Optimisation et parallélisme avec OpenMP

L'approche *grain fin*

Martin.Jucker@epfl.ch, EPFL-SB-CRPP,
Vincent.Keller@epfl.ch, EPFL-STI-ISE-LIN,
Gonçalo.Pena@epfl.ch, EPFL-SB-IACS-CMCS et
Riccardo.Puragliesi@epfl.ch, Paul Scherrer Institut (PSI)




Résumé

Cet article présente le paradigme de programmation OpenMP pour les machines à mémoire partagée et propose une méthodologie d'*OpenMP-isation* des applications scientifiques sur les machines SMP *(Symetric Multi-Processor)* . Dans la première section, les auteurs présentent en quelques mots le contexte d'utilisation d'OpenMP ainsi qu'un petit historique. La deuxième section présente OpenMP dans les grandes lignes. Une méthodologie d'utilisation des machines à mémoire partagée avec OpenMP est présentée dans la troisième section illustrée par un *benchmarking* sur le simple exemple de la multiplication matrice – matrice. Pour finir dans les cinquième et sixième sections, les auteurs présentent l'optimisation et la parallélisation avec OpenMP de deux applications réelles: l'application de physique des plasmas TERPSICHORE, ainsi qu'un solveur de Helmholtz 3D, accompagnés de résultats obtenus sur la nouvelle architecture multicœur d'Intel: le Xeon 51XX Woodcrest. Notons enfin que les travaux présentés proviennent en grande partie des travaux pratiques du cours postgrade *High Performance Computing Methods* de Ralf Gruber.

Introduction

Il fut un temps, pas si lointain, où l'utilisation de machines à mémoire partagée SMP n'était pas ouverte à tout un chacun: leur complexité matérielle permettant une expression du parallélisme plus simple que les machines à mémoire distribuée avait un coût élevé. À cela il convient d'ajouter que toute l'énergie et le financement de la recherche électronique ont été investis dans la bataille de fréquence sur des processeurs mono-cœur que se sont livrée les principaux fondeurs de microprocesseurs.

En novembre 1995, Intel annonçait le **Pentium Pro**, une révolution dans l'évolution de son processeur phare capable d'être monté sur une carte mère bi ou quadri processeurs. Mais en partie à cause de la découverte du désormais célèbre *flag erratum* (un *bug* dans la conversion entier-float causant une erreur *overflow*), le *chip* n'a pas connu le succès qu'il aurait pu mériter, à l'exception de celui d'avoir été le précurseur de toute la série qui a suivi [1]. Aujourd'hui la guerre des GHz est terminée et on assiste à un revirement de stratégie des principaux fondeurs avec l'apparition des architectures multi-processeurs à multi-cœurs, doublées de compilateurs qui s'améliorent de version en version, autorisant la programmation à mémoire partagée à un coût relativement faible. OpenMP est l'un des paradigmes de programmation parallèle adapté aux machines de type SMP, c'est l'objet de cet article.

Ces briques multi-processeurs multi-cœurs sont les nœuds des clusters de type *Beowulf* actuels (clusters construits sur la base d'éléments *grand public*). Le 6ème ordinateur le plus puissant de la planète – et le premier *Beowulf* derrière les spécifiques BlueGene d'IBM et le RedStorm de Cray Inc.– est un super-ordinateur de ce type [2]. Ainsi la programmation à plusieurs niveaux de parallélisme est maintenant à portée de bourse de tout un chacun; on utilise le paradigme de programmation parallèle MPI entre les nœuds et OpenMP à l'intérieur des nœuds.

OpenMP, késako ?

Le standard OpenMP *(Open Multi-Processing* [3]) est une interface de programmation (API) et a été publié pour la première fois en 1997 pour le langage Fortran. Aujourd'hui, OpenMP est aussi disponible pour les langages de programmation C/C++. OpenMP consiste en un ensemble de directives de compilation, d'une bibliothèque de fonctions ainsi qu'un ensemble de variables d'environnement. Les performances obtenues avec OpenMP dépendent donc de deux facteurs: d'une part d'un choix de partage de données judicieux de la part du programmeur, d'autre part de la qualité du compilateur (interprétation des directives de compilation). Tous les exemples décrits dans cet article ont été compilés avec la dernière version du compilateur d'Intel (9.1e) sur un nœud multi-cœurs du cluster Pleiades2 [4] (Intel Xeon 5150 Woodcrest, 2.66 GHz, 8 GB RAM), sauf lorsque l'auteur le précise explicitement.

Une machine à mémoire partagée se programme toujours de la même façon: plusieurs *threads* résidant sur des *process elements* différents possèdent un espace propre et accèdent à un espace de données partagé; entraînant les problèmes traditionnels d'aléas de données. L'utilisation d'OpenMP ne change en rien cet état de fait et demande toujours du programmeur une connaissance parfaite du problème qu'il veut résoudre en terme algorithmique ainsi que des bases de l'architecture cible. OpenMP peut améliorer sensiblement la performance d'une application, mais peut malheureusement aussi, mal utilisé, la fusiller. C'est ce que nous allons notamment voir dans la suite de cet article.

Comment utiliser OpenMP ?

D'autres avant nous ont tenté d'établir une méthodologie d'*OpenMP-isation* (voir par exemple l'excellent cours de l'IDRIS [13]). Il nous semblait toutefois que la présentation des résultats en terme de performances (orienté Calcul à Hautes Performances) en gardant le même exemple du début à la fin, passant en revue les améliorations, les pièges à éviter, etc. avait encore un sens.

Il existe deux cas d'utilisation: sur un code existant, ou *from scratch*. Dans le premier, on part d'un code séquentiel *mono-thread* qu'on parallélise au niveau des boucles en suivant une méthodologie que nous allons voir plus loin; c'est l'approche à grain fin *(fine grain)*. Dans le second, on part dès le début en ayant en tête que la machine cible sera à mémoire partagée; c'est l'approche à grain fort *(coarse grain)*. Il est utile de rappeler ici que les drapeaux de compilation *(compilation flags)* disponibles sont nombreux et variés [9], nous en resterons à l'optimisation `-O0` (sans optimisation), `-O1`, `-O2` (les deux sont similaires sur les architectures IA-32 et EM64T et correspondent à une optimisation avec *inlining*, propagation de constantes, allocation globale des registres, propagation des attributs des routines, analyse des adresses des variables, élimination des sections mortes, suppression des variables non référencées, optimisation inter routines au niveau d'un fichier, récursion de queue) et `-O3` (optimisation maximale: `-O2` avec en plus une analyse plus agressive des dépendances, modification des boucles, pré-extraction des données) ainsi qu'à la vectorisation des boucles via les flags *–xX* où X dépend de l'architecture cible (*–xT* sur un Woodcrest, *–xW* sur un Xeon, *–xB* sur un PentiumM, etc.).

Afin de pouvoir mesurer les peformances, les effets des optimisations proposées, nous utiliserons le temps et le rapport MFlops/s comme métriques. Ces deux valeurs peuvent se mesurer grâce à des outils spécifiques non intrusifs (Intel VTune [10] ou la simple routine `gettimeofday()` précise à la micro-seconde près) ou intrusifs (Parallel API [5]). Dans cet exemple, on mesurera le temps à la microseconde près et on calculera le rapport MFlops/s avec la connaissance du nombre d'opérations. Dans la suite de cet article nous allons prendre l'exemple simple de la multiplication matricielle (les éléments des matrices sont des nombres à virgule flottante double précision, 64 bits):

$$C = A * B$$

La taille des matrices choisie est de 1200 x 1200 (division entière par 16 possible pour l'exemple). L'exemple sera implémenté en Fortran 77[1] et compilé à l'aide du compilateur Intel Fortran 9.1e sur l'un des nœuds de Pleiades2+ (la nouvelle extension multi-processeurs multicœurs du cluster Pleiades). L'exemple est implémenté suivant le listing 1: le listing 1 présente la séquence $k - j - i$, il suffit d'inverser les indices pour obtenir les résultats présentés dans cet article.

Tous les résultats sont donnés en forçant 4 *threads* sur le nœud, soit un thread par cœur

(`export OMP-NUM-THREADS = 4`).

