ＳＰＥＣ　ＯＭＰとＨＰＦ

坂上 仁志，森井 宏幸
姫路工業大学　大学院工学研究科　電気系工学専攻

Rewriting SPEC OMP Benchmark Programs with HPF

Hitoshi Sakagami, Hiroyuki Morii
Department of Electrical Engineering and Computer Sciences
Himeji Institute of technology

1．はじめに

現在，実用的な並列プログラミング環境として MPI[1,2]，OpenMP[3]，HPF[4-6] が存在する．MPI(Message Passing Interface) は，多くの異なったアーキテクチャの並列計算機で利用可能である．しかし，ユーザがプロセッサ間のデータ転送をプログラム実行の流れを意識した上で明示的に記述しなければならず，一般ユーザには扱い難い．一方，OpenMP は一般ユーザーにも扱い易いように，従来の逐次プログラムに最小限の付加的な指示文を挿入するだけでプログラムの並列実行を可能とする記述性の高い指示文ベース処理を用いているが，共有メモリ型の並列計算機でしか利用できない．これに対して，HPF(High Performance FORTRAN) は OpenMP と同様に指示文ベース処理を用いたデータ並列言語であり，一般ユーザにも扱い易く，共有メモリ型の並列計算機だけではなく分散メモリ型の並列計算機でも利用できる．そのため，HPF は今後並列計算機を一般ユーザでも容易に扱えるツールとするために必要なプログラミング言語と言える．

そこで，共有メモリ型並列計算機システムの性能を評価するために OpenMP で記述されたベンチマークプログラムである SPEC OMP2001[7] を HPF に書き換え，並列性能を比較する．また，書き換えを行う上で明らかになった HPF の抱える問題点について議論し，その問題点の解決方法を示す．

2．HPF による並列化

2．1　並列化の方針

OpenMP との比較評価を行うため，基本的には OpenMP のプログラムで並列化されている部分（並列領域）に着目し，その部分を HPF で並列化する [8]．OpenMP とは異なり，HPF では結果を格納する配列要素を保持するプロセッサが計算を行うように処理が分割される．これを Owner Computes Rule と呼ぶ [9]．このため，OpenMP で並列化されている DO ループの DO 変数が使われている配列変数の次元を HPF でデータ分散する．また，通信の効率化のため SHADOW 指示文，REFLECT 指示文，ON-HOME-LOCAL 指示文も用いる．

実行環境には大阪大学サイバーメディアセンターの NEC SX-5 を用いた．また，一部 SX-5 では扱えない機能を使用するために地球シミュレータセンターにある PC クラスタを用いた．これらの HPF コンパイラは，まず HPF ソースを解釈して MPI ライブラリの呼び出しを含む並列実行可能な FORTRAN プログラムを出力するプリコンパイルと呼ばれる作業を行う．次に，バックエンドの FORTRAN コンパイラを用いてそのプログラムから実行可能モジュールを作成する．この作業は，通常 HPF コンパイラが自動で行うためにユーザは意識せずコンパイルできる．Table 1 に利用したコンパイラの名前とバージョンを示す．

Table 1 Compiler name and version

SX-5	HPF	HPF/SX V2 Rev.1.9.2
	OpenMP	FORTRAN90/SX Version 2.0 for SX-5 Rev.280
	FORTRAN	FORTRAN90/SX Version 2.0 for SX-5 Rev.280
PC cluster	HPF	HPF/ES for PC Cluster Rev.2.0
	FORTRAN	Intel(R) Fortran Compiler for 32-bit applications, Version 7.0

2．2　SWIM

SWIM は差分法により浅水方程式を数値計算し，海水のシミュレーションを行うプログラムである．

SWIM の書き換えでは，同一の DO ループ内で結果を格納する複数の変数の添字が一致していない場合に並列化を行うと効率よく実行されない問題が見つかった．この問題を不適切分散問題と呼ぶ [8].

```
      dimension u(10,10),v(10,10),p(10,10)
!HPF$ PROCESSORS proc(2)
!HPF$ DISTRIBUTE (*,BLOCK) ONTO proc :: u,v,p
      ●●●
!HPF$ INDEPENDENT
      do j = 1 , 9
       do i = 1 , 9
        u(i+1,j) = -p(i,j)*2
        v(i,j+1) =  p(i,j)/2
       enddo
      enddo
      ●●●
```

