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

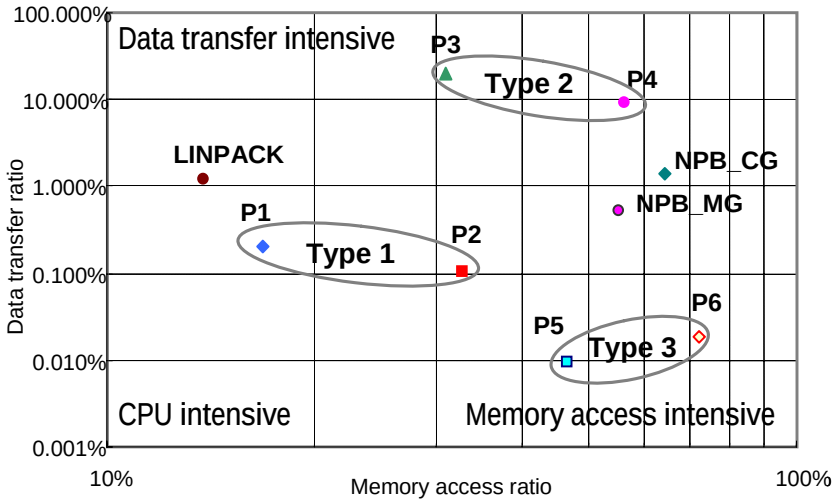

図 2.2　JAXA 並列 CFD コードの特性

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

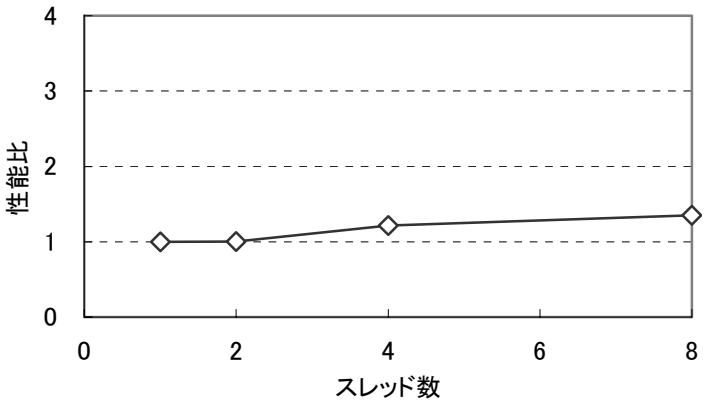

図 3.3　スレッド数に対する性能向上比

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

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

XPF の分割ループで並列化したため回転数が少ない

変更後

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

【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

変更前

```
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)

変更後

```
!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

【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

変更前

```
       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．

変更後

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

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

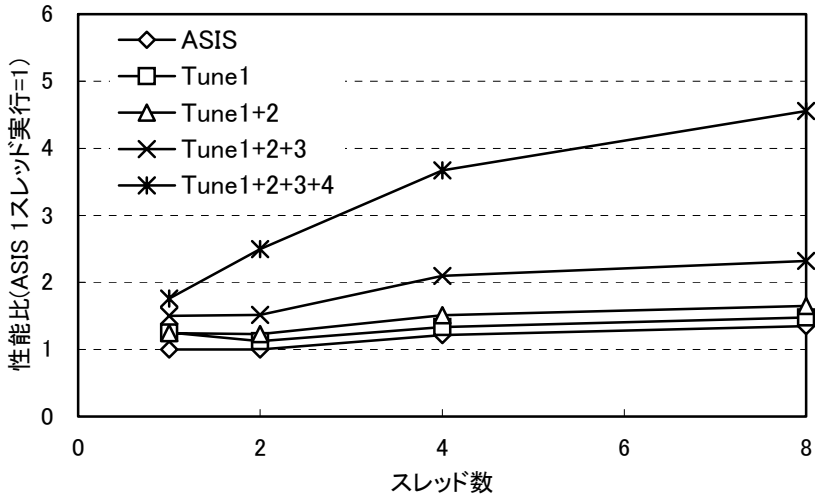

図 3.14　チューニングによるスレッド並列性能

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井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．

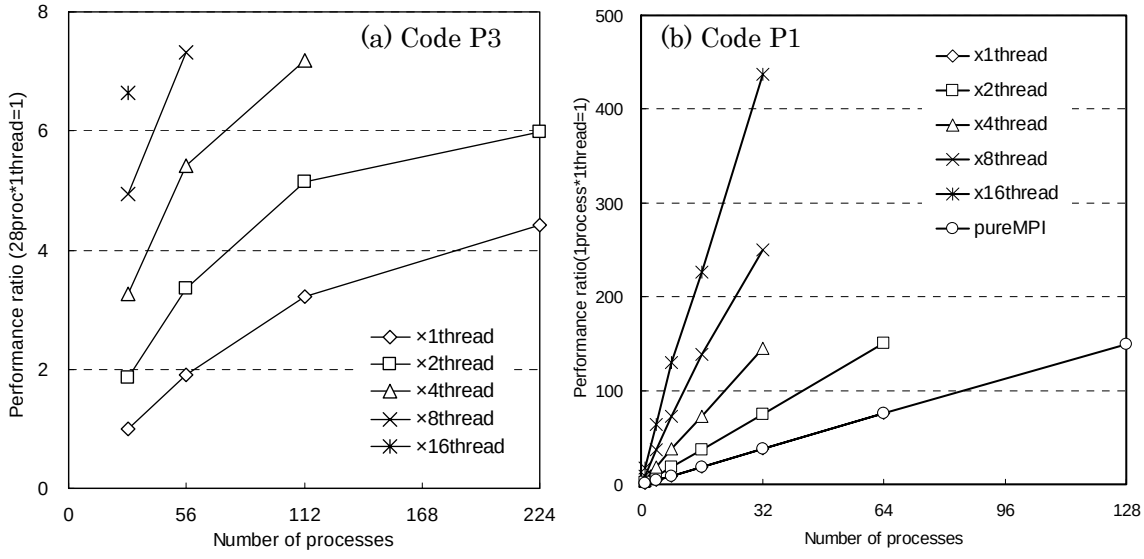

図 3.15　JAXA 並列 CFD コードのハイブリッド並列性能

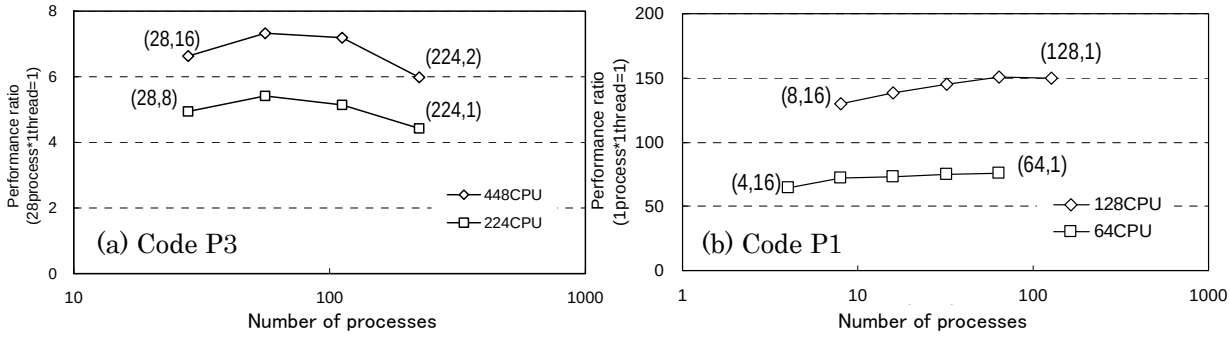

図 3.16　CPU 数一定時のハイブリッド並列性能の比較

