Contribution à la parallélisation de méthodes numériques à matrices creuses skyline. Application à un module de calcul de modes et fréquences propres de Systus

Pierre Bassomo

▶ **To cite this version:**

Pierre Bassomo. Contribution à la parallélisation de méthodes numériques à matrices creuses skyline. Application à un module de calcul de modes et fréquences propres de Systus. Web. Ecole Nationale Supérieure des Mines de Saint-Etienne; Université Jean Monnet - Saint-Etienne, 1999. Français. NNT : 1999STET4010 . tel-00822654

HAL Id: tel-00822654

https://theses.hal.science/tel-00822654

Submitted on 15 May 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ÉCOLE NATIONALE SUPÉRIEURE
DES MINES DE SAINT-ETIENNE

UNIVERSITÉ JEAN MONNET
DE SAINT-ETIENNE

N° d'ordre : 209 ID

THÈSE

Présentée par Pierre BASSOMO

pour obtenir le titre de

Docteur

DE L'UNIVERSITÉ JEAN MONNET DE SAINT-ETIENNE ET DE
L'ÉCOLE NATIONALE SUPÉRIEURE DES MINES DE SAINT-ETIENNE

Spécialité Informatique, Calcul Parallèle

Contribution à la parallélisation de méthodes numériques à matrices creuses skylines. Application à un module de calcul de modes et fréquences propres de SYSTUS

Soutenue à Saint-Etienne, le 12 Juillet 1999

Composition du jury :

Madame	C. Sayettat	Présidente
Messieurs	S. Petiton	Rapporteurs
	S. Radjopadhye	
	M. Jourlin	
	M. Ahués	Examinateurs
	P. Clauss	
	B. Vinsonneau	Invité
	I. Sakho	Directeur de thèse

ÉCOLE NATIONALE SUPÉRIEURE
DES MINES DE SAINT-ETIENNE

UNIVERSITÉ JEAN MONNET
DE SAINT-ETIENNE

N° d'ordre : 209 ID

THÈSE

Présentée par Pierre BASSOMO

pour obtenir le titre de

Docteur

DE L'UNIVERSITÉ JEAN MONNET DE SAINT-ETIENNE ET DE
L'ÉCOLE NATIONALE SUPÉRIEURE DES MINES DE SAINT-ETIENNE

Spécialité Informatique, Calcul Parallèle

Contribution à la parallélisation de méthodes numériques à matrices creuses skylines. Application à un module de calcul de modes et fréquences propres de SYSTUS

Soutenue à Saint-Etienne, le 12 Juillet 1999

Composition du jury :

Madame	C. Sayettat	Présidente
Messieurs	S. Petiton	Rapporteurs
	S. Radjopadhye	
	M. Jourlin	
	M. Ahués	Examinateurs
	P. Clauss	
	B. Vinsonneau	Invité
	I. Sakho	Directeur de thèse

Je tiens à remercier ici tous les membres du jury :

Claudette Sayettat, professeur à l'*Ecole Nationale Supérieure des Mines de Saint-Etienne*, pour l'honneur qu'elle me fait en acceptant de présider ce jury;

Serge Petiton, professeur à l'*Université des Sciences et Technologies de Lille*, pour avoir bien voulu rapporter ce travail. Ses nombreux travaux dans la parallélisation des codes de calcul scientifiques m'ont éclairé dans le portage de *SYSTUS* sur des architectures parallèles.

Sanjay Rajopadhye, chargé de recherche au *C.N.R.S.* et à l'*I.R.I.S.A.* de *Rennes*, pour avoir accepté de rapporter cette thèse.

Michel Jourlin, professeur au *C.P.E de Lyon*, pour avoir accepté de rapporter cette thèse.

Philippe Clauss, maître de conférences à l'*Université Louis Pasteur de Strasbourg*, pour avoir bien voulu être membre de ce jury;

Mario Ahués, professeur à l'*Université Jean Monnet de Saint-Etienne*, pour avoir accepté d'être membre de ce jury, et pour la grande rigueur scientifique qu'il a montrée chaque fois que j'ai eu l'occasion de travailler avec lui. Je lui suis reconnaissant pour ses nombreuses contributions dans les traitements numériques de problèmes spectraux.

Bernard Vinsonneau, ingénieur expert en dynamique des structures chez *SYSTUS International*, dont les nombreuses remarques m'ont permis de de mieux comprendre le fonctionnement du module de calcul de modes et fréquences propres et pour les exemples tests qu'il a mis à ma disposition.

Ibrahima Sakho, professeur à l'*Université de Metz*, pour avoir encadré et diriger mon travail avec sérieux malgré la distance. Ses nombreuses remarques sur le fond m'ont permis de formaliser une méthode de parallélisation à partir des expériences réalisées au tour de *SYSTUS*.

Je lui suis très reconnaissant pour tous ses conseils amicaux et fraternels qui m'ont permis de rester très autonome au cours de cette thèse.

Cette thèse s'est déroulée au sein du département informatique de l'Ecole des Mines de Saint-Etienne. Que tous les membres dudit département qui m'ont accompagé durant cette période trouvent ici mes plus sincères remerciements :

Annie Corbel, maître-assistant à *Ecole Nationale Supérieure des Mines de Saint-Etienne*, qui a co-encadré ma thèse et dont les nombreuses remarques très constructives m'ont aidé dans la rédaction de ce travail. Je lui suis très reconnaissant pour ses conseils fort utiles dans connaissance de *PVM* et de la parallélisation de codes de calculs scientifiques.

Bernard Peroche, professeur à *Ecole Nationale Supérieure des Mines de Saint-Etienne*, qui a tout fait, administrativement et diplomatiquement, pour que la thèse et sa soutenance se passent dans de meilleurs conditions.

Jean-Jacque Girardot, le responsable du département *R.I.M* dont je suis membre.

toutes et tous les collègues du département *R.I.M*: *Fernando, Jérôme, Bich-Lien, ...* pour leurs délires salvateurs;

tous les autres du centre *SIMADE*: *Jean-Michel, Marc, Michel, Marie-Line, Nicolas, ...*

Je remercie également *Alain Bourgeat* du laboratoire d'*Analyse Numérique* de l'*Université Jean Monnet de Saint-Etienne* qui m'a donné l'idée de faire cette thèse, au terme de mon stage de *DEA*.

J'exprime ma gratitude à *Marilyne* et à toute sa famille pour m'avoir soutenu durant tous ces moments.

Merci à ma famille pour toutes ses prières, son amour et sa compréhension qui m'ont accompagné durant tous les moments difficiles que j'ai enduré

Table des matières

Chapitre 1

Introduction générale

1.1 Problématique des méthodes numériques

Les méthodes numériques sont de nos jours un outil indispensable pour la modélisation et l'analyse d'un nombre de plus en plus croissant de problèmes relevant des sciences de l'ingénieur. (On trouvera par exemple dans [LP96, LT86a], une introduction aux méthodes numériques ainsi qu'aux classes de problèmes qu'elles ciblent). Toutefois, même si des méthodes numériques de plus en plus performantes sont conçues et développées avec des moyens de calcul standard, les finesses d'analyses requises par certaines classes de problèmes rendent leur utilisation prohibitive. C'est par exemple le cas du calcul des modes et fréquences propres de structures mécaniques (voir [Cha88]).

Depuis le début du siècle, les analyses vibratoires font partie de la conception des structures pour augmenter leur durée de vie et leur sécurité de fonctionnement. La détermination de modes et fréquences propres est la partie la plus classique de ces analyses. Les fréquences propres réelles servent à éviter les résonances, les fréquences propres complexes permettent d'aborder l'étude de stabilité, pour des rotors par exemple, ou bien dans des domaines autres que la construction mécanique. Au début, les modèles structuraux étaient analytiques et provenaient d'une description des structures à l'aide d'un petit nombre de degrés de liberté. Mais ces procédés sont difficilement applicables à toutes les structures mécaniques. Avec l'important progrès de la méthode des éléments finis et le développement des moyens informatiques, l'analyse des structures a connu un essor considérable. Aujourd'hui, il existe des logiciels très performants pour l'analyse des structures (par exemple *SYSTUS*) avec un nombre de degré de liberté relativement grand, améliorant ainsi, la qualité de l'analyse. Il est toutefois, des facteurs limitant à ces logiciels: les moyens informatiques utilisés.

En effet, les temps de certaines simulations augmentent. Il se pose alors le problème de la recherche de moyens de calcul permettant d'analyser le comportement des structures dans des délais raisonnables.

Les solutions offertes par les constructeurs informatiques pour effectuer les simulations à des coûts raisonnables se situent dans deux directions [AG89]: l'accroissement de la vitesse d'horloge des processeurs classiques et le parallélisme. Bien que des processeurs de plus en plus performants soient disponibles, les besoins en capacité de mémorisation et en puissance de calcul des méthodes numériques augmentent encore plus vite. Le parallélisme constitue alors une alternative intéressante.

1.2 Apport du calcul parallèle aux méthodes numériques

Une définition unanimement admise du parallélisme ou encore du calcul parallèle est le recours à plusieurs unités de traitement (processeurs) qui communiquent et se coordonnent pour résoudre un même problème. Ainsi, le parallélisme, apporte t-il aux méthodes numériques, une puissance de calcul et une capacité de mémorisation qui croissent avec le nombre de processeurs. Ceci permet d'envisager des niveaux de simulation non encore atteints, dans des délais raisonnables. Outre la puissance de calcul et la capacité de mémorisation qu'on peut qualifier d'apport quantitatif, l'apport du parallélisme aux méthodes numériques est également qualitatif. En effet, la multiplication des unités de traitement pour résoudre un même problème requiert que celui-ci soit décomposé en sous problèmes induisant ainsi, la maîtrise de la complexité de celui-ci.

Pour atteindre de tels objectifs via le parallélisme, de nombreuses difficultés subsistent. En effet, le concept de calculateur parallèle introduit des options d'un genre nouveau dans le mode de fonctionnement des ordinateurs. Celles-ci doivent être maîtrisées par les programmeurs afin d'exploiter efficacement le parallélisme potentiel d'une application. D'une manière générale, on peut distinguer les options liées:

1- aux mécanismes de communication [Rum94, Fos95],

2- à la granularité du parallélisme [CT93],

3- à l'organisation des traitements en parallèle [Fly72].

Toutes ces options donnent lieu à une diversité de modèles de programmation et de machines parallèles que *Flynn* [Fly72] classe en: *SISD*, *SIMD*, *MISD* et *MIMD*. Face à cette multitude de machines parallèles, concevoir des applications parallèles impose, soit des outils de parallélisation automatique, soit un effort du programmeur suivant des méthodologies de programmation.

Quelle que soit l'approche, l'objectif reste une **parallélisation optimale** qui se mesure par le **facteur d'accélération** (le rapport du temps d'exécution séquentielle de l'application et du temps d'exécution parallèle de l'application). Une parallélisation optimale étant celle dont le facteur d'accélération est égal au nombre de processeurs utilisés.

1.3 Contribution de la thèse

Dans cette thèse, nous proposons une méthodologie de parallélisation des méthodes numériques. En général les méthodes numériques sont une chaine d'algorithmes s'appelant les uns après les autres tout au long de leur exécution. A moins d'aborder leur parallélisation à partir du problème physique qu'elles traitent, par exemple par des techniques de *décomposition de domaines*, l'approche de parallélisation la plus réaliste est celle de type *client/serveur*. Elle consiste à préserver la structure de la méthode numérique et à paralléliser quelques uns des algorithmes qui la composent. La méthode numérique devient ainsi client et les algorithmes qui ont été parallélisés les serveurs. C'est l'approche adoptée dans cette thèse dont la contribution se situe au niveau de la méthodologie de parallélisation des serveurs.

Etant donné une exécution systolique optimale selon une fonction de temps affine ou linéaire, de la solution d'un problème exprimée sous forme d'équations récurrentes, nous proposons une méthodologie de mise en oeuvre d'une telle solution sur une machine dont le

nombre de processeurs est inférieur à celui nécessaire à l'exécution systolique, garantissant son optimalité au sens du lemme de *Brent* (voir chapitre 3) ainsi que d'un modèle d'exécution prenant en compte les coûts de communication. Pour cela, nous procédons en quatre étapes:

Etape 1 : réécriture d'un schéma d'exécution séquentiel d'une application donnée sous la forme d'un schéma d'exécution systolique,

Etape 2 : création d'un graphe quotient par agrégation en super-noeuds, des noeuds d'un graphe d'exécution systolique,

Etape 3 : ordonnancement en temps minimum du graphe obtenu à l'étape 2,

Etape 4 : allocation en des sommets du graphe obtenu à l'étape 2 en préservant "au mieux", l'ordonnancement en temps minimum de l'étape 3.

La génération des structures de données et du code découle des quatre étapes précédentes. Nous proposons une extention de cette méthodologie au cas où les méthodes numériques utilisent des matrices creuses. Les méthodes numériques ainsi parallélisées sont analysées du point de vue de leur complexité. En particulier, une analyse mathématique du temps d'exécution des algorithmes parallèles en fonction de la taille des super-noeuds, du coût des communications et du nombre de processeurs est réalisée.

Cette méthodologie est ensuite appliquée à la parallélisation du module *DLANCB* de calcul de modes et fréquences propres de *SYSTUS*. Les performances du module de calcul parallèle sont évaluées sur des machines parallèles et comparées aux performances théoriques, à travers deux exemples d'applications de l'analyse dynamique des structures.

1.4 Organisation de la thèse

La suite de la thèse est organisée en trois grands chapitres. Dans le chapitre 2, nous faisons une synthèse des grandes classes de techniques destinées à la parallélisation optimale des méthodes numériques. L'accent est mis sur les outils et méthodologies de parallélisation, afin de montrer les avantages et les limites des stratégies qui existent dans la littérature et de justifier la méthodologie proposée.

Le chapitre 3 est dédié à la partie théorique de la thèse, à savoir une présentation de la méthodologie de parallélisation ainsi qu'une approche d'analyse de complexité. Nous commençons par des rappels sur l'algorithmique systolique qui sert de fil conducteur à la méthodologie proposée. Cette méthodologie repose sur des modèles mathématiques que nous exposons. Afin d'en faciliter la compréhension, ses aspects fondamentaux sont discutés dans le cadre de modèles d'applications ciblés par le modèle systolique. Dans le cas d'applications irrégulières, nous exposons les réadaptations nécessaires.

Le chapitre 4 concerne l'application de la méthodologie à la parallélisation d'un module de calcul des modes et fréquences propres. Une présentation de ce code nous permet d'étudier les divers algorithmes exécutés pour le calcul des modes et fréquences propres. La parallélisation s'articule autour des algorithmes dont l'exécution répétitive est prohibitive. Les performances parallèles sur des exemples d'applications tels que l'etude d'une plaque mince ou d'un cylindre encastré sont fournies.

Le chapitre 5 fait le bilan de ce travail, expose les problèmes qui restent à traiter et indique quelques perspectives.

Chapitre 2

Parallélisation des méthodes numériques

Le but de ce chapitre est de modéliser le problème de la parallélisation optimale des méthodes numériques et de trouver une classification de ses solutions, en s'attardant sur les méthodologies de parallélisation, plus particulièrement aux méthodologies de parallélisation des méthodes numériques à matrices creuses.

Pour commencer cette étude bibliographique, nous posons le problème de la parallélisation en mettant l'accent sur les techniques d'optimisation du modèle mathématique d'une application parallèle.

La section 2.2 traite des approches de parallélisation des méthodes numériques. La section 2.2 aborde le cas particulier des approches de parallélisation des méthodes numériques à matrices de très grande taille.

2.1 Position du problème

Le parallélisme est une forme de traitement informatique qui permet, au cours d'exécution, l'exploitation des événements concurrents. Ces événements peuvent être de plusieurs types: procédures, instructions, etc.

Ainsi, le traitement parallèle d'une application consiste en sa transformation en une séquence d'événements concurrents sans en altérer la sémantique. Dans ce contexte, la parallélisation optimale a pour but de faire que la séquence des événements concurrents d'une application soit la plus courte possible. En général, la parallélisation optimale d'une application consiste en un ensemble de tâches qui s'obtiennent par application des quatre phases suivantes:

1– **partitionnement** d'une application en un ensemble de tâches et modélisation sous forme d'un graphe de tâches,

2– **ordonnancement** du graphe de tâches,

3– **allocation** des tâches aux processeurs d'une machine parallèle en respectant les contraintes de l'ordonnancement des tâches,

4– **mise en oeuvre** du programme parallèle.

Pour mener à bien chacune de ces phases, un modèle mathématique de l'application parallèle et de la machine d'exécution s'impose. Ainsi, avant d'expliciter les problèmes liés à la conduite de chacune de ces phases, nous discutons d'abord de la modélisation mathématique d'une application parallèle. Le modèle mathématique de la machine parallèle sera discuté plus loin (voir section 3.3.2).

2.1.1 Modèles mathématiques d'une application parallèle

Ces dernières décennies, les recherches ont permis la construction d'algorithmes parallèles ayant divers niveaux d'efficacité. Le modèle unanimement accepté est celui du graphe de tâche [CT93]. C'est à dire un ensemble de tâches $D = \{u_1, u_2, \cdots, u_n\}$ qui sont reliées par un ensemble E de relations de dépendance.

Ce modèle pose des difficultés à au moins deux niveaux. A un premier niveau, il s'agit d'evaluer théoriquement les performances des applications sous-jacentes sur des modèles de machines plus réalistes que les modèles du type *PRAM*. En effet, les modèles *PRAM* ignorent les surcoûts liés aux environnements logiciels et matériels pour la mise en oeuvre.

A un second niveau, les applications scientifiques qui motivent le recours au parallélisme se traitent via des méthodes numériques qui enchainent différentes classes d'algorithmes. Si pour chacune de ces classes d'algorithmes, le modèle de graphe de tâches peut donner lieu à des algorithmes parallèles optimaux, il n'est pas sûr qu'en réinjectant ces algorithmes parallèles optimaux dans la méthode numérique, il en résulte un programme parallèle optimal. En effet, aux coût de l'exécution des algorithmes ainsi parallélisés doivent être rajoutés les coûts de simulation des phases: de partitionnement, d'ordonnancement et d'allocation. La simulation de ces phases est très souvent prohibitive lorsque les calculs dans les algorithmes, manipulent des structures irrégulières.

2.1.2 Partitionnement d'une application en un ensemble de tâches

Considérons une application dont on connait un schéma d'exécution séquentiel correct. Le partitionnement consiste à décomposer cette application en un ensemble de tâches dont l'analyse des dépendances, permet de construire un graphe de tâches, soit $G = (D, E)$, qui traduit un parallélisme interne à l'application. De plus, l'exécution de G doit fournir des résultats identiques à ceux du schéma d'exécution séquentiel correct. Classiquement, trois stratégies existent pour décomposer une application en un ensemble de tâche. Il s'agit de la décomposition par les:

1– données,

2– calculs,

3– approches de type "*diviser pour paralléliser*".

Le partitionnement optimal consiste à choisir la meilleure décomposition de tâches possible. Le choix de la meilleure décomposition possible est un problème difficile lié au temps d'exécution de G et à la machine parallèle d'exécution [CT93]. Sans être exhaustif, les principaux facteurs de choix sont:

- le nombre de processeurs,

- le rapport unité de temps de communication/unité de temps de calcul,

- capacité de mémorisation et organisation hiérarchique de la mémoire,
- environnement de programmation parallèle.

2.1.3 Ordonnancement des tâches

Dans le cadre d'une parallélisation optimale, l'ordonnancement consiste en la recherche d'un temps d'exécution minimum du graphe graphe de tâches. Pour ce faire, on procède généralement en deux étapes: recherche d'une borne inférieure du temps d'exécution, en suite, construction d'un algorithme dont le temps d'exécution est égal à cette borne. On résout ainsi un problème d'ordonnancement en temps minimum d'un ensemble de tâches [CT93]. La complexité parallèle en terme de temps d'exécution est définie par cette borne inférieure. A cette complexité, est sous-jacente celle spaciale c'est à dire le nombre minimum de processeurs requis pour atteindre cette borne inférieure du temps d'exécution.

Plus formellement, pour un graphe de tâche $G = (D, E)$ le problème de l'ordonnancement consiste à trouver une fonction de temps:

$$\begin{array}{rcl} T & : & D \rightarrow I\!N \\ & & u \longmapsto T(u) \\ & & \text{vérifiant } T(u) < T(v) \text{ pour tout arc } (u,v) \in E \end{array} \tag{2.1}$$

L'ordonnancement optimal étant la fonction qui garantisse la quantité:

$$\min_{T} \max_{u \in D} T(u) \tag{2.2}$$

Le problème de l'ordonnancement optimal des tâches en général n'est entièrement résolu que dans des cas particuliers simples. Il est répertorié comme étant *NP-complet* [Ull75].

2.1.4 Allocation des tâches

Etant donné un graphe de tâches $G = (D, E)$ et un ordonnancement compatible T, l'allocation des tâches consiste à associer à chaque tâche un processeur d'exécution de sorte que le temps d'exécution de G soit celui indiqué par T. Plus formellement, l'allocation est une application définie par:

$$\begin{array}{rcl} A & : & D \rightarrow \mathcal{P} \\ & & u \longmapsto A(u) \end{array} \tag{2.3}$$

$A(u)$ est le processeur qui exécute u et $\mathcal{P}$ est l'ensemble des processeurs.

Une allocation optimale est une allocation garantissant un temps d'exécution induit par un ordonnancement optimal. Le problème de l'allocation optimale n'est généralement pas facile. Il est plus difficile lorsque le nombre de processeurs disponibles est inférieur au nombre nécessaire induit par l'ordonnancement.

Par ailleurs, pour une allocation donnée, il n'est pas évident qu'en préservant le temps d'exécution fixé par un ordonnancement optimal (voir section 2.1.3), le programme parallèle soit optimal. En effet, après une allocation des tâches aux processeurs, l'exécution va dépendre des paramètres tels que:

- ⋆ la connaissance de la structure de l'application parallèle qui dépend de ses composantes (code et données) et des relations entre les composantes (communications et précédence),

⋆ la puissance de calcul et la capacité de mémorisation de chaque processeur,

⋆ la performance du réseau d'interconnexion de la machine cible.

Dans sa généralité, le problème d'une allocation est repertorié *NP-complet* [GJ79]. On essaie donc rarement de trouver des solutions optimales et on se contente des solutions approchées i.e des allocations produisant des performances "acceptables" et conduisant à des programmes parallèles "réalistes".

2.1.5 Mise en oeuvre optimale

Etant donné un modèle mathématique d'une application parallèle sous forme de graphe de tâches $G = (D, E)$ et une allocation des tâches A, le problème de la mise en oeuvre parallèle consiste à choisir des structures de données et à générer à partir de ces structures de données, des algorithmes qui simulent l'exécution de G. Une mise en oeuvre optimale est un mise en oeuvre parallèle garantissant un temps d'exécution minimum.

Le problème de la mise en oeuvre optimale n'est pas facile. En effet, pour une allocation donnée, le choix des structures de données peut s'accompagner d'algorithmes moins performants ou prohibitif en espace mémoire. On essaie continuellement de gérer au mieux la puissance de calcul et la capacité de mémorisation disponibles. Ceci contribue à intégrer des paramètres supplémentaires qui compliquent la résolution des problèmes exposés dans les section 2.1.2, 2.1.3 et 2.1.4.

2.2 Taxinomie des approches de parallélisation pour les méthodes numériques

Etant donné un problème dont la résolution fait appel à des méthodes numériques, la littérature sur la parallélisation des méthodes numérique est fort abondante. Quelle que soit l'approche préconisée, elle consiste en la résolution des quatre problèmes évoqués précédement dans la section 2.1. La multiplicité des paramètres gouvernant chacun de ces problèmes fait qu'une taxinomie sur les façons de les résoudre est assez difficile à établir. Les approches toutefois peuvent être différentiées selon qu'elles intègrent les quatre problèmes depuis la formulation du problème physique à résoudre ou seulement dans la résolution de celui-ci.

Dans le premier cas, on trouve par exemple le cas général des diverses approches de décomposition de domaine [KST95, CM94], et en particulier, pour le cas du calcul des modes et fréquences propres, les méthodes numériques du style *explicitly restarded* ou *implicitly restarded Lanczos* [MS95, Sor95]. Dans le second cas, on trouve les approches de parallélisation classiques telles que par exemple celles décrites dans [KGGK94, Lav90, CT93].

Les méthodes dites de décomposition de domaine préconisent une formulation parallèle du problème physique et se placent à la croisée de plusieurs disciplines dont notamment, le domaine d'application, l'analyse numérique et l'informatique. En effet, outre la définition des différents sous problèmes physiques et leur résolution, cette classe d'approches pose aussi la question de l'interfaçage des solutions des sous problèmes qui dépassent le cadre de la présente thèse. Notre taxinomie porte donc plutôt sur les approches de parallélisation des solutions séquentielles des problèmes physiques dictées essentiellement par un souci de conservation des acquis.

Cette approche dite de parallélisation des algorithmes standards est de nos jours, de mieux en mieux maitrisée, du moins dans le monde des informaticiens du parallélisme. Pour les

gros logiciels combinant à la fois des appels à de multiples modules et sous programmes c'est loin d'être le cas. Celle-ci motive le développement d'outils et de méthodes de parallélisation pour de tels logiciels. Les outils de parallélisation permettent de produire automatiquement des logiciels parallèles tout en limitant les interventions humaines. Quant aux méthodes de parallélisation, elles permettent de produire des codes parallèles au prix d'une intervention significative du programmeur.

Dans ce qui suit, nous étudions les grandes classes d'approches de parallélisation des méthodes numériques. Le cas particulier des méthodes numériques enchainant différentes classes d'algorithmes nous permet d'aborder les limites de validité de ces approches. Il s'agit de méthodes numériques dont le programme est constitué d'une multitude de sous programmes. De nombreuses études montrent que dans de tels programmes, une grande fraction du temps d'exécution est passée dans le traitement d'un petit nombre d'instructions internes à des boucles possédant un potentiel de parallélisme très important. Enfin pour ces méthodes numériques, la structure répétitive et parfois régulière de leurs boucles imbriquées facilite la mise en oeuvre des techniques d'analyse de dépendance et la recherche de stratégie: de partitionnement, d'ordonnancement et d'allocation.

En raison de l'abondance de la littérature sur le sujet de la parallélisation (voir par exemple [Aa94, BT89, DT94, Fea96a, PE96, Lil94, HNP91]), nous ne présentons donc que les grandes classes de stratégie. Le principal critère de classification que nous retenu est le caractère automatique ou non, de la production des programmes parallèles.

2.2.1 Parallélisation automatique

La parallélisation automatique est actuellement l'objet d'un vaste effort de recherche. En témoignent les nombreuses équipes de recherche qui y travaillent et les outils proposés. On peut en guise d'illustration citer: *SUIF*, à l'université de Stanford en Californie, *PIPS*, à l'Ecole des Mines de Paris, *LooPo* à l'université de Passau, et *PAF* à l'Université Versailles, entre autres. Elle consiste à laisser à un outil spécialisé, par exemple un compilateur, le soin d'exploiter le parallélisme implicite à un programme source. Analysons maintenant l'intégration de la résolution des quatre problèmes évoqués précédement dans la section 2.1, au principe de fonctionnement d'un paralléliseur automatique (voir par exemple la figure (2.1)).

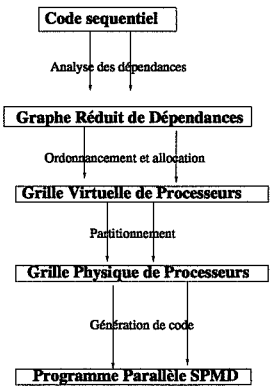

FIG. 2.1 – Principe de la parallélisation automatique.

2.2.1.1 Analyse du principe de la parallélisation automatique
Analyse de dépendance

L'analyse des dépendances a pour objectif la création d'un graphe décrivant les contraintes sur l'ordre d'exécution des instances des instructions. La détection des dépendances dans le code séquentiel, qui en est le principe, a donné lieu à de nombreuses études [Ban88, YAI95, BGS94]. Pour les méthodes numériques, l'analyse des dépendances porte sur les boucles imbriquées. La structure régulière et répétitive des classes de boucles imbriquées telles que les nids de boucles affines, facilite en effet la mise en oeuvre des techniques de détection et d'analyse automatiques de dépendance.

Mais en général, la détection automatique des dépendances est plutôt difficile car trop peu d'information sur les zones mémoires sont disponibles de façon statique, c'est à dire avant l'exécution. Pour être sûr de conserver la sémantique du programme, des algorithmes tels que le test de *Banerjee* ou l'*Omega test* [Pug92] ont été proposés. Ces algorithmes sont trop généraux pour fournir une analyse exacte des dépendances. Dans [Fea96a], on montre que sous certaines hypothèses, on peut donner de façon exacte toutes les dépendances d'un programme par des techniques de programmation linéaire paramétrée.

Notons cependant que, l'analyse des dépendances exactes ne cible pas toutes les classes de programmes. Par exemple, la parallélisation automatique telle qu'exposée dans [Fea96a], suppose des structures de données de type tableaux multidimensionnels et indexés par des fonctions affines. Ceci en effet exclu notamment le traitement des matrices creuses en général et skylines en particulier(voir section 2.3).

Ordonnancement et allocation

Dans le cadre de la parallélisation automatique, l'ordonnancement permet d'exhiber le parallélisme. Une synthèse des principaux algorithmes d'ordonnancement est fournie dans [Viv97]. Ils s'inspirent généralement de l'algorithme de *Lamport* [Lam74] ou de celui d'*Allen* et *Kennedy* [AK87], deux catégories d'algorithmes d'ordonnancement. Selon qu'on désigne la représentation exacte ou approchée des dépendances.

En parallélisation automatique, les fonctions d'allocation sont dans leur majorité supposées affines [DR94, AL93]. Cela suppose que les processeurs forment une grille multidimensionnelle. De telles hypothèses ne se justifient totalement que pour des machines dont la topologie est une grille avec un nombre suffisant de processeurs ou lorsque la parallélisation a pour langage cible *HPF*[KLS+94]. Dans les environnements d'exécution dont les caractéristiques matérielles et logicielles sont coûteux en termes de communication, la phase de l'allocation de la parallélisation automatique nécessite d'être prolongée par un partitionnement.

Le partitionnement

Il consiste à partitionner l'ensemble des processeurs dits virtuels de la phase d'allocation et à allouer chaque élément de la partition à un processeur physique dans le respect des contraintes comme le grain du parallélisme, les coût des communications, l'équilibrage de la charge de travail des processeurs.

Généralement, ces contraintes sont satisfaites au prix d'analyses relativement prohibitives

telles que les mécanismes d'*inspecteur/exécuteur* [DPM91]). Dans le cas où il existe une régularité dans les calculs ou que les processeurs virtuels forment une grille multidimensionnelle, des partitionnements par blocs [RS92, IT88] permettent par exemple de transformer les communications entre des processeurs virtuels adjacents en communications locales. Dans cet ordre d'idée, on trouve les techniques dites de "*pavage*". L'idée est de regrouper plusieurs points de l'espace, représentant chacun un processeur virtuel ou un calcul, au sein d'un même "*super-noeud*" encore appelé "*tuile*" (voir figure (2.2)). Chaque tuile est une unité de calcul "atomique" c'est à dire que ses points de synchronisation sont au début et à la fin des traitements de ses calculs. Les communications entre les tuiles sont regroupées en un message unique. Il existe une abondante littérature sur les techniques de pavage dont un état de l'art fournit dans [Bou96].

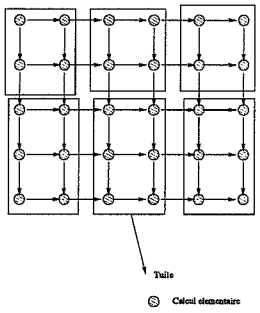

FIG. 2.2 – Exemple d'un pavage $2D$.

Génération de code

C'est la phase au cours de laquelle la parallélisation automatique produit du code parallèle. Elle consiste en la détection et l'élimination de certaines dépendances qui peuvent être cachées par l'implantation d'un code séquentiel (encore appelées "fausses dépendances") afin d'exhiber le maximum de parallélisme. Les techniques pour ce faire procèdent par renommage de scalaires ou de tableaux, expansion scalaires ou de tableaux, privatisation ou mise en assignation unique. Pour une description complète de ces techniques, nous renvoyons le lecteur aux articles de synthèse [BENP93, BGS94].

L'application directe de ces techniques à la transformation d'un code séquentiel induit un coût en espace mémoire relativement important. En intégrant ces techniques d'élimination dans les algorithmes d'ordonnancement, on peut exhiber du parallélisme avec un coût moindre en espace mémoire. Nous ne discutons pas ici, de la faisabilité d'une telle procédure et nous renvoyons le lecteur intéressé à la thèse de *Vivien* [Viv97].

Au nombre des problèmes qui compliquent la génération automatique des codes parallèles performants, on peut citer:

- l'ordonnancement et l'allocation des *tuiles* dans les techniques de *pavage* qui peuvent générer des codes trop gros [BDRV97],

- les problèmes de fusion de boucles parallèles qui réduisent le nombre de boucles parallèles produites par des algorithmes de parallélisation, mais qui sont prohibitifs en espace mémoire [BDSV97].

2.2.1.2 Limites du principe de la parallélisation automatique

L'étude des différentes étapes de la parallélisation automatique permet de constater que, son domaine de validité dépend de facteurs tels que:

- la performance des algorithmes de parallélisation,
- les architectures ciblées,
- la structure des codes sources à paralléliser.

Nous ne faisons pas ici une discussion détaillée sur la performance des algorithmes de parallélisation. Nous renvoyons le lecteur intéressé à l'article [BDSV97]. On montre ainsi, que les paralléliseurs automatiques classiques sont confrontés à deux types de difficultés:

- leur complexité en temps et en espace, en raison de la nature combinatoire des problèmes à résoudre lors de la phase de parallélisation. En effet, dans le cas des dépendances affines, la construction du graphe de dépendance, son ordonnancement et le placement de ses sommets sur les processeurs disponibles requièrent des outils mathématiques comme la programmation sous contraintes en nombre entiers dont on sait que les solutions exigent souvent des recherches exhaustives. On comprend alors qu'en général, les paralléliseurs automatiques classiques ne puissent être utilisés pour paralléliser totalement une méthode numérique composée de plusieurs classes d'algorithmes,
- l'analyse sémantique efficace des programmes, comme l'on montré [Tri84, Ber93], est performante si sa complexité en temps et en espace est grande.

Comme conséquence, les codes parallèles obtenus par parallélisation automatique sont rarement à la hauteur des attentes ainsi que le prouvent les expériences de comparaison entre des programmes parallélisés automatiquement et ceux parallélisés manuellement sont exposés dans [BENP93]. Elles montrent que les programmes parallélisés manuellement sont plus rapides. Il se pose donc un problème d'amélioration des outils de parallélisation automatique dont les travaux sont en cours. Certains travaux reposant sur les techniques inspirées de la synthèse des réseaux systoliques [Pug91]. En l'état actuel des connaissances dans la parallélisation automatique, seuls des programmes très simples sont concernés. La parallélisation des gros logiciels, est plutôt semi-automatique et utilise donc nombre de techniques qui ont cours dans le domaine de la parallélisation manuelle.

2.2.2 Parallélisation manuelle

Une autre approche de programmation des machines parallèle pour le calcul à haute performance consiste à développer des environnements de programmation qui mettent à la disposition du programmeur, des outils nécessaires à l'expression, la mise en oeuvre et l'évaluation du parallélisme. Les progrès enregistrés dans le domaine du calcul parallèle ont permis de développer plusieurs classes d'environnement de programmation justifiant ainsi, diverses méthodologies de parallélisation.

Il existe de nombreux prototypes d'environnements académiques pour la synthèse d'architectures spécialisées telles que: *Diastol*, *PEI,MMAlpha*, etc [Mau89]. Ils permettent la manipulation des *équations de récurrence affines*. Ces environnements de programmation sont très peu répandus dans le domaine de la parallélisation des méthodes numériques. Néanmoins, le modèle *polyédrique*, un des fondements de la synthèse dans ces environnements, nous semble très intéressant pour la parallélisation sur d'autres types d'environnements de programmation (voir chapitre 3) comme par exemple, les environnements de programmation les plus répandus actuellement et qui sont sont à base de:

- ⋆ langage de programmation *data-parallèle* (par exemple *HPF*) pour le parallélisme implicite,
- ⋆ bibliothèques à passage de message (par exemple *PVM* ou *MPI*) pour le parallélisme explicite.

L'intérêt de ces environnements est qu'ils sont disponibles sur une grande gamme de machines parallèles commercialisées et offrent un parallélisme virtuel gage de portabilité. Ces environnements favorisent une programmation à base de processus communicant. Le concept fondamental de ces environnements est celui du rapprochement sur un couple processeur-mémoire, d'un couple calcul-données. Un des enjeux actuels est de proposer des méthodologies de programmation prouvées correctes et efficaces pour ces environnements de programmation très répandus. Avant de discuter de ces problèmes, nous faisons un tour d'horizon de ces environnements de programmation.

2.2.2.1 Environnements de programmation parallèle

On distingue deux classes d'environnement de programmation parallèle. Dans la première, on écrit des algorithmes en explicitant toutes les phases de calcul et de communication: on parle de parallélisme explicite. Dans la seconde, on décrit le parallélisme dans les algorithmes à l'aide de directives: on parle de parallélisme implicite.

Les environnements à parallélisme explicite

Dans de tels environnements, l'utilisateur se charge explicitement de la génération des programmes qui vont être exécutés en parallèle. Dans ce, il doit prendre en compte à la fois les paramètres de la machine cible et ceux de l'application à paralléliser. Afin de simplifier l'effort d'implantation, les programmes sont souvent écrits en mode *SPMD (Single Programme Multiple Data)*. Chaque programme est une implantation d'un algorithme exécutant séquentiellement des tâches qui ont été allouées à un processeur. Lorsque l'on a besoin de données pour exécuter une tâche, on effectue une phase de communication en faisant appel aux fonctions d'une bibliothèque appropriée. Pour l'heure, ces bibliothèques de fonctions sont essentiellement à passage de message. Les plus célèbres d'entre elles sont: *PVM (Parallel Virtual Machine)* et *MPI (Message Passing Interface)*. Ces bibliothèques permettent à un ensemble d'ordinateurs de fonctionner comme une "machine parallèle virtuelle". Le principe étant de fournir à l'utilisateur des moyens de construire une application parallèle à base de processus. Chaque processus est exécuté sur un des processeurs de la machine parallèle virtuelle et peut communiquer avec d'autres processus. Les programmes utilisateurs font appel à des fonctions contenues dans des bibliothèques incluses lors de l'édition des liens. Les langages de programmation utilisés dans ce cas sont le *FORTRAN* et le *C*. Il existe également d'autres approches

telles que celles utilisant des langages entièrement dédiés à la programmation par échange de message. Par exemple le langage *OCCAM* [May83] qui est employé pour la programmation sur des *Transputers* [ltd88]. Le problème de tels environnements est leur portabilité et leur interfaçage avec des programmes existant et écrits en *FORTRAN* et *C*.

Pour des programmes échangeant des messages tels que des matrices, des vecteurs ou des scalaires, les *BLACS*[DW95] constituent une bibliothèque de haut niveau très intéressante. Dans le chapitre 4, nous présentons quelques fonctionnalités des *BLACS*. Toutefois, il demeure toujours de la responsabilité du programmeur de spécifier toutes les phases de communications.

Afin de faciliter la programmation, des opérations standards autres que les envois ou les réceptions sont intégrées dans les environnements de programmation parallèle. Elles donnent ainsi aux programmeurs plusieurs niveaux d'abstraction dans l'implantation des algorithmes.

A titre d'exemple, les *BLAS* [LHKK79] sont aujourd'hui fournis en standard par les constructeurs de machines pour des opérations d'algèbre linéaire telles que:

- ⋆ les manipulations d'opérations purement vectorielles (*BLAS-1*),
- ⋆ les manipulations d'opération du style matrice vecteurs (*BLAS-2*),
- ⋆ les manipulations d'opération du style matrice matrice (*BLAS-3*).

Les *BLAS* sont dédiés essentiellement au traitement des matrices pleines ou bandes. Les machines ciblées sont des monoprocesseurs ou des multiprocesseurs à mémoire partagée. Dans le cadre de machines à mémoire distribuée, les *PB-BLAS* [CDW96] en sont une extension. D'autres versions, non standard, sont proposées pour des matrices creuses. On peut par exemple consulter *http://www.netlib.org/scalapack/index.html* pour une présentation de telles bibliothèques. Notons qu'avec la multiplicité des modes de stockages des matrices creuses, il est difficile de trouver des bibliothèques couvrant tous les stockages creux.

Un autre exemple de bibliothèques de programmes: *LAPACK* [ABB+95]. Il s'agit d'une bibliothèque de programmes traitant la résolution de problèmes classiques de l'algèbre linéaire telles que:

- ⋆ les résolutions de systèmes linéaires,
- ⋆ les résolutions de problèmes spectraux ou de décomposition en valeurs singulières.

Les machines ciblées par *LAPACK* sont des monoprocesseurs ou des multiprocesseurs à mémoire partagée. Elle trouve une extension pour les machines à mémoire distribuée à travers la bibliothèque de programmes *ScaLAPACK* [BCC+97]. Notons que *LAPACK* ou *ScaLAPACK* ne traitent que des méthodes numériques à matrices pleines. Dans le cadre de matrices creuses, le lecteur peut par exemple se référer à [SGM95].

Au total, les bibliothèques de programmes qui sont proposées permettent de véritables environnements de développement de programmes parallèles. La figure (2.3) illustre les divers niveaux d'abstraction dans un tel développement. Le plus bas niveau est celui de la programmation à l'aide d'appel à des fonctions de communication alors que le plus élevé correspond à des appels de fonctions exécutant des calculs en parallèle. Cependant, des fonction décrivant la distribution des données et des calculs sont nécessaires afin de permettre aux processus, d'opérer sur des données en entrée, en effectuant des phases de calcul et de communication,

pour produire des résultats. Dans bon nombre de cas, les distributions de données et de calculs qui sont intégrés dans ces environnements de programmation utilisent des fonctions de placement linéaire.

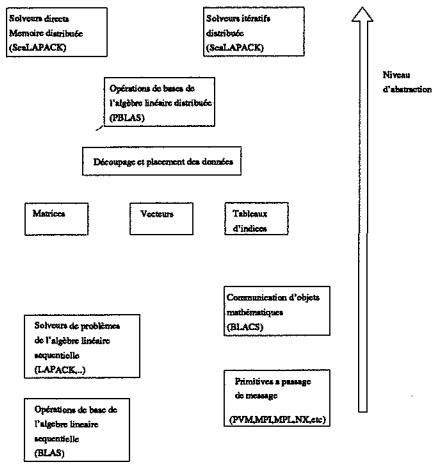

FIG. 2.3 – Niveaux d'abstraction dans la programmation parallèle explicite.

Les environnements à parallélisme implicite

Ces environnements s'articulent autour de langages de programmation tels que:

- ⋆ les langages sans séquencement tels que le *PROLOG*. Ils ne sont quasiment pas utilisés dans le domaine de la parallélisation des méthodes numériques;
- ⋆ les langages impératifs tels que les langages *data-parallèle* qui sont très répandus dans le domaine de la parallélisation automatique des codes de calculs scientifiques (comme par exemple *HPF (High Performance Fortran)* [KLS+94]).

Le data-parallélisme est une expression du parallélisme à travers une exécution simultanée d'une même opération appliquée à tout un ensemble de données. Un programme data-parallèle est donc une séquence de telles instructions. Le placement des données étant acquis, il est de la responsabilité du programmeur de spécifier les opérations dites *data-parallèles*, sur les tableaux ainsi répartis. Dans les langages data-parallèles, les placements de données imposent des règles d'écriture locales des programmes. Conformément à ces règles, le processeur qui effectue le calcul est celui qui possède la donnée. Dans le langage *HPF*, les directives *FORALL* ou *INDEPENDENT* permettent d'identifier les boucles parallèles. Ces instructions sont en général spécifiées lors de la description des boucles dans les programmes *HPF*.

La compilation génère ainsi une structure de programme de type *SPMD* en procédant comme suit:

- ⋆ explicitation des communications par la règle dite du "*propriétaire du calcul*",

⋆ optimisation du code *SPMD.*

Les enjeux majeurs des environnements tels que *HPF* se situent dans l'optimisation des communications engendrées par les calculs et la distribution des données. Celles-ci sont essentiellement liées aux traitements des indices des ensembles de données. Lorsque l'indexation de tels ensembles est réalisée par des fonctions affines, les communications qui en découlent peuvent être explicitées et optimisées à la compilation. On parle alors de communications régulières. Dans le cas où l'indexation est faite par des fonctions non affines, les communications deviennent irrégulières. Celles-ci ne pourront être explicitées et optimisées qu'à l'exécution.

Les communications régulières peuvent être optimisées à la compilation par la vectorisation des messages. Dans le cadre de communications irrégulières, les techniques utilisées sont celles dites de l'*inspecteur/exécuteur* [DPM91]. Dans une première phase, l'inspecteur exécute une version simplifiée du code *SPMD* afin de déduire des listes d'itérations locales, des données à émettre et de données à recevoir. Dans la seconde phase, l'exécution des calculs se réduit alors au parcours de ces listes.

On peut a priori, envisager deux classes d'approches pour organiser les communications et les rangements de données [Fea96b]:

1– dans un premier temps, on peut supposer que les tâches sont placées de manière arbitraire sur les processeurs d'une machine parallèle et que la circulation peut être améliorée au cours de l'exécution d'un programme en faisant soit migrer des blocs d'instructions, soit des données;

2– dans un second temps, il s'agit de rechercher une répartition idéale des calculs et des données avant le déroulement du programme.

2.2.2.2 Structures de données pour la mise en oeuvre

Tant que les machines parallèles sont à mémoire communes, les stratégies de rangement de données sont celles utilisées dans les machines séquentielles standards. Dans le cas des manines à mémoire distribuée, les données doivent être découpées et placées sur les processeurs disponibles. Ce problème est incontournable dans le cadre d'une programmation data-parallèle ou à passage de messages.

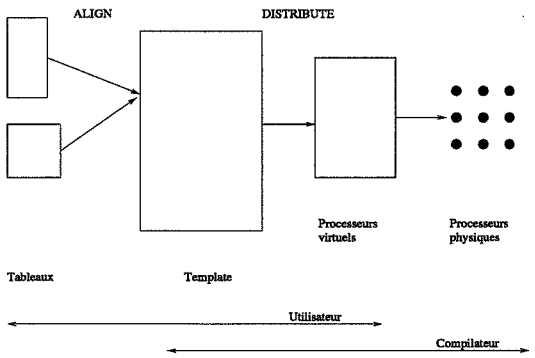

FIG. 2.4 – Distribution des données dans le cadre du parallélisme implicite.

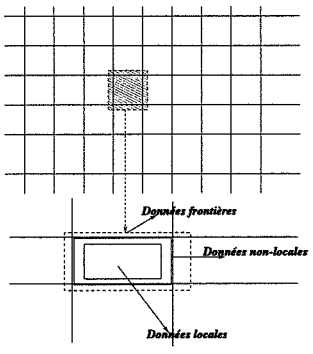

FIG. 2.5 – Distribution des données dans le cadre du parallélisme explicite.

A titre d'exemple, nous avons décrit dans la figure (2.4), le principe général de l'implantation du partitionnement des données à l'aide de *HPF* [pfF94] et qui se généralise bien aux langages data-parallèle. Il consiste en trois phases. La première phase est spécifiée dans cette figure par la directive *ALIGN*. Elle met en relation via des tableaux spéciaux, les *Templates*, les ensembles de données susceptibles de s'exécuter sur le même processeur. Dans la seconde phase, la directive *DISTRIBUTE* permet de réaliser un placement des données en relation dans un tableau de type Template sur des processeurs virtuels. Notons que les données manipulées ici sont essentiellement des tableaux et que la distribution est réalisée point par point, bloc par bloc, cyclique ou bloc cyclique. Enfin, dans la troisième phase, le placement des processeurs virtuels sur des processeurs physiques est réalisé par le compilateur, indépendamment de la machine parallèle ciblée.

Dans le cadre d'une programmation à passage de messages, le rangement des données sur chaque processeur est identique à celui d'un programme séquentiel standard. La métaphore employée pour le décrire se fonde sur le concept de décomposition de domaine. Dans la figure (2.5), nous illustrons trois types de données nécessaires à la description des algorithmes parallèles. Ainsi, on distingue:

1– les données dites "locales" qui correspondent à des données reçues par un processeur au cours d'une phase de distribution de données et qui contribuent aux calculs locaux,

2– les données "non-locales" qui sont celles qu'un processeur doit envoyer à d'autres processeurs,

3– les données dites "frontières" qui sont celles reçues par un processeur en provenance d'autres processeurs.

Ces données doivent être construites et placées explicitement.

2.2.2.3 Méthodologies de parallélisation manuelle

En calcul scientifique, les méthodologies de parallélisation peuvent être classées selon le mode de décomposition de l'application séquentielle. Ainsi, on distingue la décomposition par les données, la décomposition par les calculs et diviser pour paralléliser.

La décomposition par les données

Dans cette classe, les stratégies favorisent la décomposition des applications suivant leurs données. Une telle décomposition est motivée par la présence de donnée de grande taille dans les méthodes numériques. Il semble beaucoup plus naturel d'aborder leur parallélisation en décomposant leurs données. On crée ainsi un ensemble de tâche en procédant en deux étapes comme suit:

- un ensemble de données est décomposé en sous-structures correspondant à des données qui vont être traitées par des processeurs;
- pour chacune de ces sous-structures, une analyse du schéma d'exécution séquentiel permet de définir les tâches dont l'exécution implique la contribution de la sous-structure.

On retrouve dans ces stratégies, les similitudes avec les techniques de *décomposition de domaine* [Saa96]. Lorsqu'un processeur est en charge d'une sous-structure, il est responsable

du traitement des tâches nécessitant une contribution de la sous-structure. Dans le cas où une tâche requiert des informations liées à d'autres sous-structures, on se met en phase de communication. Une telle phase peut alors se comparer à la résolution d'un problème aux frontières. Lorsque les décompositions des données sont relativement simples, les phases de communication se modélisent assez bien. Ceci est par exemple le cas lors du traitement d'algorithmes numériques à matrice pleines [LJ93a]. Lorsque les matrices sont creuses, les phases de communications peuvent impliquer plusieurs processeurs et le graphe des communications peut devenir difficile à construire de façon systématique [Aa94].

Ces stratégies s'illustrent dans le cadre de la programmation à passage de messages. La programmation à passage de messages permet de définir des programmes parallèles en utilisant des extensions des structures de données séquentielles. Elle est ainsi à la base de l'implantation de la plupart des bibliothèques parallèles de calcul scientifique [SM95]. Un défaut majeur de la décomposition par les données est que l'utilisateur est censé connaitre la circulation des informations dans ses programmes. De plus, il n'est pas toujours simple de trouver systématiquement la bonne distribution des données. D'où la nécessité de trouver et d'automatiser de bons critères de découpage et de placement des données [Fea96b, PE96].

La décomposition par les calculs

Dans cette classe, on favorise le découpage par les calculs. Il s'agit de détecter et d'exploiter le parallélisme lié à l'exécution des calculs (analyse des dépendances). Notons que ces calculs peuvent aussi être ceux fournis par les tâches d'un graphe issu d'une décomposition des données. Dans tous les cas, on exploite les dépendances entre les tâches. D'où des similitudes avec la parallélisation automatique. La démarche adoptée procède ainsi qu'il suit:

1– décomposition à grains fins des calculs d'un algorithme séquentiel,

2– agrégation des calculs à grains fins dans des super-noeuds.

Le plus difficile est donc de définir, pour une machine d'exécution, la taille et le nombre de super-noeuds produisant l'exécution parallèle optimale. Il s'agit là d'un problème *NP-complet* [KL88] dont les solutions s'obtiennent via des heuristiques. Dans le chapitre 3, nous discutons en particulier, d'heuristiques de partitionnement utilisées dans le domaine de la synthèse des architectures systoliques.

Une des heuristiques est l'agrégation des calculs à grains fins en des super-noeuds exécutés de façon atomique [Bab84]. Cette heuristique permet de réduire les communications mais ne permet pas toujours un ordonnancement optimal si on se place sur des modèles de machines théoriques telles que les *PRAM*. Cependant, les solutions fournies sont dans la pratique parmi les plus performantes sur des machines *MIMD*. C'est à cette classe qu'appartiennent les techniques de "*pavage*" citées plus tôt (section 2.2.1).

Etant donné un graphe dont les tâches sont des super-noeuds, on a besoin de résoudre de façon optimale l'allocation des tâches. Pour accomplir une allocation des tâches, il existe des méthodes génériques qui décrivent les procédures à suivre. Il existe aussi des outils qui assurent l'allocation: soit dans les cas statiques [Pel97], soit dans les cas dynamiques [ZZWD93]. L'idée est toujours de pouvoir intégrer une méthode ou un outil d'allocation dans un programme avec pour but, des performances de plus en plus intéressantes.

Avec les machines à mémoire distribuée et à communication à passage de message, les algorithmes parallèles doivent être implantés sous la forme de processus séquentiels ayant une

mémoire privée et communicant par échange de message [Hoa78]. En particulier, l'exécution d'un programme parallèle, sur une telle machine, nécessite que les processus communicant soient chargés sur ses processeurs. Dans le cas où on suppose que tous les processus coexistent simultanément pendant toute la durée d'exécution du programme, on parle alors de problème de *placement* pour nommer le problème d'ordonnancement parallèle. Un placement est dit *statique* s'il a lieu avant l'exécution d'un programme et n'est jamais remis en cause pendant la durée d'exécution du programme, et *dynamique* si les processus peuvent être déplacés en cours d'exécution.

Diviser pour paralléliser

Cette classe favorise la technique du "*diviser pour régner*" (*divide and conquer*). Elle consiste à diviser un problème en sous-problèmes plus petits, dont les résolutions permettent de construire la solution du problème initial. Dans le cas où les sous-problèmes sont de même nature que le problème initial, ce processus peut être appliqué récursivement, jusqu'à ce que leur taille permette leur résolution.

Le graphe de tâches est un *arbre de récursion*. Les noeuds d'un même niveau sont donc indépendants, et peuvent être exécutés en parallèle. En fait, il s'agit d'un cas particulier de l'approche "*diviser pour régner*" standard dans laquelle, un appel récursif d'un *noeud fils* est exécuté par un processeur différent de celui du *noeud père* [Qui87].

Dans la section 2.3, nous illustrons ces différentes méthodologies pour les méthodes numériques à matrices de grande taille. Notons tout de même que pour ces méthodes numériques, la décomposition des données est inévitable avec la taille des matrices, tandis que pour la deuxième et la troisième classe, le problème est de déduire une stratégie de décomposition et de distribution des données qui préserve les propriétés d'optimalité de l'ordonnancement des tâches. Ainsi, de nombreux environnements de programmation offrent des outils d'allocation permettant des distribution de données cyclique, bloc et bloc cycliques [LJ93b] très régulières. Pour des applications à matrices creuses, ces distributions ne valident cependant pas toujours l'équilibrage de la charge de calcul des processeurs.

2.2.2.4 Limites de la parallélisation manuelle

Les méthodologies de parallélisation manuelle conduisent à une manipulation explicite d'un graphe de tâches. Les graphes de tâches unanimement reconnus comme décrivant le parallélisme potentiel des applications, sont des graphes de dépendance à flots de données [Kum82]. Cependant, la modélisation d'un gros logiciel manuellement sous la forme d'un graphe de dépendance à flots de données semble peu réaliste. Aussi dans ces cas, on utilise des approches de parallélisation progressives. Elles consistent comme l'illustre la figure (2.6) en un ensemble de modifications progressives, sur des sous programmes d'un code séquentiel. L'ensemble de ces modifications permet de construire une structure de code parallèle par assemblage des sous programmes parallélisés.

Dans cette boucle de portage, les mesures de performances de la parallélisation menées sur le programme ainsi construit permettent à l'utilisateur, de juger de la qualité de la parallélisation. Si celle-ci est considérée comme inefficace, les causes de cette inefficacité doivent être décelées. Au nombre de celles-ci, citons: les sous-programmes mal parallélisés, la stabilité

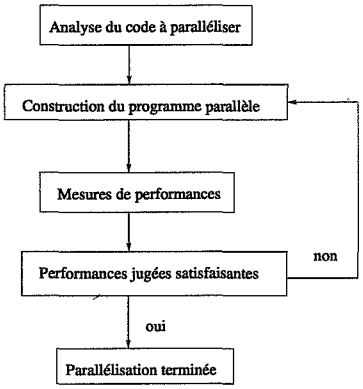

FIG. 2.6 – Boucle de portage d'un code de calcul séquentiel sur machines parallèles.

numérique et la portabilité.

les sous-programmes mal parallélisés:

la parallélisation optimale d'un sous-programme en vue de sa réutilisation dans un code de calcul nécessite des analyses qui vont au delà de celles classiquement proposées pour des algorithmes pris individuellement. En effet, il faut éviter une explosion de la place mémoire et du temps de calcul requis par l'exécution d'un code. Or, en rajoutant implicitement ou explicitement une modélisation sous forme de graphe de dépendance, les coûts en termes de structures de données pour traiter de façon optimale, le partitionnement, l'ordonnancement et l'allocation, peuvent devenir prohibitifs dans certains cas.

Un cas particulier est celui de la parallélisation des sous-programmes tels ceux manipulant des opérations sur des matrices creuses et creuses skylines. En parallélisant efficacement ces sous-programmes, on peut espérer une amélioration des performances du code séquentiel. Il existe des méthodes et outils permettant de paralléliser ces opérations (voir section 2.3). Malheureusement, les sous-programmes qui en résultent sont souvent difficiles à réintégrer dans un code ou alors le sont au prix de pré-traitements qui peuvent très vite, devenir prohibitifs. Ceci est spécifique au mode de stockage creux. En effet, pour accéder à un coefficient d'une matrice creuse, dont les indices ligne et colonne sont i et j respectivement, on calcule son adresse via des vecteurs qui dépendent de i et j. Dans la mesure où les positions des termes non-nuls par ligne varient de façon aléatoire, les accès à ces termes deviennent difficiles à traiter et compliquent la modélisation du graphe de dépendance. Un tel graphe de dépendance ne peut alors être construit que lors de l'exécution du programme. On a donc, à titre d'exemple, besoin de paralléliseurs automatiques qui construisent les graphes de dépendance à l'exécution des programmes. Dans la mesure où l'on désire paralléliser explicitement ces sous-programmes, on a besoin de méthodes de parallélisation du style *diviser pour paralléliser*. Ces méthodes de parallélisation ont tendance à modifier le profil des matrices afin de construire des schémas d'exécution parallèles intéressants.

La stabilité numérique:

elle se pose dans la mesure où, en exhibant le parallélisme dans un schéma numérique, on peut être amené à modifier l'ordre d'exécution des opérations. Or cette modification, en arithmétique à précision finie, peut être source d'erreurs. Ainsi, en réutilisant des résultats partiels, il peut y avoir une propagation des erreurs de calcul. Il n'est donc pas sain d'utiliser une stratégie de parallélisation ou un sous-programme parallèle existant sans en connaitre l'ordre d'exécution d'opérations élémentaires par rapport à celui du sous-programme séquentiel.

La portabilité:

elle a longtemps constitué le point faible de la parallélisation aussi bien implicite qu'explicite. Aujourd'hui, grâce au développement de directives standards (style *HPF*) pour la parallélisation implicite ou de bibliothèques de communications telles que *PVM* ou *MPI* pour la parallélisation explicite, il est possible de transformer un code écrit en *Fortran 77* ou en *C* en une application à base de processus communicant (selon la méthode UNIX). Ces processus peuvent alors être exécutés sur une grande classe de machines parallèles. Toutefois, ces modes de programmation portables impliquent un partitionnement implicite ou explicite des données.

Ainsi, la mise en oeuvre d'un code parallèle à partir d'appels de sous-programmes parallèles pose au moins trois questions:

- existe-t-il une adéquation entre: les rangements de données des divers sous-programmes parallèles et ceux du code séquentiel?
- peut on envisager d'intégrer raisonnablement les phases d'ordonnancement et de placement dans un code de calcul existant? Si oui comment et quelles en sont les complexités en temps et en espace?
- comment peut on garantir systématiquement un environnement permettant de choisir les meilleures fonctions d'ordonnancement et placement, étant donné un code appelant plusieurs sous-programmes parallèles.

2.3 Parallélisation de méthodes numériques a matrices de grande taille

Les méthodes numériques à matrices de grande taille présentent deux caractéristiques fondamentales. D'une part, les objets qu'elles manipulent sont des matrices, des vecteurs ou des scalaires. D'autre part, le traitement des calculs est mis en oeuvre par des boucles imbriquées. Au nombre des objets manipulés, les matrices sont les plus gourmands en espace mémoire. Afin d'étudier la parallélisation de telles méthodes, nous rappelons des concepts liés aux objets manipulés. Soient n et m deux entiers positifs et non nuls.

Définition 1 *On appelle vecteur x, de longueur n, l'ensemble des coefficients $(x_i)_{1\leq i\leq n}$.*
On appelle matrice A, de taille $n \times m$, l'ensemble des coefficients $(a_{i,j})_{1\leq i\leq n, 1\leq j\leq m}$.

Les coefficients de A ou de x sont des scalaires. Le stockage de A correspond à celui de mn scalaires et celui de x à n scalaires. La définition 1 se généralise au cas non scalaire. On parle alors de matrices ou de vecteurs par blocs (voir figure (2.7)). Lorsque la matrice A est telle que $m = n$, A est une matrice carrée d'ordre n.

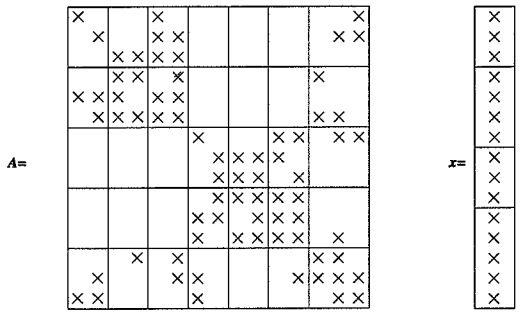

FIG. 2.7 – Découpage par bloc d'une matrice et d'un vecteur

Dans la pratique, les matrices de grande taille présentent deux particularités liées à leur ordre et leur nombre de coefficients non nuls. En effet, leur ordre peut être supérieur à 10^4 tandis que la proportion des coefficients non nuls est très petite devant 1 [LP96, RT83, Gas66]. On dit qu'elles sont de grande taille et creuses. Dans ce contexte, outre leur traitement, le premier problème que doit résoudre une méthode numérique à matrice de grande taille est le stockage (des coefficients utiles). Aussi, dans cette section, nous exposerons d'abord les stockages couramment utilisés. Puis nous montrerons comment les méthodes numériques présentées s'appliquent au cas d'un mode de stockage dit skyline (ou ligne de ciel) particulièrement utilisé dans l'industrie.

2.3.1 Méthodes numériques pour matrices creuses de grande taille

Dans cette section, nous présentons comment dans le cas général les problèmes de stockage et de traitement sont abordés pour les méthodes numériques à matrices creuses de grande taille.

2.3.1.1 Rangement des matrices

Les stratégies utilisées pour stocker les coefficients utiles d'une matrice creuse de grande taille dépendent beaucoup de la vision que le programmeur peut avoir de la mémoire. Sans nuire à la généralité, nous distinguons les stratégies pour mémoire commune et les stratégies pour mémoire distribuée.

Stratégies pour les machines à mémoire commune:

puisque dans ce cas l'utilisateur dispose d'une vision globale de la mémoire, les techniques de rangement procèdent de la même façon que sur les machines séquentielles.