上記の例では，u(i,1:5) が proc(1) に u(i,6:10) が proc(2) にデータ分散されている．同様に v，p もデータ分散されているため，proc(1) は u(i,5) を所持しているが v(i,6) は所持していない．このため，j=5 のループ処理は，Owner Computes Rule により異なるプロセッサで実行しなければならない．しかし，NEC SX-5 の HPF コンパイラは並列実行のために DO ループの処理分割を行うとき，DO ループの範囲をできる限り一塊として扱おうとするため，Owner Computes Rule との間に矛盾が生じていた．

ソースをプリコンパイルして確認したところ，コンパイラが矛盾を解決するため，以下に示すように動的に確保された一時配列 vtemp に演算結果を格納し，演算後に vtemp(i,j) を v(i,j+1) に複写するソースを生成していた．この配列の動的確保，解放と配列間の複写のため，並列実行の効率が落ちていたことが分った．

```
allocate(vtemp)
do j = 1 , 9
  do i = 1 , 9
    u(i+1,j)   = -p(i,j)*2
    vtemp(i,j) =  p(i,j)/2
  enddo
enddo
do j = 1 , 9
  do i = 1 , 9
    v(i,j+1) = vtemp(i,j)
  enddo
enddo
deallocate(vtemp)
```

不適切分散問題は，同一ループ内で結果を格納する複数の配列変数の添字が一致していない場合に生じるため，配列変数の添字が同一になるように別々の DO ループに分けることで回避できる．また，この問題はデータ分散が DO ループの処理分割に適合していないことが原因で生じるため，ALIGN 指示文を用いて結果を格納する配列変数の添字に合わせてデータ分散を行うことによっても解決することができる．

別々の DO ループに分ける方法では，簡単に不適切分散問題を解決することができる．しかし，DO ループ内でベクトルレジスタのデータが再利用できる場合，この方法ではデータのリロードが必要になり，ベクトル処理の効率が悪くなる．このため，別々の DO ループに分ける方法による不適切分散問題の解決は，ソースに修正を加える必要があるだけではなく計算効率が悪化する場合がある．一方，ALIGN 指示文を用いる方法はソース修正の必要もなくベクトル処理の効率が悪くなる欠点もない．しかし，適切なデータ分散が DO ループによって異なる場合，すべての DO ループに一つの ALIGN 指示文で対応できないという問題がある．

この場合以下に示すように，上の DO ループを分けることで不適切分散問題を解決する方法 (a) と上の DO ループは ALIGN 指示文を用いて不適切分散問題を解決し，下の DO ループは分けることで新たな不適切分散問題が生じないようにする方法 (b) が考えられる．この二つの方法を比較すると，方法 (b) が上の DO ループでベクトルレジスタのデータを再利用できて効率がよいため，SWIM では方法 (b) を用いて並列化を行う．

```
!HPF$ DISTRIBUTE (*,BLOCK) ONTO proc::u,v,p
      ●●●
!HPF$ INDEPENDENT
      do j = 1 , 9
       do i = 1 , 9
        u(i+1,j) = -p(i,j)*2
       enddo
      enddo
!HPF$ INDEPENDENT
      do j = 1 , 9
       do i = 1 , 9
        v(i,j+1) =  p(i,j)/2
       enddo
      enddo
!HPF$ INDEPENDENT
      do j = 1 , 9
       do i = 1 , 9
        u(i,j) = x(i,j)
        v(i,j) = y(i,j)
       enddo
      enddo
      ●●●
```

(a)

```
!HPF$ DISTRIBUTE (*,BLOCK) ONTO proc::u,p
!HPF$ TEMPLATE , DISTRIBUTE(*,BLOCK) ONTO
!HPF$& proc :: base(10,0:10)
!HPF$ ALIGN base(i,j) WITH v(i,j-1)
      ●●●
!HPF$ INDEPENDENT
      do j = 1 , 9
       do i = 1 , 9
        u(i+1,j) = -p(i,j)*2
        v(i,j+1) =  p(i,j)/2
       enddo
      enddo
!HPF$ INDEPENDENT
      do j = 1 , 9
       do i = 1 , 9
        u(i,j) = x(i,j)
       enddo
      enddo
!HPF$ INDEPENDENT
      do j = 1 , 9
       do i = 1 , 9
        v(i,j) = y(i,j)
       enddo
      enddo
      ●●●
```

