Premières expérimentations avec le système de transformation pour l'exécution parallèle STEP

Daniel Millot, Alain Muller, Christian Parrot, Frédérique Silber-Chaussumier

RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

Premières expérimentations avec le système de transformation pour l'exécution parallèle STEP

Daniel Millot, Alain Muller, Christian Parrot, Frédérique Silber-Chaussumier

Institut TELECOM, Télécom SudParis, Département Informatique, 91011 Évry, France,
{ Daniel.Millot, Alain.Muller, Christian.Parrot, Frédérique.Silber-Chaussumier } @it-sudparis.eu

Résumé

L'exploitation d'une application sur une plate-forme à mémoire distribuée requiert une parallélisation utilisant le paradigme de passage de messages, a priori réservée à des programmeurs experts du parallélisme. Pour faire évoluer cette situation, nous proposons une approche reposant sur un outil de transformation source-à-source appelé STEP (*Système de Transformation pour l'Exécution Parallèle*), qui permet de générer automatiquement une version par passage de messages d'un programme à partir d'un code parallèlisé à l'aide de directives OpenMP. Cet article présente puis évalue cet outil sur des benchmarks classiques. Cette évaluation montre l'intérêt de cette approche, les limites du prototype actuel et suggère des améliorations.

1. Introduction

Pour atteindre un niveau de performance satisfaisant, une application de calcul intensif (HPC) doit être adaptée à la plate-forme d'exécution cible. Il est donc inévitable d'utiliser des algorithmes parallèles différents pour des cibles aux caractéristiques architecturales aussi différentes que, par exemple, un multi-cœur, une grappe de multi-cœurs ou une grille de calcul, ce qui rend le portage d'applications parallèles très coûteux. Or, qu'il veuille développer une nouvelle application parallèle ou bien produire une version parallèle d'une application séquentielle, le développeur doit a priori opter pour l'un des deux paradigmes classiques de programmation parallèle liés aux deux principales organisations mémoire : passage de messages pour la mémoire distribuée, ou variables partagées pour la mémoire partagée. Mais, en l'absence de modèles de programmation parallèle hybrides, ce choix est difficile, car bien que la plupart des plates-formes multi-processeurs actuelles soient au plus haut niveau des architectures à mémoire distribuée, interconnectant un nombre parfois considérable de nœuds, chaque nœud a lui-même en général une architecture à mémoire partagée, souvent à plusieurs niveaux (multi-processeur avec partage de la mémoire du nœud, chaque processeur pouvant être multi-cœur, avec partage de différents niveaux de cache). Par ailleurs, chacun des deux paradigmes de programmation parallèle a des avantages et des inconvénients. L'utilisation de MPI permet un contrôle fin des communications entre processus, mais demande une expertise conséquente et donne une vue très fragmentée du programme, du fait du mélange du code purement MPI avec le code séquentiel initial qui rend la lecture du code difficile. Au contraire, l'utilisation d'OpenMP permet de conserver une vision globale du programme séquentiel initial. Cependant, elle est a priori réservée essentiellement à des plates-formes à mémoire partagée alors que l'augmentation du nombre de processeurs des plates-formes d'exécution s'appuie plutôt sur un modèle d'architecture à mémoire distribuée. Il faudrait donc pouvoir profiter du confort d'une approche OpenMP tout en bénéficiant de l'efficacité d'une programmation MPI.

Nous avons proposé dans [1] une approche qui vise à allier le meilleur des deux paradigmes grâce à une transformation source-à-source. Elle préserve la simplicité de la programmation avec des directives OpenMP tout en produisant sans effort supplémentaire des codes MPI prêts à l'emploi susceptibles d'être optimisés par un expert pour une exécution plus efficace. De nombreux autres travaux [2, 3, 4, 5, 6, 7] visent à permettre l'exécution d'un programme OpenMP sur une architecture à mémoire distribuée. La section 4 de l'article [1] est consacrée à l'état de l'art sur ces travaux. Dans l'article en question [1], une comparaison d'un point de vue conceptuel de STEP avec des travaux proches sur cette problématique a été effectuée. Le présent article est une première confrontation des performances du code produit par le prototype courant de STEP avec celles de codes produits d'autres manières, sur un ensemble de benchmarks de référence ; sans insister sur les améliorations apportées depuis cette première publication.