[1] Ce choix se justifie par une visibilité plus claire de l'impact du choix de l'ordre des indices. Une analogie avec le langage C/C++ est aussi plus aisée. La performance obtenue avec la syntaxe de base Fortran 90/95 sera présentée en fin de section

Première étape: compilation sans optimisation

La compilation sans optimisation (flag `-O0`) permet au programmeur de pointer les problèmes et surtout d'observer les améliorations qu'il peut faire lui-même à la main, notamment l'ordre des indices dans les boucles. Le listing 2 présente les résultats pour cette première étape.

```
      n = 1200
c Initialisation de A, B et C

      call initialise(n*n,A,1.0d0)
      call initialise(n*n,B,1.0d0)
      call initialise(n*n,C,1.0d0)

c     Depart du chrono
      ti = second()
c   Multiplication C=A*B
c   en suivant la sequence kji
    do k=1,n
      do j=1,n
        do i=1,n
          C(j,k)=C(j,k)+A(j,i)*B(i,k)
        enddo
      enddo
    enddo
c   Arret du chrono
    ti = second ()-ti
    mflops_kji = 2.0*(n/100.)**3/ti
```

listing 1 – Multiplication matrice*matrice

	t	Mflop/s
kji	0.3849E+02	89.78
kij	0.2339E+02	147.7
ikj	0.2367E+02	146.0
jki	0.2927E+02	118.1
ijk	0.4767E+02	72.50
jik	0.4761E+02	72.59
DGEMM	0.1319E+00	29210.0

listing 2 – Résultats du programme MMIJK avec le flag de compilation-O0

Le listing 2 démontre que l'ordre des indices dans une boucle a une importance capitale dans la performance d'un code. Les performances peuvent doubler. Ici le pire cas est de 89 MFlops/s, le meilleur à 147 MFlops/s. Cela est dû au fait que les données restent en cache au lieu d'être à chaque fois ramenées de la mémoire haute aux registres. DGEMM, la routine BLAS, élément de LAPACK[12] et écrite en assembleur, provenant de la bibliothèque MKL sur les machines Intel (ACML sur les machines à base d'AMD Opteron) a été mise comme référence. La performance de 29.21 GFlops/s est obtenue grâce aux instructions SSE2 et SSE3 (utilisant un chip spécifique permettant 4 résultats par temps de cycle contre 2 normalement). Les plus perspicaces auront noté que 29.21 GFlops/s c'est 8 GFlops/s de plus que la performance maximale du nœud ($R\infty$ = 21.28 GFlops/s)[2].

Seconde étape: unroll manuel

Attaquons le déroulement de boucles (*unrolling*) manuel de la meilleure des boucles $k-i-j$, celle dont les indices donnent la meilleure performance; la meilleure boucle est celle qui permet le meilleur agencement des données en mémoire. Plus précisément, le principe du déroulement de boucle exige de laisser en cache des grandeurs qui peuvent être utilisées plus d'une fois. Ainsi, on augmente le V_a [6] (rapport entre le nombre d'opérations et le nombre de LOAD's en mémoire haute). Dans l'exemple qui suit, on déroule sur 8 niveaux, cela signifie (voir listing 7 sans les directives OpenMP) que les grandeurs *b* et *c* restent en cache (du fait de leur grandeur), *a* est utilisé 8 fois, V_a = 16 au lieu de V_a = 2.

La compilation se fait à l'aide du *flag* de compilation `-O1`. Le déroulement de boucle permet d'effectuer plus d'opérations avec les mêmes données en cache. Le programmeur doit par contre toujours être clair quant aux indices qu'il parcourt.

	t	Mflop/s
kji	0.3889E+01	888.6
kji2	0.2133E+01	1620.0
kji4	0.1596E+01	2166.0
kji8	0.1584E+01	2182.0
kji16	0.2077E+01	1664.0

listing 3 – Résultats du programme MMIJK avec le flag de compilation -O1 et un unroll manuel

Le listing 3 démontre qu'un *unroll* manuel améliore les performances de la meilleure des boucles. Ainsi on gagne encore 1.3 GFlops/s sur le meilleur arrangement des indices.

Troisième étape: vectorisation des boucles

La documentation d'Intel [9] parle de *innermost loops parallelization* c'est-à-dire de parallélisation sur la boucle intérieure alors que la parallélisation *multi-threads* s'appelle *outer-most loops parallelization* (voir sections 3.5.1 et 3.5.2). L'option de compilation est *–xX* où *X* correspond à l'architecture cible. Cette optimisation permet d'utiliser au mieux les pipelines des processeurs en faisant plus d'opérations avec les mêmes opérandes à chaque itération des boucles. On n'utilisera cette option qu'après avoir effectué l'*unrolling à la main* (étape 2). La performance maximale (2.18 GFlops/s) est toujours celle obtenue grâce au *unroll* à 8 niveaux en regard de la moins bonne (ordre $j-i-k$), de 354 MFlops/s, c'est 1.82 GFlops/s de plus donc un facteur de 5.

Notez que le compilateur est capable de faire un *unroll* sur les boucles *j–k–i* et *j–i–k* mais n'arrive jamais à la performance obtenue grâce à un déroulement manuel. Les performances obtenues en commençant par l'indice *i* sont de l'ordre de 1 GFlops/s (soit moins de 40% de la performance avec un déroulement manuel), celles obtenues en commençant par l'indice *k* tournent autour de 600 MFlops/s (soit moins de 25% de la performance avec un déroulement manuel).

Finalement, qu'on compile avec le *flag* le plus (`-O3`) ou le moins (`-O1`) agressif, seules les boucles commençant par l'indice *j* gagnent en performance. Ceci se vérifie simplement entre les listings 5 et 4. Cela démontre que le compilateur est capable de reconnaître certaines constructions et de les optimiser, malheureusement pas toutes.

[2] $R\infty$ = freq * NbrOperations per cycle * Nbrprocess elements = $2.66 * 10^9 * 2 * 4 = 21.28$ GFlops/s

	t	Mflop/s
kji	0.3543E+01	975.6
kij	0.3543E+01	975.4
ikj	0.5978E+01	578.2
jki	0.9751E+01	354.4
ijk	0.5966E+01	579.3
jik	0.2056E+02	168.1

listing 4 – Résultats du programme MMIJK avec les flags de compilation -O1 -xT sans unroll

	t	Mflop/s
kji	0.3549E+01	973.7
kij	0.3548E+01	974.2
ikj	0.5988E+01	577.1
jki	0.1656E+01	2086.0
ijk	0.5987E+01	577.3
jik	0.1659E+01	2084.0

listing 5 – Résultats du programme MMIJK avec les flags de compilation -O3 -xT sans unroll manuel

Quatrième étape: vectorisation et optimisation

	t	Mflop/s
kji	0.1296E+01	2667.0
kji2	0.1628E+01	2123.0
kji4	0.1493E+01	2314.0
kji8	0.1567E+01	2205.0
kji16	0.2077E+01	1664.0

listing 6 – Résultats du programme MMIJK avec les flags de compilation -O3 -xT et unroll manuel

Le listing 6 présente le résultat après la compilation la plus agressive et la vectorisation. La meilleure performance obtenue est de 2.66 GFlops/s pour la boucle *k–i–j* ; ce qui représente 50 % de $R\infty$! Il est intéressant de voir ce que donnent les performances d'un code sans aide au compilateur. C'est ce qui est présenté au listing 5.

Cinquième étape: parallélisation

Une fois que le code est optimisé sur un processeur (ou un cœur), on peut commencer à paralléliser le code au niveau du nœud (puisque nous sommes sur une machine SMP). Deux possibilités s'offrent au programmeur: laisser le compilateur découvrir les sections qui peuvent être exécutées en parallèle de façon sûre *(safely parallelized sections),* c'est l'auto-parallélisation, ou alors choisir manuellement la meilleure façon de partager le travail entre chaque *thread,* c'est la **parallélisation avec OpenMP**. Nous allons démontrer que cette seconde façon est la meilleure, même pour le programme le plus simple, puisqu'on n'est jamais mieux servi que par soi-même.

