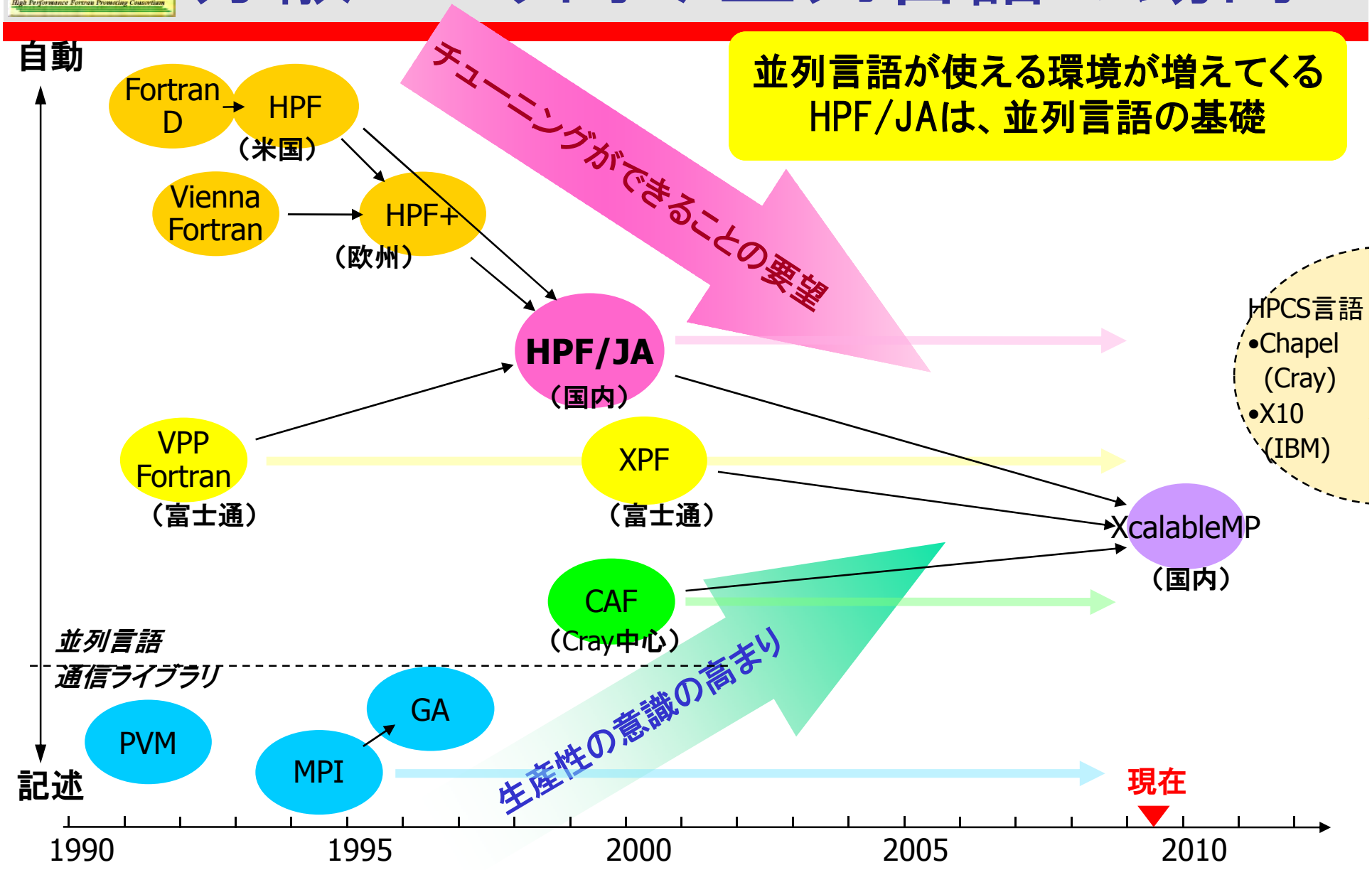
FUJITSU THE POSSIBILITIES ARE INFINITE

HPF(High Performance Fortran)講習会
入門編その1

岩下 英俊
富士通(株)
次世代テクニカルコンピューティング開発本部
2009/07/16-17

分散メモリ向け並列言語の動向



その1の内容

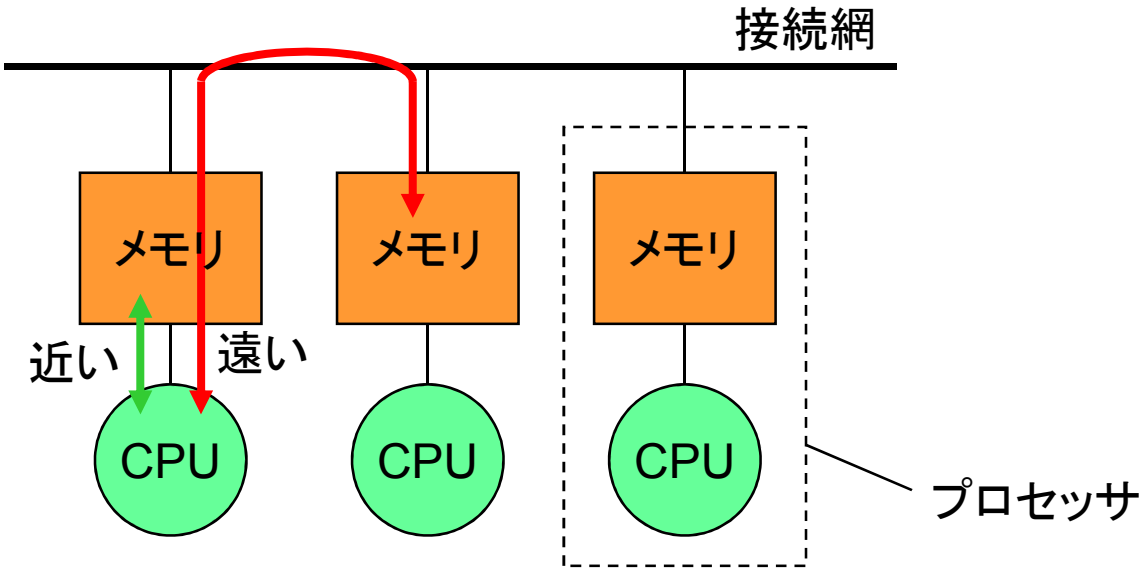
- HPF言語の考え方(3.1)
  - 想定するハードウェア
  - プログラムの並列化とは
  - HPFが提供する「仮想」
  - HPFによるプログラム並列化
- データ分散とループの並列化(3.2)
  - 指示文の書式
  - データの分散(DISTRIBUTE指示文)
  - プロセッサの宣言(PROCESSORS指示文)
  - ループの並列化(INDEPENDENT指示文)
  - ループ処理の分担(ON指示文)
  - 自動か、書くか
- 典型的な通信最適化(6.1.2)
  - SHADOW指示文＋REFLECT指示文＋LOCAL節