La section 2 décrit le fonctionnement de STEP. La section 3 commente les résultat obtenus sur une série de benchmarks. La dernière section présente des pistes d'évolution de l'outil STEP.

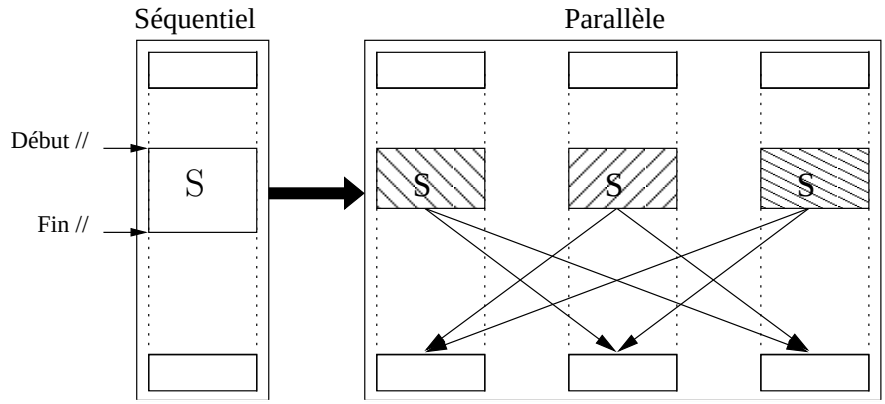

FIG. 1 – Modèle d'exécution

2. Description du projet STEP

Le développement de STEP repose sur le système d'analyse, de transformation et de compilation PIPS [8]. PIPS est pour l'instant opérationnel pour des applications écrites en Fortran. Il permet en particulier des analyses interprocédurales de régions de tableaux.

Pour produire rapidement un prototype susceptible de valider la production automatique d'un code source parallèle fonctionnant par passage de messages à partir d'un code source parallélisé à l'aide de directives OpenMP, un des critères de choix du modèle d'exécution du programme parallèle obtenu grâce à STEP a été la simplicité de mise en œuvre. Avec le modèle choisi (voir la figure FIG.1), les données sont allouées pour tous les processus et actualisées à la fin de chaque construction *workshare* d'OpenMP, en communiquant aux autres processus les données susceptibles de servir ultérieurement. Chaque processus exécute, de manière redondante, le code séquentiel situé en dehors des sections parallèles. Le fonctionnement de l'outil de transformation source-à-source développé dans le cadre de STEP peut être découpé en trois étapes : l'outlining guidé par les directives OpenMP, l'analyse de régions et le codage MPI.

2.1. L'outlining

Cette étape a pour but de faciliter l'exécution dans des contextes différents des sections parallèles d'un code. Dans le programme produit par STEP à cette étape, le code de chaque section parallèle est remplacé par l'appel d'une procédure exécutant le code de la section considérée (une procédure par section). Cette transformation préserve la sémantique du programme original. Par exemple, le listing 1 contient une boucle étiquetée 20 qui, dans le listing 2, a été remplacée par un appel à la procédure $P1\_DO20$.

Listing 1 – Avant outlining

```
      PROGRAM P1
      INTEGER I,N,F1
      PARAMETER (N=10)
      INTEGER T(N,2), A(N-1)
       ...
!$OMP PARALLEL DO
      DO 20 I = 1, N-1
         A(I) = F1(T, I)
20    CONTINUE
!$OMP END PARALLEL DO
       ...
      END
```

Listing 2 – Après outlining

```
      PROGRAM P1
      INTEGER I,N,F1
      PARAMETER (N=10)
      INTEGER T(N,2), A(N-1)
       ...
!$OMP PARALLEL DO
      CALL P1_DO20(I, 1, N-1, N, A, T)
!$OMP END PARALLEL DO
       ...
      END

      SUBROUTINE P1_DO20(I, I_L, I_U, N, A, T)
      INTEGER I, I_L, I_U, N, F1
      INTEGER A(1:N-1), T(1:N, 1:2)
      DO 20 I = I_L, I_U
         A(I) = F1(T, I)
20    CONTINUE
      END
```

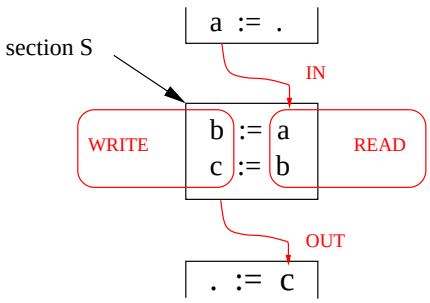

FIG. 2 – Régions IN, OUT, $READ$ et $WRITE$.