Il existe plusieurs stratégies de rangement possibles. Un état de l'art de ces stratégies est proposé dans [BBC+94]. La stratégie la plus générale consiste à ne représenter que les termes non-nuls. Elle peut être mise en oeuvre ligne par ligne ou colonne par colonne. Elle requiert

	Ligne 1			Ligne 2					...
	1	2	3	4	5	6	7	8	...
val	$a_{1,1}$	$a_{1,2}$	$a_{1,8}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,6}$	$a_{2,9}$	...

	Ligne 1			Ligne 2					.....
	1	2	3	4	5	6	7	8	.....
col_ind	1	2	8	1	2	3	6	9	.....

	1	4	3	4	5	6	7	8	9	10	11	12	13	14	15	16
ptr_row	1	4	9	14	19	23	28	33	38	45	51	55	62	66	70	74

FIG. 2.8 – Rangement ligne par ligne d'une matrice creuse.

trois tableaux au minimum, comme illustré par la figure (2.8), pour un rangement ligne par ligne des termes non-nuls d'une matrice:

- ⋆ un tableau *val* dans lequel sont rangés les coefficients non-nuls de la matrice,
- ⋆ un tableau *col_ind* dans lequel sont rangés les indices de colonne des coefficients non-nuls de la matrice,
- ⋆ un tableau de *ptr_row* qui indique pour chaque ligne l'adresse de son premier élément dans le tableau *val*.

Pour une matrice d'ordre n ayant nz coefficients, cette stratégie coûte $2nz + n + 1$ contre n^2 pour un stockage plein. Toutefois, elle n'est pas bien adaptée pour les opérations d'insertion ou de suppression de coefficient, et requiert des indirections pour l'accès aux coefficients. En effet, pour des méthodes numériques qui modifient le profil de la matrice[LT86b], l'élimination de *Gauss*, la factorisation QR, etc, on peut assister à la génération de nouveaux coefficients non nuls (nuls) qu'il va donc falloir insérer (supprimer). Quant au problème des indirections, il est commun à toutes les méthodes numériques à matrices creuses. En effet, toute référence à un coefficient stocké passe par une utilisation implicite des tableaux *ptr_row*, pour la localisation du premier terme non-nul d'une ligne et *col_ind* pour la détermination de l'indice colonne. Lorsque l'ordre et le nombre de coefficients non nuls sont très grands, on observe en pratique que les multiples indirections peuvent entrainer des surcoûts lors du traitement de simples opérations scalaires.

Les solutions apportées à ces problèmes sont complémentaires. En effet, elles consistent à accepter de stocker des termes nuls tout en réduisant le nombre total d'indirections. Dans cet ordre d'idée, on peut citer:

- les stockages par blocs de sous-matrices denses, qui sont une généralisation des stockages ligne par ligne ou colonne par colonne à des matrices par blocs [BBC+94],
- les stockages par vecteurs de longueur n, des diagonales d'une matrice [BBC+94],
- les stockages *skyline* ou *ligne de ciel* discutés dans la section 2.3.2.

Tous ces stockages peuvent être réutilisés dans la mesure où le mode de programmation parallèle n'implique pas une décomposition des données. Cependant, afin de bénéficier de l'organisation hiérarchique de la mémoire des processeurs, les stockages de matrices par blocs sont les plus utilisés. Notons également que les stockages par diagonale se prêtent bien aux calculs sur les machines vectorielles [Saa96]. C'est dans les stratégies de programmation à passage de messages ou data-parallèle que des extensions de ces stockages sont incontournables.

Stratégies pour les machines à passage de message:

pour ce cas, des matrices et des vecteurs traités sont nécessairement partitionnés. Ainsi, en raison du coût élevé des communications induit par le partitionnement des calculs, on a de plus en plus recours à une décomposition par blocs de données adjacentes ainsi que décrit dans [Eij92]. Soit $A = (a_{i,j})$ une matrice donnée. Notons E l'ensemble des couples (i,j) tels que les temes $a_{i,j}$ ont été alloués à une processeur $\mathcal{P}$. D'un point de vue mathématique, les coefficients alloués à $\mathcal{P}$ forment une matrice, notées A_E, et définie par:

$$(A_E)_{i,j} = \begin{cases} a_{i,j} \text{ si} & (i,j) \in E \\ 0 \text{ sinon} \end{cases} \tag{2.4}$$

Soit

$$\mathcal{I} = \{i : \exists j \ (i,j) \in E\} \tag{2.5}$$

et

$$\mathcal{J} = \{j : \exists i \ (i,j) \in E\} \tag{2.6}$$

En reindexant les éléments de $\mathcal{I}$ et $\mathcal{J}$, on montre que la matrice A_E est de format $|\mathcal{I}| \times |\mathcal{J}|$. En outre, elle peut être stockée comme par le biais des stockages de matrices standards. Cependant, il convient de disposer localement des informations permettant le passage de la réindexation locale à l'indexation globale. Dans le cadre de la programmation data-parallèle, ces problèmes sont résolus grâce à des directives de placement de données [PE96]. La figure (2.9) illustre des tableaux spécifiant un placement de données pour un stockage ligne par ligne. Ce mode de rangement reprend le format *Ellpack* [Saa96]. De plus, il nécessite un tableau Ic indiquant les placements de données sur des processeurs virtuels.

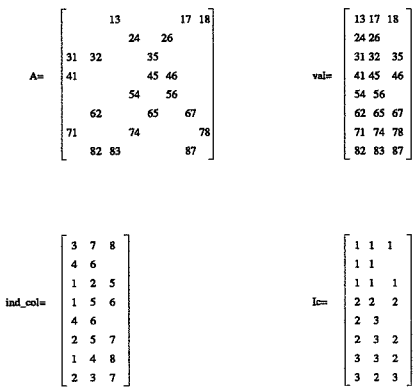

FIG. 2.9 – Exemple de stockage data-parallèle ligne par ligne d'une matrice.

2.3.1.2 Méthodologies de parallélisation

Les méthodes de parallélisation reposent peu ou prou sur une forme de détection des dépendances dans les enchainements des étapes de calcul. On distingue deux grandes approches.

La première exploite le parallélisme lié au traitement des itérations des boucles imbriquées [Lil94]. L'analyse des méthodes numériques, en tant qu'ensemble de boucles imbriquées a fait l'objet de nombreuses études. Les résultats les plus significatifs sont établis dans le cas où les algorithmes présentent une forme de régularité dans le traitement des étapes de calculs et dans les structures de données (nids de boucles affines). Cette régularité est exploitée dans les outils de parallélisation tels que les compilateurs paralléliseurs ou dans les compilateurs de langages à parallélisme implicite. Lorsqu'il apparait des irrégularités dans le traitement des calculs et dans les rangements des données, l'inconvénient de ces approches est que la taille du graphe de tâche augmente très vite et la résolution des problèmes d'ordonnancement et de placement devient très vite coûteuse. Ceci est notamment le cas lorsque les matrices sont rangées en utilisant des stockages creux.

Lorsque les matrices sont creuses, il devient difficile de réaliser l'analyse des dépendances de manière automatique. En effet, les accès aux coefficients des matrices sont faits par des vecteurs d'indirection. Ainsi, quand on fait appel à un coefficient $a_{i,j}$, on passe implicitement par l'écriture suivante:

$$a_{i,j} := val(col\_ind(ptr\_row(i) + c)) \tag{2.7}$$

où c est la position du coefficient $a_{i,j}$ par rapport au premier terme de la ligne i. De tels accès peuvent cacher des communications entre des processeurs. La maitrise du comportement de telles communications est en outre rendue impossible par une méconnaissance totale de la localisation des coefficients de la matrice. On parle alors d'algorithmes *irréguliers.*

Pour de tels algorithmes, la modélisation formelle en terme d'un ensemble de boucles imbriquées est loin d'être suffisante. D'où des méthodes qui demandent une certaine connaissance du problème physique qui est traité. Dans cet ordre d'idée, on trouve de nombreuses approches utilisant le principe du *diviser pour paralléliser* [Aa94].

La méthode habituelle pour simuler informatiquement les problèmes physiques consiste à les discrétiser dans le temps et dans l'espace. Pour cela, on calcule un graphe maillé dont les sommets portent les quantités physiques en un point de l'espace, et dont les arêtes indiquent des dépendances spatiales entre les sommets. Lorsque les dépendances entre les variables aux sommets sont linéaires, le problème peut être traité comme un système linéaire et modélisé sous forme matricielle. Ces dépendances se modélisent dans le cadre d'un *graphe d'adjacence.*

Définition 2 *Soit A une matrice d'ordre n. On appelle graphe d'adjacence de la matrice A, le graphe dont:*
les sommets correspondent chacun à une ligne de la matrice, soit $\{1, 2, \cdots, n\}$
l'ensemble des arcs est défini par $\{(i, j);\ a_{i,j} \neq 0\}$

A titre d'exemple, la figure (2.10) présente une numérotation assez simple d'un graphe d'adjacence et la figure (2.11) la structure de matrice creuse correspondante. Comme le montre la figure (2.12), on peut renuméroter les noeuds d'un tel graphe d'adjacence de façon à construire une structure de matrice creuse par blocs illustrée par la figure (2.13).

Cette approche est illustrée dans la plupart des méthodes numériques à matrices creuses proposées dans la littérature [Duf96]. L'idée la plus répandue consiste à introduire de la

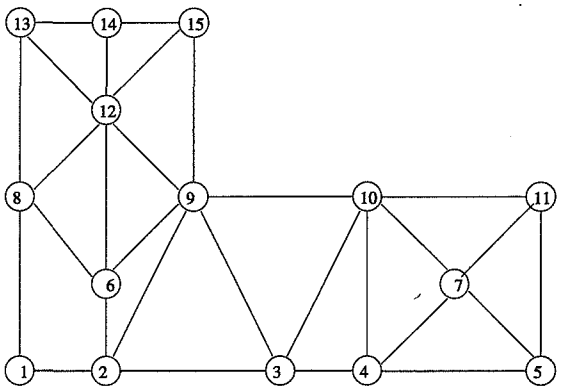

FIG. 2.10 – Graphe d'adjacence d'une matrice creuse.

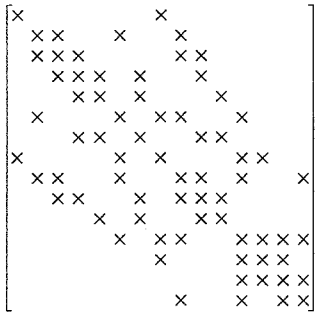

FIG. 2.11 – Exemple de matrice creuse de taille 15 × 15.

localité des données grâce à une restructuration des matrices creuses sous la forme d'une matrice par bloc. Celles-ci sont possibles avec des outils tels que *Metis* [KK95] ou *Scotch* [Pel97]. Les traitements sont alors organisés en raisonnant sur des blocs de lignes ou de colonnes consécutives. On obtient ainsi des graphes de dépendance entre des super-noeuds correspondant aux traitement de blocs de ligne ou de colonne. Le traitement de ces super-noeuds peut être partagé par plusieurs processeurs.

Malheureusement, il existe des classe de matrices creuses telles que les matrices bandes ou skylines qui ne se prêtent pas bien à cette approche. L'inconvénient de ces méthodes est qu'elles sont très sensibles à la numérotation des inconnues. Dans la section suivante, nous présentons cette famille de matrices creuses.

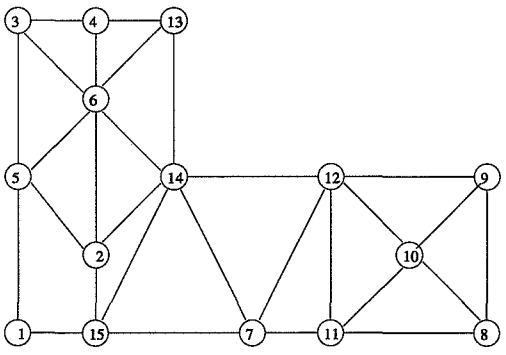

FIG. 2.12 – Graphe d'adjacence d'une matrice creuse après renumérotation.

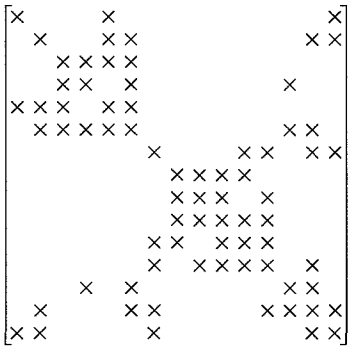

FIG. 2.13 – Exemple de matrice creuse de taille 15 × 15 obtenue après une renumérotation.

2.3.2 Méthodes numériques parallèles pour des matrices skylines

Lorsque l'on ne cherche à stocker que les termes non-nuls, la gestion des rangements des coefficients n'est pas simple. Elle implique une simulation des divers calculs sans valeurs numériques afin de définir la structure générale du graphe d'adjacence des matrices ainsi modifiées. Toutefois, cette étape de pré-traitement peut être évitée en acceptant de stocker quelques termes nuls. Ces techniques font intervenir le concept de *remplissage à l'intérieur d'un profil.*

Dans certaines méthodes numériques à matrices creuses, les opérations utilisées modifient la structure du graphe d'adjacence des matrices. Au nombre de ces opérations, on peut citer les multiplications de matrices creuses entre elles. Les principales méthodes numériques qui requièrent de telles opérations sont surtout les méthodes de résolution directes. Les méthodes directes permettent un résolution par élimination successives des inconnues d'un problème. Classiquement, deux méthodes sont utilisées: l'élimination de *GAUSS* et les transformations orthogonales. Celles-ci peuvent s'interpréter matriciellement en termes de factorisation de matrice. Par exemple l'élimination de *GAUSS* s'interprète matriciellement comme une factorisation d'une matrice A, sous la forme $A = LU$, avec L une matrice triangulaire inférieure et U, une matrice triangulaire supérieure. Les méthodes d'élimination à l'aide de transformations orthogonales quant à elles, peuvent s'interpréter en termes d'une factorisation de la matrice A, sous la forme $A = QR$, avec Q une matrice unitaire et R une matrice triangulaire, par exemple dans le problème des moindres carrés, ou de type *Hessenberg* dans le problème de recherche de vecteurs propres (voir [GL83]).

2.3.2.1 Rangement des matrices

Etant donnée une matrice A $n \times n$, appelons j_i l'indice colonne du premier élément non nul de la ligne i c'est à dire que $a_{i,j} = 0$ pour $j < j_i$ et $a_{i,j_i} \neq 0$ et k_i l'indice colonne du dernier élément non nul de la ligne i c'est à dire que $a_{i,j} = 0$ pour $j > k_i$ et $a_{i,k_i} \neq 0$

Définition 3 (Matrice skyline)
On appelle e profil de A est l'ensemble des couples (i,j) défini par:
$Prof(A) = \{(i,j);\ 1 \leq i \leq n \text{ et } j_i \leq j \leq k_i\}$.
Le stockage skyline ou ligne de ciel d'une matrice consiste à stocker son profil.
Par abus de langage, pour désigner une matrice dont les coefficients sont stockés en mode skyline, nous parlerons de matrice skyline.

Cette notion de profil induit une représentation à largeur de bande variable le long de chaque ligne (voir (2.14)). Notons lbw, la largeur maximale du profil sur chaque ligne, encore appelée largeur de bande maximale. La figure (2.15) illustre un rangement ligne par par ligne d'une matrice A. On calcule le premier élément non nul puis tous les éléments du profil.

Les matrices skylines de grande taille sont généralement très intéressantes lorsque $lbw \ll n$. Dans ce cas, on ne perd pas beaucoup de place en réservant, dès le départ, la place mémoire nécessaire au stockage de A, et on gagne en simplicité et en place mémoire nécessaire pour ranger l'information nécessaire pour ranger la structure de A. Les stockages proposés dans le cadre des matrices creuses peuvent être réutilisés. Ces stockages se trouvent simplifiés dans la mesure où tous les termes du profil sont pris en compte.

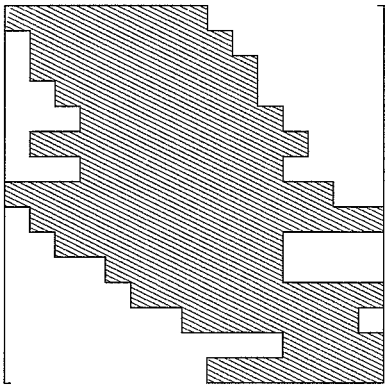

FIG. 2.14 – Profil d'une matrice creuse

	Ligne 1			Ligne 2			...
	1	...	8	9	...	17	...
val	$a_{1,1}$	...	$a_{1,8}$	$a_{2,1}$	...	$a_{2,9}$	...

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
ptr_row	1	9	18	27	35	23	54	62	75	98	98	105	115	122	126	133

FIG. 2.15 – Rangement du profil d'une matrice creuse.

2.3.2.2 Méthodologies de parallélisation

Dans la mesure où les matrices skylines présentent, de par leur nombre de termes non nuls, les caractéristiques des matrices creuses, les méthodologies de la section 2.3.1.2 peuvent être réutilisées. Cependant, du fait de la répartition des termes non nuls, les méthodes à matrice skylines introduisent des difficultés supplémentaires qui rendent inefficaces l'application de telles méthodologies. En effet, analysons la structure d'un graphe d'adjacence d'une matrice skyline en terme de dépendances entre les inconnues d'un problème. On constate que chaque inconnue dépend d'un nombre très faible d'autres inconnues ayant des numéros adjacents. Dans ces conditions, il est difficile d'exploiter un parallélisme dans le traitement des inconnues. Des renumérotations sont donc nécessaires afin de mettre en évidence des structures de graphe exhibant du parallélisme. Ceci justifie les approches diviser pour paralléliser qui sont développées [CD97]. Cependant, ces méthodes nécessitent des heuristiques puissantes permettant de conserver la localité des données et de limiter les remplissages observés dans les méthodes directes à matrices creuses.

Une autre alternative dans la parallélisation des matrices skylines est l'utilisation des méthodologies de parallélisation de méthode numérique à matrices pleines. Ces méthodologies tirent profit de la régularité de la répartition dans un profil. Ces techniques portent essentiellement sur la parallélisation automatiques des "*nids de boucles*" et sont développées dans le cadre de la parallélisation automatique.

2.4 Conclusion

Nous avons présenté un état de l'art des principales classes d'approches de parallélisation des méthodes numériques. Quelle que soit l'approche, on reste confronté à quatre problèmes récurrents: trouver une représentation de l'application parallèle, une méthode de partitionnement de l'application, l'ordonnancement des partitions et leur allocation aux processeurs de la machine d'exécution. Les graphes de tâches sont le modèle le plus utilisé pour la représentation des applications parallèles. La recherche d'une parallélisation optimale donne lieu à la résolution de problèmes d'optimisations combinatoires de la théorie des graphes tels que: le partitionnement, l'ordonnancement et l'allocation, dont les solutions sont interdépen-

dantes. Les solutions proposées pour la parallélisation optimale sont loin d'être définitives. En particulier, pour une grande classe d'algorithmes, le modèle du graphe de dépendance à flots de données entre les calculs semble être celui qui décrit le mieux, leur parallélisme interne. Ce modèle pose toutefois des difficultés notamment dans le traitement des méthodes numériques enchainant des algorithmes à matrices creuses. Quant aux problèmes de partitionnement, d'ordonnancement et d'allocation, ils se posent avec accuité dans le cas général, et avec une acuité accrue pour les méthodes numériques à matrice creuses de grande taille dont les graphes de tâches présentent des irrégularités tels que seules les heuristiques sont généralement proposées comme méthodes de résolution.

Dans le domaine de la parallélisation automatique, une modélisation exacte du graphe de dépendance à flots de données est prohibitive. Les solutions proposées restent à valider (expérimentalement) sur les algorithmes à matrices creuses, soit l'essentiel des applications rencontrées dans la pratique. Une représentation approchée des dépendances peut être utilisée dans ce cas là, mais au prix d'une perte de parallélisme. Quant aux problèmes de partitionnement, ordonnancement et allocation, ils ne sont résolus que dans le cas affin. Même si les placements dans ce cas peuvent être choisis de façon optimale, on ne le peut garantir pour l'ordonnancement qui ne prend pas en compte, le caractère combinatoire des dépendances entre les calculs.

Dans le domaine de la parallélisation manuelle, outre les limites de validité du modèle du graphe de dépendance à flots de données qui sont constatées en parallélisation automatique, le recours aux environnements de programmation à parallélisme virtuel complique les problèmes de partitionnement, d'ordonnancement et d'allocation. Les modèles mathématiques des algorithmes doivent intégrer les coûts prohibitifs des environnements logiciel et matériel pour la mise en oeuvre. On a donc besoin de critères d'évaluation des performances sur des modèles de machines beaucoup plus réalistes que les *PRAM*.

Dans cette thèse, nous proposons une méthodologie de parallélisation pour des méthodes numériques qui s'articulent autour d'un enchainement d'algorithmes à matrices ayant un stockage quelconque. Chaque algorithme est parallélisé de façon optimale tout en respectant les contraintes de coûts en espace mémoire et d'interfaçage avec la méthode numérique. C'est une approche de parallélisation de type pavage qui fusionne:

- les techniques de la synthèse systolique, pour obtenir un schéma d'exécution systolique,
- les idées du partitionnement par blocs des espaces d'itérations, utilisées pour modéliser le coût des communications,
- les propriétés de la manipulation blocs par blocs, des vecteurs et des matrices, pour déduire d'un partitionnement par blocs des espaces d'itérations, des schémas d'exécution par blocs de calcul qui se synthétisent, formellement, par des équations de récurrences.

Cette méthode a pour fondements théoriques le modèle des *systèmes d'équations de récurrence uniformes (SERU)*. Cependant, ce modèle ne traite pas les cas des matrices creuses. Par conséquent, une relaxation des contraintes sur les *SERU* est utilisée pour pouvoir traiter des méthodes numériques à matrices creuses. La relaxation du modèle des *SERU* s'opère a deux niveau:

1– on affaiblit la notion de vecteurs de translation constants. Seules les directions de dépendances sont constantes. Toutefois, dans chaque direction, les vecteurs de dépendance varient de façon aléatoire.

2– recours a des fonctions d'ordonnancement au plus tôt ou au plus tard pour ordonnancer les SERU relaxés en (1).

Les applications parallèles ainsi obtenues sont optimales au sens du lemme de *Brent* ainsi que dans un modèle prenant en compte les coûts des communications. Toutes ces particularités font que, la présente approche de parallélisation se différencie des approches de pavage existant dans la littérature et dans lesquelles, les tuiles sont des copies translatées d'une tuile dite canonique.

Chapitre 3

Nouvelle approche systolique de parallélisation de méthodes numériques

Le but de ce chapitre est de proposer une méthodologie de parallélisation optimale pour des méthodes numériques. Nous avons vu lors du chapitre précédent, que la parallélisation optimale est un problème extrêmement difficile. Nous avons donc préféré mettre en place des heuristiques: de partitionnement, d'ordonnancement et d'allocation, pour parvenir à des algorithmes parallèles optimaux dédiés à des machines *MIMD* à mémoire distribuée.

Dans la section 3.1, nous faisons quelques rappels sur la synthèse des réseaux systoliques. Dans la section 3.2, nous présentons les principes généraux de l'approche de parallélisation. Cette approche s'appuie sur les modèles mathématiques qui sont présentés dans la section 3.3. Nous détaillons ensuite les aspects fondamentaux de l'approche dans la section 3.4, en supposant que pour les méthodes numériques à paralléliser, les calculs reposent sur une organisation régulière des données. Dans la section 3.5, nous discutons des extensions de ces aspects fondamentaux dans le cas des rangements de matrices creuses qui constituent une organisation irrégulière des données, propre aux méthodes numériques. La dernière section est consacrée aux analyses de complexité des algorithmes parallèles générés par la présente méthodologie.

3.1 Rappels sur l'algorithmique systolique

Au début des années 80, l'évolution technologique permet d'envisager des circuits comportant plusieurs milliers de transistors. Ceci rend intéressant la conception d'architectures de machines ayant des facteurs de régularité élevés afin de minimiser les coûts de tels circuits. Le concept d'architectures *systoliques* introduit par *Kung* [Kun82] répond bien à de tels objectifs. Il s'agit des structures parallèles faites de cellules simples, localement et régulièrement connectées. Cette section est consacrée à une présentation des aspects fondamentaux des réseaux systoliques ainsi qu'à ceux des méthodes permettant leur synthèse. Une telle présentation nous permet de justifier le choix de l'algorithmique systolique comme "fil conducteur" d'une stratégie de parallélisation.

3.1.1 Les architectures systoliques

Définition 4 *Une architecture systolique (ou réseau systolique ou système systolique) est ainsi défini dans [Kun82]:*

- ⋆ *machine parallèle spécialisée faite de processeurs ou cellules simples,*
- ⋆ *les processeurs sont localement et régulièrement connectés,*
- ⋆ *les calculs effectués par une telle architecture utilisent à la fois la notion de pipeline et de parallélisme vrai et sont exécutés de façon synchrone.*

Une architecture systolique (figure 3.1) est donc composée d'un grand nombre de cellules élémentaires identiques et interconnectées localement. Chaque cellule (figure 3.2) reçoit des données en provenance de cellules voisines, effectue un calcul, puis transmet les résultats à des cellules voisines un temps de cycle plus tard. Seules les cellules situées à la frontière du réseau communiquent avec le monde extérieur i.e sont physiquement connectées à un ordinateur hôte ou à une mémoire externe. Les cellules évoluent en parallèle sous le contrôle d'une horloge globale. Ainsi, plusieurs calculs sont effectués simultanément et on peut enchaîner la résolution de plusieurs instances du même problème.

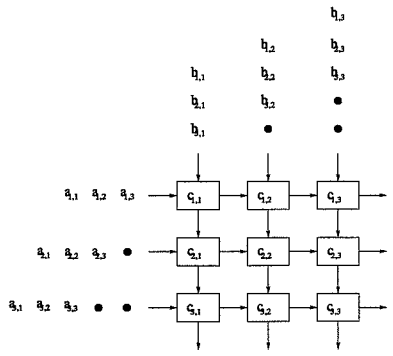

FIG. 3.1 – Architecture systolique pour le produit de deux matrices de taille 3×3.

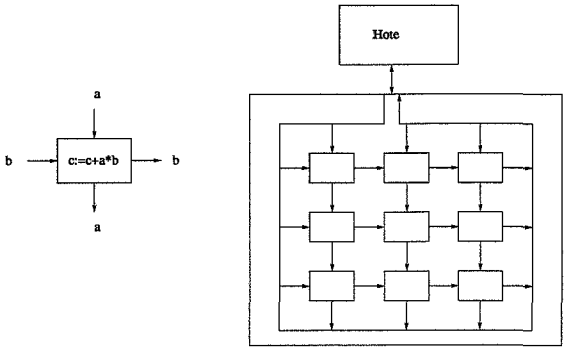

FIG. 3.2 – Organisation de l'architecture systolique et programme d'une cellule élémentaire.

L'idée fondamentale est d'utiliser de telles machines comme périphérique d'un calculateur dit "*hôte*" pour accélérer des morceaux de programmes prohibitifs en temps de calcul. La conception d'une architecture systolique, nécessite une simulation préalable, afin de s'assurer de sa validité par rapport aux critères de la définition (4). L'objectif d'une telle simulation est donc double:

1– mettre au point la description de l'algorithme systolique ou encore la "*systolisation*",

2– valider l'algorithme ainsi défini.

Or, il n'est pas rare que le nombre de processeurs requis par une telle architecture puisse atteindre des dizaines de milliers pour délivrer des puissances de calcul pouvant atteindre le milliard d'opérations à la seconde. D'où des recours à des techniques de simulation parallèle. En outre, de telles architectures se fondent sur des critères de régularité. Ceci limite, a priori, leur domaine d'application. Bien que la définition d'une telle architecture puisse apparaître comme très contraignante, peu de problèmes ont résisté à la systolisation [Kun82]. Sans être exhaustif, citons:

- ⋆ en analyse numérique, le produit de matrices, le produit matrice vecteur, la triangularisation de matrice ou encore, l'inversion de matrice,
- ⋆ en traitement d'image citons la convolution ou la transformée de Fourier discrète,
- ⋆ dans le domaine des problèmes non-numériques citons la programmation dynamique ou la recherche de composantes connexes.

Une telle énumération ne doit cependant pas masquer les problèmes difficiles que pose la conception systématique d'applications pour de telles architectures [AQ85] bien que la littérature propose diverses méthodologies. Dans la section 3.1.2, nous abordons les principes fondamentaux de telles méthodologies.

3.1.2 Les méthodologies de conception systématiques d'architectures systoliques.

Les travaux de *Moldovan* [Mol83], *Miranker* [MW84], *Moreno* [ML88] et *Quinton* [Qui83] constituent un cadre général pour les méthodologies existantes. Depuis ces travaux, la synthèse des réseaux systoliques a beaucoup évolué. Avant de discuter de ces évolutions, nous introduisons tout d'abord la synthèse des réseaux systoliques telle qu'elle a été proposée par les auteurs des travaux précédemment cités. Elle peut se résumer en trois étapes:

Etape 1– réécriture d'un schéma d'exécution séquentiel d'une application donnée sous la forme d'un système d'équations de récurrence uniformes,

Etape 2– définition d'une fonction temporelle spécifiant l'ordonnancement des calculs de façon à exhiber le parallélisme potentiel de l'application,

Etape 3– allocation des calculs aux processeurs de sorte que le réseau d'interconnexion résultant soit planaire et ait ses processeurs localement connectés.

La méthode de synthèse la plus utilisée pour ce faire est celle dite de *projection*. L'idée générale de cette méthode est de profiter de la cyclicité (donnée par le caractère récurrent des équations) et de l'uniformité des dépendances pour obtenir un réseau formé de cellules modulaires (la structure de chaque cellule ne dépend pas de la taille du problème) et dans lequel la circulation des données est synchrone. Pour cela, les calculs sont répartis sur le réseau par une fonction d'allocation linéaire et ordonnancés par des fonctions de temps linéaires.

La mise en oeuvre de ces trois étapes de la méthode de projection a aujourd'hui beaucoup évoluée. Au départ, les équations de récurrence considérées étaient uniformes. Le problème de l'ordonnancement des équations de récurrence uniformes est souvent résolu par des techniques de programmation linéaire et ses solutions appartiennent à un polyèdre. Les méthodes de résolution de ce problème reposent sur le fait que les dépendances entre les calculs peuvent

être représentées par un nombre fini de vecteurs. Plusieurs recherches ont montré par la suite que les équations de récurrence affines s'adaptent bien à la synthèse de tels réseaux [MCP94, QR89, RPF86, QD89]. Le formalisme des équations de récurrence affines et ses extensions donnent lieu à ce qui est communément appelé le *modèle polyédrique*, base du langage *Alpha* [Mau89]. Dans ce modèle, les problèmes d'ordonnancement et d'allocation sont plus difficiles. Néanmoins, dans le cas où les ensembles des indices définissant les équations récurrentes sont des polyèdres, il est possible de bénéficier d'une représentation de ces polyèdres pour formuler les contraintes d'ordonnancement et d'allocation en termes de programmation linéaire.

Quelles que soient les fonctions de temps et d'allocation choisies, l'objet d'une méthode de projection reste la production automatique, ou tout au moins semi-automatique, d'un réseau systolique calculant le problème exprimé sous forme de système d'équations de récurrence. Pour ce faire, la fonction d'allocation est choisie comme une *projection* linéaire le long d'un vecteur dit *vecteur de projection.* En général, le vecteur de projection est choisi en fonction de deux critères: la taille du réseau généré et la position des entrées-sorties qu'on aimerait voir projetées sur les bords du réseau. De nombreuses études [Dar93, MF86], ont montré que les réseaux systoliques ainsi générés par projection ont des caractéristiques, non nécessairement souhaitées, dues à la combinaison des fonctions de temps affines dont les paramètres sont des vecteurs de temps, notés τ, et des fonctions d'allocation dont les paramètres sont des vecteurs de projection, notés s. D'un coté, la taille du réseau dépend en général du graphe de dépendance et du volume de calcul à traiter. Par exemple, pour le produit matriciel, le graphe de dépendance est un cube de taille $n \times n \times n$ et les réseaux systoliques comportent un nombre de cellules de l'ordre de n^2. De l'autre coté, l'activité d'un réseau systolique dépend du choix des paramètres de la méthode, τ et s à travers leur produit scalaire. Par conséquent, si c est le produit scalaire entre les vecteurs τ et s, on montre que les cellules d'un réseau systolique sont généralement actives de façon cyclique tous les c unités de temps [Dar93].

Ainsi, si on souhaite répartir de vastes domaines de calcul sur des réseaux systoliques de taille moindre, une phase de partitionnement est donc nécessaire. Le partitionnement peut prendre deux formes:

1– la forme *Localement Parallèle Globalement Séquentiel (LPGS)* (voir [MF86]),

2– la forme *Localement Séquentiel Globalement Parallèle (LSGP)* (voir [Dar93, CMPia, BDD90]).

Dans une approche systolique du type *LPGS*, on partitionne un graphe de dépendance en blocs de taille fixe et on traite les blocs les uns après les autres. Dans un bloc, la structure induite par une projection est un réseau systolique de petite taille. Les résultats produits par un bloc sont stockés dans des mémoires externes et sont réutilisés en entrée, par des blocs qui le nécessitent. Cette approche exige des mémoires locales de petite taille mais une mémoire externe de grande taille.

Dans l'approche *LSGP*, on est amené à partitionner en blocs de cellules d'activité disjointe, un réseau systolique obtenu par une méthode de projection. Un tel partitionnement semble justifié par l'intuition suivante: puisque le domaine des calculs est dense, à un instant donné, une seule cellule sur c doit être active, où $c = \tau.s$ unités de temps. On peut donc fusionner c cellules dans une seule cellule. Le principe du partitionnement est alors d'augmenter c, en modifiant τ et s, puis de compresser le réseau d'un facteur de c. A priori, n'importe quel ensemble de blocs de cellules devrait convenir pour cette approche. Chaque bloc est alloué

à un processeur physique qui n'a plus qu'a en simuler le fonctionnement. La difficulté réside dans le choix du regroupement des cellules. En effet, les points de calcul projetés dans un même bloc doivent être ordonnancés à des instants différents de telle sorte qu'ils puissent être calculés sans conflit par le processeur physique. De plus, chaque processeur a besoin d'une large mémoire interne.

De nombreux auteurs ont abordé le problème du partitionnement *LSGP* [BDD90, MCP94, Dar93]. Il existe des approches purement vectorielles qui consistent à sélectionner k vecteurs $v_1, \cdots, v_k$ de telle sorte que les points de $u+v_1, \cdots, u+v_k$ soient alloués au même processeur. Cependant, ces points trouvés par énumération, donnent lieu à des partitionnements dont la mise en oeuvre est prohibitive.

Il existe d'autres approches consistant à regrouper les sommets d'un graphe systolique par parallélépipèdes. Classiquement, les parallélépipèdes sont des copies translatées d'un parallélépipède canonique et les pavages ainsi définis occasionnent des pertes de parallélisme. Notons cependant que, ces approches qui utilisent les techniques de pavage peuvent facilement être intégrées à un synthétiseur de réseaux systolique ou à la génération automatique de codes *SPMD*. Ces dernières classes d'approches *LSGP* sont celles qui se rapprochent le plus de l'approche de parallélisation que nous proposons. Un exposé des principes généraux de notre approche de parallélisation permettra de mettre en évidence son originalité par rapport aux approches de pavage du type *LSGP*.

3.2 Principes de l'approche de parallélisation

Pour répondre aux besoins en termes de puissance de calcul et de capacité de mémorisation que demandent certaines méthodes numériques, il existe aujourd'hui, une gamme de machines parallèles construites autour de composants standards tels que les processeurs et réseaux d'interconnexion du commerce. Les environnements logiciels installés sur ces machines favorisent le recours au parallélisme virtuel, gage de portabilité. Nous renvoyons le lecteur intéressé à l'annexe A pour une présentation de quelques unes de ces machines parallèles. Dans cette section, nous présentons les grandes lignes d'une nouvelle approche systolique de parallélisation de méthode numériques pour cette gamme de machines. Cette approche fusionne trois aspects: l'analyse quantitative de la systolisation, le partitionnement par blocs et l'allocation par blocs. Ces aspects seront détaillés et analysés dans les sections suivantes.

3.2.1 Analyse quantitative de la systolisation

Il s'agit de réutiliser les techniques existantes pour la synthèse des réseaux systoliques à fronts d'ondes, pour évaluer quantitativement, le parallélisme d'une méthode numérique. Ces techniques ont fait l'objet de nombreuses études dans le cas où les méthodes numériques sont à matrices denses. Dans le cas où les matrices sont creuses, ces techniques restent formellement valides. Notons cependant dans ce cas que beaucoup de calculs sont nuls, par construction du stockage creux, et ils ne contribuent pas à la production des résultats. Pour identifier l'ensemble des calculs dont les contributions sont nécessaires et suffisantes pour l'exécution d'une telle méthode numérique, il suffit par exemple, d'affecter le signal 0 à toute équation récurrente ayant une contribution nulle, et le signal 1 à toute équation récurrente ayant une contribution non nulle. Par élimination de tous les points ayant un signal égal à 0, on parvient ainsi à des équations de récurrence qui se caractérisent par des dépendances dynamiques. Cette spécificité des méthodes numériques influence leurs complexités en temps

et en mémoire, et soulève le problème de la parallélisation de ce type de récurrences.

On dispose aujourd'hui d'une méthodologie de parallélisation bien établie, pour la synthèse des réseaux systoliques à partir d'équations récurrentes affines (voir section 3.1.2). Bien que, pour des raisons d'ordre industrielle, les méthodes numériques à matrices creuses aient été d'un grand intérêt, il semble à notre connaissance que jusqu'à présent, aucun chercheur n'ait abordé concrètement les équations de récurrence pour les méthodes numériques à matrices creuses. Ce sujet est étroitement lié à un aspect toujours ouvert d'une portée plus générale: peut-on développer les techniques existantes de synthèse des réseaux systoliques à fronts d'ondes, pour les étendre aux dépendances dynamiques?

Cette nouvelle approche de parallélisation tente d'y répondre positivement à travers l'analyse et la synthèse des dépendances dynamiques, lorsque le nombre de directions de dépendance est fini et pour des organisations irrégulières de données telles que celles des matrices creuses. Le parallélisme potentiel ainsi défini peut être exploité pour une mise en oeuvre: sur des architectures parallèles du type *MIMD* à mémoire distribuée utilisant des bibliothèques à passage de message, et sur des architectures spécialisées utilisant la logique reconfigurable. Dans le cadre de cette thèse, nous discutons de la construction de code du style *SPMD* pour des machines *MIMD* à mémoire distribuée. Pour les résultats concernant les architectures reconfigurables, nous renvoyons le lecteur à l'article [BS99].

La construction d'un code *SPMD* pose le problème de la simulation des gros domaines de calculs générés par les systèmes d'équations récurrentes, sur des machines d'exécution dont le nombre de processeurs est borné et les coûts des communications élevé. Ce problème est résolu par agrégation des calculs dans des super-noeuds. On obtient ainsi, des algorithmes parallèles dont l'optimalité doit être démontrée. La parallélisation optimale étant difficile, on procède par des méthodes heuristiques de partitionnement et d'allocation.

3.2.2 Partitionnement par blocs

Il s'agit de définir une nouvelle structure de graphe par agrégation des équations récurrentes fournies par une analyse quantitative de la systolisation. Tout sommet de ce nouveau graphe est un super-noeud qui contient un sous-graphe définissant les dépendances entre les équations récurrentes ainsi agrégées. Un arc dans cette nouvelle structure s'obtient par réduction en une seule dépendance, de toutes les dépendances entre deux super-noeuds. Les super-noeuds sont exécutés de façon atomique.

Pour obtenir cette nouvelle structure de graphe, on considère une formulation d'un problème en terme de systèmes d'équations récurrentes uniformes *(SERU)*. L'intuition de l'agrégation par blocs réside sur les techniques de quadrillage utilisées dans les méthodes de discrétisation du type *différences finies* ou *éléments finis* [RT83, LT86b]. Ces quadrillages sont utilisés pour le maillage de domaines continus. En utilisant ces techniques dans le cadre du domaine des indices d'un *SERU*, on parvient à un partitionnement dans lequel, les super-noeuds sont les mailles. Dans ce nouveau graphe, certains super-noeuds peuvent être à contribution nulle (les super-noeuds qui ne contiennent pas de sous-graphe). Ces super-noeuds sont identifiables, en utilisant l'affectation des signaux 0 et 1 précédemment mentionnée dans la section 3.2.1.

Afin d'automatiser ce partitionnement, indépendamment de la taille d'un *SERU* donné, nous recherchons de nouvelles structures de graphes dont la structure est "similaire" à celles des graphes du *SERU* à partitionner. Il vient ainsi que la nouvelle structure de graphe, obtenue par partitionnement, se modélise formellement par un *SERU*, et hérite des propriétés

de modularité propres aux *SERU*. En outre, la notion de similitude permet, par extension des résultats de l'analyse de complexité d'un schéma d'exécution systolique, de déduire un modèle mathématique évaluant a priori, le temps d'exécution du nouveau graphe. Ce modèle mathématique dépend explicitement de la taille et du nombre de super-noeuds qui s'exécutent de façon obligatoirement séquentielle. La taille de ces super-noeuds peut se paramétrer en fonction du stockage des données en entrée, et du nombre de processeurs de la machine d'exécution.

3.2.3 Allocation par blocs

Pour une méthode numérique, il s'agit d'allouer ses données en entrée aux processeurs d'une machine d'exécution, et de dire quels sont les processeurs détenteurs de résultats finaux. Pour y parvenir, notre idée est la suivante. On considère un graphe modélisant les dépendances d'un *SERU* et ayant ses sommets tous identiques. Deux étapes permettent alors de définir une telle allocation.

La première étape a pour but de définir, par projection, une structure de donnée permettant de modéliser le fait que des sommets du graphe modifient successivement, une variable suivant une direction de dépendance. Pour ce faire, on résout un problème d'ordonnancement optimal des *SERU* et on marque chaque sommet par sa date d'activation. Le choix d'une direction de projection permet de créer, après projection, un couplage entre une donnée en entrée et les sommets qui la modifient. Ce couplage se modélise dans le cadre d'un objet abstrait, que nous nommons cellule, caractérisé par: la donnée en entrée, les algorithmes des sommets, son intervalle d'activation (borne inférieure et supérieure des dates d'activation des sommets) et son voisinage (la liste des prédécesseurs et des successeurs des sommets). Puisque tous les sommets associés à une cellule sont obligatoirement exécutés de façon séquentielle, une cellule peut être allouée à un seul processeur. La deuxième étape consiste alors à allouer les cellules aux processeurs de sorte que sur un même processeur, les intervalles d'activation ne s'intersectent pas.

3.3 Modèles mathématiques de l'approche

Les modèles mathématiques utilisés dans le cadre de cette approche de parallélisation concernent l'application parallèle et la machine parallèle d'exécution.

3.3.1 Modèle d'une application parallèle

Une application parallèle est bâtie selon le modèle *client/serveur* [GH94]. Il s'agit de deux entités qui coopèrent l'une avec l'autre à l'exécution d'une méthode numérique.

La première entité, le module client, sert d'interface utilisateur et exécute la logique des traitements. L'interface utilisateur permet de paramétrer et de résoudre le partitionnement et l'allocation des données. Les algorithmes de partitionnement et d'allocation dépendent de l'approche de parallélisation et peuvent être exécutés de façon centralisée ou de façon décentralisée. La logique des traitements se définit par le schéma d'exécution d'une méthode numérique, et enchaîne plusieurs algorithmes dont certains sont parallèles. L'exécution d'un algorithme parallèle se traduit par la formulation de deux requêtes successives: fournir des données en entrée au serveur et récupérer des résultats calculés par le serveur.

La deuxième entité, le module serveur, gère l'exécution des algorithmes parallèles d'une méthode numérique. Chaque algorithme parallèle est modélisé par un graphe, et dépend d'un

partitionnement et d'une allocation de données. Le serveur est donc un ensemble de graphes d'exécution d'algorithmes parallèles dont nous exposons la structure générale et le modèle d'évaluation des performances.

Graphe d'exécution d'un algorithme parallèle

C'est un graphe qui s'obtient par une décomposition par les calculs mentionnés précédemment dans la section 2.2.2.3. Pour ce faire, on considère d'abord une modélisation mathématique d'un algorithme parallèle sous la forme d'un graphe $G = (D, E)$ dans lequel, chaque sommet représente un calcul à grains fins. L'agrégation des sommets du graphe G permet de définir un graphe *quotient.*

Définition 5 (Graphe quotient)
Soit Π, *une partition de* D. *Le graphe quotient de* G *ou encore le graphe réduit associé à la partition* Π, *noté* G_Π *est un graphe dont:*

- ⋆ *l'ensemble des sommets, noté* D_Π, *est l'ensemble des classes d'équivalence ou encore des points de* Π;
- ⋆ *l'ensemble des arcs, noté* E_Π, *est l'ensemble des couples de point* (π, π'), *avec* π *et* π' *des points de* Π, *pour lesquels existent* $u \in \pi$ *et* $v \in \pi'$ *tels que* $(u, v) \in E$.

En réduisant en un point, tout sommet du graphe G_Π et en un arc, tous les arcs entre deux sommets ayant la même orientation, on obtient une nouvelle structure de graphe. Dans la suite de ce chapitre, ce nouveau graphe sera noté $G' = (D', E')$ avec: D' l'ensemble des sommets et E' l'ensemble des arcs de G'.

Pour des problèmes de synthèse systématique du graphe G', nous nous plaçons dans le cadre de l'hypothèse de modularité des réseaux syslotiques telle que décrite dans [Kun82]. Nous supposons que les sommets et les arcs des graphe G et G' ne dépendent pas de la taille du problème traité en exécutant un algorithme parallèle. Sous cette hypothèse, les graphes peuvent être vu comme un assemblage de cellules élémentaires, où une cellule élémentaire désigne un sommet et tous ses arcs entrant et sortant. En vertu de l'hypothèse de modularité précédente, l'ensemble des sommets (respectivement des arcs) d'un graphe peut être partitionné en classe d'équivalence de sommets (respectivement d'arcs). Par conséquent, l'ensemble des cellules élémentaires peut être partitionné en classe d'équivalence. On doit donc théoriquement pouvoir exécuter un problème de grande taille en rajoutant, aux bords d'un graphe validant la propriété de modularité, des cellules appartenant à des classes d'équivalence déjà définies.

Définition 6 (Graphes similaires)
Etant donnés deux graphes G *et* G' *vérifiant la propriété de modularité précédemment citée, notons:* $\mathcal{C}$ *(respectivement* $\mathcal{C}'$*), l'ensemble des classes d'équivalence des cellules élémentaires du graphe* G *(respectivement* G'*) résultant de cette modularité.*
Les graphes G *et* G' *sont similaires si* $\mathcal{C}$ *et* $\mathcal{C}'$ *sont identiques.*

Analyse des performances

Le modèle d'analyse des performances dépend du volume des calculs par sommet de G' et

des communications entre les sommets de G'. Cette analyse des performances s'inscrit dans le cadre de la recherche des sommets et des arcs de G' qui fournissent une borne inférieure sur la séquentialité. Pour ce faire, nous allons introduire le concept de *poids des sommets d'un graphe.*

Soit u un sommet de G'. On appelle *prédécesseur* de u, tout sommet v de G' tel que $(v,u) \in E'$. Soit $Pred(u)$, l'ensemble des prédécesseurs de u. On appelle *successeur* de u tout sommet v de G' tel que $(u,v) \in E'$. Soit $Succ(u)$, l'ensemble des successeurs de u. Pour tout noeud u de G', on lui affecte la quantité $T_{cal}(u)$. Pour tout arc (u,v) de G', on lui affecte la quantité $T_{com}(u,v)$. Appelons poids de v, soit $Poids(v)$:

$$Poids(v) = T_{cal}(v) + \max_{u \in Pred(v)} \{Poids(u) + \Phi(q, T_{com}(u,v))\} \tag{3.1}$$

Notons que, pour tout noeud v tel que $Pred(v) = \emptyset$,

$$Poids(v) = T_{cal}(v) \tag{3.2}$$

Dans la mesure où $D' \subset \mathbb{Z}^q$, la fonction Φ modélise les relations entre les prédécesseurs de v situés dans les q dimensions de l'espace.

Le poids de G' est alors égal au plus fort poids des sommets sans successeurs:

$$Poids(G') = \max_{\{u | Succ(v) = \emptyset\}} \{Poids(v)\} \tag{3.3}$$

Un tel poids est difficile, voire impossible à formuler de façon analytique. Aussi, tentons nous d'en donner une borne supérieure la plus fine possible. Pour cela, nous considérons le sous graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ qui contient des sommets à plus fort poids et construit par l'algorithme suivant:

1: $\mathcal{E} \leftarrow \emptyset$

2: $\mathcal{V} \leftarrow \{v | Poids(v) = Poids(G')\}$

3: Répéter

3.1: $Pred(\mathcal{V}) \leftarrow \{u \in D - \mathcal{V} | Poids(u) = Poids(G' - \mathcal{G})\}$

3.2: $\mathcal{V} \leftarrow \mathcal{V} \cup Pred(\mathcal{V})\}$

3.3: $\mathcal{E} \leftarrow \mathcal{E} \cup \{(u,v) \in E' | u, v \in \mathcal{V}\}$

3.4: Jusqu'à $Pred(\mathcal{V}) = \emptyset$

On appelle *chemin* entre deux sommets u et v: une succession des sommets distincts $[u = x_0, x_1, \cdots, x_{n-1}, x_n = v]$ telle que $(x_{n-1}, x_n) \in E$ pour $i = 1, \cdots, n$; n représente la longueur du chemin. Soit d, la longueur d'un plus long chemin de $\mathcal{G}$. Par construction, on peut vérifier que:

$$Poids(G') \leq (d+1) \max_{u \in \mathcal{V}} T_{cal}(u) + d \max_{(u,v) \in \mathcal{E}} \Phi(q, T_{com}(u,v)) \tag{3.4}$$

Sous des hypothèses présentées dans la section 3.4.3, nous montrons que le concept de poids tel que précédemment défini permet d'établir des contraintes d'ordonnancement d'un graphe quotient. En particulier, on peut vérifier que le graphe $\mathcal{G}$ contient la liste des sommets à

plus fort poids qui sont obligatoirement exécutés de façon séquentielle. Ces sommets à plus fort poids ne sont pas nécessairement sur un chemin. Par abus de langage, nous parlerons de "chemin à plus fort poids" pour désigner l'ensemble de ces sommets.

Définition 7 (Chemin à plus fort poids)
On appelle chemin à plus fort poids *entre deux sommets u et v, une succession des sommets distincts* $[u = x_0, x_1, \cdots, x_{n-1}, x_n = v]$ *telle que:*

$$Poids(x_{i+1}) = Poids(x_i) + \Phi(q, \acute{T}_{com}(x_i, x_{i+1})) + T_{cal}(x_{i+1}) \quad 0 \leq i \leq n-1 \tag{3.5}$$

3.3.2 Modèle de la machine parallèle d'exécution

Le modèle de machine auquel nous nous référons est une machine *MIMD* à mémoire distribuée. C'est un super-calculateur composé de P processeurs interconnectés par un réseau de communication logique. Chaque processeur, $\mathcal{P}_i$ avec $1 \leq i \leq P$ est identifié par son index i et possède une mémoire locale. Le temps mis par chaque processeur pour exécuter de façon atomique, n calculs scalaires, se modélise par:

$$T_{cal}(n) = \tau^*(n) + n * t_{scal} \tag{3.6}$$

où: t_{scal} est le temps de traitement des opérations scalaires qui est exprimé en unité de temps par opération scalaire; $\tau^*(n)$ est le temps de "start up" nécessaire à l'accès des structures de données pour le traitement de n calculs scalaires.

Les données sont échangées entre processeurs par l'intermédiaire de mécanismes de routage. On permet ainsi une approche de type parallélisme virtuel dans un tel super-calculateur. Les communications se font via des primitives telles:

- ⋆ les communications quelconques d'un processeur à un autre,
- ⋆ les diffusions d'un message d'un processeur à tous les autres,
- ⋆ les échanges des messages de tous les processeurs à tous les autres,
- ⋆ etc.

Ces fonctions sont décrites et étudiées dans [Rum94, CT93]. Pour nos besoins, seules les primitives telles que les envois ou les réceptions de données point à point sont indispensables pour simuler les graphes d'exécution. En effet, ceci est lié aux modèles de graphe qui sous-tendent nos analyses et qui sont par construction similaires aux graphes des algorithmes systoliques. Notons cependant qu'un super-noeud peut avoir besoin d'envoyer simultanément des données à plusieurs des super-noeuds voisins. Dans ce cas, la primitive d'envoi des donnée est appelée répétitivement. Il en est de même des données qui sont reçues et qui peuvent provenir simultanément de plusieurs super-noeuds voisins. Pour certaines machines parallèles telles que les systèmes *SPx* d'*IBM* ou *PARAGON* d'*INTEL*, ces données peuvent être acheminées selon des canaux distincts. Pour d'autres, tels les réseaux de stations de travail, les données passent par le même canal.

Notons respectivement *envoyer* et *recevoir*, les primitives d'envoi et de réception point à point qui sont ainsi utilisées.

La procédure envoyer:

elle permet à un processeur i d'envoyer une donnée comportant n scalaires à un processeur quelconque. Pour ce faire, la connaissance de son indice est nécessaire et suffisante. Nous admettons que tout processeur émetteur de données peut enchaîner sur une phase de calcul avant que la totalité des données émises ne soient disponibles sur le processeur récepteur. Il s'agit donc d'une opération dite "*non-blocante*" [DW95].

La procédure recevoir:

elle permet à un processeur, d'indice j, de recevoir un paquet de n scalaires émis par un processeur i. Pour ce faire, seule la connaissance de l'indice i du processeur émetteur est nécessaire. Nous admettons que, le processeur j, ne peut entamer une phase de calcul que si toutes les n données scalaires émises par i ont été reçues par j. Il s'agit d'une opération de type "*localement blocante*" [DW95].

Nous admettons qu'à chaque appel de la procédure *recevoir* par un processeur, il existe un et un seul processeur ayant fait un envoi correspondant. Une modélisation simple de la communication point à point est donnée par:

$$T_{com}(n, \tau, \theta) = \tau + n\theta \tag{3.7}$$

- $\star$ τ est le temps d'initialisation ou "start-up" exprimé en seconde,
- $\star$ θ est le taux de transmission exprimé en seconde par nombre de données scalaires transférées. Il représente l'inverse de la largeur de bande passante du réseau logique.

3.4 Méthodologie de la parallélisation

On considère une méthode numérique dont les matrices peuvent être creuses ou denses. Les étapes successives de l'approche sont les suivantes lorsque les matrices sont denses:

Etape 1 : réécriture d'un schéma d'exécution séquentiel d'une application donnée sous la forme d'un schéma d'exécution systolique,

Etape 2 : partitionnement par bloc d'un graphe d'exécution systolique en un graphe qui lui est "similaire",

Etape 3 : ordonnancement en temps minimum du graphe obtenu à l'étape 2,

Etape 4 : allocation des sommets du graphe obtenu à l'étape 2 en préservant "au mieux", l'ordonnancement en temps minimum de l'étape 3.

Afin de traiter le cas des méthodes numériques à matrices creuses, nous proposons une relaxation du schéma d'exécution systolique. Etant donné un *SERU*, il s'agit d'éliminer les équations récurrentes à contribution nulle tout en préservant les directions de dépendance du *SERU* (Nous parlerons alors de *systèmes d'équations récurrentes uniformes relaxées (SERUR)*). Par construction, l'ordonnancement des *SERUR* est obtenu grâce aux fonctions de temps *au plus tôt* ou *au plus tard*. Ainsi, étant donné une telle relaxation du schéma systolique, nous verrons que les étapes (2) à (4) procèdent de la même façon.

3.4.1 Analyse quantitative du schéma d'exécution systolique de référence

Un schéma d'exécution systolique s'obtient à partir des méthodologies de la section 3.1.2. Pour les besoins de notre approche, nous nous plaçons dans le cadre d'une synthèse de réseaux systoliques à partir d'un *SERU*. Pour construire le schéma d'exécution systoliques de référence, on résout un problème d'ordonnancement des *SERU* ainsi considérés. Notre propos ne consiste pas à résoudre un problème de synthèse de réseaux systolique, il s'agit d'analyser les calculs qui peuvent être soit liés par une relation de dépendance et dans ce cas, leur exécution est séquentielle, soit non interférantes et leur exécution peut être effectuée en parallèle. Dans cette étude, nous manipulons un parallélisme à grains fins pour détecter le parallélisme potentiel d'une application. Cette étude servira de fil conducteur pour l'ordonnancement du nouveau schéma d'exécution (voir l'approche que nous proposons dans la section 3.4.3).

3.4.1.1 Equations de récurrence uniformes

Le développement des *SERU* commence vers la fin des années 60 avec les travaux de *Karp*, *Miller* et *Winograd* [KMW67] qui proposent l'expression d'algorithmes itératifs sous forme de systèmes d'équations récurrentes uniformes. Ils étudièrent l'organisation des calculs dans des structures régulières. Ils montrèrent en particulier la présence d'un parallélisme intrinsèque important au sein de ces équations sous certaines conditions et l'intérêt du caractère cyclique des calculs. Les *SERU* ont déjà été appliqués à de nombreux algorithmes du calcul scientifique. Il s'agit d'algorithmes dont les calculs reposent sur des matrices denses. Nous ne discutons pas ici, des aspects relatifs à la calculabilité *SERU* à partir d'un problème. Nous renvoyons le lecteur intéressé au rapport [QD89] et à sa bibliographie.

Etant donné un problème à résoudre, on cherche une solution séquentielle exprimée sous la forme suivante:

$$Y_i(z) = f_i(Y_1(z - v_1), Y_2(z - v_2), \cdots, Y_p(z - v_p)) \tag{3.8}$$

pour $i = 1, \cdots, p$.

- ⋆ z parcours un domaine $D \subset \mathbb{Z}^q$ et représente un ensemble d'indices,
- ⋆ les f_i sont des fonctions à p variables décrivant les transformations élémentaires ayant lieu en z,
- ⋆ les v_i sont des éléments de $\mathbb{Z}^q$ encore appelés vecteurs de dépendance,
- ⋆ chaque Y_i est supposé connu lorsque $z \notin D$.

L'équation (3.8) exprime la relation de récurrence entre les Y_i. Dans la mesure où les indices des Y_i sont obtenus par simple translation de z, on parle alors d'équations de récurrence *uniformes*. Les dépendances entre les équations (3.8) peuvent être représentées par un graphe orienté $G = (D, E)$ dont l'ensemble des sommets est constitué par le domaine D et dont l'ensemble des arcs, soit E, est formé par les couples $(z - v_i, z)$. Tout terme $Y_i(z - v_i)$ situé dans la partie droite de (3.8) est utilisé comme donnée en z, tandis que $Y_i(z)$ est le résultat d'un calcul effectué en z. Il y a donc un arc (z', z) dans G si un résultat produit en z' est utilisé comme donnée en z.

Il existe au moins deux façons de représenter le graphe G. La première représentation est celle dite du *graphe réduit*. Il s'agit d'un graphe cyclique qui est défini dans [Dar93] par:

Définition 8 (Graphe réduit)
Le graphe réduit associé au système d'équations

$$Y_i(z) = f_i(Y_1(z - v_1), Y_2(z - v_2), \cdots, Y_p(z - v_p)) \; 1 \leq i \leq p$$

où Y_i est une variable définie sur les points entiers d'un polyèdre D_i, est le graphe orienté possédant:

- ⋆ *p sommets s_i valués par les polyèdre D_i,*
- ⋆ *une arête est orientée de s_i vers s_j si $Y_j(z) = f_j(\cdots, Y_j(z - v_i), \cdots)$ et valuée par v_i.*

Le graphe réduit est très utilisé en parallélisation automatique. Sa génération automatique se fait au prix d'une analyse de dépendance dont les limites de validité ont été exposées dans la section 2.2.1.

La deuxième représentation est une construction explicite de G par assemblage de cellules élémentaires (une cellule élémentaire est un sommet et tous ses arcs entrant et sortant). Pour pour y parvenir, on se sert de la propriété de modularité dans un modèle systolique [Kun82]. En vertu de cette propriété les sommets et les arcs peuvent peuvent être caractérisés par leurs classes d'équivalence. Il suffit donc de disposer de ces classes d'équivalence pour simuler la structure d'un graphe d'exécution systolique, indépendamment de la taille du problème qui est traité. De plus, l'intérêt que nous portons à cette deuxième représentation se justifie aussi par les stratégies de partitionnement et d'allocation des calculs qui induisent une logique de partitionnement et l'allocation des matrices et des vecteurs d'une méthode numérique.

Pour illustrer la modélisation sous la forme d'équations de récurrence uniformes, on considère le produit d'une matrice symétrique par un vecteur. La résolution numérique de la plupart des problèmes scientifiques conduisent assez souvent à la résolution d'un ou plusieurs systèmes d'équations linéaires $Ax = b$ avec $A = (a_{i,j})_{1 \leq i,j \leq n}$, $b = (b_i)_{1 \leq i \leq n}$ et $x = (x_i)_{1 \leq i \leq n}$. Le produit matrice vecteur est une opération de base dans les méthodes itératives telles que celles de *Krylov* [LT86b]. Dans les illustrations qui suivent, seule la partie triangulaire supérieure de la matrice est utilisée. Nous renvoyons le lecteur intéressé au chapitre 4 pour une étude plus détaillée.

Exemple 1 (Le produit matrice symétrique vecteur)
Compte tenu de la symétrie de la matrice A, on défini le système d'équation de récurrence:

$$\begin{cases} y_i^{(0)} = 0 & 1 \leq i \leq n \\ y_i^{(i)} = y_i^{(i-1)} - a_{i,i}x_i & 1 \leq i \leq n \\ y_i^{(j)} = y_i^{(j-1)} - a_{i,j}x_j & 1 \leq i \leq n \text{ et } j \in [i+1, n] \\ y_j^{(i)} = y_j^{(i-1)} - a_{i,j}x_i & 1 \leq i \leq n \text{ et } j \in [i+1, n] \end{cases} \tag{3.9}$$

Il s'agit d'un système d'équations de récurrence affines. On parvient à un système d'équations de récurrence uniformes en utilisant les techniques décrites dans [QD89]. Le domaine des calculs, soit $D \subset \mathbb{Z}^2$, se définit comme suit:

$$D = \bigcup_{i=1}^{n} \{(i, j) \; : \; i \leq j \leq n\} \tag{3.10}$$

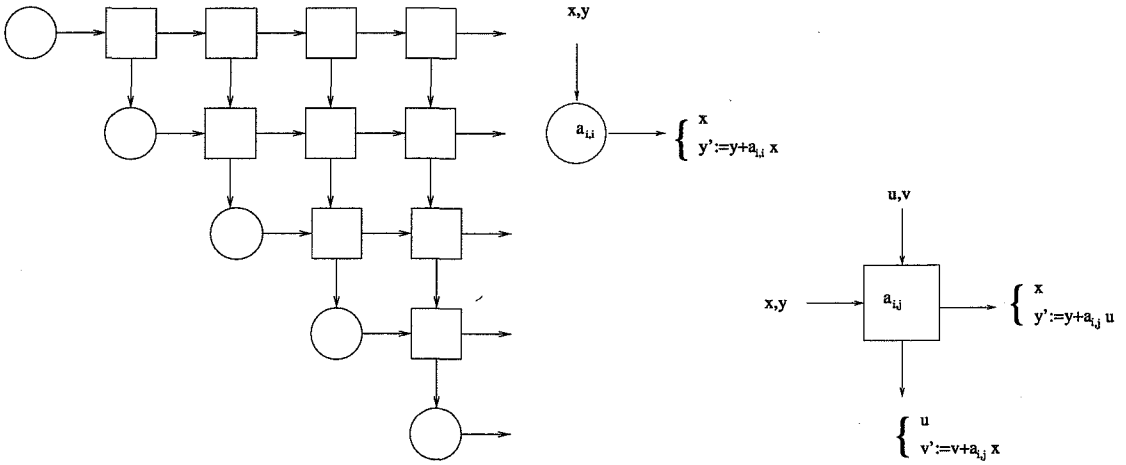

FIG. 3.3 – Graphe de dépendance et cellules élémentaires associés au produit matrice vecteur pour $n = 5$.

