FUJITSU THE POSSIBILITIES ARE INFINITE

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

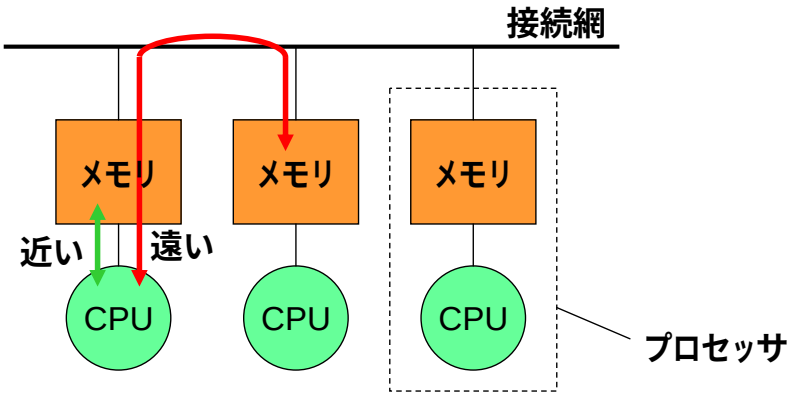
岩下 英俊
富士通 (株)
次世代テクニカルコンピューティング開発本部
2007年7月26日

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
- 最初の例：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
- スカラ変数の使い方 (4.1)
  - 重複割付け
  - NEW変数
- 集計計算 (4.2)
  - 集計計算 (REDUCTION節)
  - 演算種別の指定
  - 最大 (小) 値の位置

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

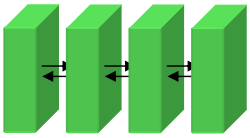
- 並列化率↑と、オーバヘッド↓
  - 計算の並列化
    - ループの並列化
    - タスク並列、手続間並列、・・・
  - 通信コストを削減
    - データの分割配置
    - 通信不要の判断
    - データの一時的移動・変形
    - 通信量・通信回数の削減
    - 通信パターンの最適化

並列化のスタイル2つ

- グローバルモデル
  - HPFのスタイル

全体の実行をプログラム.

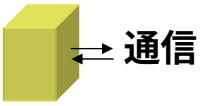
コンパイラが自動でN個に分割.



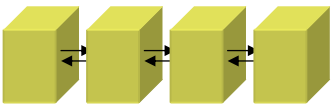
割り算の並列化

- SPMDモデル
  - MPIのスタイル

各プロセッサの実行をプログラム.



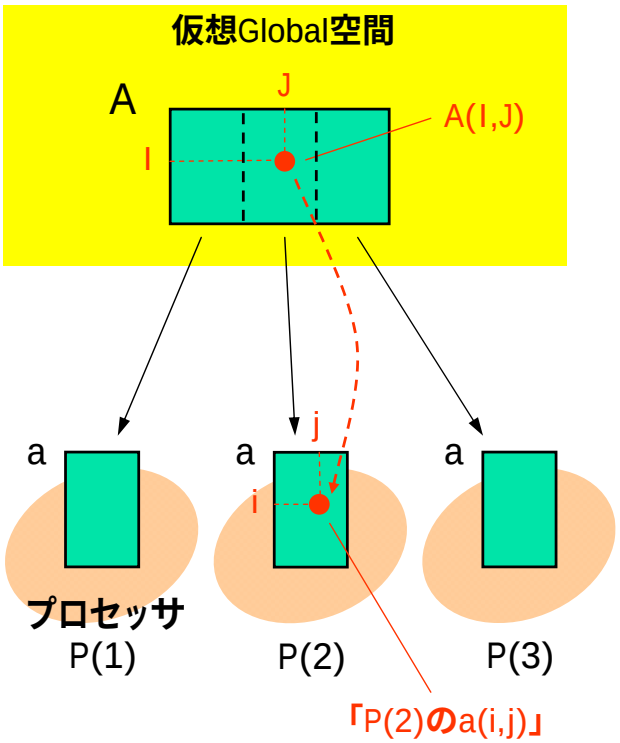
N個同時に実行する.



掛け算の並列化

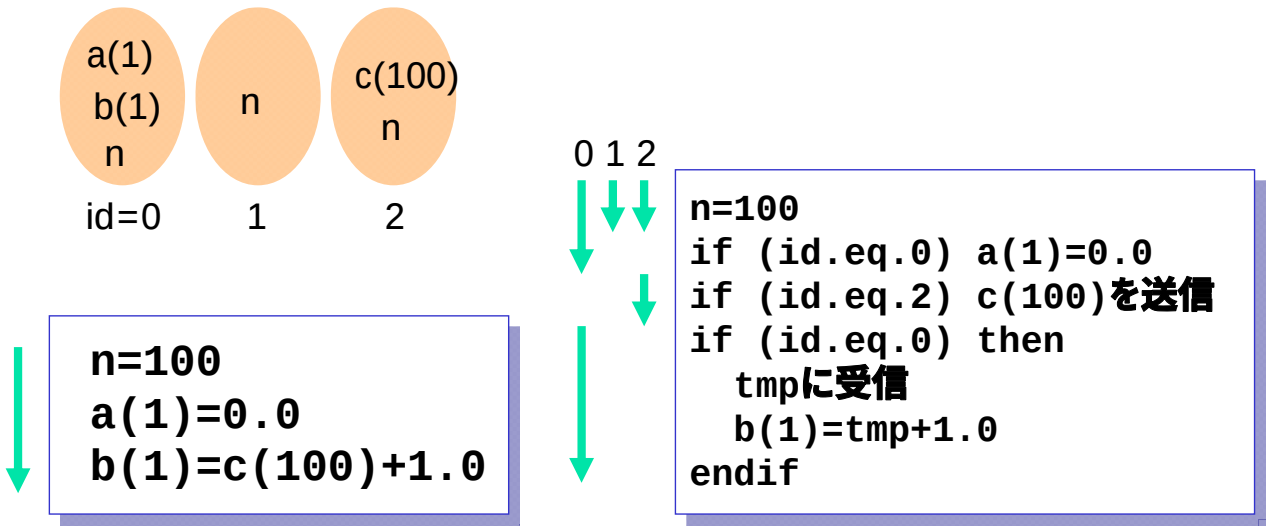
HPFが提供する「仮想」(1/2)

