à celles du code 2.23 après sa ré-écriture pour PRFX. Le nommage des fonctions de tâches sous PRFX est tout aussi libre qu'en séquentiel : le préfixe `RPC_` est purement "décoratif", il n'est ni traité ni compris par le support PRFX.

Listing 2.23 – Implémentation séquentielle de la version bloc l'algorithme de factorisation de Cholesky de matrices.

```
1  #include <essl.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sting.h>
5  //type for matrice NxN with block storage
6  typedef double elem_t;
7
8  // Generic call to ESSL API:
9  // [d|s]## essl_func_suffix // for [double|float] BLAS computations
10 #define G(essl_func_suffix) d ## essl_func_suffix
11
12 void RPC_fillBlk2D(uint32_t blk_dim, elem_t A[blk_dim][blk_dim],uint32_t blk_l,uint32_t blk_c)
13 {
14   uint32_t l,c;
15   int matrix_c, matrix_l; //abs coordinates in whole matrix in number of elem
16   for ( c = 0; c < blk_dim; c++)
17     for ( l = 0; l < blk_dim; l++)
18       {
19         matrix_l = (l+blk_l*blk_dim);
20         matrix_c = (c+blk_c*blk_dim);
21
22         A[l][c] = my_mat_get_val(NULL,matrix_l, matrix_c);
23       }
24 }
25
26 void RPC_fillSymmMatrix(uint32_t blk_dim, uint32_t nb_w_blk, elem_t A[nb_w_blk][nb_w_blk][blk_dim][blk_dim])
27 {
28   uint32_t blk_l, blk_c;
29   ulong blk_sz =  blk_dim *  blk_dim *sizeof(elem_t);
30   elem_t (*Alc)[blk_dim];
31
32   for (blk_c = 0; blk_c < nb_w_blk; blk_c++)
33     for(blk_l = blk_c; blk_l < nb_w_blk; blk_l++)
34       {
35         int place = blk_l*nb_w_blk+blk_c;
36
37         Alc = A[blk_l][blk_c];
38
39         RPC_fillBlk2D(blk_dim, Alc, blk_l, blk_c);
40       }
41 }
42
43
44 void RPC_CholeskyBlk2D(uint32_t blk_dim, elem_t A[blk_dim][blk_dim])
45 {
46   int info;  char A_uplo = 'L';
47
48  G(potrf)(&A_uplo, blk_dim, A, blk_dim, &info);
49 }
50
51 void RPC_trsm(uint32_t blk_dim,  elem_t A[blk_dim][blk_dim], elem_t A[blk_dim][blk_dim])
52 {
53   char side, transD, diag, uplo;
54   side = 'R'; transD = 'T'; uplo = 'L'; diag = 'N';
55   G(trsm)(&side,&uplo,&transD,&diag,blk_dim,blk_dim,1.0,D,blk_dim,A,blk_dim);
```

```
56  }
57
58  void RPC_gemm(uint32_t blk_dim, elem_t A[blk_dim][blk_dim],elem_t L[blk_dim][blk_dim],elem_t U[blk_dim][blk_dim])
59  {
60    char transL = 'N';  char transU = 'T';
61
62    G(gemm) (&transL, &transU, blk_dim,blk_dim,blk_dim,-1.0,L,blk_dim,U,blk_dim,1.0,A,blk_dim);
63  }
64
65  void RPC_CholeskyMatrixBlk2D( uint32_t blk_dim, uint32_t nb_w_blk, elem_t A[nb_w_blk][nb_w_blk][blk_dim][blk_dim])
66  {
67    uint32_t i,j,k;
68    ulong blk_w_sz = blk_dim * sizeof(elem_t);
69    ulong blk_sz = blk_dim *  blk_w_sz;
70    elem_t (*Akk)[blk_dim];
71    elem_t (*Lkk)[blk_dim]; elem_t (*Aki)[blk_dim];
72    elem_t (*Ukk)[blk_dim]; elem_t (*Aik)[blk_dim];
73    elem_t (*Aij)[blk_dim]; elem_t (*Lik)[blk_dim];
74    elem_t (*Ukj)[blk_dim];;
75
76    for (k = 0; k < nb_w_blk; k++)
77      {
78        Ukk = Lkk = Akk = A[k][k];
79
80        RPC_CholeskyBlk2D(blk_dim, Akk);
81
82        for (i = k + 1; i < nb_w_blk; i++)
83          {
84            Aik = A[i][k];
85            RPC_trsm(blk_dim, 'U', Aik, Ukk);
86          }
87
88        for (j = k + 1; j < nb_w_blk; j++)
89          {
90            Ukj = A[j][k];
91
92            for (i = j; i < nb_w_blk; i++)
93              {
94                Aij = A[i][j], Lik = A[i][k];
95
96                RPC_gemm( blk_dim, Aij, Lik, Ukj);
97              }
98          }
99      }
100 }
101 void RPC_storeSymmMat2DToFile(char*filename,uint32_t blk_dim,uint32_t nb_w_blk,
102                               elem_t A[nb_w_blk][nb_w_blk][blk_dim][blk_dim])
103 {
104   uint32_t matrix_l,matrix_c,blk_l,blk_c;
105   uint32_t matrix_dim = blk_dim * nb_w_blk;
106
107   fopen(filename,"+w")
108
109   fprintf(fd,"%u %u",matrix_dim, matrix_dim);
110   for( matrix_l = 0; matrix_l < matrix_dim ; matrix_l++)
111     {
112       for ( matrix_c = 0; matrix_c < matrix_dim; matrix_c++)
113         {
114           elem_t (*Alc)[blk_dim];
115           uint32_t blk_l = matrix_l/blk_dim;
116           uint32_t blk_c = matrix_c/blk_dim;
117
118           if (blk_l == blk_c)
119             {
120               Alc = A[blk_l][blk_c];
121               if ( matrix_l > matrix_c)
122                 printf("% 4.3e,",Alc[matrix_l%blk_dim][matrix_c%blk_dim]);
123               else
124                 printf("% 4.3e,",Alc[matrix_c%blk_dim][matrix_l%blk_dim]);
125             }
126           else
127             {
128               if ( matrix_l > matrix_c)
129                 Alc = A[blk_l][blk_c];
130               else
131                 Alc = A[blk_c][blk_l];
132               printf("% 4.3e,",Alc[matrix_l%blk_dim][matrix_c%blk_dim]);
133             }
134         }
135     }
136 }
137
138
139 int main(int argc,char**argv)
140 {
141   uint32_t i;
142   ulong msg_sz;
143   elem_t (*A)[nb_w_blk][blk_dim][blk_dim];
```

```
144     uint32_t nb_w_blk;
145     uint32_t blk_dim;
146     uint32_t matrix_dim;
147     ulong matrix_sz;
148     char *filename;
149
150     nb_w_blk    = atoi(argv[1]);
151     blk_dim     = atoi(argv[2]);
152     matrix_dim = nb_w_blk * blk_dim;
153     matrix_sz   = matrix_dim*matrix_dim*sizeof(elem_t);
154     filename = argv[3];
155    A = malloc(matrix_sz);
156
157     RPC_fillSymmMatrix(blk_dim, nb_w_blk,&A);
158     RPC_CholeskyMatrixBlk2D( blk_dim, nb_w_blk, &A);
159     RPC_storeSymmMat2DToFile(filename,blk_dim, nb_w_blk,&A);
160
161     free(A);
162   }
```

Dans la version PRFX, le découpage par blocs va permettre également de générer du parallélisme et d'utiliser une distribution pour équilibrer la charge des processeurs. En effet, comme nous l'avons vu à la section 4.2, nous appliquons une distribution cyclique 2D (i.e. selon les deux dimensions) des tâches (et donc des blocs) sur les *threads* exécuteurs. De plus, en adoptant la version SMP (cf. ligne 99 du listing 2.24) de cette distribution, une meilleure localité est obtenue pour les grappes de SMP. L'équilibrage est d'autant meilleur que le nombre de blocs est grandement supérieur au nombre de *threads* exécuteurs. Il est aussi plus adéquat lorsque le déploiement est de 1 processus par nœud et de 1 *thread* par processeur. Dans ce cas, la distribution cyclique des blocs correspond exactement à la hiérarchie d'un nœud SMP.

Dans la version PRFX, seules les trois fonctions présentées dans le code 2.24 changent. La fonction `main()` contient les déclarations des RPC que nous allons utiliser ainsi que leurs modes d'accès et le choix de la distribution cyclique des tâches de calcul. La matrice de blocs est allouée en iso-mémoire. Sa taille est paramétrée par des données de niveau inspection (`argv[1]` et `argv[2]`). Nous distribuons le contrôle intrinsèque à l'algorithme entre l'initialisation (phase `0.0`), le calcul (phase `1.0`) et le stockage (phase `2.0`) de la matrice dans un fichier (cf. lignes 103 à 111 du listing 2.24). Le parallélisme présent dans l'algorithme de Cholesky est intégralement décrit par le code 2.24. Un code utilisant le mode d'expression RAPID ou ATHAPASCAN peut également exploiter ce parallélisme avec le même niveau d'écriture car les données accédées sont unitaires. Notons toutefois, qu'un code ATHAPASCAN requiert un volume de code annexe important à l'image de celui de l'algorithme de la factorisation LU (cf. Annexe A.4.2).

En revanche, une version OpenMP ou HPF de ce code, en gardant le même niveau d'écriture, ne peut ni décrire ni exploiter le parallélisme présent entre les itérations de la boucle externe en k.

Comme nous avons choisi de distribuer l'initialisation, nous devons déclarer un *stencil* dans la fonction `RPC_fillSymmMatrix()` pour décrire un bloc et créer un RPC par appel d'initialisation de chacun des blocs. Tous ces appels ont un numéro de placement correspondant au numéro du bloc qui est pris en compte par la distribution préalablement fixée. Ces appels sont également fixés dans la phase 0.0.

Ensuite, dans la partie contenant les trois boucles imbriquées de l'algorithme de Cholesky (fonction `RPC_CholeskyMatrixBlk2D()`), nous déclarons autant de pointeurs qu'il y a de dénominations usuelles des blocs pour cet algorithme. Nous aurions pu utiliser seulement trois pointeurs polyvalents, mais nous avons préféré conserver dans cette implémentation les notations

algorithmiques. A chacune de ces références aux blocs de la matrice, est associé un *stencil* 1D décrivant leur forme géométrique. Une fois tous ces *stencils* déclarés, les seuls changements concernent les appels BLAS qui doivent être transformés en appels de tâche via l'utilisation de `PRFX_call()`. La numérotation utilisée pour le placement des tâches est la même que celle de l'initialisation. Par contre, tous ces RPC sont dans la phase 1.0.

Enfin, l'écriture dans un fichier étant séquentielle après recentralisation automatique de la matrice par le support PRFX, seul l'appel a besoin d'être transformé. Cette fois, du point de vue de l'ordonnancement, la phase 2.0 pour séparer cette étape des précédentes.

Listing 2.24 – Implémentation parallèle avec PRFX par blocs 2D de l'algorithme de Factorisation de Cholesky de matrices pleines.

```
1  #include [...]
2  #include "prfx.h"
3
4  //type for matrice NxN with block storage
5  typedef double elem_t;
6  [...]
7
8  void RPC_storeSymmMat2DToFile(char*filename,uint32_t blk_dim,uint32_t nb_w_blk,
9                                elem_t A[nb_w_blk][nb_w_blk][blk_dim][blk_dim]);
10 void RPC_fillBlk2D(uint32_t blk_dim, elem_t A[blk_dim][blk_dim],uint32_t blk_l,uint32_t blk_c);
11 void RPC_CholeskyBlk2D(uint32_t blk_dim, elem_t A[blk_dim][blk_dim]);
12 void RPC_trsm(uint32_t blk_dim,  elem_t A[blk_dim][blk_dim], elem_t A[blk_dim][blk_dim]);
13 void RPC_gemm(uint32_t blk_dim, elem_t A[blk_dim][blk_dim],elem_t L[blk_dim][blk_dim],elem_t U[blk_dim][blk_dim]);
14
15 void RPC_CholeskyMatrixBlk2D( uint32_t blk_dim, uint32_t nb_w_blk, elem_t A[nb_w_blk][nb_w_blk][blk_dim][blk_dim])
16 {
17   uint32_t i,j,k;
18   ulong blk_w_sz = blk_dim * sizeof(elem_t);
19   ulong blk_sz = blk_dim *  blk_w_sz;
20   blk2d Akk;
21   blk2d Lkk, Aki, Ukk, Aik;
22   blk2d Aij, Lik, Ukj_t;
23
24   PRFX_stencilDecl1D(&Akk, blk_sz);
25   PRFX_stencilDecl1D(&Lkk, blk_sz);
26   PRFX_stencilDecl1D(&Aki, blk_sz);
27   PRFX_stencilDecl1D(&Ukk, blk_sz);
28   PRFX_stencilDecl1D(&Aik, blk_sz);
29   PRFX_stencilDecl1D(&Aij, blk_sz);
30   PRFX_stencilDecl1D(&Lik, blk_sz);
31   PRFX_stencilDecl1D(&Ukj_t, blk_sz);
32
33   for (k = 0; k < nb_w_blk; k++)
34     {
35       Ukk = Lkk = Akk = A[k][k];
36       PRFX_call(k*nb_w_blk+k, 1.0, RPC_CholeskyBlk2D, blk_dim, &Akk);
37       for (i = k + 1; i < nb_w_blk; i++)
38         {
39           Aik = A[i][k];
40           PRFX_call(i*nb_w_blk+k, 1.0, RPC_trsm, blk_dim, 'U', &Aik, &Ukk);
41         }
42       for (i = k + 1; i < nb_w_blk; i+)
43         {
44           Lik = A[i][k];
45           for (j = k + 1; j <= i; j++)
46             {
47               Aij = A[i][j], Ukj_t = A[j][k];
48               PRFX_call(i*nb_w_blk+j,1.0, RPC_gemm, blk_dim, &Aij, &Lik, &Ukj_t);
49             }
50         }
51     }
52 }
53
54 void RPC_fillSymmMatrix(uint32_t blk_dim, uint32_t nb_w_blk, elem_t A[nb_w_blk][nb_w_blk][blk_dim][blk_dim])
55 {
56   uint32_t blk_l, blk_c;
57   ulong blk_sz =  blk_dim *  blk_dim *sizeof(elem_t);
58   elem_t (*Alc)[blk_dim];
59
60   PRFX_stencilDecl1D(&Alc, blk_sz);
61
62   for (blk_c = 0; blk_c < nb_w_blk; blk_c++)
63     for(blk_l = blk_c; blk_l < nb_w_blk; blk_l++)
64       {
65         int place = blk_l*nb_w_blk+blk_c;
66
```

```
67              Alc = A[blk_l][blk_c];
68
69              PRFX_call(place,0.0, RPC_fillBlk2D, blk_dim, &Alc, blk_l, blk_c);
70            }
71    }
72
73    int main(int argc,char**argv)
74    {
75      uint32_t i;
76      ulong msg_sz;
77      elem_t (*A)[nb_w_blk][blk_dim][blk_dim];
78      uint32_t nb_w_blk;
79      uint32_t blk_dim;
80      uint32_t matrix_dim;
81      ulong matrix_sz;
82
83      nb_w_blk    = atoi(argv[1]);
84      blk_dim     = atoi(argv[2]);
85      matrix_dim = nb_w_blk * blk_dim;
86      matrix_sz   = matrix_dim*matrix_dim*sizeof(elem_t);
87
88      PRFX_init(NULL);
89
90      PRFX_declRPC2(RPC_CholeskyBlk2D,1,IMM,RW,cost_CholeskyBlk2D);
91      PRFX_declRPC4(RPC_trsm,1,IMM,IMM,RW,RO,cost_trsm);
92      PRFX_declRPC4(RPC_gemm,1,IMM,RW,RO,RO,cost_gemm);
93      PRFX_declRPC3(RPC_CholeskyMatrixBlk2D,0,IMM,IMM,VR,NULL);
94      PRFX_declRPC5(RPC_fillBlk2D,1, IMM,IMM,FW,IMM,IMM,cost_fillBlk2D);
95      PRFX_declRPC3(RPC_fillSymmMatrix,0,IMM,IMM,VR,NULL);
96      PRFX_declRPC4(RPC_storeSymmMat2DToFile,1,RO,IMM,IMM,RO,NULL);
97      PRFX_declRPC1(PRFX_isofree,0,FW,NULL);
98
99      PRFX_thrDistributionSetDefault(PRFX_DISTRIB_CYCLIC2D_LINE_COLUMN_SMP, nb_w_blk, 2, 2, 3, 5);
100
101     PRFX_isoMalloc(&A, matrix_sz);
102
103     PRFX_call(0,0.0,RPC_fillSymmMatrix,blk_dim, nb_w_blk,&A);
104
105     PRFX_call(0,1.0,RPC_CholeskyMatrixBlk2D, blk_dim, nb_w_blk, &A);
106
107     PRFX_isomalloc (&filename, strlen(argv(4));
108     strcpy(filename, argv[4]);
109     PRFX_call(0,2.0,RPC_storeSymmMat2DToFile,&filename, blk_dim, nb_w_blk,&A);
110
111     PRFX_call(0,2.0,PRFX_isofree, &A);
112
113     PRFX_terminate();
114   }
```

L'ordonnancement du DAG de tâches de l'algorithme de Cholesky est coûteux car le nombre de tâches est important. Ce nombre de tâches dépend du facteur de découpage d (constante chosie en fonction des performances des BLAS3) de la matrice avec $dN = n$. Le nombre de tâches est donc de l'ordre de $O(N^3/3)$. Le nombre de dépendances est de 4 (une de contrôle et trois de données) pour la majorité des tâches (celles effectuant les calculs de contribution) ; le nombre total de dépendances est donc de l'ordre de $O(4N^3/3)$. Comme nous l'avons vu, les heuristiques d'ordonnancement utilisées ont une complexité de l'ordre de $O(|E|log(|E|))$; donc l'algorithme d'ordonnancement a une complexité de l'ordre de $O(4N^3log(N))$. L'application d'une heuristique d'ordonnancement serait coûteuse mais elle pourrait être rentabilisée en la réutilisant pour le traitement de plusieurs problèmes de taille identique, mais avec des coefficients différents. Néammoins, un calcul complet d'un ordonnancement ne serait ici pas très judicieux, car la distribution analytique bloc cyclique 2D conduit déjà à de bonnes performances. De plus, pour générer les vecteurs d'ordonnancement, une formule analytique spécifique existe et est moins coûteuse qu'une heuristique générique.

2.9 Conclusion

Nous venons de présenter dans ce chapitre le mode d'expression PRFX et les choix effectués. Ces choix sont cohérents ; ils permettent d'offrir un mode d'expression aisé et fin des algorithmes parallèles grâce à l'iso-espace d'adressage et à la génération automatique des synchronisations et de la cohérence. Ces choix sont réalistes, car ils ne requièrent pas l'existence d'un inspecteur doté de capacités d'analyse du code source. Notre inspecteur a un rôle mécanique simple portant sur les intersections des accès des tâches. Ces accès sont connus précisément car explicitement fournis par le programmeur à l'inspecteur. Les choix de l'iso-mémoire et de la modélisation du programme parallèle PRFX par un DAG de tâches complet permettent d'appliquer des optimisations avant l'exécution (ordonnancement, agrégation des arcs du DAG, préparation des compteurs de synchronisation) et pendant l'exécution parallèle (communications unilatérales, exploitation de la mémoire partagée par les *threads*). De plus, l'utilisateur peut paramétrer le déploiement et l'ordonnancement afin d'exploiter la hiérarchie de machines hétérogènes comme par exemple les grappes de nœuds SMP.

Les exemples d'utilisation présentés nécessitent l'implémentation d'un support pour pouvoir être exécutés. Ce support implémente l'interface indiquée mais également des services annexes tels que l'iso-mémoire, l'inspection, l'ordonnancement, le déploiement *thread* / processus et l'exécution. C'est la mise en œuvre de ce support que nous nous proposons de voir au chapitre 3.

Chapitre 3

Implémentation de PRFX

PRFX a été implémenté majoritairement pour le système IBM AIX (version 4.3 à 5.2) en langage C. Cette implémentation représente environ 14000 lignes de code pour l'inspecteur / ordonnanceur et 7000 pour la partie exécuteur. Un portage pour le système SGI IRIX 6.4 a été effectué au cours du développement du support PRFX. Ce portage a permis d'augmenter la portabilité de la plate-forme par rapport à l'utilisation des *threads* POSIX et à la manière de les déployer sur les processeurs. Les communications unilatérales ont été rendues génériques et indépendantes de la bibliothèque LAPI [38] d'IBM.

Les machines utilisées lors du développement sont des nœuds SMP IBM NH2 16 voies, reliés par un réseau *Colony*, situés à Montpellier et administrés par le CINES. Depuis le 20 octobre 2004, quatre nœuds IBM p690+, interconnectés par un réseau *Federation*, sont également utilisables . Le système de réservation des processeurs offre de nombreuses possibilités. Ceci nous a permis de travailler à distance, dans le contexte de production de cette machine sans trop le perturber.

Nous allons donner quelques détails de l'implémentation des principales fonctionnalités décrites au chapitre précédent, notamment du point de vue des structures de données utilisées. Ces fonctionnalités supposent l'existence d'une iso-mémoire. De plus, elles ont des comportements différents selon qu'elles sont exécutées dans le contexte de l'inspection ou dans celui de l'exécution parallèle.

3.1 Iso-mémoire et iso-allocateur

L'iso-mémoire est implémentée par saturation d'un allocateur *thread-safe* fourni par le programmeur lors de l'initialisation de PRFX. Par défaut, les fonctions `malloc()`, `realloc()` et `free()` sont utilisées. La réutilisation d'un allocateur existant permet de bénéficier de sa technologie et de ses éventuelles fonctionnalités de déboguage ou de vérification associées. Toutes les allocations effectuées avant l'initialisation de PRFX avec cet allocateur ne doivent être libérées qu'après la terminaison de `PRFX_terminate()`. La saturation de l'allocateur par les processus PRFX permet de trouver un espace d'adresses commun à tous. Cet espace d'adresses a les mêmes bornes pour tous ces processus, il correspond donc bien à un iso-espace d'adressage.

Nous implémentons une version de l'iso-allocateur qui ne nécessite pas de synchronisations entre les processus lors des allocations ou libérations d'iso-mémoire (cf. section 2.3.1). Pour ce faire, une fois l'allocateur saturé, chaque processus libère ce qui deviendra sa zone d'iso-allocation locale et exclusive. La taille de ces zones est récupérée dans le fichier de déploiement.

La mise en place de l'iso-mémoire précède le démarrage de l'inspecteur pour lui permettre d'y allouer des structures de données qui persisteront jusqu'à l'exécution (e.g. le catalogue des *stencils* ou les zones de réception des arguments des tâches que nous verrons ci-après). Par contre, les données parallèles de l'utilisateur, qui ont été allouées lors de l'inspection, sont libérées pour être ensuite réallouées lors de l'exécution parallèle. Cette séparation est nécessaire car l'exécution peut intervenir après la terminaison du processus effectuant l'inspection et l'ordonnancement (cas de l'amortissement de l'inspection). De plus, elle permet de conserver une exécution parallèle où les iso-allocations continuent de se faire dynamiquement. Nous pourrions détourner la fonction d'iso-allocation lors de la phase parallèle pour qu'elle retourne exactement les mêmes zones mémoires que celles obtenues lors de l'inspection, mais ce fonctionnement peut entraîner une sur-consommation mémoire. En effet, l'inspecteur exécute séquentiellement et de façon légère le programme ; les zones mémoires obtenues lors de la pré-exécution ne se libèrent donc pas aussi rapidement que lors de l'exécution parallèle. Reproduire le même schéma d'iso-allocation de la pré-exécution, lors de l'exécution, conduira donc inutilement à l'utilisation de vastes plages d'adresses qui risquent de forcer le système à mettre des pages mémoire sur le disque dur. L'inspecteur ne travaille que sur les adresses et non sur le contenu des données parallèles, il n'est donc pas sujet à ce phénomène.

3.2 Structures de données internes

La structure de données principale est celle du DAG de tâches complet dont on voit un exemple partiel dans la figure 9. Comme indiqué sur cette figure, le contenu des tâches utilisateurs et des tâches internes de PRFX requiert en particulier des structures de données annexes relatives aux fonctions cibles de RPC et aux *stencils* à appliquer pour instancier les communications. Au cours de l'inspection et de l'exécution du programme, certaines structures de données persistent. Pour que la taille de ces dernières reste raisonnable, nous exploitons le fait que les mêmes "objets" sont rencontrés plus ou moins souvent en fonction de la régularité du programme. Nous optimisons donc leur stockage en factorisant leurs occurrences dans des "catalogues" où un seul de leur représentant est présent. Ces catalogues ont une structure ordonnée pour permettre des opérations de manipulation et de recherche de faible complexité. Nous avons un catalogue pour la description des fonctions de tâches et un catalogue en relation avec la description des *stencils* déclarés par le programmeur.

Le catalogue des fonctions cibles de RPC décrit les modes d'accès de chacun de leurs paramètres. Il est construit lors de la pré-exécution via les informations indiquées dans les appels de type `PRFX_declRPC()`. Il stocke également le fait que l'exécution d'une fonction est nécessaire ou non à la construction du DAG de tâches lors de la pré-exécution. Ce catalogue est en mémoire normale (tas) et est identique pour tous les processus PRFX jusqu'à l'exécution parallèle.

Les structures de données associées aux *stencils* décrivent la géométrie des données parallèles en mémoire sous la forme d'un ensemble de plages d'*offsets*. Cette description est indépendante de la localisation de la donnée parallèle en mémoire. Les *stencils* décrivant des motifs géométriques répétitifs sont stockés de façon compacte. Pour cela, deux ensembles de plages d'*offsets* spéciaux $m1$ et $m2$ de même cardinalité sont décrits dans un même *stencil* et un facteur de répétition n y est stocké. L'intégralité des plages d'*offsets* décrites par ce *stencil* compact sont obtenues en répétant à intervalle régulier le motif $m1$ avec des altérations. L'intervalle de répétition est toujours le même, à savoir la distance qui sépare $m1$ de $m2$. Les altérations sont une amplification linéaire, à chaque répétition, des différences constatées entre $m1$ et $m2$.

Pour obtenir l'adresse de base servant à l'application des *stencils* dans les données parallèles, nous avons deux protocoles distincts. Ils ont l'avantage d'être indépendants des adresses mémoires effectives des données parallèles cibles. Pour ce faire, ces protocoles utilisent, soit un numéro d'argument d'entrée de la tâche, soit un numéro d'iso-allocation effectuée par cette tâche afin d'obtenir l'adresse de base requise.

Le catalogue des *stencils* du programmeur et de ceux obtenus en interne par l'inspecteur (lors des intersections d'accès) est maintenu en assurant que chaque *stencil* catalogué est unique. Les *stencils* sont normalisés de sorte que deux *stencils* décrivant la même forme géométrique avec des plages d'*offsets* différentes ont la même forme normale. Par exemple le *stencil* s1 : $\{[0,16],[16,32]\}$ et le *stencil* s2 : $\{[0,8],[8,32]\}$ ont la même forme normale : $\{[0,32]\}$. Le catalogue des *stencils* contient tous les *stencils* stockés de façon unique et normalisée. A chaque fois qu'un *stencil* interne à PRFX ou du programmeur est créé, seule sa forme normalisée est utilisée. Elle est ajoutée dans le catalogue si elle n'existait pas déjà.

Le nombre de *stencils* dans le catalogue est inférieur au nombre d'accès aux données. Il peut être égal, par exemple dans le cas d'applications irrégulières où aucun accès n'a la même géométrie tout le long de la vie du programme. Cette description normalisée permet à l'inspecteur de travailler avec un nombre réduit de *stencils* et ceci motive l'existence de ce catalogue.

Le catalogue de *stencils* est utilisé par l'inspecteur, dès que le programmeur déclare un nouveau *stencil*, pour rechercher si sa forme normale existe déjà. Ce catalogue est construit lors de l'inspection et reste présent en iso-adresse dans chaque processus PRFX lors de l'exécution. Ceci permet d'optimiser la réalisation des dépendances de contrôle en n'envoyant qu'un message de taille réduite. Cette dépendance porte les arguments du RPC (i.e. les données immédiates et les pointeurs de pointeurs de données parallèles) et leurs *stencils* associés. Notre fonctionnement optimisé n'envoit que des référence sur les *stencils* car leurs structures de données sont disponibles à destination.

3.3 Inspecteur

L'inspecteur est implémenté par un analyseur de dépendances créant, utilisant et mettant à jour les structures de données, persistant jusqu'à l'exécution, précédemment mentionnées. Il fait également usage de structures de données temporaires telles que le catalogue des accès ou l'arbre de l'historique des accès atomiques que nous découvrirons ci-après.

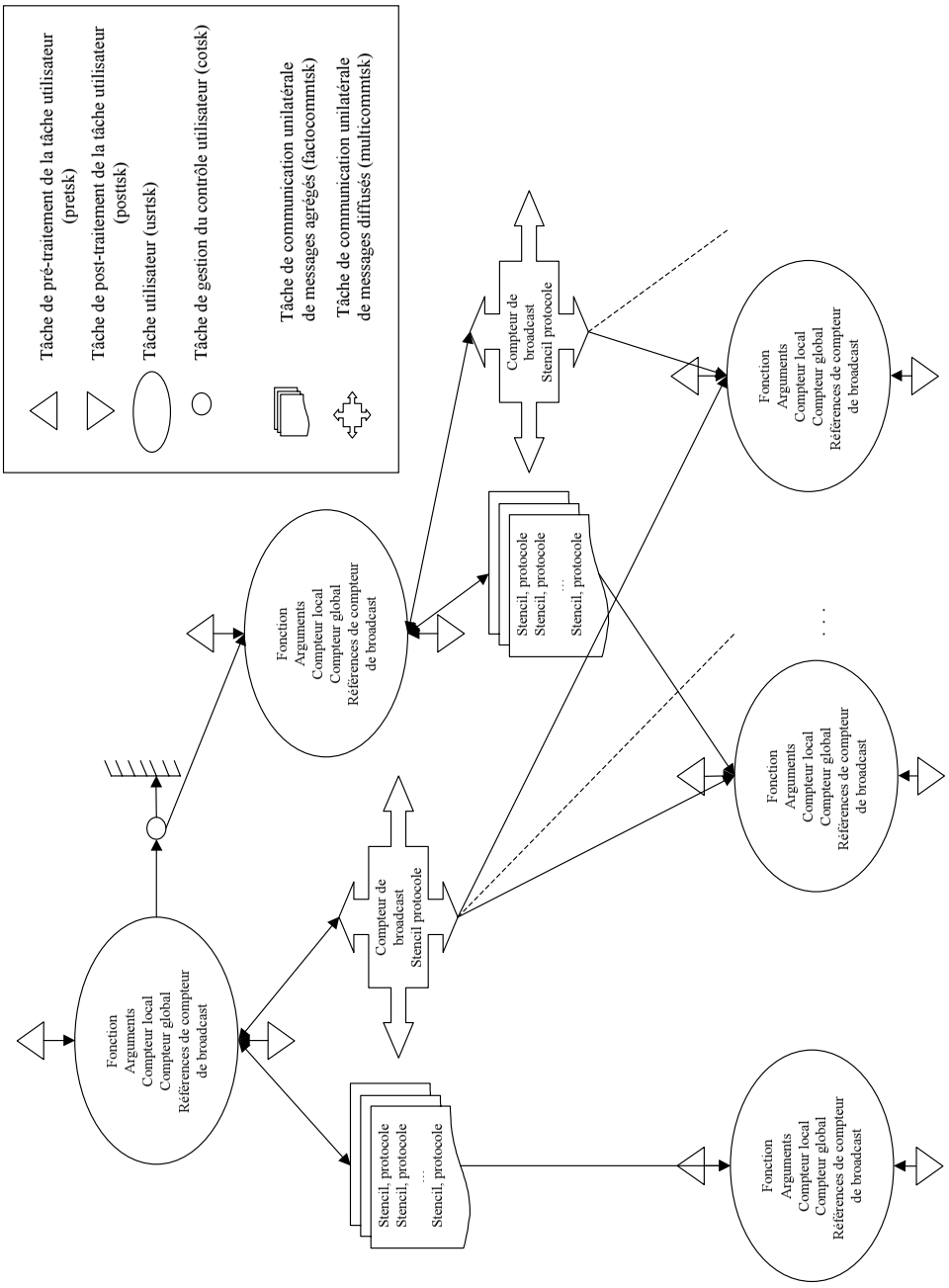

FIG. 9 – Structure de données simplifiée du DAG complet.

