HPFの概要

核融合科学研究所
シミュレーション科学研究部
坂上仁志
sakagami.hitoshi@nifs.ac.jp

アウトライン

- JAHPF/HPFPC
- HPFの概要
  - 機能とコンパイラ
  - 並列化の基本
  - 指示文と並列化
- プログラム例
  - 3次元流体コード
  - 2次元静電粒子コード
- まとめ

JAHPF

- HPF合同検討会(Japan Association for HPF)
  - http://www.hpfpc.org/jahpf/
  - 1997年1月29日に発足し,活動を開始した.
- HPFをベースとし,以下の3条件を満たす並列言語仕様を確立する.
  - 標準性:多用なプラットフォーム上で動作
  - 適用性:科学技術計算スキームの実装
  - 高速性:ハードウェア性能の活用
- HPF/JA仕様を策定した.

HPFPC

- HPF推進評議会(HPF Promoting Consortium)
  - http://www.hpfpc.org/
  - JAHPFを発展解消し,2001年7月9日に設立総会を開催して,任意団体としての活動を開始した.
- 活動の中心を拡張言語仕様の開発から,実用アプリケーションのHPF化促進,実環境でのHPFの評価にフォーカスする.
  - HPF利用を支援するために,会員所有のコードをHPF化するための個別相談や講習会を行う.

HPFの機能

- HPF 2.0基本機能
  - 単純なデータの分散とループ処理の分担
  - 外部手続の並列呼び出し
- HPF 2.0公認拡張機能
  - 複雑なデータ分散
  - タスク並列実行
- HPF/JA 1.0拡張機能
  - 集約演算の拡張
  - 明示的な通信の最適化

日本のコンパイラ

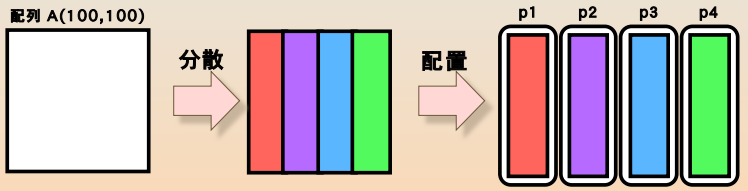
- HPF/ES - 地球シミュレータ用
  - HPF2.0+HPF/JA+独自拡張
- HPF/SX - NEC SXシリーズ用
  - HPF/ESとほぼ互換(除く:並列I/O)
- HPF/ES for PC cluster
  - HPF/ESとほぼ互換のNEC製PCクラスタ用
- HPF/VPP - 富士通VPP用
- fhpf - 富士通製
  - フリーのHPFコンパイラ(HPF推進協議会で配布)

海外のコンパイラ

- ADAPTOR
  - ドイツのGMDで開発されたパブリックドメインのHPFコンパイラ
  - HPF2.0+公認拡張機能の多く
- SHPF
  - オーストリアのVCPCで開発されたパブリックドメインのHPF2.0コンパイラ
- PGHPF Compiler
  - 米国PGI社で開発されたHPFコンパイラ
  - HPF2.0+公認拡張機能の一部

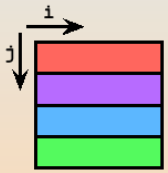
HPFによる並列化#1

- ユーザは,Fortran配列をどのように複数のプロセッサ上へ分散して配置するかだけを明示的に指定する.(データ分散)



HPFによる並列化#2

- 更新データを保持するプロセッサが処理を行うように並列化する.(処理分担)
  - Owner Computes Rule




```
dimension a(100,100),b(100,100)
!HPF$ DISTRIBUTE (*,BLOCK) :: a,b
do j = 2, 100
  do i = 1, 100
    a(i,j) = a(i,j)*b(i,j)+b(i,j-1)
  end do
end do
```

```
p1:j=2,25
p2:j=26,50
p3:j=51,75
p4:j=76,100
```

- 必要なデータ転送は,コンパイラが自動で面倒を見る.

