NS3D コードの各種並列化言語を用いた並列化による性能向上について

小林 穣*、松尾 裕一*

Performance Improvement using the Parallelization for NS3D Code

Minoru Kobayashi* and Yuichi Matsuo*

ABSTRACT

The performance evaluation for the various parallel programming languages is done using NS3D (P3) code under the condition of the large scale and the large number of processors. After the scalar optimization of the NS3D code, the code is parallelized using the various parallel programming languages and the performance is evaluated. The calculation part for the NS3D code is obtained a good parallel performance using the various parallel programming languages and the hybrid version (MPI+OpenMP and XPFortran+OpenMP). However, the communication part of the NS3D code indicates that the performance of the XPFortran (+OpenMP) version is better than that of the MPI (+OpenMP) version. It is neccessary to improve the performance of the MPI (+OpenMP) version. The performance ratio between the sequential calculation and the XPFortran+OpenMP using 1440 CPUs is about 845.

1. はじめに

これまで，5 本の CFD コードを基に，JAXA 数値シミュレータ III の CeNSS（Central Numerical Simulation System）および各種スカラー計算機（IBM Power4, SGI Onyx3400）上で，スカラー最適化および OpenMP 化による性能向上の試みを実施してきた [1]．今回は，計算体系を自由に変更できる NS3D コードを利用して，大規模な計算体系にするとともに，種々の並列化言語（XPFortran, MPI, OpenMP およびコンパイラによる自動並列機能）を用いて並列化を行ない，並列化言語間の性能を比較した．加えて，大規模な計算（プロセッサ数で 1000 台規模）にも対応できるように，2 種類の並列化言語を組合せたハイブリッド・コード（XPFortran+OpenMP と MPI+OpenMP）も開発し，その性能評価を行なった．

2. 並列化言語の概要

今回使用した並列化言語の概要は，以下のとおりである．

- MPI[2]
  プロセス間通信の業界標準言語仕様（クラスター型並列・分散処理）である．
- XPFortran[3]
  分散メモリ向けのデータ並列型拡張言語であり，VPP Fortran[4] 仕様を包含している．
- OpenMP[5]
  共有メモリ向けの業界標準の並列拡張言語であり，指示行挿入による高度で柔軟な並列プログラミングが可能である．
- コンパイラによる自動並列機能[6]
  コンパイラが自動的に行なう並列化機能であり，VPP の自動ベクトル化ループが並列化の対象のループとなっている．

上記の中で，OpenMP とコンパイラによる自動並列機能は，スレッド並列用の言語であり，ノード内のみに適用可能である．これに対して，MPI と XPFortran は，プロセス並列用の言語であり，ノード内だけでなく，ノード間の並列にも適用可能である特長を有している．プロセス並列とスレッド並列の比較は，表 1 のとおりである．

表 1: プロセス並列とスレッド並列の比較

比較項目	プロセス並列	スレッド並列
メモリ	独立	共有
適用ノード	ノード内・間	ノード内
変数	ローカル	グローバル／ローカル
並列化	プログラム全体	段階的並列化が可能
データ通信	明示的に指定	不要
言語	XPFortran, MPI	OpenMP，自動並列

3. NS3D コードの概要

3.1 数値計算方法

NS3D は，3 次元圧縮性 Navier-Stokes 方程式を TVD スキームにより差分化し，その差分式を陰的近似因子化法（Implicit Approximate Factored Scheme, IAF 法）で解くコードである．

一般座標系における（非粘性圧縮性流れの）基礎方程式は，以下のように記述できる．

*宇宙航空研究開発機構

$$\frac{\partial \hat{Q}}{\partial t}+\frac{\partial \hat{E}}{\partial \xi}+\frac{\partial \hat{F}}{\partial \eta}+\frac{\partial \hat{G}}{\partial \zeta}=0 \qquad (1)$$

ここで，$\hat{Q}$ は，保存量からなる未知ベクトルで，$\hat{E},\hat{F},\hat{G}$ は，各方向の流速ベクトルである．質量保存式，運動量保存式およびエネルギー保存式からなっており，3 次元を取り扱うので，$\hat{Q},\hat{E},\hat{F},\hat{G}$ は，5 次元のベクトルとなっている．

