HPF/ES
プログラミングテクニック
― 中級編 ―

地球シミュレータセンター
村井 均

Table of Contents

- Fortran 90 プログラミング
- リダクション(集計)演算
- 性能測定
- 外来手続き
  - Fortranライブラリ
  - MPI手続き
- 不規則問題
- HPFの限界

Fortran 90 プログラミング(1)

- 大域変数

  COMMONよりはモジュール、モジュールよりは引数渡しを使う。

	最適化	メモリ効率	機能
COMMON	×	△	×
モジュール	△	△	○
引数渡し	○	○	△

Fortran 90 プログラミング(2)

- 実際に使用するサイズの配列を割り付ける。

  サイズが実行時に決まる場合は、割り付け配列や自動配列の機能を使う。

割付け配列の例

```
      real a(1000)
!HPF$ distribute a(block)
      read(*,*) n
      do i=1, n
       a(i) = ...
      end do
```

```
      real allocatabe :: a(:)
!HPF$ distribute a(block)
      read(*,*) n
      allocate (a(n))
      do i=1, n
       a(i) = ...
      end do
```

メモリの無駄や負荷の不均等を引き起こす。

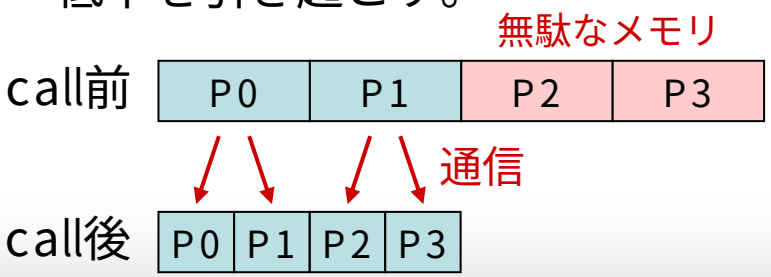
Fortran 90 プログラミング(3)

自動配列の例

```
      real a(1000)
!HPF$ distribute a(block)
      read(*,*) n
      call sub(a,n)
      ...
      subroutine sub(a,n)
      real a(n)
!HPF$ distribute a(block)
```

```
      read(*,*) n
      call sub(n)
      ...
      subroutine sub(n)
      real a(n)
!HPF$ distribute a(block)
```

メモリの無駄や通信による性能低下を引き起こす。



Fortran 90 プログラミング(4)

- 実引数と仮引数の形状は一致させる。

  Fortran 90では必ずしも一致しなくともよいが、HPFでは一致する必要がある(実行時エラーとなる)。

  A)実引数: 配列要素 → 仮引数: 配列

  ➡ 実引数を部分配列に。

```
      REAL A(10,10)
!HPF$ DISTRIBUTE A(*,BLOCK)
      CALL SUB(A(1,I))
      ...
      SUBROUTINE SUB(B)
      REAL B(10)
```

```
      REAL A(10,10)
!HPF$ DISTRIBUTE A(*,BLOCK)
      CALL SUB(A(:,I))
      ...
      SUBROUTINE SUB(B)
      REAL B(10)
```

Fortran 90 プログラミング(5)

B) 実引数: 一次元配列 → 仮引数: 二次元配列

➡ テンポラリにコピーしてから渡す。

✕

```
      REAL A(100)
!HPF$ DISTRIBUTE (BLOCK) :: A
      CALL SUB(A)
      ...
      SUBROUTINE SUB(B)
!HPF$ DISTRIBUTE (*,BLOCK) :: B
      REAL B(10,10)
```

○

```
      REAL A(100), T(10,10)
!HPF$ DISTRIBUTE (BLOCK) :: A
!HPF$ DISTRIBUTE (*,BLOCK) :: T
      AをTにコピー
      CALL SUB(T)
      TをAにコピー
      ...
      SUBROUTINE SUB(B)
!HPF$ DISTRIBUTE (*,BLOCK) :: B
      REAL B(10,10)
```

※ 非分散配列であれば、sequence属性を指定してこれらの渡し方もできる。

Fortran 90 プログラミング(6)

- 擬寸法(大きさ引継ぎ)配列は使わない。
  - 形状引継ぎ配列がほぼ等価な機能を実現。
  - 非分散配列であれば、sequence属性を指定して擬寸法配列として使える。

```
!HPF$ distribute (block) :: a
      call sub(a,b)
      ...
      subroutine sub(a,b)
      real a(*), b(*)
!HPF$ distribute (block) :: a
```

```
!HPF$ distribute (block) :: a
!HPF$ sequence :: b
      call sub(a,b)
      ...
      subroutine sub(a,b)
      real a(:), b(*)
!HPF$ distribute (block) :: a
!HPF$ sequence :: b
```

ループ構造の得失(1)

- DO文
  - 可能であれば、自動的に並列化する。
  - ループボディ全体を対象に、通信生成および各種最適化を適用する。
  - ループのネスティングは変更しない。
- 配列構文
  - 等価なFORALL文に変換する。
- FORALL文
  - 等価なDO文に変換した後、並列化する。このとき、左辺配列の一次元目から順に最内側ループでアクセスされるように、ループを構成する。

ユーザが、最適なネスティングでコーディングしておく必要がある。

ループ構造の得失(2)

- ネスティングを考慮※した上で、DO文を使用するのがベスト。
- 単純なゼロクリアやメモリコピー程度であれば、配列構文でもロスはない。
- 現状では、FORALL文を使うメリットはあまりない。

※ 最内側ループで一次元目を、最外側ループで最終次元ないしは分散次元をアクセスする。

リダクション(集計)演算

- 最新版では、ほとんどのリダクション演算を自動的に並列化できる。

総和

```
do i=1, N
  s = s + a(i)
end do
```

最大値

```
do i=1, N
  amax = max(a(i),amax)
end do
```

- 位置変数つきの最大値・最小値は、指示文の指定が必要。

```
!HPF$ independent,
!HPF$+ reduction(FIRSTMAX:amax/iloc/)
      do i=1, N
        if (a(i) > mx) then
         amax = a(i)
         iloc = i
        end if
      end do
```

性能測定(1)