HPF指示文

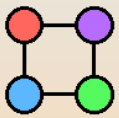
- 抽象プロセッサの定義
- 配列の分散,整列
  - 1次元配列,2次元配列,...
  - BLOCK,CYCLIC,GEN_BLOCK
- 並列実行の指示
  - ループ,変数,演算
- ループ処理分担の指示
- 袖領域の定義と通信
- 不必要な通信の抑制

抽象プロセッサの定義

- 抽象プロセッサの名前,次元およびサイズを定義する.

```
!HPF$ PROCESSORS linear(4)
```

```
!HPF$ PROCESSORS mesh(2,2)
```



- ただし,抽象プロセッサから物理プロセッサへのマッピングは,処理系に依存する.

1次元配列の分散

- BLOCK分散

```
real a(32)
!HPF$ DISTRIBUTE a(BLOCK) ONTO linear
     p1          p2          p3          p4
```

```
a(1)~a(8)   a(9)~a(16)   a(17)~a(24)   a(25)~a(32)
```

- CYCLIC分散

```
real b(32)
!HPF$ DISTRIBUTE b(CYCLIC(2)) ONTO linear
p1 p2 p3 p4
```

```
b(1)~b(2)   b(9)~b(10)   b(17)~b(18)   b(25)~b(26)
```

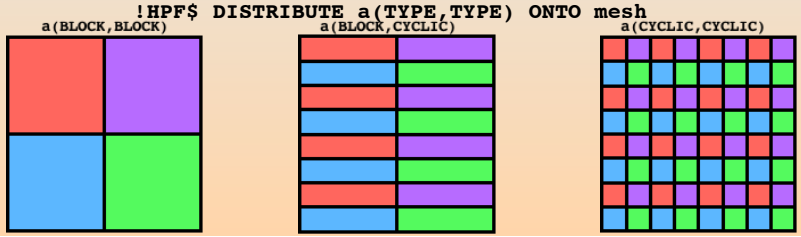
2次元配列の分散

```
real a(8,8)
!HPF$ DISTRIBUTE a(TYPE,TYPE) ONTO linear
```

a(BLOCK,*)　　a(*,CYCLIC)

```
real a(8,8)
!HPF$ DISTRIBUTE a(TYPE,TYPE) ONTO mesh
```

a(BLOCK,BLOCK)　　a(BLOCK,CYCLIC)　　a(CYCLIC,CYCLIC)



不均等分散

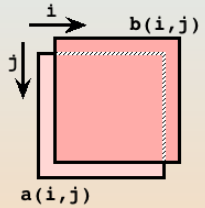
- 3角行列の演算等では,配列を均等に分散すると各プロセッサの計算負荷が不均一になり,全体的な効率が悪くなる.
- GEN_BLOCKを用いた不均等分散で,各プロセッサの計算量を均等にする.

```
      real a(256,256)
      integer,parameter::m(4)=(/128,53,41,34/)
!HPF$ DISTRIBUTE a(*,GEN_BLOCK(m)) ONTO linear
```

配列の整列

- 以下のような場合,同じ分散を指定すると境界でデータ転送が発生する.

```
do j =
  do i =
    a(i,j) = b(i+1,j-1)
  end do
end do
```



- 複数の配列間の相互関係を指定してデータ転送が発生しないように調整する.

```
!HPF$ ALIGN a(i,j) WITH b(i+1,j-1)
```

並列実行の指示(ループ)

- コンパイラだけでは判断できない場合にループを並列に実行しても問題のないことを明示する.

```
!HPF$ INDEPENDENT
      do i = 1, 100
        a(ind(i)) = a(ind(i)) + b(i)
      end do
```

  - ind(i)に重なりがあった場合,並列実行すると回帰参照により正しい実行結果が得られないが,コンパイラはind(i)に重なりがあるかどうか判断できないのでループを並列化しない.
  - そこで,ind(i)に重なりがない場合,コンパイラにループの並列化を指示する.

並列実行の指示(変数)

- ループの各繰返しで一時的に使われる変数があると厳密な意味でのINDEPENDENTにはならない.

```
!HPF$ INDEPENDENT, NEW(tmp)
      do i = 1, 100
        tmp  = a(i) + b(i)
        c(i) = tmp * c(i)
      end do
```

  - あるプロセッサがtmpに値を代入後,参照する前に別のプロセッサがtmpの値を更新するかもしれない.
  - しかし,その変数に対してNEW宣言をすると,繰返し毎に新しい実体を割り当てるため,ループが並列化される.
  - ただし,スカラ変数の場合には,コンパイラが自動的に判断することが多い.