この基礎方程式に IAF 法を適用することにより，最終的に以下の方程式を解くことに帰着する．

$$L_\xi L_\eta L_\zeta \Delta\hat{Q}^n = RHS^n \qquad (2)$$
$$\hat{Q}^{n+1} = \hat{Q}^n + \Delta\hat{Q}^n \qquad (3)$$

実際には，以下のように 3 方向に分割して，順次処理していく．

$$L_\xi x = RHS^n \qquad (4)$$
$$L_\eta y = x \qquad (5)$$
$$L_\zeta \Delta\hat{Q}^n = y \qquad (6)$$

各方程式は，5 × 5 のブロック 3 重対角連立方程式を解くことになる．

3.2 並列化方法

純 OpenMP のようなスレッド並列を用いた場合，データを自由に定義・参照することが可能である．これに対して，MPI や XPFortran のプロセス並列を用いた場合には，データの分割方法により制約を受け，データを自由に定義・参照することができない．今回採用した並列化方法では，図 1 にあるように，I と J 方向の計算では，K 方向で分割した配列を使用し，K 方向の計算では，J 方向で分割した配列を使用している．データの転置転送（図中の transpose の個所）を行なうことで，分割方向の変更を行なっている．なお，図中の SUB1 から SUB3 が，右辺ベクトルを計算するルーチンであり，SUB4 から SUB6 が，ブロック三重対角行列を解くルーチンとなっている．

3.3 スカラー最適化手法の概要

これまで用いていたスカラー最適化手法（tune-1：詳細は文献 [1] を参照のこと）に加えて，今回，以下のスカラー最適化を実施した．

- 重複している演算および配列の削除 (tune-2)
  重複している演算を削減し，演算の効率化を図った．
- 配列のパディング (tune-3)
  使用した計算体系が，514x480x480 のため，メモリのコンフリクトを回避するために，配列のパディングを行なった．ここで，パディングとは，共通ブロック内の各変数の領域の間や配列に，隙間を作ることを指す．具体的には，2 のべき乗の宣言を避けて，大きめの領域を確保したり，コンパイラのオプション（富士通コンパイラでは，-Kcommonpad[=N],-Karraypad_const[=N] 等のオプションがある）を指定したりすることで，対応する．
- 配列参照の変更 (tune-4)
  配列次元を変更することで，TLB ミス率の低減を行なった．MPI 版（XPFortran 版も）は，並列化軸を変更するために，データ交換を行なっている．本来，OpenMP 版には，並列化軸を変更する必要はないが，今回は配列参照を変更することで，高速になったため，適用している．即ち，転送先の配列を QT(5,II,JJ/PE,KK) から QT(5,II,KK,JJ/PE) に変更した．このことにより，TLB ミス率が低減（0.0000）し，計算時間の短縮が図れた．なお，I と K との配列の入れ替えも可能であるが，演算時間の短縮よりも，データの再配置に要する時間の増加が大きくなり，採用しなかった．

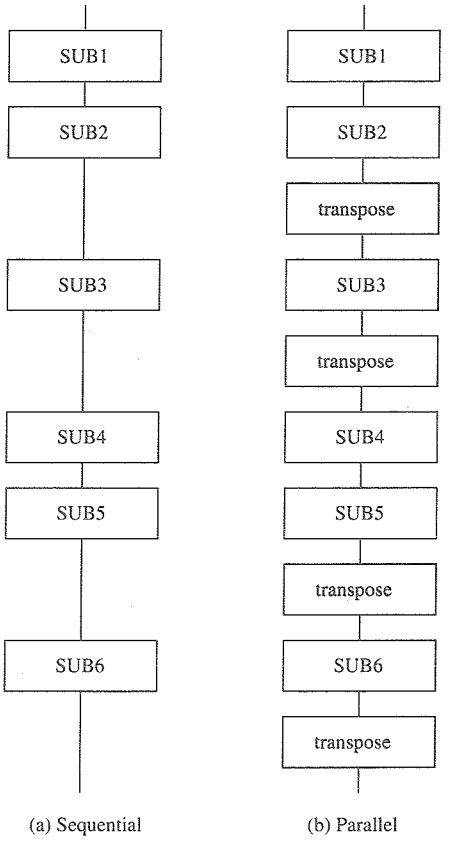

図 1: 並列処理の流れ

オリジナル版と 4 種類のチューニング版の性能を比較するために，各種測定結果を表 2 に掲げておく．言語は，OpenMP で，32 スレッドを使用して実行したものである．なお，格子サイズは，514x480x480 で，ステップ数は，5 回である．

