SMPスレッド並列WG

成果報告書
《公開版》

《活動期間：2004 年 11 月～2006 年 10 月》

2006 年 10 月 31 日作成

2007 年 5 月 11 日公開

サイエンティフィック・システム研究会

SMP スレッド並列 WG

ー 公開版 目次 ー

1．はじめに

1.1 背景と目的

2．各プログラムの検討

2.1 平行平板間乱流解析ｺｰﾄﾞの性能評価とﾊｲﾌﾞﾘｯﾄﾞ並列性能推定の試み
2.2 三次元圧縮性流体解析プログラム UPACS の性能評価
2.3 非構造格子 Euler/Navier-Stokes ｿﾙﾊﾞ JTAS のｽﾚｯﾄﾞ並列最適化
2.4 近似逆行列型前処理つき CG 法の並列性能評価
2.5 SMP におけるスレッド並列の台数効果と高速化手法
2.6 ﾏﾙﾁｶﾗｰｵｰﾀﾞﾘﾝｸﾞによるｽﾚｯﾄﾞ並列型 ICCG ｿﾙﾊﾞによる HPC2500 のベンチマーク評価
2.7 スレッド並列プログラムの性能測定

3．コンパイラ機能/性能およびチューニング情報

3.1. コンパイラの機能および性能についての情報
・プロファイラ情報の Cover(%)について
・PA ｶｳﾝﾀによる性能情報採取方法および注意点について
・構造体とｱﾛｹｰﾀﾌﾞﾙ配列の組み合わせにおける制約について
3.2. チューニングについての情報
・2 次元拡散方程式のスレッド並列チューニングについて

4．（欠番）

5．おわりに

- これは、2006 年 10 月に作成した成果報告書から、非公開情報を除いたものです。項番の抜けなどがありますが、ご了承ください。
- 2007 年 5 月 11 日(第 29 回通常総会)をもって公開しました。

１．はじめに

1.1　背景と目的

富士通が、ベクトル計算機を要素とする分散主記憶型並列計算機 VPP シリーズから SMP を要素とする分散主記憶型のスーパーコンピュータ PRIMEPOWER へと製品転換を行うに伴い、過去 4 年間にわたって SMP 単体および SMP を要素とする分散主記憶型並列計算機における並列化の技術と評価を実施し、成果をあげてきた。(スカラ並列技術 WG、SMP クラスタ WG)

しかしながら、これらの並列化技術あるいは並列計算機利用技術が必ずしも十分に普及し、活用されている状況にはない。この理由としては、既存のプログラムを「利用するだけ」の利用者が増えてきたことや、計算機パワーが飛躍的に大きくなりプログラミングよりは本来のシミュレーションによる研究に時間を割くようになったことが考えられる。

しかしながらシミュレーションソフトウェアは実験的手法の計測器に相当するものであり、よりよい研究や開発を行おうとすればソフトウェアの内容にまで立ち入った関心を持つのは当然のことと思われる。本 WG ではプログラムのチューニングと、プログラムのひいては計算機の評価に関心を当てて活動した。

プログラムチューニングはこれを実施することにより、不必要に使われていた計算機資源（CPU 時間）を産みだすという経済効果もあれば、自分自身の研究開発が効率化されるという効果、本来であれば他の利用者が利用できる計算機資源を無駄遣いしなくてすむという効果がある。もちろんチューニングに要する費用や時間とのトレードオフ、プログラムのリーダビリティやメンテナンスコストへの考慮が求められるのも確かである。

また、並列化技術（計算機利用技術）や評価の研究が重要な側面の一つとして、「次の」計算機を開発するにあたり、ハードウェアや OS、コンパイラを考える指針を与えてくれることにある。ベクトル計算機出現当初のベクトル化技術研究と計算機評価に関する研究は高性能ベクトル計算機の開発と自動ベクトル化コンパイラの開発に貢献することができた。

以上のような認識から本 WG においてはこれまで以上に幅広いアプリ分野で、スレッド並列を軸として如何にスカラ並列の性能を引き出すかという観点から、引き続き SMP 計算機の性能を追求する並列化技術について研究することとした。本報告は 2 カ年にわたる WG 活動の内容をまとめたものである。

2.1 平行平板間乱流解析ｺｰﾄﾞの性能評価とﾊｲﾌﾞﾘｯﾄﾞ並列性能推定の試み

松尾裕一（宇宙航空研究開発機構）

1. はじめに

宇宙航空研究開発機構（以下，JAXA）では，旧航空宇宙技術研究所時代の 1980 年代後半から，スーパーコンピュータの比類ない計算能力を利用して，計算流体力学 Computational Fluid Dynamics (CFD)に代表される数値シミュレーション技術を先駆的に研究開発し，流体基礎現象の物理解明や航空宇宙機の設計開発に適用してきた．2002 年 10 月には，PRIMEPOWER HPC2500 の 18 筐体（2,304CPU）から成る大規模 SMP クラスタを導入し [1)]，CFD 技術のさらなる高度化と実問題への一層の適用を推進している．最近の CFD アプリケーションの傾向として，工学系の解析では，設計への高度適用に係る現実の物体形状を見据えた複雑形状への対応が進んでおり，数 1,000 万点メッシュ上で,マルチブロック構造格子や非構造格子，重合格子といった各種の格子系を用いることにより，エンジンナセルやフラップのついた機体まわりの流れ解析やエンジン内部の複数段の流れ解析が可能になって来ている．また，時間とともに現象が変化したり，物体が移動する非定常（過渡的）問題や，流体－構造などの多分野連成問題への適用，あるいは最適化を睨んだ多量のパラメータ解析などが扱われるようになって来ており，いずれの場合も既にプロダクションレベルでの実設計開発への適用が行われている．一方，学術系の解析では，マルチスケールやマルチフィジクスといった現実の現象をできるだけ忠実に取り扱う方向へと進展しており，乱流や燃焼流のシミュレーションでは，最大 10 億点メッシュ上で高速流や化学反応を考慮しつつ出力データは量にして時に 100GB を超えるようなケースも出現している．並列計算の観点からは，単純な幾何分割やデータ構造の分割から，計算領域を分割して各領域を CPU にマッピングする並列化手法や MPI によるデータ転送が主流となりつつある．その一方で，通信負荷の重い FFT や補間処理も依然として使われており，かつての大規模な流体解析のイメージに比べ，取り扱う問題の範囲やコードの多様性は相当に拡大して来ているといえる．

本報告では，JAXAの並列CFDコードの中で，メモリアクセス負荷及び転送負荷が比較的重い部類の「並行平板間乱流解析コード」を取り上げ，スレッド並列の性能チューニングと性能測定結果を示すとともに，コードの特性と並列性能の関係について考察する．また，アムダールの法則を拡張したハイブリッド並列における簡易な実効性能推定法を提示し，その推定精度を検証し有効性を示す.

2. 平行平板間乱流解析コードの特性と処理の概要

表 2.1 は，JAXA において主に航空関係で実際に使われている並列 CFD コード [2)~7)]の概要を示したものである．このうち，コード P2，P3 が学術系の課題（P2:燃焼流，P3:平行平板間乱流）を解析するためのものであり，あとの 4 本は工学系のコードである．いずれのコードも Navier-Stokes 方程式を基礎式とし，格子形状によってメモリアクセスパターン（ローカル or グローバル or 連続 or リスト）や転送パターンが，流体以外の部分を解くか否か等で演算密度が異なる．図 2.2 は，横軸にメモリコスト，縦軸に通信コストを取って，各コードの位置をプロットしたものである．ここで，メモリコスト=メモリアクセス時間/CPU 時間，通信コスト=通信時間/経過時間として，プロファイラ等で採取したデータから持って来た．無論，同じ CPU 数でも使用したスレッド数，プロセス数の組み合わせによって，あるいは問題サイズ，チューニングの有無などによってプロットされる位置は多少ずれるが，基本的な配置は大きく変わることはない．ここにプロットした値は，プロセス数はすべて 4 で統一して測定したものである．これより，6 本の CFD コードは，計算処理中心のタイプ 1（コード P1, P2），通信コストが多い（10%以上）タイプ 2（コード P3, P4），メモリアクセスが多いタイプ 3（コード P5, P6）の 3 タイプにほぼ分類できることがわかった．ちなみに，ベンチマークとして有名な LINPACK 及び NAS Parallel Benchmarks の MG と CG を参考までにプロットした.

表 2.1　主な JAXA 並列 CFD コード

Code (Name)	Application	Simulation model	Numerical method	Parallel strategy	Language
P1 (HJET)	Combustion	DNS	FD w. Chemistry	OpenMP + MPI	F77
P2 (LES)	Aircraft components	LES	FD	OpenMP + MPI	F77
P3 (CHANL)	Turbulence	DNS	FD with FFT	OpenMP + XPF	F77
P4 (HELI)	Helicopter	URANS	FD w. Overlapped mesh	AutoParallel + XPF	F77
P5 (UPACS)	Aeronautics	RANS	FV w. Multiblock mesh	MPI	F90
P6 (JTAS)	Aeronautics	RANS	FV w. Unstructured mesh	MPI	F77

LES: Large Eddy Simulation, DNS: Direct Numerical Simulation, RANS: Reynolds-Averaged Navier-Stokes, URANS: Unsteady RANS, FD: Finite Difference, FV: Finite Volume

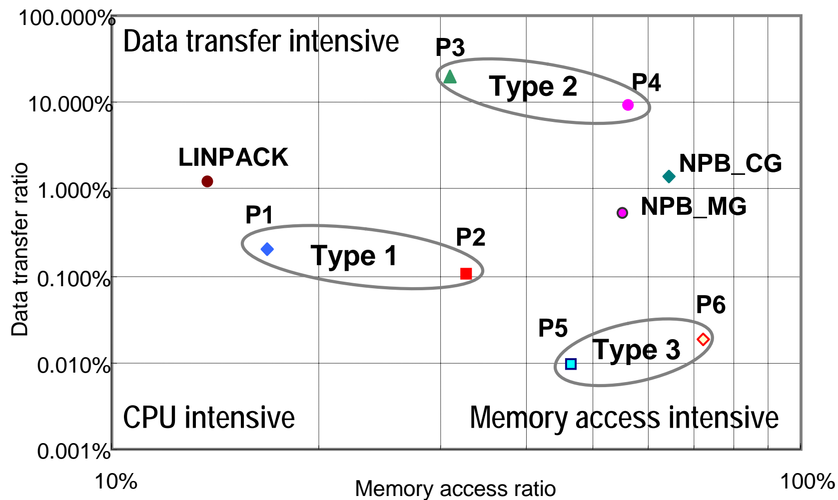


図 2.2　JAXA 並列 CFD コードの特性

本報告で扱った平行平板間乱流解析コード（以下，コード P3）は，非圧縮 Navier-Stokes 方程式を時間進行で解き，並列化に XPFortran を用いた約 20,000 行の Fortran77 プログラムである．時間進行には，粘性項に対しては２次精度クランクニコルソン法を，その他の項には３次精度ルンゲクッタ法を用いている．空間の離散化には４次精度差分を用いている．プログラムは，多くの 3 重 DO ループの計算から成り，一部に 3 重対角行列の行列解法，圧力ポアソン方程式の求解のために FFT を含む．JAXA コードの中では，図 2.2 にあるようにタイプ 2 に属し，メモリアクセス的には中程度であるが，並列軸の持ち替えを行っているために転送負荷は高い．

3. スレッド並列性能チューニングと並列性能評価

3.1 ASIS コードの性能分析

性能評価区間は，コード P3 の主計算ループ部分とし，時間計測には gettod を用いた．問題サイズは，現実には 2,048×448×1,536 であるが，チューニングの機動性を確保するため 2,048×32×1,536 とした．スレッド並列特性を調べるために，プロセスは４に固定し，スレッド並列数を変化させてプロファイラによりコスト分布，性能情報等を採取した．測定条件の概要を表 3.1 に，測定結果を表 3.2 及び図 3.3 に示す．また，プロファイラによる出力をリスト 3.4 に示す．

表 3.1　計算条件

実行ノード	32cpu×2 ノード
並列規模	プロセス並列…　XPFortran 4 プロセス スレッド並列…　自動並列 1,2,4,8 スレッド
格子サイズ	2,048×32×1,536（評価用データサイズ）
Iteration 回数	2 回 (計算処理 2 回＋統計処理 1 回を実行)
翻訳オプション	-Kfast_GP2=3,ocl,hardbarrier,mfunc=2,parallel,reduction,noeval -O5 -x40 -NRnotrap

表 3.2　性能測定結果

スレッド数	経過時間	性能比
1	747.47sec	1.00
2	745.18sec	1.00
4	614.95sec	1.22
8	553.67sec	1.35

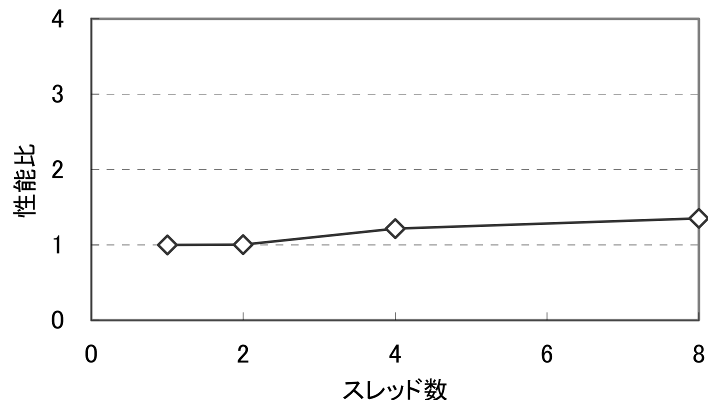


図 3.3　スレッド数に対する性能向上比

これより，ASIS コードはスレッド並列の性能向上特性は悪く，以下の問題が指摘される．

① 統計関連の処理(budget,budgek1,budgek2)のコストが大きい(8thread で合わせて約 46%)．

② バリア同期待ち(_libc_poll)のコストが大きい．

③ 実数のべき乗(g_adxi)のコストが大きい．

④ スレッド非並列のサブルーチン(ave)が残っている．

リスト 3.4

コスト分布（プログラム全体）

```
    ----------------------------------------------------------
         Samples     % Run      Barrier      Start       End
  --<1thread>-----------------------------------------------------------
     9.627300e+04    29.33  0.000000e+00        -         -   _libc_poll
     3.438300e+04    10.47  0.000000e+00        -         -   g_adxi
     1.767100e+04     5.38  0.000000e+00        -         -   _ioctl
     1.030100e+04     3.14  0.000000e+00     3621      5454   ave_
     8.121000e+03     2.47  0.000000e+00      774       785   cntdmat._PRL_10_
     5.516000e+03     1.68  0.000000e+00     1232      1241   cntdma._PRL_9_
     5.293000e+03     1.61  0.000000e+00       12       625   budgek1._PRL_1_
     5.235000e+03     1.59  0.000000e+00        -         -   prbar_probe
     5.025000e+03     1.53  0.000000e+00       12       625   budgek2._PRL_1_
     4.956000e+03     1.51  0.000000e+00      533       673   rk2._PRL_2_
  --<2thread>-----------------------------------------------------------
     3.877300e+04    11.23  0.000000e+00     5631      5885   budget._PRL_4_
     3.773200e+04    10.93  0.000000e+00       12       625   budgek2._PRL_1_
     3.729500e+04    10.80  0.000000e+00        -         -   g_adxi
     3.697400e+04    10.71  0.000000e+00       12       625   budgek1._PRL_1_
     1.033200e+04     2.99  4.000000e+00     3621      5454   ave_
     8.108000e+03     2.35  0.000000e+00     3931      3980   ave._PRL_7_
     6.839000e+03     1.98  0.000000e+00      774       785   cntdmat._PRL_10_
     6.447000e+03     1.87  0.000000e+00        -         -   prbar_probe
     5.476000e+03     1.59  0.000000e+00     1232      1241   cntdma._PRL_9_
     4.959000e+03     1.44  0.000000e+00      533       673   rk2._PRL_2_
  --<4thread>-----------------------------------------------------------
     5.842200e+04    13.34  0.000000e+00       12       625   budgek2._PRL_1_
     5.838600e+04    13.33  0.000000e+00       12       625   budgek1._PRL_1_
     5.770600e+04    13.17  0.000000e+00     5631      5885   budget._PRL_4_
     4.045400e+04     9.23  0.000000e+00        -         -   g_adxi
     1.954000e+04     4.46  0.000000e+00     3931      3980   ave._PRL_7_
     1.188300e+04     2.71  1.549000e+03     3621      5454   ave_
     6.711000e+03     1.53  0.000000e+00      774       785   cntdmat._PRL_10_
     6.653000e+03     1.52  0.000000e+00        -         -   prbar_probe
     6.288000e+03     1.44  0.000000e+00      389       454   stcs._PRL_23_
     5.860000e+03     1.34  0.000000e+00     3880      3914   ave._PRL_9_
  --<8thread>-----------------------------------------------------------
     9.751900e+04    16.34  0.000000e+00       12       625   budgek2._PRL_1_
     9.302900e+04    15.58  0.000000e+00       12       625   budgek1._PRL_1_
     8.546400e+04    14.32  0.000000e+00     5631      5885   budget._PRL_4_
     4.150100e+04     6.95  0.000000e+00        -         -   g_adxi
     4.024600e+04     6.74  0.000000e+00     3931      3980   ave._PRL_7_
     2.317900e+04     3.88  0.000000e+00      389       454   stcs._PRL_23_
     2.265300e+04     3.79  0.000000e+00      543       608   stcs._PRL_21_
     1.545800e+04     2.59  0.000000e+00     3880      3914   ave._PRL_9_
     1.425400e+04     2.39  3.892000e+03     3621      5454   ave_
     8.737000e+03     1.46  0.000000e+00        -         -   _libc_poll
```

関数性能測定状況（プログラム全体）

```
 ------------------------------------------------------------------------------------------
         CPU          Commit         MIPS         MFLOPS      L2-miss   mTLB-ir   mTLB-or   Cover
--<8thread>--------------------------------------------------------------------------------------
9.679459e+02  1.082560e+11  1.118410e+02  5.863436e+01   2.9545    0.0000    0.0000   80.0   budgek2._PRL_1_
9.232460e+02  1.080596e+11  1.170432e+02  6.137338e+01   2.8635    0.0000    0.0000   98.7   budgek1._PRL_1_
8.482010e+02  1.038536e+11  1.224398e+02  6.239195e+01   3.6745    0.0000    0.0000   99.4   budget._PRL_4_
4.111113e+02  2.529987e+11  6.154019e+02  9.890186e+01   0.3131    0.0000    0.0000   91.9   g_adxi
3.993641e+02  5.699815e+10  1.427223e+02  2.717652e+01   1.1222    0.0000    0.0000   97.5   ave._PRL_7_
2.302111e+02  3.134895e+10  1.361748e+02  2.204465e+01   3.1923    0.0000    0.0000   98.7   stcs._PRL_23_
2.249198e+02  2.945408e+10  1.309537e+02  2.009890e+01   3.3601    0.0000    0.0000   99.4   stcs._PRL_21_
1.533622e+02  1.140311e+10  7.435412e+01  1.451418e+01   4.0057    0.0000    0.0000   99.4   ave._PRL_9_
1.416606e+02  1.372047e+11  9.685453e+02  7.923003e+01   0.0135    0.0000    0.0000   86.6   ave_
2.235310e+00  2.945278e+09  1.317615e+03  4.221320e-01   0.0076    0.0000    0.0000    0.9   _libc_poll
------------------------------------------------------------------------------------------
6.119807e+02  2.270770e+12  3.710525e+03  1.244961e+03   1.1135    0.0000    0.0000   62.0   Process Total
```

3.2 スレッド並列性能チューニング

上記の測定結果を基に原因を分析し，次の性能チューニングを段階的に試みた．

【Tune １】実数のべき乗計算の乗算化

以下のようにべき乗計算を行う箇所が含まれているが，ソースは-Knoeval で翻訳されているため，べき乗を乗算化する最適化が抑止されていた．これに対し，翻訳時に-Keval を指定することで，べき乗を乗算に置き換えた．

リスト 3.5

実数のべき乗使用例

```
!XOCL SPREAD DO /ISE
      DO 61 J=1,JG
      DO 61 K=1,KG
      DO 61 I=1,IG
       U_RMST(J)=U_RMST(J)+( U(I,J,K,1)-EA_U(J) )**2
       V_RMST(J)=V_RMST(J)+( U(I,J,K,2)-EA_V(J) )**2
       W_RMST(J)=W_RMST(J)+( U(I,J,K,3)-EA_W(J) )**2
       ・・・
       UVT(2,J)=UVT(2,J)+
     &     (((U(I,J,K,1)-EA_U(J))+(U(I-1,J,K,1)-EA_U(J)))*0.5D0)
     &    *(((U(I,J,K,2)-EA_V(J))+(U(I,J-1,K,2)-EA_V(J-1)))*0.5D0)
     &          *(TMPT2)
   61 CONTINUE
!XOCL END SPREAD
```

【Tune ２】 False Sharing が発生する配列の次元追加

多重ループのワーク変数が１次元配列になっている箇所で，複数スレッドによるキャッシュライン競合(False Sharing)が発生し，スレッド数が増加するほど性能が劣化している可能性があったので，以下のように配列に１次元追加してすき間を空けることで，False Sharing を回避した．

リスト 3.6

変更前	変更後
	PARAMETER (NPad=8)
DIMENSION U11(0:JG),U33(0:JG)	DIMENSION U11(NPad,0:JG),U33(NPad,0:JG)
・・・	・・・
!XOCL LOCAL U11(/ISF),U33(/ISF)	!XOCL LOCAL U11(:,/ISF),U33(:,/ISF)
・・・	・・・
!XOCL SPREAD DO /ISE	!XOCL SPREAD DO /ISE
p DO 110 J=1,JG	p DO 110 J=1,JG
p DO 110 K=1,KG	p DO 110 K=1,KG
p DO 110 I=1,IG	p DO 110 I=1,IG
C―――――――――――――――― K	C―――――――――――――――― K
p U11(J)=U11(J)+UT(I,J,K,1)**2	p U11(1,J)=U11(1,J)+UT(I,J,K,1)**2
p U33(J)=U33(J)+UT(I,J,K,3)**2	p U33(1,J)=U33(1,J)+UT(I,J,K,3)**2
C――――――――――――――――	C――――――――――――――――
・・・	・・・
p 110 CONTINUE	p 110 CONTINUE
!XOCL END SPREAD	!XOCL END SPREAD
・・・	・・・
!XOCL SPREAD DO /ISE	!XOCL SPREAD DO /ISE
P DO 210 J=1,JG	P DO 210 J=1,JG
P AK(J)=0.5*(U11(J)+・・・+U33(J))*DOOP	P AK(J)=0.5*(U11(1,J)+・・・+U33(1,J))*DOOP
・・・	・・・
P 210 CONTINUE	P 210 CONTINUE
!XOCL END SPREAD	!XOCL END SPREAD

（変更前の吹き出し）XPF の分割ループで並列化したため回転数が少ない

【Tune ３】 ロードバランス不均等なループの融合

バリア同期待ち(_libc_poll)の呼び出しコストが大きいサブルーチンを調査したところ，特定のプロセスだけ動作するような spread do ループが複数連続して使用されていた．これに対し，以下のように複数の spread do ループを融合し，各プロセスが同時に動作するようにした．

リスト 3.7

<table>
<tr><th>変更前</th><th>変更後</th></tr>
<tr><td>

```
C================================ FOR JA
!XOCL SPREAD DO /ISC
      DO 601 J=JA,JA
        IF (J.EQ.JA) THEN
        DO 201 K=1,KG
          ・・・
  201   CONTINUE
        ENDIF
  601 CONTINUE
!XOCL END SPREAD
C================================ FOR JB
!XOCL SPREAD DO /ISC
      DO 602 J=JB,JB
        IF (J.EQ.JB) THEN
        DO 202 K=1,KG
          ・・・
  202   CONTINUE
        ENDIF
  602 CONTINUE
!XOCL END SPREAD
      ・・・
C================================ FOR JV
!XOCL SPREAD DO /ISC
      DO 611 J=JV,JV
        IF (J.EQ.JV) THEN
        DO 211 K=1,KG
          ・・・
  211   CONTINUE
        ENDIF
  611 CONTINUE
!XOCL END SPREAD
```

J=JA に対応するプロセスのみ動作(残りは同期待ち)

</td><td>

```
!XOCL SPREAD DO /ISC
      DO 601 J=JA,JV
C================================ FOR JA
        IF (J.EQ.JA) THEN
        DO 201 K=1,KG
          ・・・
  201   CONTINUE
C================================ FOR JB
        ELSE IF (J.EQ.JB) THEN
        DO 202 K=1,KG
          ・・・
  202   CONTINUE
          ・・・
C================================ FOR JV
        ELSE IF (J.EQ.JV) THEN
        DO 211 K=1,KG
          ・・・
  211   CONTINUE
C
        ENDIF
  601 CONTINUE
!XOCL END SPREAD
```

JA～JV の各プロセスが同時に動作

</td></tr>
</table>

【Tune 4】並列化率の拡大

自動並列化では構造が複雑なため並列化されていないループを OpenMP で並列化した．

リスト 3.8

<table>
<tr><th>変更前</th><th>変更後</th></tr>
<tr><td>

```
        DO 201 K=1,KG
        DO 201 I=1,IG
        IF(UT(I,JA,K,1).GE.(RPDFU(1,0))) THEN
         PDFN(0,JA,1)=PDFN(0,JA,1)+1.0D0
         GO TO 2001
        ELSE
         L=1
        ENDIF
 1001 CONTINUE
        IF(UT(I,JA,K,1).GE.(RPDFU(1,L))) THEN
         PDFN(L,JA,1)=PDFN(L,JA,1)+1.0D0
         GO TO 2001
        ELSE
         L=L+1
        ENDIF
        IF(L.LE.NBIN) THEN
         GO TO 1001
        ELSE
         PDFN(NBIN+1,JA,1)=PDFN(NBIN+1,JA,1)+1.0D0
        ENDIF
 2001 CONTINUE
  201  CONTINUE
```

jwd5133i-i:この DO ループは構造が複雑なため並列化されません．

</td><td>

```
!$omp parallel do reduction(+:pdfn) private(i,k,l)
        DO 201 K=1,KG
        DO 201 I=1,IG
        IF(UT(I,JA,K,1).GE.(RPDFU(1,0))) THEN
         PDFN(0,JA,1)=PDFN(0,JA,1)+1.0D0
         GO TO 2001
        ELSE
         L=1
        ENDIF
 1001 CONTINUE
        IF(UT(I,JA,K,1).GE.(RPDFU(1,L))) THEN
         PDFN(L,JA,1)=PDFN(L,JA,1)+1.0D0
         GO TO 2001
        ELSE
         L=L+1
        ENDIF
        IF(L.LE.NBIN) THEN
         GO TO 1001
        ELSE
         PDFN(NBIN+1,JA,1)=PDFN(NBIN+1,JA,1)+1.0D0
        ENDIF
 2001 CONTINUE
  201  CONTINUE
!$omp end parallel do
```

</td></tr>
</table>

3.3 スレッド並列チューニング後の性能測定結果

以下の表 3.9〜3.13 に，プロセスを 4 に固定しスレッド数を変化させた場合について，Tune1 から Tune4 まで段階的に適用した場合の性能測定結果を示す．ここで，「性能向上比」は，1 スレッド実行=1 としたときの性能向上比，「ASIS に対する性能比」は，ASIS の 1 スレッド実行=1 としたときの性能向上比を示す．

表 3.9　ASIS(チューニング無し)

スレッド数	経過時間[sec]	性能向上比	ASIS に対する性能比
1	747.47	1.00	1.00
2	745.18	1.00	1.00
4	614.95	1.22	1.22
8	553.67	1.35	1.35

表 3.10　Tune1 を適用した場合

スレッド数	経過時間[sec]	性能向上比	ASIS に対する性能比
1	595.46	1.00	1.26
2	664.20	0.90	1.13
4	559.61	1.06	1.34
8	506.03	1.18	1.48

表 3.11　Tune1+Tune2 を適用した場合

スレッド数	経過時間[sec]	性能向上比	ASIS に対する性能比
1	600.46	1.00	1.24
2	607.08	0.99	1.23
4	495.52	1.21	1.51
8	453.81	1.32	1.65

表 3.12　Tune1+Tune2+Tune3 を適用した場合

スレッド数	経過時間[sec]	性能向上比	ASIS に対する性能比
1	497.53	1.00	1.50
2	493.78	1.01	1.51
4	356.12	1.40	2.10
8	321.98	1.55	2.32

表 3.13　Tune1+Tune2+Tune3+Tune4 を適用した場合

スレッド数	経過時間[sec]	性能向上比	ASIS に対する性能比
1	425.36	1.00	1.76
2	299.08	1.42	2.50
4	203.61	2.09	3.67
8	164.08	2.59	4.56

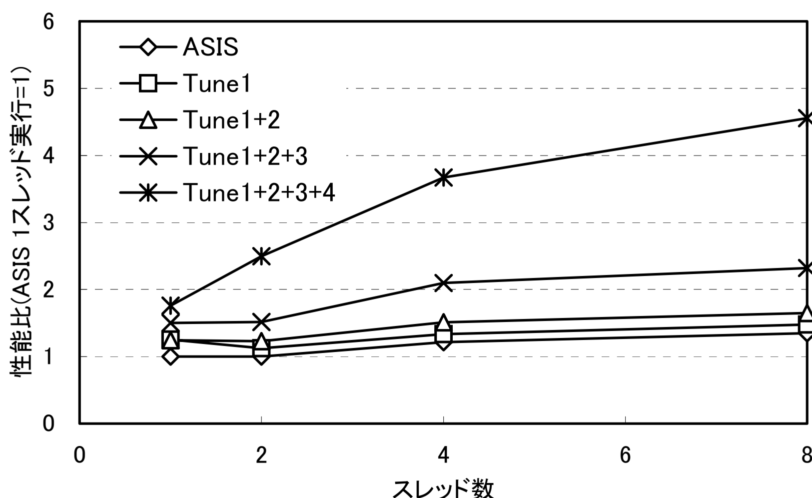


図 3.14　チューニングによるスレッド並列性能

図 3.14 は，チューニングによるスレッド並列性能（表 3.19～3.14 の ASIS に対する性能向上比）を，ASIS の 1 スレッド実行を基準にプロットしたものである．Tune1～Tune3 の効果は，スレッド並列性能に対する貢献としては大きくはないが，表からもわかるように ASIS に比べると素性能は 5 割近く向上しており，チューニングの効果は出ているといえる．Tune2 の False Sharing は，素性能の向上には繋がらないものの，スレッド並列時の不自然な性能低下は回避されている．Tune4 は，スレッド並列の効果を上げるのに大きく役立っているのがわかる．スレッド並列性能が上がらなかった主たる要因は，スレッド並列化されていないサブルーチンのせいであった．

経験によれば，チューニングの効果は，場合によっては非常に大きく出るが，そうでない場合もある．チューニングの原則論や事例集のようなものもあるようだが，ケースバイケースや問題規模によるということもあり，勘所を掴むのがなかなか難しい．経験と勘以外の形式知を如何に積み上げるかが課題である．また，ここでの性能測定は，他ジョブの影響をできる限り受けないような条件下で実施したのでチューニングの効果は明確な差分として出てきているが，現実には他ジョブの影響を受けて差分が明確に出ない場合もあり，システム側にもチューニングしやすいような環境構築を望みたいところである．

3.4 ハイブリッド並列の性能測定結果

次に，コード P3 の実際の問題サイズでの並列性能を調べた．スレッド数を固定し，プロセス数を変化させることにより，経過時間を測定した．他のジョブからの影響を避けるために，他のジョブが走って丼いない状態で測定した．

図 3.15(a)は，横軸にプロセス数，縦軸には 28 プロセス×1 スレッドのときの性能を基準(=1)としたときの性能比を取ったものを示す．何本かの線は，スレッドを 1,2,4,8,16 と変化させたものに相当する．通信が多いために，プロセス性能の直線性は良くなく，プロセス数が多くなると性能曲線の傾きは寝てくる．一方で，スレッド並列については，プロセス数一定のラインでみると，スレッドが多くなっても一定の割合で性能が向上しているのがわかる．参考のために，コード P1 の性能曲線を図 3.15(b)に示す．メモリアクセス，通信量ともに相対的に少ないこのコードの場合には，このように極めて良い直線性を示す．

これらの図は，横軸にプロセス数を取っているが，CPU 数一定で整理した場合の性能をコード P3 と P1 で比較したものを図 3.16 に示す．コード P1 の場合は，純 MPI の場合が最も良い性能を示しているのがわかる．これは，プロセス並列とスレッド並列を組み合わせた『ハイブリッド並列』の性能について一般に報告されている「純 MPI（プロセス）並列の方がハイブリッド並列より性能が良い」という事実 [8]と矛盾しない．しかし，通信の多いコード P3 の場合には，特定のプロセス数×スレッド数の組み合わせ（448CPU の場合 56 プロセス×8 スレッド）のときに最も良い性能を示し，純 MPI の場合に比べ，2 割ほど高い性能を示している．これは，P3 のようなコードの場合，プロセス数×スレッド数の組み合わせを適切に選ぶことにより，プログラムに手を加えることなしに性能向上を図ることができることを意味している．これは，何らかの性能推定モデルがあれば，適切なプロセス数×スレッド数を予め選択できることを示唆しており，ハイブリッド並列のメリットということもできる．