Le DAG complet nous offre une approche uniquement par tâche. Pour chaque tâche nous connaissons les données accédées, si bien que pour résoudre les dépendances d'un accès de tâche, nous ne disposons pas directement de la liste des tâches en relation avec cet accès. Pour accélérer cette recherche, nous utilisons une structure de données permettant de connaître, pour un accès donné, la liste des tâches en relation et la nature de cette relation (lecture RO, écriture RW, écriture totale FW, virtuelle VR) .

L'inspecteur est générique car il travaille sur des accès représentés par des plages d'adresses mémoire. Il peut donc générer les dépendances indépendamment de la signification logique des données utilisées par le programmeur. Pour rappel, ce n'est pas le cas dans des supports comme RAPID ou ATHAPASCAN où les données parallèles sont des objets de la plate-forme avec les limitations que nous avons vues, mais avec l'avantage, par contre, d'être manipulables symboliquement.

L'inspecteur travaille en interne avec de vraies adresses obtenues via l'iso-allocateur mémoire. Elles apparaissent donc dans ses structures de données temporaires ; par contre, dans le DAG complet, ces adresses sont remplacées par un couple *stencil*, protocole d'application en mémoire. Nous rappelons que ce dernier indique comment obtenir l'adresse mémoire servant de base à l'application du *stencil*. Le DAG est ainsi paramétré par les futures valeurs des iso-adresses qui seront obtenues lors des iso-allocations de l'exécution parallèle. Le fait de générer les dépendances de cette façon, en relatif, nous permet de réutiliser un fichier de DAG complet pour l'exécuter (exécuteur parallèle *standalone*) dans un processus totalement nouveau. Dans ce processus, les iso-allocations se font dynamiquement et les adresses obtenues n'ont aucune garantie d'être identiques à celles utilisées pour construire le DAG complet.

L'inspection démarre par l'exécution de la tâche `main()` du programmeur, c'est la seule tâche créée non pas par un `PRFX_call()`, mais spontanément par PRFX.

A la fin de l'inspection, le DAG de tâches complet est réalloué dans un espace contigu en mémoire pour pouvoir être envoyé en un seul message aux autres processus. Cette duplication est une solution temporaire permettant d'augmenter le coût mémoire mais économisant les coûts de gestion qu'impliquerait la distribution de ce DAG (cf. 5.4.1). Le DAG est alors disponible pour l'ordonnanceur.

3.3.1 Arbre de l'historique des accès atomiques dans les données parallèles

Les feuilles de cet arbre peuvent être vues comme la description (intervalle d'adresses) de chaque pièce du puzzle dessiné par l'application des *stencils* s'intersectant ou non dans les données parallèles. Cet arbre est construit de façon incrémentale au cours de la pré-exécution. Les nœuds de l'arbre sont donc des descriptions d'anciens accès atomiques qui se sont vus redécoupés par des accès plus fins effectués par de nouvelles tâches.

L'adjectif atomique est utilisé pour un accès *a* pour signifier qu'aucune tâche n'a effectué jusqu'alors d'accès morcelant *a*. Cette structure est au cœur de la résolution des dépendances de

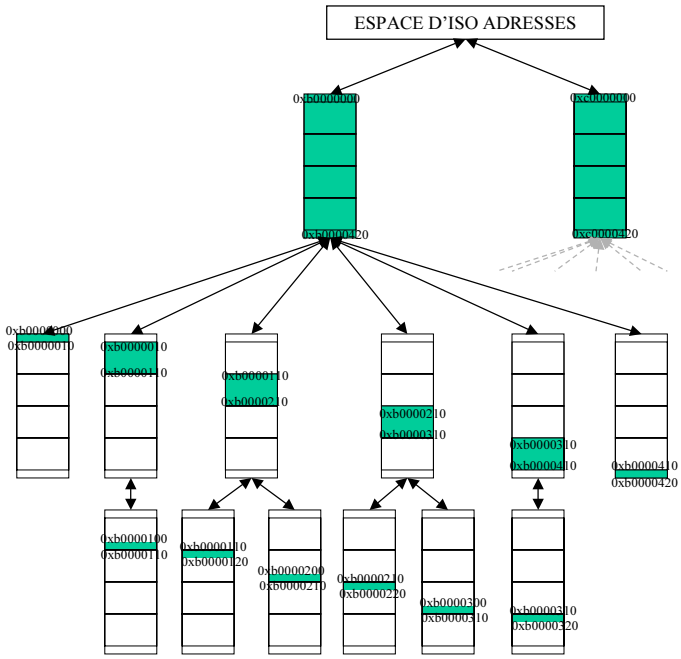

FIG. 10 – Arbre de l'historique des accès atomiques aux maillages pour l'exemple du problème de Laplace 1D.

données, elle archive le déroulement de l'exécution du point de vue des accès aux données parallèles. Elle permet à l'inspecteur d'intégrer très finement les nouveaux accès relativement aux anciens. Cette finesse de description, couplée à la connaissance des tâches en relation avec n'importe quel accès atomique, lui permet de décider si les accès de deux tâches sont indépendants ou non.

Au fur et à mesure de l'inspection, l'arbre de l'historique des accès atomiques est mis à jour lors des créations (`PRFX_isoMalloc()`), libérations (`PRFX_isoFree()`) de données parallèles et lors des instanciations d'accès aux données via les arguments des `PRFX_call()`.

Pour un appel à `PRFX_isomalloc()`, une nouvelle entrée est insérée dans l'arbre. Cette entrée est un intervalle unique d'adresses car, par définition, `PRFX_isoMalloc()` réserve une zone mémoire contiguë. Cette entrée est insérée à la racine de l'arbre car l'accès étant par nature nouveau, il n'a aucune relation avec les précédents accès répertoriés dans cet arbre. Il est inséré en respectant un ordre portant sur la borne inférieure de l'intervalle d'adresses. A chaque `PRFX_isoFree()`, le sous-arbre partant de la racine et correspondant à l'intervalle précédemment alloué est élagué.

Pour chaque nouvel accès instancié par les paramètres d'un `PRFX_call()`, sa décomposition en accès atomiques est recherchée dans l'arbre. L'arbre est parcouru en tirant parti de l'ordre et de la hiérarchie inclusive entre les pères et les fils. Si cette décomposition existe déjà, alors la relation avec la tâche appelée est sauvegardée dans la table de l'accès utilisateur existant, et dans toutes les tables des accès atomiques faisant partie de la décomposition. Par contre, si l'accès est

nouveau, les intersections qu'il produit avec les accès atomiques existant génèrent des nouvelles feuilles dans l'arbre. Elles correspondent à de nouveaux accès atomiques encore plus fins. Pour chacun de ces nouveaux accès, la liste des tâches en relation d'accès est recopiée depuis l'accès atomique père. Dans les deux cas, les tables des anciens accès atomiques sur le chemin remontant de père en père jusqu'à la racine sont mises à jour. Cette mise à jour s'effectue en insérant dans la table le numéro de la tâche créée et le numéro d'iso-allocation ou d'argument en relation.

Dans le cas de l'implémentation de l'algorithme de Jacobi où le maillage est découpé par blocs de lignes et où un maillage auxiliaire est utilisé, l'arbre de l'historique des accès est au final celui présenté dans la figure 10. Les parties en grisé correspondent effectivement au contenu de cette structure de données, les parties claires sont données à titre indicatif dans ce schéma pour améliorer sa lisibilité. Nous voyons sur ce schéma les deux iso-allocations du maillage principal (adresse fictive obtenue lors de l'inspection `0xb0000000`) et auxiliaire (adresse fictive `0xc0000000`) qui correspondent aux deux nœuds fils de la racine. Seul le sous-arbre concernant le maillage principal est détaillé car celui du maillage auxiliaire est similaire modulo un *offset* de décalage sur les adresses. Comme indiqué précédemment, ces nœuds fils sont rangés par ordre croissant de leur adresse de base de l'accès décrit. Ces fils ont été rajoutés de façon incrémentale. Les fils du premier nœud correspondent aux accès pour l'initialisation des deux lignes de début et de fin du maillage, puis aux accès pour l'initialisation des blocs. Les feuilles de l'arbre quant-à elles, sont apparues lors des premières mises à jour de blocs lorsque chaque bloc augmenté du maillage principal a été lu. Elles correspondent aux zones de chevauchement des accès de voisinage. Il faut noter que la structure de l'arbre de l'historique des accès aurait été différente si les accès aux blocs augmentés avaient précédé les accès aux blocs.

3.3.2 Utilisation des catalogues de *stencils* et d'accès

La figure 11 présente les relations entre le catalogue de *stencils* à gauche, le catalogue des accès au centre, et l'arbre de l'historique des accès à droite pour l'algorithme de Laplace en version blocs de lignes. Les *stencils* utilisés sont, en partant du haut, celui décrivant l'intégralité du maillage qu'il soit principal ou auxiliaire, celui décrivant un bloc quelconque de lignes d'un de ces maillages, celui décrivant un bloc de lignes augmenté quelconque d'un de ces maillages et enfin celui décrivant une ligne du maillage. Le catalogue des accès fait référence aux *stencils* et indique pour chacun de ces derniers, les adresses de base sur lesquelles le *stencil* a été appliqué. Pour le premier *stencil*, deux accès sont catalogués ; ils correspondent respectivement aux maillages principal et auxiliaire. Pour le deuxième *stencil*, toutes les adresses de début de bloc sont cataloguées, à savoir 4 pour le maillage principal et 4 pour le maillage auxiliaire. De même pour le *stencil* du bloc augmenté, 8 adresses de base au total sont cataloguées. Le dernier *stencil* n'a pas d'adresse de base d'accès catalogué car il s'agit d'un *stencil* interne avec lequel aucun accès utilisateur n'a été fait. En poursuivant la lecture du schéma vers la droite, on voit que chaque adresse de base est associée à autant d'accès atomiques ou anciennement atomiques. Par exemple, les accès à des blocs augmentés de l'utilisateur sont associés dans l'arbre par une décomposition en trois accès atomiques, à savoir une ligne, un bloc et une ligne.

Le catalogue des accès stocke également la liste des tâches en relation avec chaque accès et ceci dans l'ordre anti-séquentiel. Nous pouvons voir sur la figure 12 que, par exemple, l'accès

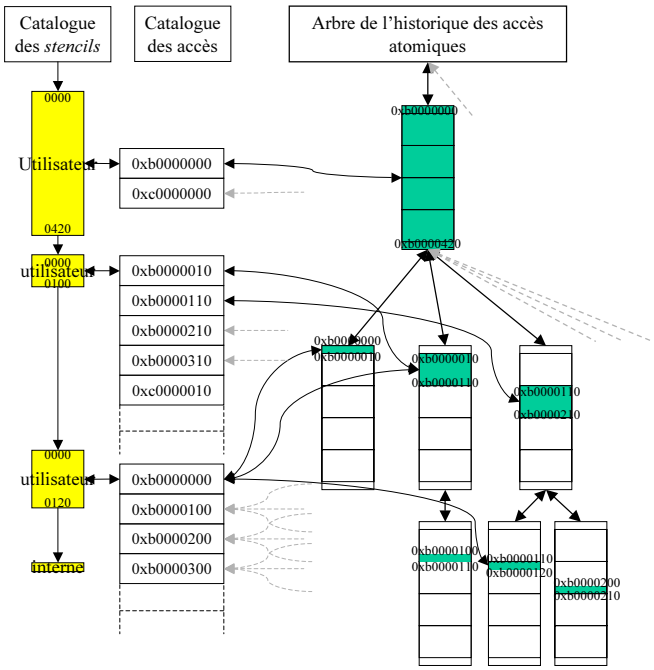

FIG. 11 – Associations entre accès utilisateur et accès atomiques.

basé en `0xb0000000` selon la géométrie du premier *stencil* a été en relation (de façon chronologique) avec la tâche *Ti* via son iso-allocation d'index 0, puis avec l'argument 2 de la tâche *Ti*, etc.... Nous utilisons abusivement la notation *Ti* pour parler d'une tâche dont le numéro n'est pas important pour la compréhension du schéma.

3.3.3 Construction des tâches utilisateurs

Lors de la pré-exécution, un contexte de tâche doit être maintenu pour chaque tâche dont l'inspection n'est pas terminée. En effet, contrairement à l'exécution parallèle où les tâches sont atomiques, l'inspection s'exécute en séquentiel. Elle descend donc dans l'arbre des appels de tâches. Ceci nécessite de maintenir et de restaurer un contexte d'inspection par tâche. Ce contexte contient la distribution courante et les associations des pointeurs de pointeur de données parallèles avec les *stencils*.

Il existe autant d'appels RPC que de nombres de paramètres possibles (`PRFX_call0()`, `PRFX_call1()`, `PRFX_call2()`, `PRFX_call3()`,...) et ceci pour éviter les surcoûts des appels de fonctions à nombre de paramètres variables. Les `PRFX_call0()` à `PRFX_call32()` sont ainsi disponibles pour lancer des tâches utilisateurs de 0 à 32 paramètres.

Lors de la construction de la tâche utilisateur, le numéro du *thread* exécuteur de destination est calculé et enregistré d'après la distribution courante et la numérotation du programmeur (premier paramètre du `PRFX_call()`). Le nombre de dépendances à satisfaire pour cette tâche est également enregistré.

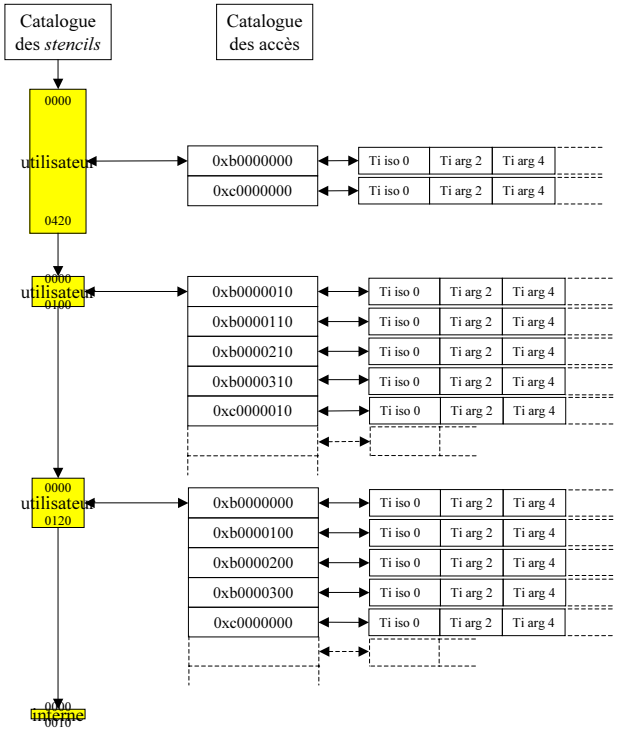

FIG. 12 – Associations entre accès, liste des tâches en relation et protocoles d'application des *stencils*.

Le compteur des synchronisations locales de la tâche est initialisé à zéro. Un compteur pour les synchronisations distantes et autant de références à des compteurs de *broadcast* que la tâche possède de précédences satisfaisant ses accès en lecture seule (RO), sont réservés en iso-mémoire. Ces compteurs sont également mis à zéro.

Le contexte de la tâche est initialisé avec la même distribution des RPC que la tâche initiatrice. L'ensemble des plages d'adresses atteintes par ses arguments en lecture ou écriture (RO, CRW, RW, FW) est calculé et associé à la tâche. Les pointeurs de pointeurs (dans la pile du *thread*) des arguments de RPC correspondant à des données parallèles sont associés à leur *stencil*.

3.3.4 Construction des tâches internes

Les tâches internes (cf. figure 9) sont des tâches annexes des tâches utilisateurs. Elles sont prévues de façon statique. Une tâche utilisateur et son cortège de tâches internes associées doivent être exécutées par le même processus, elles peuvent donc être placées par l'ordonnanceur sur n'importe quel *thread* exécuteur d'un même processus. Ces tâches sont exécutées dans l'ordre suivant : tâche de pré-traitement, puis tâche utilisateur et ses tâches de gestion du contrôle utilisateur (correspondant aux `PRFX_call()`) et enfin la tâche de post-traitement et les tâches de communication sortantes de la tâche utilisateur.

La définition de ces tâches internes permet de circonscrire les opérations relevant de la gestion du parallélisme, dans des objets manipulables.

3.3.4.1 Tâches de contrôle

Les tâches de contrôles sont créées en nombre égal à celui des `PRFX_call()` effectués par le programme utilisateur. Elles sont créées avec pour seule information le numéro de la tâche destination.

3.3.4.2 Tâches de pré et post traitement

Les tâches de pré et post traitement sont automatiquement accolées à la tâche utilisateur par l'inspecteur. Elles sont juste constituées d'un état et d'une référence vers la tâche utilisateur dont elles s'occupent. L'identification d'une tâche de préparation (resp. terminaison) qui précède (resp. suit) l'exécution de chaque tâche utilisateur permet d'ordonnancer cette gestion et d'en déporter le coût. Par conséquent, l'ordonnanceur peut par exemple dédier des *threads* exécuteurs à des tâches utilisateur et déléguer leur pré et post-traitement à un autre *thread* exécuteur. Nous nous référerons par la suite à ces tâches par la dénomination de pré-tâche et de post-tâche.

3.3.4.3 Tâches de communication

Les tâches de communication sont associées à une tâche utilisateur. Elles sont créées même si le placement indique qu'il n'y aura pas de communication ; en effet, l'inspecteur crée un DAG

complet indépendant du placement et de l'ordonnancement des tâches. Elles se chargent de réaliser les dépendances du DAG complet et sont divisées en deux groupes pour pouvoir utiliser les fonctionnalités de la bibliothèque de communication :

- les `Factocommtsk` gérant les dépendances "écriture vers lecture seule" (*broadcast* vers 1 à n tâches) ;
- les `Multicommtsk` gérant les dépendances "écriture vers écriture" ou "écriture vers écriture totale" (anti-dépendance) ou "lecture seule vers écriture" (anti-dépendance).

Lorsqu'une dépendance d'une tâche $t1$ vers une tâche $t2$, de type *broadcast*, est détectée par l'inspecteur, une recherche est faite parmi les tâches de *broadcast* associées à $t1$. Cette recherche consiste à trouver le même *stencil* et le même protocole d'application. En cas de succès, seul le numéro de la tâche destination est rajoutée avec les autres destinataires de ce *broadcast*. Sinon, une nouvelle tâche de *broadcast* est créée pour ce *stencil* avec son protocole d'application. La liste de destinataires est alors uniquement composée de la tâche cible $t2$.

Les messages agrégés correspondent au deuxième groupe et dans ce cas, on adopte une répartition par tâche destination. Ainsi, toutes les communications concernant une même tâche sont regroupées dans la même structure de données de la tâche de communication.

Pour les *broadcasts* comme pour les messages agrégés, les plages d'adresses à transmettre ne sont instanciables qu'au moment de l'exécution parallèle. Pendant l'inspection, les adresses des données parallèles utilisées sont factices, elles ne peuvent donc être stockées dans les structures de données des tâches de communication.

Pour l'instant ces tâches de communication ne comportent pas ce qui devrait être le troisième groupe de tâches spéciales pour gérer les communications entre tâches effectuant des accès commutatifs (CRW).

3.4 Implémentation de l'exécuteur parallèle

L'exécuteur parallèle est composé de processus préalablement déployés par un outil externe (e.g. poe, mpirun, etc...). Par contre, les *threads* internes sont déployés et attachés aux processeurs par PRFX d'après les indications du fichier de déploiement. Chaque *thread* exécuteur suit l'exécution du programme en faisant correspondre les événements de création (`PRFX_call()`) et terminaison de tâches utilisateurs avec l'activation des tâches internes associées de PRFX.

Les processeurs sont contrôlés en leur attachant les *threads* qui exécutent directement les instructions des tâches utilisateurs et internes sur le matériel.

Le réseau est plus difficilement contrôlable car il est accédé par l'intermédiaire des bibliothèques LAPI, ShMem ou MPI-2 qui ne proposent pas de qualité de service. Les modèles de communications sont donc peu fiables. La difficulté de modélisation est aussi due à leur accaparement de ressources processeur de façon asynchrone par rapport à l'exécution. En effet, les interruptions levées par l'arrivée des paquets réseaux interviennent de façon asynchrone sur n'importe quel processeur. Toutefois, il est à noter que les communications en espace utilisateur contribuent à un meilleur contrôle en évitant des commutations vers le noyau du système d'exploitation.

La mémoire est également cause d'incertitude. Les systèmes d'exploitation fournissent à

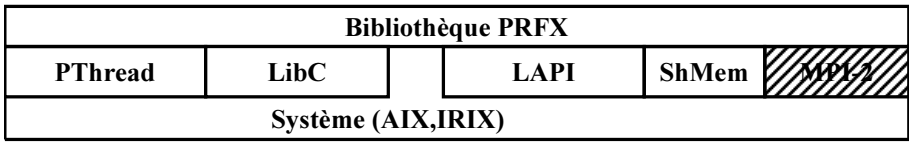

FIG. 13 – Bibliothèques sous-jacentes à PRFX.

chaque processus une mémoire virtuelle paginée dont le stockage réel en mémoire physique est peu contrôlable. Le mécanisme de translation d'adresses, les frontières de pages et les fautes de pages introduisent des coûts arbitraires. L'évolution se dirige vers un modèle avec des granularités de pages importantes, profitant des avantages du modèle segmenté. Ainsi, les temps difficilement modélisables perdus en gestion de la table des pages, en translation d'adresses, en défaut de cache TLB (*Trans Look-aside Buffer*) et mémoire sont moindres. Un autre phénomène non négligeable qui intervient maintenant dans les nœuds SMP avec un nombre important de processeurs concerne la cohérence mémoire. Elle est assurée par le matériel, mais les différents niveaux de cache et les stratégies de cohérence (avec propriétaire, migration de pages) introduisent des coûts difficilement modélisables.

Les couches logicielles comportent essentiellement les communications avec au-dessus les supports des modes d'expression de haut niveau. Leur comportement est très sensible à l'état et à la configuration de la machine (e.g. variables d'environnement). Leur utilisation est problématique et ne permet pas, de fait, un contrôle total de l'exécution sur le matériel. La figure 13 indique les couches logicielles sous-jacentes à la plate-forme PRFX, cette architecture minimise l'éloignement du support par rapport au système d'exploitation. PRFX utilise la bibliothèque de *threads* POSIX, la bibliothèque C et le système dans toutes les configurations d'exécution. Lorsque PRFX est utilisée avec un déploiement de plusieurs processus, une bibliothèque de communication unilatérale est utilisée. Cette bibliothèque est soit LAPI, soit ShMem, soit MPI-2. Cette dernière est utilisée dans un mode synchrone très pénalisant pour les performances car l'interface n'est pas adaptée à nos besoins.

3.4.1 Déploiement *threads* et processus

Comme nous l'avons vu, le déploiement est statique et les processus sont déployés par l'outil de déploiement choisi par le programmeur (poe avec AIX, mpirun (rsh, ssh) sous IRIX ou d'autres systèmes,...). Les *threads* sont créés par PRFX avec les caractéristiques du fichier de déploiement. Ces caractéristiques sont directement spécifiables lors de la création d'un *thread* POSIX. Nous exploitons ainsi la rapidité des *threads* utilisateurs POSIX lorsque plusieurs (`n`) d'entre eux sont affectés à un même *thread* noyau (mode `1:n`). Le programmeur (au travers de l'ordonnanceur) a alors intérêt à leur affecter des tâches de calcul lorsque les coûts de ces dernières sont mal cernés. Ainsi, le programme a un fonctionnement plus dynamique. En effet, pour un *thread* noyau, de multiples *thread* exécuteurs POSIX sont candidats à l'exécution de leur tâche courante. Statistiquement, ce mode de fonctionnement masque les attentes qui seraient intervenues dans un mode `1:1`. Le mode `1:1` où à un *thread* noyau correspond un *thread* exécuteur

POSIX permet de recouvrir les opérations bloquantes (e.g. tâches d'entrée / sortie) au niveau du système. Les *threads* noyaux sont ensuite attachés à un processeur physique de la machine. Cette association n'est pas standard et peut être indisponible sur certains systèmes d'exploitation.

Les *threads* permettent également d'exploiter les technologies de *multithreading* intégrées au niveau des nouvelles générations de processeurs (IBM POWER5, Intel Pentium 4,...). La structure de données liée au déploiement permet également d'enregistrer les traces d'exécutions par *thread* sans confusion.

3.4.2 Gestion des compteurs de dépendance

Lors de l'exécution, la connaissance de la satisfaction de toutes les dépendances d'une tâche est connue en faisant la somme de tous ses compteurs. Si cette somme est égale au nombre de ses précédences, alors la tâche peut passer dans l'état "prêt". Lorsque les précédences concernent des processus PRFX distants, les compteurs sont incrémentés en utilisant les fonctionnalités de la bibliothèque de communication. Notre implémentation utilisant la bibliothèque LAPI, elle exploite les compteurs nativement offerts par cette dernière. Une illustration du fonctionnement de cette bibliothèque est présentée par la figure 14 où une donnée parallèle `tab` est communiquée de façon unilatérale via LAPI. Cette communication correspond à une copie de la donnée `tab` vers l'iso-mémoire du processus distant. Au terme de cette copie, le compteur distant est automatiquement incrémenté par la bibliothèque LAPI. Ces deux opérations ne requièrent pas (d'un point de vue modèle) la participation du processus distant.

Nous avons mis en place des compteurs globaux et locaux pour être en adéquation avec l'utilisation d'une bibliothèque de communication unilatérale en extra-processus et l'utilisation d'incrémentation en mémoire partagée en intra-processus. Ces compteurs sont potentiellement incrémentés par plusieurs tâches de contrôle ou de communication aussi bien locales que distantes. La contention est diminuée grâce aux deux emplacements mémoire distincts : un pour les incrémentations intra-processus et un autre pour recevoir les incrémentations venant de l'extérieur. Les compteurs locaux sont incrémentés avec des opérations atomiques de type `__fetch_and_add()` évitant les surcoûts des *mutex* de la bibliothèque Pthread par exemple. Les compteurs distants sont incrémentés avec les fonctionnalités d'incrémentation de mots mémoire (ou compteurs) proposées par les bibliothèques de communication unilatérales.

Une autre optimisation est faite, elle consiste à réserver un compteur spécial pour chaque *broadcast*. Ainsi, ce compteur est référencé par l'ensemble des tâches utilisateurs ciblées par ce *broadcast*. Il a seulement deux valeurs possibles : 0 ou 1 selon que le *broadcast* est réalisé ou non. En dédiant un unique compteur par *broadcast* et par processus nous obtenons un fonctionnement sans séquentialisation tel qu'il est supposé par les modèles d'exécution théorique des heuristiques d'ordonnancement. En effet, au sein d'un processus, toutes les tâches dépendant d'un *broadcast* scrutent le même compteur, si bien que lorsqu'il est incrémenté, elles démarrent toutes en même temps (modulo la latence de la mémoire en intra-nœud). Sans cette optimisation, l'implémentation du *broadcast* devrait incrémenter un à un tous les compteurs des tâches concernées par ce même *broadcast*.

Lorsque l'exécuteur attend que la somme des compteurs de dépendances d'une tâche soit égale à son nombre de précédences, il peut fonctionner selon deux modes. Dans le premier mode,

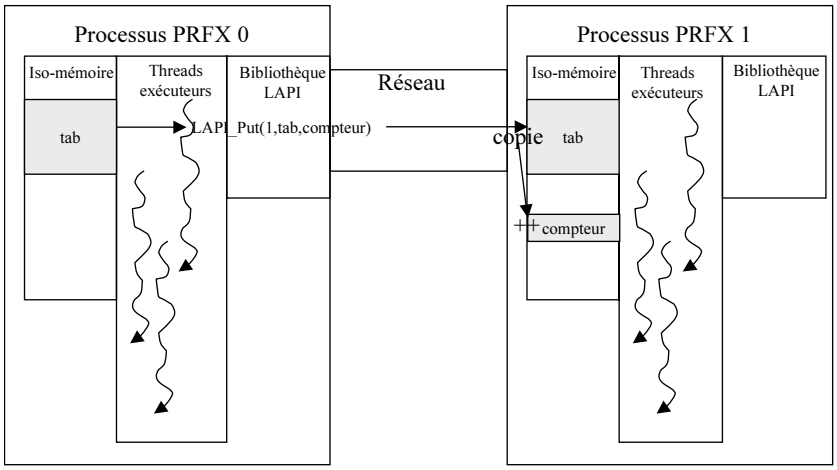

FIG. 14 – Utilisation de LAPI par PRFX.

le processeur est relâché pour d'autres *threads* (e.g. ceux de la bibliothèque de communication). Dans le second mode, le processeur est monopolisé par la sommation continuelle des compteurs.

3.4.3 Utilisation du DAG de tâches à l'exécution

Pour l'exécution, nous dupliquons le DAG de tâches dans chaque mémoire normale (tas) des processus PRFX. Nous ne mettons en iso-mémoire, qui est plus précieuse car bornée, que les parties du DAG devant être accédées de façon distante. Ces parties concernent les compteurs de dépendances pour les *broadcasts*, les compteurs associés aux tâches utilisateurs, les zones de stockage des pointeurs des arguments des tâches utilisateurs. Ces derniers sont en nombre fixe, ils peuvent être réutilisés si les dépendances entre les tâches assurent l'absence de conflits d'accès. Ce n'est pas toujours possible : par exemple, les codes des exemples que nous présentons dans cette thèse sont écrits avec un contrôle centralisé car cette approche améliore la lisibilité. De cette façon, toutes les tâches reçoivent leurs paramètres dans un laps de temps très court, si bien que la réutilisation des zones d'iso-mémoire dédiées aux arguments est impossible.

3.4.3.1 Tâches utilisateurs

Lorsque la tâche utilisateur passe dans l'état prêt et qu'un *exécuteur* est en attente pour exécuter cette tâche, alors il procède à d'ultimes préparatifs avant d'appeler la fonction de tâche. Au cours de ces préparatifs, il fait passer l'état de la tâche de prêt à l'état marche. Ensuite, il associe les pointeurs de *stencils*, reçus via la dépendance de contrôle, aux pointeurs de pointeurs d'arguments dans sa pile d'appel. Les n arguments du RPC (de la taille d'un pointeur) sont recopiés depuis l'iso-mémoire par le *thread* exécuteur dans sa propre pile. Ces deux opérations sont effectuées par ce dernier car elles ne peuvent être faites de façon valide par la tâche de préparation. En effet, supposons que la tâche de préparation soit ordonnancée sur le *thread* p et la tâche utilisateur sur le *thread* u, alors si p est différent de u alors le *thread* p devrait faire un effet de bord

sur la pile du *thread u*. Ce fonctionnement provoquerait dans ce cas des anomalies d'exécution du *thread u*.

Les tâches de contrôle sont débloquées au fur et à mesure des appels aux `PRFX_call()` correspondants effectués par la tâche utilisateur. Ces appels sont réellement implémentés de façon asynchrone. Enfin, la tâche utilisateur se termine et passe dans l'état terminé.

Une tâche utilisateur a des caractéristiques prédéfinies et un contexte pour l'exécution. Lors de l'exécution parallèle, seuls les arguments sont pris en compte. Le placement et la phase qui permet de séparer par exemple les étapes (inititalisation des données, calcul, sauvegarde des résultats) du programme, sont une information pour l'ordonnanceur. En effet, le *thread* exécuteur en charge de la tâche ne recalcule pas d'après la numérotation et la distribution spécifiée par l'utilisateur le numéro du *thread* exécuteur destination. Ce calcul a déjà été fait à l'inspection et éventuellement remis en cause à l'ordonnancement : le résultat est donc déjà stocké dans le DAG complet.

3.4.3.2 Tâches internes

Les tâches internes sont implémentées "en dur" dans l'exécuteur pour économiser le coût des appels effectués via des pointeurs de fonctions. Cette optimisation est utile car elle influe sur la latence entre l'exécution de deux tâches utilisateurs. Les moments où ces tâches internes s'exécutent peuvent être enregistrés par le système de traces proposé par PRFX.