これだけで
HPFが書けます

これだけで
性能が出ます

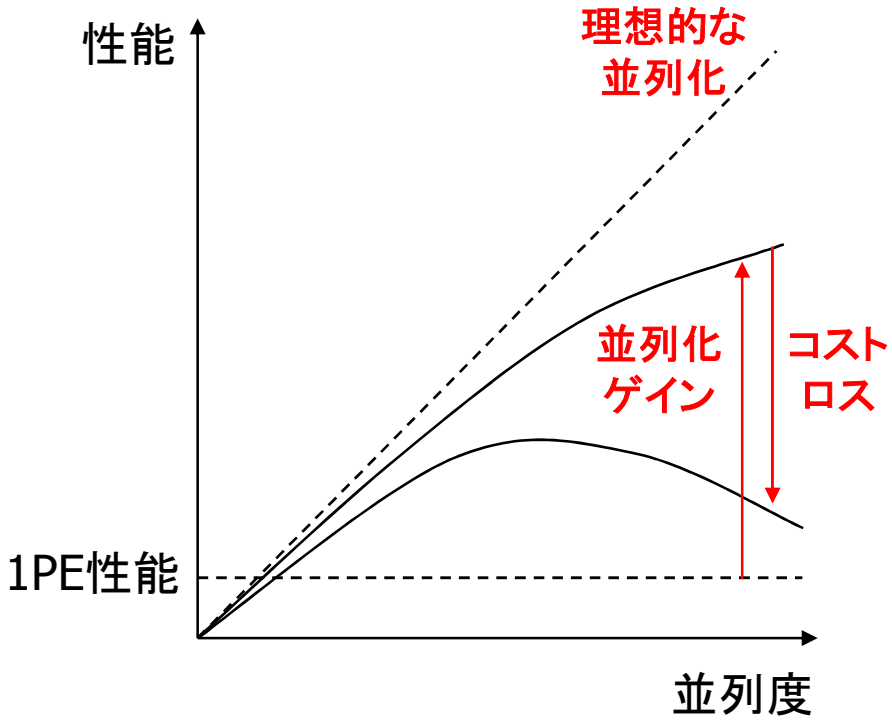
HPFのターゲット

- 「分散メモリ型計算機」
  - 自プロセッサのメモリアクセスは、通常のload/store。
  - 他プロセッサのメモリアクセスは、通信が伴う。
    - 数百～数千倍遅い



分散メモリ型計算機で性能を出す

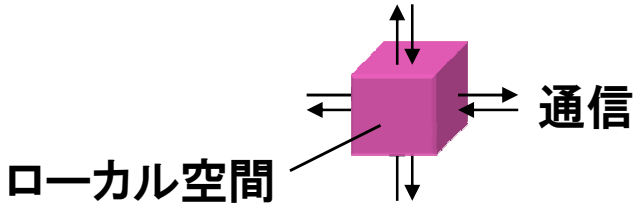
- 並列化率のゲインと、オーバヘッドのロス
- 性能を出すためには:
  - 計算をできるだけ並列化する
    - ループの並列化
    - タスク並列、手続間並列、・・・
  - 通信コストを下げる
    - データの分割配置
    - 通信不要の判断
    - データの一時的移動・変形
    - 通信量・通信回数の削減
    - 通信パターンの最適化



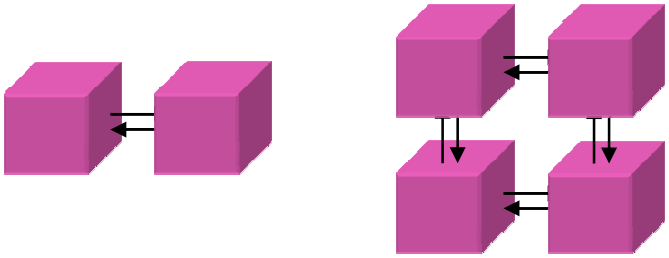
並列化のスタイル2つ

- SPMDモデル
  （Local View）
  - MPIのスタイル

各プロセッサの実行をプログラム.



それをN個同時に実行する.

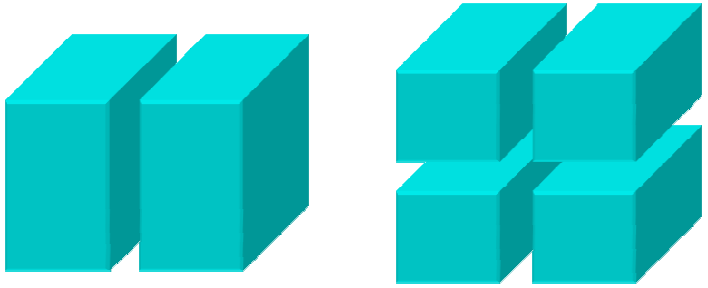


2並列　　4並列

- グローバルモデル
  （Global View）
  - HPFのスタイル

全体の実行をプログラム.

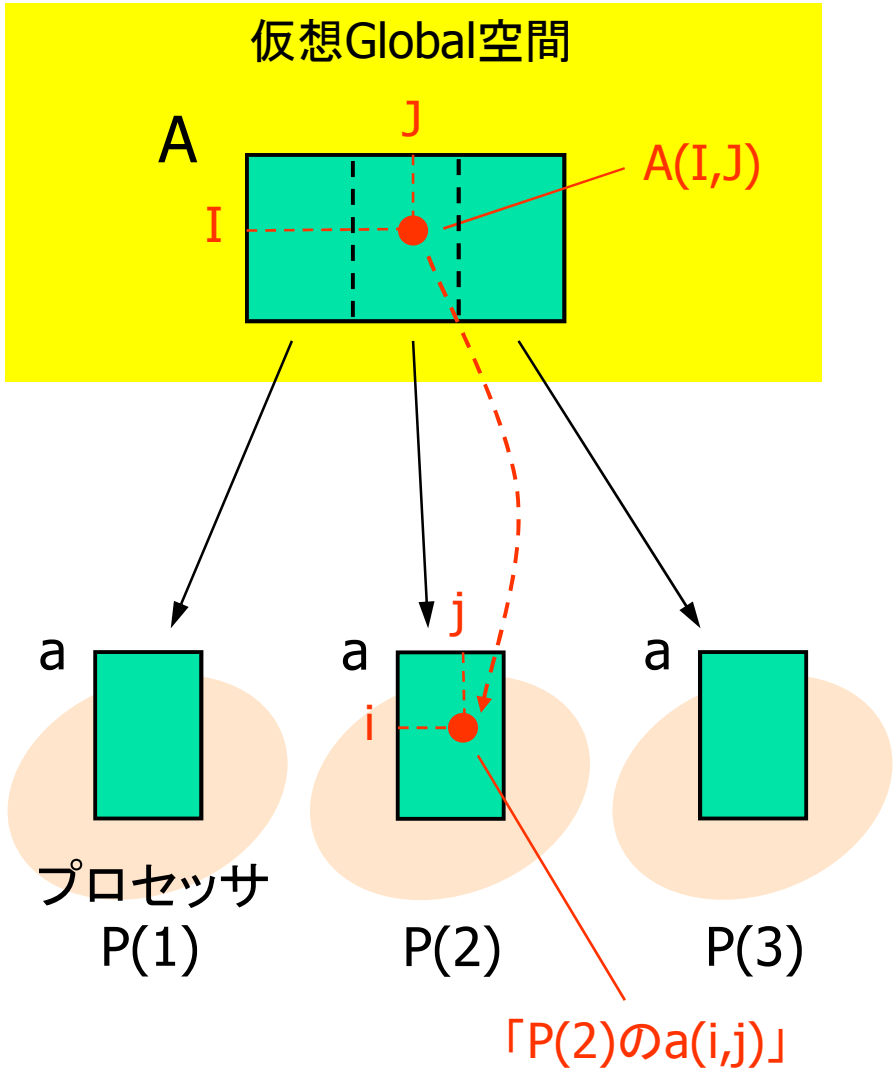
N個に分割されて並列実行.