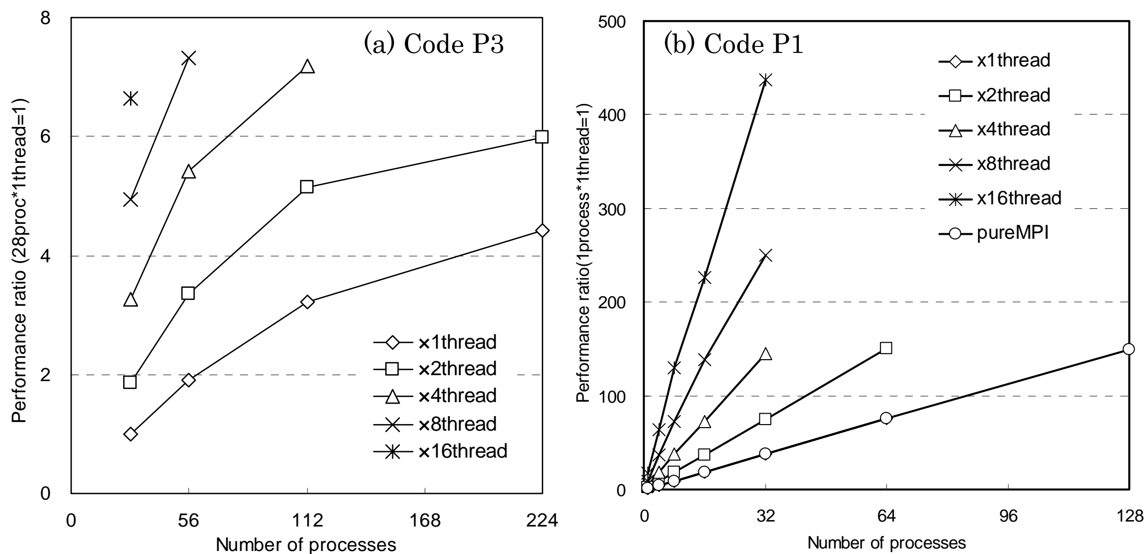


図 3.15　JAXA 並列 CFD コードのハイブリッド並列性能

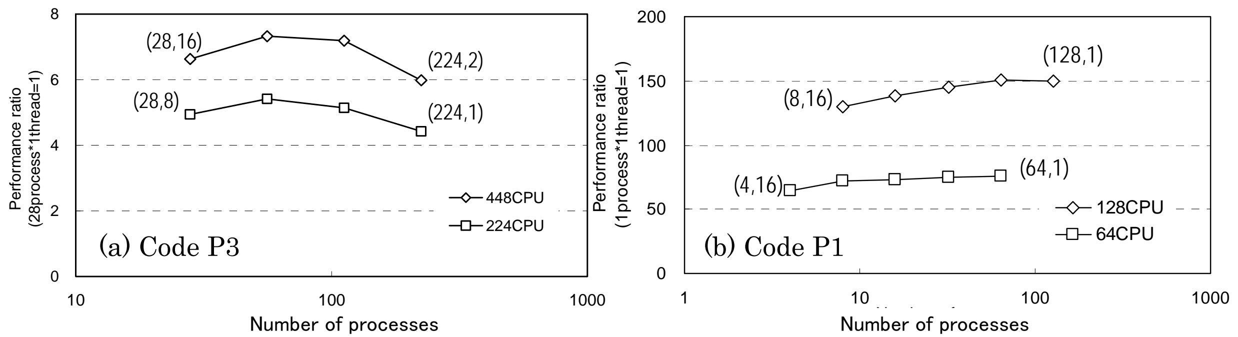


図 3.16　CPU 数一定時のハイブリッド並列性能の比較

4. 拡張アムダールの法則によるハイブリッド並列時の性能推定

現実問題として，最適なプロセス数×スレッド数を知るために，上記のような系統的な性能測定をいつも実施するわけにはいかない．ここでは，簡便な指標と性能推定モデルについて考えてみたい．

4.1 アムダールの法則

並列システムの性能を表す法則の一つにアムダールの法則がある．いま，あるプログラムにおいて，並列処理できる部分の割合（並列化率）を a とし，n 台の CPU を用いて得られる性能向上率を $S(n)$とすると，

$$S(n) = T_{serial}/T_{parallel} \tag{4.1}$$

ただし，T_{serial}，$T_{parallel}$ は，それぞれ逐次実行，並列実行時の CPU 時間をあらわし，

$$T_{parallel} = T_{serial} \times \{(1 - a) + a/n \} \tag{4.2}$$

の関係がある．もし，a = 1 なら $T_{parallel} = T_{serial} /n$，ゆえに $S(n) = n$ となり，並列性能は CPU 数の増加とともに完全にリニアに上昇する．また，$n \to \infty$ とすると $S(\infty) \to 1/(1-a)$であり，これは並列化率に応じて性能向上に上限があることを意味している．たとえば，a = 0.95 の場合 $S(\infty)$ = 20，すなわち性能向上率はたかだか 20，よって，20CPU 以上使うのは無駄ということになる．ただし，一般的には，n が大きくなると a も上昇するので，状況はそれほど悲観的ではないことに注意する．

4.2 ハイブリッド並列におけるアムダールの法則の拡張とその検証（その 1）

ハイブリッド並列における並列形態には，プロセス並列とスレッド並列が存在するので，上記の一般的なアムダールの法則は直接的には適用できない．いま，プロセス並列の並列数を n_p，並列化率を a_p，スレッド並列の並列数を n_t，並列化率を a_t とする．表 4.1(a)は，コード P1 の並列性能の実測値の一部を表にしたものである．4 プロセス×1 スレッドの場合の性能を基準(=1)としたときの性能比を示してある．プロセス並列化率 a_p は，スレッド 1 の場合の性能向上比（第 1 行）から，スレッド並列化率 a_t は，プロセス 1 の場合の性能向上比（第 1 列）から，式(1)(2)により導かれる関係 $a = (1-1/S)(1-1/n)$ を用いて求めることができ，これより平均の a_p = 1.018，a_t = 0.994 と求まる．いま，ハイブリッド並列におけるアムダールの法則として，通常のアムダールの法則(4.1)(4.2)の自然な拡張として，

$$S(n) = T_{serial}/T_{hybrid} \tag{4.3}$$

ただし，

$$T_{hybrid} = T_{serial} \times \{(1 - a_p) + a_p/n_p \} \times \{(1 - a_t) + a_t/n_t \} \tag{4.4}$$

を考える．ここに，$(1 - a_p) + a_p/n_p$ はプロセス並列による加速分，$(1 - a_t) + a_t/n_t$ はスレッド並列による加速分をあらわす．上記の a_p 及び a_t を式(4.3)(4.4)に代入して，性能向上比を推定したのが表 4.1(b)である．表 4.1(c)は，表 4.1(a)の実測値と表 4.1(b)の推定値をもとに，推定値/実測値を示したものである．塗りつぶした欄の値を除きほとんどの値が 1 に近いことから，このコード P1 の場合は，拡張アムダールの法則(4.3)(4.4)により，性能推定が可能であることがわかる．なお，高並列で誤差が大きくなるのは，このコード P1 の場合，データアクセスがオンキャッシュになり，並列数以上に性能が出てしまっているからである．

一方，コード P3 の場合の結果を表 4.2 に示す．この場合，平均の a_p = 0.916，平均の a_t = 0.915 である．表 4.2(c)によれば，塗りつぶした欄の値のように，プロセス数×スレッド数の大きいところで推定の誤差が大きくなっている．

表 4.1　拡張アムダールの法則の検証（コード P1）

(a)　性能向上比の実測値（数字は 4 プロセス×1 スレッドの値を基準）

		プロセス比				
		1	2	4	8	16
スレッド比	1	1.00	2.11	4.23	8.83	17.40
	2	2.00	4.21	8.14	17.42	34.06
	4	3.93	8.30	15.97	34.37	-
	8	7.60	16.06	28.61	67.15	-
	16	14.04	29.14	54.75	104.56	-

(b)　性能向上比の推定値

		プロセス比				
		1	2	4	8	16
スレッド比	1	1.00	2.04	4.28	9.44	23.75
	2	1.99	4.06	8.50	18.75	47.20
	4	3.93	8.03	16.80	37.04	93.22
	8	7.66	15.66	32.78	72.29	181.92
	16	14.61	29.83	62.53	137.90	347.01

(c)　推定値と実測値の比較（推定値/実測値）

		プロセス比				
		1	2	4	8	16
スレッド比	1	1.00	0.97	1.01	1.08	1.36
	2	1.00	0.97	1.04	1.08	1.36
	4	1.00	0.97	1.05	1.08	-
	8	1.01	0.98	1.15	1.08	-
	16	1.04	1.03	1.14	1.32	-

表 4.2　拡張アムダールの法則の検証（コード P3）

(a)　性能向上比の実測値（数字は 28 プロセス×1 スレッドの値を基準）

		プロセス比				
		1	2	4	8	16
スレッド比	1	1.00	1.90	3.21	4.43	-
	2	1.85	3.35	5.14	5.98	-
	4	3.25	5.41	7.19	-	-
	8	4.94	7.32	-	-	-
	16	6.63	-	-	-	-

(b)　性能向上比の推定値

		プロセス比				
		1	2	4	8	16
スレッド比	1	1.00	1.85	3.20	5.04	7.09
	2	1.84	3.40	5.90	9.30	13.08
	4	3.19	5.89	10.20	16.09	22.63
	8	5.02	9.27	16.06	25.34	35.63
	16	7.05	13.01	22.53	45.55	50.00

(c)　推定値と実測値の比較（推定値/実測値）

		プロセス比				
		1	2	4	8	16
スレッド比	1	1.00	0.97	1.00	1.14	-
	2	1.00	1.02	1.15	1.55	-
	4	0.98	1.09	1.42	-	-
	8	1.02	1.27	-	-	-
	16	1.06	-	-	-	-

表 4.3　コード P3 における通信コストの影響（並列化率は 28 プロセスを基準に算出）

	プロセス数			
	28	56	112	224
実測性能向上比	1.00	1.90	3.21	4.43
並列化率	-	0.946	0.916	0.885
経過時間（秒）	464.87	247.66	149.69	110.29
通信時間（秒）	31.65	28.78	34.76	46.14

表 4.4　高精度アムダールの法則によるプロセス性能推定（コード P3）

	プロセス数						
	28	56	112	224	448	896	1,792
実測性能向上比	1.00	1.90	3.21	4.43	-	-	-
(4.1)(4.2)による推定性能向上比	1.00	1.85	3.20	5.04	7.08	8.88	10.17
(4.5)(4.6)による推定性能向上比	1.00	1.84	3.11	4.43	4.81	3.86	2.47

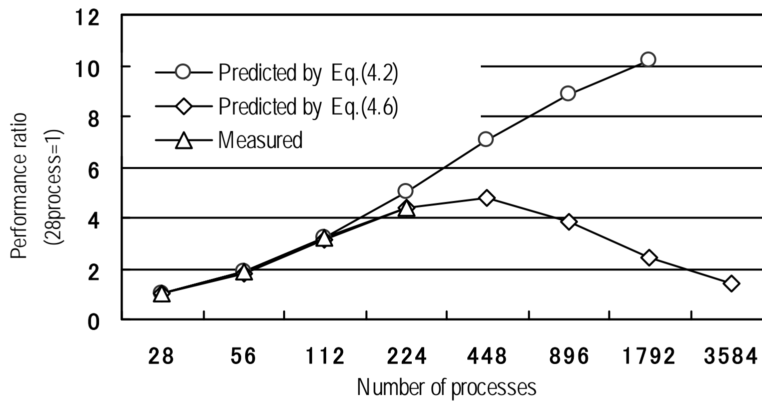


図 4.5　アムダールの法則によるプロセス性能推定の比較（コード P3）

4.3 ハイブリッド並列におけるアムダールの法則の拡張とその検証（その 2）

次に，コード P3 における性能推定誤差の原因とアムダールの法則の改良（高精度化）を考える．いま，プロセス数が変化したときの並列化率を調べてみると，表 4.3 上段のようになり，並列化率は一定ではなく，プロセス数の増加とともにかなり低下しているのがわかる．これは，プロセス数とともに増加する通信コスト等が存在するためである．プロファイラを使いコストの内訳を測定した結果，表 4.3 下段に示すようにプロセス数の増加に伴う通信コストの増大を実際に確認した．

そこで，アムダールの法則(4.1)(4.2)に対して，プロセス並列における通信コストの影響を考慮する高精度化を考える．いま，プロセスの並列化率 a_p，プロセス数 n_p に加えて，通信の影響として，プロセス数によらずコスト一定の通信の割合を c_t，袖転送のようなプロセス数にコストが比例する通信の割合を c_n とすると，アムダールの法則(4.1)(4.2)の自然な拡張として，

$$S(n) = T_{serial}/T_{parallel} \tag{4.5}$$

ただし，

$$T_{parallel} = T_{serial} \times \{(1 - a_p - c_t - c_n) + a_p/n_p + (c_t + c_n \times n_p)\} \tag{4.6}$$

という関係を考えることができる．コード P3 におけるそれぞれのコストをプロファイラによって調べてみると，28 プロセスの場合，$a_p = 0.925$，$c_t = 0.057$，$c_n = 0.005$，$1\text{-}a_p\text{-}c_t\text{-}c_n = 0.013$ となり，6%強の実際に無視できない通信コストが存在する*[1]．式(4.5)(4.6)により求めた推定性能向上比と実測値とを比較したのが表 4.4 であり，(4.1)(4.2)によるものと比べると，多プロセスにおける性能推定の精度は向上している．さらに，実測範囲以上の多プロセスの場合の性能向上比を推定してみると，表 4.4 や図 4.5 にあるように，896 プロセス以上では，通信コストの影響で性能向上比が低下する傾向が現れている．

[1] ちなみにこれはプロセス別の演算コストを合計した，いわば非並列状態での比率であり，プロセス数に応じた通信コストの比率は式(6)をもとに算出することができる．例えば,4 プロセス並列時の実行時間比は，$T_{parallel}/T_{serial} = 0.013 + 0.925/4 + 0.057 + 0.005 \times 4 = 0.321$ であり，それに含まれる通信の影響は，$c_t + c_n \times n_p = 0.057 + 0.005 \times 4 = 0.077$ であるから，通信コストの比率は，$0.077/0.321 \approx 0.24$ となり，これは図 4 で示した特性図の実測値とも概ね一致する．

この結果を利用して，ハイブリッド並列における拡張アムダールの法則を高精度化してみると，通信の部分はスレッド並列による加速の影響は受けないことから，

$$S(n) = T_{serial}/T_{hybrid} \quad (4.7)$$

ただし，$T_{hybrid} = T_{serial} \times [\{(1 - a_p - c_t - c_n) + a_p/n_p\} \times \{(1 - a_t) + a_t/n_t\} + (c_t + c_n \times n_p)] \quad (4.8)$

とすることができる．表 4.6 は，コード P3 に対して式(4.7)(4.8)による推定値と実測値を比較したものであるが，両者はすべてのレンジでよく合致しており，表 4.2(c)と比べても高プロセス×高スレッドの場合の推定精度は改善されているのがわかる．表 4.7 は，表 4.6(a)を CPU 数一定で整理したものである．図 4.2(a)で示した特定のプロセス数×スレッド数で性能が高くなる挙動がよく再現されており，高精度拡張アムダールの法則(4.7)(4.8)が通信の多いコード P3 の場合の性能推定に有効であることを示している．

表 4.6　高精度拡張アムダールの法則の検証（コード P3）

(a)　性能向上比の推定値

		プロセス比				
		1	2	4	8	16
スレッド比	1	1.00	1.84	3.11	4.43	4.81
	2	1.84	3.22	4.94	6.14	5.77
	4	3.18	5.12	7.00	7.60	6.41
	8	4.99	7.29	8.84	8.62	6.78
	16	6.97	9.23	10.18	9.24	6.99

(b)　推定値と実測値の比較（推定値/実測値）

		プロセス比				
		1	2	4	8	16
スレッド比	1	1.00	0.97	0.97	1.00	-
	2	1.00	0.96	0.96	1.03	-
	4	0.98	0.95	0.97	-	-
	8	1.01	1.00	-	-	-
	16	1.05	-	-	-	-

表 4.7　高精度拡張アムダールの法則による性能推定（コード P3）

		CPU 数（プロセス数×スレッド数）			
		112	224	448	896
スレッド数	1	3.11	4.43	4.81	3.86
	2	3.22	4.94	6.14	5.77
	4	3.18	5.12	7.00	7.60
	8	2.99	4.99	7.29	8.84
	16	2.63	4.52	6.97	9.23

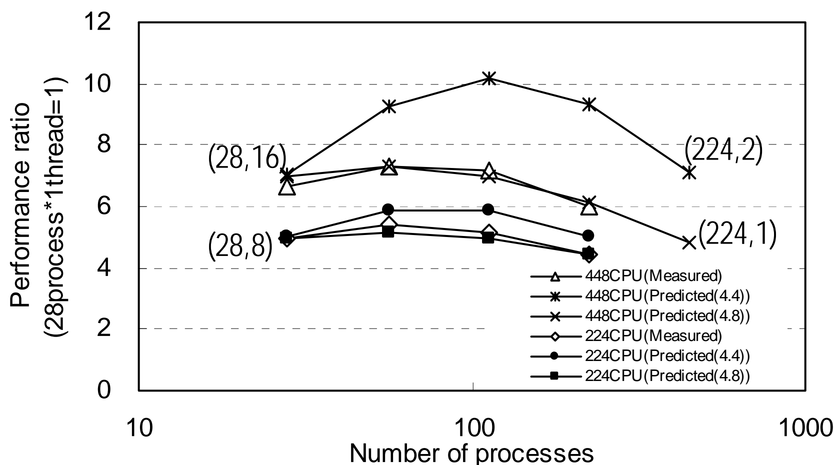


図 4.8　アムダールの法則によるコード P3 の性能推定の比較

5. まとめ

本報告では，JAXA の並列 CFD コードの中で，メモリアクセス負荷及び転送負荷が比較的重い部類の「並行平板間乱流解析コード」を取り上げ，スレッド並列の性能チューニングの概要と性能測定結果を示すとともに，コードの特性と並列性能の関係について論じた．また，アムダールの法則を拡張したハイブリッド並列における簡易な性能推定法を提示しその推定精度を検証した．本コードのように通信量が多い場合でも，拡張されたアムダールの法則で性能向上比の推定がある程度の精度で可能であることがわかった．ここで示した性能推定法は，特殊なパラメータを引用しているわけではないので，JAXA の並列システムに限らず，一般の並列システムに適用できるものであると考えられる．

しかし，今回実施したような系統的性能評価は一般には困難と考えられるから，拡張アムダールの法則(4.3)(4.4)や(4.7)(4.8)の基本になっている並列化率や通信コストを如何に簡便に見積もるかがこの推定法の鍵でありそれがまた今後の課題でもある．例えば，コード P1 のような通信量が少なく線形の性能の場合には，プロセス数×スレッド数の組み合わせとして，16×1，1×16 のように，2 ケースでプロセス並列化率とスレッド並列化率を採取すれば，(4.3)(4.4)により精度良い性能推定が可能である．しかし，本報告で扱ったコード P3 のように通信量が多い場合には，(4.7)(4.8)を使う必要があり，しかもその場合には，プロセス数に比例するかどうか等の通信の中身まで把握する必要がある．通信量の全体はプロファイラで知ることができるが，通信の中身を簡単に測る方法については今後の課題と考えている．

謝辞

今回のチューニング，性能測定に際しては，富士通の多大なるご協力をいただきました．ここに記して謝意を表します．

参考文献

1) Matsuo, Y., Tsuchiya, M., Aoki, M., Sueyasu, N., Inari, T. and Yazawa, K.: Early Experience with Aerospace CFD at JAXA on the Fujitsu PRIMEPOWER HPC2500, *Proc. SC'04*, Pittsburgh, USA (Nov. 2004).
2) 溝渕泰寛，新城淳史，小川哲：CeNSSを用いた水素噴流浮き上がり火炎詳細シミュレーション，航空宇宙数値シミュレーション技術シンポジウム2004論文集，JAXA特別資料SP-04-012, pp.202-207 (Mar. 2005).
3) 松尾裕一：差分法による翼まわり流れのLES，第12回数値流体力学シンポジウム講演論文集，pp.153-154 (Dec. 1998).
4) 阿部浩幸，松尾裕一：平行平板間乱流の大規模直接数値シミュレーション，航空宇宙数値シミュレーション技術シンポジウム2004論文集，JAXA特別資料SP-04-012, pp.21-26 (Mar. 2005).
5) 近藤夏樹，青山剛史，齊藤茂：重合格子法を用いたロータ/胴体干渉の計算，航空宇宙数値シミュレーション技術シンポジウム2003論文集，JAXA特別資料SP-03-002, pp.232-237 (Mar. 2004).
6) Takaki, R., Yamamoto, K., Yamane, T., Enomoto, S. and Mukai, J.: The Development of the UPACS CFD Environment, *High Performance Computing, Proc. ISHPC2003*, LNCS 2858, pp.307-319 (Oct. 2003).
7) 村山光宏，山本一臣：非構造格子法を用いた航空機高揚力装置周りの流れ場解析の精度検証，航空宇宙数値シミュレーション技術シンポジウム2004論文集，JAXA特別資料SP-04-012, pp.82-86 (Mar. 2005).
8) Cappelo, F. and Etiemble, D.: MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks, *Proc. SC'00*, Dallas, USA (Nov. 2000).

2.2 三次元圧縮性流体解析プログラム UPACS の性能評価

宇宙航空研究開発機構
高木　亮治

1. はじめに

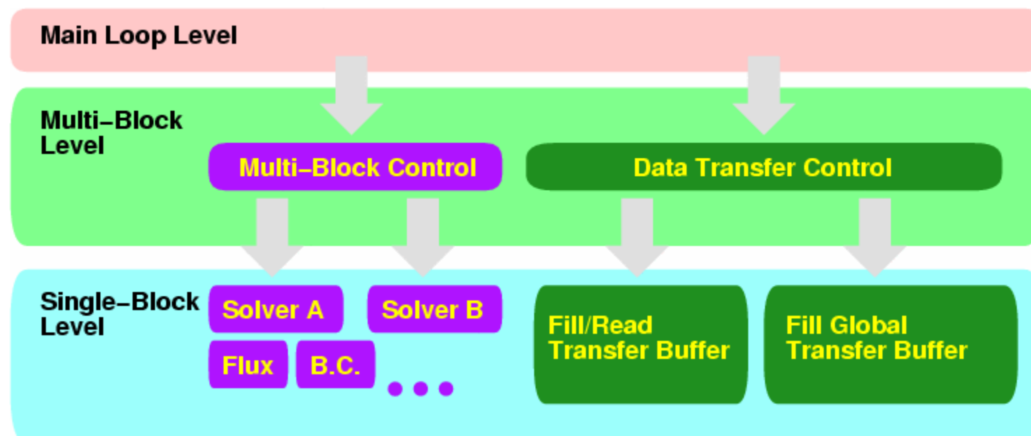
宇宙航空研究開発機構（JAXA）で開発された CFD プログラム UPACS について、富士通 PRIMEPOWER HPC2500 上で性能評価を行ったのでその結果を報告する。

2. プログラム概要

UPACS は中核となる解析ソルバである UPACS ソルバと、解析の前後処理を行う各種ツール、ユーティリティ群からなる CFD 共通基盤環境である。UPACS の特徴として A)拡張性と共有性、B)並列化、等が挙げられる。

A)　拡張性と共有性

（ア）オブジェクト指向の考え方を取り入れることで、データ、手続きのカプセル化とプログラム構造の階層化を行った。特にプログラムを三階層として、本来別の処理であるシングルブロックの解析ソルバ部とマルチブロック/オーバーセット処理部および並列処理部を分離した。下記に UPACS の階層構造を示す。最下部が単一ブロックの解析ソルバ、中間にマルチブロック/オーバーセット処理を実施する部分、最上部にプログラムの流れを制御する部分となっている。この結果解析ソルバの開発者は並列処理やマルチブロック/オーバーセット処理を考慮する必要がなく、それぞれの専門家による分散した開発が可能となった。



UPACS の階層構造

（イ）CFD 研究者による共有化とカプセル化、コードの階層化を実現するため、UPACS は Fortran90 を用いて開発された。C++など計算科学の新しい道具であるオブジェクト指向型の言語は非常に便利ではあるが、これまでの資産の継承、CFD 研究者の習熟度、更には大型計算機での実行性能と開発環境の実績を考慮すると、C++を用いて開発するのは時期尚早と判断した。一方伝統的な科学技術用開発言語である Fortran にも Fortran90 になって構造体、ポインタ等、我々の目的を実現するための機能が導入されており、開発言語として Fortran90 を選定した。

B)　並列化

（ア）複雑形状への適用性と解析精度の維持のバランスを保つためマルチブロック/オーバセット構造格子法を採用している。そのためマルチブロック/オーバセット構造格子の複数ブロックを並列化の際の領域分割にマッピングすることで並列化を行っている。複数個のブロックが並列処理単位に自由にマッピングできるため、任意の並列度数での解析が簡単に実行できる。

（イ）並列化は MPI を用いたプロセス並列を採用した。並列化には他にも VPP-Fortran、XPFortran を用いたプロセス並列、OpenMP によるスレッド並列などがあるが、並列化されたプログラム

の汎用性（移植性）を重視してMPIによる並列化を行った。MPIはPCクラスタから大型計算機まで並列計算機なら一般的に利用可能な並列環境であり、移植性を考えると非常に有望である。

	従来のCFDコード	UPACS
開発言語	Fortran77	Fortran90 構造体、ポインタ配列…
並列化	データ、手続き並列 VPP-Fortran(XPFortran)	明示的な領域分割 MPI
格子	単一構造格子	複合格子（非構造格子）
行列反転	1行列の反転を並列化 （ADI等を用いた巨大なシングルブロックでの行列反転）	1行列の反転は非並列 （マルチブロックでの並列化）
時間積分	定常解析が主	今後は非定常解析が主
データ転送	行列の転置などでAlltoAllが必要	ブロック間の陽的なデータ転送で良い

今回の性能評価ではUPACS ver.1.3を使用した。

3. 基本性能

基本的な性能としてMPI並列とOpenMP並列の組合わせによる、SpeedUp（計算量一定で並列数可変）性能計測を行った。その結果プロセスに比べてスレッドの並列効果が低く、同じCPU数ならHybridよりPureMPIの効率が良いことがわかった。

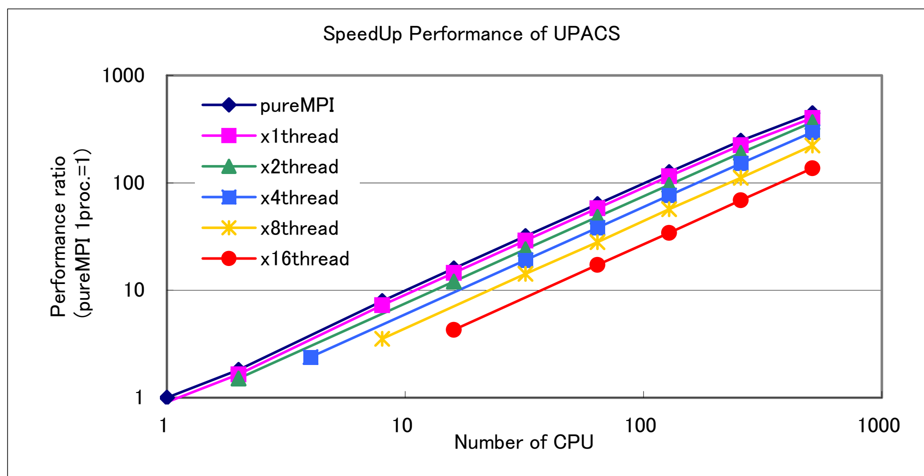
【測定条件】

実行環境	機種: 富士通 PRIMEPOWER HPC2500（SPARC64V 1.3GHz） 使用規模: 32cpu×32node 開発環境: Parallelnavi2.4（Fujitsu Fortran Compiler V5.6）
並列数	[PureMPI] MPI並列のみで1～512プロセスを使用 [Hybrid] MPI並列1～512プロセスにOpenMP並列1～16スレッドを併用
計算格子	1ブロックあたり40×20×80:計512ブロック 実行時に各プロセス均等に分散
計算反復回数	2回
翻訳時オプション	[PureMPI] mpifrt -Kfast_GP2=3,V9,largepage=2,hardbarrier -x- [Hybrid] mpifrt -Kfast_GP2=3,V9,largepage=2,OMP,hardbarrier -x-

【測定結果】

経過時間[秒]	Process数									
	1	2	4	8	16	32	64	128	256	512
PureMPI	1238.4	682.9	-	156.0	77.6	38.9	19.5	9.8	5.0	2.8
×1thread	1360.9	749.4	-	170.5	85.3	42.6	21.3	10.8	5.5	3.1
×2thread	820.7	-	-	102.9	51.3	25.7	13.0	6.5	3.4	-
×4thread	517.3	-	-	65.0	32.3	16.3	8.2	4.2	-	-
×8thread	349.8	-	87.0	44.0	21.9	11.1	5.6	-	-	-
×16thread	288.4	-	71.7	36.3	18.0	9.1	-	-	-	-

SpeedUp (MPI 1proc.=1)	Process数									
	1	2	4	8	16	32	64	128	256	512
PureMPI	1.00	1.81	-	7.94	15.95	31.85	63.53	126.38	247.47	444.99
×1thread	0.91	1.65	-	7.26	14.52	29.05	58.03	114.99	223.77	401.19
×2thread	1.51	-	-	12.03	24.14	48.20	95.63	190.23	367.85	-
×4thread	2.39	-	-	19.05	38.31	76.20	151.23	297.83	-	-
×8thread	3.54	-	14.24	28.13	56.65	111.31	222.31	-	-	-
×16thread	4.29	-	17.28	34.13	68.77	136.32	-	-	-	-



次にプロファイラを用いて、8 プロセス実行の性能情報を採取した。　L2 キャッシュミス率や TLB ミス率が高いルーチンが多く、演算性能(MFLOPS)の阻害要因となっていることが判明した。

【測定条件】

実行環境	機種： 富士通 PRIMEPOWER HPC2500（SPARC64V 1.3GHz） 使用規模： 64cpu×3node 開発環境： Parallelnavi2.4（Fujitsu Fortran Compiler V5.6）
並列数	MPI 並列 8 プロセス
計算格子	1 ブロックあたり 100×100×100：計 8 ブロック 実行時に各プロセス均等に分散
計算反復回数	5 回
翻訳時オプション	mpifrt -Kfast_GP2=3,V9,largepage=2,hardbarrier -x-

【測定結果】

①プロセス単位

CPU (Sec)	MIPS	MFLOPS	L2miss (%)	TLBmiss (%)	Cover (%)	
181.3	667.8	147.9	1.0	0.6	68.5	Process 0
179.1	676.8	149.7	1.0	0.6	69.4	Process 1
177.7	682.0	150.9	1.0	0.6	68.8	Process 2
180.2	671.1	148.7	1.0	0.6	69.8	Process 3
178.6	682.1	150.6	1.0	0.6	69.1	Process 4
178.3	682.0	150.9	1.0	0.6	69.2	Process 5
180.3	672.9	149.2	1.0	0.6	69.8	Process 6
179.0	680.2	150.3	1.0	0.6	69.3	Process 7
272.7	3560.3	787.8	1.0	0.6	69.2	Total

②ルーチン単位

Cost (%)	MIPS	MFLOPS	L2miss (%)	TLBmiss (%)	Cover (%)	ルーチン名
30.6	478.8	152.3	0.4	3.0	42.0	blk_mfgs.implhs_mfgs_
15.3	773.9	217.4	0.6	0.9	59.3	blk_rhsviscous.cellfacevariables_
6.8	127.3	13.4	13.0	0.0	99.0	blk_rhsconvect.rhs_convect_
5.1	925.0	295.6	0.3	0.0	99.1	blk_flux.flux_roe_
4.4	946.4	216.5	0.5	0.0	99.1	blk_muscl.muscl_co_
4.0	809.6	44.5	0.2	1.0	56.4	blk_metrics.calcmetrics_
3.7	161.7	24.4	14.3	0.0	99.1	blk_rhsviscous.rhs_viscous_
3.4	393.4	103.3	1.4	0.0	98.8	blk_rhsviscous.flux_vis_
2.9	769.6	191.1	0.6	2.2	37.9	blk_dt.calcdt_original_
2.9	898.4	178.5	0.5	0.0	99.1	blk_tm_spalartallmaras.muscl_2ndorder_
2.4	1639.7	333.6	0.8	0.0	99.0	blk_tm_spalartallmaras.diffusion_
1.8	1743.1	145.7	0.2	0.0	98.4	jwe_gdgemm
1.6	147.9	4.8	7.9	0.0	99.0	blk_muscl.muscl_
1.5	607.1	159.1	2.7	0.0	99.0	blk_tm_spalartallmaras.convection_ausm_
1.4	926.6	0.0	0.3	1.8	38.2	blk_metrics.calccellvrtx_

