J. Plasma Fusion Res. Vol.83, No.1 (2007) 94-100

連載コラム High Performance Fortran で並列計算を始めよう

6．見てみよう実用プログラム

坂上仁志
核融合科学研究所

（原稿受付：2006年11月22日）

今回は，シミュレーションに用いられる実用的なプログラム（コード）について，どのようにHPF化したのかを具体的に示します．

6.1 3次元流体コード

このコードは，レーザー核融合の爆縮過程における流体力学的な不安定性を研究するために使っています．非球対象かつ原点を跨ぐフル3次元の現象をシミュレーションするために，非粘性，圧縮性の3次元流体方程式をカーテシアン座標系により計算しています．5点差分による空間微分と陽的解法による時間積分を用い，多次元の時間発展は分ステップ法（fractional step method）を用いています．それでは，このコードをHPFで並列化してみましょう[1]．

6.1.1 領域の分割

陽的解法を用いているので計算には近距離の絡み合いしかなく，計算空間を部分領域に分割して，各々の部分領域を別々のプロセッサに割り振って並列計算を行うことが基本的には容易です．例えば，64個に分割する場合を考えましょう．この場合，ハムを1方向に切るように64スライスにしたり，四角いケーキを縦横に8等分ずつ切るようにしたり，ルービックキューブのように全方向に4等分ずつして小さなサイコロに切ることが考えられます．もし，X方向に分割していると，その方向の時間発展を計算するためには，X方向に隣接するプロセッサ間で境界であるYZ平面のデータを通信して交換する必要があります．このとき，1プロセッサが交換すべきデータ量は，YおよびZ方向に分割していると少なくなります．また，分割していない方向の計算には，通信の必要はありません．一般に，分割する方向が増えるほど，1プロセッサが交換するデータ量は少なくなりますが，データ交換の回数は増えます．大雑把に言うと，Ethernet接続のPCクラスタのように演算能力に比べて通信速度が比較的遅い並列計算機では多方向分割が向いており，特別なネットワークハードウェアを用いているため通信速度は速くてもその初期化時間が目立つような並列計算機では1方向分割が向いている可能性があります．一方，ベクトルプロセッサの場合，分割によってベクトル長が短くなるとパフォーマンスが低下することが多いため，特定の方向には分割しない方がいいことも考えられます[2]．このように，どの分割方法が最善かは，利用する並列計算機のアーキテクチャに大きく依存しますが，ここでは地球シミュレータを考えます[3]．地球シミュレータのネットワークは高速通信を実現していますが，多段スイッチ構成なので通信の初期化時間は目立ちます．また，プロセッサはベクトル型なのでベクトル長を確保する分割方法を採用しなければなりません．そこで，Z方向にのみ領域を分割することにしました[4]．

```
      parameter(lx=1024,ly=1024,lz=1024,lpara=64)
      common /ci3ds1/ sr(lx,ly,lz), sm(lx,ly,lz)
!HPF$ PROCESSORS proc(lpara)
!HPF$ DISTRIBUTE (*,*,BLOCK) ONTO proc :: sr, sm
```

Fig. 1　3次元領域に対するデータ分散．

このコードで使っている変数は，COMMON文によってサブルーチン間で共有されています．また，そのCOMMON文が必要な場所には，INCLUDE行によってソースを展開するようにしています．このため，HPF化するための修正は，INCLUDEされるファイル中にPROCESSORS指示文と領域分割に対応するデータ分散をDISTRIBUTE指示文として書くだけです．これをFig. 1に示します．

プロセッサ台数は，DISTRIBUTE指示文で「ONTO proc」を省略すると可変にできますが，ここでは確実に並列実行性能を得るためにPARAMETER文で与えています．この場合，利用したいプロセッサ台数毎にロードモジュールを作成する必要がありますが，実際の計算ではプロセッサ数を変更することはほとんどなかったので，それほど不便には感じませんでした．

6.1.2 ループの並列化

このコードでは，主要な計算はすべて三重のDOループになっています．最内DOループのインデックスはX方向に関連して3次元配列の1次元目をアクセスし，次のDOインデックスはY方向に関連して2次元目をアクセスし，最外インデックスはZ方向に関連して3次元目をアクセスします．したがって，並列化によるループ処理の分担は3次元目で行い，1次元目はメモリを連続アクセスして