Parallélisation manuelle avec OpenMP

Le listing 7 présente le code source pour la boucle *k–j–i* avec un *unroll* manuel à 8 niveaux. Les directives de programmation OpenMP sont placées à l'extérieur de la boucle à paralléliser. La boucle qui sera parallélisée est celle qui suit immédiatement la directive `!$OMP DO`. La directive `SCHEDULE(GUIDED, 1)` permet de choisir la façon dont seront agencées chacune des sections traitées en parallèle. Dans ce cas précis, nous avons choisi un ordonnancement GUIDED *i.e.* que chaque *thread* reçoit une quantité de travail balancé *(load balancing)* selon l'état de ce qui reste à faire. L'auteur propose au lecteur intéressé de consulter les spécifications du standard OpenMP [11] pour plus de détails.

```
  n = 1200
c Initialisation de A, B e t C

  call initialise (n*n,A,1.0d0)
  call initialise (n*n,B,1.0d0)
  call initialise (n*n,C,1.0d0)

c   Depart du chrono
ti = second ()

c Multiplication C = A * B
c en suivant la sequence kji et unroll de 8 niveaux

  !$OMP PARALLEL DEFAULT(SHARED) PRIVATE( i , j )
  !$OMP DO SCHEDULE(GUIDED, 1)
    do k=1,n , 8
      do i =1,n
    do j =1,n
          c(j,k)=c(j,k)+a(j,i)-b(i,k)
          c(j,k+1)=c(j,k+1)+a(j,i)-b(i,k+1)
          c(j,k+2)=c(j,k+2)+a(j,i)-b(i,k+2)
          c(j,k+3)=c(j,k+3)+a(j,i)-b(i,k+3)
          c(j,k+4)=c(j,k+4)+a(j,i)-b(i,k+4)
          c(j,k+5)=c(j,k+5)+a(j,i)-b(i,k+5)
          c(j,k+6)=c(j,k+6)+a(j,i)-b(i,k+6)
          c(j,k+7)=c(j,k+7)+a(j,i)-b(i,k+7)
        enddo
      enddo
    enddo
  !$OMP END DO
  !$OMP END PARALLEL

c Ar ret du chrono
  ti = second () - ti
  mflops_kji8 = 2 . 0*(n/100.)**3/ti
```

listing 7 – Multiplication matrice*matrice parallélisée avec OpenMP

Le listing 8 démontre que la parallélisation par OpenMP sur 4 *threads* suit un *speedup* parfaitement linéaire (voir la comparaison avec les listings 2 et 3). Pour certaines boucles (*i–k–j* par exemple), on observe un *speedup* superlinéaire[3].

Auto-parallélisation

Cette étape consiste à laisser le compilateur chercher (et trouver) les boucles qui peuvent s'exécuter de façon indépendante *(embarrassingly parallel)*. Pour cela, on invoque le compilateur à l'aide du *flag* de compilation `-parallel`. Cette opération se fait après l'optimisation `-O2` (ou `-O3`) A noter que si le programmeur a lui-même procédé à la parallélisation de ses boucles en utilisant OpenMP (comme décrit à la section

[3] Cette dernière remarque permet aussi à l'auteur de mettre en garde le lecteur ne jurant que par le *speedup*: si la version monoprocesseur affiche une performance médiocre, il est relativement aisé d'obtenir ce genre d'effet !

précédente), l'invocation de `-parallel -openmp` ne change pas le programme final: le compilateur parallélise d'abord les boucles OpenMP, puis autoparallélise le reste. L'autoparallélisation ne peut se faire qu'avec le *flag* d'optimisation `-O1` au minimum, la comparaison entre la parallélisation OpenMP et le niveau `-O0` est donc difficile. On remarque tout de même que le maximum que puisse faire le compilateur sans *unroll* manuel est 2.84 GFlops/s (voir listing 9) alors que si le programmeur a suivi les étapes proposées par l'auteur aux points précédents, notamment l'*unrolling* manuel, le compilateur arrive à des performances de l'ordre de 7.77 GFlops/s pour un *unrolling* manuel à 8 niveaux. L'autoparallélisation optimisée de la façon la plus agressive donne des résultats proches de ceux d'OpenMP optimisé pour certains indices de boucle. Voir le listing 11. Il est intéressant de voir que la suite d'index la plus utilisée *i–j–k* donne la performance la plus médiocre.

	t	Mflop/s
kji	0.9550E+01	361.9
kij	0.5862E+01	589.6
ikj	0.6714E+01	514.7
jki	0.7697E+01	449.0
ijk	0.1531E+02	225.7
jik	0.1396E+02	247.5
kji2	0.5589E+01	618.3
kji4	0.5504E+01	628.0
kji8	0.5800E+01	595.9
kji16	0.5602E+01	616.9

listing 8 – Résultats du programme MMIJK avec le flag de compilation -O0 -openmp

	t	Mflop/s
kji	0.2871E+01	1204.0
kij	0.1213E+01	2848.0
ikj	0.2409E+01	1435.0
jki	0.2471E+01	1398.0
ijk	0.6027E+01	573.5
jik	0.5439E+01	635.4

listing 9 – Résultats du programme MMIJK avec le flag de compilation -O1 -parallel sans unroll manuel

	t	Mflop/s
kji2	0.6550E+00	5276.0
kji4	0.5193E+00	6655.0
kji8	0.4448E+00	7769.0
kji16	0.5081E+00	6802.0

listing 10 – Résultats du programme MMIJK avec le flag de compilation -O1 -parallel avec unroll manuel

	t	Mflop/s
kji	0.1177E+01	2936.0
kij	0.1171E+01	2952.0
ikj	0.2449E+01	1411.0
jki	0.4249E+00	8134.0
ijk	0.3780E+01	914.4
jik	0.4249E+00	8133.0

listing 11 – Résultats du programme MMIJK avec le flag de compilation -O3 -xT -parallel sans unroll manuel

Sixième étape: optimisation du code parallèle

Que se passe-t-il maintenant si on force le compilateur à optimiser les boucles parallélisées avec OpenMP ?

Le listing 12 présente les résultats du code parallélisé avec OpenMP et le *flag* d'optimisation `-O3`. La performance maximale obtenue est de 7.73 GFlops/s pour la séquence *k–i–j* avec un *unrolling* de 8. Les résultats sont exactement les mêmes qu'avec le *flag* de compilation `-O2` !

Cela signifie que le compilateur n'est plus capable de faire mieux. Finalement, le listing 13 présente les résultats du code parallélisé avec OpenMP, le flag d'optimisation `-O3` et la vectorisation des boucles intérieures avec *–xT*. La performance maximale obtenue est de 8.04 GFlops/s pour la séquence *k–i–j* avec un unrolling de 8. C'est le meilleur résultat possible[4].

	t	Mflop/s
kji	0.2847E+01	1214.0
kij	0.1211E+01	2853.0
ikj	0.3092E+01	1118.0
jki	0.2470E+01	1399.0
ijk	0.8818E+01	391.9
jik	0.7624E+01	453.3
kji2	0.6555E+00	5273.0
kji4	0.5216E+00	6626.0
kji8	0.4468E+00	7736.0
kji16	0.5124E+00	6745.0

listing 12 – Résultats du programme MMIJK avec le flag de compilation -O3 -openmp

	t	Mflop/s
kji	0.1166E+01	2964.0
kij	0.1165E+01	2967.0
ikj	0.3190E+01	1083.0
jki	0.6315E+00	5473.0
ijk	0.6405E+01	539.6
jik	0.6076E+00	5688.0
kji2	0.6163E+00	5607.0
kji4	0.5256E+00	6576.0
kji8	0.4298E+00	8042.0
kji16	0.4658E+00	7420.0

listing 13 – Résultats du programme MMIJK avec le flag de compilation -O3 -openmp -xT

Septième étape: toutes voiles dehors

Pour finir, regardons ci-après ce que donne le résultat compilé avec `-O3-xT-parallel- openmp`. Le compilateur *unroll* parallélise les boucles selon OpenMP, puis auto-parallélise ce qui reste et finalement vectorise.

Le listing 14 présente les mêmes performances que celles obtenues avec le code parallélisé à la main avec OpenMP (listing 13). L'auto-parallélisation sur un code optimisé à la main n'améliore pas les performances même si le compilateur annonce (lors de la compilation) qu'il a réussi à auto-paralléliser des boucles partielles.