Tâches gérant le contrôle utilisateur Pour chaque appel à `PRFX_call()` effectué par une tâche utilisateur, une tâche de contrôle est prévue statiquement lors de l'inspection. Ces tâches de contrôle sont dans une liste à laquelle le *thread* exécuteur se référera pour savoir quelle est la tâche cible correspondante. Cette liste permet au *thread* exécuteur de suivre le déroulement du programme dans le DAG en se positionnant sur la tâche courante. Pour un appel à `PRFX_call()`, la tâche de contrôle a deux comportements différents selon que la tâche cible est ordonnancée sur le même processus ou non. En intra-processus, l'appel à `PRFX_call()` effectue la recopie des *n* pointeurs d'arguments de la fonction et des *stencils* associés dans le tableau en iso-mémoire réservé à cet effet et, immédiatement après, il incrémente le compteur local de la tâche cible. Ces opérations allongent très peu l'exécution de la tâche utilisateur car elles sont simples et non bloquantes. Elles sont également suffisantes ce qui permet de désactiver les tâches internes de contrôle.

En extra-processus, seuls les arguments passés en paramètres du `PRFX_call()` et les *stencils* associés sont recopiés dans la zone iso-mémoire réservée à cet effet. Ensuite, la tâche de contrôle est activée. Lorsque vient son tour, elle calcule le processus destination d'après le numéro de *thread* exécuteur de la tâche cible. La suite consiste alors en une communication unilatérale opérant sur le tableau (en iso-mémoire) contenant les *n* pointeurs d'arguments et de *stencils* vers le processus dont le numéro a été précédemment calculé.

La figure 15 présente l'enchaînement des traitements provoqués lors de l'exécution parallèle de la tâche utilisateur numéro 0 (cf. en haut à gauche de la figure). La légende de ce schéma est la même que celle de la figure 9.

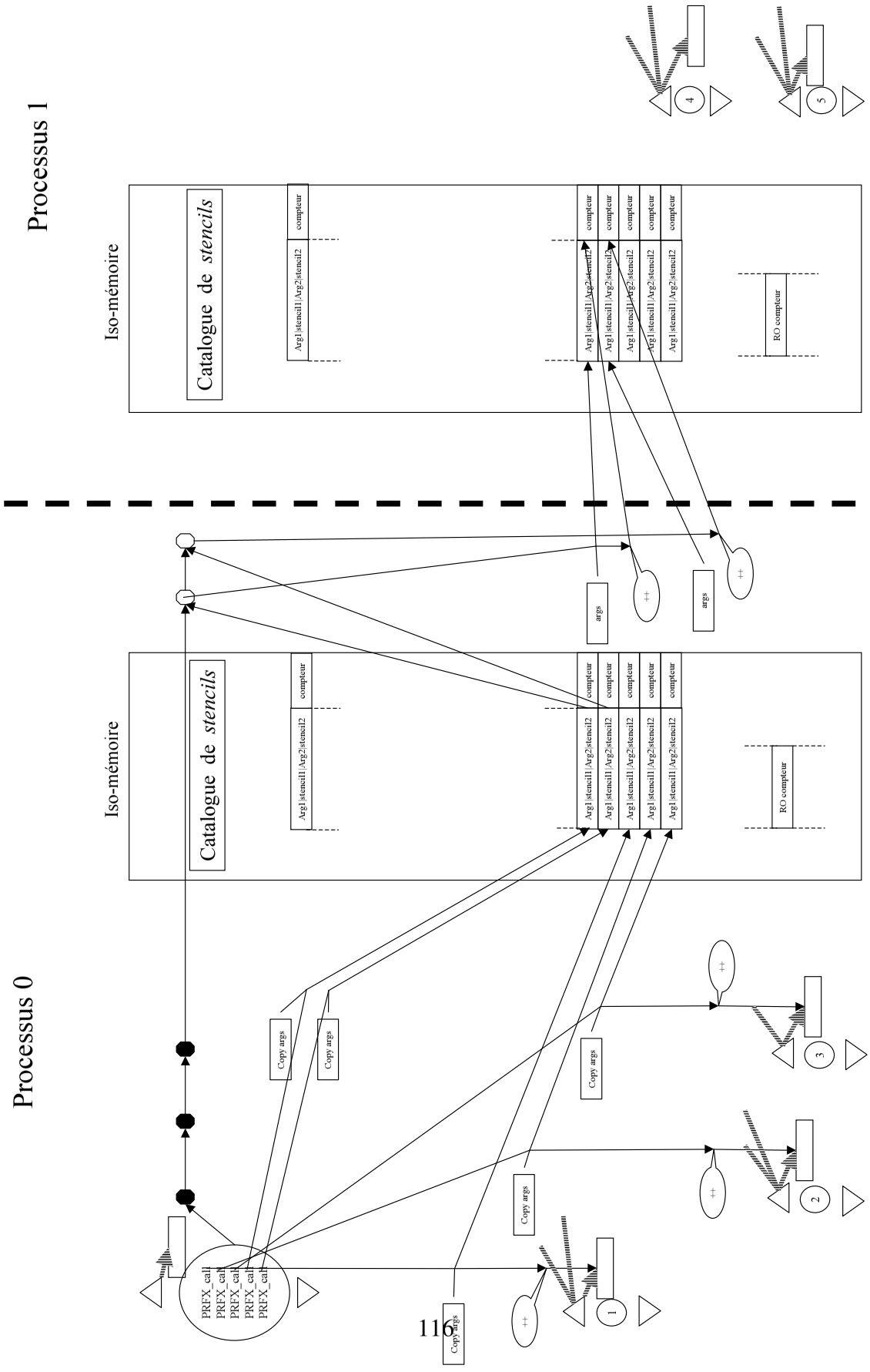

FIG. 15 – Actions des tâches de contrôle.

La tâche 0 dans laquelle sont entourés cinq `PRFX_call()` effectue cinq appels de tâches. Ces tâches sont supposées placées comme indiqué, à savoir les tâches 0, 1, 2, 3 sur le processus 0 et les tâches 4 et 5 sur le processus 1. Les trois premiers appels de tâche sont en local, tous les traitements sont donc faits séquentiellement dans le contexte d'exécution de la tâche initiatrice. De fait, les trois tâches de contrôle prévues en cas d'appel délocalisé peuvent être annulées (disques noirs en haut du schéma). Les pointeurs d'arguments et de *stencils* sont récupérés et copiés en iso-mémoire locale dans un emplacement préalablement convenu lors de l'inspection comme indiqué par la flèche allant des `PRFX_call()` vers l'iso-mémoire. Après quoi, le compteur de synchronisation de chacune des trois tâches locales est incrémenté (sigle de la bulle).

Les deux appels de tâches suivants sont délocalisés. Le `PRFX_call()` dans la tâche utilisateur n'a besoin que de copier les pointeurs d'arguments et de *stencils* en iso-mémoire et d'activer la tâche de contrôle associée à cet appel de tâche. Ces deux tâches de contrôle s'exécuteront quand elles seront atteintes dans le vecteur d'ordonnancement d'un *thread* exécuteur. A ce moment là, elles communiqueront de façon unilatérale la zone iso-mémoire des pointeurs d'arguments et *stencils* vers l'iso-mémoire du processus 1. Le compteur global des tâches utilisateur ainsi appelées est incrémenté après l'arrivée de ce message.

Tâches de pré et post traitement : Lors de l'exécution, la tâche préparatoire scrute les deux compteurs (local et global) de sa tâche utilisateur associée, ainsi que les compteurs de *broadcast* dont cette dernière est la cible. Lorsque la somme de ces compteurs est égale au nombre de dépendances requises par la tâche utilisateur, la pré-tâche fait passer dans l'état "prêt" sa tâche utilisateur associée et se termine. La scrutation est paramétrable selon les deux modes d'attente active ou non.

La tâche de post-traitement scrute l'état de sa tâche utilisateur associée. Dès qu'elle est terminée, la tâche de libération procède à la libération mémoire de la tâche.

Tâches de communication de données parallèles : Toutes les tâches de communication de données parallèles provoquent l'incrémentation d'un compteur à destination lorsque la donnée est arrivée. Cette fonctionnalité est intégrée à la couche de communication LAPI mais non à celle de ShMem ou de MPI-2. Lorsque ces dernières bibliothèques sont utilisées, la communication est effectuée en trois étapes : l'envoi du message, le *flush* du message puis l'incrémentation d'un compteur.

Nous avons également implémenté une version utilisant les écritures dans les fenêtres distantes MPI-2. L'interface MPI-2 est inadaptée aux communications unilatérales telles que nous voulons les utiliser.

Contrairement aux tâches de contrôle, les tâches de communication n'ont aucun lien direct avec l'interface de PRFX. Par contre, elles ont la même dualité de comportement selon que la tâche cible est locale au processus ou non. Les tâches de communication scrutent l'état de leur tâche utilisateur associée. Dès que cette dernière passe dans l'état "terminé", elles entrent en action sous les deux formes que nous avons déjà mentionnées à la section 2.4.2.3 :

– Envoi de données parallèles agrégées : Les envois de données agrégées factorisent la latence. De plus, en augmentant la taille du message, la bande passante se rapproche de la

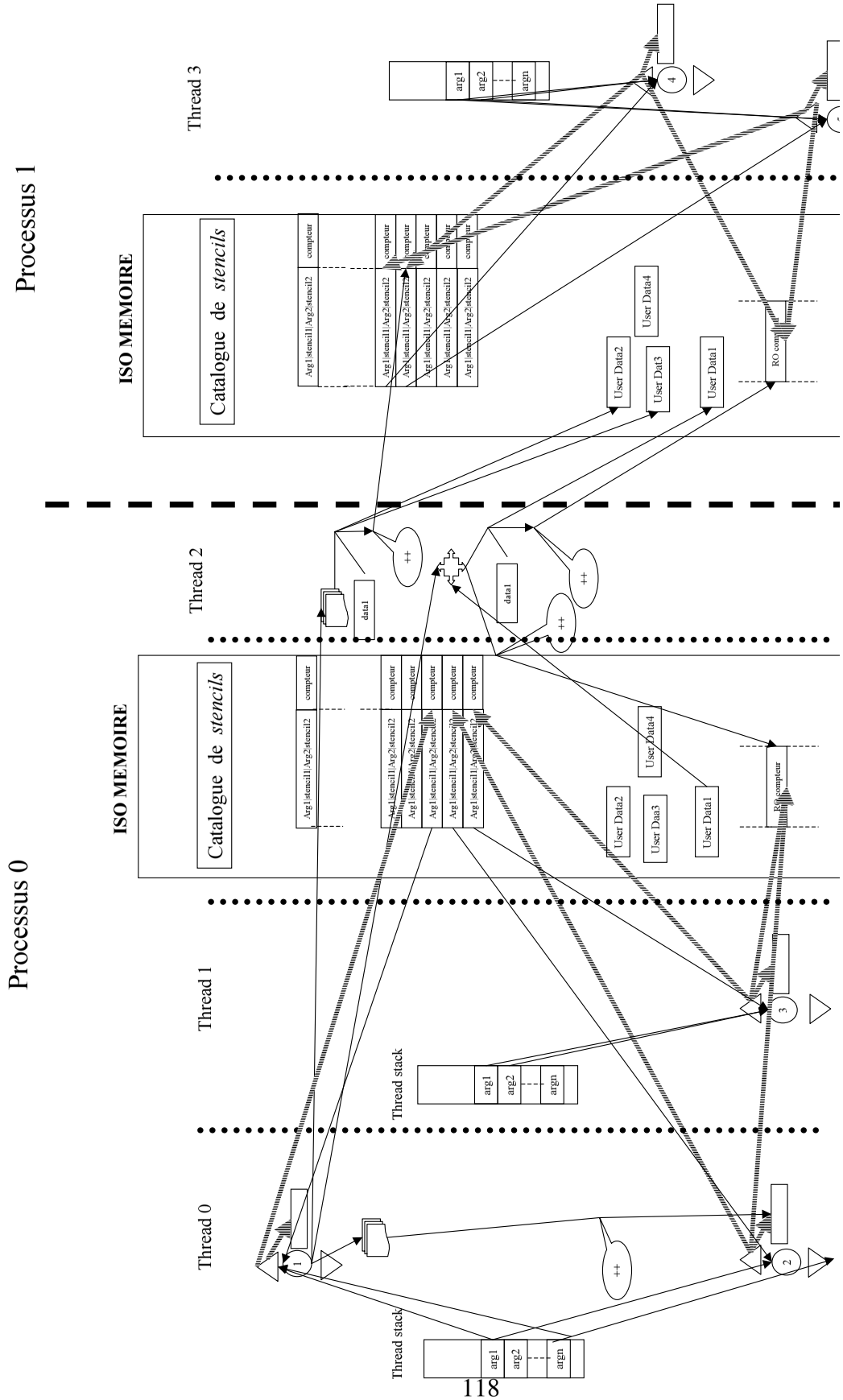

FIG. 16 – Actions des tâches de communication.

limite imposée par le matériel.
La tâche s'occupant des dépendances portant des données agrégées contient dans sa structure les informations sur le numéro de la tâche de destination ainsi que la liste des *stencils* et la manière de les appliquer en iso-mémoire relativement aux données de la tâche associée. Le numéro du processus destination est calculé d'après le numéro de *thread* exécuteur de la tâche destination. Si cette dernière est locale au processus, alors le compteur local de la tâche est incrémenté. Sinon, les données correspondant à l'application des *stencils* sont envoyées de façon unilatérale au processus distant.
L'implémentation de PRFX au-dessus de LAPI utilise une de ses fonctionnalités, proposée par de nombreuses bibliothèques de communication, qui permet d'envoyer une série de vecteurs de données décrits par des plages d'adresses. Sous ShMem, cette fonctionnalité n'est pas disponible, et l'implémentation est donc effectuée par une succession de *put* vers le même destinataire.

– *Broadcast* de données parallèles : Le *broadcast* d'un message est implémenté sous la forme de *n* envois de ce message en séquentiel vers les *n* processus le requérant, suivi d'une incrémentation d'un compteur unique par processus. Les envois s'effectuent par processus et non par tâche pour éviter la redondance des envois vers un même processus détenteur de plusieurs tâches visées par un *broadcast*. L'unicité du compteur, quant à lui, permet le signalement simultané au sein d'un processus de toutes les tâches visées.
La tâche de *broadcast* dispose de la liste des tâches cibles et de leurs numéros respectifs de *thread* exécuteur auquel elles sont affectées. Etant donné que cette tâche est en charge des appels à la bibliothèque de communication dont la durée dépend de phénomènes matériels asynchrones, nous commençons par assurer le déblocage des tâches locales. Pour cela, nous positionnons localement à 1 le compteur de dépendances associé à ce broadcast. Ensuite, nous établissons la liste des numéros de processus destination du *broadcast* d'après l'ensemble des numéros de *thread* exécuteur des tâches. Cette liste des processus destinataires est calculée à la volée pour qu'en cas de ré-ordonnancement dynamique, l'exécution reste valide.
La donnée à envoyer est, quant à elle, déterminée en appliquant le *stencil* associé à la tâche de communication relativement à la base indiquée. Enfin, la tâche de *broadcast* se termine.

La nouvelle version de LAPI propose une interface pour les *broadcasts* que nous n'avons pas encore intégrée. ShMem propose également une fonctionnalité de *broadcast* mais qui nécessite que tous les processus fassent appel à la procédure `shmem_bcast()`, ce qui ne permet pas de réaliser une communication unilatérale. Elle correspond aux besoins de MPI qui est implémenté au-dessus de ShMem sur IRIX.

La figure 16 présente la suite du déroulement de l'exécution de la figure 15 après la terminaison de la tâche de contrôle 0 (omise ici). Sur ce schéma, les affectations des tâches aux *threads* sont désormais indiquées. La tâche de préparation (de forme triangulaire) associée à la tâche 1 est lancée par le *thread* exécuteur 0. Elle scrute uniquement les deux compteurs (1 local + 1 global) de la tâche associée. Cette tâche ne référence pas de compteurs de *broadcast* car elle ne possède pas de précédences de ce type. La dépendance de contrôle étant satisfaite, la pré-tâche fait passer

la tâche utilisateur 1 dans l'état prêt. Le *thread* 0 fait donc l'appel vers la fonction de cette tâche après avoir recopié les arguments dans sa pile et enregistré les *stencils*. Ensuite, le code de l'utilisateur de la tâche 1 s'exécute et se termine sans avoir appelé la bibliothèque PRFX. La tâche passe dans l'état terminé ce qui fait passer simultanément dans l'état prêt la tâche de terminaison associée et les deux tâches de messages agrégés et de *broadcast*. Les tâches de terminaison et de messages agrégés locales sont exécutées toujours par le *thread* 0. Le message agrégé ne nécessite qu'une incrémentation car la dernière version de la donnée `data4` est déjà disponible localement. En revanche, lorsque le *thread* 2 exécute la deuxième tâche du message agrégé, c'est pour envoyer la dernière version de `data2` et `data3` à la tâche 4. Cet envoi se fait en un seul message de façon unilatérale en agrégeant les deux données à transmettre. Dès que ces deux données sont disponibles dans la mémoire du processus 1, le compteur global de la tâche 4 est incrémenté. Le *thread* 2 enchaîne ensuite sur la tâche de *broadcast* qui est dans l'état prêt. Cette tâche incrémente le compteur local RO associé à ce broadcast lors de l'inspection. Ensuite, elle construit la liste des processus atteints d'après le placement des tâches destination de ce *broadcast*. Cette liste comporte le processus 0 et le processus 1. Donc la donnée `data1` est envoyée de façon unilatérale en iso-adresse vers le processus 1. Le compteur RO distant est incrémenté au terme de cette communication. Ainsi, la pré-tâche de la tâche 4 voit ses dépendances satisfaites et donc ses compteurs incrémentés. La somme de ces derniers (compteur global = 2, local = 0, compteur de *broadcast* = 1) correspond aux nombres de dépendances requises. Le *thread* 3 lance donc la tâche 4 qui se termine, puis il lance la post-tâche associée à la tâche 4. Il effectue ensuite les mêmes opérations pour la tâche 5. Ceci termine l'exécution du processus 1. Le processus 0 se termine également après que les tâches 2 et 3 ainsi que leurs pré et post tâches se sont exécutées.

3.5 Conclusion

L'implémentation du support pour le mode d'expression de PRFX est effective (hormis la prise en compte des accès commutatifs). Ce support est uniquement dédié aux programmes prévisibles. Il est constitué de peu de couches logicielles. Il exploite une iso-mémoire, les *threads* POSIX et les communications unilatérales (LAPI principalement, ShMem et MPI-2 de façon dégradée) qui sont les technologies les plus performantes actuellement sur les grappes de nœuds SMP. Il permet de générer statiquement le DAG de tâche complet d'un programme PRFX, d'ordonnancer également statiquement les tâches de calcul et les tâches internes de ce DAG, et enfin de déployer les *threads* de la plate-forme selon le schéma indiqué par l'utilisateur tout en respectant l'ordonnancement des tâches et des communications lors de l'exécution parallèle.

Chapitre 4

Expérimentations

Les expérimentations ont été réalisées sur deux grappes de SMP IBM. La première est actuellement composée de 4 nœuds p690+ (32 processeurs POWER4 1700MHz) de dernière génération interconnectés par un switch *Federation*. La seconde, plus ancienne, compte 27 nœuds NH2 de 16 processeurs POWER3 (375MHz) reliés par un réseau *Colony*. Nous ferons référence à ces machines via les diminutifs p690 et NH2. Les nœuds p690+ comptent deux fois plus de processeurs POWER 4 ; chacun d'eux est théoriquement 4.5 fois plus puissant que les POWER 3. Le réseau *Federation* des p690+ a une bande passante quatre fois plus importante et une latence 30 % plus faible que le réseau *Colony*. Le niveau d'hétérogénéité entre les performances de la mémoire partagée et celles du réseau est plus marqué pour les p690.

La grappe p690 est exploitée par le système AIX 5.2 avec l'environnement PSSP 3.4 (Parallel System Support Programs) alors que la grappe NH2 est exploitée par le système AIX 5.1 et l'environnement PSSP 3.2. La bibliothèque de communication unilatérale LAPI utilisée est donc plus ancienne sur les nœuds NH2, elle ne permet pas d'exploiter toute la bande passante du réseau. L'implémentation de cette ancienne version ne respecte ni les spécifications d'asynchronisme ni celles d'unilatéralité et ni celles de ré-entrance mais elle est *thread-safe*. Une version entièrement re-écrite de LAPI existe, mais elle est uniquement installée sur les nœuds p690. Cette nouvelle version de la bibliothèque prend de l'importance car elle est utilisée par l'implémentation de MPI proposée par IBM ainsi que par le système de fichiers IBM GPFS. Cette version respecte les spécifications sur les réseaux utilisant le *switch* IBM *Federation* grâce à ses fonctionnalités matérielles (capacité d'adressage de la mémoire distante).

Les conditions expérimentales sont nombreuses, elles visent à faire tendre les caractéristiques de la machine vers celles supposées par l'ordonnanceur. Les machines sont donc réservées dans un mode dédié à l'application. L'iso-mémoire consommant beaucoup d'espace d'adressage, nous indiquons au système AIX qu'il doit allouer effectivement la mémoire demandée lors de la saturation effectuée par l'iso-allocateur (positionnement de variables d'environnement `PS_ALLOC=early` et `PS_DISCLAIM=no`). L'iso-allocateur utilise `malloc()` de la libC (réentrante) fournie avec le système AIX. L'ordonnanceur affecte toutes les communications à un seul *thread* exécuteur par processus PRFX pour que les tâches traitées par les autres *threads* exécuteurs du processus aient un service de communication non bloquant. Ces *threads* exécuteurs

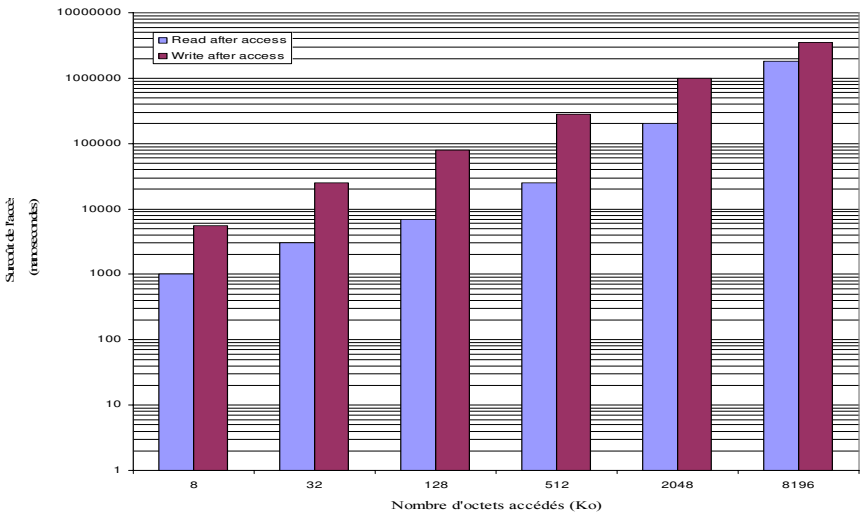

FIG. 17 – Surcoût de la cohérence matérielle pour les nœuds IBM NH2 en fonction du volume de données accédées.

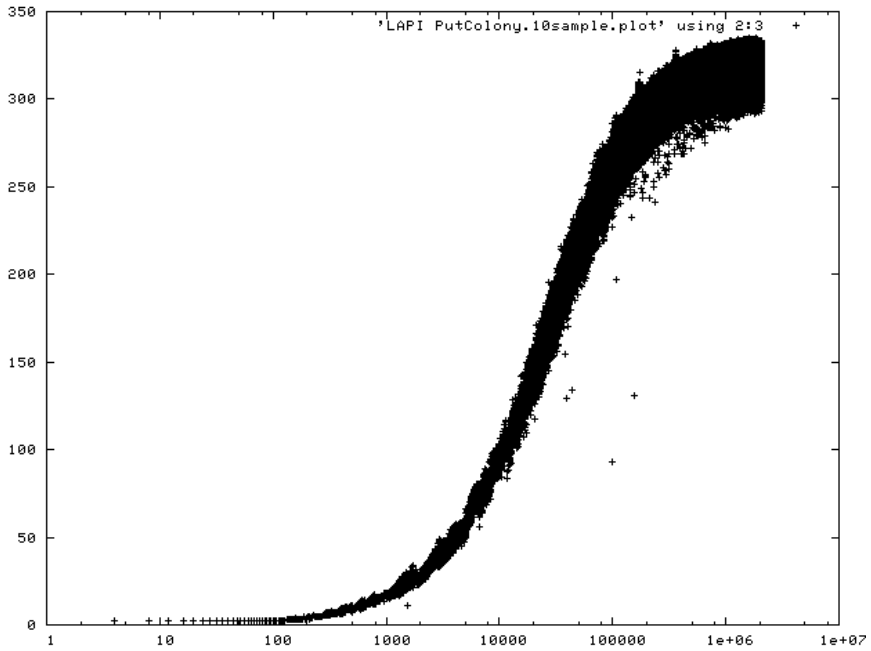

FIG. 18 – Bande passante du réseau IBM *Colony* en fonction de la taille du message envoyé via `LAPI_Put()`.

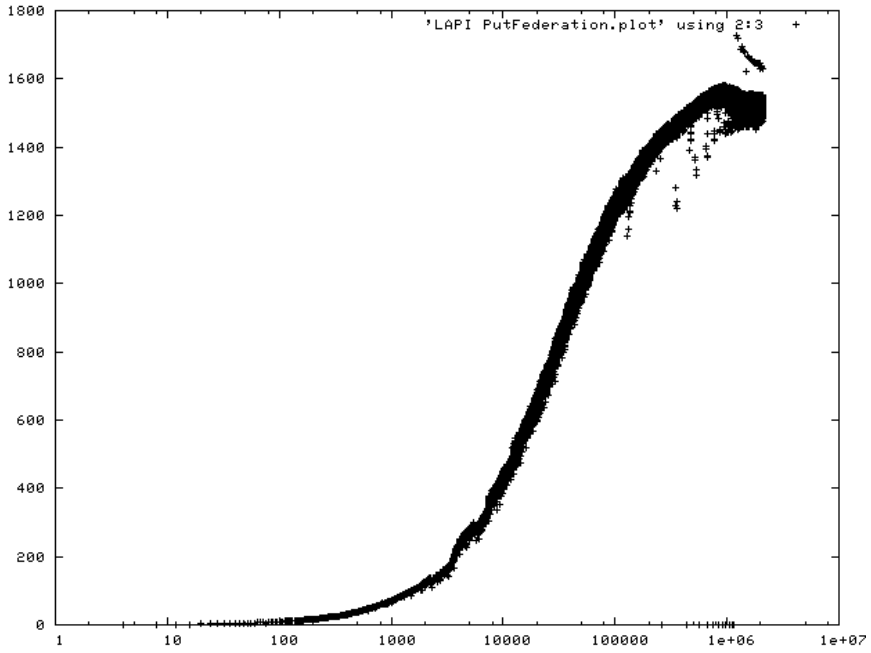

FIG. 19 – Bande passante du réseau IBM *Federation* en fonction de la taille du message envoyé via `LAPI_Put()`.

traitent les pré-tâches, les tâches utilisateurs et les post-tâches. Les algorithmes sont découpés en quatre phases en valuant le deuxième paramètre des `PRFX_call()` :

- phase d'initialisation,
- phase de contrôle utilisateur,
- phase de calcul,
- phase de vérification du calcul.

Une fois la validité de notre exécution assurée, la phase de vérification (consommatrice de ressources) est retirée pour effectuer des prises de temps sur la phase de calcul.

Pour les NH2, la configuration d'exécution de chaque processus est de 19 *threads* :

- 1 *thread* `poe`,
- 2 *threads* LAPI,
- 16 *threads* exécuteurs PRFX.

Dans le cas des p690, chaque processus comporte 35 *threads* :

- 1 *thread* `poe`,
- 2 *threads* LAPI,
- 32 *threads* exécuteurs PRFX.

Le *thread* `poe` gère les signaux et les entrées-sorties standards des processus ; nous le rendons peu intrusif en ne récupérant que la sortie standard du processus 0 et en désactivant une vérification de fonctionnement (via les variables `MP_STDOUTMODE=0, MP_PULSE=0`). Ce *thread* n'est pas fixé sur un processeur.

La bibliothèque LAPI est utilisée dans un mode de scrutation toutes les 1000 microsecondes (`MP_POLLING_INTERVAL=1000`). Les deux *threads* LAPI sont affectés via le déploiement au

processeur 0, tout comme le *thread* exécuteur en charge des tâches de communication. Ainsi, les opérations de synchronisation entre ces trois *threads* POSIX, nécessitées par la bibliothèque LAPI, sont locales à un même processeur. Seul un des deux *threads* LAPI est sollicité par PRFX pour effectuer les communications unilatérales (`LAPI_Put()`). Le *thread* inutilisé est en charge de fonctionnalités (signaux de complétion de messages, messages actifs) dont PRFX ne fait pas usage, il est donc inactif. Nous améliorons la performance des synchronisations en positionnant les variables : `AIXTHREAD_COND_DEBUG=OFF, AIXTHREAD_MUTEX_DEBUG=OFF, AIXTHREAD_RWLOCK_DEBUG=OFF, RT_GRQ=OFF, AIXTHREAD_SCOPE=S, AIXTHREAD_MINKTHREADS=19, MP_SINGLE_THREAD=yes`. Les autres *threads* exécuteurs sont attachés au processeur spécifié dans le fichier de déploiement.

Les *threads* exploitent la mémoire partagée dont la cohérence est assurée en intra-nœud directement par le matériel. Les coûts associés à cette cohérence sont indiqués par l'histogramme de la figure 17. L'axe des ordonnées correspond au surcoût en nano secondes lié à la gestion de la cohérence mémoire et l'axe des abscisses présente deux cas de cohérence pour chacune des tailles de données accédées. Ces deux cas sont la lecture d'une donnée (*read after access*) ou son écriture (*write after access*) par une carte C_1 de processeurs du nœud après qu'elle a été accédée (i.e indifféremment lue ou écrite) par une autre carte C_2 de processeurs du même nœud. L'opération de lecture provoque la remontée de la donnée dans les caches L2 et L1 de la carte C_1. L'opération d'écriture est plus coûteuse car elle nécessite, en plus, d'invalider les caches L1 et L2 de la carte C_2.

La cohérence mémoire est assurée par PRFX en extra-nœud via la bibliothèque LAPI dont les bandes passantes sont indiquées par les nuages de points des figures 18 et 19. La bande passante (en ordonnée) a été mesurée sur un réseau IBM *Colony* entre deux nœuds NH2 et sur un réseau *Federation* entre deux nœuds p690. Les tailles de message considérées (en abscisse) vont par pas de 4 octets de 0 à 2 Mo selon une échelle logarithmique. Ces informations servent à l'ordonnanceur pour calculer les dates d'exécution des tâches.

L'implémentation des deux algorithmes (Jacobi et LU) qui vont suivre ont fait l'objet de deux publications [21, 20] sur des tailles de problème différentes de celles présentées ici. Les résultats obtenus pour le troisième algorithme (Cholesky) appliqué à des matrices creuses ont également été publiés [22]. Ce dernier papier présente, par rapport au précédent, les nouvelles fonctionnalités (ordonnancement par phase, ordonnancement des communications et accès commutatifs) pour capturer davantage de savoir-faire du programmeur afin d'obtenir des performances pour les algorithmes irréguliers.

4.1 Algorithme de Jacobi

Comme nous l'avons vu à la section 2.8.2.1, l'algorithme de Jacobi s'écrit très simplement avec le mode d'expression PRFX. Pour obtenir de meilleures performances, nous modifions cet algorithme de sorte à économiser la recopie du maillage à chaque itération. Cette modification consiste à faire deux itérations de l'algorithme au lieu d'une à chaque tour de boucle. En effet,

plutôt que de recopier directement les éléments du maillage temporaire vers le maillage principal, nous inversons les rôles de ces maillages : le maillage temporaire devient principal et inversement. Ceci nous permet d'effectuer une deuxième itération à la place de la copie.

Trois maillages, de tailles 126000×646, 126000×6460 et 126000×64600 sont utilisés. Leurs occupations en iso-mémoire sont respectivement de 70 Mo, 700 Mo et 7 Go. Pour être traités en parallèle, ces maillages sont découpés en autant de blocs de lignes que de processeurs à exploiter.

Ces blocs sont distribués avec l'option `PRFX_DISTRIB_BLOCK_BLOCK1D` (cf. section 2.3.3.1) de la fonction `PRFX_thrDistributionSetDefault()`. Ceci permet de faire traiter un paquet contigu de blocs (super bloc) sur les processeurs de chaque nœud SMP. Ainsi, un processus PRFX donné n'a que deux messages extra-nœuds à envoyer à chaque itération de l'algorithme. Ces deux messages correspondent aux deux frontières du super bloc. Ce nombre de messages extra-nœud est donc linéaire par rapport au nombre de nœuds SMP et d'itérations de l'algorithme.