Let Us Start Parallel Processing Using High Performance Fortran ! 6. Let Us Look into Application Programs

SAKAGAMI Hitoshi
author's e-mail: sakagami.hitoshi@nifs.ac.jp

©2007 The Japan Society of Plasma Science and Nuclear Fusion Research

```
!HPF$ INDEPENDENT
      do iz = 1, lz-1
        do 10 iy = 1, ly
          do 10 ix = 1, lx
             wr(ix,iy,iz) = sr(ix,iy,iz) + …
   10     continue
      end do
      …
!HPF$ INDEPENDENT, REDUCTION(max:wram)
      do iz = 1, lz
        do 20 iy = 1, ly
          do 20 ix = 1, lx
             wram = max(wram, …, … )
   20     continue
      end do
```

Fig. 2　ループの並列化.

高速にベクトル処理することができます．コンパイラによっては，1 次元目と 2 次元目のループを一重化し，より効率の良いベクトル処理をするかもしれません．

コンパイラの並列処理に対する解析能力が高いと，Fig. 1 に示した HPF 指示文だけで自動並列化されるかもしれません．しかし，どんな場合には自動並列化されて，どんな場合にはされないかをソースを見ただけで判断するには，多様なケースについての知識が不可欠であり，一般には，コンパイラの専門家でないと難しいです．そこで，ここでは大した手間でもないので，INDEPENDENT 指示文を最外 DO ループに対して無条件に書いて，その並列化をコンパイラに指示します．また，時間ステップの計算には，スカラ変数の全空間における最大値が必要なので，この計算ループを並列化するために INDEPENDENT 指示文に REDUCTION 節も書きます．これを Fig. 2 に示します．

6. 1. 3　通信の最適化

3 次元計算領域を Z 方向に分割しているだけなので，X 方向および Y 方向の時間発展の計算には，通信は必要なく非常に効率のよい並列実行ができます．しかし，Z 方向の時間発展計算には，通信が必要です．この通信をコンパイラにすべて任せてもいいのですが，必ずしも効率の良い通信になるとは限りません．また，ここでも，どんな場合には効率の良い通信になり，どんな場合にはならないかを判断することは難しいですし，その判断基準もコンパイラにより異なってしまいます．そこで，SHADOW，REFLECT および ON-HOME-LOCAL 指示文[1]を用いてこの通信を最適化し，どんな HPF コンパイラでも効率の良い適切な通信が行われるようにします．

まず，Fig. 3 に示す陽的解法の典型的なプログラムを使って SHADOW および REFLECT 指示文の説明をします．

Fig. 3 のプログラムで配列 a，b を BLOCK 分散すると，プロセッサの境目では，a (i) の値を計算するために b (i + 1)

```
real a(1000), b(1000)
…
do i = 2, 999
  a(i) = b(i-1) + b(i) + b(i+1)
end do
```

Fig. 3　陽的解法の典型的なプログラム.

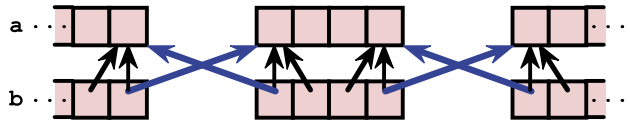
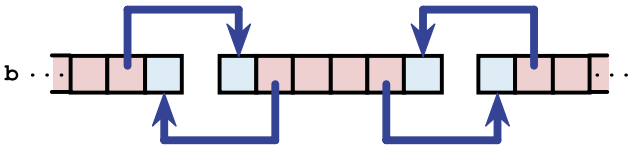
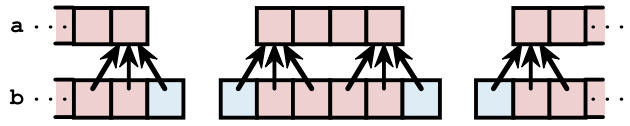


Fig. 4　プロセッサ境目における通信.



（a）　袖領域と一括通信



（b）　袖領域を使った計算

Fig. 5　袖領域による通信の最適化.

