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

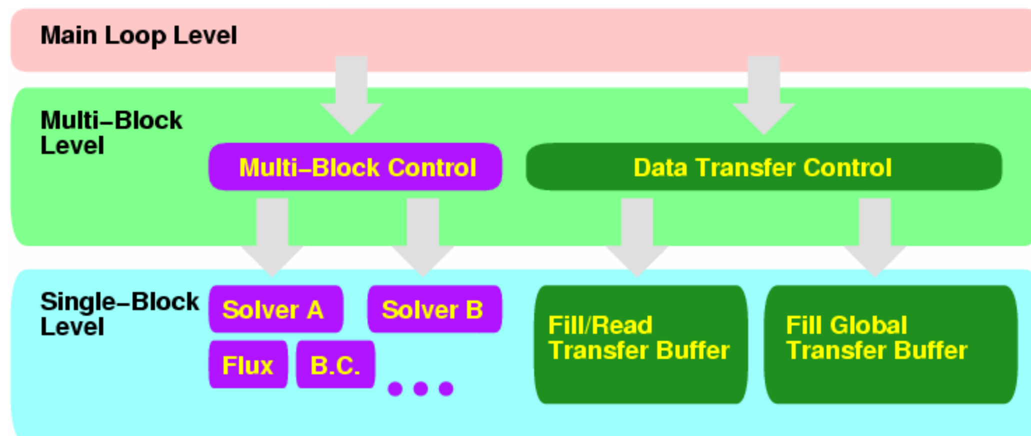
宇宙航空研究開発機構（JAXA）で開発された CFD プログラム UPACS について、富士通 PRIMEPOWER HPC2500 上で性能評価を行ったのでその結果を報告する。