Les paramètres de la procédure obtenue par outlining sont :

– les paramètres de boucle : identifiant de l'indice de boucle et bornes inférieure et supérieure respectives de cet indice (paramètres I, I_L et I_U dans le listing 2),
– les paramètres intra-boucle : identifiant de toutes les variables internes à la boucle et taille de ces variables lorsqu'il s'agit de tableaux (paramètres N, A et T dans le listing 2).

2.2. L'analyse de régions

L'objectif de cette étape est de déterminer les messages qui vont être échangés entre les processus à l'issue de chaque section parallèle, lors de chaque phase d'actualisation des données des processus, imposée par notre modèle d'exécution. Cette description repose sur l'analyse de régions de tableaux possible avec PIPS.

À chaque tableau présent dans la section de code analysée par PIPS est associé un polyèdre convexe qualifiable de plusieurs manières possibles. Il peut être étiqueté $READ$ ou $WRITE$ si les cellules qu'il contient sont respectivement accédées en lecture ou bien en écriture par des instructions contenues par la section de code en question ; voir la figure FIG.2.

Il peut être étiqueté IN (respectivement OUT) si les données qu'il contient sont lues (respectivement écrites) dans la section de code considérée et écrites dans une section de code précédente (respectivement lues dans une section de code suivante) ; voir la figure FIG.2.

L'interprocéduralité de l'analyse produite par PIPS rend applicable les règles d'analyse évoquées ci-dessus non seulement aux instructions contenues par la section de code considérée, mais aussi au code des procédures appelées par la section de code en question.

Cette analyse peut fournir un résultat qualifié d'exact lorsque tous les éléments contenus par la région étiquetée $READ$, $WRITE$, IN ou OUT peuvent être qualifiés de la même manière que la région concernée. Dans le cas contraire, la région étiquetée constitue une surapproximation.

L'introduction de directives OpenMP dans un programme séquentiel découpe implicitement son code en une succession de sections de codes dont certaines sont parallèles. En combinant les étiquettes ($READ$, $WRITE$, IN ou OUT) obtenues grâce à ces analyses, STEP peut minimiser la taille des régions de tableaux qui vont être communiquées entre processus, pour garantir la cohérence des données entre les différents processus à la fin de chacune des constructions *workshare* de code. Ces régions de tableaux reçoivent l'étiquette $SEND$. Pour une section de code parallèle donnée, la région $SEND(T)$ d'un tableau T est nécessairement contenue dans la région $OUT(T)$ de ce tableau. Comme il est inutile d'échanger avec un autre processus une région de tableau dans laquelle la section de code parallèle concernée n'a pas écrit, il convient de restreindre le message à envoyer à la région $WRITE(T)$. En définitive,

$$SEND(T) = OUT(T) \cap WRITE(T).$$

Une région $SEND$ est un polyèdre convexe quelconque, comme intersection de polyèdres convexes quelconques. Pour simplifier la construction du message, c'est une enveloppe rectangulaire de ce polyèdre qui constitue le message effectivement envoyé.

Si la région de tableau reçue dans le message n'a pas été modifiée durant l'exécution de la section parallèle par le processus receveur, alors cette région est intégralement recopiée à la place correspondante dans le tableau considéré. Sinon, une différentiation cellule par cellule des deux régions de tableaux (celle résidente et celle reçue) permet de savoir quelles cellules de la région de tableau contenue dans le message doivent être recopiées dans le tableau résident. Cette opération qui peut être coûteuse a lieu lors de l'exécution du programme et réduit ses performances.