- 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。

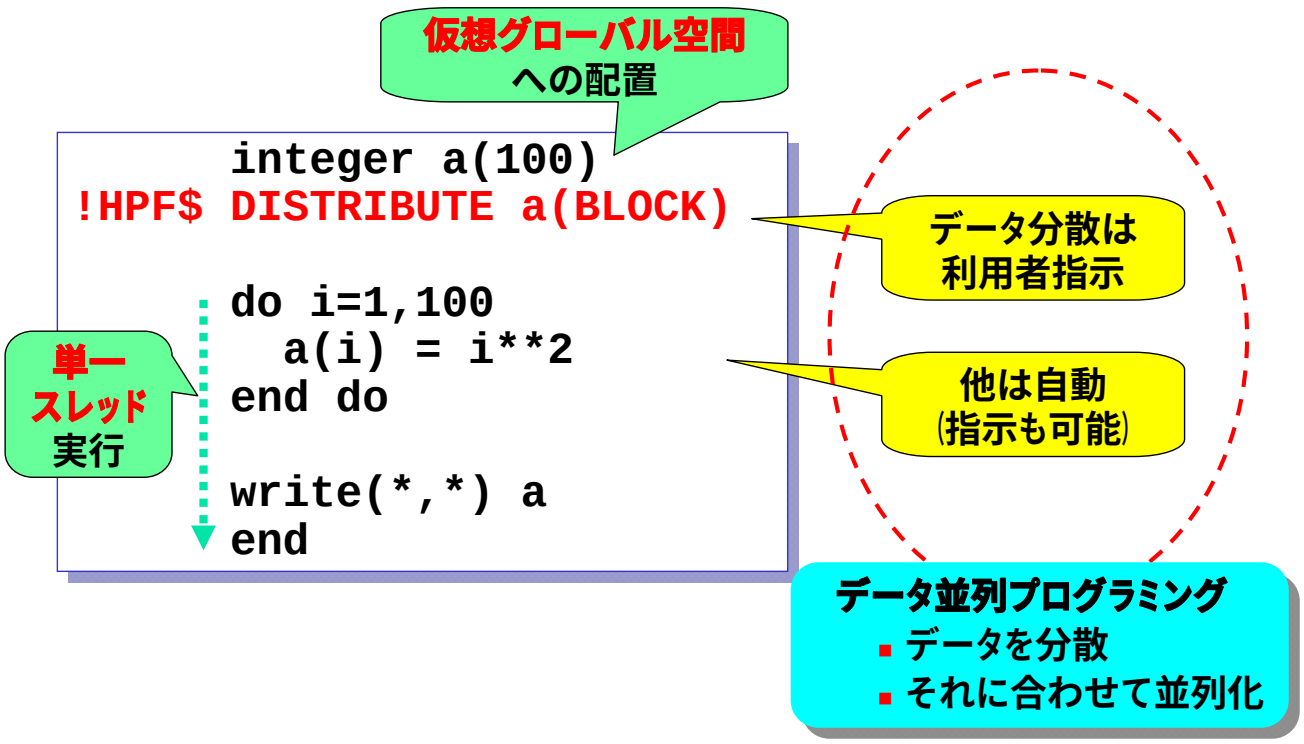


HPFが提供する「仮想」(2/2)

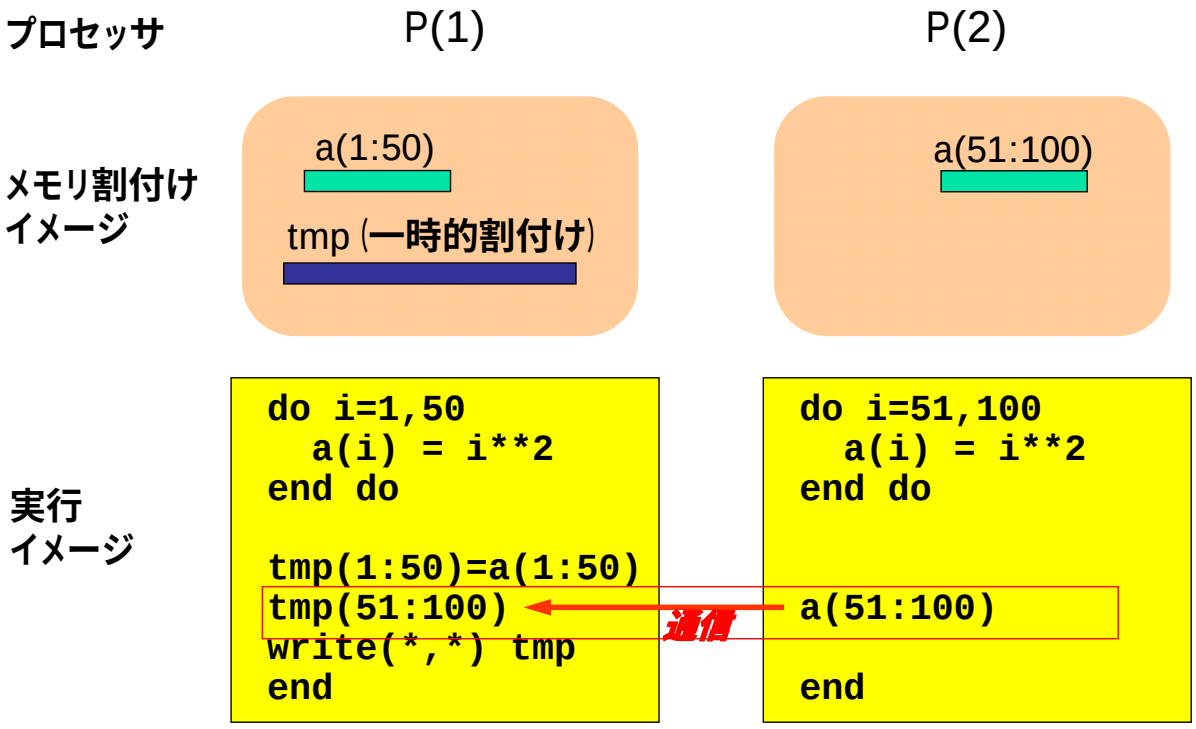
- 単一スレッド
  - プロセッサ番号で場合分けすることなく、一筋の実行列を記述する。



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
- 最初の例：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
- スカラ変数の使い方 (4.1)
  - 重複割付け
  - NEW変数
- 集計計算 (4.2)
  - 集計計算 (REDUCTION節)
  - 演算種別の指定
  - 最大 (小) 値の位置

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.

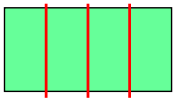
<分散形式> は，分散次元は BLOCK CYCLIC など
分散しない次元は *

【例】

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

4プロセッサ使用のとき

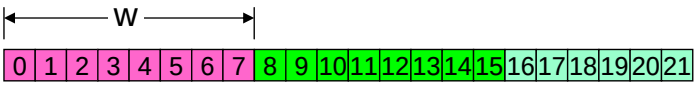
a(100,200)



b(100,200)

分散形式(1/2)

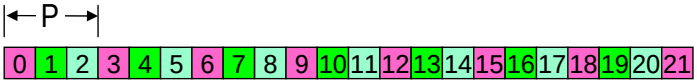
- DISTRIBUTE a(BLOCK)



w = ceil (N / P)
N: **配列のサイズ**
P: **プロセッサ数**

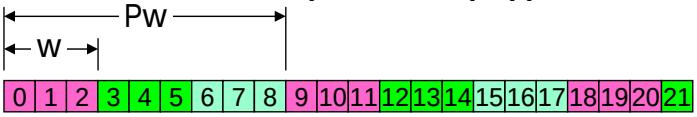
  - **近傍要素の参照が多い場合 (差分法など)**

- 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**

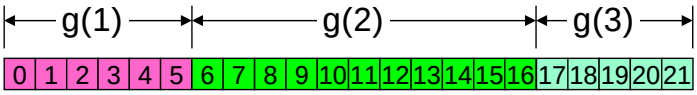


w: **指定ブロック幅**

  - **上二者の性質がある場合の中間的な手段**

分散形式(2/2)

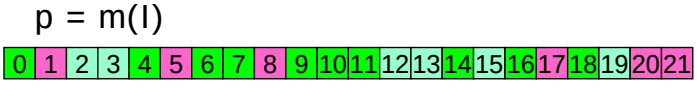
- !hpf$ distribute a(GEN_BLOCK(g)) : **不均等**block



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)**
  - **ヘテロ環境に跨るデータ分散**

- !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 (浮遊粒子など)**
  - **長方形･直方体にマップできない領域 (四面体、複雑な構造など)**

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** $p(n_1, \ldots, n_N)$, ...

p はプロセッサ配列名.