4. スカラチューニング

単体性能向上のため、データアクセスの効率化を促進するスカラチューニングを実施した。

4.1 チューニング概要

オリジナルソースに対して、以下の性能チューニングを段階的に適用した。

項目名	内容
Tune1	・配列の軸入替え
Tune2	・サブルーチンにまたがるループ融合＋ワーク配列の次元削減

【Tune1 － 配列の軸入替え】

TLB ミス率の高いルーチンでは､配列の最外次元が変化するデータアクセスが多用されており、ストライド幅が大きくなっていた。そこで、最内次元に入れ替えることにより、データアクセスを連続化した。

ソース変更前

```
subroutine implhs_mfgs(blk,sweepID,cdt,cdiag)
  !
  type(blockDataType),intent(inout) :: blk
 ........
  real(8), pointer, dimension(:,:,:,:) :: dq_star
  allocate(dq_star(0:blk%in+1,0:blk%jn+1, &
                0:blk%kn+1,bdtv_nFlowVar))
  dq_star(:,:,:,:) = 0.0
 ........
    do k=is(3),ie(3),istep(3)
      do j=is(2),ie(2),istep(2)
        do i=is(1),ie(1),istep(1)
          rho = blk%q(i,j,k,1)
          rhoi = 1.d0/(rho+epsilon(rho))
          u(:) = blk%q(i,j,k,2:4)*rhoi
         ........
          nv(:)= blk%fNormal   (i-1,j,k,1,:)
          nt    = blk%fNormal_t(i-1,j,k,1)
          q(:) = q0(:)+dq_star(i-1,j,k,:)
         ........
          nv(:)= blk%fNormal   (i,j-1,k,2,:)
          nt    = blk%fNormal_t(i,j-1,k,2)
          q(:) = q0(:)+dq_star(i,j-1,k,:)
         ........
        enddo
      enddo
    enddo
```

ソース変更後

```
subroutine implhs_mfgs(blk,sweepID,cdt,cdiag)
  !
  type(blockDataType),intent(inout) :: blk
 ........
  real(8), pointer, dimension(:,:,:,:) :: dq_star
  allocate(dq_star(bdtv_nFlowVar,0:blk%in+1, &
                0:blk%jn+1,0:blk%kn+1))
  dq_star(:,:,:,:) = 0.0
 ........
    do k=is(3),ie(3),istep(3)
      do j=is(2),ie(2),istep(2)
        do i=is(1),ie(1),istep(1)
          rho = blk%q(1,i,j,k)
          rhoi = 1.d0/(rho+epsilon(rho))
          u(:) = blk%q(2:4,i,j,k)*rhoi
         ........
          nv(:)= blk%fNormal   (:,i-1,j,k,1)
          nt    = blk%fNormal_t(1,i-1,j,k)
          q(:) = q0(:)+dq_star(:,i-1,j,k)
         ........
          nv(:)= blk%fNormal   (:,i,j-1,k,2)
          nt    = blk%fNormal_t(2,i,j-1,k)
          q(:) = q0(:)+dq_star(:,i,j-1,k)
         ........
        enddo
      enddo
    enddo
```

実際の入替えは、ソース変更の代わりに以下の C プリプロセッサマクロを使用して行った。

common.f90inc

```
#if !defined(common_f90inc)
#define common_f90inc

#define q(i,j,k,n)   Q(n,i,j,k)
#define fNormal_t(i,j,k,n)   FNORMAL_T(n,i,j,k)
#define fNormal(i,j,k,n,A)   FNORMAL(A,i,j,k,n)
#define dq_star(i,j,k,n)   DQ_STAR(n,i,j,k)
・・・・

#endif /* !defined(common_f90inc) */
```

【Tune2 – サブルーチンにまたがるループ融合＋ワーク配列の次元削減】

L2 キャッシュミス率の高いルーチンでは、ソースが複雑なため自動的にループ融合できない箇所があった。そこで、ループ融合した状態にソースを書換えた。

ソース変更前

```
subroutine rhs_viscous(blk)
  type(visCellFaceType), pointer, dimension(:,:,:) :: cface
  ・・・
    if(bv_viscous%fullns) then
      call cellFaceVariables(blk,cface,dir)
    else if(bv_viscous%thinlayer) then
      call cellFaceVars_thinlayer(blk,cface,dir)
    else
      write(6,*) ' error: Unknown viscous term model '
      write(6,*) ' rhs_viscous '
    end if

    call flux_vis(blk,cface,dir)
    call blk_saveBoundaryFlux_viscous(blk,cface,dir)
  ・・・
end subroutine rhs_viscous
```

```
subroutine cellFaceVariables(blk,f,dir)
 ・・・
  do k = isrt(3),iend(3)
    do j = isrt(2),iend(2)
      do i = isrt(1),iend(1)
     ・・・
        f(i,j,k)%nv     = blk%fNormal(i,j,k,ixi,:)
        f(i,j,k)%area   = blk%fArea   (i,j,k,   ixi)

      end do
    end do
  end do

end subroutine cellFaceVariables
```

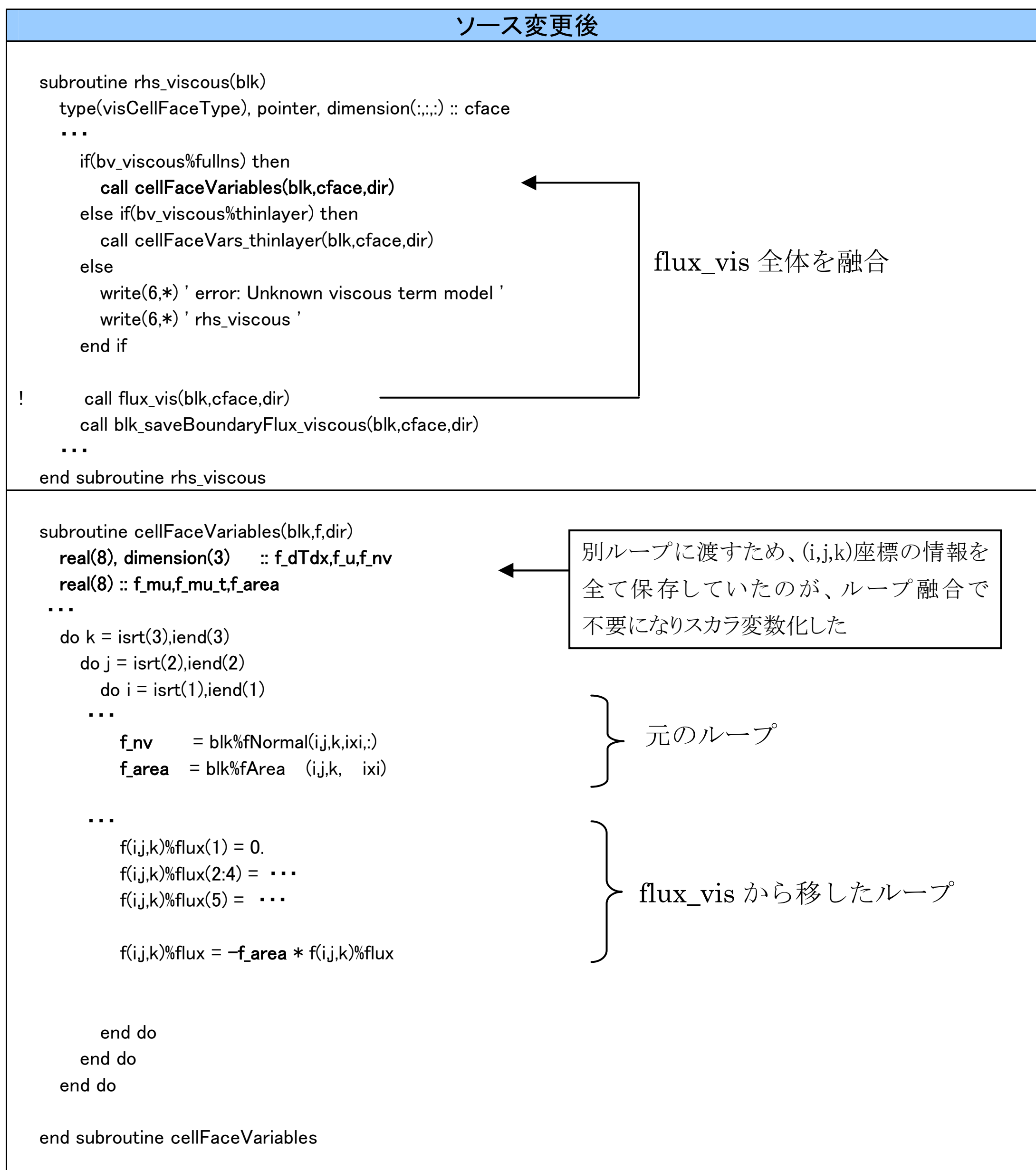
```
subroutine flux_vis(blk,f,dir)
 ・・・
  do k = isrt(3),iend(3)
    do j = isrt(2),iend(2)
      do i = isrt(1),iend(1)
     ・・・
        f(i,j,k)%flux(1) = 0.
        f(i,j,k)%flux(2:4) = ・・・
        f(i,j,k)%flux(5) = ・・・

        f(i,j,k)%flux = -f(i,j,k)%area * f(i,j,k)%flux

      end do
    end do
  end do

end subroutine flux_vis
```

また、このループ融合により大きな領域を取る必要がなくなった作業配列については、配列次元数を削減した状態にソースを書換えた。



ソース変更後

```
subroutine rhs_viscous(blk)
  type(visCellFaceType), pointer, dimension(:,:,:) :: cface
  ・・・
    if(bv_viscous%fullns) then
      call cellFaceVariables(blk,cface,dir)
    else if(bv_viscous%thinlayer) then
      call cellFaceVars_thinlayer(blk,cface,dir)
    else
      write(6,*) ' error: Unknown viscous term model '
      write(6,*) ' rhs_viscous '
    end if

!        call flux_vis(blk,cface,dir)
    call blk_saveBoundaryFlux_viscous(blk,cface,dir)
  ・・・
end subroutine rhs_viscous
```

```
subroutine cellFaceVariables(blk,f,dir)
  real(8), dimension(3)     :: f_dTdx,f_u,f_nv
  real(8) :: f_mu,f_mu_t,f_area
 ・・・
  do k = isrt(3),iend(3)
    do j = isrt(2),iend(2)
      do i = isrt(1),iend(1)
     ・・・
        f_nv       = blk%fNormal(i,j,k,ixi,:)
        f_area    = blk%fArea    (i,j,k,    ixi)

     ・・・
        f(i,j,k)%flux(1) = 0.
        f(i,j,k)%flux(2:4) =  ・・・
        f(i,j,k)%flux(5) =   ・・・

        f(i,j,k)%flux = -f_area * f(i,j,k)%flux


      end do
    end do
  end do

end subroutine cellFaceVariables
```

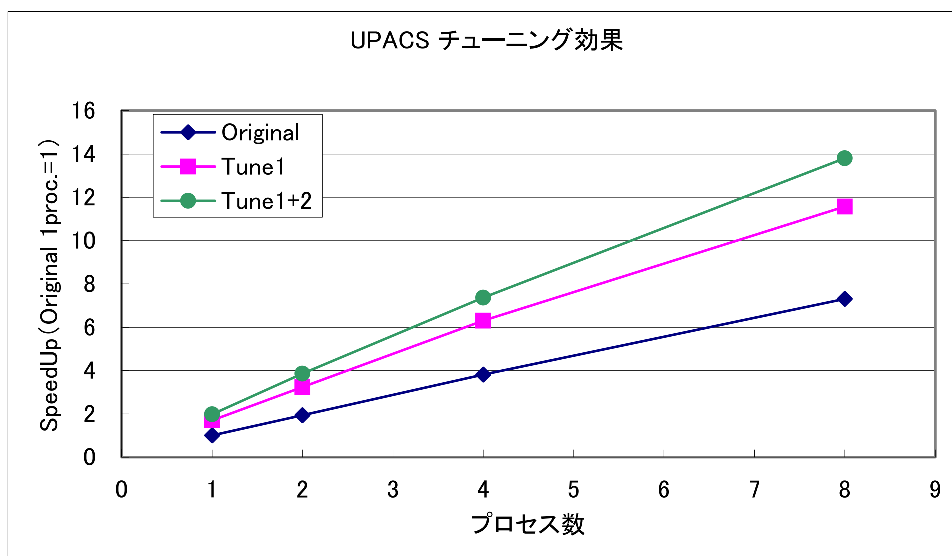

4.2 性能測定

「３．基本性能」と同じ測定条件で、以下の３パターンの性能を測定した。

パターン名	内容
Original	オリジナルソース
Tune1	Original に Tune1 を適用したソース
Tune1+2	Tune1 に Tune2 を適用したソース

【測定結果】

MPI プロセス数	実行時間[秒]			SpeedUp(Original 1proc.=1)		
	Original	Tune1	Tune1+2	Original	Tune1	Tune1+2
1	2048.4	1211.8	1037.7	1.00	1.69	1.97
2	1059.6	635.1	532.6	1.93	3.23	3.85
4	538.8	325.4	278.1	3.80	6.30	7.37
8	280.5	177.1	148.5	7.30	11.57	13.80



2段階のスカラチューニングにより、オリジナルソースから約2倍の性能向上が得られた。測定パターンごとに、8プロセス実行のプロファイラ情報を採取した結果、L2キャッシュミス率やTLBミス率の改善に応じて、全体の演算性能も改善されていることがわかった。チューニング後もL2キャッシュミス率の高い箇所がいくつか残っているが、これらの中にはプログラム構造が複雑なため有効なループ融合が出来なかったルーチンも含まれている。強制的に融合するにはアルゴリズムの変更が必要なため、今回は対象外とした。さらに、プロファイラを用いて以下の詳細情報を計測した。

【コスト比率】実行時間ベースのコスト分布と､その中を占めるメモリアクセス時間(MEM)およびそれ以外の命令処理時間(CPU)の比率
【命令数比率】発行命令数における、以下の命令の割合
Load/Store命令(Ld/St),浮動少数点演算命令(Float),プリフェッチ命令(Pref),分岐命令(Branch), その他命令(Other)
【実効性能】命令数情報とコスト情報から算出した、MIPS値およびMFlops値の情報

	コスト比率			命令数比率					実効性能	
		MEM	CPU	Ld/St	Float	Pref	Branch	Other	MIPS	MFlops
全体	100.0%	39%	61%	42.9%	22.4%	1.8%	5.0%	27.9%	937.8	210.9

コスト比率ではCPU時間(61%)がメモリアクセス時間(39%)に比べて高いのに対し、命令数比率ではFloatの割合(22.4%)が少なかった。ポインタや構造体のアドレス計算など、その他の命令数の割合が多いため、MFlopsが向上しないと考えられる。

5. 自動並列化

自動並列化オプションを追加して翻訳した場合の並列化状況を調査した。また、ソース解析能力を比較するため、ベクトル機での自動ベクトル化状況も併せて調査した。

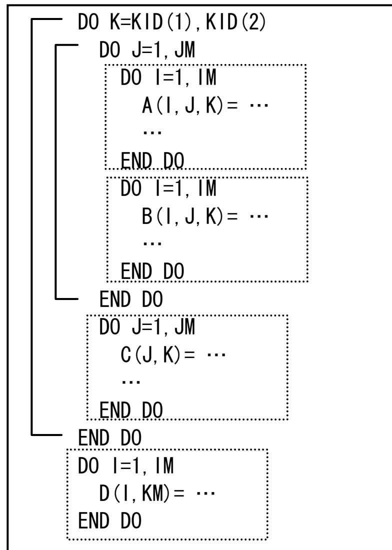
	自動並列化	自動ベクトル化
機種	富士通 PRIMEPOWER HPC2500 (SPARC64V 1.3GHz)	富士通 VPP5000
言語環境	Parallelnavi2.4 (Fujitsu Fortran Compiler V5.6)	UXP/V Fortran V20L20
翻訳時 option	mpifrt -Kfast_GP2=3,V9,largepage=2,hardbarrier -x- -Kparallel,reduction	-Pa -Wv,-m3
使用 program	UPACS (前回報告の Tune1+2 版) ※自動並列化や自動ベクトル化を促進するための追加変更は行っていない。	

5.1 調査方法

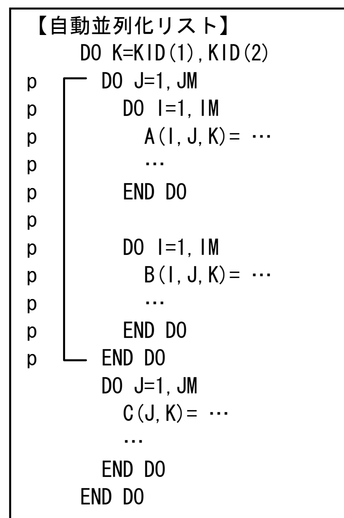
並列化/ベクトル化状況を調査するにあたり、コンパイラが出力するメッセージ数を単純にカウントする方法だけでは、以下の問題が考えられる。

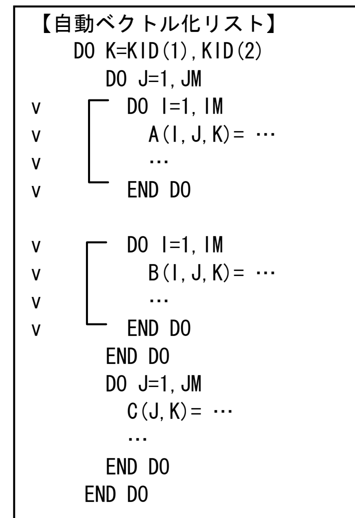
- 並列化の規模が判りにくい（外側の大きなループでも内側の小さなループでもカウント数は同じ）
- 並列化とベクトル化の比較が難しい（並列化は外側から、ベクトル化は内側からの解析で軸が異なる場合がある）

そこで、軸になった DO ループそのものではなく、階層構造の末端にある最内ループ（左下リストの点線範囲）が並列化あるいはベクトル化されたかどうかをカウント対象とした。



自動並列化/自動ベクトル化のコンパイルリストをそれぞれ出力し、並列化やベクトル化の軸の内部に含まれる最内ループ数をカウントする。例えば下記リストの場合、自動並列化と自動ベクトル化で軸になる DO ループは異なるが、最内ループのカウント数はどちらも２となる。





5.2 調査結果

(a) 全体情報

前回測定した、8 プロセス並列（スレッド並列無し）実行コストの上位ルーチンを対象に集計したところ、以下のように、自動並列化/自動ベクトル化ともに最内ループ数はゼロとなった。

サブルーチン名	HPC2500 8 プロセス 実行コスト	サブルーチン内 最内ループ数	HPC2500 自動並列化 最内ループ数	VPP5000 自動ベクトル化 最内ループ数
blk_mfgs.implhs_mfgs_	14.0%	2	0	0
blk_rhsviscous.cellfacevariables_	10.8%	1	0	0
blk_muscl.muscl_co_	10.2%	1	0	0
blk_flux.flux_roe_	10.1%	1	0	0
blk_tm_spalartallmaras.muscl_2ndorder_	7.8%	2	0	0
blk_tm_spalartallmaras.diffusion_	5.4%	2	0	0
blk_rhsconvect.rhs_convect_	5.1%	2	0	0
blk_muscl.minmod_co_	2.4%	0	0	0
blk_rhsviscous.rhs_viscous_	2.3%	2	0	0
blk_metrics.calcmetrics_	1.6%	2	0	0
top_timeint.implicit_onestep_	1.5%	0	0	0
blk_tm_spalartallmaras.production_destruction_	1.4%	1	0	0
blk_tm_spalartallmaras.lhs_gaussseidel_	1.4%	3	0	0
blk_tm_spalartallmaras.vanalbada_	1.3%	0	0	0
blk_tm_scalar_measure.vorticity_	1.3%	1	0	0
blk_dt.calcdt_original_	0.9%	1	0	0
上位ルーチン合計	77.6%	21	0	0

以下、UPACS のコスト上位５ルーチンについて、ループ構造と並列化阻害要因を調べた。代表例としてサブルーチン blk_mfgs.implhs_mfgs_のコンパイルリストから抜粋したループ構造とメッセージ情報を以下に示す。下線部の DO ループは、いずれもループ内部のポインタ引用が自動並列化の制約となっている。

blk_mfgs.implhs_mfgs_:コンパイルリスト（抜粋）

```
 20                 subroutine implhs_mfgs(blk,sweepID,cdt,cdiag)
 21                   !       Matrix Free Gauss-Seidel (MFGS) method by E. Shima (KHI)          *
 22                   !
 23                   type(blockDataType),intent(inout) :: blk
 24                   integer,intent(in) :: sweepID
 25                   real(8),intent(in) :: cdt,cdiag
 26
 27                   real(8), pointer, dimension(:,:,:,:) :: dq_star
 :
 54                   allocate(dq_star(0:blk%in+1,0:blk%jn+1,0:blk%kn+1,bdtv_nFlowVar))
 55         p u       dq_star(:,:,:,:) = 0.0
 56                   imax(1)=blk% in ; imax(2)=blk% jn ; imax(3)=blk% kn
 57
 58                   n = sweepID
 59    1              do ifb = 1,2
 :
 72    2                do k=is(3),ie(3),istep(3)
 73    3                  do j=is(2),ie(2),istep(2)
 74    4                    do i=is(1),ie(1),istep(1)
 75    4                      rho = blk%q(i,j,k,1)
 76    4                      rhoi = 1.d0/(rho+epsilon(rho))
 77    4      u               u(:) = blk%q(i,j,k,2:4)*rhoi
 78    4                      p     = blk%p(i,j,k)
 79    4                      c     = sqrt(abs(GAMMA*p*rhoi))
 80    4
 81    4      u               uu_ui = abs(dot_product(u(:),blk%fNormal(i   j   ,k   ,1,:)) + blk%fNormal_t(i   j   ,k   ,1))
 :
238    4      u               dq0(:) = dq_star(i,j,k,:)
239    4    p u              dq_star(i,j,k,:) = (dh*df(:)*blk%inv_vol(i,j,k) + blk%dq(i,j,k,:))*inv_diagonal
240    4
241    4      u               ddq(:) = dq_star(i,j,k,:) - dq0(:)
242    4                      if(abs(ddq(1)) > 1.D5) dq_star(i,j,k,1) = dq0(1)
243    4                      if(abs(ddq(2)) > 1.D5) dq_star(i,j,k,2) = dq0(2)
244    4                      if(abs(ddq(3)) > 1.D5) dq_star(i,j,k,3) = dq0(3)
245    4                      if(abs(ddq(4)) > 1.D5) dq_star(i,j,k,4) = dq0(4)
246    4                      if(abs(ddq(5)) > 1.D5) dq_star(i,j,k,5) = dq0(5)
247    4
248    4                    enddo
249    3                  enddo
250    2                enddo
251    1
252    1              end do
 :
267                   deallocate(dq_star)
268
269                 end subroutine implhs_mfgs
 :
Module subprogram name(implhs_mfgs)
 jwd5101i-i  "blk_mfgs.f90", line 59: DO ループ内に,自動並列化の制約となる文が存在します.
 jwd5101i-i  "blk_mfgs.f90", line 72: DO ループ内に,自動並列化の制約となる文が存在します.
 jwd5101i-i  "blk_mfgs.f90", line 73: DO ループ内に,自動並列化の制約となる文が存在します.
 jwd5101i-i  "blk_mfgs.f90", line 74: DO ループ内に,自動並列化の制約となる文が存在します.
```

ループ内部のポインタ引用が自動並列化の制約となっているほか、ユーザ定義の関数呼び出しを含んでいる場合、DO 変数がモジュール内のデータ実体である場合も阻害要因となっている。これらコスト上位５ルーチンに共通する、ループ内ポインタ引用の自動並列化について、現在のコンパイラの対応状況および回避方法は以下の通りである。

【機能改善について】

コンパイラでポインタの振る舞いを完全に解析することは不可能であり、汎用的な自動並列化は対応困難。もしポインタを使わなくても書ける処理内容であれば、後述の回避方法による改善の可能性がある。

【回避方法】

下記の 2 種類の方法がある。

1) ループ内のポインタ変数同士に領域の重なりが無い場合、ディレクティブ(!ocl noalias)あるいは翻訳時オプション(-Knoalias)で指示することにより、自動並列化が促進される場合がある（効果があるかどうかはプログラム依存）。

2) ソース修正により、配列ポインタを割付配列(allocatable)または形状明示配列（F77 の整合配列）などに置き換える。

そこで実際に、今回のソースについて、1)の翻訳時オプション（自動並列：-Knoalias，自動ベクトル：-Wv,-noalias）を追加して翻訳したところ、以下のように、自動並列化/自動ベクトル化ともに最内ループ数は増加した。

サブルーチン名	HPC2500 8 プロセス 実行コスト	サブルーチン内 最内ループ数	HPC2500 自動並列化 最内ループ数	VPP5000 自動ベクトル化 最内ループ数
blk_mfgs.implhs_mfgs_	14.0%	2	0 → 1	0 → 1
blk_rhsviscous.cellfacevariables_	10.8%	1	0	0
blk_muscl.muscl_co_	10.2%	1	0	0
blk_flux.flux_roe_	10.1%	1	0	0
blk_tm_spalartallmaras.muscl_2ndorder_	7.8%	2	0	0 → 1
blk_tm_spalartallmaras.diffusion_	5.4%	2	0	0
blk_rhsconvect.rhs_convect_	5.1%	2	0 → 1	0 → 1
blk_muscl.minmod_co_	2.4%	0	0	0
blk_rhsviscous.rhs_viscous_	2.3%	2	0 → 1	0 → 1
blk_metrics.calcmetrics_	1.6%	2	0	0
top_timeint.implicit_onestep_	1.5%	0	0	0
blk_tm_spalartallmaras.production_destruction_	1.4%	1	0	0 → 1
blk_tm_spalartallmaras.lhs_gaussseidel_	1.4%	3	0	0 → 3
blk_tm_spalartallmaras.vanalbada_	1.3%	0	0	0
blk_tm_scalar_measure.vorticity_	1.3%	1	0	0
blk_dt.calcdt_original_	0.9%	1	0	0
上位ルーチン合計	77.6%	21	0 → 3	0 → 8

ただし、新たに並列化/ベクトル化されたのは、比較的小規模のループであり、コスト比率の高い大規模ループには変化が無かった。

(b) ポインタ引用の変更

自動並列化の阻害要因と考えられるポインタ引用の書き換えを行なった。実行コスト上位ルーチンを対象に、ソース中で配列のポインタ引用が使われている箇所を、同じ動的割当て方式で最適化への制約が少ないと想定される、アロケータブル配列に書き換えた。単独で宣言されている配列の場合、下線部のように、宣言文の pointer 属性を、target + allocatable 属性に変更した。

書き換え前（配列ポインタ）	書き換え後（アロケータブル配列）

type(cellFaceType),dimension(:,:,:),**pointer**:: cface
integer :: ii,jj,kk
allocate(cface(-1:blk%in+1, -1:blk%jn+1, -1:blk%kn+1))
 	
do kk=-1,blk%kn+1
  do jj=-1,blk%jn+1
    do ii=-1,blk%in+1
      cface(ii,jj,kk)%area = 0.0
      cface(ii,jj,kk)%nt = 0.0
     ・・・
    enddo
  enddo
enddo
	type(cellFaceType),dimension(:,:,:),**target,allocatable**:: cface
	integer :: ii,jj,kk
	allocate(cface(-1:blk%in+1, -1:blk%jn+1, -1:blk%kn+1))
	
	do kk=-1,blk%kn+1
	  do jj=-1,blk%jn+1
	    do ii=-1,blk%in+1
	      cface(ii,jj,kk)%area = 0.0
	      cface(ii,jj,kk)%nt = 0.0
	     ・・・
	    enddo
	  enddo
	enddo

また構造型の成分として宣言されている配列の場合、Fortran の仕様により、構造型の成分には target 属性を指定できないため、allocatable 属性に変更した。

書き換え前（配列ポインタ）	書き換え後（アロケータブル配列）

type blockDataType
・・・
  real(8),**pointer**,dimension(:,:,:) :: inv_vol
  real(8),**pointer**,dimension(:,:,:,:,:):: fNormal,xix
・・・
end type blockDataType
	type blockDataType
	・・・
	  real(8),**allocatable**,dimension(:,:,:) :: inv_vol
	  real(8),**allocatable**,dimension(:,:,:,:,:):: fNormal,xix
	・・・
	end type blockDataType

なお、今回の変更に関して大半の実行文は変更不要であるが、別のポインタに代入される箇所については、target 属性あるいは pointer 属性が無いと翻訳時エラーになるが、今回の調査ではコスト上位に含まれないため対象外とした。

変更後に実行コスト上位ルーチンを対象に集計すると、自動並列化ループ数が前回に比べて 3 箇所増加したが、コストの大部分を占めるサブルーチンには変化がなかった。ほかにも自動並列化の阻害要因が含まれている可能性が考えられるが、コンパイラの出力メッセージ上では変化が見られないため、オブジェクト内部レベルの調査が必要と考えられる。

サブルーチン名	HPC2500 8 プロセス 実行コスト	サブルーチン内最内ループ数	自動並列化最内ループ数	
			書き換え前（前回の結果）	書き換え後（今回の結果）
blk_mfgs.implhs_mfgs_	14.0%	2	1	1
blk_rhsviscous.cellfacevariables_	10.8%	1	0	0
blk_muscl.muscl_co_	10.2%	1	0	0
blk_flux.flux_roe_	10.1%	1	0	0
blk_tm_spalartallmaras.muscl_2ndorder_	**7.8%**	**2**	**0**	**0→1**
blk_tm_spalartallmaras.diffusion_	5.4%	2	0	0
blk_rhsconvect.rhs_convect_	**5.1%**	**2**	**1**	**1→2**
blk_muscl.minmod_co_	2.4%	0	0	0
blk_rhsviscous.rhs_viscous_	**2.3%**	**2**	**1**	**1→2**
blk_metrics.calcmetrics_	1.6%	2	0	0
top_timeint.implicit_onestep_	1.5%	0	0	0
blk_tm_spalartallmaras.production_destruction_	1.4%	1	0	0
blk_tm_spalartallmaras.lhs_gaussseidel_	1.4%	3	0	0
blk_tm_spalartallmaras.vanalbada_	1.3%	0	0	0
blk_tm_scalar_measure.vorticity_	1.3%	1	0	0
blk_dt.calcdt_original_	0.9%	1	0	0
上位ルーチン合計	77.6%	21	3	6

6. まとめ

三次元圧縮性流体解析プログラム UPACS について、富士通 PRIMEPOWER HPC2500 上でスカラチューニングを実施した。その結果オリジナルに比べて約 2 倍程度の速度向上が見られた。本プログラムはコスト比率では CPU 時間（61%）の割合がメモリアクセス時間（39%）に比べて比較的高いのに対し、命令数比率では Float の割合が少なく 22.4%程度であった。ポインタや構造体のアドレス計算など他の命令数の割合が多いため FLOPS 値が向上しないことが判明した。

自動並列化の阻害要因に関して調査を行なった。ループ内部のポインタ引用が阻害要因となっているが、アロケータブル配列に変更することで自動並列化が適用されたループが 3 から 6 に増加したが、コストの大部分を占めるルーチンに関しては改善は見られなかった。オブジェクト内部レベルでの調査が必要と考えられる。

7. 謝辞

性能測定及びプログラムの書き換えには富士通の稲荷氏を始めとして富士通の関係各位のご協力をいただきました。ここで厚く御礼を申し上げます。

2.3 非構造格子 Euler/Navier-Stokes ソルバ JTAS のスレッド並列最適化

坂下雅秀（宇宙航空研究開発機構）

概　要

3 次元ハイブリッド非構造格子有限体積法 Euler／Navier-Stokes ソルバ JTAS（JAXA Tohoku university Aerodynamic Simulation code）は、元来、ベクトル計算機用に開発されたものであり、スレッド並列による実行が可能である。しかし、テストデータによる性能測定において、時間積分計算の部分で、8 スレッド実行によるスレッド並列加速率が約 5 倍と、理論値（8 倍）の 7 割を下回る性能しか得られていなかった。そこで、全体性能の向上とスレッド並列化の最適化を目的とした変更を加え、JTAS スレッド並列版を開発した。このスレッド版について、同様にテストデータによる性能測定を行った結果、全体の性能が約 1.5 倍向上し、時間積分計算部分で、8 スレッド実行によるスレッド並列化加速率が約 6.2 倍と、理論値の 7 割を越える性能が得られることが確認された。

1.　はじめに

現在、宇宙航空研究開発機構(JAXA)では、次世代超音速機技術の基礎研究として小型超音速実験機（NEXST－1）に関するプロジェクトが進められている[1][2]。このプロジェクトにおいては、複雑な形状の回りにおける剥離や再付着を伴う複雑な流れ場に対するCFD（Computational Fluid Dynamics）解析技術が求められている。このような解析には非構造格子法（Unstructured Grid Method）が良く用いられる。JAXAにおいては、非構造格子ソルバとして、主にJTASが用いられている[3][4][5]。JTASは、東北大学で開発されたTAS(Tohoku university Aerodynamic Simulation code)[6]をもとにJAXAに導入されているCeNSS(Central Numerical Simulation System)と呼ばれる大規模SMP（Symmetric Multiple Processor）クラスタシステム（富士通製PRIMEPOWER HPC2500）に適合するように若干の変更が加えられたコードであり、オリジナルのTASと区別する意味でJTASと呼ばれている。

