RenPar'2000
Besançon, 19 juin - 22 juin 2000

Redistributions entre appels de routines ScaLAPACK

Frédéric Desprez(1), Cyrille Randriamaro(2)

(1)LIP, Ecole Normale Supérieure de Lyon, 69364 Lyon Cedex 07, France, desprez@ens-lyon.fr
(2)LaRIA, Université de Picardie Jules Vernes, 5, rue du Moulin Neuf, 80000 Amiens, crandria@laria.u-picardie.fr

---

Résumé
Nous nous intéressons à la transformation de programmes scientifiques qui utilisent des routines `LAPACK` vers des programmes `HPF` utilisant des routines `ScaLAPACK` sur des matrices distribuées grâces à des directives. Le temps d'exécution de chaque routine dépend directement de la distribution des matrices qu'elle manipule. De plus, si une matrice est distribuée différemment pour deux routines `ScaLAPACK`, il faut ajouter une redistribution, d'où un coût supplémentaire. Nous proposons ici des algorithmes et heuristiques pour trouver la ou les distributions optimales en tenant compte de coûts de redistribution.

Mots-clés : tableaux distribués, distribution bloc-cyclique, redistribution, ScaLAPACK, HPF.

---

1. Introduction

La programmation d'applications numériques sur machines parallèles à mémoire distribuée est aujourd'hui facilitée par l'utilisation de bibliothèques de haut niveau, comme ScaLAPACK, et de compilateurs de langages data-parallèles, comme HPF. Cependant, l'utilisateur doit gérer lui-même non seulement la distribution initiale des matrices sur les processeurs, mais aussi les redistributions éventuellement nécessaires au maintien d'un équilibrage correct de la charge au cour du temps.

Cet article s'intègre donc dans le cadre de l'optimisation de programmes faisant appel à des routines d'algèbre linéaire. Notre objectif est de trouver une distribution pour toutes les matrices d'un programme et d'intercaler ou non des redistributions entre les différents appels de fonctions d'algèbre linéaire.

Nos motivations viennent de l'observation du comportement des routines des bibliothèques `PBLAS` et `ScaLAPACK` [7]. Leurs performances dépendent beaucoup de la distribution des matrices utilisées, c'est à dire de la grille de processeurs $L \times C$ et de la taille $r \times s$ des blocs de données. Ainsi, pour optimiser un programme, une méthode intuitive serait d'utiliser, pour chaque instruction matricielle, la distribution optimale des matrices concernées. Si une matrice n'a pas la même distribution pour deux instructions matricielles, il suffit alors de la redistribuer. Or, ces redistributions génèrent un surcoût. Il peut donc être plus avantageux de garder une distribution pour l'exécution de plusieurs instructions, le surcoût occasionné par une mauvaise distribution pouvant être inférieur à celui d'une redistribution.

Après une présentation du problème, nous présentons un tour d'horizon des algorithmes permettant de gérer l'allocation dynamique des données. Nous construisons ensuite notre heuristique de façon à contrecarrer les difficultés rencontrées par les algorithmes précédents.

2. Problème de l'allocation dynamique

L'approche généralement utilisée repose sur le schéma suivant : le programme est considéré comme une succession de phases (chaque phase est généralement un nid de boucles ou l'appel d'une fonction qui travaille sur des matrices distribuées). Pour chaque phase, on choisit un ensemble de distributions, appelé choix de distributions, puis on sélectionne une des distributions de l'ensemble dans le but d'optimiser le temps d'exécution du programme global, en tenant compte du coût des redistributions éventuelles. Dans notre cas, nous considérons les phases comme des appels aux fonctions des bibliothèques `ScaLAPACK` et `PBLAS`. Dans cette section, nous modélisons les coûts de ces fonctions parallèles et des redistributions, puis, nous présentons une formulation du problème de l'allocation dynamique de données.

2.1. Modélisation des coûts des routines parallèles

Chaque routine `ScaLAPACK` et `PBLAS` est constituée d'une succession d'appels aux routines `BLAS` et `BLACS`. Les `BLAS` sont séquentiels et leurs temps d'exécution peuvent être mesurés et extrapolés pour toutes les tailles de matrices. Les routines séquentielles les plus coûteuses sont en $O(n^3)$. Il en est de même des routines `BLACS` de communication qui répondent au modèle $\beta + L \times \tau$ avec des communications point à point. Ainsi le coût total des routines `ScaLAPACK` et `PBLAS` est une somme de polynômes de degrés inférieurs ou égaux à 3, en fonction des paramètres de la distribution et de la taille des matrices.