Dans le cas de notre implémentation, la taille du problème n'influe donc pas sur le nombre de tâches du DAG associé. L'exécution de cet algorithme est modélisée par un DAG complet contenant un nombre linéaire de tâches par rapport au nombre de processeurs utilisés et au nombre d'itérations effectuées (ici 1000). Ce DAG est ordonnancé avec l'heuristique du chemin critique, la restriction à un seul *thread* candidat et le respect des phases (cf. section 2.5).

Le graphique de la figure 20 montre les coûts de l'inspection, de l'ordonnancement (schedCP) et de l'exécution (jacobi) sur des p690 pour les trois problèmes de taille croissante (de gauche à droite). Pour chacun de ces problèmes, les coûts (en ordonnée avec échelle logarithmique) sont reportés pour des déploiements (en abscisses) sur 1, 2, 3 et 4 nœuds. Le graphique de la figure 21 est similaire, les nœuds considérés sont des NH2 et ils sont en plus grand nombre pour chaque problème (jusqu'à 6 nœuds). A nombre de nœuds identique, ces coûts d'inspection et d'ordonnancement sont constants quelle que soit la taille du problème. Ces coûts sont faibles par rapport à l'exécution parallèle ; ils peuvent donc être rapidement amortis.

Le *speedup* (cf. figure 22) de l'algorithme implémenté croît de façon linéaire avec le nombre de nœuds NH2 utilisés. Le *speedup* est également linéaire pour les p690 mais de croissance moins rapide. Ces deux scalabilités sont bonnes car les transferts de données extra-nœuds sont peu nombreux.

La courbe de rendement présentée par la figure 23 est exprimée en nombre de points du maillage traités par seconde (en ordonnées) en fonction du nombre de nœuds utilisés et ceci pour les trois tailles de problème considérées. Ce choix du nombre de points par seconde est moins sujet à variation que le nombre d'opérations qui conduit à des résultats incohérents selon les optimisations du compilateur C. Dans le cas des p690, le *thread* exécuteur auquel l'ordonnanceur a affecté toute la gestion des synchronisations est surchargé. En effet, il doit réaliser toutes les tâches internes correspondant aux tâches utilisateurs que les 31 autres *threads* exécuteurs traitent. Les temps d'exécution de ces tâches internes deviennent plus importants lorsque les communications transitent via le réseau. Ceci explique la plus faible performance obtenue lorsque davantage de nœuds sont utilisés (cf. figure 23). Pour les NH2, le rendement décroît légèrement au fur et à mesure que davantage de nœuds sont utilisés. En effet, l'augmentation du coût des tâches de communication ne conduit pas à des retards importants du *thread* exécuteur en charge des tâches

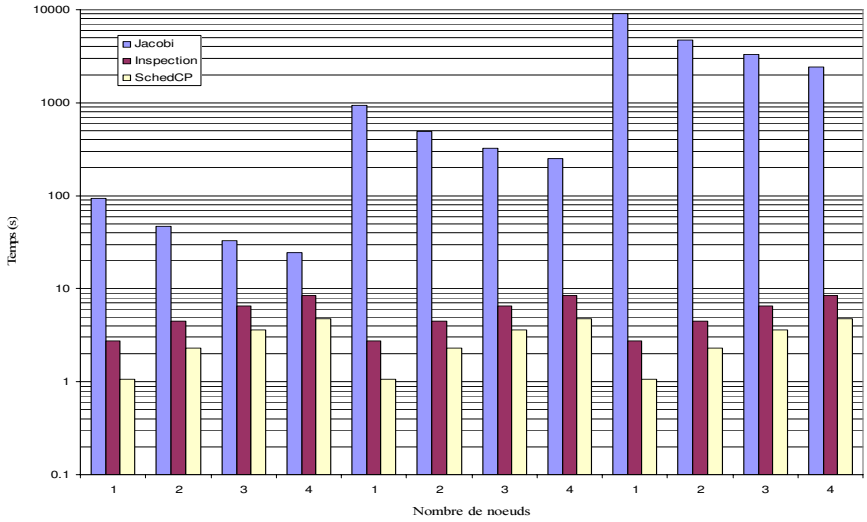

FIG. 20 – Temps d'exécution, d'inspection et d'ordonnancement sur des nœuds IBM p690 pour trois maillages de largeur 646, 6460 et 64600 et de hauteur 126000.

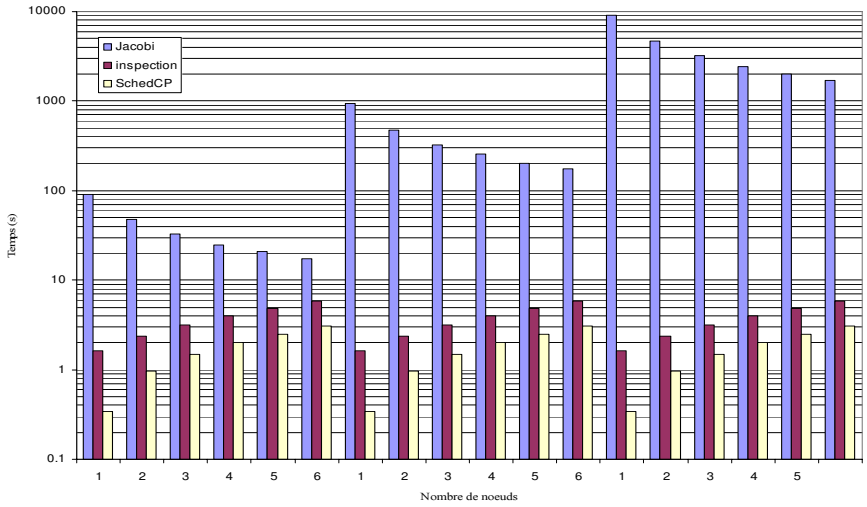

FIG. 21 – Temps d'exécution, d'inspection et d'ordonnancement sur des nœuds IBM NH2 pour trois maillages de largeur 646, 6460 et 64600 et de hauteur 126000.

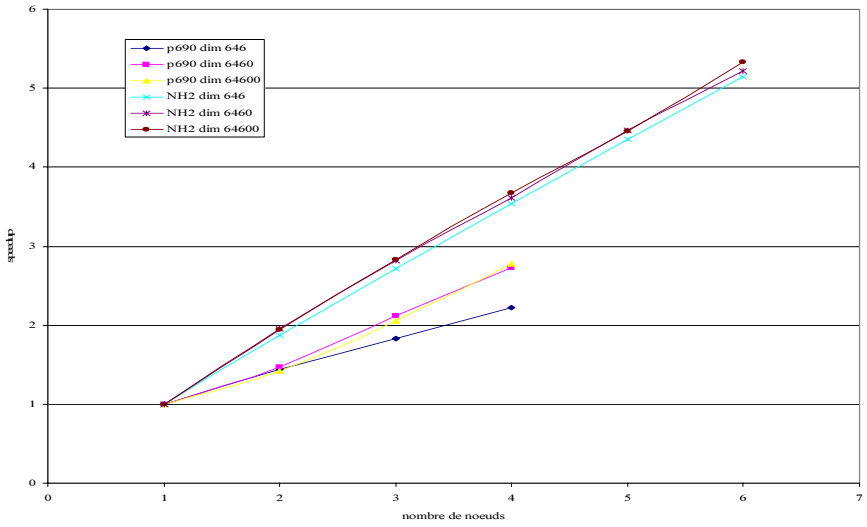

FIG. 22 – Scalabilité pour des nœuds IBM NH2 et p690 de l'implémentation de l'algorithme de Jacobi pour 3 maillages de largeur 646, 6460 et 64600 et de hauteur 126000.

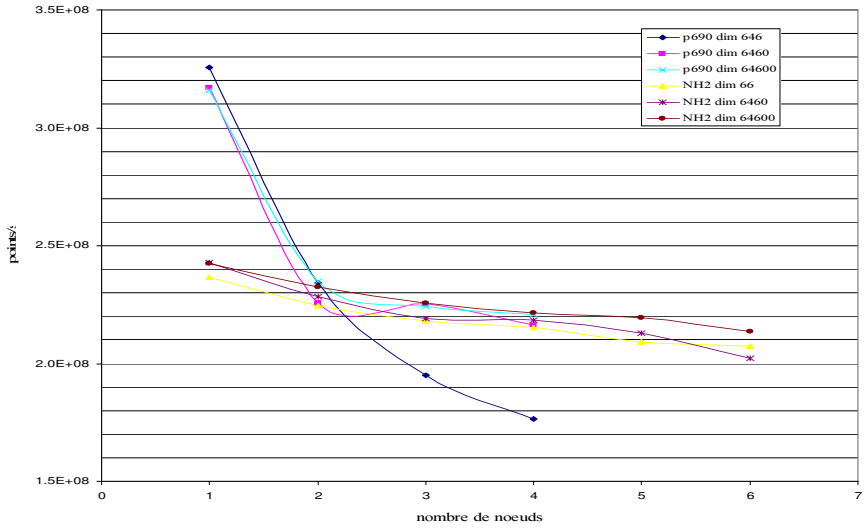

FIG. 23 – Rendement pour des nœuds IBM NH2 et p690 de l'implémentation de l'algorithme de Jacobi pour 3 maillages de largeur 646, 6460 et 64600 et de hauteur 126000.

internes de PRFX.

Cet exemple est un algorithme générant beaucoup de parallélisme avec un DAG ayant une structure régulière et un faible nombre de tâches de calcul. La majorité des tâches de communication est gérée par la cohérence matérielle intra-nœud, les performances obtenues avec PRFX sont donc bonnes. Nous allons étudier un cas moins favorable dans la section qui suit où ce nombre de tâches doit dépendre de la taille du problème traité pour conserver suffisamment de parallélisme.

4.2 Factorisation LU de matrices denses sans pivotage

Nous considérons la version par bloc 2D de la factorisation LU de matrices décrit par l'algorithme 2. Nous supposons ici que la matrice est découpée en $N \times N$ blocs. Des opérateurs BLAS3 correspondent aux opérations nécessitées par les lignes 4 et 5 (`trsm`) ainsi que par la ligne 9 (`gemm`). Ces opérateurs sont paramétrés par la taille des blocs qu'ils traitent. Cet algorithme classique de factorisation LU peut avoir des propriétés imprévisibles (pivotage) que nous n'avons pas considérées ici pour répondre aux contraintes de notre support. L'algorithme étant alors prévisible, nous pouvons construire son DAG de tâches, l'ordonnancer et l'exécuter avec PRFX.

Algorithme 2 Algorithme de factorisation LU de matrices denses (version par bloc).

Entrée Une matrice A de $N \times N$ blocs inversible bien conditionnée
Sortie La matrice A modifiée où L et U sont respectivement les parties triangulaires inférieures et supérieures telles que $A = LU$

1: **pour** k de 1 à N **faire**
2: factoriser A_{kk} en $L_{kk}U_{kk}$, L_{kk} et U_{kk} écrasent A_{kk}
3: **pour** i de $k+1$ à N **faire**
4: résoudre $L_{kk}U_{ki} = A_{ki}$ où U_{ki} écrase A_{ki}
5: résoudre $U_{kk}^t L_{ik}^t = A_{ik}^t$ où L_{ik} écrase A_{ik}
6: **fin pour**
7: **pour** i de $k+1$ à N **faire**
8: **pour** j de $k+1$ à N **faire**
9: $A_{ij} \leftarrow A_{ij} - L_{ik}U_{kj}$
10: **fin pour**
11: **fin pour**
12: **fin pour**

Pour cet algorithme LU, nous avons choisi de nous intéresser à la résolution d'une matrice de taille 10080×10080 stockée par blocs ; son stockage nécessite environ 800 Mo de mémoire. Cette dimension de matrice a la propriété d'être multiple de 84, 90, 96, 105, 112, 120, 126, 140, 144, 160, 168, 180, 210, 224, 240, 252 ,280, 288, 315 et 336. Ces vingts nombres seront les différentes largeurs de blocs que nous utiliserons pour factoriser cette matrice ($N = 10080/$

largeur de bloc). Ainsi, pour un problème de taille donnée, nous allons observer les variations induites par les changements de granularité des calculs. Chaque point des courbes présentées pour cet exemple a été calculé à partir du meilleur temps d'exécution parmi 5 mesures.

Le DAG de tâches est ici relativement dense, il contient $O(N^3)$ tâches. Dans cette version où le découpage est bi-dimensionnel, le nombre de tâches utilisateurs correspond au nombre de fois où les opérateurs BLAS sont utilisés : $2/3N^3 - 1/2N^2$. Le nombre de tâches internes de gestion est du même ordre car le degré moyen d'une tâche utilisateur est 2. Le nombre total de tâches (internes et utilisateurs confondues) va respectivement de 1800000 à 30000 pour les largeurs de blocs précédemment citées.

Les deux machines (p690 et NH2) du banc d'essais ont une configuration d'exécution comportant P processus PRFX et leurs *threads* exécuteurs. Le déploiement épouse leur hiérarchie avec 1 processus par nœud et 1 *thread* exécuteur par processeur.

Leur hiérarchie est également exploitée avec une version SMP de la distribution cyclique (`PRFX_DISTRIB_CYCLIC2D_SMP`, cf. section 2.3.3.1) classique pour cet algorithme. Ce choix de distribution permet d'obtenir de bonnes performances. Les dimensions de la grille de processus dépendent du nombre de processus utilisés. Cette grille est respectivement de dimensions 1×1, 1×2, 1×3, 2×2, 1×5, 2×3, 1×7, 2×4, 3×3 pour des déploiements de 1 à 9 processus. La sous-grille de *threads* pour chaque processus est de taille 5×6 (=30 *threads*) pour les nœuds p690 et de taille 3×5 (=15 *threads*) pour les nœuds NH2. Dans le cas des p690, nous préférons faire travailler l'ordonnanceur avec 30 *threads* exécuteurs de BLAS et 1 *thread* exécuteur de communications plutôt qu'avec 31 *threads* exécuteurs de BLAS et 1 *thread* exécuteur de communications car 31 est un nombre premier, ce qui conduit à des grilles de *threads* inadaptées au problème. Un *thread* exécuteur par nœud ne sera donc jamais chargé de traiter des tâches utilisateurs.

Les RPC sont numérotés $i*N+j$ si A_{ij} est le bloc dans lequel ils écrivent, de façon à obtenir l'adéquation entre la distribution choisie pour les RPC et la distribution des blocs de données. La distribution des données tient donc compte de la hiérarchie physique de la machine. Cette distribution assure un équilibrage de charge d'autant meilleur que le nombre de blocs est grandement supérieur au nombre de *threads* exécuteurs. D'autre part, elle permet de réduire les volumes de communications extra-processus.

Les temps pris par les tâches d'inititalisation d'un bloc et par celles effectuant les calculs ont été mesurés pour les largeurs de blocs utilisées. Ces mesures sont faites dans un cadre idéal où les blocs sont dans les caches. La majeure partie du temps de calcul de l'algorithme est passée dans l'exécution de la routine BLAS `dgemm()`. Nous avons constaté, en séquentiel, que le rendement de cet opérateur BLAS était stable pour les tailles de blocs utilisées. Ce rendement est d'environ 1.25 +/-0.05 GFlops pour les POWER3 et 3.95 +/- 0.05 Gflops pour les POWER4. Les autres RPC correspondent au contrôle, ils sont indiqués ici par l'utilisateur comme étant d'une durée constante pour des raisons expliquées ci-après. Ces prévisions de la durée des tâches sont fournies à l'ordonnanceur via des fonctions de coût de l'utilisateur. Elles permettront aussi de calculer un temps prévisionnel directement sur le DAG que l'on pourra comparer avec le temps réel d'exécution (cf. fin du paragraphe sur l'étude du rendement).

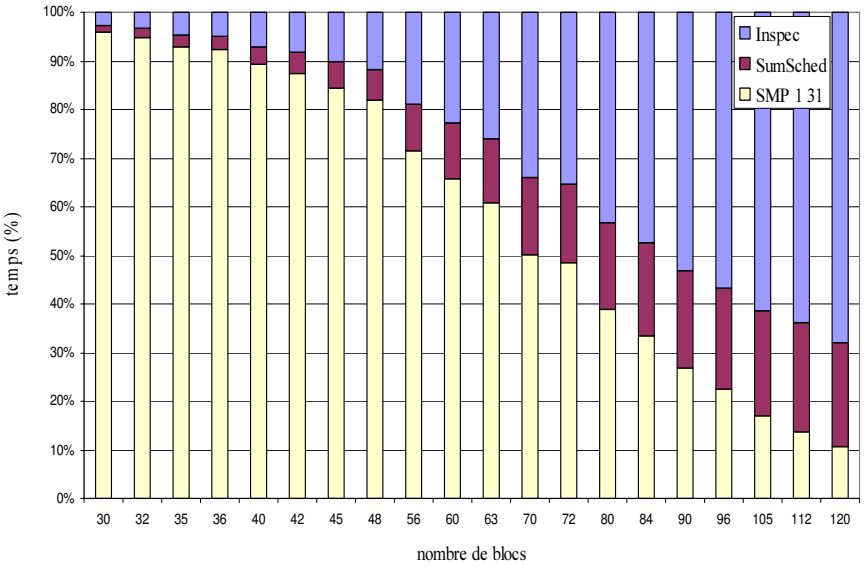

FIG. 24 – Durées relatives sur un p690 de l'inspection, de l'ordonnancement PRFX et de la phase parallèle de l'algorithme LU pour différents nombres de blocs (30 à 120 blocs pour une matrice 10080×10080).

L'ordonnancement est celui du chemin critique à choix contraint à un seul processeur candidat et respectant les phases (cf. section 2.5). Après application de l'ordonnancement (qui ne remet pas en cause la distribution choisie), le nombre de communications extra-processus (via LAPI) est de l'ordre de $O(NP)$ (avec P le nombre de processus) et le nombre de communications intra-processus (via la cohérence mémoire matérielle) est de l'ordre de $O(N^3)$.

Dans la figure 24, nous présentons, en ordonnée, les coûts relatifs d'inspection (inspec), d'ordonnancement (Sum Sched) et d'exécution parallèle sur 1 nœud p690 avec 31 *threads* exécuteurs (SMP 1 31) pour une granularité de calcul de plus en plus fine en abscisse. Les mesures de temps ont été effectuées sur un seul nœud p690 aussi bien pour les pré-traitements séquentiels (inspection et ordonnancement) que pour la phase parallèle. Nous ne comptabilisons pas le temps pris par les tâches de contrôle utilisateur dans la phase parallèle. Ces tâches sont au nombre de 3 sur les milliers que compte le DAG, mais elles sont importantes car elles décrivent le contrôle de l'algorithme (arbre de création des tâches). Ce contrôle est ici centralisé du fait du choix d'implémentation de l'algorithme LU. Ceci va provoquer l'envoi de nombreux messages de petites taille perturbant les autres communications. Comme à l'avenir, nous prévoyons de dupliquer ces tâches de contrôle utilisateur, nous effectuons nos mesures de temps une fois ces 3 tâches terminées.

Les tâches et les dépendances, présentes en nombre cubique, sont coûteuses à générer. On constate que les coûts cumulés de l'inspection et de l'ordonnancement franchissent un palier, dès que le découpage de la matrice est plus fin que 48×48 blocs. Néanmoins, le coût de ces pré-traitements peut être rentabilisé lorsque la structure de la matrice reste la même (i.e. seuls les coefficients de la matrice changent).

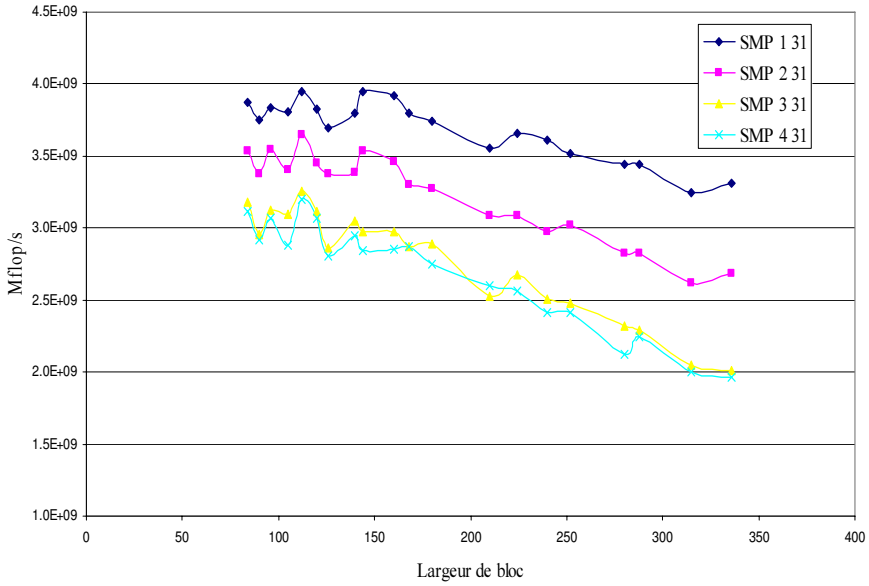

FIG. 25 – Rendement pour des nœuds IBM p690 de l'algorithme LU avec différentes largeurs de bloc (matrice 10080×10080).

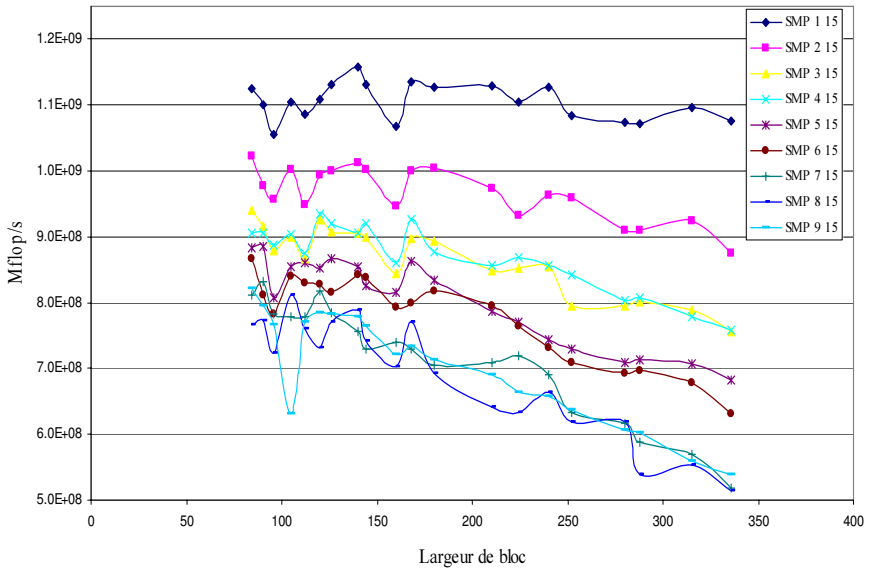

FIG. 26 – Rendement pour des nœuds IBM NH2 de l'algorithme LU avec différentes largeurs de bloc (matrice 10080×10080).

Etude du rendement. Les courbes des figures 25 et 26 présentent le rendement en MFlops (en ordonnée) obtenu par processeur (hors processeur exécutant les tâches de communication) en fonction du grain de calcul (largeur de bloc en abscisse). Ces rendements sont indiqués pour des configurations de 1 à 4 nœuds (SMP 1 31 à SMP 4 31) pour les p690 et de 1 à 9 nœuds (SMP 1 15 à SMP 9 15) pour les NH2.

Pour les plus petites largeurs de blocs que nous avons considérées (de 84 à 160), le nombre de blocs est important, donc le nombre de tâches (qui est cubique) l'est également. Ce grand nombre de tâches, combiné à la distribution cyclique, améliore l'équilibrage pour des configurations d'exécution avec plus de nœuds.

Pour toutes les courbes (NH2 et p690), les premières valeurs de rendement stagnent au lieu de décroître avec l'augmentation du grain de calcul (largeur de bloc). En effet, l'équilibrage est moins bon mais ce phénomène est compensé par un surcoût moindre de gestion à l'exécution de ce nombre cubique de tâches et par un meilleur rendement lors de l'enchaînement des BLAS.

Intéressons nous plus spécifiquement aux rendements obtenus avec les p690 (figure 25). Ils oscillent à 10 % près autour des mêmes valeurs (3.8 Gflops par processeur pour 1 SMP, 3.5 Gflops pour 2 SMP, 3 Gflops pour 3 ou 4 SMP) tant que la largeur de bloc est inférieure à 200, ensuite le rendement baisse. Les quatre courbes sont décalées les unes par rapport aux autres, sauf la dernière (SMP 4) et l'avant dernière (SMP 3). La perte de rendement entre ces courbes s'explique par l'utilisation de plus en plus importante des communications réseau.

Les trois premières courbes correspondent à des exécutions avec une grille de processus 1×1, 2×1 et 3×1 alors que la dernière utilise une grille 2×2 mieux adaptée au problème. Ceci explique la proximité des deux dernières courbes. En passant de 1 à 4 nœuds, environ 25% de la puissance de crête a été perdue. Ce pourcentage est bon, il indique que les communications extra-nœuds sont bien gérées par PRFX. Pour les p690, les petites largeurs de blocs sont moins pénalisées car la bibliothèque de communication et le réseau sont plus performants. Par contre le nombre de tâches n'est pas suffisant pour alimenter quatre nœuds (i.e. 128 processeurs) lorsque la largeur des blocs augmente.

On retrouve les mêmes phénomènes sur les NH2 (cf. figure 26) mais avec plus d'ampleur car la configuration d'exécution va jusqu'à 9 nœuds. Par exemple, la grille 1×3 du SMP 3 a un rendement équivalent à 4 % près de celle 2×2 du SMP 4, de même le 2×3 du SMP 6 permet de conserver le même rendement que la grille 1x5 du SMP 5. Ensuite pour les configurations SMP 7, SMP 8 ou SMP 9, le problème résolu n'est pas d'une taille suffisante pour que l'effet pipeline de l'algorithme soit exploité durablement en régime continu. On constate donc une dégradation des performances plus accentuée car plus de ressources sont mises en jeu.

Les figures 27 et 28 contiennent les traces d'exécutions réelles sur trois p690 de notre implémentation de l'algorithme du LU. Chaque tâche est coloriée avec un code couleur indiqué en légende. Ce code correspond au pourcentage de temps supplémentaire pris par la tâche par rapport aux prévisions utilisées lors de l'ordonnancement par PRFX. Par exemple, toutes les tâches de la même couleur que `worstpercent_0_10` sont entre 0 et 10 % plus lentes que prévu. On constate sur ces traces d'exécution que le rendement est stable (entre 10 et 20% en dessous des prévisions) pour les plus gros blocs (de dimension 120 à 336). Il se dégrade (perte de 20 à 40%)

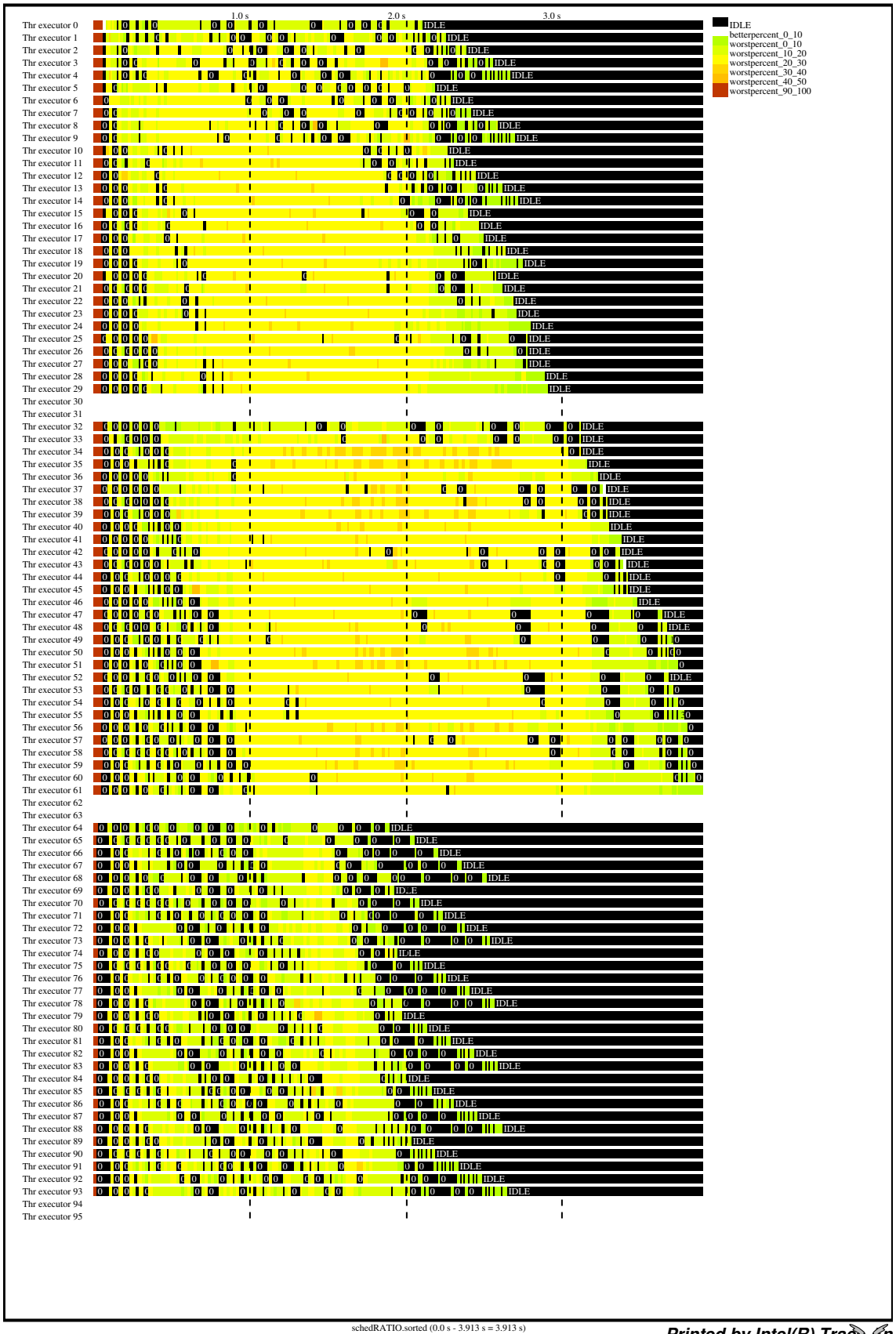

FIG. 27 – Traces d'exécution de la factorisation LU avec PRFX d'une matrice 10080×10080 découpée en 30×30 blocs de 336×336 sur trois nœuds p690.

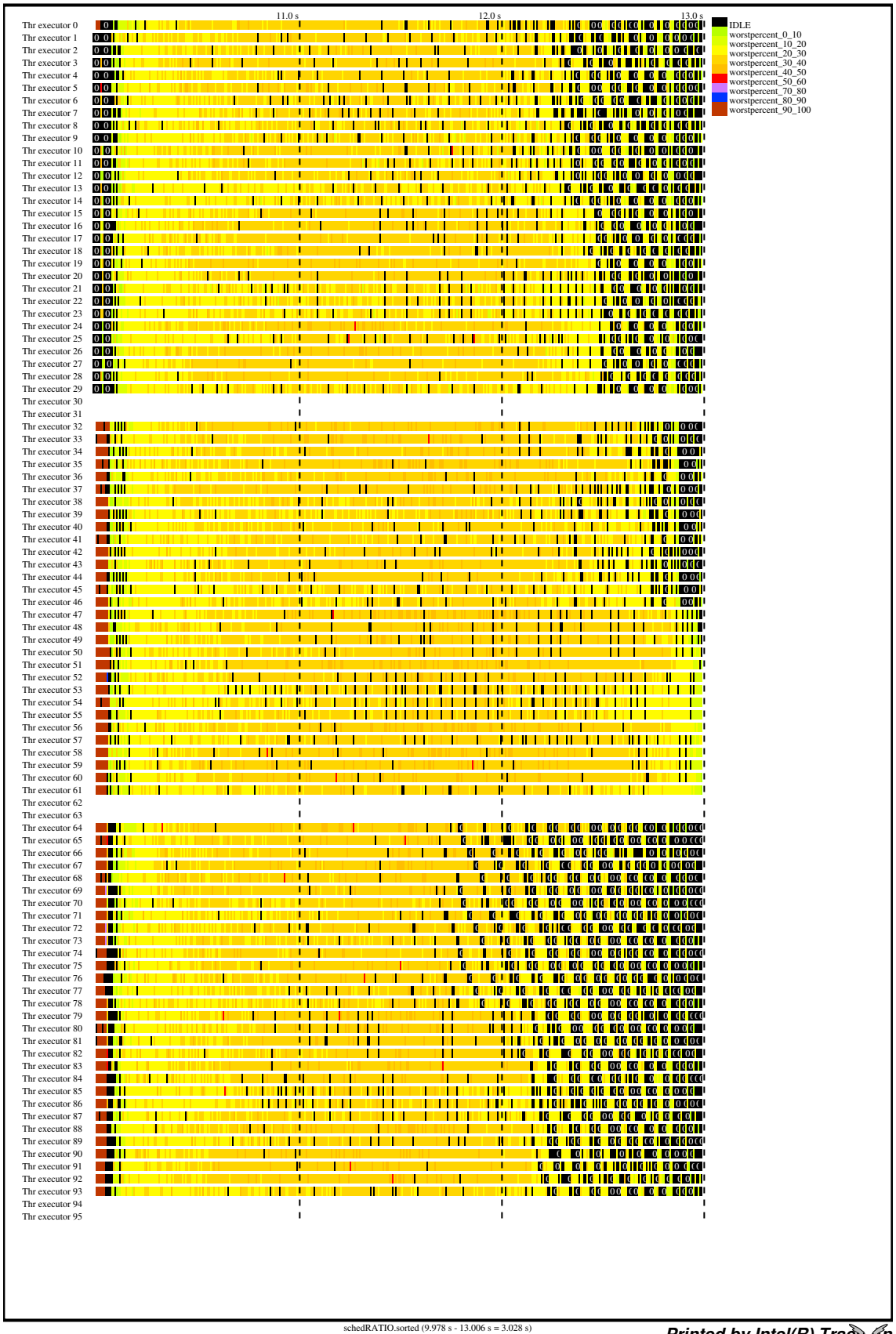

FIG. 28 – Traces d'exécution de la factorisation LU avec PRFX d'une matrice 10080×10080 découpée en 120×120 blocs de 84×84 sur trois nœuds p690.

pour les blocs de largeur plus petite. Pour ces petites largeurs, l'effet de cache qui était présent lors des mesures de calibrage des BLAS ne se reproduit pas complètement lors de la phase parallèle.