上記の 4 種類のチューニングを実施することにより，オリジナルと比較して，約 3 倍の高速化が図れた．

表 2: チューニングの効果

name	original	tune-1	tune-2	tune-3	tune-4
main	54.5852	28.7262	30.2161	27.2741	39.2155
sub1	29.9159	24.6846	12.7411	13.3834	12.7927
sub2	30.5161	39.8532	26.0182	15.4004	14.6677
sub3	78.0581	36.3166	37.0786	34.6509	14.9348
sub4	144.9113	54.7280	33.6963	34.4048	33.7843
sub5	83.2952	74.1799	54.8571	35.1494	34.8799
sub6	146.4433	70.9075	49.2592	49.1994	34.8579
total	567.7255	329.3964	243.8669	209.4628	185.1332

単位：秒

使用した 3 種類の並列化言語の測定結果を表 3 に示す．使用したプロセッサ数は，32 である．この表は，OpenMP だけでなく，他のプロセス並列の言語でも，本チューニングが有効であることを示している．

表 3: 各言語での測定結果の比較

name	MPI	XPFortran	OpenMP
main	7.2425	5.7886	39.2155
sub1	12.9627	12.9564	12.7927
sub2	14.9886	15.0999	14.6677
trans-1	12.9242	9.5956	-
sub3	15.2641	15.0842	14.9348
trans-2	5.2224	5.3222	-
sub4	32.2924	33.3071	33.7843
sub5	32.6912	34.6299	34.8799
trans-3	4.8737	5.1322	-
sub6	32.8500	34.4877	34.8579
trans-4	4.8950	4.6724	-
total	178.5470	176.0762	185.1332

単位：秒

4. 性能評価

4.1 並列化言語単独の性能

オリジナル版だけでなく，チューニング版のソースも並列化言語固有の部分を除いて，同一のソースを用いることで，各種並列化言語そのものの性能を比較している．

個々の並列化言語を単独で使用した場合について，コード全体の測定結果を図 2 に示す．オリジナル版のみ実行した自動並列を除いて，オリジナル版（凡例のサフィックスが O のグラフ）とチューニング版（凡例のサフィックスが T のグラフ）の両方の結果を掲げている．使用したプロセッサ数は，プロセス並列（XPFortran と MPI）で 240 CPU，スレッド並列（OpenMP と自動並列）で 60 CPU である．なお，自動並列とオリジナル版を用いた OpenMP に関しては，コンパイル・オプション（`-Karraypad_const=1`）を指定することにより，自動的に配列のパディングを行なっている．チューニング版と差がでないように配慮している．なお，サフィックスの `O` と `T` は，それぞれ，オリジナル版とチューニング版であることを示している．

この結果は，以下のことを示している．

- コンパイラの自動並列機能は，少数スレッドであれば，並列化効果が期待できる．
- チューニング版は，どの並列化言語を使用しても，実行したプロセス数内においては，良い並列効果が得られている．
- オリジナル版もチューニング版と比較して，絶対性能では遅いが，どの並列化言語を使用しても，良い並列化効率が得られている．
- チューニング版だけでなくオリジナル版でも，同じ版を使用する限り，並列化言語間に大きな性能差が生じていない．