または b（i-1）の値を隣のプロセッサから通信して自プロセッサに持ってくる必要があります．これを Fig. 4 に青色の矢印で示します．

この b（i± 1）の値を通信により自プロセッサの配列 b とは別の領域で受けると，境目における計算では，配列 b とその別の領域のどちらを参照するか判定して処理するようにプログラムしなければなりません．たとえこれをコンパイラが自動で行ってくれたとしても，実行効率の良いプログラムになるとは保障されません．また，コンパイラがナイーブに通信を生成すると，効率の悪い通信パターンになるかもしれません．そこで，まず，配列 b の両側を大きめにとって（この領域は，一般には袖領域と呼ばれますが，HPF ではシャドウ領域と呼びます），隣接プロセッサからのデータは，その袖領域に受けるようにします．そして，境目におけるデータ交換のための通信は，一括して効率良く実行するようコンパイラに指示します．これを Fig. 5 (a) に示します．袖領域は水色で，一括通信は青矢印で表されています．いったん袖領域にデータが転送されると，Fig. 5 (b) に示すように，計算そのものは，通信しないで効率良く行われます．

この袖領域を宣言するのが SHADOW 指示文で，一括通信を指示するのが REFLECT 指示文です．なお，REFLECT 指示文は，宣言文ではなく実行文に相当します．Fig. 6 に袖領域への通信を明示した HPF プログラムを示します．

1　文法的には，HOME 節と LOCAL 節を持つ ON 指示文です．よく使われるパターンなので，こう呼ぶことにします．

```
      real a(1000), b(1000)
!HPF$ PROCESSORS proc(4)
!HPF$ DISTRIBUTE (BLOCK) ONTO proc :: a, b
!HPF$ SHADOW b(1)
      …
!HPF$ REFLECT b
!HPF$ INDEPENDENT
      do i = 2, 999
        a(i) = b(i-1) + b(i) + b(i+1)
      end do
```

Fig. 6 袖領域の通信を明示した HPF プログラム.

```
      …
!HPF$ REFLECT b
!HPF$ INDEPENDENT
      do i = 2, 999
!HPF$ ON HOME(a(i)), LOCAL
        a(i) = b(i-1) + b(i) + b(i+1)
      end do
      call sub ( b )
!HPF$ INDEPENDENT
      do i = 1, 999
!HPF$ ON HOME(c(i)), LOCAL
        c(i) = ( b(i) + b(i+1) ) / 2.0
      end do
```

Fig. 7 ON-HOME-LOCAL 指示文による無駄な通信の抑制.

さて，これで袖領域を使った通信の最適化は実現できましたが，ユーザには通信の必要がないとわかっていても，コンパイラはプログラムを解析してそのことを確信できない限り安全サイドを見込むため，無駄な通信をするかもしれません．また，コンパイラには通信をしなくていいかどうか判断できない場合もあります．こういう場合には，ON-HOME-LOCAL 指示文により，完全にローカルな変数だけで実行できることを明示して，無駄な通信を抑制することができます．Fig. 7 に示したプログラムでは，最初の DO ループに必要な通信は REFLECT 指示文で実行していますので，コンパイラが無駄な通信を生成しないように ON-HOME-LOCAL 指示文を書いています．また，2 番目の DO ループでは，配列 b がサブルーチン sub の呼出しによって更新されるかどうかコンパイラには判断できないため，通信をしてしまいます．サブルーチン内で配列 b が参照しかされないなら，既に REFLECT 指示文によって袖領域の値は確定していますので通信の必要はありませんから，これを ON-HOME-LOCAL 指示文により明示します．

このコードは 5 点差分を使っていますが，実際には，i, i + 1 の値から i+1/2 の値を計算したり，i±1/2 の値から i の値を計算する複数のステップから構成されています．このため，袖領域は，変数によって上側か下側のどちらかしか必要ありません．しかし，Fig. 6 の SHADOW 指示文では両側に袖領域を確保し，REFLECT 指示文はその両方に通信します．このため，コードは正しく実行されますが，アクセスされることがない袖領域に対して無駄な通信が行われてしまいます．そこで，必要な片側の袖領域だけを SHADOW 指示文において明示し，更なる通信の最適化を行います．これを Fig. 8 に示します．

```
!HPF$ SHADOW (0:0,0:0,0:1) :: wr
!HPF$ SHADOW (0:0,0:0,1:0) :: wp
!HPF$ REFLECT wr
!HPF$ INDEPENDENT
      do iz = 1, lz-1
!HPF$ ON HOME(wp(:,:,iz)), LOCAL BEGIN
        do 30 iy = 1, ly
          do 30 ix = 1, lx
            wp(ix,iy,iz) = wr(ix,iy,iz) +
     &                     wr(ix,iy,iz+1)
   30   continue
!HPF$ END ON
      end do
!HPF$ REFLECT wp
!HPF$ INDEPENDENT
      do iz = 2, lz-1
!HPF$ ON HOME(we(:,:,iz)), LOCAL BEGIN
        do 40 iy = 1, ly
          do 40 ix = 1, lx
            we(ix,iy,iz) = wp(ix,iy,iz) +
     &                     wp(ix,iy,iz-1)
   40   continue
!HPF$ END ON
      end do
```

Fig. 8 片側 SHADOW 指示文による通信の最適化.