Le temps d'exécution s'écarte de la prévision au fur et à mesure que les nœuds sont rajoutés et que les blocs sont plus petits. Cet écart est de l'ordre de 6%, 14%, 20%, 21%, 25%, 27%, 32%, 34%, 32% pour 1 à 9 NH2 et de 7%, 12 %, 19% et 22% pour les exécutions sur 1 à 4 p690. Ces écarts sont satisfaisants ; néanmoins, ils montrent que les prévisions que nous avons définies à partir de mesures expérimentales de la durée des tâches dans un cadre idéal sont moins pertinentes pour l'exécution réelle, et ce d'autant plus que le nombre de nœuds augmente.

Etude du *speedup*. Les courbes des figures 29 et 30 indiquent le *speedup* (en ordonnée) de notre implémentation de l'algorithme LU de 1 à 4 nœuds dans le cas des p690 et de 1 à 9 nœuds dans le cas des NH2. Pour les p690 (figure 29), nous avons reporté les 20 courbes correspondant aux différentes granularités de calcul étudiées. Dans le cas des NH2, pour que les courbes restent identifiables, nous n'avons conservé que 10 granularités de problème différentes. Ces granularités sont référencées par la légende avec deux nombres, le premier indique le nombre de blocs sur la largeur de la matrice et le second la largeur d'un bloc.

Le *speedup* croît par paliers car la conformation de la grille de *threads* provoque un équilibrage plus ou moins bon selon que N (la largeur en nombre de blocs de la matrice) est un multiple ou non des dimensions de cette grille. Ce *speedup* est également meilleur lorsque la grille de processus de la distribution est carrée.

Cela est visible dans le cas des NH2 lorsque le nombre de nœuds est égal à un carré : 4 (grille 2×2) et 9 (grille 3×3). En revanche, le *speedup* est logiquement moins bon pour les nombres de nœuds correspondant à des nombres premiers.

La scalabilité de l'implémentation est globalement meilleure sur p690 que sur NH2 car le réseau *Federation* est mieux exploité par LAPI.

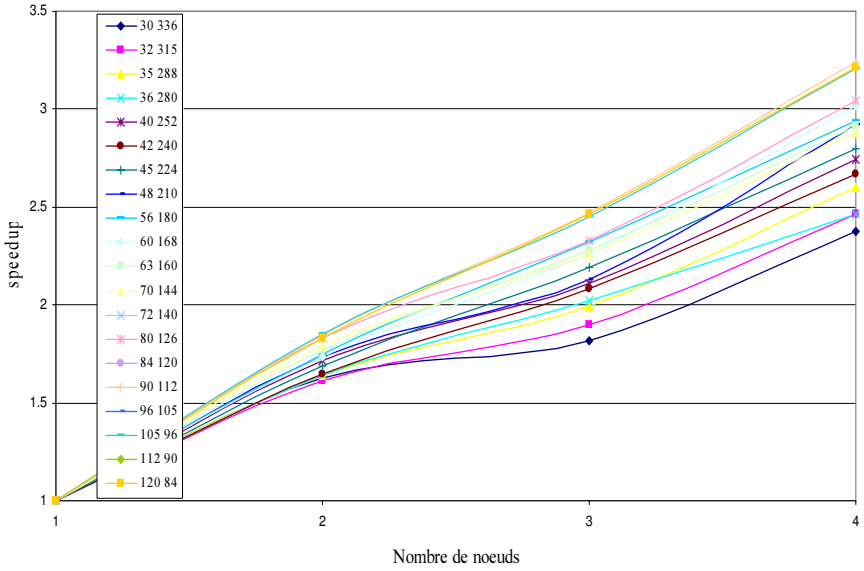

FIG. 29 – Scalabilité pour des nœuds IBM p690 de l'algorithme LU avec différentes largeurs de bloc (matrice 10080×10080).

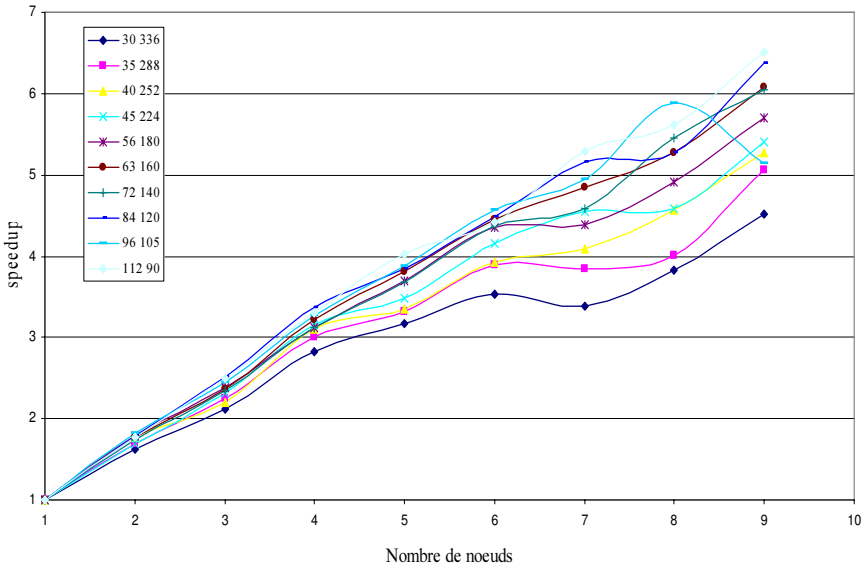

FIG. 30 – Scalabilité pour des nœuds IBM NH2 de l'algorithme LU avec différentes largeurs de bloc (matrice 10080×10080).

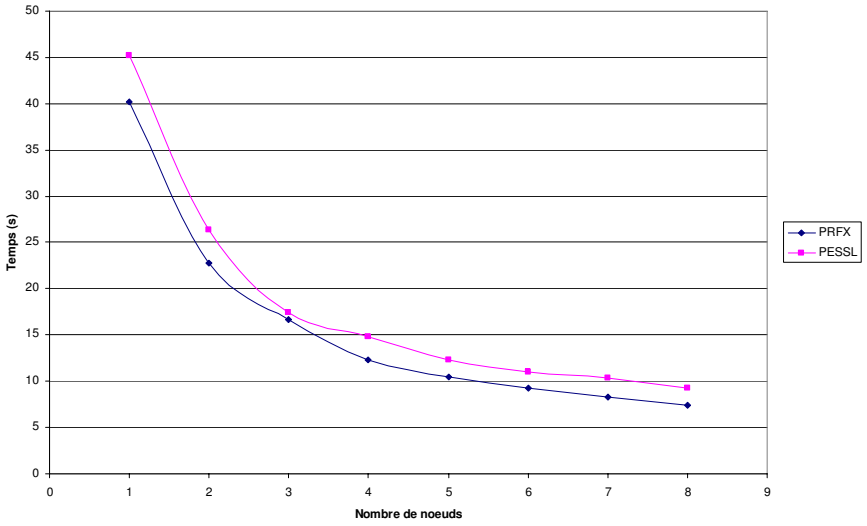

FIG. 31 – Comparaison de PRFX avec PESSL pour une matrice 10080×10080 découpée en 84×84 blocs de 120×120 .

Comparaison de PRFX avec PESSL. PESSL (Parallel Engineering and Scientific Subroutine Library) est une bibliothèque de fonctions scientifiques (équivalente à ScaLAPACK) fournie par le constructeur et dédiée aux machines SMP IBM. Cette bibliothèque permet de programmer et d'exploiter efficacement ces grappes de SMP. Nous avons choisi PESSL plutôt que sa version SMP (PESSLSMP) car l'algorithme implémenté dans PESSLSMP n'est qu'une version macroscopique de celui de PESSL. Ceci séquentialise les calculs et les communications ; ces dernières ne peuvent donc pas être envoyées de façon anticipée dans PESSLSMP. Les appels BLAS sont des BLAS SMP opérant sur des plus gros blocs. Ils sont exécutés par autant de *threads* qu'il y a de processeurs sur un nœud. Pour satisfaire un BLAS SMP sur un nœud NH2 (16 processeurs), il faut une taille de bloc au moins de l'ordre de 1000×1000. Avec cette granularité de calcul, les communications sont moins fréquentes, le parallélisme est donc faible pour une matrice de taille 10000×10000. Cette bibliothèque PESSLSMP est performante pour des nœuds avec moins de processeurs (e.g. quadri-processeurs).

Nous avons reporté sur le graphique de la figure 31 les temps d'exécution sur NH2 de PRFX et PESSL lorsque la matrice 10080×10080 est composée de 84×84 blocs de taille 120×120. Le paramétrage de ce cas est favorable à PRFX car la largeur de blocs choisie est telle que le niveau de parallélisme est suffisant tout en ayant une gestion interne peu coûteuse. Une distribution SMP est utilisée dans le cas de PRFX alors qu'une simple distribution cyclique sur les deux dimensions est (au mieux) mise en place avec PESSL. L'algorithme de PESSL calcule systématiquement la ligne pivot alors que notre algorithme non. Toutefois, nous avons conditionné la matrice de telle sorte qu'aucune ligne ne soit permutée par PESSL. Cette dernière est utilisée en configuration multi-processus (i.e. 16 processus par nœud) mono-*thread*. Les communications entre ces processus sont effectuées par une version de MPI capable d'exploiter des

communications via des segments de mémoire partagée. PRFX, quant à lui, est multi-processus multi-*thread* avec le déploiement que nous avons décrit précédemment. Quinze processeurs pour PRFX et seize pour PESSL sont utilisés pour faire les opérations BLAS. Pour PRFX, les *threads* de la bibliothèque de communication au nombre de deux sont confinés sur le processeur restant. Dans le cas de PESSL, lorsqu'un processus appelle `MPI_init()`, la bibliothèque MPI crée trois *threads* qu'elle utilise pour gérer les communications intra et extra-nœud. Si nous comptons le *thread* du flux d'exécution principal du processus PESSL, nous obtenons un total de 64 *threads* par nœud. Les temps indiqués dans la version PRFX n'incluent pas le contrôle utilisateur qui est exprimé de façon centralisée. Ces éléments permettent de comprendre pourquoi PESSL est légèrement moins performant que PRFX. L'implémentation de PESSL est entièrement écrite à la main avec un travail important de mise au point pour obtenir des performances. Ceci explique que malgré les handicaps liés aux choix de son implémentation, et malgré les conditions expérimentales favorables à PRFX, elle soit ici 10 % moins performante qu'un programme PRFX exploitant les communications unilatérales et les *threads*. PRFX se compare donc favorablement à une bibliothèque permettant de programmer les grappes de SMP, si seule l'exécution parallèle est considérée.

Cet exemple est un algorithme avec un parallélisme potentiel important. Le DAG de tâches présente une irrégularité avec une décroissance du nombre de tâches au cours du déroulement de l'algorithme. Les performances obtenues avec PRFX sont bonnes jusqu'à des grands nombres de processeurs au vu de la taille du problème traité.

4.3 Factorisation de matrices creuses par la méthode de Cholesky

La factorisation de Cholesky est utilisée (cf. algorithme 3) par exemple en mécanique pour le calcul de structures par éléments finis. Les matrices de ces problèmes sont creuses et la structure de données irrégulière à manipuler complique notoirement l'algorithmique. L'implémentation efficace de l'algorithme parallèle par blocs de Cholesky appliquée à une matrice creuse nécessite un pré-traitement avec trois étapes préliminaires : la renumérotation, l'identification des blocs pour la matrice L et l'ordonnancement des calculs.

Durant la factorisation d'une matrice creuse, les calculs de contribution ($A_{ij} \leftarrow A_{ij} - L_{ik} \times L^t_{jk}$) de la factorisation mettent à jour des termes précédemment nuls ou non nuls de la matrice. Le creux de la matrice peut donc varier au cours de la factorisation. Pour conserver le creux de la matrice initiale, on renumérote les inconnues en utilisant par exemple la méthode des dissections emboîtées [53]. Dans un cadre parallèle, cette renumérotation qui est de type "*Divide and Conquer*" a aussi l'avantage d'induire du parallélisme en maximisant l'indépendance dans les calculs [1].

L'identification des blocs colonnes de L est une deuxième étape purement symbolique [18]. Cette structure basée sur la partition des inconnues en N parties est effectuée de telle sorte que les calculs de contributions ne concernent à chaque fois qu'un seul bloc colonne "en face". Le

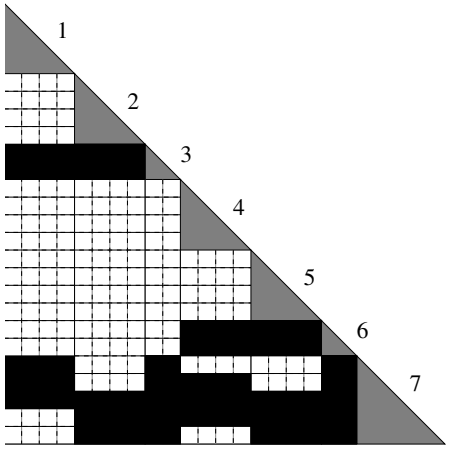

FIG. 32 – Matrice creuse de 7 blocs colonnes, avec 7 blocs diagonaux et 10 blocs extra-diagonaux.

Algorithme 3 Algorithme de factorisation de matrices creuses par la méthode de Cholesky.

Entrée Une matrice A creuse, symétrique et définie positive dont seule la partie triangulaire inférieure est stockée sous forme de N blocs colonnes. Ces blocs colonnes contiennent un bloc diagonal et des blocs extra-diagonaux ; le bloc colonne k est décrit par une liste de n_k blocs, numérotés de 1 à n_k et stockés de façon contiguë.

Sortie La matrice creuse L qui vérifie $A = LL^t$

1: **pour** k de 1 à N **faire**
2: factoriser A_{1k} en $L_{1k}U_{1k}$ où L_{1k} et U_{1k} écrasent A_{1k}
3: résoudre $L_{1k}L^t_{(2...n_k)k} = A^t_{(2...n_k)k}$ où $L_{(2...n_k)k}$ écrase $A_{(2...n_k)k}$
4: **pour** j de 2 à n_k **faire**
5: $k' \leftarrow$ numéro de bloc colonne en face du bloc L_{jk}
6: $A_{*k'} \leftarrow A_{*k'} - L_{(j...n_k)k}L^t_{jk}$
7: **fin pour**
8: **fin pour**

bloc colonne “en face” d'un bloc extra-diagonal B est celui dont les numéros de colonne correspondent aux numéros de ligne du bloc B. Par exemple, pour la matrice creuse représentée par la figure 32, le bloc colonne en face du premier bloc extra-diagonal du premier bloc colonne est le bloc colonne 3 et celui en face du deuxième bloc extra-diagonal est le bloc colonne 7.

L'algorithme 3, essentiellement à base de BLAS3, est semblable à celui traitant des matrices pleines (cf. algorithme 1 chapitre 2). Chaque bloc colonne a un stockage compact ce qui permet l'application d'opérateurs BLAS sur la totalité des blocs d'un même bloc colonne. Ainsi, les opérations BLAS se font sur des blocs plus gros. Ceci factorise la latence d'appel au BLAS3 et permet d'atteindre un régime de calcul où le rendement du processeur est meilleur.

Le placement des blocs va favoriser la localité des accès et des communications. Le parallélisme exploité est premièrement celui généré par le creux (indépendance entre les blocs), et

deuxièmement celui exploité par redécoupage des blocs denses de taille suffisamment grande. Il est irrégulier et nécessite un ordonnancement spécifique pour être correctement exploité.

Nous supposons être en possession d'une matrice renumérotée, structurée par blocs et dont le placement est indiqué en vue d'une factorisation sur la machine considérée. Nous avons utilisé pour ce faire le logiciel BLEND et les travaux décrits dans ([36]).

4.3.1 Prise en compte des accès commutatifs avec PRFX

Algorithme 4 Algorithme de factorisation de matrices creuses par la méthode de Cholesky avec réduction explicite pour PRFX.

Entrée Une matrice A creuse, symétrique et définie positive dont seule la partie triangulaire supérieure est stockée sous forme de N blocs colonnes. Ces blocs colonnes contiennent un bloc diagonal et des blocs extra-diagonaux ; le bloc colonne k est décrit par une liste de n_k blocs, numérotés de 1 à n_k et stockés de façon contiguë

Sortie La matrice creuse A dont la partie triangulaire inférieure modifiée : L vérifie $A = LL^t$

1: **pour** k de 1 à N **faire**
2: $c \leftarrow$ nombre de processus contribuant au $k$ième bloc colonne noté A_k
3: **pour** z de 1 à c **faire**
4: RPC pour tâche de synchronisation accédant à la duplication locale du bloc colonne A_k notée A_k^z
5: **fin pour**
6: **si** $c > 0$ **alors**
7: RPC pour tâche de réduction : $A_k \leftarrow \sum_{z=1}^{c} A_k^z$
8: **fin si**
9: RPC pour tâche de factorisation : factoriser sur place A_{1k} en $L_{1k}U_{1k}$
10: RPC pour tâche de résolution : résoudre $L_{1k}L^t_{(2\ldots n_k)k} = A^t_{(2\ldots n_k)k}$ où $L_{(2\ldots n_k)k}$ écrase $A_{(2\ldots n_k)k}$
11: **pour** j de 2 à n_k **faire**
12: $k' \leftarrow$ numéro de bloc colonne en face du bloc L_{jk}
13: $z \leftarrow$ indice du bloc colonne local correspondant à $A_{k'}$
14: $A^z_{k'} \leftarrow$ duplication locale du bloc colonne $A_{k'}$
15: RPC pout tâche de contribution locale : $A^z_{*k'} \leftarrow A^z_{*k'} - L_{(j\ldots n_k)k}L^t_{jk}$
16: **fin pour**
17: **fin pour**

L'algorithme 3 est séquentiel et l'opération d'accumulation de la ligne 6 (un bloc B dans un bloc colonne k' va être modifié par un ensemble de contributions issues de calculs réalisés au titre des blocs colonnes de numéros inférieurs à k' dans lesquels il y a un bloc extra-diagonal en face de k') est critique pour les performances dans le cas parallèle. Cette opération d'accumulation irrégulière doit être implémentée sous forme d'agrégations locales à chaque processus et d'une réduction. Cela suppose l'existence d'accès commutatifs (CRW). Ce type d'accès est fourni par PRFX mais non implémenté pour l'instant de façon optimisée (accès CRW considérés comme RW). Pour exploiter tout de même le parallélisme induit par les accès commutatifs

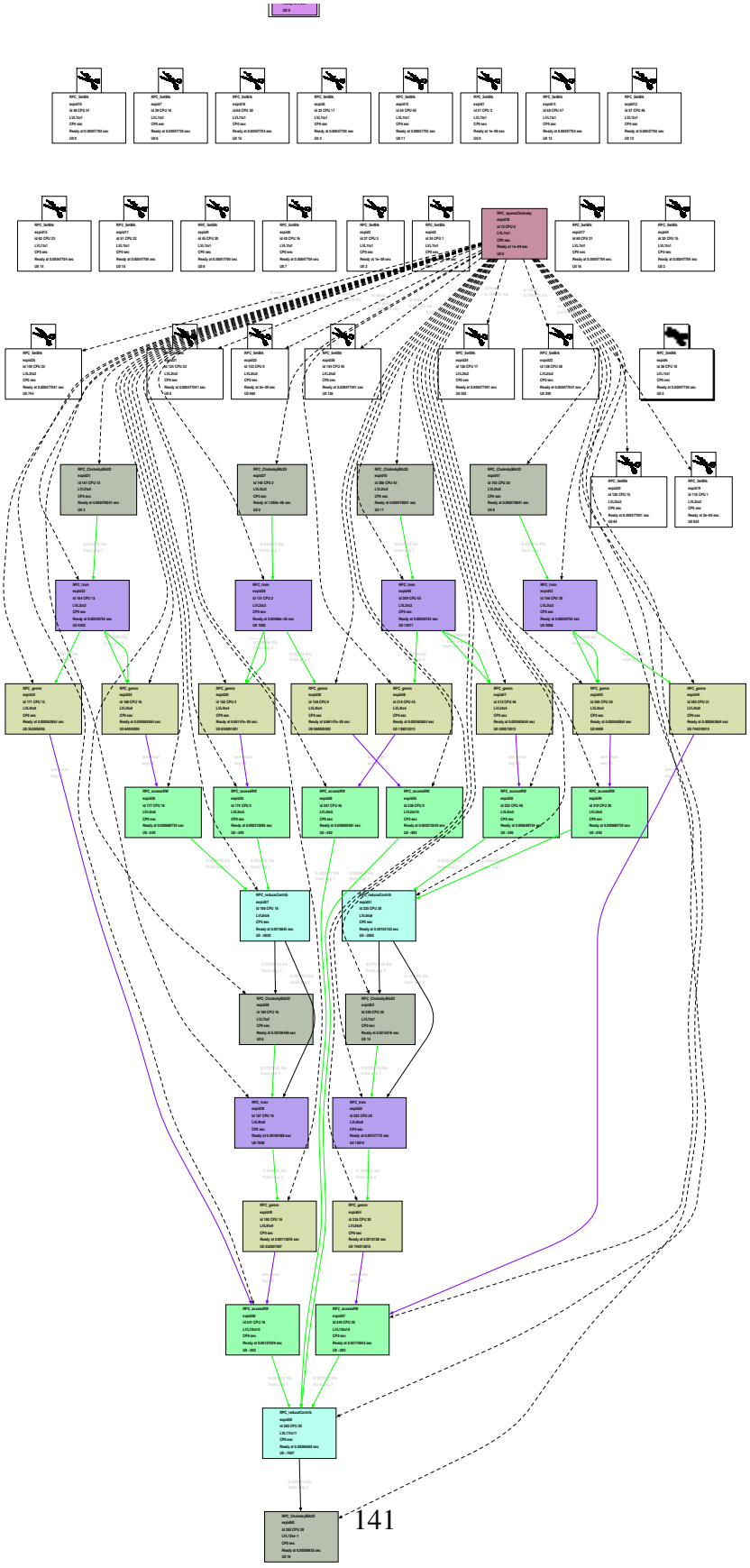

FIG. 33 – DAG daVinci de l'exécution de l'algorithme de Cholesky pour la matrice de la figure 32.

de l'algorithme de Cholesky, nous implémentons plutôt l'algorithme 4 décrivant explicitement les tâches de synchronisation permettant la gestion de ces accès CRW. Ces tâches devront être générées automatiquement par PRFX dans le futur. Ici, les accès commutatifs concernent une réduction que nous décrirons également explicitement avec une tâche. Qui plus est, nous implémentons cette réduction irrégulière en tenant compte du placement des processus pour obtenir de meilleures performances. Cette prise en compte passe par une agrégation par nœud SMP, i.e. toutes les tâches qui contribuent à un même bloc colonne modifie un même bloc colonne local dans lequel on agrège les contributions (la duplication du bloc colonne distant). L'algorithme 4 nécessite donc des versions locales des blocs colonnes subissant les contributions (lignes 13 et 14). Une seule duplication de ce bloc colonne est effectuée par processus contributeur. Tous les *threads* d'un même processus peuvent accéder en parallèle avec des opérateurs atomiques au même bloc colonne local. Chaque ensemble de blocs colonnes locaux associé à un bloc colonne k doit être rassemblé dans un seul processus PRFX (a priori celui qui traitera ce bloc colonne k) en vue d'effectuer leur réduction. Nous utilisons pour cela une solution intermédiaire qui consiste à attendre la fin des contributions dans chaque bloc colonne local (boucle de la ligne 3 à la ligne 5) avec une tâche factice ne faisant rien, puis à créer une tâche effectuant la réduction (ligne 7). Ceci se traduit par exemple dans le cas de la matrice creuse de la figure 32 par la génération du DAG de la figure 33. Ce DAG ne représente que les tâches utilisateurs. Chacune des tâches valuant le contenu initial des blocs a été manuellement déconnectée (tâches de couleur blanche avec un ciseau) dans un but simplificateur. Seules restent les tâches de contrôle (marron et rose en haut), les tâches de calcul (opérateur BLAS de factorisation `potrf` en gris, de résolution `trsm` en violet et de multiplication `gemm` en beige) et les tâches liées à la gestion des contributions commutatives des tâches `gemm` (tâches factices en vert et tâches de réduction en bleu).

La tâche de contrôle marron exécute la partie contrôle de l'algorithme 4, d'où les nombreux arcs de contrôle en pointillés sortant de cette tâche. Pour la matrice considérée ici, les traitements des blocs colonnes 1, 2, 4 et 5 peuvent démarrer en parallèle car ils ne subissent pas de contributions (absence de bloc sur la ligne de leur bloc diagonal). Ce parallélisme est bien détecté par l'inspecteur, ce qui est visuellement vérifiable (voir la cinquième ligne du DAG). Le traitement de chacun de ces blocs colonnes se poursuit, avec les `trsm` puis les `gemm`. A la septième ligne du DAG apparaissent les tâches de synchronisation implémentant la réduction voulue sur les contributions. Les deux synchronisations de gauche et les deux de droite sont utilisées à la ligne suivante du DAG pour effectuer la réduction. Les deux synchronisations centrales ne seront utilisées que pour la dernière réduction du DAG. Une fois ces réductions faites, le traitement des blocs colonnes reprend en suivant ce même schéma.

4.3.2 Résultats

Les mesures sont obtenues avec une implémentation de l'algorithme 4 utilisant la bibliothèque PRFX sur une grappe de NH2 et une grappe de p690+.

Les expérimentations ont été réalisées sur quatre matrices creuses au format RSA ; les valeurs des métriques indiquées dans le tableau 2 sont celles obtenues avec une factorisation symbolique par colonne. Nous utilisons le logiciel BLEND [36] pour redécouper ces matrices avec un facteur indicatif de blocage égal à 100 et obtenir une distribution initiale des blocs. BLEND adapte le

TAB. 2 – Description des matrices traitées. NNZ_A est le nombre de termes extra-diagonaux dans la partie triangulaire de A, NNZ_L est le nombre de termes extra-diagonaux dans la matrice factorisée et *OPC* est le nombre d'opérations requises.

Name	Columns	NNZ_A	NNZ_L	OPC
THREAD	29736	2.220e+06	2.404e+07	3.884e+10
OILPAN	73752	1.761e+06	8.912e+06	2.985e+09
BMWCRA1	148770	5.247e+06	6.597e+07	5.702e+10
SHIP001	34920	2.304e+06	1.428e+07	9.034e+09

découpage au nombre de processeurs à exploiter ; plus ce nombre est grand, plus le découpage est fin (modulo une taille critique pour l'efficacité des BLAS3) afin que le parallélisme soit plus important. Les tâches traitant ces blocs sont ordonnancées par PRFX en sélectionnant l'heuristique dite du *ready time* avec un ensemble de CPU candidats limité à ceux de la carte du processeur initialement choisi par BLEND. Les blocs de la matrice sont traités avec les opérateurs BLAS de la bibliothèque IBM ESSL.

Les histogrammes des figures 34, 35, 36 et 37 décrivent les temps nécessaires à la génération du DAG (inspection), à son ordonnancement et à son exécution sur la grappe de nœuds NH2. Pour l'inspection, le temps de construction de l'arbre des accès atomiques à partir des données structurantes contenues dans le fichier décrivant la structure creuse de la matrice n'est pas comptabilisé. Ce temps est conséquent (entre 2 et 5 fois le temps d'inspection) car le nombre de motifs d'accès est important à cause des grains d'accès variables à l'intérieur de chaque bloc colonne. Nous avons considéré que cette information statique relative aux des données structurantes, et facilement connue à l'issue de la factorisation symbolique, n'a pas à être retrouvée par l'inspecteur. La durée de l'inspection peut être raccourcie dès lors que ces informations sont fournies.

Pour chaque histogramme, la matrice concernée est indiquée par le commentaire. Pour chaque phase (inspection, ordonnancement et exécution), les quatre barres correspondent successivement aux quatre déploiements sur 1,2,3 et 4 nœuds NH2. La largeur de blocage adaptative de BLEND en fonction du nombre de nœuds utilisés conduit à des DAG contenant plus de tâches. De fait, l'inspection et l'ordonnancement sont plus coûteux au fur et à mesure que le nombre de nœuds utilisés croît. Le temps supplémentaire investi en pré-traitements (inspection plus ordonnancement) lors du passage de 1 à 2 nœuds NH2 permet en contre-partie d'améliorer significativement le temps d'exécution parallèle. Par contre, les surplus de temps investis par la suite (jusqu'à 4 nœuds) ne sont pas récompensés par une amélioration rentable des performances de la phase parallèle, mais cela est dû à une trop faible granularité de calcul pour 4 nœuds SMP (64 processeurs).

Ces temps investis dans l'inspection et l'ordonnancement sont importants vis à vis du temps passé dans la phase parallèle. Néanmoins, sur des exemples très irréguliers tels que la factorisation de matrices creuses, ces pré-traitements permettent d'obtenir de réelles performances contrairement à une exécution n'exploitant pas statiquement les données structurantes du problème.

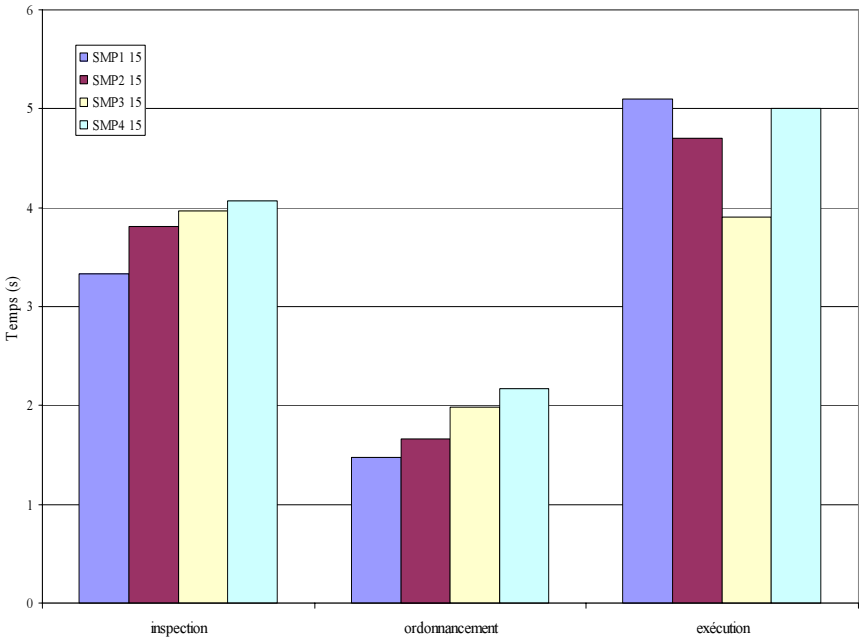

FIG. 34 – Représentation des durées des trois étapes sur NH2 : inspection, ordonnancement et exécution pour la matrice THREAD .