JTAS は、CeNSS 向けに変更が加えられているものの、その変更は配列の次元入れ替え等限定的なものであり、CeNSS の性能を十分有効に活用出来ていないという問題があった。そこで、FORTRAN コンパイラによる自動並列化のより効率的な活用を促進することで CeNSS に対する適合性を向上させることを目的として、より内容に踏み込んだ変更を行った。また、テストデータを用いた性能測定を実施し、変更による性能向上を確認した。

2.　JTAS 概要

性能評価には、支配方程式として３次元Euler方程式を用いた。JTASではこれを、空間方向にはセル節点有限体積法により、時間方向にはEuler陰解法により離散化し、時間積分にはLU-SGS陰解法[10][11]を用いて計算する。流束の評価は、HLLEW法[8]により行われる。高次精度差分における制限関数としては、Venkatakrishnanの制限関数[9]が用いられている。

JTASは、元来ベクトル計算機用に開発されているため、LU－SGS法の適用にあたっては非構造格子に適用可能な超平面（Hyper Plane）を構成し再帰参照を回避している[10][11]。ところで、有限体積法において、流束ベクトルは、各検査体積を構成する面ごとに求める必要がある。一方で、ある面は異なる二つの検査体積の境界を構成するのであるから、その面を通る流束はその二つの検査体積に対して同じ量でかつ符号が反対であるように寄与することになる。従って、流束ベクトルを検査体積ごとにそれを構成する全ての面に対して計算することは、流束ベクトルを二重に計算することとなり効率的ではない。面ごとに計算すれば流束ベクトルの計算を最小限に押さえることが出来る。ここで、セル節点体積法の場合、検査体積がその唯一含む節点の番号で指定されるのと同様に、面はそれが唯一含む辺の番号で指定されるので、実際のプログラムにおける流束の計算は、辺IEが含む両端の節点の番号N1 及びN2 を保持する配列の名前を"NEDG2ND"とした場合、例えばリスト 2.1 のようになる。

リスト 2.1　流束計算の例

```
    DO 100 IE = 1, Nedges
       N1=NEDG2ND(1,IE)
       N2=NEDG2ND(2,IE)
       HLLEW=FLUX_FUNC(Same arguments required)
       FLUX(N1)=FLUX(N1)+HLLEW
       FLUX(N2)=FLUX(N2)-HELLW
100 CONTINUE
```

ここで、このDOループは配列"FLUX"に対して再帰参照になっていることに注意されたい。なぜならば、検査体積は複数の面から構成されているため、異なる面（辺）IEにおいて同じ検査体積番号（節点番号）がN1 又はN2 に表れるからである。

JTASでは、この再帰参照を回避しベクトル計算を行うため、色分け（Coloring）によるグループ化の手法が用いられている。これは、ある検査体積（節点）

に含まれる全ての面（辺）は必ず別の色を持つように前もって色分けしておき、DOループ内では同じ色を持った面（辺）のみを計算することにより、再帰参照を避ける方法である。この場合、実際のプログラムは、例えばリスト 2.2 に示すような二重ループになる。外側のDOループが全ての色に対する処理のループであり、内側のDOループがその内のある一つの色に対する処理を行うDOループである。色分けを適切に行うことにより、内側のループで一度に処理される面（辺）は必ず異なる検査体積（節点）を構成するものとなり、再帰参照が回避されベクトル化される[12]。

リスト 2.2　流束計算のベクトル化例

```
      DO 100 IC = 1, MAX_Colors
*VOCL LOOP,NOVREC
      DO 110 IE = 1, Num_edges(IC)
         N1=NEDG2ND(1,IE,IC)
         N2=NEDG2ND(2,IE,IC)
         HLLEW=FLUX_FUNC(Some arguments required)
         FLUX(N1)=FLUX(N1)+HLLEW
         FLUX(N2)=FLUX(N2)-HELLW
  110 CONTINUE
  100 CONTINUE
```

同様のベクトル化手法は、速度、圧力、密度及び温度の勾配（Gradient）の計算（以下、単に勾配の計算という）、制限関数の計算、LU－SGS法における計算の一部においても使用されている。但し、勾配の計算においては、計算が面（辺）ごとではなく要素毎に行われることが異なる。

JTASでは、MPIを利用したプロセス並列化もなされている。プロセス並列化を行うにあたっては、まず計算に先立って各プロセスに割り当てるために格子空間を領域分割し、この分割された領域をそれぞれのプロセスが分担して計算する[13]。

3.　計算性能最適化のための変更

全体の性能及びスレッド並列における性能向上を目的に JTAS に加えた変更内容は、以下の二点である。

(1) LU－SGS 法における節点番号の付け替え
(2) 色分けの削除及び DO ループの分割

以下、この変更を加えた JTAS をスレッド版 JTAS、または単にスレッド版という。より具体的な変更内容を以下に示す。

3.1.　LU-SGS 法における節点番号の付け替え

JTAS では非構造格子に LU－SGS 法を適用するために超平面が構成されている。ところが、節点における各物理量等を配列に保存する際には、格子生成時に付されたオリジナルの節点の番号でインデックスされる場所に保存されている。このような方法は、或る一つの超平面内に存在する節点の番号が不連続であるため、効率的なメモリアクセスを疎外する要因となることが予想された。そこで、LU－SGS 法の計算を行う部分では、ハイパー面を考慮した節点番号の付け替えを行うこととし、新たな節点番号として超平面内でオリジナルの節点番号の昇順に連続な番号を与えた。この際、LU－SGS 法に関係する部分以外では従来通りの番号付けとし、LU－SGS 法で必要となる保存量ベクトル等のデータは、従来の番号付けで保存されている配列から新たな番号付けで保存される配列に一旦コピーした後 LU－SGS 法の計算を行い、新しい時間ステップでの値を計算する際に従来の番号付けの配列に戻す方法をとった。

3.2.　色分けの削除及び DO ループの分割

オリジナルの JTAS は、既にベクトル化されているため一切の変更を加えることなしに、FORTRAN の持つ自動並列化機能によりスレッド並列化することが可能である。ところで、ベクトル化は基本的に最内側 DO ループに対してなされる。一方で、スレッド並列化では、スレッド生成のオーバーヘッドをなるべく小さくするために、スレッド生成回数の少ない、より外側の DO ループで並列化することが望ましい。色分けによるベクトル化では、例えばリスト 2.2 において内側の DO ループである DO 110 がベクトル化、即ちスレッド並列化されることとなり、スレッド生成のオーバーヘッドによる性能低下が予想された。加えて、スレッド並列化される DO ループの回転数は、生成されたスレッドの中でなるべく多くの演算が行われるように、ベクトル化における場合同様なるべく多い方が望ましいが、色分けによるベクトル化では、一度に計算されるのは一つの色に属する辺（勾配の計算の場合要素）のみであり、全ての辺を一度に処理するのに比べて性能が低下することが予想される。一方で、全ての辺について同時に計算することにすれば、リスト 2.1 に示すように DO ループは一重ループとなり、外側かつ回転数の多い理想的なループの構成となるが、再帰参照を含むため、ベクトル化もスレッド並列化も行うことが出来ない。

そこで、リスト 3.1 に示すように色分けの削除をすると共にDOループの分割を行うことで、色分けによる方法で問題になると予想される点の改善を図った。リスト 3.1 は、流束ベクトルの計算を簡略化した例であり、DO 100 において辺ごとに計算された流束ベクトルは、一旦、作業用配列"EDG_WK"に辺ごとの値として保存された後、DO 110 において、節点（検査体積）ごとの配列"FLUX"に足し込まれている。ここで、DO 100 における配列"EDG_WK"及びDO 110 における配列"FLUX"は、インデックス参照ではなく直接参照となっていることに注意されたい。これにより、メモリアクセスの効率化も期待される。

尚、この変更は、LU－SGS法の計算に対しては適用しなかった。これは、色分けを行った方が良い性能が得られたためである（「5.5 LU－SGS法の計算における測定結果及び評価」参照）。

リスト 3.1　流束計算の変更例

```
    DO 100 IE = 1, Nedges
       HLLEW=FLUX_FUNC(Some arguments required)
       EDG_WK(IE)=HLLEW
100 CONTINUE
    DO 110 N = 1, Nnode
       NEDGE = IEDGE_in_NODE(0,N)
       DO 111 IE=1,NEDGE
          IEDGE=IEDGE_in_NODE(IE,N)
          XSIGN=SIGN(1.0D0,IEDGE)
          IEDGE=ABS(IEDGE)
          FLUX(N)=FLUX(N) + XSIGN*EDG_WK(IEDGE)
111    CONTINUE
110 CONTINUE
```

4.　計算性能測定条件

4.1.　ハードウエア

表 4.1 に、性能測定に使用した大規模クラスタ SMP システム CeNSS のハードウエア諸元について示す。

表 4.1　CeNSS ハードウエア諸元

ハードウエア	Fujitsu PRIMEPOWER HPC2500
論理ピーク性能	9.3TFLOPS, 5.2GFLOPS/CPU
メモリ容量	3.6TBytes, 64GBytes/node
ノード数	56
ＣＰＵ数	1,792, 32/node
ノード内アーキテクチャ	SMP
ＣＰＵ	SPARC64V
L2 キャッシュ容量	2MBytes, on chip
インターコネクト形状	Full crossbar
インターコネクト性能	4GBytes × 2　(送受信)

4.2.　ソフトウエア

表 4.2 に使用したコンパイラ、リンケージエディタ及び MPI ライブラリのバージョン情報を示す。また、表 4.3 に翻訳時に指定したコンパイルオプションを示す。

表 4.2　ソフトウエアバージョン情報

ソフトウエア	バージョン情報
コンパイラ	Fujitsu Fortran Version 5.6
リンケージエディタ	Software Generation Utilities Solaris Link Editors: 5.8-1.296
MPI ライブラリ	MPI Patchlevel 2.21.0 MPLib version MPLIB-sr2.3.1

表 4.3　指定したコンパイルオプション一覧

コンパイルオプション
-Umpi -Qi,p -NRtrap -Kparallel -Kvppocl -Et

4.3.　JTAS 実行条件

4.3.1.　初期条件

性能測定のために設定した初期条件を表 4.4 に示す。

表 4.4　性能測定に使用した主な初期条件

設定データ	設定値	
時間積分ステップ数	500 ステップ	
クーラン数	$1.0\text{x}10^5$	
迎角	0.0 度	
レイノルズ数	$1.0\text{x}10^6$	
比熱比 γ	1.4	
自由流温度	273.15 K	
壁面温度	0.5	
自由流マッハ数	0.8	
流れの種別	層　流	
空力係数計算	あ　り	
境界流入条件	初期密度	1.0
	流入速度(x 方向)	$9.5\text{x}10^{-1}$
	初期圧力	1.0
	初期渦動粘性係数	20.0
	非スリップ境界	あ　り

4.3.2.　格子点数

性能測定のために設定した計算格子点の数を表 4.5 に示す。MPI による並列実行時には、分割された領域の間で情報の授受を行うために余分の格子点が付与される。ここでは、この情報授受のためにオーバーラップして設けられた格子点数が示されている。実際に解析が行われる格子点数は「正味」として示した。

表 4.5　性能測定に使用した計算格子点の数

要素名	要素			辺	格子点
	三角錐	三角柱	四角錐		
2 process					
proc.0	716,199	0	0	861,171	128,802
proc.1	481,662	141,636	879	870,779	160,656
小　計	1,197,861	141,636	879	1,731,950	289,458
合　計	1,340,376			1,731,950	289,458
正　味					
小　計	1,173,840	141,636	879	1,690,955	280,969
合　計	1,316,355			1,690,955	280,969

5.　計算性能測定結果及び評価

以下に、JTAS の性能測定結果及びその評価について示す。測定は、MPI による並列化におけるプロセス数を２で固定とし、スレッド並列については１プロセス当たりのスレッド数１、２、４及び８の４ケースについて行った。尚、測定は他のジョブも存在する通常の運用状態の元で行った。このため、特に経過時間の測定結果には変動要因が含まれていることに注意されたい。以下、先ず全体の測定結果について概観した後、主な処理ごとの測定結果について示し、簡単な評価を行う。

5.1.　全体の測定結果及び評価

表 5.1 及び図 5.1 に JTAS 全体及び時間積分計算に要した経過時間を測定した結果及びスレッド並列化により得られた加速率を示す。表 5.1 において、各欄の上段が秒を単位とした経過時間であり、下段が加速率

である。また、経過時間はバリア同期を取ってルートプロセス（ランク０）における測定結果のみを示している（以下同様）。

表 5.1 より、いずれのスレッド数による実行においても、オリジナルに比べてスレッド版における経過時間が短縮されており、最適化の効果が確認できた。また、８スレッド実行において、加速率が全体では６倍を下回っているものの、時間積分の計算においては６倍を上回っており、当初の目標を達成する十分な加速率が得られた。

表 5.2 及び図 5.2 にスレッド版のプロファイラによる実行状況の解析結果について、その抜粋を示す。表中、各解析項目の"Rank0"及び"Rank1"は、MPI プロセスのランク０及びランク１における解析結果である。また、"Average"は、ランク０とランク１の測定値を単純平均した値であり、"Total"はプロファイラが"Process Total"として出力した結果である。L2 キャッシュ・ミス率、アドレス変換バッファ・ミス率共に十分に小さいとは言えず、この結果、浮動小数点演算性能も 150MFLOPS を若干上回る程度に留まった。

表 5.3 及び図 5.3 に経過時間を元に算出した JTAS オリジナルに対してスレッド版でどの程度性能が向上したかを表す性能向上率を示す。性能向上率は、オリジナルの実行に要した経過時間をスレッド版の実行に要した経過時間で除したものとして計算した。いずれのスレッド数による実行でも性能が向上することが確認出来た。

表 5.1　JTAS 全体の経過時間測定結果

上段：経過時間［秒］
下段：加 速 率［－］

測定区間	1Thread	2Thread	4Thread	8Thread
オリジナル	8803.71	4978.60	2770.26	1814.81
全体	1.00	1.77	3.18	4.85
スレッド版	6012.07	3229.03	1810.94	1075.91
全体	1.00	1.86	3.32	5.59
オリジナル	8733.57	4908.60	2703.89	1748.62
時間積分	1.00	1.78	3.23	4.99
スレッド版	5889.41	3106.27	1688.14	954.76
時間積分	1.00	1.90	3.49	6.17

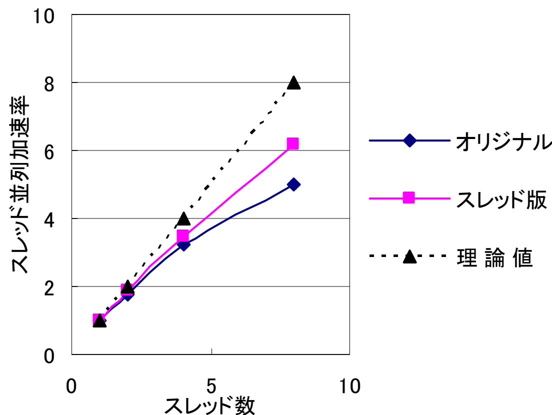
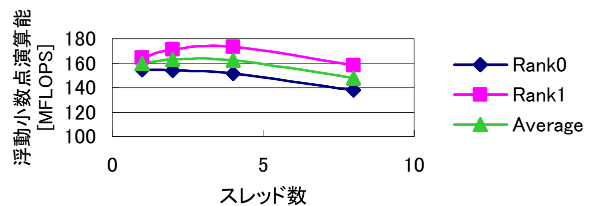


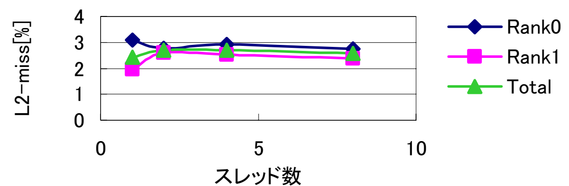
図 5.1　時間積分計算におけるスレッド並列実行加速率

表 5.2　スレッド版プロファイラ情報抜粋

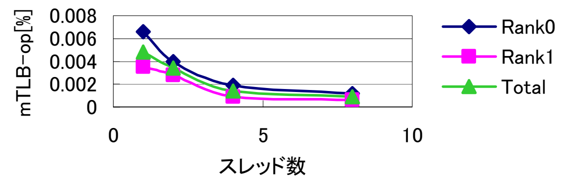
Rank	1Thread	2Thread	4Thread	8Thread
浮動小数点演算性能 FLOPS ［MFLOPS］				
Rank0	154.8	154.0	151.4	137.7
Rank1	164.4	171.1	173.1	158.1
Average	159.6	162.6	162.3	147.9
L2 キャッシュ・ミス率 L2-miss ［%］				
Rank0	3.08	2.78	2.91	2.75
Rank1	1.96	2.61	2.53	2.39
Total	2.42	2.69	2.71	2.57
アドレス変換バッファ・ミス率 mTLB-op ［%］				
Rank0	0.0066	0.0040	0.0019	0.0012
Rank1	0.0035	0.0028	0.0009	0.0006
Total	0.0048	0.0034	0.0014	0.0009



(a) 浮動小数点演算性能



(b) L2キャッシュ・ミス率



(c) アドレス変換バッファ・ミス率

図 5.2　スレッド版プロファイラ情報抜粋

表 5.3　最適化による性能向上率

1Thread	2Thread	4Thread	8Thread
1.46	1.54	1.53	1.69

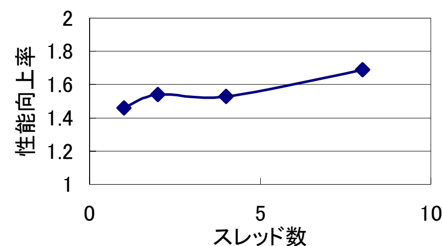


図 5.3　最適化による性能向上率

5.2. 流束の計算における測定結果及び評価

表 5.4 及び図 5.4 から 5.5 に、流束の計算について経過時間の測定を行なった結果を示す。

流束の計算においては、オリジナルに比べて処理に要する経過時間が短縮されると同時に、表 5.4 及び図 5.5 に示すように８スレッド（２プロセス、16CPU）実行においてオリジナルで 5.55 倍だったスレッド並列による加速率がスレッド版において 7.42 倍に向上しており、最適化による効果が確認できた。

表 5.4 流束の計算における測定結果

上段：経過時間［秒］
下段：加 速 率［ー］

version	1Thread	2Thread	4Thread	8Thread
オリジナル	1141.31	773.14	374.34	205.75
	1.00	1.48	3.04	5.55
スレッド版	1036.89	531.88	273.57	139.73
	1.00	1.95	3.79	7.42

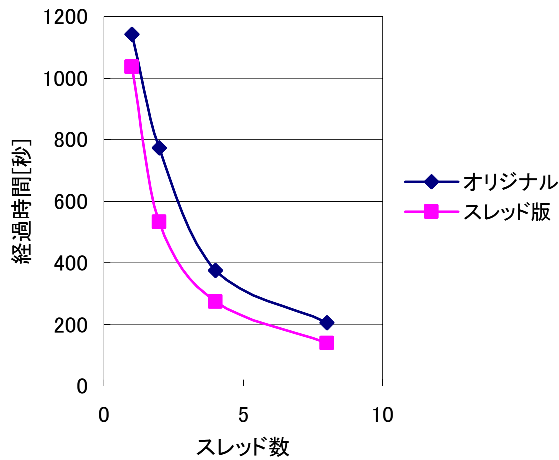


図 5.4 流束の計算における測定結果

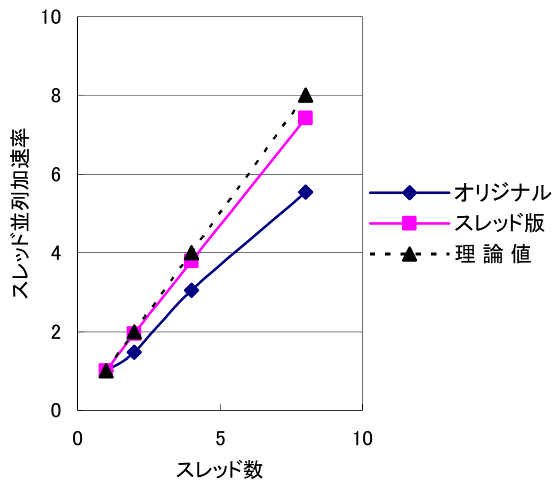


図 5.5 流束の計算におけるスレッド並列実行加速率

5.3. 勾配の計算における測定結果及び評価

表 5.5 及び図 5.6 から 5.7 に、勾配の計算について経過時間の測定を行なった結果を示す。

勾配の計算においては、８スレッド実行の場合にオリジナルで 7.39 倍あったスレッド並列化による加速率がスレッド版では 7.00 倍に低下している（表 5.5 参照）。しかし、処理に要した経過時間は 362.70 秒から 237.36 秒に短縮されており最適化による計算性能向上の効果が確認された。

表 5.5 勾配の計算における測定結果

上段：経過時間［秒］
下段：加 速 率［ー］

version	1Thread	2Thread	4Thread	8Thread
オリジナル	2679.61	1432.21	689.44	362.70
	1.00	1.87	3.89	7.39
スレッド版	1660.64	862.60	454.79	237.36
	1.00	1.93	3.65	7.00

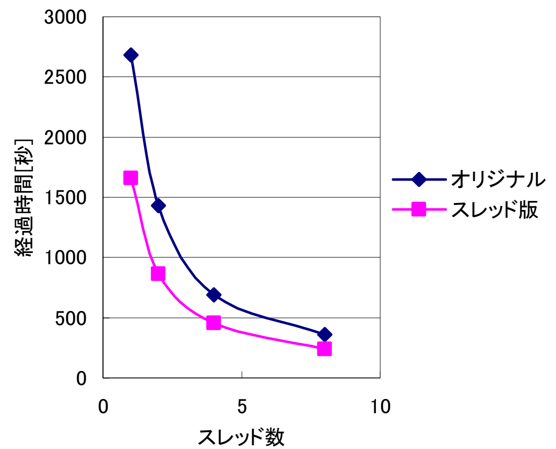


図 5.6 勾配の計算における測定結果

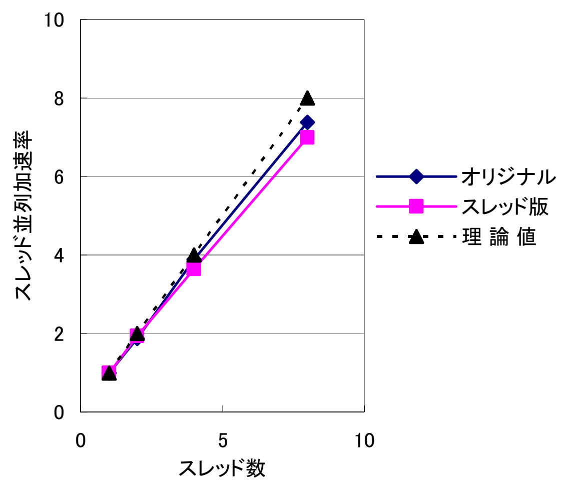


図 5.7 勾配の計算におけるスレッド並列実行加速率

5.4. 制限関数の計算における測定結果及び評価

表 5.6 及び図 5.8 から 5.9 に、制限関数の計算について経過時間の測定を行なった結果を示す。

制限関数の計算においては、8 スレッド実行の場合にオリジナルで 7.54 倍あったスレッド並列化による加速率がスレッド版では 6.71 倍に低下している。しかし、処理に要した経過時間は、211.96 秒から 161.96 秒に短縮されており最適化の計算性能向上の効果が確認された。

表 5.6 制限関数の計算における測定結果

上段：経過時間［秒］
下段：加 速 率［－］

version	1Thread	2Thread	4Thread	8Thread
オリジナル	1599.02	861.88	410.54	211.96
	1.00	1.86	3.89	7.54
スレッド版	1087.35	581.07	310.65	161.96
	1.00	1.87	3.50	6.71

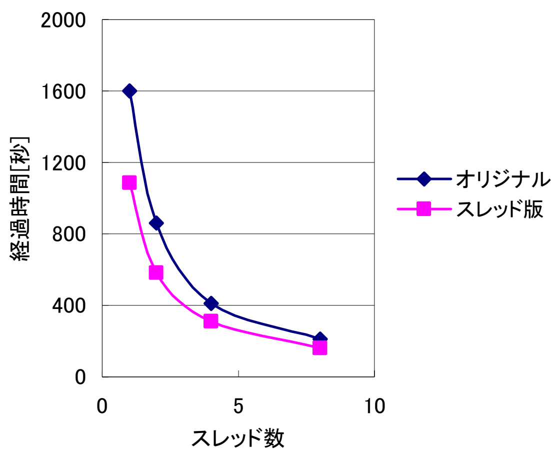


図 5.8 制限関数の計算における測定結果

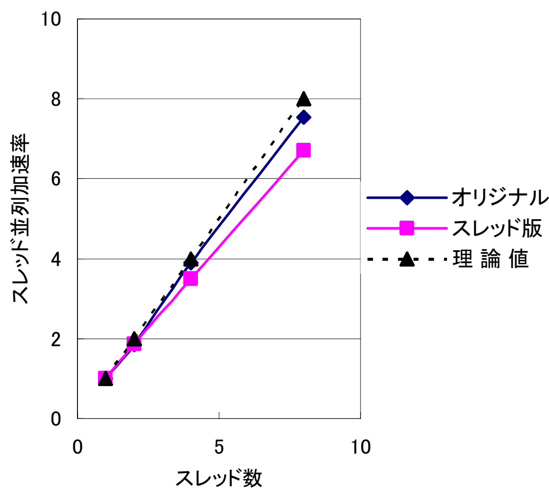


図 5.9 制限関数の計算におけるスレッド並列実行加速率

5.5. LU－SGS 法の計算における測定結果及び評価

表 5.7 及び図 5.10 から 5.11 に、LU－SGS 法の計算について経過時間の測定を行なった結果を示す。

LU－SGS 法の計算においては、8 スレッド実行の場合にオリジナルで 2.97 倍であったスレッド並列化による加速率がスレッド版では 5.76 倍に向上した。同時に、処理に要した経過時間も 773.07 秒から 238.55 秒に短縮されており最適化の効果が確認された。

表 5.8 及び図 5.12 にコーディング方法を変更した場合に、LU－SGS 法の計算に要する経過時間がどのように変化するかを示す。コーディング方法としては、(1) 色分けにより再帰参照を回避した場合（オリジナル、リスト 2.2 参照、「色分け」と表記）、(2) 色分けを削除し DO ループを分割した場合（リスト 3.1 参照、「色分け削除」と表記）、(3) 色分けによる再帰参照の回避を行うとともに、超平面を考慮した節点番号の付け替え（Reordering）を行った場合（スレッド版、「色分け+Reo.」と表記）、(4) 色分けを削除するとともに DO ループの分割を行い、さらに、節点番号の付け替えを行った場合（「色除+Reo.」と表記）の四種類を検討した。これより明らかなように、色分けの削除を行うとオリジナル（色分けによる再帰参照の回避）に比べて性能が低下すること、節点番号の付け替えを行った場合に最も良い性能が得られることが確認できた。色分けの削除を行った場合について LU－SGS 法の計算に含まれるあるサブルーチンについて調べてみると、MPI 並列のみによる実行の場合にリスト 3.1 DO 100 に相当する部分の計算時間に対して、単に節点毎の配列に足し込むのみである DO 110 に相当する部分の計算に 60%以上の時間を要しており、このことが色分けを削除し DO ループを分割した場合に性能が低下する原因であると考えられる[14]。以上のことから、スレッド版では節点番号の付け替えのみを行い色分けの削除は行わなかった。

表 5.7 LU－SGS 法の計算における測定結果

上段：経過時間［秒］
下段：加 速 率［－］

version	1Thread	2Thread	4Thread	8Thread
オリジナル	2297.07	1414.62	964.67	773.07
	1.00	1.62	2.38	2.97
スレッド版	1374.72	699.87	389.80	238.55
	1.00	1.96	3.53	5.76

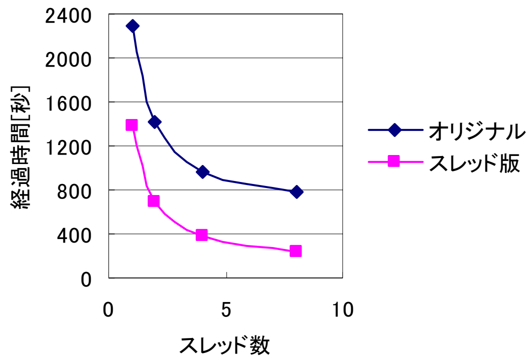


図 5.10　LU－SGS 法の計算における測定結果

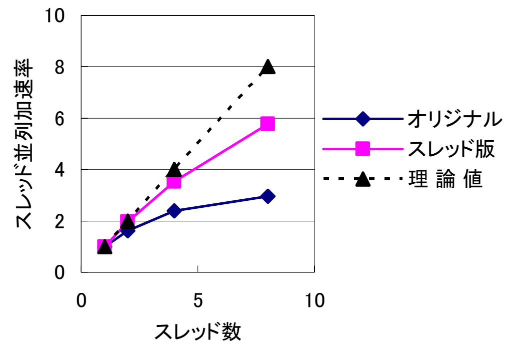


図 5.11　LU－SGS 法の計算におけるスレッド並列実行加速率

表 5.8　LU－SGSの計算におけるコーディング方法と性能の関係

経過時間［秒］

version	01Thread	02Thread	04Thread	08Thread
(1) 色　分　け　（オリジナル）	2297.07	1414.62	964.67	773.07
(2) 色分け削除	2671.32	1950.94	1638.87	1269.93
(3) 色分け＋Reo.（スレッド版）	1374.72	699.87	389.80	238.55
(4) 色　除＋Reo.	1789.36	896.81	485.75	278.96

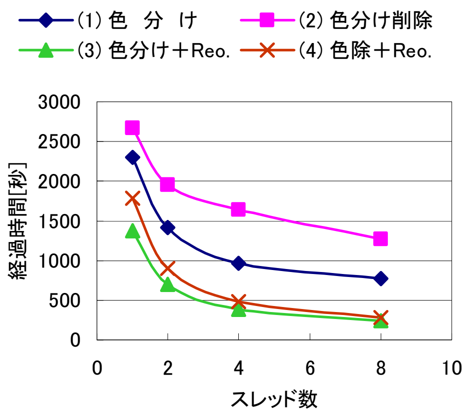


図 5.12　LU－SGS法の計算における
コーディング方法と性能の関係

5.6.　スレッド並列化におけるオーバーヘッド

勾配の計算及び制限関数の計算において、オリジナルに比べてスレッド版の計算性能は向上したものの、スレッド並列化による加速率は低下した。

そこで、加速率 S 、並列化率 α 及びスレッド数 n としてアムダールの法則、

$$S = \frac{1}{\frac{\alpha}{n} + (1-\alpha)} \tag{6.1}$$

を適用し、スレッド並列化によるオーバーヘッド O を

$$O = (1-\alpha)T \tag{6.2}$$

によって評価してみる。但し、 T は当該測定区間の計算に要した経過時間であり、従って、オーバーヘッドは、当該区間の計算に要する時間の内、並列化されていない部分の処理に要した時間である。この時、勾配の計算において、オリジナルで 98.8%であった並列化率 α がスレッド版では 98.0%に低下し、オーバーヘッドは、オリジナル版の 4.2 秒に対してスレッド版では 4.9 秒に増加している。この性能低下の原因は、リスト 3.1 の DO 110 に相当する計算のオーバーヘッドが大きいためである[14]。とはいえ、計算時間はオリジナル版に比べて短縮されておりチューニングとしての効果はあったと言える。

6.　まとめ

3 次元ハイブリッド非構造格子有限体積法 Euler／Navier-Stokes ソルバ JTAS（JAXA Tohoku university Aerodynamic Simulation code）について、全体性能の向上とスレッド並列化の最適化を目的とした変更を加え、JTAS スレッド並列版を開発して、全体の性能が約 1.5 倍向上し、時間積分計算部分で、8 スレッド実行によるスレッド並列化加速率が約 6.2 倍と、理論値の 7 割を越える性能が得られることを確認した。

参考文献

[1] Iwamiya, T., "NAL SST Project and Aerodynamic Design of Experimental Aircraft", Proceedings of the 4th ECCOMAS Computational Fluid Dynamics Conference, Wiley, Chichester, England, U.K., pp. 580-585, 1998.

[2] 高木正平, 坂田公夫 他.,"［特集］超音速実験機計画について", 日本流体力学会誌ながれ 18-5, pp.275-307, 1999.

[3] 藤田健, 松島紀左, 中橋和博., "非構造格子 CFD を用いた逆問題設計システムの高度化", 第 15 回数値 流体力学シンポジウム予稿集 D05-3, 2001.

[4] 高橋克倫, 藤田健 他., "NAL 小型超音速実験機 NEXST-1 の結合分離金具形状修正の CFD 解析", 第 17 回数値流体力学シンポジウム予稿集 F2-3, 2003.

[5] "小型超音速実験機（NEXST-1）の舵角変化時における空力特性変化の数値解析", http://www.ista.jaxa.jp/res/c02/a06_01.html

[6] Nakahashi, K., Ito, Y., and Togashi, F, "Some challenges of realistic flow simulations by unstructured grid CFD", Int. J. for Numerical Methods in Fluids, Vol.43, pp.769-783, 2003.

[7] "並列型の非構造格子ソルバープログラム　ユーザーズマニュアル"　平成 15 年 2 月 28 日　財団法人青葉工業振興会.