コンパイラは，ここに述べたすべての最適化を自動的に行ってくれるかもしれませんが，最善な袖領域および通信になるとは限りませんので，マニュアルでやることをお奨めします．もちろん，このようなマニュアル最適化をやらずに並列実行してみて，性能が悪かったら一歩ずつ最適化を試みて，満足な性能が得られればそれ以上の最適化をしないという手もあります．このように，ステップバイステップで並列化できることは，HPF の大きな利点です．

また，ここでは 3 次元計算領域を Z 方向にしか分割しませんでしたが，複数方向に分割したい場合でも HPF を使えば，ほとんど同じくらいの簡単な手間だけでできてしまいます．ですから，いろいろな分割方法を試してみるのも興味深いと思います．このような分割方法の変更を簡単に試せるのも，また，HPF の利点と言えます．

差分法を陽的解法で解くコードについては，SHADOW，REFLECT，ON-HOME-LOCAL 指示文を使えば，大抵の場合は効率良く並列化できますので，ぜひ，ご自分のコードでもチャレンジしてみてください．なお，前回までの連載で解説した姫野ベンチ HPF 版でも，これらの指示文が使用されていますので参考にしてください．

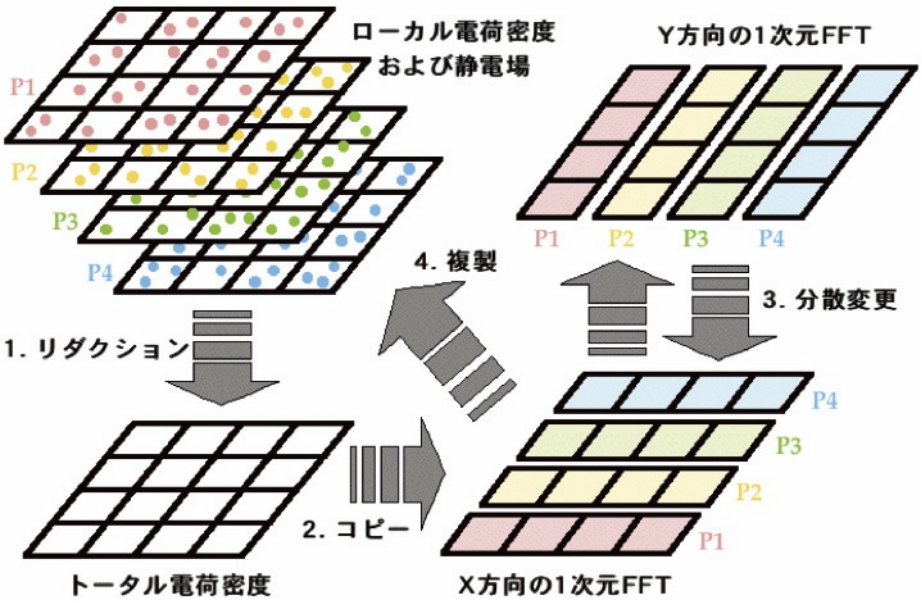


Fig. 9 フィールドの配列についての分散.