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。

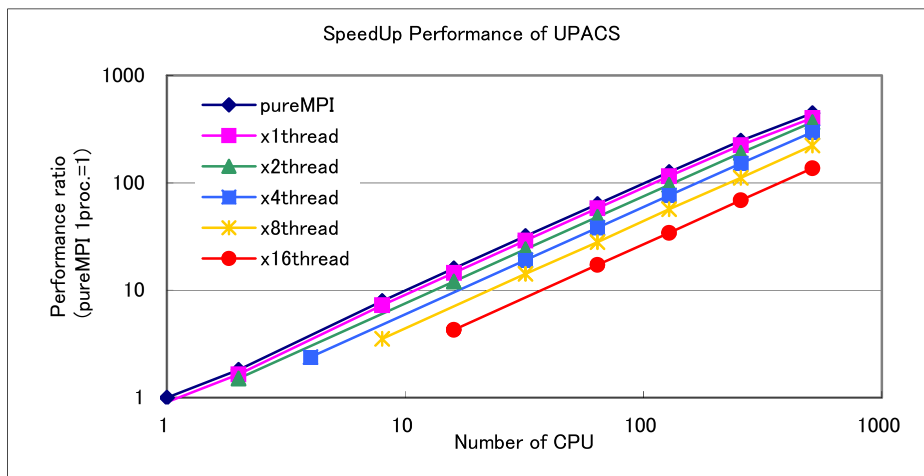
【測定条件】

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	Process数									
(MPI 1proc.=1)	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
```

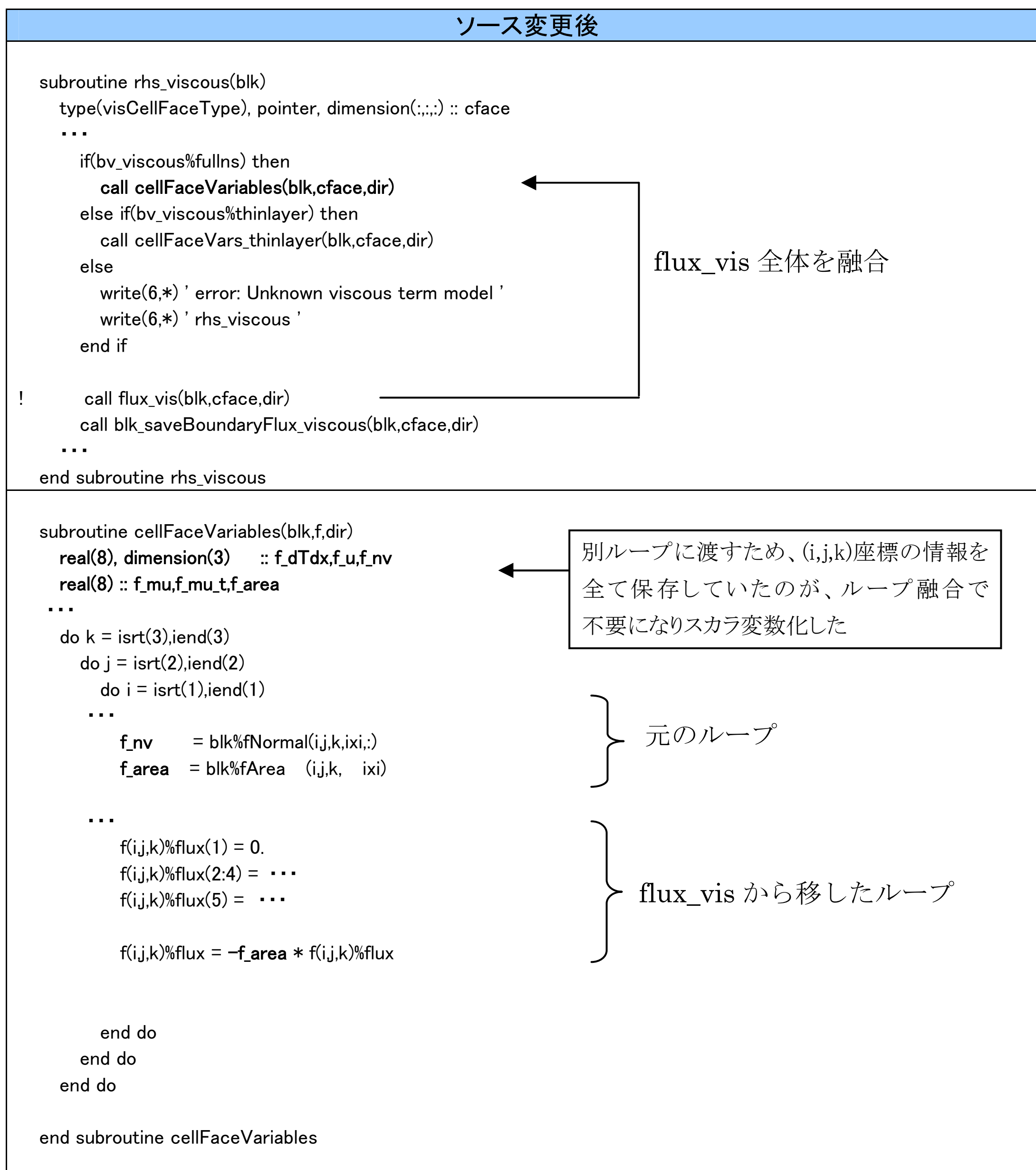
```
subroutine flux_vis(blk,f,dir)
 ・・・
  do k = isrt(3),iend(3)
    do j = isrt(2),iend(2)
      do i = isrt(1),iend(1)
     ・・・
         f(i,j,k)%flux(1) = 0.
         f(i,j,k)%flux(2:4) = ・・・
         f(i,j,k)%flux(5) = ・・・