(b)

このように不適切分散問題は，主にALIGN指示文を用いて解決し，ALIGN指示文のみで対応できない場合は効率を考慮にいれて別々のDOループに分ける方法を併用することで解決した．Table 2に不適切分散問題解決前と以上の方法により解決したHPFプログラムの実行時間と高速化率を示す．

Table 2 Calculation time and speedup ratio of SWIM

		Calculation time[sec]			Speedup ratio		
		without solution	with solution	OpenMP	without solution	with solution	OpenMP
Number of Processors	1	505.3	294.7	283.2	1.00	1.00	1.00
	2	265.9	153.3	150.4	1.90	1.92	1.88
	3	183.7	109.2	103.2	2.75	2.70	2.74
	4	144.4	82.3	81.4	3.50	3.58	3.48
	8	81.8	51.3	48.4	6.18	5.74	5.85

Table 2の結果より，不適切分散問題を解決することで性能を大幅に改善することができ，HPFでOpenMPと同等の並列性能が実現できたことが分る．

2．3 MGRID

MGRIDはマルチグリッドで3次元のポテンシャル場を計算するプログラムである．

MGRIDでは以下のように大きな1次元配列を主プログラムで用意し，副プログラムではその1次元配列の一部を3次元配列として用いている．

```
dimension a(M),is(k),il(k)
●●●
do i = 1 , k
  call sub(a(is(i)),il(i))
enddo
●●●
```

```
      stop
      end

      subroutine sub(a,m)
      dimension a(m,m,m)
      ●●●（計算）
      return
      end
```

引数として渡す配列変数の次元数が呼ぶ側と呼ばれる側のプログラムで異なる，いわゆる順序結合はFORTRAN77を用いて大規模なプログラムを作成するときには，よく用いられた．HPFは並列化を行うためにデータ分散を行うが，データ分散を行った配列の順序結合には対応していない．そのため，MGRIDのプログラムはHPFの指示文を追加するだけでは並列化できなかった．HPFが分散配列の順序結合をサポートしていない問題を引数の次元数不一致問題と呼ぶ[8].

呼び出される側での引数の次元数を呼ぶ側のプログラムの次元数に合わせて定義し，呼び出されるプログラムで本来扱う次元数で必要なサイズの配列を動的に用意し，引数からデータを複写して使用することで引数の次元数不一致問題を回避した．副プログラムで扱う配列のサイズが呼ばれるタイミングで異なるために，副プログラムで用意する配列は動的に確保するのが妥当である．なぜなら，配列は必要以上に確保すると処理の分割が均等にならないためである．そこで，主プログラム側から変数配列aの一部をm×m×mのサイズで副プログラムsubに1次元配列aaとして渡し，整合配列を用いてデータを副プログラム側で受け，本来の3次元配列aにデータを複写するように書き換える．以下にMGRIDで引数の次元数不一致問題を動的配列へ複写して解決した例を簡略化して示す．

```
      dimension a(M),is(k),il(k)
      ●●●
      do i = 1 , k
        call sub(a(is(i)),il(i))
      enddo
      ●●●
      stop
      end

      subroutine sub(aa,m)
      dimension aa(m**3)
      allocatable a(:,:,:)
!HPF$ PROCESSORS proc(NP)
!HPF$ DISTRIBUTE (*,*,BLOCK) ONTO proc :: a
      allocate a(m,m,m)
      call subin(aa,a,m,m,m)       ! aa から a へ複写
      ●●●                          !（計算）
      call subout(aa,a,m,m,m)      ! a から aa へ複写
      deallocate(a)
      return
      end
```