La figure (3.3), illustre le graphe de dépendance G pour $n = 5$. Dans ce graphe notons qu'il existe deux classes de cellules élémentaires:

⋆

les cellules rondes représentent les calculs effectués aux points (i, i). Une telle cellule reçoit en entrée des données de type $(x_i^{(i-1)}, y_i^{(i-1)})$ de la cellule $(i-1, i)$. Après le calcul de $x_i^{(i)} = x_i^{(i-1)}$ et de $y_i^{(i)} = y_i^{(i-1)} + a_{i,i}x_i^{(i)}$, elle envoie $(x_i^{(i)}, y_i^{(i)})$ à la cellule $(i, i+1)$

⋆

les cellules carrées représentent les calculs effectués aux points (i, j) avec $i < j$. Une telle cellule reçoit en entrée des données de type $(x_j^{(i-1)}, y_j^{(i-1)})$ de la cellule $(i-1, j)$ et $(x_i^{(j-1)}, y_i^{(j-1)})$ de la cellule $(i, j-1)$. Après le calcul de $x_i^{(j)} = x_i^{(j-1)}$, $x_j^{(i)} = x_j^{(i-1)}$, $y_i^{(j)} = y_i^{(j-1)} + a_{i,j}x_j^{(i)}$ et $y_j^{(i)} = y_j^{(i-1)} + a_{i,j}x_i^{(j)}$, elle envoie $(x_i^{(j)}, y_i^{(j)})$ à la cellule $(i, j+1)$ et $(x_j^{(i)}, y_j^{(i)})$ à la cellule $(i+1, j)$.

3.4.1.2 Complexité en temps

Etant donnée une modélisation d'une application en terme de *SERU* et le graphe de dépendance $G = (D, E)$ qui en découle, on cherche à résoudre le problème de l'ordonnancement optimal de ce *SERU*. Dans les méthodes de synthèse de réseaux systoliques, ce problème admet une solution. Pour ce faire, on suppose qu'en chaque point de D, les calculs ont la même durée. Dans la pratique, ceci n'est pas vrai car les calculs ne sont pas les mêmes suivant les sommets du graphe. Cependant, il est possible de supposer une exécution synchrone des calculs en prenant comme unité de temps le maximum des durées associées aux différents points de D. Une fonction de temps T, est une application définie par:

$$\begin{array}{rcl} T & : & D \rightarrow I\!N \\ & & u \longmapsto T(u) \\ & & \text{vérifiant } T(u) < T(v) \text{ pour tout arc } (u, v) \end{array} \qquad (3.11)$$

$T(u)$ représente la date à laquelle sera exécuté le calcul associé au point u. La contrainte $T(u) < T(v)$ pour tout arc (u, v) est naturelle car certaines données nécessaires aux calculs en v sont produites par des opérations effectués en u. Il est donc clair que les points de D situés sur un chemin de G seront traités séquentiellement. Comme tous les points de D ont un temps d'exécution unitaire, le temps minimum d'exécution du schéma de récurrence est égal au nombre de sommets du plus long chemin de G. Une fonction de temps est donc "optimale" si elle conduit à un temps d'exécution égal à cette borne. Il existe plusieurs façons de construire de telles fonctions de temps.

Le problème de l'ordonnancement des *SERU* est souvent résolu par des techniques de la programmation linéaire, comme nous l'avons précédemment vu dans la section 3.1.2. Dans [Qui83], les méthodes de projection développées exploitent des caractéristiques des fonctions affines. Le caractère analytique de telles fonctions est intéressant dans le cadre d'applications traitant de matrices pleines.

L'ordonnancement des *SERU* peut aussi se résoudre par des fonctions combinatoires. Dans [ST89], les fonctions de temps utilisées exploitent le caractère combinatoire des dépendances dans les graphes et sont non analytiques. Le point commun entre les graphes qui sont traités dans [ST89] et ceux modélisant les dépendances des *SERU* est que leurs sommets sont localement connectés. L'intérêt que nous portons aux fonctions de nature combinatoire se justifie par l'optique de la parallélisation de méthodes numériques à matrices creuses.

Dans l'exemple (1) du produit matrice vecteur, on montre que la fonction T définie par:

$$\begin{array}{rcl} T & : & D \rightarrow I\!N \\ & & (i,j) \longmapsto T(i,j) = i + j - 1 \end{array} \tag{3.12}$$

est une fonction de temps optimale. En outre, elle est unique. Dans la figure (3.4), nous avons illustré cette fonction pour $n = 5$.

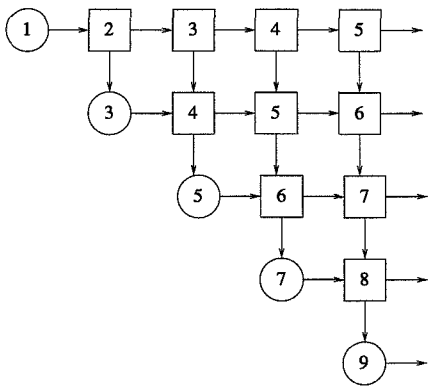

FIG. 3.4 – Fonction de temps affine pour le produit matrice pleine symétrique par un vecteur.

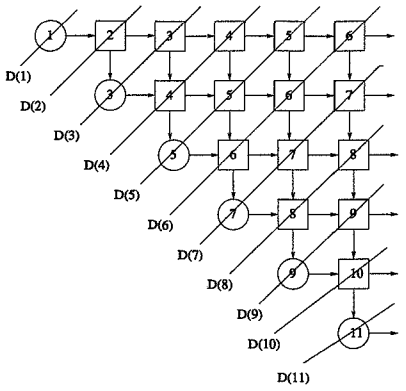

FIG. 3.5 – Sous ensembles exécutés à la même date pour le produit matrice pleine symétrique par un vecteur.

3.4.1.3 Complexité en nombre de processeurs

Etant donnée une fonction de temps optimale (au sens de la section 3.4.1.2), nous définissons maintenant le nombre minimum de processeurs requis pour exécuter les calculs avec un temps minimum. Une telle définition est possible dans le cadre de la synthèse des réseaux systoliques. Pour ce faire, on se place sous les hypothèses précédemment citées dans la section 3.4.1.2 pour caractériser une fonction de temps optimale. On introduit par la suite, une famille de sous-ensembles $D(t)$ de D défini par:

$$D(t) = \{u \in D \ : \ T(u) = t\} \tag{3.13}$$

$D(t)$ représente les points de D exécutés à la même date t. Par conséquent, le nombre de calculs élémentaires exécutés à la date t, soit $m(t)$, est défini par:

$$m(t) = |D(t)| \tag{3.14}$$

$m(t)$ est également le nombre de processeurs requis à la date t. Dans la suite, T_{sys} désigne le temps d'exécution optimal d'un algorithme systolique modélisé par le graphe G. Ainsi, toute stratégie de parallélisation conduisant à un algorithme "optimal" en temps nécessite au minimum P_{sys} processeurs, avec

$$P_{sys} = \min_{T} \max\{m(t) : \ 1 \leq t \leq T_{sys}\} \tag{3.15}$$

Dans le cadre de l'ordonnancement des *SERU*, il existe des techniques permettant d'évaluer de façon explicite, les formules (3.13), (3.14) et (3.15). Nous ne cherchons pas des expressions explicites pour ces formules dans le cas des dépendances dynamiques. Il existe au moins une méthode de calcul automatique du nombre optimal de processeurs P_{sys}. Il s'agit d'une méthode calculant le nombre paramétrique exact de points à coordonnées entières d'un polyèdre convexe paramétré, sous la forme d'un polynôme d'*Ehrhart*. Cette méthode a été implantée dans la librairie polyédrique *Polylib (http://icps.u-strasbg.fr/polylib)*.

La présence des formules (3.13), (3.14) et (3.15) dans le cadre de cette approche se justifie à double titre. D'une part, pour un découpage par blocs, ces formules interviennent dans la définition de critères d'optimalité au sens du lemme de *Brent* sur une *PRAM*. Ces critères d'optimalité permettent de quantifier formellement la perte du parallélisme occasionnée par des découpages par blocs. D'autre part, dans le cadre de la stratégie d'allocation que nous proposons, elles servent à définir un nombre minimum de processeurs requis par la méthode heuristique d'allocation.

Pour illustrer notre propos, on considère l'ordonnancement, pour une exécution parallèle optimale, des calculs d'un produit matrice vecteur (voir (3.4)). Pour les matrices pleines, la fonction de temps étant affine, il est possible de caractériser explicitement les sous ensembles $D(t)$. En effet, on a:

$$D(t) = \{(x_1, x_2) \in D \ : \ x_2 = -x_1 + 1 + t\}. \tag{3.16}$$

La figure (3.5) illustre les sous ensembles $D(t)$ lorsque la matrice symétrique est de taille 6×6. En représentant, pour chaque ligne, le nombre minimum de processeurs nécessaires pour exécuter en parallèle les calculs associés aux points de $D(t)$ aux instants successifs $t = 1, \ldots, 2n-1$, on obtient un tableau semblable à celui illustré dans la figure (3.6).

	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$	$t=7$	$t=8$	$t=9$	$t=10$	$t=11$
$j=1$	1	1	1	1	1	1	0	0	0	0	0
$j=2$	0	0	1	1	1	1	1	0	0	0	0
$j=3$	0	0	0	0	1	1	1	1	0	0	0
$j=4$	0	0	0	0	0	0	1	1	1	0	0
$j=5$	0	0	0	0	0	0	0	0	1	1	0
$j=6$	0	0	0	0	0	0	0	0	0	0	1
$m(t)$	1	1	2	2	3	3	3	2	2	1	1

FIG. 3.6 – *Calcul du nombre de processeurs pour le produit d'une matrice symétrique pleine de taille* 6×6 *par un vecteur.*

Dans ce tableau, on évalue P_{sys}, en calculant le maximum des $m(t)$, et chaque $m(t)$ s'obtient en sommant tous les éléments de la même colonne. Ainsi, on vérifie que c'est pour $t^* = 5, 6, 7$, le nombre de processeurs maximal est atteint et vaut $P_{sys} = 3$. On montre aisément que dans le cas général $t^* = n-1, n, n+1$ et $P_{sys} = \frac{n}{2}$, si n est pair, et $t^* = n$ et $P_{sys} = \frac{n+1}{2}$ si n est impair (on utilisera la notation $P_{sys} = \lceil \frac{n}{2} \rceil$, dans les deux cas).

3.4.2 Partitionnement du schéma d'exécution systolique de référence

3.4.2.1 Position du problème

Soit $G = (D, E)$ le graphe d'exécution d'un algorithme systolique dont les complexités en temps et en nombre de processeurs sont respectivement T_{sys} et P_{sys}. Etant donnée une machine d'exécution à P processeurs, où $P < P_{sys}$, on cherche un partitionnement (parmi tous les partitionnements possibles du graphe G) donnant lieu à une exécution optimale du graphe G sur une machine parallèle à P processeurs. Un tel partitionnement, s'il existe, constitue un partitionnement optimal. La recherche d'un tel partitionnement est un problème difficile.

Dans le cadre du modèle systolique, il existe des résultats d'optimalité du partitionnement pour des machines d'exécution qui ignorent le surcoût des communications. Il suffit pour ce faire, de considérer la méthode des hyperplans de *Lamport* [Lam74].

Dans le domaine des machines parallèles à mémoire distribuée, on se contente de solutions approchées. En effet, l'impact des communications et de l'architecture sur le temps d'exécution n'est pas toujours modélisé de façon satisfaisante. L'état de l'art et les perspectives dans ce domaine sont présentés dans des thèses comme [Bou96, Dar93]. La construction d'un partitionnement optimal dans ce domaine a fait l'objet de nombreuses démonstrations de *NP-complétude*. Les méthodes heuristiques qui font l'unanimité se fondent sur des partitionnements par blocs et la recherche de compromis calcul/communication. Ceci conduit à des algorithmes dont les résultats de complexité les plus intéressants, à notre connaissance, sont ceux des algorithmes s'exécutant en $O(n \log n)$, où n est la dimension de l'espace des itérations.

Dans ce qui suit, nous présentons une stratégie de partitionnement permettant de résoudre le compromis calcul/communication. L'originalité de cette stratégie de partitionnement est un nouveau schéma d'exécution modulaire (au sens des réseaux systoliques) et tel que, sur un modèle de machine où les coûts des communications sont négligeables, le meilleur grain

de parallélisme se trouve en considérant une optimalité au sens du lemme de *Brent*.

3.4.2.2 Une stratégie de partitionnement

On considère un graphe $G = (D, E)$ modélisant les dépendances du *SERU* (3.8). Le domaine des indices est donc tel que $D \subset \mathbb{Z}^q$. Les étapes du partitionnement sont les suivantes:

1– Choisir le plus petit rectangle B de $\mathbb{Z}^q$ contenant D. On a

$$B = B_1 \times \cdots \times B_q \text{ où } B_i \subset \mathbb{Z}$$

2– Définir un quadrillage de B en découpant B_i en $n_i - 1$ points intermédiaires. Soit

$$x_{i,1} < \cdots < x_{i,n_i-1}$$

3– Définir une partition Π de D en considérant des droites d'équation $x_{i,j}$ parallèles aux cotés de B,

4– Construire un graphe quotient G_Π, d'après la définition 5, et en déduire un graphe $G' = (D', E')$, en réduisant en un point, tout sommet du graphe G_Π et en un arc, tous les arcs entre deux sommets de G_Π ayant la même orientation,

5– Choisir à l'étape (2) des quadrillages des B_i de sorte que le graphe G', construit à l'étape (4), soit similaire au graphe G (au sens de la définition 6).

Afin de traduire mathématiquement la similitude entre les graphes G et G', on cherche un graphe G' qui modélise les dépendances d'un *SERU* qui est sous la forme:

$$X_i(z) = g_i(X_1(z - u_1), X_2(z - u_2), \cdots, X_p(z - u_p)) \tag{3.17}$$

pour $i = 1, \cdots, p$.

- ⋆ z parcourt un domaine $D' \subset \mathbb{Z}^q$ et représente un ensemble d'indices,
- ⋆ les g_i sont des fonctions à p variables décrivant les calculs ayant lieu en z,
- ⋆ les u_i sont des éléments de $\mathbb{Z}^q$ encore appelés vecteurs de dépendance,
- ⋆ chaque X_i est supposé connu lorsque $z \notin D'$.

L'équation (3.17) peut être représentée par un graphe orienté dont l'ensemble des sommets est constitué par le domaine D' et dont l'ensemble des arcs, soit E', est formé par les couples $(z-u_i, z)$. Tout terme $X_i(z-u_i)$ situé dans la partie droite de l'équation 3.17 est utilisé comme donnée en z, tandis que $X_i(z)$ est le résultat d'un calcul effectué en z. De la même manière que pour le *SERU* (3.8), la modularité du modèle systolique permet une caractérisation des sommets de G' en termes de classes d'équivalence.

La définition 6 de la similitude pose la question de l'existence d'un partitionnement par blocs, d'un graphe G, conduisant à une graphe G' tels que G et G' soient similaires. Cette question admet une réponse positive au moins dans le cas des dépendances uniformes de méthodes numériques telles que: la factorisation de *Gauss*, les produits matrices vecteurs et les résolutions de systèmes triangulaires (voir chapitre 4). Il s'agit de dépendances caractérisées

par des vecteurs de dépendance formant la base canonique de l'espace vectoriel $\mathbb{R}^q$, avec $q > 0$. Dans ce cas, on parvient ainsi à une formulation de la méthode numérique utilisant des matrices et des vecteurs blocs par blocs.

Pour illustrer notre propos, nous considérons le graphe G de la figure (3.3). Pour ce graphe, l'ensemble des sommets de G est tel que:

$$D = \bigcup_{i=1}^{5} \{(i,j) \ : \ i \leq j \leq 5\} \tag{3.18}$$

et le plus petit rectangle de $\mathbb{Z}^2$ contenant D est défini par $B = [1,5] \times [1,5]$. Dans la figure (3.7), nous avons illustré le quadrillage dans les deux dimensions de B. Le long de la direction $\vec{i}$, les points intermédiaires du quadrillage de l'intervalle $[1,5]$ sont $i \in \{2,3\}$. Le long de la direction $\vec{j}$, le point intermédiaire du quadrillage de l'intervalle $[1,5]$ est $j = 3$. On obtient l'agrégation des points de D illustré dans la figure (3.7). Dans la figure (3.8), nous avons illustré le graphe G' déduit du découpage de la figure (3.7). Notons que pour l'exemple du produit matrice vecteur, en choisissant deux quadrillages distincts pour les deux dimension, les graphes G et G' ne sont plus similaires. Par exemple, dans la figure (3.8), le graphe G' n'est pas similaire à un graphe G définit par les équations de récurrence uniformes de l'exemple (1). En effet, le point $(1,1)$ a deux voisins dans la structure de G' au lieu d'un seul comme c'est le cas dans la structure de G.

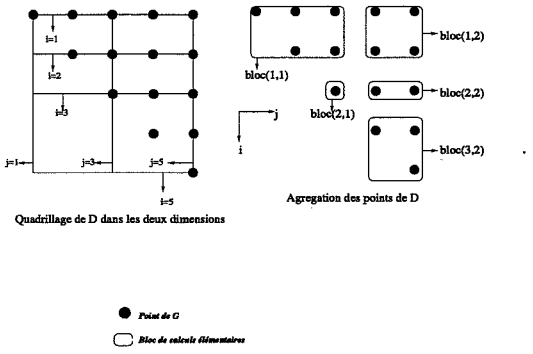

FIG. 3.7 – Découpage par blocs de calculs élémentaires du graphe G.

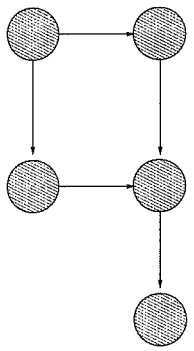

FIG. 3.8 – Structure du graphe G'.

(3.9).

Dans la figure (3.10), le graphe obtenu est similaire au graphe G de la figure (3.9). Pour obtenir la similitude dans le cadre du produit matrice vecteur, il suffit de considérer la même suite de points intermédiaires pour les deux dimensions lorsque les matrices sont pleines. Un tel découpage défini une matrice par blocs dont la manipulation est formellement semblable à celle d'une matrice par points.

Dans la mesure où, pour un graphe G, la stratégie de partitionnement précédemment décrite permet de construire de nouveaux graphes G', il se pose la question de l'optimalité des algorithmes parallèles modélisés par les nouveaux graphes G'. Pour répondre à cette question,

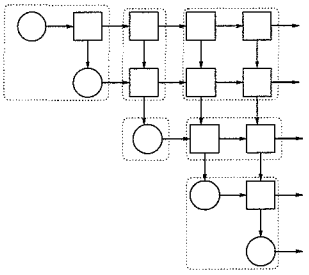

FIG. 3.9 – Découpage de G

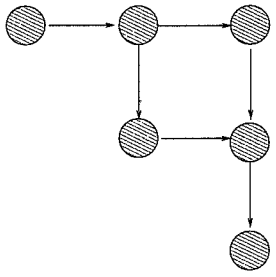

FIG. 3.10 – Structure de graphe G' similaire à G.

nous devons établir les conditions nécessaires et suffisantes pour que des partitionnements du graphe G soient optimaux sur des machines parallèles à P processeurs. Pour des modèles de machines telles que les *PRAM*, ces conditions peuvent être formulées explicitement. Pour des modèles de machines parallèles à coûts de communication non nuls, on se contente généralement d'heuristiques.

CNS de partitionnement optimal pour un modèle d'exécution PRAM

On considère une *PRAM* à P processeurs et un schéma d'exécution systolique. Les analyses de la section 3.4.1 permettent de définir le parallélisme potentiel du schéma d'exécution systolique. En reprenant les notations de la section 3.4.1, nous nous intéressons à la construction d'un algorithme parallèle optimal avec $P < P_{sys}$ processeurs à partir du schéma d'exécution systolique optimal. Pour ce faire, nous procédons en deux étapes comme suit:

étape1:

étant donné t, $1 \leq t \leq T_{sys}$, on considère les sous ensembles $D(t)$ tels que définis à la section 3.4.1.3. Chaque $D(t)$ est partitionné en $m(t)$ parties égales, lorsque $m(t) \leq P$ et équitablement en P parties si $m(t) > P$ (on dira également que $D(t)$ est partitionné équitablement en $\lceil \frac{m(t)}{P} \rceil$ parts). Soit alors

$$D(t) = \bigcup_{i=1}^{\min(P,m(t))} D(t,i) \tag{3.19}$$

où les $D(t,i)$ sont deux à deux disjoints

étape2:

les $D(t,i)$ sont alloués aux $\min(P, m(t))$ processeurs disponibles, pour

$$i = 1, \cdots, min(P, m(t))$$

Ces deux étapes définissent une condition nécessaire et suffisante caractérisant un algorithme exécutant le graphe G avec une complexité optimale sur P processeurs d'une *PRAM*.

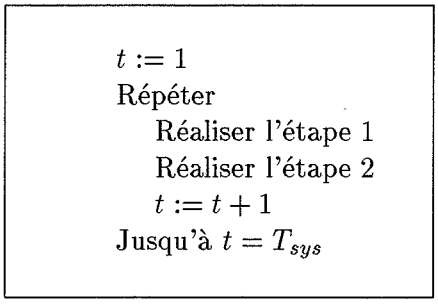

FIG. 3.11 – *Allocation des calculs élémentaires à P processeurs*

Ceci se vérifie aisément à l'aide de l'algorithme de la figure 3.11. La mise en oeuvre d'une telle condition se fonde sur le partitionnement équitable des sous-ensembles $D(t)$. Elle conduit à un algorithme parallèle dont la complexité en temps est égal à $T_{opt}(P)$. Ainsi, avec P processeurs, le i-ème pas d'exécution peut être subdivisé en $\lceil \frac{m(i)}{P} \rceil$. On vérifie que le temps d'exécution optimal sur une *PRAM*, soit $T_{opt}(P)$, est tel que:

$$T_{opt}(P) = \sum_{i=1}^{T_{sys}} \lceil \frac{m(i)}{P} \rceil \tag{3.20}$$

Comme nous le verrons dans la section 3.6.1, il est difficile qu'un partitionnement par blocs satisfasse à ces conditions nécessaires et suffisantes d'optimalité sur une *PRAM*. Dans la figure (3.12), nous illustrons un partitionnement des sous ensembles $D(t)$ pour l'exemple du produit matrice vecteur avec $P = 2$. On vérifie que $T_{opt}(2) = 10$. C'est un partitionnement qui valide la condition nécessaire et suffisante avec une *PRAM* à $P = 2$ processeurs. Il induit une agrégation des coefficients de la matrice le long d'une diagonale. Chaque processeur agit sur des termes de la matrice. Sur chaque processeur, le stockage de ces termes requiert la connaissance des indices lignes et des indices colonnes et conduit à un mode de stockage dit *aléatoire* [LT86b]. De tels stockages sont prohibitifs en espace mémoire.

Une heuristique de partitionnement

Il s'agit de définir des conditions suffisantes permettant une optimalité au sens de l'application du lemme de *Brent* [Bre73] au schéma d'exécution systolique de la section 3.4.1. Pour cela, rappelons le lemme (1) du à *Brent.*

Lemme 1 (de Brent)
On considère une machine parallèle de type PRAM (Parallel Random Access Machine) *composée de P processeurs séquentiels indépendants. Si un algorithme parallèle nécessite t unités de temps et T_{seq} opérations pour résoudre un problème donné, alors il existe un algorithme avec P processeurs et dont le temps d'exécution est inférieur ou égal à $t + \frac{T_{seq}-t}{P}$.*

Avant d'appliquer le lemme de *Brent* au schéma d'exécution systolique de la section 3.4.1, remarquons que le nombre d'unités de temps t requis par l'algorithme parallèle est égal à

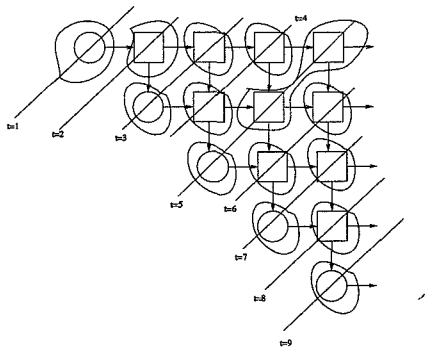

FIG. 3.12 – Partitionnement optimal des calculs pour une PRAM à 2 processeurs.

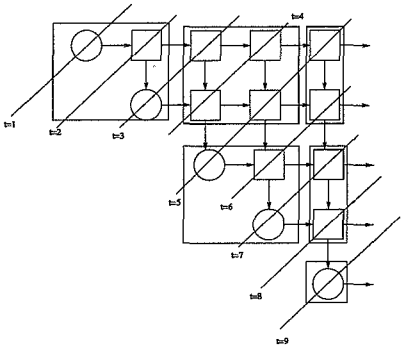

FIG. 3.13 – Découpage par bloc du graphe G

T_{sys}. Par conséquent, $T_{opt}(P)$ est majoré par:

$$T_{opt}(P) \leq (1 - \frac{1}{P})T_{sys} + \frac{1}{P}T_{seq} \tag{3.21}$$

Dans la figure (3.13), pour un découpage par bloc de taille 2×2, on peut vérifier que $T_{exe}(G') = 13$ et que $P = 2$. On a $T_{exe}(G') > T_{opt}(2)$. Afin de contrôler l'écart entre $T_{exe}(G')$ et $T_{opt}(P)$ sur une *PRAM*, l'heuristique que nous proposons pour le choix des tailles de bloc est la suivante:

Heuristique 1
On considère les étapes (1) *à* (4) *du partitionnement par blocs des sommets de G tel que définis dans la section 3.4.2.2. Si* T_{sys} *est très petit devant* T_{seq}*, alors un partitionnement par blocs du graphe G est optimal si le temps d'exécution sur une* PRAM*, soit* $T_{exe}(G')$*, de l 'algorithme parallèle qui en résulte reste majorée par:*

$$T_{exe}(G') \leq (1 - \frac{1}{P})T_{sys} + \frac{1}{P}T_{seq} \tag{3.22}$$

Cette heuristique possède deux applications intéressantes. La première application est obtenue avec un modèle de machine *PRAM* pour un schéma d'exécution systolique pour lequel T_{sys} est négligeable devant T_{seq}. En effet, s'il existe G' tel que

$$T_{exe}(G') = (1 - \frac{1}{P})T_{sys} + \frac{1}{P}T_{seq} \tag{3.23}$$

Le facteur d'accélération $\frac{T_{seq}}{T_{exe}(G')}$ est proche de P, car:

$$\frac{T_{seq}}{T_{exe}(G')} = P\frac{1}{1 + (p-1)\frac{T_{sys}}{T_{seq}}} \tag{3.24}$$

Une deuxième application est obtenue pour des machines d'exécution à coûts de communication non nuls. En effet, en considérant un graphe d'exécution G' construit en minimisant le

surcoût des communications, le temps d'exécution dépend quasiment de l'exécution des calculs. Si de plus T_{sys} est négligeable devant T_{seq}, on peut obtenir des facteurs d'accélération proche de P. Dans la section 3.6, l'analyse effectuée permet de montrer qu'il en existe des partitionnements par blocs qui soient optimaux au sens de l'heuristique 1. Dans le chapitre 4, les résultats expérimentaux ont permis de valider une telle heuristique.

3.4.3 Ordonnancement du nouveau schéma d'exécution

3.4.3.1 Position du problème

Etant donné un graphe de tâches $G' = (D', E')$ obtenu par partitionnement d'un graphe $G = (D, E)$, plusieurs algorithmes parallèles peuvent décrire l'exécution du graphe G'. Notons $\mathcal{A}$, un algorithme parallèle décrivant une exécution possible de G' et $T_{\mathcal{A}}(G')$ le temps d'exécution de l'algorithme $\mathcal{A}$. Quel que soit l'algorithme $\mathcal{A}$, $T_{\mathcal{A}}(G')$ est le temps de fin d'exécution du sommet de G', sans successeur, qui est exécuté en dernier. Formellement, on a:

$$T_{\mathcal{A}}(G') = \max_{\{u \in D' :\ Succ(u)=\emptyset\}} \{Deb(u) + Ex(u)\} \tag{3.25}$$

où $Deb(u)$ est le temps de début d'exécution du sommet u et $Ex(u)$ est la durée d'exécution du sommet u.

L'algorithme parallèle optimal est un algorithme $\mathcal{A}_0$ tel que:

$$T_{\mathcal{A}_0}(G') = \min_{\mathcal{A}} T_{\mathcal{A}}(G') \tag{3.26}$$

L'ordonnancement optimal est la fonction de temps qui atteint la borne inférieure définie par l'expression (3.26).

Dans le cadre de la synthèse des réseaux systoliques, le problème de l'ordonnancement optimal se résout en négligeant les communications et en supposant que toutes les tâches ont un temps d'exécution unitaire. Nous avons précédemment vu dans la section 3.1.2 que les fonctions de temps proposées étaient affines et le temps d'exécution d'un algorithme systolique optimal était égal au temps d'exécution du plus long chemin du graphe.

Avec les machines à mémoire distribuée, les résultats de l'ordonnancement sans communication peuvent être étendus en prenant en compte les communications. Le modèle le plus fréquemment utilisé revient à considérer un graphe valué, dont le coût des arcs dépend des données acheminées entre les tâches reliées par ces arcs. Formellement, si $(u, v) \in E'$, on a la relation suivante:

$$Deb(v) \geq Deb(u) + Ex(u) + C(u, v) \tag{3.27}$$

où $Deb(u)$ est le temps de début d'exécution du sommet u, $Ex(u)$ est la durée d'exécution du sommet u et $C(u, v)$, le coût de l'acheminement des données entre les sommets u et v.

Dans le modèle défini par la relation 3.27, la quantité $C(u, v)$ dépend de la fonction de routage de la machine parallèle. Par conséquent, sa modélisation n'est pas toujours satisfaisante et donne lieu à de nombreuses heuristiques.

Dans ce qui suit, nous proposons un ordonnancement selon le modèle défini par l'expression 3.27. Pour calculer le temps d'exécution parallèle, on montre par récurrence que le temps de fin d'exécution de chaque super-noeud est égal au temps d'exécution du chemin de plus fort poids reliant ce super-noeud à un super-noeud sans prédécesseur. Pour ce faire, nous renvoyons le lecteur à la section 3.3.1.

3.4.3.2 Une stratégie d'ordonnancement

Pour définir un ordonnancement des sommets du graphe de dépendance G', on considère les notions de prédécesseur et successeur définies dans la section 3.3.1, et on suppose que l'exécution de tout sommet du graphe G' est atomique. Pour tout sommet u de G', soit $T_{cal}(u)$ le temps d'exécution des transformations élémentaires de u. Pour tout arc (u, v) de G', soit $T_{com}(u, v)$ le temps d'acheminement des messages de u à v. Soit $Pred(u)$, l'ensemble des prédécesseurs de u et $Succ(u)$ l'ensemble des successeurs de u. La fonction Φ modélise le coût des communications sur une architecture donnée lorsque ces communications doivent se faire simultanément avec les prédécesseurs d'un noeud.

La présente stratégie d'ordonnancement se caractérise par une fonction de temps, soit T, qui intègre les coûts de calculs et de communications. Elle a comme fil conducteur, la recherche de "*chemin à plus fort poids*", dont la définition (7) est fournie dans la section 3.3.1. L'idée de la fonction de temps T est qu'un super-noeud u, à l'extrémité d'un chemin à plus fort poids, ne peut être complété qu'à une date égale à $Poids(u)$. Par construction, T peut être définie par:

$$\begin{array}{rcl} T & : & D' \rightarrow I\!N \\ & & u \longmapsto T(u) = Poids(u) \end{array} \tag{3.28}$$

Le marquage des sommets de G' par les valeurs de T peut se traduire en termes d'un algorithme parallèle, noté $\mathcal{A}_1$, qui procède comme suit, pour chaque super-noeud:

1.1 recevoir des données de ses prédécesseurs,

1.2 exécuter tous ses calculs de façon atomique,

1.3 envoyer les résultats à ses successeurs.

En vertu de la formule (3.3), le temps d'exécution de l'algorithme $\mathcal{A}_1$, soit $T_{exe}(G')$, est au moins égal au plus fort poids des sommets sans successeurs. Formellement, on a:

$$T_{\mathcal{A}_1}(G') \geq Poids(G') = \max_{Succ(v)=\emptyset} \{Poids(v)\} \tag{3.29}$$

Cependant, l'algorithme $\mathcal{A}_1$ est-il optimal? Pour y répondre, nous examinons les conditions nécessaires et suffisantes pour qu'un algorithme parallèle soit optimal sur une machine *PRAM*.

CNS d'ordonnancement optimal pour un modèle d'exécution PRAM

Quel que soit l'algorithme parallèle qui simule l'exécution de G', il simule implicitement celle du graphe $G = (D, E)$ qui a été partitionné. Dans le cadre du modèle systolique, nous avons défini dans la section 3.4.1.2, un ordonnancement optimal du graphe G. Pour cet ordonnancement optimal de G, considérons les sous ensembles $D(t)$ de D, formés de sommets de G ayant la même date d'exécution tels que définis dans la section 3.4.1.3. Par construction de G', on partitionne implicitement les ensembles $D(t)$, pour $t = 1, \cdots, T_{sys}$, où T_{sys} est le temps d'exécution optimal du schéma systolique. En vertu des contraintes de dépendance dans le graphe G, une borne minimum du temps d'exécution de G' est obtenue en considérant un algorithme parallèle, soit $\mathcal{A}_0$, dont les calculs, sur chaque super-noeud, procèdent de façon non atomique et par parcours des $D(t)$, pour $t = 1, \cdots, T_{sys}$. Sur un modèle de type *PRAM*, une condition nécessaire et suffisante pour avoir un ordonnancement optimal est de construire une

fonction de temps qui atteint cette borne. Clairement, la fonction T définie par l'expression (3.28) n'est pas optimale sur une *PRAM*.

L'algorithme $\mathcal{A}_0$ est intéressant sur une *PRAM* lorsque son temps d'exécution est borné au sens du lemme de *Brent*. Nous avons vu sous certaines conditions, décrites dans la section 3.4.2.2, que le facteur d'accélération pouvait être proche du nombre de processeurs utilisés. Malheureusement, sur des modèles de machines à mémoire distribuée tels que définis dans la section 3.3.2, mettre en oeuvre $\mathcal{A}_0$ est relativement prohibitif en termes de structures de données et de contrôle. Nous avons donc besoin d'une heuristique d'ordonnancement permettant de gérer au mieux la capacité de mémorisation et la puissance de calcul des machines parallèles d'exécution.

Une heuristique d'ordonnancement

Considérons l'algorithme $\mathcal{A}_1$ précédemment introduit dans cette section. L'heuristique d'ordonnancement est la suivante:

Heuristique 2
Soit un marquage des sommets de G' par la fontion de temps T définie par la formule (3.28). L'algorithme $\mathcal{A}_1$ est optimal si $T_{exe}(G')$ est majoré au sens du lemme de Brent *sur une* PRAM.

Une application de cette heuristique est obtenue lorsqu'il existe des exemples d'applications et des partitionnements par blocs qui valident l'heuristique 2. Lorsque ces deux heuristiques sont validées, une conséquence sur les machines parallèles à mémoire distribuée est qu'avec la réduction du coût des communications induites par un partitionnement par blocs et une exécution atomique des super-noeuds, les temps d'exécution peuvent devenir presque optimaux. Ceci a été validé expérimentalement au chapitre 4.

3.4.4 Allocation du nouveau schéma d'exécution

3.4.4.1 Position du problème

Etant donné un graphe de tâches $G' = (D', E')$ et un ordonnancement compatible T, l'allocation des tâches consiste à associer à chaque tâche, un processeur d'exécution de sorte que le temps d'exécution de G' soit celui indiqué par T. Plus formellement, l'allocation est une application définie par:

$$\begin{array}{rcl} A & : & D' \rightarrow \mathcal{P} \\ & & u \longmapsto A(u) \end{array} \tag{3.30}$$

$A(u)$ est le processeur qui exécute u et $\mathcal{P}$ est l'ensemble des processeurs. Une allocation optimale est une allocation garantissant un temps d'exécution induit par un ordonnancement optimal. Dans le cas où on suppose que la structure du graphe de tâches est définie a priori, on cherche un *placement* optimal des tâches. On peut y parvenir de deux façons.

Une première façon de faire consiste à considérer un placement et à en déduire le meilleur ordonnancement des calculs. Cette façon de faire est classiquement utilisée dans le domaine du *data-parallélisme*. Toutefois, la question qui se pose ici est celle de savoir si le meilleur ordonnancement déduit d'un placement, défini a priori, donne lieu à une exécution parallèle optimale. A défaut de définir un placement "optimal", on suppose généralement que le placement est résolu par des heuristiques permettant: de minimiser le coût des communications, des fonctions de placement linéaires.

Une deuxième façon de procéder consiste à définir un ordonnancement et à en déduire une allocation préservant le temps d'exécution induit par cet ordonnancement. Il existe au moins une telle allocation. Lorsque l'ordonnancement est optimal, on peut espérer une allocation optimale. Cependant, quel est le coût en termes de structures de données et de contrôle pour la mise en oeuvre d'une telle allocation? Quels sont les coûts de communications qui en résultent lors de l'exécution du programme parallèle.

Nous proposons une stratégie qui se situe dans le cadre de la deuxième façon précédemment évoquée, et s'applique à l'algorithme $\mathcal{A}_1$ défini dans la section 3.4.3.2. Une première propriété de cette stratégie est que le temps d'exécution de $\mathcal{A}_1$ reste borné inférieurement (respectivement supérieurement) par le poids de $Poids(G')$ (respectivement par le terme de droite de la formule (3.4). Une deuxième propriété est qu'on alloue implicitement les super-noeuds via une structure de donnée construite par projection.

3.4.4.2 Une stratégie d'allocation

Il s'agit d'un placement de tâches sous la contrainte de la taille mémoire. En effet, notre propos est d'étendre les structures de données déjà choisies pour l'algorithme séquentiel que l'on désire paralléliser. Au lieu d'allouer explicitement les super-noeuds aux processeurs, on les couple à des structures de donnée qui dépendent de cet algorithme séquentiel. Ce couplage induit de nouveaux types de données qui sont explicitement allouées aux processeurs. Pour ce faire, la méthode proposée se fonde sur les méthodes de projection telles que définies dans [BR90, CMPia, Tou90], et sur l'hypothèse selon laquelle les sommets de G' sont tous identiques. Soit $proj_{\vec{d}}$, la projection dans la direction $\vec{d}$, et T une fonction de temps optimale au sens de la section 3.4.1.2. Nous procédons en deux étapes comme suit:

étape1:

on considère le graphe $\mathcal{G} = (\mathcal{D}, \mathcal{E})$ obtenu par projection du graphe G' dans la direction $\vec{d}$. Un sommet c de $\mathcal{G}$ se définit comme l'image des super-noeuds situés sur une droite de direction $\vec{d}$. Un arc est orienté de c_1 vers c_2, soit $(c_1, c_2) \in \mathcal{E}$, si un super-noeud projeté sur c_1 a un successeur projeté sur c_2. Pour tout $c \in \mathcal{D}$, posons:

$$\mathcal{I}_c = \{u \in D' \ : \ proj_{\vec{d}}(u) = c\} \tag{3.31}$$

et

$$T(\mathcal{I}_c) = \{T(u) \ : \ u \in \mathcal{I}_c\} \tag{3.32}$$

Chaque sommet $c \in \mathcal{D}$ est étiquetté par un intervalle dont les bornes correspondent aux dates d'activation des sommets u_c et v_c, avec:

$$T(u_c) = \min_{u \in \mathcal{I}_c} T(u) \tag{3.33}$$

et

$$T(v_c) = \max_{u \in \mathcal{I}_c} T(u) \tag{3.34}$$

Formellement on a une fonction de temps $\mathcal{T}$ sur $\mathcal{G}$:

$$\mathcal{T} : c \longmapsto [T(u_c), T(v_c)] \tag{3.35}$$

On peut aisément vérifier que la fonction de temps $\mathcal{T}$ induite par cette projection préserve le temps d'exécution des sommets situés sur le plus long chemin.

étape2:

elle consiste à définir une allocation des sommets $\mathcal{G}$ à P processeurs de façon compatible avec la fonction $\mathcal{T}$. Nous proposons pour cela une fonction d'allocation définie comme suit:

1: $k \leftarrow 0$

2: $K \leftarrow \emptyset$(ensemble de processeurs auxquels les noeuds ont été alloués)

3: Pour tous les noeuds c créés à l'étape 1 faire

3.1: Chercher dans K le plus petit élément k_{min} tel que le dernier intervalle d'activation du processeur $\mathcal{P}_{k_{min}}$ n'intersecte pas $\mathcal{T}(c)$

3.2: Si k_{min} existe, allouer c au processeur $\mathcal{P}_{k_{min}}$

3.2: Sinon

$k \leftarrow k + 1$

allouer c à $\mathcal{P}_k$

$K \leftarrow K \cup \{k\}$

finsi

La question qui se pose est la suivante: *la stratégie ainsi définie par les étapes* 1 *et* 2 *qui précèdent, préserve t-elle la quantité* $Poids(G')$*, qui est le temps minimum d'exécution de l'algorithme* $\mathcal{A}_1$*, défini dans la section 3.4.3.2?* Une fois de plus, nous y répondons en examinant le cas d'une allocation optimale sur un modèle *PRAM.*

CNS d'allocation optimale pour un modèle d'exécution PRAM

Sur une *PRAM*, une condition nécessaire et suffisante pour une allocation optimale du graphe G' est de préserver le temps d'exécution $T_{\mathcal{A}_0}(G')$, où $\mathcal{A}_0$ est l'algorithme précédemment introduit dans la section 3.4.3.2. Par construction, il est difficile, voire impossible, d'avoir l'égalité $T_{\mathcal{A}_0}(G') = T_{exe}(G')$, du fait de l'exécution atomique des super-noeuds dans l'algorithme $\mathcal{A}_1$.

Une heuristique d'allocation

Soit T, une fonction de temps qui préserve $T_{exe}(G')$ sur une *PRAM*. Une condition suffisante pour que $T_{exe}(G') = Poids(G')$ consiste à allouer les super-noeuds aux processeurs de façon à préserver le plus long chemin à plus fort poids $[u, v]$, où u est un sommet sans prédécesseur, et v est un sommet sans successeur. Une telle condition n'est pas toujours validée par les étapes 1 et 2 de la stratégie d'allocation précédemment décrite. Sa validation remet en cause le choix de la projection de l'étape 1, qui permet de passer d'une allocation des tâches à celle des données. De plus, les sommets de G' n'étant pas toujours identiques et il est très coûteux d'évaluer le poids des sommets puis de procéder à une allocation aux processeurs les moins chargés. D'où la recherche de méthodes approchées permettant de générer un code réaliste tout en restant le plus près possible de $Poids(G')$.

En supposant que tous les super-noeuds de G' ont le même temps d'exécution, par exemple la valeur maximale, les fonctions de temps définies dans la section 3.4.1.2 fournissent de bonnes heuristiques d'ordonnancement des noeuds de G'. Elles permettent pour l'algorithme $\mathcal{A}_1$, un temps d'exécution après allocation, soit $T_{exe}(G')$, qui reste majoré par la borne supérieure de l'expression (3.4), et minoré par la quantité $Poids(G')$. Etant donnée une fonction de temps T, optimale au sens où elle conduit à un temps d'exécution égal au nombre de sommets du plus long chemin, appelons date d'activation d'un super-noeud u, le nombre $T(u)$. La figure (3.14) illustre un ordonnancement des sommets de G' par une telle fonction de temps. Les super-noeuds ayant la même date d'activation peuvent être exécutés en "parallèle". Pour qu'un placement soit compatible avec un tel ordonnancement, il suffit que $P \geq P'_{sys}$, où P'_{sys} est le nombre maximum de super-noeuds activés à la même date.

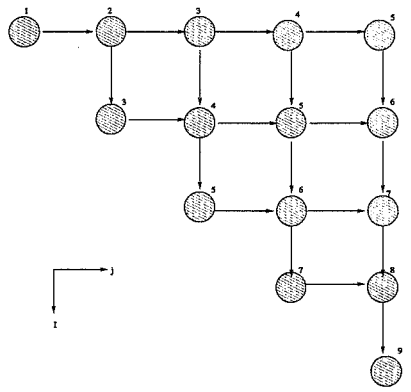

FIG. 3.14 – Ordonnancement du graphe G'

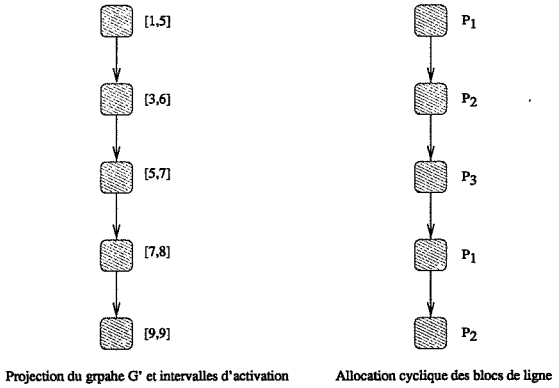

FIG. 3.15 – Projection de G' dans la direction $(0,1)$ et allocation des calculs aux processeurs.

Dans le cas du produit matrice vecteur, l'ordonnancement par une fonction de temps linéaire (voir section 3.4.1.2) est illustré dans la figure (3.14). En choisissant $\vec{d} = (0,1)$, on obtient le graphe illustré par la figure (3.15) où chaque sommet est étiqueté par un intervalle dont les bornes correspondent aux dates d'activation des sommets (i,i) et (i,n). Dans ce cas la direction $\vec{d}$ est la direction de propagation de la donnée $\{x_i, y_i\}$ dans laquelle on obtient le résultat final. On obtient une allocation des blocs de lignes de façon cyclique à $P'_{sys} = 3$ processeurs.

Avec $P \geq P'_{sys}$ processeurs, l'heuristique est la suivante:

Heuristique 3

Soient les deux étapes de la stratégie d'allocation définie dans la section 3.4.4.2. Une telle stratégie d'allocation des sommets de G' qui préserve le temps d'exécution des sommets situés sur le plus long chemin dans G' est considéré comme optimal si le temps $T_{exe}(G')$ qu'elle induit reste majoré au sens du lemme de Brent.

3.4.5 Génération de code et des structures de données

Nous détaillons maintenant la dernière phase de la parallélisation qui consiste à générer un code *SPMD* pour l'exécution du graphe $G' = (D', E')$. Le choix d'un langage à parallélisme explicite semble judicieux pour:

- contrôler la génération des communications,
- favoriser l'expression de tous le parallélisme exhibé par la stratégie d'ordonnancement de la section 3.4.3.2.

Le couplage de la stratégie de partitionnement de la section 3.4.2.2 et de la stratégie d'ordonnancement de la section 3.4.3.2 induit implicitement une réindexation du domaine des indices D du schéma d'exécution systolique précédemment défini dans la section 3.4.1. Une telle réindexation se traduit par une énumération de tous les points de D.

Dans le domaine de la parallélisation automatique, les algorithmes d'énumération utilisés s'appuient sur des techniques de programmation linéaire en nombre entiers. Nous ne discutons pas ici, de la mise en oeuvre de ces techniques de programmation linéaire. L'approche qui est proposée repose sur la synthèse systématique, via une structure de données appropriée, du graphe G'. Cette structure de données se définit comme une collection de cellules, qui s'obtiennent par projection du graphe G' dans une direction, soit $\vec{d}$. La direction $\vec{d}$ qui est choisie est celle de l'étape 1 de la stratégie d'allocation précédemment décrite dans la section 3.4.4.2, et chaque cellule est un sommet du graphe $\mathcal{G} = (\mathcal{D}, \mathcal{E})$. Une cellule c peut donc être définie à travers:

- le processeur auquel elle a été allouée,
- l'ensemble $\mathcal{I}_c$ des super-noeuds projetés sur c,
- les prédécesseurs et les successeurs de $u \in \mathcal{I}_c$,
- ses dates d'activation, soit $T(\mathcal{I}_c)$, avec T une fonction de temps telle que choisie dans la section 3.4.4.2.

En vertu de la stratégie de partitionnement par bloc précédemment définie dans la section 3.4.2.2, on associe à tout super-noeud de G' un sous système d'équations récurrentes. Nous proposons de simuler l'exécution de ce sous système dans le cadre d'un algorithme séquentiel faisant appel à des structures de données qui sont définies par: les données associés par construction à chaque cellules, les données envoyées par les prédécesseurs et les données envoyées aux successeurs d'un super-noeud. Dans la mesure où le graphe G' est similaire au graphe systolique G, la construction des algorithmes pour tous les sommets de G' procède par classe. Ainsi, les cellules précédemment définies peuvent encore être caractérisées par:

- une liste chainée de coordonnées de super-noeuds $u \in \mathcal{I}_c$, chaque super-noeud pointant sur les coordonnées de ses successeurs et prédécesseurs,
- des classes d'algorithmes associés aux super-noeuds,
- des structures de données dépendant des classes d'algorithmes,
- le processeur auquel est alloué la cellule.

Le code parallèle généré est illustré dans la figure (3.16). Dans ce code, notons que si le super-noeud u et ses prédécesseurs (respectivement successeurs) sont alloués au même processeur, la réception (respectivement l'envoi) correspond à une lecture (respectivement écriture) en mémoire locale.

```
Pour toutes les cellules c allouées à un processeur
 Pour u ∈ I_c
    Recevoir de tous les prédécesseurs de u
    Exécuter l'algorithme de u
    Envoyer à tous les successeurs de u
 Fin_pour
Fin_pour
```

FIG. 3.16 – *Exemple de code parallèle généré.*

Dans la figure (3.18), nous illustrons les structures de données requises pour la mise en oeuvre d'un algorithme par bloc dont le graphe est défini dans la figure (3.17). Avec la stratégie d'allocation précédemment décrite dans la section 3.4.4.2, chaque processeur est responsable du traitement des super-noeuds projetés sur les cellules qui lui sont allouées.

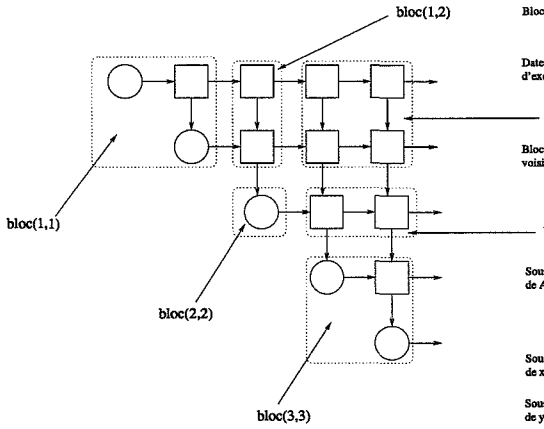

FIG. 3.17 – Découpage d'un graphe de produit matrice vecteur.

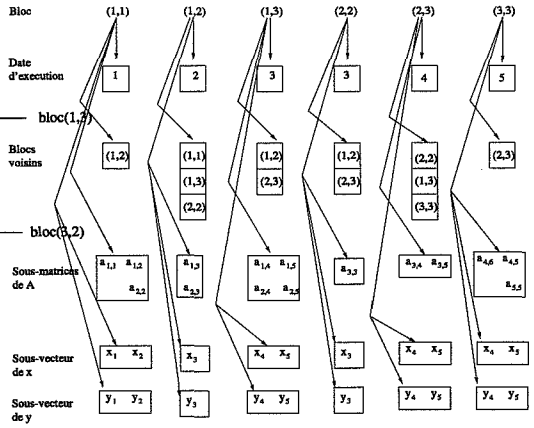

FIG. 3.18 – Structures de données requises pour les calculs en parallèle.

3.5 Cas des matrices creuses

Dans ce qui précède, nous avons présenté et analysé une nouvelle approche de parallélisation des applications. Elle repose sur des concepts tels que les *systèmes d'équations de récurrence uniformes (SERU)* la synthèse d'algorithmes systoliques et le *partitionnement par blocs* pour produire des algorithmes pour des modèles de machines telles que des machines *MIMD* à mémoire distribuée. Les principes généraux de cette approche ont été établis en supposant une organisation des calculs au tour de structures régulières. Nous proposons maintenant une adaptation de ces principes aux cas où les calculs reposent sur une organisation irrégulière des calculs. Cette adaptation est nécessaire dans la mesure où, pour bon nombre de traitements de problèmes de taille industriels, il existe des techniques telles que les *éléments finis*, les *différences finies*, les *décompositions de domaine*, etc, qui produisent des formulations de méthodes numériques dans lesquelles les matrices de très grande taille sont creuses. Dans la section section 2.3, nous avons discuté de l'intérêt du recours aux rangements creux. Malheureusement, la parallélisation des méthodes numériques ainsi formulées reste problématique.

Si à ce jour, le partitionnement par blocs est utilisé aussi bien pour le stockage des matrices creuses que pour la parallélisation des méthodes numériques, en revanche, très peu de choses qui découlent des *SERU* s'appliquent concrètement aux méthodes numériques à matrices creuses. Certes, on peut toujours manipuler une matrice creuse sous sa forme dense, toutefois, les stockages denses pour des matrices de très grande taille sont prohibitifs. Nous proposons donc de rajouter aux étapes 1 à 4 exposées dans la section 3.4, une étape dite de relaxation. La procédure de parallélisation devient donc la suivante:

Etape 1 : réécriture d'un schéma d'exécution séquentiel d'une application donnée sous la forme d'un schéma d'exécution systolique,

Etape 2 : partitionnement par bloc d'un graphe d'exécution systolique en un graphe qui lui est "similaire",

Etape 2.1 : élimination dans le graphe construit en 2, de tous les sommets dont la contribution est nulle,

Etape 3 : ordonnancement en temps minimum du graphe obtenu à l'étape 2.1,

Etape 4 : allocation des sommets du graphe obtenu à l'étape 2.1 en préservant "au mieux", l'ordonnancement en temps minimum de l'étape 3.

Dans l'étape 2.1, il suffit par exemple, de considérer le *SERU* défini à l'étape 2, et d'affecter le signal 0 à tout super-noeud dont le sous système d'équations récurrentes a une contribution nulle, et le signal 1 à tout super-noeud dont le sous système d'équations récurrentes a une contribution non nulle. Par élimination de tous les super-noeuds ayant un signal égal à 0, on parvient ainsi à des équations de récurrence qui se caractérisent par des dépendances irréguliers. Dans ce qui suit, nous discutons de la synthèse systématique de ces dépendances et de l'ordonnancement des équations de récurrence qui en découlent. Puisque les graphes $G = (D, E)$ et $G' = (D', E')$, construits respectivement aux étapes 1 et 2 sont similaires, il suffit d'étudier formellement la modification des *SERU* définis par l'expression (3.8). Nous renvoyons le lecteur intéressé au chapitre 4 pour les illustrations.

3.5.1 Extension des SERU

On considère un problème défini sous la forme d'un *SERU* (voir section 3.4.1.1). Pour un tel *SERU*, les indices des équations récurrentes sont obtenue par translation et le nombre de vecteurs de dépendance est fini. Lorsqu'un tel *SERU* est utilisé pour la description d'une méthode numérique à matrice creuse, il induit obligatoirement la représentation des calculs à contribution nulle, par définition du rangement creux. Une élimination totale de ces calculs peut être obtenue en appliquant formellement la procédure de l'étape 2.1. Cependant une synthèse systématique et automatique de ces dépendances est difficilement envisageable dans tous les cas. Afin de permettre une synthèse systématique de ces dépendances, nous acceptons une représentation de quelques équations récurrentes à contribution nulle. Pour ce faire, nous procédons comme suit:

1– préservation de toutes les directions de dépendance,

2– dans chaque direction de dépendance, les points représentant les calculs non nuls sont localement connectés.

On parvient ainsi, à une relaxation d'un *SERU*. Pour un problème à résoudre, on cherche une solution séquentielle exprimée sous la forme suivante:

$$Y_i(z) = f_i(Y_1(v_1(z)), Y_2(v_2(z)), \cdots, Y_p(v_p(z))) \tag{3.36}$$

pour $i = 1, \cdots, p$.

- ⋆ z parcours un domaine $D \subset \mathbb{Z}^q$ et représente un ensemble d'indices,
- ⋆ les f_i sont des fonctions à p variables décrivant les transformations élémentaires ayant lieu en z,
- ⋆ les fonction v_i sont définies dans $\mathbb{Z}^q$ et décrivent des vecteurs de dépendance de longueur variable,
- ⋆ chaque Y_i est supposé connu lorsque $z \notin D$.

L'équation (3.36) est une relaxation de l'équation (3.8). Nous parlerons de *systèmes d'équations de récurrence uniformes relaxées(SERUR)*. L'équation (3.36) peut être représentée par un graphe orienté G dont l'ensemble des sommets est constitué par le domaine D et dont les arcs sont les couples $(v_i(z), z)$. Tout terme $Y_i(v_i(z))$ situé dans la partie droite de (3.36) est utilisé comme donnée en z, tandis que $Y_i(z)$ est le résultat d'un calcul effectué en z. Il y a donc un arc (z', z) dans G si un résultat produit en z' est utilisé comme donnée en z.

3.5.2 Ordonnancement des SERUR

Pour un *SERU*, nous avons précédemment vu, dans la section 3.1.2, que le problème de l'ordonnancement est souvent résolu par des techniques de programmation linéaire et ses solutions appartiennent à un polyèdre. Les méthodes de résolution de ce problème reposent sur le fait que les dépendances entre les calculs peuvent être représentées par un nombre fini de vecteurs. Pour les *SERUR*, ces résultats ne s'appliquent plus car les vecteurs de dépendance ne sont plus en nombre fini. En effet, dans une direction de dépendance, la loi décrivant la variation des vecteurs de dépendance est définie en fonction du mode de rangement creux choisi.

Dans ce qui suit, nous supposons qu'en chaque point de D, les calculs ont la même durée. Dans la pratique, ceci n'est pas vrai car les calculs ne sont pas les mêmes suivant les sommets du graphe. Dans ce cas, les fonctions de temps construites autour de la notion de poids des chemins d'un graphe peuvent être considérées (voir section 3.3.1 et 3.4.3). Cependant, il est possible de supposer une exécution synchrone des calculs en prenant comme unité de temps le maximum des durées associées aux différents points de D. Ceci nous permet d'exhiber des fonctions de temps qui pourront être utilisées dans le cadre des deux étapes d'allocation de la section 3.4.4.2.

En ce qui concerne les applications à matrices creuses, bien que les sommets restent localement connectés, les vecteurs de dépendance peuvent varier suivant une loi quelconque. Ainsi, les fonctions de temps considérées, pour être optimales, doivent tirer profit du mode de rangement creux et partant, du caractère combinatoire du graphe G. Les fonctions au plus tôt et au plus tard constituent deux possibilités de construction de telles fonctions pour des graphes dont les noeuds sont localement connectés. Dans ce qui suit, soit d la longueur du plus long chemin du graphe G.

Description d'une fonction de temps au plus tôt

Puisque chaque point u de D qui est à l'extrémité d'un chemin de longueur $d(u)$ ne peut être exécuté qu'à la date $1 + d(u)$, on définit la date d'exécution au plus tôt d'un point u par:

$$T_1(u) = 1 + d(u) \tag{3.37}$$

où $d(u)$ est la longueur d'un plus long chemin d'extrémité u.

Si G comporte un arc (u, v) on vérifie aisément que $T_1(u) < T_1(v)$. Par conséquent T_1 est une fonction de temps. Il est donc clair, par construction, que:

$$\max\{T_1(u) : \ u \in D\} = 1 + d \tag{3.38}$$

et que T_1 est optimale, si le graphe ne contient pas de sommets à contribution nulle.

Description d'une fonction de temps au plus tard

La construction d'une fonction de fonction de temps au plus tard procède en deux étapes. Dans un premier temps, on affecte la date $1 + d$ à tous les points de G qui sont à l'extrémité d'un plus long chemin. Dans un second temps, tant qu'il y a des points non encore datés, on en choisit un, soit u, tel que pour tout arc (u, v), v est déjà daté et on affecte à u la date:

$$T_2(u) = -1 + \min\{T_2(v) : \ (u, v) \ \text{ arc de } G\} \tag{3.39}$$

Par construction, cette fonction vérifie les conditions requises pour une fonction de temps et est optimale, si le graphe ne contient pas de sommets à contribution nulle.

3.6 Analyse de complexité de la méthodologie de parallélisation

Pour une méthode numérique donnée, nous avons étudié deux schémas d'exécutions possibles. D'un coté, un schéma d'exécution purement systolique caractérisé par:

⋆ son graphe de dépendance $G = (D, E)$,

⋆ une fonction de temps optimale, soit T, avec T_{sys} (respectivement P_{sys}) la complexité en termes de temps (respectivement du nombre de processeurs).

De l'autre coté, pour une machine parallèle du type *MIMD* à mémoire distribuée, nous avons proposé un deuxième schéma permettant de simuler le graphe G. Il s'agit d'un schéma qui se caractérise par:

⋆ son graphe de dépendance $G' = (D', E')$, qui est similaire au graphe G,

⋆ un algorithme $\mathcal{A}_1$ qui exécute G' dans un temps $T_{exe}(G')$, après allocation des tâches.

Soient d, la longueur d'un plus long chemin de G', u_0 le sommet de G' tel que:

$$T_{cal}(u_0) = \max_{u \in D'} T_{cal}(u) \tag{3.40}$$

et $(u_1, v_1) \in E'$ tel que:

$$\Phi(q, T_{com}(u_1, v_1)) = \max_{(u,v) \in E'} \Phi(q, T_{com}(u, v)) \tag{3.41}$$

En réutilisant ici, la notion de *Poids* introduite dans la section 3.3.1, on peut vérifier que:

$$Poids(G') \leq T_{exe}(G') \leq (d+1)T_{cal}(u_0) + d\Phi(q, T_{com}(u_1, v_1)) \tag{3.42}$$

La question qui se pose est celle de savoir si un algorithme $\mathcal{A}_1$ qui exécute G' dans un temps $T_{exe}(G')$, est optimal. Pour y répondre, nous analysons a priori, le degré d'optimalité de $T_{exe}(G')$ dans les trois sens suivant:

1– exécution sur une *PRAM*,

2– exécution sur une *PRAM* avec majoration au sens du lemme de *Brent*,

3– modèle d'exécution prenant en compte les coûts de communication.

