HPF応用編1

平成19年7月
NEC

NEC

内容

- コードのクリーンナップ
- HPFによる並列化の基本手順
- 配列宣言とデータマッピング
- 不均等なデータマッピングの利用
- 動的なマッピングの変更
- HPFプログラム高速化手法
- アプリケーションの並列化
- HPFプログラムにおける計時
- Information

NEC

マップできない配列

- 他の変数とメモリ領域を共有する配列はマップできない。(翻訳時にエラーとなる)
  - EQUIVALENCE文に記述した配列
    - 作業配列や、動的に大きさを決めたい配列には、割付け配列や自動割付け配列を使う
    - ある型の配列を別の型で参照するようなプログラミングは避ける。
  - POINTER/TARGET属性を持つ配列
    - 動的にメモリを確保したい場合には、割付け配列や自動割付け配列を利用する。

```
・自動割付け配列
SUBROUTINE SUB(N)
COMMON /COM/ M
REAL A(N,M)  ！大きさが、引数や共通ブロック変数により実行時に決まる局所配列
```


NEC

コードのクリーンナップ: マップできない配列

- EQUIVALENCEされた配列の修正例
  - どうしても、場所によって、異なる形状で記述する必要がある場合

```
REAL A(10,10),B(100)
EQUIVALENCE (A,B)
```

必要に応じて配列を割り付け、値をコピー

```
      REAL, ALLOCATABLE :: A(:,:),B(:)
!HPF$ DISTRIBUTE A(*,BLOCK)
!HPF$ DISTRIBUTE B(BLOCK)
      ALLOCATE(B(N*N))          ! Bを割付け
       ....                     ! Bを参照
      ALLOCATE(A(N,N))          ! Aを割付け
      DO I=1,N
        DO J=1,N
          A(J,I) = B(J+N*I)     ! BからAへコピー
        ENDDO
      ENDDO
      DEALLOCATE(B)             ! Bを解放
       ....                     ! Aを参照
```


NEC

Fortranとの非互換部分

- **手続間で結合する配列**の形状等は、原則として同一でなければならない
  - **実引数と仮引数**: 例1) アドレス渡し

```
      real a(100,100)
!hpf$ distribute a(*,block)
      do i=1,100
         call sub(a(1,i))
      enddo
      end
      subroutine sub(a)
      real a(100)
```

例2) 実引数と仮引数の形状が異なる

```
      real a(10000)
      call sub(a,10)
      end


      subroutine sub(a,n)
      real a(n,n)
!hpf$ distribute a(*,block)
```

  - **共通ブロック**: 例1) マッピング指定漏れ

```
      common /com/a(100,100)
!hpf$ distribute a(*,block)
      end
      subroutine sub()
      common /com/a(100,100)
```

例2)使わない手続で宣言省略

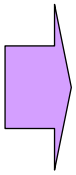
```
      common /com/a(100),b(100)
!hpf$ distribute (block) :: a,b
      end
      subroutine sub()
      common /com/a(100)
!hpf$ distribute (block) ::a
```


NEC

コードのクリーンナップ: 引数結合(1)

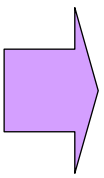
- 対応する実引数と仮引数の形状等は一致させる。

```
      real a(100,100)
!hpf$ distribute a(*,block)
      do I=1,100
         call sub(a(1,I)) ! アドレス渡し
      enddo
      end
      subroutine sub(a)
      real a(100)
```



```
      real a(100,100)
!hpf$ distribute a(*,block)
      do I=1,100
         call sub(a(:,I)) ! 部分配列
      enddo
      end
      subroutine sub(a)
      real a(100)
```

```
real a(100000)    ! 大きめに
call sub(a,10)
end
subroutine sub(a,n)
real a(n,n)
```

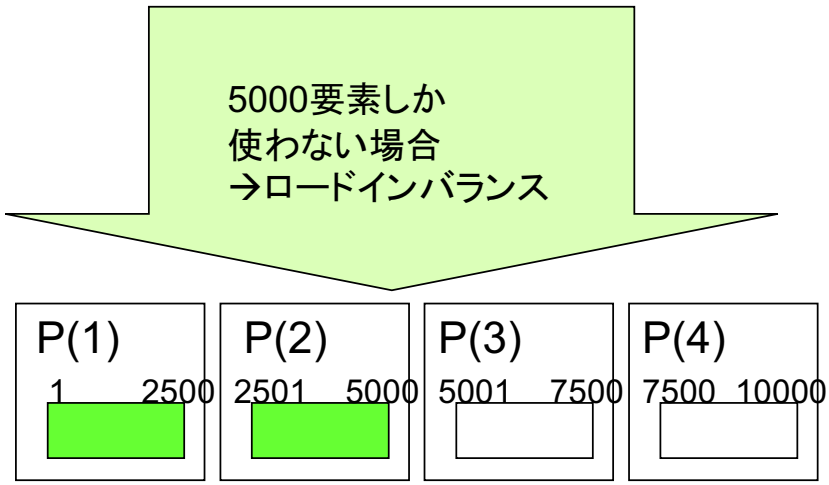


```
real,allocatable :: a(:,:) ! 割付け配列
read(5) n
allocate(a(n,n)) !使用する形状で割付け
call sub(a,n)
end
subroutine sub(a,n)
real a(n,n)
```


NEC

配列サイズを大きめに宣言する弊害

```
      REAL  A(10000)
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P
```



•サイズが動的に決まる場合は、割付け配列、自動割付け配列を利用する

NEC

コードのクリーンナップ： 引数結合(2)

·大きさ引継ぎ配列(擬寸法配列)は、最後の次元の寸法がないためマップしたり、マップされた実引数と結合できない。
→整合配列、または形状引継ぎ配列を利用する。

```
real a(100,100)
call sub(a)
   :
end

subroutine sub(a)
real a(100,*)
```

整合配列

```
real a(100,100)
call sub(a,100)
   :
end

subroutine sub(a,n)
real a(100,n)
```

形状引継ぎ配列:
Interfaceブロック等により、呼出し側で、呼ばれ側手続の引数が形状引継ぎ配列であること等を明示する必要がある

```
real a(100,100)
interface
  subroutine sub(a)
  real a(:,:)
  end subroutine
end interface
call sub(a,100)
   :
end

subroutine sub(a)
real a(:,:)
```


NEC

コードのクリーンナップ： 引数結合(3)

■並列化しない手続など、配列をマップしなくて良い場合は、SEQUENCE指示文を実引数、仮引数に指定する、という方法もある

- SEQUENCE指示文を指定した配列はマップできない代わりに、手続間で形状を一致させる必要はない

```
 subroutine fsub()
real a(10,10),b(10,10)
call fsub2(a(1,i),b,i)
end
subroutine fsub2(a,b,i)
real a(10),b(10,10)
```

これらの手続は、並列化しない場合

```
      subroutine fsub()
      real a(10,10),b(10,10)
!HPF$ SEQUENCE a,b
      call fsub2(a(1,i),b,i)
      end
      subroutine fsub2(a,b,i)
      real a(10),b(10,10)
!HPF$ SEQUENCE
```

- 変数名を省略したSEQUENCE指示文は、出現する全ての非分散配列や共通ブロックを指定したのと同じ意味になる。
- コンパイラによっては、翻訳時オプションで、変数指定なしのSEQUENCE指示文が指定されたかの様に扱うことも可能。(HPF/SX V2の-Msequence等)

NEC

コードのクリーンナップ： 共通ブロック(1)

- 共通ブロック中の変数の数、各変数の形状、マッピングは、全手続で一致させる

```
      subroutine sub()
      common /com1/a(100,100)
      common /com2/b(100,100),c(100,100)
!hpf$ distribute (*,block) :: a
!hpf$ distribute (*,block) :: b
```

sub()とsub2()の共通ブロック変数の宣言を同一に

```
      subroutine sub2()
      common /com1/a(100,100)
      common /com2/b(100,100),c(100,100)
!hpf$ distribute (*,block) :: a
!hpf$ distribute (*,block) :: b
```


NEC

コードのクリーンナップ　共通ブロック(2)

•大域変数はモジュールやINCLUDEファイル中に宣言すると、宣言の記述は一回で済むため、書き忘れや書き誤りが防げる。

モジュールによる大域変数の宣言

```
      module com1
      dimension a(100,100)
!hpf$ distribute (*,block) :: a
      end module
      module com2
      dimension b(100,100),c(100,100)
!hpf$ distribute (*,block) :: b,c
      end module
      subroutine sub()
      use com1
      use com2
      subroutine sub2()
      use com1
      use com2
```

INCLUDEファイルによるCOMMONの宣言

```
%> cat  com1.h
       common /com1/ a(100,100)
!hpf$ distribute (*,block) :: a
%> cat  com2.h
       common /com2/ b(100,100),c(100,100)
!hpf$ distribute (*,block) :: b,c
```

```
      subroutine sub()
      include com1.h
      include com2.h
      subroutine sub2()
      include com1.h
      include com2.h
```


NEC

内容

- コードのクリーンナップ
- HPFによる並列化の基本手順
- 配列宣言とデータマッピング
- 不均等なデータマッピングの利用
- 動的なマッピングの変更
- HPFプログラム高速化手法
- アプリケーションの並列化
- HPFプログラムにおける計時
- Information

NEC

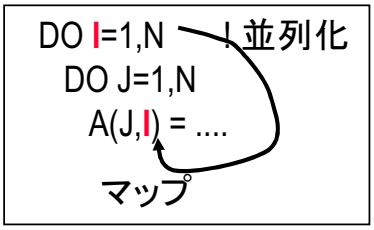
HPFプログラミングの基本手順

1. コードのクリーンナップ
2. どのループを並列化するか決める
   - 一番実行コストの高いループ
3. 配列のマッピングを決める
   - 並列化するループに対応する次元でマップ
4. HPFコンパイラで翻訳してみる
5. 診断メッセージ等を見て、指示文を追加、修正
   - 配列のマッピングの追加、修正
   - 並列化可能であること(INDEPENDENT)や通信不要であること(ON, LOCAL) を明示する指示文の追加
   - その他の指示文の追加
6. 上記の4、5を繰り返す。

NEC

配列のマッピングの決め方(1)