$n_1 \times \ldots \times n_N$ 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*

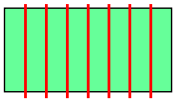
または

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

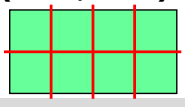
【例】

```
!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
```

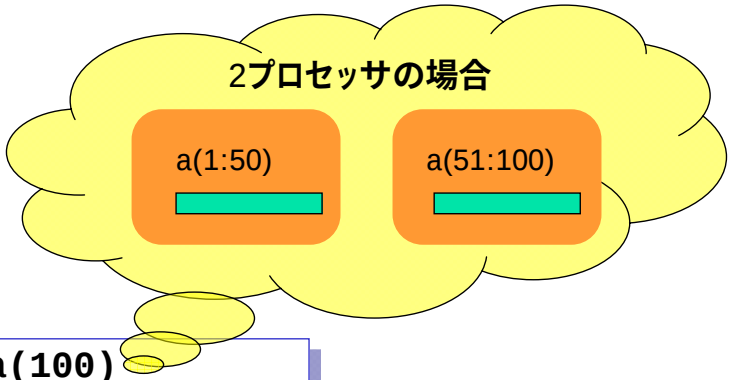
e(100,200)



f(100,200)



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。

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)

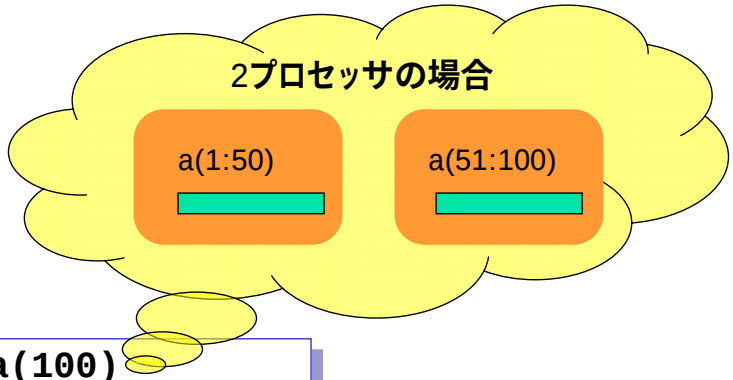
- 飛び出しのあるループは並列化不可
  - ループ内から外への分岐
    - ループ内での分岐は可
  - STOP文、EXIT文

- 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       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

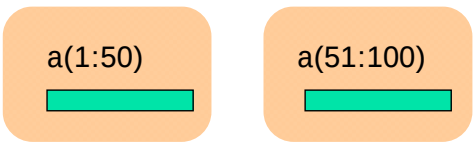
処理分担の決定とは

- 並列化可能なループについて、
  各繰り返しをどのプロセッサに担当させるか？
- 目的は、性能
  - 均等な負荷分散 & 通信コストの削減
  - 最適解は自明ではない。

【例】

```
      real a(100)
!HPF$ DISTRIBUTE a(block)
      do i=1,80
        a(i)=i**2
      enddo
```

2プロセッサの場合



✕ 負荷均等優先の実装

```
do i=1,40
 a(i)=i**2
enddo
a(41:50)
```

通信

```
do i=41,80
 a(i)=i**2
enddo
a(41:50)
```

○ 通信量削減優先の実装

```
do i=1,50
 a(i)=i**2
enddo
```

```
do i=51,80
 a(i)=i**2
enddo
```

データの所有者が、計算する。

処理分担の決定:自動 or 指示

- **自動に任せる場合**
  - “Owner Computes” Rule **が基本**
    - **代入文左辺のデータの持ち主が計算する。**
- **処理分担を記述する場合**
  - ON**指示文**
    - **配列要素を指定することで、処理分担を指示する。**
    - “Owner Computes” Rule**以外も指示できる。**

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**

```
!HPF$ INDEPENDENT
      do i=1,100
!HPF$ ON HOME(a(i)) BEGIN
        a(i) = i**2
!HPF$ END ON
      end do
```

ホーム変数 a(i)を持つプロセッサが、その繰り返しを担当

a(i) のアクセスでは通信は起こらない。

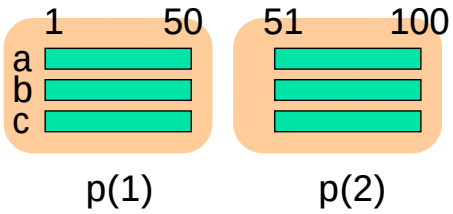
処理分担の選択

- 最善の方法は自明ではない。
- データ配置を微調整する、という選択肢も。(→応用編)

[例]

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

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



ON HOME(a(i)) とすると
- do i=1,50
- do i=51,99

- a(i) の更新は通信なし
- p(1) が b(51), c(51) を参照 *→通信*

ON HOME(b(i+1)) とすると
- do i=1,49
- do i=50,99

- b(i+1), c(i+1) の参照は通信なし
- p(2) が a(50) を更新 *→通信*

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

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
- 最初の例：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
- スカラ変数の使い方 (4.1)
  - 重複割付け
  - NEW変数
- 集計計算 (4.2)
  - 集計計算 (REDUCTION節)
  - 演算種別の指定
  - 最大 (小) 値の位置

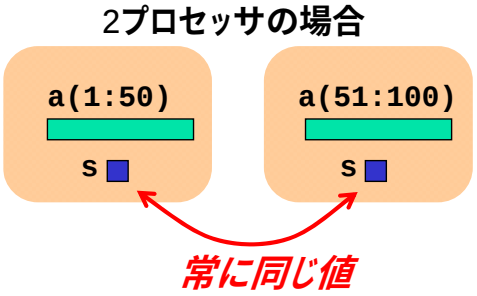
スカラ変数の割付け

- 分散と重複
  - 配列は (原則として) 分散割付け
  - スカラは重複割付け
    - 値の一意性は、システムが保証

【例】

```
      real a(100),s
!HPF$ DISTRIBUTE a(block)

      ...
```



スカラ変数のアクセス(1/2)

- 並列実行の外では、値の一意性を自動保証
  - 読み出しは、常に自プロセッサから
  - 書き込みは
    - ① 全プロセッサが、それぞれ行う（冗長計算）。または、
    - ② 1プロセッサで書き込んだ値を、全プロセッサに放送

【例】

```
      real a(100),s
!HPF$ DISTRIBUTE a(block)

      s=1.0
      a(1)=s+1.0
      a(51)=s+2.0
      s=a(1)*2
      ...
```

2プロセッサの場合

```
real a(1:50),s

s=1.0 ①
a(1)=s+1.0

s=a(1)*2
broadcast s
...
```

② 通信

```
real a(51:100),s

s=1.0 ①

a(51)=s+2.0

s
...
```


スカラ変数の割付け(2/2)

- 並列実行中は、一時的なばらつきが起こる
  - 読むだけの変数なら、問題なし
  - 書き込みのある変数は、用途毎に分類
    - 自動並列では、自動的に識別

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

(e) ループ内局所変数
ループを繰り越さないで役目を終える変数。
通常これがほとんど。

NEW変数

```
do i=...
 if(...) ifound=i
enddo
```

(f) 選択的代入
ループ内で1度だけ代入される変数。
（複数回代入なら、並列化可能条件に反する。）

```
do i=...
 asum=asum+a(i)
 if(a(i)>ax) ax=a(i)
enddo
```

(g) 集計計算
プロセッサを跨ぐ総和、最大値など、特別なパターン。

集計変数

INDEPENDENT指示文のNEW節

NEW変数を含むループの並列化

!HPF$ INDEPENDENT [, *<節>*] ...

