HPFによる
実用コードの並列化

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

- 3次元流体コード
- 2次元静電粒子コード
- SPEC OMP2001
  - SWIM
  - MGRID
  - APSI
  - APPLU

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

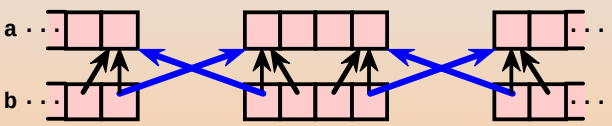
- INDEPENDENT指示文を並列実行できるDOループに追加し，並列実行を明示する．

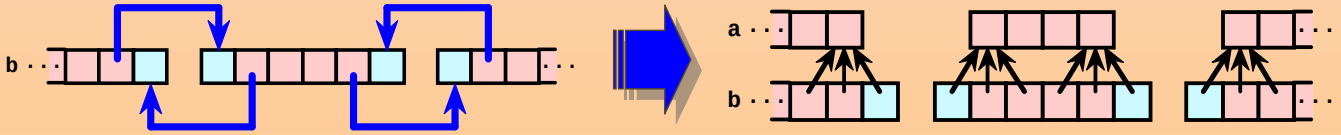
```
!HPF$ INDEPENDENT
    do iz = 1, lz-1
      do 10 iy = 1, ly
        do 10 ix = 1, lx
          wr(ix,iy,iz) = sr(ix,iy,iz) + sr(ix,iy,iz+1)
          …
  10   continue
      end do
    …
!HPF$ INDEPENDENT, REDUCTION(max:wram)
    do iz = 1, lz
      do 20 iy = 1, ly
        do 20 ix = 1, lx
   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

袖領域の定義とその通信

- 分散配列に袖領域を定義して，通信する．
  - 効率のよいブロック通信
  - メモリの連続アクセス

```
      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
```





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
```

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
```

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
```

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.
  - 自動では, 最適化されなかった.

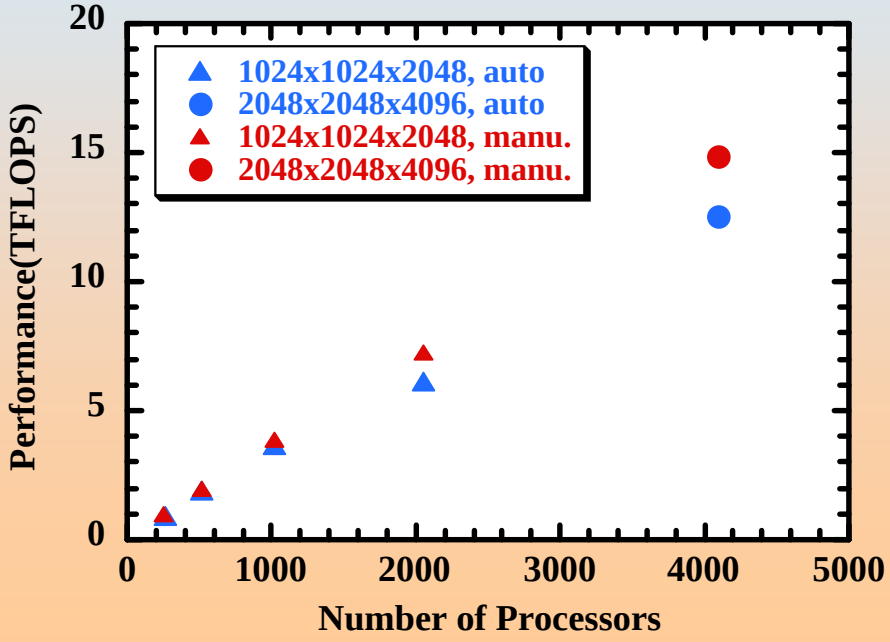
```
      do j = 2, maxy-1
        do i = 2, maxx-1
          isign = 1
          if( un(i,j) .lt. 0.0d0 ) isign = -1
          jsign = 1
          if( vn(i,j) .lt. 0.0d0 ) jsign = -1
          im1 = i - isign
          jm1 = j - jsign
!HPF$ ON HOME(un(i,j)), LOCAL BEGIN
          a8 = fs(i,j) - fs(im1,j)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

ESにおける手動最適化の効果

- LOCAL, REFLECTによる最適化は，コンパイラにより自動的に行われていた．
- 自動並列化では，通信スケジュールは，サブルーチンの呼び出し毎に少なくとも一回，実行情報を元に作成される．
- 手動最適化では，コンパイラが通信スケジュールを静的に決定できるため，全体の実行を通じて一回作成するだけである．

ESにおける自動／手動の性能比較

- アムダールの法則により，並列台数が増えると差が広がる．



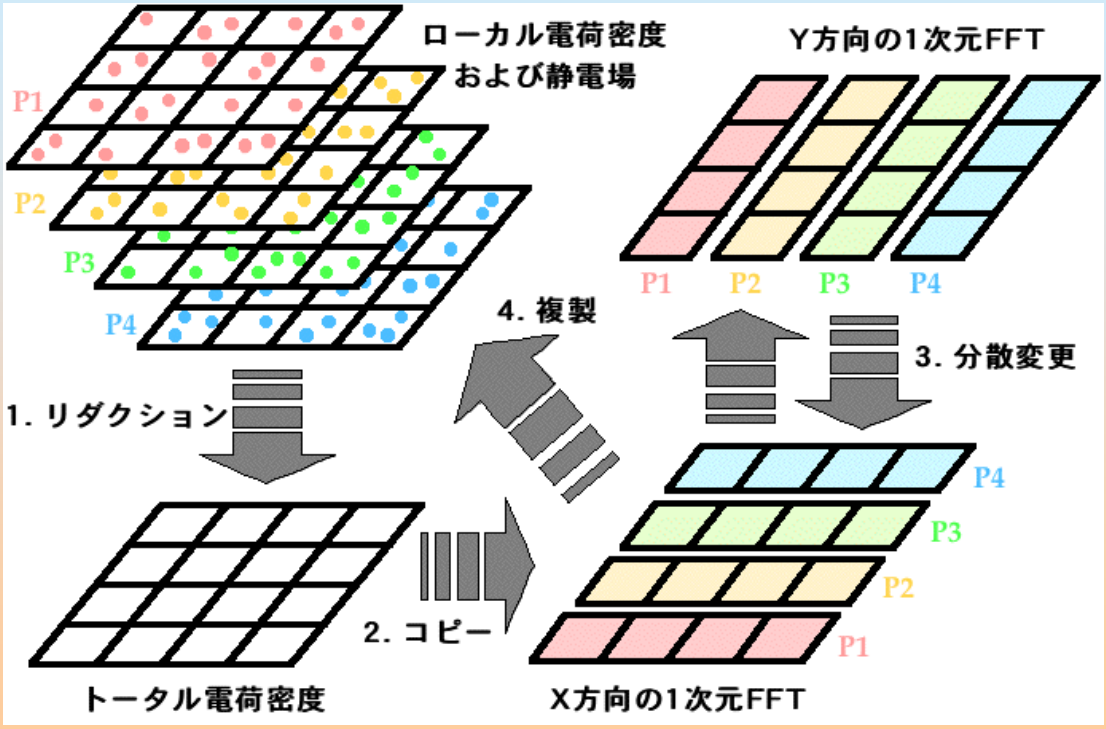
2次元静電粒子コード

- 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．
- 場データの分散は，必要に応じて動的に変更する．

場データの動的再分散



電荷密度計算のHPF化

ベクトル化