- どのループを並列化するか決める
  - 最も実行コストの高い手続の最も実行コストの高いループ
  - 最終結果を計算しているループ
- 並列化するループでアクセスされる配列の次元をマップ
  - 多重ループの場合、通常は
    ・最外側を並列化
    するのがもっとも効率的
    - 並列化のオーバヘッドを最小にするため



  - 多次元配列の場合、メモリアクセスの連続性から、通常は
    ・最終次元をHPFにより分散
    するのがもっとも効率的

NEC

配列のマッピングの決め方(2) 指示文

- 配列宣言と指示文
  - 分散したい次元の上下限が、全ての配列で同一の場合、マッピングはDISTRIBUTE指示文だけでほぼOK
    （マッピングを指定する指示文としては、他にALIGN指示文もある）

```
        PARAMETER (IX=100,IY=100)
         REAL*8  A(IX,IY),B(0:IX,IY),C(IX+1,IY)
!HPF$ DISTRIBUTE (*,BLOCK) :: A,B,C
                ...
        DO J=1,IY -1    ! 並列化
          DO I=1,IX
            A(I,J) = B(I,J)-B(I-1,J)+C(I,J+1)-C(I,J)
          END DO
        END DO
```


NEC

配列のマッピングの決め方(3) 分散形式

- 分散形式の選択
  - 規則的な問題なら、通常BLOCK分散またはGEN_BLOCK分散を利用。(通信や参照時の効率がよい)
  - 何次元分散するかは、何重並列化するかで決める。（＝プロセッサ構成の次元数）通常は1次元で十分。

```
DO K=1,N ! 並列化                    1次元分散
  DO J=1,N                           !HPF$ PROCESSORS P(4)
    DO I=1,N ! ベクトル化            !HPF$ DISTRIBUTE  A(*,*,BLOCK) ONTO  P
      A(I,J,K)  = ....
```

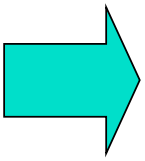
```
DO K=1,N ! 並列化                    2次元分散
  DO J=1,N ! 並列化                  !HPF$ PROCESSORS P(2,2)
    DO I=1,N ! ベクトル化            !HPF$ DISTRIBUTE  A(*,BLOCK,BLOCK) ONTO  P
      A(I,J,K)  = ....
```


NEC

配列のマッピングの決め方(4)

- 既に分散された配列を基準に、他の配列をマップする
  - 分散配列が右辺に出現するループ中で定義される配列は分散配列と同じプロセッサに配置する。

```
      real a(10),b(10)
!HPF$ distribute (block)  :: a
      do i=1,n
        b(i) = a(i)
      enddo
```



```
      real a(10),b(10)
!HPF$ distribute (block) :: a,b
      do i=1,n
        b(i) = a(i)
      enddo
```

  - 分散配列が実引数になっている場合、対応する仮引数にも同一のデータマッピングを指定する。
  - 分散配列が仮引数になっている場合、対応する実引数にも同一のデータマッピングを指定する。
  - 共通ブロック変数を分散する場合、出現する全ての手続で同一の分散を指定する。

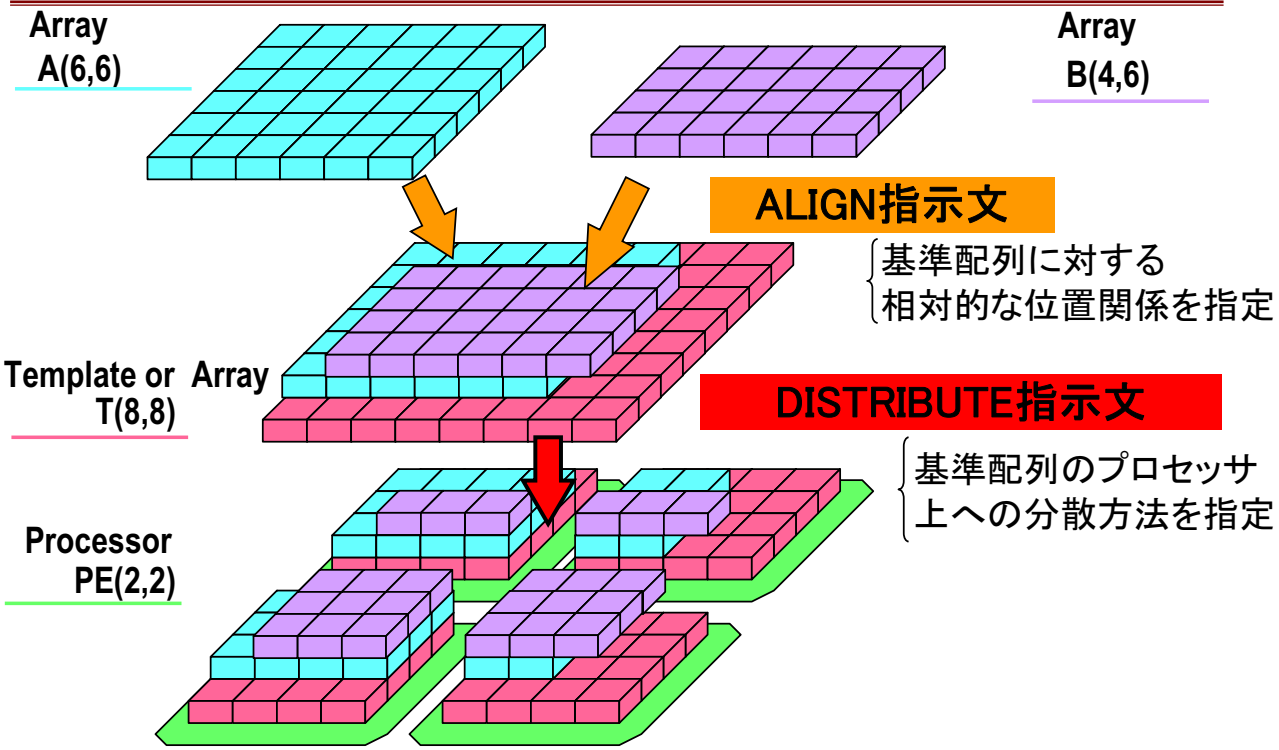
NEC

内容

- コードのクリーンナップ
- HPFによる並列化の基本手順
- 配列宣言とデータマッピング
- 不均等なデータマッピングの利用
- 動的なマッピングの変更
- HPFプログラム高速化手法
- アプリケーションの並列化
- HPFプログラムにおける計時
- Information

NEC

HPFのデータマッピングモデル



NEC

ALIGN指示文(1)

- 基準配列に対する相対的な位置関係(整列)により、間接的にデータマッピングを指定
- 4通りの整列方法
  - 3つ組
  - 縮退
  - 複製
  - Single整列
- *Ai*(整列元)を *T*(整列先)の対応する要素と同じ抽象プロセッサへマップする。

形式

・1つの配列の整列

ALIGN *A1*(*i* | : | * ,...) WITH *T*(*整数式* | [*l*]:[*u*][:*s*] | * ,...)

・複数の配列の整列

ALIGN (*i* | : | * ,...) WITH *T*(*整数式* | [*l*]:[*u*][:*s*] | * ,...) :: *A1,A2,...*

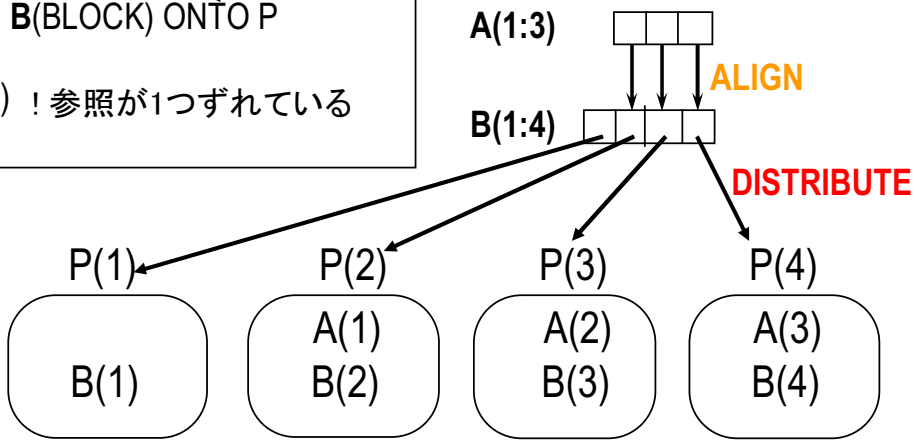
NEC

ALIGN指示文(2) 3つ組

- 整列元と整列先の特定次元の添字を一次式で対応させる

```
        REAL A(3), B(4)
!HPF$ PROCESSORS P(4)
!HPF$ ALIGN A(i) WITH B(i+1)
!HPF$ DISTRIBUTE B(BLOCK) ONTO P
        DO i=1,3
          A(i)=B(i+1) ！参照が1つずれている
        ENDDO
```

A(i)をB(i+1)と同じプロセッサへ



- ストライドを指定する事もできる。

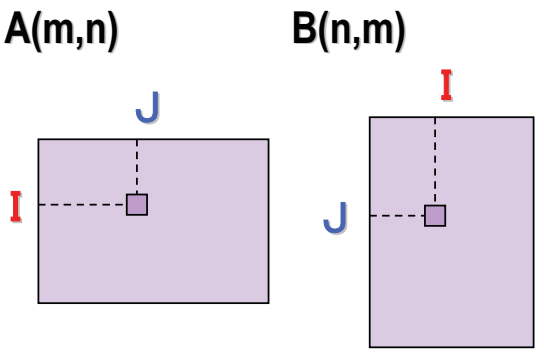
例） ALIGN C(i) WITH D(2*i-1)　　！Dの奇数添字に整列

NEC

ALIGN指示文(3) 3つ組(多次元の場合)

- 同じ変数(*align-dummy*)が記述された次元同士が対応

ALIGN A(I,J) WITH B(J,I)



適用例:

```
DO I=1,m
  DO J=1,n
    A(I,J)=B(J,I)
  END DO
END DO
```


NEC

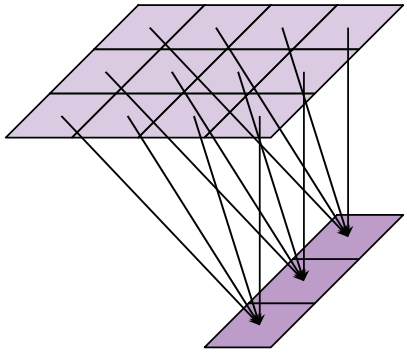
ALIGN指示文(4) 縮退

- 次元数が異なる場合(1) 縮退
  - 整列先の方が次元数が小さい場合
- ‘＊’が指定された次元は、マップされない(整列の相対的位置に影響しない)

```
ALIGN A(I,*) WITH B(I)
```