Une somme de polynômes est dérivable. Ainsi, pour une grille $L \times C$ donnée, on est capable de calculer la taille optimale de blocs. D'une manière plus générale, on peut évaluer la distribution optimale d'une routine `PBLAS` ou `ScaLAPACK`, ainsi que la forme de la grille optimale, c'est-à-dire le ratio L/C. De même, on peut connaître la distribution statique optimale d'un ensemble d'instructions matricielles. Des exemples de telles modélisations sont donnés dans [13] et [2].

2.2. Modélisation des coûts des redistributions

Nous posons comme hypothèse que le nombre de processeurs est fini et fixé. Nous n'acceptons qu'une modification de la forme de la grille, c'est-à-dire du ratio L/C. Ainsi, toutes les redistributions se font sur la grille complète de processeurs. Il en est de même pour les matrices : même si les instructions matricielles n'utilisent qu'une partie de la matrice, les routines de redistribution modifient la répartition de la matrice entière. Pour estimer le coût d'une redistribution, nous calculons les messages que doit envoyer chaque processeur. Nous prenons alors comme coût celui de l'envoi des messages effectué par le processeur qui en a le plus. Cette modélisation coûteuse ne donnant pas de fonction dérivable.

2.3. Formulation du problème

Nous considérons, pour commencer, un programme parallèle qui ne contient que des appels à des fonctions `ScaLAPACK` et `PBLAS`. Une phase du programme est alors un simple appel de fonctions. Un tel programme peut se modéliser avec un graphe que nous appelons graphe des distributions :

Definition 1 *Soit un programme S, le* **graphe des distributions** *$G = (V, E)$ du programme S est le graphe pour lequel à chaque sommet correspond une instruction matricielle. Une arête relie alors deux sommets du graphe si, et seulement si, les deux phases correspondantes utilisent une même matrice A et toutes les instructions exécutées entre ces deux phases n'utilisent pas A.*

Ainsi, chaque sommet $v \in V$ est étiqueté par la distribution des différentes matrices utilisées. En utilisant ces distributions, le temps d'exécution de la phase correspondant à $v \in V$ définit le poids du sommet. De même, le poids de chaque arête $e \in E$ est défini par le temps d'exécution des redistributions.

Definition 2 *Soit $G = (V, E)$, le graphe des distributions d'un programme S, le* **coût $\Gamma(G)$ du graphe** *G est la somme des poids des sommets de V additionné de la somme des poids des arêtes de E, ce qui représente le temps d'exécution de S avec les distributions et redistributions déterminées par G.*

Nous pouvons à présent formuler le problème de la façon suivante :

Definition 3 *Soit $G = (V, E)$ le graphe des distributions d'un programme S. Le* **problème de l'allocation dynamique de données** *consiste à déterminer la distribution de chaque sommet du graphe G de façon à minimiser $\Gamma(G)$.*

Les fonctions de coût des routines des bibliothèques `ScaLAPACK` et `PBLAS` sont dérivables. Pour obtenir la meilleure distribution statique de tout un programme, il suffit de dériver la somme des fonctions de coûts des routines utilisées. Cependant, les fonctions de coût des routines de redistribution n'étant pas dérivables, cette stratégie simple n'est pas applicable pour le problème de l'allocation dynamique de données.

3. Etat de l'art

Dans cette section sont présentés des articles traitant de l'allocation dynamique de données. Les algorithmes sont décrits ici dans des termes différents de ceux utilisés dans les articles de ce paragraphe : nous ne faisons pas état de l'alignement des données (inutile avec les routines `ScaLAPACK`) et considérons qu'il existe une distribution optimale pour chaque instruction ou groupe d'instructions matricielles. Notre

objectif est de faciliter leur comparaison avec l'algorithme que nous présentons ici. Les principaux articles qui traitent de la redistribution de données considèrent le code source comme une succession de phases atomiques. Les algorithmes déterminent ensuite la distribution à attribuer à chaque phase dans le but d'optimiser le temps global d'exécution du programme.