2.3. Le codage MPI

À partir du code source produit à l'étape 1, et en s'appuyant sur les régions $SEND$ obtenues lors de l'étape 2, le code source appelant les primitives de communication MPI est généré lors de cette troisième étape. Par exemple, pour une directive OpenMP de type "parallel do", STEP crée une nouvelle procédure suffixée par $\_MPI$. Cette procédure remplace la procédure produite par outlining. Ainsi, sur le listing 3 on peut voir que l'appel à la procédure $P1\_DO20\_MPI$ remplace celui à la procédure $P1\_DO20$ du listing 2. Cette nouvelle procédure est composée de quatre parties :

- La première partie, lignes 15 à 17, découpe la boucle de façon à répartir le traitement sur les différents processus en fonction du rang de chacun.
- Puis, compte-tenu de ce découpage de la boucle, les régions $SEND$ (voir section 2.2) des différents tableaux impliqués dans la boucle sont déterminées, ligne 20.
- Dans la partie trois, lignes 23 à 26, la procédure remplacée est appelée avec les bornes de l'indice de boucle (précédemment déterminées) adaptées à chaque processus .
- Enfin, chaque processus échange ses régions de tableaux qualifiées $SEND$ avec les autres processus, ligne 29.

Listing 3 – Fichier source du code parallèle MPI.

```
1       PROGRAM P1
2       include "step.h"
3  C    declarations
4       CALL STEP_Init
5       ...
6       CALL P1_DO20_MPI(I, 1, N-1, N, A, T)
7       ...
8       CALL STEP_Finalize
9       END
10
11      SUBROUTINE P1_DO20_MPI(I, I_L, I_U, N, A, T)
12      include "step.h"
13 C    declarations
14
15 C    Loop splitting
16      CALL STEP_SizeRank(STEP_Size, STEP_Rank)
17      CALL STEP_SplitLoop(I_L, I_U, 1, STEP_Size, I_STEP_SLICES)
18
19 C    SEND region computing
20      CALL STEP_COMP_SEND(A_STEP_SR,...)
21
22 C    Where the work is done
23      I_IND = STEP_Rank+1
24      I_LOW = I_STEP_SLICES(LOWER,I_IND)
25      I_UP = I_STEP_SLICES(UPPER,I_IND)
26      CALL P1_DO20(I_IND, I_LOW, I_UP, N, A, T)
27
28 C    SEND regions communications
29      CALL STEP_AllToAllRegion(...)
30      END
```

3. Premiers résultats sur quelques applications en calcul scientifique

L'objectif de cette section est de cerner expérimentalement un contexte favorable à l'utilisation de STEP. Pour cela, les performances (durée d'exécution, accélération) de diverses exécutions de programme sont comparées.

Différents types d'exécution.

Dans la suite, les différents types d'exécutions sont désignés par :
- SER pour l'exécution séquentielle obtenue à partir du code source séquentiel ;
- OMP pour l'exécution parallèle en mémoire partagée obtenue à partir du code source séquentiel auquel des directives OpenMP (exprimant le parallélisme) ont été ajoutées ;
- STEP pour l'exécution parallèle en mémoire distribuée obtenue à partir du même code source que celui utilisé pour l'exécution OMP, transformé par STEP ;
- KMP pour l'exécution parallèle en mémoire distribuée obtenue à partir du même code source que celui utilisé pour l'exécution OMP en utilisant l'outil Cluster OpenMP (distribué avec le compilateur fortran Intel [9]). Rappelons que Cluster OpenMP s'appuie sur un système à mémoire partagée distribuée (DSM) qui donne une vision unifiée de la mémoire des cœurs de calcul, qui peut être physiquement distribuée, et permet sans effort de programmation supplémentaire une exécution en mémoire distribuée ;
- MPI pour désigner l'exécution parallèle obtenue à partir d'un code source produit manuellement et contenant des appels aux primitives de communication MPI entre les processus.

Les performances obtenues par l'éxécution STEP sont systématiquement comparées à celles obtenues par les exécutions SER et OMP. Cette comparaison a lieu avec les performances obtenues par une exécution MPI lorsqu'elle existe et avec celles obtenues par l'exécution KMP lorsque ses résultats sont exploitables. L'objectif de cette confrontation est d'analyser les points faibles et les points forts de STEP afin d'orienter les développements futurs.

Différents benchmarks.