[4] Il s'agit aussi tout simplement de la performance maximale R1 du processeur vectoriel NEC SX-5 (R1 = 8 GFlops/s)

	t	Mflop/s
kji	0.1166E+01	2964.0
kij	0.1166E+01	2964.0
ikj	0.2981E+01	1159.0
jki	0.6556E+00	5272.0
ijk	0.6453E+01	535.5
jik	0.5951E+00	5807.0
kji2	0.6160E+00	5611.0
kji4	0.5263E+00	6566.0
kji8	0.4300E+00	8037.0
kji16	0.4658E+00	7419.0

listing 14 – Résultats du programme MMIJK avec le flag de compilation -O3 -xT -parallel-openmp

Et Fortran 90 alors ?

Afin d'être complet, donnons quelques performances pour qui utiliserait les nouvelles fonctionnalités offertes par Fortran 90. La principale nouveauté apparue avec Fortran 90 sur Fortran 77 était la gestion *simplifiée* des tableaux et l'apparition de nouvelles fonctions intrinsèques. Dans le cas de notre petit exemple, la multiplication matricielle se note tel que présenté au listing 15.

```
    n = 1200
! Initialisation de A, B e t C
    call initialise(n*n,A,1.0d0)
    call initialise(n*n,B,1.0d0)
    call initialise(n*n,C,1.0d0)

    ti = second()
    C = matmul(A,B)
    ti = second ( ) - t i
    mflops-rate = 2.0*(n/100.)**3/ti
```

listing 15 – Multiplication matricielle en Fortran 90

Il est évident que cette notation *à la Matlab* est bien plus lisible pour un humain, mais qu'en est-il des performances ? Elles sont catastrophiques:

	t	Mflop/s
F90	0.1096E+02	315.3
DGEMM	0.1203E+00	29120.0

listing 16 – Performance MATMULT en Fortran 90 avec flags -O3 -xT

On a laissé la performance de DGEMM de côté. Ce n'est pas moins d'un facteur 100 qui différencie les deux méthodes! Une OpenMP-isation devrait améliorer la performance (voir listing 17)

	t	Mflop/s
F90//	0.2905E+01	1190.0

listing 17 – Performance MATMULT en Fortran 90 avec flags -O3 -xT - openmp

Et Fortran 95 alors ?

Le standard Fortran 95 a introduit des nouvelles fonctionnalités *parallèles*. C'est le cas notamment du bloc FORALL. Dans le cas de la multiplication matricielle, cela n'a pas le moindre sens.

Moralité

Les résultats présentés ci-dessus démontrent – à l'aide d'un code source extrêmement simple – qu'en suivant une stratégie d'optimisation on peut arriver à des performances de l'ordre de 37 % de la performance maximale R_∞ du processeur considéré. Sans optimisation manuelle, en plaçant une totale confiance dans le compilateur, on obtient un résultat proche (*flags* de compilation `-parallel -xT -O3`) 8.1 GFlops/s au mieux dans le cas où l'utilisateur a aidé manuellement le compilateur (1.4 GFlops/s au pire !). Le compilateur est capable de reconnaître certaines boucles et de changer l'ordre des indices. Malheureusement pas partout.

Les performances de l'auto-parallélisation démontrent que le compilateur est capable de reconnaître certaines boucles et de les paralléliser exactement comme on le fait avec OpenMP. Pour certains ordres d'indice, le compilateur n'en est pas capable. Il est aussi à noter que suivant la complexité du calcul à l'intérieur de la boucle externe, le compilateur ne sera pas capable de l'auto-paralléliser. C'est ce que nous allons voir dans la dernière partie de cet article avec l'application de physique des plasmas TERPSICHORE d'abord, avec un solveur de Helmholtz 3D ensuite.

Cela montre surtout que si le programmeur a une connaissance exacte de l'algorithmique de son code, il peut alors parfaitement maîtriser la localisation de ses données.

Premier exemple: TERPSICHORE

TERPSICHORE est un code MHD développé au CRPP dès 1988 [7].

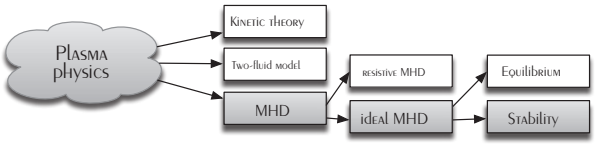

Fig. 1 – TERPSICHORE étudie la stabilité MHD idéale des appareils de fusion thermonucléaire

La physique de TERPSICHORE

TERPSICHORE est un code MHD idéal *(MagnetoHydroDynamics),* nommé ainsi en référence à la déesse grecque de la danse et de la poésie. Il recherche la stabilité des géométries tridimensionnelles à l'aide d'un équilibre provenant d'un autre code nommé VMEC [8]. L'équilibre est trouvé en résolvant le système non linéaire:

$$\nabla \mathbf{p} = \mathbf{j} \times \mathbf{B}, \qquad \mathbf{j} = \nabla \times \mathbf{B}, \qquad \nabla \cdot \mathbf{B} = 0,$$

avec **p**, **j**, **B**: la pression, la densité de courant et le champ magnétique respectivement. Ensuite, pour déterminer la stabilité du plasma, les équations linéarisées MHD sont traitées par TERPSICHORE. Elles peuvent être écrites sous la forme variationnelle :

$$\delta W_p + \delta W_v - \omega^2 \delta W_k = 0,$$

avec δW_p l'énergie potentielle du plasma, δW_v l'énergie magnétique du vide autour du plasma, δW_k l'énergie cinétique et ω^2 la valeur propre du système. Si $\omega^2 < 0$ alors la configuration est instable. La fig. 2 montre un exemple d'équilibre traité avec TERPSICHORE.

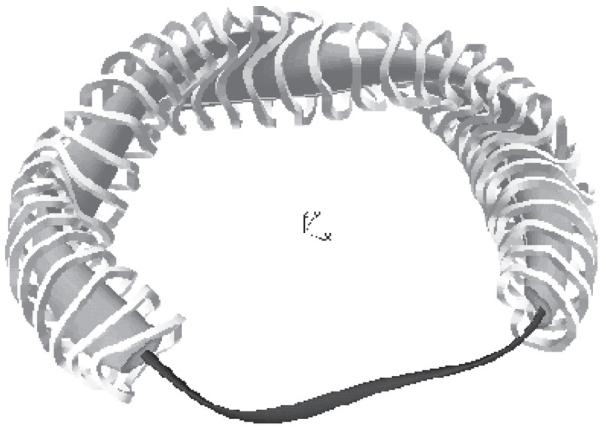

Fig. 2 – un exemple de configuration: le W7-X à Greifswald, Allemagne.

Une des spécialités de TERPSICHORE est son système de coordonnées. La MHD idéale prédit que les lignes de champ magnétique sont sur des surfaces à pression constante, ce qu'on appelle les surfaces de flux, et que ces lignes de champ magnétique sont *gelées* dans le plasma, c'est-à-dire que si les particules bougent, les lignes de champ magnétique bougent avec elles. Avec cette connaissance, il est possible d'introduire des coordonnées du flux magnétique, ou **coordonnées de Boozer**. Ces coordonnées de Boozer utilisent une étiquette du flux $s \in [0, 1]$ comme variable radiale et les variables angulaires polaires et toroïdales θ, ψ de façon à ce que les lignes de champ deviennent droites dans le système de coordonnées choisi. Si l'on veut ensuite considérer une particule dans une géométrie donnée, on sait d'avance qu'elle ne va pas bouger le long de la variable radiale s du fait de sa condition de gel. Il s'agit là de l'un des nombreux avantages de ce nouveau système de coordonnées.