<節> はNEW節またはREDUCTION節

NEW節は **NEW(*v*$_1$, *v*$_2$, ...)**

v$_1$ や *v*$_2$ はNEW変数. 変数名で指定.

```
!HPF$ INDEPENDENT,NEW(k,tmp,j)
      do i=m,n
        k=i+1
        do j=k,n
          tmp=a(i-1,j)+a(i,j)
          b(k)=b(k)+tmp
        enddo
      enddo
```

多くのコンパイラでは、無指定のスカラ変数をNEW変数と判断する最適化を行う。

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
- 最初の例：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
- スカラ変数の使い方 (4.1)
  - 重複割付け
  - NEW変数
- 集計計算 (4.2)
  - 集計計算 (REDUCTION節)
  - 演算種別の指定
  - 最大 (小) 値の位置

集計計算の並列化の原理

- 並列化可能条件を少し緩和
  - 特定の演算に限って、実行順序の変更を許す。
    - 交換則と結合則が成り立つ演算
  - 従来並列化できなかったループが、並列化できる。

```
asum=0.0
asum=asum+a(1)
asum=asum+a(2)
asum=asum+a(3)
asum=asum+a(4)
asum=asum+a(5)
asum=asum+a(6)
asum=asum+a(7)
asum=asum+a(8)
```

```
asum=0.0
asum=asum+a(1)
asum=asum+a(2)
asum=asum+a(3)
asum=asum+a(4)
```

```
asum=0.0
asum=asum+a(5)
asum=asum+a(6)
asum=asum+a(7)
asum=asum+a(8)
```

asum=(asumのプロセッサ間総和)

  - 浮動小数点演算では、誤差の範囲で逐次と結果が違う。

プログラム例：集計計算

- 分散配列の総和を返す関数
  - input: a(n), n, a0
  - output: a0+ Σ {a(i) | i=1,…,n}

```
1        real function asum(a,n,a0)
2        real a(n),a0
3  !HPF$ DISTRIBUTE a(CYCLIC)
4
5        asum=a0
6  !HPF$ INDEPENDENT,REDUCTION(asum)
7        do i=1,n
8         asum=asum+a(i)
9        end do
10       return
11       end
```

INDEPENDENT指示を書くならREDUCTION節は必須

集計文
このパターンから加算のreductionであることを自動判定

INDEPENDENT指示文のREDUCTION節

集計（リダクション）計算

!HPF$ INDEPENDENT [, <節>] ...

<節>は、NEW節またはREDUCTION節

REDUCTION節は

REDUCTION([<種別1>:] r_1, r_2, ...)

REDUCTION(<種別2>: r_1/i_{11}, i_{12}, .../, ...)

<種別1>は　+ * .AND. .OR. .EQV. .NEQV.
MAX MIN IAND IOR または IEOR

<種別2>は　FIRSTMAX FIRSTMIN LASTMAX
または LASTMIN

r_1 や r_2 は集計変数（変数名）

i_{11} や i_{12} は位置変数（スカラ変数名）

3つの書式
- 種別指定なし
- 種別1を指定
- 種別2を指定

REDUCTIONの書式1

- **書式1：演算種別指定なし（HPF2.0仕様）**

  REDUCTION(asum, amax)

  - 集計変数は、集計文だけに表れてよい。
  - 集計文の形式は以下に限定

    （rは集計変数、expはrを含まない式）

    - r = r *op* exp　または　r = exp *op* r

      op は + * .AND. .OR. .EQV. .NEQV. のいずれか
    - r = *f*(r, exp)　または　r = *f*(exp, r)

      f は MAX MIN IAND IOR IEOR のいずれか

      [例] `amax = MAX(amax, a(i))`
  - 制限違反はエラー検出される。
    - 集計文以外での集計変数の参照
    - 集計文の形式の違反

REDUCTIONの書式1の例

分散配列 a から、以下の値を得る。

- aamax　a の配列要素の絶対値の最大値

```
1       parameter(m=100)
2       real a(m)
3  !HPF$ DISTRIBUTE a(BLOCK)
4
5       aamax = -1.0
6  !HPF$ INDEPENDENT, REDUCTION(aamax)
7       do i=1,m
8         aamax = max( aamax, abs(a(i)) )
9       end do
```


REDUCTIONの書式2

- **書式2：種別1の指定** (HPF/JA拡張仕様)

  REDUCTION(+:asum, MAX:amax)

  - 集合演算の演算子・関数名を明示
    - + * .AND. .OR. .EQV. .NEQV. MAX MIN IAND IOR IEOR のいずれか
    - OpenMPと同じ書式
  - 書式1の機能を包含し、制限を緩和
    - 集計変数の出現場所は自由。呼出し手続で更新してもよい。
    - 意味的に、集計計算でなければならない。
      - 利用者責任。コンパイラはエラー検出しない（できない）。
- **書式1と2の使い分け**
  - 書式1の方が安全。エラー検出される。
  - 書式1で表現できなければ、注意して書式2を使う。

REDUCTIONの書式2の例

分散配列 a から、以下の値を得る。

- aamax　a の配列要素の絶対値の最大値

```
1       parameter(m=100)
2       real a(m)
3  !HPF$ DISTRIBUTE a(BLOCK)
4
5       aamax = -1.0
6  !HPF$ INDEPENDENT, REDUCTION(MAX:aamax)
7       do i=1,m
8         if ( aamax < abs(a(i)) ) then
9           aamax = abs(a(i))
10        end if
11      end do
```


REDUCTIONの書式3

- **書式3：種別2の指定** (HPF/JA**拡張仕様**)

  REDUCTION(FIRSTMAX:amax/i,j/)

  - MAX, MIN**の機能拡張**
    - FIRSTMAX, FIRSTMIN, LASTMAX, LASTMIN **のいずれか**
  - **最大 (小) 値を探すループ実行で、最大 (小) 値が得られた位置や、その位置での他の情報を、最大 (小) 値と同時に得る。**

    【例】分散配列 a(M,N) **の要素の最大値 と、最大値を取る添え字** (i,j) **を得る。**
  - **ループ実行の順序で、最初に見つかる最大 (小) 値と、最後に見つかる最大 (小) 値を区別。**
    - FIRST- と LAST-
    - **逐次実行時の意味に合うように、利用者が選択**

REDUCTIONの書式3の例

分散配列 a から、以下の値を得る。

- aamax　a の配列要素の絶対値の最大値
- i1　aamax を持つ最初の配列要素のindex
- a1　a(i1) の値。すなわち、aamax = abs(a1)

```
1       parameter(m=100)
2       real a(m)
3 !HPF$ DISTRIBUTE a(BLOCK)
4
5       aamax = -1.0
6 !HPF$ INDEPENDENT, REDUCTION(FIRSTMAX:aamax/i1,a1/)
7       do i=1,m
8         if ( aamax < abs(a(i)) ) then
9           i1 = i
10          a1 = a(i1)
11          aamax = abs(a1)
12        end if
13      end do
```

-1.0, “<=” なら、LASTMAX
huge(a), “>” なら、FIRSTMIN
huge(a), “>=” なら、LASTMIN

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
- 最初の例：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
- スカラ変数の使い方 (4.1)
  - 重複割付け
  - NEW変数
- 集計計算 (4.2)
  - 集計計算 (REDUCTION節)
  - 演算種別の指定
  - 最大 (小) 値の位置

FUJITSU THE POSSIBILITIES ARE INFINITE

HPF (High Performance Fortran) 講習会
入門編その2

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
2007年7月27日

その2の内容

- 多次元配列と重複割付け (4.3)
  - 行列積のプログラム例
  - 重複配列の使い方
  - 多次元配列の分散方法
- 手続呼出しの種類 (5.1)
- グローバル手続の呼出し (5.2)
  - 手続間のデータ渡し
  - 自動再マッピング
  - 手続を跨ぐデータの分散
- ローカル手続の呼出し (5.3)
  - EXTRINSIC接頭辞
- 分散配列に関する制限事項 (5.4)
- HPFによる並列化の方法 (5.5)