2並列　　4並列

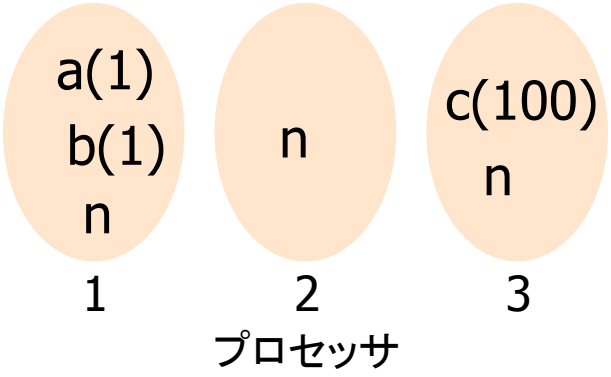
HPFが提供する「仮想」(1/2)

- グローバル空間
  - 分散メモリが、一つの巨大なメモリに見える。
  - 利用者は、配列の属性として分散を指示。



HPFが提供する「仮想」(2/2)

- 単一スレッド
  - 個々のプロセッサの動作を気にするのでなく、一筋の実行列を記述する。



プログラム

```
n=100
a(1)=0.0
b(1)=c(i)+1.0
```

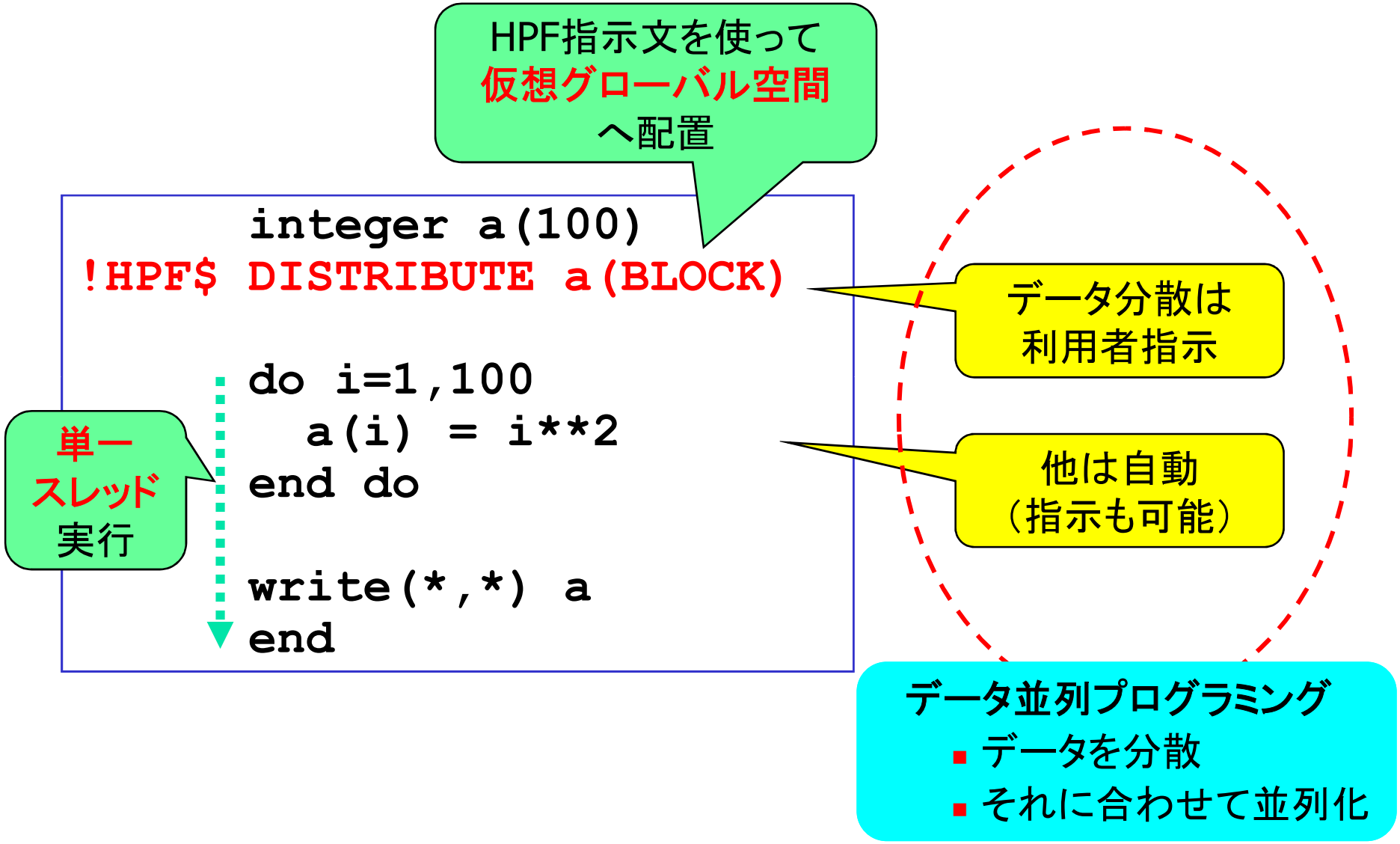
プロセッサ

1 2 3

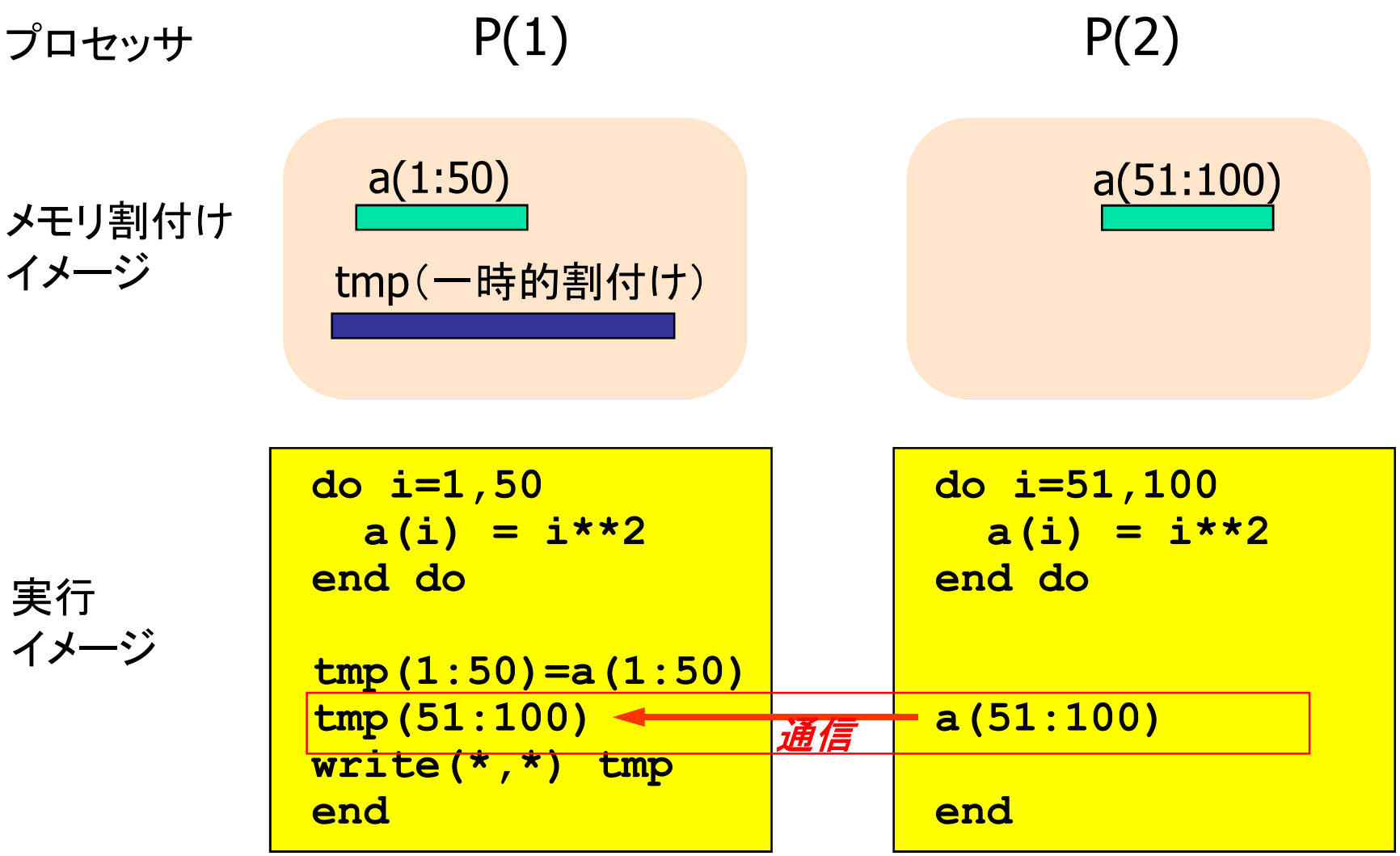
実行イメージ

```
n=100
if (プロセッサ=1) a(1)=0.0
if (プロセッサ=3) c(i)を1へ送信
if (プロセッサ=1) then
  3からのデータをtmpに受信
  b(1)=tmp+1.0
endif
```

HPFプログラム、最初の例



実行イメージ(2並列の実現例)



HPFによるプログラム並列化

- 並列プログラムの開発手順(ざっくり)

(1) 逐次(Fortran)プログラムで実行確認

- 最初は逐次で動作する問題規模で。

(2) HPF指示文を加え、翻訳・実行確認

- 逐次実行と同じ結果ならOK。

(3) 性能が不十分なら、(2)を繰り返す。

- 自動でうまくいかなかった部分に指示を追加。

その1の内容

- HPF言語の考え方(3.1)
  - 想定するハードウェア
  - プログラムの並列化とは
  - HPFが提供する「仮想」
  - HPFによるプログラム並列化

- データ分散とループの並列化(3.2)
  - 指示文の書式
  - データの分散(DISTRIBUTE指示文)
  - プロセッサの宣言(PROCESSORS指示文)
  - ループの並列化(INDEPENDENT指示文)
