RenPar'9,
Lausanne. Suisse, 20–23 mai 1997.

Performance des macro-pipelines dans les programmes data-parallèles

Frédéric Desprez [a*], Julien Zory [b]

[a]LIP, CNRS URA 1398 et INRIA Rhône-Alpes, ENS-Lyon, Frederic.Desprez@inria.fr

[b]CRI, Ecole des Mines de Paris, Julien.Zory@cri.ensmp.fr

Résumé

Dans cet article, nous présentons une étude de performances des calculs pipelinés dans le cadre de l'optimisation des communications pour un compilateur de langages data-parallèles du type High Performance Fortran.

Mots clef: compilation, calculs pipelinés, analyse de performances, recouvrements

1. Introduction

L'utilisation des machines parallèles pour le calcul numérique passe maintenant par l'emploi de compilateurs data-parallèles et de langages comme High Performance Fortran. Ces compilateurs s'occupent plus ou moins automatiquement d'extraire le parallélisme du code et de générer les communications nécessaires à l'exécution correcte du programme parallèle. Ces communications doivent être optimisées afin d'obtenir des performances proches de celles des programmes codés "à la main" s'appuyant sur des bibliothèques de passage de message.

Dans cet article, nous abordons le problème de l'optimisation des calculs pipelinés, c'est-à-dire des algorithmes dont les dépendances entre les calculs et la distribution des données sur les processeurs font que l'exécution du programme sur une machine parallèle conduit à une exécution partiellement séquentielle. Une solution pour contourner ce problème consiste à redistribuer les données de manière à éviter que les dépendances ne traversent l'espace des processeurs; mais cela n'est bien sûr pas toujours possible et le coût de ces mouvements de données peut s'avérer important. Une autre solution consiste, lorsque c'est possible, à pipeliner les calculs et les communications. Il faut alors transformer les nids de boucles en y ajoutant un pas supplémentaire sur les boucles pipelinées et ainsi envoyer les données par blocs dès qu'elles ont été calculées. On obtient alors un macro-pipeline. Le pas choisi dépend des rapports entre les coûts de calcul, la latence des communications et leur vitesse. De plus, on peut utiliser des émissions asynchrones pour masquer les communications. Cet article propose donc une étude analytique du gain en performances obtenu par utilisation du macro-pipeline dans un programme data-parallèle exécuté sur P processeurs. Notre étude fait suite à un article récent de M.J. Quinn et P.J. Hatcher [4] qui présente le gain issu de l'utilisation de diverses techniques de recouvrement en ne considérant qu'un seul processeur. Nous étudions les diverses méthodes d'implémentation de macro-pipelines pour les distributions mono-dimensionnelles et nous étendons ces résultats aux cas multi-dimensionnels. Cette étude se situe dans le cadre de l'implantation de ces techniques dans le compilateur HPF Adaptor du GMD/SCAI.

Après une rapide présentation du problème abordé et de l'état de l'art dans le paragraphe 2, nous présentons dans le paragraphe 3 l'étude du gain dans les cas mono et multi-dimensionnel. Enfin, nous terminons par quelques conclusions et perspectives pour des travaux futurs.

*. Recherche soutenue par le projet CNRS–ENS Lyon–INRIA *ReMaP* et le projet EUREKA-EUROTOPS.

2. Problème et état de l'art

La présence de dépendances entre les communications et les calculs interdit l'utilisation directe des communications asynchrones sans utiliser un macro-pipeline. Ainsi, des codes très simples tels que celui présenté dans la figure 1, présentent un flux de données traversant l'espace des processeurs (figure 2) et dont l'exécution data-parallèle sans optimisation aboutirait à de mauvaises performances. La réduction du grain de calcul schématisée sur la figure 3, permet un démarrage plus rapide des processeurs. Il sera cependant nécessaire de trouver un bon compromis entre démarrage au plus tôt et exécution séquentielle du programme, et ceci en raison des surcoûts occasionnés par le nombre croissant de communications (et de leur temps d'initialisation).

```
      REAL A(N,N)
!HPF$ DISTRIBUTE A(BLOCK,BLOCK)
      ...
    DO I=2,N
     DO J=2,N
(S)   A(I,J) = f(A(I-d,J),
$       A(I,J-d'))
     END DO
    END DO
```

FIG. 1 –. Type de code concerné.