主に以下の4つの方法がある。

- HPFPROF
- ftrace
- MPIPROGINF
- 計時関数

※ UNIX標準のprofコマンドも利用可能。

	情報量	測定の単位	操作性
HPFPROF	×	○	○
ftrace	△	△	△
MPIPROGINF	○	×	◎

性能測定(2) HPFPROF

ループ単位の性能情報をGUIで表示(CUIでも利用可能)

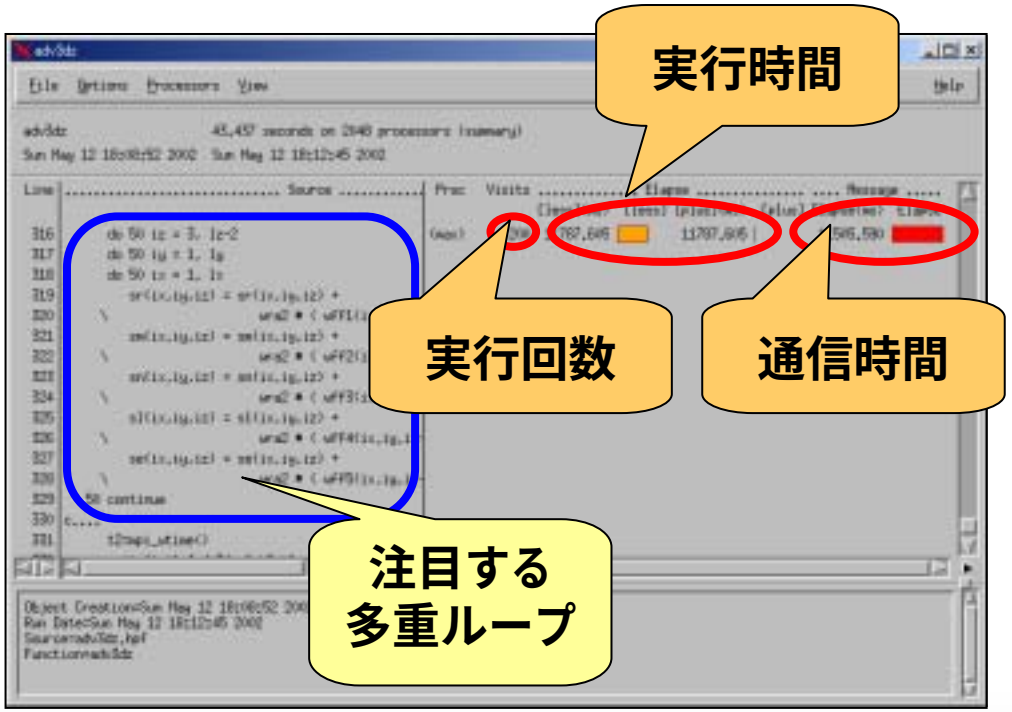
```
% hpf -Mprof=loop foo.hpf
```

↓

実行（**hpfprof.out** 出力）

↓

```
% hpfprof &
```



○直感的に情報を把握できる。
○ループ単位の情報を得られる。
×オーバヘッドはやや大きい。

性能測定(3) MPIPROGINF

ハードウェア機能により収集した情報を、
実行終了後に表示

```
% setenv MPIPROGINF ALL_DETAIL
```

↓

実行（標準エラーに表示)

○コンパイル時に特別な処理は不要。
○オーバヘッドはない。
×プログラム全体の情報のみ。

```
Real   Time (sec)              :   101.612306
User   Time (sec)              :   101.506830
System Time (sec)              :     0.094371
Vector Time (sec)              :    96.497723
Instruction Count              :  11823252381
Vector Instruction Count       :   3385739196
Vector Element Count           : 811052249951
FLOP Count                     : 319016327628
MOPS                           :  8073.247504
MFLOPS                         :  3142.806520
Average Vector Length          :   239.549535
Vector Operation Ratio (%)     :    98.970394
Memory size used (MB)          :  1874.048325
MIPS                           :   116.477407
Instruction Cache miss (sec):     0.511975
Operand     Cache miss (sec):     0.339069
Bank     Conflict Time (sec):     0.410926
```

性能測定(4) ftrace

- FORTRAN90/SXの簡易性能解析機能
- 手続き毎の性能情報を表示

```
% hpf -ftrace *.hpf
```

↓

実行（プロセス毎にftrace.**out** 出力）

↓

```
% ftrace
```

○手続き毎の詳細な情報が得られる。
×通信など、並列実行に関する情報は得られない。
×操作がやや複雑。

```
*-------------------------*
  FLOW TRACE ANALYSIS LIST
*-------------------------*

Execution : Tue Jun  3 13:15:55 2003
Total CPU : 0:00'08"958


PROG.UNIT  FREQUENCY  EXCLUSIVE       AVER.TIME   MOPS      BANK
                      TIME[sec](  % )    [msec]             CONF

adv3dx          300     3.806( 42.5)    12.687 5788.3    0.0012
adv3dz          200     2.523( 28.2)    12.613 5727.0    0.0013
adv3dy          200     2.296( 25.6)    11.482 6357.0    0.0012
cmp3dp          700     0.182(  2.0)     0.259 3935.2    0.0000
cmpram          100     0.136(  1.5)     1.357 4882.4    0.0000
unnamed$main
                  1     0.016(  0.2)    15.871 2272.7    0.0001
init              1     0.000(  0.0)     0.373 4019.8    0.0000
pghpf$static$init
                  1     0.000(  0.0)     0.011   54.4    0.0000
-------------------------------------------------------  ------
total          1503     8.959(100.0)     5.961 5859.2    0.0039
```

※ プローブ関数を埋め込むことで、ライン単位の情報を得ることも可能。

性能測定(5) 計時関数

以下の計時関数(手続き)が利用可能

- **MPI_Wtime**
- **SYSTEM_CLOCK**
- **etime**

○任意のポイントで情報を得られる。
×0番プロセスの情報のみ。
×並列ループ内には書けない。

```
      double precision t1,
     +       t2, MPI_Wtime
      ...
      t1 = MPI_Wtime()
!HPF$ independent
      do i=1, N
       ...
      end do
      t2 = MPI_Wtime()
      write(*,*) t2-t1
```