```
      real xe(no), rho(lx)
      do i = 1, no
        ixe = xe(i)
        dxe = xe(i) - ixe
        rho(ixe) = rho(ixe)
     &           + (1.0-dxe)
      end do
```

```
      real xe(no), rho(lx), rhotmp(lvec,lx)
      do iv = 1, lvec
        do i = 1, no, lvec
          ixe = xe(i+iv-1)
          dxe = xe(i+iv-1) - ixe
          rhotmp(iv,ixe) = rhotmp(iv,ixe)
     &                   + (1.0-dxe)
        end do
      end do
c.... reduction rhotmp to rho
      do iv = 1, lvec
        do i = 1, lx
          rho(i) = rho(i) + rhotmp(iv,i)
        end do
      end do
```

HPF化

```
      real xe(no), rho(lx), rhotmp(lvec,lx)
!HPF$ DISTRIBUTE xe(CYCLIC),rhotmp(BLOCK,*)
!HPF$ INDEPENDENT
      do iv = 1, lvec
!HPF$ ON HOME(rhotmp(iv,:)), LOCAL BEGIN
        do i = 1, no, lvec
          ixe = xe(i+iv-1)
          dxe = xe(i+iv-1) - ixe
          rhotmp(iv,ixe) = rhotmp(iv,ixe)
     &                   + (1.0-dxe)
        end do
!HPF$ ENDON
      end do
```

ソースは無修正だが
並列性能がでない！

一時配列によるＨＰＦ化＃１

- ベクトル化とは別に，HPFによる並列化を考える．
  - ベクトル化と共存するときは，更に工夫が必要である．

```
      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
```

SXにおける並列化とベクトル化