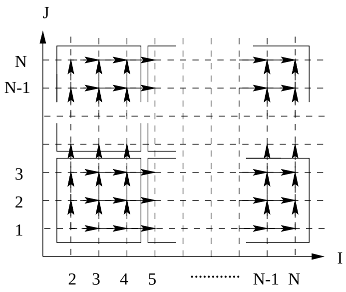


FIG. 2 –. Le schéma pipeline.

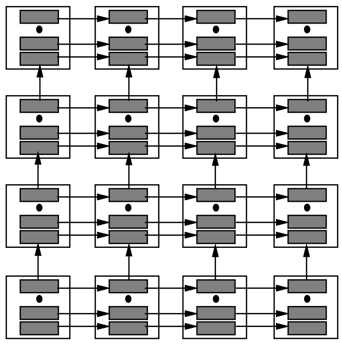


FIG. 3 –. La solution du macro-pipeline.

Plusieurs compilateurs de langages data-parallèles proposent aujourd'hui une implémentation de la technique présentée ci-dessus et parmi eux FortranD [2], Paradigme [3], Adaptor [1] ou encore le récent compilateur HPF d'IBM. Cependant, la seule étude poussée concernant les performances envisageables par utilisation de ces techniques, est à notre connaissance celle de M.J. Quinn et P.J. Hatcher [4]. Les auteurs présentent plusieurs résultats, en limitant leur étude au gain obtenu sur un processeur.

3. Étude des performances

3.1. Modélisation

Nous supposons que la machine utilisée possède une mémoire distribuée et qu'elle échange les données à travers un réseau de communications. Les calculs seront notés $C_r(n)$ et nous appelons $T(n)$ le coût de communication de n éléments entre deux processeurs. Ce dernier est modélisé par une fonction linéaire de la taille du message $T(n) = \beta_T + \tau_T n$. De même, nous représentons la latence des communications (bufferisation + appel à `send/receive`) par la fonction $L(n) = \beta_L + \tau_L n$. Nous appelons $L'(n)$ le coût de réception du message et supposons qu'il est égal à $L(n)$. Nous supposons que nous pouvons recouvrir totalement le coût $T(n)$ d'une communication mais pas le terme dû à la mise en place de cette communication et à la bufferisation système ($L(n)$ et $L'(n)$). Cela est réaliste pour les machines actuelles

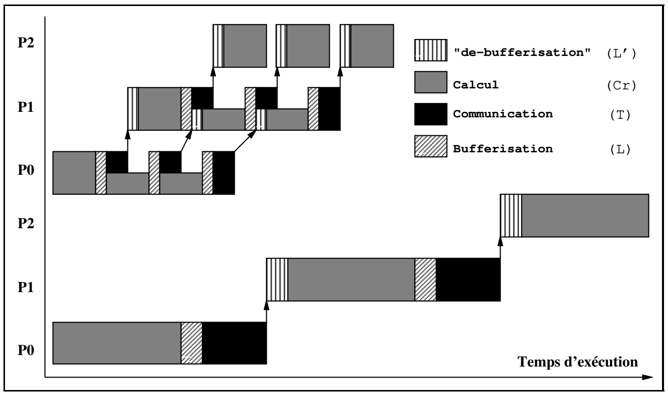


FIG. 4 – Gain en performance pour P processeurs.

et leurs bibliothèques de communication. Les paramètres qui peuvent varier sont : la dimension des matrices (N_m), le nombre de dimensions distribuées (N_d) et les dépendances (vecteurs de la forme $(0,\ldots,0,d_i>0,0,\ldots,0)$ pour $0 \leq i \leq N_m-1$). Nous supposons qu'il existe au moins un vecteur de dépendances pour chacune des dimensions distribuées d_i. Sans quoi plusieurs macro-pipelines de dimensions inférieures seront lancés en parallèle.