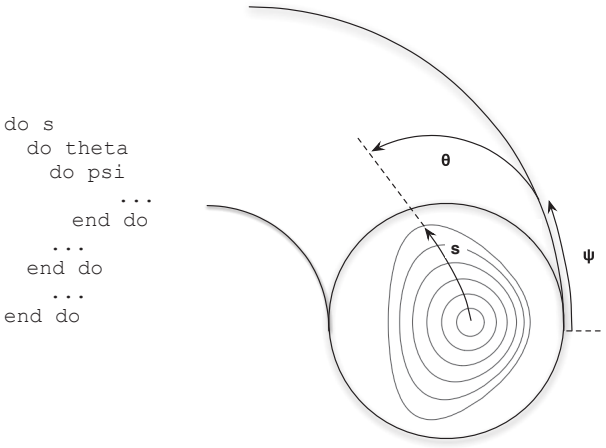

Fig. 3 – structure schématique

Le code

L'utilisation de coordonnées magnétiques permet de calculer chaque surface de flux (c'est-à-dire une position radiale) de façon indépendante. Ainsi, la structure générale est très similaire à un code dans lequel il y aurait des boucles sur chaque variable radiale et ce, sur chaque sous-routine. Schématiquement, cette structure est présentée dans la fig. 3.

Organisation du code

La structure de TERPSICHORE est organisée en six principales routines: `eqinvm`, `veqrec`, `mtaskb`, `stabin`, `mtaska` et `mtasks-ap`. Les deux premières établissent l'interface avec la sortie du code VMEC et reconstruisent les variables d'équilibre MHD. La routine `mtaskb` est responsable principalement de la mise en correspondance du système de coordonnées de VMEC et du système de coordonnées de Boozer ainsi que du calcul de la métrique. La construction des matrices de stabilité est effectuée dans `stabin`. Le solveur PAMERA calcule les valeurs propres ω^2 et les modes. Ce solveur prend peu de temps (ne domine pas) pour le petit cas test considéré.

Performance de TERPSICHORE

Le code a été optimisé dans le passé pour les machines vectorielles, spécialement pour le NEC SX-5 dans sa dernière version et donc, la structure montrée à la fig. 3 est plus précisément donnée par:

```
do s
  do theta
    do psi
      variable(s,theta,psi)
    end do
  end do
end do
```

La performance sur la machine NEC SX-5 pour le *run* de test choisi *(Large Helical Device)* est de 30s. Cette performance devient l'objectif à atteindre en optimisant le code sur l'architecture Xeon 5100 (Woodcrest) de Pleiades2.

Le tab. 1 présente le *profile* de l'application non optimisée sur une architecture XEON. Quatre routines attirent immédiatement l'attention : `lgikvm`, `lamcal`, `metric` et `fourin`. La suite de cette section concernera l'optimisation de ces quatre routines principalement.

main routine	subroutine	subroutine	time
eqinvm			0.12
veqrec	cospol		0.64
	lgikvm		35.45
	mtaskl	lamcal	24.78
mtaskb	vmtobo		24.16
	extint		5.73
	metric		15.73
	bophys		4.96
stabin			0.14
mtaska	fourin		10.01
Total			132.97

Tab. 1 – TERPSICHORE: profile obtenu sur un Xeon 64 HT (Pleiades2) de l'application non optimisée

Optimisation

Performance sur le Xeon 5160 (WoodCrest)

Tout d'abord, nous avons voulu savoir où commencer notre optimisation. La fig. 4 ainsi que le tab. 2 présentent la

performance du code obtenue sur un Xeon 5160 *(Woodcrest)*. On voit facilement que le compilateur utilisé (`ifort` version 9.1e) peut améliorer les résultats d'environ 50%, mais, comme nous l'avons vu plus haut, il est recommandé d'aider le compilateur manuellement plutôt que d'accorder une trop grande confiance en lui. Dans tous les cas, la nature du code laisse présager d'excellentes optimisations possibles.

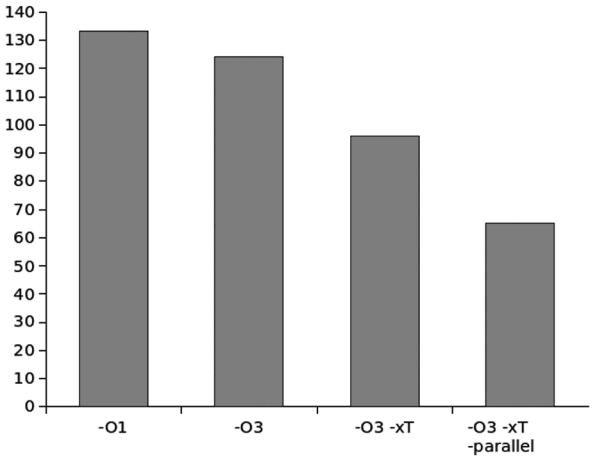

Fig. 4 – performance sur un nœud Xeon 5160 (WoodCrest) avant l'optimisation

`-O1`	133s
`-O3`	124s
`-O3-xT`	96s
`-O3 -xT -parallel`	65s

Tab. 2 – TERPSICHORE: performance sur un nœud Xeon 5160 (WoodCrest) avant l'optimisation

Travailler sur l'optimisation -O1

Comme déjà vu plus haut, il y a quelques changements qui doivent être apportés manuellement à ce niveau d'optimisation. Alors que le code a été déjà hautement optimisé, mais pour une machine vectorielle, le gain principal que l'on peut attendre vient d'une simple inversion des indices des boucles DO. La fig. 5 présente un exemple d'une telle inversion. D'autres possibilités ont été utilisées comme la division des grandes boucles, le déroulement dans des plus petites boucles, etc. mais l'inversion des indices reste le plus efficace. Les résultats après cette première optimisation sont présentés à la fig. 6 et au tab. 3.

Notons tout d'abord que toutes les exécutions montrent une meilleure performance que celle obtenue sans optimisation (valeur initiale), c'est-à-dire que l'optimisation à la main avec le *flag* de compilation (`-O1` par exemple) était meilleure d'environ 15% que ce dont est capable le compilateur (avec `-O3 -xT`). Deuxièmement, on observe qu'il n'y a quasiment pas de différence de performance sur la version optimisée manuellement en changeant de *flag* de compilation (excepté avec `–parallel`). Ceci nous encourage donc, comme précisé dans la première partie de cet article, à paralléliser le code manuellement pour voir si on peut encore aider le compilateur et obtenir les meilleures performances possibles (meilleures que ce que pourrait faire le compilateur seul).

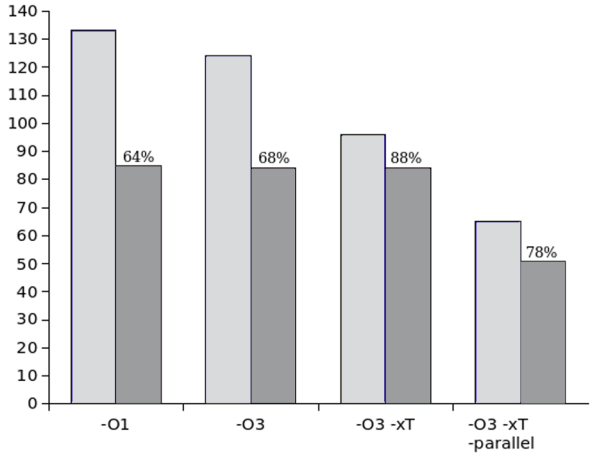

Fig. 6 – la performance après optimisation sur un processeur au niveau `-O1`

		opt
`-O1`	133s	85s
`-O3`	124s	84s
`-O3 -xT`	96s	84s
`-O3 -xT -parallel`	65s	51s

Tab. 3 – TERPSICHORE: performance après optimisation

Stratégie de parallélisation

Le code sera parallélisé en utilisant les directives OpenMP sur le code optimisé sur un processeur.