A(3,4)

B(3)



適用例:

```
DO I=1,3
   DO J=1,4
      B(I)=B(I)+A(I,J)
   END DO
END DO
```


NEC

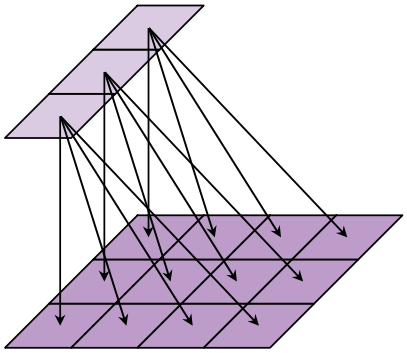
ALIGN指示文(5) 複製

- 次元数が異なる場合(2) 複製
  - 整列先の方が次元数が大きい場合
- ‘＊’が指定された整列先の次元に沿って、整列元を複製

```
ALIGN A(I) WITH B(I,*)
```

A(3)

B(3,4)



適用例:

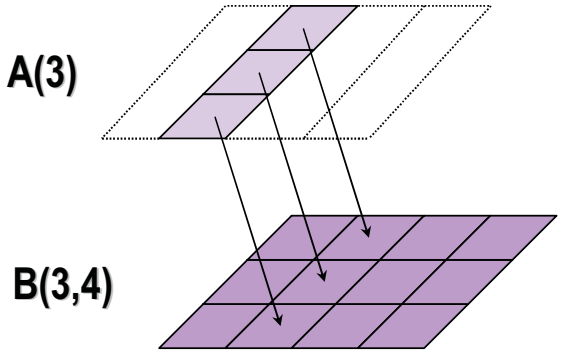
```
DO J=1,4
   DO I=1,3
      B(I,J)=A(I)
   END DO
END DO
```


NEC

ALIGN指示文(6) Single整列

- 整列先の特定次元の1断面への整列

ALIGN A(I) WITH B(I,2)



適用例:

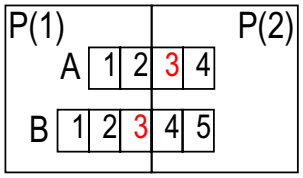
```
        REAL A(3,4),B(3,4)
!HPF$ ALIGN A(I,J) WITH B(I,J) ! 1対1に整列
      CALL SUB(A(:,2),B) ! 2列目を渡す
        ...
        SUBROUTINE SUB(A,B)
        REAL A(3),B(3,4)
!HPF$  ALIGN A(I) WITH B(I,2) ! 2列目に整列
```


NEC

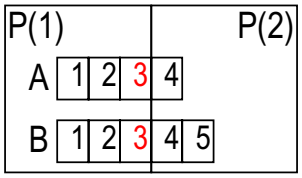
ALIGN指示文を使った方がよい場合(1)

- 関連する配列の宣言範囲が異なる場合

```
      real a(4),b(5)
!hpf$ processors p(2)
!hpf$ distribute (block) onto p :: a,b
      do i = 1,4
        a(i) = b(i)  ! 通信発生
```




```
      real a(4),b(5)
!hpf$ processors p(2)
!hpf$ align a(i) with b(i)
!hpf$ distribute (block) onto p :: b
      do i = 1,4
        a(i) = b(i)  ! 通信不要
```



- 関連する配列の宣言範囲が見かけ上異なる場合も含む

```
      subroutine sub(n,m)
      real a(n),b(m)
!hpf$ distribute (block) :: a,b
```

```
      subroutine sub(n,m)
      real a(n),b(m)
!hpf$ align a(i) with b(i)
!hpf$ distribute (block) :: b
```


NEC

ALIGN指示文を使った方がよい場合(2)

・上下限値が翻訳時に不明の場合、DISTRIBUTE指示文のみでは、2つの配列の分割幅が一致するか否か分からないので、通信発生

•割付け配列の場合

```
      real,allocatable :: a(:),b(:),c(:)
!hpf$ distribute (block) :: a,b,c
      do i = 1,n
        a(i) = b(i) + c(i)  !通信発生
```

```
      real,allocatable :: a(:),b(:),c(:)
!hpf$ align (i) with c(i) :: a,b
!hpf$ distribute (block) :: c
      do i = 1,n
        a(i) = b(i) + c(i)  !通信不要
```

•例えば、a(1:100),b(1:50),c(1:10)のように、割付け範囲がばらばらになる可能性がある

•形状引継ぎ配列の場合

```
      subroutine sub(a,b,c)
      real  a(:),b(:),c(:)
!hpf$ distribute (block) :: a,b,c
      do i = 1,n
        a(i) = b(i) + c(i)  !通信発生
```

```
      subroutine sub(a,b,c)
      real  a(:),b(:),c(:)
!hpf$ align (i) with c(i) :: a,b
!hpf$ distribute (block) :: c
      do i = 1,n
        a(i) = b(i) + c(i)  !通信不要
```

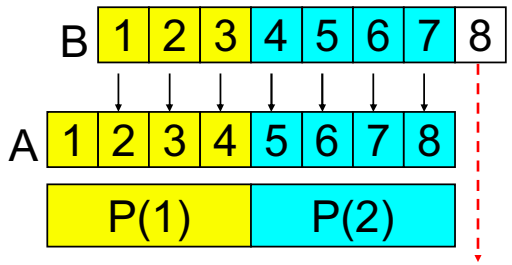

NEC

TEMPLATE指示文 メモリを持たない配列の宣言

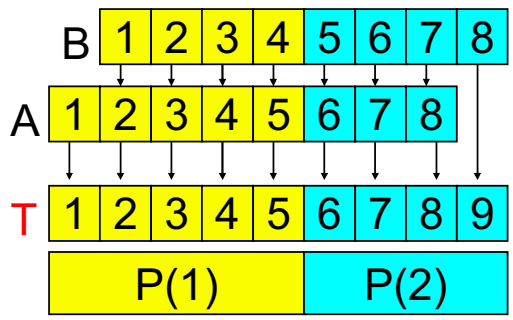
・マッピングを指定する適当な配列がない場合

・ON指示構文に指定する適当な配列がない場合

```
      INTEGER A(8),B(8)
!HPF$ PROCESSORS P(2)
!HPF$ ALIGN B(i) WITH A(i+1)  ！エラー
!HPF$ DISTRIBUTE A(BLOCK) ONTO P
      DO i=1,7
        B(i) = A(i+1)  ！参照が1つずれている
```




```
      INTEGER A(8),B(8)
!HPF$ PROCESSORS P(2)
!HPF$ TEMPLATE T(9)
!HPF$ ALIGN B(i) WITH T(i+1)
!HPF$ ALIGN A(i) WITH T(i)
!HPF$ DISTRIBUTE T(BLOCK) ONTO P
      DO i=1,7
        B(i) = A(i+1)
```



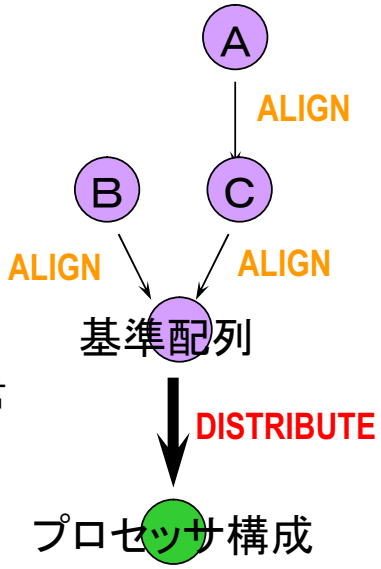
NEC

HPFのデータマッピングまとめ

1. どのループを並列化するかを決定
2. 並列化するループによりアクセスされる次元をマップ

```
DO I=1,N    ! 並列化
  DO J=1,N
    A(J,I) = B(J,I+1)
```

Aの2次元目をマップ



- 関連の深いデータは整列(ALIGN)
  - 参照パタンにより芋づる式に決定
    例) Aの2次元目 "I"とBの2次元目の"I+1"

```
!HPF$ ALIGN A(J,I) WITH B(J,I+1)
```

  - 必要に応じて、基準配列としてTEMPLATEを宣言
- 基準配列(TEMPLATE等)を分散(DISTRIBUTE)
  - 他の変数は整列関係を守ってマップされる。

NEC

内容

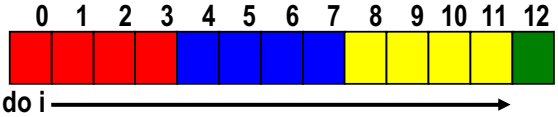
- コードのクリーンナップ
- HPFによる並列化の基本手順
- 配列宣言とデータマッピング
- 不均等なデータマッピングの利用
- 動的なマッピングの変更
- HPFプログラム高速化手法
- アプリケーションの並列化
- HPFプログラムにおける計時
- Information

NEC

不均等分散によるロードバランスの改善

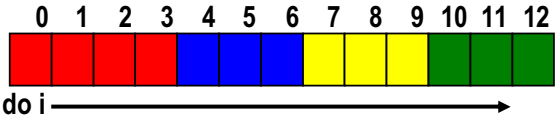
- BLOCK分散の場合、分散ブロック幅は、(配列要素数－1)/ プロセッサ数＋1
- プロセッサと配列要素数次第では、後方のプロセッサにマップされる配列要素数が少なくなったり、まったくなくなったりする場合あり。

```
      real a(0:12),b(0:12)
!HPF$ DISTRIBUTE (BLOCK) onto P :: a,b
!HPF$ PROCESSORS P(4)
      do i=1,11
        b(i) = a(i-1)+a(i) + a(i+1)
      enddo
```