3.1. Complexité du problème (Kennedy et Kremer)

Kennedy et Kremer [11] définissent une phase comme un ensemble d'opérations sur des matrices distribuées. Leur approche est principalement basée sur deux structures de données.

Premièrement, ils créent **le Graphe de Flot de Contrôle des Phases** (GFCP) : c'est un graphe de flot de contrôle dans lequel chaque sommet représente toutes les instructions d'une même phase. Les arêtes sont pondérées par la fréquence et la probabilité d'exécution.

Ils construisent alors **le Graphe de Distribution de Données** (GDD) : c'est une extension du GCFP. Chaque sommet est multiplié par le nombre de ses distributions potentielles et contient ainsi une phase et une distribution potentielle. Il y a une arête entre deux sommets si au moins une des matrices communes aux deux sommets doit être redistribuée entre le sommet source et le sommet destination. Les sommets (et arêtes) sont pondérés par l'estimation du temps d'exécution des instructions (et redistributions).

La détection d'une distribution pour chaque étape revient alors à chercher un *chemin minimal*: un chemin de poids minimum passant une fois par chacune des phases. Ils ont montré que chacun des deux problèmes était NP-Complet [8], aussi bien le choix de la liste des distributions pour construire le GDD, que la recherche du chemin minimal.

3.2. Restriction sur les distributions (Garcia et al, Lee et al)

Garcia, Ayguadé et Labarta [4] se limitent aux distributions 1D cycliques ou par bloc. Chaque instruction matricielle n'a alors le choix qu'entre deux distributions, ce qui permet de construire un GDD avec deux fois plus de sommets qu'il n'y a d'instructions matricielles. Les auteurs trouvent ensuite le chemin minimal en utilisant les techniques de Kennedy et Kremer.

Lee, Wang et Yang [9] intègrent de plus la distribution bloc-cyclique avec une taille de blocs unique et fixée (1D et 2D). En s'intéressant aux programmes d'imagerie, ils étudient des programmes qui ne travaillent que sur une seule matrice distribuée. Les boucles sont traitées séparément, ce qui évite les cycles dans le GDD. La recherche du chemin minimal est donc optimale avec l'algorithme de Dijkstra.

3.3. Regroupement des instructions (Anderson et Lam)

L'algorithme d'Anderson et Lam [6] travaille sur un graphe dont les sommets correspondent aux boucles et nids de boucles, et les arêtes aux redistributions de données potentielles entre les nids de boucles. Ces arêtes sont pondérées par le coût de leurs redistributions. Le poids des sommets représente le temps d'exécution des nids de boucles lorsque les matrices sont distribuées de façon optimale.

Ils sélectionnent la plus grosse arête. Les deux sommets des extrémités sont alors réunis, puis on calcule la meilleure distribution commune à ces sommets (dite *distribution statique*). Si le coût d'exécution des deux phases fusionnées, avec la distribution statique, est inférieur à la somme des coûts des deux précédentes distributions et de la redistribution, alors les deux sommets sont regroupés en un seul. Ceci est répété avec la plus grande arête suivante et ainsi de suite jusqu'à la dernière arête.

3.4. Division des phases (Palermo et Banerjee)

A l'inverse, Palermo et Banerjee [3] considèrent tout d'abord le programme comme une seule phase englobant toutes les instructions. La distribution statique optimale est choisie pour cette phase. Elle est alors de scindée en deux, générant deux phases de plus petites tailles, chacune ayant sa distribution statique optimale. Ainsi, toutes les divisions possibles sont testées pour choisir celle qui permet de minimiser le coût du programme, sans tenir compte du coût des redistributions éventuelles. Les deux phases sont alors également divisées en deux, et ainsi de suite, de façon récursive, jusqu'à ne plus avoir de division générant de coût inférieur à celui de la phase à diviser.

A chaque phase non divisée (dite *phase de base*) est alors associée sa distribution statique optimale, ainsi que la distribution de la phase mère (dont la division a produit cette phase de base), celle de la phase mère de cette dernière, et ainsi de suite jusqu'à la phase regroupant le programme global. Enfin, un graphe ressemblant au GDD est construit. Il est utilisé pour la recherche d'un chemin minimal.