Certains des programmes qui ont servi à cette étude ont été sélectionnés car ils font partie des benchmarks de référence et d'autres parce que leurs caractéristiques (essentiellement le ratio calcul/communication) permettent d'illustrer certaines propriétés de STEP. Les benchmarks de référence ont été choisis parmi ceux proposés par la NASA [10]. Le benchmark FT résout un système d'équations aux dérivées partielles (équation de la chaleur) en dimension trois par la transforméee de Fourier rapide. Alors que celui nommé MG résout l'équation de Poisson avec conditions aux limites périodiques par une méthode multigrille. Ils se distinguent par leurs modèles de communication. Le premier oblige tous les nœuds de calcul à communiquer entre eux alors que le second privilégie les communications entre certains noeuds seulement. Pour ces deux benchmarks, une exécution MPI existe, en plus des exécutions SER et OpenMP. Les deux autres benchmarks, respectivement MD et Matmul, sont réalisés à partir des programmes `md.f` et `matmul.f`. Le premier des deux est un programme de simulation de la dynamique moléculaire disponible dans une version OpenMP [11] du code source. Ce programme a été choisi pour l'importance de son rapport calcul/communication. Le second est l'implémentation "classique" du produit de matrices carrées. Ce programme a été choisi pour la possibilité qu'il offre de faire varier le rapport calcul/communication.

La plate-forme de test.

La plateforme utilisable pour réaliser les tests est constituée de quatorze nœuds, en réseau (20 Go/s). Chaque nœud est constitué de quatre quadricœurs (Xeon-1.6 GHz) ; soit seize cœurs par nœud. Pour les exécutions en mémoire distribuée (MPI, KMP ou STEP), un seul cœur par nœud est utilisé ; les autres cœurs restant inactifs. Pour les exécutions en mémoire partagée (OMP), le programme est exécuté sur les différents cœurs d'un même nœud. L'optimisation de l'implémentation du code pour l'exécution MPI des benchmarks NAS (MG et FT) oblige à lancer l'exécution de ces programmes sur une plateforme constituée d'un nombre de cœurs égal à une puissance de deux. Compte tenu du nombre de nœuds disponibles, les tests ont été effectués sur deux, quatre ou huit nœuds.

Pour ces deux benchmarks, l'exécution KMP ne fournit pas de résultats satisfaisants : dead lock pour un des programmes et résultat erroné pour l'autre. Cette exécution n'est, pour cette raison, pas prise en compte dans la comparaison. Pour certains benchmarks, lorsque l'exécution KMP fournit des résultats, d'importants écarts types sur les mesures de temps ont été constatés, vraisemblablement dus à des effets de cache au niveau du fonctionnement du mécanisme de la DSM.

L'évaluation des performances des différentes exécutions de chaque benchmark repose sur la moyenne de dix mesures du temps écoulé durant l'exécution de chaque programme.

Un examen plus détaillé du code source généré a permis d'établir que l'analyse de régions de tableaux

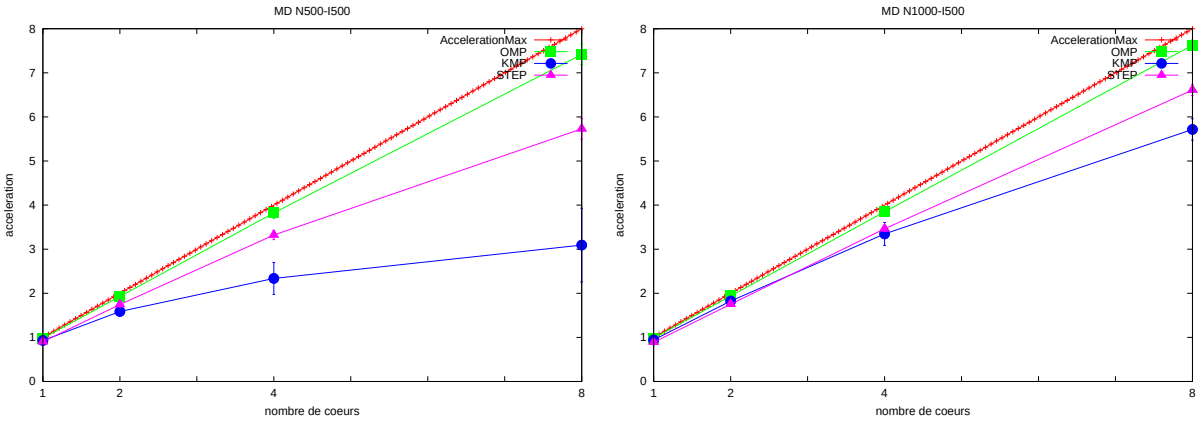

FIG. 3 – Accélération (MD)

sur ces benchmarks optimise la taille des messages transmis. Les régions de tableaux modifiées sont déterminées de manière exacte pour tous ces benchmarks. Il n'y a donc pas d'entrelacement de zones de tableaux qui engendrerait, lors de l'exécution, le traitement coûteux de reconstruction par différenciation. Cette constatation renforce le parti pris dans STEP d'analyser finement la dépendance des données ; déjà motivé par la volonté de minimiser la taille des messages échangés.