並列実行の指示(演算)

- ループ中に総和等の集約演算があると,その演算結果を求める変数に対してはNEW宣言ができず,ループはINDEPENDENTにはならない.そこで,各プロセッサが部分的な結果を求め,最後に全プロセッサで最終的な結果を求める特別な指示をする.

```
      s = 0.0
!HPF$ INDEPENDENT, REDUCTION(+:s)
      do i = 1, 100
        s = s + a(i) * b(i)
      end do
```

  - 総和計算であることを明示する.

ループ処理分担の指示

- コンパイラのループ処理分担が不適切な場合,データ転送が多発して並列効率が低下するので,分担の仕方を指示する.

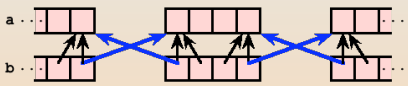
```
      real a(100), b(100), c(100), d(100)
!HPF$ DISTRIBUTE(BLOCK) ONTO linear::a,b,c,d
      do i = 1, 99
!HPF$ ON HOME(b(i+1))
        a(i) = b(i+1) + c(i+1) + d(i+1)
      end do
```

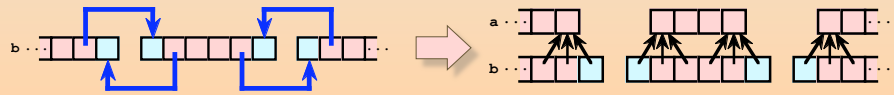
  - この場合,配列(i)を保持するプロセッサより配列(i+1)を保持するプロセッサで実行する方が,データ転送量が少ない.

袖領域の定義とその通信

- 分散配列に袖領域を定義して,その部分を一括通信すると効率が良い.

```
      real a(100), b(100)
!HPF$ DISTRIBUTE(BLOCK) ONTO linear::a,b
!HPF$ SHADOW b(1)
      do i = 1, 100
         b(i) = …
      end do
!HPF$ REFLECT b
!HPF$ INDEPENDENT
      do i = 2, 99
        a(i) = b(i-1) + b(i) + b(i+1)
      end do
```





不必要な通信の抑制

- コンパイラには解析できないために行われる不必要な通信を明示的に抑制する.

```
!HPF$ REFLECT b
!HPF$ INDEPENDENT
      do i = 2, 99
!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, 99
!HPF$ ON HOME(c(i)), LOCAL
        c(i) = ( b(i) + b(i+1) ) / 2.0
      end do
```

LOCALが有効な別の例

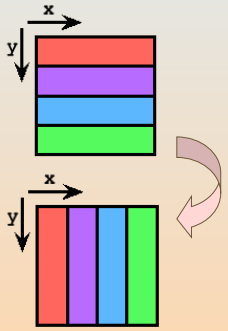
- 上流差分のため,通信すべき方向が実行時の条件により異なる.
  - 常に袖領域が有効かどうか判断できなかった.

```
      do j = 2, maxy-1
        do i = 2, maxx-1
          isign = 1
          if( un(i,j) .lt. 0.0d0 ) isign = -1
          jsign = 1
          if( vn(i,j) .lt. 0.0d0 ) jsign = -1
          im1 = i - isign
          jm1 = j - jsign
!HPF$ ON HOME(fs(i,j)), LOCAL BEGIN
          a8 = fs(i,j) - fs(im1,j) - fs(i,jm1) + fs(im1,jm1)
          ...
!HPF$ END ON
      end do
      end do
```

動的な分散の変更

- 配列に対する最適な分散が異なる場合,分散を動的に変更すると一括して通信が行われ,全体的な効率が良い.

```
      dimension a(256,256)
!HPF$ DISTRIBUTE a(*,BLOCK)
      call subx ( a )
      call suby ( a )
      …
      subroutine subx ( a )
      dimension a(256,256)
!HPF$ DISTRIBUTE a(*,BLOCK)
      …
      subroutine suby ( a )
      dimension a(256,256)
!HPF$ DISTRIBUTE a(BLOCK,*)
```



  - サブルーチンの呼出し時に自動的に変更される.