[8] Obayashi, S. and Guruswamy, G. P., "Convergence Acceleration of a Navier-Stokes Solver for Efficient Static Aeroelastic Computation", AIAA Journal, Vol. 33, No. 6, pp.1134-1141, 1995.

[9] Venkatakrishnan V., "On the Accuracy of Limiters and Convergence to Steady State Solutions.", AIAA Paper, 93-0880, 1993.

[10] Sharov, D. and Nakahashi, K., "Reordering of 3-D Hybrid Unstructured Grids for Vectorized LU-SGS Navier-Stokes Computations", AIAA 97-2102, 1997.

[11] Sharov, D. and Nakahashi, K., "Reordering of 3-D Hybrid Unstructured Grids for Lower-Upper Symmetric Gauss-Seidel Computations", AIAA Journal, Vol. 36, No. 3, 1998

[12] Sharov, D. , Luo, H. and Baum. J. D., "Implementation of Unstructured Grid GMRES+LU-SGS Method on Shared-Memory, Cache-based parallel Computers.", AIAA 2000-0927, 2000.

[13] Fujita, T, et. al, "Evaluation of Parallelized Unstructured-grid CFD for Aircraft Applications", Proc. of Parallel CFD 2002.

[14] 坂下雅秀,松尾裕一,村山光宏., 「非構造格子 Euler/Navier-Stokes ソルバ JTAS の計算性能最適化」, 宇宙航空研究開発機構研究開発報告, 2006, 投稿中.

2.4　近似逆行列型前処理つき CG 法の並列性能評価

藤野 清次 (九州大学情報基盤センター)

1　はじめに

一般に, 共役勾配 (Conjugate Gradient：以下, CG と略す) 法の前処理行列に望まれる性質として, 元の行列の疎 (スパース) 性の保持とその近似度の高さが上げられる. さらに, 反復計算の高い並列性を実現するには, 前処理行列のタイプも重要になってくる. 代表的な前処理の 1 つが不完全コレスキー (Incomplete Cholesky： 以下, IC と略す) 分解である. IC 分解は, 連立 1 次方程式の係数行列を三角行列どうしの積の形に不完全分解する方法である. しかしながら, この前処理はその計算順序が本質的に逐次的であり, そのため CG 法の算法中の前処理部分の並列化が困難になる.

最もシンプルな並列化手法は, ブロック IC 分解と呼ばれる方法である. この方法は 80 年代にすでに提案され, 通称並列ブロックジャコビ法とも呼ばれる. また, ブロック IC 分解は Argonne 研究所で開発された PETSc と呼ばれるライブラリーに実装され, 広く使われている [3][26]. また, Hypre という並列ライブラリも知られている [15]. しかし, ブロック IC 分解はノードに跨るフィルインをまったく考慮しないので, 収束性があまりよくない. そこで, 要素番号を並べ直すことによる改良版ブロック IC 分解が開発された [17]. また, Azte と呼ばれるライブラリーも Sandia 研究所で開発され公開されている [2]. さらに, 最小 2 乗法を利用し近似逆行列を計算するタイプの並列化手法として, Grote らが実装した SPAI[13] や Chow が実装した ParaSails[11] という並列ライブラリーもよく知られている.

一方, 選択的ブロック化によりフィルインを選択する並列化手法も研究されている [16][23][24]. これは, フィルインの発生するブロックや位置に応じて，幾つかの分解処理手順の中から適切なものを選択 (select) するという考え方に基づく方法である. さらに, シュール補元を利用した並列化手法も研究されている [25]. また, Benzi らにより, 近似逆行列に基づく並列化法の要素番号のオーダリングに基づく解析なども行なわれている [5]. また, 色付け順序法による並列化手法として, 岩下らによる ABRB 法 [17] や ABRB 法においてフィルインを考慮する井上らによる並列化手法 [16] が研究されている. この他にも並列版の前処理の有望な方法に MultiGrid 法や AMG (Algebraic MultiGrid) 法などがある [31]. 並列化の最近の動向は文献 [12][14][26] などを参照されたい.

本論文では, IC 分解あるいはロバスト IC 分解に基づき不完全分解された三角行列の逆行列を近似的に計算し前処理行列を構成する方法を提案する [32]. これと同様の考え方は, 須田により回路解析においてすでに提案され, 並列計算機 AP1000 上で実装されている [27]. しかし, 本研究の特長は, 反復法の収束安定性をよりロバストにするために, フィルインの棄却のとき対角要素を補償することにより, 並列性能のより一層の安定化を図ったところにある. また, IC 分解つき CG 法の反復計算中に現れる前進 (後退) 代入計算を行列とベクトルの積の計算に置き換えたところも大きな特長の 1 つである.

2　各種前処理の特徴

実数対称正定値行列 $A(=(a_{ij}))\in R^{n\times n}$ を持つ線形方程式

$$Ax \;=\; \boldsymbol{b} \tag{2.1}$$

を反復法の 1 つである共役勾配 (Conjugate Gradient： 以下, CG と略す) 法で解くことを考える. $\boldsymbol{x}$, $\boldsymbol{b}$ は n 次元の解ベクトルおよび右辺ベクトルとする. 特に, 行列 A が大型で疎行列の場合, 線形方程式 (2.1) は前処理つき CG 法で解かれることが多い.

ここで, 前処理 (preconditioning) とは, $K\approx A^{-1}$ となるような適当な行列 K を定め,

$$KA\boldsymbol{x}=K\boldsymbol{b} \tag{2.2}$$

のように変換することを指す. 具体的な計算においては，CG 法の反復中で，前処理行列 K とベクトルの積を計算することになる. 前処理の結果，反復法の

収束性が高められ，収束までの反復回数が大幅に減少することが多い. 代表的な前処理の 1 つである IC 分解では, 前処理行列 K を直接求めず, その代わり次のように分解した上三角行列 $U(=(u_{ij}))$ を求める.

$$A \approx U^T U \quad = \quad K^{-1}. \tag{2.3}$$

ここで, 上付き添え字 “T” は転置を表す. このように行列 A を上三角行列 U とその転置 U^T の積の形で近似する分解は不完全分解型の前処理と呼ばれる. また, CG 法の反復ループ中に現れる前処理行列とベクトルの積の計算は, 前進代入と後退代入という二段階で求められる.

2.1 IC(tol) 分解と RIC(tol) 分解

IC 分解は (完全) コレスキー分解を閾値などを使って不完全に分解するもので, 様々な改良版がある. 例えば, IC 分解で分解過程で生じたフィルインを閾値 (=tol) で棄却 (dropping) する方法がある. 以下では, この分解を IC(tol) 分解と呼ぶ. この方法は, 前処理行列の疎性の保持と前処理行列の生成時間の短縮を主な目的に開発された. IC(tol) 分解の算法を以下に示す.

- **IC(tol) 分解の算法**

for $i = 1, \ldots, n$

$$u_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} u_{ki}^2} \tag{2.4}$$

for $j = i+1, \ldots, n$

$$u_{ij} = (a_{ij} - \sum_{k=1}^{i-1} u_{ki} u_{kj}) / u_{ii}$$

dropping fillins with tolerance (2.5)

end for

end for

式 (2.5) において, 以下の棄却判定を行う.

if ($|u_{ij}| \le$ tol) then $u_{ij} = 0$
end if

しかし, 式 (2.4) の右辺の根号の中が零または負の数になり分解が途中で破綻 (breakdown) することがある. そこで, 対称正定値行列に対して, 理論的に破綻が起きないことを保証する前処理として, **RIC(tol) 分解**が提案された [1] [19]. この分解では, $A \approx U^T U + R + R^T - D$ と分解する. ここで, D は対角行列, R は形式的な行列とする. RIC(tol) 分解では, フィルインの棄却を行うと同時に, 前処理行列の対角要素にフィルインの絶対値に対応する量を加えることで破綻を未然に防いでいる. 具体的には, 式 (2.5) においてフィルイン u_{ij} を棄却するとき, u_{ij} に対応する 2 つの対角要素 u_{ii}, u_{jj} を次のように修正・補償する.

if ($|u_{ij}| \le$ tol) then $u_{ij} = 0$
$u_{ii} = (1 + |u_{ij}|/\sqrt{u_{ii}u_{jj}})u_{ii}$
$u_{jj} = (1 + |u_{ij}|/\sqrt{u_{ii}u_{jj}})u_{jj}$
end if

図 1 にフィルイン u_{ij} と対応する 2 つの対角要素 u_{ii}, u_{jj} の配置の様子を示す. 補償の対象は対角要素 u_{ii} だけでなく, 行列の対称性から, フィルイン u_{ji} に対する対角要素 u_{jj} も補償の対象になる.

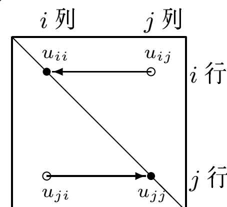


図 1: RIC(tol) 分解におけるフィルインと修正される対角要素の対応

2.2 SAInv 前処理

SAInv 前処理は, 行列 A の逆行列 A^{-1} を

$$A^{-1} \approx Z Z^T \tag{2.6}$$

と近似分解する方法である. この前処理は, Benzi らにより, **SAInv** (Stabilized Approximate Inverse) **前処理**と呼ばれた [6]. SAInv 前処理は, 行列が対称正定値行列のとき, 理論的に分解の破綻を起こさない, 前処理行列の計算が行列ベクトル積の計算で済む, など並列処理に向いた方法である.

3 前処理の並列化

ここでは, IC 分解の並列化についてを考える. IC 分解の前処理行列の作成部分および CG 法の反復ループの中の前進 (後退) 代入計算は本質的に逐次的なアルゴリズムである. そのため, IC 分解の並列化は非常に難しいことが知られている [18]. 次に, 古典的だが有用な IC 分解の並列手法であるブロック IC 分解と改良版について述べる.

3.1 ブロック IC(tol), ブロック RIC(tol) 分解

まず, 通常のブロック IC 分解による前進 (後退) 代入の並列化について記述する [4][28]. 図 2 に, 通常のブロック IC 分解において, 行列の非零要素が無視される領域および下三角行列 U^T の非零要素が含まれるブロック (図 2 中の Bl_0, Bl_1, Bl_2, Bl_3 を指す) さらに, 各ブロックとノードとの対応関係を示す. ノード数は 4 とする. このように, ブロック IC 分解では, 前進 (後退) 代入計算の並列実行を可能にするために, 非零要素は前処理行列を作成するとき, ノード間に跨る非零要素は無視される. したがって, ブロック IC 分解では, ノード数が増えれば増える程非零要素を無視する領域が拡大する. その結果, 前処理行列の近似精度が低下し, 反復回数が急激に増加することが多い.

一方, 本研究で扱う**ブロック IC(tol) 分解**とは, 非零要素が含まれるブロック (図 2 中の BL_0, BL_1, BL_2, BL_3) 内では, あらかじめ定めた閾値より大きいフィルインは前処理行列の要素として認めるという方法である. この処置により, 反復回数の急激な増加を抑えることを図る. さらに, 分解の破綻を防ぐために, 第 2.1 節で述べたフィルインを棄却するときの対角要素の補償処理を行なう. 以下, **ブロック RIC(tol) 分解**と呼ぶ.

3.2 RIC ベースの近似逆行列生成

ここでは, $A \approx U^TU$ と不完全分解する閾値つき IC(tol) 分解 あるいは $A \approx U^TU + R + R^T - D$ と不完全分解する閾値つきロバストなト RIC(tol) 分解において, 上三角行列 U の逆行列を求めることを考える [9]. そのために, ガウス・ジョルダン (Gauss-Jordan: 以下, GJ と略す) 法と同様の手順を採用する. ただし, GJ 法は線型方程式求解のための数値解法ではなく, あくまで逆行列を求めるための数値解法と位置づける [8] [10][30]. GJ 法自体の並列化については文献 [21][22] を参照されたい. GJ 法で逆行列を求めるとき, 行列 U と逆行列 U^{-1} が各々上三角行列であることから, 作業用の行列 S を用いて, 以下に示すように計算法が得られる.

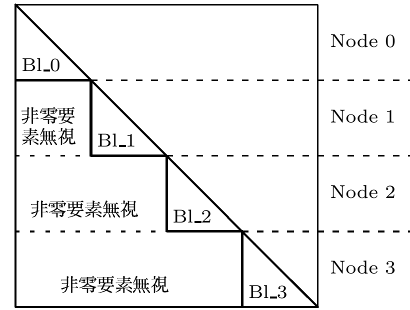


図 2: ブロック IC 分解において, 非零要素が無視される領域および下三角行列 U^T の各ブロックとノードとの対応 (ノード数は 4)

- **上三角行列に対する逆行列計算**

$$S = I \tag{3.7}$$

for $j = 1, \ldots, n$

$$s_{jj} = \frac{s_{jj}}{u_{jj}} \tag{3.8}$$

for $i = j+1, \ldots, n$

$$u_{ji} = \frac{u_{ji}}{u_{jj}} \tag{3.9}$$

end for

for $i = j+1, \ldots, n$

for $k = i+1, \ldots, n$

$$u_{jk} = u_{jk} - \frac{u_{ji}}{u_{ii}} u_{ik} \tag{3.10}$$

end for

$$s_{ji} = -\frac{u_{ji}}{u_{ii}} s_{ii} \tag{3.11}$$

end for

end for

ここで, 式 (3.7) は行列表現とする. 以上から, IC(tol) 分解および RIC(tol) 分解をベースにした近似逆行列型分解が以下のように得られる. この分解法を **IC(tol)AInv** (AInv は Approximate Inverse の略) 分解および **RIC(tol)AInv** 分解と各々呼ぶ.

- **IC(tol)AInv 分解の算法**

for $i = 1, \ldots, n$

$$u_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} u_{ki}^2}$$

for $j = i+1, \ldots, n$

$$u_{ij} = (a_{ij} - \sum_{k=1}^{i-1} u_{ki}u_{kj})/u_{ii} \quad (3.12)$$

dropping fillins with tolerance

(3.13)

end for

end for

$***$ IC(tol) decomposition is over. $***$

$$W = U \quad (3.14)$$

$$Z = I \quad (3.15)$$

for $j = 1, \ldots, n$

$$z_{jj} = \frac{z_{jj}}{w_{jj}} \quad (3.16)$$

for $i = j+1, \ldots, n$

$$w_{ji} = \frac{w_{ji}}{w_{jj}} \quad (3.17)$$

end for

for $i = j+1, \ldots, n$

for $k = i+1, \ldots, n$

$$w_{jk} = w_{jk} - \frac{w_{ji}}{w_{ii}} w_{ik} \quad (3.18)$$

end for

$$z_{ji} = -\frac{w_{ji}}{w_{ii}} z_{ii} \quad (3.19)$$

end for

end for

IC(tol) 分解が終了した直後の式 (3.14) では, 前処理行列 U が一時的作業用配列 W に収められ, 次に GJ 法に従って上三角行列 Z の要素が求まる. また, 式 (3.19) において近似逆行列を構成する行列 Z の要素が確定する. さらに, 式 (3.13) は IC(tol) 分解において閾値 (=tol) によるフィルインの棄却の判定処理であり, IC(tol) 分解をベースにした IC(tol)AInv 分解および RIC(tol) 分解をベースにした RIC(tol)AInv 分解における棄却処理は次のように行なわれる.

if ($|u_{ij}| \leq$ tol) then $u_{ij} = 0$

if IC(tol)AInv is adopted

then goto (3.14)

else if RIC(tol)AInv is adopted then

$u_{ii} = (1 + |u_{ij}|/\sqrt{u_{ii}u_{jj}})u_{ii}$

$u_{jj} = (1 + |u_{ij}|/\sqrt{u_{ii}u_{jj}})u_{jj}$

end if

end if

4 数値実験

4.1 計算環境と計算条件

表 1 に計算機環境を示す. 計算はすべて倍精度演算で行なった. CG 法の収束判定条件は反復第 k 回目の残差ベクトル $\boldsymbol{r}_k$ の 2 ノルムが, $||\boldsymbol{r}_k||_2/||\boldsymbol{r}_0||_2 < 10^{-12}$ を満たしたとき収束とした. 初期近似解 $\boldsymbol{x}_0$ はすべて 0 とした. 係数行列は対角スケーリングを行い対角項を 1.0 に揃える正規化処理をした.

表 1: 計算機環境

項目	仕様
計算機	IBM eServer p5 モデル 595
設置場所	九州大学情報基盤センター
CPU	Power5 (1.9GHz)
CPU 数	16
許容メモリ量	48GB
OS	AIX 5L
コンパイラ	XL Fortran 9.1
並列化ライブラリ	OpenMP

表 2: テスト行列の特徴

行列	次元数	非零要素	非零要素/次元数	分野
TUBE1-2	21,498	459,277	21.4	タイヤチューブ [20]
NASASRB	54,870	1,366,097	24.9	シャトル ロケット [29]
TCASE	134,856	4,830,936	35.8	応力解析
ENGINE	143,571	2,424,822	35.8	エンジン 応力解析 [29]
SBEAM	352,496	13,528,771	38.4	応力解析
MODEL3	374,229	11,165,692	29.8	応力解析
GRID_w	634,335	22,385,096	35.3	設計計算
GRID_s	838,395	13,566,835	16.2	設計計算

表 2 にテストに用いた 8 つの行列を示す．そのうち TUBE1-2 は R. Kouhia の疎行列データベースから，NASASRB と ENGINE は Florida 大学の疎行列データベースから選んだ [20][29]．それ以外の 5 つの行列は工学分野の解析で実際に現れた行列である．

4.2 逐次版の安定性能の評価

表 3 と表 4 に 8 つの係数行列に対する逐次版の前処理つき CG 法の CPU 時間を掲載した．調べた前処理は IC(tol) 分解, RIC(tol) 分解, IC(tol)AInv 分解, RIC(tol)AInv 分解の 4 つである．ただし, 表中では, 各々IC 分解, RIC 分解, ICAInv 分解, RICAInv 分解と略記した．各前処理の閾値 (=tol) は 0.01, 0.05, 0.1 と 3 通り変化させ, そのうち最も合計時間が少なかったものを表に載せた．CPU 時間の測定は Fortran90 の組み込み関数 etime を用いて行なった． 表中で,「precond.」は前処理,「tol」は閾値,「fill-in ($\times 10^6$)」は前処理行列で使われたフィルインの個数を表す．単位は 10^6 である．「Itr.」は反復回数,「pre-t」は前処理行列作成に要した時間,「CG-t」は CG 法の反復計算の時間,「tot-t」はそれらの合計時間を各々表す．また, 反復回数の欄で「max」は最大反復回数までで CG 法が収束しなかったことを表す．表 3 と表 4 から以下のことがわかる．

- IC(tol) 分解と IC(tol)AInv 分解は 8 つの行列のうち, 僅か 2 つの行列 (行列 ENGINE, SBEAM) しか収束せず, 収束が非常に不安定である．
- 一方, RIC(tol) 分解と RIC(tol)AInv 分解は 8 つすべての行列で収束し, 収束が安定である．
- メモリ量については, IC(tol) 分解が収束した行列 ENGINE, SBEAM の場合でも, RIC(tol) 分解つきの方が IC(tol) 分解の場合よりも少ない．

以上の逐次処理の結果から，収束が安定である RIC(tol) 分解および RIC(tol)AInv 分解の場合の並列性能について次の 4.3 節で報告する．また, 比較のために SAInv 分解の性能についても報告する．

4.3 並列性能評価

並列ライブラリ OpenMP を使用し, スレッド数を 1, 2, 4, 8, 16 と変化させて並列性能を調べた．スレッド数の上限を 16 としたのは計算機が設置された情

表 3: 係数行列 (TUBE1-2, NASASRB, TCASE and ENGINE) に対する前処理つき CG 法の時間

precond.	tol	fill-in ($\times 10^6$)	Itr.	pre-t	CG-t	tot-t
IC	-	-	max	-	-	-
RIC	0.01	0.38	1973	0.19	10.80	10.99
ICAInv	-	-	max	-	-	-
RICAInv	0.01	0.71	2147	1.21	15.59	16.80

(a)matrix TUBE1-2

precond.	tol	fill-in ($\times 10^6$)	Itr.	pre-t	CG-t	tot-t
IC	-	-	max	-	-	-
RIC	0.05	0.47	5987	0.2	71.4	71.6
ICAInv	-	-	max	-	-	-
RICAInv	0.05	0.34	7349	0.7	80.6	81.3

(b)matrix NASASRB

precond.	tol	fill-in ($\times 10^6$)	Itr.	pre-t	CG-t	tot-t
IC	-	-	max	-	-	-
RIC	0.01	4.49	1743	4.5	200.4	204.9
ICAInv	-	-	max	-	-	-
RICAInv	0.05	1.02	5272	3.7	254.1	257.8

(c)matrix TCASE

precond.	tol	fill-in ($\times 10^6$)	Itr.	pre-t	CG-t	tot-t
IC	0.01	3.46	453	2.6	29.8	32.4
RIC	0.01	2.96	874	1.8	50.7	52.5
ICAInv	0.05	1.57	1884	2.8	71.1	73.9
RICAInv	0.05	0.78	2127	1.4	62.2	63.5

(d)matrix ENGINE

表 4: 係数行列 (SBEAM, MODEL3, GRID_w and GRID_s) に対する前処理つき CG 法の時間

precond.	tol	fill-in ($\times 10^6$)	Itr.	pre-t	CG-t	tot-t
IC	0.01	10.61	556	11.0	164.4	175.4
RIC	0.01	9.02	1008	8.3	239.3	247.6
ICAInv	0.01	14.30	814	48.0	221.2	269.2
RICAInv	0.01	8.58	1282	32.1	267.1	299.3

(e)matrix SBEAM

precond.	tol	fill-in ($\times 10^6$)	Itr.	pre-t	CG-t	tot-t
IC	-	-	max	-	-	-
RIC	0.01	6.57	2303	5.0	403.0	408.0
ICAInv	-	-	max	-	-	-
RICAInv	0.01	7.15	2486	20.2	418.8	439.0

(f)matrix MODEL3

precond.	tol	fill-in ($\times 10^6$)	Itr.	pre-t	CG-t	tot-t
IC	-	-	max	-	-	-
RIC	0.05	5.46	4052	3.5	994.4	997.9
ICAInv	-	-	max	-	-	-
RICAInv	0.05	2.84	4435	8.0	926.3	934.3

(g)matrix GRID_w

precond.	tol	fill-in ($\times 10^6$)	Itr.	pre-t	CG-t	tot-t
IC	-	-	max	-	-	-
RIC	0.01	16.44	15749	9.6	5917.1	5926.7
ICAInv	-	-	max	-	-	-
RICAInv	0.1	2.43	40348	4.1	6958.0	6962.1

(h)matrix GRID_s

報基盤センターの運用上の制約である．経過時間の測定は合計 3 度行いその平均値を表に掲載した．また経過時間の測定は Fortran90 の組み込み関数 SYSTEM_CLOCK を用いて行なった．本節の並列性能評価では, 収束が安定な前処理であるブロック RIC(tol) 前処理 (以下, 表中では BRIC と略す) つき CG 法 , SAInv 前処理 (以下, 表中では SAInv と略す) つき CG 法, RIC(tol)AInv 前処理 (以下, 表中では RICAInv と略す) つき CG 法の並列性能を評価した．調べた閾値は 0.1 と 0.05 の場合の 2 ケースで, あまり性能面で差がなかったので, 閾値が 0.1 の場合の結果を以下の表では掲載した．また, 行列は対称であるため一般には下三角部分のみ保持すればよいが, 全非零要素を保持する方が並列計算において経過時間が短かったので, 数値実験では行列の全非零要素を行列ベクトルの積の計算で使用した．

表 5 から表 12 に各行列に対する実験結果を示す．表中で, 「precond.」は前処理, 「th」はスレッド数, 「fill-in ($\times 10^6$)」は前処理行列で使われたフィルインの個数を表す．単位は 10^6 である．「Itr.」は反復回数, 「pre-t」は前処理行列の作成に要した時間, 「CG-t」は CG 法の反復計算の時間, 「tot-t」はそれらの合計時間, 「speedup」はスレッド数が 1 のときの経過時間を 1.0 としたときの各スレッドでの経過時間の比率, すなわち「台数効果」を各々表す．時間の単位はすべて秒とする．台数効果は, スレッド数が 1 のケースの合計時間を 1.0 としたときに, 各スレッド数で並列化によって得られた速度向上の倍率を指す．また各表の合計時間の欄で太字の数字はスレッド数が 16 のとき経過時間が最も短かったものを指す．表 10 の行列 MODEL3 の前処理 SAInv で, 反復回数の欄で「max」は最大反復回数までで CG 法が収束しなかったことを表す．

表 5: 行列 TUBE1-2 に対する並列版 PCG 法の性能

pre-cond.	th	fill-in ($\times 10^6$)	Itr.	pre-t [s]	CG-t [s]	tot-t [s]	speed-up
BRIC	1	0.071	6137	0.04	28.45	28.50	1.00
	2	0.070	6193	0.05	14.67	14.72	1.94
	4	0.069	6275	0.04	7.44	7.48	3.81
	8	0.068	6347	0.04	3.90	3.94	7.23
	16	0.067	6514	0.05	2.80	2.85	10.00
SAInv	1	0.194	3363	0.80	20.20	21.01	1.00
	2	0.194	3555	0.81	10.90	11.71	1.79
	4	0.194	3416	0.81	5.32	6.13	3.43
	8	0.194	3361	0.82	2.80	3.62	5.81
	16	0.194	3426	0.83	2.00	**2.83**	7.42
RIC-AInv	1	0.056	7603	0.08	37.49	37.57	1.00
	2	0.056	7601	0.08	19.28	19.36	1.94
	4	0.056	7601	0.08	9.76	9.84	3.82
	8	0.056	7599	0.08	5.32	5.40	6.96
	16	0.056	7600	0.08	3.41	3.49	10.78

表 6: 行列 NASASRB に対する並列版 PCG 法の性能

pre-cond.	th	fill-in ($\times 10^6$)	Itr.	pre-t [s]	CG-t [s]	tot-t [s]	speed-up
BRIC	1	0.201	10885	0.1	186.2	186.3	1.00
	2	0.201	10883	0.1	76.4	76.5	2.44
	4	0.199	10940	0.1	38.8	39.0	4.78
	8	0.196	10938	0.1	20.4	20.6	9.07
	16	0.189	11066	0.1	14.1	14.3	13.08
SAInv	1	0.576	11401	3.0	276.7	279.7	1.00
	2	0.576	11410	3.1	138.4	141.5	1.98
	4	0.576	11412	3.1	52.6	55.6	5.03
	8	0.576	11398	3.0	25.8	28.8	9.70
	16	0.576	11404	3.0	13.7	16.7	16.75
RIC-AInv	1	0.111	14522	0.2	253.8	254.0	1.00
	2	0.111	14519	0.2	115.9	116.1	2.19
	4	0.111	14525	0.2	51.3	51.6	4.92
	8	0.111	14521	0.2	26.0	26.2	9.68
	16	0.111	14525	0.2	13.9	**14.1**	17.97

表 7: 行列 TCASE に対する並列版 PCG 法の性能

pre-cond.	th	fill-in ($\times 10^6$)	Itr.	pre-t [s]	CG-t [s]	tot-t [s]	speed-up
BRIC	1	0.845	5445	0.6	413.3	413.9	1.00
	2	0.837	5482	0.6	214.0	214.6	1.93
	4	0.815	5446	0.6	88.4	89.0	4.65
	8	0.790	6945	0.6	48.5	49.1	8.43
	16	0.731	7437	0.6	26.2	26.8	15.46
SAInv	1	2.125	6446	18.3	615.5	633.8	1.00
	2	2.125	6449	18.4	323.1	341.4	1.86
	4	2.125	6437	18.4	149.7	168.1	3.77
	8	2.125	6443	18.4	56.8	75.2	8.42
	16	2.125	6438	18.4	26.6	44.9	14.10
RIC-AInv	1	0.328	7697	1.1	549.3	550.4	1.00
	2	0.328	7686	1.1	272.5	273.6	2.01
	4	0.328	7673	1.1	104.6	105.6	5.21
	8	0.328	7676	1.1	50.4	51.5	10.69
	16	0.328	7677	1.1	23.6	**24.7**	22.30

表 8: 行列 ENGINE に対する並列版 PCG 法の性能

pre-cond.	th	fill-in ($\times 10^6$)	Itr.	pre-t [s]	CG-t [s]	tot-t [s]	speed-up
BRIC	1	0.452	2612	0.3	112.2	112.5	1.00
	2	0.408	2915	0.3	50.8	51.0	2.20
	4	0.376	2997	0.3	24.1	24.3	4.63
	8	0.362	3042	0.2	12.8	13.1	8.61
	16	0.346	3074	0.2	7.9	**8.2**	13.78
SAInv	1	0.736	1390	6.6	69.1	75.7	1.00
	2	0.736	1390	6.6	31.9	38.5	1.97
	4	0.736	1390	6.7	15.7	22.4	3.38
	8	0.736	1390	6.8	8.4	15.2	5.00
	16	0.736	1390	6.7	5.0	11.7	6.49
RIC-AInv	1	0.301	2904	0.5	130.0	130.4	1.00
	2	0.301	2903	0.5	57.0	57.5	2.27
	4	0.301	2903	0.5	25.3	25.8	5.06
	8	0.301	2902	0.5	15.0	15.4	8.45
	16	0.301	2902	0.5	9.1	9.6	13.59

表 9: 行列 SBEAM に対する並列版 PCG 法の性能

pre-cond.	th	fill-in $\times 10^6$	Itr.	pre-t [s]	CG-t [s]	tot-t [s]	speed-up
BRIC	1	0.604	2859	1.0	531.7	532.8	1.00
	2	0.604	2859	1.1	276.4	277.4	1.92
	4	0.601	3067	1.0	156.8	157.9	3.37
	8	0.594	3261	1.0	85.8	86.8	6.13
	16	0.581	3430	1.0	43.4	44.5	11.98
SAInv	1	.1304	1748	17.4	362.2	379.6	1.00
	2	1.304	1748	17.5	185.6	203.0	1.87
	4	1.304	1748	17.5	96.2	113.6	3.34
	8	1.304	1746	17.5	50.2	67.6	5.61
	16	1.304	1747	17.6	25.0	**42.7**	8.90
RIC-AInv	1	0.410	4126	1.1	805.1	806.3	1.00
	2	0.410	4121	1.1	412.0	413.2	1.95
	4	0.410	4116	1.1	213.0	214.2	3.76
	8	0.410	4106	1.1	113.3	114.4	7.05
	16	0.410	4107	1.1	52.1	53.2	15.14

表 10: 行列 MODEL3 に対する並列版 PCG 法の性能

pre-cond.	th	fill-in ($\times 10^6$)	Itr.	pre-t [s]	CG-t [s]	tot-t [s]	speed-up
BRIC	1	0.888	7413	1.0	1223.0	1224.0	1.00
	2	0.882	7422	1.0	641.9	642.9	1.90
	4	0.875	7443	1.0	317.1	318.1	3.85
	8	0.860	7494	1.0	149.6	150.6	8.13
	16	0.823	7615	1.0	66.8	**67.8**	18.06
SAInv	1	-	max	-	-	-	-
	2	-	max	-	-	-	-
	4	-	max	-	-	-	-
	8	-	max	-	-	-	-
	16	-	max	-	-	-	-
RIC-AInv	1	0.568	7875	1.2	1337.9	1339.1	1.00
	2	0.568	7873	1.2	685.2	686.5	1.95
	4	0.568	7872	1.2	354.3	355.6	3.77
	8	0.568	7871	1.2	161.8	163.1	8.21
	16	0.568	7870	1.2	75.8	77.1	17.37

表 11: 行列 GRID_w に対する並列版 PCG 法の性能

pre-cond.	th	fill-in ($\times 10^6$)	Itr.	pre-t [s]	CG-t [s]	tot-t [s]	speed-up
BRIC	1	1.625	5457	1.9	1797.6	1799.5	1.00
	2	1.620	5522	1.9	918.1	920.0	1.96
	4	1.615	5553	1.9	468.9	470.9	3.82
	8	1.574	5720	1.9	254.0	255.9	7.03
	16	1.535	5762	1.9	161.2	163.1	11.03
SAInv	1	3.979	2504	52.8	912.7	965.4	1.00
	2	3.979	2501	52.7	462.7	515.5	1.87
	4	3.979	2501	53.7	251.4	305.2	3.16
	8	3.979	2501	53.0	127.4	180.3	5.35
	16	3.979	2501	53.0	69.4	**122.3**	7.89
RIC-AInv	1	0.815	6458	2.3	2116.7	2119.0	1.00
	2	0.815	6455	2.3	1041.1	1043.4	2.03
	4	0.815	6454	2.3	537.0	539.3	3.93
	8	0.815	6454	2.3	283.5	285.7	7.42
	16	0.815	6454	2.3	152.8	155.1	13.67

表 12: 行列 GRID_s に対する並列版 PCG 法の性能

pre-cond.	th	fill-in ($\times 10^6$)	Itr.	pre-t [s]	CG-t [s]	tot-t [s]	speed-up
BRIC	1	3.796	35099	1.9	9411.3	9413.2	1.00
	2	3.788	36218	1.9	4651.3	4653.2	2.02
	4	3.768	38638	1.9	2736.9	2738.8	3.44
	8	3.732	40237	1.9	1345.3	1347.1	6.99
	16	3.705	43245	1.9	755.1	756.9	12.44
SAInv	1	9.712	14947	70.3	5155.8	5226.0	1.00
	2	9.712	14947	70.7	2651.8	2722.4	1.92
	4	9.712	14947	71.6	1392.0	1463.6	3.57
	8	9.712	14947	70.3	692.0	762.4	6.86
	16	9.712	14946	70.5	355.8	**426.3**	12.26
RIC-AInv	1	2.439	40348	4.0	10291.7	10295.7	1.00
	2	2.439	40345	4.0	5166.3	5170.3	1.99
	4	2.439	40344	4.0	2684.5	2688.6	3.83
	8	2.439	40342	4.0	1335.0	1339.1	7.69
	16	2.439	40341	4.0	692.2	696.3	14.79

これらの表から以下の観察ができる.