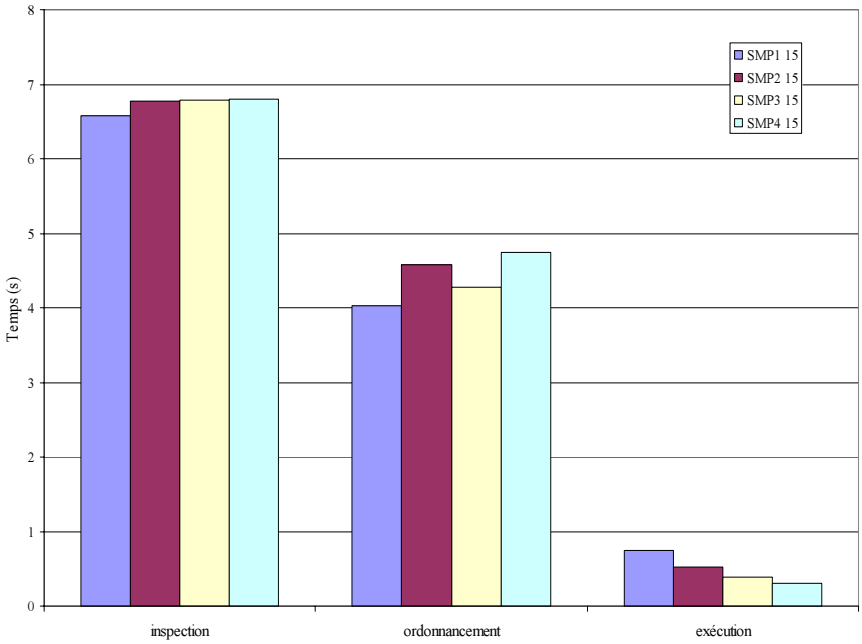

FIG. 35 – Représentation des durées des trois étapes sur NH2 : inspection, ordonnancement et exécution pour la matrice OILPAN .

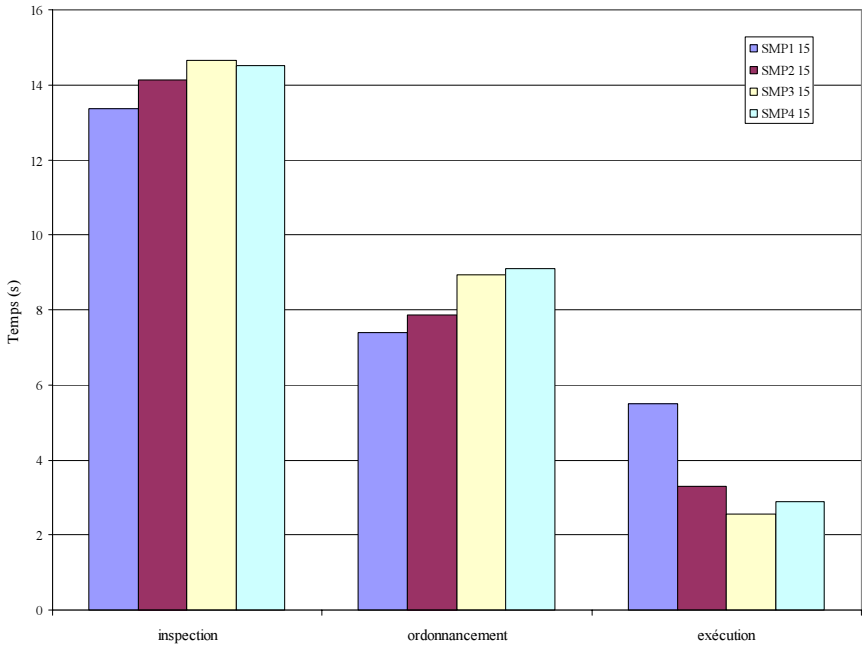

FIG. 36 – Représentation des durées des trois étapes sur NH2 : inspection, ordonnancement et exécution pour la matrice BMWCRA1 .

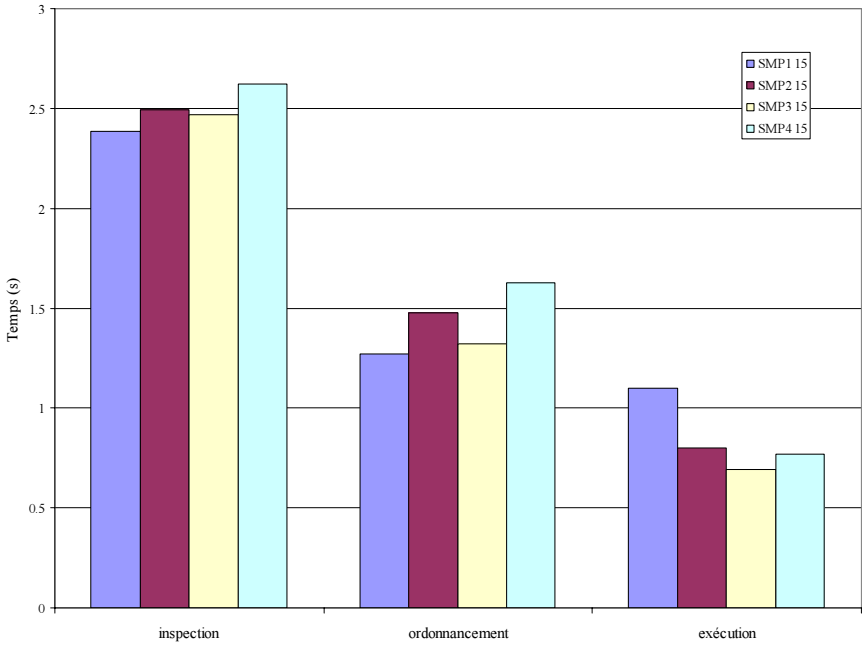

FIG. 37 – Représentation des durées des trois étapes sur NH2 : inspection, ordonnancement et exécution pour la matrice SHIP001 .

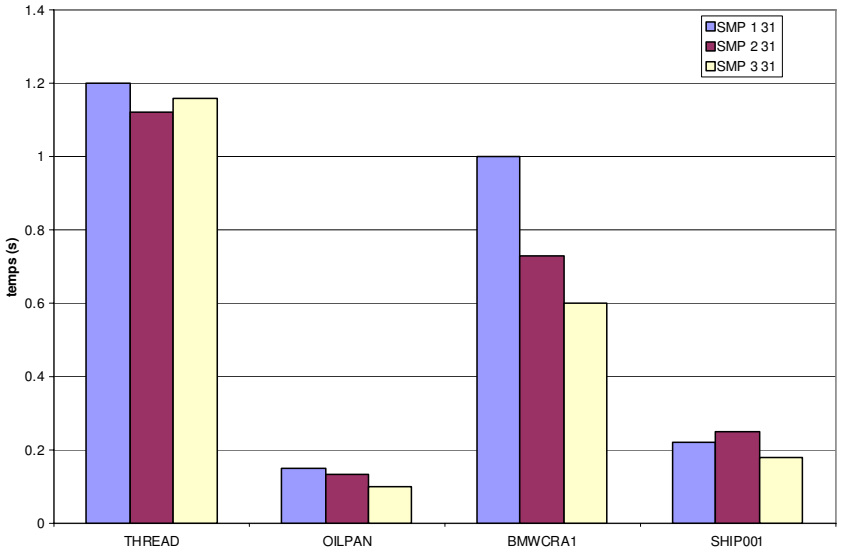

FIG. 38 – Temps de factorisation avec une grappe de p690+.

On constate également que les temps d'inspection varient comme les temps d'ordonnancement. Ceci indique que notre implémentation de l'inspecteur est d'une complexité du même ordre que l'ordonnancement pour ces cas irréguliers. Ces deux technologies sont génériques et ne sont de fait pas optimisées pour le cas de l'algorithme de Cholesky opérant sur des matrices creuses.

Nous avons également procédé à des tests sur la grappe de nœuds p690. Nous avons obtenu les temps de factorisation indiqués dans l'histogramme de la figure 38. Pour chaque matrice, chaque barre successive correspond à l'utilisation d'un nœud (soit 32 processeurs) supplémentaire. En comparant sur une configuration égale de 32 processeurs, le temps de factorisation sur NH2 avec celui sur p690, on observe un gain moyen d'un facteur 3 pour les p690 (nous rappelons qu'en théorie ce facteur pourrait être de 4.5). L'amélioration des temps de factorisation obtenus lors du passage de 32 à 64 processeurs est en moyenne de 6 % pour les NH2 et de 11 % pour les p690. La scalabilité est donc meilleure pour les p690, et cela malgré une taille de problème relativement petite par rapport au nombre de processeurs engagés. Cette différence s'explique notamment par les fonctionnalités matérielles (capacités d'adressage) du réseau *Federation* utilisées par la bibliothèque LAPI, ce qui conduit à des communications unilatérales plus performantes.

Nous revenons à présent aux grappes de NH2 et nous allons donner quelques chiffres sur le pourcentage de temps perdu par PRFX par rapport aux prévisions de l'ordonnanceur et par rapport à un logiciel dédié comme PaStiX [36]. Pour cet exemple de factorisation de matrices creuses, notre implémentation est entre 5 et 15 % plus lente que les prévisions (matrice OILPAN

mise à part) jusqu'à 3 nœuds. Cet écart s'agrandit à partir de quatre nœuds pour les matrices THREAD et BMWCRA1, indiquant une perte de signification des prévisions de l'ordonnanceur. En effet, les matrices traitées ne sont pas d'une taille suffisante, si bien que les surcoûts de gestion inhérents à notre plate-forme ne sont plus négligeables au-delà de trois nœuds NH2.

PaStiX est capable de prolonger les performances jusqu'à quatre nœuds pour ces tailles de problème. PaStiX, qui est écrit en langage C, utilise la bibliothèque MPI et exploite les nœuds SMP avec des *threads*. PaStiX prend en entrée non seulement le placement calculé par BLEND mais également l'ordonnancement qu'il produit. De plus, PaStiX exploite les 16 processeurs de chaque nœud NH2 avec des *threads* entrelaçant calculs et communications alors que PRFX dédie 1 processeur (*thread*) aux communications et 15 processeurs (*threads*) aux calculs.

Relativement à une solution dédiée comme PaStiX, notre solution est environ de 10 et 35% plus lente si l'on ne considère que la phase parallèle. Cette différence de performance est à relativiser par rapport au niveau d'expression utilisé. PaStiX est un solveur dédié aux problèmes de factorisation de matrices creuses. PRFX offre plus de fonctionnalités (migration de données) qui ne l'oblige pas à travailler à placement fixe comme PaStiX. PRFX est également une approche plus générale du point de vue des problèmes traitables.

4.4 Conclusion

Les expérimentations présentées ici sont un panel représentatif des algorithmes scientifiques implémentables avec le mode d'expression PRFX en restant aux frontières des capacités du support. Ils démontrent ses capacités à gérer des accès se chevauchant (Jacobi), des grands nombres de tâches (LU) ainsi que des accès irréguliers à grain variable (Cholesky pour des matrices creuses). Il ressort de ces expérimentations que le choix du modèle du DAG de tâches est utile au contrôle de l'exécution (ordonnancement) et à sa compréhension par l'utilisateur. Ces deux éléments permettent d'améliorer les performances par essais successifs de différentes stratégies algorithmiques ou d'ordonnancement. Ces changements s'opèrent aisément grâce aux concepts de tâches et d'espace mémoire partagée au niveau expression, et grâce au mécanisme d'inspection / exécution. Malgré ce niveau d'expression plus élevé par rapport aux bibliothèques dédiées, le support PRFX est capable d'offrir un niveau de performances intéressant. Les coûts des pré-traitements (inspection et ordonnancement) liés à ce DAG peuvent être rentabilisés pour les algorithmes présentés.

Chapitre 5

Perspectives

Plusieurs domaines du parallélisme sont abordés par cette thèse, en particulier ceux relevant des modes d'expression, des heuristiques d'ordonnancement, du mécanisme inspection/exécution et des supports d'exécution. Le regroupement de ces domaines au sein de la plate-forme PRFX nous permet de définir des évolutions réalisables tout en maintenant la cohérence de l'ensemble.

5.1 Mode d'expression

Dans le mode d'expression actuel, la description des accès aux données est fastidieuse dès lors que les structures de données misent en jeu sont des compositions de *stencils* de base. Un moyen de simplifier ces descriptions passerait par de nouvelles fonctionnalités pour décrire des partitions dans des données parallèles structurées avec des pointeurs (e.g. graphe). Elles intégreraient à la fois des informations de partitionnement et de sérialisation de la donnée. Le partitionnement serait indiqué par le programmeur en enregistrant les liaisons (pointeurs) du graphe de données chevauchant les frontières du dit partitionnement. La sérialisation de ces structures de données complexes peut être faite en demandant au programmeur les informations sur les emplacements des chaînages par pointeurs et sur leurs valeurs terminales (sérialisation guidée).

D'autre part, de même que le programmeur déclare les accès mémoire, il pourrait déclarer ses accès aux fichiers. Ainsi, le parallélisme présent lors de ces accès serait modélisé dans le DAG. Ceci permettrait par exemple d'exploiter des programmes avec une technologie *out-of-core*. L'implémentation de ce service au niveau du support PRFX serait géré via MPI-IO ou des services de la grille (e.g. GridFTP).

Le mode d'expression pourrait également être enrichi pour permettre au programmeur de relâcher (perte de propriété) une donnée parallèle explicitement. Cette fonctionnalité autoriserait le support, au cours de l'exécution de la tâche, à effectuer au plus tôt des communications correspondant aux données relâchées.

En l'état actuel, pour ce faire, le programmeur doit créer une tâche effectuant l'accès à la donnée sur laquelle porte le relâchement. Ce type de fonctionnalité est utile pour exploiter le fait que certaines tâches n'utilisent plus de données parallèles à partir d'un certain moment de leur

exécution. Toutefois, ce fonctionnement ne correspond pas à celui supposé par les heuristiques d'ordonnancement où les communications n'interviennent qu'à partir de la terminaison de la tâche.

Un autre moyen d'enrichir le mode d'expression est d'intégrer totalement le concept de tâche multi-*threads* (e.g. opérateurs BLAS SMP, code OpenMP ou utilisant les *threads* POSIX). Le mode d'expression actuel et le support permettent l'utilisation de ce type de tâche, mais elles ne sont pas prises en compte dans le DAG car non déclarées au niveau expression. Les *threads* supplémentaires créés ne sont pas contrôlés par PRFX : ils préemptent le processeur des *threads* exécuteurs PRFX. Il faudrait donc ajouter une fonctionnalité de création de tâches multi-*threads*. Cette fonctionnalité permettrait d'exploiter un grain de parallélisme fin en intra-processus sans que l'inspecteur ait besoin de générer le sous-DAG interne à cette tâche multi-*threads*. Le grain de ces tâches est plus important du point de vue de l'inspecteur si bien que la fonctionnalité de relâchement de données précédemment décrite serait alors très utile pour conserver du parallélisme. Comme les données parallèles restent à l'intérieur du même processus, les dépendances entrantes et sortantes de cette tâche multi-*threads* pourraient être générées avec la même méthode que les tâches séquentielles.

En revanche, pour prendre en compte une tâche exécutant par exemple un code MPI, l'inspecteur devrait connaître le nouveau placement des données à la terminaison de cette tâche multi-processus. Ainsi, l'inspecteur peut continuer à générer correctement la cohérence et la synchronisation des tâches suivantes. Pour ces tâches multi-processus, il faut enrichir l'interface de PRFX pour permettre au programmeur d'indiquer la nouvelle distribution des données parallèles à la terminaison de cette tâche.

Enfin, une tâche pourrait être un DAG complet auquel cas le programmeur devrait spécifier par exemple un fichier décrivant ce DAG ou un moyen de le construire de façon paramétrée (bibliothèque de DAG). L'inspecteur pourrait alors travailler en générant juste les dépendances entre ces DAG.

5.2 Extension aux programmes quasi-prévisibles

Pour pouvoir continuer à offrir des performances, nous ne pouvons pas être trop généralistes en proposant de traiter tous les types de dynamicité des programmes (classe générale des programmes imprévisibles). Nous proposons donc de ne traiter que des programmes de la classe quasi-prévisible. Nous considérons qu'un algorithme est **quasi-prévisible** lorsque ses schémas de synchronisation sont connus mais que ses accès aux données sont surestimés. De fait, la géométrie des données portées par les arcs de synchronisation est également surestimée.

Pour traiter ces algorithmes, nous devrons étendre le mode de fonctionnement statique actuel vers un fonctionnement plus dynamique. Ainsi, nous pourrons traiter des algorithmes quasi-prévisibles comportant des étapes de calcul structurées par des données de niveau exécution (e.g. algorithme de factorisation LU avec pivotage). La dynamicité de ces algorithmes quasi-prévisibles ne doit porter que sur les accès aux données. Ces accès sont dits flous par analogie avec "le flou optique". Les objets n'ont pas de contours précis, mais il est tout de même possible

de les borner dans l'espace. Par opposition à l'adjectif flou, nous utiliserons celui de net pour parler des accès des programmes prévisibles. Un accès flou possède au moins une des propriétés suivantes :

- pointeur de base dans la donnée parallèle cible pour l'application du *stencil* dépendant de données structurantes de niveau exécution ;
- géométrie du *stencil* dépendant de données structurantes de niveau exécution.

Dans le cas des programmes quasi-prévisibles, les boucles et les conditionnelles doivent rester, comme pour un programme prévisible, uniquement dépendantes de données de niveau inspection. Cette contrainte permet de continuer à modéliser statiquement le programme par un DAG de tâches complet. Nous parlerons de tâche floue (resp. nette) lorsque cette dernière effectue au moins un accès flou (resp. aucun accès flou). Dans ce DAG, les dépendances de données relatives à des accès flous sont également dites floues. Ces dépendances sont sur-évaluées et de ce fait les tâches floues ont une durée potentiellement approximative. Les parties prévisibles du programme restent quant à elles modélisées finement. Si ces dernières sont majoritaires, le programme peut être supporté de façon performante, car ses parties prévisibles bénéficient d'une inspection, d'un ordonnancement et d'une exécution statique.

En adoptant un fonctionnement quasi-statique, nous minimisons les coûts de résolution des dépendances en cours d'exécution. En effet, nous ne résolvons que les dépendances floues, celles nettes ont déjà été résolues lors de l'inspection quasi-statique (avec possibilité d'amortissement). Par contre, les dépendances floues nécessitent deux résolutions : lors de la pré-exécution lorsqu'elles sont surestimées et lors de l'exécution lorsqu'elles sont instanciées. Toutefois le temps investi dans la résolution surestimée est très utile car il va alléger la tâche de résolution dynamique en restreignant son domaine de recherche. Après l'inspection quasi-statique, chaque dépendance floue est associée avec un ensemble des tâches potentiellement en relation avec elle. Ainsi, la marge de variation du comportement imprévisible de chaque dépendance floue est circonscrite à une zone strictement connue.

La classe quasi-prévisible s'étend jusqu'aux programmes dont toutes les dépendances de données sont floues et seules les dépendances de contrôle sont nettes. Dans ce cas, la seule information dont dispose la plate-forme avant l'exécution parallèle est l'arbre des créations de tâches avec leurs accès flous, et pour chacun de ces accès flous à résoudre, un ensemble de tâches candidates. La construction de cet arbre lors de l'inspection avec les tâches candidates puis son raffinement à l'exécution sont donc coûteux. Un apport d'informations pour l'inspecteur est donc souhaitable pour éviter d'avoir une exécution faisant fréquemment appel au résolveur de dépendances dynamiques.

Le flou qualifie la connaissance du DAG après le passage de la phase d'inspection quasi-statique. Pour rendre ce DAG net, l'exécuteur parallèle doit être doté d'un résolveur de dépendances à la volée. Ce résolveur évolue sur une zone restreinte du DAG (ensemble de tâches candidates) ce qui optimise ses coûts de fonctionnement. Lors de l'exécution parallèle, les parties floues du DAG complet se précisent à chaque lancement de tâche anciennement flou car les accès flous de chaque `PRFX_call()` sont enfin instanciés avec les données de niveau exécution.

Nous pourrons conserver un ordonnancement statique, car toutes les tâches de l'exécution sont dans le DAG. Ainsi, nous bénéficions d'une prise en compte globale de l'exécution. L'usage de modèles de coûts par l'ordonnanceur pour les dépendances et les tâches floues est impossible, d'où la nécessité de pouvoir procéder à un ré-ordonnancement dynamique (autre perspective).

5.2.1 Prise en compte des accès flous

L'expression des algorithmes quasi-prévisibles est possible grâce à une légère modification de la partie du mode d'expression servant à informer l'inspecteur. Cette partie concerne les accès flous qui doivent être indiqués par le programmeur.

La déclaration d'un accès flou s'effectue en utilisant une fonction de déclaration de *stencils* spéciale suffixée par `fuzzy`. Ces fonctions `fuzzy` ont le même comportement que leurs équivalentes nettes lors de l'exécution parallèle car les données de niveau inspection comme celles de niveau exécution sont alors connues. Ce n'est que pendant l'inspection qu'elles conduisent à des interprétations particulières par l'inspecteur.

Nous faisons remarquer que les traitements spécifiques liés au flou se propagent à distance 1 dans le DAG, c'est à dire qu'un accès net fait par une tâche dépendant d'une tâche floue devra être résolu à la volée.

Nous avons choisi de ne pas faire spécifier par le programmeur l'accès sur-évalué, pour limiter le nombre d'appels à la bibliothèque PRFX. Lors de la pré-exécution, les accès flous sont donc instanciés par l'inspecteur de façon maximale. Cet accès maximal, circonscrivant tous les accès d'une tâche fille, est construit en fusionnant l'ensemble des données parallèles dont la tâche initiatrice est propriétaire. Le programmeur peut toutefois obtenir l'accès sur-évalué plus restreint souhaité en créant une tâche intermédiaire nette effectuant un unique accès net sur-évalué puis en lançant la tâche aux accès flous.

Listing 5.25 – Exemple d'un accès flou.

```
1  void RPC_partition(int dim, int *a, double*b)
2  {
3  ...
4    globalCompute(dim,b);
5    a[0] = computePartition(dim,b);
6  }
7  void RPC_compute(int dim, double *b){
8  ...
9      localCompute(dim,b);
10 ...
11 }
12
13 void RPC_launchComputation(int dim,int*a,double*b){
14   ...
15   int dim1 = a[0];
16   int dim2= dim - a[0];
17
18   PRFX_stencil1Dfuzzy(&b1, dim1*sizeof(double));
19   PRFX_stencil1Dfuzzy(&b2, dim2*sizeof(double));
20
21   b1 = b;
22   b2 = &b[dim1];
23   PRFX_call(-1,0.0,RPC_compute,dim1,b1);
24   PRFX_call(-1,0.0,RPC_compute,dim2,b2);
25   ...
26 }
27 void main(void)
28 {
29   ...
30   PRFX_isoMalloc(&a,sizeof(int));
31   PRFX_isoMalloc(&b,n*sizeof(double));
32
```

```
33      PRFX_call(-1,0.0,RPC_partition,n,a,b);
34      PRFX_call(-1,0.0,RPC_launchComputation,n,a,b);
35      PRFX_call(-1,0.0,RPC_storeToFile,n,b);
36      ...
37  }
```

Le listing 5.25 présente un code qui calcule l'indice `a` (ligne 33) servant à scinder le tableau `b` en deux. Chacune des deux partitions est décrite par un *stencil fuzzy* (lignes 18 et 19) par la fonction `RPC_launchComputation()`. La première partition référencée par la donnée parallèle `b1` a sa géométrie paramétrée par une donnée de niveau exécution : `dim1`. La deuxième partition, référencée par `b2` a non seulement sa géométrie paramétrée par une donnée de niveau exécution mais aussi son pointeur de base. Les deux tâches créées aux lignes 23 et 24 utilisent les données parallèles floues `b1` et `b2`. Le support devra affiner dynamiquement à l'exécution les dépendances des accès relatifs aux deux appels à `RPC_compute()` mais également celles de l'accès effectué par `RPC_storeToFile()` (ligne 35) car ce flou se répercute à distance 1 dans le DAG.

5.2.2 Prise en compte des accès commutatifs

La liberté de l'ordre d'exécution autorisée par les accès commutatifs peut être pleinement exploitée pour des programmes prévisibles ou quasi-prévisibles. Cependant, la prise en compte des accès commutatifs ne peut se faire simplement car le modèle de DAG utilisé ne modélise qu'un des ordres possibles entre les tâches effectuant des accès commutatifs. Une solution simple qui pourrait fonctionner, consisterait à supprimer les dépendances associées à ces accès commutatifs, puis à utiliser une exclusion mutuelle opérant en distribué. Mais, cette solution est invalide pour une exécution statique car dans ce cas, les dépendances doivent être préalablement établies en utilisant la dernière tâche effectuant un accès en écriture. Or, dans le cas de tâches effectuant des accès commutatifs, elles sont toutes potentiellement candidates pour ce dernier accès. Une première solution consiste à ajouter une résolution dynamique des accès dépendants d'accès commutatifs.

Une deuxième solution statique est de faire modéliser par l'inspecteur l'indéterminisme induit par les accès commutatifs en supprimant l'orientation des arcs concernés du DAG. En effet, l'ordre entre les tâches effectuant des accès commutatifs n'est pas unique. L'ordonnanceur est alors chargé de choisir un ordre (orientation des arêtes) entre ces tâches qui favorise les performances.

5.3 Extension aux programmes ayant un flot de contrôle imprévisible

Dans un fonctionnement statique, une part de dynamicité peut être prise en compte à moindre coût en supposant que le DAG complet dont dispose l'exécuteur est un sur-DAG qui, par définition contient le DAG qui sera effectivement utilisé lors de l'exécution parallèle. Un exemple serait typiquement une création de tâche dans une conditionnelle dépendant de données de niveau exécution. Ceci se produit dans l'algorithme de Jacobi, où à chaque itération, le test de convergence est effectué. Ceci ne va pas sans poser de problèmes lors de la génération de ce DAG et

de son ordonnancement. L'inspecteur doit être capable d'explorer tous les scénarios d'exécution dont le nombre peut exploser combinatoirement. Les heuristiques d'ordonnancement sont également mises à mal, car elles doivent intégrer des coûts par exemple probabilistes sur la réalisation de tel ou tel scénario dans le DAG.

Lors de l'exécution de parties conditionnelles du DAG, un seul chemin est emprunté. Ainsi, dans l'arbre des possibilités d'exécution, tous les sous-arbres de contrôle non atteints doivent être élagués afin de supprimer des dépendances devenues inutiles. Si le sommet du sous-arbre à élaguer n'a pour dépendance que l'initiateur, alors tout le sous-arbre des appels de RPC peut être supprimé ainsi que les dépendances de données sortant de ce sous-arbre. Par contre, si le sommet du sous-arbre a d'autres précédences que son initiateur alors l'élagage ne peut commencer qu'à une profondeur de 1 dans le sous-arbre des appels. Ainsi, la racine du sous-arbre, bien que non exécutée, est conservée pour servir de relais aux données arrivées avant l'appel. Ce système d'élagage suppose, pour être performant, d'adopter les mêmes contraintes qu'ATHAPASCAN de propagation des données dans l'arbre d'appel : les accès d'une tâche fille sont inclus dans ceux de la tâche mère (hors données nouvellement créées par la tâche fille).

5.4 Inspection

L'inspection peut être dupliquée et effectuée par tous les processus au démarrage du programme. Ainsi, le temps pris par la diffusion du DAG est économisé par rapport à un fonctionnement avec un seul processus inspecteur. Cela est en quelque sorte le cas lorsqu'un fichier de DAG complet est chargé en parallèle par tous les processus. Une version parallèle de l'inspecteur n'est pas possible car nous ne forçons pas le programmeur à restreindre ses accès au fur et à mesure des créations de tâches imbriquées. La résolution d'une dépendance nécessite donc d'attendre que tous les accès des tâches précédentes soient connus, ce qui est intrinsèquement séquentiel.

5.4.1 Propriétés et optimisations liées au DAG de tâches complet

Toutes les informations sur les accès, allocations et libérations mémoire, contenues dans le DAG complet, permettent de vérifier des propriétés statiquement et à l'exécution. Statiquement, l'absence de fuite mémoire concernant les données parallèles pourrait être vérifiée. A l'exécution, chaque accès aux données parallèles effectué par le code de la tâche est vérifiable en changeant les modes d'accès aux pages mémoire et en vérifiant lors des exceptions levées si l'accès est conforme aux déclarations. Cette vérification ne peut se faire qu'à un niveau de granularité par page, ce qui ne permet pas de capturer finement toutes les erreurs d'accès possibles.

On peut également détecter des schémas de synchronisation autres que les diffusions et les agrégations de données (e.g. tous vers un ou tous vers tous) dans le DAG. Ces schémas de synchronisation correspondent à des appels spécifiques d'échanges collectifs implémentés de façon optimisée par certaines bibliothèques de communication (e.g. MPI). Ces schémas sont détectables car les emplacements et la géométrie mémoire des données parallèles sont décrits par les informations associées aux arcs du DAG.

Le DAG de tâches complet peut également être optimisé en le restructurant tout en conservant une modélisation valide du programme initial. La restructuration peut concerner la duplication des tâches utilisateur. Les tâches dupliquées doivent être marquées pour que l'ordonnanceur et l'exécuteur parallèle puissent les appréhender correctement. Une tâche dupliquée est exécutée par chacun des processus PRFX. Ses effets de bords extra-processus (synchronisation et cohérence) sont filtrés par le support pour garantir la validité de l'exécution. La duplication peut surtout être intéressante pour les tâches utilisateur contenant le contrôle de l'algorithme, car notre mode d'expression ne force pas le programmeur à distribuer ce contrôle (cf. section 2.7). Le programmeur a donc tendance à écrire un programme séquentiel où l'arbre de création des tâches est un père avec n fils. En dupliquant la tâche père, l'utilisateur bénéficie automatiquement d'une distribution du contrôle intrinsèque à son algorithme. Ainsi, le coût des petits messages correspondants à la gestion de ce contrôle utilisateur est économisé.

D'autres transformations du DAG, comme le changement de grain par fusion automatique des tâches, sont intéressantes pour adapter l'exécution à des machines hétérogènes. Cette transformation est réalisable si les accès des tâches aux données sont connus et si ces tâches correspondent à l'application d'opérateurs de calculs simples. Si l'on dispose de règles de fusion associées à ces routines de calcul (e.g. BLAS), le grain du DAG peut être automatiquement modulé.

L'inspecteur pourrait également modéliser le fonctionnement de PRFX dans le DAG complet. Nous avons commencé à intégrer dans ce DAG les tâches interne de communication et celles gérant la préparation et la finalisation des tâches utilisateurs. Nous pourrions rajouter des tâches de résolution des dépendances qui prendraient en paramètre la tâche à résoudre et la structure de données contenant l'historique des accès.

Le DAG complet est une structure qui occupe d'autant plus de mémoire que le grain de l'application est fin. Pour limiter la quantité de mémoire utilisée, ce DAG peut être généré par phase. Chaque nouvelle phase du DAG remplace l'ancienne en mémoire. Un autre moyen est de stocker le DAG de façon compacte. Par exemple cela est possible pour des programmes répétitifs. En effet, il suffit de modéliser une itération et de faire boucler le DAG correspondant. Des variations peuvent même être appliquées en faisant apparaître ou disparaître des tâches à partir de certains numéros d'itération.

Le DAG complet peut également être partitionné et distribué entre les processus PRFX en fonction du placement des tâches. Chaque processus doit posséder le sous DAG correspondant aux tâches que ses *threads* exécuteurs traitent. Ce sous-DAG doit être augmenté à distance 1 pour avoir localement les informations nécessaires à la satisfaction des dépendances extra-processus.

5.5 Ordonnancement et ré-ordonnancement dynamique

L'ordonnancement peut répondre à des contraintes autres que la minimisation de la durée d'exécution. Il peut faire en sorte que la somme des tailles des données parallèles utilisées à chaque instant soit inférieure à une valeur seuil. Les informations nécessaires à cette optimisation sont contenues dans le DAG complet qui décrit les allocations, les accès et les libérations des

données parallèles. Ainsi, si on suppose que les données parallèles représentent la majorité des données allouées, l'exécution parallèle peut théoriquement se contenter du volume mémoire seuil fixé.