また，MGRIDでは，マルチグリッドによる計算法により配列のサイズが4から258まで変化する．このため，以下のinterpという副プログラムにおいてプロセッサ数以下の配列サイズのときには工夫が必要であった．

```
      SUBROUTINE interp(z, m, u, n)
      ●●●
      DIMENSION u(n, n, n), z(m, m, m)
!HPF$ PROCESSORS proc(NP)
!HPF$ TEMPLATE , DISTRIBUTE (*,*,BLOCK) ONTO proc :: utemp(n,n,n+2)
!HPF$ ALIGN u(i,j,k) with utemp(i,j,k)
!HPF$ ALIGN z(:,:,i) with utemp(:,:,2*i-1)
!HPF$ SHADOW (0,0,2) :: z
      ●●●
!HPFJ REFLECT z
      do k = 2 , m-1
!HPFJ ON HOME(u(:,:,k)),LOCAL BEGIN
```

```
      •••                                    ! （データ通信が必要でない計算）
!HPFJ END ON
      enddo
      •••
```

Fig.1 に副プログラム interp で n=6，m=4 であるときのデータ分散の様子を示す．SHADOW 指示文ではシャドウ領域を配列の上限，下限から SHADOW 指示文で指定された幅だけ確保するためデータを保持していない配列のシャドウ領域は確保できない．そのため，Fig.1(a) のように配列サイズがプロセッサ数以上である場合はすべてのプロセッサに z がデータ分散されるため正しくシャドウ領域を確保できる．しかし，プロセッサ数が 8 と配列サイズよりプロセッサ数の方が大きい場合は，Fig.1(b) に示すようにデータ分散でデータが割り振られないプロセッサが生じる．このため，データ分散で z が割り振られている proc1,3,5,7 ではシャドウ領域を確保できるが，proc2,4,6,8 ではシャドウ領域を確保することができない．その結果，proc2,4,6,8 では REFLECT 指示文によりデータを得ることができず，OH-HOME-LOCAL 指示文が用いられていると proc2,4,6,8 は z の正しい値を持たないために正しい演算結果が得られない．

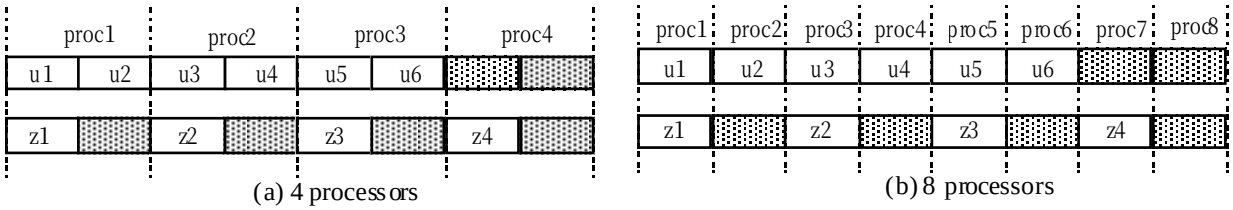

Fig.1 Data distribution

そこで，REFLECT，ON-HOME-LOCAL 指示文を用いずにコンパイラが自動で通信コードを生成するのにまかせることで問題を回避した．しかし，REFLECT，ON-HOME-LOCAL 指示文を用いないと通信の効率が悪い．そのために REFLECT，ON-HOME-LOCAL 指示文を挿入した副プログラムと挿入しない interp_no_on_home という副プログラムを別に用意し，以下のように if 文により呼びわけた．これにより，プロセッサ数より配列サイズが小さなときは必要な通信を行い，プロセッサ数より配列サイズが大きいときは必要のない通信を行わないことで効率の良い計算を行うことが可能となる．

```
if(mm(k) .ge. NP)then
  CALL interp(u(ir(k-1)), mm(k-1), u(ir(k)), mm(k))
else
  CALL interp_no_on_home(u(ir(k-1)), mm(k-1), u(ir(k)), mm(k))
endif
```

プロファイラの結果とプリコンパイルしたソースからプログラムを調べたところ，データを複写するコードを何度も書く必要がないように作成した subin，subout の副プログラムにおいて無駄が生じていた．

```
      subroutine subin(a,atemp,ix,iy,iz)
      double precision a,atemp
      integer*8 ix, iy, iz
      dimension a(ix,iy,iz) , atemp(:,:,:)
!HPF$ PROCESSORS proc(NP)
!HPF$ DISTRIBUTE (*,*,BLOCK) ONTO proc :: atemp
!HPF$ SEQUENCE a
       do k = 1 , iz
         do j = 1 , iy
           do i = 1 , ix
            atemp(i,j,k) = a(i,j,k)
           enddo
         enddo
       enddo
       return
       end
```