3次元流体コード

- 3次元流体方程式(非粘性,圧縮性)
- カーテシアン座標系
- 5点差分による空間微分
- 陽的解法による時間積分
- 多次元の時間発展は分ステップ法
- 通常の領域分割で並列化可能

並列化の方針

- Z方向にのみ配列を分散する.
  - 1次元抽象プロセッサ
  - ベクトル処理の効率を考慮
  - Z方向の計算時のみ通信が必要
  - 多方向分散に比べて多い通信データ量
- 時間ステップの計算には,スカラ変数の全空間における値が必要である.
  - REDUCTION演算

並列化の方法

- 並列化を段階的に四つのレベルで行う.
  - DISTレベル(PROCESSORS+DISTRIBUTE)
  - INDレベル(INDEPENDENT)
  - SHADレベル(SHADOW+REFLECT)
  - LOCレベル(ON HOME LOCAL+ENDON)
- HPFでは,指示文を徐々に追加しながら,より高度な並列化ができる!
  - MPIによるプログラミングでは,このような段階的な並列化は難しい.

DISTレベルの並列化

- PROCESSORS指示文による1次元プロセッサの定義とDISTRIBUTE (*,*,BLOCK)指示文によるデータ分散を指示する.

```
      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
```

- 実際には,includeファイルに記述する.
  - 編集作業をしたHPF指示文の行数は少ない.

INDレベルの並列化

- INDEPENDENT指示文を並列実行できるDOループに追加し,並列実行を明示する.

```
!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) + sr(ix,iy,iz+1)
               ...
   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
```

SHADレベルの並列化

- SHADOW+REFLECT指示文により通信を最適化する.
  - 明示的なブロック通信
  - 最適な袖領域による通信量の削減

```
!HPF$ SHADOW (0:0,0:0,0:1) :: sr
!HPF$ SHADOW (0:0,0:0,1:0) :: sp
!HPF$ SHADOW (0:0,0:0,1:1) :: se
      ...
!HPF$ REFLECT sr, sp, se
!HPF$ INDEPENDENT
      do iz = 2,lz-1
         do 30 iy = 1,ly
            do 30 ix = 1,lx
               wr(ix,iy,iz) = sr(ix,iy,iz) + sr(ix,iy,iz+1)
               wp(ix,iy,iz) = sp(ix,iy,iz) + sp(ix,iy,iz-1)
               we(ix,iy,iz) = se(ix,iy,iz-1) + se(ix,iy,iz+1)
   30    continue
      end do
```

LOCレベルの並列化

- LOCAL指示文により不必要な通信を明示的に抑制する.

```
!HPF$ REFLECT sr
!HPF$ INDEPENDENT
      do iz = 1, lz-1
!HPF$ ON HOME(wr(:,:,iz)), LOCAL BEGIN
         do 10 iy = 1, ly
            do 10 ix = 1, lx
               wr(ix,iy,iz) = sr(ix,iy,iz) + sr(ix,iy,iz+1)
               ...
   10    continue
!HPF$ END ON
      end do
```

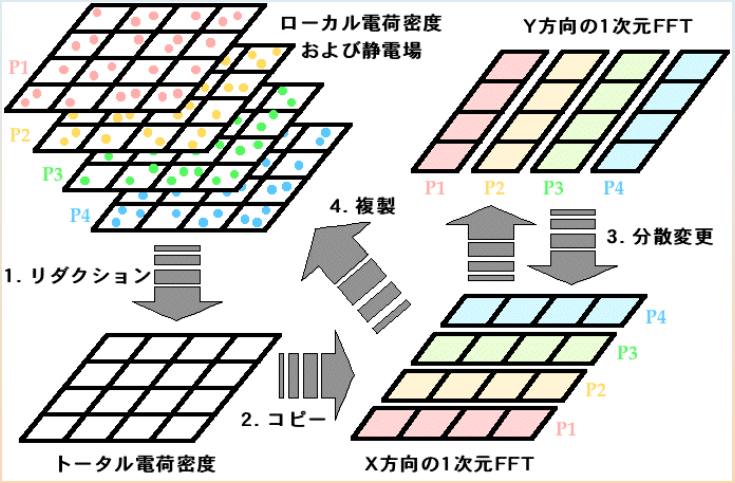
2次元静電粒子コード

- 典型的なParticle In Cell法
- 電子とイオンの運動方程式
- 電荷密度の計算
  - 間接インデックスを用いたランダムな配列アクセス
- 2次元ポアソン方程式
  - 2次元フーリエ変換／逆変換
  - 1次元FFTルーチンの多重呼び出し