  - ループ処理の分担(ON指示文)
  - 自動か、書くか

これだけで
HPFが書けます

- 典型的な通信最適化(6.1.2)
  - SHADOW指示文＋REFLECT指示文＋LOCAL節

これだけで
性能が出ます

HPF指示文の書式

- 接頭辞＋指示文名[＋･･･]

```
!HPF$ DISTRIBUTE a(BLOCK)
```

  - Fortranコンパイラでは、コメント行と見なされる

- 継続行のルール
  - 自由形式(free source form)

```
!HPF$ DISTRIBUTE    &
!HPF$    a(BLOCK)
```

    - `&'で終わると、次の行に継続
  - 固定形式(fixed source form)

```
!HPF$ DISTRIBUTE
!HPF$*    a(BLOCK)
```

    - `!HPF$'の直後(6桁目)に文字があると、前の行から継続

HPFプログラム、最初の例(再掲)

```
1          integer a(100)
2  !HPF$ DISTRIBUTE a(BLOCK)
3
4          do i=1,100
5            a(i) = i**2
6          end do
7
8          write(*,*) a
9          end
```

DISTRIBUTE指示文:
配列データをプロセッサに
分散する方法を指示。

- 分散種別: block
- プロセッサ数: 無指定

DISTRIBUTE指示文(プロセッサ数可変)

データ分散の宣言(プロセッサ数可変のとき)

!HPF$ **DISTRIBUTE *a*(*〈分散形式〉*, ...)**

または

!HPF$ **DISTRIBUTE (*〈分散形式〉*, ...) :: *a, b, ...***

a や ***b*** は配列変数.