6.2 2次元静電粒子コード

このコードは，古典的 Particle-In-Cell 法を用いた 2 次元静電粒子コードで，プラズマの基礎物理を研究するために使っています．粒子の位置から電荷密度分布を求め，2 次元 FFT を用いてポアソン方程式を解き，静電場を求めます．そして，この静電場内における粒子の運動を運動方程式を時間積分することにより計算しています．それでは次に，このコードを HPF で並列化してみましょう[1]．

6.2.1 並列化の方針

MPIを用いた粒子コードの並列化には領域分割法を用いることが多いです[5]．この場合，各プロセッサは自身が受け持つ領域内に存在する粒子およびフィールドに関する計算を行います．この手法では，粒子がある領域から別の領域に移動すると，その粒子の計算を受け持つプロセッサが替わるので，その粒子の情報をプロセッサ間でやり取りしなければなりません．しかし，このような通信パターンを HPF で記述するためには，かなりトリッキーなコーディングが必要になり，付加的な指示文を挿入するだけでプログラムの並列化を行う HPF 本来の趣旨から逸脱します．そこで，粒子情報を格納している配列を単純に BLOCK 分散して並列化を行うことにします．この粒子分割法では，粒子の運動を計算する DO ループは，無理なく効率よく並列化できます．しかし，粒子は 2 次元領域を自由に移動するため，各プロセッサはフィールドの情報を全空間について重複して持つ必要があります．各プロセッサは，自身が受け持つ粒子の位置情報から全空間に分布するローカルな電荷密度を計算し，そのローカル電荷密度に対して集計演算を行うことにより，トータルな電荷密度を求めます．トータルな電荷密度が求まると，それを 2 次元FFTして電荷密度のフーリエ係数を求めますが，2 次元 FFT は 1 次元 FFT の重ね合わせで行います．そこで，まず，X 方向の 1 次元 FFT が並列実行できるように，トータル電荷密度を Y 方向に分散された配列にコピーし，1 次元FFTサブルーチンの並列呼出しを可能にします．次に，Y 方向の 1 次元 FFT が並列実行できるように配列の分散を自動再マッピングにより変更し，1 次元FFTサブルーチンを並列に呼び出します．そして，電荷密度のフーリエ係数が求まると，それにフォームファクタを乗じて電位のフーリエ係数を計算し，それを 2 次元逆フーリエ変換することで電位を求めます．このときも，1 次元逆フーリエ変換の方向にあわせて，同様に配列を自動再マッピングします．各プロセッサは，電位について全空間の情報が必要なので，求まった電位を複製します．この様子を Fig. 9 に示します．

6.2.2 電荷密度の計算

電荷密度を計算する典型的なループには，電荷密度を格納する配列に対しての不規則な間接参照が含まれていますので，単純に並列化して実行すると同じ配列要素への書込みが起こり，計算結果が不正になります．この配列に分散配列を用いて，かつ不正な計算結果にならないよう並列化するためには，配列への複雑なアクセスパターンが必要になり，HPF コンパイラは効率よく並列化することができないのが現状です．そこで，プロセッサ毎にローカルな電荷密度をいったん一時配列に求め，最後に集計演算を行ってトータルな電荷密度を計算することを考えます．このとき，重複配列を一時配列として用いると，各プロセッサは自身の受け持つ粒子の情報からローカルな電荷密度を効率よく並列計算できるようになります．また，全プロセッサのローカル電荷密度から集計演算によりトータル電荷密度を求める面倒な通信を含んだ計算は，重複配列を REDUCTION 節に記述することにより，すべて HPF コンパイラに任せてしまいます．最後に，トータル電荷密度を格納している重複配列から並列 FFT のための分散配列にデータをコピーします．この HPF プログラムを Fig. 10 に示します．

```
      parameter( lx=256, ly=256, no=lx*ly*100)
      parameter( lpara=32 )
      dimension xe(no),ye(no),rhotmp(lx,ly),
     &          rho(lx,ly)
!HPF$ PROCESSORS proc(lpara)
!HPF$ DISTRIBUTE (BLOCK) ONTO proc :: xe,ye
!HPF$ DISTRIBUTE (*,BLOCK) ONTO proc :: rho
      do j = 1, ly
         do i = 1, lx
            rhotmp(ix,iy) = 0.0
         end do
      end do
!HPF$ INDEPENDENT, REDUCTION(+:rhotmp)
      do i = 1, no
         ixe  = xe(i)
         dxe  = xe(i) - ixe
         ddxe = 1.0-dxe
         iye  = ye(i)
         dye  = ye(i) - iye
         ddye = 1.0 - dye
         rhotmp(ixe  ,iye  ) =
     &          rhotmp(ixe  ,iye  ) - ddxe * ddye
         rhotmp(ixe+1,iye  ) =
     &          rhotmp(ixe+1,iye  ) -  dxe * ddye
         rhotmp(ixe  ,iye+1) =
     &          rhotmp(ixe  ,iye+1) - ddxe *  dye
         rhotmp(ixe+1,iye+1) =
     &          rhotmp(ixe+1,iye+1) -  dxe *  dye
      end do
!HPF$ INDEPENDENT
      do j = 1,ly
         do i = 1,lx
            rho(i,j) = rhotmp(i,j)
         end do
      end do
```