※ **MPI_Wtime** を使う場合は、明示的な型宣言「**double precision MPI_Wtime**」が必要。

HPF外来機能(1)

HPF外来機能を使って、ASL等のFortranライブラリやMPIライブラリを呼ぶことができる。

➡ 性能ボトルネック部分を、より高速なFortranライブラリやMPI手続きで置き換える。

外来機能: 他言語で書かれた手続きや、異なる実行モデルに基づく手続きを呼ぶための機能

HPF外来機能(2) Fortranライブラリ

インタフェースブロック内で、Fortranライブラリを「FORTRAN_LOCAL手続き」として宣言する。

引数の分散次元には「:」を指定して、形状引継ぎ配列とする。

分散指定は、呼び出し前の状態を示す(必要に応じ、再分散が起こり得る)。

```
      interface
        extrinsic(FORTRAN_LOCAL)
     +  subroutine sub(a)
        real a(:)
!HPF$   distribute (block) :: a
        end subroutine
      end interface
```

HPF外来機能(3) Fortranライブラリ

注意点

A) 引数として、ローカル配列のサイズを渡す。

B) 配列引数は、最終次元を分割する方が良い。

C) 最終次元以外を分割するときは、シャドウのサイズを0にする方が良い。

一次元FFTライブラリを呼び出す例

```
      real r(m,n), wk(m,n)
!HPF$ distribute (block,*) :: r, wk
!HPF$ shadow (0,0) :: r, wk
      ...
      np = number_of_processors()
      call DFRMBF(n,m/np,r,m/np,1,
     +      isw,ifax,trigs,wk,ierr)
```

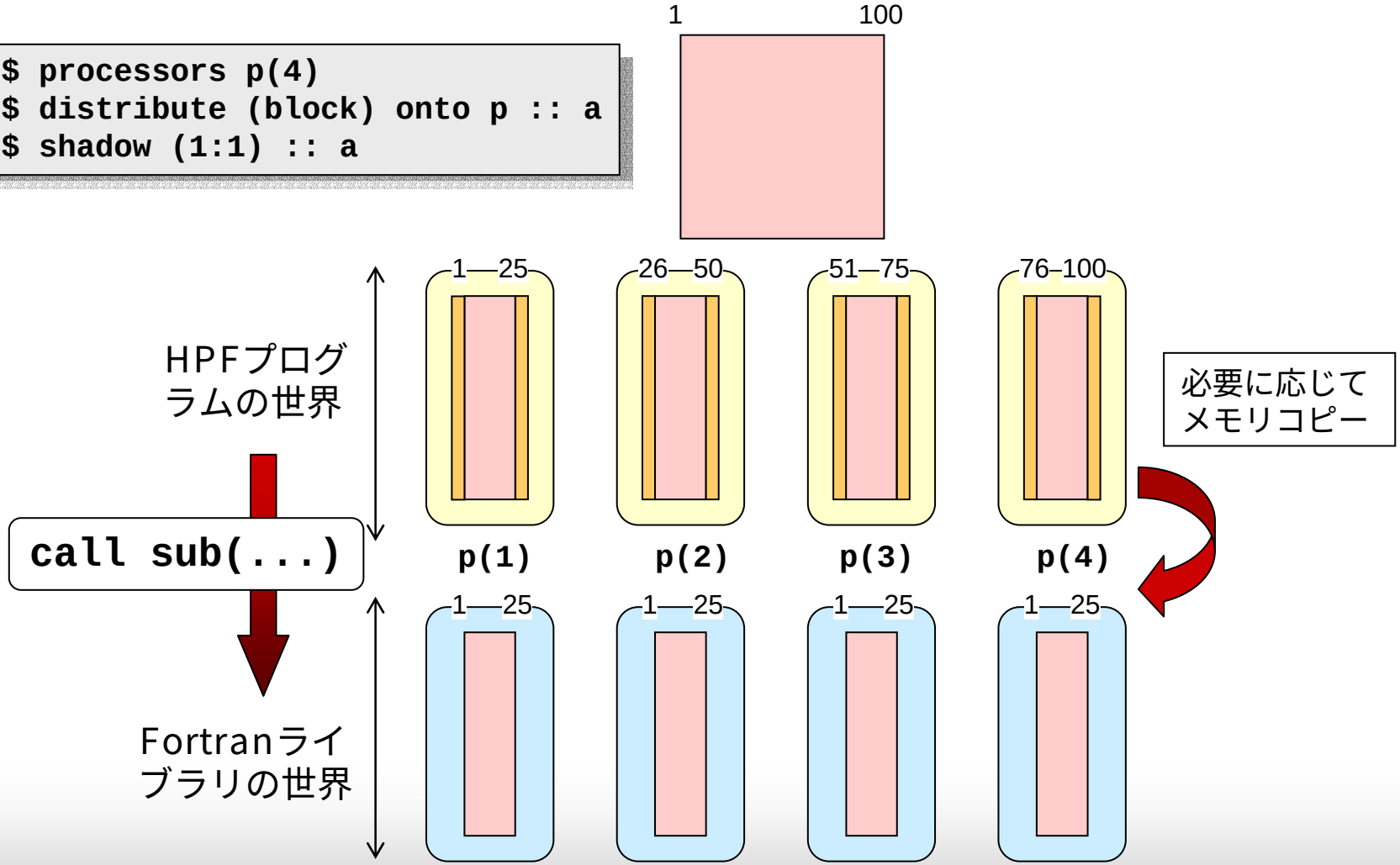
B)

C)

A)

HPF外来機能(4) Fortranライブラリ

```
!HPF$ processors p(4)
!HPF$ distribute (block) onto p :: a
!HPF$ shadow (1:1) :: a
```



HPF外来機能(5) MPI手続き

- 呼び側(caller)

  インタフェースブロック内で、MPI手続きを「HPF_LOCAL手続き」として宣言する。

引数の分散次元には「:」を指定して、形状引継ぎ配列とする。

分散指定は、呼び出し前の状態を示す(必要に応じ、再分散が起こり得る)。

```
      interface
        extrinsic(HPF_LOCAL)
+       subroutine sub(a)
        real a(:)
!HPF$   distribute (block) :: a
        end subroutine
      end interface
```