Nous avons observé que, pour ce code, la meilleure stratégie de partage des données entre les *threads* OpenMP était (dans l'ordre):

- `!$OMP DO` sur la boucle la plus exterieure;

```
do L = 1,LMNV
 do JK = 1,NJK
    VJAC(JK,I)   = VJAC(JK,I)   + FVJAC(L,I)*TCOS(JK,L)
    SIGMAV(JK,I)= SIGMAV(JK,I)+ FSIGMV(L,I)*TCOS(JK,L)
    TAUV(JK,I)   = TAUV(JK,I)   + FTAUV(L,I)*TCOS(JK,L)
    PARPV(JK,I)  = PARPV(JK,I)  + FPARPV(L,I)*TCOS(JK,L)
    PERPV(JK,I)  = PERPV(JK,I)  + FPERPV(L,I)*TCOS(JK,L)
    RV(JK,I)     = RV(JK,I)     + FRV(L,I)*TCOS(JK,L)
    ZV(JK,I)     = ZV(JK,I)     + FZV(L,I)*TSIN(JK,L)
    RVT(JK,I)    = RVT(JK,I)    - FRV(L,I)*MX(L)*TSIN(JK,L)
    ZVT(JK,I)    = ZVT(JK,I)    + FZV(L,I)*MX(L)*TCOS(JK,L)
    RVP(JK,I)    = RVP(JK,I)    + FRV(L,I)*NX(L)*TSIN(JK,L)
    ZVP(JK,I)    = ZVP(JK,I)    - FZV(L,I)*NX(L)*TCOS(JK,L)
 end do
end do
```

35s

```
do JK = 1,NJK
 do L = 1,LMNV
    VJAC(JK,I)   = VJAC(JK,I)   + FVJAC(L,I)*TCOS(JK,L)
    SIGMAV(JK,I)= SIGMAV(JK,I)+ FSIGMV(L,I)*TCOS(JK,L)
    TAUV(JK,I)   = TAUV(JK,I)   + FTAUV(L,I)*TCOS(JK,L)
    PARPV(JK,I)  = PARPV(JK,I)  + FPARPV(L,I)*TCOS(JK,L)
    PERPV(JK,I)  = PERPV(JK,I)  + FPERPV(L,I)*TCOS(JK,L)
    RV(JK,I)     = RV(JK,I)     + FRV(L,I)*TCOS(JK,L)
    ZV(JK,I)     = ZV(JK,I)     + FZV(L,I)*TSIN(JK,L)
    RVT(JK,I)    = RVT(JK,I)    - FRV(L,I)*MX(L)*TSIN(JK,L)
    ZVT(JK,I)    = ZVT(JK,I)    + FZV(L,I)*MX(L)*TCOS(JK,L)
    RVP(JK,I)    = RVP(JK,I)    + FRV(L,I)*NX(L)*TSIN(JK,L)
    ZVP(JK,I)    = ZVP(JK,I)    - FZV(L,I)*NX(L)*TCOS(JK,L)
 end do
end do
```

11s

Fig. 5 – La principale accélération (speedup) a été obtenue par inversion des indices dans les boucles DO

```
!$OMP DO
      do JK = 1,NJK
       do L = 1,LMNV
          VJAC(JK,I)   = VJAC(JK,I)   + FVJAC(L,I)*TCOS(JK,L)
          SIGMAV(JK,I) = SIGMAV(JK,I) + FSIGMV(L,I)*TCOS(JK,L)
          TAUV(JK,I)   = TAUV(JK,I)   + FTAUV(L,I)*TCOS(JK,L)
          PARPV(JK,I)  = PARPV(JK,I)  + FPARPV(L,I)*TCOS(JK,L)
          PERPV(JK,I)  = PERPV(JK,I)  + FPERPV(L,I)*TCOS(JK,L)
          RV(JK,I)     = RV(JK,I)     + FRV(L,I)*TCOS(JK,L)
          ZV(JK,I)     = ZV(JK,I)     + FZV(L,I)*TSIN(JK,L)
          RVT(JK,I)    = RVT(JK,I)    - FRV(L,I)*MX(L)*TSIN(JK,L)
          ZVT(JK,I)    = ZVT(JK,I)    + FZV(L,I)*MX(L)*TCOS(JK,L)
          RVP(JK,I)    = RVP(JK,I)    + FRV(L,I)*NX(L)*TSIN(JK,L)
          ZVP(JK,I)    = ZVP(JK,I)    - FZV(L,I)*NX(L)*TCOS(JK,L)
       end do
      end do
!$OMP END DO
```

→

```
      do l=1,lmnv
!$OMP PARALLEL SECTIONS
!$OMP SECTION
         do JK = 1,NJK
            VJAC(JK,I)   = VJAC(JK,I)   + FVJAC(L,I)*TCOS(JK,L)
            SIGMAV(JK,I) = SIGMAV(JK,I) + FSIGMV(L,I)*TCOS(JK,L)
            TAUV(JK,I)   = TAUV(JK,I)   + FTAUV(L,I)*TCOS(JK,L)
            PARPV(JK,I)  = PARPV(JK,I)  + FPARPV(L,I)*TCOS(JK,L)
         end do
!$OMP SECTION
         do jk=1,njk
            PERPV(JK,I)  = PERPV(JK,I)  + FPERPV(L,I)*TCOS(JK,L)
            RV(JK,I)     = RV(JK,I)     + FRV(L,I)*TCOS(JK,L)
            ZVT(JK,I)    = ZVT(JK,I)    + FZV(L,I)*MX(L)*TCOS(JK,L)
            ZVP(JK,I)    = ZVP(JK,I)    - FZV(L,I)*NX(L)*TCOS(JK,L)
         end do
!$OMP SECTION
         do jk=1,njk
            ZV(JK,I)     = ZV(JK,I)     + FZV(L,I)*TSIN(JK,L)
            RVT(JK,I)    = RVT(JK,I)    - FRV(L,I)*MX(L)*TSIN(JK,L)
            RVP(JK,I)    = RVP(JK,I)    + FRV(L,I)*NX(L)*TSIN(JK,L)
          end do
!$OMP END PARALLEL SECTIONS
        end do
```

2.8s **1.8a**

Fig. 7 – un exemple d'utilisation de la directive de compilation `!$OMP SECTIONS` dans une boucle indépendante (embarrassingly parallel loop)

- division des grandes boucles indépendantes et placement dans des directives `!$OMP SECTIONS`;
- combinaisons de petites boucles;
- appels OpenMP imbriqués;

La fig. 7 montre un exemple de partage en sections indépendantes d'une boucle et la performance associée (en secondes).

Après la parallélisation avec OpenMP, la performance a augmenté d'un facteur de 3 en comparaison de la version optimisée non parallèle (fig. 8 et tab. 4).

On peut observer qu'une parallélisation manuelle est meilleure que ce que peut faire le compilateur : `-O1 -openmp` est environ 2 fois plus rapide que `-O3 -xT -parallel` sur le code non optimisé. Notons aussi qu'il n'y a quasiment pas de différence entre les *flags* de compilation et que la performance ainsi obtenue est comparable à celle obtenue sur le NEC SX-5.

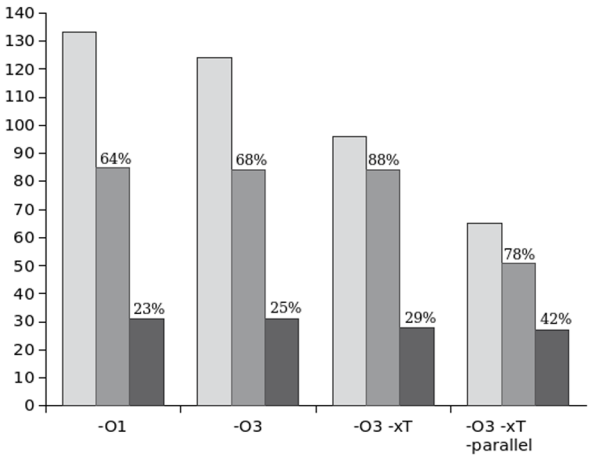

Fig. 8 – performance après la parallélisation avec OpenMP

		opt	OMP
-O1	133s	85s	31s
-O3	124s	84s	31s
-O3 -xT	96s	84s	28s
-O3 -xT -parallel	65s	51s	27s
NEC SX-5		**30s**	

Tab. 4 – performance après parallélisation. Les pourcentages sont relatifs à avant la parallélisation avec le même flag de compilation

Résultats

Comme petit résumé de ce qu'on a pu observer de la parallélisation avec OpenMP, nous pouvons établir que –> **l'optimisation et la parallélisation à la main est meilleure que le compilateur –> sur des codes instables peut être aussi rapide que sur des codes stables avec OpenMP –> le Xeon 5160 (WoodCrest) est comparable au NEC SX-5**... mais il est 30 fois moins cher. Le tab. 5 présente un petit résumé des performances obtenues selon les *flags* de compilation ainsi que le paradigme de parallélisation utilisé (autoparallélisation ou manuellement avec les directives OpenMP).

main routine	routine	routine	non opt	sp[5] opt
eqinvm			0.12	0.13
veqrec	cospol		0.64	0.0.64
	lgikvm		35.45	13.15
	mtaskl	lamcal	24.78	17.77
mtaskb	vmtobo		24.16	19.35
	extint		5.73	5.78
	metric		15.73	6.18
	bophys		4.96	5.50
stabin			0.14	0.14
mtaska	fourin		10.01	7.74
Total			132.97	84.69

Tab. 5 – résumé des performances des quatre principales routines posant problème

Il est intéressant de noter que le gain de performance entre la version OpenMP-isée et la version optimisée est d'un facteur de 3.1 pour 4 *threads* OpenMP, signifiant ainsi que la scalabilité est bonne. Difficile de faire mieux.