HPFプログラミングの基本

- 基本的な書き方3種（復習）

DISTRIBUTE (+PROCESSORS)	INDEPENDENT (+REDUCTION)	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

データマッピング　　計算マッピング

- 次に、通信の削減・改善
  - 基本は自動生成、自動最適化
    - 翻訳時メッセージなどで確認
  - 性能改善には
    - ON HOME指示の追加、改善
    - データ分散の改善
    - 通信方法の指示、通信不要の指示 → 応用編で

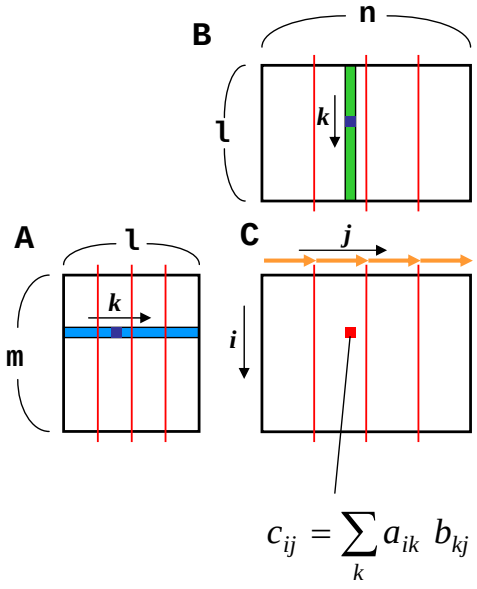
行列積プログラム（1）(1/2)

- 行列積 C = A × B を計算するサブルーチン
  - Input: [m, l] 行列 A, [l, n] 行列 B; サイズ m, n, l
  - Output: [m, n] 行列 C
  - A, B, C とも2次元目でblock分散

```
1       subroutine mmul(A,B,C,m,n,l)
2       real A(m,l),B(l,n),C(m,n)
3 !HPF$ DISTRIBUTE (*,BLOCK) :: A,B,C
4
5       C=0.0
6       do j=1,n
7        do i=1,m
8         do k=1,l
9          C(i,j)=C(i,j)+A(i,k)*B(k,j)
10        end do
11       end do
12      end do
13      return
14      end
```

j で自動並列化

4並列の場合



$$c_{ij} = \sum_{k} a_{ik}\, b_{kj}$$

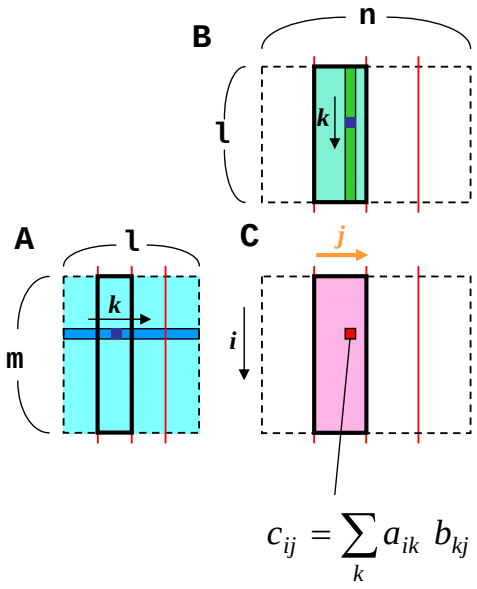
行列積プログラム(1) (2/2)

- 各プロセッサの動作イメージ
  - 並列化した結果、Aの参照で通信が必要
    → コンパイラが自動生成

```
1       subroutine mmul(A,B,C,m,n,l)
2       real A(m,□),B(l,□),C(m,□)
3
4
5       C=0.0
6       do j∈担当範囲
7        do i=1,m
8         do k=1,l
9          C(i,j)=C(i,j)+A(i,k)*B(k,j)
10        end do
11       end do
12      end do
13      return
14      end
```

通信発生
(自動生成)

4並列の場合



$$c_{ij} = \sum_k a_{ik}\ b_{kj}$$

行列積プログラム(2)

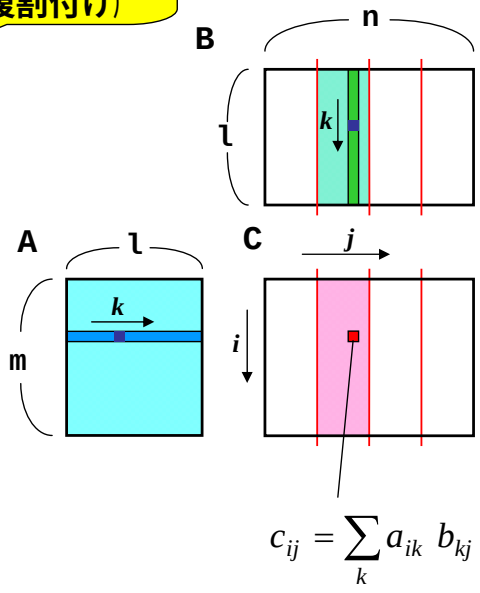
- 性能改善版
  - Aのコピーを全員が持つ ・・・重複配列

Aは分散指示なし
(重複割付け)

```
1       subroutine mmul(A,B,C,m,n,l)
2       real A(m,l),B(l,n),C(m,n)
3  !HPF$ DISTRIBUTE (*,BLOCK) :: B,C
4
5       C=0.0
6       do j=1,n
7        do i=1,m
8         do k=1,l
9          C(i,j)=C(i,j)+A(i,k)*B(k,j)
10        end do
11       end do
12      end do
13      return
14      end
```

j で自動並列化

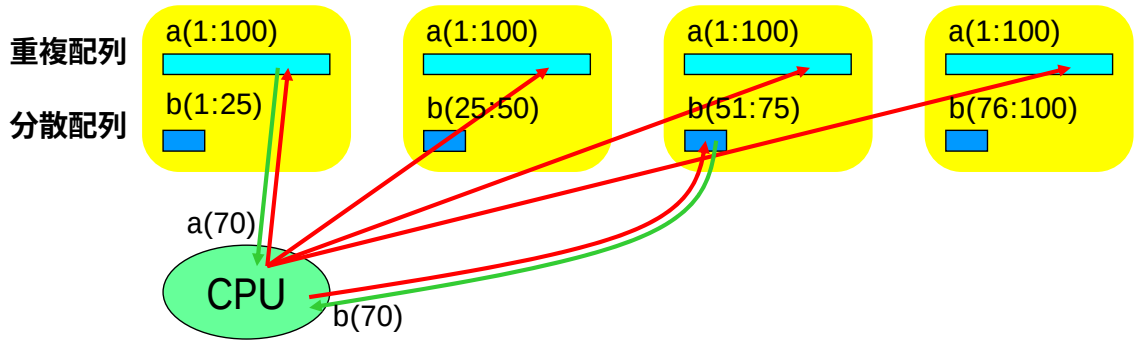
どれも通信なし



$$c_{ij} = \sum_k a_{ik}\ b_{kj}$$

重複配列

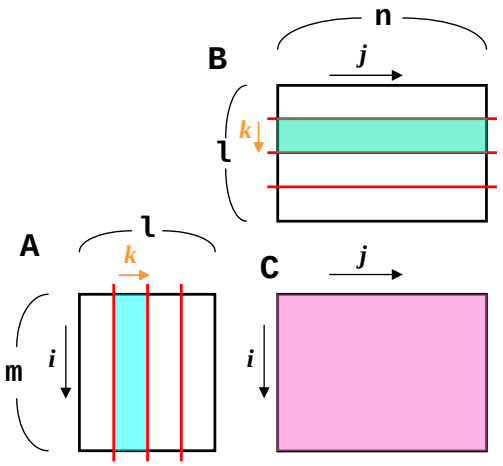
- 分散配列と比較して
  - 参照はいつも通信なしで高速。
  - 更新は遅い。1プロセッサからbroadcast、または全プロセッサの冗長実行（並列性なし）。
    - コンパイラは、常に同じ値を保証。
  - メモリを食う。メモリ消費量に台数効果なし。
- 用途
  - 値の更新が少ない、読むだけのデータ