- 16 スレッドのとき, 合計時間が最も短かったのは RIC(tol)AInv 前処理のときが 2 ケース (行列 NASASRB, TCASE), BRIC(tol) 分解のときが 2 ケース (行列 ENGINE, MODEL3) そして SAInv 前処理のときが 4 ケースであった.
- スレッド数が増加したときの収束までの反復回数について, BRIC(tol) 分解では段々と増加するのに対して, SAInv 前処理と RIC(tol)AInv 前処理ではほぼ一定回数で変動が少ない. このように似た傾向を示すのは, SAInv 前処理と RIC(tol)AInv 前処理が, プロセッサ台数によって前処理行列が変わらないことから, 至極当然の結果であると考えられる.
- RIC(tol)AInv 前処理で必要な前処理行列のフィルインの個数は相対的に他の 2 つの前処理のときよりも少なくて済むことが多い.
- SAInv 前処理では, 表 10 のときのように収束しないケースがあり, BRIC(tol) 分解と RIC(tol)AInv 分解に比べてその収束が不安定である.

解析結果をより詳細に検討するために, 表 13 に並列版前処理つき CG 法の 16 スレッドを使用した場合の台数効果の比をまとめた. 表中の太字の数字は各行列で最も台数効果が高かった前処理を示す. 表 13 の結果から, RIC(tol)AInv 前処理が他の 2 つの前処理に比べて台数効果が高いことがわかる. SAInv 前処理と RIC(tol)AInv 前処理は, このように CG 法の反復時間に関していずれも高い台数効果を示すが, 前処理行列の作成の負荷は, SAInv 前処理の負荷の方がずっと重い. したがって, 前処理行列の作成部分を並列化しない場合には, 全体での台数効果は RIC(tol)AInv 前処理の方が高くなる. また, 16 スレッドを越えるような多数スレッドの場合には RIC(tol)AInv 前処理が有用になると思われるが, 前述のように今回の数値実験では運用上の制約から実験出来なかった.

同様に, 16 スレッドを使用した場合において, 表 14 に SAInv 前処理と RIC(tol)AInv 前処理において, 前処理つき CG 法における前処理行列作成の時間と CG 法の反復計算の時間との関係をまとめた. 表中の比 (ratio) は CG 法の反復時間を 1 としたときの前処理時間の比率である. SAInv 前処理のときの前処理行列作成の時間に比べて RIC(tol)AInv 前処理のときの同時間が大幅 (約 10%以下) に短くなったことがわかる. その結果, SAInv 前処理のときのように, 多数のスレッドを使用したとき前処理行列を作成する時間が顕在化するという現象は現れなくなった.

表 13: 並列版前処理つき CG 法の台数効果の評価 (16 スレッドのとき)

Matrix	preconditioning		
	BRIC	SAInv	RICAInv
TUBE1-2	10.00	7.42	**10.78**
NASASRB	13.08	16.75	**17.97**
TCASE	15.46	14.10	**22.30**
ENGINE	13.78	6.49	**13.59**
SBEAM	11.98	8.90	**15.14**
MODEL3	**18.06**	-	17.37
GRID_w	11.03	7.89	**13.67**
GRID_s	12.44	12.26	**14.79**

表 14: 前処理行列作成の時間と CG 法の反復計算の時間との関係 (16 スレッドのとき)

Matrix	SAInv			RICAInv		
	pre-t [s]	CG-t [s]	ratio	pre-t [s]	CG-t [s]	ratio
TUBE1-2	0.8	2.0	0.415	0.08	3.41	0.023
NASASRB	3.0	13.7	0.219	0.2	13.9	0.014
TCASE	18.4	26.6	0.692	1.1	23.6	0.047
ENGINE	6.7	5.0	1.340	0.5	9.1	0.055
SBEAM	17.6	25.0	0.704	1.1	52.1	0.021
MODEL3	-	-	-	1.2	75.8	0.016
GRID_w	53.0	69.4	0.764	2.3	152.8	0.015
GRID_s	70.5	355.8	0.198	4.0	692.2	0.006

5 HPC2500 並列計算機上での性能評価

評価した解法は近似逆行列 (SAINV) 前処理つき CG 法である. テスト行列は構造解析の実数対称正定値行列 (行列 Grid_s と行列 Grid_w) である. 使用した計算機は, 富士通社 HPC2500 である. 最適化オプションは-KOMP -Kfast,largepage,parallel -Cpp -KV9 とした. 表 15 と表 16 に測定結果を示す. ただし, 京都大学での測定は, 通常運転のときの計測であり, 単独実行による計測でないことを付記する. 表中の時間の単位はすべて秒とする.

表 15: 近似逆行列型前処理 (SAINV), 行列 Grid_s

Thread	HPC2500			p5/595
	1.3GHz	1.56GHz	2.08GHz	1.9GHz
	富士通沼津	京都大学	京都大学	九州大学
1	-	-	-	5226
2	-	4480.9	4598.6	2722
4	-	3293.8	2445.9	1463
8	-	1736.4	1748.5	762
16	-	977.0	986.6	426
32	401.7	700.5	520.3	-
64	318.4	450.1	-	-
126	289.2	254.4	-	-

表 16: 近似逆行列型前処理 (SAINV), 行列 Grid_w

Thread	HPC2500			p5/595
	1.3GHz	1.56GHz	2.08GHz	1.9GHz
	富士通沼津	京都大学	京都大学	九州大学
1	-	-	-	965.4
2	-	767.6	755.2	515.5
4	-	555.2	420.1	305.2
8	-	328.9	391.1	180.3
16	-	210.3	210.8	122.3
32	133.0	178.1	129.6	-
64	121.3	126.1	-	-
126	115.8	92.2	-	-

6 おわりに

RIC(tol) 分解ベースに近似的に逆行列を計算するという並列計算向きの CG 法の前処理を新しく提案した. 提案した RIC(tol)AInv 前処理は, (i) 前処理行列の作成に要する時間が従来の SAInv 前処理のときよりも少なくて済み, (ii) 並列計算のときの CG 法の収束性が安定化し, (iii) 並列台数に係わらず, ほぼ一定の反復回数で収束し, 台数効果が得やすい, などのいくつかの優れた特徴を有することがわかった.

今後の課題は, (i)16 スレッドを越えるような多数スレッドでの性能評価, (ii) 最適な閾値の場合の各前処理の性能の比較をすること, そして (iii) 前処理行列の作成部分の並列化などである.

参考文献

[1] Ajiz, M.A., Jennings, A.: A Robust Incomplete Choleski-Conjugate Gradient Algorithm, *Int. J. Numer. Methods Engrg.*, Vol. 20, pp. 949–966 (1984).

[2] Aztec: A massively Parallel Iterative solver library for solving sparse linear systems の HP, http://www.cs.sandia.gov/CRF/aztec1.html

[3] Balay, S., et al.: PETSc Users manual, ANL-95/11 Rev.2.1.5, Argonne National Laboratory, (2004).

[4] Barton,M.L.: Three-dimensional magnetic field computation on a distributed memory parallel processor, *IEEE Trans. Magns.*, Vol. 26-2, pp. 834–836 (1990).

[5] Benzi, M., Marin, J., Tuma, M.: A two-level parallel preconditioner based on sparse approximate inverse, In D. Kincaid ed., Iterative methods in Scientific computation IV, *IMACS series in Comput. and Appl. Math.*, Vol. 5, IMACS, pp. 167–178 (1999).

[6] Benzi, M., Cullum, J.K. and Tuma, M.: Robust Approximate Inverse Preconditioning for the Conjugate Gradient Method, *SIAM J. Sci. Comput.*, Vol. 22, pp. 1318–1332 (2000).

[7] Benzi, M., Tuma, M.: A Robust Incomplete Factorization Preconditioner for Positive Definite Matrices, *Numerical Linear Algebra with Applications*, Vol. 10, pp. 385–400 (2003).

[8] Bogacki, P.: HINGES - An illustration of Gauss-Jordan reduction, MathDL: *J. of Online Math. and its Appli.*, (2006).

[9] Bollhöfer, M., Saad, Y.: On the relations between ILUs and factored approximate inverses, *SIMAX*, Vol. 24, pp. 219–237 (2002).

[10] Chen, K., Evans, D.: An efficient variant of Gauss-Jordan type algorithms for direct and parallel solution of dense linear systems, *Int. J. of Computer Math.*, Vol. 76, pp. 387–410 (2000).

[11] Chow, E.: Parallel implementation and practical use of sparse approximate inverses with a priori sparsity patterns, *Int. J. High Perf. Comput. Appl.*, Vol. 15, pp. 56–74 (2001).
Vol. 10, pp. 385–400 (2003).

[12] Duff, I.S., van der Vorst, H.A.: Developments and trend in the parallel solution of linear systems, *Parallel Computing*, Vol. 25, pp. 1931–1970 (1999).

[13] Grote, M., Huckle, T.: Parallel preconditioning with sparse approximate inverses, *SIAM J. Sci. Comput.*, Vol. 18, pp. 838–853 (1997).

[14] Henon, P., et al.: Blocking issues for an efficient parallel block ILU preconditioner, *Abstracts of International Workshop Preconditioning 2005*, Atlanta U.S., May (2005).

[15] Hypre: Scalable Linear Solvers の HP, http://www.llnl.gov/CASC/linear_solvers/

[16] 井上明彦, 藤野清次: フィルインの選択に基づく改良版 ABRB 順序付け法による ICCG 法の並列化, 情報処理学会論文誌 コンピュータシステム, Vol. 46 No.SIG16 (ACS12), pp. 119–128 (2005).

[17] Iwashita, T., Shimasaki, M.: Parallel precessing of 3-D eddy current analysis with moving conductor using parallelized ICCG solver with renumbering process, *IEEE Trans. Magn.*, Vol. 36, pp. 1504–1509 (2000).

[18] 岩下 武史, 島崎 眞昭, 同期点の少ない並列化ICCG 法のためのブロック化赤－黒順序付け, 情報処理学会論文誌: Vol. 43, No.4, pp. 893–904 (2002).

[19] 柿原正伸, 藤野清次: 緩和係数 ω を自動決定する対角緩和準ロバスト ICCG 法の収束性, 情報処理学会論文誌 コンピュータシステム, Vol. 46 No.SIG4(ACS9), pp. 45–55, (2005).

[20] R. Kouhia, Sparse Matrices web page: http://www.hut.fi/~kouhia/sparse.html.

[21] Melab, N., Petiton, S. and Talbi, E.-G.: A new parallel adaptive block-based Gauss-Jordan algorithm, Publication Interne AS-180, LIFL, Université de Lillel, Fév (1998).

[22] Melab, N.: A parallel adaptive Gauss-Jordan algorithm, *The Journal of Supercomputing*, Vol. 17, pp. 167–185 (2000).

[23] Nakajima, K., Okuda, H.: Parallel iterative solvers with selective blocking preconditioning for simulations of fault-zone contact, *Numer. Linear Algebra Appl.*, Vol. 11, pp. 831–852 (2004).

[24] Raghavan, P., Teranishi, K., Ng, E.: A latency tolerant hybrid sparse solver using incomplete Cholesky factorization, *Numer. Linear Algebra Appl.*, Vol. 10, pp. 541–560 (2003).

[25] Saad, Y.: Distributed Schur complement techniques for general sparse linear systems, SIAM J. Sci. Comput., Vol. 21, pp. 1337-1356 (1999).

[26] Saad, Y.: Iterative methods for sparse linear systems 2nd ed., SIAM Philadelphia (2003).

[27] Suda, R.: Large scale circuit analysis by preconditioned relaxation methods, *Proc. of PCG'94*, Keio Univ., pp. 189–197 (1994).

[28] Vollaire, C., Nicolas, L.: Preconditioning techniques for the Conjugate Gradient solver on a parallel distributed memory computer, *IEEE Trans. Magn.*, Vol. 34, pp. 3347–3350 (1998).

[29] University of Florida Sparse Matrix web page: http://www.cise.ufl.edu/research/sparse/matrices.

[30] 山本 哲朗, 数値解析入門 [増訂版], サイエンス社 (2002).

[31] Yavneh, I. (Editor): Special Issue: Multigrid methods, *Numer. Linear Algebra Appl.*, Vol. 11, No.2–3 (2004).

[32] 吉田正浩, 不完全分解の逆行列を用いた前処理つき CG 法の並列化, 九州大学大学院システム情報科学府情報工学専攻, 修士論文, 2006.3.

2.5　SMPにおけるスレッド並列の台数効果と高速化手法

金澤 正憲[1] 義久 智樹[1] 杉崎 由典[2] 青木 正樹[2]
京都大学[1] 富士通(株)[2]

この報告は、『杉崎由典，青木正樹，義久智樹，金澤正憲：SMP におけるスレッド並列の台数効果と高速化手法について，情報処理学会研究報告「2005 年並列／分散／協調処理に関する「武雄」サマー・ワークショップ（SWoPP 武雄 2005)」，2005-EVA-14，pp.1-6，2005　8 月』をもとに、手を加えたものである。

1.　はじめに

京都大学学術情報メディアセンターでは、2004 年 3 月にスーパーコンピュータを、ベクトル並列機（富士通 VPP800/63）から、SMP クラスタ（富士通 PRIMEPOWER　HPC2500/12 ノード）に置き換えた。ここでは、VPP800 導入時に、ベクトル計算機の性能評価に用いられたベンチマーク 3 本を取り上げ、ＳＭＰクラスタで実行し、並列台数効果を測定した。台数効果の十分でないベンチマークに対して、ソースプログラムの簡単な書き換えを行えば、自動並列で、台数効果が著しく改善されることを確認した。また、MPI で書き換えたベンチマークを実行し、MPI による並列処理との比較を行ったが、自動並列によるスレッド並列と大差なかった。

2.　ベンチマークについて

ここで用いたベンチマークプログラムは、3 本で、grad、 product5、 shift3 と呼んでいる。概要を表 1 に示す。

表1：ベンチマーク概要

名称	プログラムの概要
grad	近傍との差分
product5	行列積
shift3	値の伝播（他の影響を含む）

それぞれのプログラムの主要部分を附録に記した。どれも 2 次元配列を扱うプログラムで、配列の大きさを増減することにより、いろいろな演算速度のコンピュータで実行できるようになっている。

3.　SMP クラスタでの測定結果

3 本のプログラムについて、オリジナルソースのまま自動並列化した場合、高速化のためのチューニングを行った場合、MPI で記述した場合について実測し、3 つの場合の結果をまとめて図示する。測定結果については、特に断らない限り、1CPU 向きの最適化を施したプログラムの実行時間を 1 として表示している。また、実行時間とともにモニタ機能から収集された性能情報をもとに、結果を分析する。

3.1　測定システムおよび環境

ここで使用した測定マシンと条件（パラメータなど）を表 2 に示す。

表 2．測定マシンと条件

実行環境	富士通 PRIMEPOWER HPC2500　(SPARC64V 1.82GHz) Parallelnavi2.4
並列数	128CPU,512GB の 1 ノード上にて 1～120CPU を使用
翻訳時オプション	-Kfast_GP2=3,prefetch=4, parallel=3,V9, largepage=2,hardbarrier -w

3.2　オリジナルソースプログラムの場合

各プログラムに手を加えることなく実行させて得られた結果から判明した性能上の問題を以下に示す。

(1)grad

TLB ミス、L2 キャッシュミスが多発している。

同一ループ内に配列の次元アクセス順序が異なる文が存在するため、連続アクセスとストライドアクセスが混在している。混在によりコンパイラの最適化が効かず、ストライドアクセスによる TLB ミスにより性能が劣化している。

(2)product5

matmul 並列ライブラリ呼出しに展開されている。

(3)shift3

L2 キャッシュミスが多発している。

10 個の do 文があるが、1 つ目の do 文を実行した後の do 文実行ではアクセスするデータがキャッシュに残っていない。そのためミスが多発し、性能が劣化している。

3.3 最適化を行なった場合

オリジナルソースコードの性能検証から、性能影響要因への対処方法を検討・実施し、性能向上を図った。実施した性能向上策と効果を以下に示す。

(1)grad

メモリアクセスを全て連続アクセスにするため、ループ分配及びループ交換を行った。図 a 参照。これにより、データの局所性が高まり、モニタ機能による解析情報から、TLB ミス及び L2 キャッシュミスが削減されたことを確認した。

```
          DO 300 J=1,N-1
            DO 300 I=1,N
              B(I,J)=A(I,J+1)-A(I,J)
300           C(J,I)=A(J+1,I)-A(J,I)
                    ↓  変更
          DO 300 J=1,N-1
            DO 300 I=1,N
300           B(I,J)=A(I,J+1)-A(I,J)
          DO 310 I=1,N
            DO 310 J=1,N-1
310           C(J,I)=A(J+1,I)-A(J,I)
```

図 a. チューニング例（grad N=50000）

チューニング前後のメモリ性能を表 3 に示す。図 1 に台数効果を示す。この 2 つの結果から、ループ分解と交換が非常に効果の大きいことが判る。

表 3. メモリ性能(grad)

チューニング	L2 キャッシュミス(%)	TLB ミス (%)	メモリアクセス(%)
前	32.9950	0.5789	98.39
後	3.4604	0.0000	41.98

(N=50000, CPU=4)

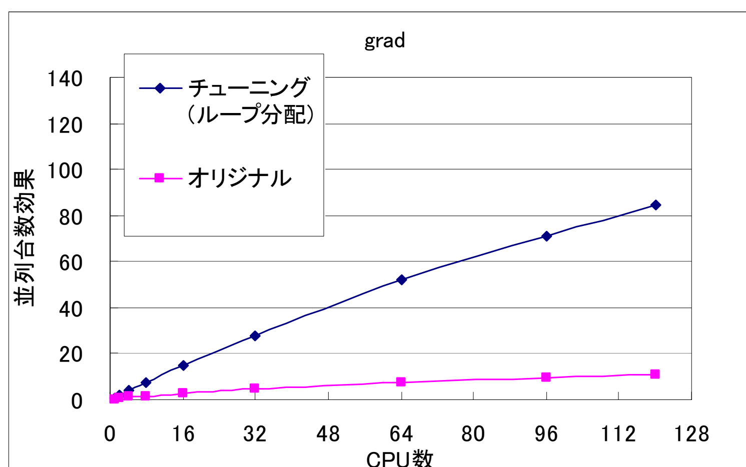


図 1. grad チューニング効果

(2)product5

matmul 並列ライブラリ呼出しに展開されているため、図 2 に示すとおり、十分高速な結果が得られている。ループ分配最適化を行うことで、配列のゼロクリア処理の部分に、ループ分配最適化を行ったが、余り大きな効果はなかった。なお、64CPU 以上で並列台数効果が低下するのは、行列積(N=12000)を並列処理した場合の計算粒度が十分でないためと思われる。N=100000（プログラムの大きさ約 230GB）とし、2006 年 6 月に測定をしなおすと、図 3 に示すようになり、並列効果は、むしろ、スーパーリニアになった。

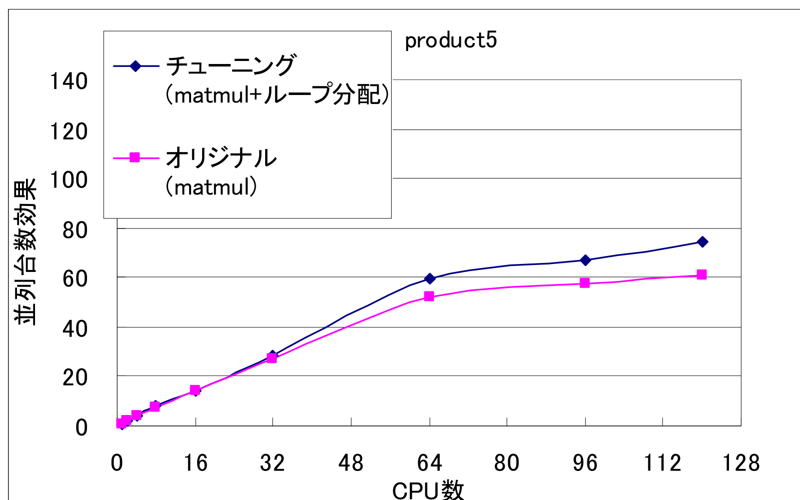


図 2．product5 チューニング効果（N=12000）

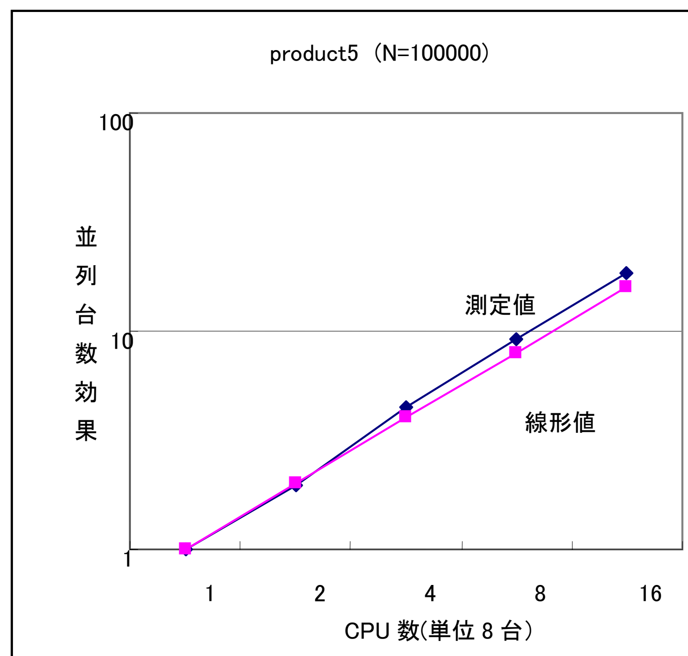


図 3．　Product5　自動並列　(N=100000)

(3)shift3

ループ融合を行うことで、内側のループでアクセスする次元方向の配列領域が局所化され、次のループ処理の際にはその次元方向のデータは全てオンキャッシュとなる。チューニングについては、図 b を参照。図 4 に台数効果を示す。

チューニング前後のメモリ性能を表 4 に示す。明らかに、非常に大きな効果があったことが判る。

表 4.　メモリ性能(shift3)

チューニング	L2 キャッシュミス(%)	TLB ミス (%)	メモリアクセス(%)
前	0.9596	0.0000	42.89
後	0.0998	0.0000	10.07

(N=50000, CPU=4)

```
      DO 10 J=1,NN
      DO 10 I=1,NN
       A(I,J)=A(I+1,J)+0.48*B(I,J)+0.122*(B(I-1,J)+B(I,J-1)+B(I+1,J)+B(I,J+1))
10    CONTINUE
       …
      DO 100 J=1,NN
      DO 100 I=1,NN
       A(I,J)=A(I+2,J)+0.46*B(I,J)+0.118*(B(I-1,J)+B(I,J-1)+B(I+1,J)+B(I,J+1))
100   CONTINUE
                 ↓  変更
      DO J=1,NN
        DO 10 I=1,NN
         A(I,J)=A(I+1,J)+0.48*B(I,J)+0.122*(B(I-1,J)+B(I,J-1)+B(I+1,J)+B(I,J+1))
10      CONTINUE
       …
        DO 100 I=1,NN
         A(I,J)=A(I+2,J)+0.46*B(I,J)+0.118*(B(I-1,J)+ B(I,J-1)+B(I+1,J)+B(I,J+1))
100     CONTINUE
      ENDDO
```

図 b.　チューニング例（shift3　N=100000）

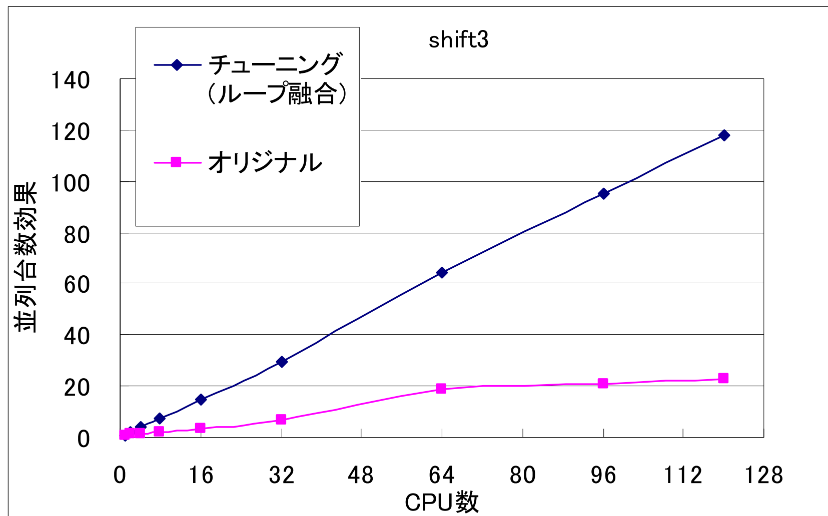


図 4.　shift3 チューニング効果

3.4 MPI で書き直した場合

チューニングを施したソースコードを用い、これを MPI へ書き換えを行った。書き換えに際しては、主要ループ内での通信が発生しないような方法にて行っている。MPI への書き換え方法と性能結果を以下に示す。

(1)grad

スレッド並列と同様に、配列の 2 次元目を並列化した。

主要ループ中には通信処理がないこともあり、並列台数効果もスレッド並列と同等の性能を示している。図 5 参照。

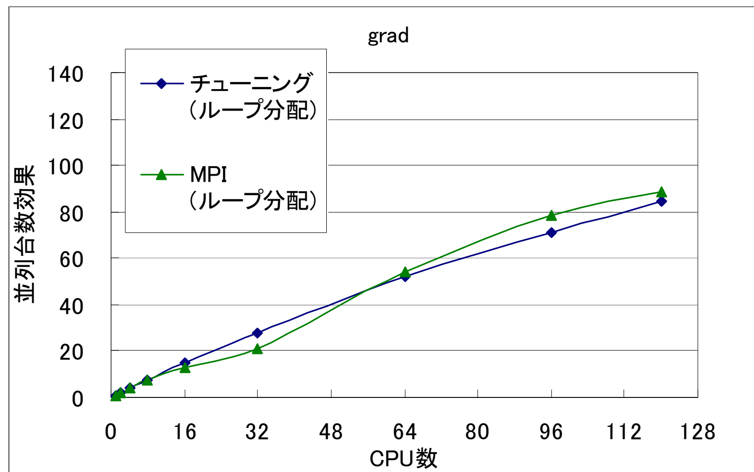


図 5．grad 自動並列と MPI の性能比較

(2)product5

一部の配列を各プロセスで重複して保持することにより、主要ループ中に通信処理が入らないようにしている。その影響で matmul 関数ライブラリ呼出しに展開されなくなった。そのため、性能がスレッド並列と比較して大きく低下している。図 6 参照。

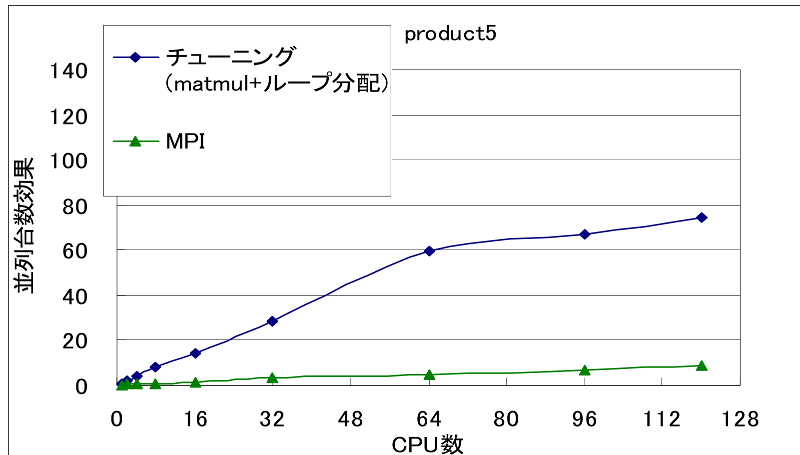


図 6．product5 自動並列と MPI の性能比較

(3)shift3

スレッド並列と同様に、配列の 2 次元目を並列化した。

主要ループ中には通信処理がないこともあり、並列台数効果もスレッド並列と同等の性能を示している。図 7 参照。

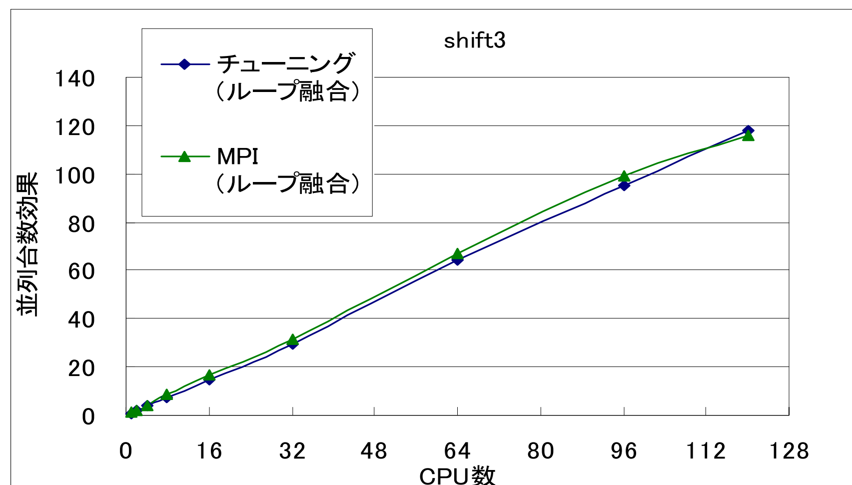


図 7．shift3 自動並列と MPI の性能比較

4. VPP との性能比較

Fujitsu　VPP5000 の 1CPU で同じプログラムを実測し、HPC2500 の実測値と比較した。さらに、スレッド並列で 32CPU の場合も測定した。表 5 に測定値及び比較結果を示す。効果の欄については HPC2500 の 1CPU を 1 とした際の性能効果を表している。

VPP5000 はピーク性能 9.6GFlops、HPC2500 はピーク性能 7.28GFlops である。LINPACK によると、SMP の性能はピークの 50%と換算するのが、妥当と考えられ、VPP5000 と HPC2500 は、約 2.6 倍と推測される。shift3 では、同程度の性能が出ているといえる。

表 5：VPP5000 と HPC2500 の性能比較

Bench mark	Size(N)	CPU 台数	VPP5000 1	HPC2500 1	HPC2500 32
grad	500	時間	2,864	25,611	967
		効果	8.94	1.00	26.49
	5000	時間	393,423	2,679,202	101,226
		効果	6.81	1.00	26.47
product5	12000	時間	2,147	903	33
		効果	0.42	1.00	27.36
shift3	1000	時間	14,766	46,520	1,411
		効果	3.15	1.00	32.97
	10000	時間	1,344,125	4,765,837	154,533
		効果	3.55	1.00	30.84

注：時間は grad, shift3 がμsec, product5 が sec。

5. おわりに

京都大学学術情報メディアセンターで従来から用いてきたベンチマークプログラムによって、SMP による自動並列処理の効果を実測し、評価した。共有メモリ型システムにおいて、自動並列化による効果が、十分多数の台数まで（今回は 120CPU まで)、ほぼ線形的に向上することがわかった。

高速処理をするためには、よく知られた並列化手法であるループ分配、ループ交換、ループ融合が大いに有効であることが実証できた。このことから、コンパイラがループ分配、交換、融合などの典型的な並列化技法を自動的に取り入れる、または、プログラマに候補の技法と場所を推定して、助言を行う必要がある。さらに、会話的に並列化機能とその効果を直ちに判定できる機能が不可欠であろう。

自動ベクトル化コンパイラが種々の機能を順次、取り入れたように、自動並列化コンパイラも今後絶えず新しい機能を盛り込むことが重要である。

product5 の行列積のように、並列化、最適化を駆使するよりも、既存の並列ライブラリを用いたほうが、圧倒的に性能が高いことがわかった。その上、プログラムも

```
c=matmul(b,c)
```

と簡潔な形で記述できるので便利である。

したがって、まず、第 1 の解決策として、並列ライブラリの普及にセンターとしては努める必要があると自覚した。一方、コンパイラがソースを読みきって並列ライブラリを組み込むようにすることも非常に有効である。導入当初のコンパイラは、行列積の計算であると読み切る能力が弱かったが、現在は改善されている。

上述したように、自動並列化コンパイラは、ソース解析技術の向上が必要であると考えられる。

SMP マシンは、巨大な主記憶を共有するため、CPU 数が多くなると、メモリアクセスで競合が起こるという問題が指摘されていた。そこで、ベンチマークプログラムを MPI で書き直して、その実行時間を実測したところ、メモリアクセスの競合に関しては、ほとんどないと推測される。

さらに、VPP5000 の 1CPU で実行した結果と比較すると、同じ 1CPU なら VPP5000 が 3 倍から 8 倍程度速いことが確認できた。

これらの結果は、ハードおよびソフトの並列技術の発展に有意義なデータを与えるものと考えている。

今後も利用者プログラムを収集して、ベンチマークを整備していきたい。

最後に、助言をいただいた京都大学学術情報メディアセンター岩下武史助教授、富士通㈱荒川征己氏、並びに、関係者各位に深く感謝します。

参考文献

1）草野義博，新庄直樹；ハイパフォーマンスコンピュータ：PRIMEPOWER HPC，Fujitsu，Vol.53, No.6, pp.444-449，2002.