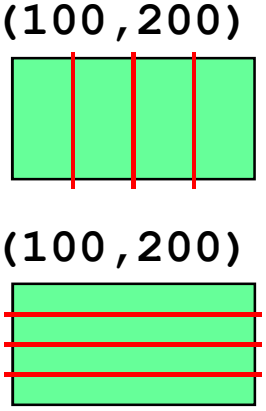
〈分散形式〉 は, 分散次元は **BLOCK CYCLIC** など

分散しない次元は *

【例】

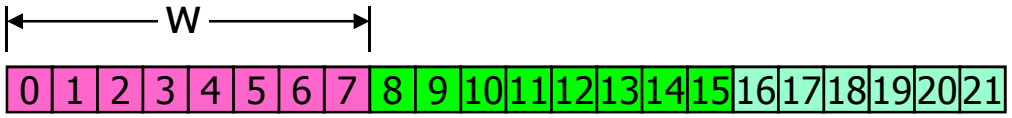
```
      real a(100,200),b(100,200)
!HPF$ DISTRIBUTE a(*,BLOCK)
!HPF$ DISTRIBUTE b(BLOCK,*)
      ingeger,dimension(10,50,5) :: c,d
!HPF$ DISTRIBUTE (*,CYCLIC,*) :: c,d
```

4プロセッサ使用のとき



分散形式(1/2)

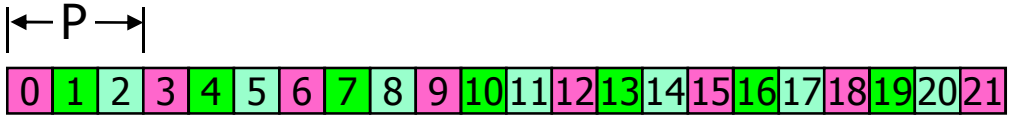
- DISTRIBUTE a(BLOCK)



w = ceil (N / P)
N: 配列のサイズ
P: プロセッサ数

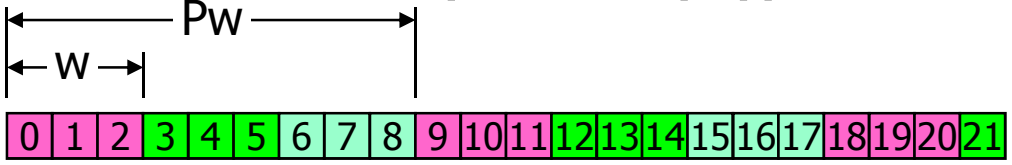
  - 近傍要素の参照が多い場合（差分法など）

- DISTRIBUTE a(CYCLIC)



  - 計算負荷にばらつきがあっても、ほぼ均等に配分

- DISTRIBUTE a(CYCLIC(w))：block-cyclic分散

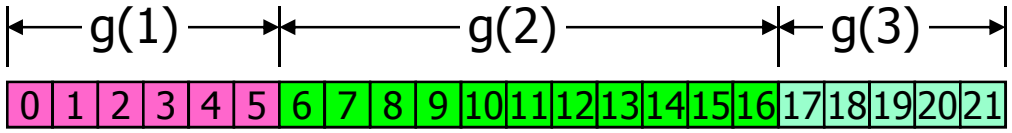


w: 指定ブロック幅

  - 上二者の性質がある場合の中間的な手段

分散形式(2/2)

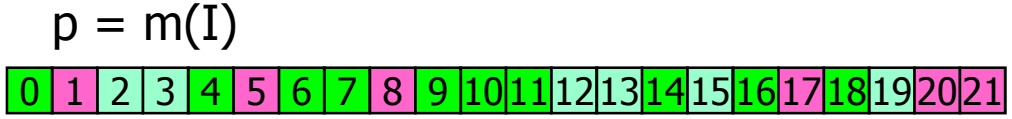
- !hpf$ distribute a(GEN_BLOCK(g))：不均等block



g: マッピング配列
g(k): 第kプロセッサが担当する大きさ

  - 負荷の偏りが事前に分かっている場合（三角行列など）

- !hpf$ distriubte a(INDIRECT(m))：不規則分散



m: マッピング配列
p: プロセッサ番号

  - データとプロセッサの対応が不規則な場合
  - 長方形・直方体にマップできない領域（非構造格子など）

プロセッサ数指定の有無

```
      integer a(100)
!HPF$ DISTRIBUTE a(BLOCK)

      do i=1,100
        a(i) = i**2
      end do

      write(*,*) a
      end
```

分散種別：block
プロセッサ数：無指定

- プロセッサ数可変
  - 実行開始時に指定
- 実行開始時にサイズ確定

```
      integer a(100)
!HPF$ PROCESSORS p(2)
!HPF$ DISTRIBUTE a(BLOCK) ONTO p

      do i=1,100
        a(i) = i**2
      end do

      write(*,*) a
      end
```

分散種別：block
プロセッサ数：2

- プロセッサ数固定
  - プロセッサ数変更時は、再翻訳が必要
- コンパイル時にサイズ確定

PROCESSORS指示文と DISTRIBUTE指示文(プロセッサ指定)

プロセッサの名前と形状の宣言

!HPF$ **PROCESSORS** ***p*(*n*$_1$, ..., *n*$_N$), ...**

p はプロセッサ配列名.

n*$_1$** × **...** × ***n*$_N$** はプロセッサ数で, ***Nは分散次元の数.

データ分散の宣言(プロセッサを指定するとき)

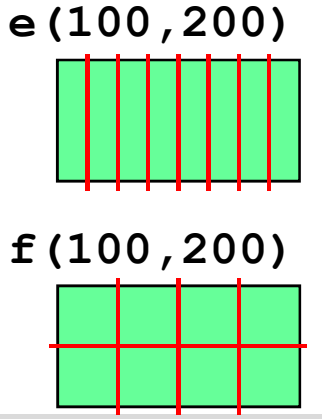
!HPF$ **DISTRIBUTE *a*(**〈*分散形式*〉**, ...) ONTO *p***

または

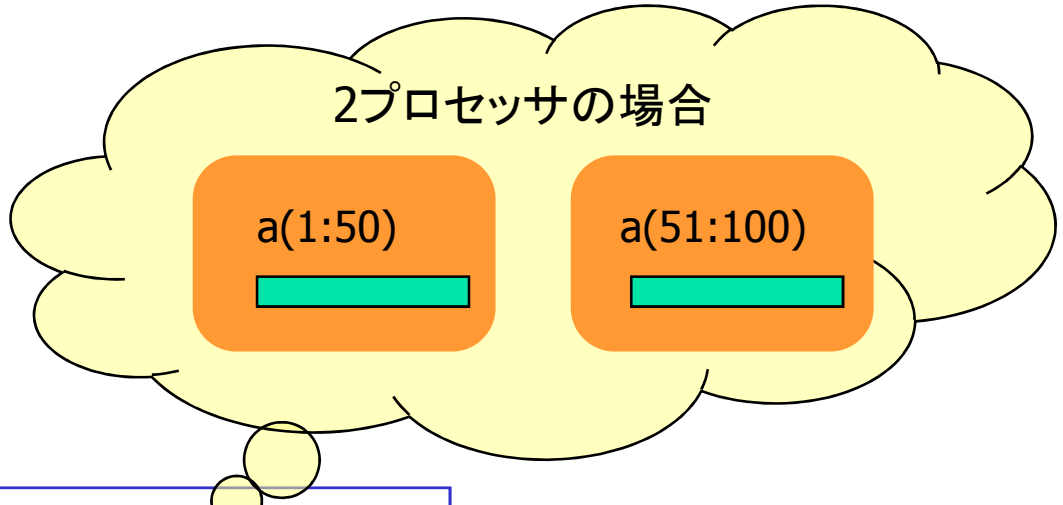
!HPF$ **DISTRIBUTE (**〈*分散形式*〉**, ...) ONTO *p* :: *a, b, ...***