行列積プログラム（3）

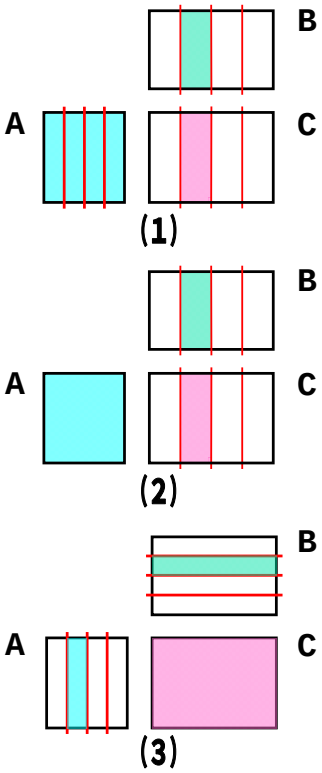
- C を重複配列にする方法
  - 配列の集計計算

```
1       subroutine mmul(A,B,C,m,n,l)
2       real A(m,l),B(l,n),C(m,n)
3 !HPF$ DISTRIBUTE A(*,BLOCK)
4 !HPF$ DISTRIBUTE B(BLOCK,*)
5
6       C=0.0
7 !HPF$ INDEPENDENT,NEW(j,i),REDUCTION(C)
8       do k=1,l
9 !HPF$  ON HOME(A(:,k))
10       do j=1,n
11        do i=1,m
12         C(i,j)=C(i,j)+A(i,k)*B(k,j)
13        end do
14       end do
15      end do
16      return
17      end
```



分散方法の比較

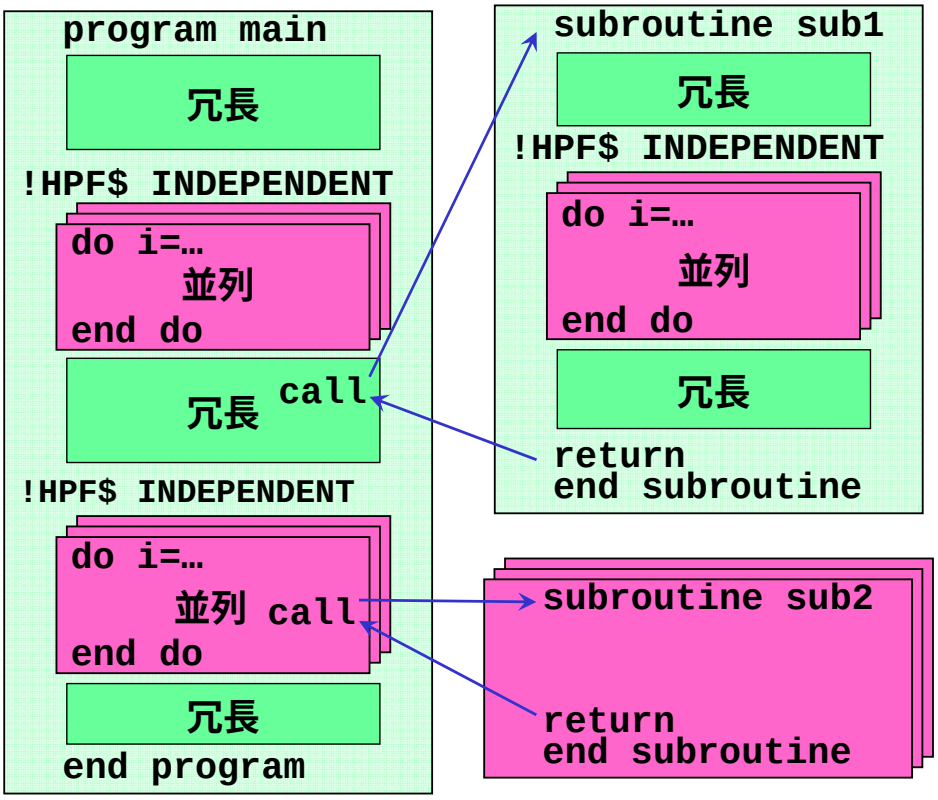
- データ配置、通信コスト、メモリ効率
  - (1) A,B,Cとも2次元目で分散
    - Aの参照で通信が発生
    - 配置のメモリ使用量は最小
      - 実行時に大きな一時領域が必要
  - (2) Aは重複、B,Cは2次元目で分散
    - 通信が発生しないので、最も高速
    - Aのメモリ使用量が大きい
  - (3) A/Bは2/1次元目で分散、Cは重複
    - Cの全配列集計計算のコスト
    - Cのメモリ使用量が大きい
- 全体の性能のためには、プログラム全体を考慮して分散を選択



その2の内容

- 多次元配列と重複割付け (4.3)
  - 行列積のプログラム例
  - 重複配列の使い方
  - 多次元配列の分散方法
- 手続呼出しの種類 (5.1)
- グローバル手続の呼出し (5.2)
  - 手続間のデータ渡し
  - 自動再マッピング
  - 手続を跨ぐデータの分散
- ローカル手続の呼出し (5.3)
  - EXTRINSIC接頭辞
- 分散配列に関する制限事項 (5.4)
- HPFによる並列化の方法 (5.5)

手続呼出しの場所による違い



手続呼出しの分類

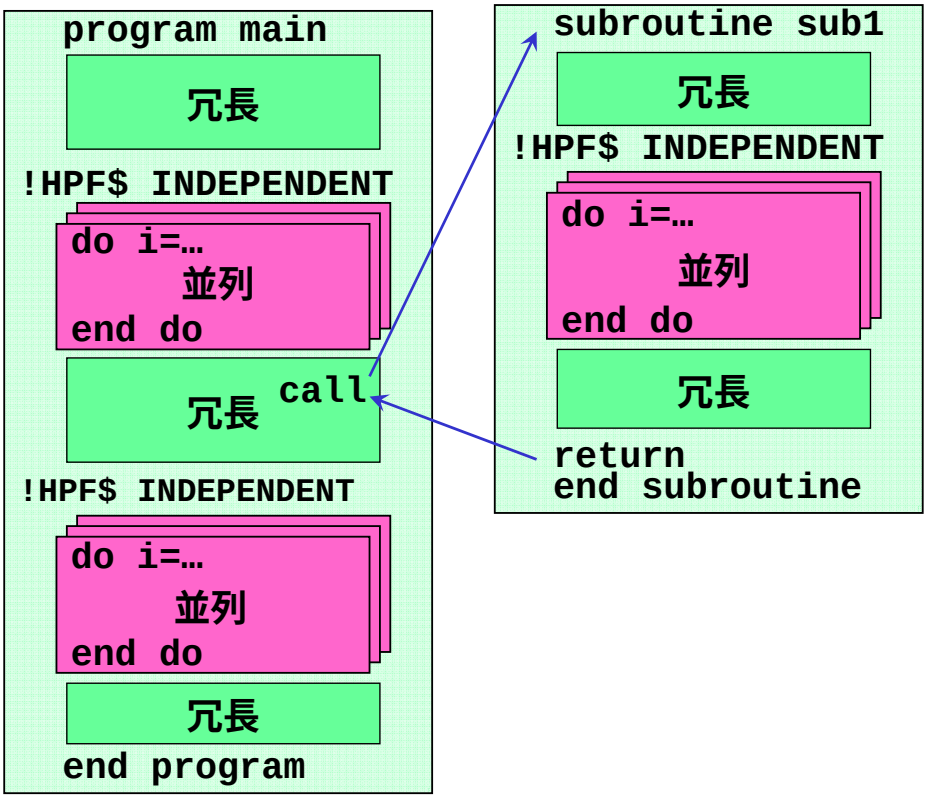
- 呼出し元
  - 冗長実行部 ・・・ 並列ループの外側
  - 並列実行部 ・・・ 並列ループの内側
- サブプログラム
  - グローバル手続 ・・・ 複数プロセッサによる実行
    (グローバルモデル。通常のHPF)
  - ローカル手続 ・・・ 個々のプロセッサの実行
    (SPMDモデル。Fortran, MPI)
  - (シリアル手続) ・・・ 強制的に1プロセッサで行う実行
    (逐次モデル。Fortran)