De même que le DAG peut être généré par phase, l'ordonnancement peut suivre le même fonctionnement. Ceci permet d'entamer l'exécution parallèle plus tôt. De plus, le calcul de l'ordonnancement de la phase suivante peut se faire en parallèle avec l'exécution du programme. Enfin, les modèles de coût utilisés à chaque nouvelle phase peuvent être mis à jour.

Ces modèles de coût peuvent également être rendus plus réalistes. Par exemple les coûts des tâches peuvent prendre en compte la localisation des données dans la hiérarchie de cache mémoire d'un nœud SMP. Plus les données sont éloignées du cache et plus la tâche durera longtemps.

Les nouvelles fonctionnalités que nous voulons ajouter pour prendre en compte les programmes flous induisent que le DAG initialement ordonnancé contient une sur-estimation du nombre et du volume de données des dépendances. Ces dépendances ne sont connues de façon exacte qu'en cours d'exécution. Pour prendre en compte la nouvelle structure du DAG affiné, la partie de ce dernier qui est au-delà du front d'exécution doit être réordonnancée. Ce ré-ordonnancement serait opéré par des tâches de gestion de la plate-forme en distribué car le DAG de tâche complet est dupliqué. Ce ré-ordonnancement doit modifier les vecteurs de tâches de façon "totalement transparente" pour les *threads* exécuteurs. Ces modifications à distance seront facilitées par l'existence du support iso-mémoire. Dans les cas favorables, le ré-ordonnancement peut s'opérer sans interruption de l'exécution du programme utilisateur.

5.6 Exécution

L'exécution des vecteurs de tâches peut être rendue plus dynamique pour maximiser l'occupation du processeur. Ceci pourrait être fait lorsque l'attente du passage à l'état prêt d'une tâche est plus longue que prévue. Par exemple, les *threads* exécuteurs en attente peuvent avoir un *thread* auxiliaire qu'ils contrôlent et auquel ils assignent dynamiquement des tâches prêtes de leur vecteur d'ordonnancement. L'avantage d'un *thread* auxiliaire est qu'il peut être préempté par le système pour donner régulièrement la main au *thread* exécuteur. Dès que la tâche prévue par l'ordonnancement est prête, le *thread* exécuteur garde la main et le *thread* auxiliaire est endormi jusqu'à la prochaine attente. Ainsi, l'usage du processeur est maximisé sans pour autant nuire à l'ordonnancement.

D'autre part, les traces d'exécution peuvent être enrichies d'informations provenant des compteurs de performances intégrés aux processeurs. Le logiciel VAMPIR que nous utilisons est capable d'intégrer de telles informations. Enfin, la visualisation du déroulement de l'exécution peut être faite avec le visualiseur de graphe DaVinci. Ce dernier propose une interface pour piloter l'affichage et la construction du DAG de tâches à distance.

5.7 Extension à la grille

Le terme grille fait ici référence à une grille de calcul dédiée à ce rôle et non à des collections de machines disparates et volatiles.

Ces machines peuvent être programmées et exploitées efficacement avec PRFX si leur comportement est modélisable. En effet, la modélisation du programme par un DAG de tâche complet effectuée par PRFX est universelle. Il suffit d'utiliser une heuristique adéquate et suffisamment informée à la fois via le DAG de tâches complet, les modèles de coût et la topologie de la machine pour que l'utilisation performante d'une grille de calcul soit effective.

Pour cela, des aspects techniques doivent être résolus, mais ils sont en petit nombre car PRFX repose sur des concepts et des services simples que des environnements de type grille (e.g. Globus) fournissent. Il faudrait par exemple que l'inspecteur soit capable de reconnaître le type des données et de les encoder au format XDR (eXternal Data Representation) pour les RPC et que l'exécuteur parallèle les utilise. Un autre moyen est d'intégrer les types de données MPI (`MPI_DataType`) dans les *stencils* de PRFX car des versions de MPI dédiées à la grille existent (e.g. MPICH-G). Enfin, il est possible d'utiliser des format d'encodage liés aux *web services* comme XML-RPC ou SOAP (Simple Object Access Protocol).

L'usage d'une bibliothèque de communication unilatérale propriétaire (i.e. LAPI ou ShMem) doit être remplacé par celui d'une bibliothèque plus classique comme Nexus ou MPI supportant les communications dans un environnement de type grille. Cela ne va pas sans poser problème au niveau de l'utilisation de l'interface MPI comme nous l'avons vu (fin de section 1.2.1). Il faudrait donc programmer au-dessus de MPI des communications unilatérales répondant mieux à nos besoins que celles proposées par MPI-2.

L'iso-mémoire peut être conservée si toutes les architectures ont la même capacité d'adressage (e.g. 64 bits). En effet, son implémentation consiste juste à trouver une plage d'adresse commune d'une taille suffisante à laquelle un allocateur mémoire doit être associé.

Conclusion

Nous avons vu que l'implémentation d'algorithmes parallèles conduisant à des exécutions performantes ne suivait pas la logique du monde séquentiel. En effet, d'une part l'irrégularité d'un algorithme séquentiel ne rend pas son implémentation impossible ou problématique, et d'autre part, une fois une implémentation performante effectuée, l'utilisation d'une machine séquentielle plus puissante garantit de meilleures performances.

Cette logique est moins souvent constatée dans le domaine du parallélisme car l'architecture des machines parallèles est complexe. Des modes d'expression standards de haut niveau règlent des problèmes de portabilité (abstraction des processeurs et de la mémoire) mais assez peu les problèmes d'expression d'algorithmes parallèles irréguliers ni les problèmes de performances sur les machines à mémoire hiérarchique (e.g. grappes de SMP). Ces dernières ne sont pas originellement prises en compte car ces modes d'expression standards leur sont antérieurs. Les spécifications de ces standards sont ambitieuses car trop généralistes et simplificatrices en confiant un rôle mineur aux programmeurs si bien que le support ou le compilateur du langage sont supposés être capables de transformer des codes séquentiels annotés en des codes parallèles optimisés. Cette transformation est difficile car l'algorithme exprimé est séquentiel. Les annotations pour aider le compilateur et le support peuvent être complexes si bien que leur implémentation est parfois partielle ou non optimisée.

Les approches utilisant une modélisation de l'exécution du programme par un DAG de tâches sont, grâce à ce modèle, mieux adaptées à l'expression d'algorithmes irréguliers. Cependant, elles ne proposent au programmeur ni les avantages d'une mémoire partagée, ni l'exploitation des aspects prévisibles des algorithmes afin de les exécuter de façon performante sur des grappes de SMP.

Nous avons proposé un nouveau mode d'expression basé sur le langage C pour programmer des algorithmes parallèles notamment irréguliers. L'écriture des codes est aisée grâce à la conservation de la majorité des caractéristiques des codes séquentiels. Ce mode d'expression permet au programmeur non seulement de décrire finement le parallélisme d'une application qu'elle soit irrégulière ou non, mais il lui permet également de gérer ses données précisément en mémoire et de spécifier éventuellement le déroulement et les modalités de l'exécution. La gestion des synchronisations et de la cohérence est implicite grâce à l'utilisation d'un inspecteur traitant les informations données par le programmeur sur le découpage du programme en tâches et sur leurs accès aux données. Le déploiement de l'application est spécifiable en termes de nombre

de *threads* et de processus pour prendre en compte les grappes de SMP. Chaque *thread* est assignable à un processeur d'une machine donnée, y compris les *threads* annexes (e.g. *threads* créés par les bibliothèques utilisées par PRFX). Ce déploiement est utilisé par un algorithme d'ordonnancement des tâches et des communications. L'heuristique d'ordonnancement peut être choisie (chemin critique ou *ready time* par exemple) et appliquée avec des contraintes de placement ou de phase. La spécification du placement et de l'ordonnancement des tâches et des communications en vecteurs de tâches exécutées par chaque *thread* permet au programmeur de contrôler chaque instant de l'exécution.

Nous avons également implémenté un support pour ce mode d'expression. Il exploite un fonctionnement de type inspection / exécution où l'inspecteur modélise l'exécution de ces algorithmes parallèles par un DAG de tâches complet.

La vision globale et totale du DAG de tâches modélisant le programme permet d'appliquer des méthodes d'optimisations externes et globales à l'algorithme : placement des données, ordonnancement des calculs et des communications et équilibrage de charge.

Des tâches de gestion internes à PRFX sont mises en place pour effectuer les communications (arcs) du DAG de tâches. Une fonction de coût leur est associée et ces tâches sont prises en compte lors de l'ordonnancement.

Notre support est dédié aux grappes de nœuds SMP car ces machines offrent actuellement les meilleures puissances de calcul exploitables. Il offre des fonctionnalités pour factoriser les synchronisations et agréger les messages à destination d'un même nœud SMP. Il utilise une bibliothèque de communications unilatérales en extra-nœud et des instructions de synchronisations atomiques en intra-nœud. Ceci remplit une deuxième condition pour l'obtention de performances : un support tenant compte de la hiérarchie mémoire et réseau des grappes de SMP.

A la lumière du travail réalisé, nous avons imaginé des extensions pour le support ainsi que pour le mode d'expression. La principale voie de recherche consiste à prendre en compte des programmes quasi-prévisibles pour lesquels la disponibilité d'informations sur la majeure partie du déroulement de l'exécution permet de mettre en place statiquement des optimisations.

Dans le cadre des programmes prévisibles et quasi-prévisibles, nous commencerons tout d'abord par implémenter dans le support d'exécution actuel une gestion plus optimisée des accès commutatifs aux données. Ensuite à plus long terme, nous implémenterons un résolveur des dépendances à la volée pour raffiner les dépendances des programmes quasi-prévisibles et enfin nous implémenterons un équilibreur de charge dynamique par migration des tâches en iso-mémoire.

Bibliographie

[1] P. Amestoy, J. Roman, and F. Pellegrini. Hybridizing Nested Dissection and Halo Approximate Minimum Degree for Efficient Sparse Matrix Ordering. *Concurrency : Practice and Experience*, 12 :69–84, 2000.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks : Shared memory computing on networks of workstations. In *IEEE Computer*, volume 29(2), pages 18–28, 1996.

[3] F. André, M. Le Fur, Y. Mahéo, and J.-L. Pazat. The Pandore Data Parallel Compiler and its Portable Runtime. In *The International Conference and Exhibition on High-Performance Computing and Networking*, pages 176–183, Milan, Italy, May 1995. LNCS 919, Springer Verlag.

[4] G. Antoniu, L. Bougé, and R. Namyst. An efficient and transparent thread migration scheme in the PM2 runtime system. In *3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*, volume 1586 of *Lecture Notes in Computer Science*, pages 496–510. Springer-Verlag, April 1999.

[5] Takuya Araki, Hitoshi Murai, Tsunehiko Kamachi, and Yoshiki Seo. Optimization of HPF Programs with a Dynamic Recompilation Technique. *ISHPC*, 43 :551–562, 2002.

[6] E. Ayguade, X. Martorell, J. Labarta, M. Gonzalez, and N. Navarro. Exploiting Multiple Levels of Parallelism in OpenMP : A Case Study. In *Int. Conf. on Parallel Processing, Aizu*, 1999.

[7] B-novative. http ://www.b-novative.com.

[8] Basic Linear Algebra Subroutines (BLAS). http ://www.netlib.org.

[9] S. Benkner and T. Brandes. High-Level Data Mapping for Clusters of SMP's . In *6th International Workshop, HIPS*, volume 2026, pages 1–15, 2001.

[10] B. Bentley. Validating the Intel Pentium 4 Microprocessor. In *The International Conference on Dependable Systems and Networks (DSN'01)* , pages 493–502, 2001.

[11] Georges Bosilca, Gilles Fedak, and Franck Cappello. OVM : Out-of-order execution parallel Virtual Machine. In *Future Generation Computer Systems, FGCS*, volume 18(4), pages 525–537, 2002.

[12] L. Bougé, P. Hatcher, R. Namyst, and C. Pérez. A multithreaded runtime environment with thread migration for a HPF data-parallel compiler. In *The 1998 Intl Conf. on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 418–425, Paris, France, October 1998. IFIP WG 10.3 and IEEE.

[13] Thomas Brandes. HPF Library and Compiler Support for Halos in Data Parallel Irregular Computations. *Parallel Processing Letters 10*, 2/3 :189–200, 2000.

[14] F. Brégier. *Extensions du langage HPF pour la mise en œuvre de programmes parallèles manipulant des structures de données irrégulières*. PhD thesis, Décembre 1999.

[15] C. Brunschen. OdinMP/CCp - A Portable Compiler for C with OpenMP to C with POSIX Threads. Master's thesis, MSc Thesis, Dept of Info. Teeh. Lund University, Sweden, 1999.

[16] B. Chapman, F. Bregier, A. Patil, and A. Prabhakar. Achieving High Performance under OpenMP on ccNUMA and Software Distributed Share Memory Systems. In *Concurrency and Computation Practice and Experience*, volume 14, pages 1–17, 2002.

[17] B. Chapman, P. Mehrotra, and H.P. Zima. Vienna Fortran and the Path towards a Standard Parallel Language. *IEICE Transactions in Information and Systems*, E80-D :409–416, 1997.

[18] P. Charrier and J. Roman. Algorithmique et calculs de complexité pour un solveur de type dissections emboîtées. *Numerische Mathematik*, 55 :463–476, 1989.

[19] J. Choi, J. Dongarra, R. Pozo, and D.W. Walker. SCALAPACK : a scalable linear algebra for distributed memory concurrent computers. *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127, 1992.

[20] B. Cirou. PRFX : une plate-forme pour les applications parallèles utilisant les grappes de nœuds SMP. In *Actes de RenPar'15*, pages 101–108. INRIA, October 2003.

[21] B. Cirou, M. C. Counilh, and J. Roman. PRFX : a runtime library for high performance programming on clusters of SMP nodes. In *PARALLEL COMPUTING : Software Technology, Algorithms, Architectures & Applications*, pages 189–200. Elsevier, September 2003.

[22] B. Cirou, M. C. Counilh, and J. Roman. Programming irregular scientific algorithms with static properties on clusters of SMP nodes. In *HPSEC Proceedings*. IEEE Computer Society Press, June 2005. to be published.

[23] B. Cirou and E. Jeannot. Triplet : A clustering scheduling algorithm for heterogeneous systems. In *2001 ICPP Workshops*, pages 231–236. IEEE Computer Society, September 2001.

[24] R. Das, Y.-S. Hwang, M. Uysal, J. Saltz, and A. Sussman. Applying the CHAOS/PARTI Library to Irregular Problems in Computational Chemistry and Computational Aerodynamics. In *Proceedings of the 1993 Scalable Parallel Libraries Conference*, pages 45–56. IEEE Computer Society Press, October 1993.

[25] Chris H. Q. Ding. High Performance Fortran for Practical Scientific Algorithms : An up-to-date Evaluation. *Journal of Future Generation Computer Systems*, 15(3) :343–352, May 1999.

[26] M. Doreille. *Athapascan-1 : vers un modèle de programmation parallèle adapté au calcul scientifique*. PhD thesis, 1999.

[27] P. Feautrier. Automatic parallelization in the polytope model. In *The data parallel programming model : foundations, HPF realizations , and scientific applications*, volume 1132, pages 79–103. Springer-Verlag, Lecture Notes In Computer Science, 1996.

[28] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1) :23–53, 1991.

[29] Message Passing Interface Forum. MPI : A message-passing interface standard. Technical report, UT-CS-94-230, 1994.

[30] Cong Fu and Tao Yang. Run-time techniques for exploiting irregular task parallelism on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 42(2) :143–156, 1997.

[31] Thierry Gautier, Jean-Louis Roch, and Gilles Villard. Regular versus irregular problems and algorithms. In *IRREGULAR*, pages 1–25, 1995.

[32] A. Gerasoulis and T. Yang. PYRROS : Static scheduling and code generation for message passing multiprocessors. In *Proc. of 6th ACM International Conference on Supercomputing, Washington D.C*, pages 428–437, 1992.

[33] A. Gerasoulis and T. Yang. DSC : Scheduling parallel tasks on an unbounded number of processors. In *IEEE Transactions on Parallel and Distributed Systems*, volume 10, pages 951–967, 1994.

[34] Apostolos Gerasoulis and Tao Yang. A comparison of clustering heuristics for scheduling dags on multiprocessors. *Journal of Parallel and Distributed Computing*, 16 :276–291, 1992.

[35] Salim Hariri Haluk Topcuoglu and Min-You Wu. Task scheduling algorithms for heterogeneous processors. *8th IEEE Heterogeneous Computing Workshop (HCW'99)*, pages 3–14, April 1999.

[36] P. Hénon, F. Pellegrini, P. Ramet, and J. Roman. High Performance Complete and Incomplete Factorizations for Very Large Sparse Systems by using Scotch and PaStiX softwares. In *Eleventh SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, Californie, USA*, Feb 2004.

[37] HPF. High Performance Fortran langage specification http ://www.crpc.rice.edu/HPFF.

[38] IBM. LAPI (one-sided communication library), RSCT for AIX 5L : LAPI Programming Guide, SA22-7936.

[39] IEEE Threads POSIX 1003.1c standard. http ://www.unix-systems.org/single_unix_specification_v2/xsh/threads.html, 1995.

[40] J.F. Collard and P. Feautrier and T. Risset. Construction of DO Loops from Systems Of Affine Constraints. Technical report, Resarch Report 93-15, LIP, 1993.

[41] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, 1994.

[42] H. J. Siegel L. Wang and V. P. Roychowdhury. A genetic-algorithm-based approach for task matching and scheduling in heterogeneous computing environements. *5th IEEE Heterogeneous Computing Workshop (HCW'96)*, pages 82–95, april 1996.

[43] M. Le Fur, J.-L. Pazat, and F. André. An Array Partitioning Analysis for Parallel Loop Distribution. In *EUROPAR*, Stockholm, Sweden, August 1995. LNCS, Springer Verlag.

[44] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th Symposium on Principles of Distributed Computing*, pages 229–39, Calgary, Alberta, Canada, August 1986.

[45] B. Lisper. Detecting sttaic algorithms by partial evaluation. In *ACM Sigplan Symposium on Partial Evaluation and Semi-Based Program Manipulation*, pages 31–42, 1991.

[46] MOSIX. http ://www.mosix.org, 2004.

[47] R. Namyst. *PM2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières.* PhD thesis, Université de Lille 1, 1997.

[48] G. Narlikar and G. Blelloch. Pthreads for Dynamic and Irregular Parallelism. In *Proceedings of SC98 : High Performance Networking and Computing*, Orlando, Florida, November 1998.

[49] Oshinori Ojima, Mitsuhisa Sato, Hiroshi Harada, and Yutaka Ishikawa. Performance of Cluster-enabled OpenMP for the SCASH Software Distributed Shared Memory System. In *CCGRID*, pages 450–456, 2003.

[50] Omni project. http ://phase.hpcc.jp/Omni/home.html, 2004.

[51] OpenMP Language Application Program Interface Specification. http ://www.openmp.org.

[52] Pallas. http ://www.pallas.com.

[53] F. Pellegrini, J. Roman, and P. Amestoy. Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering. In *Proceedings of IRREGULAR'99, Puerto Rico, LNCS 1586*, pages 986–995, April 1999.

[54] M. Rinard. *The Design, Implementation and Evaluation of Jade, a Portable, Implicitly Parallel Programming Language*. PhD thesis, Dept. of Computer Science, Stanford University, 1994.

[55] M. C. Rinard. Applications experience in jade. *Concurrency Practice & Experience*, 10(6) :417–448, May 1998.

[56] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of jade. *ACM Transactions on Programming Languages and Systems*, 20(3) :483–545, May 1998.

[57] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A simple load balancing scheme for task allocation in parallel machines. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, 1991.

[58] V. Sarkar. *Partitioning and scheduling parallel programs for execution on multiprocessors*. The MIT Press, Cambridge,MA, 1989.

[59] D. J. Scales, K. Gharachorloo, and C. Thekkath. Shasta : A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–85, Cambridge, Massachusetts, October 1996. Also available as DEC WRL Research Report 96/2.

[60] SCore. http ://www.pccluster.org, 2004.

[61] SGI IRIX. http ://techpubs.sgi.com, 2004.

[62] Silicon Graphics Inc. Parallel processing on Origin series. System MIPSpro 7 Fortran 90 Commands and Directives Reference Manual. In *http ://techpubs.sgi.com*, 2004.

[63] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L : Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the 16th Symposium on Operating Systems Principles*, pages 170–83, Saint Malo, France, October 1997.

[64] T. Brandes. ADAPTOR : OpenMP Programmer's Guide. http ://www.scai.fhg.de/EP-CACHE/adaptor/ompguide/ompguide.html, 2004.

[65] T. Brandes. ADAPTOR Parallel Fortran Compilation System. http ://www.scai.fhg.de/EP-CACHE/adaptor/www/adaptor_home.html, 2004.

[66] J. D. Ullman. NP-complete scheduling problem. *Journal of Computer and System Sciences*, 10 :384–393, june 1975.

[67] Dhabaleswar K. Panda Vladimir Yarmolenko, Jose Duato and P. Sadayappan. Characterization and enhancement of static mapping heuristics for heterogeneous systems. *Proceedings of the 7th International Conference on High Performance Computing*, pages 37–48, 2000.

[68] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS : A multigrain shared memory system. In *Proc. of the 23rd Annual Int'l Symp. on Computer Architecture (ISCA'96)*, pages 45–55, 1996.

[69] H. Zima and B. Chapman. Supercompiler for parallel vectors computers. 1991.

Annexe A

Extraits de code

A.1 Langage OpenMP

A.1.1 Jacobi SPMD

Listing A.26 – Implémentation SPMD de l'algorithme de Jacobi avec OpenMP sur un maillage du plan avec une distribution 1D.

```
1       subroutine jacobi (n,m,dx,dy,alpha,omega,u,f,tol,maxit)
2       integer n,m,maxit
3       double precision dx,dy,f(n,m),u(n,m),alpha, tol,omega
4   !Local variables
5       integer i,j,k,k_priv
6       double precision error,resid,rsum,ax,ay,b
7       double precision error_priv, uold(n,m)
8
9   !Initialize coefficients
10      ax = 1.0/(dx*dx)          ! X-direction coef
11      ay = 1.0/(dy*dy)          ! Y-direction coef
12      b  = -2.0/(dx*dx)-2.0/(dy*dy) - alpha ! Central coeff
13
14  !No Worksharing Do Construct No Worksharing Do Construct
15
16      nthreads = omp_get_max_threads()
17      jlo = 2
18      jhi = m-1
19      nrem = mod ( jhi - jlo + 1, nthreads )
20      nchunk = ( jhi - jlo + 1 - nrem ) / nthreads
21
22  !$omp parallel private(me,js,je,resid,k_priv,err_priv)
23      me = omp_get_thread_num()
24      if ( me < nrem ) then
25         js = jlo + me * ( nchunk + 1 )
26         je = js + nchunk
27      else
28         js = jlo + me * nchunk + nrem
29         je = js +nchunk - 1
30      end if
31      k_priv = 1
32      error_priv = 10.0 * tol
33
34      do while (k_priv .le. maxit .and. error_priv .gt. tol)
35         do j=js,je
36            do i=1,n;
37               uold(i,j) = u(i,j)
38            enddo
39         enddo
40  !$omp barrier
41  !$omp single
42         error = 0.0
43  !$omp end single
44         error_priv = 0.0
45         do j = js,je
46            do i = 2,n-1
47               resid = (ax*(uold(i-1,j) + uold(i+1,j))
48               &      + ay*(uold(i,j-1) + uold(i,j+1))
```

```
49                &      + b * uold(i,j) - f(i,j))/b
50                u(i,j) = uold(i,j) - omega * resid
51                error_priv = error_priv + resid*resid
52             enddo
53          enddo
54
55  !$omp critical
56          error = error + error_priv
57  !$omp end critical
58          k_priv = k_priv + 1
59  !$omp barrier
60          error_priv = sqrt(error)/dble(n*m)
61        enddo
62  !$omp single
63  k = k_priv; error = error_priv
64  !$omp end single nowait
65  !$omp end parallel
```

A.2 Langage HPF

A.2.1 Jacobi

Listing A.27 – Implémentation de l'algorithme de Jacobi avec HPF sur un maillage du plan avec une distribution 1D.

```
1         PROGRAM LAPLACE
2         implicit NONE
3   c     solving the laplace pde
4         real, parameter :: EPS = 0.001
5         integer, parameter :: MAXX = 96, MAXY = 96
6         integer :: ITER
7         integer :: I, J
8         real, allocatable, dimension (:,:) :: F
9   !hpf$ distribute (*,block) :: F
10
11  c     allocating memory
12        ALLOCATE ( F (MAXX,MAXY))
13
14  c     initialization
15        F = 2.
16        F(:,MAXY) = 1.
17        forall (J=2:MAXY-1,I=2:MAXX-1)
18           F(I,J) =  0
19        end forall
20
21  !     iterate  with shadow edges
22        call ITERATE (F, MAXX, MAXY, EPS, ITER)
23        print *, 'RUN ', ITER, ' Iterations'
24        deallocate (F)
25        END
26
27  !     the following routine uses SHADOW EDGES
28
29        SUBROUTINE ITERATE (F, N1, N2, EPS, NITER)
30
31        implicit NONE
32
33        integer, intent (in)  :: N1, N2
34        real,    intent (in)  :: EPS
35        integer, intent (out) :: NITER
36
37        real, dimension (N1,N2) :: F, DF
38
39  !hpf$ distribute F(*,block)
40  !hpf$ align DF(:,:) with F(:,:)
41
42        real    :: FMAX
43        integer :: I, J
44
45        DF   = 0.0
46        NITER = 0
47        FMAX = 1.0
48
49  c     iteration
50
51        DO WHILE (FMAX .gt. EPS)
52
53           NITER = NITER + 1
```

```
54
55  !adp$     profile FORALL
56            forall (J=2:N2-1,I=2:N1-1)
57              DF(I,J) = ( F(I,J+1) + F(I,J-1)
58       &              +   F(I-1,J) + F(I+1,J) ) * 0.25 -F (I,J)
59            end forall
60
61  !adp$     profile SUM
62            forall (J=2:N2-1,I=2:N1-1)
63               F(I,J) = F(I,J) + DF(I,J)
64            end forall
65
66            DF = ABS(DF)
67  !adp$     profile MAX
68            FMAX = MAXVAL (DF)
69
70        END DO
71
72        END
```

A.3 Bibliothèque RAPID

A.3.1 Factorisation de Cholesky de matrices creuses avec un découpage par blocs bi-dimensionnel

Listing A.28 – Implémentation d'une factorisation de Cholesky par blocs 2D de matrices creuses avec la bibliothèque RAPID.

```
1   #include <stdio.h>
2   #include <rapid.h>
3   #include "block.h"
4
5   int chol_reg (char *dagname, int nblock, COL_LINK **blk_index)
6   {
7     int i, j, k;
8
9     set_clusteropt (USERCLUSTER);
10    dag_begin (dagname);
11
12    for (j = 1; j <= nblock; ++j)
13      {
14        i = j;
15        while (i)
16          {
17            object ("blk", DataSize (i, j), 2, i, j);
18            mark_commute ("blk", 2, i, j);
19            i = GetNextNonzero (blk_index, i, j, 0);
20          }
21        object ("Linv", DataSize (j, j), 1, j);
22      }
23
24    for (k = 1; k <= nblock; ++k)
25      {
26        unmark_commute ("blk", 2, k, k);
27        task_begin ("factor_kk", TaskSize (k, k, k), 1, k);
28        set_cluster (k, k);
29        task_read ("blk", 2, k, k); task_write ("blk", 2, k, k); task_write ("Linv", 1, k);
30        task_end ();
31
32        j = GetNextNonzero (blk_index, k, k, 0);
33        while (j)
34          {
35            unmark_commute ("blk", 2, j, k);
36            task_begin ("scale_ik", TaskSize (j, k, k), 2, j, k);
37            set_cluster (j, k);
38            task_read ("blk", 2, j, k); task_read ("Linv", 1, k);
39            task_write ("blk", 2, j, k);
40            task_end ();
41            j = GetNextNonzero (blk_index, j, k, 0);
42          }
43        j = GetNextNonzero (blk_index, k, k, 0);
44        while (j)
45          {
46            i = j;
47            while (i)
48              {
```

```
49                task_begin ("modify_ij", TaskSize (i, j, k), 3, i, j, k);
50                set_cluster (i, j);
51                task_read ("blk", 2, i, k); task_read ("blk", 2, j, k); task_read ("blk", 2, i, j);
52                task_write ("blk", 2, i, j);
53                task_end ();
54                i = GetNextNonzero (blk_index, i, k, 1);
55              }
56            j = GetNextNonzero (blk_index, j, k, 0);
57          }
58      }
59    task_func ("factor_kk", factor_kk);
60    task_func ("scale_ik", scale_ik);
61    task_func ("modify_ij", modify_ij);
62    dag_end ();
63  }
64
65  main (int argc, char *argv[])
66  {
67    int ndim;                       /* order of the element-wise matrix */
68    int nblock;                     /* order of the block-wise matrix */
69    int nonzeros;                   /* number of the nonzero elements */
70    int nzblocks;                   /* number of the nonzero blocks */
71    int i, bsize = BSIZE;
72    [...]
73    long exec_time;
74
75    rapid_init (&argc, argv);
76    fortran_init ();
77
78    if (argc < 3)
79      {
80        fprintf (stderr, "Usage: %s matri-file permute-file [maximum-block-size]\n
81  ", argv[0]);
82        exit (0);
83      }
84
85    /* read in matrix from file argv[1] */
86    org_index = ReadStructure (argv[1], &ndim, &nonzeros);
87    /* do symbolic factorization */
88    col_index = SymbolicFact (org_index, ndim, &col_nz, &ET_ele, &nonzeros);
89
90    if (argc >= 4)
91      bsize = atoi (argv[3]);
92
93    /* do supernode partitioning and construct the block structure */
94    FindSuperNodes (ndim, ET_ele, col_index, col_nz, &nblock,
95                    &super, &start, &perm, &invp, bsize, 1);
96    blk_index = BuildBlockIndex (ndim, col_index, super, start, perm, invp,
97                                 nblock, &blk_nz, &ET_blk, &nzblocks);
98    /* free element wise index structure of F */
99    FreeStructre (ndim, col_index, col_nz, ET_ele);
100
101   /* print out the permutation vector generated by supernode amalgamation */
102   PrintPermute (argv[2], ndim, perm);
103
104   chol_reg ("sparse_chol", nblock, blk_index);
105   dag_sched ("sparse_chol");
106
107   InitDiagDom (ndim, org_index, nblock, blk_index, invp);
108
109   rapid_barrier ();
110   exec_time = clock ();
111   exec_dag ("sparse_chol");
112
113   exec_time = clock () - exec_time;
114   if (!me_no)
115     fprintf (stderr, "The execution time of the dag is %d msecs.\n", exec_time /
116  1000);
117
118   PrintFactor (ndim, nblock, blk_index);
119 }
```

A.4 Bibliothèque ATHAPASCAN

A.4.1 Interface et implémentation d'une matrice de blocs et de ses opérateurs de calcul (classe `matrix`)