HPF外来機能(6) MPI手続き

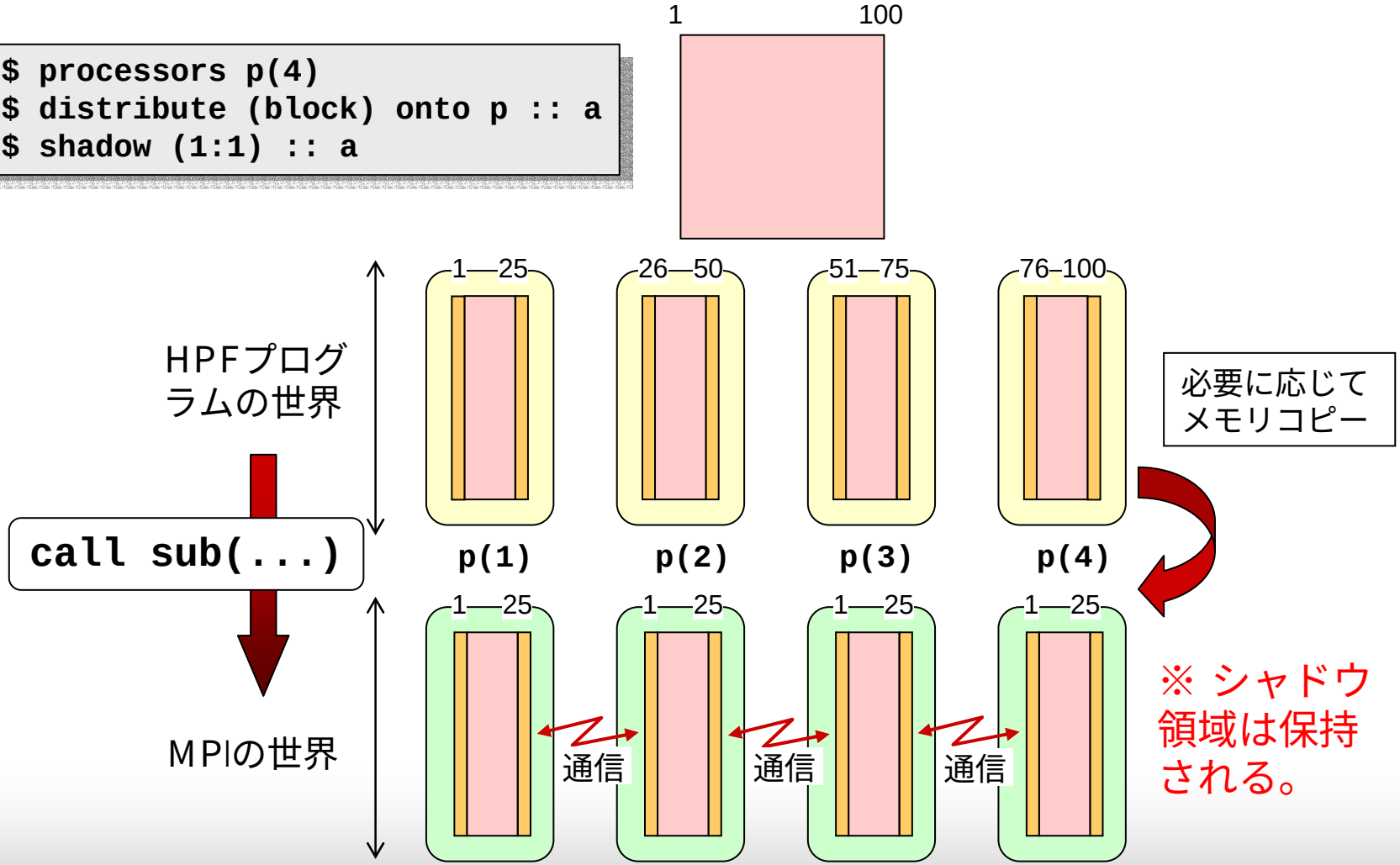
- 呼ばれる側(callee)
  - HPF_LOCAL手続きとして宣言する。
  - 「**mpif.h**」をインクルードする。
  - **MPI_Init**と**MPI_Finalize**を除く任意のMPIライブラリを使える。

```
      EXTRINSIC (HPF_LOCAL)
     +   SUBROUTINE SUB(A)
      USE HPF_LOCAL_LIBRARY
      INCLUDE 'mpif.h'
      ...
      CALL MPI_Send(...)
      ...
      END
```

※ 以下が保証される。
- コミュニケータ == **MPI_COMM_WORLD**
- MPIプロセス数 == **number_or_processors()**
- MPIランク == **my_processor()**