3.2. Analyse du pipeline sur des matrices distribuées selon une dimension

Le gain en performance obtenu par utilisation du macro-pipeline est donné par le rapport entre les temps d'exécution avec (T_{np}) et sans macro-pipeline (T_p), et sera noté $P(n)$. Nous supposons que nous découpons les données et les calculs pour obtenir une taille n_0.

$$\begin{aligned} T_{np} &= PC_r(n) + (P-1)[L'(n) + T(n) + L(n)] \quad (1) \\ T_p &= PC_r(n_0) + (P-1)[L'(n_0) + T(n_0) + L(n_0)] \\ &\quad + \left(\frac{n}{n_0} - 1\right)[\max(L'(n_0) + C_r(n_0) + L(n_0), T(n_0))] \quad (2) \end{aligned}$$

T_{np} décrit une exécution séquentielle des P processeurs à laquelle s'ajoutent les phases de communication (partie basse de la figure 4). T_p est quant à lui composé d'un premier terme décrivant le démarrage de tous les processeurs, puis d'une seconde partie caractéristique de l'exécution *avec recouvrements* des calculs du dernier processeur (partie haute de la figure 4). Le résultat présenté par l'équation 3 prouve la réelle efficacité du macro-pipeline pour des problèmes dont la taille est suffisante pour compenser les coûts de latence de communications ($n^2\tau_c \gg \lambda_T + \beta_T n$).

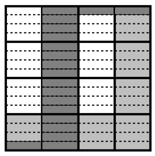
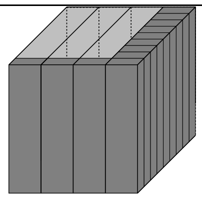
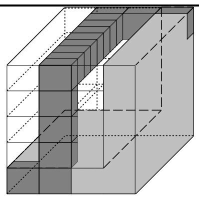
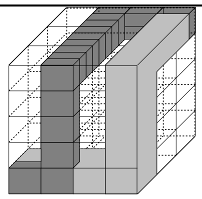
$$\lim_{n\to+\infty} P(n) = \frac{P \times O(n^2) + 3(P-1) \times O(n)}{(P-1) \times O(n) + 3(p-1) \times O(1) + O(n^2) + 2O(n) - 2O(1)} = P \quad (3)$$

3.3. Extension aux distributions multi-dimensionnelles

Nous généralisons maintenant l'étude précédente à des matrices multi-dimensionnelles distribuées par blocs sur des grilles multi-dimensionnelles de P processeurs. Avec ce type de distribution et de dépendances (une dans chaque direction), il existe du parallélisme intrinsèque puisque les processeurs peuvent travailler par "vagues". Une optimisation macro-pipeline est toutefois possible et nous permet d'obtenir des performances améliorées, notamment dans le cas de matrices de taille suffisamment importante.

Le temps d'exécution de l'algorithme *sans* macro-pipeline peut être exprimé comme la somme de $N_d \times \sqrt[N_d]{P}$ étapes de calculs, séparées par des phases de communications. L'algorithme progresse de manière séquentielle le long de chacune des N_d dimensions distribuées sur $\sqrt[N_d]{P}$ processeurs. Nous avons donc $T_{np}^{N_d} \simeq N_d \times \sqrt[N_d]{P} \times O(n^{N_m}) + C_1 \times O(n^{N_m-1})$ (où C_1 est un terme constant). Soit i la dimension choisie pour effectuer le macro-pipeline. L'exécution le long de cette dimension est séquentielle. Ainsi, si NP_i est le nombre de processeurs selon la dimension i, nous avons NP_i étapes de calculs de complexité $O(n^{N_m})$. Par ailleurs, comme la complexité des calculs le long des autres dimensions est réduite à $O(n^{N_m-1})$ en raison du macro-pipeline, nous obtenons $T_p^{N_d} \simeq NP_i \times O(n^{N_m}) + C_2 \times O(n^{N_m-1})$ (où C_2 est un terme constant). L'étude aux limites du gain en performances nous donne le résultat général présenté par l'équation 4. Le tableau 3.3 résume les performances qui peuvent être obtenues dans les cas classiques.