【例】

```
!HPF$ PROCESSORS proc(8),p2(2,4)
      real e(100,200),f(100,200)
!HPF$ DISTRIBUTE e(*,BLOCK) ONTO proc
!HPF$ DISTRIBUTE f(BLOCK,BLOCK) ONTO p2
```



ループの並列化




```
1         integer a(100)
2  !HPF$  DISTRIBUTE a(BLOCK)
3
4         do i=1,100
5           a(i) = i**2
6         end do
7
8         write(*,*) a
9         end
```

ループ並列性：無指定
処理分担：無指定

- 並列化可能の判定
- 処理分担の決定

並列化可能の判定：自動 or 指示

- 自動に任せる場合
  - コンパイラがデータ依存解析して決める。
  - 並列化可能でも並列化されないことがある。
    - コンパイラの能力不足の場合
    - コンパイラでは論理的に解析できない場合
- 並列性を指示する場合
  - INDEPENDENT指示文
    - 並列化可能であることをコンパイラに教える。
  - 正しい記述は利用者の責任
    - 並列化できないループに指定すると、動作保証できない。

並列性を指示する場合：INDEPENDENT指示文

ループの並列性の指示　・・・並列化するDOループの直前に

!HPF$ **INDEPENDENT [, *〈節〉*] ...**

〈節〉 はNEW節またはREDUCTION節（後述）.

```
1         integer a(100)
2  !HPF$  DISTRIBUTE a(BLOCK)
3
4  !HPF$  INDEPENDENT
5         do i=1,100
6           a(i) = i**2
7         end do
8
9         write(*,*) a
10        end
```

並列化可能であることの宣言

並列性を指示する場合：
並列化可否の考え方(1/3)

- 順序通りに実行しないと結果が変わるループは、並列化不可

→ (c)は並列化できない。

```
do i=1,50
 a(i)=a(i-1)+a(i)
enddo
```

```
do i=51,100
 a(i)=a(i-1)+a(i)
enddo
```

i = 50のとき a(50) へ書く

順序依存あり

先 → 後

i = 51のとき a(50) を読む

○
```
do i=...
 a(i)=a(i)+1.0
 b(i)=a(i)
enddo
```
(a)

○
```
do i=...
 b(i)=a(i-1)+a(i)
enddo
```
(b)

×
```
do i=...
 a(i)=a(i-1)+a(i)
enddo
```
(c)

○
```
do i=...
 c(i,1)=c(i-1,2)
enddo
```
(d)

並列性を指示する場合:
並列化可否の考え方(2/3)

- コンパイラでは並列化できないケース
  - (h)は、複数の繰り返しで同じ要素を定義するかもしれない。
  - (i)は、サブルーチン内でaの同じ要素をアクセスするかもしれない。

→利用者の責任でINDEPENDENT指示可能

?
```
do i=...
 a(ix(i))=...
enddo
```

(h) 間接参照

?
```
do i=...
 call subx(a,i)
enddo
```

(i) 手続呼出し

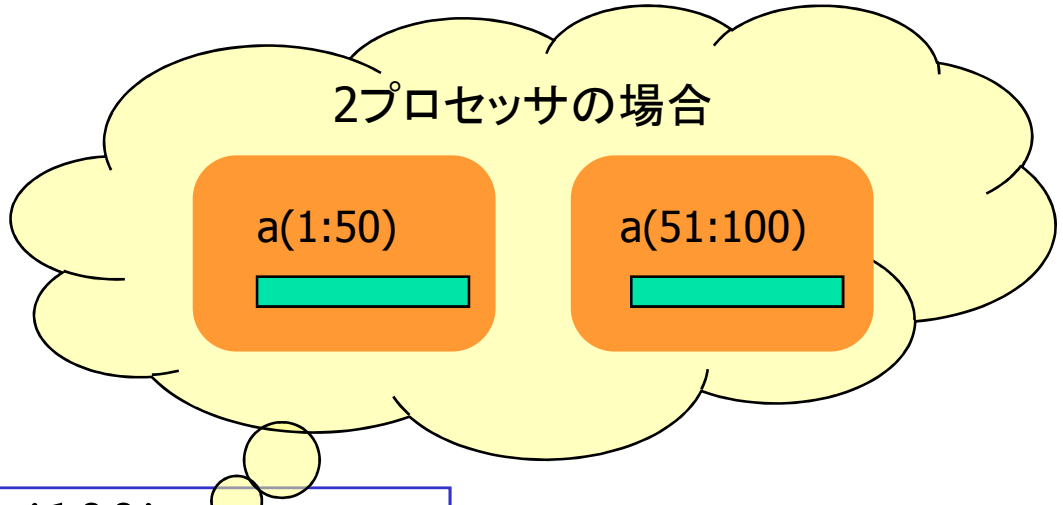
並列化可否の考え方(3/3)

- 飛び出しのあるループは並列化不可
  - ループ内から外への分岐
    - ループ内での分岐は可
  - STOP文、EXIT文
- 多重ループ
  - ループ毎に、並列化可否を判断

×
```
    do i=...
     if(...) goto 99
    enddo
99  ...
```
(k)

○ ×
```
do k=...
 do j=...
○  do i=...
    ... =d(i,j+1,k)
    d(i,j,k)= ...
   enddo
 enddo
enddo
```
(j)

ループの並列化（再掲）




```
1         integer a(100)
2  !HPF$ DISTRIBUTE a(BLOCK)
3
4  !HPF$ INDEPENDENT
5         do i=1,100
6           a(i) = i**2
7         end do
8
9         write(*,*) a
10        end
```

ループ並列性：指定
処理分担：無指定

- 並列化可能の判定
- 処理分担の決定

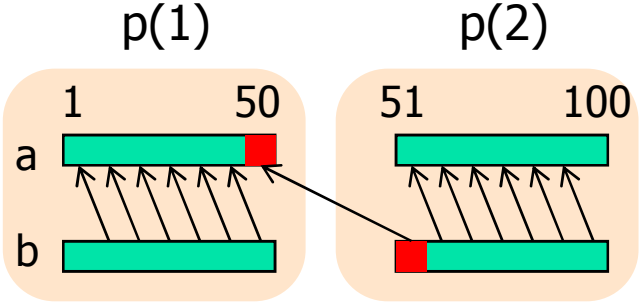
処理分担の決定とは

- 並列ループの処理分担とは
  - 反復のどの範囲を、どのプロセッサに担当させるか。
  - どう選択しても計算結果は正しいが、効率が著しく下がることがある。
- 処理分担を選択する目的
  - 計算効率を上げる。特に、通信量の削減。