3.6.1 Optimalité de la méthode dans un modèle d'exécution PRAM

Pour établir les conditions nécessaires et suffisantes pour une telle optimalité, nous reconsidérons les notations issues du modèle systolique de la section 3.4.1. Plus précisément, rappelons les notations de la section 3.4.1.3. Pour $t = 1, \cdots, T_{sys}$, nous avons défini $D(t)$ et $m(t)$ par:

$$D(t) = \{u \in D \ : \ T(u) = t\} \text{ et } m(t) = |D(t)| \tag{3.43}$$

Dans l'expression (3.43), T désigne une fonction de temps optimale au sens précisé dans la section 3.4.1.2. Par construction de G', tout sommet u de G' contient un sous graphe de G. L'ensemble des noeuds de ce sous graphe intersecte quelques $D(t)$, pour $t = 1, \cdots, T_{sys}$. De façon formelle, on a:

$$u = \bigcup_{t=1}^{T_{sys}} (D(t) \cap u) \tag{3.44}$$

Dans l'expression (3.44), nous avons fait un abus de langage en confondant le sommet u et l'ensemble des noeuds du sous graphe qu'il contient. Notons également $|u|$, le nombre de

noeuds de G contenus dans u. Puisque les $D(t)$, pour $t = 1, \cdots, T_{sys}$ forment une partition de D (ensemble des sommets de G), la valeur de $|u|$ est définie par:

$$|u| = \sum_{t=1}^{T_{sys}} |D(t) \cap u| \tag{3.45}$$

L'évaluation de $T_{exe}(G')$ conduit à analyser les sommets de G' qui sont traités obligatoirement de façons atomique et séquentielle. Soit $\mathcal{S}$, l'ensemble de ces sommets. En supposant que les sommets de G ont un temps d'exécution unitaire, on vérifie que $T_{exe}(G')$ vaut:

$$T_{exe}(G') = \sum_{u \in \mathcal{S}} |u| = \sum_{u \in \mathcal{S}} \sum_{t=1}^{T_{sys}} |D(t) \cap u| \tag{3.46}$$

Nous cherchons maintenant à quantifier l'écart entre $T_{exe}(G')$ et le temps d'exécution optimal du graphe G sur une *PRAM* à P processeurs, soit $T_{opt}(P)$. Dans la section 3.4.2.2, nous avons montré que:

$$T_{opt}(P) = \sum_{t=1}^{T_{sys}} \lceil \frac{m(t)}{P} \rceil \tag{3.47}$$

Si on pose:

$$\epsilon(t) = \sum_{u \in \mathcal{S}} |D(t) \cap u| - \lceil \frac{m(t)}{P} \rceil \tag{3.48}$$

où $m(t) = |D(t)|$, alors on a:

$$T_{exe}(G') = T_{opt}(P) + \sum_{t=1}^{T_{sys}} \epsilon(t) \tag{3.49}$$

Notons qu'avec P processeurs utilisés pour exécuter G', on a nécessairement:

$$\epsilon(t) \geq 0 \text{ pour } t = 1, \cdots, T_{sys} \tag{3.50}$$

Par conséquent on a:

$$T_{exe}(G') = T_{opt}(P) \Leftrightarrow \epsilon(t) = 0 \text{ pour } t = 1, \cdots, T_{sys} \tag{3.51}$$

L'expression (3.51) est une condition nécessaire et suffisante pour une optimalité sur une machine *PRAM* à P processeurs. Cette condition semble peu réaliste pour des découpages par blocs.

3.6.2 Optimalité de la méthode au sens du Lemme de Brent

Dans cette section, nous considérons les notations de la section 3.6.1. Dans la section 3.4.2.2, l'heuristique (1) est proposée pour définir un sens de l'optimalité pour un partitionnement. En reprenant la formule (3.40), l'expression (3.42) devient:

$$Poids(G') \leq T_{exe}(G') \leq (d+1) T_{cal}(u_0) \tag{3.52}$$

Soit T_{seq}, le temps d'exécution séquentiel. Pour que $T_{exe}(G')$ soit majorée au sens du lemme de *Brent*, il suffit que:

$$(d+1)T_{cal}(u_0) \leq (1-\frac{1}{P})T_{sys} + \frac{1}{P}T_{seq} \tag{3.53}$$

Nous avons précédemment remarqué que cette majoration peut fournir une exécution parallèle optimale si T_{seq} est très grand par rapport à T_{sys}. Sous cette hypothèse et en résolvant un problème d'optimisation en nombres entiers dont les inconnues sont:

- ⋆ le nombre de calculs scalaires contenus dans u_0,
- ⋆ la longueur du plus long chemin de G'.

on peut prédire, a priori, l'optimalité du temps d'exécution $T_{exe}(G')$. Les expériences effectuées dans le chapitre 4 nous ont permis de valider les conditions suffisantes pour l'optimalité au sens du lemme de *Brent*:

- ⋆ T_{seq} très grand devant T_{sys},
- ⋆ existence de u_0 et d solution de l'équation (3.53).

3.6.3 Analyse de complexité de la méthodologie dans un modèle d'exécution avec communication

Dans ce qui suit, nous supposons acquise, l'optimalité au sens du lemme de *Brent* sur une *PRAM*. Nous cherchons des conditions suffisantes sur le surcoût des communications pour qu'il reste négligeable par rapport au coût des calculs. Pour ce faire, nous partons de la majoration suivante de la fonction Φ:

$$\max_{(u,v)\in E'} \Phi(q, T_{com}(u,v)) \leq qd \max_{(u,v)\in E'} T_{com}(u,v) \tag{3.54}$$

Cette majoration de Φ s'établit en considérant le pire des cas c'est à dire le cas où, non seulement le temps d'acheminement des données entre deux sommets est le maximum, mais les données qui sont envoyées par les prédécesseurs sont acheminées les unes après les autres. Dans ces conditions, le temps d'exécution devient borné par:

$$Poids(G') \leq T_{exe}(G') \leq (d+1)T_{cal}(u_0) + dqT_{com}(u_2, v_2) \tag{3.55}$$

où $(u_2, v_2) \in E'$ tel que:

$$T_{com}(u_2, v_2) = \max_{(u,v)\in E'} T_{com}(u,v) \tag{3.56}$$

Une condition suffisante pour se rapprocher d'un modèle à coût de communications négligeable est de trouver des partitionnements par blocs tels que:

$$\frac{dqT_{com}(u_2, v_2)}{(d+1)T_{cal}(u_0)} \ll 1 \tag{3.57}$$

Ceci est possible dans la mesure où, en partant d'un modèle systolique et en utilisant une agrégation par blocs, les communications entre les super-noeuds dépendent du nombre de dépendances qui traversent le coté du rectangle de $\mathbb{Z}^q$ qui contient le super-noeud, tandis que les calculs dans un super-noeud dépendent du volume du rectangle de $\mathbb{Z}^q$ qui le contient. Dans le chapitre 4, les exemples d'application choisis valident une telle condition suffisante. Ainsi, pour avoir une optimalité sur un modèle de machine à coût de communications non nul, il suffit:

- ⋆ d'une optimalité au sens du lemme de *Brent* sur une *PRAM*,

⋆ de rendre la quantité (3.57) très petite devant 1.

3.7 Conclusion

Dans ce chapitre, nous nous sommes intéressés à la mise en oeuvre d'une méthodologie de parallélisation optimale pour méthodes numériques. Cette méthodologie fusionne trois aspects: l'analyse quantitative de la systolisation, le partitionnement par blocs et l'allocation par blocs.

L'analyse quantitative de la systolisation consiste en une réutilisation des techniques existantes pour la synthèse des réseaux systoliques à fronts d'ondes, pour évaluer quantitativement, le parallélisme d'une méthode numérique. Ces techniques ont fait l'objet de nombreuses études dans le cas où les méthodes numériques sont à matrices denses. Dans le cas où les matrices sont creuses, ces techniques restent formellement valides. Notons cependant dans ce cas que beaucoup de calculs sont nuls, par construction du stockage creux, et ils ne contribuent pas à la production des résultats. Pour identifier l'ensemble des calculs dont les contributions sont nécessaires et suffisantes pour l'exécution d'une telle méthode numérique, il suffit par exemple, d'affecter le signal 0 à toute équation récurrente ayant une contribution nulle, et le signal 1 à toute équation récurrente ayant une contribution non nulle.

Le partitionnement par blocs conduit à définir une nouvelle structure de graphe par agrégation des équations récurrentes fournies par une analyse quantitative de la systolisation. Pour obtenir cette nouvelle structure de graphe, on considère une formulation d'un problème en terme de systèmes d'équations récurrentes uniformes *(SERU)*. L'intuition de l'agrégation par blocs réside sur les techniques de quadrillage utilisées dans les méthodes de discrétisation du type *différences finies* ou *éléments finis*. Ces quadrillages sont utilisés pour le maillage de domaines continus. En utilisant ces techniques dans le cadre du domaine des indices d'un *SERU*, on parvient à un partitionnement dans lequel, les super-noeuds sont les mailles. Dans ce nouveau graphe, certains super-noeuds peuvent être à contribution nulle (les super-noeuds qui ne contiennent pas de sous-graphe). Ces super-noeuds sont identifiables, en utilisant l'affectation des signaux 0 et 1 précédemment mentionnée.

Enfin, l'allocation par blocs consiste en un placement de tâches sous la contrainte de la taille mémoire. En effet, notre propos est d'étendre les structures de données déjà choisies pour l'algorithme séquentiel que l'on désire paralléliser. Au lieu d'allouer explicitement les super-noeuds aux processeurs, on les couple à des structures de donnée qui dépendent de cet algorithme séquentiel. Ce couplage induit de nouveaux types de données qui sont explicitement allouées aux processeurs.

Afin de détailler et d'analyser ces trois aspects nous avons introduit une modélisation de l'application parallèle et de la machine parallèle d'exécution.

Chapitre 4

Application: calcul des modes et fréquences propres dans SYSTUS

Dans ce chapitre, nous nous intéressons à la parallélisation d'un module de calcul de modes et fréquences propres du logiciel *SYSTUS*. En particulier, nous appliquons la nouvelle approche systolique étudiée au chapitre 3, aux algorithmes dont l'exécution est prohibitive dans le cadre d'un calcul de modes et fréquences propres. Dans la section 4.1, une introduction à *SYSTUS* permet de présenter le module de calcul de modes et fréquences propres à paralléliser. Il s'agit du module *DLANCB* dont la parallélisation est discutée dans la section 4.2. Enfin, nous terminons ce chapitre par une évaluation des performances avant et après la parallélisation.

4.1 Introduction à SYSTUS

4.1.1 Présentation générale de SYSTUS

SYSTUS, développé à partir de 1962 par des bureaux d'étude de génie civil, puis à partir de 1976 par la société *FRAMATOME*, est actuellement commercialisé par la société *SYSTUS International (Groupe ESI)*. Il est destiné à l'analyse du comportement thermo-mécanique des structures par la méthode des éléments finis. Les maillages par éléments finis interviennent à toutes les étapes du traitement: modélisation physique, conditions aux limites, chargement et expression des résultats. Les types d'analyses disponibles dans *SYSTUS* sont:

- ⋆ l'analyse statique linéaire; il s'agit de problèmes de l'élasticité linéaire en petites déformations;
- ⋆ l'analyse dynamique d'une structure à partir de ses modes et fréquences propres;
- ⋆ l'analyse non-linéaire;
- ⋆ les analyses stochastiques.

Dans le cadre de nos travaux, nous nous intéressons à la partie de *SYSTUS* destinée à l'analyse dynamique d'une structure non-amortie à partir de ses modes et fréquences propres.

4.1.2 Analyse dynamique des vibrations

Soit Ω, le domaine de définition d'un milieu élastique linéaire sans amortissement. D'après le principe de d'*Alembert* [Lare], on peut écrire à un instant t donné:

$$\left\{\begin{array}{lcll} \sigma_{ij,j}+f_i & = & \rho\frac{\partial^2 u_i}{\partial t^2} & \text{sur } \Omega \\ \epsilon_{ij} & = & \frac{1}{2}(u_{i,j}+u_{j,i}) & \\ \sigma_{ij} & = & E_{ijkl}\epsilon_{kl} & 1\leq i\leq n \end{array}\right. \tag{4.1}$$

Avec les conditions aux limites du type:

$$\left\{\begin{array}{lcll} \sigma_{ij}n_j & = & \overline{T}_i & \text{sur } S_\sigma \\ u & = & \overline{u}_i & \text{sur } S_u \end{array}\right. \tag{4.2}$$

$\overline{u}_i$ est la i-ème composante du déplacement sur S_u et $\overline{T}_i$ la i-ème composante de la pression appliquée sur S_σ. Par application de la méthode des éléments finis aux équations (4.1) et (4.2), on se ramène à un système d'équations différentielles ordinaires en dimension finie défini par:

$$\left\{\begin{array}{ll} \text{Trouver } U(t) & \text{vérifiant:} \\ KU(t)+M\ddot{U}(t) = & F(t) \end{array}\right. \tag{4.3}$$

Le problème de l'analyse des vibrations consiste donc à déterminer les conditions sous lesquelles, l'équation (4.3), sans terme d'excitation ($F(t)=0$), permet un mouvement. En faisant l'hypothèse que

$$U(t)=\Phi\sin(\omega t+\tau) \tag{4.4}$$

l'équation (4.3) s'écrit:

$$(K\Phi-\omega^2 M\Phi)\sin(\omega t+\tau)=0 \quad \forall t \tag{4.5}$$

d'où

$$K\Phi-\omega^2 M\Phi=0 \tag{4.6}$$

Nous avons donc à résoudre un problème de recherche *d'éléments propres généralisés* (voir section B.1). Pour les structures considérées dans le cadre de nos études, les matrices K et M sont symétriques et à coefficients réels. Lorsqu'elles sont définies positives, alors les valeurs propres recherchées sont réelles et positives. Lorsque la matrice K est singulière, on introduit un décalage de manière à rendre régulier le problème spectral généralisé (voir définition 11 de l'annexe B). Si α est la valeur de ce décalage, on résout alors:

$$[K+\alpha M)-(\omega^2+\alpha)M]\Phi=0 \tag{4.7}$$

Nous présentons dans l'annexe B les principales classes de méthodes numériques de recherche d'éléments propres généralisés [Cha88, GL83, LT86b]. Pour chacune de ces classes, nous indiquons celles disponibles dans *SYSTUS*. La résolution des problèmes d'éléments propres dans *SYSTUS* se fait par le biais de deux classes de méthodes numériques:

1– la méthode de *Givens* destinée au traitement de problèmes d'éléments propres à matrices pleines et de taille relativement petite,

2– les méthodes de *sous espaces* de *Krylov* destinées aux traitements de problèmes d'éléments propres à matrices creuses et de taille relativement grande. Par exemple la méthode du sous-espace itératif, les méthodes de *Lanczos*, etc.

Dans le cas de matrices symétriques creuses et de grande taille, les méthodes de *Lanczos* par blocs sont les plus performantes et sont à la base du module *DLANCB* de *SYSTUS*.

4.1.3 Présentation du module DLANCB

Il permet de calculer quelques dizaines de fréquences propres (éventuellement multiples), autour d'une fréquence f_0, en utilisant une procédure en quatre phases illustrées par la figure (figure 4.1):

phase A: la construction du problème spectral généralisé,

phase B: factorisation de *Gauss* LDL^T,

phase C: calcul des valeurs propres,

phase D: divers post-traitements.

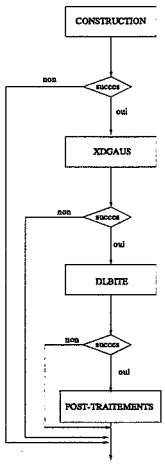

FIG. 4.1 – Fonctionnement général de DLANCB

4.1.3.1 La phase de construction du problème spectral

Elle s'appuie sur la discrétisation par éléments finis d'une structure, qui a été réalisée en amont par un mailleur. La construction se fait par lecture des fichiers décrivant le maillage par les éléments finis puis, procède à l'assemblage d'un couple de matrices, noté $(\overline{K}, \overline{M})$, avec:

$$\overline{K} = K + \alpha M \text{ et } \overline{M} = \frac{trace(K)}{trace(M)} M$$

Une des particularité de *DLANCB* est que les matrices $\overline{K}$ et $\overline{M}$ étant symétriques, on utilise leurs parties triangulaires supérieures pour effectuer les calculs. De plus le stockage de ces parties triangulaires supérieures est fait en mode skyline de sorte que, les deux matrices ont le même profil. Formellement, ce profil équivaut à celui d'une matrice $A = (a_{i,j})_{1 \leq i,j \leq n}$ dont la partie triangulaire supérieure stricte est formée par la réunion des sous ensembles d'indices:

$$struct(A_i) = \{j > i;\ a_{i,j} \neq 0\} \text{ pour } i = 1, \cdots, n \tag{4.8}$$

Le stockage skyline dans ces conditions revient à considérer la réunion:

$$[i, j_i] \text{ pour } i = 1, \cdots, n \tag{4.9}$$

où

$$j_i = \max\{j : j \in struct(A_i)\} \text{ pour } i = 1, \cdots, n \tag{4.10}$$

La largeur maximale de bande, soit lbw, est donnée par:

$$lbw = \max_{1 \leq i \leq n} \{l_i : l_i = j_i - i + 1\} \tag{4.11}$$

Le nombre de termes non nuls de la matrice A est défini par:

$$nz = \sum_{i=1}^{n} l_i \tag{4.12}$$

Dans la description des algorithmes à matrices creuses, nous utiliserons la notation $struct(A_i)$, pour chaque ligne i.

4.1.3.2 La phase de factorisation LDL^T

Dans le cadre des méthodes de *Lanczos par bloc* (voir les discussions des sections B.2 et B.3), la détermination d'une base orthonormale d'un sous espace de *Krylov* implique successivement des calcul de produits $\overline{K}^{-1}U$, avec U qui est une matrice rectangulaire. Or, au cours de ces calculs successifs, la matrice $\overline{K}$ est invariante. Dans ces conditions, il semble judicieux de procéder à une factorisation de $\overline{K}$ avant le calcul des produits $\overline{K}^{-1}U$. La matrice $\overline{K}$ étant seulement symétrique et inversible, car l'on souhaite trouver les fréquences autour d'une fréquence centrale f_0, sa factorisation sous la forme $\overline{K} = LDL^T$ permet de réduire les coûts de traitement de ces résolution de systèmes linéaires. Ainsi, le calcul $\overline{K}^{-1}U$ se fait par le biais de résolution de systèmes triangulaires inférieur puis supérieur. Nous allons dans ce qui suit décrire formellement la factorisation $A = LDL^T$.

Pour des raisons de commodité, nous avons noté: $A = (a_{i,j})_{1 \leq i,j \leq n}$, $L = (l_{i,j})_{1 \leq i,j \leq n}$ et $D = (d_{i,j})_{1 \leq i,j \leq n}$. Le calcul de la factorisation $A = LDL^T$ se fait dans ce cas par identification des coefficients situés dans la partie supérieure ou inférieure des matrices. Par exemple, en notant $L^T = (l^t_{i,j})_{1 \leq i,j \leq n}$, la matrice triangulaire supérieure de la factorisation LDL^T, l'identification des coefficients de la partie triangulaire supérieure de A, D et L^T donne:

$$a_{i,j} = \sum_{k=1}^{i} l^t_{k,i} d_{k,k} l^t_{k,j} \text{ pour } i \leq j \tag{4.13}$$

En supposant que nous effectuons un calcul ligne par ligne, on obtient l'algorithme suivant. Ayant calculé les $(i-1)$ premières lignes, les termes $l_{i,j}$ et $d_{i,i}$ s'obtiennent par les formules suivantes:

$$\begin{cases} d_{i,i} &= a_{i,i} - \sum_{k=1}^{i-1} (l^t_{k,i})^2 d_{k,k} \\ l^t_{i,j} &= (a_{i,j} - \sum_{k=1}^{i-1} l^t_{k,i} d_{k,k} l^t_{k,j}) / d_{i,i} \quad \text{pour } i < j \leq n \end{cases} \tag{4.14}$$

En vertu de la formule (4.14), il se peut que pour $a_{i,j} = 0$, $l^t_{i,j} \neq 0$. C'est ce qui se passe s'il existe $k < i$ avec $l^t_{k,j}$ et $l^t_{k,i}$ non nuls. Une *factorisation symbolique* [LT86a] permet de réserver la place mémoire et remplir des tableaux d'entiers nécessaires à la représentation de L^T. La

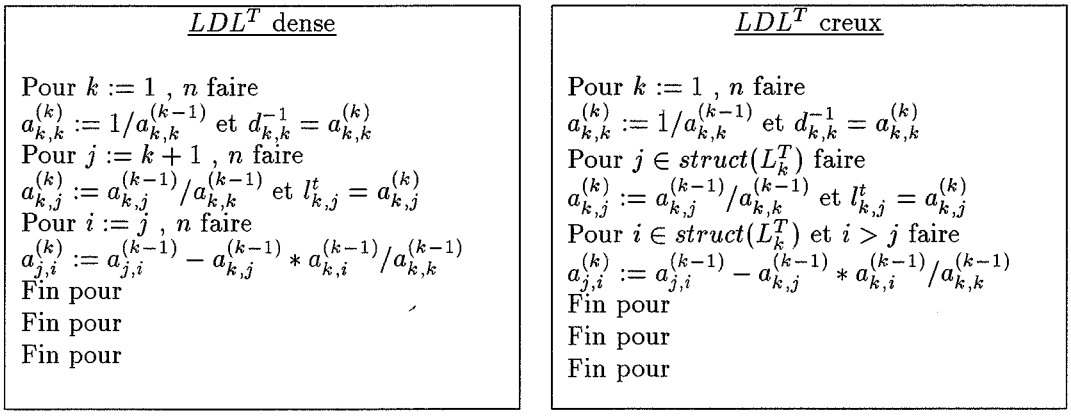

FIG. 4.2 – *Factorisation LDL^T pour matrices pleines et creuses.*

figure 4.2 illustre deux versions d'algorithmes permettant de calculer la factorisation LDL^T dans le cas où A est dense (LDL^T dense) et dans le cas où la matrice A est creuse (LDL^T creux), avec:

- ⋆ $a_{i,j}^{(k)}$ représente la valeur du terme $a_{i,j}$ après le calcul des k premières lignes,
- ⋆ $struct(L_i^T) = \{j > i;\ l_{i,j}^t \neq 0\}$ pour $i = 1, \cdots, n$ est calculé après la factorisation symbolique.

Pour le sous-programme de *XDGAUS* qui exécute la factorisation LDL^T à matrice skyline dans *DLANCB*, on montre que le nombre d'opérations scalaires effectué est majoré par (voir section 4.2.6.1):

$$n * lbw^2 \tag{4.15}$$

4.1.3.3 La phase de calcul des valeurs et fréquences propres

Le calcul des fréquences propres se fonde sur le principe des méthodes de *Lanczos* par blocs tel que décrit dans la section B.3. Toutefois, afin d'éviter les problèmes de perte d'orthogonalité qui retardent la convergence, des réorthogonalisations sont nécessaires. Tous ces calculs sont implantés dans le sous programme *DLBITE* et procèdent théoriquement comme suit:

a– formation d'une matrice tridiagonale par bloc, notée T_i (voir figure (4.3)) à la i-ème itération;

b– calcul des valeurs propres de la matrice tridiagonale par bloc;

c– sélection des valeurs propres et test de convergence;

d– calcul des fréquences propres.

L'organigramme général du programme implanté est celui décrit dans la figure (B.1) de l'annexe B. Nous renvoyons le lecteur intéressé à la section B.4 de l'annexe B, pour une présentation approfondie de l'implantation de la méthode de *Lanczos par blocs* dans *DLBITE*.

$$T_i = \begin{pmatrix} A_1 & F_1 & 0 & . & . & 0 \\ G_2 & A_2 & F_2 & 0 & . & 0 \\ 0 & G_3 & A_3 & F_3 & 0 & 0 \\ . & . & . & . & . & . \\ . & . & . & . & . & F_{i-1} \\ . & . & . & . & G_i & A_i \end{pmatrix}$$

FIG. 4.3 – *Exemple de matrice tridiagonale par bloc construite à l'étape i.*

Dans cette implantation de la méthode de *Lanczos*, les opérations standards telles que les produits de matrices skylines par des vecteurs et les résolutions de systèmes triangulaires à matrices skylines sont appelées plusieurs fois et sont de coût élevé. Avant de présenter la phase des post-traitements, nous exposons ces deux opérations prohibitives.

4.1.3.4 Le produit matrice vecteur

C'est une opération qui est décrite matriciellement par $y = Ax$ avec $A = (a_{i,j})_{1 \leq i,j \leq n}$ une matrice de dimension $n \times n$ et x et y des vecteurs de dimension n. Les coefficients de y sont obtenus par identification comme suit:

$$y_i = \sum_{j=1}^{n} a_{i,j}\, x_j \text{ pour } i = 1, \cdots, n \tag{4.16}$$

Dans le module *DLANCB*, seule la partie triangulaire supérieure de A est stockée et ses coefficients sont rangés ligne par ligne. Par conséquent, en prenant les $a_{i,j}$ tels que $j \geq i$, la formule (4.16) devient:

$$y_i = \sum_{j>i} a_{i,j}\, x_j + \sum_{j<i} a_{j,i}\, x_j + a_{i,i}\, x_i \tag{4.17}$$

$y = Ax$ dense

Pour $i := 1$, n faire
  $y_i^{(i)} := y_i^{(i-1)} + a_{i,i}x_i$
  Pour $j := i+1$, n faire
    $y_i^{(j-1)} := y_i^{(j-1)} + a_{i,j}x_j$
    $y_j^{(i)} := y_j^{(i-1)} + a_{i,j}x_i$
  Fin pour
Fin pour

$y = Ax$ creux

Pour $i := 1$, n faire
  $y_i^{(i)} := y_i^{(i-1)} + a_{i,i}x_i$
  Pour $j \in struct(A_i)$ faire
    $y_i^{(j)} := y_i^{(j-1)} + a_{i,j}x_j$
    $y_j^{(i)} := y_j^{(i-1)} + a_{i,j}x_i$
  Fin pour
Fin pour

FIG. 4.4 – *Produit matrice vecteur: cas de matrices symétriques pleines et creuses.*

La figure (4.4) illustre deux versions d'algorithmes permettant de calculer le produit matrice

vecteur $y = Ax$. Notons que:

- ⋆ $y_i^{(j-1)}$ représente la valeur de y_i avant l'addition de la contribution de $a_{i,j}x_j$,
- ⋆ $struct(A_i)$ est défini par la formule (4.8),
- ⋆ $y_i^{(0)}$ est la valeur initiale de y_i. Elle vaut 0 en règle générale.

Le nombre d'opérations scalaires ainsi exécuté est égal à (voir section 4.2.6.1):

$$\boxed{4nz - 2n} \tag{4.18}$$

4.1.3.5 Résolution de systèmes triangulaires

Les systèmes triangulaires sont des systèmes d'équations linéaires dont la représentation matricielle est $Ax = b$, avec $A = (a_{i,j})_{1 \leq i,j \leq n}$ une matrice qui est soit:

- ⋆ triangulaire supérieure i.e $a_{i,j} = 0$ pour $i > j$,
- ⋆ triangulaire inférieure i.e $a_{i,j} = 0$ pour $i < j$.

Lorsque A est une matrice triangulaire supérieure, on résout les équations "en remontant" dans l'ordre $n, n-1, \ldots, 1$, ceci permet de calculer successivement les inconnues dans cet ordre. Ainsi, à partir de la i-ème équation:

$$\sum_{j=i}^{n} a_{i,j}\, x_j = b_i \text{ pour } i = n, \cdots, 1 \tag{4.19}$$

on calcule

$$x_i = (b_i - \sum_{j=i+1}^{n} a_{i,j}\, x_j)/a_{i,i} \text{ pour } i = n, \cdots, 1 \tag{4.20}$$

Lorsque A est une matrice triangulaire inférieure, on résout les équations "en descendant" dans l'ordre $1, 2, \ldots, n$, ce qui permet de calculer successivement les inconnues dans cet ordre. Ainsi, à partir de la i-ème équation:

$$\sum_{j=1}^{i} a_{i,j}\, x_j = b_i \text{ pour } i = 1, \cdots, n \tag{4.21}$$

on calcule

$$x_i = (b_i - \sum_{j=1}^{i-1} a_{i,j}\, x_j)/a_{i,i} \text{ pour } i = n, \cdots, 1 \tag{4.22}$$

Il existe plusieurs façons de programmer les résolutions de systèmes triangulaires qui diffèrent: par l'ordre dans lequel sont effectués les calculs dans les équations (4.20) et (4.22), et par le mode d'accès aux données. Dans *DLANCB*, une même matrice triangulaire supérieure, soit A, est utilisée pour résoudre $Ax = y$ et $A^T u = v$. Ceci permet des économies en place mémoire. Notons que, d'après les formules (4.20) et (4.22), il n'est pas nécessaire de stocker x. En effet, comme b_i ne sert plus après le calcul de x_i, on peut ranger x_i à la place de b_i.

Compte tenu de ces hypothèses, nous explicitons maintenant les algorithmes séquentiels qui en découlent. En ce qui concerne la résolution de systèmes triangulaires inférieurs, ou encore la *descente*, la figure (4.5) décrit des algorithmes à matrice pleine et creuse. Dans ces deux algorithmes, il s'agit de résoudre $A^T x = b$, avec A une matrice triangulaire supérieure. D'où une mise en oeuvre des calculs par modifications successives des termes b_j, avec $j > i$ une colonne de la i-ème ligne telle que $a_{i,j} \neq 0$.

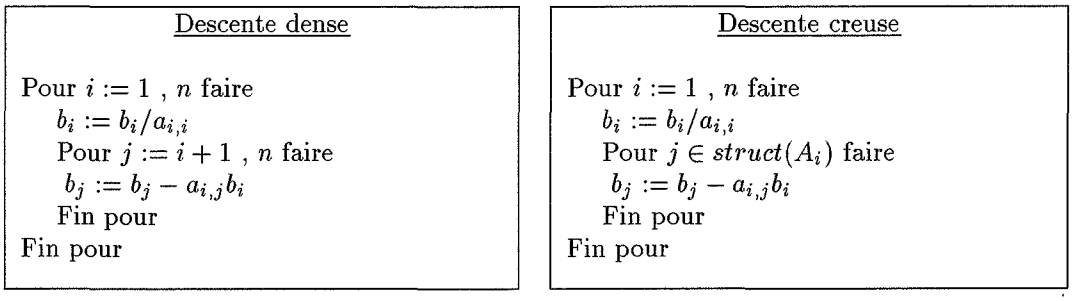

FIG. 4.5 – *Algorithmes de descente: cas de matrices symétriques denses et creuses stockées dans la partie triangulaire supérieure.*

La figure (4.6) décrit des algorithmes de remontée sur des matrices pleines et creuses. Dans ces algorithmes, les calculs procèdent conformément à la formule (4.20).

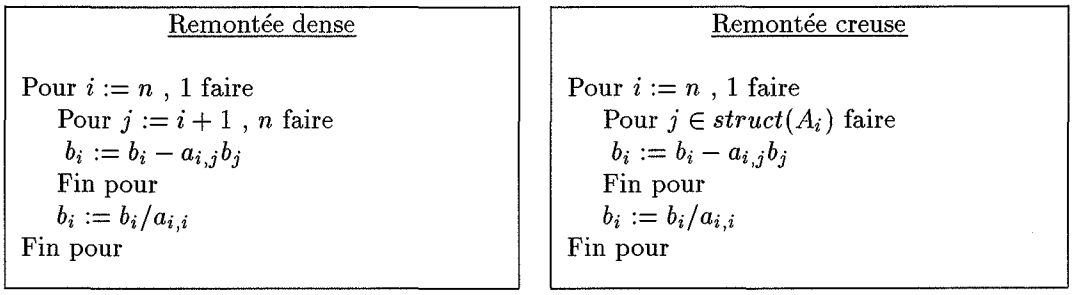

FIG. 4.6 – *Algorithmes de remontée: cas de matrices symétriques denses et creuses.*

Notons que $struct(A_i)$ est donné par l'expression (4.8). Dans le module *DLANCB*, la résolution de systèmes triangulaires intervient avec le calcul de $LDL^T x = b$ avec x et b des des vecteurs de taille n. Le nombre d'opérations scalaires qui sont exécutées est égal à (voir section 4.2.6.1):

$$4nz - 3n \tag{4.23}$$

4.1.3.6 La phase de post traitement

Les post-traitements sont basés sur le calcul des vecteurs propres, pour les valeurs propres ayant convergées et les tests de convergence entre les éléments propres calculés numériquement et théoriques. Cette partie de *DLANCB* va au delà de nos objectifs de parallélisation. De plus, la non disponibilité des programmes sources ne permet pas d'évaluer la complexité de cette phase. Aussi, nous rappelons simplement les opérations qui y sont exécutées.

Dans cette phase, on procède d'abord, au calcul des vecteurs propres de la matrice T_i de la figure (4.3), à partir des valeurs propres convergentes en appliquant la méthode de la puissance inverse. En notant l, le nombre de vecteurs propres, X, la matrice rectangulaire dont les colonnes sont les vecteurs propres de T_i, et Q, la matrice dont les colonnes sont les vecteurs forment une base orthonormale générée par les itérations de *Lanczos*, les vecteurs propres du problème généralisé sont obtenus par le produit matriciel:

$$Y = QX \tag{4.24}$$

En suite, étant donné Y, il convient de vérifier que:

- ⋆ $Y^T KY$ est une matrice diagonale si les colonnes de Y sont des vecteurs propres du problème spectral généralisé. Il s'agit donc d'une relation qu'il convient de vérifier en pratique;
- ⋆ si on note Y_i une colonne de Y correspondant à la valeur propre λ_i, alors le vecteur résiduel, noté F_i, et défini par:

$$F_i = KY_i - \lambda_i MY_i \tag{4.25}$$

est nul.

4.2 Parallélisation de DLANCB

Nous exposons dans cette section, la stratégie que nous proposons pour le portage progressif de *DLANCB* sur des machines parallèle de type *MIMD* à mémoire distribuée. Elle est basée sur la notion d'architecture logicielle *client/serveur*. Avant de discuter de cette approche, il nous a sembler judicieux de présenter quelques travaux sur la parallélisation des opérations matricielles telles que l'élimination de *Gauss*, des produits matrices vecteurs et des résolutions de systèmes linéaires.

4.2.1 Autres travaux de parallélisation

La parallélisation de l'élimination de *Gauss* est très étudiée dans la littérature [HNP91, GKK94], et sert de cadre à bon nombre d'approches de parallélisation. Fondamentalement, nous distinguons deux classes de méthodes pour les machines *MIMD* à mémoire distribuée:

1: les méthodes fondées sur le partitionnement du graphe d'adjacence [GHLN88];

2: les méthodes *multifrontales* [Liu92, DR83].

Dans les méthodes de la classe (1), le parallélisme dépend de la répartition des termes de la matrice à factoriser. Une analyse du graphe d'adjacence permet non seulement de construire

une factorisation symbolique, mais aussi la production d'une organisation du graphe d'adjacence de la matrice dans lequel les sommets sont des blocs de lignes ou des blocs de colonnes. L'idée est de pouvoir traiter les sommets en parallèle. Malheureusement, ce niveau de parallélisme est insuffisant et on a besoin d'introduire à nouveau du parallélisme dans le traitement en parallèle des sommets. Pour ce faire on utilise des découpage $2D$ de la matrice en sous-matrices qui sont ensuite placées sur des processeurs en vertu de quelques heuristiques [Rot94, RG93].

Dans la deuxième classe de méthodes, chaque étape de la factorisation est effectuée comme une suite de factorisations appliquées à des matrices denses de petite taille [Liu92, DR83]. La production de ces sous-matrices se fait au regard de l'arbre d'élimination de la factorisation et de façon récursive. L'intérêt majeur des méthodes multifrontales se situe dans la manipulation de sous-matrices denses pour lesquelles on peut utiliser les divers niveaux des bibliothèques *BLAS*. La structure d'arbre d'élimination est généralisée à des blocs de lignes et est dérivée d'un graphe d'adjacence. Le problème qui se pose est celui du placement efficace des sommets de l'arbre.

Ces deux classes de méthodes induisent des prétraitements dans la mesure où le parallélisme du graphe d'adjacence ou de l'arbre d'élimination est souvent insuffisant. En particulier, pour des matrices stockées en mode skyline (voir [Pel95]), il est difficile de trouver de bonnes heuristiques offrant un parallélisme intéressant, sans une profonde modification du profil de la matrice. Une telle analyse n'est alors intéressante que si la factorisation est effectuée plusieurs fois. Pour le code de calcul que nous nous proposons de paralléliser, on effectue la factorisation un fois pour toute. Il semble donc inadéquat, d'envisager de telles stratégies de parallélisation.

4.2.2 Le modèle client serveur pour DLANCB

La parallélisation de *DLANCB* peut se ramener à une boucle de portage de code (voir figure (2.6) du chapitre 2). Dans ce cadre, nous proposons de modifier le code source de *DLANCB* de façon à en faire, un programme "frontal" alimentant en données, un programme s'exécutant en mode *SPMD*. Le fonctionnement de cet ensemble est illustré dans la figure (4.7) et la nouvelle structure de *DLANCB* dans la figure (4.8). Dans ce qui suit, nous décrivons les rôles du client et du serveur.

Le client:

il exécute sur une machine séquentielle (ou sur un processeur d'une machine parallèle), une version modifiée du programme de *DLANCB* existant dans *SYSTUS*. Les matrices et les vecteurs étant construits au niveau du programme client, les phases de partitionnement, d'ordonnancement et d'allocation sont centralisées, et leur exécution est réalisée via les sous-programmes SE_CONF et SE_RECONF. Puisque la parallélisation s'articule au tour des méthodes numériques à matrices creuses de *DLANCB*, le client fait sous-traiter leur exécution par le serveur.

Par exemple, pour effectuer le factorisation LDL^T, le client exécute le sous-programme *CL_XDGAUS*. Ce sous-programme permet de diffuser un signal demandant au serveur de démarrer la factorisation en parallèle. Lorsque la factorisation est terminée, l'exécution du sous-programme SE_RECONF, permet de réadapter les rangements de données en parallèle pour les traitements ultérieurs.

Afin de calculer les valeurs propres, le client exécute le sous-programme *CL_DLBITE*. Ce

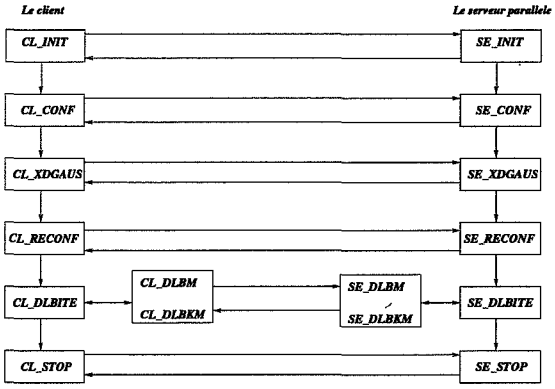

FIG. 4.7 – Mode de fonctionnement du module DLANB en parallèle

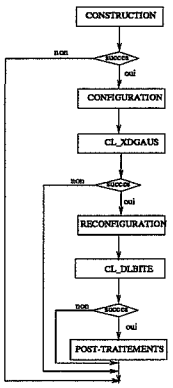

FIG. 4.8 – Exemple de modification du code source de DLANCB.

sous-programme est une version de *DLBITE* dans laquelle nous avons remplacé les appels des sous-programmes *DLBM* et *DLBKM* respectivement par *CL_DLBM* et *CL_DLBKM*. Ces deux derniers sous-programmes sont exécutés de façon répétitives et produisent à chaque exécution, des vecteurs ou des matrices rectangulaires. Grâce aux rangements de données réalisés par SE_RECONF, on limite les échanges de données entre le client et le serveur, à la distribution et à la récupération des vecteurs ou des matrices rectangulaires.

Le serveur:

il s'exécute en mode *SPMD* sur une machine parallèle en utilisant des communications à passage de message. Ce serveur exécute en parallèle des opérations à matrices symétriques stockées en mode skyline telles que: $A = LDL^T$ (*SE_XDGAUS*), $y = Ax$ (*SE_DLBM*), $LDL^Tx = b$ (*SE_DLBKM*) à la demande d'un programme client. Pour ce faire, il doit être configuré grâce aux procédures SE_CONF et SE_RECONF. Dans ce qui suit, nous appliquons l'approche de parallélisation définie au chapitre 3, aux méthodes numériques à matrices creuses de *DLANCB*. L'application de cette approche de parallélisation induit un fonctionnement du type *client/serveur*.

4.2.3 Les serveurs parallèles de DLANCB

On considère une méthode numérique dont les matrices peuvent être creuses ou denses. Les étapes successives de l'approche du chapitre 3 sont les suivantes:

Etape 1 : réécriture d'un schéma d'exécution séquentiel d'une application donnée sous la forme d'un schéma d'exécution systolique,

Etape 2 : partitionnement par blocs d'un graphe d'exécution systolique, en un graphe qui lui est "similaire",

Etape 2.1 : élimination dans le graphe construit en 2, de tous les sommets dont la contribution est nulle,

Etape 3 : ordonnancement en temps minimum du graphe obtenu à l'étape 2.1,

Etape 4 : allocation des sommets du graphe obtenu à l'étape 2.1 en préservant "au mieux", l'ordonnancement en temps minimum de l'étape 3.

Pour illustrer notre propos dans le cadre des matrices creuses, nous considérons les matrices de la figure (4.9). Il s'agit de deux matrices creuses dont les stockages sont décrits par les structures de donnée *ptr* et *adr* des figures (4.10) et (4.11).

$$A = \begin{pmatrix} x & 0 & 0 & 0 & x & 0 \\ 0 & x & x & 0 & 0 & 0 \\ 0 & x & x & x & 0 & x \\ 0 & 0 & x & x & 0 & 0 \\ x & 0 & 0 & 0 & x & x \\ 0 & 0 & x & 0 & x & x \end{pmatrix} \qquad L^T = \begin{pmatrix} x & 0 & 0 & 0 & x & 0 \\ & x & x & 0 & 0 & 0 \\ & & x & x & 0 & x \\ & & & x & 0 & x \\ & & & & x & x \\ & & & & & x \end{pmatrix}$$

FIG. 4.9 – *Profil de A et de L^T*

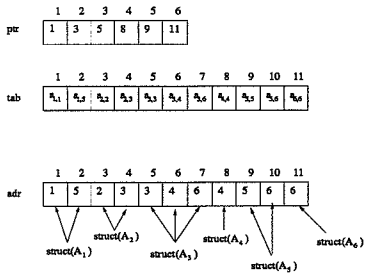

FIG. 4.10 – Stockage de la partie triangulaire supérieure d'une matrice creuse.

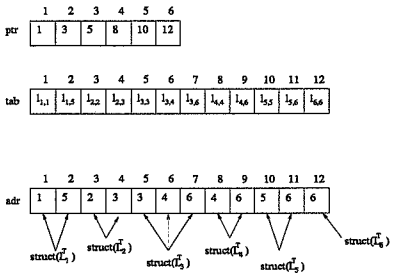

FIG. 4.11 – Stockage de la partie triangulaire supérieure de la matrice L^T.

4.2.3.1 Schémas systoliques de référence

L'étude des schémas systoliques de référence se concentre sur leur modélisation à partir de *SERU* (cas dense) ou de *SERUR* (cas creux). Les dépendances entre ces équations récurrentes se modélisent par une graphe de dépendance, soit $G = (D, E)$. Dans le cas où les matrices sont supposées pleines, on obtient des *SERU* via la procédure d'uniformisation utilisée dans [BPST96]. Dans le cas où les matrices sont creuses, il suffit d'appliquer la procédure de relaxation de la section 3.5.1.

Notons que pour les matrices symétriques de *DLANCB*, on peut se permettre d'éliminer toutes les équations récurrentes à contribution nulle sans augmenter le nombre de directions de dépendance. En effet, la construction de ces matrices est telle que les termes diagonaux

sont non nuls. Or, comme nous le verrons dans ce qui suit, seule leur suppression induit l'apparition de nouvelles directions de dépendance.

Exemple 2 (Le produit matrice symétrique vecteur)

D'après les algorithmes de la figure (4.4), on peut définir le système d'équations de récurrence (4.26). Notons que ces équations ont été déduites de la version liée aux matrices creuses. Pour les matrices pleines, il suffit de remplacer, à i fixé, struct(A_i) par l'intervalle $[i+1, n]$.

$$\begin{cases} y_i^{(0)} = 0 & 1 \leq i \leq n \\ y_i^{(i)} = y_i^{(i-1)} - a_{i,i}x_i & 1 \leq i \leq n \\ y_i^{(j)} = y_i^{(j-1)} - a_{i,j}x_j & 1 \leq i \leq n \text{ et } j \in struct(A_i) \\ y_j^{(i)} = y_j^{(i-1)} - a_{i,j}x_i & 1 \leq i \leq n \text{ et } j \in struct(A_i) \end{cases} \tag{4.26}$$

Par conséquent, le domaine de définition des calculs, noté D est défini par:

$$D = \{(i,j) \in \mathbb{Z}^2;\ 1 \leq i \leq n \text{ et } j \in struct(A_i) \cup \{i\}\} \tag{4.27}$$

Dans la figure (4.12), nous illustrons le graphe de dépendance d'un produit d'une matrice symétrique et creuse (la matrice A de la figure (4.9)), par un vecteur. Les sommets et les arcs en pointillé indiquent la présence d'équations récurrentes à contribution nulle. Tant que ces sommets en pointillé ne sont pas des cercles, les directions de dépendance des *SERUR* sont identiques à celles des *SERU*.

Dans les figures (4.35) et (4.36), on illustre des graphes modélisant respectivement les produit matrice vecteurs pour matrices 6×6 pleine et skyline. Dans toutes ces figures, nous avons utilisé les notations suivantes:

- ⋆ un cercle est un point de D de type (i,i). Il est dédié au calcul de $y_i^{(i)}$. Il reçoit $(y_i^{(i-1)}, x_i)$ dans la direction $(0,1)$. Après avoir calculé $y_i^{(i)} = y_i^{(i-1)} - a_{i,i}x_i$, il envoie $(y_i^{(i)}, x_i)$ dans la direction $(1,0)$,
- ⋆ un carré est un point de D de la forme (i,j), $j \in struct(A_i)$. Il est destiné au calcul de $y_i^{(j)}$ et $y_j^{(i)}$. Il reçoit $(y_j^{(i-1)}, x_j)$ dans la direction $(0,1)$ et $(y_i^{(j-1)}, x_i)$ dans la direction $(1,0)$. Après le calcul de $y_i^{(j)} = y_i^{(j-1)} - a_{i,j}x_j$ et de $y_j^{(i)} = y_j^{(i-1)} - a_{i,j}x_i$, il envoie $(y_j^{(i)}, x_j)$ dans la direction $(0,1)$ et $(y_i^{(j)}, x_i)$ dans la direction $(1,0)$.

Exemple 3 (La descente)

Les algorithmes de la figure (4.5) ont un comportement analogue, tout au moins leurs itérations, à ceux de la figure (4.4). Pour s'en convaincre, considérons une représentation de calculs de ces algorithmes de la figure (4.5) sous forme d'équations de récurrence. Soit:

$$\begin{cases} b_i^{(0)} = b_i & 1 \leq i \leq n \\ b_i^{(i)} = b_i^{(i-1)}/a_{i,i} & 1 \leq i \leq n \\ b_j^{(i)} = b_j^{(i-1)} - a_{i,j}b_i^{(i)} & 1 \leq i \leq n \text{ et } j \in struct(A_i) \end{cases} \tag{4.28}$$

Par conséquent, D est défini par:

$$D = \{(i,j) \in \mathbb{Z}^2;\ 1 \leq i \leq n \text{ et } j \in struct(A_i)\} \tag{4.29}$$

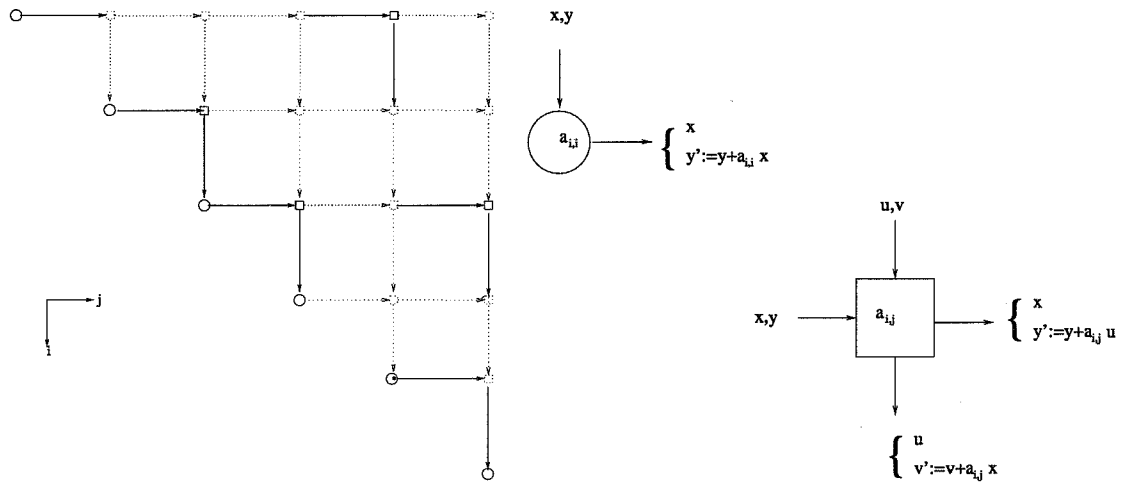

FIG. 4.12 – Graphe de dépendance et cellules élémentaires associés à la factorisation LDL^T. La matrice étant creuse, les carrés et les cercles en pointillé représentent des calculs à contribution nulle.

La figure (4.13) illustre le graphe de dépendance de la descente, où la matrice creuse utilisée est la matrice A de la figure (4.9). Dans cette figure, nous avons utilisé les notations suivantes:

- ⋆ Un cercle est un point de D de type (i,i). Il est dédié au calcul de $b_i^{(i)}$. Il reçoit $b_i^{(i-1)}$ dans la direction $(0,1)$. Après avoir calculé $b_i^{(i)} = b_i^{(i-1)}/a_{i,i}$, il envoie $b_i^{(i)}$ dans la direction $(1,0)$.
- ⋆ Un carré est un point de D de la forme (i,j), $j \in struct(A_i)$. Il est destiné au calcul de $b_j^{(i)}$. Il reçoit $b_j^{(i-1)}$ dans la direction $(0,1)$ et $b_i^{(i)}$ dans la direction $(1,0)$. Après le calcul de $b_j^{(i)} = b_j^{(i-1)} - a_{i,j}b_i^{(i)}$, il envoie $b_j^{(i)}$ dans la direction $(0,1)$ et $b_i^{(i)}$ dans la direction $(1,0)$.

Ainsi, la représentation du graphe de dépendance d'une descente est identique à celle du produit matrice vecteur telle que défini dans l'exemple 2. La parallélisation de la descente procède de la même façon que celle du produit matrice vecteur.

Exemple 4 (La remontée)
Les algorithmes de la figure (4.6) permettent de construire le système d'équations de récurrence suivant:

$$\begin{cases} b_i^{(n+1)} = b_i & 1 \le i \le n \\ b_i^{(j)} = b_i^{(j+1)} - a_{i,j}b_j^{(j)} & 1 \le i \le n \text{ et } j \in struct(A_i) \\ b_i^{(i)} = b_i^{(i+1)}/a_{i,i} & 1 \le i \le n \end{cases} \tag{4.30}$$

Par conséquent, D est défini par:

$$D = \{(i,j) \in \mathbb{Z}^2;\ 1 \le i \le n \text{ et } j \in struct(A_i)\} \tag{4.31}$$

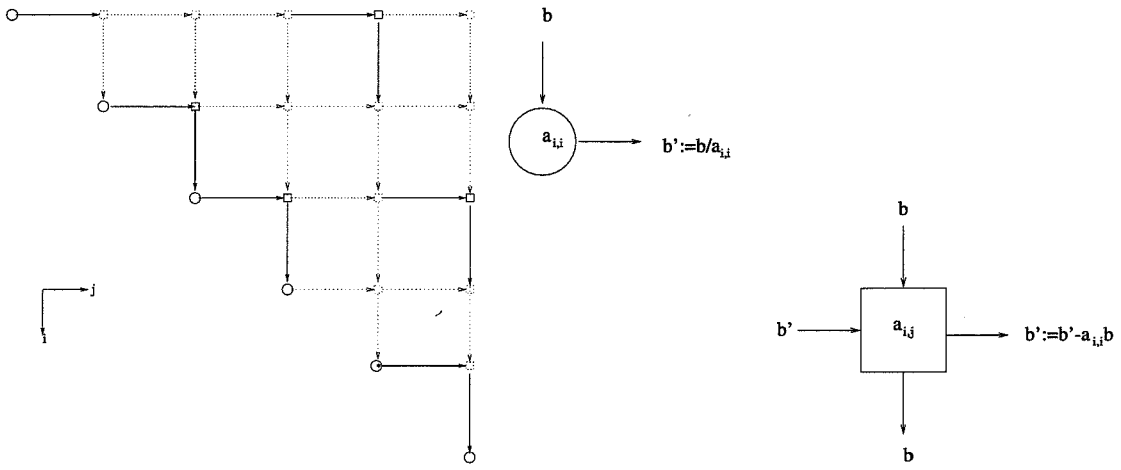

FIG. 4.13 – Graphe de dépendance et cellules élémentaires associés à la descente. La matrice étant creuse, les carrés et les cercles en pointillé représentent des calculs à contribution nulle.

Remarquons que compte tenu de l'organisation des calculs dans les algorithmes de la figure (4.6), il suffit d'effectuer, pour chaque ligne, les calculs par ordre d'indices colonne décroissants pour introduire un parallélisme lors du traitement des lignes consécutives.

La figure (4.14) illustre le graphe de dépendance pour la matrice creuse A de la figure (4.9). Dans cette figure, nous avons utilisé les notations suivantes:

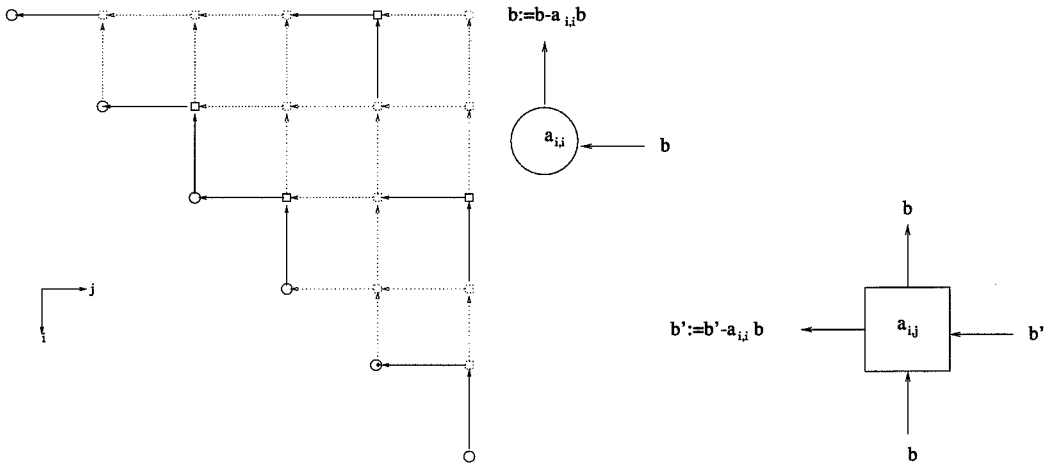

FIG. 4.14 – Graphe de dépendance et cellules élémentaires associés à la remontée. La matrice étant creuse, les carrés et les cercles en pointillé représentent des calculs à contribution nulle.

⋆ Un cercle est un point de D de type (i,i). Il est dédié au calcul de $b_i^{(i)}$. Il reçoit $b_i^{(i+1)}$ dans la direction $(1,0)$. Après avoir calculé $b_i^{(i)} = b_i^{(i+1)}/a_{i,i}$, il envoie $b_i^{(i)}$ dans la direction

$(0,1)$.

⋆ Un carré est un point de D de la forme (i,j), $j \in struct(A_i)$. Il est destiné au calcul de $b_i^{(j)}$. Il reçoit $b_j^{(j)}$ dans la direction $(0,1)$ et $b_i^{(j+1)}$ dans la direction $(1,0)$. Après le calcul de $b_i^{(j)} = b_i^{(j+1)} - a_{i,j} b_j^{(j)}$, il envoie $b_j^{(j)}$ dans la direction $(0,1)$ et $b_i^{(j)}$ dans la direction $(1,0)$.

Dans cette structure de graphe, il est clair que l'ensemble des sommets du graphe G est identique à celui du graphe d'une opération de descente $A^T x = b$. En outre, les dépendances de données portent sur les mêmes sommets. Bien que le sens des vecteurs de dépendance soit opposé à celui d'une opération de descente, l'étude de parallélisation procède de façon identique.

Exemple 5 (La factorisation LDL^T)
En partant des formulations des algorithmes de la figure (4.2), on définit le système d'équations de récurrence (4.32). Dans ces équations, les notations choisies sont celles associées aux matrices creuses. Pour les matrices pleines, il suffit de remplacer les ensembles $struct(L_k^T)$ par les intervalles discrets $[k+1, n]$.

$$\begin{cases} a_{i,j}^{(0)} = a_{i,j} & 1 \le i \le n \text{ et } j \in struct(L_i^T) \cup \{i\} \\ a_{i,i}^{(i)} = 1/a_{i,i}^{(i-1)} & 1 \le i \le n \\ a_{i,j}^{(i)} = a_{i,j}^{(i-1)}/a_{i,i}^{(i-1)} & 1 \le i \le n \text{ et } j \in struct(L_i^T) \\ a_{i,j}^{(k)} = a_{i,j}^{(k-1)} - a_{k,i}^{(k-1)} * a_{k,j}^{(k-1)}/a_{k,k}^{(k-1)} & 1 \le k \le n \text{ et } i,j \in struct(L_k^T) \end{cases} \tag{4.32}$$

On peut donc définir le domaine des calculs scalaires, D, par:

$$D = \bigcup_{k=1}^{n} \{(i,j,k) \in \mathbb{Z}^3;\ i,j \in struct(L_k^T) \cup \{k\} \text{ et } i \le j\} \tag{4.33}$$

La figure (4.15) illustre le graphe de dépendance pour la matrice creuse L^T de la figure (4.9). Le lecteur trouvera dans les figures (4.37) et (4.38), des illustrations de graphes de dépendance de matrices pleine et skyline respectivement. Dans ces figures, nous avons utilisé les notations suivantes:

⋆ Le cercle noir est un point de D de type (i,i,i). Il représente le calcul de la valeur $d_{i,i}^{-1} = a_{i,i}^{(k)}$. Il reçoit la valeur $a_{i,i}^{(i-1)}$ dans la direction $(0,0,1)$. Après avoir calculé $a_{i,i}^{(i)} = 1/a_{i,i}^{(i-1)}$, il envoie $a_{i,i}^{(i-1)}$ dans la direction $(1,0,0)$.

⋆ Un cercle blanc est un point de D de type (i,i,k) tel que $i \in struct(L_k^T)$. Il est dédié au calcul de $a_{i,i}^{(k)}$. Il reçoit $a_{i,i}^{(k-1)}$ dans la direction $(0,0,1)$ et $(a_{k,k}^{(k-1)}, a_{k,i}^{(k-1)})$ dans la direction $(0,1,0)$. Après avoir calculé $a_{i,i}^{(k)} = a_{i,i}^{(k-1)} - a_{k,i}^{(k-1)} * a_{k,i}^{(k-1)}/a_{k,k}^{(k-1)}$, il envoie $(a_{k,k}^{(k-1)}, a_{k,i}^{(k-1)})$ dans la direction $(1,0,0)$ et $a_{i,i}^{(k)}$ dans la direction $(0,0,1)$.

⋆ Un carré noir, est un point de D dont les coordonnées sont de la forme (i,j,i) où $j \in struct(L_i^T)$. Il est destiné au calcul de $l_{i,j}$. Il reçoit la valeur $a_{i,j}^{(i-1)}$ dans la direction $(0,0,1)$ et la valeur $a_{i,i}^{(i-1)}$ dans la direction $(1,0,0)$. Une fois le calcul $a_{i,j}^{(i)} = a_{i,j}^{(i-1)}/a_{i,i}^{(i-1)}$

effectué, il envoie $a_{i,i}^{(i-1)}$ dans la direction $(1,0,0)$, $a_{i,j}^{(i)}$ dans la direction $(0,0,1)$, et $(a_{i,i}^{(i)}, a_{i,j}^{(i-1)})$ dans la direction $(0,1,0)$.

⋆ Un carré blanc est un point de D de la forme (i,j,k), $i,j \in struct(L_k^T)$ et $i \leq j$. Il est destiné au calcul de $a_{i,j}^{(k)}$. Il reçoit $a_{i,j}^{(k-1)}$ dans la direction $(0,0,1)$, $(a_{k,k}^{(k-1)}, a_{k,i}^{(k-1)})$ dans la direction $(1,0,0)$ et $(a_{k,k}^{(k-1)}, a_{k,j}^{(k-1)})$ dans la direction $(0,1,0)$. Après le calcul de $a_{i,j}^{(k)} = a_{i,j}^{(k-1)} - a_{k,i}^{(k-1)} * a_{k,j}^{(k-1)} / a_{k,k}^{(k-1)}$, il envoie $(a_{k,k}^{(k-1)}, a_{k,i}^{(k-1)})$ dans la direction $(1,0,0)$, $a_{i,j}^{(k)}$ dans la direction $(0,0,1)$ et $(a_{k,k}^{(k-1)}, a_{k,j}^{(k-1)})$ dans la direction $(0,1,0)$.

Enfin, pour des besoins de lisibilité du graphe G, nous avons représenté les calculs à contribution nulle par des carrés ou des cercles en pointillé.

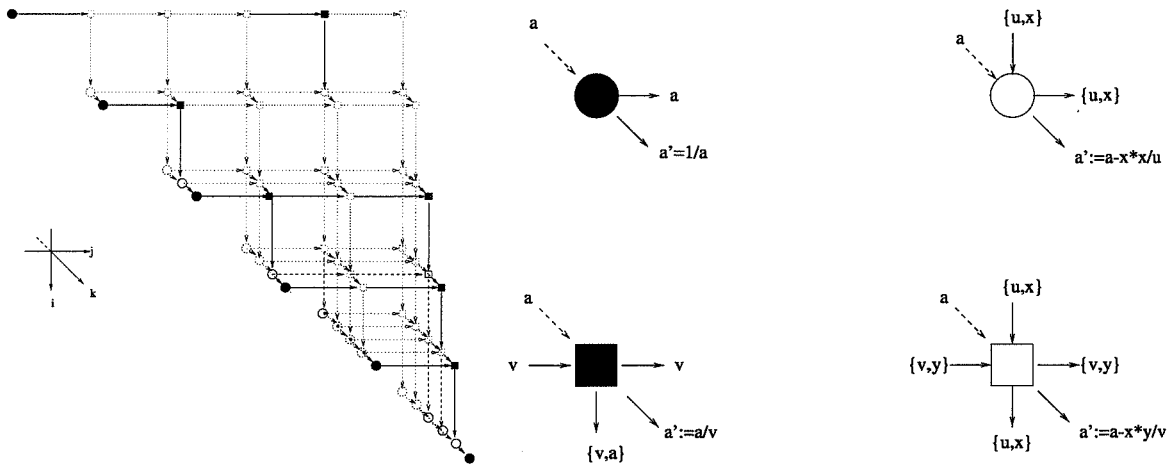

FIG. 4.15 – Graphe de dépendance et cellules élémentaires associés à la factorisation LDL^T. La matrice étant creuse, les carrés et les cercles en pointillé représentent des calculs non exécutés.

Synthèse des dépendances

Dans *DLANCB*, les matrices skylines ont toutes le même profil. Nous présentons formellement la synthèse des dépendances dans le cas général d'une matrice creuse. Les dépendances n'étant plus uniformes, une exécution symbolique des méthodes numériques est nécessaire pour construire la structure du graphe $G = (D, E)$. La synthèse automatique des dépendances peut se faire par listes d'adjacences. Celles-ci permettent d'associer à tout point $u \in D$, la liste des points $v \in D$, tels que (u,v) ou (v,u) sont des arcs de G. Pour le produit matriciel et les résolutions des systèmes triangulaires, les dépendances à construire sont dans les directions $(1,0)$ et $(0,1)$, quant à la factorisation LDL^T, ses dépendances sont dans les directions $(1,0,0)$, $(0,1,0)$ et $(0,0,1)$.