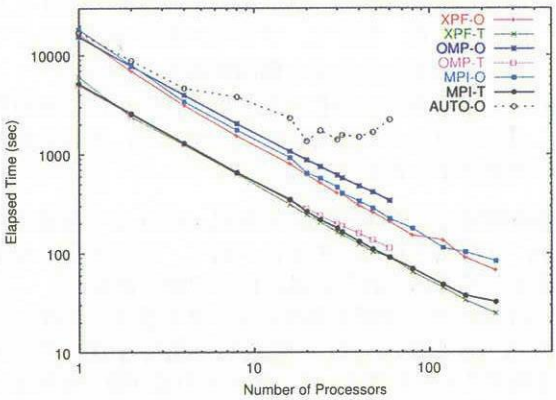

図 2: 並列言語単独でのコード全体の性能比較

チューニング版について，各言語（OpenMP, MPI および XPFortran）の性能を演算性能と通信性能（除 OpenMP）に分けて比較した．結果を図 3 に示す．ここで，`Calc`，`Comm` は，それぞれ，演算部分と通信部分の測定結果であることを示している．

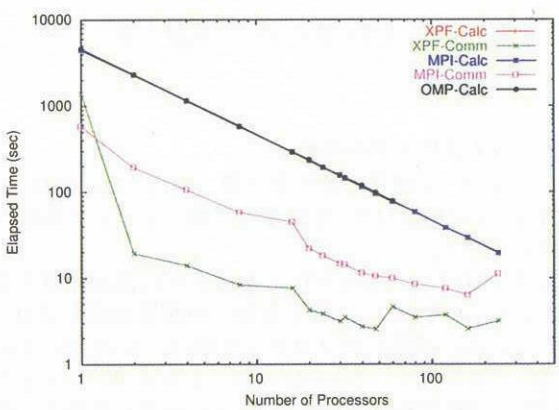

図 3: 並列言語単独での個別の性能比較

この結果は，以下のことを示している．

- どの並列化言語でも，演算処理の時間は，ほとんど同じである．
- 演算処理時間は，並列化効率が非常に高い．
- 本コードでは，スレッド並列（OpenMP）の並列化効率が非常に高く，60 スレッドまで，並列化効果がある．

- 通信処理の時間は，XPFortran の方が，MPI より高速である．だだし，プロセス数が増加するにつれて，差が小さくなる傾向がある．
- コード全体の並列化効率が低下するのは，通信処理がプロセスが増えるにしたがって，並列化効率が低下するためである．

通信時間が，XPFortran の方が，MPI より高速であるが，プロセスが増大するにつれて，差が小さくなる原因として考えられるのは，以下の点である．

MPI 版のデータ転置部分のソースを図 4 に示す．この図で，mpi_alltoall の転送元の配列 q は，データの連続アクセスを優先して，データの並び換えを行なっていない．このため，プロセス数が少ない場合に，do 250 ループの回転数が多くなり，小さなパケットが多数発生することになり，性能が低下した可能性が高い．配列 q のデータ転送に必要な部分を別の配列に格納することにより，通信回数を削減することは可能である．だたし，データのコピーに要する時間と通信時間の短縮時間とのトレードオフとなる（5.3 節で比較する）．

```
      do 250 kt = k11, k12
        call mpi_alltoall(
     1    q(1,1,1,kt), mm, mpi_double_precision,
     2    buff, mm, mpi_double_precision,
     3    mpi_comm_world, ierror )
        ic = 0
        do 240 lt = kt, kk, ks
          do 230 jt = j11, j12
            ic = ic + 1
            do 220 it = 1, ii
              do 210 n = 1, 5
                qt(n,it,lt,jt) = buff(n,it,ic)
210           continue
220         continue
230       continue
240     continue
250   continue
```

図 4: MPI 版のデータ転置方法

4.2 ハイブリッドの性能

ここでは，2 種類の並列化言語（MPI+OpenMP と XPFortra+OpenMP）を組合せた場合について性能の比較を行なう．

まず，MPI と OpenMP とを組合せた測定結果を図 5 に示す．あわせて，MPI 単独での測定結果もプロットしている．使用したプロセッサ数は，MPI が，1 から 96 プロセスで，OpenMP が，1 から 30 スレッド（ただし，96 プロセスのみ，15 スレッド）である．最大のプロセッサ数は，1440 CPU となっている．

この結果は，以下のことを示している．

- 1 スレッドの測定結果は，MPI 単独の測定結果と同等の性能である．
- 同一プロセッサ数で比較すると，スレッド数を多くするほど経過時間がかかっている．
- スレッド並列が有効なのは，8 スレッドぐらいまでである．