Fig. 10 電荷密度の並列計算.

```
      parameter( lx=256, ly=256, lpara=32)
      dimension rho(lx,ly),phi(lx,ly),ck(lx,ly)
!HPF$ PROCESSORS proc(lpara)
!HPF$ DISTRIBUTE (*,BLOCK) ONTO proc :: rho, ck
      dimension fftsx1(lx),fftsx2(lx),lftsx3(15),
     &          fftsy1(ly),fftsy2(ly),lftsy3(15)
c....
      interface
         subroutine rfftfx ( kx, ky, fdat,
     &                       fsx1,fsx2,ksx3)
         parameter( lpara = 32 )
!HPF$ PROCESSORS proc(lpara)
         dimension fsx1(kx),fsx2(kx),ksx3(15),
     &             fdat(kx,ky)
!HPF$ DISTRIBUTE (*,BLOCK) ONTO proc :: fdat
         end subroutine
         subroutine rfftfy ( kx, ky, fdat,
     $                       fsy1, fsy2, ksy3)
         parameter( lpara = 32 )
!HPF$ PROCESSORS proc(lpara)
         dimension fsy1(ky),fsy2(ky),ksy3(15),
     &             fdat(kx,ky)
!HPF$ DISTRIBUTE (BLOCK,*) ONTO proc :: fdat
         end subroutine
      end interface
c....
      (rho の計算)
c..                                        * 順フーリエ変換 *
      call rfftfx (lx,ly,rho,fftsx1,fftsx2,lftsx3)
      call rfftfy (lx,ly,rho,fftsy1,fftsy2,lftsy3)
c..                                     * フォームファクター *
!HPF$ INDEPENDENT
      do j=1,ly
         do i=1,lx
            rho(i,j)=rho(i,j)*ck(i,j)
         end do
      end do
c..                                        * 逆フーリエ変換 *
      call rfftby (lx,ly,rho,fftsy1,fftsy2,lftsy3)
      call rfftbx (lx,ly,rho,fftsx1,fftsx2,lftsx3)
c..                                                * 複製 *
      do j=1,ly
         do i=1,lx
            phi(i,j) = rho(i,j)
         end do
      end do
      stop
      end
c--------------------
      subroutine rfftfx (kx,ky,fdat,fsx1,fsx2,ksx3)
      parameter( lx=256, ly=256, lpara=32 )
!HPF$ PROCESSORS proc (lpara)
      dimension fsx1(kx),fsx2(kx),ksx3(15),
     &          fdat(kx,ky)
!HPF$ DISTRIBUTE (*,BLOCK) ONTO proc :: fdat
c....
      interface
         extrinsic('FORTRAN','LOCAL')
     $   subroutine rfftf( k, ftmp, f1, f2, k3 )
         dimension ftmp(k),f1(k),f2(k), k3(15)
         intent(inout) :: ftmp,f1
         intent(in) :: k,f2,k3
         end subroutine
      end interface
c....
!HPF$ INDEPENDENT
      do iy = 1, ky
ccc        call rfftf ( kx,fdat(1,iy),fsx1,fsx2,ksx3 )
         call rfftf ( kx,fdat(:,iy),fsx1,fsx2,ksx3 )
      end do
      return
      end
c--------------------
      subroutine rfftfy (kx,ky,fdat,fsy1,fsy2,ksy3)
      parameter( lx=256, ly=256, lpara=32 )
!HPF$ PROCESSORS proc(lpara)
      dimension fsy1(ky),fsy2(ky),ksy3(15),
     &          fdat(kx,ky),ftmpy(ly)
!HPF$ DISTRIBUTE (BLOCK,*) ONTO proc :: fdat
c....
      interface
         extrinsic('FORTRAN','LOCAL')
     $   subroutine rfftf( k, ftmp, f1, f2, k3 )
         dimension ftmp(k),f1(k),f2(k), k3(15)
         intent(inout) :: ftmp,f1
         intent(in) :: k,f2,k3
         end subroutine
      end interface
c....
!HPF$ INDEPENDENT, NEW(ftmpy)
      do ix = 1, kx
         do iy = 1, ky
            ftmpy(iy) = fdat(ix,iy)
         end do
         call rfftf ( ky,ftmpy,fsy1,fsy2,ksy3 )
         do iy = 1, ky
            fdat(ix,iy) = ftmpy(iy)
         end do
      end do
      return
      end
```

Fig. 11 FFTの並列計算.