         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
```

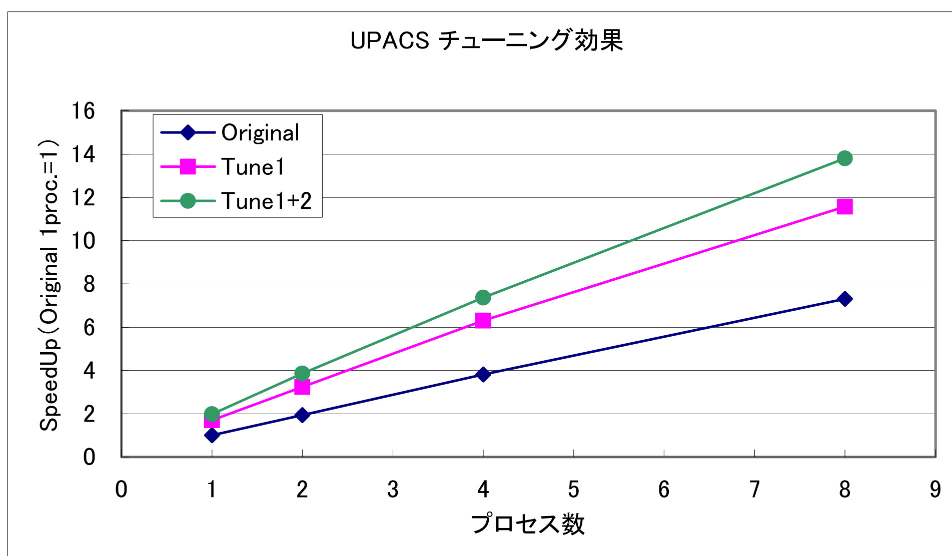

4.2 性能測定

「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。

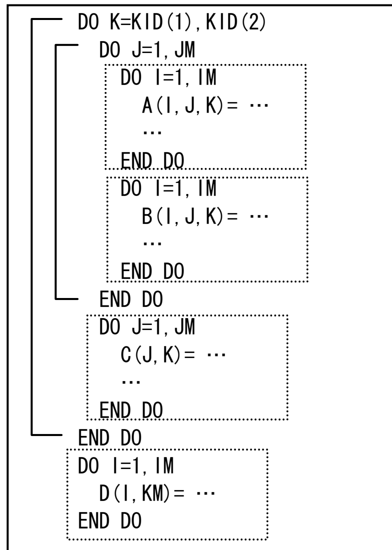
	自動並列化	自動ベクトル化
機種	富士通 PRIMEPOWER HPC2500 (SPARC64V 1.3GHz)	富士通 VPP5000
言語環境	Parallelnavi2.4 (Fujitsu Fortran Compiler V5.6)	UXP/V Fortran V20L20
翻訳時 option	mpifrt -Kfast_GP2=3,V9,largepage=2,hardbarrier -x- -Kparallel,reduction	-Pa -Wv,-m3
使用 program	UPACS (前回報告の Tune1+2 版) ※自動並列化や自動ベクトル化を促進するための追加変更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

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。

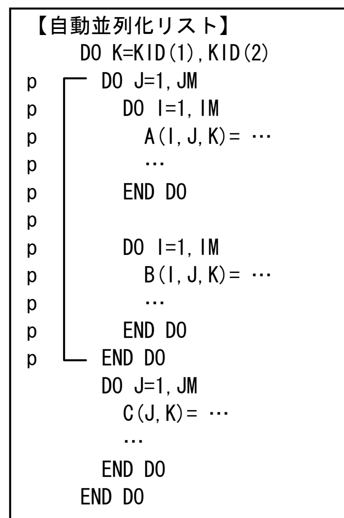
- 並列化の規模が判りにくい（外側の大きなループでも内側の小さなループでもカウント数は同じ）
- 並列化とベクトル化の比較が難しい（並列化は外側から、ベクトル化は内側からの解析で軸が異なる場合がある）

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。

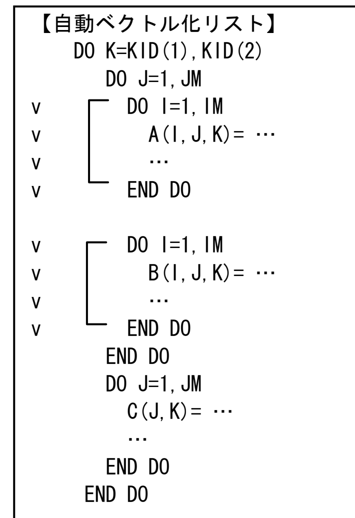


```
DO K=KID(1),KID(2)
  DO J=1,JM
    DO I=1,IM
      A(I,J,K)= …
      …
    END DO
    DO I=1,IM
      B(I,J,K)= …
      …
    END DO
  END DO
  DO J=1,JM
    C(J,K)= …
    …
  END DO
END DO
DO I=1,IM
  D(I,KM)= …
END DO
```

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。



```
【自動並列化リスト】
    DO K=KID(1),KID(2)
p     DO J=1,JM
p       DO I=1,IM
p         A(I,J,K)= …
p         …
p       END DO
p
p       DO I=1,IM
p         B(I,J,K)= …
p         …
p       END DO
p     END DO
      DO J=1,JM
        C(J,K)= …
        …
      END DO
    END DO
```



```
【自動ベクトル化リスト】
    DO K=KID(1),KID(2)
      DO J=1,JM
v       DO I=1,IM
v         A(I,J,K)= …
v         …
v       END DO

v       DO I=1,IM
v         B(I,J,K)= …
v         …
v       END DO
      END DO
      DO J=1,JM
        C(J,K)= …
        …
      END DO
    END DO
```

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。