Second exemple: Solveur de Helmholtz 3D

La problématique

L'équation de Helmholtz peut être utilisée pour résoudre des problèmes complexes par exemple un flux thermal induit par les forces de flottaison *(buoyancy forces)* à l'intérieur d'un

volume fermé où la différence de température est imposée entre les bords.

En partant des équations de Navier-Stockes et de l'équation de l'énergie sous l'hypothèse de Boussinesq, il est possible de diviser le problème initial comme suit [14, 15]:
[Première étape]

$$a^* + \nabla p = f \text{ in } \Omega \quad (1)$$

$$\nabla \cdot a^* = 0 \text{ in } \overline{\Omega} \quad (2)$$

$$a^* \cdot n = \left(\frac{\delta u}{\delta t} \Delta u\right) \cdot n \text{ in } \delta\Omega \quad (3)$$

[Seconde étape]

$$\left(\frac{\delta u}{\delta t} \Delta u\right) = a^* \text{ in } \Omega \quad (4)$$

plus les conditions de bord.

La méthode de projection-diffusion brièvement décrite ci-dessus entraîne quatre problèmes de Helmholtz utiles pour résoudre chaque itération: un pour chaque composante cartésienne de la vitesse et un pour la température. Puisque chaque variable primitive est étendue en polynôme de Chebyshev, pour le cas présent, une méthodologie pseudo-spectrale est utilisée en imposant une erreur nulle aux points de la grille (plus précisément aux points de Gauss-Lobatto), une grille de 1693 points est suffisante pour exécuter une simulation numérique directe *(DNS Direct Numerical Simulation)* dans un régime turbulent.

Donc, la matrice discrète du problème pour une quantité scalaire générale peut être écrite comme:

$$\nabla^2 \text{Ø}n+1 - a(\Delta t)\text{Ø}n+1 = f^* \text{ in } \Omega \quad (5)$$

ou, en séparant l'opérateur Laplacien discret modifié

$$(L^*_x + L^*_y + L^*_z - a(\Delta t) I)\, \text{Ø}n+1 = F^* \quad (6)$$

En outre, résolvant un problème de valeur propre – vecteur propre de l'opérateur Laplacien discret modifié pendant une étape de prétraitement, alors le problème peut être résolu simplement en utilisant la multiplication matrice – matrice pour chaque direction cartésienne. En fait, le problème final établit:

$$H_x H_y H_z\, [\Lambda_x + \Lambda_y + \Lambda_z + a(\Delta t) I]\, H_x^{-1} H_y^{-1} H_z^{-1} \text{Ø}n+1 = F_1^* \quad (7)$$

où

$$H_i \Lambda_i H_i^{-1} = L_1^* \quad (8)$$

et il est possible de démontrer que H_i et Λ_i sont proportionnels à la matrice identité dans les directions $j \neq i$.

Optimisation sur un processeur

Avant de commencer une quelconque parallélisation du code, il est très important, comme nous l'avons vu précédemment, de tirer le maximum de performance de l'application sur un processeur en optimisant manuellement. L'objectif étant d'obtenir un exécutable qui offre une performance similaire, quels que soient les *flags* de compilation, même si le déroulement des boucles et la vectorisation des boucles internes sont faits par le compilateur.

Un bon point de départ est d'identifier la routine qui prend le plus de temps sur l'entier du code, parce qu'une infime amélioration de cette routine entraîne une amélioration globale non négligeable. En utilisant l'outil *gprof*[6] couplé avec l'option de compilation `-qp(-pg)`, on obtient directement un *profile* du code. Le tableau 6 présente le *profiling* de l'entier du solveur de Navier-Stokes.

À première vue, il semble que la routine `helm3col` (le solveur de Helmholtz) est la partie du code la plus gourmande en temps. C'est sur cette routine que nous allons nous concentrer.

Le tab. 7 présente les performances en terme de temps passé pour une seule itération enregistrée sur un processeur de la machine Pleiades2 (Xeon 64, 2.8 GHz). La première colonne présente le cas d'une compilation non agressive `-O1` sans optimisation à la main, la seconde présente le résultat obtenu avec une compilation agressive `-O3 -xW`, ensuite les résultats présentent les mêmes *flags* de compilation mais avec une optimisation à la main. Par optimisation manuelle, nous entendons simplement que toutes les boucles `DO` ont été modifiées afin d'aider le compilateur à les traiter (réordonnancement des index, déroulement, division, etc.).

NH-NACO [s/iter]	NH-ACO [s/iter]	NH-ACO [s/iter]	NH-ACO [s/iter]
33.39(33.32)	6.90(4.42)	3.60(3.63)	3.10(3.12)

Tab. 7 – Différentes performances obtenues par modification des flags de compilation et optimisation manuelle

Each sample counts as 0.01 seconds						
% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
38.20	447.73	447.73	200	2.24	2.82	helm3col–
6.53	524.26	76.53				mkl–blas–p4–dgcopyan
6.36	598.79	74.53	1252	0.06	0.06	permyz–
6.17	671.07	72.28	1252	0.06	0.06	permyz–
5.28	732.91	61.84				mkl–blas–p4–dcopy
5.00	791.51	58.60				mkl–blas–p4–dgcopybn
...						

Tab. 6 – Profile plat en utilisant gprof avec les options de compilation adéquates. La routine qui prend le plus de temps est le solveur lui-même: `helm3col`

[6] sur une architecture sur laquelle on peut faire confiance à cet outil (ndt)

En outre, nous avons testé la capacité HT *(HyperThreading)* (voir [9]) des processeurs Xeon afin d'obtenir un gain pour l'exécution d'une instruction parallèle très simple (la capacité HT autorise une augmentation infime de performance sur un Pentium 4). Le tab. 8 résume ce qui a été obtenu en utilisant le *flag* de compilation `-parallel` (l'autoparallélisation automatique du compilateur). Les deux versions du compilateur sont à nouveau comparées.

NH-ACO [s/iter]	NH-ACO [s/iter]
33.00(4.85)	3.15(3.21)

Tab. 8 – Influence de l'HyperThreading sur les performances avec l'auto-parallélisation du compilateur

Il apparaît clairement que la toute dernière version du compilateur Intel (la version 9.1e) n'est pas capable de corriger les erreurs d'écriture du code, les performances sont ainsi dégradées drastiquement. En effet, dans ce cas, l'auto-parallélisation semble annuler toutes les autres optimisations et donne une performance obtenue avec une optimisation basse `-O2`. D'un autre côté, la version précédente du compilateur (la version 9.0 disponible sur Pleiades2) donne des résultats bien meilleurs même s'ils ne sont pas comparables avec une optimisation manuelle. Notons finalement que la compilation du code optimisé manuellement avec l'une ou l'autre des versions du compilateur donne les mêmes résultats.

Parallélisation

L'étape suivante est la parallélisation du code sur une machine SMP. Pour ce faire, nous utiliserons les nouvelles architectures multi-cœurs d'Intel Woodcrest. Tout ce qui va suivre a été obtenu sur un seul nœud du calculateur Pleiades2+. Les résultats qui vont suivre présentent uniquement une parallélisation manuelle; l'avantage de l'utilisation des directives OpenMP à la place de l'auto-parallélisation va être démontrée.