```
      dimension xe(nm,np)
      dimension w(nx,np)
      dimension rho(nx)
!HPF$ DISTRIBUTE (*,BLOCK):: x,w
!HPF$ INDEPENDENT
      do ip = 1,np
       do m = 0, 1
!CDIR LISTVEC
        do j = 1, nm
         ix = x(j,ip)
         dx = x(j,ip) - ix
         if(m.eq.0) s1 = 1.0 - dx
         if(m.eq.1) s1 = dx
         ix = ix + m
         w(ix,ip) = w(ix,ip) + s1
        end do
       end do
      end do
!HPF$ INDEPENDENT,REDUCTION(+:rho)
      do ip = 1,np
       do I = 1, nx
        rho(i) = rho(i) + w(i,ip)
       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
```

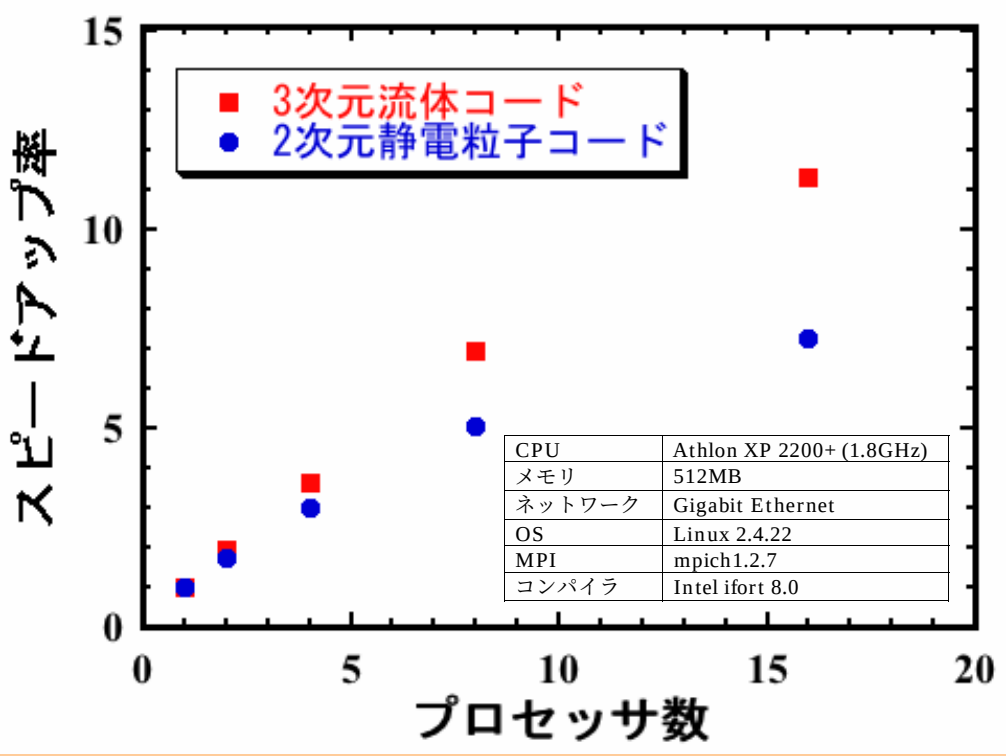
2次元FFTのHPF化#3

```
      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

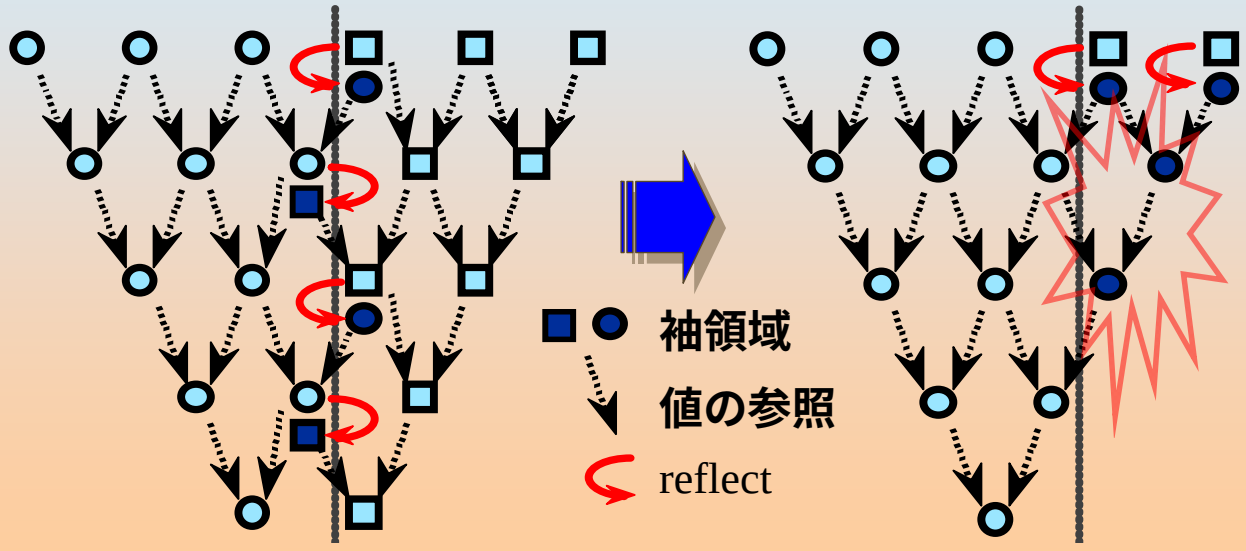


更なる最適化

- EXT_HOME節
  - 袖領域までプロセッサの計算境界を拡張する.
  - 重複計算 vs. 通信コスト
- 非同期通信
  - 通信と計算を重複実行する.
  - 通信時間の隠蔽
- 手動による再分散
  - サブルーチン呼び出し時の自動再マップによる無駄な通信の抑制

EXT_HOME指示文による通信の最適化

- プロセッサ境界を拡張することで袖領域の計算を重複して行い, REFLECTの回数を一回に減らすことができる.