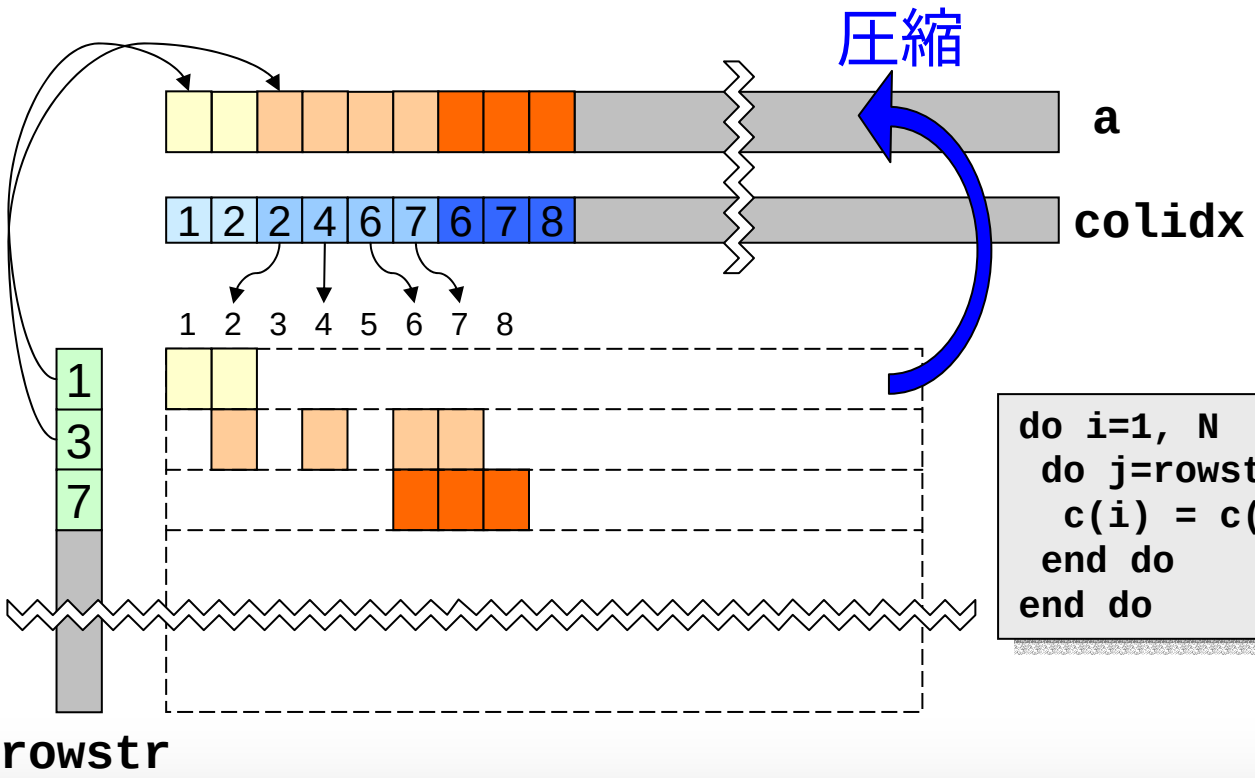
HPF外来機能(7) MPI手続き

```
!HPF$ processors p(4)
!HPF$ distribute (block) onto p :: a
!HPF$ shadow (1:1) :: a
```



不規則問題(1) 疎行列

- 圧縮された疎行列に対する行列積

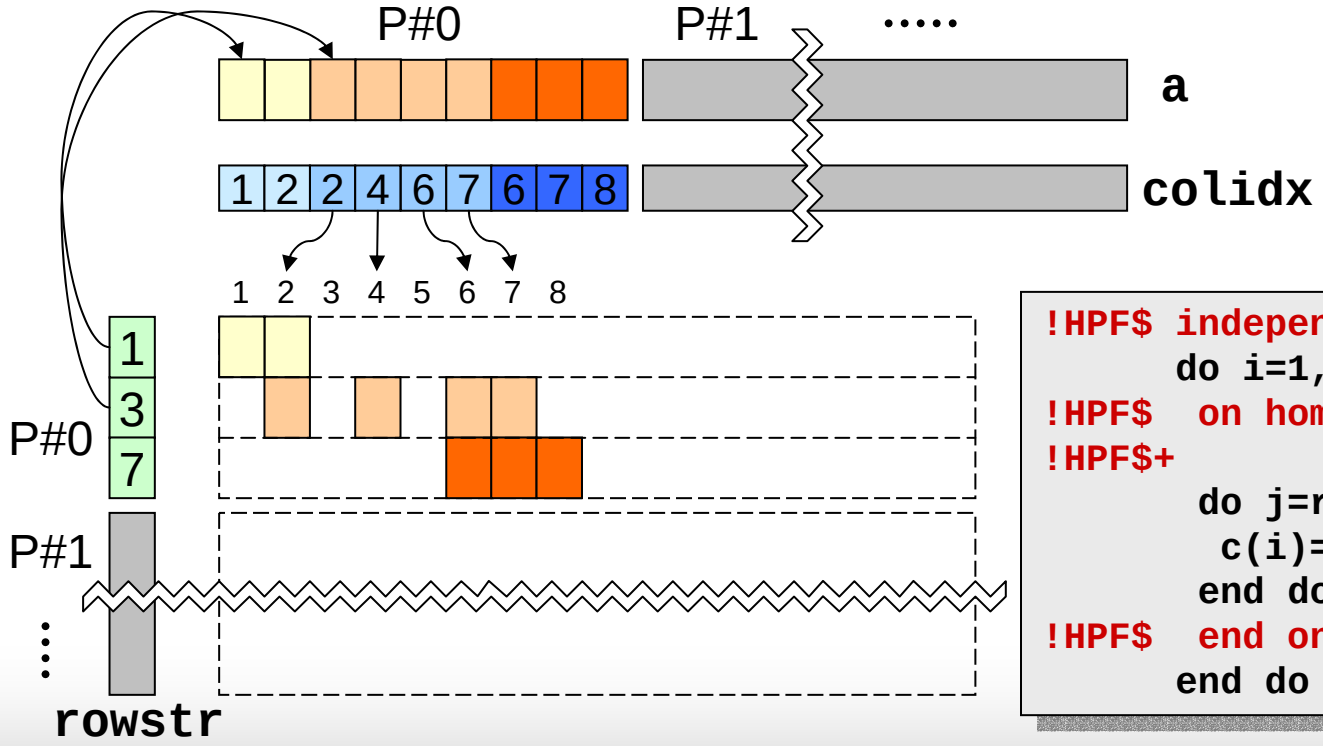



```
do i=1, N
 do j=rowstr(i), rowstr(i+1)-1
  c(i) = c(i) + a(j)*b(colidx(j))
 end do
end do
```

不規則問題(2) 疎行列

rowstrの分散に合わせて、圧縮された配列aのマッピング配列mapを設定する。

```
      integer :: map(NP) = (/9, .../)
!HPF$ DISTRIBUTE (BLOCK) :: c, rowstr
!HPF$ DISTRIBUTE (GEN_BLOCK(map)) :: a, colidx
```