L'utilisation des directives OpenMP a permis de paralléliser des opérations complexes qui concernent principalement une quadruple boucle imbriquée où une multiplication matice – matrice avec des indices *shiftés* est effectuée entre une matrice 2D et une matrice 3D (10% du temps de l'entier du code est utilisé pour cette opération).

D'un point de vue plus précis, il est clair que l'exécution du code sur quatre cœurs d'un nœud Woodcrest est environ 20% plus rapide que l'auto-parallélisation, même optimisée manuellement. Cela signifie qu'après avoir aidé le compilateur (déroulement des boucles, réordonnancement d'index, etc.), il est possible d'augmenter la performance d'au moins 20%. Afin d'être complet, il a été dit plus haut que la version parallèle du code prenait 11.07 [s/iter] compilé avec la version 9.1e et 1.99 [s/iter] avec la version 9.0; ceci sur 4 cœurs. Cela signifie que cette version du code tourne sur 4 cœurs aussi rapidement que la version optimisée manuellement sur un seul cœur !

Avant de conclure, un dernier aspect doit être abordé. Afin de se rapprocher le plus possible de la performance maximale autorisée sur un cœur, nous avons observé que certaines routines BLAS 1 hautement optimisées par Intel (la bibliothèque MKL) ne supportaient pas la scalabilité ! Ainsi il est fortement recommandé de réécrire les sous-routines BLAS 1 (`dscal`, `dcopy`, etc.) en utilisant les directives OpenMP. Le tab. 10 présente la loi d'échelle pour la routine `dcopy` (de la bibliothèque MKL version 8.0) ainsi que pour celle écrite à la main `ucopy` sur 1693 éléments de matrice. Cela même si les BLAS sont implémentés dans MKL avec les directives OpenMP.

compiling options	1cpu [s/iter]	2cpus [s/iter]	4cpus [s/iter]
-parallel	2.07	1.37	1.07
-openmp	2.07	1.20	0.87

Tab. 9 – Performance auto-parallélisation vs. directives manuelles OpenMP sur 1, 2 et 4 cœurs d'un nœud multi-CPU's multi-cœurs Woodcrest avec le code totalement parallélisé

subroutine	1cpu [s]	2cpus [s]	4cpus [s]
dcopy	2.56×10^{-2}	2.56×10^{-2}	2.56×10^{-2}
ucopy	2.54×10^{-2}	1.61×10^{-2}	1.58×10^{-2}

Tab. 10 – Loi d'échelle des deux routines dcopy (de MKL) et ucopy (écrite avec OpenMP)

Quelques observations

Dans cette dernière section, nous donnons quelques conseils sur la base d'observations faites lors de l'optimisation et la parallélisation d'applications à l'aide d'OpenMP.

De façon générale, la procédure proposée est celle communément utilisée en génie logiciel lorsqu'il s'agit d'optimiser une application: **Monitoring –> Optimisation –> Tests**. Dans cette optique, il faut disposer d'une bonne mesure du temps. Des outils existent (VTUNE, gprof, etc.). Mais nous avons pu observer que le meilleur monitoring reste l'utilisation de la fonction `gettimeofday()`. Le *profiling gprof* donne des résultats erronnés.

La version du compilateur ainsi que celle des versions utilisées peuvent jouer un rôle. Dans cette optique, les ingénieurs système laissent généralement le choix de la version à l'utilisateur. C'est ainsi que sur la machine Pleiades [4], pas moins de 5 versions du compilateur Intel, 3 versions de la bilbiothèque MKL sans compter le compilateur libre *gfortran* sont disponibles pour les utilisateurs.

Conclusion

On a vu que parvenir à une performance aussi proche que possible de la performance maximum d'un nœud multiprocesseurs multi-cœurs demande du travail. Il est illusoire de se dire que le compilateur y arrivera seul. Malgré les efforts constants des compagnies qui développent des compilateurs de plus en plus performants, tout utilisateur/développeur HPC devrait être capable de fournir un code source à la performance identique quelle que soit l'option de compilation, en d'autres termes, un code source optimisé à la main pour un type d'architecture.

En résumé, la méthodologie présentée dans cet article se décompose en:

I Optimisation sur un processeur
  (a) Compilation avec l'optimisation la plus basse
  (b) Modification de l'ordre des index
  (c) Déroulement des boucles
  (d) Vectorisation

II Parallélisation
  (a) OpenMP
  (b) ev. Optimisation maximum
  (c) ev. auto-parallélisation

Et que ce même utilisateur/développeur garde toujours en mémoire que la magie n'existe pas; qu'un code médiocre mono-processeur ne donnera jamais de bonnes performances sur plusieurs processeurs.

On ne fait pas une bonne soupe avec des légumes avariés

Machines disponibles à l'école

Le tab. 11 décrit les différentes machines SMP ou NUMA disponibles à l'école. Leur utilisation passe par une politique qui varie selon la machine. Les auteurs conseillent aux utilisateurs de se renseigner auprès des ingénieurs système attitrés aux machines.

Le tab. 11 présente les machines sur lesquelles il est possible d'exécuter un code implémenté avec les directives de compilation OpenMP. Notons que la performance d'une machine NUMA (Accès non uniforme de la mémoire) peut varier en fonction de l'utilisation de cette dernière.

Machine	Type	#Nœuds	#cœurs	#threads max per node
Pleiades2+	SMP	99	4	4
Mizar Cluster	SMP	224	2	2
Alcor	SMP	24	4	4
Mizar ICT	NUMA	8	16	-
Altix MX	NUMA	1	16	-

Tab. 11 – Machines connues disponibles à l'école

Références

[1] Histoire des micro-processeurs, site Web, 2007, ***www.cpu-info.com***

[2] Liste des 500 superordinateurs les plus puissants de la planète, site Web, 2007, ***www.top500.org***

[3] Site officiel du développement d'OpenMP, site Web, 2007, ***www.openmp.org***

[4] Site du projet Pleiades, site Web, 2007, ***pleiades.epfl.ch***

[5] Performance API, site Web , 2007, ***icl.cs.utk.edu/papi/index.html***

[6] Gruber, R., Volgers, P., De Vita, A., Stengel, M., Tran, T.-M. , *Parameterisation to tailor commodity clusters to applications*. Future Generation Computer Systems, 19:111–120, 2003.

[7] Anderson D.V., Cooper W.A., Fu G.Y., Gengler M., Gruber R., Merazzi S., Schwenn U., *TERPSICHORE. A Three-Dimensional Ideal Magnetohydrodynamic Stability Program, EPFL* - Supercomputing Review 3, 29 - 32 , 1991

[8] W. A. Cooper, S. P. Hirshman, S. Merazzi and R, Gruber, *3D Magnetohydrodynamic Equilibria With Aniostropic Pressure,* Computer Physics Communications 72 (1992), 1-13.

[9] *Intel Fortran Compiler: Optimizing Applications*, Document Number 307781-003US, Intel Fortran Compiler version 9.1 Documentation, Online ***ftp://download.intel.com/support/performancetools/fortran/linux/v9/optaps–for.pdf***

[10] Intel VTune, Available for download on Intel site, ***www.intel.com***

[11] *OpenMP Application Program Interface Specifications*, Version 2.5, May 2005, Available online: ***www.openmp.org/drupal/mp-documents/spec25.pdf***

[12] LAPACK library, OnLine on ***www.netlib.org***

[13] Etienne Gondet, Pierre-François Lavallée, Cours OpenMP – CNRS-IDRIS, Version 1.3, 2000, Online, ***www.idris.fr/data/publications/openmp.pdf***

[14] Haldenwang,P. and Labrosse, G. and Abboudi, S.A. and Deville, M. , Chebyshev, *3D Spectral and 2D Pseudospectral Solvers for the Helmholtz Equation*, Journal of Computational Physics , Vol 55, pp. 115-128 , 1984

[15] Leriche, E. , *Direct Numerical Simulation of a Lid-Driven Cavity Flow by a Chebyshev Spectral Method,* PhD Thesis Numéro 1932, École Polytechnique Fédérale de Lausanne, 1999 ■