La figure FIG.3 montre que STEP est très efficace pour le benchmark MD qui est caractérisé par l'importance de son rapport calcul/communication.

L'expression du parallélisme sous-jacente au code source des exécutions MPI et OMP du benchmark FT est sensiblement différente ; contrairement au code source des exécutions MPI et OMP du benchmark MG. Pour ces deux applications l'exécution STEP requiert plus de temps que l'exécution SER. L'outil de profiling Vampir [12] a permis de constater la faiblesse du rapport calcul/communication pour l'exécution STEP des benchmarks MG et FT. Ce mauvais rapport calcul/communication est inhérent à l'algorithme implémenté pour le benchmark FT. En revanche, ce rapport peut être augmenté (en réduisant le volume global des communications) pour le benchmark MG, grâce à une amélioration envisagée de STEP qui est décrite ci-dessous.

Ainsi, avec le modèle actuel, une donnée peut être qualifiée $SEND$ alors qu'elle n'est pas nécessairement lue immédiatement par la section de code qui suit l'échange. Or seule la dernière écriture d'une donnée dans une section parallèle précédant la lecture de cette donnée justifie sa présence dans le message. Autrement dit, avec le modèle d'exécution actuel, une région de tableau peut être incluse inutilement de nombreuses fois dans des messages. Pour pallier ce problème, il faut envisager d'utiliser un autre modèle d'exécution qui relierait de manière plus forte les sections parallèles de code productrices de données ($OUT(T) \cap WRITE(T)$) de celles consommatrices de données. La mise en place de cet autre modèle nécessite une analyse plus complète des régions de tableaux. Car pour déterminer si une section de code consomme un tableau T, il faut calculer

$$RECV(T) = IN(T) \cap READ(T).$$

Une section de code est respectivement productrice et consommatrice des régions $SEND$ et $RECV$ qui lui sont associées. L'utilisation d'un tel modèle présente en outre l'avantage de permettre la désynchronisation de l'actualisation de la mémoire des différents processus et d'autoriser le recouvrement calcul/communication ; en permettant de différer l'échange de données non immédiatement utilisées.

Pour étudier l'incidence du ratio calcul/communication sur l'efficacité de STEP le programme matmul.f a été exécuté avec différentes tailles de matrices. Le rapport calcul/communication augmente avec la taille de la matrice. On constate sur la figure FIG.4 que la croissance de l'accélération en fonction du ratio calcul/communication est plus grande pour les exécutions STEP et KMP que pour l'exécution OMP. Cela confirme l'intérêt croissant avec le rapport calcul/communication d'une exécution sur plateforme à mémoire distribuée. Par ailleurs, on constate que l'accélération obtenue avec STEP est meilleure que celle obtenue avec KMP pour le plus petit ratio calcul/communication (N=1000). Le système à mémoire partagée distribuée sur lequel s'appuie Cluster OpenMP trouve sans doute là une limite. En effet, pour le produit de petites matrices, le mécanisme de pagination requiert un nombre restreint de pages mémoire pour stocker les tableaux. Aussi, la faiblesse de l'accélération pour l'exécution KMP peut s'expliquer par une augmentation relative de la concurrence des accès (par des processus différents) à ces pages, quand le nombre de processus augmente.