3.5. Tous les regroupements de voisinage (Peizong Lee)

L'algorithme de Peizong Lee [10] considère le programme comme une liste d'instructions matricielles. Tous les regroupements possibles entre voisins sont définis : pour N phases, pour chaque sommet x, les regroupements définis sont (x), $(x,\ x+1)$, $(x,\ x+1,\ x+2)$, ..., $(x,\ ...,\ N)$. Pour chaque regroupement la distribution statique optimale est calculée. Ensuite, on construit un graphe dans lequel chaque regroupement est un sommet ; une arête relie deux sommets u et v si la dernière instruction de u précède directement la première instruction de v. En pondérant les sommets par leur temps d'exécution et les arêtes par les temps de redistribution des matrices, on peut alors chercher le chemin minimal.

3.6. Conclusion

Le tableau suivant présente un récapitulatif des caractéristiques des algorithmes présentés dans la section précédente. Nous appelons ici *regroupement* le fait d'unir les phases séquentiellement, à opposer au *découpage* qui fait une liste de tous les regroupements souhaités puis qui choisit parmi cette liste.

Algorithmes	Distributions possibles	Choix des distributions	Recherche d'un chemin
Garcia et al	cyclique et par bloc	-	NP-Complet
Lee et al	idem et Cyclic(r)	-	polynomial
Anderson et Lam	bloc-cyclique	regroupement	-
Palermo et Banerjee	bloc-cyclique	découpage	polynomial
Peizong Lee	bloc-cyclique	découpage	polynomial

Nous orientons nos recherches pour trouver une heuristique qui s'appuie sur les atouts des algorithmes existants : l'algorithme d'Anderson et Lam est puissant pour éliminer les grosses distributions en regroupant deux à deux les instructions. Cette méthode présente l'inconvénient de ne pas avoir une vue globale du programme et de risquer d'éliminer des redistributions au détriment d'autres plus petites. A l'inverse, les algorithmes de Palermo et Banerjee et de Peizong Lee ont cette vue globale. En revanche, les instructions ne sont réunies qu'avec leurs voisines : des instructions peuvent ne pas être réunies malgré une grosse redistribution qui pourrait fortement optimiser le programme.

4. Nouvel algorithme : l'algorithme de regroupement

Dans ce paragraphe, nous allons regrouper les avantages de chacun des algorithmes qui ont une vue globale. Pour cela, nous étudions chaque algorithme dont nous extrayons les éléments positifs pour construire le nôtre. Nous appellerons cet algorithme l'**algorithme de regroupement**.

4.1. Groupement en phases

Nous le prenons comme algorithme de référence celui de Peizong Lee, les paragraphes suivants ne feront que modifier son graphe en y apportant les éléments avantageux de chacun des autres algorithmes : au début, l'algorithme de regroupement calcule toutes les distributions de voisinage. Il crée un graphe avec les $N^2/2$ sommets représentant les différentes distributions et les instructions correspondantes. Ensuite, pour obtenir le chemin minimal, il applique sur ce graphe l'algorithme de Dijkstra, dont le coût est $O(N)$. Son coût total est donc $O(N^2)$.

4.2. Division des phases

L'algorithme de Palermo et Banerjee divise successivement les phases en deux. Ainsi, chaque phase ne contient que des instructions voisines. Les distributions générées par cet algorithme sont un sous-ensemble de celles générées par l'algorithme de Peizong Lee. Les choix des distributions de l'algorithme de regroupement ne sont donc pas modifiés.

En revanche, le graphe construit n'est pas le même car l'algorithme de Palermo et Banerjee considère que l'on peut couper un groupe d'instructions, alors que celui de Peizong Lee le refuse. Pour l'admettre, il faudrait modifier le graphe en attribuant à chaque instruction toutes les distributions correspondant aux groupes dont il fait parti. Ainsi, pour chaque instruction i, on associe toutes les distributions $D_{a,...,b}$ telles que $a \leq i \leq b$. On peut alors construire le même graphe que précédemment, dans lequel chaque sommet représente une instruction et une des distributions avec laquelle elle est associée.