P(1):3要素、P(2),P(3):4要素、P(4):0要素

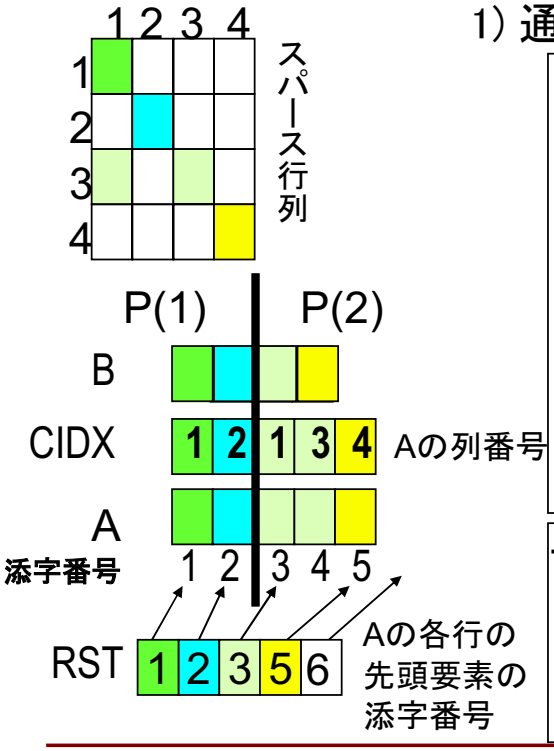
```
      real a(0:12),b(0:12)
      integer :: map = (/4,(3,i=1,3)/)
!HPF$ DISTRIBUTE (GEN_BLOCK(map)) onto P :: a,b
!HPF$ PROCESSORS P(4)
      do i=1,11
        b(i) = a(i-1)+a(i) + a(i+1)
      enddo
```



P(1),P(2),P(3):3要素、P(4):2要素

NEC

疎行列に対する不均等分散(1)



1) 通信が発生しないようマッピングを指定

```
      REAL A(5),B(4),X(4)
      INTEGER CIDX(5),RST(5)
      INTEGER :: GB(2) = (/2,3/)
!HPF$ DISTRIBUTE (GEN_BLOCK(GB)) :: A,CIDX
!HPF$ DISTRIBUTE (BLOCK) :: B
      DO I=1,4    ! 各行
        B(I) = 0.0
        DO J=RST(I),RST(I+1)-1   ! I行目のAを参照
          B(I)=B(I)+A(J)*X(CIDX(J)) ! 行列ベクトル積
        ENDDO
      ENDDO
```

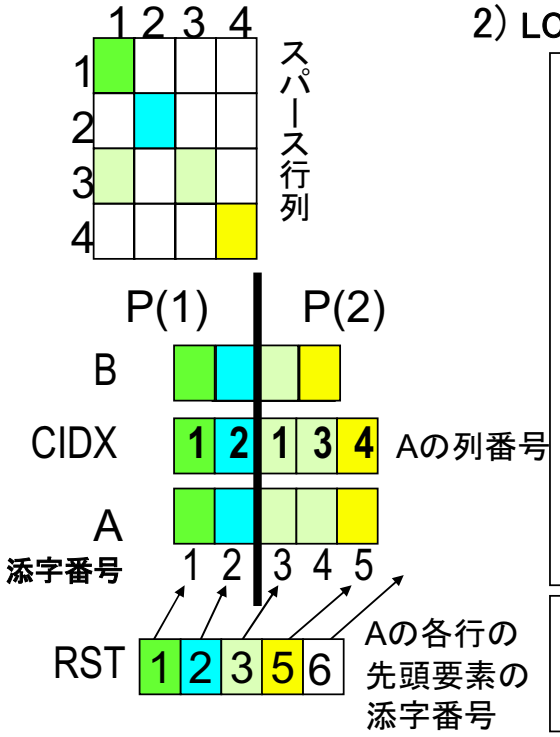
並列化対象

-Minfo：自動では通信が不要である事がわからない

```
7,  Independent loop
    Array a not aligned with home array; array copied
    Array cidx not aligned with home array; array copied
    Independent loop parallelized : b(i)
```


NEC

疎行列に対する不均等分散(2)



2) LOCAL節付きのON指示文により通信不要を明示

並列化対象

```
      REAL A(5),B(4),X(4)
       INTEGER CIDX(5),RST(5)
       INTEGER :: GB(2) = (/2,3/)
!HPF$ DISTRIBUTE (GEN_BLOCK(GB)) :: A,CIDX
!HPF$ DISTRIBUTE (BLOCK) :: B
       DO I=1,4   ! 各行
!HPF$    ON HOME(B(I)), LOCAL(A,CIDX)  BEGIN
         B(I) = 0.0
         DO J=RST(I),RST(I+1)-1  ! I行目のAを参照
           B(I)=B(I)+A(J)*X(CIDX(J))
         ENDDO
!HPF$    ENDON
       ENDDO
```

```
-Minfo
  7,  Independent loop
      Independent loop parallelized : b(i)
```


NEC

GEN_BLOCK分散時の宣言法

・GEN_BLOCK分散のためのマッピング配列は、
実行時に決めたい場合が多い。
→ 呼び側手続でマッピング配列を設定し、呼ばれ側で分散

```
      integer, allocatable :: map(:)        ! 割付け配列
      allocate(map(number_of_processors())) ! プロセッサ数分割付け
      n = 0 ! 配列サイズの初期化
      do I=1,number_of_processors()
        map(I) = …      !  I番目のプロセッサに配置する要素数
        n = n+ map(I)   ! 配列サイズの計算
      enddo
      call realmain(map,n) ! 真の主プログラム起動
      end

      subroutine realmain(map,n) ! 真の主プログラム
!hpf$ distribute a(*,gen_block(map))
      integer a(100,n),map(number_of_processors())
```


NEC

内容

- コードのクリーンナップ
- HPFによる並列化の基本手順
- 配列宣言とデータマッピング
- 不均等なデータマッピングの利用
- 動的なマッピングの変更
- HPFプログラム高速化手法
- アプリケーションの並列化
- HPFプログラムにおける計時
- Information

NEC

REDISTRIBUTE指示文

- DISTRIBUTE指示文が宣言時に配列の分散を指定するのに対して、実行時に、配列の分散を変更する
- 対象となる配列には、宣言部でDYNAMIC指示文が必要
- 再分散対象に整列している配列は全て再マップされる

```
!HPF$ DYNAMIC  A     ! DYNAMIC指示文が必要
!HPF$ DISTRIBUTE A(*,BLOCK)
      DO J=1,N      ! 並列化可能:2次元目をマップ
        DO I=2,N-1
          A(I,J)=(A(I-1,J)+2*A(I,J)+A(I+1,J))/4.0
        END DO
      END DO
!HPF$ REDISTRIBUTE  A(BLOCK,*)
      DO I=1,N      ! 並列化可能:1次元目をマップ
        DO J=2,N-1
          A(I,J)=(A(I,J-1)+2*A(I,J)+A(I,J+1))/4.0
        END DO
      END DO
```


NEC

REALIGN指示文

- ALIGN指示文が宣言時に整列を指定するのに対して、実行時に、配列の整列を変更する。
- 対象となる配列には、宣言部でDYNAMIC指示文が必要。

```
!HPF$ DYNAMIC A   !   再整列するためには、DYNAMIC指示文が必要
!HPF$ DISTRIBUTE B(BLOCK)
!HPF$ ALIGN A(*,J) WITH B(J)
      DO J=1,N     ! 並列化可能:2次元目をマップ
        DO I=2,N-1
          A(I,J)=(A(I-1,J)+2*A(I,J)+A(I+1,J))/4.0
        END DO
      END DO
!HPF$ REALIGN A(I,*) WITH B(I)
      DO I=1,N     ! 並列化可能:1次元目をマップ
        DO J=2,N-1
          A(I,J)=(A(I,J-1)+2*A(I,J)+A(I,J+1))/4.0
        END DO
      END DO
```


NEC

動的再マッピングの副作用

- DYNAMIC指示文を指定された配列は、翻訳時にマッピングが不明となる。

  → 最適な並列化の方法をコンパイラが判断できない
  （分散されていない次元で並列化すると性能低下）

```
DO I=1,N     ! 並列化可能
  DO J=1,N   ! 並列化可能
    A(I,J) =0.0.
```

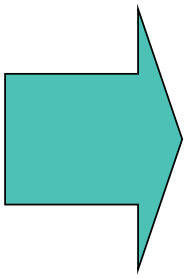
上記ループは、どちらも並列化可能なので、コンパイラは、通常分散されている配列次元に対応するループで並列化する。翻訳時にマッピングが不明の場合は、どのような分散でも動作するような非効率な並列コードが生成される。

NEC

回避策(1)

- マッピングを変えたい場合、手続を分割し、手続引数経由で変更する。1つの手続中では1つのマッピングとなるようにする。

```
!HPF$ DISTRIBUTE  A(*,BLOCK)
      REAL A(N,N)
      DO J=1,N  ！並列可能
        DO I=2,N
          A(I,J) = A(I-1,J)
        ENDDO
      ENDDO
      DO I=1,N  ！並列可能
        DO J=2,N
          A(I,J) = A(I,J-1)
            :
```



```
!HPF$ DISTRIBUTE  A(*,BLOCK)
      REAL A(N,N)
      DO J=1,N  ！並列可能
        DO I=2,N
          A(I,J) = A(I-1,J)
        ENDDO
      ENDDO
      CALL SUB(A,N)
```

```
      SUBROUTINE SUB(A,N)
!HPF$ DISTRIBUTE  A(BLOCK,*)
      REAL A(N,N)
      DO I=1,N  ！並列可能
        DO J=2,N
          A(I,J) = A(I,J-1)
            :
```


NEC

回避策(2)

- 計算マッピングの指定を通じて、データマッピングの変更をコンパイラにやらせる

```
      REAL A(N,N)
!HPF$ DISTRIBUTE A(*,BLOCK)
!HPF$ TEMPLATE T(N,N)
!HPF$ DISTRIBUTE T(BLOCK,*)
      DO J=1,N     ！並列化可能
        DO I=2,N-1 ！並列化不可
          A(I,J) = A(I-1,J)+A(I,J)
        END DO
      END DO

      DO I=1,N     ！並列化可能
!HPF$   ON HOME(T(I,:)), BEGIN
        DO J=2,N-1  ！ 並列化不可
          A(I,J) = A(I,J-1)+A(I,J)
        END DO
!HPF$   ENDON
      END DO
```