スレッド数が多くなるにつれて，性能が低下する原因を明らかにするために，コード全体でなく，演算処理時間とデータ転送時間について，それぞれの値を示したのが，図 6 である．ここで，同一の色は，同一のプロセス数であることを示しており，四角印と丸印は，それぞれ，演算部分と通信部分の測定結果であることを示している．

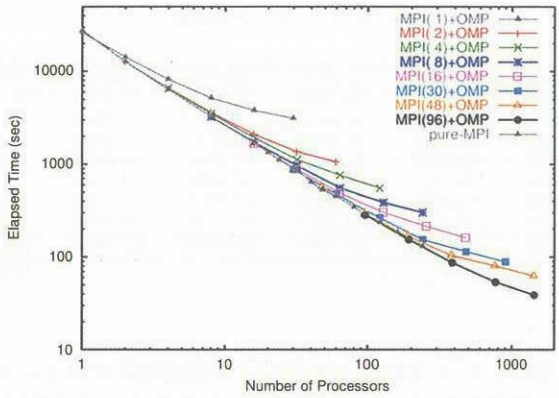

図 5: プロセス数を固定した MPI+OpenMP のコード全体の測定結果

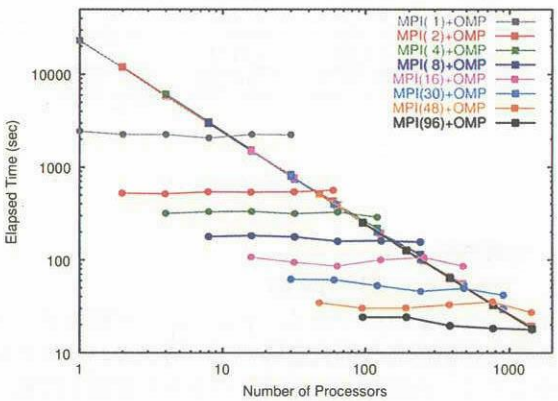

図 6: 各処理毎の MPI+OpenMP の測定結果

この結果は，以下のことを示している．

- 演算部分は，測定プロセス数の範囲内で，非常に高い並列化効率が得られている．
- 通信部分は，プロセス並列に対して，並列化効果があるが，スレッド並列に対して並列効果がない．
- 通信部分は，スレッド数に依存しないので，スレッド数が多くなると，データ通信時間の割合が増大し，並列性能が低下する．
- コード全体では，スレッド数が多くなるにつれて，並列性能が低下してくる．

次に，XPFortran と OpenMP を組合せた測定結果を図 7 に示す．図の内容は，MPI+OpenMP の場合と同様である．あわせて，XPFortran 単独で測定した結果もプロットしている．使用プロセッサ数は，MPI+OpenMP の場合と同じである．

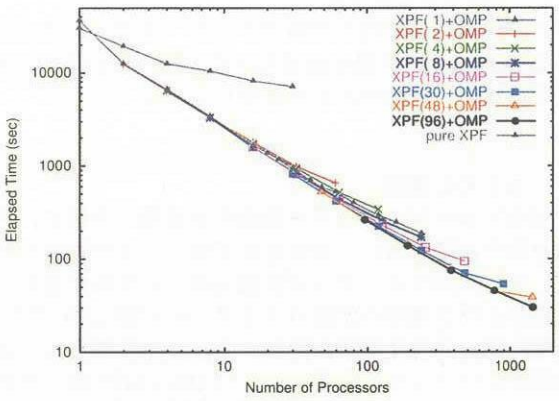

図 7: プロセス数を固定した XPFortran+OpenMP のコード全体の測定結果

この結果は，以下のことを示している．

- 1 スレッドの測定結果は，XPFortran 単独の測定結果と同等の性能である．
- 同一プロセッサ数で比較すると，スレッド数を多くするほど遅くなる傾向が見られるが，MPI+OpenMP 版ほど顕著ではない．
- XPFortran+OpenMP 版の方が，MPI+OpenMP 版よりも並列化効果が良い．
- スレッド並列は，16 スレッドぐらいまで有効である．