2）富士通；Parallelnavi Fortran 使用手引書，マニュアル

3）富士通；Parallelnavi Fortran 文法書，マニュアル

附録 1:ベンチマークプログラムの主要ループ部

各プログラムの主要ループ部を以下に示す。

附表1：grad

```
      DO 300 J=1,N-1
        DO 300 I=1,N
          B(I,J)=A(I,J+1)-A(I,J)
 300      C(J,I)=A(J+1,I)-A(J,I)
      DO 400 I=1,N
        B(I,N)=A(I,1)-A(I,N)
 400    C(N,I)=A(1,I)-A(N,I)
      DO 500 J=1,N
        DO 500 I=1,N
  500     A(I,J)=ATAN2(B(I,J),C(I,J))
```

gradでは、配列のｘ軸方向及びｙ軸方向の差分を求め、結果に対してatan2演算を行っている。

附表2：product5

```
      DO 300 I=1,N
        DO 300 J=1,N
          C(I,J)=0
          DO 300 K=1,N
 300        C(I,J)=C(I,J)+A(I,K)*B(K,J)
```

product5では行列積を行っている。

附表3：shift3

```
      DO 10 J=1,NN
      DO 10 I=1,NN
      A(I,J)=A(I+1,J)+0.48*B(I,J)+0.122*(B(I-1,J)+B(I,J-1)+B(I+1,J)+B(I,J+1))
10    CONTINUE
      DO 20 J=1,NN
      DO 20 I=1,NN
      A(I,J)=A(I+3,J)+0.44*B(I,J)+0.114*(B(I-1,J)+B(I,J-1)+B(I+1,J)+B(I,J+1))
20    CONTINUE
      DO 30 J=1,NN
      DO 30 I=1,NN
      A(I,J)=A(I+5,J)+0.40*B(I,J)+0.106*(B(I-1,J)+B(I,J-1)+B(I+1,J)+B(I,J+1))
30    CONTINUE
      DO 40 J=1,NN
      DO 40 I=1,NN
      A(I,J)=A(I+7,J)+0.36*B(I,J)+0.098*(B(I-1,J)+B(I,J-1)+B(I+1,J)+B(I,J+1))
40    CONTINUE
      DO 50 J=1,NN
      DO 50 I=1,NN
      A(I,J)=A(I+9,J)+0.32*B(I,J)+0.090*(B(I-1,J)+B(I,J-1)+B(I+1,J)+B(I,J+1))
50    CONTINUE
      DO 60 J=1,NN
      DO 60 I=1,NN
      A(I,J)=A(I+10,J)+0.30*B(I,J)+0.086*(B(I-1,J)+B(I,J-1)+B(I+1,J)+B(I,J+1))
60    CONTINUE
      DO 70 J=1,NN
      DO 70 I=1,NN
      A(I,J)=A(I+8,J)+0.34*B(I,J)+0.094*(B(I-1,J)+B(I,J-1)+B(I+1,J)+B(I,J+1))
70    CONTINUE
      DO 80 J=1,NN
      DO 80 I=1,NN
      A(I,J)=A(I+6,J)+0.38*B(I,J)+0.102*(B(I-1,J)+B(I,J-1)+B(I+1,J)+B(I,J+1))
80    CONTINUE
      DO 90 J=1,NN
      DO 90 I=1,NN
      A(I,J)=A(I+4,J)+0.42*B(I,J)+0.110*(B(I-1,J)+B(I,J-1)+B(I+1,J)+B(I,J+1))
90    CONTINUE
      DO 100 J=1,NN
      DO 100 I=1,NN
      A(I,J)=A(I+2,J)+0.46*B(I,J)+0.118*(B(I-1,J)+B(I,J-1)+B(I+1,J)+B(I,J+1))
 100   CONTINUE
```

shift3では、配列Aのx軸方向に1～10要素離れた要素に対して、配列B及び配列Bの4近傍の係数付きの加算結果を加えている。

別紙：

SMP におけるスレッド並列の多数台の台数効果

京都大学学術情報メディアセンター
金澤　正憲

前述の**『SMPにおけるスレッド並列の台数効果と高速化手法』**でのベンチマークgradにより、多数のCPUでの台数効果を調べてみた。1回だけの実行で得られた結果を表1に示す。

測定の結果、127台の場合を除いて、台数効果は見られた。複数回実行させて、比較しなければならないが、未だ台数効果があるという傾向は明白になったと判断できる。

表１　台数効果（120～128CPU）

Grad（N=100000）測定日　2006.9.1

台数	経過時間	比率	台数比率
120	12.6848	1.000	1.000
121	12.6360	1.004	1.008
122	12.5131	1.014	1.017
123	12.4507	1.019	1.025
124	12.3664	1.026	1.033
125	12.3001	1.031	1.042
126	12.2559	1.035	1.05
127	12.2646	1.034	1.058
128	12.1948	1.040	1.067

注。京都大学学術情報メディアセンターの HPC2500（128CPU、512GB、1.82GHz）

2.6 マルチカラーオーダリングによるスレッド並列型ICCGソルバによるHPC2500のベンチマーク評価

岩下武史[†]　金澤正憲[†]
杉崎由典[††]　青木正樹[††]

Performance Evaluation of By Using Thread Parallel ICCG Solver Based on Multi-Color Ordering

Takeshi Iwashita,[†] Masanori Kanazawa ,[†]
Yoshinori Sugisaki [††] and Masaki Aoki[††]

1. はじめに

京都大学学術情報メディアセンターでは，2004 年 3 月にスーパーコンピュータを置き換え，それまでの並列ベクトル型計算機 富士通 VPP800 から SMP クラスタ型並列計算機 富士通 HPC2500 に移行した[1]．SMP クラスタ型並列計算機は，近年スーパーコンピュータのシステムとして主流の形態となっており，パーソナルコンピュータにおけるプロセッサのマルチコア化を鑑みると，一層その傾向は強まると考えられる．そこで本稿では，代表的な SMP クラスタ型並列計算機の一つである富士通 HPC2500 を対象として，並列化 ICCG ソルバベンチマークによる性能評価を行う．

富士通 HPC2500 は，富士通社製の SPARC 互換プロセッサを使用した，所謂スカラ型の並列計算機システムである．本計算機システムはノードと呼ばれるメモリを共有したマルチプロセッサシステムを高速のクロスバネットワークにより相互に接続した形態を持ち，一般に SMP クラスタ型と呼ばれるカテゴリに属する．同計算機は，1 ノード内に 128 個の CPU を持つノード内高密度プロセッサ実装を特徴とし，512GB の共有メモリ空間を有している．一般にノード内の並列処理には OpenMP などの API によるスレッド並列処理が用いられる．一方，複数のノードを使用した並列処理の場合には，ノード間のデータ転送に MPI Library 等の通信ライブラリが使用され，マルチプロセスによる処理が行われる．本稿では，HPC2500 の 1 ノードを用い，OpenMP により記述されたスレッド並列処理ベンチマークプログラムにより評価を行う．

† 京都大学学術情報メディアセンター
Academic Center for Computing and Media Studies, Kyoto University
†† 富士通株式会社
Fujitsu Ltd.

本稿で用いるベンチマークは線形反復解法の一つである ICCG 法[2] の並列化ソルバである．同手法は有限要素法や差分法から生ずる対称な係数行列をもつ連立一次方程式の解法として最も一般的なものの一つである．ICCG 法は共役勾配法に不完全コレスキー分解前処理を施した手法であるが，一般に共役勾配法に関する処理は並列化が容易であり，前処理に伴う計算は並列処理が難しいとされている．そこで，不完全コレスキー分解やその広いカテゴリである ILU 分解前処理の並列化に関して様々な提案がなされている[3),4]．その中で，本稿で用いるソルバでは，マルチカラーオーダリング法[4),5] を用いる．同手法は，未知変数を並列処理に適した形に並び替えるリオーダリング手法の一種で，古くから知られた古典的手法である．しかし，その有用性は様々なアプリケーションで知られ，最新の数値シミュレーションにおいてもその概念が利用されている[6]．マルチカラーオーダリング法の実装方法は複数考えられるが，本稿では構造型の差分解析と非構造型の有限要素解析の 2 種を対象とし，それぞれ異なる方法を用いる．差分解析では，ストライドアクセスによる方法を用い，有限要素解析では，間接アドレッシングによる方法を使用する．本ベンチマークにより，スカラ型並列計算機において重要なキャッシュの有効利用性，並列台数効果等について検討を行う．

2. 並列化 ICCG 法ベンチマーク

2.1 マルチカラーオーダリング法による並列化 ICCG 法

ICCG 法ソルバは大きく分けて，反復解法により連立一次方程式の求解を行う反復部と前処理行列を求める等の反復解法に必要なセッティングを行うセットアップ部に分けられる．このうち，計算時間の点では一般的に反復部が支配的であり，大規模問題ではより顕著である．本

稿では並列処理のための付加的に必要な処理を含めて反復の前段階のセットアップ部は並列化せず，逐次処理とする．ICCG 法の反復部分は以下のような計算により構成される．(i) 前進・後退代入計算，(ii) ベクトルの内積計算，(iii) 行列ベクトル積，(iv) ベクトルの更新およびノルムの計算 である．このうち，計算量として支配的であるのは，前進・後退代入計算と行列ベクトル積計算である．また，これらの計算部分のうち内積計算や行列ベクトル積計算は並列化が容易であり，前進・後退代入計算の並列化が一般に困難であるとされている．そこで，従来から前進・後退代入計算の並列処理に関する様々な提案がなされている[3),4)]．

本稿では，IC 分解前処理に伴う前進・後退代入計算の並列化のために，マルチカラーオーダリング法を用いる．同手法は，未知変数の順序を入れ替えることにより係数行列の非ゼロパターンを並列処理可能な形に変換するリオーダリング手法の一種である．リオーダリング手法に関する研究は様々になされており，多くのオーダリングが提案されているが，マルチカラーオーダリングはスレッド並列処理に向いたオーダリングである．マルチカラーオーダリングでは，互いに依存関係にない未知変数を一つのグループとし，これを 1 色とみなす．ここで，i 番目の未知変数と j 番目の未知変数が依存関係にないとは係数行列の (i,j) 要素 a_{ij} が 0 であることを意味する．この結果，各色内で前進・後退代入計算の並列処理が可能となる．一方，異なる色に属する未知変数間は依存関係がある可能性があるので，各色に関する処理が終了するごとに同期，または通信が発生する．そのため，同期・通信コストを削減するためになるべく少ない色数での利用が好ましいと考えれてきたが，ILU(IC) 分解前処理では一般に多くの色数を用いるほうが前処理効果が高く，30 色から 100 色の利用が有効であるという報告がある[5)]．このような従来の研究結果によると，マルチカラーオーダリングは他のオーダリングと比べて前処理効果が高く，反復法の収束性に優れる．一方，同手法は元来ベクトル計算機のために開発された方法であり，共有メモリ型の並列処理での実装に向いている．即ち，同手法は色毎にステージ化されている方法であるが，あるプロセッサが利用するデータは，その直前の色に関する処理では複数のプロセッサが処理を行った結果となる．従って，同手法を分散メモリ型並列計算機上のプロセス並列で利用することを考えると，非常に多数の 1 対 1 通信を実装する必要があり，オーバーヘッドの点から現実的ではない．

マルチカラーオーダリングの実装法には大きく 2 種類ある．即ち，陽的に未知変数を並べ替える方法と計算順序のみを入れ替える方法である．ここで，前者の場合には係数行列は図 1 のような形になる[8)]．本解析例では，非構造型の解析において同手法を用いる．一方，メモリ上の係数行列データの格納に別の順序付けを用い，計算順序のみを入れ替える方法は，未知変数間の関係が簡単な構造型の差分解析に適しており，文献 9 においてその具体的手法が提案されている．本解析においても，差分解析ベンチマークでは本手法によりマルチカラーオーダリングの実装を行う．

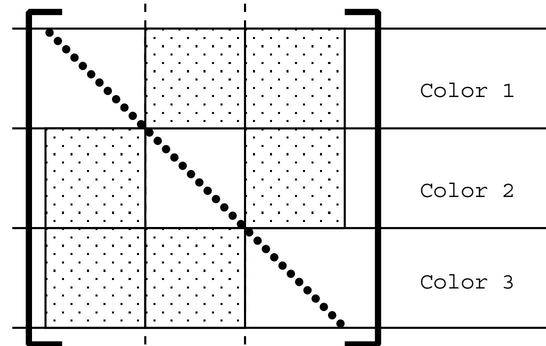


図 1　マルチカラーオーダリング法における係数行列
Fig. 1　Coefficient matrix in multi-color ordering method

2.2　差分解析ベンチマーク

本ベンチマークでは，次式で与えられる 2 次元ポアソン方程式の境界値問題の差分解析を用いる．

$$
\begin{aligned}
&-\nabla\cdot(\kappa\nabla u(x,y)) = f\ in\ \Omega(0,1)\times(0,1) \qquad (1)\\
&u(x,y) = 0\ on\ \delta\Omega\\
&if\ (\frac{1}{4}\leq x\leq\frac{3}{4}\ \&\ \frac{1}{4}\leq y\leq\frac{3}{4})\ then\\
&\kappa = 100\\
&else\quad \kappa = 1
\end{aligned}
$$

ここで f は, 格子点を辞書式順序付けで並べた場合の格子番号を k として, $0.5\sin(k+1)$ である．未知数の個数は 1001×1001 とする．式 (1) を 5 点差分公式により離散化し, 得られる連立一次方程式を ICCG 法により解く．ここで, 解析プログラム上において, 係数行列と前処理行列は辞書式順序付け法に基づきそれぞれ 3 つの一次元配列と 1 つの一次元配列に格納される[10)]．また，本ベンチマークではマルチカラーオーダリングの実装は，Washio, Hayami が文献 9 に示している間接参照を必要とせず，計算順序のみを入れ替える方法を用いる．その結果，本ベンチマークにおいて前進・後退代入計算は色数をストライド幅とするストライドアクセスをもつ計算手順により行われる．図 2 に本ベンチマークにおける前進代入計算のプログラムを示す．図 2 において，myid は各スレッドのスレッド ID を表す．また，icolor は色数であり，is(myid), ie(myid) は各スレッドが計算する領域の開始行番号と終了行番号を表す．内側のループが各スレッドで並行的に行われ，その後バリア同期をとる．この操作が色数の数だけ行われる．なお，後退代入計算も同様の手順で並列化される．

2.3　有限要素解析ベンチマーク

本ベンチマークでは，電磁場解析の一種である 3 次元渦電流場の解析を対象とする．解析対象内の電磁界を記

```
!$OMP PARALLEL PRIVATE(myid,i,icolo)
:
:
do icolo=1,icolor-2
 do i=is(myid)+icolo,ie(myid),icolor
 z(i)=z(i)-b(i-1)*z(i-1)/diag(i-1) &
       -c(i-nx)*z(i-nx)/diag(i-nx)
enddo
!$OMP BARRIER
enddo
```

図 2 マルチカラーオーダリング法による並列化前進代入計算（差分解析ベンチマーク）

Fig. 2 Parallelized forward substitution by multi-color ordering method (finite difference analysis benchmark).

述する方程式は, マクスウェル方程式において変位電流の項を無視することにより与えられる. 本解析では, 辺要素を使用し, 磁気ベクトルポテンシャルのみによる定式化を行う A-法を用いるので, 支配方程式は次式で与えられる.

$$\nabla \times (\nu \nabla \times \boldsymbol{A}_m) = -\sigma \frac{\partial \boldsymbol{A}_m}{\partial t} + \boldsymbol{J}_0 \quad (2)$$

ここで, $\boldsymbol{A}_m$ は磁気ベクトルポテンシャル, $\boldsymbol{J}_0$ は強制電流の電流密度, ν は磁気抵抗率, σ は導電率を表す. 磁気ベクトルポテンシャルをベクトル補間関数により近似展開し, 式 (2) にガラーキン法を適用することにより, 次式が得られる.

$$[K]\{A_m\} + [M_A]\frac{\partial \{A_m\}}{\partial t} - \{J\} = 0 \quad (3)$$

ここで, $\{A_m\}$ は未知変数 A_{mi} からなる列ベクトルを表す. $[K]$, $[M_A]$ は行列, $\{J\}$ は列ベクトルを表し, 以下のように与えられる.

$$K_{ij} = \sum_e^m \iiint_e (\nabla \times \boldsymbol{N}_i) \cdot (\nu \nabla \times \boldsymbol{N}_j) dV \quad (4)$$

$$M_{Aij} = \sum_e^m \iiint_e \sigma \boldsymbol{N}_i \cdot \boldsymbol{N}_j dV \quad (5)$$

$$J_i = \sum_e^m \iiint_e \boldsymbol{N}_i \cdot \boldsymbol{J}_0 dV \quad (6)$$

ここで, e は各要素, m は全要素数, $\boldsymbol{N}$ はベクトル補間関数を表す. 未知変数の総数を n として, 行列 $[K]$, $[M_A]$ は n 次正方行列, $\{A_m\}$ および $\{J\}$ は n 次元ベクトルである. 式 (3) 中の時間微分項を後退差分法により解くと,

$$[Q]\{A_m\} = \{f\} \quad (7)$$

但し,

$$[Q] = ([K] + \frac{1}{\Delta t}[M_A]), \quad (8)$$

```
!$OMP PARALLEL
:
:
do icolo=1,icolor-2
!$OMP DO PRIVATE(jj)
do i=icspo(ic),icspo(ic+1)-1
  do j=lnrowptr(i),lnrowptr(i+1)-1
  jj=llnt(j)
  z(i)=z(i)-z(jj)*alic(j)/adic(jj)
  enddo
enddo
enddo
```

図 3 マルチカラーオーダリング法による並列化前進代入計算（有限要素解析ベンチマーク）

Fig. 3 Parallelized forward substitution by multi-color ordering method (finite element analysis benchmark).

表 1 有限要素解析ベンチマーク（3 次元渦電流解析）の諸元

Table 1 3-d eddy-current analysis test model

Number of unknowns	1011920
Number of edge elements	327680
Number of nodes	342225

$$\{f\} = \frac{1}{\Delta t}[M_A]\{A_{mold}\} + \{J\} \quad (9)$$

の連立一次方程式が得られる．ここで，本稿では解析対象として電気学会 3 次元渦電流解析モデル[13] を用いる．表 1 に解析の諸元を示す．本解析では，解析領域中に非導電性の部分（空気領域）が含まれるため, 係数行列 $[Q]$ は半正定値となる.

本解析では，時間発展問題のある 1 ステップをベンチマーク問題とする．本解析のような辺要素を用いた電磁場解析では，係数行列は正値性を失っている場合がほとんどあり，ICCG 法をそのまま用いることができない．そこで，加速パラメータを 1.3 としたシフト付き ICCG 法[14] を用いることとする.

本解析は非構造型の解析であるため，並列化 ICCG 法ソルバにおける行列・ベクトル積演算や前進・後退代入計算では間接アドレッシングが用いられる．また，マルチカラーオーダリング法の実装については未知変数を陽的に並び替える手法を使用した．このとき，前進代入計算の並列化プログラムは図 3 のように与えられる．図中において，icspo(ic) は ic 番目の色の未知変数の開始番号であり，各色毎に代入計算が並列化される.

3. 数値実験

3.1 実行環境

本ベンチマークは京都大学学術情報メディアセンターの富士通 HPC2500（SPARC64V 1.3GHz）上で行った．プ

ログラムはFORTRAN90により書かれ，並列処理のAPIとしてOpenMPを用いている．コンパイル時の最適化オプションには-Kfast_GP2=3を指定した．また，ICCG法の収束基準として右辺ベクトルノルムと残差ノルムの比を用い，その値が 10^{-7} 以下となった時点で収束とみなす．

3.2 差分解析ベンチマーク結果

表2に差分解析ベンチマークの結果を示す．なお，表中において計算時間は反復の開始から終了までの間の経過時間を示しており，不完全コレスキー分解などの反復解法部のセットアップ部分の時間は含まれていない．まず，マルチカラーオーダリング法による並列化ICCG法の収束性については，色数を増やすほど向上しており，従来の研究結果と合致している[5),10),11)]．差分解析におけるオーダリングの影響については，文献11において，Incompatible nodeの数が多いほど収束性が悪化することが示されており，マルチカラーオーダリングにおいて色数が増加するに従い収束性が増すことを説明することができる．また，著者らの一部も文献12において，オーダリングと収束性の関係を示す指標を提案しており，その値からも本現象を説明することができる．一方，表3に1CPU時における1反復あたりの計算時間と色数の関係を示すが，色数が増すにつれて大幅に1反復あたりの計算時間を要していることがわかる．これはストライド幅が増加することにより，キャッシュのヒット率が大きく低下したことによる．ベクトル計算機上での実行では，バンクコンフリクトを起こす場合を除けば1反復あたりの計算時間は色数に対してあまり変化しないため，高い収束性が得られる色数の多い場合が有効となるがスカラ型の計算機では注意が必要である．次に，速度向上については概ね高い並列化効率を得ている．特に色数が多い場合にはスーパーリニアな性能を示しており，色数100の場合には顕著である．これは，並列化によりデータアクセスの局所性が高まりキャッシュのヒット率が向上したためである．プロファイラによる分析では，色数100の場合の代入計算ループについて1CPU時と16CPU時で，L1キャッシュミス率，L2キャッシュミス率，TLBミス率がそれぞれ23% → 1.5%，31% → 0.3%，1.86% → 0.0031% のように大幅に向上している．これらの結果から上記のような速度向上が得られたものと考えられる．最後に総合的な計算性能を考えると，本解析では色数100，CPU数16の場合が最もよい結果となった．これは，色数を多くした場合でも十分に各プロセッサが扱うデータサイズが小さい場合にはキャッシュのヒット率が向上し，収束性に優れた色数の多い場合が有効であることを示している．しかし，問題サイズに対してプロセッサ数が十分ではない場合には，最小の色数である2色の場合が有効となると考えられる．

3.3 有限要素解析ベンチマーク結果

表4に有限要素解析ベンチマークの結果を示す．なお，表中において計算時間は反復の開始から終了までの間の経過時間を示している．まず，色数と反復回数との関係では，40から120程度の色数の変化ではそれほど大きな差はなかった．但し，色数を500以上とした場合には改善が見られることが分かっている．次に，1反復あたりの計算時間については，色数が増大するに従って増加しているが差分解析ほど顕著ではない．これは色数の増加に従ってキャッシュのヒット率が下がることが原因と考えられるが，差分解析と比べて未知変数間のデータ関係が

表2 差分解析ベンチマーク結果
Table 2 Benchmark result of finite difference analysis

(a) 色数2の場合

CPU数	計算時間（秒）	反復回数	速度向上
1	248	1600	1.0
2	138	1600	1.80
4	72.3	1599	3.42
8	45.3	1599	5.48
12	27.5	1599	9.03
16	18.9	1600	13.1

(b) 色数4の場合

CPU数	計算時間（秒）	反復回数	速度向上
1	304	1333	1.0
2	176	1332	1.73
4	92.3	1331	3.29
8	61.3	1331	4.95
12	33.3	1332	9.12
16	20.1	1331	15.1

(c) 色数8の場合

CPU数	計算時間（秒）	反復回数	速度向上
1	444	1176	1.0
2	258	1176	1.72
4	136	1176	3.25
8	95.8	1171	4.61
12	43.7	1171	10.2
16	21.7	1174	20.4

(d) 色数20の場合

CPU数	計算時間（秒）	反復回数	速度向上
1	595	1074	1.0
2	327	1073	1.82
4	185	1070	3.22
8	133	1069	4.47
12	57.8	1068	10.3
16	23.7	1073	25.1

(e) 色数100の場合

CPU数	計算時間（秒）	反復回数	速度向上
1	773	1012	1.0
2	314	1013	2.47
4	98.1	1012	7.88
8	39.0	1012	19.8
12	23.2	1012	33.3
16	14.6	1012	52.9

表3 1反復あたりの計算時間と色数の関係
Table 3 Relationship between computational time in one iteration and number of colors

色数	2	4	8	20	100
計算時間（ミリ秒）	155	228	378	554	764

表 4　有限要素解析ベンチマーク結果
Table 4　Benchmark result of finite element analysis
(a) 色数 40 の場合

CPU 数	計算時間（秒）	反復回数	速度向上
1	823	496	1.0
2	416	496	1.97
4	212	496	3.88
8	115	496	7.14
12	77.6	496	10.6
16	61.1	496	13.5

(b) 色数 80 の場合

CPU 数	計算時間（秒）	反復回数	速度向上
1	899	509	1.0
2	454	509	1.98
4	228	509	3.95
8	122	509	7.39
12	83.4	509	10.8
16	67.1	509	13.4

(c) 色数 120 の場合

CPU 数	計算時間（秒）	反復回数	速度向上
1	1190	497	1.0
2	592	497	2.01
4	288	497	4.13
8	149	497	7.96
12	101	497	11.7
16	81.6	497	14.6

複雑な有限要素解析では元来ランダムアクセスとなっているためにその低下率は小さい．次に並列化効率については全体的に高い値を得ており，マルチカラーオーダリング法の有効性を示している．総合的なベンチマーク性能では，色数 40 の場合が最も計算時間が短かった．

次に，HPC2500 が備えるラージページ機能について本ベンチマークにより評価を行った．当該の機能はページサイズを大きくすることにより，大きな配列を扱う科学技術計算において TLB ミスの軽減を狙った機能である．表 5 にその結果を示す．また，図 4 に本ベンチマークにおいて色数 40，CPU 数を 1 とした場合を基準とする速度向上を示す．表 5，図 4 より，いずれの色数の場合にも性能が改善され，ラージページ機能が有効であることが分かる．

4. おわりに

本稿では，マルチカラーオーダリング法による並列化 ICCG 法ソルバベンチマークにより HPC2500 の性能評価を行った．色数をストライド幅とするストライドアクセスをもつ差分解析ベンチマークでは，色数の増加に従い，キャッシュのヒット率が著しく低下し，1 反復あたりの計算時間が増加する現象が見られた．しかし，使用プロセッサ数を増加することによりアクセスするデータの局所性が高まり，スーパーリニアな並列台数効果を得ることにより，色数が大きい場合でも問題サイズが適度に小さければ高い性能を得ることができることがわかった．有限要素解析ベンチマークにおいても差分解析の場合と

表 5　有限要素解析ベンチマークにおけるラージページの効果
Table 5　Effects of use of largepage in finite element analysis benchmark
(a) 色数 40 の場合

CPU 数	計算時間（秒）	反復回数	速度向上
1	749	496	1.0
2	382	496	1.96
4	195	496	3.84
8	111	496	6.77
12	73.2	496	10.2
16	57.9	496	13.0

(b) 色数 80 の場合

CPU 数	計算時間（秒）	反復回数	速度向上
1	815	509	1.0
2	413	509	1.97
4	208	509	3.91
8	116	509	7.03
12	77.3	509	10.5
16	61.2	509	13.3

(c) 色数 120 の場合

CPU 数	計算時間（秒）	反復回数	速度向上
1	1076	497	1.0
2	536	497	2.00
4	261	497	4.13
8	142	497	7.56
12	92.8	497	11.6
16	73.1	497	14.7

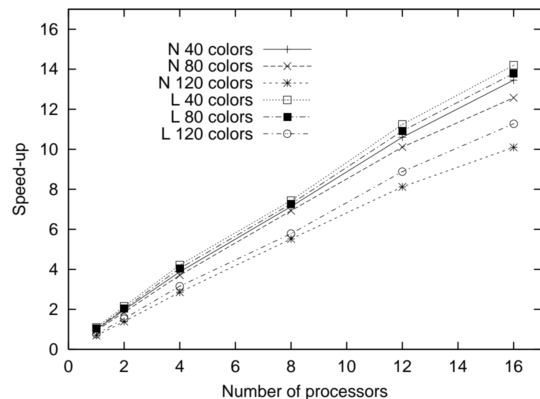


図 4　有限要素解析ベンチマークにおける速度向上（N: ノーマルページ使用，L: ラージページ使用）
Fig. 4　Speed-up in finite element analysis benchmark (N: normal page case, L: large page case)

同様の傾向が見られたが，色数に対する 1 反復あたりの計算時間の変化は差分解析の場合と比べて小さかった．また，本ベンチマークでは HPC2500 が備える Largepage 機能について評価を行ったが，100 万自由度の当該ベンチマーク問題において性能改善が見られた．

参 考 文 献

1) 杉崎　由典, 青木　正樹, 義久　智樹, 金澤　正憲,「SMP におけるスレッド並列の台数効果と高速化手法について」, 情報処理学会　研究報告, 2005-EVA-14, (2005).
2) J. Meijerink and H. A. van der Vorst, “An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix Is a Symmetric M-matrix,” *Mathematics of Computation*, 31, (1977), pp. 148-162.

3) H.A. van der Vorst and T.F. Chan, "Parallel Preconditioning for Sparse Linear Equations", *ZAMM. Z. angew. Math. Mech.*, 76 (1996), pp. 167–170.
4) Y. Saad, "Iterative Methods for Sparse Linear Systems", Second ed., SIAM, Philadelphia, PA, 2003.
5) S. Doi and T. Washio, "Ordering Strategies and Related Techniques to Overcome the Trade-off Between Parallelism and Convergence in Incomplete Factorization", *Parallel Computing*, 25, (1999), pp. 1995-2014.
6) K. Nakajima, "Preconditioned Iterative Linear Solvers for Unstructured Grids on the Earth Simulator ", HPC Asia Proceedings, (2004), pp. 150-169.
7) I. S. Duff and G. A. Meurant, "The Effect of Ordering on Preconditioned Conjugate Gradients", *BIT*, 29, (1989), pp.635-657.
8) T. Iwashita and M. Shimasaki, "Algebraic Multi-color Ordering for Parallelized ICCG Solver in Finite Element Analyses," *IEEE Trans. Magn.*, vol. 38, (2002), pp. 429-432.
9) T. Washio and K. Hayami, "Overlapped Multicolor MILU Preconditioning," *SIAM Journal on Scientific Computing*, 16, (1995), pp. 636-650.
10) 岩下　武史，島崎　眞昭；「同期点の少ない並列化 ICCG 法のためのブロック化赤ー黒順序付け」，情報処理学会論文誌，Vol.43 No. 4，(2002)，pp. 893-904.
11) S. Doi and A. Lichnewsky, "A Graph-Theory Approach for Analyzing the Effects of Ordering on ILU Preconditioning," INRIA report 1452, (1991).
12) Iwashita, T., Nakanishi, Y. and Shimasaki, M.: Comparison Criteria for Parallel Orderings in ILU Preconditioning, *SIAM J. Sci. Comput.*, Vol.26, No.4, pp.1234–1260 (2005).
13) T. Nakata, N. Takahashi, T. Imai, and K. Muramatsu, "Comparison of Various Methods of Analysis and Finite Elements in 3-D Magnetic Field Analysis," *IEEE Trans. Magn.*, vol. 27, (1991), pp.4073-4076.
14) K. Fujiwara, T. Nakata, and H. Fusayasu, "Acceleration of Convergence Characteristic of the ICCG Method," *IEEE Trans. Magn.*, vol. 27, (1993), pp.1958-1961.