・計算マッピングを指定するためのTEMPLATEの宣言と分散の指定

・ON指示構文に、上記のTEMPLATEを記述して、計算マッピングを指定。

・コンパイラは、指定された計算マッピングに従ってループを並列化し、リモートアクセスの発生する配列Aに対して、自動的にテンポラリ配列への置換えと、配列Aとテンポラリ領域間の通信・コピーを行う。

NEC

回避策(3)

- HPFコンパイラによる変形例のイメージ

```
        REAL A(N,N), tmp(N,N)
!HPF$ DISTRIBUTE A(*,BLOCK)
!HPF$ TEMPLATE T(N,N)
!HPF$ DISTRIBUTE (BLOCK,*)  :: T, tmp
      DO J=1,N      ! 並列化
        DO I=2,N-1  ! 並列化不可
          A(I,J) = A(I-1,J)+A(I,J)
        END DO
      END DO
      tmp = A
      DO I=1,N      ! 並列化
!HPF$   ON HOME(T(I,:)), BEGIN
        DO J=2,N-1  ! 並列化不可
          tmp(I,J) = tmp(I,J-1)+tmp(I,J)
        END DO
!HPF$   ENDON
      END DO
      A = tmp
```

・ループ前後で、1次元目で分散したテンポラリtmpと、配列A間の(通信を伴う)コピーを行う。
・ループ中は、1次元目で分散されたテンポラリtmpに置き換え、分散次元に対応するDO I のループで並列化する

NEC

内容

- コードのクリーンナップ
- HPFによる並列化の基本手順
- 配列宣言とデータマッピング
- 不均等なデータマッピングの利用
- 動的なマッピングの変更
- HPFプログラム高速化手法
- 次元拡張による並列化
- HPFプログラムにおける計時
- Information

NEC

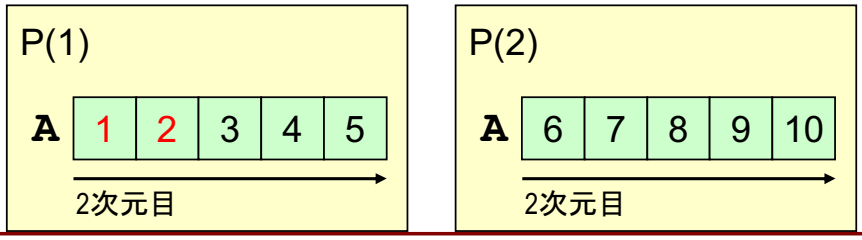
境界処理ループの記述法(1)

```
      real a(10,10)
!hpf$ processors p(2)
!hpf$ distribute (*,block) onto p :: a
      do i=1,10
        a(i,1)=a(i,2)
      enddo
```

```
      real a(10,10)
!hpf$ processors p(2)
!hpf$ distribute (*,block) onto p :: a
!hpf$ on home(a(:,1)), local begin
      do i=1,10
        a(i,1)=a(i,2)
      enddo
!hpf$ endon
```

分散されていない次元は":"を指定

•例えば、2プロセッサで実行する場合、**A(:,1)**を保持するプロセッサだけで実行すると、通信なしで実行可能。(例えば、10プロセッサで実行する場合は通信が必要)
•**ON指示構文とLOCAL節**の指定がないと、**P(2)**も実行に参加し、通信が発生



NEC

境界処理ループの記述法(2)

・マップされた次元の添字が定数の場合、DO変数との対応が取れないため、適切な実行プロセッサが選択できず、全プロセッサで実行するため通信が発生する可能性あり。
・配列のマップされた次元の添字は、定数ではなくDO変数を使って記述する。
•前頁のようにON指示構文とLOCAL節を使って通信なしでの実行を指定しても良い

```
!hpf$ distribute (*,block) :: a,c
      do j=1,n
        if(j .eq. 2)then
          do I=1,n
            a(i,1)=a(i,1)-b(i)*c(i,1)
          enddo
        endif
      enddo
```

```
!hpf$ distribute (*,block) :: a,c
      do j=1,n
        if(j .eq. 2)then
          do I=1,n
            a(i,j-1)=a(i,j-1)-b(i)*c(i,j-1)
          enddo
        endif
      enddo
```


NEC

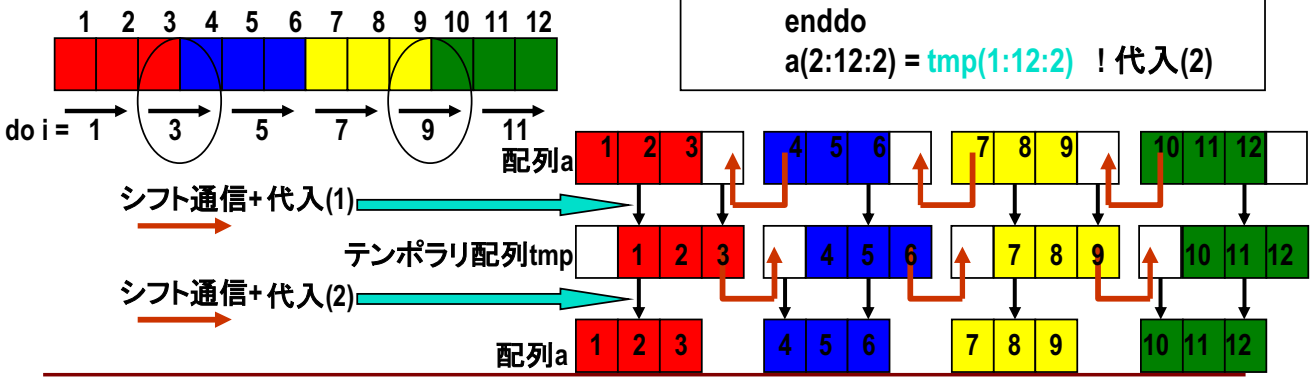
ループ中の配列添字に不一致がある場合(1)

・定義側の添字にずれがある場合、配列全体がテンポラリに置換される

HPFコンパイラによる変形例のイメージ

```
      real a(12)
!HPF$ DISTRIBUTE (BLOCK) onto P :: a
!HPF$ PROCESSORS P(4)
      do i=1,12,2
        a(i) = c/cos(2.*pai/i)
        a(i+1) = a(i)
      enddo
```

```
      real a(12),tmp(12)
!HPF$ DISTRIBUTE (BLOCK) onto P :: a,tmp
!HPF$ PROCESSORS P(4)
      tmp(1:12:2) = a(2:12:2)  ! 代入(1)
      do i=1,12,2
        a(i) = c/cos(2.pai/i)
        tmp(i) = a(i)
      enddo
      a(2:12:2) = tmp(1:12:2)  ! 代入(2)
```



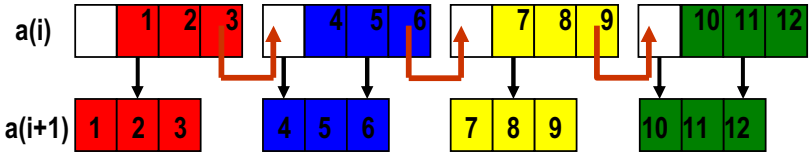
NEC

ループ中の配列添字に不一致がある場合(2)

・ループを分割して、左辺の配列の添字のずれを解消すれば、シフト通信だけで効率の良い並列化が可能

```
      real a(12)
!HPF$ DISTRIBUTE (BLOCK) onto P :: a
!HPF$ PROCESSORS P(4)
!HPF$ SHADOW a(1:0)  ! シャドウの割付け
      do i=1,12,2
        a(i) = c/cos(2.*pai/i)
      enddo
!HPF$  REFLECT a
      do i=1,12,2
!HPF$    ON HOME(a(i+1)), LOCAL  ! 通信なしでシャドウを参照できることを明示する。
        a(i+1) = a(i)
      enddo
```

HPF/SX V2は、左の3つの指示文の追加（SHADOW指示文、REFLECT指示文、ON HOME-LOCAL指示文)なしでも、ループを分割するだけで自動で同等の処理を行う



NEC

同一の手続が複数個所で呼ばれる場合

- 1つの手続呼出しに対する実引数のマッピングが場所により異なる場合。

•サブルーチンchgの仮引数dを分散しないと呼出し(1)で、分散すると呼出し(2)で手続呼出し/戻り時の通信が発生。

```
!hpf$ distribute (*,block) :: a
      real a(n,n),b(n,n)
      call chg(a,n)   !  (1)
      call chg(b,n)   !  (2)
        :
      end

      subroutine chg(d,n)
      real d(n,n)
```

実引数に合わせて仮引数を分散したサブルーチンのコピーを作成し、呼び分ける

手続のクローニング

```
!hpf$ distribute (*,block) :: a
      real a(n,n),b(n,n)
      call chg_dist(a,n)   !  (1)
      call chg(b,n)        !  (2)
        :
      end

      subroutine chg_dist(a,n)
      real a(n,n)
!hpf$ distribute d(*,block)
        :
      end
      subroutine chg(d,n)
      real d(n,n)
        :
      end
```


NEC

同一の作業配列が複数個所で参照される場合

- 1つの作業配列に対応するデータ配列のマッピングが場所により異なる場合。

•作業配列tmpを分散しないと最初のループで、分散すると2つ目のループで通信が発生。

```
!hpf$ distribute (*,block) :: a,b
      real a(n,n),b(n,n),tmp(n)
      do j=1,n  !  (1)
        tmp(j) = a(1,j)
        b(1,j) = tmp(j)
      enddo
!hpf$ on home(a(:,1)),local
      do i=1,n
        tmp(i) = a(i,1)
        b(i,1) = tmp(i)
      enddo
```

データ配列に合わせてマッピングした作業配列のコピーを作成し、使い分ける。(この例ではスカラ変数に書き換えても良い)

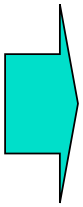
```
!hpf$ distribute (*,block) :: a,b
      real a(n,n),b(n,n),tmp(n)
      real tmpmap(n)
!hpf$ align tmpmap(i) with a(*,i)
      do j=1,n
        tmpmap(j) = a(1,j)
        b(1,j) = tmpmap(j)
      enddo
!hpf$ on home(a(:,1)),local
      do i=1,n
        tmp(i) = a(i,1)
        b(i,1) = tmp(i)
      enddo
```


NEC

同一配列に複数個所でシフト通信が出る場合(1)