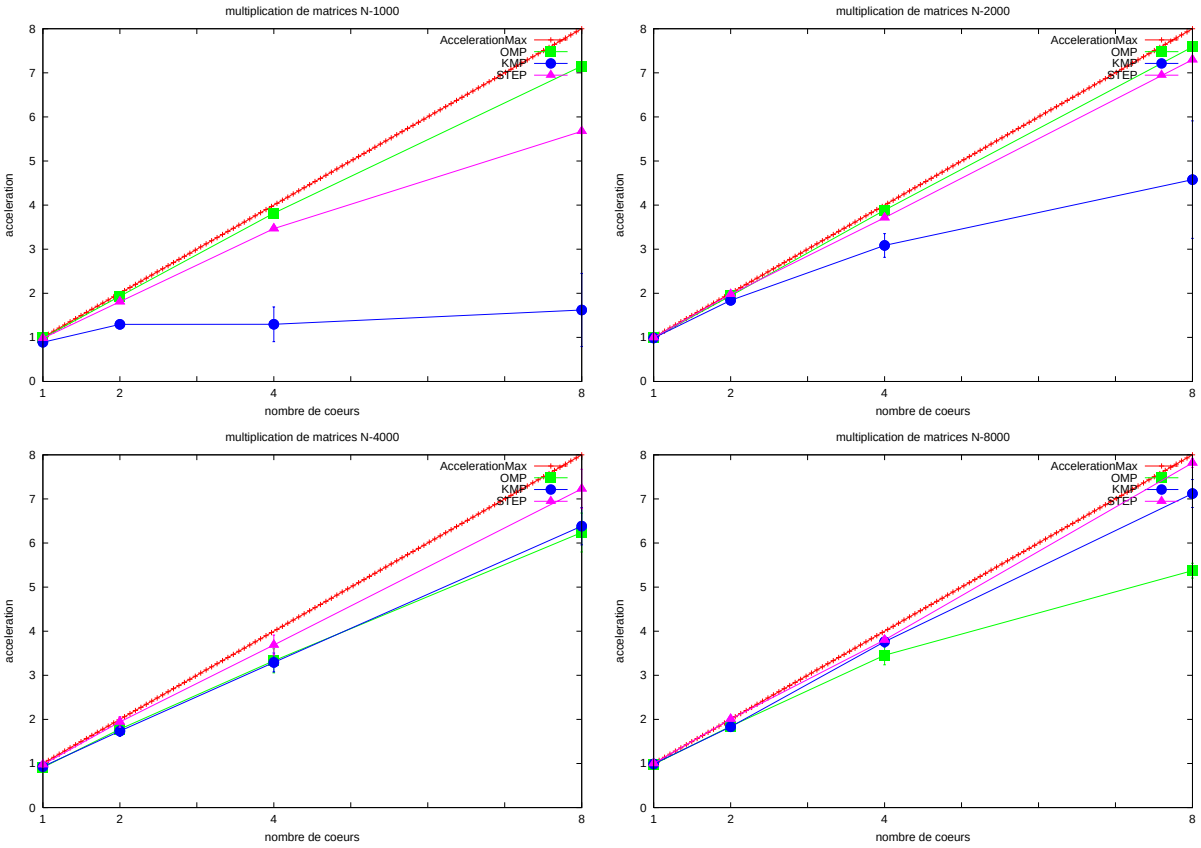

FIG. 4 – Accélération (Matmul)

Ces tests confirment expérimentalement que STEP est adapté aux applications à fort ratio calcul/communication. Bien entendu, les performances obtenues avec STEP ne peuvent soutenir la comparaison avec celles obtenues à partir d'un code produit manuellement par un expert en programmation MPI. Mais on constate que l'approche adoptée avec le développement de STEP conduit à la production d'un code parallèle qui, sur certaines applications, fournit une accélération intéressante par rapport à celle des exécutions OMP ou KMP.

4. Conclusion et perspectives

L'outil STEP de transformation source-à-source qui a été présenté dans cet article s'appuie sur PIPS pour générer un code source MPI à partir d'un code source contenant des directives OpenMP. La qualité des analyses interprocédurales de régions effectuées par PIPS permet à STEP de réduire le volume de communications nécessaires et le risque d'entrelacement responsable de coûteuses opérations de synchronisation. Grâce à la richesse de ses analyses, PIPS a permis un développement rapide du prototype courant de STEP. Nous avons présenté dans cet article le résultat des premières expérimentations sur des applications représentatives du calcul scientifique, dont certaines sont des benchmarks de référence. Ces expérimentations montrent la pertinence de l'approche adoptée pour des applications présentant un rapport calcul/communication suffisant et ouvrent des perspectives d'amélioration.