2.7 スレッド並列プログラムの性能測定

名古屋大学情報連携基盤センター
永井　亨

1. はじめに

名古屋大学情報連携基盤センターの Fujitsu PRIMEPOWER HPC2500（CPU の動作周波数 2.08Hz、総 CPU 数 1664、ノード数 24、トータルの理論最大性能 13.5Tflops、総主記憶容量 12TB）を使用してスレッド並列性能を測定した。本報告における測定では 1 ノード（64CPU、主記憶 512GB）を占有しておこなった。

2. 行列積

2-1. コンパイラオプション

スレッド数を 60 まで変化させたときの 4096×4096 行列積の経過時間を測定した。測定はそれぞれ 10 回おこない、その平均値をとった。これを flops 値にしたものを図 1 に示す。配列の宣言は

```
dimension a(4096+1,4096),b(4096+1,4096),c(4096+1,4096)
```

で、すべて倍精度実数型である。

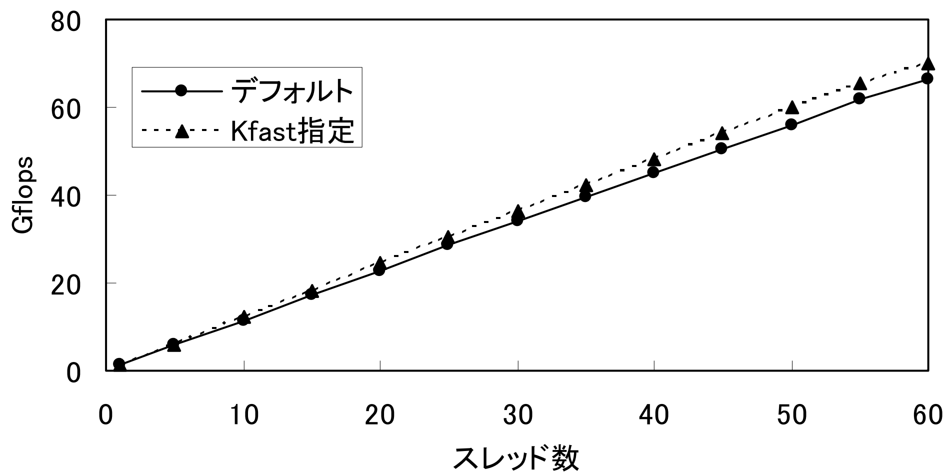


図 1

図 1 にはコンパイラの最適化オプションをデフォルト値（–Kfast_GP2=3,largepage=2,V9）のまま実行した場合（これは通常のプログラムでは最も性能が出る富士通推奨のオプションセット）と–Kfast を指定した場合の測定結果が示されている。デフォルトのほうが–Kfast を指定した場合よりも 7%程度おそい。これは–Kfast を指定した場合には 32 ビットアドレッシングモード（–KV8PLUS）が有効になるのに対し、デフォルトの 64 ビットアドレッシングモードでは 4 バイト整数から 8 バイト整数への変換やアドレス計算の増加、それに伴う一部最適化の違いなどの影響により 32 ビットアドレッシングモードより実行性能が低下する場合があるためである。

2-2. パディング

図 1 では 50 スレッドで約 60Gflops となっている。1CPU あたり 8Gflops の理論最大性能を持つことを考えるとこれはかなり低い値である。そこで、52 スレッドの場合について、配列宣言を

```
dimension a(ndim+m,ndim),b(ndim+m,ndim),c(ndim+m,ndim)
```

とし、ndim が 1024、2048、4096 のときに m を 1 から 10 まで変えてそれぞれ測定した。結果

を表 1 に示す。測定はそれぞれ 10 回おこない平均値をとった。表 1 にはコンパイラオプションとしてデフォルトの場合と–Kfast を指定した場合とが示されている。パディングの大きさに依存して性能が改善されるのはキャッシュ競合が緩和されるためと考えてよい。4096×4096 の m=1 では L1 キャッシュミスが大きくなるため、ほかの場合に比べて性能が低下する。

表 1　（単位は Gflops）

m	1024×1024		2048×2048		4096×4096	
	デフォルト	–Kfast	デフォルト	–Kfast	デフォルト	–Kfast
1	83.2	193.5	145.2	150.6	58.0	61.9
2	86.9	202.6	149.4	152.7	121.8	122.5
3	90.6	204.5	143.4	149.7	134.5	133.1
4	82.3	156.8	137.4	143.4	156.4	157.6
5	91.8	204.5	138.3	144.4	152.2	153.4
6	78.7	204.5	142.0	145.0	157.0	157.3
7	91.4	204.5	141.3	142.6	157.9	157.4
8	71.8	127.8	132.9	136.7	152.9	154.7
9	91.8	202.6	137.8	141.2	159.7	161.3
10	92.2	204.5	140.5	145.0	156.6	157.2

表 1 で–Kfast を指定した場合の 1024×1024 がほかの場合よりも性能が大きく上回ることは注目される。これは、主にコンパイル時に組み込まれる matmul 関数の処理論理に起因する。matmul 関数は、CPU 数により配列を最適な小行列に分割して行列積を実行する。 HPC2500 では L2 キャッシュは 4MB (1MB×4WAY)であるのにたいして、たとえば、52 スレッドでは 1024×1024 行列の matmul 関数内での小行列サイズは 2.8MB 程度になるが、2048×2048 行列および 4096×4096 行列の場合のサイズは 4MB を越える。このため後者では L2 キャッシュミスが増加し、性能が低下する。ちなみに 52 スレッドで実測した際のプロファイラが出力した L2 キャッシュミス率を表 2 に示す。

表 2

配列宣言	L2 キャッシュミス率(%)
(1024+10,1024)	0.0356
(2048+10,2048)	0.1051
(4096+10,4096)	0.1076

富士通によれば、行列積(matmul)の性能見積もりは 1CPU あたり 4.6Gflops 程度である。したがって、52 スレッド×4.6 = 239Gflops であるから–Kfast を指定した場合の 1024×1024 行列の値は妥当であるといえる。そこで m を 10 より大きい値にすることによりほかの場合でも 性能がさらに向上するか測定してみた。52 スレッドで m の値を 20、50、100 と変えた場合の表 3 に示す。表 1 と比べると、1024×1024 行列と 4096×4096 行列では明らかに改善されている。実効性能をあげるにはパディングの大きさを適切に選ぶことが重要である。

表 3　（単位は Gflops）

m	1024×1024		2048×2048		4096×4096	
	デフォルト	–Kfast	デフォルト	–Kfast	デフォルト	–Kfast
20	176.0	200.7	139.8	146.1	164.0	169.1
50	146.1	150.2	138.0	148.2	168.4	168.9
100	195.2	202.6	151.0	153.0	180.6	183.0

3．2 次元拡散方程式

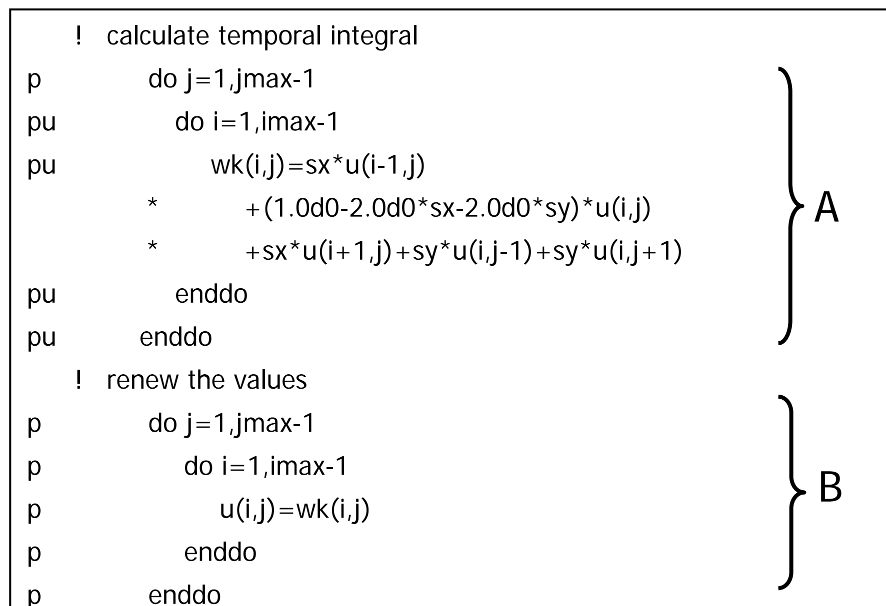
2 次元拡散方程式(1)を有限差分法で解く問題を考える。

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \qquad (\alpha > 0) \tag{1}$$

時間を 1 次精度前進差分、空間を 2 次精度中心差分で離散化し、$t = k\Delta t$ 、 $x = i\Delta x$ 、 $y = j\Delta y$ 、とおいて $u(t, x, y)$ を $u_{i,j}^{k}$ と表すと、

$$u_{i,j}^{k+1} = s_x u_{i+1,j}^{k} + (1 - 2s_x - 2s_y) u_{i,j}^{k} + s_x u_{i-1,j}^{k} + s_y u_{i,j+1}^{k} + s_y u_{i,j-1}^{k} \tag{2}$$

となる。ただし、$s_x = \alpha \Delta t / \Delta x^2$ 、 $s_y = \alpha \Delta t / \Delta y^2$ である。式(2)の計算は図 2 のように書ける。時間積分を行う A の部分と、積分値の更新を行う B の部分の経過時間を測定した結果を表 4 に示す。ただし、$0 \le i \le 2000$ 、$0 \le j \le 2000$ とし、A および B をそれぞれ 1000 回繰り返して平均値をとった。表 4 をみると A の部分と B の部分とで処理時間に大きな差はないことがわかる。prefetch 機能はコンパイラの標準オプションとなっているため、より高速に実行するためにはメモリコピーをおこなう B の部分をなくすプログラミングが必要になる。B の部分をなくす方法としては以下の 3 つが考えられるであろう。



```
    !  calculate temporal integral
p          do j=1,jmax-1
pu            do i=1,imax-1
pu               wk(i,j)=sx*u(i-1,j)
          *         +(1.0d0-2.0d0*sx-2.0d0*sy)*u(i,j)
          *         +sx*u(i+1,j)+sy*u(i,j-1)+sy*u(i,j+1)
pu            enddo
pu         enddo
    !  renew the values
p          do j=1,jmax-1
p             do i=1,imax-1
p                u(i,j)=wk(i,j)
p             enddo
p          enddo
```


図 2

表 4　（単位は msec）

スレッド数	A	B
1	42.07	41.77
10	6.20	6.11
20	0.67	0.47
30	0.46	0.33
40	0.35	0.27
50	0.30	0.23
60	0.26	0.21

[案 1]

図 3 に示すように u を 3 次元配列に変更し、計算ステップ数 istep が奇数のとき k1=1、k2=2、偶数のとき k1=2、k2=1 としてスイッチする。ただし、この場合には do ループの直前に指示行を挿入し、配列 u が回帰参照ではないことを明示する必要がある。指示行を有効にするためコンパイルオプションに–Kocl,vppocl を追加した。

```
           dimension u(:,:,2)
                 :
           if(mod(istep,2).eq.1) then
              k1=1
              k2=2
           else
              k1=2
              k2=1
           endif
       !ocl norecurrence(u)
p          do j=1,jmax-1
pu            do i=1,imax-1
pu               u(i,j,k2)=sx*u(i-1,j,k1)+(1.0d0-2.0d0*sx-2.0d0*sy)*u(i,j,k1)
         *             +sx*u(i+1,j,k1)+sy*u(i,j-1,k1)+sy*u(i,j+1,k1)
pu            enddo
p           enddo
```

図 3

[案 2]

図 4 に示すように、計算ステップ数 istep の奇偶よって、u の 3 次元目の値を明示的に指定する。通常のコンパイルオプションのみで並列化される。

```
           dimension u(:,:,2)
                 :
           if(mod(istep,2).eq.1) then
p             do j=1,jmax-1
pu               do i=1,imax-1
pu                  u(i,j,2)=sx*u(i-1,j,1)+(1.0d0-2.0d0*sx-2.0d0*sy)*u(i,j,1)
         *                +sx*u(i+1,j,1)+sy*u(i,j-1,1)+sy*u(i,j+1,1)
pu               enddo
p             enddo
            else
p             do j=1,jmax-1
pu               do i=1,imax-1
pu                  u(i,j,1)=sx*u(i-1,j,2)+(1.0d0-2.0d0*sx-2.0d0*sy)*u(i,j,2)
         *                 +sx*u(i+1,j,2)+sy*u(i,j-1,2)+sy*u(i,j+1,2)
pu               enddo
p             enddo
            endif
```

図 4

[案 3]

図 5 に示すように、同じサイズの 2 次元配列 u1、u2 を用意し、計算ステップ数 istep の奇偶よって左辺と右辺の配列を入れ替える。この場合も通常のコンパイルオプションのみで並列化される。

案 1〜案 3 までを実測した結果を表 5 に示す。比較のため、図 2 の A と B の部分の処理時間（表 4 の A、B および A+B）もならべて示した。また、表 5 の A+B の 1 スレッドの経過時間を 1 としたときの台数効果を図 6 に示す。案 1 から案 3 までのどの場合も明らかな性能向上がみてとれる。案 1 と案 3 は値がほとんど同じであるためグラフが重なって一本の線のようにみえる。案 2 が他の 2 つより若干性能が下回るのはループ内のレジスタ見積りに関するコンパイラの最適化状況に差があるためである。なお、いずれの場合についても配列 u の 1 次元目のパディングの大きさを変えて実行してみたが、有意な性能差はみられなかった。

これらの結果より作業領域をもちいずに時間積分するプログラミングが有効であることが示された。もちろん、ここで示した例は非常に単純なプログラムであるから、より実用的なプログラムで検証する必要がある。

```
                dimension u1(:,:),u2(:,:)
                      :
                if(mod(istep,2).eq.1) then
p                  do j=1,jmax-1
pu                    do i=1,imax-1
pu                       u2(i,j)=sx*u1(i-1,j)+(1.0d0-2.0d0*sx-2.0d0*sy)*u1(i,j)
               *            +sx*u1(i+1,j)+sy*u1(i,j-1)+sy*u1(i,j+1)
pu                    enddo
p                  enddo
                else
p                  do j=1,jmax-1
pu                    do i=1,imax-1
pu                       u1(i,j)=sx*u2(i-1,j)+(1.0d0-2.0d0*sx-2.0d0*sy)*u2(i,j)
               *            +sx*u2(i+1,j)+sy*u2(i,j-1)+sy*u2(i,j+1)
pu                    enddo
p                  enddo
                endif
```

図 5

表 5 （単位は msec）

スレッド数	図 2			案 1	案 2	案 3
	A	B	A+B			
1	42.07	41.77	83.84	41.51	41.99	41.77
10	6.20	6.11	12.31	7.16	7.16	7.14
20	0.67	0.47	1.14	0.72	0.82	0.73
30	0.46	0.33	0.79	0.51	0.57	0.50
40	0.35	0.27	0.62	0.40	0.45	0.40
50	0.30	0.23	0.53	0.34	0.38	0.34
60	0.26	0.21	0.47	0.31	0.35	0.31

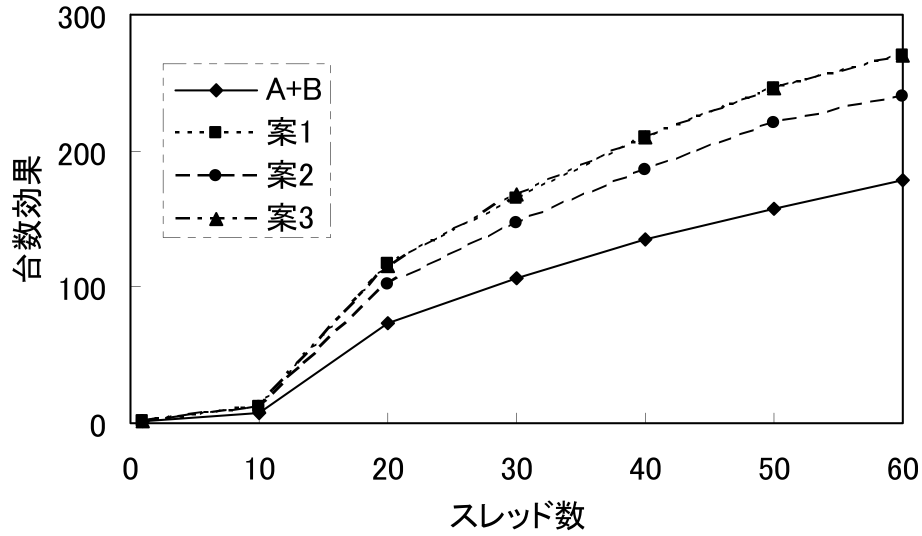


図 6

4. おわりに

簡単な 2 つのプログラム（行列積および 2 次元拡散方程式）を使用して HPC2500 上でのスレッド並列処理性能を測定した結果を報告した。SMP 上で処理性能を向上させるにはキャッシュミスの有無を意識したプログラミングが必要である。陽解法の場合に作業領域を用いずに時間積分する手法が実用的プログラムにおいても有効かどうかは今後の課題である。

Ⅲ. プロファイラ情報の Cover(％)について

富士通株式会社
鈴木 清文

以下の報告は、「2.3 非構造格子Euler/Navier-StokesソルバJTASのスレッド並列最適化」の資料に関し、プロファイラ情報の「cover(%)」が100%にならないのはなぜか、との質問があがり、それに対して回答したものである。

(WG 会議室 投稿日：2005/04/28(Thu))

網羅率(Cover(%))が 100%にならない要因には以下のようなものがあります。

- データメイン TLB ミス率が高い
- プログラムの実行時に入出力処理が多い
- プログラムの実行時に領域の確保・解放処理が多い

目安としては、網羅率がおよそ 90%以上であれば、そのハードウェアモニタ情報はハードウェアの実際の動作をほぼ網羅しており信頼してよいと言えます。

もう少し詳しい説明は、プログラミング支援ツール使用手引書の「8.3.2.2 ハードウェアモニタ情報を利用した改善」の「網羅率」の項に書かれていますので、ご参照下さい。

以上

Ⅳ. PAカウンタによる性能情報採取方法および注意点について

富士通株式会社
杉崎　由典

プログラム中の特定の部分について PA カウンタにより性能情報を採取する際には、測定対象箇所を PA サービスルーチン(hpc_start/hpc_stop)で挟み、測定対象箇所の正確な性能情報を採取します。

PA カウンタ情報を thread 毎に採取するためには、情報採取を行う各 thread 上で hpc_start/hpc_stop を呼び出す必要があります。

【性能情報採取方法】

スレッドごとに hpc_start/hpc_stop を実行させるには二つの方法(自動並列と OpenMP)があります。
以下に具体的な方法を示します。

(1)OpenMP を用いた方法

```
----------------------------------------
          …
!$omp parallel              -+
          call hpc_start(1)   |各 thread で hpc_start()を実行
!$omp end parallel          -+

          測定対象部            !!この部分の PA 情報を採取したい

!$omp parallel              -+
          call hpc_stop(1)    |各 thread で hpc_stop()を実行
!$omp end parallel          -+
          …
----------------------------------------
```

(2)自動並列を用いた方法

```
----------------------------------------
          …
!ocl   independent(hpc_start)
          do i=1,N              -+
             call hpc_start(1)   |各 thread で hpc_start()を実行
          end do                -+

          測定対象部            !!この部分の PA 情報を採取したい

!ocl   independent(hpc_stop)
          do i=1,N              -+
             call hpc_stop(1)    |各 thread で hpc_stop()を実行
          end do                -+
          …
----------------------------------------
```

以上

Ⅴ．構造体とアロケータブル配列の組合せにおける制約について

富士通株式会社
鈴木 清文

この報告は、「2.2 三次元圧縮性流体解析プログラム UPACS の性能評価」の自動並列化状況調査において、ポインタ配列をアロケータブル配列に書き換えている際に翻訳時エラー（文法エラー）となった現象が発生したため、構造体にアロケータブル配列を指定する場合の制約事項について報告したものです。

1．アロケータブル配列の指定方法

UPACS において、構造体の中の配列ポインタが自動並列化の阻害要因かどうかを確認するため、ソース中で、構造体成分が配列ポインタで宣言された箇所を、同じ動的割り当て方式のアロケータブル配列に書き換えました。
ソース書き換え例を以下に示します。

書き換え前（配列ポインタ）	書き換え後（アロケータブル配列）
type blockDataType ・・・ real(8),**pointer**,dimension(:,:,:) :: inv_vol real(8),**pointer**,dimension(:,:,:,:,:):: fNormal,xix ・・・ end type blockDataType	type blockDataType ・・・ real(8),**allocatable**,dimension(:,:,:) :: inv_vol real(8),**allocatable**,dimension(:,:,:,:,:):: fNormal,xix ・・・ end type blockDataType

2．アロケータブル配列指定時の制約事項

書き換えは宣言文のみで、大部分の実行文は変更不要ですが、例外的に以下の箇所は翻訳時エラーになりました。

```
  68              real(8),allocatable,dimension(:,:,:,:,:):: fNormal,xix
 ・・・             ・・・
 186              type(blockDataType),intent(inout):: blk
 187              character(len=*),intent(in):: name
 188              real(8),pointer,dimension(:,:,:,:,:):: dp
 189    1         select case(name)
 190    1         case('fNormal')
 191    1           dp => blk%fNormal
 192    1         case('xix')
 193    1           dp => blk%xix
 194    1         case('cellVrtx')
 195    1           dp => blk%cellVrtx

jwd2395i-s "base_blockDataType.f90", line 191, column 13: ポインタ代入文の指示先は,TARGET 属性をもつか,
TARGET 属性をもつ実体の部分実体であるか,または POINTER 属性をもたなければなりません.
jwd2395i-s "base_blockDataType.f90", line 193, column 13: ポインタ代入文の指示先は,TARGET 属性をもつか,
TARGET 属性をもつ実体の部分実体であるか,または POINTER 属性をもたなければなりません.
```

つまり、ポインタ代入文で構造体成分のアロケータブル配列を指示するのは制約事項になります。

通常、アロケータブル配列をポインタ代入文で指示するには、宣言時に ALLOCATABLE 属性に加え、TARGET 属性を付加しますが、構造体成分の場合、ALLOCATABLE 属性と TARGET 属性は同時に指定できない仕様になっています。

3．書き換え例

以下のように、構造体全体に TARGET 属性を指定することで翻訳可能となります。

```
  68              real(8),allocatable,dimension(:,:,:,:,:):: fNormal,xix
 ・・・             ・・・
 186              type(blockDataType),intent(inout),target:: blk
```

以上

Ⅸ．２次元拡散方程式のスレッド並列チューニングについて

富士通株式会社
谷村 恭伸

この報告は、「2.7 スレッド並列プログラムの性能測定」内の「３．２ 次元拡散方程式」のチューニング初期段階において発生した性能問題に関しての分析結果をまとめたものです。なお、この版には、最終版に存在しない暫定的に記述したデータコピー処理が含まれているため、それが影響して発生した問題も含まれています。

指摘された性能問題は、表１の測定値における以下の３点です。
⑴ 10 並列から 20 並列で、スケーラビリティが 2 倍以上出ている
⑵ 案 1，2，3 の 60 並列が遅い
⑶ 案 1 と案 2 の 20 並列が遅い
表中のプログラムは、それぞれ図１～図４に対応します。案１～案３のチューニングは、オリジナルの B の部分のコストが大きいので、これを解消するために A と B の部分を１つにまとめたチューニングを何パターンか試されたものです。

表１

	案１	案２	案３	オリジナル (A)	オリジナル (B)
1 並列	42.13	42.07	41.65	42.07	42.07
10 並列	7.30	7.29	7.12	6.20	6.20
20 並列	1.75	1.79	0.70	0.67	0.67
30 並列	0.69	0.80	0.62	0.46	0.46
40 並列	0.41	0.46	0.38	0.35	0.35
50 並列	0.35	0.40	0.33	0.30	0.30
60 並列	0.73	0.85	0.62	0.26	0.26

１． 10 並列から 20 並列で、スケーラビリティが 2 倍以上出ている原因について

永井委員の予想どおり、20 並列からキャッシュに乗り切るようになるためです。
対象ループで使用されているデータの大きさは、約 60M バイト(2001 要素*2001 要素*8 バイト*2 個)です。並列実行時に各 CPU で扱うデータ量は、以下のようになります。
10 並列：約 6M バイト
20 並列：約 3M バイト
したがって、L2 キャッシュの４M バイトに乗り切ることになります。また、プロファイラにより L2 キャッシュミスが軽減されることも確認済みです。

２． 案 1，2，3 の 60 並列が遅い原因について

キャッシュミスが発生するために、パラレルバランスが悪くなっていることが原因です。
同期待ち時間を除いた各スレッドの並列実行時間（パラレルバランス）を確認したところ、各スレッドの実行時間の幅は以下のとおりでした。
50 並列：約 50μ秒
60 並列：約 400μ秒
表１の 50 並列と 60 並列の時間の差分から、これが低下要因であるということが分かります。原因は以下のとおりです。

チューニング版では、サブルーチン diffuse の先頭でのデータコピーのループ(X)が、0～jmax で並列化されているのに対して、測定区間のループ(Y)では、1～jmax-1 で並列化されています。そのため、ループ X とループ Y で、各 CPU に割り当てられる要素の端にずれが生じます。これが、キャッシュミスの要因となり、高並列度での影響が大きくなるためです。しかし、オリジナル版では、配列 wk は 1～jmax-1 でしか書き込みされないので、各 CPU に割り当てられる要素にずれは生じることはありません。

３． 案 1 と案 2 の 20 並列が遅い原因について

コンパイラの最適化状況に違いがあるためです。

これは、定義と参照が同一配列の場合（案 1，2）に、コンパイラでデータ依存関係を見切れないため、定義と参照が別配列の場合（案３、オリジナル）には動作するループの回転を跨った最適化がされていないためです。今後、コンパイラを改善する方向で検討いたします。

図１　　オリジナル（定義と参照で依存があるため、ワーク配列を使用）

```
      parameter (imax=2000,jmax=2000)
      implicit real*8 (a-h,o-z)
      dimension u(0:imax,0:jmax),wk(0:imax,0:jmax)

c ---- step-A ------------------------
      call gettod()
      do j=1,jmax-1
         do i=1,imax-1
            wk(i,j)=sx*u(i-1,j)+(1.0d0-2.0d0*sx-2.0d0*sy)*u(i,j)
      *        +sx*u(i+1,j)+sy*u(i,j-1)+sy*u(i,j+1)
         enddo
       enddo
       call gettod()
c ---- step-B -----------------------
       call gettod()
       do j=1,jmax-1
          do i=1,imax-1
             u(i,j)=wk(i,j)
          enddo
       enddo
       call gettod()
```

図２　　案１（3 次元にして依存をなくした(ocl で指示)）

```
       dimension u(0:imax,0:jmax,2)

       if(kk.eq.1) then
          k1=1
          k2=2
       else
          k1=2
          k2=1
       endif

       do j=0,jmax
          do i=0,imax
             u(i,j,k1)=v(i,j)
          enddo
       enddo

       call gettod()
!ocl norecurrence(u)
       do j=1,jmax-1
          do i=1,imax-1
             u(i,j,k2)=sx*u(i-1,j,k1)+(1.0d0-2.0d0*sx-2.0d0*sy)*u(i,j,k1)
      *         +sx*u(i+1,j,k1)+sy*u(i,j-1,k1)+sy*u(i,j+1,k1)
          enddo
       enddo

       call gettod()

       do j=0,jmax
          do i=0,imax
             v(i,j)=u(i,j,k2)
          enddo
       enddo
```

図３　　案２（案１を、ocl 指定なしにできるように if で分けた）

```
      dimension v(0:imax,0:jmax,2)

      k1=mod(k-1,2)+1

      do j=0,jmax
        do i=0,imax
          v(i,j,k1)=u(i,j)
        enddo
      enddo

      call gettod()
      if(k1.eq.1) then
      do j=1,jmax-1
        do i=1,imax-1
          v(i,j,k1+1)=sx*v(i-1,j,k1)+(1.0d0-2.0d0*sx-2.0d0*sy)*v(i,j,k1)
     *         +sx*v(i+1,j,k1)+sy*v(i,j-1,k1)+sy*v(i,j+1,k1)
        enddo
      enddo
      else
      do j=1,jmax-1
        do i=1,imax-1
          v(i,j,k1-1)=sx*v(i-1,j,k1)+(1.0d0-2.0d0*sx-2.0d0*sy)*v(i,j,k1)
     *         +sx*v(i+1,j,k1)+sy*v(i,j-1,k1)+sy*v(i,j+1,k1)
        enddo
      enddo
      call gettod()

      if(k1.eq.1) then
      do j=0,jmax
        do i=0,imax
          u(i,j)=v(i,j,k1+1)
        enddo
      enddo
      else
      do j=0,jmax
        do i=0,imax
          u(i,j)=v(i,j,k1-1)
        enddo
      enddo
      endif
```

図４　　案３（案２の３次元配列を、２つの２次元配列に分けた）

```
      dimension u1(0:imax,0:jmax),u2(0:imax,0:jmax)

      k1=mod(k,2)

      if(k1.ne.1) then
      do j=0,jmax
        do i=0,imax
          u2(i,j)=u1(i,j)
        enddo
      enddo
      endif

      call gettod()
      if(k1.eq.1) then
      do j=1,jmax-1
        do i=1,imax-1
          u2(i,j)=sx*u1(i-1,j)+(1.0d0-2.0d0*sx-2.0d0*sy)*u1(i,j)
     *          +sx*u1(i+1,j)+sy*u1(i,j-1)+sy*u1(i,j+1)
        enddo
      enddo
      else
      do j=1,jmax-1
        do i=1,imax-1
          u1(i,j)=sx*u2(i-1,j)+(1.0d0-2.0d0*sx-2.0d0*sy)*u2(i,j)
     *          +sx*u2(i+1,j)+sy*u2(i,j-1)+sy*u2(i,j+1)
        enddo
      enddo
      endif
      call gettod()

      if(k1.eq.1) then
      do j=0,jmax
        do i=0,imax
          u1(i,j)=u2(i,j)
        enddo
      enddo
      endif
```

- 以上 -

５．おわりに

本 WG の活動を振り返ってみたとき、会員、富士通の参加者の関心は３つに分類できる。

ア）既存のプログラム、アルゴリズムをチューニングも含めて如何に並列化するか。

イ）そもそもより並列処理に向いたアルゴリズムはないのか。

ウ）次の計算機を考える糧につながるような計算機の評価、プログラムの実行性能の解析はできないのか。

ア）はもっとも現実的な関心事であり、既存の計算機資源をよりよく利用するという観点からも重要である。スカラ機の実効性能に影響を与えるキャッシュや TLB などハードウェアの動きがプログラムからは見えにくいこと、レジスタ割付やプリフェッチの影響を見るにはコンパイラがどのようなオブジェクトを生成しているかを知る必要があるが、今やユーザレベルではオブジェクトコードを解析するのが困難なことなどから、分かり易い性能評価モデルひいてはプログラミングモデルを立てて理解するには至らないことが多い。本報告では宇宙航空研究開発機構の松尾氏が試みている他、“なぜこのような性能になるのか”をできるだけ理解するためにキャッシュミス率や TLB ミス率、メモリアクセス率などを計測しチューニングに対する指針を得た。個別的ではあるが得られたデータを基にした議論は WG に参加して頂いた皆さんの今後の活動に幾ばくかなりとも貢献できることと思う。また、本報告書およびチューニング事例集（SS 研ホームページで公開中．添付資料参照）は、広く会員の皆さまのお役に立つものと考える。これらの情報を参考にして、エンドユーザのアプリケーションプログラムにあっても是非一度はこのようなデータを取得して自らのプログラムの動きを認識してもらいたい。そのためにはセンターユーザ、ベンダーの協力も必要である。

イ）については、今回は九州大学の藤野氏に線型方程式の解法について提案・検討していただいたが、エンドユーザアプリケーションの段階でも同じように、大規模並列計算機が利用できるという新しい環境に適した計算法、プログラミングスタイルを考え出す試みが必要である。ただ、ア）は時間をかければ何がしかの効果を得られるケースが殆どであるが、こちらは本質的に「創り出す」ものであるため時間をかければ手に入れることができる、というものではない。

ウ）は、ア）を掘り下げていくことによりある程度は実現できると考えているが、本 WG のような場は必ずしも適切ではなさそうである。

わが国の計算機を作る能力、計算機を使いこなす能力を維持・発展させるためには、ア）、イ）、ウ）の活動全般を再構築する必要があるように感じている。特にア）の活動については、ややもすれば枝葉末節的な“技術”と捉えられているのか、エンドユーザレベルでの認識が弱いように思う。また、ア）の活動については本 WG ではエンドユーザアプリの視点で取り組んだが、I/O やファイルシステムなども含めたシステムとしての評価や運用技術からみた評価などセンターユーザの視点からの評価も重要である。これらのことに取り組んでいくにはセンターユーザを基点としてエンドユーザ、ベンダーの協力体制の構築が必須と思料する。

最後になるが、貴重な時間を割いて2年間のWG活動に参加して頂いた会員の皆さん、富士通の担当者、WGの円滑な運営に尽力して頂いたSS研事務局の皆さんに感謝する。また、評価のためのデータ取得に協力をいただいた富士通担当者に感謝の意を表する。

（SMP スレッド並列 WG まとめ役　福田正大）

商標について

- SPARC 商標は，米国 SPARC International,Inc. の登録商標です．SPARC 商標のついた製品は 米国 Sun Microsystems, Inc.が開発したアーキテクチャーに基づくものです．
- Sun, Sun Microsystems, Sun ロゴ, Solaris およびすべての Solaris に関連する商標及びロゴは，米国およびその他の国における米国 Sun Microsystems, Inc.の商標または登録商標です．
  注：Solaris(TM) Operating System および Solaris(TM) オペレーティングシステムは，本書では「Solaris OS」または「Solaris」と記述しています．
- Parallelnavi は、富士通株式会社の商標です。
- その他各種製品名は，各社の製品名称，商標または登録商標です．
- 本書に記載されているシステム名，製品名等には，必ずしも商標表示（®，™）を付記していません．

SS研 SMPスレッド並列WG (2004/11-2006/10) 成果報告書《公開版》

【発行】　サイエンティフィック・システム研究会

【編集/著者】　SMPスレッド並列WG

※　著作権は各原稿の著者または所属機関に帰属します。無断転載、引用を禁じます。

本資料に関するお問合せは、下記へお願いします。

サイエンティフィック・システム研究会（SS研）事務局
東京都港区東新橋1-5-2 汐留シティセンター（〒105-7123）
富士通株式会社　カスタマーリレーション部内
TEL：03-6252-2582(直通)　FAX：03-6252-2934
Email：ssken@ssken.gr.jp
Web：http://www.ssken.gr.jp/index.shtml

2006年10月31日作成　2007年5月11日発行　禁無断転載