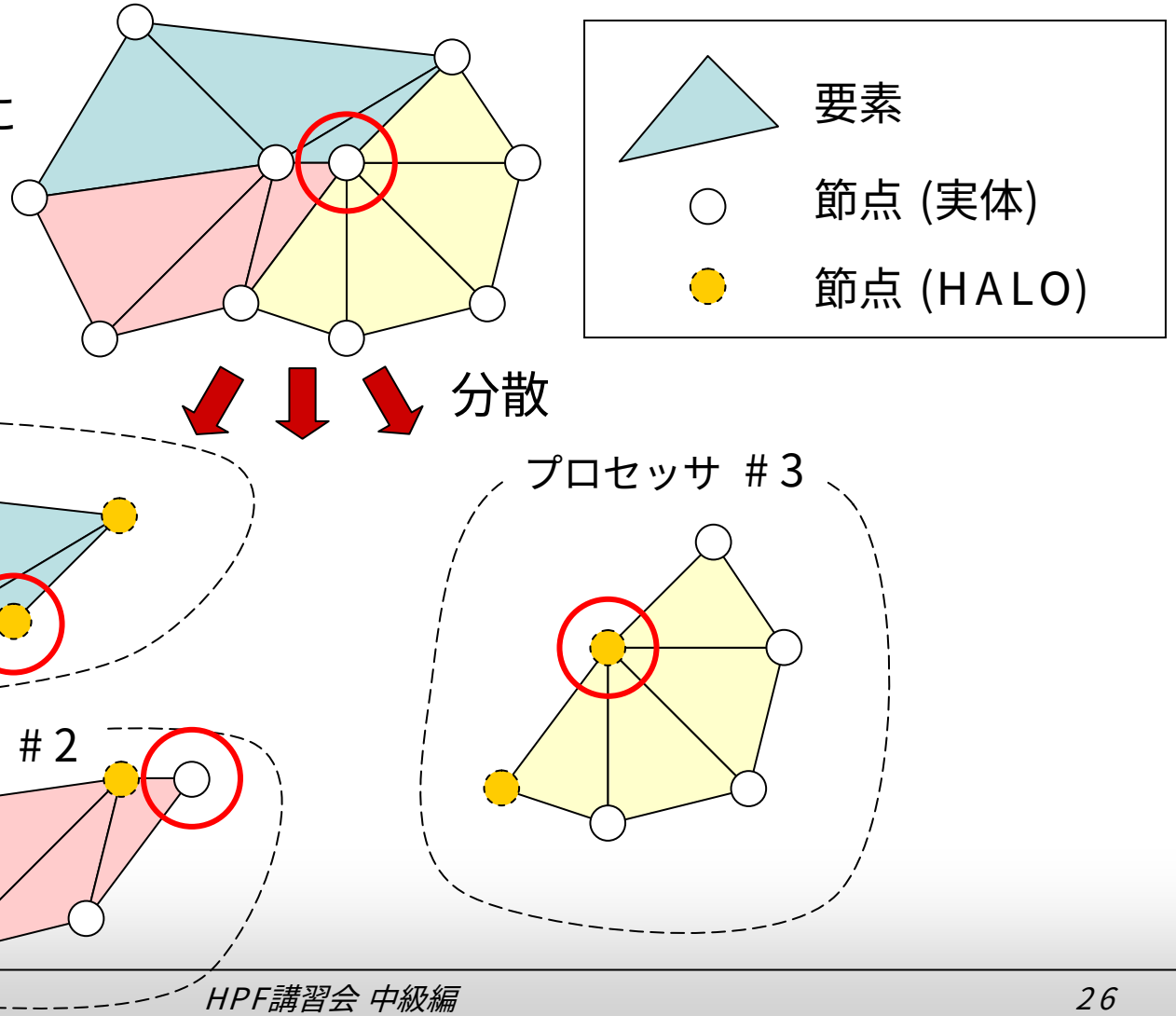



```
!HPF$ independent
      do i=1, N
!HPF$  on home(rowstr(i)),
!HPF$+        local(a, colidx) begin
       do j=rowstr(i),rowstr(i+1)-1
        c(i)=c(i)+a(j)*b(colidx(j))
       end do
!HPF$  end on
      end do
```

不規則問題(3) 有限要素法

分散境界上の節点は、
- 一つのプロセッサに実体として、
- 他のプロセッサにHALOとして、

割り付ける。

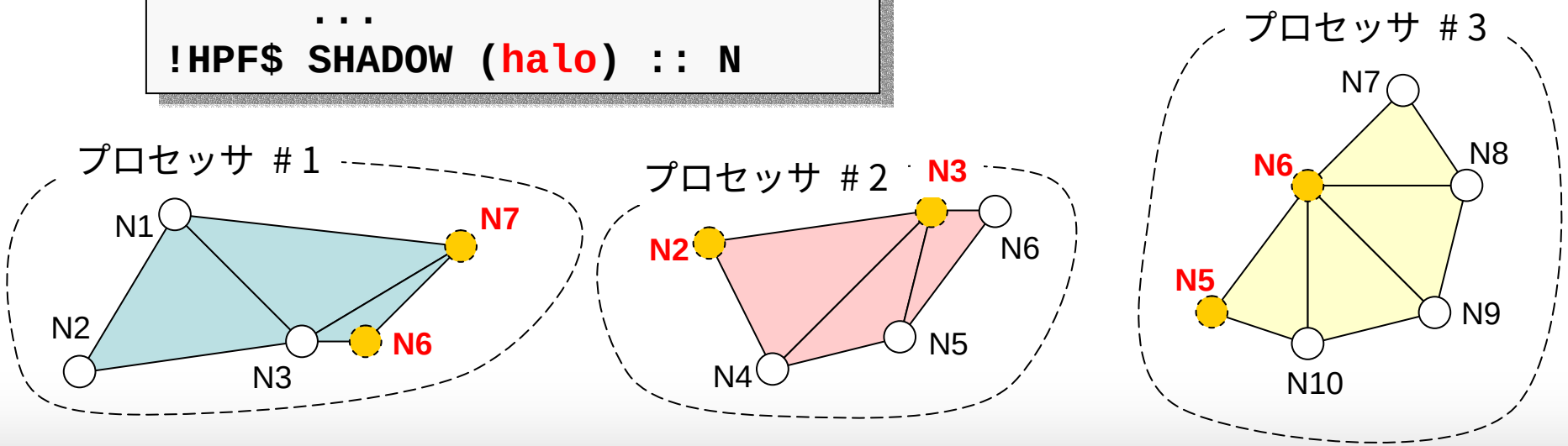


不規則問題(4) HALOの宣言

SHADOW指示文に「HALO初期化配列」を指定する。

※ あらかじめ、メッシュのパーティショニングとリオーダリングを済ませておく必要がある。

```
        halo(1)%index = (/6,7/)
        halo(2)%index = (/2,3/)
        halo(3)%index = (/5,6/)
        ...
!HPF$ SHADOW (halo) :: N
```