通信の初期化が通信速度に比べて遅いネットワークでは有効である. ただし, 重複計算のための新たなオーバーヘッドは発生する.

非同期通信

- 非同期通信を使うと，データの転送と計算を同時に実行できる．
  - DOループ実行の前に6変数のREFLECTが必要である．
  - DOループを二つに分割し，非同期転送を使う．

```
!HPF$ REFLECT sr, sm
!HPF$ ASYNC(id) BEGIN
!HPF$ REFLECT sn, sl, se, sp
!HPF$ END ASYNC
!HPF$ INDEPENDENT
    do iz = 1,lz-1
!HPF$ ON HOME( wtmp(:,:,iz)), LOCAL BEGIN
     (sr, smだけを使った計算)
!HPF$ END ON
    end do
!HPF$ ASYNCWAIT(id)
!HPF$ INDEPENDENT
    do iz = 1,lz-1
!HPF$ ON HOME( wtmp(:,:,iz)), LOCAL BEGIN
     (すべての変数を使った計算)
!HPF$ END ON
    end do
```

再分散の最適化

- 手動で再分散を行うと4回が2回になる．
  - REDISTRIBUTEが使えるとソース無修正．

```
!HPF$ DISTRIBUTE rho(*,BLOCK)
!HPF$ DISTRIBUTE ck(*,BLOCK)
c.... forward FFT
     call fftfx2 ( lx, ly, rho )
c--- remap (*,BLOCK) -> (BLOCK,*)
     call fftfy2 ( lx, ly, rho )
c--- remap (BLOCK,*) -> (*,BLOCK)
c.... form factor
     rho = ck * rho
c.... backward FFT
c--- remap (*,BLOCK) -> (BLOCK,*)
     call fftby2 ( lx, ly, rho )
c--- remap (BLOCK,*) -> (*,BLOCK)
     call fftbx2 ( lx, ly, rho )
```

```
!HPF$ DISTRIBUTE rho(*,BLOCK)
!HPF$ DISTRIBUTE (BLOCK,*) ::rho2,ck
c.... forward FFT
     call fftfx2 ( lx, ly, rho )
c--- remap (*,BLOCK) -> (BLOCK,*)
     rho2 = rho
     call fftfy2 ( lx, ly, rho2 )
c.... form factor
     rho2 = ck* rho2
c.... backward FFT
     call fftby2 ( lx, ly, rho2 )
c--- remap (BLOCK,*) -> (*,BLOCK)
     rho = rho2
     call fftbx2 ( lx, ly, rho )
```

SPEC OMP2001

- SPECによって開発されたOpenMPによる共有メモリ型並列計算機システムの性能評価を行うベンチマークプログラムである．
  - OpenMPは，HPFと同様に指示文ベースであるので，両者のプログラミング適応性や並列性能を比較することは，興味深い．
  - 単なるベンチマークコードではなく，実用的なコードである．
  - FORTRANコード8本とCコード2本からなる．

OpenMPの並列化

- 基本ルールとしてOpenMPの並列領域をHPFで並列化することとする．
  - PARALLEL,END PARALLEL指示文
  - DO指示文

```
      ...
!$OMP PARALLEL
!$OMP DO
      do j = 1, n
        do i = 1, m
          u(i,j) = p(i,j)
          v(i,j) = p(i,j+1)
        end do
      end do
!$OMP END DO
!$OMP END PARALLEL
      ...
```

HPFによる並列化＃1

- データ分散と処理分割
  - PROCESSORS指示文
  - DISTRIBUTE指示文
  - INDEPENDENT指示文

```
    ...
!HPF$ PROCESSORS proc(NPROC)
!HPF$ DISTRIBUTE (*,BLOCK) ONTO proc :: u,v,p
    ...
!HPF$ INDEPENDENT
    do j = 1, n
     do i = 1, m
      u(i,j) = p(i,j)
      v(i,j) = p(i,j+1)
     end do
    end do
    ...
```

HPFによる並列化＃2

- 明示的な指示による通信効率の向上
  - SHADOW, REFLECT指示文
  - ON-HOME-LOCAL指示文

```
    ...
!HPF$ SHADOW (0,0:1) ::p
    ...
!HPF$ REFLECT p
!HPF$ INDEPENDENT
    do j = 1, n
!HPF$ ON HOME(u(i,j)), LOCAL BEGIN
     do i = 1, m
      u(i,j) = p(i,j)
      v(i,j) = p(i,j+1)
     end do
!HPF$ END ON
    end do
    ...
```