XPFortran+OpenMP の演算処理時間とデータ転送時間について，それぞれの値を示したのが，図 8 である．ここで，同一の色は，同一のプロセス数であることを示しており，四角印と丸印は，それぞれ，演算部分と通信部分の測定結果であることを示している．

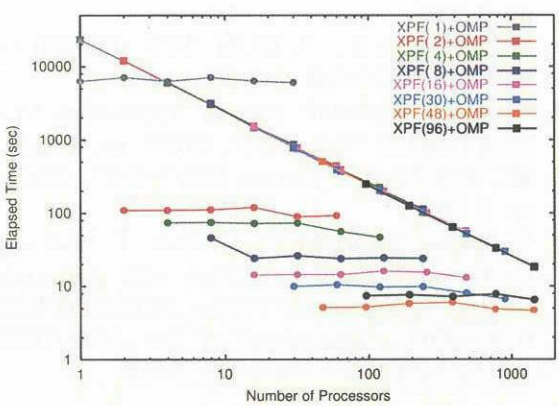

図 8: 各処理毎の XPFortran+OpenMP の測定結果

XPFortran+OpenMP の場合も MPI+OpenMP の場合と同様に，以下のようになっている．

- 演算部分は，測定プロセス数の範囲内で，非常に高い並列化効率が得られている．
- 通信部分は，プロセス並列に対して，並列化効果があるが，スレッド並列に対して並列効果がない．
- 通信部分の経過時間が短かいので，スレッド数が多くなっても，並列性能があまり低下しない．

最後に，両者の結果を比較すると，以下のようになる．

- 演算処理は，測定プロセス数（1 から 1440 CPU まで）の範囲内で，非常に高い並列化効率が得られている．
- データ通信処理は，プロセス並列に対していは並列化効率があるが，スレッド並列に対して，ほとんど並列化効率が得られていない．
- 通信部分は，MPI+OpenMP 版より XPFortran+OpenMP 版の方が高速である．

5. 議論

（特に，MPI+OpenMP の）ハイブリッド版の性能を向上させるための方策として，以下のことが考えられる．

- 計算体系を大きくする．
- 2 次元分割の MPI を採用する．
- 通信処理を削減する．
- マルチブロックを適用する．

5.1 計算体系の検討

1 タイム・ステップあたりの演算量とデータ転送量は，グリッド数に依存しないため，計算体系を大きくしても両者の比率は同じである．また，現在の計算体系での 1 回あたりのデータ転送量は約 5 GB になり，48 プロセスを使用しても，1 プロセスあたり約 100 MB のデータ転送量となる．CeNSS におけるデータ転送能力は，1 MB 程度で飽和する [7]．したがって，これ以上計算体系を大きくしても，通信性能の向上は，期待できない．

5.2 2 次元分割の MPI の採用

スレッド部分のデータ転送も含めて，並列化する方法として，MPI を用いた 2 次元分割が考えられる．ただし，ハードウェア資源の制限から今回のケースでは，768（48 x 16）が最大となり（1 ノードあたりの DTU が 16 個しかない），1000 台規模の計算が実行できない難点がある．また，2 次元分割にすることにより，データ転送回数が 2 倍になる（軸の入れ替えが，余分に必要となる）こともあり，今回は適用しなかった．

5.3 通信時間の削減

通信時間を削減するためには，以下の方法が考えられる．

- 演算と通信とをオーバラップさせ，通信処理（あるいは，演算処理）を隠蔽する．

- パケットの発生を抑止する.
- 複数個の三重対角行列を解くアルゴリズムを見直し，データ転送量を削減する.

最初の方法は，16 スレッドぐらいになると，通信部分と対応する演算部分の処理時間が同程度になり，オーバラップさせても通信時間を短縮することにならないので（オーバラップに伴なうオーバヘッドの分だけ，遅くなる可能性がある），今回は採用しなかった.

2 番目に，パケットの発生回数を抑止させる方法を述べる.

現行 MPI 版の測定結果を表 4 に示す. "total" と "alltoall" は，それぞれ，25 回反復させた通信関連部分の合計値と "MPI_ALLTOALL" 部分の合計値である. また，"max_all" と "min_all" は，25 回反復させた"alltoall" 部分の最大値と最小値である.