不規則問題(5) ＨＡＬＯのアクセス

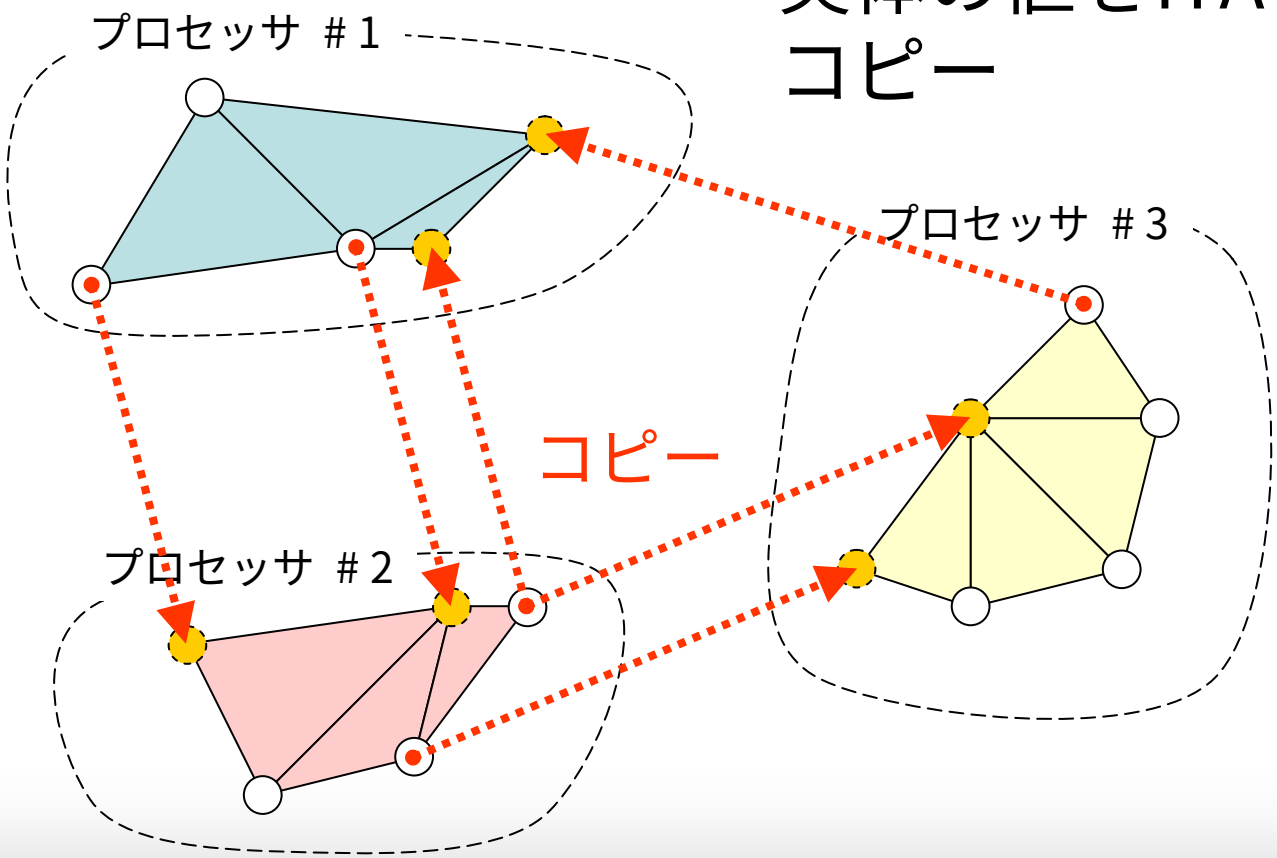
ＨＡＬＯに対し、次の3つのアクションを指定できる。

- 更新 (reflect)
- 参照
- リダクション

不規則問題(6) ＨＡＬＯの更新

!HPFJ REFLECT N

実体の値をＨＡＬＯへコピー

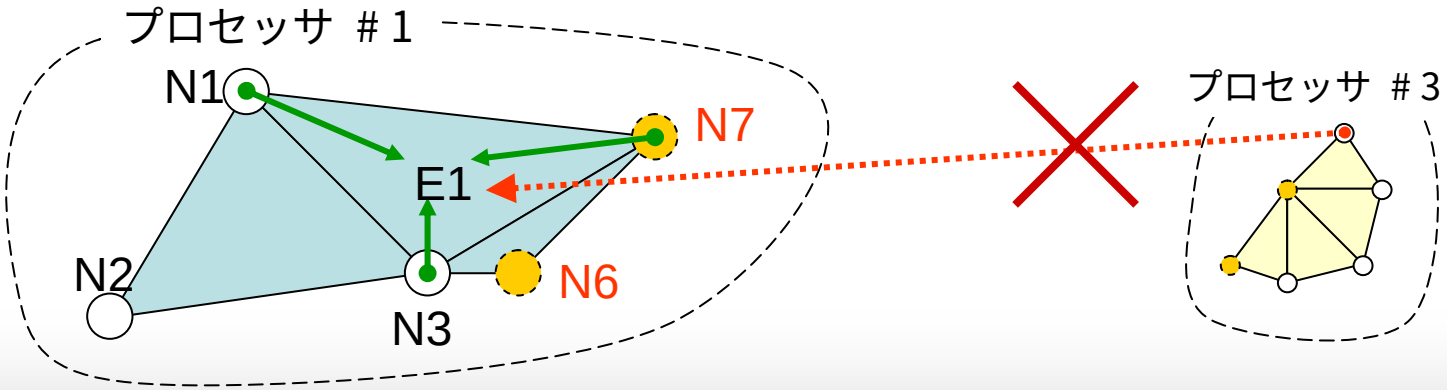


不規則問題(7) HALOの参照

```
!HPFJ INDEX_REUSE (.true.) N
!HPF$ INDEPENDENT
      DO i=1, ENUM
!HPF$  ON HOME(E(i)), LOCAL(N) BEGIN
       DO j=1, 3
        E(i) = E(i) + N(idx(i,j)) + 1
       END DO
!HPF$  END ON
      END DO
```