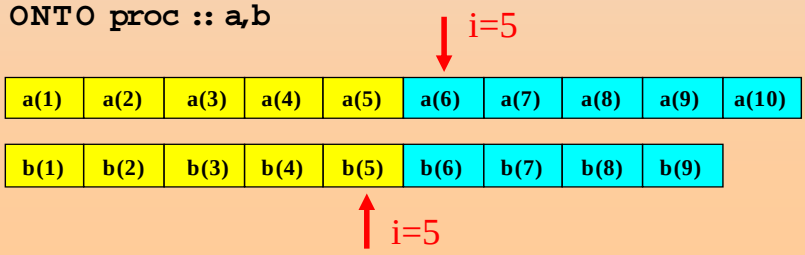
HPF/OpenMPの実行環境

- NEC SX-5/128M8
  - 大阪大学サイバーメディアセンター
  - ノード当り16プロセッサ
  - ノード当り160GFLOPS, 128GB
- OpenMP
  - FORTRAN90/SX Version 2.0 for SX-5 Rev.280
- HPF
  - HPF/SX V2 Rev.1.9.2
  - -Mnoentry
    - エラー処理コードの生成を抑制する.

SWIMの並列化

- 浅水方程式による海水のシミュレーションプログラムである.
- 同一のDOループ内で結果を格納する配列の添字が異なる場合に並列性能が悪い.
  - Owner Computes Ruleとの矛盾

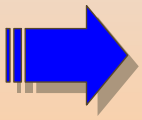
```
      dimension a(10),b(9)
!HPF$ PROCESSORS proc(2)
!HPF$ DISTRIBUTE (BLOCK) ONTO proc :: a,b
      do i = 1, 9
       a(i+1) = ...
       b(i)   = ...
      end do
```



並列性能低下の原因

- OCRと矛盾する場合は，一時変数を動的に確保して計算後，本来の変数にコピーする．
  - 並列化はできるが，大きなオーバーヘッド

```
do i = 1, 9
 a(i+1) = ...
 b(i)   = ...
end do
```

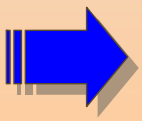


```
allocate(atemp)
do i = 1, 9
 atemp(i) = ...
 b(i)     = ...
end do
do i = 1, 9
 a(i+1) = atemp(i)
end do
deallocate(atemp)
```

問題の解決方法＃１

- 添字が異なる配列の処理を別々のDOループに分ける．
  - ソースを修正する必要があるが，容易に並列性能を改善できる．
  - ベクトル処理の効率が低下する場合がある．

```
do i = 1, 9
 a(i+1) = ...
 b(i)   = ...
end do
```

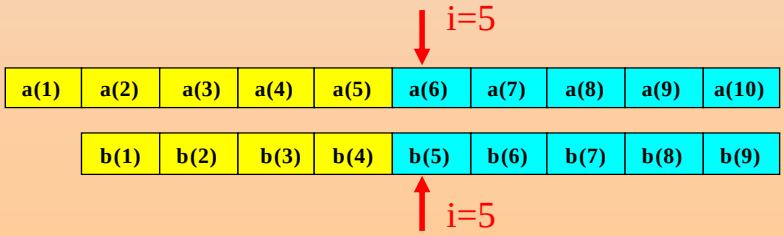


```
do i = 1, 9
 a(i+1) = ...
 end do
 do i = 1, 9
 b(i)   = ...
end do
```

問題の解決方法＃2

- OCRとDOループ処理分担が矛盾しないよう配列の分散をALIGN指示文で調整する．
  - 指示文のみで並列性能を改善できる．
  - ただし，異なるDOループで最適なALIGNが異なる場合は，対応できない．

```
!HPF$ ALIGN b(i) WITH a(i+1)
    ...
    do i = 1, 9
     a(i+1) = ...
     b(i)   = ...
    end do
```



両方の方法の併用

- ベクトル処理の効率を考慮する．

×

```
    ...
!HPF$ INDEPENDENT
    do i = 1, 9
     a(i+1) = p(i)*2
    end do
!HPF$ INDEPENDENT
    do i = 1, 9
     b(i)   = -p(i)/2
    end do
    ...
!HPF$ INDEPENDENT
    do i = 1, 9
     a(i) = 0
     b(i) = 1
    end do
```

```
do i = 1, 9
 a(i+1) = p(i)*2
 b(i)   = -p(i)/2
end do
...
do i = 1, 9
 a(i) = 0
 b(i) = 1
end do
```

○

```
!HPF$ ALIGN b(i) WITH a(i+1)
    ...
!HPF$ INDEPENDENT
    do i = 1, 9
     a(i+1) = p(i)*2
     b(i)   = -p(i)/2
    end do
    ...
!HPF$ INDEPENDENT
    do i = 1, 9
     a(i) = 0
    end do
!HPF$ INDEPENDENT
    do i = 1, 9
     b(i) = 1
    end do
```

SWIMの性能評価

- OpenMPと同程度の並列性能を得ることができた.
- ただし, OpenMPプログラムはNEC SX-5の1ノード内でしか実行できないが, HPFは複数ノードでも実行できる！

SWIM	Execution Time [s]			Speedup Ratio		
#proc	before	after	OpenMP	before	after	OpenMP
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

MGRIDの並列化

- マルチグリッドにより3次元のポテンシャル場を計算をするプログラムである.
- 主プログラムと副プログラムで引数の次元数が異なるため, そのままでは並列化することができない.