- 同一の配列にシフト通信が必要なループが複数あり、シフト対象の配列が更新されない場合、シフト通信は最初の1度だけでよい。
  - •コンパイラによって、自動的にこのような最適化が行われる場合もある。

```
      real,dimension(n,n) :: f,p,q,r
!HPF$ DISTRIBUTE (*,BLOCK) :: f,p,q,r
      do j=2,n-1
        do i=1,n
          r(i,j)=f(i,j)- (p(i,j-1)+p(i,j+1))
        enddo
      enddo
      do iter=1,1000 ! 時間発展loop
          :
        do j=1,n
          do i = 1,n
            q(i,j) = f(i,j)–(p(i,j-1)+p(i,j+1))
          enddo
        enddo
          :
      enddo
```



```
      real,dimension(n,n) :: f,p,q,r
!HPF$ DISTRIBUTE (*,BLOCK) :: f,p,q,r
!HPF$ SHADOW p(0,1)
!HPF$ REFLECT p  ！シフト通信
      do j=2,n-1
!HPF$   ON HOME(r(:,j)), LOCAL(p)
        do i=1,n
          r(i,j)=f(i,j)- (p(i,j-1)+p(i,j+1))
        enddo
      enddo
      do iter=1,1000 ! 時間発展loop
          :
        do j=1,n
!HPF$      ON HOME(r(:,j)), LOCAL(p)
          do i = 1,n
            q(i,j) = f(i,j)–(p(i,j-1)+p(i,j+1))
          enddo
        enddo
          :
      enddo
```


NEC

同一配列に複数個所でシフト通信が出る場合(2)

- 複数の手続で、同じ配列に対してシフト通信が出る場合

シフト通信の対象配列が、複数の手続呼出し間で定義されていなければ、呼出側にまとめることが可能

```
      common /com/a(100),b(100),c(100)
!hpf$ distribute (block) :: a,b,c
!hpf$ shadow b(1)

      do i=1,iter ！時間発展ループ
          :
        call sub1()
          :              ! bの定義なし
        call sub2()
          :
        do j=1,n
          b(j) = t * b(j) + c(j)
        enddo
      enddo
```

```
       subroutine sub1()
      common /com/a(100),b(100),c(100)
!hpf$ distribute (block) :: a,b,c
!hpf$ shadow b(1)
!hpf$ reflect b
      do i=2,N
!hpf$   on home(a(i)), local(b)
        a(i) = t * (b(i) + b(i-1))
      enddo
          :
      end
      subroutine sub2()
      common /com/a(100),b(100),c(100)
!hpf$ distribute (block) :: a,b,c
!hpf$ shadow b(1)
!hpf$ reflect b
      do i=1,N-1
!hpf$   on home(a(i)), local(b)
        a(i) = t *( b(i) + b(i+1))
      enddo
          :
      end
```


NEC

同一配列に複数個所でシフト通信が出る場合(3)

- 呼出側にまとめると高速化可能
- 時間発展ループ中でbが更新されなければ、時間発展ループの前に出せば、より高速化が可能

```
      common /com/a(100),b(100),c(100)
!hpf$ distribute (block) :: a,b,c
!hpf$ shadow b(1)

      do i=1,iter   ! 時間発展ループ
        :
!hpf$   reflect b
        call sub1()
        :                    ! bの定義なし
        call sub2()
        :
        do j=1,n
          b(j) = t * b(j) + c(j)   ! bの定義
        enddo
      enddo
```

```
      subroutine sub1()
      common /com/a(100),b(100),c(100)
!hpf$ distribute (block) :: a,b,c
!hpf$ shadow b(1)

      do i=2,N
!hpf$   on home(a(i)), local(b)
        a(i) = t * (b(i) + b(i-1))
      enddo
        :
      end
      subroutine sub2()
      common /com/a(100),b(100),c(100)
!hpf$ distribute (block) :: a,b,c
!hpf$ shadow b(1)

      do i=1,N-1
!hpf$   on home(a(i)), local(b)
        a(i) = t *( b(i) + b(i+1))
      enddo
        :
      end
```


NEC

同一パタンの通信が複数の配列に発生する場合(1)

HPFコンパイラによる変形例のイメージ

•3次元目でBLOCK分散された配列から、非分散のテンポラリに複数の配列がコピーされる例

```
!hpf$ distribute (*,*,block) :: x,y,z,cx,cy,cz
      real,dimension(n,n,n) :: x,y,z
      real,dimension(n,n,n) :: cx,cy,cz
  do k=1,n
    do j=1,n
      do i=1,n
        ix = idxx(i,j,k)
        iy = idxy(i,j,k)
        iz = idxz(i,j,k)
        cx(i,j,k) = x(ix,iy,iz)
        cy(i,j,k) = y(ix,iy,iz)
        cz(i,j,k) = z(ix,iy,iz)
      enddo
    enddo
  enddo
```

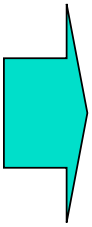
```
!hpf$ distribute (*,*,block) :: x,y,z,cx,cy,cz
      real,dimension(n,n,n) :: x,y,z
      real,dimension(n,n,n) :: cx,cy,cz
      real,allocatable,dimension(:,:,:) :: tx,ty,tz
      allocate(tx(n,n,n),ty(n,n,n),tz(n,n,n))
      tx = x; ty = y; tz = z ！ 通信(3回)
  do k=1,n
    do j=1,n
      do i=1,n
        ix = idxx(i,j,k)
        iy = idxy(i,j,k)
        iz = idxz(i,j,k)
        cx(i,j,k) = tx(ix,iy,iz)
        cy(i,j,k) = ty(ix,iy,iz)
        cz(i,j,k) = tz(ix,iy,iz)
      enddo
    enddo
  enddo
```


NEC

同一パタンの通信が複数の配列に発生する場合(2)

・通信対象の配列群を、分散した作業配列にコピーすることで、通信回数を減らす。
・効果は、コピーする配列数、実行プロセッサ数、通信パタン、通信性能等に依存する。
・一般に配列数、実行プロセッサ数が多いほど、効果は大きくなる。

```
!hpf$ distribute (*,*,block) :: x,y,z,cx,cy,cz
      real,dimension(n,n,n) :: x,y,z
      real,dimension(n,n,n) :: cx,cy,cz
  do k=1,n
    do j=1,n
      do i=1,n
        ix = idxx(i,j,k)
        iy = idxy(i,j,k)
        iz = idxz(i,j,k)
        cx(i,j,k) = x(ix,iy,iz)
        cy(i,j,k) = y(ix,iy,iz)
        cz(i,j,k) = z(ix,iy,iz)
      enddo
    enddo
  enddo
```



```
!hpf$ distribute (*,*,block) :: x,y,z,cx,cy,cz
      real,dimension(n,n,n) :: x,y,z
      real,dimension(n,n,n) :: cx,cy,cz
      real, dimension(n,n,3,n) :: tmp,tmpcpy
!hpf$ align (*,*,*,i)  with x(*,*,i) :: tmp
      tmp(:,:,1,:)=x; tmp(:,:,2,:)=y; tmp(:,:,3,:)=z
      tmpcpy = tmp ！ 通信(1回)
  do k=1,n
    do j=1,n
      do i=1,n
        ix = idxx(i,j,k)
        iy = idxy(i,j,k)
        iz = idxz(i,j,k)
        cx(i,j,k) = tmpcpy(ix,iy,1,iz)
        cy(i,j,k) = tmpcpy(ix,iy,2,iz)
        cz(i,j,k) = tmpcpy(ix,iy,3,iz)
      enddo
    enddo
  enddo
```


NEC

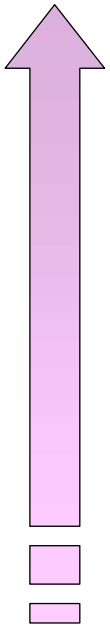
内容

- コードのクリーンナップ
- HPFによる並列化の基本手順
- 配列宣言とデータマッピング
- 不均等なデータマッピングの利用
- 動的なマッピングの変更
- HPFプログラム高速化手法
- アプリケーションの並列化
- HPFプログラムにおける計時
- Information

NEC

種々の応用へのHPFの適応度

容易

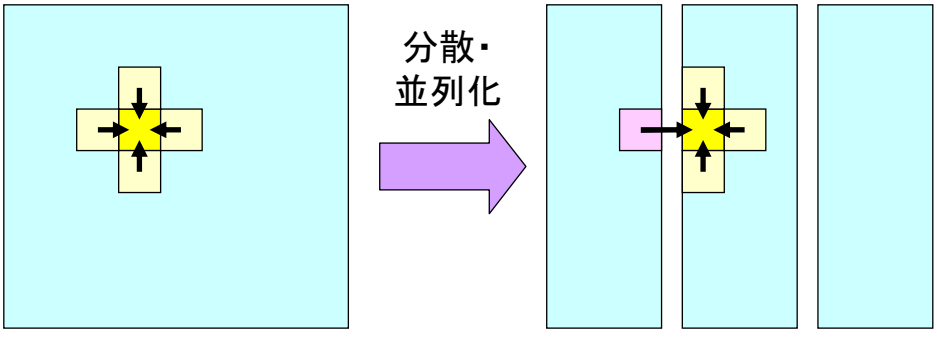


- 規則的／データアクセスの局所性が高い問題
  - 例）偏微分方程式の陽解法
- 規則的／データアクセスの局所性が低い問題
  - 例）多次元FFT、ADI法
- 不規則／データアクセスの局所性が高い問題
  - 例）有限要素法
- 不規則／データアクセスの局所性が低い問題
  - 例）粒子系の問題