Nous proposons une approche permettant de construire dans le cadre d'une même structure, appelée *matrice de cellules*, les graphes des opérations matricielles précédemment citées. Le

rangement des cellules dans une telle matrice procède de la même façon que celui des matrices creuses de *DLANCB*. Une *cellule* est caractérisée par un couple (i,j) et est associée au coefficient $a_{i,j}$ de la matrice A. En outre, elle doit contenir des informations permettant de générer les dépendances de la factorisation $A = LDL^T$, du produit matrice vecteur $y = Ax$ et de la résolution du système $LDL^T x = b$. Pour compléter ces informations, nous considérons d'abord la construction des dépendances de la factorisation LDL^T qui procède en deux étapes:

1– Construction de toutes les dépendances dans les directions $(1,0,0)$ et $(0,1,0)$ associées à la k-ème itération des algorithmes de la figure (4.2),

2– construction pour chaque (i,j), une liste de points (i,j,k) représentant les transformations successives, soit $a_{i,j}^{(k)}$, associées au terme $a_{i,j}$.

L'algorithme de la figure (4.16) illustre la mise en oeuvre de ces deux étapes. Dans cet algorithme, lorsque l'on ajoute (i,j,k) à la liste des transformations successives de $a_{i,j}$, si (i,j,k') est le point précédemment ajouté à une telle liste, on crée naturellement l'arc $((i,j,k'),(i,j,k))$. La synthèse automatique des dépendances induit une extension du stockage de la matrice creuse.

Pour $k = 1,\ldots,n$
 Pour $i \in struct(L_k^T) \cup \{k\}$
  Pour $j \in struct(L_k^T) \cup \{k\}$ et $j \geq i$
   Arcs dans $(1,0,0)$: $((i, pred_k(j), k),(i,j,k))$ et $((i,j,k),(i, succ_k(j),k))$
   Arcs dans $(0,1,0)$: $((pred_k(i), j, k),(i,j,k))$ et $((i,j,k),(succ_k(i),j,k))$
   Ajouter (i,j,k) à la liste des transformations successives de $a_{i,j}$
  Fin pour
 Fin pour
Fin pour

FIG. 4.16 – *Mise en oeuvre du graphe G.*

Notons que pour chaque k, avec $1 \leq k \leq n$, les indices représentés dans $struct(L_k^T)$ sont rangés par ordre croissant. L'expression $pred_k(j)$, désigne le prédécesseur de j dans la liste ordonnée $struct(L_k^T)$. Si j est le plus petit élément de cette liste, alors son prédécesseur vaut k et k n'admet pas de prédécesseur. De la même manière, $succ_k(j)$ désigne l'indice suivant j dans la liste ordonnée $struct(L_k^T)$. Si j est le plus grand élément de $struct(L_k^T)$, il n'admet pas de suivant. Le suivant de k est le plus petit élément de $struct(L_k^T)$. Lorsqu'un prédécesseur ou un successeur n'est pas défini pour un indice, le point correspondant n'existe pas. Par conséquent, on ne construit pas d'arcs correspondant à une telle situation.

Pour le produit matrice vecteur et la résolution de systèmes triangulaires, les dépendances se déduisent de celles générées par l'algorithme de la figure (4.16). En effet, les prédécesseurs ou les successeurs d'un points (i,j) s'obtiennent dans la direction $(1,0)$ pour $k = i$, et dans

la direction $(0,1)$ pour $k = j$. La matrice de cellule ainsi obtenue, sera complétée lors de l'ordonnancement et de l'allocation. Toutefois, une telle représentation est prohibitive si on considère un stockage point à point des matrices. Elle sera appliquée aux matrices stockées par blocs qui résulteront du partitionnement par blocs.

4.2.3.2 Partitionnement par blocs

Le point de départ du partitionnement par bloc est un graphe de dépendance d'un des schémas d'exécution systolique exposés dans la section 4.2.3.1, dont les notations sont entièrement reprises dans ce qui suit. Dans la mesure où les graphes définis dans la section 4.2.3.1 s'articulent autour d'une matrice carrée, il suffit de considérer le même *quadrillage* dans toutes les dimensions pour produire un graphe similaire. Le quadrillage qui est proposé ici, se fonde sur ceux utilisés dans les techniques de discrétisation du type *différences finies* ou *élément finis* [RT83, LT86b].

Soit la suite de nombres entiers définie par:

$$\mu = (\mu_i)_{1 \leq i \leq N} \tag{4.34}$$

vérifiant

$$n = \sum_{i=1}^{N} \mu_i \tag{4.35}$$

où $N \leq n$. La suite μ permet de définir un quadrillage de D à partir de celui de l'intervalle discret $[1,n]$ de $\mathbb{N}$ à l'aide de $N-1$ points intermédiaires. Soit:

$$1 = u_0 < u_1 < \ldots < u_N = n \tag{4.36}$$

avec

$$u_i = \sum_{j=1}^{i} \mu_j \text{ pour } i > 0 \tag{4.37}$$

On considère les fonctions i_1 et i_2 définies par:

$$\begin{aligned} i_1 \ : \ & \mathbb{Z} \to \mathbb{Z} \\ & i \longmapsto i_1(i) = u_i - \mu_i + 1 \end{aligned} \tag{4.38}$$

et

$$\begin{aligned} i_2 \ : \ & \mathbb{Z} \to \mathbb{Z} \\ & i \longmapsto i_2(i) = u_i \end{aligned} \tag{4.39}$$

Ainsi on a

$$[1,n] = \bigcup_{I=0}^{N} [i_1(I), i_2(I)] \tag{4.40}$$

Un super-noeud sera défini par:

Définition 9 (Super-noeud)

Pour un intervalle $[1,n] \subset \mathbb{Z}$, on considère son partitionnement sous la forme (4.40). Un super-noeud est un sous graphe de G contenu dans une maille rectangulaire B définie par:

- ⋆ *si $D \subset \mathbb{Z}^2$, alors il existe deux nombre I et J tels que $1 \leq I, J \leq N$ et B a pour sommets les points de $\mathbb{Z}^2$ tels que:*

$$(i_1(I), i_1(J)),\ (i_1(I), i_2(J)),\ (i_2(I), i_2(J)) \ , (i_2(I), i_1(J)) \tag{4.41}$$

et les coordonnées du super-noeud sont: (I,J),

⋆ *si* $D \subset \mathbb{Z}^3$, *alors il existe trois nombres* I, J *et* K *tels que* $1 \leq I, J, K \leq N$ *et* B *a pour sommets les points de* $\mathbb{Z}^3$ *tels que:*

$$\begin{array}{ll} (i_1(I), i_1(J), i_1(K)) & , (i_1(I), i_2(J), i_1(K)) \\ (i_2(I), i_2(J), i_1(K)), & (i_2(I), i_1(J), i_1(K)) \\ (i_1(I), i_1(J), i_2(K)), & (i_1(I), i_2(J), i_2(K)) \\ (i_2(I), i_2(J), i_2(K)), & (i_2(I), i_1(J), i_2(K)) \end{array} \tag{4.42}$$

et les coordonnées du super-noeud sont: (I, J, K).

Un tel quadrillage permet de définir un graphe $G(\mu) = (D(\mu), E(\mu))$ similaire à G, où $D(\mu)$ et $E(\mu)$ sont respectivement les ensembles de sommets et d'arcs. La matrice A est alors découpée dans la direction $(0,1)$, en N blocs de lignes consécutives et dans la direction $(1,0)$, en N blocs de colonnes consécutives. Soit $A(\mu) = (A_{I,J}(\mu))_{1 \leq I,J \leq N}$ la matrice par blocs ainsi définie.

En supposant que la matrice utilisée soit dense, tout sommet de $G(\mu)$ est obtenu en réduisant en un seul point tout super-noeud conforme à la définition 9. Ce point aura pour coordonnées les coordonnées du super-noeud ainsi réduit. Un arc de $G(\mu)$ est obtenu en réduisant en un seul arc tous les arcs de G existant entre deux sommets de $G(\mu)$. Les vecteurs de dépendance dans $G(\mu)$ se mesurent par rapport aux coordonnées des super-noeuds.

Dans le cas où la matrice est creuse, le quadrillage est d'abord appliqué en considérant une représentation en terme de matrice dense. Ensuite, dans chaque super-noeud, on procède à une élimination des sommets de G dont les équations récurrentes sont à contribution nulle (les ronds et les carrés en pointillés).

Dans le cas où les super-noeuds ne contiennent plus de sommets de G, nous les appellerons super-noeuds à contribution nulle. Pour obtenir le nouveau graphe de dépendance, il suffit d'éliminer tous les super-noeuds à contribution nulle, et, dans chaque direction de dépendance, garder les connexions entre les super-noeuds, géométriquement les plus proches, par rapport aux coordonnées des super-noeuds. Pour la synthèse automatique de telles dépendances, l'algorithme de la figure (4.16) s'applique formellement. En effet, les graphes G et $G(\mu)$ étant similaires, il suffit de considérer formellement un parcours des indices des sous-matrices de la matrice par blocs $A(\mu)$.

Dans ce qui suit, nous illustrons le quadrillage sur le produit matrice vecteur et la factorisation LDL^T. Nous renvoyons le lecteur intéressé à la section 4.2.6 pour l'analyse de la complexité.

Exemple du produit matrice vecteur

Par construction, on a:

$$D \subset [1, n] \times [1, n]$$

Le quadrillage de D est obtenu par les droites parallèles aux cotés de $[1, n] \times [1, n]$. Soient $x = u_I$ et $y = u_J$, avec $0 \leq I, J \leq N$. On trouvera dans les figures (4.17), (4.19), (4.21) et (4.23) des exemples de quadrillages ainsi définis.

La figure (4.17) illustre le quadrillage du graphe G associé à une matrice pleine de taille 6×6. Il s'agit d'un quadrillage simple de D par pas constant $\mu_i = \frac{n}{N}$. Dans ce cas, $N = 3$ et $\mu_i = 2$. Le quadrillage de la figure (4.17) induit des rangements de matrices par blocs 3×3 et des rangements de vecteurs par blocs illustrés dans la figure (4.18). Pour le même graphe

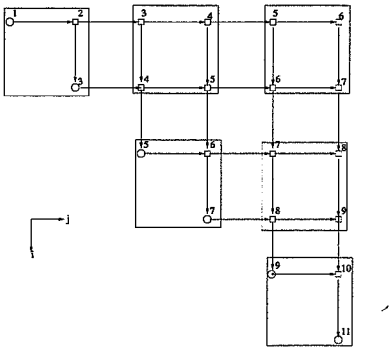

FIG. 4.17 – Quadrillage d'un graphe plein avec $\mu_i = 2$.

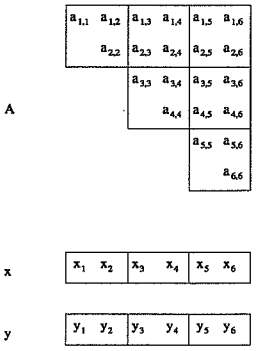

FIG. 4.18 – Quadrillage des données avec $\mu_i = 2$ pour une matrice pleine

de dépendance, la figure (4.19) illustre un quadrillage avec μ_i variable. Le graphe obtenu après partitionnement est similaire à celui d'un produit matrice vecteur dont la matrice est de taille 4×4. La figure (4.20) illustre les stockages par blocs des matrices et des vecteurs qui en résultent.

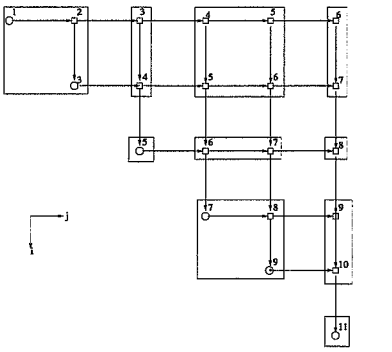

FIG. 4.19 – Quadrillage d'un graphe plein avec μ_i variable.

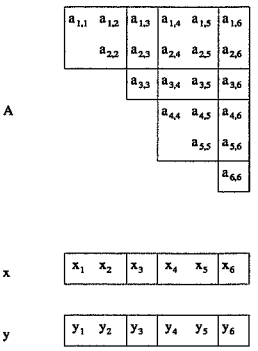

FIG. 4.20 – Quadrillage des données avec μ_i variable pour une matrice pleine

Dans la figure (4.21), nous illustrons un quadrillage avec $\mu_i = 2$ pour le produit matrice vecteur avec la matrice creuses A de la figure (4.9). D'après ce quadrillage, la matrice A utilisée est partitionnée en une matrice bloc 3×3 illustrée dans la figure (4.22). La figure (4.23) illustre un quadrillage, avec μ_i variable, du graphe associé à la matrice creuse de la figure (4.9). Il en résulte un graphe similaire à celui d'une matrice creuse 5×5 par blocs de taille variable (cf figure (4.23)).

Le partitionnement par blocs de G induit celui de la matrice A dans ses deux dimensions et celui des vecteurs x et y. En partant du stockage par blocs des vecteurs et de la matrice ainsi défini, on peut donc caractériser par classes les différents super-noeuds $G(\mu)$. Comme le montrent les figures (4.25) et (4.26), on distingue deux classes de super-noeuds.

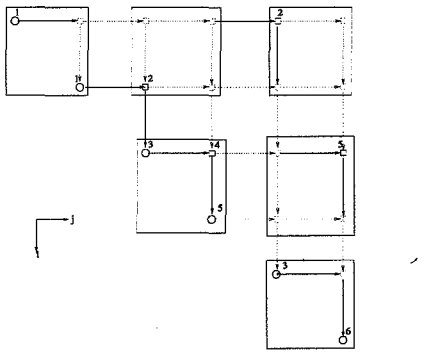

FIG. 4.21 – Quadrillage d'un graphe creux avec $\mu_i = 2$.

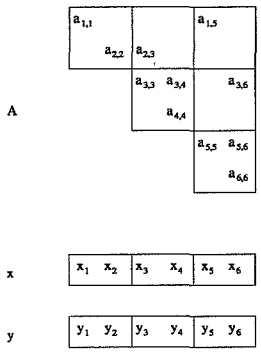

FIG. 4.22 – Quadrillage des données avec $\mu_i = 2$ pour une matrice creuse.

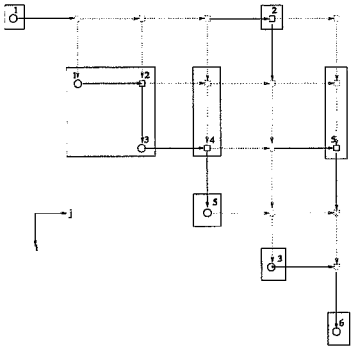

FIG. 4.23 – Quadrillage d'un graphe creux avec μ_i variable.

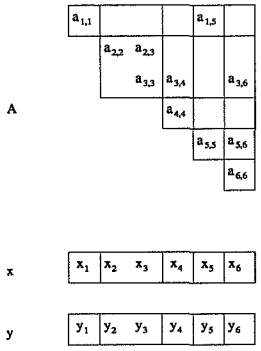

FIG. 4.24 – Quadrillage des données avec μ_i variable pour une matrice creuse.

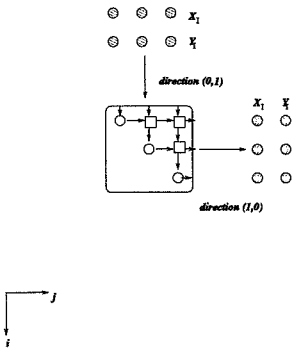

FIG. 4.25 – Super-noeud de type (I, I).

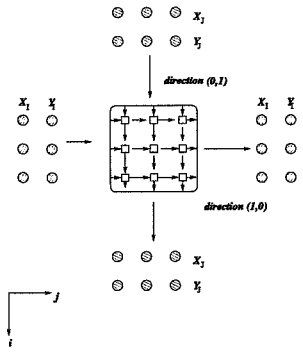

FIG. 4.26 – Super-noeud de type (I, J).

Un super-noeud (I, I), reçoit dans la direction $(0, 1)$ et envoie dans la direction $(1, 0)$. On retrouve donc un fonctionnement semblable à celui des cellules rondes utilisées dans le modèle du graphe G. Toutefois, les données échangées sont des paires de vecteurs de longueur μ_I dont les éléments sont des composantes des vecteurs x et y appartenant à l'intervalle $[i_1(I), i_2(I)]$. Notons ces deux vecteurs par $X_I(\mu)$ et $Y_I(\mu)$ respectivement.

Un super-noeud (I, J) reçoit dans les directions $(0, 1)$ et $(1, 0)$ d'une part, et, d'autre part, envoie dans les directions $(0, 1)$ et $(1, 0)$. On retrouve donc un fonctionnement semblable à celui des cellules carrées utilisées dans le modèle du graphe G. Toutefois, les données échangées sont des paires de vecteurs de longueur μ_I et μ_J dans les directions $(1, 0)$ et $(0, 1)$, respectivement. Pour les sous-vecteurs extraits de x et y, les composantes appartiennent à l'intervalle $[i_1(I), i_2(I)]$ pour la direction $(1, 0)$ et à l'intervalle $[i_1(J), i_2(J)]$ pour la direction $(0, 1)$.

Dans la mise en oeuvre du quadrillage nous n'avons fait aucune hypothèse sur l'orientation des arcs. On peut donc réitérer cette procédure de quadrillage sur des graphes G modélisant une descente ou une remontée.

Exemple de la factorisation LDL^T

Par construction, D vérifie:

$$D \subset [1, n] \times [1, n] \times [1, n]$$

En réutilisant les notations introduites précédemment dans cette section, le quadrillage de D est obtenu par les droites orthogonales aux faces de $[1, n] \times [1, n] \times [1, n]$ et d'équations $x = u_I$, $y = u_J$ et $z = u_K$, avec $1 \leq I, J, K \leq N$. On trouvera dans les figures (4.27) et (4.29) des exemples de tels quadrillages qui permettent de définir un graphe $G(\mu)$ similaire à G.

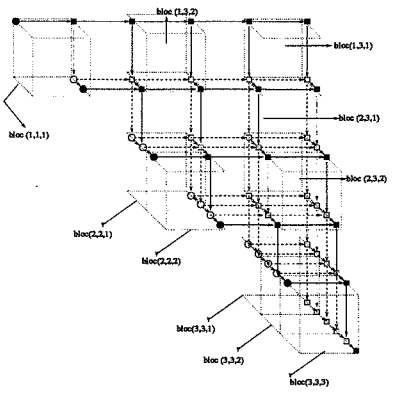

FIG. 4.27 – Quadrillage avec $\mu_I = 2$.

$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$
	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$
		$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$
			$a_{4,4}$	$a_{4,5}$	$a_{4,6}$
				$a_{5,5}$	$a_{5,6}$
					$a_{6,6}$

FIG. 4.28 – Quadrillage d'une matrice pleine avec $\mu_I = 2$

La figure (4.27) illustre le quadrillage du graphe G associé à une matrice pleine de taille 6×6. Il s'agit d'un quadrillage simple de D par pas constant $\mu_I = \frac{n}{N}$. Dans ce cas, $N = 3$ et $\mu_I = 2$. Le quadrillage de la figure (4.27) induit une matrice bloc 3×3 illustrée dans la figure (4.28).

La figure (4.29) illustre un quadrillage, avec des valeurs de μ_I variables, du graphe associé à la matrice creuse L^T de la figure (4.9). Il induit une matrice creuse 4×4 par blocs de taille variable illustrée dans la figure (4.30).

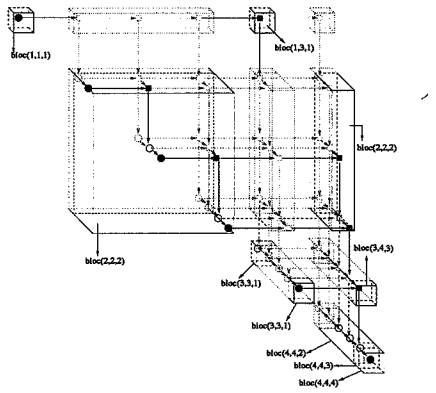

FIG. 4.29 – Quadrillage avec μ_I variable.

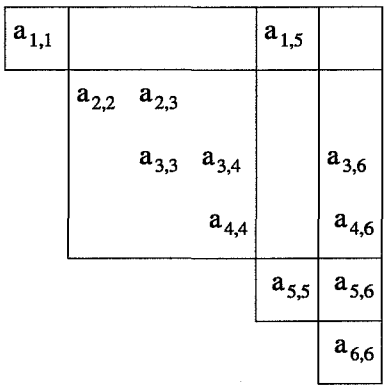

FIG. 4.30 – Quadrillage de la matrice correspondante

En réutilisant la définition des $A_{I,J}(\mu)$ introduite précédemment, on peut caractériser les principales classes de super-noeud (I, J, K) en fonction de la sous-matrice $A_{I,J}(\mu)$. Les figures (4.31), (4.32), (4.33) et (4.34) illustrent les principales classes de super-noeuds. Pour des raisons de lisibilité, nous n'avons pas représenté les données qui circulent dans la direction $(0, 0, 1)$. On peut vérifier qu'il s'agit d'une sous-matrice dont le profil est identique à celui de la sous-matrice $A_{I,J}(\mu)$. Pour toutes les classes de super-noeuds (I, J, K), les communications se résument à des transferts de sous-matrices que nous allons maintenant caractériser. Notons également que ces sous-matrices ne sont pas toujours pleines. D'où la nécessité de les transférer en tant que matrice creuse si l'on désire simuler uniquement, dans chaque super-noeud, le calcul des noeuds de G.

Pour un super-noeud de type (I, I, I) (voir figure (4.31)), on reçoit une matrice dont le profil est identique à celui de $A_{I,I}(\mu)$ dans la direction $(0, 0, 1)$. La sous-matrice produite a également le même profil que $A_{I,I}(\mu)$ et est envoyée dans la direction $(1, 0, 0)$ vers le point de $G(\mu)$ le plus proche. Le super-noeud qui reçoit une telle matrice peut alors constituer les couples $(a_{i,i}^{(i-1)}, a_{i,j}^{(i-1)})$ pour simuler le calcul des noeuds de G qu'il contient. Ainsi, on retrouve un fonctionnement semblable à celui des cellules rondes et noires de la modélisation de G.

Pour un super-noeud de type (I, J, I) (voir figure (4.33)), on reçoit une sous-matrice dont le profil est identique à celui de $A_{I,J}(\mu)$ dans la direction $(0, 0, 1)$; on reçoit une sous-matrice triangulaire dont le profil est identique à celui de $A_{I,I}(\mu)$ dans la direction $(1, 0, 0)$. La sous-matrice produite a également le même profil que $A_{I,J}(\mu)$ et est envoyée dans la direction $(0, 1, 0)$ vers le point de $G(\mu)$ le plus proche. Notons que, pour chacune de ses lignes, on rajoute le coefficient diagonal extrait de la même ligne dans la sous-matrice triangulaire reçue dans la direction $(1, 0, 0)$. Ceci permet au super-noeud qui la reçoit de former le couple

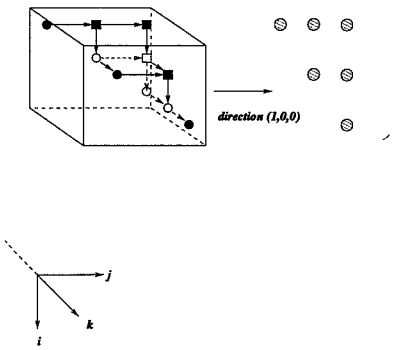

FIG. 4.31 – Super-noeud du type (I, I, I).

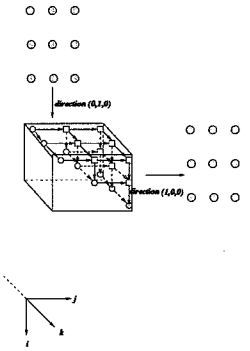

FIG. 4.32 – Super-noeud du type (I, I, K) avec $K < I$.

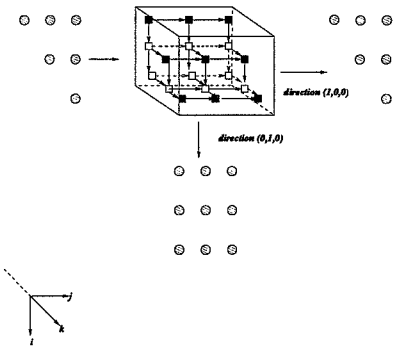

FIG. 4.33 – Super-noeud du type (I, J, I) avec $J > I$.

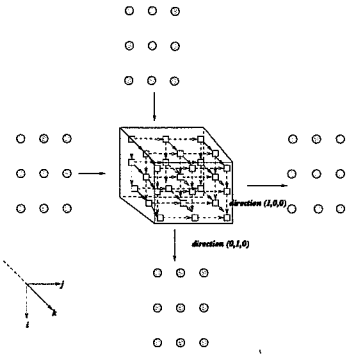

FIG. 4.34 – Super-noeud du type (I, J, K) avec $J > I$ et $K < I$.

$(a_{i,i}^{(i-1)}, a_{i,j}^{(i-1)})$ pour simuler le calcul des noeuds de G qu'il contient. Ainsi, on retrouve un fonctionnement semblable à celui des cellules carrées et noires de la modélisation de G.

Pour un super-noeud de type (I, I, K) (voir figure (4.32)), on reçoit une matrice dont le profil est identique à celui de $A_{I,I}(\mu)$ dans la direction $(0, 0, 1)$ et une sous-matrice dont le profil est identique à celui de la sous-matrice $A_{K,I}(\mu)$ dans la direction $(0, 1, 0)$. La sous-matrice produite a également le même profil que $A_{I,I}(\mu)$ et est envoyée dans la direction $(0, 0, 1)$ vers le point de $G(\mu)$ le plus proche; la sous-matrice reçue dans la direction $(0, 1, 0)$ est envoyée dans la direction $(1, 0, 0)$ au noeud de $G(\mu)$ le plus proche. Ainsi, on retrouve un fonctionnement semblable à celui des cellules rondes et blanches de la modélisation de G.

Pour un super-noeud de type (I, J, K) (voir figure (4.34)), on reçoit une sous-matrice dont le profil est identique à celui de $A_{I,J}(\mu)$ dans la direction $(0, 0, 1)$; on reçoit une sous-matrice dont le profil est identique à celui de $A_{K,I}(\mu)$ dans la direction $(1, 0, 0)$; on reçoit une sous-matrice dont le profil est identique à celui de $A_{K,J}(\mu)$ dans la direction $(0, 1, 0)$. La sous-matrice produite a également le même profil que $A_{I,J}(\mu)$ et est envoyée dans la direction $(0, 0, 1)$ vers le point de $G(\mu)$ le plus proche. Les sous-matrices reçues dans les directions $(1, 0, 0)$ et $(0, 1, 0)$ sont respectivement envoyées dans les mêmes directions. Ainsi, on retrouve un fonctionnement semblable à celui des cellules carrées et blanches de la modélisation de G.

4.2.3.3 Ordonnancement des nouveaux schémas d'exécution

Dans la section 3.6, nous avons établi les bornes supérieure et inférieure du temps d'exécution des algorithmes parallèles après allocation des nouveaux schémas d'exécution. La borne inférieure est égal au poids du graphe $G(\mu)$, au sens défini dans la section 3.3.1, et la borne supérieure dépend de la longueur du plus long chemin de $G(\mu)$. La stratégie d'allocation permettant simplement de préserver la longueur du plus long chemin, il nous semble judicieux de décrire des fonctions de temps compatibles avec une telle allocation, et d'évaluer le nombre minimum de processeurs requis pour que cette compatibilité soit respectée.

Pour évaluer la longueur du plus long chemin de $G(\mu)$, nous proposons de procéder à une étude classique d'un problème d'ordonnancement de systèmes d'équations de récurrence. Les fonctions de temps ainsi définies seront réutilisées lors de l'allocation. Sans nuire à la généralité, nous utilisons les notations relatives aux schémas systoliques de référence de la section 4.2.3.1.

Complexité en termes de temps

Nous avons vu que les cercles et les carrés dans le graphe G n'exécutent pas le même nombre d'opérations scalaires. Notre propos n'est pas de quantifier toutes ces opérations scalaires telles que l'aurait demandé une analyse de complexité rigoureuse. Afin de d'évaluer le plus long chemin dans G, nous admettrons que le traitement de tout sommet, (cercle ou carré) équivaut à une unité de temps. La complexité d'un schéma systolique de référence sera notée T_{sys} et la longueur du plus long chemin vaut $T_{sys} - 1$. Nous construisons maintenant, les fonctions de temps qui atteignent la borne T_{sys}.

Dans le cas où les matrices sont pleines ou skylines, nous avons le lemme suivant:

Lemme 2 (Cas des matrices pleines et skylines)
Pour le produit matrice vecteur de l'exemple (2), la fonction de temps optimale est définie par:

$$T(i, j) = i + j - 1 \tag{4.43}$$

Pour la factorisation LDL^T de l'exemple (5), la fonction de temps optimale est définie par:

$$T(i,j,k) = i + j + k - 2 \tag{4.44}$$

La preuve du lemme 2, procède par des techniques telles que celles développées dans [BR90, BPST96]. Ces techniques exploitent le caractère linéaire des dépendances entre sommets. Pour le produit matrice vecteur et la descente, les arcs du graphe G sont de la forme $((i,j),(i,j+1))$ ou $((i,j),(i+1,j))$. Pour la remontée, les arcs sont de la forme $((i,j+1),(i,j))$ ou $((i+1,j),(i,j))$. Enfin, pour la factorisation LDL^T, les arcs sont de la forme $((i,j,k),(i,j+1,k))$, $((i,j,k),(i+1,j,k))$ ou $((i,j,k),(i,j,k+1))$. Par conséquent, pour les opérations à matrice pleines et skylines décrites dans les exemples (2), (3), (4) et (5), les fonctions de temps optimales sont affines.

Pour le produit matrice vecteur, les figures (4.35) et (4.36), illustrent une fonction de temps affine avec des matrices pleine et skyline respectivement. Il suffit d'un parcours ligne par ligne de la matrice pour déterminer i et j, et partant, calculer explicitement $T(i,j)$.

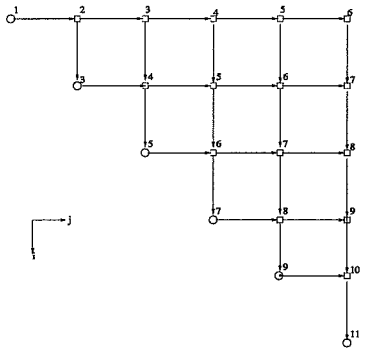

FIG. 4.35 – Fonction de temps affine pour une matrice pleine

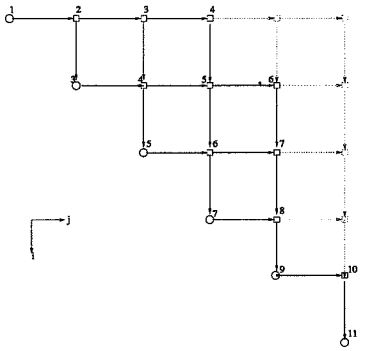

FIG. 4.36 – Fonction de temps affine pour une matrice skyline

Pour la factorisation LDL^T, les figures (4.37) et (4.38), illustrent une fonction de temps affine avec des matrices pleine et skyline respectivement. Notons que, pour chaque coefficient $a_{i,j}$, on peut calculer explicitement $T(i,j,k)$ dans le cas où la matrice est dense. En effet, par construction de la factorisation LDL^T, le nombre k pour le résultat partiel $a_{i,j}^{(k)}$ est défini si $1 \leq k \leq j$. Dans le cas où la matrice est skyline, bien que la fonction de temps soit affine, le calcul de $T(i,j,k)$ n'est plus immédiat. En effet, le nombre de coefficients par ligne étant variable, le nombre k pour le résultat partiel $a_{i,j}^{(k)}$ n'est défini que si $l_j \leq k \leq j$, où l_j est tel que $a_{k,j} = 0$ pour $1 \leq k < l_j$. Un parcours de la matrice est donc nécessaire pour déterminer précisément l_j, pour $j = 1, \cdots, n$.

Dans le cadre de matrices creuses quelconques, les fonctions affines ne sont plus appropriées. Il convient dès lors, de définir des fonctions de temps qui utilisent à fond le caractère combinatoire des graphes de dépendance. Dans la section 3.5.2, nous avons proposé les fonctions de temps au plus tôt et au plus tard pour résoudre le problème de l'ordonnancement. La mise en oeuvre de fonction de temps optimale requiert un parcours de l'ensemble des listes d'adjacence modélisant le graphe G.

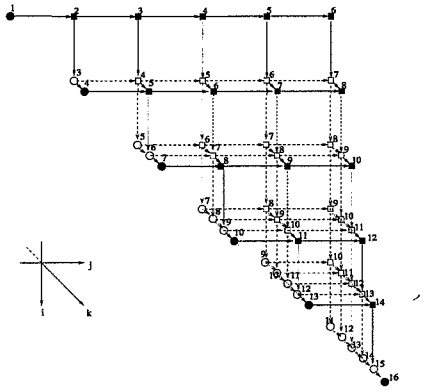

FIG. 4.37 – Fonction de temps affine pour une matrice pleine

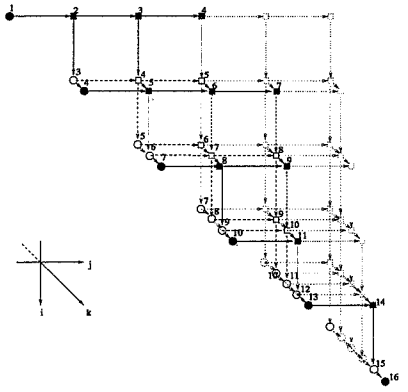

FIG. 4.38 – Fonction de temps affine pour une matrice skyline

Pour le produit matrice vecteur, nous illustrons dans les figures (4.39) et (4.40) les fonctions de temps au plus tôt et au plus tard. Outre un parcours ligne par ligne de la matrice creuse, la construction d'une telle fonction nécessite la connaissance, pour chaque point u, de tous les points v tels que (u, v) ou (v, u) sont des arcs de G. Ainsi, la date au plus tôt du point u se déduit de celle des points v tels que les (v, u) sont des arcs de G. Il suffit dès lors d'un parcours de toutes les lignes, pour $i = 1, \ldots, n$, pour dater tous les points de D. De la même manière, la date au plus tard du point u se déduit de celle des points v tels que les (u, v) sont des arcs de G. Il suffit donc d'un parcours de toutes les lignes, pour $i = n, n-1, \ldots, 1$, pour décrire une telle fonction de temps.

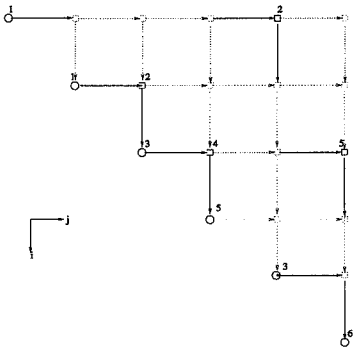

FIG. 4.39 – Fonction de temps au plus tôt

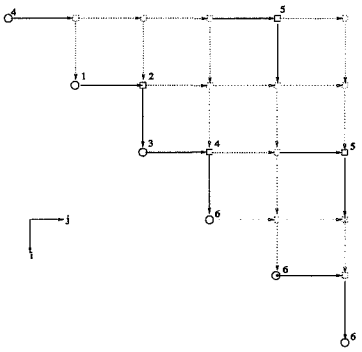

FIG. 4.40 – Fonction de temps au plus tard

Pour l'exemple de la factorisation LDL^T à matrice creuse, nous illustrons dans les figures (4.41) et (4.42) les fonctions de temps au plus tôt et au plus tard. Outre un parcours ligne par ligne de la matrice creuse qui permet de déterminer les indices (i, j), la construction d'une telle fonction nécessite le parcours des listes des transformations successives de $a_{i,j}$ et, pour

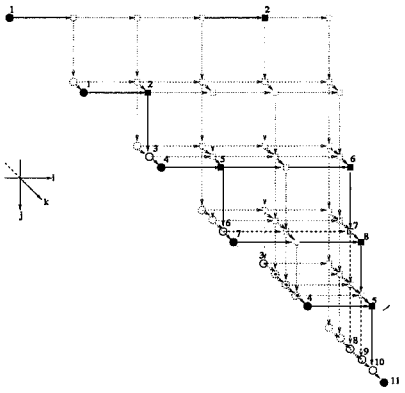

FIG. 4.41 – Fonction de temps au plus tôt

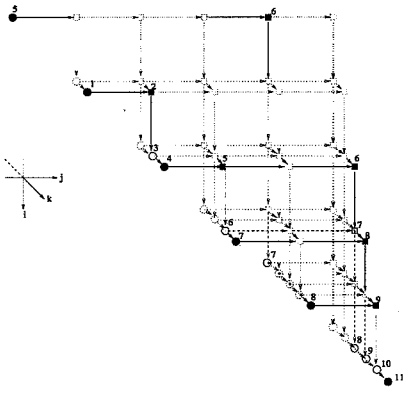

FIG. 4.42 – Fonction de temps au plus tard

chaque point $u = (i, j, k)$, de tous les points v tels que (u, v) ou (v, u) sont des arcs de G. Ainsi, la date au plus tôt du point u se déduit de celles des points v tels que les (v, u) sont des arcs de G. Il suffit dès lors d'un parcours de toutes les lignes, d'après l'algorithme 1 de la figure (4.43), pour créer un ordonnancement au plus tôt des points de D. De la même manière, la date au plus tard du point u se déduit de celle des points v tels que les (u, v) sont des arcs de G. Il suffit donc d'un parcours de toutes les lignes, d'après l'algorithme 2 de la figure (4.43), pour décrire une telle fonction de temps. Notons que la présence de -1, dans l'expression "*Pour* $i \in struct(A_k) \cup \{k\}$, -1 *faire*" signifie un parcours par ordre décroissant de $struct(A_k) \cup \{k\}$.

Algorithme 1

Pour $k := 1$, n faire
 Pour $i \in struct(A_k) \cup \{k\}$ faire
  Pour $j \in struct(A_k) \cup \{k\}$ et $j \geq k$ faire
  $u := (i, j, k)$
  $\mathcal{A} := \{v : (v, u)$ arc de $G\}$
  Si $\mathcal{A} = \emptyset$ alors $T(u) = 1$
  Sinon $T(u) = 1 + \max\{T(v) : v \in \mathcal{A}\}$
  Fin pour
 Fin pour
Fin pour

Algorithme 2

Pour $k := n$, 1, -1 faire
 Pour $i \in struct(A_k) \cup \{k\}$, -1 faire
  Pour $j \in struct(A_k) \cup \{k\}$ et $j \geq k$, -1 faire
  $u := (i, j, k)$
  $\mathcal{A} := \{v : (u, v)$ arc de $G\}$
  Si $\mathcal{A} = \emptyset$ alors $T(u) = d$
  Sinon $T(u) = -1 + \min\{T(v) : v \in \mathcal{A}\}$
  Fin pour
 Fin pour
Fin pour

FIG. 4.43 – *Mise en oeuvre des fonctions de temps combinatoires.*

Complexité en termes de processeurs

Notons P_{sys}, le nombre de processeurs minimum pour exécuter le schéma systolique de référence en T_{sys} unités de temps, et considérons les notations de la section 3.4.1.3. Nous discutons de la détermination de P_{sys} pour trois classes de matrices: pleines, bandes et creuses.

Lemme 3 (Cas d'une matrice pleine)
Pour le produit d'une matrice symétrique $n \times n$ par un vecteur, on a:

$$P_{sys} = \lceil \frac{n}{2} \rceil$$

Pour la factorisation LDL^T, on a:

- ⋆ *Si $n = 2p$ alors $P_{sys} = \frac{p(p+1)}{2}$*
- ⋆ *Si $n = 2p+1$ et p pair alors $P_{sys} = \frac{(p+1)^2+1}{2}$*
- ⋆ *Si $n = 2p+1$ et p impair alors $P_{sys} = \frac{(p+1)^2}{2}$*

Lemme 4 (Cas d'une matrice bande)
Soit une matrice $n \times n$ symétrique, bande et de largeur de bande $lbw \ll n$ (Il s'agit ici du nombre maximum de termes par lignes dans la partie triangulaire supérieure qui est très petit devant le nombre de lignes).
Pour le produit d'une matrice symétrique $n \times n$ par un vecteur, on a:

$$P_{sys} = \lceil \frac{lbw}{2} \rceil$$

Pour la factorisation LDL^T, on a:

$$\begin{array}{llll} Si & lbw = 3p & alors & P_{sys} = \frac{3p^2+p}{2} \\ si & lbw = 3p-1 & alors & P_{sys} = \frac{3p^2-p}{2} \\ si & lbw = 3p-2 & alors & P_{sys} = \frac{3p^2-3p+2}{2} \end{array}$$

On trouvera dans [BR90, BPST96] une technique permettant de démontrer ces deux derniers lemmes. Cette technique procède couche par couche. Nous présentons ici les éléments d'une telle technique dans le cas où les matrices sont bandes. La fonction de temps étant affine, il est possible de caractériser les sous ensembles $D(t)$ de sommets ayant la même date d'exécution.

Pour le produit matrice vecteur, on a:

$$D(t) = \{(x_1, x_2) \in D \ : \ x_2 = -x_1 + 1 + t\}. \tag{4.45}$$

Si lbw est la largeur de bande, on vérifie que, pour $t^* = lbw$, le nombre de processeurs maximal est atteint et vaut $P_{sys} = \frac{lbw}{2}$, si lbw est pair, et $P_{sys} = \frac{lbw+1}{2}$ si lbw est impair. Ce résultat s'obtient, en analysant, pour chaque ligne, les points de $D(t)$ aux instants successifs $t = 1, \ldots, 2n-1$.

Pour la factorisation LDL^T le calcul de P_{sys} procède par couches. Les couches ainsi considérées sont définies pour $k = 1, \ldots, n$ par:

$$D_k = \{(i, j, x) \in D : \ x = k\}$$

Sur chacune de ces couches, soit $D_k(t)$, la restriction de $D(t)$ à D_k. La cardinalité de $D_k(t)$, soit $m_k(t)$, est obtenue de la même manière que dans un produit matrice vecteur. Puisque

$$m(t) = \sum_{k=1}^{n} m_k(t)$$

on peut donc définir un tableau dont les lignes décrivent les valeurs de $m_k(t)$ aux instants successifs $t = 1, \ldots, T_{sys}$. La mise en oeuvre de $m(t)$ à partir des $m_k(t)$, pour $k = 1, \ldots, n$, se fait en additionnant les colonnes du dit tableau.

Dans le tableau (4.44), on effectue le calcul du nombre de processeurs en calculant $m_k(t)$ ligne par ligne. Il s'obtient en sommant tous les éléments de la même colonne du tableau. Ainsi, on vérifie par exemple que, pour $t = lbw$, le nombre de processeurs maximal est atteint.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$m_1(t)$	1	1	2	1	1	0	0	0	0	0	0	0	0	0	0	0
$m_2(t)$	0	0	0	1	1	2	1	1	0	0	0	0	0	0	0	0
$m_3(t)$	0	0	0	0	0	0	1	1	2	1	1	0	0	0	0	0
$m_4(t)$	0	0	0	0	0	0	0	0	0	1	1	2	1	1	0	0
$m_5(t)$	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0
$m_6(t)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
$m(t)$	1	1	2	2	2	2	2	2	2	2	2	2	2	2	1	1

FIG. 4.44 – *Calcul du nombre de processeurs pour une matrice bande.*

Pour des matrices skylines quelconques ou plus généralement pour les matrices creuses, il est difficile d'expliciter le calcul de P_{sys}, compte tenu du caractère combinatoire des graphes. Lorsque la matrice est à largeur de bande variable, le lemme (4) permet de calculer une borne supérieure pour P_{sys} à partir de la largeur de bande maximale de la matrice.

4.2.3.4 Allocation des nouveaux schémas d'exécution

Il s'agit d'allouer les sommets des graphes $G(\mu)$ décrits dans la section 4.2.3.2. Ces graphes se caractérisent par des *SERUR*, dont l'ordonnancement a été formellement étudié dans la section 4.2.3.3. L'idée de l'allocation des nouveaux schémas d'exécution réside dans l'allocation de la matrice de cellules créée dans la section 4.2.3.1 et associée à la matrice $A(\mu)$. Nous illustrons une telle allocation dans un premier temps, sur la factorisation LDL^T, puis, nous en exposons les grandes lignes pour le produit matrice vecteur. Les notations utilisées dans ce qui suit, sont celles introduites dans les sections 4.2.3.1, 4.2.3.2 et 4.2.3.3. Puisque nous manipulons des graphes similaires à ceux construits dans la section 4.2.3.1, nous discutons formellement de l'allocation des sommets des graphes $G = (D, E)$.

Exemple de la factorisation LDL^T

Nous considérons d'abord une allocation induite par projection de G dans l'une des directions de l'espace qui le contient, puis allouerons les sommets du graphe résultant aux processeurs.

Dans chacune des étapes, la fonction d'allocation est astreinte à la compatibilité avec la fonction de temps à savoir des tâches (cellules) ayant les mêmes dates d'activation ne peuvent être allouées aux même processeurs.

Etape 1:

pour le problème de la factorisation LDL^T nous procéderons à une projection de G dans la direction $(0,0,1)$. Dans le cas d'une matrice pleine, le graphe résultant est un réseau triangulaire comme illustré par la figure (4.46) où chaque sommet est étiqueté par un intervalle dont les bornes correspondent aux dates d'activation des sommets (i,j,k_0) et (i,j,i) de G. Notons que k_0 est la première ligne qui induit une modification partielle de $a_{i,j}$. Formellement on a:

$$\mathcal{T}: \ (i,j) \rightarrow [T(i,j,k_0), T(i,j,i)] \tag{4.46}$$

On peut aisément vérifier que la fonction de temps $\mathcal{T}$ induite par cette projection préserve T_{sys} comme temps d'exécution. Par contre, le nombre de sommets est égal au nombre de coefficients de la matrice, soit nz, et supérieur à P_{sys}.

Etape 2:

Elle consiste à définir une allocation des sommets de G à P_{sys} processeurs de façon compatible avec la fonction de temps $\mathcal{T}$. Nous proposons pour cela une allocation ligne par ligne:

1: $k \leftarrow 0$

2: $K \leftarrow \emptyset$ (Ensemble de processeurs auxquels des tâches ont été allouées)

3: Pour $i = 1, \cdots, n$ faire

3.1: Pour $j = 1, \cdots, n$ faire

3.2: Chercher dans K le plus petit élément k_{min} tel que le dernier intervalle de temps d'activation de $P_{k_{min}}$ n'intersecte pas $\mathcal{T}(i,j)$

3.3: Si k_{min} existe alors allouer (i,j) à $P_{k_{min}}$

3.4: Sinon

$k \leftarrow k+1$

allouer (i,j) à P_k

$K \leftarrow K \cup \{k\}$ (au sens ensembliste)

Fin si

Fin pour

Fin pour

Cette stratégie est illustrée par les figures (4.47) (resp. (4.49)) pour le cas des matrices pleines (resp creuses). La répartition de données induite pour la matrice est donnée par la figure (4.48) pour une matrice pleine et (4.50) pour une matrice creuse. On peut alors montrer que la cardinalité de K vaut P_{sys} pour les matrices pleines et skyline. En effet, au regard de l'étape 2 il suffit de montrer que lorsque les dates d'activation associées à tout couple (i,j) sont consécutives, $|K| = P_{sys}$.

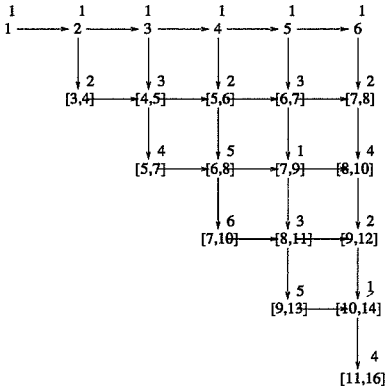

FIG. 4.45 – Projection de G dans la direction $(0,0,1)$ pour une matrice pleine.

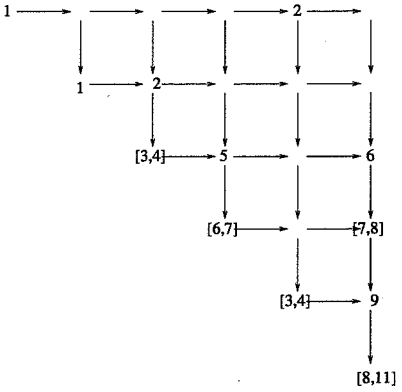

FIG. 4.46 – Projection de G dans la direction $(0,0,1)$ pour une matrice pleine.

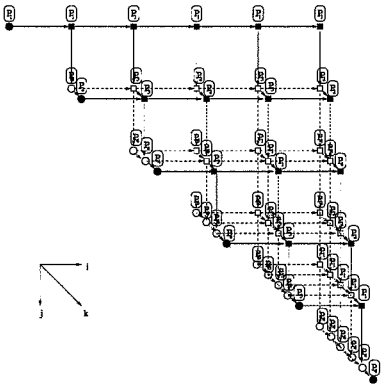

FIG. 4.47 – Allocation des calculs pour une matrice pleine.

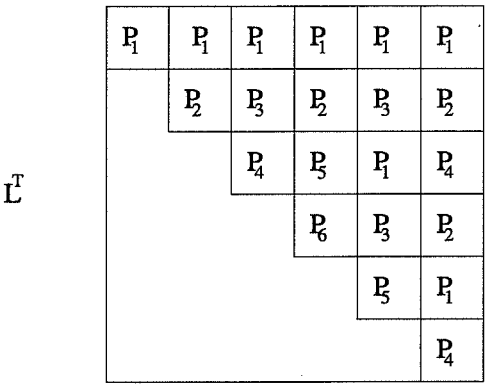

FIG. 4.48 – Allocation des coefficients d'une matrice pleine

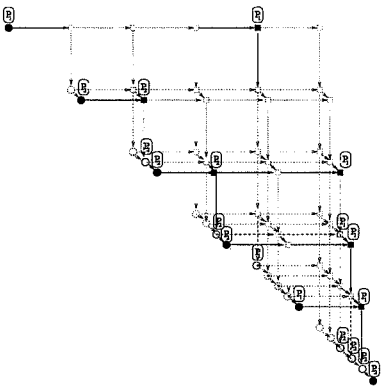

FIG. 4.49 – Allocation des calculs pour une matrice creuse.

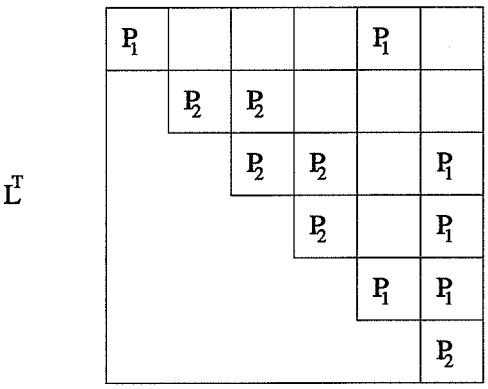

FIG. 4.50 – Allocation des coefficients d'une matrice creuse

Exemple du produit matrice vecteur

Pour les besoins de l'allocation, il suffit de prendre pour direction de projection $\vec{d} = (0, 1)$. Puisque les super-noeuds situés sur des droites parallèles à $\vec{d}$ ont des dates d'activation toutes distinctes et permettent des transformations successives des données $\{y_i^{(j)}, x_i\}$, pour $j \geq i$, ils peuvent être alloués à un même processeur. Un tel processeur est donc le détenteur du résultat $\{y_i, x_i\}$. Par conséquent, une allocation ligne par ligne de la matrice de cellule est suffisante. De plus, pour démarrer les calculs en parallèle, il suffit d'allouer les termes x_j et y_j aux processeurs possédant les lignes i où la colonne j apparaît en premier.

Dans la figure (4.51), nous illustrons la stratégie de la section 3.4.4.2 pour le cas d'un graphe similaire à celui d'un produit matrice vecteur plein avec $n = 6$. Elle induit une allocation des données illustrée par la figure (4.52). Il en résulte une allocation de la matrice A ligne par ligne et cyclique.

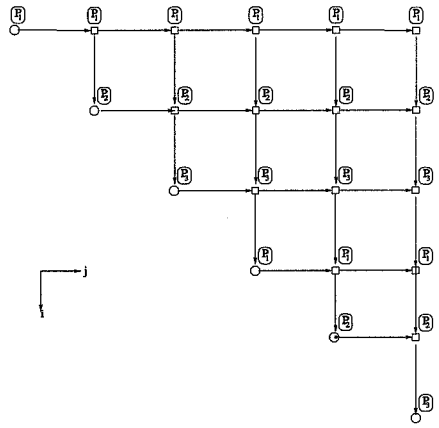

FIG. 4.51 – Allocation des calculs aux processeurs dans le cas plein.

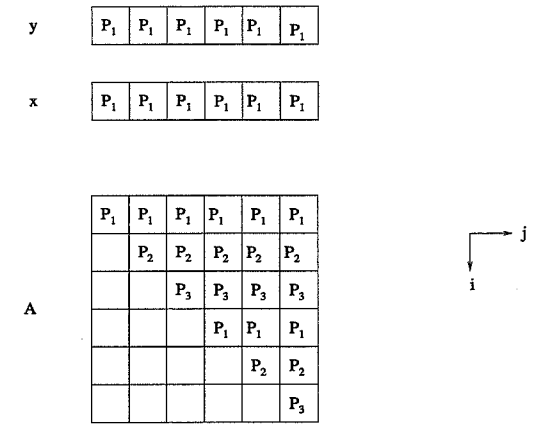

FIG. 4.52 – Allocation des données aux processeurs dans le cas plein.

Dans le cas de matrices creuses, bien que l'allocation demeure ligne par ligne, elle n'est plus cyclique. Les figures (4.53) et (4.54) illustrent respectivement les allocations de calculs et de données pour une produit matrice vecteur creux, avec $n = 6$ et le profil de la matrice étant celui de la figure (4.9).

4.2.4 Génération de codes parallèles

Pour exécuter les algorithmes des sommets de $G(\mu)$ alloués à un processeur, la génération de code produit principalement deux boucles imbriquées, comme l'illustre l'algorithme de la figure (4.55). Le premier niveau d'imbrication est un parcours de toutes les cellules allouées à un processeur. Le second niveau d'imbrication est le parcours des super-noeuds dont les coordonnées sont décrites dans une cellule. Ces super-noeuds pouvant être regroupés par classes, "exécuter" $P(z)$ signifie, exécuter le programme correspondant à la classe du super-noeud z. Si z et son prédécesseur (respectivement successeur) sont alloués au même processeur, la

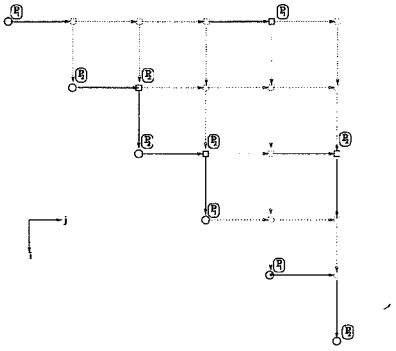

FIG. 4.53 – Allocation des calculs aux processeurs dans le cas creux.

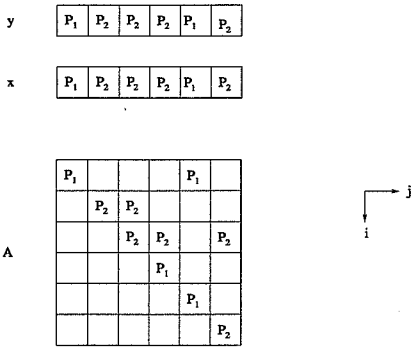

FIG. 4.54 – Allocation des données aux processeurs dans le cas creux.

réception (respectivement l'envoi) correspond à une lecture (respectivement écriture).

```
Pour toutes les cellules allouées à un processeur
 Pour tous les super-noeuds z d'une cellule
 Recevoir des prédécesseurs
 Exécuter P(z)
 Envoyer aux successeurs
 Fin pour
Fin pour
```

FIG. 4.55 – *Génération de code parallèle.*

Dans ce qui suit, nous réutilisons les notations de la section 4.2.3, et présentons les classes d'algorithmes associées aux super-noeuds pour le produit matrice vecteur et pour la factorisation LDL^T. Les sous graphes de G associés à ces classes sont exposés dans la section 4.2.3.2.

4.2.4.1 Classes d'algorithmes pour le produit matrice vecteur

Les algorithmes relatifs aux points $z \in D(\mu)$ appartiennent à deux classes illustrées dans la figure (4.56). Dans ces algorithmes, $\mathcal{L}_{I,J}$ désigne les indices de lignes de la sous-matrice $A_{I,J}(\mu)$. Pour $i \in \mathcal{L}_{I,J}$, l'ensemble $\mathcal{C}_{I,J}(i)$ est formé par les colonnes de la ligne i dont les termes $a_{i,j}$ sont les coefficients non-nuls de $A_{I,J}(\mu)$.

La première classe est constituée de super-noeuds de coordonnées du type $z = (I, I)$, dont l'algorithme est défini dans la figure (4.56). Dans cet algorithme, les termes x_i et y_i pour $i \in \mathcal{L}_{I,I}$ ou x_j et y_j pour $j \in \mathcal{C}_{I,I}(i)$ et $i \in \mathcal{L}_{I,I}$ sont respectivement les composantes des vecteurs $X_I(\mu)$ et $Y_I(\mu)$. Le couple $\{X_I(\mu), Y_I(\mu)\}$ est reçu dans la direction $(0, 1)$. Après

modification des coefficients de $Y_I(\mu)$, ce couple est sauvegardé en mémoire par le processeur en charge du super-noeud (I, I).

La deuxième classe est constituée de super-noeuds de coordonnées du type $z = (I, J)$, avec $I < J$, dont l'algorithme est défini dans la figure (4.56). Dans cet algorithme, les termes x_i et y_i pour $i \in \mathcal{L}_{I,J}$ sont respectivement les composantes des vecteurs $X_I(\mu)$ et $Y_I(\mu)$ qui résident dans la mémoire locale du processeur en charge de z. Les termes x_j et y_j, pour $j \in \mathcal{C}_{I,J}(i)$ et $i \in \mathcal{L}_{I,J}$, quant à eux sont les composantes des vecteur $X_J(\mu)$ et $Y_J(\mu)$. Le couple $\{X_J(\mu), Y_J(\mu)\}$ est reçu dans la direction $(0, 1)$. Après modification des coefficients de $Y_I(\mu)$ et de $Y_J(\mu)$, le couple $\{X_J(\mu), Y_J(\mu)\}$ est envoyé dans la direction $(0, 1)$. Le couple $\{X_I(\mu), Y_I(\mu)\}$ quant à lui est sauvegardé en mémoire par le processeur en charge du super-noeud (I, J).

$z = (I, I)$

Pour $i \in \mathcal{L}_{I,I}$ faire
  $y_i := y_i + a_{i,i}x_i$
  Pour $j \in \mathcal{C}_{I,I}(i)$ faire
    $y_i := y_i + a_{i,j}x_j$
    $y_j := y_j + a_{i,j}x_i$
  Fin pour
Fin pour

$z = (I, J)$

Pour $i \in \mathcal{L}_{I,J}$ faire
  Pour $j \in \mathcal{C}_{I,J}(i)$ faire
    $y_i := y_i + a_{i,j}x_j$
    $y_j := y_j + a_{i,j}x_i$
  Fin pour
Fin pour

FIG. 4.56 – *Algorithme de $P(z)$.*

4.2.4.2 Classes d'algorithmes pour la factorisation LDL^T

Les algorithmes relatifs aux points $z \in D(\mu)$ sont dans quatre classes. Les deux premières classes sont relatives aux traitement de sous-matrices de la forme $A_{I,I}(\mu)$. Deux classes d'algorithmes sont associées à de telles sous-matrices. Elles correspondent aux points $z = (I, I, I)$ ou $z = (I, I, K)$, avec $K < I$ et sont illustrées dans la figure (4.57). Les deux dernières quant à elles sont relatives aux sous-matrices de la forme $A_{I,J}(\mu)$, avec $I \neq J$. Elles correspondent aux points $z = (I, J, I)$ ou $z = (I, J, K)$, avec $K < I$ et sont illustrées dans la figure (4.58).

Dans ces quatre classe d'algorithmes, $\mathcal{I}_{I,J}$ représente la liste des indices de ligne de la matrice $A_{I,J}(\mu)$. Pour chaque indice $i \in \mathcal{I}_{I,J}$, $\mathcal{J}_{I,J}(i)$ désigne la liste des indices de colonne des termes à modifier à la i-ème ligne. $\mathcal{K}_{K,I}$ et $\mathcal{K}_{K,J}$ correspondent respectivement aux ensembles des indices de lignes des matrices $A_{K,I}(\mu)$ et $A_{K,J}(\mu)$ induisant des modifications de la matrice $A_{I,J}(\mu)$. Dans ces algorithmes, $a_{i,j}^{(k-1)}$ signifie la valeur partielle de $a_{i,j}$ avant la k-ème itération.

Pour $z = (I, I, I)$, la sous-matrice $A_{I,I}(\mu)$ est modifiée uniquement à partir de ses propres termes. L'algorithme de $P(z)$ permet alors de construire un ensemble de données, noté $\mathcal{H}$, à partir des valeurs $a_{i,j}^{(i-1)}$. La donnée $\mathcal{H}$ est ensuite envoyée dans la direction $(1, 0, 0)$. Ses indices de ligne et colonne sont donc ceux de $A_{I,I}(\mu)$.

Pour $z = (I, I, K)$, la sous-matrice $A_{I,I}(\mu)$ est modifiée à partir de valeurs reçues dans la direction $(0, 1, 0)$ et stockées dans une structure de donnée $\mathcal{V}$. $\mathcal{V}$ est formée de termes $a_{k,k}^{(k-1)}$

de la diagonale de $A_{K,K}(\mu)$ et de termes $a_{k,i}^{(k-1)}$ de $A_{K,I}(\mu)$. Cette donnée a été créée par le bloc (K,I,K). $\mathcal{V}$ est ensuite transmise dans la direction $(1,0,0)$.

$\underline{z=(I,I,I)}$

Pour $i \in \mathcal{I}_{I,I}$ faire
$a_{i,i}^{(i)} := 1/a_{i,i}^{(i-1)}$
Pour $j \in \mathcal{J}_{I,I}(i)$ faire
$a_{i,j}^{(i)} := a_{i,j}^{(i-1)}/a_{i,i}^{(i-1)}$
Pour $k \in \mathcal{J}_{I,J}(i)$ et $k \geq j$ faire
$a_{j,k}^{(i)} := a_{j,k}^{(i-1)} - a_{i,j}^{(i-1)} * a_{i,k}^{(i-1)}/a_{i,i}^{(i-1)}$
Fin pour
Fin pour
Fin pour

$\underline{z=(I,I,K)}$

Pour $k \in \mathcal{K}_{K,I}$ faire
Pour $i \in \mathcal{I}_{I,J}$ faire
Pour $j \in \mathcal{J}_{I,J}(i)$ faire
$a_{i,j}^{(k)} := a_{i,j}^{(k-1)} - a_{k,i}^{(k-1)} * a_{k,j}^{(k-1)}/a_{k,k}^{(k-1)}$
Fin pour
Fin pour
Fin pour

FIG. 4.57 – *Sous-programmes $P(z)$ relatifs aux traitement d'une matrice $A_{I,I}(\mu)$.*

Pour $z=(I,J,I)$, la sous-matrice $A_{I,J}(\mu)$ est modifiée à partir de ses propres termes et de ceux de la structure de donnée $\mathcal{H}$ précédemment créée par le bloc (I,I,I). $\mathcal{H}$ est formée par les termes $a_{i,j}^{(i-1)}$, pour $i \in \mathcal{I}_{I,I}$ et $j \in \mathcal{J}_{I,J}(i)$. L'algorithme de $P(z)$ permet alors de construire un ensemble de données, noté $\mathcal{V}$, à partir des valeurs $a_{i,j}^{(i-1)}$ telles que $a_{i,j}$ est un terme de $A_{I,J}(\mu)$ et $a_{i,i}^{(i-1)}$ telles que $a_{i,i}$ est un terme de $A_{I,I}(\mu)$. La donnée $\mathcal{H}$ est ensuite envoyée dans la direction $(1,0,0)$ tandis que $\mathcal{V}$ l'est dans la direction $(0,1,0)$.