- 実用的には、2つのパターンで十分
  - 冗長実行部で、グローバル手続 (HPF) を呼び出す。
  - 並列実行部で、ローカル手続 (Fortran) を呼び出す。

その2の内容

- 多次元配列と重複割付け (4.3)
  - 行列積のプログラム例
  - 重複配列の使い方
  - 多次元配列の分散方法
- 手続呼出しの種類 (5.1)
- グローバル手続の呼出し (5.2)
  - 手続間のデータ渡し
  - 自動再マッピング
  - 手続を跨ぐデータの分散
- ローカル手続の呼出し (5.3)
  - EXTRINSIC接頭辞
- 分散配列に関する制限事項 (5.4)
- HPFによる並列化の方法 (5.5)

グローバル手続の呼出し



手続間のデータ渡し

```
  subroutine sub0
  common /aaa/a
!HPF$ DISTRIBUTE a …
!HPF$ DISTRIBUTE b …


  call sub1(b)


  end subroutine
```

```
  subroutine sub1(y)
  common /aaa/x
!HPF$ DISTRIBUTE x …
!HPF$ DISTRIBUTE y …



  return
  end subroutine
```

- COMMON変数
  - 分散は利用者責任で一致。
- モジュール変数
  - 記述は一箇所なので、当然一致。
- 実引数と仮引数
  - 分散は一致していなくてもよい。→ 自動再マッピング

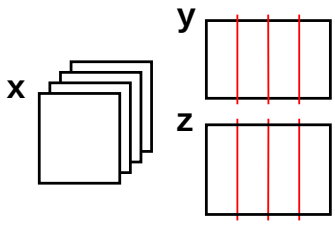
引数の分散が一致する呼出し

```
      real x(n1,m),y(m,n2),z(n1,n2)
!HPF$ DISTRIBUTE (*,BLOCK) :: y,z
      ...
      call mmul(x,y,z,n1,n2,m)
      ...
```

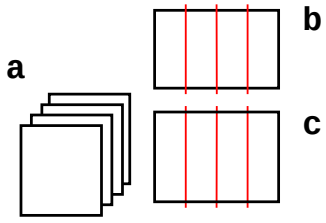
再マッピングなし
(アドレス渡し)

```
      subroutine mmul(a,b,c,m,n,l)
      real a(m,l),b(l,n),c(m,n)
!HPF$ DISTRIBUTE (*,BLOCK) :: b,c

      c=0.0
      do j=1,n
       do i=1,m
        do k=1,l
         c(i,j)=c(i,j)+a(i,k)*b(k,j)
        end do
       end do
      end do
      return
      end
```



分散の有無、
分散次元・形式
が完全に一致



引数の分散が一致しない呼出し

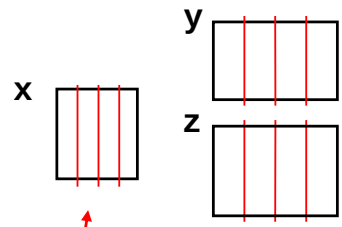
```
      real x(n1,m),y(m,n2),z(n1,n2)
!HPF$ DISTRIBUTE (*,BLOCK) :: x,y,z
      ...
      call mmul(x,y,z,n1,n2,m)
      ...
```

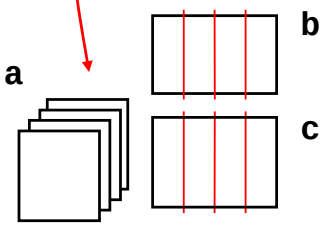
a(m,l) を割付け
x→a のコピー

a→x のコピー
aの開放

ただし、(文法では)
呼出し側に
サブプログラムの
引数に関する情報
(明示的引用仕様)
が必要

x⇔aの分散が
不一致






```
      subroutine mmul(a,b,c,m,n,l)
      real a(m,l),b(l,n),c(m,n)
!HPF$ DISTRIBUTE (*,BLOCK) :: b,c

      ...
```


明示的引用仕様 (F90文法)

- 再マッピングを起こすには、「明示的引用仕様」が必要
  (仕様では)

(a) INTERFACE文を使う方法

```
subroutine yyy

 interface
 subroutine xxx
   宣言部
 end subroutine
 end interface

 call xxx

end subroutine
```

```
subroutine xxx
   宣言部
   実行部
end subroutine
```

(b) モジュールを使う方法

```
module mmm
contains
 subroutine xxx
   宣言部
   実行部
 end subroutine
end module
```

```
subroutine yyy

 use mmm


 call xxx

end subroutine
```

(c) 内部手続にする方法

```
subroutine yyy

 call xxx

contains
 subroutine xxx
   宣言部
   実行部
 end subroutine
end subroutine
```

インタフェースブロックの記述例

```
      real x(n1,m),y(m,n2),z(n1,n2)
!HPF$ DISTRIBUTE (BLOCK,*) :: x,y,z
      interface
       subroutine mmul(a,b,c,m,n,l)
       real A(m,l),B(l,n),C(m,n)
!HPF$  DISTRIBUTE (*,BLOCK) :: b,c
      end interface
      ...
      call mmul(x,y,z,n1,n2,m)
      ...
```

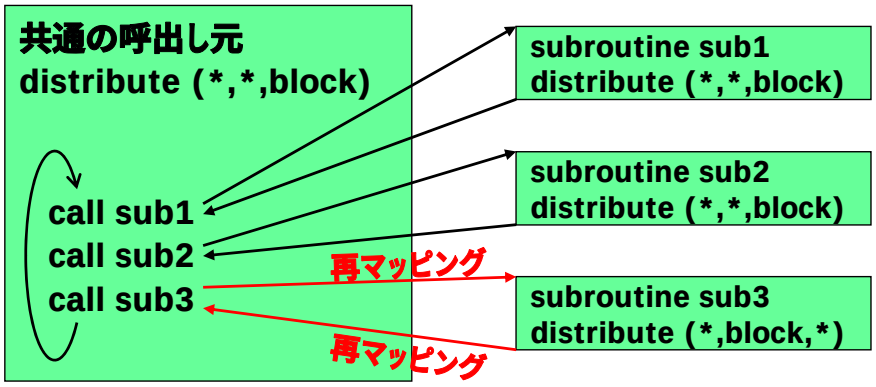
```
      subroutine mmul(a,b,c,m,n,l)
      real a(m,l),b(l,n),c(m,n)
!HPF$ DISTRIBUTE (*,BLOCK) :: b,c

      ...
```