【例】

```
      DIMENSION (100) :: a,b
!HPF$ DISTRIBUTE (BLOCK) :: a,b

!HPF$ INDEPENDENT
      do i=1,99
        a(i)=b(i+1)**2
      end do
```

2プロセッサの場合



i=50のとき
a(50)=b(51)**2

- i=50 を p(1) が担当すると、
  - p(1) はリモートの b(51) の値を読む → *通信*
- i=50 を p(2) が担当すると、
  - p(2) は計算結果をリモートの a(50) に書く → *通信*

一般に、どちらがよいとは言い切れない。

処理分担の決定:自動 or 指示

- 自動に任せる場合
  - コンパイラは、“Owner Computes” Rule を選択
    - 代入文左辺のデータの持ち主が計算する。
    - 多くの場合、通信量が少なくなり、効率がよい。
- 処理分担を指示する場合
  - ON指示文
    - 配列要素を指定することで、計算負荷分散を指示する。
    - “Owner Computes” Rule以外も指示できる。

処理分担を指示する場合：ON指示文

ループ処理の分担の指示　・・・DO文の直後（構文内）に

- 単純指示文（ループ本体が1実行文か1構文だけのとき）

!HPF$ **ON HOME(*<ホーム変数>*)**

- 指示構文　・・・ *ループ本体の全体を囲うように*

!HPF$ **ON HOME(*<ホーム変数>*) BEGIN**

<DOループ本体>

!HPF$ **END ON**

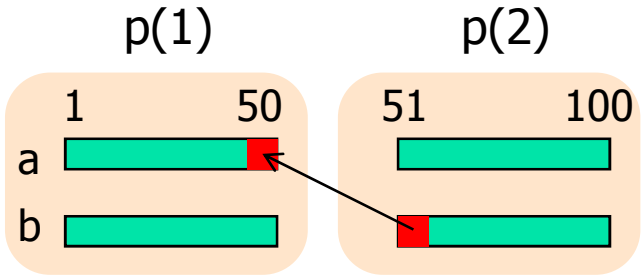
```
      DIMENSION(100) :: a,b
!HPF$ DISTRIBUTE(BLOCK) :: a,b

!HPF$ INDEPENDENT
      do i=1,99
!HPF$   ON HOME(a(i)) BEGIN
          a(i)=b(i+1)**2
!HPF$   END ON
      end do
```

a(i)を持つプロセッサがiを担当（デフォルト）

```
      DIMENSION(100) :: a,b
!HPF$ DISTRIBUTE(BLOCK) :: a,b

!HPF$ INDEPENDENT
      do i=1,99
!HPF$   ON HOME(b(i+1)) BEGIN
          a(i)=b(i+1)**2
!HPF$   END ON
      end do
```

b(i+1)を持つプロセッサがiを担当

処理分担を指示する場合：処理分担の選択



i=50のとき
a(50)=b(51)**2

- 実行イメージ
  - ON HOME(a(i)) のとき

b(51)

```
do i=1,50
  a(i)=b(i+1)**2
end do
```

```
do i=51,99
  a(i)=b(i+1)**2
end do
```

  - ON HOME(b(i+1)) のとき

```
do i=1,49
  a(i)=b(i+1)**2
end do
```

a(50)

```
do i=50,99
  a(i)=b(i+1)**2
end do
```

- 通信コストが小さくなるように、処理分担を選択する。
  - この例では大差なし。

自動に任すか、記述するか

- 基本的な書き方3種

データの分散 DISTRIBUTE	ループ並列化 INDEPENDENT	ループ処理分担 ON HOME
書く	書かない	書かない
書く	書く	書かない
書く	書く	書く

  - 必要度に合わせて
    - 自動でやってみて、だめなら書く
    - 性能的にシビアな部分は、最初から全部書く

- まだ性能が不十分なら
  - 通信の明示　　・・・次章で少し
  - 並列アルゴリズムの変更　　・・・「その2」で

その1の内容

- HPF言語の考え方(3.1)
  - 想定するハードウェア
  - プログラムの並列化とは
  - HPFが提供する「仮想」
  - HPFによるプログラム並列化

- データ分散とループの並列化(3.2)
  - 指示文の書式
  - データの分散(DISTRIBUTE指示文)
  - プロセッサの宣言(PROCESSORS指示文)
  - ループの並列化(INDEPENDENT指示文)
  - ループ処理の分担(ON指示文)
  - 自動か、書くか

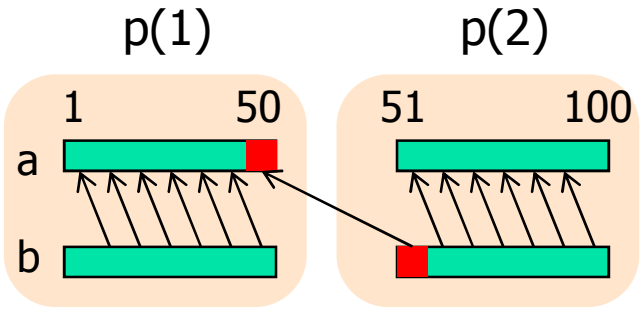
これだけで
HPFが書けます

- 典型的な通信最適化(6.1.2)
  - SHADOW指示文＋REFLECT指示文＋LOCAL節

これだけで
性能が出ます

ありがちな性能問題

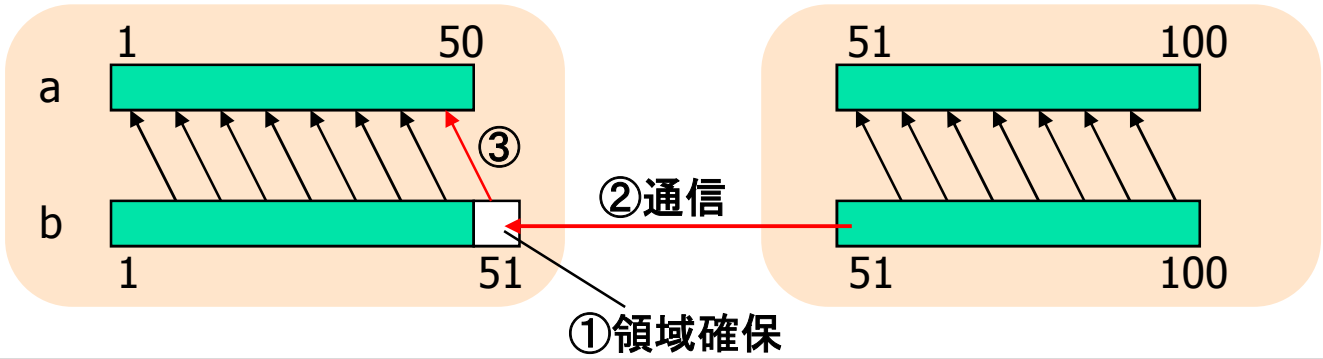
- 先ほどの例



i=50のとき、a(50)=b(51)**2
この些細な通信が、性能ネック！

```
      DIMENSION(100) :: a,b
!HPF$ DISTRIBUTE(BLOCK) :: a,b

!HPF$ INDEPENDENT
      do i=1,99
!HPF$  ON HOME(a(i)) BEGIN
        a(i)=b(i+1)**2
!HPF$  END ON
      end do
```