上記は subin のソースであるが，プリコンパイルしたソースでは以下の処理を行っていた．

```
subroutine subin(a,atemp,ix,iy,iz,…
●●●
call pghpf_qopy_in(…
do k = k$indl, k$indu
   do j = 1, iy
      do i = 1, ix
         atemp$$(i,j,k) = a(i,j,k)
      enddo
   enddo
enddo
●●●
call pghpf_copy_out(atemp$$,atemp, …
```

本来複写は一度で済むが，atemp ではなく atemp$$に一度複写された後，atemp に再度複写されていたために性能が悪化していた．そこで subin，subout を展開してプログラム中で直接複写することで問題を回避し，性能を改善した．この結果を Table 3 に示す．

Table 3 Calculation time and speedup ratio of MGRID

		Calculation time[sec]		Speedup ratio	
		HPF	OpenMP	HPF	OpenMP
Number of Processors	1	331.1	230.2	1.00	1.00
	2	207.1	118.6	1.60	1.94
	3	160.3	81.1	2.07	2.84
	4	141.0	63.1	2.35	3.65
	8	95.2	36.0	3.48	6.39

Table 3 より OpenMP に比べて 1 台での結果が劣っていることが分る．これは，配列の動的確保・解放，データの複写による影響であると考えられる．そこで，元のソースと動的確保・解放と複写を加えたソースを FORTRAN としてコンパイルし実行すると結果は変更前 226[sec]，変更後 304[sec] であった．これにより，HPF と OpenMP の 1 台での性能差の原因の大部分が配列の動的確保・解放，データの複写を追加したことであることが分った．

MGRID では主プログラムで大きな 1 次元配列を用意し，開始アドレスとサイズを変えることでその配列の一部を副プログラムへ渡していた．そこで，配列の動的確保・解放，データの複写を止めるために副プログラム側に合わせて必要なサイズでそれぞれ個別に定義してデータ分散を行うことが考えられる．しかし，この方法は以下のように大幅にプログラムに変更を加える必要がある．

```
C      dimension a(M),is(k),il(k)
      dimension a1(n1,n1,n1),a2(n2,n2,n2),...ak(nk,nk,nk),il(k)
      ●●●
      do i = 1 , k
C        call sub(a(is(i)),il(i))
        if(i .eq. 1)then
          call sub(a1,il(1))
        else if(i .eq. 2)then
          call sub(a2,il(2))
        ●●●
        else if(i .eq. k)then
          call sub(ak,il(k))
        endif
      enddo
      ●●●
      stop
```

そこで，改善案として Intel Fortran Compiler（以下，ifc）の loc() と%val() を用いて以下のように書き換えることで DO ループを展開することなく，配列の動的確保・解放，データの複写なしに並列化を行う方法を考えた．

loc は引数のアドレスを返す関数で，%val は FORTRAN でアドレス参照ではなくデータ参照による副プログラム呼び出しを行うために用いられる関数である．これらの関数により larg=loc(arg) で arg のアドレスを larg に代入し，%val(larg) で larg の値を参照することで arg のアドレスを手続きに渡すことができる．これを応用することで if 文による呼び分けを行わなくても元のプログラムと同様にアドレスを渡すことを実現している．この方法を，間接的アドレス引用法（Indirect Address Quoting Method : IAQM）と呼ぶ．

```
C      dimension a(M),is(k),il(k)
       dimension a1(n1,n1,n1),...,ak(nk,nk,nk),il(k)
!HPF$ PROCESSORS proc(NP)
!HPF$ DISTRIBUTE (*,*,BLOCK) ONTO proc :: a1,a2,...ak
       dimension 1a(k)      ┐
       la(1)=loc(a1)        │
       ●●●                  │ (追加)
       la(k)=loc(ak)        ┘
       do i = 1 , k
C         call sub(a(is(i)),il(i))
          call sub(%val(la(i)),il(i))
       enddo
       ●●●
       stop
       end

       subroutine sub(a,m)
       dimension a(m,m,m)
       ●●●（計算）
       return
       end
```