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，

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

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，

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

の関係がある．もし，$a = 1$ なら $T_{\text{parallel}} = T_{\text{serial}}/n$，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 $S(\infty) \to 1/(1\text{-}a)$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 $a = (1\text{-}1/S)(1\text{-}1/n)$ 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，

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

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

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

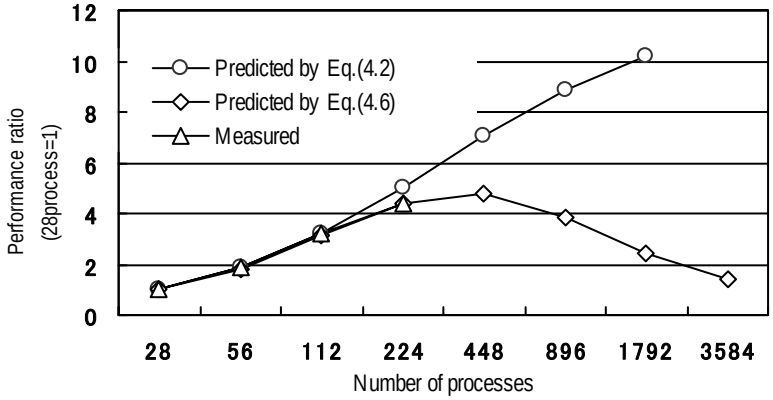

図 4.5　アムダールの法則によるプロセス性能推定の比較（コード P3）

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，$a_p = 0.925$，$c_t = 0.057$，$c_n = 0.005$，$1-a_p-c_t-c_n = 0.013$ 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，

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

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

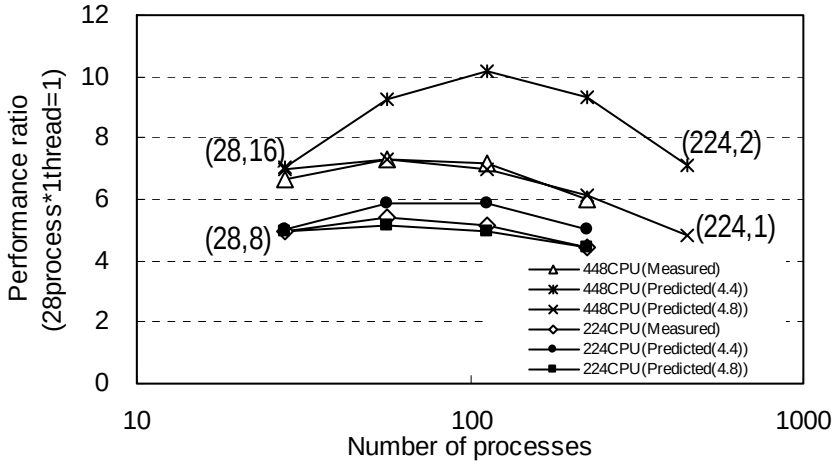

図 4.8　アムダールの法則によるコード P3 の性能推定の比較

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).