- 確実に性能を出すには
  - ① シャドウ領域を使用する宣言 ・・・ SHADOW指示文
  - ② シャドウ領域への通信の明示 ・・・REFLECT指示文
  - ③ シャドウ領域からのアクセスの明示 ・・・ON指示文のLOCAL節



シャドウを使ったチューニング

- 記述例

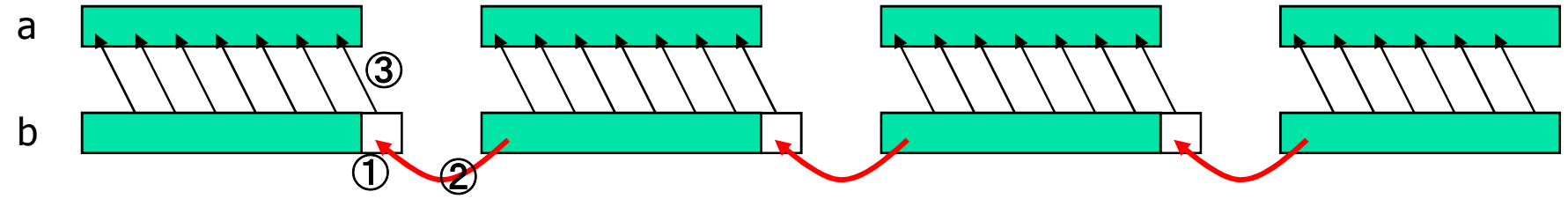
```
      DIMENSION(100) :: a,b
!HPF$ DISTRIBUTE(BLOCK) :: a,b
!HPF$ SHADOW b(0:1)

!HPF$ REFLECT b
!HPF$ INDEPENDENT
      do i=1,99
!HPF$  ON HOME(a(i)),LOCAL BEGIN
        a(i)=b(i+1)**2
!HPF$  END ON
      end do
```

①シャドウ領域の宣言
左に0、右に1、割当てを拡大

②シャドウ領域に値を埋める
通信の指示

③通信なしで計算できることを
宣言

4プロセッサの例



SHADOW指示文とREFLECT指示文

シャドウ領域の指示　・・・*宣言部で*

!HPF$ **SHADOW *a*(*〈シャドウ幅〉*, ...)　, ...**

または

!HPF$ **SHADOW (*〈シャドウ幅〉*, ...) :: *a, b, ...***

a や ***b*** は配列変数.

〈シャドウ幅〉 は，*〈整数〉* のとき、両側に拡張するシャドウのサイズ

〈整数〉：*〈整数〉* のとき、それぞれ下、上のシャドウのサイズ

シャドウ領域に対する通信　・・・*実行部で*

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

a や ***b*** は、SHADOWを持つ配列変数.

ON指示文のLOCAL節

ループ処理の分担の指示　・・・*DO文の直後(構文内)に*

- 単純指示文(ループ本体が1実行文か1構文だけのとき)

!HPF$ **ON HOME(**〈*ホーム変数*〉**),** 〈*LOCAL節*〉

- 指示構文　・・・ *ループ本体の全体を囲うように*

!HPF$ **ON HOME(**〈*ホーム変数*〉**) ,** 〈*LOCAL節*〉 **BEGIN**

　　〈*DOループ本体*〉

!HPF$ **END ON**

〈*LOCAL節*〉は，**LOCAL [(*a, b, ...*)]**

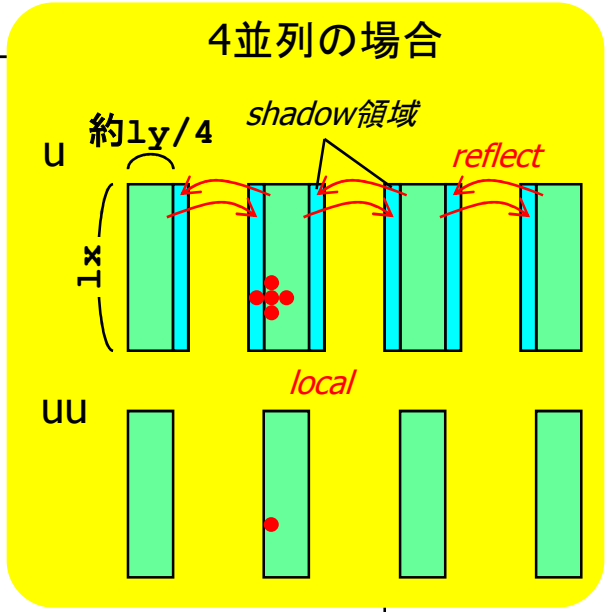
変数 ***a, b, ...*** について、通信なしでアクセスできることを宣言

(*a, b, ...*) が省略されたとき、構文内の全ての変数について、通信なしでアクセスできることを宣言

シャドウの使用例

- 実習で使用する熱伝導方程式プログラム diffusion.hpf

```
      ...
5     parameter( lx = 101, ly = 101, akap = 0.25 )
6     c
7           dimension u(lx,ly), uu(lx,ly)
8     !HPF$ DISTRIBUTE (*,BLOCK) :: u,uu
9     !HPF$ SHADOW (0,1) :: u
            ...

36    !HPF$       REFLECT u
37    c..
38    !HPF$       INDEPENDENT, NEW(ix,iy)
39                do iy = 2, ly-1
40    !HPF$         ON HOME(uu(:,iy)), LOCAL BEGIN
41                  do ix = 2, lx-1
42                    uu(ix,iy) = u(ix,iy) + akap *
43          $                   ( u(ix+1,iy  ) -2.0*u(ix,iy)+u(ix-1,iy)
44          $                    +u(ix  ,iy+1) -2.0*u(ix,iy)+u(ix  ,iy-1) )
45                  end do
46    !HPF$         ENDON
47                end do
            ...
```



その1の内容　おしまい

- HPF言語の考え方(3.1)
  - 想定するハードウェア
  - プログラムの並列化とは
  - HPFが提供する「仮想」
  - HPFによるプログラム並列化
- データ分散とループの並列化(3.2)
  - 指示文の書式
  - データの分散(DISTRIBUTE指示文)
  - プロセッサの宣言(PROCESSORS指示文)
  - ループの並列化(INDEPENDENT指示文)
  - ループ処理の分担(ON指示文)
  - 自動か、書くか
- 典型的な通信最適化(6.1.2)
  - SHADOW指示文＋REFLECT指示文＋LOCAL節

これだけで
HPFが書けます

これだけで
性能が出ます