主プログラム

```
dimension big(NALL),is(k),il(k)
...
do i = 1, k
  call sub( big(is(i)), il(i) )
end do
...
stop
end
```

副プログラム

```
subroutine sub( a, m )
dimension a(m, m, m)
...
return
end
```

一時配列を用いた問題解決

- 動的に配列を確保／分散し，非分散の仮引数から値を複写して，並列化する．

```
      subroutine sub( aa, m )
      dimension aa(m*m*m)
!HPF$ PROCESSORS proc(NPROC)
!HPF$ DISTRIBUTE (*,*,BLOCK) ONTO proc :: a
      allocatable a(:,:,:)
      allocate(a(m,m,m))
      (copying a ← aa)
      ...
      ...
      (copying aa ← a)
      deallocate(a)
      return
      end
```

MGRIDの性能評価＃1

- サブルーチンが呼出される度に動的な配列の確保／解放およびデータ複写のオーバーヘッドのため，OpenMP程の並列性能が得られなかった．
  - そのオーバーヘッドを測定すると約80秒

MGRID	Exec. Time [s]		Speedup Ratio	
#proc	HPF	OpenMP	HPF	OpenMP
1	326.0	230.2	1.00	1.00
2	191.4	118.6	1.70	1.94
3	141.3	81.1	2.31	2.84
4	117.2	63.1	2.78	3.65
8	78.4	36.0	4.16	6.39

広域並列化による性能改善

- 基本ルールから離れてサブルーチンを並列呼出することで広域的な並列化ができる.

```
      ...
      do i = 1, ndata
        call zero3 0( u,v,n,n,n )
        call norm2u3( r,n,n,n,rnm2(i),rnmu(i) )
        ...
      end do
```

```
      ...
!HPF$ INDEPENDENT, NEW(u,v,r,...
      do i = 1, ndata
!HPF$ ON HOME(rnm2(i))
        call puresub( u,v,r,n,n,n,rnm2(i),rnmu(i),...)
      end do
        ...
      pure extrinsic('FORTRAN','LOCAL') subroutine
     &        puresub( u,v,r,n,n,n,rnm2,rnmu,...)
        ...
      call zero3 0( u,v,n,n,n )
      call norm2u3( r,n,n,n,rnm2,rnmu,...)
      ...
```

MGRIDの性能評価＃2

- 広域並列化では, OpenMPに近い並列性能が得られた.
  - 配列はNEW宣言するため各プロセッサ毎に全体を確保するので多くのメモリが必要である.

MGRID	Execution Time [s]			Speedup Ratio		
#proc	local	global	OpenMP	local	global	OpenMP
1	326.0	275.2	230.2	1.00	1.00	1.00
2	191.4	149.7	118.6	1.70	1.84	1.94
3	141.3	105.5	81.1	2.31	2.61	2.84
4	117.2	76.0	63.1	2.78	3.62	3.65
8	78.4	44.2	36.0	4.16	6.23	6.39

APSIの並列化＃1

- 3次元流体方程式を解き，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．
- 実引数と仮引数で次元数が異なる．
  - 変数ごとに別々に宣言するように修正した．

```
allocatable :: work(:)
allocate(work(ntotal))
・・・
lc=1
lstepc = 1 + nx*ny*nz
・・・
call run(nx,ny,nz,work(lc),work(lstepc))
・・・
subroutine run(nx,ny,nz,c,stepc)
dimension c(*),stepc(*)
・・・
call dctdx(nx,ny,nz,c)
・・・
```

```
allocatable :: c(:,:,:),stepc(:,:,:)
・・・
allocate(c(nx,ny,nz))
allocate(stepc(nx,ny,nz))
・・・
call run(nx,ny,nz,c,stepc)
・・・
subroutine run(nx,ny,nz,c stepc)
dimension c(nx,ny,nz),stepc(nx,ny,nz)
・・・
call dctdx(nx,ny,nz,c)
・・・
```

APSIの並列化＃2

- 並列処理を行うべき配列の次元が，場所によって異なっている．
  - サブルーチン呼出し時の再マッピングで分散次元を変更した．

```
      ・・・
!HPF$ DISTRIBUTE (*,*,BLOCK) ONTO proc :: c
      ・・・
      ・・・（並列処理は3次元目）
      ・・・
      call advc(nx,ny,nz,c)
      ・・・
c-------------------------------------
      subroutine advc(nx,ny,nz,c)
      ・・・
!HPF$ DISTRIBUTE (*,BLOCK,*) ONTO proc :: c
      ・・・
      ・・・（並列処理は2次元目）
      ・・・
```