Pour N instructions, chacune peut être associée au pire à N^2 distributions, d'où un nombre total de sommets inférieur à N^3, rajoutés aux précédents. On applique l'algorithme de Dijkstra, le coût de l'algorithme de regroupement est donc $O(N^3)$.

4.3. Ajout des distributions particulières

Les algorithmes de Garcia, Ayguadé et Labarta et de Lee, Wang et Yang présentent l'avantage de tester les distributions cycliques, par bloc et bloc-cycliques avec une taille de bloc fixée. Nous les intégrons dans l'algorithme de regroupement pour chaque grille de processeur possible. Sur des grilles complètes de processeurs, le nombre de grilles possibles est inférieur au nombre total de processeurs P. Ainsi, pour chaque distribution ajoutée, il suffit de dupliquer P fois les $N^2/2$ sommets générés par l'algorithme de Peizong Lee. A chaque duplication, nous associons la distribution sur une des grilles de processeurs possibles. Le coût de l'algorithme est donc $O((P+N)\times N^2)$.

4.4. Conclusion

L'algorithme de regroupement commence par la construction d'un graphe semblable à celui de Peizong Lee, dans lequel les phases sont divisées. Nous avons intégré les distributions cycliques, par bloc et $bloc-cyclique$ pour une taille de blocs fixée : cela n'ajoute que $3P\times N^2/2$ sommets au graphe, ce qui maintient le coût de l'algorithme de regroupement à $O((P+N)\times N^2)$.

Nous avons créé un algorithme qui regroupe les qualités de plusieurs algorithmes. Il a cependant le défaut de ne s'appliquer que sur une séquence d'instructions, sans tenir compte des redistributions coûteuses qu'il serait intéressant de retirer, si celles-ci ne relient pas deux instructions consécutives.

5. Algorithme des chemins maximaux

Nous avons remarqué que l'algorithme d'Anderson et Lam ne tenait compte que des distributions les plus coûteuses, sans avoir de vue globale du programme. Notre objectif est de garder la vision globale du programme comme le fait l'algorithme de regroupement, mais plutôt que de regrouper les instructions suivant leur voisinage, notre but est de le faire suivant le coût des redistributions potentielles.

5.1. Graphe non orienté

Le point de départ de l'algorithme est le graphe des distributions décrit au paragraphe 2.3. On peut extraire le chemin élémentaire maximal et y appliquer l'algorithme de regroupement. On remarque alors que l'orientation des arêtes n'apporte rien : d'une part, le coût de redistribution reste le même si on échange la source et la destination ; d'autre part, l'algorithme de regroupement ne s'en trouve pas affecté puisque le regroupement de phases ne sert qu'à trouver une distribution statique commune.

5.2. Application de l'algorithme de regroupement

Nous disposons donc d'un graphe des distributions non orienté pour lequel chaque sommet représente une phase, donc une instruction. Chaque arête représente la redistribution qui sera effectuée si les deux phases aux extrémités n'ont pas la même distribution. Notre objectif est maintenant d'en extraire un chemin élémentaire maximal. Puisque le graphe n'est pas orienté et que plusieurs matrices peuvent être utilisées par une même instruction, le graphe peut comporter des cycles. Dans ce cas, le problème de recherche du chemin élémentaire maximal est NP-complet et nécessite l'utilisation d'heuristiques [5]. Cependant, pour un faible nombre d'arêtes, comme c'est souvent le cas, on peut faire une recherche exhaustive de tous les chemins possibles et prendre le plus coûteux. Une fois le chemin trouvé, nous appliquons l'algorithme de regroupement sur la suite des instructions ainsi formée.

5.3. Récursion

Nous avons maintenant un ensemble de sommets pour lesquels nous avons trouvé une distribution qui minimise le temps global d'exécution des instructions qu'ils représentent. Cependant, le chemin maximal ne contient pas forcément tous les sommets du graphe. Dans ce cas, pour traiter les autres sommets, nous regroupons les sommets du chemin précédemment trouvé en un seul sommet C. Une arête reliera un sommet n avec C, si et seulement si, il existait déjà une arête entre n et un des sommets regroupés dans C. Le poids du sommet C est le coût trouvé lors de l'application de l'algorithme de regroupement. On peut ainsi recommencer le choix d'un chemin élémentaire maximal, et l'application de l'algorithme de regroupement, et ainsi de suite jusqu'à ce qu'il ne reste plus qu'un seul sommet dans le graphe.