$$G \simeq \frac{N_d \times \sqrt[N_d]{P} \times O(n^{N_m}) + C_1 \times O(n^{N_m-1})}{NP_i \times O(n^{N_m}) + C_2 \times O(n^{N_m-1})} \simeq \frac{N_d \times \sqrt[N_d]{P}}{NP_i} \quad (4)$$

Chemin critique	Performances	Chemin critique	Performances
	$N_m = 2$, $N_d = 2$ $\sqrt{P} \times \sqrt{P}$ $G \simeq 2$		$N_m = 3$, $N_d = 1$ $P \times 1 \times 1$ $G \simeq P$
	$N_m = 3$, $N_d = 2$ $\sqrt{P} \times \sqrt{P} \times 1$ $G \simeq 2\sqrt{P}$		$N_m = 3$, $N_d = 3$ $\sqrt[3]{P} \times \sqrt[3]{P} \times \sqrt[3]{P}$ $G \simeq 3$

TAB. 1 –
Performances pour N_d dimensions distribuées

3.4. Macro-pipeline multi-dimensionnel

Lorsque les matrices utilisées sont de taille moyenne, il est possible d'augmenter le nombre de recouvrements calculs-communications en pipelinant plusieurs dimensions simultanément. En effet, pour obtenir de bonnes performances lorsque les communications coûtent cher (n est trop petit pour que l'on puisse supposer $C_r(n) \gg T(n)$), il devient nécessaire de réduire au maximum les termes de complexité $O(n^{N_m-1})$. Le macro-pipeline multi-dimensionnel va alors permettre de recouvrir les communications présentes selon la dimension pipelinée. De plus, le démarrage des processeurs sera encore une fois avancé. La figure 5 montre l'exécution d'un pipeline à deux dimensions sur une matrice 3D.

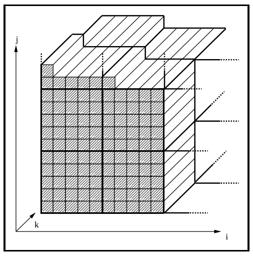


FIG. **5** – Macro-pipeline 2D sur une matrice 3D

4. Conclusion et travaux futurs

L'étude des performances des macro-pipelines nécessite encore de nombreuses analyses, notamment en ce qui concerne le choix des tailles de messages, celui des dimensions à pipeliner et surtout le choix entre pipeline et redistributions. Nous avons démontré dans cet article que ce type d'optimisation pouvait conduire à des performances intéressantes dans le cas des distributions mono et multi-dimensionnelles. Nos travaux actuels concernent la poursuite de l'implantation de ces techniques sous forme de bibliothèques dans le compilateur Adaptor et des expérimentations sur quelques machines cibles (IBM SP-2, Intel Paragon et Cray T3D).

Bibliographie

1. T. Brandes and F. Desprez. Implementing Pipelined Computation and Communication in an HPF Compiler. In *Europar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 459–462. Springer Verlag, August 1996.
2. S. Hiranandani, K. Kennedy, and C. Tseng. Evaluating Compiler Optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, 21:27–45, 1994.
3. D.J. Palermo, E. Su, J.A. Chandy, and P. Banerjee. Communication Optimizations Used in the Paradigm Compiler for Distributed Memory Multicomputers. In *International Conference on Parallel Processing*, St Charles, IL, August 1994.
4. Michael J. Quinn and Philip J. Hatcher. On the Utility of Communication–Computation Overlap in Data-Parallel Programs. *Journal of Parallel and Distributed Computing*, 33(02):197, 1996.