並列化の方針

- 粒子データを分散する.
  - 領域を分散するためには,かなりトリッキーなコーディングが必要である.
- 一時配列の導入により電荷密度の並列計算を可能にする.
- 2次元FFTは,1次元FFTルーチンの並列呼び出しにより並列化する.
  - これの並列効率が悪いなら,ここだけ並列化しないという選択肢もある.
- フィールド(場)のデータの分散は,必要に応じて動的に変更する.

場データの動的再分散



一時配列によるHPF化#1

- 各プロセッサ毎に非分散配列を用意して,その配列にreduction演算をする.

```
      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
```

一時配列によるHPF化#2

```
!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
```

2次元FFTのHPF化#1

```
      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
```

2次元FFTのHPF化#2

```
      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
```

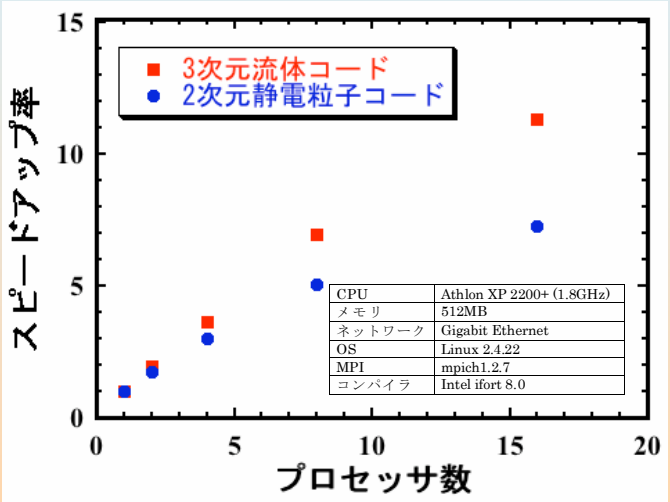
2次元FFTのHPF化#3

```
            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
```

2次元FFTのHPF化#4

```
            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
```

fhpf+PCクラスタによる性能



HPFによるプログラミング

1. プログラムにおいて並列化する部分を決定する.
   - 10%のサブルーチンがCPU時間の90%を費やす.
2. その部分において,並列化のためのデータ転送が最も少いデータ分散方法を決定する.
   - PROCESSORS, DISTRIBUTE指示文
3. コンパイラが自動並列化できないループについて明示的に並列化を指示する.
   - INDEPENDENT指示文
4. 各種最適化を行う.
   - ループ処理分担の追加補足のための指示
   - データ転送を最適化するための指示

まとめ#1

- HPFでは,挿入する指示文を順次増やすことで,ステップバイステップに段階的に並列化やチューニングができる.
  - 並列化の手間を考慮して,得られた並列性能に納得できれば,並列化をそこで止めればよい.
  - 問題があった場合,すぐに前の段階に戻れる.
- とりあえず,手軽にやってみる!
  - 最初から苦労してMPIで不可逆的なプログラミングをするより,まず,HPFによる並列化を試してみる価値は,十分にある!

まとめ#2

- 規則的な構造の問題なら,HPFでも十分な並列性能が得られることが多い.
  - プログラムの構造から,できるだけ通信が起こらないデータ分散を考える.
- 並列化できても性能が出ない場合がある.
  - 原因の追及は,やや難しい.
    - コンパイラの詳細メッセージ
  - 原因がわかれば,比較的容易に解決できる場合も結構多い.

まとめ#3

- 不規則な構造を持つ問題は,HPFでの並列化が難しいことが多い.
  - 一般に,MPIでも並列化は難しいが…
- 圧縮された疎行列形式を用いている場合などは,GEN_BLOCKで対応できる.
- FEMのように不規則構造が静的である場合は,HPFのNEC独自拡張であるHALOを用いれば,対応できる.