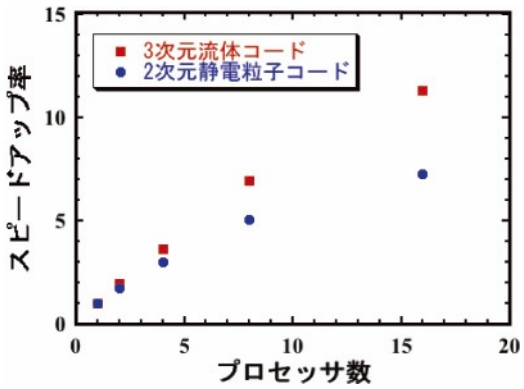


Fig. 12 スピードアップ率.

Table 1　PC クラスタの諸元.

CPU	Athlon XP 2200+ (1.8GHz)
メモリ	512MB
ネットワーク	Gigabit Ethernet
OS	Linux 2.4.22
MPI	mpich1.2.7
コンパイラ	Intel ifort 8.0

6.2.3　FFT の並列化

2 次元FFTの並列化は，分散配列の自動再マッピングおよび 1 次元FFTルーチンの並列呼出しで実現します．分散配列を手続呼出し時に再マップするためには，前回解説したように，呼出し側に INTERFACE 構文を書いて，実引数と仮引数の分散を明示する必要があります．また，Fortran で書かれた 1 次元 FFT サブルーチンを並列に呼び出すためには，呼出し側に INTERFACE 構文を書いて，呼び出されるのが Fortran 手続であることや引数の属性をすべて指定する必要がありました．そして，Fortran のサブルーチンを呼び出すときは，実引数で渡されたアドレスを仮引数では配列の先頭と見做すことはできません．このため，X 方向の FFT では部分配列を使うようにソースを若干修正して並列化しました[2]．Y 方向のFFTでは，一時配列を重複配列とすることで簡単に並列化できます．このHPFプログラムを Fig. 11 に示します．なお，Fig. 11 において，逆フーリエ変換のサブルーチンは，順フーリエ変換のサブルーチンと同様なので省略しています．

6.3　並列性能

今回，HPF 化を行った 2 つのコードの完全なソースは，HPF 推進協議会のホームページ[6]からダウンロードできますので，この解説記事ではページ数の都合で説明しきれなかった部分もご覧になれます．このコードを fhpf でコンパイルしてTable 1のPCクラスタで実行したときのスピードアップ率を Fig. 12 に示します．また，シミュレーションパラメータは，ダウンロードできるソースに書かれている値を，そのまま使っています．両コード共，お手軽に並列化しただけですが，安価なPCクラスタでも，そこそこの並列実行性能が得られています．

なお，3 次元流体コードでは，5 点空間差分を複数のステップで構成していますので，そのステップ毎に RE-FLECT 指示文による袖領域の通信が必要となります．プロセッサ間で計算の重複を許せば，本シリーズでは未説明の ON-EXT_HOME 指示文を使って，この通信回数を 1 回に減らすことができます[7]．演算能力と比べて通信性能が劣っている並列計算機では，このようなチューニングによって性能向上が見込めます．また，2 次元静電粒子コードでは，自動再マッピングを用いて配列の分散を並列化に最適な形に変更していましたが，この通信を伴う分散の変更は，全部で 4 回起こります．でも，ソースを少し修正して新しいサブルーチンを作るか，再マッピングを配列間の代入により手動で行えば，この通信を 2 回に減らすことができますので，やはり，通信速度が遅い並列計算機では，有効なチューニングになると思います．

このように，並列性能の向上を目指して段階的にチューニング作業ができることは，HPF の大きな利点です．

通信の最適化

・袖領域の指示　・・・*宣言部で*

```
!HPF$ SHADOW a(<シャドウ幅>,...),...
```

または

```
!HPF$ SHADOW (<シャドウ幅>,...)::a,b,...
```

aやbは，配列変数.