Pour $z=(I,J,K)$, la sous-matrice $A_{I,J}(\mu)$ est modifiée à partir de valeurs reçues dans la direction $(0,1,0)$, soit $\mathcal{V}$, et dans la direction $(1,0,0)$, soit $\mathcal{H}$. $\mathcal{V}$ est formée par les termes construits par le bloc (K,I,K) à partir des données $a_{k,i}^{(k-1)}$, associés à la matrice $A_{K,I}(\mu)$ et $a_{k,k}^{(k-1)}$, associés à la matrice $A_{K,K}(\mu)$. $\mathcal{H}$ est formée par les termes construits par le bloc (K,J,K) à partir des données $a_{k,j}^{(k-1)}$, dus à la matrice $A_{K,J}(\mu)$ et $a_{k,k}^{(k-1)}$, dus à la matrice $A_{K,K}(\mu)$. $\mathcal{V}$ et $\mathcal{H}$ sont ensuite envoyées dans les directions $(0,1,0)$ et $(1,0,0)$ respectivement.

4.2.5 Description des structures de données pour la mise en oeuvre

Ces structures de données concernent les méthodes à matrices skylines du module *DLANCB*. Un point commun dans la description des traitements de ces algorithmes est qu'ils peuvent se modéliser à partir d'une même structure de donnée, encore appelée matrice de cellule, construite à l'aide du profil de la matrice induite par le découpage par blocs. La différence entre ces algorithmes se situe dans les structures de données servant à stocker les résultats intermédiaires qui peuvent être échangés entre les processeurs ou réutilisés par un même processeur.

4.2.5.1 Structure de donnée commune

C'est une matrice constituée de cellules dont le rangement est à l'identique de celui de la matrice $A(\mu)$. Chaque cellule (i,j) est allouée à un processeur et comporte des informations

$z = (I, J, I)$

Pour $i \in \mathcal{I}_{I,J}$ faire
Pour $j \in \mathcal{J}_{I,J}(i)$ faire
$a_{i,j}^{(i)} := a_{i,j}^{(i-1)} / a_{i,i}^{(i-1)}$
Fin pour
Pour $j \in \mathcal{J}_{I,J}(i)$ faire
Pour $k \in \mathcal{J}_{I,J}(i)$ et $k \geq j$ faire
$a_{k,j}^{(i)} := a_{k,j}^{(i-1)} - a_{i,j}^{(i-1)} * a_{i,k}^{(i-1)} / a_{i,i}^{(i-1)}$
Fin pour
Fin pour
Fin pour

$z = (I, J, K)$

Pour $k \in \mathcal{K}_{I,J}$ faire
Pour $i \in \mathcal{I}_{I,J}$ faire
Pour $j \in \mathcal{J}_{I,J}(i)$ faire
$a_{i,j}^{(k)} := a_{i,j}^{(k-1)} - a_{k,i}^{(k-1)} * a_{k,j}^{(k-1)} / a_{k,k}^{(k-1)}$
Fin pour
Fin pour
Fin pour

FIG. 4.58 – *Programmes $P(z)$ relatifs aux traitement d'une matrice $A_{I,J}(\mu)$.*

sur des super-noeuds telles que leurs coordonnées. Ainsi:

- ⋆ pour le produit matrice vecteur ou les résolutions de systèmes triangulaires, les super-noeuds ont pour coordonnées (i, j),
- ⋆ pour la factorisation LDL^T, notons l_j le numéro du premier bloc de ligne où apparaît le bloc de colonne j. Les super-noeuds ont pour coordonnées (i, j, k), avec $l_j \leq k \leq i$.

Ces super-noeuds sont reliés à des prédécesseurs ou à des successeurs par des arcs dont la construction est définie par l'algorithme de la figure (4.16). Nous discutons de la mise en oeuvre de ces arcs dans le cadre d'un rangement des cellules en mode skyline.

Dans le cas du produit matrice vecteur ou des résolutions des systèmes triangulaires à matrice skyline, les prédécesseurs ou les successeurs d'un super-noeud, de coordonnées (i, j), sont:

- ⋆ au nord, le super-noeud $(i-1, j)$, si $i \neq 1$,
- ⋆ au sud, le super-noeud $(i+1, j)$, si $1 \leq i \leq N-1$ et $i \neq j$,
- ⋆ à l'ouest, le super-noeud $(i, j-1)$, si $i \neq 1$ et $i \neq j$,
- ⋆ à l'est, le super-noeud $(i, j+1)$, si $i < N$ et $i+1 \leq j \leq N-1$.

Notons qu'un prédécesseur ou un successeur de (l, k) n'existe que si la sous-matrice $A_{l,k}(\mu)$ contient des coefficients de A. Cette extension des stockages par blocs peut également servir pour la factorisation LDL^T. En effet, un processeur en charge de l'exécution d'un super-noeud (i, j, k), communique avec les mêmes processeurs en charge des prédécesseurs ou des successeurs du super-noeud (i, j, l_j). Ainsi, pour simuler le traitement des super-noeuds associés à la cellule (i, j), il suffit de rajouter à la description de cette cellule, une information concernant l'ordre d'exécution de ses super-noeuds dans l'intervalle $[l_j, i]$. Pour une cellule, nous définissons le type *Cellule* par les instructions de la figure (4.59).

Puisque la matrice de cellule a un profil identique à celui de la matrice $A(\mu)$, on peut donc utiliser la définition d'un stockage par blocs standard [BBC+94] pour la représenter. On

```
Type Cellule = record
    i : integer;
    j : integer;
    lj : integer;
    nord, sud, est, ouest : integer;
    Aij : SousMatrice;
    End
```

FIG. 4.59 – *Description du type Cellule.*

définit ainsi un type *MatriceCellule* par les instructions de la figure (4.60). Les cellules sont rangées bloc de lignes par bloc de lignes, en mode skyline. Dans le cadre d'une machine à mémoire distribuée, on applique formellement la représentation des matrices distribuées telle que exposée dans la section 2.3.1.1.

```
Type MatriceCellule = record
    n_ligne_cel : integer;
    n_total_cel : integer;
    ptr_ligne_cel : array[1...n_ligne_cel_max] of integer;
    val_cel : array[1...n_total_cel_max] of Cellule;
    End
```

FIG. 4.60 – *Description du type MatriceCellule.*

4.2.5.2 Structure de donnée pour les résultats intermédiaires

On distingue deux sortes de résultats intermédiaires: les vecteurs et les matrices. Nous décrivons maintenant les structures de données permettant de les représenter.

Les vecteurs:

ils sont échangés entre les processeurs lorsqu'on effectue des opérations telles que le produit matrice vecteur ou les résolutions de systèmes triangulaires. Pour le produit matrice vecteur, soit $y = Ax$ avec $x = (x_i)_{1 \leq i \leq n}$ et $y = (y_i)_{1 \leq i \leq n}$, on échange dans chaque direction, deux morceaux de vecteurs, soient X et Y, dont les coefficients sont respectivement les x_i et y_i, pour $i \in [i_1, i_2]$. Pour les résolutions de systèmes triangulaires, on échange des morceaux de vecteurs dont les coefficients sont x_i pour $i \in [i_1, i_2]$. Le stockage par blocs du vecteur qui en résulte se caractérise par le type *VecteurBloc* défini par les instructions de la figure (4.61).

Remarquons qu'en effectuant une résolution de système triangulaire inférieur puis supérieur, il convient de disposer d'une variable de type *VecteurBloc* à plusieurs éléments, au niveau du serveur afin de réduire les phases de communication entre le programme client et le serveur. En effet tout processeur est en charge du même bloc de ligne aussi bien pour la

```
Type VecteurBloc = record
    n_bloc_ligne : integer;
    n_ligne : integer;
    ptr_bloc_ligne : array[1...n_bloc_ligne_max] of integer;
    val_vec : array[1...n_ligne_max] of real;
    End
```

FIG. 4.61 – *Description du type VecteurBloc*

résolution du système triangulaire inférieur que supérieur. Comme le montre la figure (4.62), un processeur qui produit un morceau de vecteur B_i lors de la résolution d'un systèmes triangulaire inférieur, doit réutiliser ce même morceau lors de la résolution du système triangulaire supérieur. Si ce morceau n'est pas disponible, alors une communication explicite est requise.

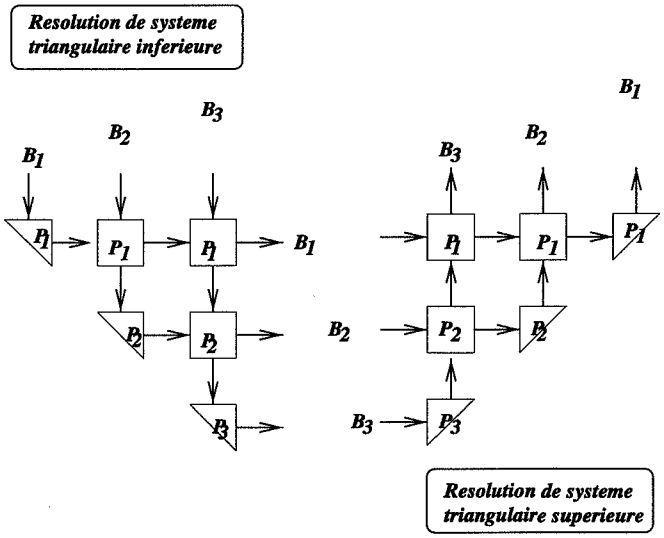

FIG. 4.62 – Réutilisation de données dans le cadre de résolution de systèmes triangulaires.

Les matrices:

elles sont échangées dans le cadre de la factorisation LDL^T. Toutefois, pour chacune de ces sous-matrices, il faut rajouter lors de la description d'une ligne, soit i, le coefficient diagonal de la matrice $a_{i,i}$. Ainsi, on peut modéliser les sous-matrices échangées à l'aide d'une structure de donnée du type *SousMatrice* illustrée dans la figure (4.63). Chaque sous matrice se caractérise alors par un stockage skyline dont les paramètres sont:

⋆ le nombre de lignes de A contenues dans cette sous-matrice, soit *ssm_n*;

- ⋆ le nombre de termes de $a_{i,j}$ contenus dans cette sous-matrice, soit *ssm_nz*;
- ⋆ un tableau, noté *indice*, qui indique pour chaque ligne de la sous-matrice l'adresse du coefficient de la première colonne;
- ⋆ un tableau, noté *val*, qui contient tous les termes $a_{i,j}$ contenus dans cette sous-matrice.

```
Type SousMatrice = record
    ssm_n : integer;
    ssm_nz : integer;
    indice : array[1...ssm_n_max] of integer;
    val : array[1...ssm_nz_max] of real;
    End
```

FIG. 4.63 – *Description du type SousMatrice.*

4.2.6 Analyse de complexité

Dans l'optique d'une étude comparative des performances théoriques et expérimentales des serveurs parallèles de *DLANCB*, l'analyse de complexité se concentre sur des matrices stockées en mode skyline. Soit $A = (a_{i,j})_{1 \leq i,j \leq n}$ une telle matrice, avec n, son nombre de lignes, nz, le nombre de termes stockés dans le profil de la partie triangulaire supérieure de A, $(l_i)_{1 \leq i \leq n}$, la suite décrivant le nombre de termes par lignes, lbw le nombre maximum de termes par ligne.

Dans la section 3.3.1, nous avons proposé une majoration du temps d'exécution parallèle qui dépend de l'algorithme parallèle et de la machine parallèle cible. On construit alors une estimation a priori du temps d'exécution en fonction de 4 paramètres:

- ⋆ t_{scal}, la vitesse de traitement des opérations élémentaires,
- ⋆ τ^*, la temps d'initialisation engendré par l'identification puis le traitement des blocs de calcul,
- ⋆ τ, le temps de "start-up" pour les communications,
- ⋆ θ, le taux de transfert des messages entre deux noeuds de la machine parallèle.

Pour les graphes étudiés dans la section 4.2.3.1, le partitionnement par blocs que nous recherchons produit un graphe $G(\mu) = (D(\mu), E(\mu))$ similaire à G, où $D(\mu)$ et $E(\mu)$ sont respectivement les ensembles de sommets et d'arcs, μ est une suite de N termes définie par l'équation (4.34). Pour tout sommet u de $G(\mu)$, soit $T_{cal}(u)$ le temps d'exécution des opérations scalaires de u. Pour tout arc (u, v) de $G(\mu)$, soit $T_{com}(u, v)$ le temps d'acheminement des messages de u à v. Nous avons montré qu'une majoration du temps d'exécution $T_{exe}(G(\mu))$ est donnée par:

$$T_{exe}(G(\mu)) \leq (d(\mu)+1) \max_{u \in D(\mu)} T_{cal}(u) + d(\mu) \max_{(u,v) \in E(\mu)} \Phi(q, T_{com}(u, v)) \tag{4.47}$$

avec $d(\mu)$ la longueur du plus long chemin dans $G(\mu)$ et la fonction Φ modélise le coût des communications sur une architecture donnée lorsque ces communications doivent se faire simultanément avec les prédécesseurs d'un sommet. On pose:

$$\mu_{max} = \max_{1 \leq i \leq N} (\mu_i) \tag{4.48}$$

4.2.6.1 Complexité séquentielle

l'opération $y = Ax$:

on se sert des coefficients stockés dans la partie triangulaire supérieure de la matrice A. On exécute deux boucles imbriquées pour $i = 1, \cdots, n$ et pour $j := i+1, \cdots, i+l_i-1$. Lors du traitement de la contribution du coefficient $a_{i,i}$, on effectue une addition et une multiplication. Pour les termes extra-diagonaux $a_{i,j}$, avec $j > i$, on effectue deux additions et deux multiplications. Par conséquent, pour la ligne i, on effectue $4l_i - 2$ opérations sur des scalaires. Pour les n lignes, on a au total:

$$f_0((l_i)_{1 \leq i \leq n}) = \sum_{i=1}^{n} (4l_i - 2) = 4nz - 2n \tag{4.49}$$

l'opération $LDL^T x = b$

on suppose que la diagonale de D^{-1} est stockée dans celle d'une matrice A et que les termes du profil de la partie triangulaire supérieure stricte de L^T sont stockés dans celle de la matrice A qui contient déjà la diagonale de D^{-1}. Ceci est possible grâce au sous-programme qui effectue $A = LDL^T$. La résolution de $LDL^T x = b$ comporte une descente et une remontée dans lesquelles on accède aux $a_{i,j}$ ligne par ligne. Au cours de la descente, on exécute deux boucles imbriquées. La contribution de $a_{i,i}$ est une multiplication de deux scalaires. Celle de $a_{i,j}$ comporte une addition et une multiplication de scalaires. Au total, on effectue $2l_i - 1$ opérations scalaires. Pour n lignes on a:

$$\sum_{i=1}^{n} (2l_i - 1) = 2nz - n \tag{4.50}$$

Pour la remontée, on se sert uniquement de la partie triangulaire strictement supérieure de A. On exécute deux boucles imbriquées. La contribution de $a_{i,j}$, avec $j > i$ se résume à une addition et une multiplication de scalaires. Soit, pour la ligne i, $2(l_i - 1)$ opérations. On effectue donc, pour la remontée,

$$\sum_{i=1}^{n} (2(l_i - 1)) = 2nz - 2n \tag{4.51}$$

Le nombre d'opérations sur les scalaires effectué pour $LDL^T x = b$ est obtenu en additionnant les quantités calculées en (4.50) et (4.51). D'où:

$$f_0((l_i)_{1 \leq i \leq n}) = 2nz - 2n + 2nz - n = 4nz - 3n \tag{4.52}$$

l'opération $A = LDL^T$

on stocke dans le profil de la partie triangulaire supérieure de A, la diagonale de D^{-1} et la partie triangulaire supérieure stricte de L^T. Ceci permet d'effectuer l'opération $LDL^T x = b$

telle que précédemment décrite. On effectue trois boucles imbriquées. Les deux premières boucles permettent de parcourir ligne par ligne la matrice A. Pour une ligne, le terme $a_{i,i}$ est modifié par une division de deux scalaires. Le terme $a_{i,j}$ est modifié par une multiplication de deux scalaires. La troisième boucle permet de modifier partiellement $\frac{(l_i-1)l_i}{2}$ termes de la matrice A. Pour chacune de ces modifications partielles, on effectue une addition et une multiplication. Au total, le nombre d'opérations sur des scalaires effectué lors du traitement de la ligne i est donné par:

$$l_i + (l_i - 1)l_i = l_i^2 \tag{4.53}$$

Le nombre total d'opérations sur des scalaires ainsi effectué est donné par:

$$f_0((l_i)_{1\leq i\leq n}) = \sum_{i=1}^{n} l_i^2 \tag{4.54}$$

4.2.6.2 Complexité parallèle en termes de calculs

Pour l'opération $y = Ax$,

$d(\mu) = 2N - 2$ et le nombre d'opérations sur les scalaires effectué par chaque bloc est au plus égal à $4\mu_{max}^2$ (pour les blocs extra-diagonaux). Au total, le nombre d'opérations sur des scalaires ainsi effectué lors du traitement sur une machine parallèle est majoré par:

$$T_{cal}(G(\mu)) \leq (2N - 1)(4t_{scal}\mu_{max}^2 + \tau^*) \tag{4.55}$$

Nous posons:

$$f_1(\alpha, \beta) = 2\beta - 1 \tag{4.56}$$

et

$$f_2(\alpha, \beta) = 4\alpha^2(2\beta - 1) \tag{4.57}$$

On construit ainsi, une borne supérieure de $T_{cal}(G(\mu))$ qui dépend linéairement des paramètres t_{scal} et τ^*. Soit

$$T_{cal}(G(\mu)) \leq \tau^* f_1(\mu_{max}, N) + t_{scal} f_2(\mu_{max}, N) \tag{4.58}$$

Pour l'opération $LDL^T x = b$,

le nombre d'opérations sur des scalaires, aussi bien lors de la descente que de la remontée, est majoré par $2\mu_{max}^2$. Pour la descente et pour la remontée, $d(\mu) = 2N - 2$. Lorsque l'on effectue une descente puis une remontée, on obtient la majoration suivante pour $T_{cal}(G(\mu))$:

$$T_{cal}(G(\mu)) \leq \tau^* f_1(\mu_{max}, N) + t_{scal} f_2(\mu_{max}, N) \tag{4.59}$$

avec

$$f_1(\alpha, \beta) = 4\beta - 2 \tag{4.60}$$

et

$$f_2(\alpha, \beta) = 4\alpha^2(2\beta - 1) \tag{4.61}$$

Pour l'opération $A = LDL^T$,

$d(\mu) = 3N - 2$ et le nombre d'opérations sur des scalaires est majoré, pour chaque bloc par $2\mu_{max}^3$. Au total, nombre d'opérations sur les scalaires effectué en parallèle est majoré par:

$$T_{cal}(G(\mu)) \leq (2\mu_{max}^3 t_{scal} + \tau^*)(3N - 2) \tag{4.62}$$

On définit alors une majoration de $T_{cal}(G(\mu))$ par:

$$T_{cal}(G(\mu)) \leq \tau^* f_1(\mu_{max}, N) + t_{scal} f_2(\mu_{max}, N) \tag{4.63}$$

avec

$$f_1(\alpha, \beta) = 3\beta - 2 \tag{4.64}$$

et

$$f_2(\alpha, \beta) = 2\alpha^3(3\beta - 2) \tag{4.65}$$

4.2.6.3 Etude des communications

Pour l'opération $y = Ax$:

deux morceaux de vecteur de taille au plus égal à μ_{max} sont acheminés. Ces deux morceaux de vecteur sont représentés par des tableaux de nombres réels. Avec l'utilisation des *BLACS*, il suffit d'un tableau de $2\mu_{max}$ réels pour acheminer les messages entre u et v. Dans le programme qui a été implanté, on utilise des nombres réels en double précision. Ainsi, on achemine l'équivalent de $4\mu_{max}$ nombres de type entier entre u et v. D'où la majoration suivante de $T_{com}(u, v)$:

$$T_{com}(u, v) \leq \tau + 4\theta\mu_{max} \tag{4.66}$$

Pour l'opération $LDL^T x = b$:

on achemine un morceau de vecteur contenant au plus μ_{max} réels, aussi bien lors de la descente que de la remontée. Puisque l'implantation suit les mêmes contraintes que celles de l'opération $y = Ax$, on obtient la majoration suivante de $T_{com}(u, v)$:

$$T_{com}(u, v) \leq \tau + 2\theta\mu_{max} \tag{4.67}$$

Pour l'opération $A = LDL^T$:

on achemine une structure de donnée de type *SousMatrice*. Avec l'implantation des programmes en fortran, on simule cette structure de donnée par une matrice stockée en mode skyline. Dans un tel stockage, deux tableaux sont utilisés: un tableau d'au plus $\mu_{max} + 1$ nombres entiers et un tableau d'au plus $\mu_{max}^2 + \mu_{max}$ nombres réels. Ainsi, on va d'abord acheminer le tableau de nombres entiers (à l'aide de la fonction *IGEBS2D*), puis, celui de nombres réels (à l'aide de la fonction *DGEBS2D*). Notons que les réels sont codés en double précision. L'appel de *IGEBS2D*, puis de *DGEBS2D*, induit implicitement deux phases "start-up". D'où la majoration suivante de $T_{com}(u, v)$:

$$T_{com}(u, v) \leq 2\tau + \theta(2\mu_{max}^2 + 3\mu_{max}) \tag{4.68}$$

Ainsi, $T_{com}(u, v)$ peut être majoré, pour les trois opérations précédentes, par une fonction linéaire. Nous proposons maintenant, une modélisation mathématique de $\Phi(q, ax + b)$, basée sur l'implantation des q émissions de messages à partir du sommet u en direction de ses q successeurs et de la disponibilité des canaux de communications pour acheminer les messages, une fois qu'ils ont été initialisés.

Lorsque le traitement des calculs élémentaires, associés au sommet u, est achevé, le processeur en charge de u doit effectuer au plus q émissions de messages. Avec la stratégie d'allocation, on émet $q - 1$ messages dans le cas des opérations $y = Ax$ et $LDL^T x = b$. En effet, les blocs de calculs élémentaires correspondant au traitement d'un même bloc de lignes vont réutiliser des morceaux de vecteurs stockés en mémoire centrale. Toutefois, lorsque ce processeur achève le traitement d'un bloc de lignes, il envoie le morceau de vecteur au processeur qui exécute le programme client, pour l'opération $y = Ax$ et pour la remontée dans l'opération $LDL^T x = b$. Pour les opérations $y = Ax$ et $LDL^T x = b$, on peut donc dire qu'un processeur va émettre au plus q messages. Pour l'opération $A = LDL^T$, on effectue, en vertu de la stratégie d'allocation, au plus $q-1$ émissions de messages. Lorsque les résultats définitifs de la factorisation $A = LDL^T$, pour une sous-matrice, sont disponibles, on passe simplement au traitement de la sous-matrice suivante. Ce n'est qu'au terme de la factorisation $A = LDL^T$ en parallèle que les sous-matrices ainsi traitées sont restituées à la demande du processeur qui exécute le programme client.

Ainsi, pour les opérations $y = Ax$ et $LDL^T x = b$, au plus q phases de "start-up" sont requises alors que pour l'opération $A = LDL^T$, on effectue au plus $q - 1$ phases de "start-up". Lorsque les messages sont initialisés dans des zones tampons, il peuvent être acheminés en parallèles suivant la multiplicité des canaux disponibles pour les communications. Tel est le cas des machines parallèles telles que la *PARAGON* et le *SP1*. Pour ces machines, nous proposons de modéliser $\Phi(q, ax + b)$ par:

$$\Phi(q, ax + b) = qb + ax \tag{4.69}$$

Pour le réseau de stations de travail, tous les message passent par le bus de communication et vont être acheminés un par un. Par conséquent, nous proposons d'évaluer $\Phi(q, ax + b)$ par:

$$\Phi(q, ax + b) = q(ax + b) \tag{4.70}$$

Nous allons maintenant évaluer une borne supérieure de $T_{com}(G(\mu))$ en fonction des nombres N et μ_{max} associés à la suite μ.

Pour l'opération $y = Ax$:

puisque $d(\mu) = 2N - 2$, on peut établir que $T_{com}(G(\mu))$ est majoré par:

$$T_{com}(G(\mu)) \leq (2N - 2)\Phi(2, \tau + 4\theta\mu_{max}) \tag{4.71}$$

On définit alors une majoration de $T_{com}(G(\mu))$ par:

$$T_{com}(G(\mu)) \leq \tau f_3(\mu_{max}, N) + \theta f_4(\mu_{max}, N) \tag{4.72}$$

avec,

$$f_3(\alpha, \beta) = 4\beta - 4 \tag{4.73}$$

et, pour la *PARAGON* et le *SP1*,

$$f_4(\alpha, \beta) = 4\alpha(\beta - 1) \tag{4.74}$$

pour le réseau de stations de travail,

$$f_4(\alpha, \beta) = 8\alpha(\beta - 1) \tag{4.75}$$

Pour l'opération $LDL^T x = b$:

$d(\mu) = 2N - 2$ à la descente et à la remontée. On établit donc que $T_{com}(G(\mu))$ est majoré par:

$$T_{com}(G(\mu)) \leq (4N - 4)\Phi(2, \tau + 2\theta\mu_{max}) \tag{4.76}$$

On définit alors une majoration de $T_{com}(G(\mu))$ par:

$$T_{com}(G(\mu)) \leq \tau f_3(\mu_{max}, N) + \theta f_4(\mu_{max}, N) \tag{4.77}$$

avec,

$$f_3(\alpha, \beta) = 8\beta - 8 \tag{4.78}$$

et, pour la *PARAGON* et le *SP1*,

$$f_4(\alpha, \beta) = 8\alpha(\beta - 1) \tag{4.79}$$

pour le réseau de stations de travail,

$$f_4(\alpha, \beta) = 16\alpha(\beta - 1) \tag{4.80}$$

Pour l'opération $A = LDL^T$:

$d(\mu) = 3N - 3$. Par conséquent, on a:

$$T_{com}(G(\mu)) \leq (3N - 3)\Phi(3, 2\tau + \theta(2\mu_{max}^2 + 3\mu_{max}) \tag{4.81}$$

On définit alors une majoration de $T_{com}(G(\mu))$ par:

$$T_{com}(G(\mu)) \leq \tau f_3(\mu_{max}, N) + \theta f_4(\mu_{max}, N) \tag{4.82}$$

avec,

$$f_3(\alpha, \beta) = 18\beta - 18 \tag{4.83}$$

et, pour la *PARAGON* et le *SP1*,

$$f_4(\alpha, \beta) = 3(\beta - 1)(2\alpha^2 + 3\alpha) \tag{4.84}$$

pour le réseau de stations de travail,

$$f_4(\alpha, \beta) = 9(\beta - 1)(2\alpha^2 + 3\alpha) \tag{4.85}$$

4.2.6.4 Etude des temps d'exécution parallèles:

D'un point de vue théorique, nous avons modélisé le temps d'exécution des algorithmes parallèles par:

$$T_{exe}(G(\mu)) = T_{cal}(G(\mu)) + T_{com}(G(\mu)) \tag{4.86}$$

Dans les sections 4.2.6.2 et 4.2.6.3, nous avons proposé, respectivement pour $T_{cal}(G(\mu))$ et $T_{com}(G(\mu))$, des majorations qui dépendent linéairement des paramètres $\overline{x}_1 = \tau^*$, $\overline{x}_2 = t_{scal}$, $\overline{x}_3 = \tau$ et $\overline{x}_4 = \theta$. On peut alors construire une fonction f telle que:

$$f(\alpha, \beta) = \sum_{i=1}^{4} \overline{x}_i f_i(\alpha, \beta) \tag{4.87}$$

Toutefois, f ne constituant pas une majoration fine de $T_{exe}(G(\mu))$, il est difficile d'estimer le temps d'exécution.

4.3 Evaluation des performances

4.3.1 Etude du problème de la plaque

4.3.1.1 Présentation du problème de la plaque

Dans *SYSTUS*, une plaque est un milieu dont les caractéristiques géométriques sont sa longueur, sa largeur et son épaisseur, comme l'illustre la figure (4.64). C'est un milieu élastique pour lequel on s'intéresse à l'étude du comportement dynamique, lors d'un mouvement de flexion. Les hypothèses de résolution de cette étude sont celles de l'élasticité linéaire sans amortissement et d'une épaisseur mince. Dans le cadre d'une discrétisation par éléments finis, on se ramène à une surface rectangulaire, en négligeant l'épaisseur de la plaque. La discrétisation d'une telle surface est illustrée dans la figure (4.65). Dans cette figure, nx (resp. ny) représente le nombre de mailles le long de l'axe x (resp y). Le nombre de mailles est égal à $nx * ny$ et le nombre de noeuds à $(nx+1)*(ny+1)$. On montre pour les équations de d'*Alembert* appliquées au problème de la plaque que chaque noeud du maillage est soumis à 3 degrés de liberté [Lare]. Le nombre total de degrés de liberté est égal à $n = 3*(nx+1)*(ny+1)$.

Le nombre n représente le nombre d'inconnues du problème d'éléments propre généralisé. En reprenant les notations de la section 4.1, relatives au stockage en mode skyline dans *DLANCB*, nous avons illustré dans le tableau de la figure (4.66), quelques exemples de profils de matrices, correspondant à des exemples tests $PLAQ_i$, pour $i = 1, \cdots, 6$.

Outre ces profils de matrice, les exemples tests $PLAQ_i$ se caractérisent également par le paramétrage du nombre de vecteurs de Lanczos par bloc, soit nv, du nombre de modes propres demandé, soit *nbmode* et du nombre de blocs de vecteurs de *Lanczos* à construire. Ces trois paramètres sont résumés dans le tableau de la figure (4.67). Pour tous les profils de matrice du tableau de la figure (4.66) plusieurs simulations de calcul de modes et fréquences propres par *DLANCB* ont été effectuées. Ces simulations nous permettent de constater que pour le problème de la plaque, il est difficile d'utiliser des tailles de blocs de vecteurs de *Lanczos* supérieures à 2. Au delà de deux vecteurs par blocs, on note la présence de vecteurs colinéaires. Il est alors difficile de mener la simulation à son terme.

4.3.1.2 Performances séquentielles

Le problème de la plaque permet d'évaluer les performances de *DLANCB* lorsque les milieux élastiques étudiés présentent des fréquences propres de multiplicité inférieure ou égal à 2. Ces

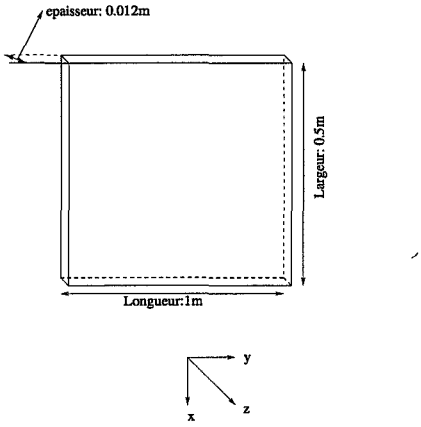

FIG. 4.64 – Géométrie de la plaque mince.

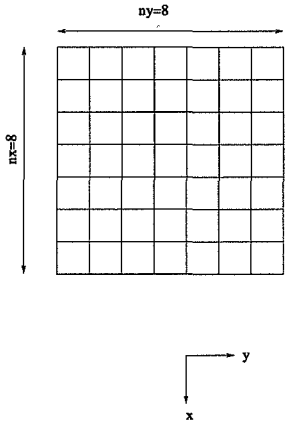

FIG. 4.65 – Maillage de la plaque.

	nx	*ny*	*n*	*nz*	*lbw*
$PLAQ_1$	*25*	*100*	7878	697272	159
$PLAQ_2$	*35*	*100*	12423	1773897	249
$PLAQ_3$	*50*	*50*	7803	1220031	159
$PLAQ_4$	*60*	*60*	11163	2080536	189
$PLAQ_5$	*70*	*70*	15123	3272541	219
$PLAQ_6$	*80*	*80*	19683	6415542	489

FIG. 4.66 – *Exemple de matrices skylines de grande taille pour le problème de la plaque.*

	nv	*itmax*	*nbmode*
$PLAQ_1$	2	16	15
$PLAQ_2$	1	49	30
$PLAQ_3$	2	8	7
$PLAQ_4$	2	8	7
$PLAQ_5$	2	20	20
$PLAQ_6$	2	20	20

FIG. 4.67 – *Paramètres en entrée pour le calcul de modes et fréquences propres par DLANCB. nv: nombre de vecteurs par blocs de Lanczos; itmax: nombre maximum de blocs de vecteurs; nbmode: nombre de vecteurs propres recherchés.*

évaluations sont réalisées sur une station de travail *SUN SPARCstation5*, présentée dans la section A.1.1. Dans le tableau de la figure (4.68), les chiffres de la colonne E illustrent le temps d'exécution, exprimé en secondes, des exemples tests $PLAQ_i$, pour $i = 1, \cdots, 6$. Pour l'exemple $PLAQ_6$, le temps d'exécution dépasse l'heure.

matrices	*valeurs en secondes*				
	A	B	C	D	E
$PLAQ_1$	7.9	31.1	321.6	209.0	569.6
$PLAQ_2$	14.6	129.4	1456.6	1364.4	2965.0
$PLAQ_3$	9.4	91.3	235.3	54.6	390.6
$PLAQ_4$	14.7	186.0	677.1	103.3	981.1
$PLAQ_5$	41.3	339.3	1553.1	910.4	2844.1
$PLAQ_6$	133.7	1127.1	3025.1	1465.2	5751.0

FIG. 4.68 – *Mesure des temps d'exécution séquentiels des phases de calcul de DLANCB pour la plaque*

Afin de détecter les phases de calcul les plus coûteuses, nous reprenons ici, les notations A à C, adoptées dans la section 4.1.3. Si on augmente le nombre de modes propres, on observe que les phases C et D sont les plus coûteuses. Si on raffine le maillage par éléments finis de la plaque, les phases B, C et D sont les plus coûteuses.

Les temps d'exécution élevés, observés dans la phase B se justifient par la complexité de la factorisation $A = LDL^T$. En effet, le nombre d'opérations scalaires effectuée dans une telle factorisation est de l'ordre de $n * lbw^2$, avec n et lbw dépendant des paramètres du maillage nx et ny.

En ce qui concerne la phase C, une analyse des performances des sous programmes qu'elle appelle a été réalisée. Comme illustré dans la figure (4.69), les appels répétitifs des sous programmes: *DLBM* (produit matrice skyline vecteur) et *DLBKM* (résolution de systèmes triangulaires à matrices skylines) sont les plus élevés. En effet, l'étude des appels des sous programmes dans la phase C, réalisée dans la section B.4, montre que sous certaines hypothèses, telles que des matrices présentant des valeurs de nz très grandes (nombre de termes dans le stockage skyline), l'exécution des algorithmes comme *DLBM* et *DLBKM* peuvent devenir coûteuses en nombres de calculs scalaires. Ainsi, lorsque le nombre de vecteurs par blocs de vecteurs de *Lanczos* est de 1 ou 2, la majeure partie du temps d'exécution est passée dans l'exécution des sous-programmes *DLBM* et *DLBKM*.

Toutefois, le nombre d'appels à ces sous-programmes est constant, quel que soit le numéro de l'itération (sauf en cas de régénération de vecteurs au cours de la factorisation de *Gramm-Schmidt*), tandis que celui des appels de sous programmes à matrices denses, pour les calculs d'orthogonalisation, augmentent avec le numéro de l'itération. Les sous programmes à matrices denses peuvent donc devenir coûteux en présence d'un grand nombre de blocs de vecteurs de *Lanczos* ou de blocs de tels vecteurs à $nv > 2$ vecteurs. Dans la section 4.3.2, le problème du cylindre encastré nous permet d'analyser le comportement de *DLANCB* lorsque les valeurs de nv sont proches des valeurs maximales utilisées en pratique.

	valeurs en secondes					
	$PLAQ_1$	$PLAQ_2$	$PLAQ_3$	$PLAQ_4$	$PLAQ_5$	$PLAQ_6$
DLBM	158.3	642.4	153.7	505.7	973.8	1947.4
DLBKM	35.8	139.4	31.4	54.2	209.6	513.4
Coût total du calcul itératif de C	309.2	1440.6	213.7	612.8	1495.4	2897.3

FIG. 4.69 – *Temps d'exécution, lors du calcul itératif des valeurs propres, de tous les appels des sous-programmes* DLBM *et* DLBKM *pour la plaque.*

4.3.2 Etude d'un cylindre encastré

4.3.2.1 Présentation du problème du cylindre encastré

Dans *SYSTUS*, un cylindre se modélise par un milieu élastique dont les caractéristiques géométriques sont son rayon, sa hauteur et son épaisseur (voir figure 4.70). Le problème à résoudre est celui de l'étude du comportement dynamique de ce cylindre lors d'une révolution autour d'un axe. Ce problème est résolu sous les hypothèses de l'élasticité linéaire sans amortissement et en négligeant l'épaisseur du cylindre. Afin de discrétiser les équations de d'*Alembert* et de construire un problème d'éléments propres généralisé, nous avons considéré deux classes de discrétisation par éléments finis fournies par *SYSTUS* (voir figures (4.71) et (4.72)).

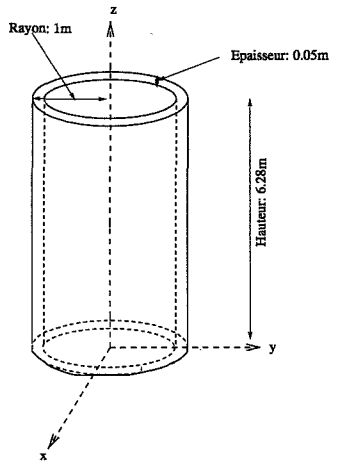

FIG. 4.70 – Géométrie d'un cylindre à coque mince.

Dans la figure (4.71), on réalise un quadrillage de la surface du cylindre par des quadrangles. Le long de l'axe z, le nombre de mailles est noté nh et, sur la circonférence du cercle contenu dans le plan passant par les axes x et y, le nombre de mailles est noté nc. Le nombre total de mailles est égal à $nc * nh$ et le nombre total de noeud à $nc * (nh + 1)$. Par construction, le

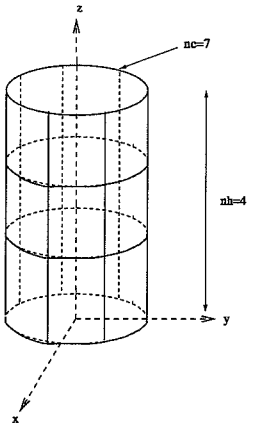

FIG. 4.71 – Maillage par des quadrangles.

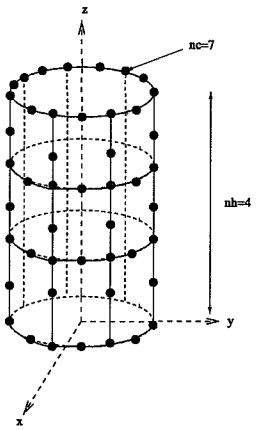

FIG. 4.72 – Maillage par des quadrangles ayant un noeud supplémentaire sur chaque face.

nombre de degrés de liberté par noeuds est égal à six. Ainsi, en faisant varier les valeurs nc et nh, on peut construire divers profils de matrice. Nous notons $CYL_{1,i}$, pour $i = 1, \cdots, 3$, les exemples de problèmes d'éléments propres que nous traitons. Dans le tableau 1 de la figure (4.73), nous avons illustré les caractéristiques des matrices associées à ces exemples.

Dans la figure (4.72), le maillage se déduit de celui de la figure (4.71) en rajoutant un point aux cotés des mailles. Ainsi, tout en laissant invariant le nombre de mailles, on augmente le nombre de noeuds qui est alors égal à $nc * (3nh + 2)$. Nous notons $CYL_{2,i}$, pour $i = 1, \cdots, 3$, les exemples de problèmes d'éléments propres que nous traitons. Dans le tableau 2 de la figure (4.73), nous avons illustré les caractéristiques des matrices associées à ces exemples. La largeur maximale de bande peut dépasser 1000.

Tableau 1

	nc	*nh*	*n*	*nz*	*lbw*
$CYL_{1,1}$	*50*	*50*	15300	4772250	318
$CYL_{1,2}$	*38*	*80*	18468	4458906	246
$CYL_{1,3}$	*60*	*60*	21960	8169300	378

Tableau 2

	nc	*nh*	*n*	*nz*	*lbw*
$CYL_{2,1}$	*20*	*20*	7440	2895456	522
$CYL_{2,2}$	*30*	*25*	13860	9205002	1074
$CYL_{2,3}$	*25*	*25*	11550	5544141	642

FIG. 4.73 – *Exemples de matrices skylines pour le problème du cylindre.*

Outre les profils des matrices *skylines*, qui contiennent de plus en plus de termes, le problème du cylindre nous permet d'atteindre les limites d'utilisation, en terme de nombre de vecteurs par blocs de *Lanczos*, de *DLANCB*. En effet, comme l'illustre le tableau de la figure (4.74), les problème $CYL_{i,j}$ permettent d'aborder le traitement de milieux élastiques pour lesquels la multiplicité des fréquences propres atteint 5. Pour bon nombre d'exemples d'application, cette borne supérieure est suffisante pour les études du comportement dynamique des structures.

matrices	*Données du problème*		
	nv	*itmax*	*nbmode*
$CYL_{1,1}$	5	8	13
$CYL_{1,2}$	5	13	20
$CYL_{1,3}$	4	23	30
$CYL_{2,1}$	3	38	20
$CYL_{2,2}$	5	9	10
$CYL_{2,3}$	4	8	8

FIG. 4.74 – *Nombre de mode et fréquences propres demandés.*

4.3.2.2 Performances séquentielles

Les performances séquentielles sont mesurées sur une station de travail *SUN SPARCstation5*, présentée dans la section A.1.1. Les mesures de temps de calcul sont illustrées dans le tableau de la figure (4.75). Les notations de A à E restent identiques à celles de la section 4.3.1.2. Nous avons observé des temps de calcul pouvant atteindre sept heures. Ces temps deviennent plus longs si on augmente le nombre de modes propres demandés. Pour ces exemples, les temps de la factorisation LDL^T peuvent atteindre l'heure de calcul. Le temps de calcul des valeurs propres peut atteindre les six heures de calcul.

Nous allons maintenant analyser les coûts des appels des sous-programmes exécutant le produit d'une matrice symétrique par un vecteur (*DLBM*)et la résolution de systèmes triangulaires inférieur, puis, supérieur (*DLBKM*). Ces temps d'exécution sont exposés dans la figure (4.76). Pour le problème du cylindre, les exemples considérés montrent qu'une majeure

	valeurs en secondes				
	A	*B*	*C*	*D*	*E*
$CYL_{1,1}$	65.7	721.3	3653.4	577.1	5017.5
$CYL_{1,2}$	54.4	516.9	8300.3	1701.7	10573.3
$CYL_{1,3}$	268.5	1505.2	18569.8	4717.6	25061.1
$CYL_{2,1}$	18.5	557.8	6937.0	586.9	8100.2
$CYL_{2,2}$	439.2	3370.8	11578.6	405.6	15794.2
$CYL_{2,3}$	58.6	1300.9	3357.3	219.5	4936.3

FIG. 4.75 – *Mesures des temps d'exécution séquentiels des phases de calcul de DLANCB pour le cylindre.*

partie du temps d'exécution dans la partie itérative de C est consacrée aux appels répétitifs des sous-programmes *DLBM* et *DLBKM*. Cependant, lorsque le nombre de degré de liberté du problème (c'est à dire n) est grand, les méthodes d'orthogonalisation et de réorthogonalisation deviennent de plus en plus coûteuses. En poursuivant le processus itératif, ces temps vont être de plus en plus élevés. Ceci s'illustre à travers les écarts entre la somme des deux premières lignes et la dernière du tableau de la figure (4.76) pour les exemples $CYL_{1,3}$ et

$CYL_{2,1}$.

	valeurs en secondes					
	$CYL_{1,1}$	$CYL_{1,2}$	$CYL_{1,3}$	$CYL_{2,1}$	$CYL_{2,2}$	$CYL_{2,3}$
DLBM	2641.1	5834.9	12475.0	4778.6	6421.2	2550.9
DLBKM	310.1	639.4	2619.6	525.3	3503.0	288.1
Coût total du calcul itératif de C	3283.4	7946.7	17745.6	6796.9	10302.6	3000.3

FIG. 4.76 – *Temps d'exécution, lors du calcul itératif des valeurs propres, de tous les appels des sous-programmes* DLBM *et* DLBKM *pour le cylindre*

4.3.3 Mesures sur le réseau de stations de travail

Il s'agit d'un réseau de station de travail, de type *SUN SPARCstation4*, dont nous présentons les principales caractéristiques dans la section A.1. Ces stations de travail sont interconnectées via un réseau *Ethernet* à $10Mb/s$, entre elles et à une station de travail *SUN SPARCstation5* sur laquelle est installé le code *SYSTUS*. Grâce au couplage des bibliothèques à passage de message *PVM* et *BLACS*, ce réseau peut être vu comme une machine parallèle virtuelle sur laquelle a été implanté le modèle *client/serveur* précédemment défini dans la section 4.2.2. Les évaluations de performances du client/serveur ont été réalisées sur les exemples tests: $PLAQ_1$, $CYL_{1,3}$ et $CYL_{2,2}$. Pour les simulations effectuées, nous avons: proposé un choix de partitionnement par blocs, présenté les temps d'exécution parallèle du client serveur et discuté des perspectives d'amélioration des performances.

4.3.3.1 Partitionnement par blocs

Le partitionnement par blocs est paramétré par une suite $\mu = (\mu_i)_{1 \leq i \leq N}$ et conduit à la génération d'un matrice de cellules rangées de façon identique au rangement des sous matrices de la matrice $A(\mu) = (A_{I,J}(\mu))_{1 \leq I,J \leq N}$, comme il a été précédemment établi dans la section 4.2.5. Pour les besoins des simulations, le choix de μ est tel que:

$$\mu_i = \begin{cases} \mu_{max} \text{ si} & 1 \leq i < N \\ n - (N-1) * \mu_{max} \text{ sinon} & \end{cases} \tag{4.88}$$

Ce choix conduit à une matrice de cellules dont la largeur maximale de bande est définie par:

$$lbw(\mu) \approx \frac{lbw + \mu_{max}}{\mu_{max}} \tag{4.89}$$

Pour une telle matrice de cellules, les lemmes (3) et (4) permettent de définir le nombre minimum de processeur, soit $P_{sys}(\mu)$, suffisant pour que la stratégie d'allocation précédemment décrite dans la section 4.2.3.4, préserve le temps d'exécution des sommets du plus long chemin du graphe $G(\mu) = (D(\mu), E(\mu))$. Un algorithme permet d'allouer la matrice de cellules ainsi générée à $P \geq P_{sys}(\mu)$ processeurs.

Dans ce qui suit, chaque exemple est simulé quatre fois, pour quatre choix de (μ_{max}, P), résumés dans les tableaux des figures (4.77), (4.81) et (4.85). Par exemple dans le tableau de la figure (4.77), la simulation de $PLAQ_6$ est remplacée par celle de quatre sous exemples tests, notés $PLAQ_{6,i}$, avec $i = 1, \cdots, 4$. Pour le tableau de la figure (4.77) (respectivement (4.81) et (4.85)), les couples associés à $PLAQ_{6,i}$ (respectivement $CYL_{1,3,i}$ et $CYL_{2,2,i}$), sont les choix de (μ_{max}, P) pour la factorisation LDL^T (*XDGAUS*), le produit matrice vecteur (*DLBM*) et les résolutions de systèmes $LDL^T x = b$ (*DLBKM*). Pour les sous exemples $PLAQ_{6,i}$ (respectivement $CYL_{1,3,i}$ et $CYL_{2,2,i}$), les paramètres de la recherche de modes et fréquences propres restent ceux de $PLAQ_6$ (respectivement $CYL_{1,3}$ et $CYL_{2,2}$), décrits par les tableaux des figures (4.67) et (4.74). Dans ce qui suit, les lettres A à D désignent les phases de calcul de *DLANCB*, E désigne le surcoût lié à la mise en oeuvre: du partitionnement, de l'ordonnancement et de l'allocation, enfin, la lettre F désigne le temps d'exécution du client/serveur.

4.3.3.2 Temps d'exécution parallèle
Cas de l'exemple $PLAQ_6$

Etant donnés les sous exemples de $PLAQ_6$ qui sont définis par $PLAQ_{6,i}$, pour $i = 1, \cdots, 4$, et les choix de (μ_{max}, P) illustrés dans le tableau de la figure (4.77), le tableau de la figure (4.78) résume les mesures de performances. Les temps d'exécution globaux, illustrés par F,

	Découpage par blocs		
	XDGAUS	*DLBM*	*DLBKM*
$PLAQ_{6,1}$	(65, 16)	(75, 16)	(75, 16)
$PLAQ_{6,2}$	(70, 16)	(75, 16)	(75, 16)
$PLAQ_{6,3}$	(75, 13)	(80, 13)	(80, 13)
$PLAQ_{6,4}$	(80, 13)	(85, 13)	(85, 13)

FIG. 4.77 – *Choix d'un découpage et nombre de processeurs pour l'exemple* $PLAQ_6$.

montrent une quasi stagnation des performances malgré la parallélisation. Pour comprendre cette quasi stagnation, considérons le modèle client/serveur précédemment défini dans la section 4.2.2, et analysons les chiffres associés à B, C et E.

La phase B consiste à faire exécuter en parallèle, la factorisation LDL^T, par le serveur. Le serveur améliore les traitements de B d'un facteur de 3.

Pour la phase C, le modèle client/serveur améliore les performances de 750 secondes. Les causes de cette légère amélioration peuvent se justifier en analysant les performances des serveurs parallèles qui y sont appelés. Comme illustré dans la figure (4.79), le serveur parallèle qui exécute *DLBM* (le produit matrice skyline vecteur), permet de diviser quasiment par deux, les temps de calcul, tandis que pour le serveur parallèle qui exécute *DLBKM* (résolution de $LDL^T x = b$), les performances sont celles d'une exécution séquentielle.

Enfin, la phase E peut être décomposée en une partie purement calculatoire et en une partie purement dédiée aux communications. En effet, étant donnés les paramètres du partitionnement par blocs, on construit une matrice de cellules. Les temps mis pour construire

	valeurs en secondes				
	$PLAQ_6$	$PLAQ_{6,1}$	$PLAQ_{6,2}$	$PLAQ_{6,3}$	$PLAQ_{6,4}$
A	133.7	85.3	98.0	99.4	100.8
B	1127.1	354.2	357.4	391.5	386.9
C	3025.1	2279.9	2244.3	2273.2	2238.9
D	1465.2	1522.7	1558.9	1545.1	1546.5
E	0.0	1081.5	887.8	805.7	857.7
F	5751.0	5323.6	5146.4	5114.9	5130.8

FIG. 4.78 – *Mesures des temps d'exécution pour le problème $PLAQ_6$ sur réseaux de station de travail.*

	valeurs en secondes				
	$PLAQ_6$	$PLAQ_{6,1}$	$PLAQ_{6,2}$	$PLAQ_{6,3}$	$PLAQ_{6,4}$
DLBM	1947.2	989.1	981.4	998.2	978.3
DLBKM	513.4	388.5	387.4	401.9	397.7
Partie itérative de C	2897.3	1936.5	1931.3	1963.7	1945.3

FIG. 4.79 – *Temps d'exécution, sur réseaux de station, de tous les appels DLBM et DLBKM dans la partie itérative de la phase C.*

cette matrice de cellules, avant les phases B et C, sont les valeurs associées à E_1, dans le tableau de la figure (4.80). Si ces valeurs sont relativement les mêmes pour les exemples $PLAQ_{6,i}$, pour $i = 2, \cdots, 4$, pour $PLAQ_{6,1}$ le temps élevé se justifie par le nombre de cellules à construire. Toutefois, E_1 reste négligeable comparé aux valeurs prises par E_2 (distribution des cellules aux processeurs avant la phase B), E_3 (récupération des résultats de la factorisation LDL^T) et E_4 (distribution des cellules aux processeurs avant la phase C). Les coûts élevés observés en E_2, E_3 et E_4 se justifient par le caractère centralisé, au niveau du client, de la résolution du partitionnement, de l'ordonnancement et de l'allocation.

Au total, pour le problème $PLAQ_6$, le recours au modèle client/serveur permet simplement de préserver les temps d'exécution séquentiels. Ceci est du aux faibles accélérations, fournies par les serveurs parallèles et aux pré-traitements que nécessitent ces serveurs. Nous allons dans ce qui suit, étudier les performances des exemples tests $CYL_{1,3}$ et $CYL_{2,2}$ qui fournissent des matrices skylines à profils relativement grands.

Cas de l'exemple $CYL_{1,3}$

C'est le plus gourmand en temps de calcul. Les simulations réalisées atteignent sept heures de calcul! Pour cet exemple du problème du cylindre, nous avons considéré quatre sous exemples tests: $CYL_{1,3,i}$, pour $i = 1, \cdots, 4$. Chaque sous exemple correspond à des choix de (μ_{max}, P) que nous résumons dans le tableau de la figure (4.81). Le choix des valeurs de μ_{max} nous permet d'analyser les accélérations obtenues lorsque μ_{max} augmente tout en utilisant un

	valeurs en secondes				
	$PLAQ_6$	$PLAQ_{6,1}$	$PLAQ_{6,2}$	$PLAQ_{6,3}$	$PLAQ_{6,4}$
E_1	0.0	13.3	4.0	4.8	4.8
Distribution	0.0	270.7	178.3	119.4	197.7
Récupération	0.0	491.0	401.3	413.1	386.4
Redistribution	0.0	306.5	304.2	268.4	268.7
Phase E	0.0	1081.5	887.8	805.7	857.7

FIG. 4.80 – *Temps de distribution des données aux processeurs: cas de l'exemple $PLAQ_6$ sur réseaux de stations de travail.*

nombre de processeurs suffisant pour la stratégie d'allocation.

	Découpage par blocs		
	XDGAUS	*DLBM*	*DLBKM*
$CYL_{1,3,1}$	(60, 16)	(70, 16)	(70, 16)
$CYL_{1,3,2}$	(70, 13)	(80, 13)	(80, 13)
$CYL_{1,3,3}$	(80, 11)	(90, 11)	(90, 11)
$CYL_{1,3,4}$	(100, 8)	(110, 8)	(110, 8)

FIG. 4.81 – *Choix d'un découpage et nombre de processeurs pour l'exemple $CYL_{1,3}$.*

Nous avons résumé dans le tableau de la figure (4.82), les mesures des temps d'exécution concernant l'exemple $CYL_{1,3}$. La lecture de ce tableau procède de la même façon que dans le cas $PLAQ_6$. Pour la première colonne, les mesures sont celles observées et présentées dans la section 4.3.2. Pour les phases de calcul qui ont été parallélisées, soient B et C, on observe des diminutions des temps de calcul atteignant des facteurs de 4. Pour la phase B, des accélérations de l'ordre de 4.2 peuvent être atteintes. Cependant, lorsque μ_{max} croit, on note une augmentation du temps d'exécution de la factorisation en parallèle. Pour la phase C, on note une accélération de 2.9 pour l'exemple $CYL_{1,3,2}$, c'est à dire pour $\mu_{max} = 80$. Pour $\mu_{max} = 70$ ou $\mu_{max} > 80$, les accélérations observées sont plus faibles.

Nous avons ensuite analysé le coût global des appels aux fonctions *DLBM* et *DLBKM* dans la partie itérative de la phase C (voir le tableau de la figure (4.83)). La réduction du temps de calcul peut atteindre des facteurs de 8 pour les appels de *DLBM* et de 3.6 pour les appels de *DLBKM*. Toutefois, l'accélération observée au niveau de la partie itérative de C est de l'ordre de 3.

Analysons maintenant le surcoût produit par la phase E. Nous avons observé des surcoûts, par rapport aux traitements séquentiels pouvant atteindre la demi-heure. Cependant, avec des simulations de l'ordre de quatre heures, ces temps sont "acceptables". Dans le tableau de la figure (4.84), nous illustrons les temps de construction et de distribution des données. On note que la majeure partie du temps est passée dans les échanges "client/serveur" portant sur les coefficients des matrices skylines. Ces temps, par rapport à ceux observés dans l'exemple

	valeurs en secondes				
	$CYL_{1,3}$	$CYL_{1,3,1}$	$CYL_{1,3,2}$	$CYL_{1,3,3}$	$CYL_{1,3,4}$
A	268.5	304.33	305.8	313.4	286.7
B	1505.2	452.5	602.6	653.1	362.3
C	18569.8	6853.4	6301.2	6467.5	4623.4
D	4717.6	4621.2	4722.8	4968.3	4472.7
E	0.0	1651.0	1095.0	971.6	728.4
F	25061.1	13882.4	13027.4	13373.9	10473.5

FIG. 4.82 – *Mesures des temps d'exécution pour le problème* $CYL_{1,3}$ *sur réseaux de station de travail.*

	valeurs en secondes				
	$CYL_{1,3}$	$CYL_{1,3,1}$	$CYL_{1,3,2}$	$CYL_{1,3,3}$	$CYL_{1,3,4}$
DLBM	12475.0	2264.8	2211.4	2248.7	1557.8
DLBKM	2619.6	1152.9	1102.8	1147.7	729.4
Partie itérative de C	17745.6	5923.3	5770.3	5886.3	4113.9

FIG. 4.83 – *Temps d'exécution, sur réseaux de station, de tous les appels DLBM et DLBKM dans la partie itérative de la phase C.*

$PLAQ_6$, sont un peu plus grands. Nous terminons cet étude par l'exemple $CYL_{2,2}$ qui nous fournit les profils des matrices ayant de plus grandes largeurs de bande par rapport aux exemples que nous avons testé.

Cas de l'exemple $CYL_{2,2}$

Pour cet exemple du problème du cylindre, nommons $CYL_{2,2,i}$, pour $i = 1, \cdots, 4$ les sous exemples correspondant aux choix de (μ_{max}, P). Ces choix sont résumés dans le tableau de la figure (4.85). Le choix des valeurs de μ_{max} nous permet d'analyser les accélérations obtenues lorsque les matrices traitées ont une grande largeur de bande. Dans l'exemple $CYL_{2,2}$, les temps d'exécution de *XDGAUS*, *DLBM* et *DLBKM* sont les plus élevés.

Nous avons résumé dans le tableau de la figure (4.86), les mesures des temps d'exécutions concernant les sous exemples de $CYL_{2,2}$. Pour les phases de calcul qui ont été parallélisées, soient B et C, on observe des diminutions des temps d'exécution atteignant des facteurs de 6. Pour la phase B, des accélérations de l'ordre de 6 peuvent être atteintes. Cependant, lorsque μ_{max} croit, on note une augmentation du temps d'exécution de la factorisation en parallèle. Pour la phase C, on note une accélération de 6.4.

4.3.3.3 Discussion et analyse

En remplaçant simplement les appels des sous-programmes *XDGAUS*, *DLBM* et *DLBKM* respectivement par *CL_XDGAUS*, *CL_ DLBM* et *CL_DLBKM*, nous avons observé une diminution des temps de calculs sur les exemples tests $CYL_{1,3}$ et $CYL_{2,2}$. Les accélérations des

	valeurs en secondes				
	$CYL_{1,3}$	$CYL_{1,3,1}$	$CYL_{1,3,2}$	$CYL_{1,3,3}$	$CYL_{1,3,4}$
Découpage et allocation	0.0	44.7	36.0	4.6	4.4
Distribution	0.0	388.0	233.2	145.0	136.8
Récupération	0.0	544.3	453.7	522.4	297.1
Redistribution	0.0	674.0	372.1	299.6	290.1
Phase E	0.0	1651.0	1095.0	971.6	728.4

FIG. 4.84 – *Temps de distribution des données aux processeurs: cas de l'exemple $CYL_{1,3}$ sur réseaux de stations de travail.*

	Découpage par blocs		
	XDGAUS	*DLBM*	*DLBKM*
$CYL_{2,2,1}$	(90, 32)	(100, 32)	(100, 32)
$CYL_{2,2,2}$	(105, 27)	(100, 27)	(100, 27)
$CYL_{2,2,3}$	(110, 27)	(95, 27)	(95, 27)
$CYL_{2,2,4}$	(120, 25)	(105, 25)	(105, 25)

FIG. 4.85 – *Choix d'un découpage et nombre de processeurs pour l'exemple $CYL_{2,2}$.*

	valeurs en secondes				
	$CYL_{2,2}$	$CYL_{2,2,1}$	$CYL_{2,2,2}$	$CYL_{2,2,3}$	$CYL_{2,2,4}$
A	439.2	447.0	412.3	453.2	402.7
B	3370.8	553.6	566.1	579.9	739.8
C	11578.6	1875.7	1810.0	2867.6	1828.1
D	405.6	359.8	373.8	362.5	362.4
E	0.0	1188.5	1147.2	1482.6	1081.1
F	15794.2	4424.6	4309.4	5745.8	4414.1

FIG. 4.86 – *Mesures des temps d'exécution pour le problème $CYL_{2,2}$ sur réseaux de stations de travail.*

temps d'exécution observées, sur le réseau de stations de travail, peuvent atteindre des facteurs de l'ordre de 3, pour le module *DLANCB*. Ces performances peuvent être améliorées en utilisant un réseau d'interconnexion beaucoup plus performant comme nous le verrons dans la section 4.3.4. Nous discutons maintenant de trois directions selon lesquelles les performances du client serveur peuvent être optimisées, quelle que soit la machine parallèle utilisée.

Première direction:

il s'agit de poursuivre le travail de remplacement des appels des sous-programmes *DLBM* et *DLBKM* respectivement par *CL_DLBM* et *CL_DLBKM* dans le module *DLANCB*. On peut ainsi libérer de l'espace mémoire au niveau du processeur qui exécute le programme client. L'espace ainsi libéré peut être utilisé pour le stockage des blocs de vecteurs de *Lanczos* en mémoire centrale. On évite de ce fait, les accès systématiques en mémoire auxiliaire lors des opérations d'orthogonalisation et de réorthogonalisation. Puisque de telles opérations s'expriment à l'aide des divers niveaux des *BLAS*, on dispose d'une source intéressante d'optimisation des calculs sur des matrices pleines. Dans cette première direction, l'assemblage des matrices reste de la responsabilité du programme client. Avec un nombre de termes de plus en plus grand dans les profils des matrices skyline, il convient de disposer de capacités de mémorisation adéquates. D'où une deuxième direction pour les optimisations futures que nous proposons.

Deuxième direction:

il s'agit de réduire les coût des communications engendrées par les opérations de distribution et de redistribution des coefficients des matrices skylines lorsque leurs tailles sont croissantes. Ces deux opérations impliquent des échanges du type *one-to-all* ou *all-to-one*. De tels échanges s'imposent dans la mesure où c'est par le biais du programme client que sont spécifiés les paramètres du découpage par bloc. L'idée que nous proposons consiste à simuler "partiellement" la construction de la structure de donnée de type *MatriceCellule* au niveau du client. Nous entendons par simulation "partielle" la construction d'une variable de type *MatriceCellule* dans laquelle on ne remplit pas explicitement l'espace mémoire destiné au rangement des coefficients des matrices. Pour chacune de ces nouvelles cellules, il faudrait rajouter les informations facilitant l'assemblage en "parallèle" de la sous-matrice correspondante. Ainsi, à partir de ces informations, l'assemblage peut être effectué, sous-matrice par sous-matrice, par le serveur. Le problème qui est posé est alors celui de la présentation des résultats de la discrétisation par la méthode des éléments finis dans *SYSTUS*. Avec la structure actuelle du module *DLANCB*, les résultats de la discrétisation sont dans des fichiers accessibles uniquement au niveau du client. Peut-on réorganiser ces fichiers de façon à faciliter un tel assemblage? De la réponse à cette question, dépend la minimisation du surcoût engendré par la distribution des coefficients des matrices.