NEC

規則的／データアクセスの局所性が高い問題ー◎

- 偏微分方程式の陽解法



ある格子点の次の値は
近傍の値のみから計算される

切断面のデータのみ通信すればよい
（通信はHPFコンパイラが自動で挿入）

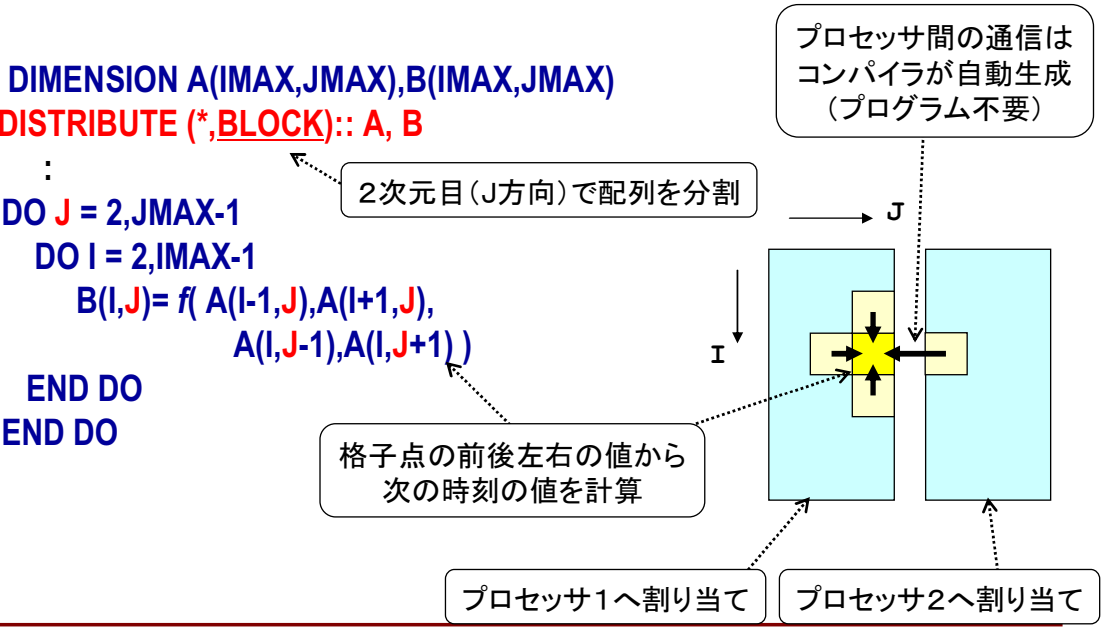
- この種の問題は、HPFにより容易にMPIと遜色ない性能が出せる。
- 通信量が少ないため、並列性能(台数効果)も出やすい。

NEC

HPFプログラムの例

■ 偏微分方程式の陽解法（2次元の例）

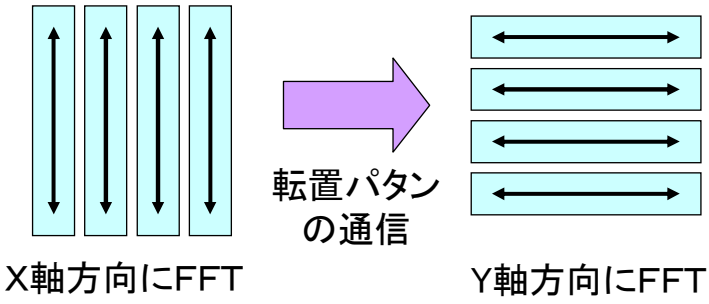
```
      DIMENSION A(IMAX,JMAX),B(IMAX,JMAX)
!HPF$ DISTRIBUTE (*,BLOCK):: A, B
        :
      DO J = 2,JMAX-1
        DO I = 2,IMAX-1
          B(I,J)= f( A(I-1,J),A(I+1,J),
                     A(I,J-1),A(I,J+1) )
        END DO
      END DO
```



NEC

規則的／データアクセスの局所性が低い問題ー○

■ 多次元FFT



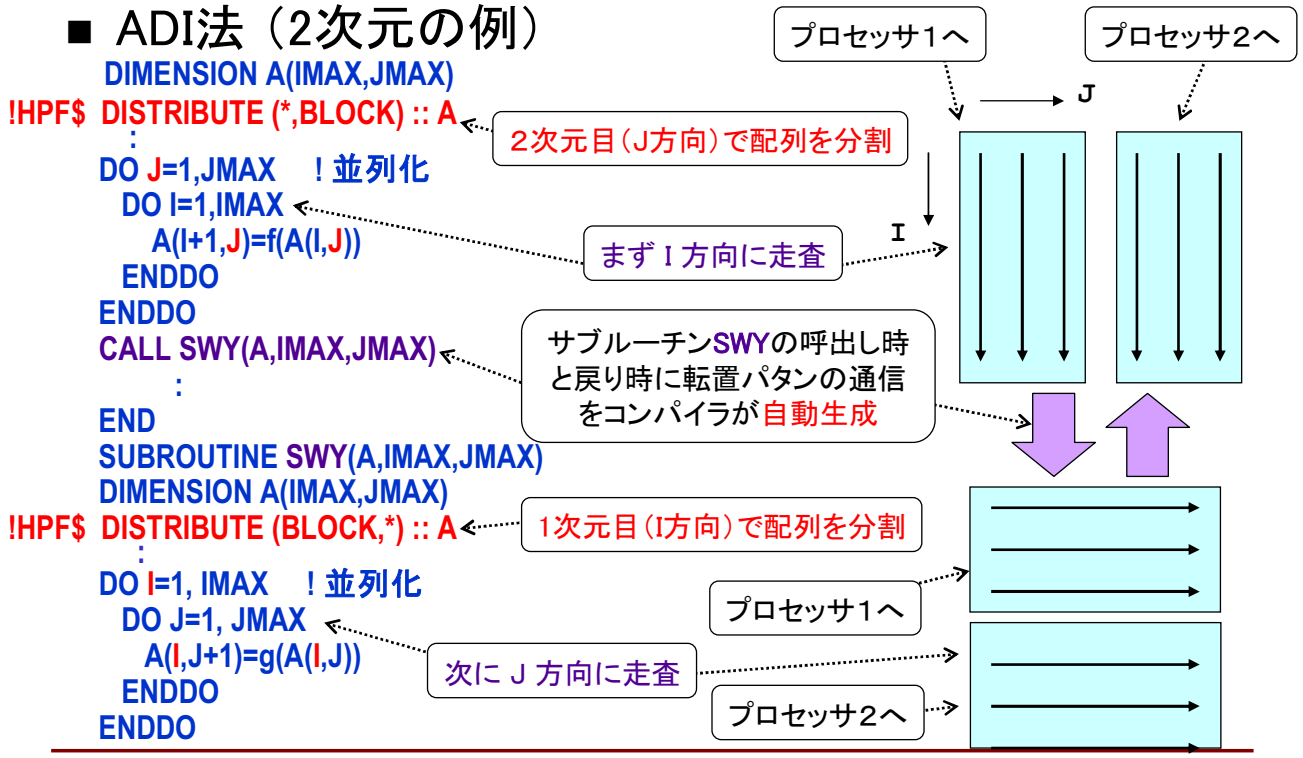
- ◆ この種の問題は、HPFならばMPIと比べてはるかに記述が容易で、MPIと遜色ない性能が出せる。
- ◆ 転置パタンの通信はややコストが大きいので、この回数をいかに減らせるかが並列性能向上のポイント（MPIでも同じ）。

NEC

HPFプログラムの例

■ ADI法（2次元の例）

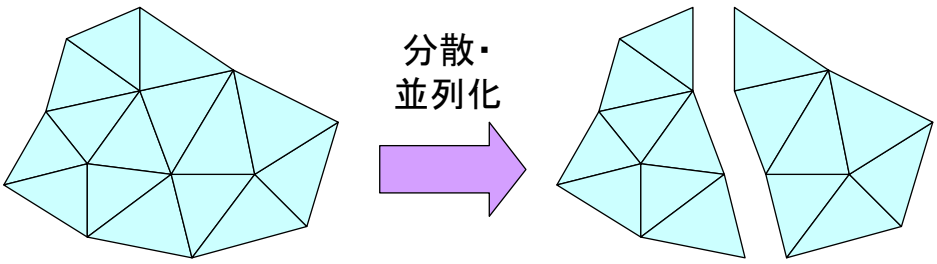
```
      DIMENSION A(IMAX,JMAX)
!HPF$ DISTRIBUTE (*,BLOCK) :: A
        :
      DO J=1,JMAX    ! 並列化
        DO I=1,IMAX
          A(I+1,J)=f(A(I,J))
        ENDDO
      ENDDO
      CALL SWY(A,IMAX,JMAX)
        :
      END
      SUBROUTINE SWY(A,IMAX,JMAX)
      DIMENSION A(IMAX,JMAX)
!HPF$ DISTRIBUTE (BLOCK,*) :: A
        :
      DO I=1, IMAX    ! 並列化
        DO J=1, JMAX
          A(I,J+1)=g(A(I,J))
        ENDDO
      ENDDO
```



NEC

不規則／データアクセスの局所性が高い問題－△

■ 有限要素法



◆ この種の問題は、並列性能を出すためには前処理が必要（うまく分割できるようなデータの並べ替え、分割境界部分の指定等）。

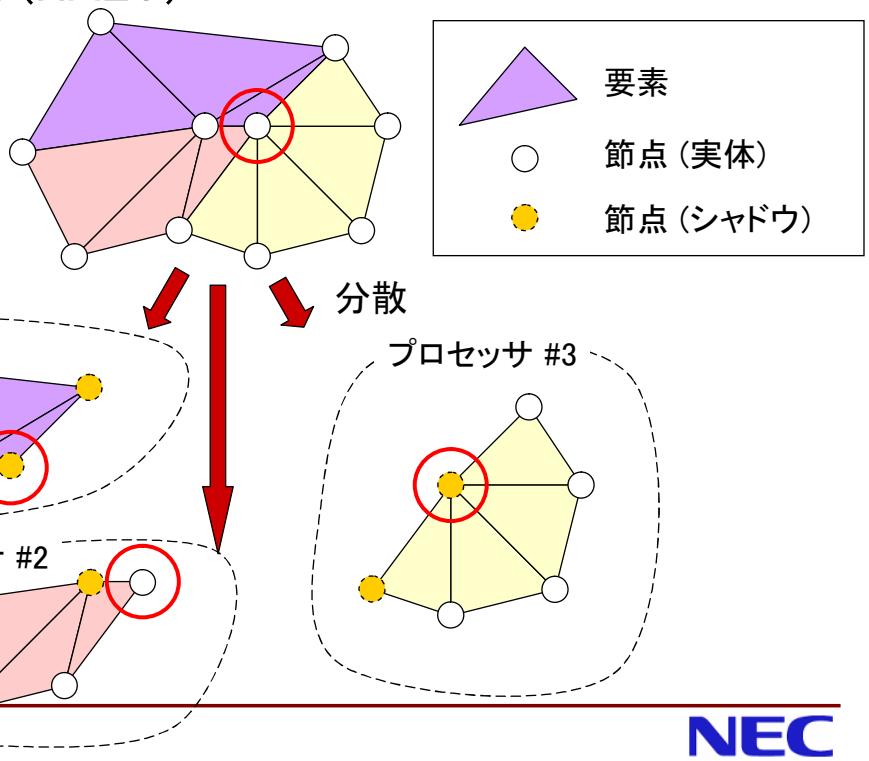
◆ MPIと比べて記述は容易だが、「既存のFortranプログラムに指示文を挿入」しただけでうまくいくというわけではない。