5.4. Analyse et conclusion

Nous avons montré que le coût de l'algorithme de regroupement était polynomial. Le chemin élémentaire maximal d'un graphe peut être trouvé, par heuristique, en un temps polynomial. Le plus petit chemin maximal d'un graphe connexe est de taille 3, aussi l'algorithme de regroupement est-il appelé moins de $N/(3-1)$ fois (les sommets d'un chemin maximal sont regroupés en un nouveau sommet qui peut appartenir au chemin suivant). Le coût de l'algorithme des chemins maximaux, dans sa globalité, est donc polynomial en fonction du nombre d'instructions (appels à des routines parallèles) du programme.

Jusqu'à présent, nos programmes sources ne comportaient que des appels à des fonctions parallèles. L'insertion des boucles et des conditions se fait en ajoutant des arêtes suivant la méthode d'Anderson et de Kremer [12]. Ces ajouts ne changent en rien l'algorithme, le graphe contenant déjà des cycles positifs.

6. Conclusion

Pour réaliser un algorithme qui gère l'allocation dynamique des données, nous avons étudié 5 algorithmes, et nous avons regroupé les 4 qui fonctionnaient suivant le même schéma : la construction d'un graphe dans lequel les sommets sont des phases associés à une redistribution statique. Cependant, cet algorithme ne regroupe que des instructions consécutives.

Pour pallier cette limitation, nous prenons un chemin élémentaire maximal du graphe des distributions. Les successions d'instructions sont alors liées aux coûts des redistributions et non à l'ordre d'exécution. Une analyse de l'heuristique nous a montré que son coût est polynomial. Nous avons observé [13] que, pour certains programmes, quelques-uns des algorithmes présentés pouvaient *garantir* une solution optimale. Dans ces mêmes cas, l'algorithme des chemins maximaux peut le faire aussi.

Bibliographie

1. J.M. Anderson and M.S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. *ACM Sigplan Notices*, 28(6):112–125, June 1993.
2. S. Domas. *Contribution à l'Ecriture d'une Bibliothèque d'Algèbre Linéaire Parallèle.* PhD thesis, Ecole Normale Supérieure de Lyon, October 1998.
3. D.J.Palermo and P.Banerjee. Interprocedural Array Redistribution with Data-flow Analysis. In *Proceedings of the 9th Annual Workshop on Languages and Conpilers for Parallel Computing*, Aug. 96.
4. J. Garcia, E. Ayguadé, and J. Labarta. Dynamic Data Distribution with Control Analysis. In *Supercomputing'96*, November 1996.
5. M. Gondran and M. Minoux. *Graphes et Algorithmes.* Editions Eyrolles, 1985.
6. J.Anderson. *Automatic Computation and Data Decomposition for Multiprocessors.* PhD thesis, Stanford University, 1997.
7. J.Choi, J.J. Dongarra, R. Pozo, and D.W.Walker. ScaLAPACK : A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers. In *Frontiers'92 : Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. Mac Lean, October 1992.
8. U. Kremer. Np-completeness of Dynamic Remapping. In *Fourth Workshop on Computers for Parallel Computers*, December 1993.
9. G. C. Lee, Y.-F. Wang, and T. Yang. Global Optimization for Mapping Parallel Image Processing Tasks on Distributed Memory. volume 45, pages 29–45. Academic Press, 1997.
10. Peyzong Lee. Efficient Algorithms for Data Distribution on Distributed Memory Parallel Computer. 8:825–839, August 1997.
11. R.Bixby, K.Kennedy, and U.Kremer. Automatic Data Layout using 0-1 Integer Programming. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT94)*, pages 111–122, Montreal, Canada, August 1994.
12. E.Ayguadé J.Garcia U. Kremer Tools On Techniques for Otomatic Data Layout: A Case Study in *Parallel Computing* 24:557-578 , 1998
13. C. Randriamaro. *Optimisation des Redistribution et Allocation Dynamique en Parallélisme de Données.* PhD thesis, Ecole Normale Supérieure de Lyon, January 2000.