En ce qui concerne la redistribution des coefficients au terme de la factorisation LDL^T, le problème se complique puisqu'il faut redécouper la matrice. En effet, si les découpages correspondant au produit matrice skyline vecteur et aux résolutions de systèmes triangulaires à matrices skylines peuvent être choisis à l'identique, ils sont toutefois différents de celui qui a été choisi pour la factorisation LDL^T. Quand bien même les découpages seraient tous identiques, se poserait alors le problème de l'allocation. Certes, on peut réutiliser la même allocation pour les trois opérations à matrices skyline dans *DLANCB*. Cette allocation peut être soit celle de la factorisation LDL^T, soit celle du produit matrice skyline vecteur. Dans

le cas où on utilise l'allocation de la factorisation LDL^T, on note l'exécution de phases de communication supplémentaires, pour le produit matrice skyline vecteur et les résolution de systèmes triangulaires, lors du traitement des blocs de lignes. Lorsque l'on applique l'allocation du produit matrice skyline à la factorisation LDL^T, on augmente le nombre de calculs à effectuer et les performances de cette dernière opération peuvent être débridées. Les questions que nous nous posons sont donc les suivantes:

1 : Doit-on utiliser un découpage unique pour toutes les opérations, si oui lequel?

2 : Doit-on utiliser la même allocation pour toutes les opérations, si oui laquelle?

3 : Peut-on envisager de redécouper les matrices après la factorisation LDL^T? Si oui comment assembler localement les sous-matrices?

L'analyse des performances de l'implantation des algorithmes parallèles nous permettra de répondre aux deux première questions. Quant à la dernière, elle nécessite d'imaginer une redistribution des sous-matrices sans intervention du client.

Troisième direction:

il s'agit d'accélérer les temps d'exécution en facilitant le traitement en concurrence de plusieurs sous-programmes lors du calcul des valeurs propres. Implanter une telle amélioration conduit à une représentation sous forme de graphe, de tous les calculs au cours du déroulement des itérations de la méthode de *Lanczos*. La construction d'un tel graphe conduit à une superposition des graphes des différentes opérations exécutées à chaque itération. La modélisation d'un tel graphe est longue et coûteuse en temps de calcul et en espace mémoire.

L'idée de notre approche pour faciliter le traitement en parallèle de plusieurs sous-programmes se fonde sur la remarque suivante: dans la méthode de *Lanczos*, une matrice rectangulaire représentant un bloc de vecteur sera transformée au cours d'une itération jusqu'à la production d'un bloc de vecteur de *Lanczos*. Les différentes étapes de cette transformation sont représentées par les sous-programmes suivant (voir l'annexe B) : *DLBDQR*, *DLBREO*, *DLBKM*, *DPIVOT*, *DLBORT* et *REORT*.

Chacun de ces sous-programmes peut constituer un étage d'un pipeline s'il est parallélisé suivant la méthodologie du chapitre 3. Ce pipeline doit être simulé au niveau du serveur. Pour ce faire, nous proposons d'utiliser certains processeurs pour les traitements de matrices skylines et d'autre pour le traitement des orthogonalisations et réorthogonalisations. Cependant, il est difficile, compte tenu des classes de problèmes que nous ciblons, d'utiliser efficacement un tel pipeline. Par exemple, avant d'exécuter *DLBREO*, il convient de vérifier que la factorisation de *Gramm-Schmidt*, exécutée par *DLBDQR*, a produit des blocs de vecteurs linéairement indépendants. En outre, il faudrait définir un système d'équations de récurrence pour décrire la circulation des données entre les étages du pipeline.

4.3.4 Mesures des temps d'exécution sur la PARAGON et le SP1

Dans cette section, nous discutons des mesures de performances observées en validant les serveurs parallèles sur le *SP1* et la *PARAGON*. Pour une présentations détaillée des environnements matériels et logiciels pour la mise en oeuvre, nous renvoyons le lecteur intéressé aux sections A.1 et A.2. Nous ne discutons pas ici, des surcoûts de la construction des structures de données car, une telle construction procédant de façon identique à celle de la section 4.2.5,

les surcoûts: du partitionnement, de l'ordonnancement et de l'allocation s'analysent de la même manière que dans la section 4.3.3. Dans ce qui suit, nous présentons successivement les performances séquentielles, les performances parallèles, et faisons une étude comparative des évaluations de performances théoriques et expérimentales.

4.3.4.1 Temps d'exécution séquentiels sur le SP1 et la PARAGON

Nous avons mesuré les temps d'exécution séquentielle des opérations suivantes:

- ⋆ la factorisation $A = LDL^T$ stockée en mode skyline (*XDGAUS*);
- ⋆ le produit matrice vecteur $y = Ax$ (*DLBM*);
- ⋆ la résolution de système linéaire du type $LDL^Tx = b$ (*DLBKM*).

Ces temps d'exécution ont été mesuré à l'aide de la fonction *DCPUTIME00* (implantée dans avec les *BLACS*). Plusieurs mesures ont été effectuées pour *XDGAUS*, *DLBM* et *DLBKM*, et ont montré des valeurs qui fluctuaient autour de valeurs moyennes. Les valeurs moyennes ainsi obtenues sont résumées dans le tableau de la figure (4.87).

	Temps d'exécution sur la PARAGON			
	$PLAQ_1$	$PLAQ_5$	$CYL_{1,1}$	$CYL_{2,3}$
$A = LDL^T$	18.65	215.8	458.45	843.55
$y = Ax$	0.38	3.75	4.63	5.36
$LDL^Tx = b$	0.83	3.85	5.61	6.52
	Temps d'exécution sur le SP1			
	$PLAQ_1$	$PLAQ_5$	$CYL_{1,1}$	$CYL_{2,3}$
$A = LDL^T$	30.21	334.20	701.59	1271.94
$y = Ax$	0.58	2.71	3.94	4.59
$LDL^Tx = b$	1.43	6.74	9.79	11.36

FIG. 4.87 – *Mesures des temps d'exécution séquentielle, exprimés en secondes, sur le SP1 et la PARAGON*

Pour une architecture donnée, on observe que les temps d'exécution augmentent avec le nombre de termes nz. L'opération $A = LDL^T$ est de loin, la plus coûteuse. Bien que les opérations $y = Ax$ et $LDL^Tx = b$ aient une complexité en nombre d'opérations scalaires du même ordre pour chaque test considéré, on note un "bon comportement" des processeurs sur $y = Ax$, par rapport à $LDL^Tx = b$. Ces différences peuvent s'expliquer par les modes d'accès à la mémoire, la structure du pipeline, les optimisations pour les calculs sur les nombre flottants, etc. Une étude de l'architecture dans chaque cas est nécessaire afin de comprendre les valeurs des temps d'exécution ainsi observés. Une telle étude va au delà de nos objectifs.

4.3.4.2 Temps d'exécution parallèle sur le SP1 et la PARAGON

Le choix du partitionnement par blocs et celui du nombre de processeurs P procédant de la même manière que dans la section 4.3.3.1, nous reprenons ici, toutes les notations de la section précédemment citée. Ainsi, on se ramène au choix d'un couple (μ_{max}, P), où μ_{max}

est lié au quadrillage des axes, et P est le nombre de processeurs requis par le client/serveur. Par soucis de simplicité, puisque le nombre de processeurs utilisés par un serveur est égal à $P-1$, nous avons choisi $P = P_{sys}(\mu) + 1$, où $P_{sys}(\mu)$ est le nombre minimum de processeurs requis par la stratégie d'allocation de la section 4.2.3.4. Afin de mesurer les performances, nous avons *DWALLTIME00* (implantée dans les *BLACS*). La fonction *DWALLTIME00* est construite autour de la fonction *UNIX gettimeofday* qui mesure un temps écoulé prenant en compte, les aléas de la machine.

	Temps d'exécution sur la PARAGON			
$P_{sys}(\mu)$	$PLAQ_1$	$PLAQ_5$	$CYL_{1,1}$	$CYL_{2,3}$
26	3.28	10.29	16.38	39.83
22	3.24	11.49	18.30	45.37
19	3.86	13.23	21.25	54.69
15	3.94	14.02	24.23	69.80
13	4.51	16.96	31.92	74.96
	Temps d'exécution sur le SP1			
$P_{sys}(\mu)$	$PLAQ_1$	$PLAQ_5$	$CYL_{1,1}$	$CYL_{2,3}$
15	6.50	46.02	111.85	127.57
13	6.87	38.66	118.96	140.38
10	9.33	26.84	84.07	212.87
7	9.46	37.54	94.77	296.20
5	10.76	62.38	126.01	369.62

FIG. 4.88 – *Mesures des temps d'exécution parallèle, exprimés en secondes, sur le SP1 et la PARAGON pour l'opération* $A = LDL^T$.

Dans le tableau de la figure (4.88), nous avons résumé les temps d'exécution parallèle observés sur la *PARAGON* et le *SP1* pour l'opération $A = LDL^T$. Ces chiffres donnent la valeur moyenne des temps d'exécution de 10 simulations consécutives. Si nous comparons, pour chaque exemple, ces chiffres et ceux exposés dans le tableau de la figure (4.87), on observe globalement que la parallélisation améliore les temps d'exécution. Pour la *PARAGON*, on observe qu'en augmentant le nombre de processeurs on produit une diminution du temps de d'exécution. Pour le *SP1*, en augmentant le nombre de processeurs, nous avons observé pour les problèmes $PLAQ_5$ et $CYL_{1,1}$, une dégradation du temps d'exécution. D'autres simulations sont en cours pour confirmer ou infirmer cette observation. Enfin, dans le tableau de la figure (4.89), nous illustrons les facteurs d'accélération obtenus, en divisant pour chaque exemple test, les chiffres du tableau de la figure (4.87), par ceux du tableau de la figure (4.88).

Dans le tableau de la figure (4.90), nous illustrons les temps de calcul expérimentaux de l'opération $y = Ax$ sur le *SP1* et la *PARAGON*. A la lecture de ces résultats, nous avons constaté qu'une augmentation de $P_{sys}(\mu)$ n'impliquait pas, de façon systématique, celle des temps d'exécution. Si nous comparons ces chiffres à ceux qui sont exposés dans le tableau de la figure (4.87), on n'observe pas toujours des améliorations des temps d'exécution. Cependant, moyennant un bon choix de la suite μ, les temps expérimentaux peuvent devenir

	Temps d'exécution sur la PARAGON			
$P_{sys}(\mu)$	$PLAQ_1$	$PLAQ_5$	$CYL_{1,1}$	$CYL_{2,3}$
26	5.69	20.97	27.99	21.18
22	5.76	18.78	25.05	18.59
19	4.83	16.31	21.25	15.42
15	4.73	15.39	18.92	12.09
13	4.31	12.72	14.36	11.13
	Temps d'exécution sur le SP1			
$P_{sys}(\mu)$	$PLAQ_1$	$PLAQ_5$	$CYL_{1,1}$	$CYL_{2,3}$
15	4.65	7.26	6.27	9.97
13	4.40	8.64	5.90	9.06
10	3.23	12.45	8.34	5.97
7	3.20	8.90	7.40	4.29
5	2.81	5.36	5.57	3.44

FIG. 4.89 – *Mesures des facteurs d'accélération expérimentaux sur le SP1 et la PARAGON pour l'opération* $A = LDL^T$.

très favorables. En outre, pour des matrices ayant des profils de plus en plus larges, les améliorations sont de plus en plus intéressantes. Du point de vue expérimental et pour les tests effectués, nous remarquons qu'avec des suites μ telles que $P_{sys}(\mu) \leq 4$, les temps d'exécution semblent être les plus favorables. Dans le tableau de la figure (4.91), nous illustrons les temps d'exécution expérimentaux de l'opération $LDL^T x = b$. Un lecture de ces résultats expérimentaux conduit aux mêmes remarques que celles de l'opération $y = Ax$.

Notons que le signe – désigne des mesures non encore effectuées.

4.3.4.3 Discussion et analyse

Si on compare les temps d'exécution séquentiels et parallèles qui sont observés expérimentalement, on constate dans certains cas, des accélérations super-linéaires, alors que par construction des stratégies de partitionnement, d'ordonnancement et d'allocation, on ne peut théoriquement diviser le nombre total de calculs scalaires que par des facteurs inférieurs au nombre de processeurs utilisés. Dans d'autres cas, alors que l'on utilise des tailles de blocs de plus en plus fines, permettant de diviser le nombre total d'opérations scalaires par des facteurs de plus en plus proche du nombre de processeurs, nous observons des temps d'exécution de plus en plus prohibitifs bien que la stratégie d'allocation utilise un nombre de processeurs suffisant.

Nous présentons dans cette section, une étude comparative entre les temps d'exécution expérimentaux observé sur la *PARAGON* et les valeurs d'une estimation a priori du temps d'exécution parallèle pour la *PARAGON*. Deux exemples tests sont choisis pour ce faire: $PLAQ_1$ et $PLAQ_5$. Pour ces exemples tests, nous étudions simplement le cas des opérations $y = Ax$ et $A = LDL^T$. Une telle étude peut ensuite être généralisée aux autres exemples tests et architectures. Enfin, notons que le choix du partitionnement par blocs et celui du nombre de processeurs P procèdent de la même manière que dans la section 4.3.3.1.

	Temps d'exécution sur la PARAGON			
$P_{sys}(\mu)$	$PLAQ_1$	$PLAQ_5$	$CYL_{1,1}$	$CYL_{2,3}$
2	0.23	0.51	–	–
4	0.34	0.54	0.52	–
6	0.56	0.91	0.62	–
8	0.67	1.21	0.82	0.39
10	0.92	2.03	1.15	0.42
12	1.31	2.22	1.18	0.45
	Temps d'exécution sur le SP1			
$P_{sys}(\mu)$	$PLAQ_1$	$PLAQ_5$	$CYL_{1,1}$	$CYL_{2,3}$
2	0.17	0.40	–	–
4	0.23	0.43	0.42	0.35
6	0.31	0.54	0.64	0.31
8	1.21	2.45	3.13	0.90
10	1.60	3.30	6.23	1.00
12	2.00	3.99	7.92	1.04

FIG. 4.90 – *Mesures des temps d'exécution expérimentaux sur le SP1 et la PARAGON pour l'opération* $y = Ax$.

	Temps d'exécution sur la PARAGON			
$P_{sys}(\mu)$	$PLAQ_1$	$PLAQ_5$	$CYL_{1,1}$	$CYL_{2,3}$
2	0.28	0.66	–	–
4	0.28	0.53	0.56	–
6	0.34	0.58	0.55	–
8	0.41	0.72	0.59	0.42
10	0.53	1.08	0.77	0.41
12	0.67	1.18	0.79	0.42
	Temps d'exécution sur le SP1			
$P_{sys}(\mu)$	$PLAQ_1$	$PLAQ_5$	$CYL_{1,1}$	$CYL_{2,3}$
2	0.24	0.52	–	–
4	0.26	0.52	0.51	0.43
6	0.33	0.59	0.61	0.39
8	1.20	0.67	5.80	1.00
10	1.80	3.47	8.98	1.29
12	2.12	4.17	13.01	1.42

FIG. 4.91 – *Mesures des temps d'exécution expérimentaux sur le SP1 et la PARAGON pour l'opération* $LDL^T x = b$.

Etude de $PLAQ_1$

Nous analysons, d'un point de vue expérimental, les temps d'exécution observés en augmentant progressivement le grain du parallélisme. Dans ce cas d'espèce, le grain du parallélisme est donné par la taille des blocs de calculs qui vont êtres traités et par la taille des messages qui vont être acheminés. Ces deux aspects de la granularité, pour le découpage que nous avons considéré, peuvent être modélisés par μ_{max}. Pour les trois opérations précédemment citées, le choix de μ_{max}, la stratégie d'allocation de la section 4.2.3.4 fixe à $P_{sys}(\mu)$ le nombre minimum de processeurs pour les serveurs parallèle.

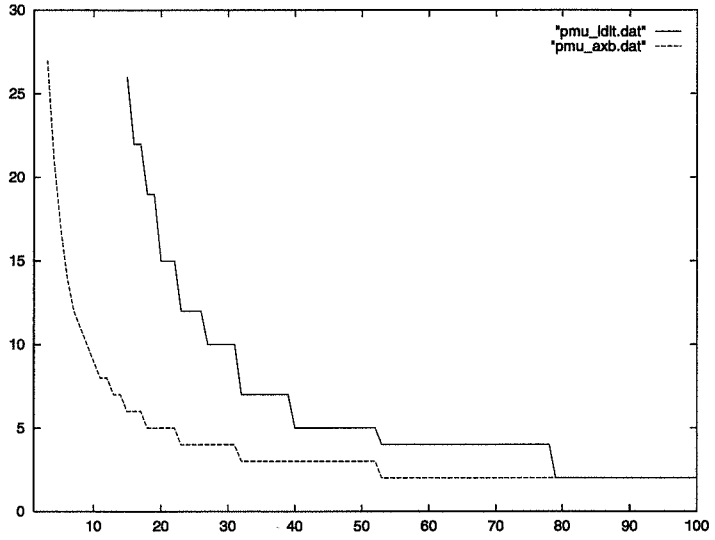

FIG. 4.92 – Evolution de $P_{sys}(\mu)$ en fonction de μ_{max} pour le problème $PLAQ_1$.

Dans la figure (4.92), les courbes illustrent l'évolution de $P_{sys}(\mu)$ en fonction de μ_{max}. La courbe *pmu_axb.dat* décrit l'évolution de $P_{sys}(\mu)$ pour l'opération $y = Ax$. La courbe *pmu_ldlt.dat* décrit l'évolution de $P_{sys}(\mu)$ pour l'opération $A = LDL^T$. Les plateaux observés montrent que, dans certains intervalles, $P_{sys}(\mu)$ est constant quel que soit la valeur de μ_{max}.

Dans la figure (4.93), nous illustrons l'évolution des temps d'exécution théoriques (voir la courbe *yax_pgon.th*) et expérimentaux (voir la courbe *yax_pgon.dat*), en fonction de μ_{max}, pour l'opération $y = Ax$. Pour les valeurs expérimentales, nous avons testé, toutes les valeurs de $\mu_{max} \in [3, 67]$. Pour ces deux courbes, nous observons un comportement semblable.

Dans la figure (4.94), nous illustrons l'évolution des temps d'exécution théoriques (voir la courbe *ltlt_pgon.th*) et expérimentaux (voir la courbe *ltlt_pgon.dat*), en fonction de μ_{max}, pour l'opération $A = LDL^T$. Pour ces expériences, nous avons testé toutes les valeurs de $\mu_{max} \in [15, 89]$. Pour des grains de parallélisme correspondant à des valeurs de $\mu_{max} < 15$, le nombre de processeurs requis, soit $P_{sys}(\mu)$, n'est pas disponible.

Au total, nous avons observé expérimentalement sur le problème $PLAQ_1$ qu'en augmentant progressivement le "grain du parallélisme", il existait une valeur de μ_{max}, et partant, du "grain

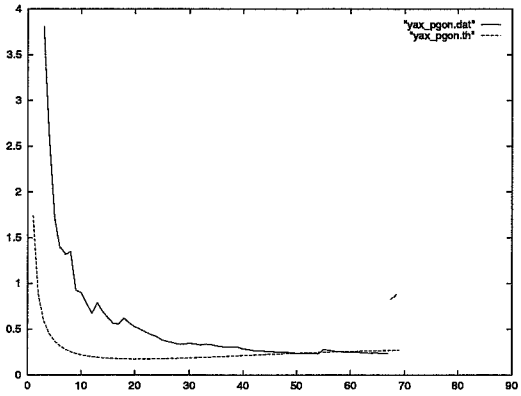

FIG. 4.93 – Comparaison des temps d'exécution expérimentaux et théoriques de l'opération $y = Ax$ pour l'exemple test $PLAQ_1$ sur la PARAGON

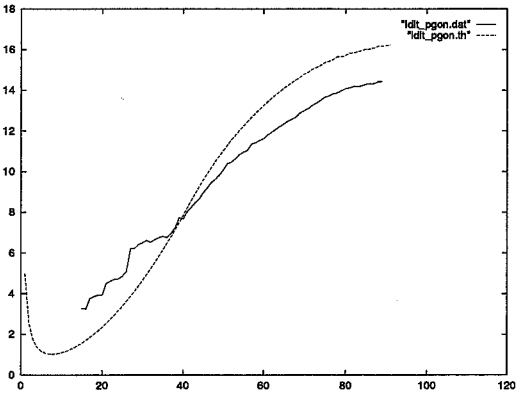

FIG. 4.94 – Comparaison des temps d'exécution expérimentaux et théoriques de l'opération $A = LDL^T$ pour l'exemple test $PLAQ_1$ sur la PARAGON

du parallélisme" pour laquelle on obtenait des performances favorables. Ces observations semblent être confirmées par le modèle d'estimation de temps d'exécution qui a été proposé. Toutefois, entre les valeurs estimées et les valeurs expérimentales, nous avons observé des écarts que nous pouvons justifier par le choix des paramètres τ, θ, t_{sca} et τ^* que nous avons supposé constants dans le modèle d'estimation a priori. Dans la pratique, ces paramètres subissent l'influence d'autres facteurs, tels l'espace mémoire requis par le programme parallèle, les chemins utilisés par les messages pour aller d'un processeur vers un autre, l'allocation de zones tampon aléatoires, pour stocker les messages émis ou reçus, etc.

Etude de $PLAQ_5$:

il s'agit d'un exemple qui se distingue de $PLAQ_1$ non seulement en terme de régularité du remplissage de son profil mais aussi en terme du nombre d'éléments du profil. En effet, il suffit par exemple de comparer le rapport nz/n dans les deux cas. Pour le problème $PLAQ_1$, on a $nz/n = 88$ alors que pour $PLAQ_5$, on a $nz/n = 216$. Ainsi, dans $PLAQ_1$, très peu de lignes ont 159 termes. Par contre pour l'exemple $PLAQ_5$, la matrice est quasiment bande et le remplissage est beaucoup plus régulier.

Au cours des expériences, nous faisons augmenter progressivement le grain du parallélisme. En reprenant les notations introduites dans le paragraphe précédent concernant $PLAQ_1$, μ_{max} symbolise le grain du parallélisme et $P_{sys}(\mu)$, le nombre maximum de blocs de calculs élémentaires exécutés théoriquement en parallèle. Pour les opérations $y = Ax$ et $LDL^T x = b$, la courbe *pmu_axb.dat* de la figure (4.95) illustre l'évolution de $P_{sys}(\mu)$ en fonction de μ_{max}, et pour l'opération $A = LDL^T$, il s'agit de la courbe *pmu_ldlt.dat*.

Dans la figure (4.96), nous illustrons l'évolution des temps d'exécution théoriques (*yax_pgon.dat*) et expérimentaux (*yax_pgon.th*), en fonction de μ_{max}, pour l'opération $y = Ax$. Pour les valeurs expérimentales, nous avons testé, toutes les valeurs de $\mu_{max} \in [6, 87]$. Pour ces deux courbes, nous observons un comportement semblable.

Dans la figure (4.97), nous illustrons l'évolution des temps d'exécution théoriques (*ldlt_pgon.th*)

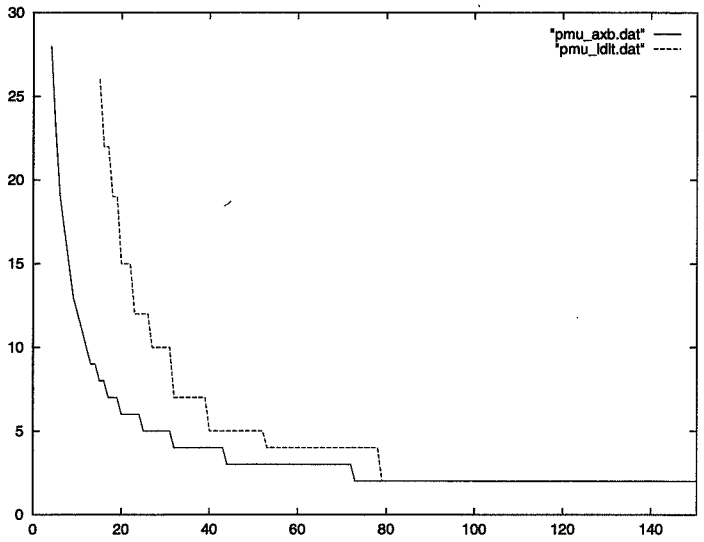

FIG. 4.95 – Evolution de $P_{sys}(\mu)$ en fonction de μ_{max} pour le problème $PLAQ_5$.

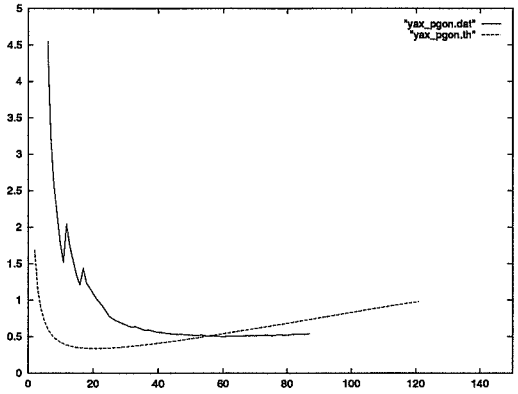

FIG. 4.96 – Comparaison des temps d'exécution expérimentaux et théoriques de l'opération $y = Ax$ pour l'exemple test $PLAQ_5$ sur la PARAGON

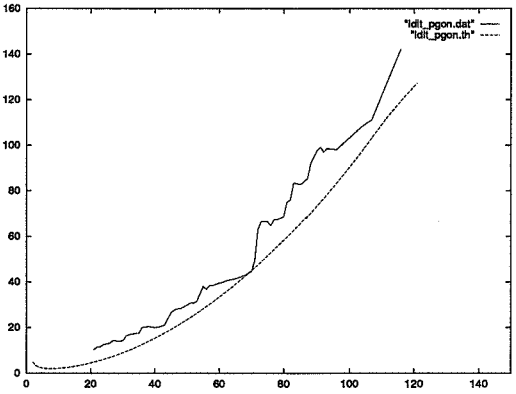

FIG. 4.97 – Comparaison des temps d'exécution expérimentaux et théoriques de l'opération $A = LDL^T$ pour l'exemple test $PLAQ_5$ sur la PARAGON

et expérimentaux (*ldlt_pgon.dat*), en fonction de μ_{max}, pour l'opération $A = LDL^T$. Pour ces expériences, nous avons testé toutes les valeurs de $\mu_{max} \in [20, 116]$. Pour des grains de parallélisme correspondant à des valeurs de $\mu_{max} < 20$, le nombre de processeurs requis, soit $P_{sys}(\mu)$, n'est pas disponible.

4.4 Conclusion

Dans ce chapitre, nous nous sommes intéressés à la parallélisation d'un module de calcul de modes et fréquences propres de *SYSTUS* (le module *DLANCB*). Trois aspects de cette étude ont été exposés.

Le premier aspect est une étude des schémas numériques implantés dans *DLANCB*. Ils s'appuient sur des algorithmes itératifs dont les calculs manipulent des structures, régulières (matrices rectangulaires) ou irrégulières (matrices skylines). Grâce à une étude théorique de la complexité de ces algorithmes, nous avons montré (voir section B.4), qu'en présence de matrices skylines de grande taille et de matrice rectangulaires de faible taille, l'exécution de méthodes numériques à matrice creuses skylines telles que le produit matrice vecteur, les résolution de systèmes triangulaires et la factorisation LDL^T étaient prohibitives.

Le deuxième aspect de cette étude a consisté à paralléliser efficacement ces méthodes numériques à matrice skylines et à les reinjecter dans le module *DLANCB*. Pour ce faire, nous avons appliqué une nouvelle approche "systolique" développée au chapitre 3. Cette nouvelle approche systolique est intéressante à plus d'un titre. Elle permet une organisation de *DLANCB* en architecture logicielle du type *client/serveur*. En cela, elle favorise la récupération du code séquentiel existant, et son utilisation en tant que programme frontal alimentant un serveur (code *SPMD*). Dans la littérature, il existe d'autres alternatives à cette nouvelle approche systolique. Toutefois, pour des méthodes numériques à matrice skyline, nous avons remarqué que l'application de ces approches pour la parallélisation de *DLANCB* était possible au prix d'une remise en cause complète du mode de stockage skyline ou de la méthode de *Lanczos par blocs*. Ceci pose des problèmes d'interfaçage avec le code de calcul existant. C'est entre autre pour ces raisons qu'il n'y a pas de comparaisons avec d'autres outils de parallélisation.

Le dernier aspect de cette étude a été l'analyse des performances séquentielles et parallèles sur trois classes de machines parallèles. Sur les réseaux de stations de travail, nous avons présenté et analysé les performances du module *DLANCB* en versions séquentielle et parallèle. Ces performances sont très satisfaisantes sur des matrices skylines de grande taille. Nous avons ensuite discutés des amélioration du modèle client/serveur pour traiter efficacement, des exemples d'applications de plus en plus coûteuses.

Sur le *SP1* et la *PARAGON*, nous avons analysé les performances du programme *SPMD* qui décrit le serveur. Les performances de la parallélisation montrent des facteurs d'accélération linéaires (voir super-linéaires) dans bon nombre de cas. Une discussion portant sur une étude comparative des performances théoriques et expérimentales, a montré que ces performances satisfaisantes résultaient bien de l'effet de la parallélisation.

Chapitre 5

Conclusions et perspectives

Les machines parallèles deviennent de plus en plus disponibles à partir de composants standards du commerce tels que les processeurs et les réseaux de communication. Il suffit par exemple d'interconnecter des équipement tels que des stations de travail à l'aide d'un réseau tel que *Ethernet*. La programmation sur ce type de machine se fait alors par échange de message. Un des but de la recherche en calcul parallèle est de fournir aux utilisateurs des méthodologies d'analyse, de conception et de preuves de programmes parallèles. Ces méthodologies, dans la mesure où elles sont constructives permettent alors aux utilisateurs de mettre en oeuvre efficacement, leurs applications sur de telles machines.

Bilan

Dans cette thèse, nous avons étudié la parallélisation d'une application industrielle. Il s'agit d'un module du code de calcul *SYSTUS*: DLANCB. Ce module met en oeuvre divers algorithmes numériques. Certains sont à matrices creuses skylines et leur exécution représente près de 80% du temps de traitement des problèmes. L'approche choisie pour paralléliser un tel module repose sur la notion de bibliothèques de sous-programmes. Son modèle d'exécution parallèle repose sur deux entités: un programme "*client*" et un "*serveur*" formés de programmes qui s'exécutent en mode *SPMD*.

Le recours au passage de message nous a conduit à développer une bibliothèque de sous-programmes conformément à une méthodologie que nous proposons. Cette méthodologie permet de concevoir des programmes parallèles exécutant des calculs de façon atomique sur des blocs de calculs et échangeant des blocs de données adjacentes. Elle a pour fil conducteur l'algorithmique *systolique*. La définition des blocs de calculs et de données est réalisée dans le cadre d'une logique de découpage. Leur placement est réalisé dans le cadre d'une logique d'allocation par bloc.

Cette méthode de parallélisation a été illustrée sur des méthodes numériques à matrice creuse telles que la factorisation LDL^T, les produits matrices vecteurs et les résolutions de systèmes triangulaires. Elle a été validée dans le cadre des algorithmes à matrices skylines qui ont ensuite été intégrés au programme *DLANCB* existant. Grâce à une bibliothèque de sous-programmes de communication *BLACS*, on peut porter *DLANCB* sur toutes les machines parallèles à base de composants standards du commerce.

Des tests ont été effectués avec succès sur des machines parallèles telles que les *IBM SPx*, les *Paragon* ou les réseaux de stations de travail. Sur des exemples tests, représentatifs de

ceux rencontrés par les utilisateurs de *DLANCB*, les gains de temps ont dépassé 3 lors d'un portage sur un réseau de stations de travail. Afin de permettre un choix de la taille des blocs dans une distribution des données par blocs, nous avons fourni un modèle d'étude a priori du comportement des temps d'exécution des différents algorithmes parallèles.

Perspectives

Le modèle choisi pour la parallélisation de *DLANCB*, à savoir l'organisation de la parallélisation autour de sous-programmes gourmands en temps de calcul n'est qu'un premier pas vers une parallélisation complète incluant la partie fréquences et modes propres complexes. La réalisation du code parallèle a soulevé un grand nombre de problèmes dont entre autre la parallélisation de la phase de distribution des données et l'exploitation du parallélisme entre les itérations de la méthode de *Lanczos* par bloc. Nous avons choisi de réaliser la phase de distribution des données séquentiellement en raison de la structure actuelle de la phase de distribution du module *DLANCB*. Cependant, ce choix se trouve confronté à un certains nombres de limitations au regard non seulement de la taille des matrices skylines que peut assembler actuellement le programme *client*, mais aussi au coût de la distribution des matrices de très grande taille. En outre, une parallélisation uniquement axée sur celle des sous-programmes critiques est loin d'être suffisante. En effet, dans le cadre du module *DLANCB*, nous avons vu que le temps de traitement des opérations à matrices non skylines augmentait avec le nombre d'itérations de la méthode de *Lanczos*, c'est à dire en première approximation avec le nombre d'éléments propres recherchés.

Une réponse à ces problèmes serait une version parallèle de *DLANCB* amalgamant des techniques de distribution de données en parallèle et une exploitation du parallélisme potentiel lié au traitement des itérations de *DLANCB*. La parallélisation de la distribution permettrait alors d'assembler en parallèle les matrices skylines. Ceci libérerait ainsi de l'espace mémoire et permettrait d'aborder le traitement des problèmes beaucoup plus "gros". Le parallélisme dans le traitement des itérations de la méthode de *Lanczos* peut se faire par le jeu du *pipeline*, tel que nous l'avons illustré dans le cadre des méthodes telles que la factorisation LDL^T. Elle permet alors d'aborder la réduction des temps de calculs liés aux algorithmes à matrice non skylines.

Annexe A

Exemples de machines parallèles

L'objet de cette annexe est la présentation de trois exemples de machines *MIMD* à mémoire distribuée utilisées au chapitre 4. Ces machines se différencient par leurs environnements matériels et présentent plusieurs points communs quant à leurs environnements logiciels.

A.1 Les environnements matériels

Les trois types de machines parallèles étudiées sont:

1– le réseau de stations de travail de type *SUN* de l'*ENSMSE (Ecole Nationale Supérieure des Mines de Saint-Etienne)*;

2 : le *SP1* d'*IBM* mis à disposition par l'Institut *IMAG* de *Grenoble*;

3 : la *PARAGON* d'*INTEL* mise à disposition par le *CDCSP Centre pour le Développement du Calcul Scientifique Parallèle* de l'Université de *Lyon1*.

Les architectures de ces machines sont constituées d'un ensemble d'unités de calcul (encore appelées *noeuds*), reliées entre elles par différents types de réseau d'interconnexion. Les principales différences entre ces architectures se trouvent dans les caractéristiques de leurs noeuds et dans le mode de fonctionnement de leur réseau d'interconnexion.

A.1.1 Le réseau de stations de travail

Il se compose d'un parc de quarante stations de travail reliées à un serveur fortement configuré. Cet ensemble est connecté au réseau général de l'*ENSMSE* et à Internet. A ce parc de stations de travail, nous avons ajouté une station sur laquelle s'exécute le code *SYSTUS* dont la parallélisation est discutée dans le chapitre 4. Cet ensemble de stations de travail interconnectées donne lieu à la réalisation d'une machine parallèle virtuelle dont les noeuds sont les stations de travail. Nous présentons maintenant, les caractéristiques des noeuds et du réseau d'interconnexion d'une telle machine parallèle virtuelle.

les noeuds:

chaque noeud est une station de travail *SUN* de type *SPARCstation4* ou *SPARCstation5* monoprocesseur sous *Solaris* 2.6. Plus précisément, on dispose de 40 stations de type *SPARCstation4* et une station de type *SPARCstation5*. Chaque station est équipée d'un processeur de type

microsparc – II qui a une fréquence d'horloge de $110Mhz$. Les *SPARCstation4* disposent d'une mémoire centrale de $64Mo$ et la *SPARCstation5* de $128Mo$ de mémoire centrale. Chaque processeur dispose également d'une mémoire cache de taille $24Ko$. Ces noeuds peuvent être configurés pour remplir plusieurs rôles logiques (serveurs de connectivités, d'image système, de systèmes de fichiers, etc).

le réseau d'interconnexion:

il s'agit d'un réseau *Ethernet* 10 base T dont la topologie physique est l'étoile [Spu94, Spu93]. Pour y accéder, chaque noeud est équipé d'un adaptateur *Ethernet*. Les machines ne sont pas connectées directement entre elles (voir figure (A.1)). Elles sont reliées individuellement aux entrées ou *ports* d'un *concentrateur*, appelé *Hub*. Le *Hub* est chargé de simuler la topologie logique en *BUS* propre à *Ethernet*(voir figure (A.2)).

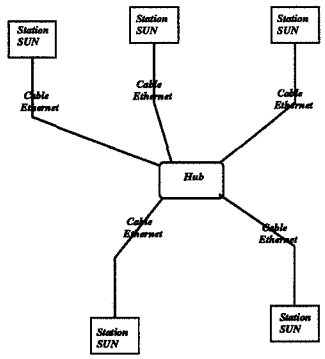

FIG. A.1 – Réseau de stations de travail en étoile.

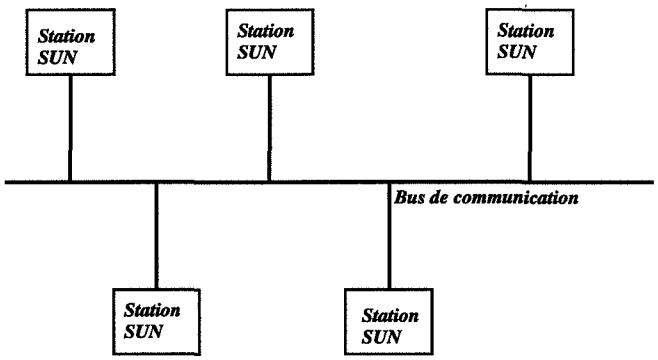

FIG. A.2 – Réseau de stations de travail en bus.

Le réseau *Ethernet* utilise une topologie logique en bus: les trames émises sont diffusées en parallèle à tous les noeuds du réseau. Cependant, sa topologie physique étant étoilée, on dit parfois que la topologie du bus réseau est *BUS étoilé*. La vitesse théorique d'un tel réseau est de $10Mb/s$. Dans la pratique, des facteurs tels que la longueur des câbles et les moyens utilisés pour organiser et réglementer la circulation des informations (protocoles) font chuter le débit de transfert des informations et introduisent des latences.

Afin d'organiser et de gérer la circulation des informations, *Ethernet* utilise des méthodes, dites *aléatoires*. Ces méthodes ne peuvent pas garantir le temps que met une information pour aller d'un noeud à un autre. De plus, elles acceptent des collisions sur le câble et une compétition entre machines. La méthode utilisée est *CSMA/CD (Carrier Sense Multiple Access with Collision Detection)*. Dans cette méthode, plusieurs stations accèdent à un support de transmission ("*Multiple Access*") et attendent que plus aucun signal ne soit détecté ("*Carrier*

Sense"). Elles transmettent alors leur données et vérifient s'il y a plusieurs signaux présents ("*Collision Detection*"). Chaque station tente de transmettre lorsqu'elle "pense" que le réseau est libre. En cas de collision, chaque station tente de retransmettre après un délai alléatoire. Chaque station sait qu'il y a collision lorsqu'elle ne reçoit pas sa propre transmission dans un laps de temps déterminé. Si une collision est détectée, un signal d'embouteillage est diffusé à tous les noeuds.

Pour communiquer, les stations de travail utilisent un schéma de communications réduit, par rapport à la norme internationale *ISO*. Deux protocoles ont été développés pour régir l'échange et le trafic des données entre les ordinateurs du réseau : le *Transmission Control Protocol (TCP)* et *l'Internet Protocol (IP)*. L'utilisation conjointe de ces deux protocoles est souvent désignée sous l'appellation *TCP/IP*. Pour des raisons physiques, la taille des données transportée sur un réseau *Ethernet* est limitée à un certain nombre d'octets (1518 – soit 1500 octets utiles au maximum). Le protocole *IP* est fait pour s'adapter à des média de communication variés et adapte la taille des paquets transportés en conséquence (1500 octets sur *Ethernet*).

A certains endroits dans le réseau, des ordinateurs ou des équipements dédiés, appelés routeurs servent à connecter entre eux les différents réseaux. Grâce au protocole *IP*, les routeurs sont en mesure de déterminer la destination des "paquets" et de les acheminer de proche en proche vers leur destination. Quant au protocole *TCP*, il s'occupe de segmenter l'information en paquets *IP* et se charge de les rassembler correctement lors de leur arrivée à destination, offrant de cette manière des communications en mode connecté aux utilisateurs. Une des particularités de la transmission par paquets est que ceuxci n'empruntent pas nécessairement le même chemin pour arriver à destination, mais plutôt celui qui est libre lors de la transmission. Etant donné que le réseau est de plus en plus occupé, il arrive fréquemment que plusieurs paquets parviennent en même temps à une "intersection". Si le routeur posté à cette intersection est débordé, les paquets sont alors automatiquement redirigés vers un routeur libre.

Les applications distribuées sur un réseau s'appuient sur le modèle *Client/Serveur* et sur des objets de communication que sont les *sockets* et les *streams*. Au dessus de ces objets sont construites les bibliothèques de communications (style *PVM*). Elles offrent aux programmeurs des fonctions de haut niveau pour le développement de ses applications.

Ainsi, un réseau de station de travail permet-il de construire une machine *MIMD* à mémoire distribuée. Les communications dans une telle machine parallèle constituent un goulot d'étranglement. Cependant, avec les performances toujours croissantes des stations de travail individuelles et la disponibilités de supports de communications rapides à faible coût tels que *Myrinet* [BCF+95] ou *SCI* [Gus92], ce type d'architecture peut constituer une bonne alternative pour le développement des applications parallèles.

A.1.2 Le SP1 d'IBM

IBM a annoncé en septembre 93 les machines parallèles *SP1*. La gamme se compose de 2 à 512 noeuds. Plus d'une centaine de systèmes *SP1* sont installés dans le monde. L'*IMAG*, dans le cadre de ses programmes de recherche sur le parallélisme, a installé un *SP1* à 32 processeurs

qui évolue actuellement en configuration *SP2*. Pour une présentation plus approfondie des machines parallèles *SPx*, nous renvoyons les lecteurs intéressés aux publications de la société *IBM* (*http://www.redbooks.ibm.com/redbooks*). Chaque configuration d'une *SPx* est pilotée par une station de contrôle *RS/6000*. Cette station est connectée à l'ensemble des noeuds pour le suivi de l'activité et le contrôle matériel des processeurs. La station de contrôle est également connectée à chaque processeur par un réseau *Ethernet* permettant le contrôle *logiciel* des processeurs. Nous présentons les principales composants du *SP1*: les noeuds et le réseau d'interconnexion.

Les noeuds:

ce sont de véritables stations de travail articulées autour d'un unique processeur *RS/6000* et d'un *UNIX* maison (*AIX*). Notons que pour le *SP1* de l'*IMAG*, certains noeuds sont configurés en *AIX3.2.5* et d'autres en *AIX4.2*. Le processeur *RS/6000* est un produit d'*IBM*. Il possède une fréquence d'horloge de $62.5Mhz$ et développe jusqu'à $125MFLOPS$ de puissance crête. Il dispose d'une mémoire vive de $64Mo$ et de $64Ko$ de cache. Les noeuds peuvent jouer divers rôles logiques:

- $\star$: exécuter des applications;
- $\star$: assurer des services de connectivité avec divers réseaux externes;
- $\star$: servir au démarrage du système d'exploitation;
- $\star$: servir pour pour l'installation initiale de logiciels et pour la propagation de ces logiciels aux processeurs devant en être équipés;
- $\star$: servir à l'exportation du système de fichier (*/usr*) pour des noeuds à capacité de disques internes réduits;
- $\star$: serveurs de données réparties, gérées par *NFS (Network File Service)*, etc

Il est possible de combiner, sans limite, le nombre de processeurs dans chaque rôle.

le réseau d'interconnexion:

il existe deux réseaux de communication dans le *SP1*. Le premier permet de connecter les noeuds via *Ethernet* avec un débit de $10Mb/s$. Pour y accéder, chaque noeud dispose d'adaptateurs *Ethernet*. Le principe d'échange de messages est alors semblable à celui exposé dans la section A.1.1. Le deuxième est un réseau multiétages rapide: le *High Performance Switch (HPS)*. Pour y accéder, chaque noeud dispose d'un adaptateur appelé *TB2*. Les adaptateurs *TB2* sont conçus autour d'un processeur *i*860 et sont usuellement installés sur la gamme de machines *SP2*.

Le *HPS* du *SP1* est un réseau réseau multiétage de type *omega* [Rum94]. Il s'agit d'une catégorie de réseaux dont la topologie est dynamique, c'est à dire pour lesquels le support physique est fixé, mais òu des *commutateurs*, permettent de modifier le schéma de connexion. Le terme multiétage signifie que les commutateurs, qui sont les briques de bases du réseau, sont organisés en étages. Un étage est composé de commutateurs. Chaque commutateur possède un certain nombre de liens d'entrées et de sorties.

Les réseaux *omega* appartiennent donc à la classe des réseaux dits *réarrangeables*. Un réseau est réarrangeable s'il permet de réaliser toute bijection entre une partie quelconque de ses entrées et une partie de ses sorties. Un tel réseau peut se bloquer puisque pour établir une liaison entre une entrée et une sortie inactives, on est conduit à modifier les liaisons existantes. La notion de blocage est liée au temps d'établissement des liaisons dans un réseau, sachant que les liaisons déjà établies, en modifient la propriété. Le niveau de blocage dépend non seulement de la topologie du réseau, mais aussi de l'algorithme de commande.

Le *HPS* dispose de liens ayant un débit de $40Mo/s$ en *full-duplex*. Comme le montre la figure (A.3), le *SP1* est constitué dé plusieurs noeuds regroupés dans des clusters. Chaque cluster contient seize noeuds (rectangles grisés) et deux étages de commutateurs (rectangles blancs). De chaque *cluster*, sortent seize liens: douze vers les autres clusters et quatre servant aux connexions externes (par exemple des disques).

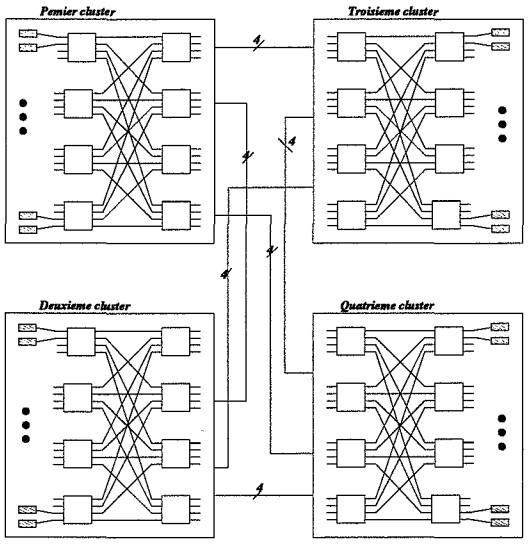

FIG. A.3 – Réseaux d'interconnexion d'un SP1 à 64 processeurs.

Une couche logicielle permet au *SP1* de réaliser des interconnexions directes entre deux noeuds. Pour cela, un calcul des commutateurs est effectué pour déterminer par où doivent passer les messages. Ce calcul ne tient pas compte de la charge du réseau. Avant d'être acheminé entre deux noeuds, un message est découpé en paquets. Les paquets sont alors routés en mode *wormhole* ou *ver de terre*. Dans ce mode de routage, les messages progressent dans le réseau de processeurs *flit* par *flit* (*flow control digit*). Le premier *flit* encore appelé la tête du message, contient l'adresse de destination. Le premier *flit* avance, à chaque fois que cela est possible et le reste du message le suit. Dans ce mode de routage, le coût des communications est relativement élevé et ne dépend pas de la distance entre les deux noeuds [Rum94].

Afin de construire des applications à base de passage de messages, le réseau *HPS* peut être utilisé suivant deux protocoles:

1 : le protocole *User Space (US)* qui est le plus performant, mais n'autorise qu'un processus par noeud. Le débit de transfert des messages dans ce protocole est de $40Mo/s$ (*full-duplex*);

2 : le protocole "*IP*" qui permet une utilisation multiple des noeuds et du *HPS*. Le débit de transfert des messages dans ce protocole celui d'un réseau *Ethernet* à $10Mo/s$.

Des bibliothèques offrent au programmeur des fonctions de haut niveau pour le développement de ses applications et l'utilisation des protocoles "*US*" et "*IP*".

A.1.3 La PARAGON d'INTEL

Il s'agit d'un produit de *Intel Corporation Supercomputer Systems Division (Intel SSD).* C'est la version commerciale du prototype *Touchstone Delta System.* La *PARAGON* se compose de noeuds interconnectés par un réseau d'interconnexion de type *grille 2D.* Le premier modèle a été livré au mois de *Septembre* 1992. Un état de l'art des fonctionnalités de la *PARAGON* est fournit dans [Coo91].

les noeuds:

chaque noeud est un bi-processeur dont les principales composantes sont illustrées par la figure (A.4). Le système d'exploitation de base de chaque noeud est *UNIX.* Il a été développé de façon à être compatible avec la norme *OSF/1.* D'un point de vue logique, on distingue trois classes de noeuds:

$\star$: les noeuds dédiés aux calculs,

$\star$: les noeuds dédiés aux services;

$\star$: les noeuds d'entrées et sorties.

La *PARAGON* du *CDCSP* est composées de trente deux noeuds:

$\star$: un noeud de service avec $32Mo$ de mémoire centrale;

$\star$: un noeud d'entrées/Sorties avec $32Mo$ de mémoire centrale;

$\star$: trente noeuds de calcul, dont dix de $32Mo$ et vingt de $16Mo$ de mémoire vive.

Dans chaque noeud, deux processeurs travaillent en parallèle et partagent une mémoire commune. Il s'agit de processeurs $i860XP$ possédant une horloge cadencée à $50Mhz$ et pouvant fournir une puissance crête de $75MFLOPS$. Un premier processeur de calcul est destiné à l'exécution des programmes à la demande des utilisateurs. Un second processeur assure la gestion des émissions et des réceptions des messages. Chaque noeud dispose d'une unité de contrôle de communication avec le réseau.

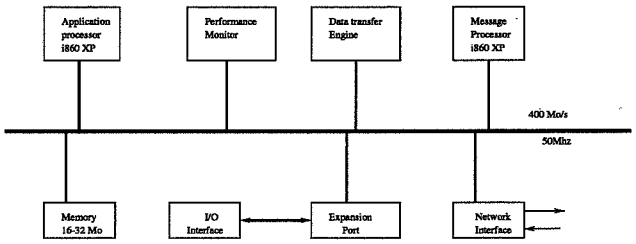

FIG. A.4 – Structure générale d'un GP node.

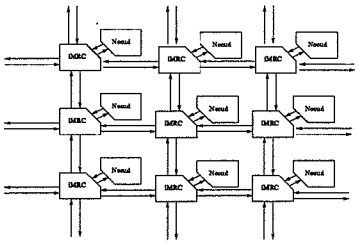

FIG. A.5 – Réseau d'interconnexion de la PARAGON

le réseau d'interconnexion:

Tout comme le *SP1*, la *PARAGON* possède deux réseaux de communications. Le premier réseau est un *Ethernet* qui relie les noeuds entre eux et est utilisé pour les tâches d'administration système. Le second est un réseau rapide rapide destiné uniquement aux calculs. La topologie physique du réseau rapide consiste en une grille $2D$. Cette grille est conçue à partir de routeurs spécifiques, appelés les *Mesh Routing Chips (iMRC)*.

Il s'agit d'un réseau d'interconnexion qui se fonde sur une topologie statique, fixée une fois pour toutes. Elle est formée de 8 lignes reliant chacune 4 processeurs. Chaque processeur dispose de 4 voisins. La grille torique est obtenue en reliant entre eux les processeurs de la première ligne et ceux de la dernière ligne puis ceux de la première colonne et ceux de la dernière colonne.

La figure (A.5) illustre une grille $2D$ d'une paragon. Chaque routeur est relié à l'unité de contrôle des messages d'un noeud par deux canaux et à quatre autres routeurs. Entre deux routeurs voisins, il existe une paire de canaux. Chaque canal a un débit de transfert des message de $200Mo/s$.

Un logiciel permet à la *PARAGON* d'acheminer les message entre deux noeuds. Ceci nécessite le calcul du plus court chemin entre deux noeuds du réseau d'interconnexion. Lorsque ce chemin a été identifié, un message est décomposé en paquets. Les paquets sont alors routés en mode *wormhole*. Les paquets avancent dans le même sens que le premier paquet émis. Tous les paquets avancent horizontalement puis verticalement. Ils changent de direction au plus une fois. Lorsque le premier paquet est bloqué, tous les paquets qui le suivent le sont également. Le mode de routage *wormhole* par paquets assure des débits théoriques de transfert de message de l'ordre de $75Mo/s$.

A.2 Description de l'environnement logiciel

Les trois machines parallèles qui ont été présentées dans la section A.1 possèdent des environnements de programmation séquentiels et parallèles relativement robustes. Pour le

développement des applications séquentielles, les outils disponibles sont:

- ⋆ langages de programmation (*C, FORTRAN, C++*) fournis par le constructeur ou domaine publique);
- ⋆ les bibliothèques de calcul *BLAS, LAPACK* du domaine publique ou optimisées par les constructeurs;
- ⋆ les bibliothèque graphiques;
- ⋆ etc.

En ce qui concerne l'implantation des applications parallèle, deux modes de développement sont possibles:

- ⋆ la programmation data-parallèle en fortran (*ADAPTOR*) où l'utilisateur incorpore des directives de placement de données dans ses programmes sources;
- ⋆ la programmation à passage de messages où l'utilisateur ajoute à ses programmes sources écrits en *C* ou *FORTRAN* des appels à des fonctions chargées de créer des tâches parallèles et des échanges de données entre processeurs.

Dans cette section, nous exposons les environnements qui ont servi à l'implantation des algorithmes parallèles. Ces environnements se classent en deux familles: les langages de programmations séquentiels classiques et les bibliothèques de communication à passage de messages.

A.2.1 Les langages de programmation

L'implantation des algorithmes parallèles que nous proposons utilise un parallélisme explicite. Classiquement, deux langages de programmation séquentiels sont utilisés en calcul scientifique: le *FORTRAN* et le *C*. Notons que bon nombre de codes de calcul scientifiques sont écrits en *FORTRAN*. Tel est par exemple le cas du module *DLANCB*. Toutefois, afin de gérer les allocations dynamiques et les affichages graphiques, on a parfois besoin de sous-programmes écrits en *C*. En général, les constructeurs proposent des variantes de compilateurs *FORTRAN* permettant d'allouer dynamiquement des tableaux. Malheureusement les codes qui en résultent sont peu portables.

Ainsi, pour l'implantation des algorithmes et des structures de données correspondantes, nous avons utilisé les directives du *FORTRAN 77* et pour la gestion des allocations dynamiques de la mémoire, nous avons utilisé des sous-programmes écrits en *C*. Ceci confère au programmes qui en résultent, une portabilité sur toutes les machines disposant de compilateurs *FORTRAN* supportant des appels de sous-programmes *C* dans du code *FORTRAN*. La réutilisation des programmes sur diverses architectures relève alors de la modification des commandes de compilation et d'édition des liens.

Sous *UNIX*, il suffit de créer un fichier décrivant toutes ces commandes de compilation et d'édition des liens (*makefile*). Seul ce fichier sera modifié en passant d'une machine à une autre. En effet, les noms des compilateurs et les options utilisées pour la génération de codes parallèles ne sont plus les mêmes. Par exemple, les noms des compilateurs sont *f77* et *cc* pour les stations de travail *SUN*, *if77* et *icc* pour la *PARAGON* et *xlf* et *xcc* pour le *SP1*. Pour ce qui est des options offertes par les compilateurs, il convient de noter que, pour le

SP1 et la *PARAGON*, les constructeurs ont rajouté des options permettant aux noeuds de disposer des programmes exécutables et d'utiliser les divers protocoles de communication qui ont été implantés. Par exemple, en utilisant les options *-nx* ou *-node* pour le compilateur de la *PARAGON*, des programmes qui peuvent être exécutés en parallèle tout en utilisant la bibliothèque de communication native *NX*. Pour le compilateur *xlf* du *SP1*, on utilise les options *-ip* ou *-us*. En revanche, pour le réseau de station de travail, les compilateurs que nous avons utilisés ne disposent pas de telles options. Les programmes sont compilés ou recopiés manuellement sur les différents noeuds.

A.2.2 Les bibliothèques de communications

Pour la parallélisation explicite, le programmeur dispose à ce jour de nombreuses bibliothèques de programmes. Certaines d'entre elles sont devenues des standards adoptés par les constructeurs de machines parallèles. Il s'agit de *PVM* ou *MPI*. D'autre, tels que *MPL* pour le *SP1* et *NX* pour la *PARAGON* sont spécifiques aux constructeurs et donc peu portables. Pour toutes ces bibliothèques, il est indispensable de gérer les échanges de messages dans le cadre de zones tampons qui doivent être étiquetées. Une telle procédure d'étiquetage complique la mise en oeuvre des phases de communication et la conception des programmes parallèles. En effet, il n'est pas rare que l'on ait besoin de recourir simultanément à plusieurs zones tampons pour recevoir ou envoyer des messages aux processeurs de façon asynchrone. Dans ce cas, il convient d'automatiser la gestion de ces zones tampons de manière à préserver l'intégrité des messages. Tel est le cas des algorithmes que nous devons implanter. Pour ces raisons, nous avons choisi les fonctions de la bibliothèque des *BLACS* afin d'implanter les communications entre les différents programmes parallèles. Les *BLACS* sont portables et optimisés sur la *PARAGON* et le *SP1*. Pour le réseau de station de travail, nous avons installé une version domaine publique au dessus de la bibliothèque *PVM* (toujours du domaine publique). D'une machine parallèle à une autre, les appels de fonctions sont les mêmes. Seules les commandes de l'édition des liens vont être variables. D'où une portabilité totale des programmes parallèles. Nous décrivons brièvement, les fonctionnalités des *BLACS* dont nous nous sommes servis. Il s'agit des fonctions de création ou de destruction de processus et des fonctions de communication.

Création et destruction des processus

Tout comme *PVM*, les *BLACS* offrent au programmeur, la possibilité de faire fonctionner un ensemble d'ordinateurs comme une machine parallèle virtuelle. L'application parallèle qui en découle est alors à base de processus (style *UNIX*) et fonctionnent également sur des machines telles que la *PARAGON* et le *SP1*. Plusieurs processus peuvent alors partager le même processeur ou le même ordinateur. Les processus ainsi générés sont repérés par des indices d'un tableau bidimensionnel, appelé grille de processus *BLACS*. Dans la figure (A.6) nous avons illustré une grille de processus *BLACS* de taille 3×4. Pour générer une telle grille, il suffit de faire exécuter à tous les processus *BLACS*, le sous-programme illustré dans la figure (A.7). Notons que le nombre de processeurs réellement utilisés peut être inférieur au nombre de processus ainsi crées.

Le sous-programme *BLACS_PINFO* retourne un identificateur du processus qui est un entier, soit IAM, et le nombre de processus utilisés, soit $NPROCS$. Ceci est vrai sur des

GRILLE DE PROCESSUS BLACS

0	***1***	***2***	***3***
4	***5***	***6***	***7***
8	***9***	***10***	***11***

FIG. A.6 – Ensemble de 12 processus BLACS ordonnés ligne par ligne et placés sur une grille 3×4.

machines telles que le *SP1* et la *PARAGON*. Pour les réseaux de stations de travail disposant de *PVM*, il faut utiliser le sous-programme *BLACS_SETUP* qui simule le lancement des processus à partir d'un processus maitre, soit le processus $IAM = 0$. Notons que le sous-programme *BLACS_SETUP* n'est opérationnel que pour les machines virtuelles construites à partir de *PVM*.

Le sous-programme *BLACS_GRIDINIT* initialise la grille de processus *BLACS*. Il s'agit ici d'une grille de taille $NPROW \times NPCOL$. Dans ce sous-programme, les paramètres en entrée sont: $NPROW$ le nombre de lignes de la grille; $NPCOL$ le nombre de colonne de la grille; une chaine de caractère qui indique si la numérotation des processus se fait ligne par ligne (*'R'*) ou colonne par colonne (*'C'*). En sortie, *BLACS_GRIDINIT* calcule un entier qui est appelé le *contexte* de la grille, soit $CONTXT$.

```
CALL BLACS_PINFO(IAM, NPROCS)
IF (NPROCS .LT. 1) THEN
      IF (IAM .EQ. 0) THEN
      WRITE(*, *)
      READ(*, *) NPROCS
      END IF
CALL BLACS_SETUP(IAM, NPROCS)
END IF
NPROW = INT( SQRT( REAL(NPROCS) ) )
NPCOL = NPROCS / NPROW
CALL BLACS_GRIDINIT(CONTXT, 'R', NPROW, NPCOL)
```

FIG. A.7 – *Création d'une grille de processus BLACS.*

A la fin des traitements en parallèle, chaque processus peut quitter la grille. Pour ce faire, il suffit d'appeler le sous-programme *BLACS_EXIT*. Il prend en entrée, un entier et permet

de libérer la mémoire utilisée par la gestion des contextes *BLACS* ainsi que les zones tampons utilisées pour les communications.

Communication entre processus

Les communications se font point à point ou par des diffusions entre des processus d'un même contexte. Pour ces deux types de communication, il existe des fonctions dites d'émission et de réception.

Pour les communications point à point l'émission peut se faire par les fonctions *vGESD2D* et *vTRSD2D*; la réception peut se faire par les fonctions *vGERV2D* et *vTRRV2D*.

Pour les diffusions l'émission peut se faire par les fonctions *vGEBS2D* et *vTRBS2D*; la réception peut se faire par les fonctions *vGEBR2D* et *vTRBR2D*.

Lorsque le nom d'une fonction *BLACS* contient le mot TR, il est destiné à des communications dans une grille de processus *BLACS* ayant la forme trapézoidale; pour des noms de sous-programmes dont le nom comporte le mot GR, il s'agit de communications dans une grille rectangulaire. A chaque émission, correspond une réception. Les noms des fonctions (*vXXXXXX*), dépendent du type de données traitées. Ainsi, v doit être remplacé par:

I dans le cas de données tels que des nombre entiers;

S dans le cas de données tels les réels simple précision;

D dans le cas de données tels les réels double précision;

C dans le cas de données tels les complexes simple précision;

Z dans le cas de données tels les complexes double précision.

Lors d'une communication, tout processus doit connaitre explicitement l'ensemble des processus avec lesquels cette communication est requise. Les messages arrivent dans l'ordre où ils ont été émis. Les messages échangés sont des tableaux bidimensionnels pouvant contenir des entiers, des réels ou des complexes. Dans le tableau de la figure (A.8), nous illustrons deux sous-programmes permettant à des processus, repérés par leurs coordonnées dans une grille de processus, de communiquer. Les données qui sont acheminées sont des matrices de taille $m \times n$. Les coefficients de la matrice A sont des nombres entiers et ceux de B des réels. Le processus émetteur a pour coordonnées $(RSRC, CSRC)$ et le processus récepteur $(RDEST, CDEST)$. Dans le sous-programme 1, le processus $(RSRC, CSRC)$ envoie dans l'ordre les tableaux A puis B au processus $(RDEST, CDEST)$. Dans le sous-programme 2, le processus de coordonnées $(RDEST, CDEST)$ reçoit A puis B émis par le processeur $(RSRC, CSRC)$. Une telle communication n'est possible que si les nombres de lignes, soit m et de colonnes, soit n sont connus au niveau des processus communiquant. On trouvera dans [DW95], d'autres exemples concernant les diffusions ou les opérations dites combinées telles que la somme, le maximum et le minimum portant sur des vecteurs distribués.