En particulier, le modèle d'exécution mis en œuvre par cette version de STEP contraint à une actualisation coûteuse des données pour chaque processus et pénalise par conséquent les applications nécessitant un volume important de données. C'est aussi pour cette raison que l'on n'obtient une accélération intéressante que lorsque le rapport calcul/communication est vraiment favorable ; pour compenser le surcoût de communication inhérent à un tel modèle d'exécution. Afin de pallier ce problème, un développement futur de STEP pourrait être de remplacer ce modèle d'exécution par un modèle plus élaboré de type "producteur-consommateur" pour lequel une analyse plus complète des dépendances optimiserait le volume de données communiquées.

Par ailleurs, un objectif à court terme de STEP est d'intégrer dans la transformation qu'il opère la

possibilité de prendre en compte du parallélisme multiniveau. Pour cela il faut envisager de produire automatiquement un code source hybride MPI/OpenMP à partir d'un code source contenant des directives OpenMP. La parallélisation gros grain, adaptée aux architectures à mémoire distribuée, serait implémentée par appel aux primitives de passage de messages (MPI) alors que la parallélisation grain fin, adaptée aux architectures à mémoire partagée, serait exprimée à l'aide de directives OpenMP. Le code parallèle ainsi produit devrait être adapté aux grappes de multi-cœurs. Les analyses fournies par PIPS permettent de calculer le ratio calcul/communication de chaque section parallèle. Ce ratio est déterminant pour décider du type d'exécution, à mémoire distribuée ou bien partagée, le mieux adapté à une section parallèle.
En facilitant la production automatique d'un code parallèle adapté aux architectures à mémoire distribuée, STEP est utilisable par le concepteur de l'application. Aussi, un autre objectif à plus long terme est de tirer un meilleur parti de la connaissance qu'a le concepteur de son application, en lui permettant de guider STEP dans la prise de certaines décisions, pour améliorer la qualité du code source produit.

5. Remerciements

Le travail réalisé pour produire cet article a été partiellement financé par le projet Européen ITEA2 ParMA (Parallel programming for Multicore Architectures) [1].

Bibliographie

1. Daniel Millot, Alain Muller, Christian Parrot, and Frédérique Silber-Chaussumier. STEP : a distributed OpenMP for coarse-grain parallelism tool. In *Proc. International Workshop on OpenMP '08, OpenMP in a New Era of Parallelism, Purdue University, USA, IWOMP*, 2008.
2. M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP Compiler for an SMP Cluster. In *Proceedings of First European Workshop on OpenMP (EWOMP)*, 1999.
3. Ayon Basumallik, Seung-Jai Min, and Rudolf Eigenmann. Towards OpenMP Execution on Software Distributed Shared Memory Systems. In *International Workshop on OpenMP : Experiences and Implementations, WOMPEI'2002*, 2002.
4. Rudolf Eigenmann, Jay Hoeflinger, Robert H. Kuhn, David Padua, Ayon Basumallik, Seung-Jai Min, and Jiajing Zhu. Is OpenMP for Grids. In *NSF Next Generation Systems Program Workshop held in conjunction with IPDPS*, 2002.
5. Yang-Suk Kee, Jin-Soo Kim, and Soonhoi Ha. ParADE : An OpenMP Programming Environment for SMP Cluster Systems. In *Conference on High Performance Networking and Computing*, 2003.
6. Taisuke Boku, Mitsuhisa Sato, Masazumi Matsubara, and Daisuke Takahashi. OpenMPI - OpenMP like tool for easy programming in MPI. In *Sixth European Workshop on OpenMP*, 2004.
7. Ayon Basumallik, Seung-Jai Min, and Rudolph Eigenmann. Programming Distributed Memory Systems Using OpenMP. In *12th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2007.
8. Corinne Ancourt, Béatrice Apvrille, Fabien Coelho, François Irigoin, Pierre Jouvelot, and Ronan Keryell. PIPS — A Workbench for Interprocedural Program Analyses and Parallelization. In *Meeting on data parallel languages and compilers for portable parallel computing*, 1994.
9. Jay Hoeflinger. Extending OpenMP to Clusters. Technical report, Intel Corporation, 2006.
10. NASA. NAS parallel benchmarks. http ://www.nas.nasa.gov/Resources/Software/npb.html.
11. Bill Magro. OpenMP samples. Kuck and Associates Inc.(KAI), http ://www.openmp.org/samples/md.f.
12. Vampir. http ://www.vampir-ng.com.

[1] http ://www.parma-itea2.org/