<シャドウ幅>は，nまたは，l:uで袖の大きさを指定.

nは，n:nと同じ.

・袖領域に対する通信　・・・*実行部で*

```
!HPF$ REFLECT a,b,...
```

aやbは，シャドウを宣言した配列変数.

・無駄な通信の抑制　・・・*実行部で*

```
!HPF$ ON HOME(<ホーム変数>),LOCAL
      <1実行文か1構文>
```

または

```
!HPF$ ON HOME(<ホーム変数>),LOCAL BEGIN
      <複数の実行文か構文>
      ...
!HPF$ END ON
```

Fig. 13 今回解説した構文のまとめ.

6.4　まとめ

実用的なシミュレーション用のコードでも，HPF でお手軽に並列化ができて，安価な PC クラスタでも並列実行ができ，そこそこの性能が得られます．また，1 台ではメモリが足りなくて実行できない規模の問題でも，主な配列を分散できるなら，複数台で並列実行すると 1 台当りに必要なメモリ量を減らすことができますので，大規模な問題でもシミュレーションできるようになります．ぜひ，ご自分のコードでも HPF をトライしてみてください．

残念ながら現状では，不規則な構造を持つコードは，HPF による並列化は難しいことが多いですが，規則的な構造を持つコードなら比較的簡単にHPF化できて，十分な並列実行性能が得られることが多いです．ですから，コードの構造が規則的な場合には，最初から苦労して MPI を用いて並列化するより，まず，HPF による並列化を試してみる価値はあると思います．このときのポイントは，プログラムの構造を考慮して，できるだけ通信が起こらないようなデータの分散方法を考えることです．ただし，HPF で並列化できても並列実行すると，十分な性能が得られない場合があります．この場合，その原因を追及することは，慣れ

2　このためには，fhpf バージョン1.4.2が必要です．

ないと取っ付きにくいのがHPFの難点ですが，コンパイラの出力する詳細メッセージを参考にすればよいでしょう．そして，いったんその原因がわかれば，比較的簡単に解決できる場合も多いです[8]．しかし，その解決方法もHPFでプログラムした経験が少ないと直感的にわかりにくいかもしれません．もし，ご自分だけで解決できない場合は，HPF推進協議会に御相談していただければ，その原因究明および解決のお手伝いができますので，ぜひ，ご連絡ください．

謝辞

PCクラスタでの並列性能の計測については，兵庫県立大学大学院工学研究科の情報制御機構研究グループに協力いただきましたので，ここに謝意を表します．

参考文献

［1］坂上仁志，水野貴夫：国産HPFコンパイラの性能評価と互換性検証，情報処理学会研究報告，2001-HPC-87, 73-78 (2001).
［2］坂上仁志，高橋豊：3次元流体コードの領域分割法における並列効率，電情通学論 J80-D-I, 683-690 (1997).
［3］http://www.es.jamstec.go.jp/esc/jp/.
［4］H. Sakagami, H. Murai, Y. Seo and M. Yokokawa: 14.9 TFLOPS Three-dimensional Fluid Simulation for Fusion Science with HPF on the Earth Simulator, SC2002, Baltimore, USA, November 16-22, pap147 (2002). (Gordon Bell Awards)
［5］村田健史，上岡功治，高橋誠治，岡田雅樹，上田裕子，大村善治，松本紘：プラズマ電磁粒子コードの並列化手法と速度向上率の評価，情処学論 数理モデル化と応用，43, 118-131 (2002).
［6］http://www.hpfpc.org/.
［7］H. Sakagami, T. Mizuno and S. Furubayashi: Parallelization Methods for Three-Dimensional Fluid Code using High Performance Fortran, Parallel Computational Fluid Dynamics (eds. K. Matsuno, et al.), 203-210 (Elsevier, North-Holland, 2002).
［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, 143-148 (2003).

坂上仁志（さかがみひとし）

1993年兵庫県立大学（旧：姫路工業大学）大学院工学研究科助教授．2005年核融合科学研究所，理論・シミュレーション研究センター教授．レーザー核融合に関する流体および粒子シミュレーション，大規模並列計算の研究に従事しています．HPFとの関わりは．1997年に発足したHPF合同検討会（JAHPF）にユーザ側の議長として参加したときから始まりました．現在は，HPF推進協議会（HPFPC）の幹事として活動しています．