表 4: 現行 MPI 版の測定結果

npe	total	alltoall	max_all	min_all
008	46.23325	20.08145	0.91780	0.72948
016	27.36360	13.39197	0.61898	0.50120
032	11.31217	4.17991	0.29981	0.14604
048	8.30770	3.55248	0.28991	0.11036

単位：秒

パケットの発行回数を削減した MPI 版の測定結果を表 5 に示す. 各項目の内容は，表 4 と同じである.

表 5: パケット削減 MPI 版の測定結果

npe	total	alltoall	max_all	min_all
008	87.22441	19.98321	1.16966	0.37598
016	47.63717	12.49501	0.63442	0.30201
032	23.46732	4.48630	0.29895	0.13306
048	15.80380	3.31886	0.19795	0.10599

単位：秒

表 4 と表 5 の "min_all"（運用中に測定を行なっているので，最小値を基に評価する）を比較すると，プロセス数が少ない（16 以下）と，改造した版の方が性能が良くなっている. しかしながら，表 4 と 5 とを比較して問題となるのは，データの並び換えに要する時間（"total" - "alltoall"）である. パケット発行回数を削減することで，"alltoall" の経過時間は（少数プロセスの場合）短縮できるが，データの並び換えに要する時間が増大するために，通信部分全体の経過時間は，2 倍になっている. 現在，上記の MPI の性能の妥当性を調査しているところである.

最後の方法は，複数個の三重対角行列を解くプログラムを作成し，評価を行なっているところである. ただし，本アルゴリズムを適用する場合，sub6 のループ構造を変更する必要がある. これまで，最内にしていた K ループをオリジナルと同じように，最外に変更する必要がある. しかも，アルゴリズムの変更により，データ転送量が削減できたとしても，今回適用したキャッシュミスを低減する方法がそのまま活用できなくなる. 今後，更に検討し，有効な方法であれば，本プログラムへの適用を考えていきたい.

5.4 マルチブロック化

どの程度通信時間が短縮できるか（多分，3 次元から 2 次元にデータ量が削減できると思っている），不明であるが，効果があると期待している.

6. まとめと課題

NS3D コードのスカラー最適化を実施した後に，各種並列化言語を用いた並列化を実施し，その性能を評価した. NS3D コードの演算部分は，いずれの並列化言語および 2 種類の言語のハイブリッド版においても，非常に良い並列化効率が得られた. これに対して，通信部分については，XPFortran (+OpenMP) 版と比較して，MPI (+OpenMP) 版の測定結果が悪く，改良の余地が残っていることが明らかとなった. 全体としては，XPFortran+OpenMP 版を 1440 CPU で実行することにより，逐次処理と比較して，約 845 倍の高速化が図れた.

今後の課題として，MPI (+OpenMP) の処理の高速化，通信時間の短縮化およびマルチブロックへの適用等を行なっていくことで，更なる高速化を図っていく必要がある.

謝辞

本研究で大規模な計算をするあたって，JAXA の IT センターの運用グループのメンバに多大な協力をしていただきました. ここに記して，感謝の意を表します.

参考文献

1. 小林穣，松尾裕一，"CFD コードのスカラー最適化と OpenMP 化による性能向上の試み", 第 17 回数値流体力学シンポジウム, (2003), pp. 128.
2. MPI Forum, "MPI-2: Extensions to the Message-Passing Interface", (1997), pp. 1–362.
3. 富士通株式会社, "XPFortran 使用手引書", (2003), pp. 1–330.
4. 富士通株式会社, "UXP/M VPP FORTRAN EX/VPP 使用手引書 V12 用", (1993).
5. OpenMP, "OpenMP Fortran Application Program Interface Version 2.0", (2000), pp, 1–116.
6. 富士通株式会社, "Fortran 使用手引書", (2002), pp. 1-346.
7. Y. Matsuo, M. Tsuchiya, M. Aoki, T. Inari and K. Yazawa, "Early Experience with Aerospace CFD at JAXA on the Fujitsu PRIMEPOWER HPC 2500", *Proceedings of the ACM/IEEE SC2004 conference*, CD-ROM, (2004).