HPF ではデータ分散された配列が引数として扱われた場合，プリコンパイル時にその配列のデータ分散に関する情報を記憶している変数を副プログラムに渡す引数に追加するという処理を行っている．しかし，上記の書き換えでは副プログラムに分散されている配列を渡すのではなくアドレス値を渡しているため，データ分散についての情報を副プログラムに渡すための引数追加処理が行われない．そこで，コンパイラの代わりにプリコンパイル後にデータ分散の情報を副プログラムに渡すように書き換えを行った．

```
       dimension la2(k)          ┐
       la2(1) = loc(a1$sd1)      │
       la2(2) = loc(a2$sd1)      │ (追加)
       ●●●                       │
       la2(k) = loc(ak$sd1)      ┘
       do i = 1, k
C          call sub(%val(la(i)),il(i),pghpf_type(25),pghpf_type(25))
          call sub(%val(la(i)),il(i),%val(la2(i)),pghpf_type(25))
       enddo
```

また，SX-5 の FORTRAN コンパイラでは，上記の loc(), %val() 機能が実装されていないために地球シミュレータセンターにある PC クラスタを用いてプログラムの実行を行った．IAQM を用いたプログラムと全体並列化の改善後の実行時間と高速化率を Table 4 に示す．

Table 4 Improvement by IAQM

		Calculation time[sec]		Speedup ratio	
		w/o IAQM	IAQM	w/o IAQM	IAQM
Number of Processors	1	14960.9	6841.4	1.00	1.00
	2	8252.9	4265.3	1.81	1.60
	4	4458.3	2387.7	3.36	2.87
	8	2511.3	1443.0	5.97	4.74
	16	1682.1	1026.5	8.89	6.66

Table 4 より，IAQM を用いることで約 15000 秒から約 7000 秒と大幅に性能を改善することができた．この方法は，性能改善に有効な手段であるので HPF コンパイラでの実装が望まれる．

２．４　APSI

APSI は，3 次元流体を用いて数値計算を行い，湖環境における汚染物質の拡散をシミュレーションするプログラムである．

APSI では，MGRID と同様に大きな 1 次元配列を確保し，副プログラムではその一部を配列変数として用いており，引数の次元数不一致問題が存在した．MGRID の HPF 化により，引数の次元数不一致問題を解決するために配列の動的確保・解放，データの複写を行ったのでは効率が悪く，大きな 1 次元配列を解体して個別に配列を定義するように書き換えを行うことで効率良く並列化できることが分っている．そこで，APSI の引数の次元数不一致問題も大きな 1 次元配列を解体して個別に配列を定義することで並列化を行った．

また，APSI では並列処理を行う次元が並列化されている場所によって異なる場合がある．HPF ではデータ分散を行っている次元でしか並列処理ができないので，並列化に適した次元で適宜データを再分散しなければならない．このデータ再分散は，副プログラム呼び出し時の引数再マッピングで行った．

更に，以下のように副プログラムが並列に呼び出されるときに i の値に応じて c の nxny 個のデータが引数として渡されている場合があった．

```
      dimension c(nx*ny*nz)
      ●●●
      mlag = 1 - nxny
!$OMP PARALLEL DO PRIVATE(mlag)
      do i = 1 , nz
        mlag = mlag + nxny
        call sub(nx,ny,c(mlag))
      enddo
!$OMP END DO
!$OMP END PARALLEL
      ●●●
      subroutine sub(nx,ny,c)
      dimension c(nx,ny)
      ●●●
```

HPF では，分散された配列をそのまま並列呼び出しされる副プログラムへは渡せないため，以下に示すように必要なデータのみを渡す形状引継ぎ配列を用いて書き換えた．

```
      dimension c(nx,ny,nz)
!HPF$ DISTRIBUTE (*,*,BLOCK) ONTO proc :: c
      interface
        pure extrinsic('fortran','local') subroutine sub(nx,ny,c)
          ●●●
        end subroutine
      end interface
      ●●●
!HPF$ INDEPEDENT
      do i = 1 , nz
        call sub(nx,ny,c(:,:,i))
      enddo
      ●●●
      subroutine sub(nx,ny,c)
      dimension c(nx,ny)
      ●●●
```