NEC

不規則問題への対応例：HPF/SX V2独自拡張機能

■ 不規則シャドウ(HALO)

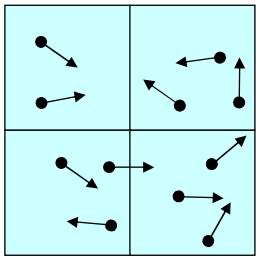
分散境界上の節点は、
- 一つのプロセッサに実体として、
- 他のプロセッサにシャドウ(コピー)として、割り付ける。



NEC

不規則／データアクセスの局所性が低い問題－△

■ 粒子系の問題

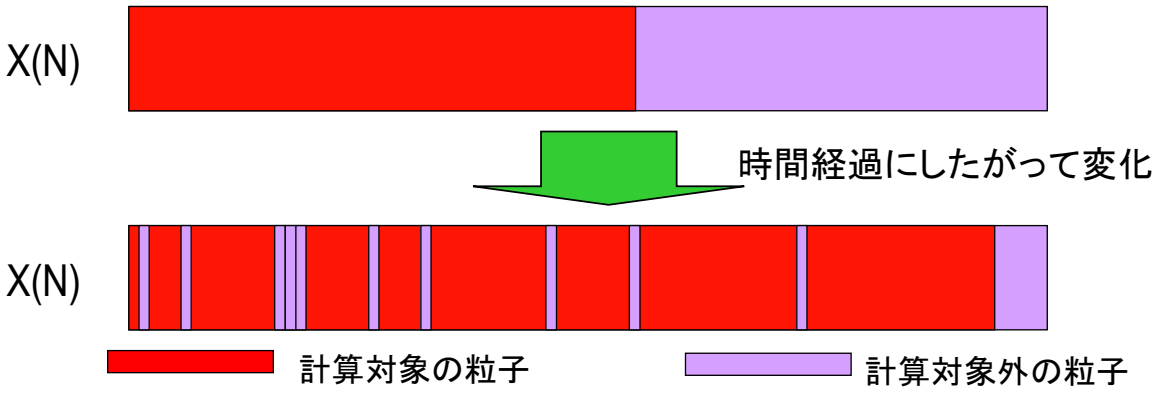


各プロセッサへの
場と粒子の割り当て方を整合させれば、
アクセスの局所性を高められる。

- この種の問題は、並列性能を出すためには粒子データの局所性を高めるための工夫が必要(MPIでも同じ)。場のデータに比べて粒子データ数が圧倒的に多い場合、粒子データだけ分散し、場のデータは分散しない方法もある。
- MPIと比べて記述は容易だが、「既存のFortranプログラムに指示文を挿入」しただけでは必ずしもうまくいくわけではない。

NEC

次元拡張による並列化(1)



•一次元配列上で行われる右のような処理は、効率よく分散並列化するのは困難。

粒子のパッキング

```
do i=1,n
    if(...)then
        k=k+1
        x(k) = x(i)
```

粒子の入れ替え

```
do i=1,n
    k = ...
    tmp = x(i)
    x(i) = x(k)
    x(k) = tmp
```


NEC

次元拡張による並列化(2)

Step1
Fortran

コードのクリーンナップ

Step2
Fortran

並列化のための次元拡張

real*8 x(M)

real*8 x(M/P,P)
P:分散並列*共有並列

このステップが必要

Step3
HPF

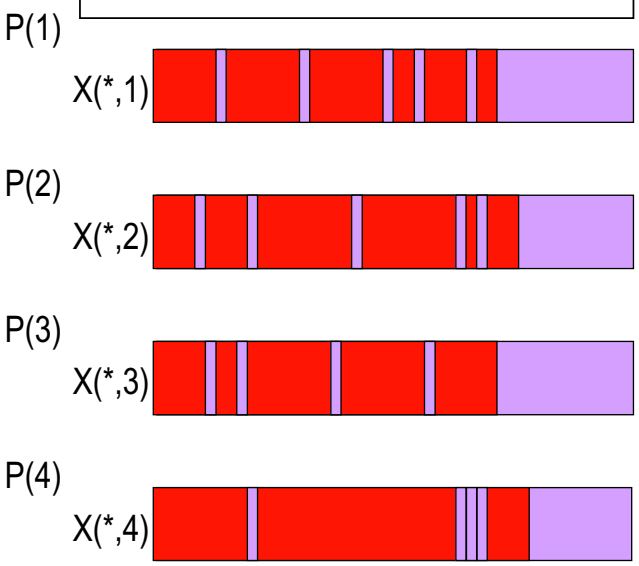
HPF指示文の挿入による分散並列化

!HPF$ distribute x(*,block)

NEC

次元拡張による並列化(3)

```
      double precision X(N/4, 4)
!hpf$ distribute (*,block) :: X
```



- 粒子データに、並列用の次元を付加し、増やした次元で分割する。
- 場のデータはマップせず、各プロセッサがそれぞれ全体を持つ
- パッキングやソートは、極力各プロセッサ内だけで行い、通信の発生を避ける。

```
do ip = 1,4 ！プロセッサ
  do i=1,n    ！粒子
    if(...)then
      k(ip)=k(ip)+1
      x(k(ip),ip) = x(i,ip)
```

- 粒子数が不均等になって、ロードバランスが取れない場合は、粒子数を均等化する。(この際通信が発生)

計算対象の粒子

計算対象外の粒子

NEC

内容

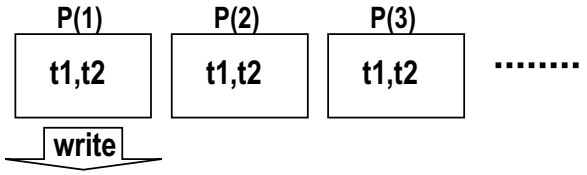
- コードのクリーンナップ
- HPFによる並列化の基本手順
- 配列宣言とデータマッピング
- 不均等なデータマッピングの利用
- 動的なマッピングの変更
- HPFプログラム高速化手法
- アプリケーションの並列化
- HPFプログラムにおける計時
- Information

NEC

HPFにおける計時(1)

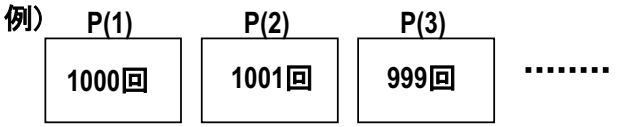
·通常の計時手続は、各プロセッサ毎独立に値を返す。同期はとられない。
·以下の例のwrite文では、1つのプロセッサの値のみが出力される。

```
double precision t1,t2
call etime(t1)
 call jacobi(nn,gosa)
call etime(t2)
write(*,*)t2-t1
```



·姫野ベンチマークプログラムの場合、3回の試行にかかる実行時間によって、本番の実行回数を決める。
  ·プロセッサ毎に、微妙に実行時間は異なるため、実行回数がプロセッサ毎に異なる可能性がある。
    ·デッドロック等が発生する可能性あり。

```
double precision t1,t2
call etime(t1)
 call jacobi(nn,gosa)
call etime(t2)
実行回数 = f / (t2-t1)
```

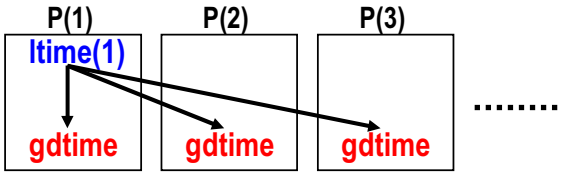


NEC

HPFにおける計時(2)

·HPF版姫野ベンチマークでは、以下のようにしてこの問題を回避している。

```
      real function gdtime(time)
      real ltime(1),time(2),gdtime
!hpf$ distribute (block) :: ltime
       ltime(1) = dtime(time)
       gdtime = ltime(1)
       end
```



·HPF専用の計時手続HPF_WCLOCK()も使用可能(HPF/JA拡張)。
  ·引数は倍精度実数型スカラ変数
  ·計時直前に同期が取られるため、ロードインバランスがある場合、一番遅いプロセッサに相当する時間が測定される。
  ·(通信により)自動的に値の一致が保障される。

```
double precision t1,t2
call hpf_wclock(t1)
 call jacobi(nn,gosa)
call hpf_wclock(t2)
実行回数 = f / (t2-t1)
```

HPFコンパイラによる変形例

```
バリア
計時
call broadcast(t1)
 call jacobi(nn,gosa)
バリア
計時
call broadcast(t2)
```


NEC

内容

- コードのクリーンナップ
- HPFによる並列化の基本手順
- 配列宣言とデータマッピング
- 不均等なデータマッピングの利用
- 動的なマッピングの変更
- HPFプログラム高速化手法
- アプリケーションの並列化
- HPFプログラムにおける計時
- Information

NEC

Information

- HPF推進協議会 (HPFPC)
  - HPFによる並列計算の普及促進を目的とし、HPCユーザとベンダーが参加
  - HPF講習会開催、HPFプログラミングガイドの公開、サンプルコードの公開
  - HPF2.0仕様書の和訳、HPF/JA拡張仕様の策定
- HPFPCのホームページ：http://www.hpfpc.org/

  現在、以下のようなドキュメントや**フリーのHPFコンパイラ**がダウンロード可能。
  - High Performance Fortranで並列計算を始めよう
  - HPFプログラミングガイド
  - HPF2.0仕様書(日本語訳)、HPF/JA仕様書
  - 過去の講習会資料
  - サンプルコード
  - 参加(会費無料)を希望される方はhttp://www.hpfpc.org/nyuukai.htmlの

  フォーマットで、apply@hpfpc.orgまでE-mailをお送りください。
- 核融合科学研究所のQ&A
  - http://www.dss.nifs.ac.jp/workgr/QandA/QandA-f90hpf.html

NEC