Listing A.29 – ATHAPASCAN : interface module matrix définissant opérations matricielles (`matrix.h`).

```
#ifndef _MATRIX_H_
#define _MATRIX_H_

#include <iostream>
#include <vector>
#include <algorithm>
#include <typeinfo>

//-------------------------------------------------------------------
//Class matrix (stockage par colonne)

template<int s>
struct bloc_list {
  struct item {
    item* next;
    int dummy1;
  };
  static void* allocate()
  {
    if (head ==0) {
      //logm << s << ' ';
      item* r = (item*)new char[s+sizeof(item)];
      r->next = 0;
      return sizeof(item) + (char*)r;
    }
    item* r = head; head = r->next;
    return sizeof(item) + (char*)r;
  }
  static void deallocate( void* p )
  {
    item* r = (item*)(-sizeof(item) + (char*)p);
    //delete [] r; return;
    r->next = head;
    head = r;
  }
  static item* head;
};
template<int s>
bloc_list<s>::item* bloc_list<s>::head =0;

template<class T>
struct MyAllocator {
  typedef size_t    size_type;
  typedef ptrdiff_t difference_type;
  typedef T*        pointer;
  typedef const T*  const_pointer;
  typedef T&        reference;
  typedef const T&  const_reference;
  typedef T         value_type;
  pointer address( reference x) const { return &x; }
  const_pointer address( const_reference x) const { return &x; }

  template <class _Tp1> struct rebind {
    typedef MyAllocator<_Tp1> other;
  };

  void construct( pointer p, const T& val )
  { new (p) T(val); }

  bool operator==( const MyAllocator<T>& A) { return true; }
  T* allocate( size_type n )
  {
    //llogm<T>() << n << ' ';
    if (n ==0) return 0;
    if (n == 16384) return (T*)bloc_list<16384*sizeof(T)>::allocate();
    if (n == 64) return (T*)bloc_list<64*sizeof(T)>::allocate();
    return new T[n];
  }

  void deallocate( pointer p, size_type n )
  {
    if (n ==0) return;
```

```
73      if (n == 16384) { bloc_list<16384*sizeof(T)>::deallocate(p); return; }
74      if (n == 64) { bloc_list<64*sizeof(T)>::deallocate(p); return; }
75      delete [] p;
76    }
77
78    void destroy(pointer p) {p->~T();}
79    size_type max_size() const throw(){return 10000000;};
80
81  };
82
83  #endif
84
85  template<class T>
86  class matrix : public std::vector<T/*,MyAllocator<T>*/ > {
87  public:
88
89    typedef typename std::vector<T/*,MyAllocator<T>*/> inherited;
90    typedef typename inherited::iterator iterator;
91    typedef typename inherited::const_iterator const_iterator;
92
93    matrix(int row_dim_ = 0, int col_dim_ = 0, const T& value = T())
94        : inherited(col_dim_*row_dim_,value), _col_dim(col_dim_), _row_dim(row_dim_) {};
95
96    int col_dim() const { return _col_dim; }
97    int row_dim() const { return _row_dim; }
98
99
100   const T& operator()(int i, int j) const { return operator[](i+j*_row_dim); }
101   T& operator()(int i, int j) { return operator[](i+j*_row_dim); }
102
103   void realloc(int row_dim_ = 0, int col_dim_ = 0) {
104     _col_dim = col_dim_;
105     _row_dim = row_dim_;
106     if(size() < col_dim_*row_dim_)
107       insert(end(),col_dim_*row_dim_-size(),T());
108     if(size() > col_dim_*row_dim_)
109       erase(begin()+col_dim_*row_dim_,end());
110   }
111
112   void alloc() {
113     if( empty() ) {
114       inherited tmp(_row_dim * _col_dim,0);
115       inherited::swap(tmp);
116     }
117   }
118
119   matrix<T>& init_value( T val_diag );
120   matrix<T>& init_value_norand( T val_diag );
121
122
123   matrix<T>& submatrix(const matrix<T>& a,int i1, int j1, int i2, int j2) {
124     realloc(i2-i1,j2-j1);
125     iterator t = begin();
126     const_iterator j;
127     for(j=a.begin()+j1*a._row_dim; j<a.begin()+j2*a._row_dim; j+=a._row_dim)
128       t = copy(j+i1,j+i2,t);
129     return *this;
130   }
131
132   matrix<T>& set_size(int col_dim, int row_dim) {
133     _col_dim = col_dim;
134     _row_dim = row_dim;
135     return *this;
136   }
137
138
139   void insert_submatrix(iterator t, const matrix<T>& m) {
140     const_iterator t_stop = t+row_dim()*m.col_dim();
141     const_iterator i = m.begin();
142     for( ; t < t_stop; t+= row_dim()) {
143       copy(i, i+m.row_dim(), t);
144       i += m.row_dim();
145     }
146   }
147
148   void destroy();
149
150   //protected:
151   int _col_dim;
152   int _row_dim;
153 };
154
155
156
157 template<class T>
158 a1::OStream& operator<<  (a1::OStream& out, const matrix<T>& m) {
159   out << m._col_dim << m._row_dim;
160
```

```
161     if (m.size() ==0)
162       out << (int)0;
163     else {
164       out << (int)1;
165       out.write( a1::OStream::DA, &m[0], sizeof(T)*m._col_dim*m._row_dim);
166     }
167     return out;
168   }
169
170   template<class T>
171   class myvector : public std::vector<T/*,MyAllocator<T>*/ >
172   {
173   public:
174           myvector(  int sz, T* data )
175           {
176                   /* not ok for intel compiler
177                   _M_start = data;
178                   _M_end_of_storage = _M_start + sz;
179                   _M_finish = _M_end_of_storage;
180                    */
181           }
182   };
183
184   template<class T>
185   a1::IStream& operator>>  (a1::IStream& in, matrix<T>& m) {
186     int s;
187     in >> m._col_dim >> m._row_dim >> s;
188
189     int size = m._row_dim * m._col_dim;
190     if (s ==0) return in;
191
192     // here in case of the WS protocol
193
194     void* tmp =0;
195     if (m.size() != size) {
196       m.resize(size);
197     }
198           tmp = &m[0];
199     in.read( a1::IStream::DA, tmp, sizeof(T)*size);
200
201     return in;
202   }
203
204
205
206
207   template<class T>
208   void _two_dim_partitioning(matrix<T>& a, int bloc_size,
209                              matrix<matrix<T> >& res
210                              ) {
211     int nb_bloc_row = a.row_dim()/bloc_size + ( a.row_dim()%bloc_size == 0 ? 0 : 1 );
212     int nb_bloc_col = a.col_dim()/bloc_size + ( a.col_dim()%bloc_size == 0 ? 0 : 1 );
213     res = matrix<matrix<T> >(nb_bloc_row,nb_bloc_col);
214     for(int i = 0; i<res.row_dim(); i++)
215       for( int j = 0; j<res.col_dim(); j++) {
216         res(i,j).submatrix(a, i*bloc_size, j*bloc_size,
217                               min((i+1)*bloc_size,a.row_dim()),
218                               min((j+1)*bloc_size,a.col_dim())
219                               );
220       }
221   }
222
223   template<class T>
224   matrix<matrix<T> > two_dim_partitioning(matrix<T>& a, int bloc_size) {
225     matrix<matrix<T> > tmp;
226     _two_dim_partitioning(a,bloc_size,tmp);
227     return tmp;
228   }
229
230   template<class T>
231   void _regroup(const matrix<matrix<T> >& m, matrix<T>& res)
232   {
233     int col_dim = 0;
234     int row_dim = 0;
235     {
236       for(int j = 0; j<m.col_dim(); j++)
237         col_dim += m(0,j).col_dim();
238       for( int i = 0; i<m.row_dim(); i++)
239         row_dim += m(i,0).row_dim();
240     }
241
242     res = matrix<T>(row_dim,col_dim);
243     typename matrix<T>::iterator t = res.begin();
244
245     for(int j = 0; j<m.col_dim(); j++) {
246       for( int i = 0; i<m.row_dim(); i++) {
247         res.insert_submatrix(t, m(i,j));
248         t += m(i,j).row_dim();
```

```
249       }
250       t += ( m(0,j).col_dim() - 1 ) * res.row_dim();
251     }
252   }
253
254   template<class T>
255   matrix<T> regroup(const matrix<matrix<T> >& m)
256   {
257     matrix<T> tmp;
258     _regroup(m,tmp);
259     return tmp;
260   }
261
262
263   template<class T>
264   std::ostream& operator<<(std::ostream& out, const matrix<T>& m) {
265     typename matrix<T>::const_iterator ptr = m.begin();
266     std::cout << "array([" << std::endl;
267     for(int j = 0; j<m.col_dim(); ++j)
268     {
269       std::cout << "[ " ;
270       for(int i = 0; i < m.row_dim()-1; i++) {
271         std::cout << *ptr << ", ";
272         ++ptr;
273       }
274       std::cout << *ptr;
275       ++ptr;
276       if (j+1 < m.col_dim())
277         std::cout << "]," << std::endl;
278       else
279         std::cout << "]" << std::endl;
280     }
281     std::cout << "]);" << std::endl;
282     return out;
283   }
284
285
286   template<class T>
287   void matrix<T>::separeLU( matrix<T>& L, matrix<T>& U ) const
288   {
289   //   std::cout << "A=" << *this << endl;
290   //   std::cout << "row_dim=" << row_dim() << ", col_dim=" << col_dim() << std::endl;
291     matrix<T> tL( row_dim(), col_dim(),0 );
292     matrix<T> tU( row_dim(), col_dim(),0 );
293   //   std::cout << "L=" << tL << endl;
294   //   std::cout << "U=" << tU << endl;
295     for (int j=0; j< col_dim(); ++j)
296     {
297       for (int i=0; i< row_dim(); i++)
298       {
299   //std::cout << ptr << ", A[" << i << "," << j << "]=" << *ptr << std::endl;
300         if (i ==j) {
301           tU(i,j) = 1.0;
302           tL(i,j) = (*this)(i,j); //*ptrL = *ptr;
303         }
304         else if (i > j) {
305           tU(j,i) = 0.0;
306           tL(j,i) = (*this)(j,i);
307         } else if (i < j) {
308           tL(j,i) = 0.0;
309           tU(j,i) = (*this)(j,i);
310         }
311   //std::cout << ptrL << ", L[" << i << "," << j << "]=" << *ptrL << std::endl;
312   //std::cout << ptrU << ", U[" << i << "," << j << "]=" << *ptrU << std::endl;
313       }
314     }
315   //   std::cout << "L:=array(" << tL << ");" << endl;
316   //   std::cout << "U:=array(" << tU << ");" << endl;
317     L.swap(tL);
318     U.swap(tU);
319   }
320
321   #  include "matrix.inl"
322
323   #endif
```

Listing A.30 – ATHAPASCAN : implémentation du module matrix (`matrix.inl`).

```
1   template<class T>
2   inline matrix<T>& matrix<T>::minus_times_equal(const matrix<T>& a, const matrix<T>& b) {
3     matrix_trace_calc("minus_times_equal");
4     const_iterator ptr1_a = a.begin();
5     for(const_iterator ptr1_b = b.begin();
6         ptr1_b< b.begin()+b._row_dim;
7         ptr1_b++, ptr1_a += a._row_dim
```

```
8         ) {
9       iterator ptr_tmp = begin();
10      for(const_iterator ptr2_b = ptr1_b;
11          ptr2_b< b.end();
12          ptr2_b+=b._row_dim
13          ) {
14        T tmp_b = *ptr2_b;
15        const_iterator ptr_stop_a = ptr1_a+a._row_dim;
16        for(const_iterator ptr2_a = ptr1_a;
17            ptr2_a < ptr_stop_a;
18            ptr2_a++, ptr_tmp++
19            )
20          *ptr_tmp -= *ptr2_a * tmp_b;
21      }
22    }
23    return *this;
24  }
25
26  template<class T>
27  inline matrix<T>& matrix<T>::factorization_LU()
28  {
29    matrix_trace_calc("factorization_LU");
30
31    int k_int;
32    int k_int_stop = std::min(row_dim(),col_dim());
33    iterator k;
34
35    for ( k = begin(), k_int = 0;
36          k_int < k_int_stop;
37          k += row_dim(), k_int++
38         ) {
39      T pivot = _ONE_ / *(k+k_int);
40      iterator i_stop =  k + row_dim();
41      iterator  i;
42      for (i = k+k_int+1 ; i < i_stop ; i++ )
43        *i *= pivot;
44      for (iterator j= k + row_dim() ; j < end() ; j+= row_dim() ) {
45        iterator i_stop = j + row_dim();
46        iterator k_j = j + k_int;
47        iterator i_k;
48        for (i = j + k_int + 1, i_k = k + k_int + 1;
49             i < i_stop ;
50             i++,i_k++ )
51          *i -=  *i_k * *k_j;
52      }
53    }
54    return *this;
55  }
56
57  template<class T>
58  inline matrix<T>& matrix<T>::this_times_inverse_U(const matrix<T>& u)
59  {
60    matrix_trace_calc("this_times_inverse_U");
61
62    int j;
63    iterator ptr_j;
64    const_iterator u_j_j = u.begin();
65    for(j = 0, ptr_j = begin();
66        j<col_dim();
67        j++, ptr_j+=row_dim(), u_j_j++
68        ) {
69
70      T tmp = _ONE_ / *u_j_j;
71      iterator l_stop = ptr_j + row_dim();
72      for(iterator l = ptr_j; l<l_stop; l++)
73        *l *= tmp;
74
75      iterator this_l_t = ptr_j + row_dim();
76       u_j_j += u.row_dim();
77      for( const_iterator u_j_t = u_j_j;
78           u_j_t<u.end();
79           u_j_t+=u.row_dim()
80           ) {
81        T tmp1 = *u_j_t;
82        iterator this_l_j_stop = ptr_j + row_dim();
83        for(iterator this_l_j = ptr_j;
84            this_l_j < this_l_j_stop;
85            this_l_j++, this_l_t++
86            )
87          *this_l_t -= *this_l_j * tmp1;
88      }
89    }
90    return *this;
91  }
92
93  template<class T>
94  inline matrix<T>& matrix<T>::inverse_L_times_this(const matrix<T>& u)
95  {
```

```
96    matrix_trace_calc("inverse_L_times_this");
97
98    for(iterator l = begin(); l<end(); l+=row_dim()) {
99      const_iterator u_t_i = u.begin();
100     for(int i = 0; i <row_dim(); i++) {
101       T tmp = *(l+i);
102       iterator this_t_l = l+i+1;
103       iterator this_t_l_stop = l+row_dim();
104       u_t_i += i+1;
105       for(; this_t_l < this_t_l_stop; this_t_l++, u_t_i++)
106         *this_t_l -= tmp * *u_t_i;
107     }
108   }
109   return *this;
110 }
```

A.4.2 Factorisation LU de matrices pleines avec un découpage par bloc bi-dimensionnel utilisant la classe `matrix`

Listing A.31 – Partie d'implémentation pour la factorisation LU par blocs 2D de matrices pleines avec la bibliothèque ATHAPASCAN (`LU.cpp`).

```
1   #include <iostream>
2   #include <math.h>
3   #include <cassert>
4   #include "athapascan-1.h"
5   #include "matrix.h"
6
7   typedef double TYPEREAL;
8
9   struct BlocC2D : Distribution
10  {
11    // p nb ligne par bloc, q nb col par bloc
12    // nb ligne total = n
13    BlocC2D (int p, int q, int n) : _p(p), _q(q), _n(n){}
14    int distribute(int i, int j)
15    {
16      return (int)((i/_q) * (_n/_p) + (j/_p));
17    }
18  private:  int _p;  int _q;  int _n;
19  };
20
21  struct my_sync { //: public a1_task {
22    my_sync( int cost = 0 ) {} //: a1_task(cost)  {}
23    my_sync( int x, int y ) {} //: a1_task(x,y) {}
24    struct add
25    {
26      void operator()(  my_sync&, const  my_sync&) {}
27    };
28    void set_null() {};
29  };
30
31  a1::OStream& operator<<( a1::OStream& out, const my_sync& x) {
32    int i = 1;
33    return out;
34  };
35
36  a1::IStream& operator>>( a1::IStream& in, my_sync& x) {
37    int i;
38    return in;
39  };
40
41  a1::OStream& operator<<( a1::OStream& out, const std::string& x) {
42    return out;
43  };
44
45  a1::IStream& operator>>( a1::IStream& in, std::string& x) {
46    return in;
47  };
48
49  template<class T> void bloc_factorization_LU(matrix<T>& A, BlocCyclic2D& bc)
50  {
51    int i, j, k;
52    for ( k=0 ; k < A.row_dim() ; k++ ) {
53      A(k,k).factorization_LU(bc);
54      for ( i=k+1 ; i < A.col_dim() ; i++ )
55        A(i,k).this_times_inverse_U(A(k,k), bc);
56      for ( j=k+1 ; j < A.row_dim() ; j++ )
57        A(k,j).inverse_L_times_this(A(k,k), bc);
```

```
58      for (i=k+1 ; i < A.col_dim() ; i++ ) {
59        for (j=k+1 ; j < A.row_dim() ; j++ )
60          A(i,j).minus_times_equal( A(i,k), A(k,j), bc );
61      }
62    }
63  }
64
65  template<class T> void minus_times_equal( matrix<T>& A, matrix<T>& L, matrix<T>& U,
66                                            BlocCyclic2D& bc)
67  {
68    int i, j, k;
69    for ( i=0 ; i < A.row_dim() ; i++ )
70      for ( j=0 ; j < A.col_dim() ; j++ )
71        for ( k=0 ; k < A.col_dim() ; k++ )
72        {
73          A(i,j).minus_times_equal(L(i,k), U(k,j), bc );
74        }
75  }
76
77  template<class T> class matrix_sh : public a1::Shared<matrix<T> >
78  {
79  public:
80    matrix_sh()
81      : Shared<matrix<T> >() {}
82
83    matrix_sh( matrix<T>*m, int _x, int _y )
84      : a1::Shared<matrix<T> >( m ),
85              x(_x), y(_y), cost( m->row_dim() * m->row_dim() * m->col_dim() ), _size_bloc(m->row_dim() * m->col_dim()) {}
86
87    void factorization_LU(BlocCyclic2D& bc) {
88      a1::Fork<task_factorisation_LU>(Distribute(bc, x, y))
89      //a1::Fork<task_factorisation_LU>(SetCost((double)cost))
90          ((a1::Shared<matrix<T> >&)*this);
91    }
92
93    void this_times_inverse_U( a1::Shared<matrix<T> >& u, BlocCyclic2D& bc) {
94      a1::Fork<task_m_times_inverse_U> (Distribute(bc, x, y))
95      //a1::Fork<task_m_times_inverse_U> (SetCost((double)cost))
96          ((a1::Shared<matrix<T> >&)*this, u);
97    }
98
99    void inverse_L_times_this( a1::Shared<matrix<T> >& l, BlocCyclic2D& bc) {
100     a1::Fork<task_inverse_L_times_m>(Distribute(bc, x, y))
101         ((a1::Shared<matrix<T> >&)*this, l);
102   }
103
104   void minus_times_equal( a1::Shared<matrix<T> >& a,  a1::Shared<matrix<T> >& b, BlocCyclic2D& bc) {
105     a1::Fork<task_minus_times_equal>(Distribute(bc, x, y))
106     //a1::Fork<task_minus_times_equal>(SetCost((double)cost))
107         ((a1::Shared<matrix<T> >&)*this, a, b );
108   }
109
110   void copyto( a1::Shared<matrix<T> >& dest ) {
111     a1::Fork<task_copyto >() ((a1::Shared<matrix<T> >&)*this, dest );
112   }
113
114 private:
115   struct task_factorisation_LU {
116     task_factorisation_LU( int cost = 0 )
117       {}
118     task_factorisation_LU( int x, int y )
119       {}
120     void operator()(Shared_r_w<matrix<T> > a) {
121       a.access().factorization_LU();
122     }
123   };
124   struct task_m_times_inverse_U {
125     task_m_times_inverse_U( int cost = 0 )
126       {}
127     task_m_times_inverse_U(  int x, int y )
128       {}
129     void operator()(Shared_r_w<matrix<T> > m, Shared_r<matrix<T> > u) {
130       m.access().this_times_inverse_U(u.read());
131     }
132   };
133   struct task_inverse_L_times_m {
134     task_inverse_L_times_m( int cost = 0 )
135       {}
136     task_inverse_L_times_m(  int x, int y )
137       {}
138     void operator()(Shared_r_w<matrix<T> > m, Shared_r<matrix<T> > l) {
139       m.access().inverse_L_times_this(l.read());
140     }
141   };
142
143   struct task_minus_times_equal {
144     task_minus_times_equal( int cost = 0 )
145       {}
```

```
146       task_minus_times_equal( int x, int y )
147         {}
148       void operator()(Shared_cw<add,matrix<T> > a,
149                       Shared_r<matrix<T> > b,
150                       Shared_r<matrix<T> > c) {
151         matrix<T> tmp(b.read().row_dim(),c.read().col_dim(),T(0));
152         a.cumul(tmp.times(b.read(), c.read()));
153       }
154     };
155
156     int x, y;  int cost;  int _size_bloc;
157   };
158
159   template<class T> void two_dim_partitioning_any_mode(
160      int size, int bloc_size,
161      matrix<matrix_sh<T> >& res,
162           BlocCyclic2D& bc
163   )
164   {
165     matrix<matrix_sh<T> > tmp(
166        size/bloc_size + ( size%bloc_size == 0 ? 0 : 1 ),
167        size/bloc_size + ( size%bloc_size == 0 ? 0 : 1 )
168     );
169     res.swap(tmp);
170     for( int j = 0; j<res.col_dim(); j++)
171       for(int i = 0; i<res.row_dim(); i++) {
172         res(i,j) = matrix_sh<T>
173           ( & ((new matrix<T>)->set_size(  std::min((i+1)*bloc_size,size) - i*bloc_size,
174                                            std::min((j+1)*bloc_size,size) - j*bloc_size
175                                         )),
176             i, j
177             );
178         res(i,j).generate( i==j ? 2*size : 1, bc);
179       }
180   }
181
182   int main(int argc, char** argv)
183   {
184     a1::Community com = a1::System::create_initial_community( argc, argv );
185     a1::Timer tim;
186     tim.start();
187
188     int n      = atoi( argv[1] );
189     int p      = atoi( argv[2] );
190     int K      = atoi( argv[3] );
191     int proc_x = atoi(argv[4]);
192     int proc_y = atoi(argv[5]);
193     int size_C = n%p == 0 ? n/p : n/p + 1;
194
195     matrix<TYPEREAL> IN;
196
197     matrix<matrix_sh<TYPEREAL> > OUT3;
198     BlocCyclic2D bc( proc_x , proc_y);
199
200     two_dim_partitioning_any_mode( n, p, OUT3,  bc  );
201
202     tim.start();
203     bloc_factorization_LU(OUT3, bc);
204     com.sync();
205     tim.stop();
206     if (com.is_leader())
207       {
208         std::cout << i << "::time " << p << " " << n << " "<< tim.realtime() << std::endl;
209         double mflop = n;
210         mflop = ((mflop*mflop*mflop*2.0/3.0)/tim.realtime())/1000000.0;
211         std::cout << "Mflops = " << mflop << std::endl;
212       }
213   }
```

A.5 Bibliothèque Jade

A.5.1 Factorisation de Cholesky de matrices creuses avec un découpage par colonnes

Listing A.32 – Noyau de l'implémentation de l'algorithme de factorisation de Cholesky de matrices creuses par bloc 1D avec Jade.

```
1   HandlePanel(panel, l, mat)
2   int panel;
3   rd_FMatrix l;
4   FMatrix mat;
5   {
6     int i, j, lastcol;
7     int row, dest_panel, dest_last;
8     int theFirst, theLast, length;
9     int is_column;
10    int my_gran;
11
12  #ifdef DEBUG
13    printf("HandlePanel %d\n", panel);
14  #endif
15
16  /* complete task begins here, parameter panel */
17    my_gran = gran;
18
19    is_column = (l->panel[panel] == 1);
20
21    withonly {
22      rd_wr(l->col_to_panel[panel]);
23      rd(mat);
24    } AT_PROC(l->panel_to_index[panel], my_gran) do (panel, mat) {
25      rd_FMatrix l;
26  /*
27      printf("%d : CompletePanel %d\n", get_this_proc(), panel);
28  */
29  #if !defined(NO_COMM_COMP)
30      l = mat;
31      CompletePanel(panel, l);
32  #endif
33    }
34
35    lastcol = panel + l->panel[panel];
36    length = l->lastnz[panel]-l->firstnz[panel];
37
38    theFirst = l->panel[panel];
39  #if defined(PARALLEL_SPAWN)
40    withonly {
41      rd(l->col_to_panel[panel]);
42      rd(mat);
43      while (theFirst < length) {
44        row = l->row[l->startrow[panel]+theFirst];
45        dest_panel = row;
46        if (l->panel[dest_panel] < 0)
47          dest_panel += l->panel[dest_panel];
48        dest_last = dest_panel + l->panel[dest_panel];
49        theLast = theFirst+1;
50        while (theLast < length && l->row[l->startrow[panel]+theLast] < dest_last)
51          theLast++;
52
53        df_rd_wr(l->col_to_panel[dest_panel]);
54        theFirst = theLast;
55      }
56    } do (panel, mat, my_gran) {
57      int lastcol;
58      int row, dest_panel, dest_last;
59      int theFirst, theLast, length;
60      int is_column;
61      rd_FMatrix l;
62
63      l = mat;
64  #endif
65      lastcol = panel + l->panel[panel];
66      length = l->lastnz[panel]-l->firstnz[panel];
67
68      theFirst = l->panel[panel];
69      while (theFirst < length) {
70        row = l->row[l->startrow[panel]+theFirst];
71        dest_panel = row;
72        if (l->panel[dest_panel] < 0)
73          dest_panel += l->panel[dest_panel];
74        dest_last = dest_panel + l->panel[dest_panel];
75        theLast = theFirst+1;
76        while (theLast < length && l->row[l->startrow[panel]+theLast] < dest_last)
77          theLast++;
78
79  /*
80        printf("%d : create %d %d\n", get_this_proc(), panel, dest_panel);
81  */
82        withonly {
83  #if defined(RD_WR_UPDATE)
84          rd_wr(l->col_to_panel[dest_panel]);
85  #endif
86  #if defined(CM_UPDATE)
87          cm(l->col_to_panel[dest_panel]);
```

```
88  #endif
89          rd(l->col_to_panel[panel]);
90          rd(mat);
91        } AT_PROC(l->panel_to_index[dest_panel], my_gran) do (panel, dest_panel, theFirst, theLast, length, mat) {
92          rd_FMatrix l;
93          int *loc;
94          double *storage;
95          int size, i, is_column;
96  /*
97      printf("%d : Modify %d %d\n", get_this_proc(), panel, dest_panel);
98  */
99  #ifdef DEBUG
100       printf("Modify %d,%d by %d,%d (%d,%d)\n",
101           dest_panel, dest_last, panel, lastcol, theFirst, theLast);
102 #endif
103 #if !defined(NO_COMM_COMP)
104         l = mat;
105         is_column = (l->panel[panel] == 1);
106         if (is_column) {
107           ModifyPanelByColumn(panel, dest_panel, theFirst, theLast, l);
108         } else {
109           if (theLast-theFirst == l->panel[dest_panel] &&
110               length-theFirst == l->lastnz[dest_panel]-l->firstnz[dest_panel]) {
111             /* same structure; add update directly into destination */
112             ModifyPanelByPanel(panel, dest_panel, theFirst, theLast, length, l);
113 /*
114                                 &L->nz[L->firstnz[dest_panel]]);
115 */
116           } else {
117             size = l->firstnz[dest_panel + l->panel[dest_panel]] -
118                    l->firstnz[dest_panel];
119             storage = (double *)
120               new_mem((size * sizeof(double)) + (l->n * sizeof(int)));
121             loc = (int *)(storage + size);
122             ZeroUpdate(theFirst, theLast, length-theFirst, storage);
123             ModifyPrivatePanelByPanel(panel, theFirst, theLast, length, storage, l);
124             FindRelativeIndicesRight(panel, theFirst, dest_panel, loc, l);
125             ScatterPanelUpdate(panel, theFirst, theLast, length-theFirst,
126                                dest_panel, storage, loc, l);
127             old_mem(storage);
128           }
129         }
130 #endif
131       }
132       theFirst = theLast;
133     }
134 #if defined(PARALLEL_SPAWN)
135   }
136 #endif
137 }
```

Modifications apportées au document

Suite aux remarques et conseils reçus, nous avons amélioré et raccourci le document. Les parties suivantes sont profondément remaniées :

- le résumé,
- l'introduction générale qui est raccourcie et structurée selon le plan préconisé,
- l'introduction du chapitre 1 qui est augmentée avec une section comportant une simplification des notions algorithmiques de l'ancienne introduction,
- un bilan pour chaque mode d'expression étudié au chapitre 1,
- les citations des travaux concernant Pandore et la parallélisation automatique,
- la conclusion du chapitre 1 qui est raccourcie,
- l'introduction du chapitre 2 qui est augmentée avec le discours de l'ancienne conclusion du chapitre 1.

Nous avons déplacé le discours sur LAPI de l'introduction du chapitre 3 vers celle du chapitre 4.

Nous avons intégré au chapitre 4 de nouveaux résultats qui ont fait l'objet d'une publication récente [22] pour la factorisation de matrices creuses sur une grappe de nœuds p690+.

Nous avons divisé l'ancien chapitre "Conclusion et perspectives" en un chapitre 5 "Perspectives" et un chapitre "Conclusion".

Expression d algorithmes scientifiques et gestion performante de leur parallélism avec PRFX pour les grappes de SMP

Résumé : Nous proposons un mode d'expression destiné à l'implémentation performante des algorithmes parallèles scientifiques, notamment irréguliers, sur des machines à mémoires distribuées hétérogènes (e.g. grappes de SMP). Ce mode d'expression parallèle est basé sur le langage C et l'API de la bibliothèque PRFX. Le découpage de l'algorithme en tâches est obligatoire mais les synchronisations et communications entre ces tâches sont implicites. L'identification des tâches et de leurs caractéristiques (identificateur de la fonction cible, paramètres, coût) doit être effectuée via l'interface de la bibliothèque PRFX. Cette interface doit également être utilisée pour allouer et libérer les données ainsi que pour déclarer les accès des tâches aux données. La structuration des données (référencement par pointeurs) est aisée car PRFX fournit un espace d'adressage global. D'autres fonctionnalités optionnelles de la bibliothèque PRFX permettent d'exprimer des optimisations génériques et lisibles. En effet, il est possible d'appliquer dans le cas des programmes prévisibles (i.e. structurés par un jeu de données d'entrée), des heuristiques d'ordonnancement globales paramétrées par des modèles de coût et guidées par le programmeur (e.g. spécification d'un placement initial).

Le support utilise une iso-mémoire assurant la validité des pointeurs en mémoire distribuée. Il adopte un fonctionnement de type inspection/exécution. L'inspecteur modélise l'exécution de l'algorithme parallèle par un DAG de tâches. Ce type de fonctionnement et ce modèle sont adaptés à l'exploitation performante des programmes irréguliers prévisibles. Les tâches et leurs accès aux données sont capturés dans ce DAG lors de l'inspection statique consistant en une pré-exécution partielle du programme. Les communications et synchronisations sont déduites de l'analyse des accès aux données faite par l'inspecteur. Ensuite, un ordonnanceur séquentiel applique l'heuristique d'ordonnancement intégrant les optimisations explicitées par le programmeur. Enfin, un exécuteur parallèle exécute le DAG de tâches en respectant l'ordonnancement des tâches et des communications. Il exploite la performance des communications unilatérales pour les communications inter-nœuds et tire profit de la mémoire partagée d'un nœud avec des *threads*.

Dans ce cadre, nous avons validé notre approche par l'implémentation d'algorithmes scientifiques (résolution de l'équation de Laplace, factorisation LU de matrices pleines et factorisation de matrices creuses par la méthode de Cholesky) et par des expérimentations en vraie grandeur sur des grappes de SMP IBM® NH2 et p690+.

Discipline : Informatique

Mots-Clefs : iso-mémoire, synchronisations implicites, algorithmes irréguliers prévisibles, DAG de tâches, grappes de SMP, inspection / exécution, ordonnancement, *threads*, communications unilatérales.

Implementing scientific algorithms and managing their parallelism efficiently with PRFX onto SMP clusters

Abstract : We present a programing model aimed to efficiently implement parallel scientific algorithms, in particular irregular ones, onto distributed memory machines like clusters of SMP nodes. This programming model is based on the C language and calls to our PRFX library. The programmer must split its algorithm into tasks nervertheless synchronizations and communications between tasks are implicit. These tasks operate on data that are dynamically allocated in an iso-memory and may access sub-data through the declaration of their geometry with stencils. Data structures (with pointers) may be easily implemented thanks to the global address space provided. We also provide functionalities to take into account the static properties of irregular algorithms (with structuring data) like scheduling heuristics using cost functions and an initial task mapping.

Our parallel runtime uses an iso-memory keeping pointer validity through migration in the distributed memory. It adopts an inspection/execution scheme. Our inspector modelizes the parallel execution by a task DAG which is well suited to predicitble irregular algorithms. Tasks and their accesses are statically analyzed by the inspector. This allows to build data dependencies thanks to a partial pre-execution of the code, and to produce a task DAG with all necessary information for the static scheduler and the parallel executor. Then, a static sequential scheduler applies a heuristic taking into account the optimizations specified by the programmer. Finally, the parallel executor executes the tasks and the communications of the DAG with respect to this schedule. Our parallel executor uses POSIX threads with one-sided communications and works on shared and distributed memory machines.

We validate its performances by experimental results for a Laplace equation solver, a LU factorization of dense matrices and a sparse Cholesky factorization algorithm on SMP IBM® cluster with NH2 and p690+ nodes.

Discipline : Computer-Science

Keywords : iso-memory, implicit synchronisations, predictible irregular algorithms, task DAG, SMP cluster, inspector/executor, scheduling, threads, one-sided communications

LaBRI,
Université Bordeaux 1,
351 cours de la Libération,
33405 Talence Cedex (FRANCE).