以上により並列化されたプログラムの実行時間と高速化率を Table 5 に示す．Table 5 の HPF の実行結果は，高速化率は出ているが OpenMP に比べ実行時間は長い．プロファイラの情報から原因を調べたところ，HPF 化によりベクトル長が一部短くなっている点，プログラムの並列呼び出し時に形状引継ぎ配列を用いてデータを渡しているが，そのときに複写が行われている点，コンパイラがエラー発生時にそれが起こった場所を特定するためのソースコードを追加する点が実行時間の増加を招いたことが分った．HPF 化によってベクトル長が短くなった点とデー

タが複写される点はプログラミングの工夫では避けられなかった．最後のエラー対応のコードが付加される問題は，それを禁止するコンパイラオプション（-Mnoentry）を用いて改善を行った．その結果，Table 5 に示すようにコンパイラオプションを追加するだけで大きく性能を改善できた．

Table 5 Calculation time and SpeedUp rate of APSI

		Calculation time[sec]			Speedup ratio		
		HPF	with -Mnoentry	OpenMP	HPF	with -Mnoentry	OpenMP
Number of Processors	1	1495.9	1020.5	928.4	1.00	1.00	1.00
	2	763.2	541.6	477.6	1.96	1.88	1.94
	3	524.3	358.9	311.6	2.83	2.84	2.98
	4	388.8	267.4	235.4	3.84	3.82	3.94
	8	271.0	171.2	136.6	5.52	5.96	6.80

3. まとめ

SPEC OMP2001 ベンチマーク中のプログラム SWIM，MGRID，APSI を HPF で書き換え，大阪大学サイバーメディアセンターの NEC SX-5 と地球シミュレータセンターの PC クラスタを用いて実行時間と高速化率を計測した．また，その中で明らかになった問題点の解決に取組み性能評価を行った．

SWIM の書き換えでは，不適切分散問題が見つかった．この問題は，ALIGN 指示文の利用と DO ループを分けることで解決できることが分った．MGRID の書き換えでは，引数の次元数不一致問題が見つかった．この問題自体は，必要な次元数の配列を動的に確保し，その配列にデータを複写することで回避できたが高い高速化率を得られなかった．その改善案として ifc の loc() と%val() という関数を用いる IAQM によりプログラムに修正を加えることで配列の動的確保・解放とデータの複写，DO ループを展開を行うことなく並列化することができた．

APSI では，並列化を行う次元が異なる場合が存在した．この問題は副プログラム毎に適したデータ分散を指定して副プログラム呼び出し時に再マッピングを行うことで，OpenMP と同様に異なる次元での並列化を実現した．

HPF は不適切分散問題，引数の次元数不一致問題のように十分な対応ができていない部分もあるが，工夫することで並列化が可能であることが分った．

謝辞

本研究の実施において，並列計算機 NEC SX-5 の利用環境を提供していただいた大阪大学サイバーメディアセンターと PC クラスタの利用環境を提供していただいた地球シミュレータセンターに謝意を表する．また，地球シミュレータセンターの村井均氏の協力に謝意を表する．

参考文献

[1] Message Passing Interface Forum:A Massage-Passing Interface Standard, Int. J. Supercomputing. Applications and High Performance Computing, Vol.8, No.3/4, pp.165-416 (1994).

[2] Message Passing Interface Forum:Extensions to the Message-Passing Interface (1997).

[3] OpenMP Architecture Review Board:“OpenMP Fortran Application Program Interface”，富士通株式会社 訳，(1999).

[4] High Performance Fortran Forum:High Performance Fortran language specification, version2.0 (1996).

[5] High Performance Fortran Forum:High Performance Fortran2.0 公式マニュアル，シュプリンガー・フェアラーク東京株式会社，(1999).

[6] http://www.hpfpc.org/jahpf/.

[7] http://www.specbench.org/.

[8] 森井宏幸，坂上仁志，新居学，高橋豊:HPF の性能評価と応用に関する研究，情報処理学会研究報告 2003-HPC-95，情報処理学会，pp.143-148 (2003).

[9] 妹尾義樹:HPF 言語の現状と将来，情報処理，38，情報処理学会，pp.90-99 (1997).