APSIの並列化＃3

- 副プログラムの並列呼び出し時に，分散配列の一部を渡す．
  - 部分配列を用いるように修正した．

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

```
    dimension c(nx,ny,nz)
    ...
!HPF$ DISTRIBUTE (*,*,BLOCK) ONTO proc :: c
    interface
      pure extrinsic(fortran_local)
     &  subroutine sub(nx,ny,c)
    ...
    end interface
    ...
!HPF$ INDEPENDENT
    do i = 1, nz
      call sub(nx,ny,c(:,:,i))
    end do
    ...
    subroutine sub(nx,ny,c)
    dimension c(nx,ny)
    ...
```

APSIの性能評価

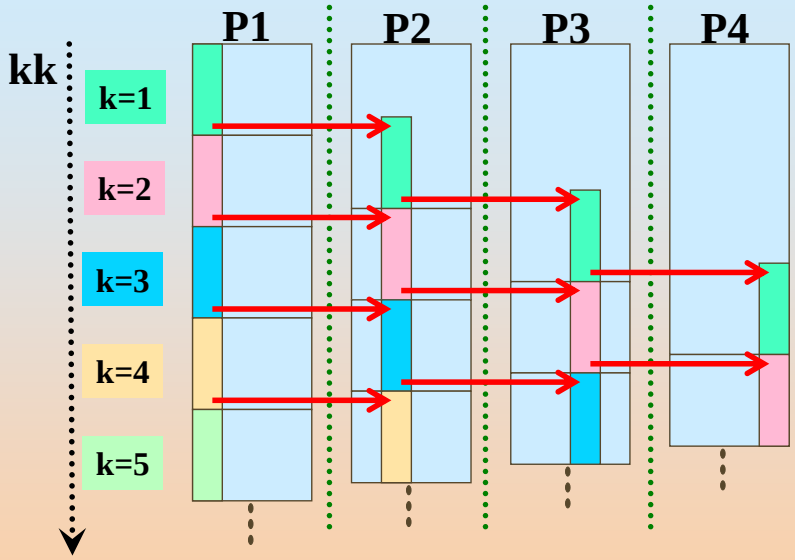
- OpenMPに近い並列性能を得ることができた．

APSI	Exec. Time [s]		Speedup Ratio	
#proc	HPF	OpenMP	HPF	OpenMP
1	1020.5	928.4	1.00	1.00
2	541.6	477.6	1.88	1.94
3	358.9	311.6	2.84	2.98
4	267.4	235.4	3.82	3.94
8	171.2	136.6	5.96	6.80

APPLUの並列化

- 非線形ブロックSORを用いている．
- 通常のSORは，回帰参照のためベクトル化も並列化もできない．
  - HPF指示文の挿入だけでは並列化できない．
- そこで，擬ハイパープレーン法を用いる．
  - ソースコードの修正が少なくて済む．
  - 残念ながら，通信はMPIのサブルーチンを直接呼び出すことで実現する．

擬ハイパープレーン法



HPF_LOCALサブルーチンから呼び出されるMPIサブルーチンによって実現されるシフト型通信

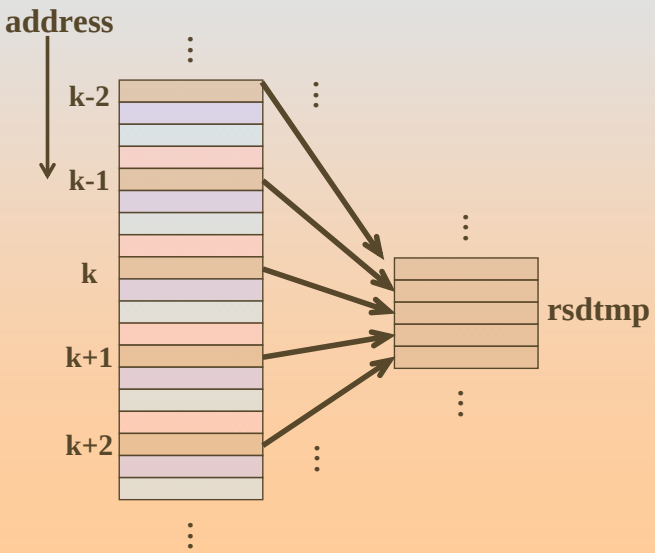
```
do k = 1, nz
  ...
end do
```

```
do kk = 1, nz+nproc-1
  k = kk - myid
  if (k .lt. 1 .or. k .gt. nz) cycle
  ...
end do
```