LOCAL節により、各プロセッサが、指定されたデータを保持している(通信は必要ない)ことを明示する。

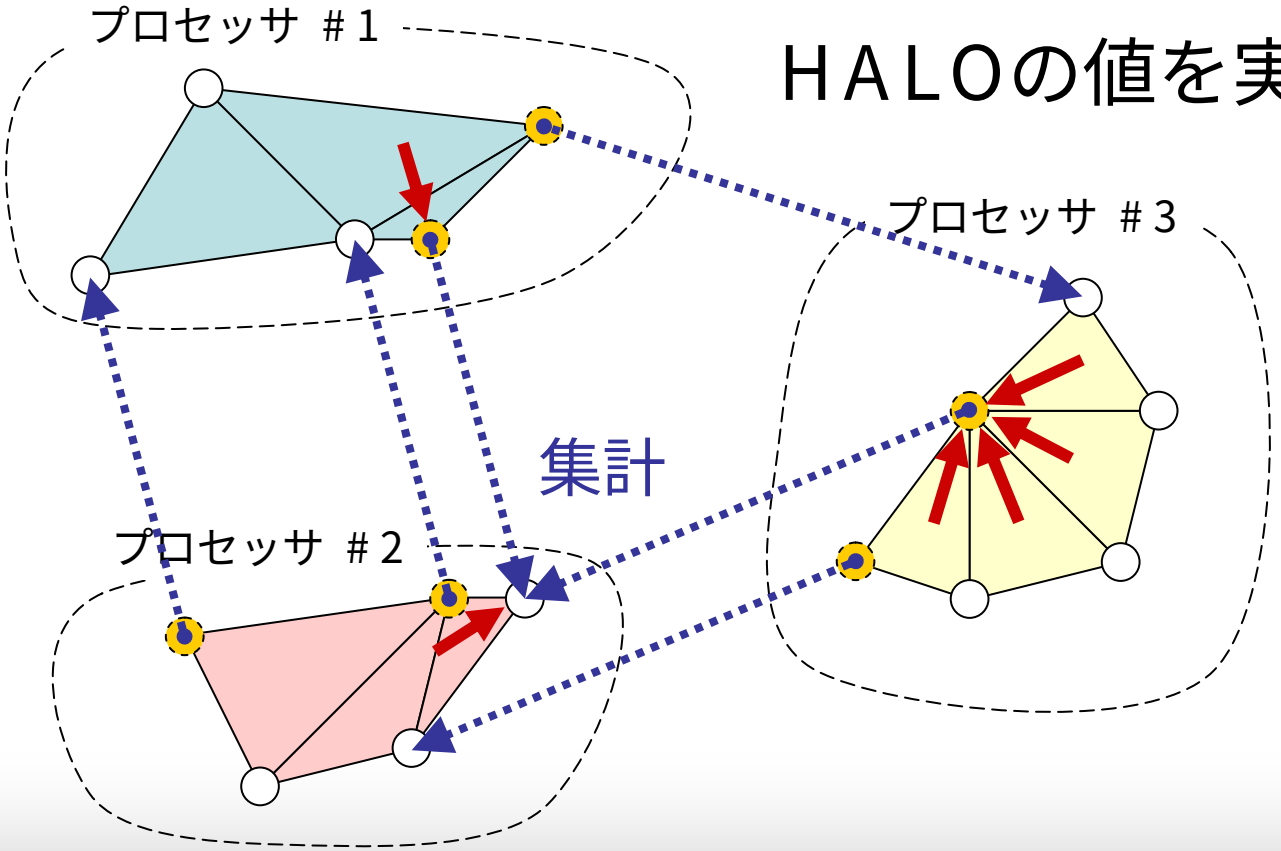
INDEX_REUSE指示文でインデックス配列が不変であることを示せば、さらに高速化できる。



不規則問題(8) HALOのリダクション

```
!HPF$ INDEPENDENT, REDUCTION(LOCAL:N)
```

HALOの値を実体へ集計



不規則問題(9) 問題点

疎行列のマッピング配列と有限要素法のHALOは、実行時に定まる。

➡ 静的な分散(メインの宣言部やモジュール等)には書けない。

再分散を使う方法

```
      real a(100)
!HPF$ distribute a(block)
!HPF$ dynamic a
      ...
      read(10) map

!HPF$ reshadow (halo) :: a
```

assertion指示文を使用しないと性能が低下する。

手続き呼び出しを介する方法

```
      read(10) map

      call sub(map)
      ...
      subroutine sub(map)
      real a(100)
!HPF$ distribute a(gen_block(map))
```

手続きの呼び出し関係が複雑になる。

HPF/ESの限界(1)

HPF/ESでうまく並列化できないパターン

- 回帰演算

```
do i=2, N
 a(i) = a(i) + a(i-1)
end do
```

- パック＆アンパック

```
do i=1, N
 if (cond) then
  j = j + 1
  b(j) = a(i)
 end if
end do
```

※ 現状では、これらをうまく扱うには、MPI手続きを呼ぶしかない。

- ソート

```
do i=1, N
 do j=N, i, -1
  if (a(j-1) > a(j)) then
   swap = a(j-1)
   a(j-1) = a(j)
   a(j) = swap
  end if
 end do
end do
```

- サーチ

```
do i=1, N
 if (a(i) == 0) exit
end do
```

HPF/ESの限界(2)

- I/Oを含むループ

  デバッグやエラーチェックのためのI/O

```
DO I=1, N
  A(I) = ...
  IF (A(I) /= 0) THEN
    WRITE(*,*) ...
  END IF
END DO
```

※ 特定の条件を満たす場合には、並列化できるよう改良を予定している。