Enfin, on construit alors une estimation a priori du temps d'exécution en fonction de 4 paramètres:

- t_{scal} est la vitesse de traitement des opérations élémentaires,
- τ^* est le temps d'initialisation engendré par l'identification puis le traitement des blocs de calcul,

```
Programme 1
CALL IGESD2D(CONTXT, m, n, A, m, RDEST, CDEST)
CALL DGESD2D(CONTXT, m, n, B, m, RDEST, CDEST)
Programme 2
CALL IGERV2D(CONTXT, m, n, A, m, RSRC, CSRC)
CALL DGERV2D(CONTXT, m, n, B, m, RSRC, CSRC)
```

FIG. A.8 – *Communication point à point à l'aide des BLACS.*

⋆ τ est le temps de "start-up" pour les communication,

⋆ θ est le taux de transfert des messages entre les noeuds de la machine parallèle.

	SPARC5	*RS/6000*	*I860*
t_{scal} (LDL^T)	0.45	0.45	0.28
t_{scal} $(y = Ax)$	0.25	0.14	0.12
t_{scal} $(LDL^T x = b)$	0.44	0.30	0.27

FIG. A.9 – *valeurs expérimentales de t_{scal} en nombre de micro-secondes par opérations.*

Pour évaluer t_{scal}, nous avons évalué les performances des trois opérations $y = Ax$, $LDL^T x = b$ et $A = LDL^T$ sur des matrices symétriques stockées en mode skyline. Soit μ_{max} le nombre de lignes de telles matrices. En faisant varier μ_{max} dans l'intervalle $[5, 150]$, nous avons observé les valeurs de t_{scal} illustrées dans le tableau de la figure (A.9). Pour les trois architectures ciblées, nous avons observé des temps inférieur à 10 micro-secondes.

Nous avons mesuré expérimentalement les valeurs θ et τ. Pour ce faire, nous avons procédé au test dit "*ping pong*" (voir [DD96]). Dans ce test, on considère deux programmes utilisateur. Le premier programme (*master*) envoie des données à un deuxième programme (*slave*). Lorsque le *slave* reçoit les données émises par le *master*, il les lui retourne immédiatement. En divisant par 2, le temps mis pour effectuer le *ping pong*, on obtient une estimation du temps de communication entre le processeur qui exécute *master*, et celui qui exécute *slave*.

Notons que les données ainsi échangées dans le cadre d'un *ping pong* sont stockés dans des zones tampon qui peuvent être allouées de façon aléatoire suivant l'espace mémoire disponible sur un processeur. Les données échangées sont stockées dans des tableaux de nombres entiers dont nous faisons varier la longueur. Etant donné un tableau dont la longueur est fixée, nous avons effectué 50 *ping pong* et avons relevé la valeur valeur maximale observée entre le premier et le dernier *ping pong*. Pour toutes les machines parallèles, les programmes *master* et *slave* s'exécutent sur des processeurs distincts. Ils utilisent les sous programmes *IGERV2D* et *IGESD2D* des *BLACS* et ont été compilés avec les mêmes options que celles utilisées dans les programmes que nous validons.

Dans les figures (A.11) et (A.12), nous illustrons les temps de communication de données telles que précédemment définis pour des longueurs de tableaux situés dans l'intervalle

	RS SUN	*SP1*	*PARAGON*
τ	2307.2	110.35	50
θ	7.20	0.31	0.071

FIG. A.10 – *Valeurs expérimentales de τ en (micro-secondes) et de θ (en micro-secondes par nombre de type entier) observées sur les architectures ciblées.*

$[0, 10000]$. Dans cet intervalle, on peut ainsi utiliser la méthode des *moindres carrés* pour retrouver les paramètres τ, pour la phase de *start-up* et θ, pour le taux de transmission de données scalaires de type nombre entiers. Dans le tableau de la figure (A.10), nous résumons les valeurs ainsi obtenues sur les trois architectures parallèles ciblées.

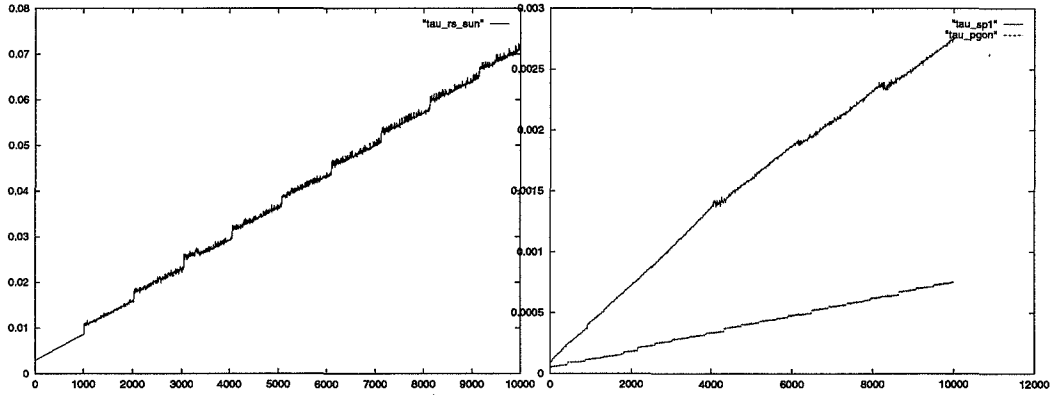

FIG. A.11 – Evolution des temps de communication expérimentaux en fonction de la taille des messages sur réseau de stations de travail.

FIG. A.12 – Evolution des temps de communication expérimentaux en fonction de la taille des messages sur la PARAGON et le SP1

Annexe B

Méthodes de calcul des valeurs et vecteurs propres

Dans cette annexe, nous traitons des spécificités du problème de recherche d'éléments propres généralisé ainsi que des méthodes numériques destinées à sa résolution. Ce problème, par rapport à une recherche d'éléments propres standard, introduit de nouveaux concept qu'il nous semble judicieux de rappeler. En outre, résoudre numériquement un problème d'éléments propres généralisé passe implicitement par un problème d'éléments propres standard. Nous rappelons les deux stratégies qui sont classiquement utilisées à cet effet.

B.1 Quelques rappels

Soient K et M deux matrices $n \times n$.

Définition 10 *On appelle problème spectral généralisé ou problème de valeurs propres généralisé, le problème:*

$$\begin{cases} \textit{trouver } (x, \lambda) \textit{ vérifiant } x \neq 0 \textit{ et} \\ Kx - \lambda Mx = 0 \end{cases} \tag{B.1}$$

Le nombre, λ, est appelé valeur propre du couple (K, M).
Le vecteur $x \in \mathbb{C}^n$, est appelé vecteur propre associé à λ.
L'ensemble des valeurs propres de (K, M), noté $\mathrm{sp}(K, M)$, est généralement appelé le spectre de (K, M).
Le couple (x, λ) est encore appelé élément propre.
On appelle faisceaux de matrices ou (faisceaux matriciels) le couple (K, M).

Le problème (B.1) généralise le problème de valeurs propres standard correspondant au cas particulier $(M = I)$. De plus le problème (B.1) présente relativement au problème standard, des différences significatives. Par exemple la notion de valeurs propres *infinies*. Ce type de valeurs propres peut apparaître par exemple lorsque dans l'équation du problème (B.1), on suppose que la matrice M est singulière [LT86a]. Une autre différence entre le problème spectral généralisé et le problème de valeurs propres standard réside dans le fait que le polynôme caractéristique $p(\lambda) = \det(K - \lambda M)$ peut être nul indépendamment de λ. Cette situation est à l'origine de la notion de *régularité* qui induit une classification sur l'ensemble

des faisceaux de matrices. On distingue ainsi des faisceaux dits *réguliers* et des faisceaux dits *singuliers* [Att93].

Définition 11 *Un faisceau (K, M) est dit singulier si et seulement si*

$$\forall \lambda \in \mathbb{C}, \text{ on a } p(\lambda) = 0 \tag{B.2}$$

Une présentation de quelques types de faisceaux singuliers ainsi que celle de quelques difficultés rencontrées lors de leur étude est faite dans [Att93]. Ces faisceaux sont assez difficiles à traiter mathématiquement et numériquement. C'est une des raisons pour lesquelles ils ne sont pas abordés dans ce rapport.

Définition 12 *Un faisceau (K, M) est dit régulier si et seulement si*

$$\exists \lambda \in \mathbb{C}, \text{ tel que } p(\lambda) \neq 0 \tag{B.3}$$

Dans le cas du problème de valeurs propres standard, la présence d'une matrice hermitienne garantit l'existence de valeurs propres réelles. Lorsque cette matrice est définie (positive ou négative), ses valeurs propres ont toutes le même signe. Dans le cas des faisceaux de matrices, cette notion est généralisée par:

Définition 13 *Un faisceau régulier (K, M) est défini si et seulement si il existe deux scalaires α et β tels que la matrice $\alpha K + \beta M$ est inversible.*

B.2 Passage du problème généralisé au standard

Une des démarches les plus naturelles pour résoudre un problème spectral généralisé associé à un faisceau de matrices régulier, consiste à effectuer un passage à un problème de valeurs propres standard équivalent pour lequel il existe des moyens assez variés pour le calcul des éléments propres. Ainsi, deux étapes sont nécessaires à la résolution de ces problèmes spectraux généralisés. C'est cette démarche qui est suivie par les méthodes numériques implantées dans *DYNSUB*.

Nous discutons de deux techniques qui peuvent être utilisées pour le passage de l'équation $Kx - \lambda Mx = 0$ à l'équation équivalente $Sy = \mu y$. Dans chaque cas nous précisons les relations donnant S, μ ou y.

Nous présentons essentiellement deux classes de transformations parmi celles généralement utilisées. La première classe est la classe de transformations dites *naturelles* et la seconde classe celle de transformations dites *spectrales*.

Les transformations naturelles

Le principe général est que, sous l'hypothèse M inversible, on a:

$$Kx - \lambda Mx = 0 \iff M^{-1}Kx - \lambda x = 0 \tag{B.4}$$

En posant $S = M^{-1}K$, on a le problème:

$$\begin{cases} \text{trouver } (y, \mu) \text{ vérifiant } y \neq 0 \text{ et} \\ Sy - \mu y = 0 \end{cases} \tag{B.5}$$

Cette formulation admet plusieurs variantes. Dans le cas où M possède une factorisation $M = LU$, alors:

$$K - \lambda M = K - \lambda LU$$

$$\text{i.e} \quad L^{-1}(K - \lambda M)U^{-1} = L^{-1}KU^{-1} - \lambda I$$

$$\text{i.e} \quad K - \lambda M = L(L^{-1}KU^{-1} - \lambda I)U$$

Si on pose $S = L^{-1}KU^{-1}$ et $y = Ux$; on obtient un problème standard du type (B.5). Dans le cas où les matrices M et K sont hermitiennes et M admet une factorisation de *Cholesky* $M = CC^T$, on montre facilement que la matrice S est hermitienne.

Notons également que lorsque K est régulière, en décomposant K sous la forme $K = LU$, on construit également un passage à un problème spectral standard de type (B.5), avec:

$$S = LMU^{-1} \text{ et } \mu = \frac{1}{\lambda} \tag{B.6}$$

Mais cette approche souffre de quelques faiblesses majeures. Par exemple, un mauvais conditionnement de M ou de K relativement à l'inversion peut influencer la qualité du spectre des valeurs propres. Autre faiblesse de cette approche, le problème de la séparation des valeurs propres qu'elle ne résout pas. En effet, pour certains problèmes tels que ceux issus de la mécanique des structures et pour lesquels on recherche des valeurs propres de faibles modules, la distance relative entre celles-ci peut être de l'ordre de 10^{-7}. Cette mauvaise séparation peut ralentir la vitesse de convergence. Ce qui peut être une motivation pour l'utilisation de la deuxième classe de transformations.

Les transformations spectrales

Les transformations *spectrales* ont pour principe, l'idée de *Ruhe* et *Ericson*. Cette idée est fondée sur la translation de l'origine. Si on écrit:

$$K - \lambda M = (K - \sigma M) - (\lambda - \sigma)M \tag{B.7}$$

avec $K_\sigma = K - \sigma M$ et on choisi σ tel que K_σ soit inversible, alors

$$K - \lambda M = K_\sigma(I - (\lambda - \sigma)K_\sigma^{-1}M) \tag{B.8}$$

En posant $\mu = 1/(\lambda - \sigma)$ et $S = K_\sigma^{-1}M$, on obtient un problème standard du type (B.9).

$$\begin{cases} \text{trouver } (y, \mu) \text{ vérifiant } y \neq 0 \text{ et} \\ Sy - \mu y = 0 \end{cases} \tag{B.9}$$

On peut imaginer plusieurs variantes à cette approche. Dans le cas où les matrices K et M sont hermitiennes, cette approche détruit visiblement le caractère hermitien de S. Mais en revanche, on peut montrer que S est auto-adjointe par rapport au produit scalaire définit par la matrice M. En effet si on note $(.|.)_M$, ce produit scalaire, on a:

$$(K_\sigma^{-1}Mu|v)_M = v^T M K_\sigma^{-1} Mu = (K_\sigma^{-1}Mv)^T Mu = (u|K_\sigma^{-1}Mv)_M \tag{B.10}$$

Cette approche est intéressante lorsqu'on désire calculer des valeurs propres d'une région quelconque du spectre. Mais pour cela il est très utile de se donner de bonnes heuristiques pour les divers choix de σ.

B.3 Taxinomie des méthodes pour le calcul des valeurs propres

La résolution des problèmes d'éléments propres comporte deux aspects fondamentaux:

1−) le calcul des valeurs propres

2−) le calcul des sous-espaces invariants

En général, la démarche suivie consiste à calculer dans un premier temps les valeurs propres puis dans un second temps les sous-espaces invariants associés aux valeurs propres. Le calcul des valeurs propres peut dépendre de critères assez variés tels que le nombre de degrés de liberté du problème, le profil des matrices, le caractère hermitien ou non des matrices, le nombre de valeurs propres recherchées, etc. Suivant ces critères, plusieurs classifications peuvent être proposées. La classification que nous utilisons ne tient compte que du nombre de valeurs propres recherchées, du profil et de la taille des matrices. Cette classification comprend deux catégories de méthodes [Cha88, GL83, LT86b].

Les méthodes de Type QZ

Le principe général de ces méthodes est la construction d'une suite de matrices qui converge vers une forme de *Schur* d'une matrice $S \in \mathbb{C}^{n\times n}$, le plus souvent triangulaire ou bloc-triangulaire, par l'utilisation de matrices de transformations orthogonales. Parmi les transformations orthogonales très souvent utilisées, citons la transformation de *Jacobi*, celle de *Householder*, celle de *Givens* (cf. [GL83]). Ces méthodes existent sous diverses formulations. Il est donc difficile d'en faire une présentation exhaustive. Généralement, le calcul des éléments propres procède en cinq étapes:

1: le passage d'un problème généralisé à un problème standard. Il se fait via une transformation naturelle ou spectrale, (voir section B.2). Par exemple, en utilisant une transformation naturelle, on passe d'un problème du type (B.1), à un autre du type (B.5) avec:

$$S = LM(L^{-1})^T \text{ et } \mu = \frac{1}{\lambda} \tag{B.11}$$

2: la tridiagonalisation de S, par exemple par par la méthode de *Givens*:

$$C = T_r T_{r-1} \cdots T_1 S T_1^T \cdots T_{r-1}^T T_r^T \tag{B.12}$$

où les T_i, pour $i = 1, \cdots, r$ sont des matrices orthogonales. Ces transformations ne modifient pas les valeurs propres μ.

3: recherche des valeurs propres de C par la méthode de QR de *Francis*. La méthode QR de *Francis* met en oeuvre un processus itératif au cours duquel, à partir de la matrice C, on calcule une suite de matrices $C^{(k)}$ telles que:

$$\begin{array}{ll} C^{(0)} = C & \\ C^{(k)} = Q^{(k)}R^{(k)} & \text{pour } k \geq 0 (factorisationdematrice) \\ Q^{(k)} & \text{est une matrice orthogonale pour } k \geq 0 \\ R^{(k)} & \text{est une matrice triangulaire supérieure pour } k \geq 0 \\ C^{(k+1)} = R^{(k)}Q^{(k)} & \text{pour } k \geq 0 (produitmatriciel) \end{array} \tag{B.13}$$

on démontre que $R^{(k)}$ tend vers une matrice triangulaire supérieure avec sur la diagonale les valeurs propres de C [Cha88, GL83, LT86b].

4: il s'agit de rechercher des vecteurs propres de C. Pour cela, connaissant les valeurs propres μ, on calcule le vecteur propre associé, par exemple, par une méthode de puissance inverse avec déflation [Att93] dans le cas des valeurs propres multiples.

5: On calcule les éléments propres du problème généralisé à partir de ceux de C, puis de ceux de S.

Au regard des opérations mises en oeuvre par les transformations orthogonales, il est clair que cette classe de méthodes est bien adaptée aux problèmes spectraux généralisés de petite taille i.e avec des matrice d'ordre $n < 1000$. En effet, les transformations orthogonales modifient les profils des matrices lorsqu'elles sont creuses. Ainsi, à partir d'une matrice creuse de grande taille, on peut obtenir une matrice pleine de grande taille dont le stockage est peu envisageable dans le cadre d'ordinateurs conventionnels. Dans *SYSTUS* cette classe de méthode est essentiellement utilisée pour des problèmes spectraux de taille $n \leq 500$ [Fra94].

Les méthodes à matrice creuses de grande taille

Le principe utilisé dans les méthodes de cette classe, et qui est celui des solveurs de problèmes spectraux à matrices creuses, est très simple. Il consiste à projeter orthogonalement et de manière itérative, le problème posé sur un sous-espace approprié de dimension m, souvent de *Krylov*. On récupère ainsi un problème spectral à résoudre de façon précise, par les méthodes de type QZ. Ce qui rend cette classe particulièrement intéressante lorsque le nombre de valeurs propres recherchées est très petit devant n, le nombre de degrés de liberté du problème.
On désigne par G_l le sous espace de dimension m et π_l la projection orthogonale en question, où l'indice l représente le compteur des itérations de la méthode.
Le problème (B.9) est approché par le problème

$$\begin{cases} \text{trouver } (x_l, \lambda_l) \text{ tel que } x_l \neq 0 \text{ et } x_l \in G_l, \text{ vérifiant} \\ \pi_l(Sx_l - \lambda_l x_l) = 0 \end{cases} \tag{B.14}$$

Le problème (B.14) s'appelle l'*approximation Galerkin* dans le cas où S est non-hermitienne et dans le cas hermitien c'est l'*approximation de Rayleigh-Ritz.*
Soit Q une base orthonormale de G_l. Si on pose $x_l = Qs_l$ alors on montre que (B.14) peut être mise sous la forme

$$\begin{cases} \text{trouver } (s_l, \theta_l) \text{ vérifiant } s_l \neq 0 \text{ et} \\ T_l s_l - \theta_l s_l = 0 \end{cases} \tag{B.15}$$

avec la matrice T_l qui est $m \times m$.

Définition 14 *Les vecteurs propres du problème (B.15) sont appelés les "vecteurs de* Ritz *"*
Les valeurs propres du problème (B.15) sont appelées les "valeurs de Ritz*"*
*Le couple (s_l, θ_l) est encore appelé une paire de "*Ritz*"*

Les méthodes de cette classe consistent à construire une base orthonormale de G_l et à résoudre (B.1) dans cette base. Ainsi, soit (x_l, λ_l) un élément propre quelconque. Pour diverses valeurs de l, on construit une suite (x_l, λ_l) et on peut se poser la question suivante:

Etant donnés les éléments propres (y, μ) de (B.9) existe t-il une suite (x_l, λ_l) qui converge vers (y, μ) lorsque l augmente?

Pour répondre à cette question, le sous espace G_l est construit dans la pratique à partir des suites de *Krylov* engendrées par un ensemble de r vecteurs indépendants $\{u_i\}_{i=1}^{r}$. Soit $W = lin(u_1, ...u_r)$ $1 \leq r < n$. Il y a deux sous-classes principales qui découlent du choix de G_l.

La première sous-classe est constituée de méthodes qui permettent d'approcher une seule valeur de *Ritz*. Dans ce cas la dimension de G_l est constante au cours des itérations, $dim(G_l) = r$. Suivant que l'on soit dans la situation $r = 1$ ou non, diverses méthodes sont à considérer. La plupart des méthodes de cette sous-classe sont des variantes de la méthode de la *puissance*. A titre d'illustration, nous présentons quelques étapes d'une méthode de dite du "*sous-espace itératif*" utilisant une transformation naturelle pour le passage à un problème d'éléments propres standard. Ces étapes sont les suivantes:

1: factorisation de la matrice K du problème (B.1);

2: choix de $r \geq 1$ vecteurs initiaux $U^{(0)} = (u_1, \cdots, u_r)$;

3: on calcule itérativement des pairs de *Ritz*. Ainsi, à partir de $i = 0$, on a:

$$
\begin{array}{ll}
(a) & K\overline{U}^{(i+1)} = MU^{(i)} \\
 & \overline{K}^{(i+1)} = \overline{U}^{(i+1)T} K \overline{U}^{(i+1)} \\
 & \overline{M}^{(i+1)} = \overline{U}^{(i+1)T} M \overline{U}^{(i+1)} \\
(b) & \text{résolution par la méthode de } \textit{Jacobi} \text{ de:} \\
 & (\overline{K}^{(i+1)} - \theta^{(i+1)} \overline{M}^{(i+1)}) s^{(i+1)} = 0 \\
(c) & U^{(i+1)} = \overline{U}^{(i+1)} s^{(i+1)} \\
(d) & \text{retour en } (a)
\end{array}
\qquad \text{(B.16)}
$$

Dans *SYSTUS*, les méthodes implantées sont: la méthode de la *puissance inverse* et la méthode du *sous espace itératif*. Il s'agit d'une généralisation méthode de puissance inverse. Dans le cas des sous espaces itératifs, on détermine en général une valeur propre multiple.

Pour la deuxième sous-classe, on met en oeuvre le calcul de plusieurs valeur de *Ritz*, éventuellement multiples. Pour ce faire deux étapes sont requises: l'orthonormalisation d'une suite de vecteurs et la détermination dans la base orthonormale ainsi calculée d'un problème de valeurs propres standard du type (B.5). La suite de vecteurs en question est une suite du type $\{U^{(0)}, SU^{(0)}, \cdots, S^{l-1}U^{(0)}\}$ avec $U^{(0)} = (u_1, \cdots, u_r)$ et $l \geq 1$. Pour cette suite, la procédure d'orthonormalisation de *Gramm-Schmidt* semble la plus appropriée pour l'orthonormalisation. En effet dans la base orthonormale qu'elle fournit, la structure de la matrice du problème à résoudre simplifie le calcul des éléments propres. En effet, la matrice est, suivant la valeur de r, tridiagonales ou tridiagonale par blocs. Les deux méthode de base de Lanczos qui sont implantées dans *SYSTUS* se fondent sur deux choix de r: $r = 1$ qui conduit à une méthode de *Lanczos par points* et $r > 1$ qui conduit à une méthode de *Lanczos par blocs*. La

principale différence entre les deux méthodes se situe au niveau de la multiplicité des valeurs propres qu'elles approchent.

Nous détaillons maintenant, le principe général des méthodes de *Lanczos* pour des matrices symétriques. Une méthode de *Lanczos* par point est construite au tour des résultats de l'orthonormalisation par la procédure de *Gramm-Schmidt* qui suivent. Pour j tel que $1 \leq j < l$, on montre que le vecteur $Q^{(j+1)}$ vérifie la relation de récurrence

$$\begin{cases} R^{(j)} = \beta_j Q^{(j+1)} = SQ^{(j)} - \alpha_j Q^{(j)} - \beta_{j-1} Q^{(j-1)} \\ Q^{(0)} = 0 \end{cases} \tag{B.17}$$

Par construction, $Q^{(0)}$ provient de la normalisation de $U^{(0)}$. De plus, on a:

$$\alpha_j = (Q^{(j)}|SQ^{(j)})_M \quad \text{pour } 1 \leq j \leq l \tag{B.18}$$

$$Q^{(j+1)} = R^{(j)}/\beta_j \text{ et } \beta_j = (R^{(j)}|R^{(j)})_M^{1/2} \text{ pour } j = 1, 2, ..., l-1 \tag{B.19}$$

La méthode de *Lanczos par blocs* s'obtient par généralisation des relations (B.17), (B.18) et (B.19) aux blocs de matrices. Les nouvelles relations obtenues en remplaçant α_j par la matrice $A_j \in \mathbb{R}^{r \times r}$, β_j par la matrice triangulaire supérieure $B_j \in \mathbb{R}^{r \times r}$ et $Q^{(j)}$ par une matrice rectangulaire $n \times r$, doivent être établies de sorte que la matrice $\overline{T}_j = Q^T MSQ$ soit tridiagonale par blocs, avec Q, la matrice formée par les blocs de vecteurs $Q^{(j)}$.

La relation (B.17) est remplacée par: pour tout j telle que $1 \leq j < l$ on montre que $Q^{(j+1)}$ vérifie

$$\begin{cases} R^{(j)} = Q^{(j+1)} B_j = SQ^{(j)} - Q_{(j)} A_j - Q^{(j-1)} B_{j-1}^T \\ Q^{(0)} B^{(0)} = 0 \end{cases} \tag{B.20}$$

La relation (B.18) est remplacée par: pour $j = 1, 2, ..., l$, A_j est donnée par

$$A_j = (SQ^{(j)})^T (MQ^{(j)}) \tag{B.21}$$

Par construction, $Q^{(0)}$ provient de l'orthonormalisation de $U^{(0)}$. Enfin la relation (B.19), qui détermine $Q^{(j+1)}$ devient une étape de factorisation *Q–R* de *Gramm-Schmidt*, à l'aide du produit scalaire $(.|.)_M$, de la matrice R_j avec $R_j = Q^{(j+1)} B_j$.

Il s'agit de méthodes qui sont très bien adaptées aux problèmes de valeurs propres de grande taille. En outre, elles sont beaucoup plus précises, du point de vue numérique dans le cadre d'une recherche de valeurs propres multiples que les méthodes des sous espaces itératifs précédemment décrites. En arithmétique exacte, les vecteurs (resp. les blocs de vecteurs) de *Lanczos* calculés par les méthodes de *Lanczos par points* (resp. *par blocs*) sont tous orthogonaux. Dans la pratique, l'utilisation de calculateurs à précision finie peut détériorer cette propriété: c'est le phénomène dit de la perte d'orthogonalité. La perte d'orthogonalité a pour effet de retarder essentiellement la converge des pairs de *Ritz*. Face à cette situation, des stratégies de réorthogonalisation sont à considérer. Dans le chapitre 4, consacré à la présentation du module *DLANCB* de *SYSTUS*, nous exposons une implantation d'une méthode de *Lanczos* par bloc mettant en oeuvre des réorthogonalisations systématiques et totales.

B.4 Calcul des valeurs et fréquences propres dans DLANCB

Le calcul des fréquences propres se fonde sur le principe des méthodes de *Lanczos* par bloc tel que décrit dans la section B.3. Toutefois, afin d'éviter les problèmes de perte d'orthogonalité qui retardent la convergence, des réorthogonalisations sont nécessaires. Dans le module

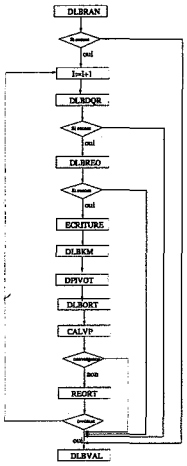

FIG. B.1 – Fonctionnement général de DLBITE

DLANCB, les réorthogonalisations sont totales dans ce sens qu'elles portent sur toute la base des vecteurs de *Lanczos*. Elles sont systématiques dans la mesure où elles sont effectuées après chaque calcul d'un bloc de vecteurs de *Lanczos*. Tous ces calculs sont implantés dans le sous programme *DLBITE* et procèdent théoriquement comme suit:

a– formation d'une matrice tridiagonale par bloc, notée T_i (voir (B.2)) à la i-ème itération;

b– calcul des valeurs propres de la matrice tridiagonale par bloc;

c– sélection des valeurs propres et test de convergence;

d– calcul des fréquences propres.

$$T_i = \begin{pmatrix} A_1 & F_1 & 0 & . & . & 0 \\ G_2 & A_2 & F_2 & 0 & . & 0 \\ 0 & G_3 & A_3 & F_3 & 0 & 0 \\ . & . & . & . & . & . \\ . & . & . & . & . & F_{i-1} \\ . & . & . & . & G_i & A_i \end{pmatrix}$$

FIG. B.2 – *Exemple de matrice tridiagonale par bloc construite à l'étape i.*

L'organigramme général du programme implanté est celui décrit dans la figure (B.1). Afin de décrire les principaux sous-programmes illustrés par cet organigramme, considérons les notations suivantes pour la définition de la suite de *Krylov*: $\{U_0, SU_0, \cdots, S^{l-1}U_0\}$ avec $U_0 = (u_1, \cdots, u_r)$, $S = \overline{K}^{-1}\overline{M}$ et $l \geq 1$. On peut alors procéder à une description des sous-programmes comme suit:

DLBRAN:

l'objet de ce sous-programme est de générer:

1−: le bloc de vecteurs U_0 qui est stocké dans un tableau bidimensionnel $n \times r$;

2−: calcul du bloc de vecteurs résiduel $R^{(0)}$ qui est ensuite rangé dans un tableau bidimensionnel $n \times r$. Le calcul de $R^{(0)}$ procède comme suit:

$$R^{(0)} = \overline{M}U$$

avec $U = (LDL^T)^{-1}\overline{M}U_0$.

Au cours du calcul de $R^{(0)}$, on effectue donc deux produits de matrices skyline par une matrice rectangulaire (deux appels au sous-programme *DLBM*); une résolution de systèmes triangulaires inférieure puis supérieure (un appel au sous-programme *DLBKM*). Dans le tableau de la figure (B.3), nous illustrons une borne supérieure du nombre d'opérations scalaires effectuée lors du calculs de $R^{(0)}$.

Opérations	*Appels*	*Complexité*
DLBM	2	$4nz\,r$
DLBKM	1	$4nz\,r$
DLBRAN	1	$12nz\,r$

FIG. B.3 – *Complexité de DLBRAN*

DLBDQR:

cette procédure est appelée de manière répétitive dans l'organigramme de la figure (B.1). Elle effectue la i-ème itération, la factorisation *Gramm-schmidt* d'une matrice rectangulaire $R^{(i-1)}$ et d'ordre $n \times r$ construite par récurrence, à l'itération $i-1$ par le sous-programme:

$$\begin{cases} \text{DLBRAN} & \text{si } i = 1 \\ \text{REORT} & \text{si } i > 1 \end{cases} \tag{B.22}$$

Cette procédure permet alors de calculer:

⋆: la factorisation $R^{(i-1)} = Q_i G_{i-1}$ avec Q_i une matrice $n \times r$ orthonormale au sens du produit scalaire $(.|.)_{\overline{M}}$ et G_{i-1} $r \times r$ une matrice triangulaire supérieure;

⋆: le produit matriciel $P = \overline{M}Q_i$;

Notons que pour des raisons d'ordre numérique, les colonnes de Q_i peuvent devenir linéairement dépendantes. Aussi, lorsqu'une telle matrice est calculée, un test est effectué afin d'éliminer les colonnes linéairement dépendantes. On procède alors à un remplacement des vecteurs ainsi éliminés. La procédure utilisée est alors *DLBRAN*. Ainsi, à la i-ème itération, on peut alors constituer la base $\{Q_1, \cdots, Q_i\}$. Les différentes étapes de cet algorithme sont

décrites dans [LT86a]. D'un point de vue théorique, les étapes de calculs qui sont éffectuées dans une telle factorisation sont:

$\star$: la normalisation des vecteurs colonnes de $R^{(i-1)}$ à l'aide du produit scalaire $(.|.)_{\overline{M}}$:

$$\frac{u}{||u||_{\overline{M}}}$$

où u est un vecteur de taille n et

$$||u||^2_{\overline{M}} = (u|u)_{\overline{M}} = u^T \overline{M} u$$

On effectue ainsi, r normalisations des vecteurs;

$\star$: les calculs de produits scalaires de type $u^T v$ où u et v sont des vecteurs colonnes de $R^{(i-1)}$ et sont de taille n. On effectue ainsi, $\frac{r(r-1)}{2}$ produits scalaires;

$\star$: le calcul de combinaisons linéaires du type $u + \alpha v$, où u et v sont des vecteurs colonnes de $R^{(i-1)}$ et sont de taille n et α un scalaire. On effectue ainsi, $\frac{r(r-1)}{2}$ combinaisons linéaires.

Dans le tableau de la figure (B.4), nous illustrons une borne supérieure du nombre d'opérations scalaires effectuée lors de la factorisation de *Gramm-Schmidt*. Dans ce tableau, nous avons noté r_0, le nombre de vecteurs colonnes défaillants.

Opérations	*Appels*	*Complexité*
$\frac{u}{\|\|u\|\|_{\overline{M}}}$	r	$4nz + 3n$
$u^T v$	$\frac{r(r-1)}{2}$	$2n$
$u + \alpha v$	$\frac{r(r-1)}{2}$	$2n$
DLBRAN	r_0	$12nz\, r_0$
DLBDQR	1	$4nz(r + 3r_0) + r(r+1)n$

FIG. B.4 – *Complexité de DLBDQR*

DLBREO

met en oeuvre une réorthogonalisation systématique des colonnes d'une matrice Q_i, par rapport à celles de la suite de matrices Q_j, pour $j = 1, \cdots, i-1$. L'algorithme implanté est celui proposé par *Ojalvo* et *Newman* []. Dans cet algorithme, on effectue k_0 fois, une réorthogonalisation totale standard. Une réorthogonalisation totale standard se fait en $i-1$ itérations suivant l'algorithme:

1: $j = 0$

2: Répéter

2.1: $j := j + 1$

2.2: "Lecture" de $P := \overline{M}Q_j$

2.3: Calcul de $A = P^T Q_i$ (produit matriciel)

2.4: "Lecture" de $P := Q_j$

2.5: Calcul de $Q_i := Q_i - PA$ (addition et produit matriciels)

3: Jusqu'à $j = i - 1$

Formellement, la réorthogonalisation met en oeuvre, à la j-ème itération, les opérations suivantes:

$\star$: le produit matriciel: $R = U^T V$ avec U et V des matrices $n \times r$ et R une matrice $r \times r$;

$\star$: la somme de deux matrices: $U + V$ avec U et V des matrices $n \times r$;

$\star$: le produit matriciel: UR avec U une matrice $n \times r$ et R une matrice $r \times r$.

Dans le tableau de la figure (B.5), nous illustrons une borne supérieure du nombre d'opérations scalaires effectuée lors de la réorthogonalisation systématique proposé par *Ojalvo* et *Newman.*

Opérations	*Appels*	*Complexité*
$R = U^T V$	$k_0(i-1)$	$2r^2 n$
$U + V$	$k_0(i-1)$	$n r^2$
UR	$k_0(i-1)$	$2r^2 n$
DLBREO	1	$5k_0(i-1)n r^2$

FIG. B.5 – *Complexité de DLBREO*

WRITEM

permet de sauvegarder les matrices Q et P sur les fichiers *NFQ* et *NFMQ*. Dans le cadre de *DLANCB*, il s'agit de sauvegarder à chaque itération i, pour $i \geq 1$ les blocs de vecteurs Q_i et $\overline{M}Q_i$. Lorsque l'on fait référence à ces matrices, elles sont transférées des fichiers à la mémoire centrale.

DLBKM

permet le calcul de $R = \overline{K}^{-1} P$ avec R et P des matrices rectangulaires $n \times r$. Il met en oeuvre la résolution de système triangulaires inférieur, soit $LY = P$ avec mise à jour de $D^{-1}P$, puis supérieure, soit $L^T R = D^{-1}P$. Le nombre d'opérations scalaires qui sont exécutées est majorée par $4nz\,r$. Dans *DLBITE*, cette procédure permet de calculer le i-ème terme de la suite de blocs de vecteurs de *Krylov*. Dans le cadre de *DLBITE*, on construit le vecteur dit résiduel $R^{(i)} = \overline{K}^{-1} P$ avec $P = \overline{M}Q_i$

DPIVOT

calcule et stocke les pivots de la matrice $S = R^T \overline{M} R$ dans un vecteur D d'ordre r. Pour ce faire, le calcul de S est requis. Dans *DLBITE*, il s'agit à l'itération, i, de calculer des valeurs destinées à vérifier numériquement que, dans *DLBDQR* la factorisation de *Gramm-Schmidt* a produit des vecteurs linéairement indépendants à la $(i+1)$-ème itération. Formellement, les opérations effectuées sont:

$\star$: $V := \overline{M}U$ (appel de *DLBM*) avec U et V des matrices $n \times r$;

$\star$: $R = U^T V$ avec U et V des matrices $n \times r$ et R une matrice $r \times r$;

$\star$: la transformation de la matrice R en une matrice triangulaire supérieure, avec R une matrice $r \times r$ (triangularisation).

Dans le tableau de la figure (B.6), nous illustrons une borne supérieure du nombre d'opérations scalaires effectuée en exécutant *DPIVOT*.

Opérations	*Appels*	*Complexité*
$R = U^T V$	1	$2r^2 n$
triangularisation	1	r^3
DLBM	1	$4nz\,r$
DPIVOT	1	$4nz\,r + r^2(2n + r)$

FIG. B.6 – *Complexité de DPIVOT*

DLBORT

permet d'orthogonaliser le vecteur résiduel $R^{(i)}$, par rapport à une suite de matrices Q_j, pour $j = 1, \cdots, i-1$ et le calcul de matrices A_i $r \times r$, et F_{i-1} $r \times r$, dans le cadre de la i-ème étape de la procédure de *Lanczos*. Pour cela, on procède en trois étapes:

1: on déduit F_{i-1} en transposant la matrice G_i;

2: on orthogonalise le vecteur $R^{(i)}$ par rapport aux blocs de *Lanczos* Q_j, pour $j = 1, \cdots, i-1$. La procédure d'orthogonalisation équivaut formellement à une exécution de l'algorithme de réorthogonalisation standard utilisé dans *DLBREO*;

3: on construit la matrice A_i telle que:

$$A_i = Q_i^T \overline{M} R^{(i)} = (\overline{M} Q_i)^T R^{(i)}$$

avec $R^{(i)}$, la matrice obtenue après orthogonalisation.

Dans l'étape (3), on lit la quantité $\overline{M}Q_i$. Dans le tableau de la figure (B.7), nous illustrons une borne supérieure du nombre d'opérations scalaires effectuée en exécutant *DLBORT*.

Opérations	*Appels*	*Complexité*
2	1	$5(i-1)r^2\,n$
3	1	$2r^2\,n$
DLBORT	1	$(5i-3)r^2\,n$

FIG. B.7 – *Complexité de DLBORT*

CALVP

désigne par soucis de simplicité, l'ensemble des sous-programmes dédiés au calcul des valeurs propres d'une matrice tridiagonale par bloc, notée T_i. Deux sous-programmes sont ainsi appelés pour effectuer:

1: la décomposition de T_i sous forme de *Hessenberg*, soit H_i la matrice de *Hessenberg* ainsi obtenue;

2: le calcul des valeurs propres de la matrice de *Hessenberg* H_i par la méthode QR de *Francis*.

Par construction, la matrice T_i est une matrice bande. Son nombre de ligne est de $i * r$. Le nombre maximum de termes par ligne est de $2r$. A l'aide de $i\,r$ transformations orthogonales successives, T_i est transformé en une matrice de *Hessenberg* H_i. On montre que (pour une factorisation de *Householder* appliquée à une matrice pleine [GL83, Cha88]) le nombre d'opérations scalaires requis par une telle factorisation est majoré par:

$$\boxed{\frac{2}{3}i^3 * r^3} \tag{B.23}$$

La méthode QR de *Francis* procède de façon itérative. Dans le pire des cas, nous notons k le nombre d'itération à réaliser, quel que soit H_i. Lorsque la méthode QR de *Francis* est appliquée à une matrice de *Hessenberg* de taille $i\,r \times i\,r$ (voir [GL83, Cha88]), le nombre d'opérations scalaires à chaque itération est d'environ:

$$\boxed{4 * i^2 * r^2} \tag{B.24}$$

Au total, le nombre d'opérations scalaires requis par le résolution du problème spectral réduit est majoré par:

$$\boxed{i^2\,r^2(4\,k + \frac{2}{3}i\,r)} \tag{B.25}$$

Pour les l itérations de la méthode de *Lanczos*, on montre que:

$$\sum_{i=1}^{l} i^2 r^2 (4k + \frac{2}{3} i r) \leq l^3 r^2 (\frac{1}{6} l r + \frac{4}{3} k) \tag{B.26}$$

REORT

réalise une réorthogonalisation systématique d'une matrice rectangulaire R par rapport à une suite de matrices Q_j, pour $j = 1, \cdots, i-1$. Cette réorthogonalisation est effectuée successivement k_1 fois. D'où un nombre d'opérations scalaires dont la majoration est obtenue formellement, selon la procédure utilisée pour *DLBREO*.

XLBVAL

met en oeuvre le calcul des fréquences propres à partir des valeurs propres ayant convergées. Pour ce faire, les formules utilisées sont déduite de l'expression du problème spectral de la section 4.2.

L'application de *DLBITE* dans le calcul des valeurs et fréquences propres coûte très cher. Elle met en oeuvre des opérations standard de l'algèbre linéaire à matrices symétrique stockées en mode skyline, à matrices bande, à matrice *Hessenberg*, à ou vectorielles. Le tableau de la figure B.8 illustre pour chacune de ces opération, sa complexité et le nombre de fois où elle est appelée dans *DLBITE*. Dans ce tableau, nous notons k: le nombre maximum d'itérations exécutés par un appel de la méthode QR de *Francis* et k_0 et k_1 sont respectivement le nombre de fois que l'on effectue des réorthogonalisations standards de *DLBREO* et *REORT*.

Opérations	*Appels*	*Complexité*
DLBRAN	1	$12nz\,r$
DLBDQR	l	$(4nz + (r+1)n)r\,l$
DLBREO	l	$5\frac{(l-1)l}{2} k_0 n\, r^2$
DLBKM	l	$4nz\,r\,l$
DPIVOT	l	$(4nz + r(2n+r))r\,l$
DLBORT	l	$\frac{(5l-1)l}{2} r^2\, n$
CALVP	l	$l^3 r^2 (\frac{1}{6} l\, r + \frac{4}{3} k)$
REORT	l	$5\frac{(l-1)l}{2} k_1 n\, r^2$

FIG. B.8 – *Coût des principales opérations dans DLBITE*

Bibliographie

[Aa94] G. Authié and al., *Algorithmes parallèles, analyse et conception*, Hermes, year 1994.

[ABB+95] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. Mckenney, S. Ostrouchov, and D. Sorensen, *Lapack users' guide*, SIAM Philadelphia PA, year 1995.

[AG89] G.S Almasi and A. Gottlieb, *Highly parallel computing*, The Benjamin/Cummings Publishing Company Inc., Redwood City, 1989.

[AK87] R. Allen and K. Kennedy, *Automatic translation of fortran programs to vector form*, ACM TOPLAS **9(4)** (1987), 491–542.

[AL93] J. M. Anderson and M. S. Lam, *Global optimization for parallelism and locality scalable parallel machines*, ACM Sigplan Notices **28(6)** (1993), 112–125.

[AQ85] F. André and P. Quinton, *Algorithmique systolique*, Parallélisme , communication et synchronisation (1985), 467–488.

[Att93] K.K. Attoh, *Contribution à l'analyse numérique du problème généralisé de valeurs propres et applications*, Ph.D. thesis, Université Jean Monnet de Saint-Etienne., Avril 1993.

[Bab84] R.G Babb, *Parallel processing with large grain data flow techniques*, Computer (1984), 55–61.

[Ban88] U. Banerjee, *Dependence analysis for supercomputing*, Kluwer Academic, 1988.

[BBC+94] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the solution of linear systems: Building blocks for iterative methods, 2nd edition*, SIAM, Philadelphia, PA, 1994.

[BCC+97] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, K. Stanley, D. Walker, and R. C. Whaley., *Scalapack users' guide*, SIAM Philadelphia PA, year 1997.

[BCF+95] J. Boden, D. Cohen, R.E. Felderman, A.E Kulawik, C.L Seitz, J.N Seizovic, and W.-K Su, *Myrinet: a gigabit-pet-second local area network*, IEEE Micro (1995), 29–35.

[BDD90] J. Bu, E.F. Deprettere, and P. Dewilde, *A design methodology for fixed-size systolic arrays*, (IEEE Computer Society Press, ed.), 1990, pp. 591–602.

[BDRV97] P. Boulet, J. Dongarra, Y. Robert, and F. Vivien, *Tiling for heterogeneous computing platforms*, Tech. report, University of Tennessee Knoxville USA, 1997, UT-CS-97-373.

[BDSV97] P. Boulet, A. Darte, G. A. Silber, and F. Vivien, *Loop parallelisation algorithms: from parallelism extraction to code generation*, Tech. report, LIP ENS Lyon France, 1997, 97-17.

[BENP93] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua, *Automatic program parallelization*, Proceeding of the IEEE **81** $N°2$ (1993), 211–243.

[Ber93] J.Y Berthou, *Construction d'un parallèliseur interactif de logiciels scientifiques de grande taille guidé par des mesures de performances.*, Ph.D. thesis, Université de Paris 6, Octobre 1993.

[BGS94] D. F. Bacon, S. L Graham, and O. J. Sharp, *Compiler transformations for high-performance computing*, ACM computing surveys **26** $N°2$ (1994), 345–420.

[Bou96] P. Boulet, *Outils pour la parallélisation automatique*, Ph.D. thesis, Ecole normale supérieure de Lyon, 1996.

[BPST96] J-C. Bermond, C. Peyrat, I. Sakho, and M. Thuenté, *Parallelization of the gaussian elimination algorithm on systolic arrays.*, Journal of parallel and distributed computing **33** (1996), 69–75.

[BR90] A. Benaini and Y. Robert, *Space-minimal systolic arrays for gaussian elimination and the algebraic path problem*, Parallel Computing **15** (1990), 211–225.

[Bre73] R. P. Brent, *"the parallel evaluation of arithmetic expressions in logarithmic time"*, in Complexity of Sequential and Parallel Numerical Algorithms (1973), 83–102.

[BS99] P. Bassomo and I. Sakho, *Une architecture systolique programmable optimale et reconfigurable pour la factorisation* LDL^T, SYMPA5 (Rennes) (1999), 69–80.

[BT89] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and distributed computation. numerical methods*, Prentice-Hall,Inc, year 1989.

[CD97] A. Cleary and J. Dongarra, *Implementation in scalapack of divide-and-conquer algorithms for banded and tridiagonal linear systems*, Tech. report, University of Tennessee Computer Science, 1997, CS-97-358.

[CDW96] J. Choi, J. Dongarra, and D. Walker, *Pb-blas: A set of parallel block basic linear algebra subroutines*, Concurrency: Practice and Experience (1996), 517–535.

[Cha88] F. Chatelin, *Valeurs propres de matrices.*, Masson, 1988.

[CM94] T. F. Chan and T. P. Mathew, *Domain decomposition algorithms*, Acta numerica (1994), 61–143.

[CMPia] P. Clauss, C. Mongenet, and G.-R. Perrin, *Calculus of space-optimal mappings of systolic algorithms on processor arrays*, IEEE computer society press (1990 Los Alamos California), 4–18.

[Coo91] Intel Coorporation, *Paragon xp/s , product overview*, Intel Coorporation, 1991.

[CT93] M. Cosnard and D. Trystram, *Algorithmes et architechtures parallèles*, InterEditions, Paris, 1993.

[Dar93] A. Darte, *Techniques de parallélisation automatique de nids de boucles*, Ph.D. thesis, Ecole Normale Supérieure de Lyon, Avril 1993.

[DD96] J. Dongarra and T.H. Dunigan, *Message-passing performance of various computers*, Tech. Report ORNL/TM-13006, Oak Ridge National Laboratory, 1996.

[DPM91] R. Das, R. Ponnusamy, and D. Mavriplis, *Distributed memory compiler methods for problems-data copy reuse and runtime partitionning.*, Tech. Report 91-73, ICASE NASA Langley Research Center, 1991.

[DR83] I. S. Duff and J. K. Reid, *The multifrontal solution of indefinite sparse symetric linear equations.*, ACM Transaction on mathematical software **9** (1983), 302–325.

[DR94] M. Dion and Y. Robert, *Mapping affine loop nests: New result*, Tech. report, LIP ENS Lyon France, 1994, 94-30.

[DT94] J. Dongarra and B. Tourancheau, *Environment and tools for parallel and scientific computing II*, SIAM Publisher Townsend, Tennessee, 1994.

[Duf96] I. S. Duff, *Sparse numerical linear algebra: direct methods and preconditioning.*, Tech. Report TR/PA/96/22, CERFACS, 1996.

[DW95] J. Dongarra and R. Clint Whaley, *A user's guide to the blacs v1.1*, Tech. Report LAPACK working note 94, University of Tennessee, 1995.

[Eij92] V. Eijkhout, *Distributed sparse data structures for linear algebra operations*, Tech. Report CS-92-169, University of Tennessee, 1992.

[Fea96a] P. Feautrier, *Automatic parallelisation in the polytope model*, The data parallel programming model: Foundations HPF Realizati on and Scientific Applications (1996), 79–103.

[Fea96b] P. Feautrier, *Distribution automatique des données et des calculs*, T.S.I-Technique et Science Informatiques **15** (1996), 529–558.

[Fly72] M.J Flynn, *Some computer organization and their effectiveness*, IEEE Transaction on computer **C-21(9)** (1972), 948–960.

[Fos95] I. Foster, *Designing and building parallel programs*, Addison-Wesley, year 1995.

[Fra94] Framasoft+CSI, *Systus-manuel d'utilisation*, Framasoft+CSI, 1994.

[Gas66] N. Gastinel, *Analyse numérique linéaire*, Hermann, Paris, Janvier 1966.

[GH94] S. Ghernaouti-Hélie, *Communication en mode client/serveur. Les outils du traitement réparti coopératif*, Masson, 1994.

[GHLN88] A. George, M. T. Heath, J. W. H. Liu, and E. G. Y. Ng, *Sparse cholesky factorization on on local memory multiprocessor*, SIAM Journal on scientific and statistical computing **9** (1988), 327–340.

[GJ79] M. R. Garey and D.S Johnson, *Computer and intractability. a guide into the theory of NP-Completeness*,W.H. Freeman and Company, 1979.

[GKK94] A. Gupta, G. Karypis, and V. Kumar, *Highly scalable parallel algorithms for sparse matrix factorization*, Tech. report, University of Minnesota, 1994, TR 94-63(Last modified on December 18 , 1995).

[GL83] G. H. Golub and C. Van Loan, *Matrix computation*, Johns Hopkins University Press, 1983.

[Gus92] B. Gustavson, *The scalable coherent interface and related standards projects*, IEEE Micro (1992), 10–22.

[HNP91] M. T. Heath, E. G. Ng, and B. W. Peyton, *Parallel algorithm for sparse linear systems*, SIAM Review **33(3)** (1991), 420–460.

[Hoa78] C. A. R. Hoare, *Communicating sequential processes*, Communications of the ACM **21(8)** (1978), 666–677.

[IT88] F. Irigoin and R. Triolet, *Supernode partitionning*, ACM Symp. of programming languages (1988), 319–329.

[KGGK94] V. Kumar, A. Grama, A. Gupta, and G. Kapyris, *Parallel computing*, Benjamin Cummings Redwood City CA, year 1994.

[KK95] G. Karypis and V. Kumar, *METIS unstructured graph partitioning and sparse matrix ordering system*, Tech. report, University of Minnesota, 1995, (Version 2.0).

[KL88] B. Kruatrachue and T. Lewis, *Grain size determination for parallel processing*, IEEE Software **1(23)** (1988), 23–32.

[KLS+94] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel, *The high performance fortran handbook*, The MIT Press, 1994.

[KMW67] R.M. Karp, R.E. Miller, and S. Winograd, *The organization of computations for uniform recurrence equations*, Journal of the association for computing machinery **14(3)** (1967), 563–590.

[KST95] D. E. Keyes, Y. Saad, and D.G Truhlar, *Domain based parallélism and problem domain decomposition methods in computational science and engineering.*, SIAM Philadelphia PA, year 1995.

[Kum82] S. P. Kumar, *Parallel algorithms for solving linear equation on MIMD computers*, Ph.D. thesis, Washington State University, 1982.

[Kun82] H. T. Kung, *Why systolic architectures?*, IEEE Computer **15** (1982), 37–46.

[Lare] D. Lavenier and al., *Digital signal processing for multimedia systems (ch 5)*, Eds: Parhi and Nishitan, (à paraitre).

[Lam74] L. Lamport, *The parallel execution of do loop*, Communications ACM **17,2** (1974), 83–93.

[Lav90] I. Lavallée, *Algorithmique parallèle et distribuée*, Hermes, 1990.

[LHKK79] C.L. Lawson, R.J. Hanson, D. Kincaid, and F.T. Krogh, *Basic linear algebra subprograms for fortran usage*, ACM Trans. Math. Soft 5 (1979), 308–323.

[Lil94] D. J. Lilja, *Exploiting the parallelism available in loops*, Computer **27(2)** (1994), 13–26.

[Liu92] J. W. H. Liu, *The multifrontal method for sparse matrix solution: Theory and pratice*, SIAM Review **34(1)** (1992), 82–109.

[LJ93a] W. Lichtenstein and S-L. Johnsson, *Block-cyclic dense linear algebra*, SIAM J. Sci. Stat. Comput **14** (1993), 1259–1288.

[LJ93b] W. Lichtenstein and S-L. Johnsson, *Block-cyclic dense linear algebra*, SIAM J. Sci. Stat. Comput **14** (1993), 1259–1288.

[LP96] B. Lucquin and O. Pironneau, *Introduction au calcul scientifique*, Masson, Paris, Janvier 1996.

[LT86a] P. Lascaux and R. Théodor, *Analyse numérique matricielle appliquée à l'art de l'ingénieur*, vol. 1, Masson, 1986.

[LT86b] P. Lascaux and R. Théodor, *Analyse numérique matricielle appliquée à l'art de l'ingénieur*, vol. 2, Masson, 1986.

[ltd88] Inmos ltd, *Transputer reference manual*, Prentice Hall, 1988.

[Mau89] C. Mauras, *Alpha: un langage équationnel pour la conception et la programmation d'arctitectures parallèles synchrones*, Ph.D. thesis, Université de Rennes 1, Décembre 1989.

[May83] D. May, *The occam language*, SIGPLAN Notices **33** (1983).

[MCP94] C. Mongenet, Ph. Clauss, and G.-R. Perrin, *Geometrical tools to map systems of affine recurrence equations on regular arrays*, Acta Informatica **31** (1994), 137–160.

[MF86] D.I Moldovan and .A.B Fortes, *Partitionning and mapping algorithms into fixed-size systolic arrays*, Journal IEEE Transaction on computers **(35)1** (1986), 1–12.

[ML88] J.H Moreno and T. Lang, *Graph-based partitioning for matrix algorithms for systolic arrays: Application to transitive closure.*, ICPP **1** (1988), 28–31.

[Mol83] D.I Moldovan, *On the design of algorithms for vlsi systolic arrays*, Proc.IEEE **71** (1983), 113–120.

[MS95] K. J. Maschhoff and D. C. Sorensen, *A portable implementation of arpack for distributed memory parallel architechtures*, Tech. report, Departement of Computational and Applied Mathematics, Rice University, October 1995, Houston, Texas 77251-1829.

[MW84] W. L. Miranker and A. Winkler, *Space-time representation of computational structures*, Computing **32** (1984), 93–114.

[PE96] S. Petiton and N. Emad, *A data parallel scientific computing introduction*, The data parallel programming model: Foundations HPF Realization and Scientific Applications (1996), 45–64.

[Pel95] F. Pellegrini, *Application de la méthode de partition à la résolution de problèmes de graphes issus du parallélisme*, Ph.D. thesis, Université Bordeaux 1, Janvier 1995.

[Pel97] F. Pellegrini, *Scotch* 3.1 *user guide*, Université de Bordeaux 1, 1997.

[pfF94] High performance fortran Forum, *High performance fortran language specification , version* 1.1, Tech. report, Rice University, 1994.

[Pug91] W. Pugh, *Uniform techniques for loop optimization*, ACM Conf. on Supercomputing (1991), 341–352.

[Pug92] W. Pugh, *The Omega test: a fast and practical integer programming algorithm for dependence analysis*, Communications of the ACM **8** (1992), 102–114.

[QD89] P. Quinton and V. Van Dongen, *The mapping of linear recurrence equations on regular arrays*, Tech. Report 485, IRISA, 1989.

[QR89] P. Quinton and Y. Robert, *Systolic algorithms and architectures*, Prentice Hall and Masson, 1989.

[Qui83] P. Quinton, *The systematic design of systolic arrays.*, Tech. Report 193, IRISA, 1983.

[Qui87] M.J Quinn, *Designing efficient algorithms for parallel computers*, Mc Graw, 1987.

[RG93] E. Rothberg and A. Gupta, *An efficient block-oriented approach to parallel sparse cholesky factorization*, Supercomputing '93, 1993.

[Rot94] E. Rothberg, *Performance of panel and block approches to sparse cholesky factorization on the ipsc/860 and paragon multicomputers*, Scalable High performance computing conference, 1994.

[RPF86] S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto, *On synthesizing systolic arrays from recurrence equations with linear dependencies*, (Springer Verlag, LNCS 241, ed.), 1986, pp. 488–503.

[RS92] J. Ramanujam and P. Sadayappan, *Tiling multidimensional iteration spaces for multicomputers*, Journal of Parallel and Distributed Computing **16(2)** (1992), 108–120.

[RT83] P.A Raviart and J.M Thomas, *Introduction à l'analyse numérique des équations aux dérivées partielles*, Masson, Paris, 1983.

[Rum94] J. De Rumeur, *Communication dans les réseaux de processeurs*, Masson, Paris, 1994.

[Saa96] Y. Saad, *Iterative methods for sparse linear systems*, PWS publishing company, year 1996.

[SGM95] B. Smith, W. Gropp, and L.C. Mcinnes, *Petsc 2.0 users manual*, Tech. Report 95/11, Argonne National Laboratory, 1995.

[SM95] Y. Saad and A. Malevsky, *PSPARSLIB: A portable library of distributed memory sparse iterative solvers*, Proceedings of Parallel Computing Technologies (PaCT-95), 3-rd international conference, St. Petersburg, Russia, Sept. 1995 (V. E. Malyshkin et al., ed.), 1995.

[Sor95] D. C. Sorensen, *Implicitly Restarted Arnoldi / Lanczos Methods For Large Scale Eigenvalue Calculation*, Tech. report, Departement of Computational and Applied Mathematics Rice University, October 1995, Houston, Texas 77251-1829.

[Spu93] C. Spurgeon, *Network reading list: TCP/IP, UNIX, and ethernet*, Tech. Report ftp://ftp.utexas.edu/pub/netinfo/ethernet/net-read-ethernet.ps, UTnet Network Information Center University of Texas at Austin, 1993.

[Spu94] C. Spurgeon, *Guide to ethernet*, Tech. report, Networking services University of Texas at Austin, 1994, ftp://ftp.utexas.edu/pub/netinfo/ethernet/ethernet-guide-A4.ps.

[ST89] I. Sakho and M. Tchuente, *Méthodologie de conception d'algorithmes paralleles pour réseaux réguliers*, T.S.I-Technique et Science Informatiques **8** N^o1 (1989), 63–72.

[Tou90] A. Benaini Patrice Quinton Yves Robert Y. Saouter Bernard Tourancheau, *Synthesis of a new systolic architecture for the algebraic path problem*, Science of Computer Programming **15(2-3)** (1990), 35–158.

[Tri84] R. Triolet, *Contribution à la parallélisation automatique de programmes fortran comportant des appels de procédure*, Ph.D. thesis, UPMC, 1984.

[Ull75] J. D. Ullman, *NP-complete scheduling problems*, Computer system science **10** (1975), 384–393.

[Viv97] F. Vivien, *Détection de parallélisme dans les boucles imbriquées*, Ph.D. thesis, Ecole Normale Supérieure de Lyon, Décembre 1997.

[YAI95] Y.-Q. Yang, C. Ancourt, and F. Irigoin, *Minimal data dependence abstractions for loop transformations: Extended version*, International journal of parallel programming **23** $N°4$ (1995), 359–388.

[ZZWD93] S. Zhou, X. Zheng, J. Wang, and P. Delisle, *Utopia: a load sharing facility for large heterogenous distributed computer systems*, Software Practice and experience **24** $N^0 11$ (1993), 1305–1336.

Table des figures

RÉSUMÉ. L'augmentation continue de la puissance des ordinateurs personnels (stations de travail ou PCs) et l'émergence de réseaux à haut débits fournissent de nouvelle opportunités de réalisation de machines parallèle à faible coût, en comparaison des machines parallèles traditionnelles. On peut aujourd'hui construire de véritables machines parallèles en interconnectant des processeurs standards. Le fonctionnement de cet ensemble de processeurs en tant que machines parallèle est alors assuré par des logiciels tels que PVM *et* MPI. *Quelle que soit la machine parallèle considérée, concevoir des applications parallèles impose, soit des outils de parallélisation automatique, soit un effort du programmeur suivant des méthodologies de programmation.*

Dans cette thèse, nous proposons une méthodologie de parallélisation des méthodes numériques. En général les méthodes numériques sont une chaîne d'algorithmes s'appelant les uns après les autres tout au long de leur exécution. A moins d'aborder leur parallélisation à partir du problème physique qu'elles traitent, par exemple par des techniques de décomposition de domaines, *l'approche de parallélisation la plus réaliste est celle de type* client/serveur.

MOTS-CLÉS : Machine parallèle, Algorithme systolique, Découpages par blocs, Allocation par blocs, Problème spectral généralisé, Méthode de Lanczos, Matrice Creuse, Stockage skyline.

ABSTRACT. Distributed memory machines consisting of multiple autonomous processors connected by a network are becoming commonplace. Unlike specialized machines like systolic arrays, *such systems of autonomous processors provide virtual parallelism through standard* message passing *libraries (*PVM *or* MPI*). In the area of parallelizing existing numerical algorithms, two main approaches have been proposed:* automatic parallelization techniques *and* explicit parallelization.

In the present work, we focus our studies on the second approach. The parallelization paradigm found to be most effective for numerical algorithms on distributed memory machine was to provide the user with a client/server *architecture. The most difficult part to design is the* SPMD *code which is initiated by a client process to speedup the computing time. To do this, our methodology aims: at reusing the* systolic model *principles for the display of the potential parallelism inside nested loops, and justifying the the aggregation of iteration of loops so as to reduce communication overheads while exploiting coarse-grained parallelism. Each aggregation is a* bloc *of fine-grained computations not located in the same hyperplan of a given space. It also defines an atomic unit of computation i.e no synchronization or communication is necessary during the execution of the fine-grained computations inside a bloc. Thus all necessary data must be available before such atomic executions. This imposes the constraint that splitting the set of fine-grained computations does not result in deadlocks.*

KEY WORDS : Parallel machine, Systolic algorithm, Block splitting, Block allocation, Generalized eigenproblem, Lanczos method, Sparse matrix, Skyline storage.