HPF_LOCALサブルーチンの並列呼出におけるオーバーヘッド

- 最終次元以外で分散された配列を引数にすると内部的に一時配列にコピーするコードが生成される．

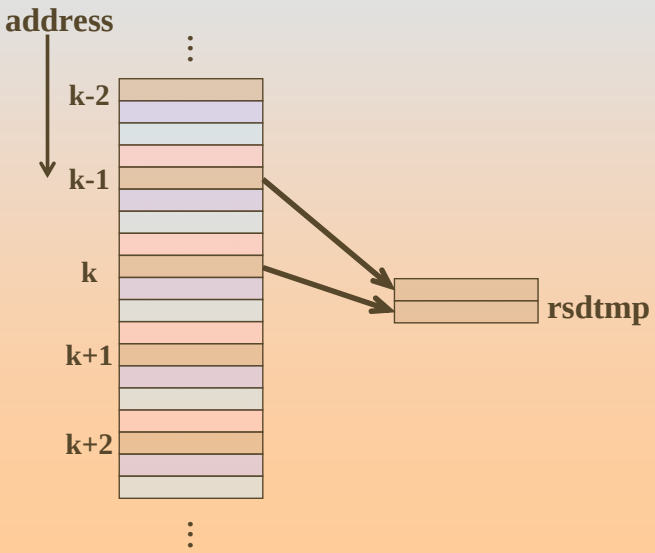
```
!HPF$ DISTRIBUTE (*,*,BLOCK,*) :: rsd
      ...
      do kk = 1, nz+nporc-1
        k = kk - myid
        if (k .lt. 1 .or. k .gt. nz) cycle
        call jalcd( ...)
        call blts( ..., k, rsd, ...)
      end do
      ...
      extrinsic('HPF','LOCAL')
     &   subroutine blts( ...,k,rsd, ...)
      dimension rsd(:,:,:,:)
      call MPI_COMM_SIZE(COMM,nproc,ierr)
      call MPI_COMM_RANK(COMM,myid,ierr)
      jmax = ... myid ... nproc
      do j = 1, jmax
       do i = 1, nx
        do m = 1, 5
          ... = rsd(m,i,j,k) + rsd(m,i,j,k-1)
        end do
       end do
      end do
      ...
```



部分配列を引数にすることによるオーバーヘッド抑制

- kとk-1しか添字として使われていないので，実引数として部分配列を使う．

```
!HPF$ DISTRIBUTE (*,*,BLOCK,*) :: rsd
      ...
      do kk = 1, nz+nporc-1
        k = kk - myid
        if (k .lt. 1 .or. k .gt. nz) cycle
        call jalcd( ...)
        call blts( ..., k, rsd(:,:,:,k-1:k), ...)
      end do
      ...
      extrinsic('HPF','LOCAL')
     &   subroutine blts( ...,k,rsd, ...)
      dimension rsd(:,:,:,2)
      call MPI_COMM_SIZE(COMM,nproc,ierr)
      call MPI_COMM_RANK(COMM,myid,ierr)
      jmax = ... myid ... nproc
      do j = 1, jmax
       do i = 1, nx
        do m = 1, 5
          ... = rsd(m,i,j,2) + rsd(m,i,j,1)
        end do
       end do
      end do
      ...
```



APPLUの性能評価

- 擬ハイパープレーン法が非効率なため，そこそこの並列性能しか得られなかった．
  - ループ当り5*224*224*2データについて4*218回一時配列へコピーされる！

APPLU	Exec. Time [s]		Speedup Ratio	
#proc	HPF	OpenMP	HPF	OpenMP
1	3472.6	2526.1	1.00	1.00
2	2139.4	1279.9	1.62	1.97
4	1170.1	663.2	2.97	3.81
8	604.7	351.9	5.74	7.18

並列化のまとめ＃１

- SWIM
  - ソースを修正して，DOループの処理分担とOCR間の矛盾を解消する必要があった．
- MGRID
  - 実引数と仮引数で配列の次元数が異なるため，動的な配列確保とデータ複写を用いた．
  - 基本ルールから離れると，サブルーチンの並列呼出による広域並列化が可能であった．

並列化のまとめ＃2

- APSI
  - 引数の次元数が異なる問題を解決するため，個々に配列を宣言した．
  - サブルーチンを並列呼出するため，分散配列の部分配列を実引数として使った．
- APPLU
  - MPIサブルーチンを直接呼び出して通信することで擬ハイパープレーン法により並列化した．
  - サブルーチンの実引数に部分配列を使うことで性能を改善した．

性能評価のまとめ

- SWIM
  - OpenMPと同等の並列性能が得られた．
- MGRID
  - 動的な配列の確保およびデータ複写のオーバーヘッドが大きく，良い並列性能は得られなかった．
  - 広域並列化では，良い並列性能が得られた．
- APSI
  - OpenMPに近い並列性能が得られた．
- APPLU
  - そこそこの並列性能しか得られなかった．

指示文および追加･修正行数

プログラム	オリジナル	OpenMP	HPF	追加･修正
SWIM(改善前)	約300	10	41	0
SWIM(改善後)	約300	10	77	20
MGRID	約400	84	127	208
MGRID(広域並列)	約600	84	8	70
APSI	約4300	144	166	約400
APPLU	約3800	93	153	約500

注：HPF指示文の行数は，includeを展開して計数しているので，実際に作業した行数は，もっと少ない．

HPF化のまとめ

- 規則的な構造の問題なら，HPFでも十分に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，比較的簡単に解決できる場合が多い．
    - HPFPCに御相談下さい！