宣言部を
そのままコピー

手続きを跨ぐデータ分散のコツ

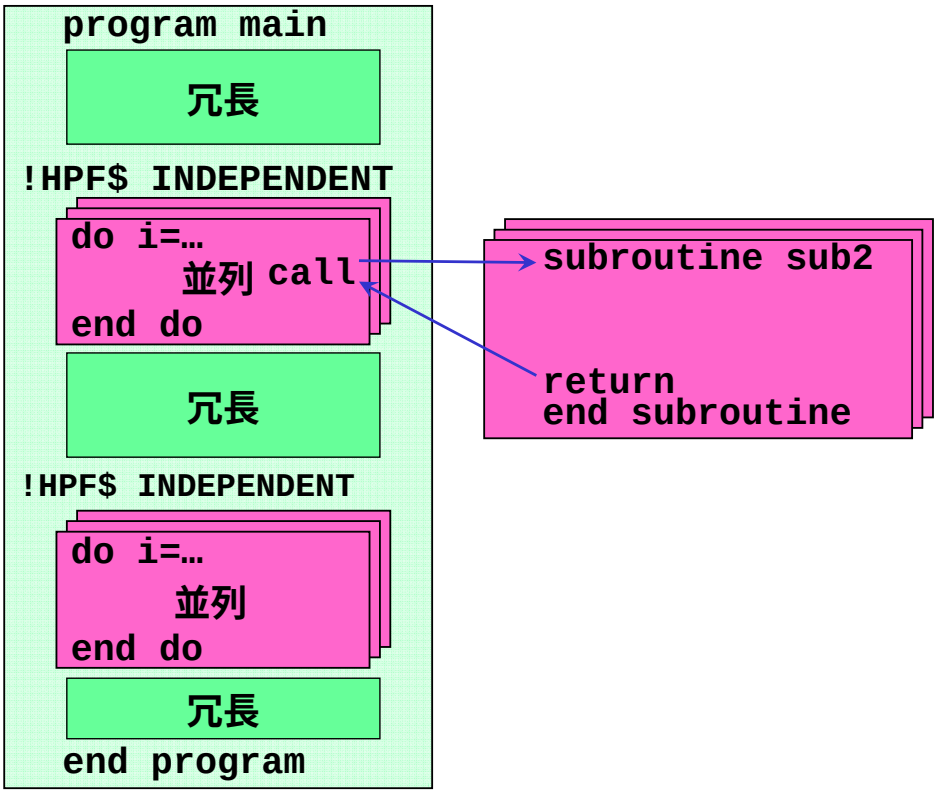
- アプリケーション全体で、通信が少なくなるデータ分散を考える。
  - 再マッピングを避ける
    - コストの高いサブプログラムで分散を決める。
    - 呼出し元や、他のサブプログラムも、同じ分散にする。
  - 再マッピングを積極的に利用
    - 再マッピング回数を減らす工夫を。



その2の内容

- 多次元配列と重複割付け (4.3)
  - 行列積のプログラム例
  - 重複配列の使い方
  - 多次元配列の分散方法
- 手続呼出しの種類 (5.1)
- グローバル手続の呼出し (5.2)
  - 手続間のデータ渡し
  - 自動再マッピング
  - 手続を跨ぐデータの分散
- ローカル手続の呼出し (5.3)
  - EXTRINSIC接頭辞
- 分散配列に関する制限事項 (5.4)
- HPFによる並列化の方法 (5.5)

ローカル手続の呼出し



ローカル手続の書き方・使い方

- Fortranコンパイラでコンパイルしてリンク
  - 可能。詳細手順は実装による。
- HPFコンパイラでコンパイルする方法
  - EXTRINSIC接頭辞（指示文ではない）を使用
    - その手続が（HPFでなく）Fortranであることの宣言

言語と言語モデルの宣言（Fortran仕様の拡張）
EXTRINSIC(*<言語>*, <*モデル*>) SUBROUTINE ...
EXTRINSIC(*<言語>*, <*モデル*>) FUNCTION ...
EXTRINSIC(*<言語>*, <*モデル*>) MODULE ...
<言語> **は** 'HPF' **または** 'FORTRAN'
<モデル> **は** 'GLOBAL' 'LOCAL' **または** 'SERIAL'
デフォルトは ('HPF', 'GLOBAL')
許される組合せは処理系による.
手続本体と明示的引用仕様に必要.

ローカル手続をHPFで記述する例

```
      real a(2,n), c(n)
!HPF$ DISTRIBUTE a(*,BLOCK)
!HPF$ DISTRIBUTE c(BLOCK)

      interface
       EXTRINSIC('Fortran','LOCAL') subroutine foolocal(x,r)
       real x(2),r
       end subroutine
      end interface

!HPF$ INDEPENDENT
      do i=1,n
        call foolocal(a(:,i),c(i))
      enddo
```

```
      EXTRINSIC('Fortran','LOCAL') subroutine foolocal(x,r)
      real x(2),r
      r = sqrt(x(1)**2 + x(2)**2)
      return
      end subroutine
```

その2の内容

- 多次元配列と重複割付け (4.3)
  - 行列積のプログラム例
  - 重複配列の使い方
  - 多次元配列の分散方法
- 手続呼出しの種類 (5.1)
- グローバル手続の呼出し (5.2)
  - 手続間のデータ渡し
  - 自動再マッピング
  - 手続を跨ぐデータの分散
- ローカル手続の呼出し (5.3)
  - EXTRINSIC接頭辞
- 分散配列に関する制限事項 (5.4)
- HPFによる並列化の方法 (5.5)

分散配列の制限事項

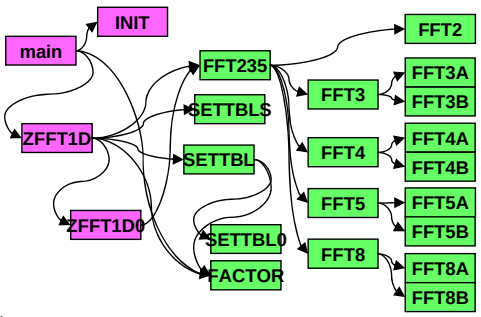
- 分散配列に関して、Fortran仕様の一部が制限される。
  - 理由：メモリの連続性 (記憶列結合、順序結合) が保証できない
  - 例えば、a(1) の次のアドレスに a(2) があるとは限らない。次のプロセッサかもしれない。
- 主な制限事項
  - EQUIVALENCE文中の指定は不可。
  - COMMON文で指定するとき、形状 (次元数と大きさ) と分散がすべて一致すること。
  - 実引数と仮引数は、形状が一致すること。
    - 分散不一致は再マッピング条件を満たせば可。
  - 大きさ引継ぎ配列 (仮引数 a(*) など) は不可。
- このような配列を含む手続の並列化は、
  - 重複配列として扱う。
  - あきらめて、Fortran手続として呼び出す。

その2の内容

- 多次元配列と重複割付け (4.3)
  - 行列積のプログラム例
  - 重複配列の使い方
  - 多次元配列の分散方法
- 手続呼出しの種類 (5.1)
- グローバル手続の呼出し (5.2)
  - 手続間のデータ渡し
  - 自動再マッピング
  - 手続を跨ぐデータの分散
- ローカル手続の呼出し (5.3)
  - EXTRINSIC接頭辞
- 分散配列に関する制限事項 (5.4)
- HPFによる並列化の方法 (5.5)

並列化の手順の提案

1. プログラム全体の手続呼出し関係を把握。
   - 呼出しグラフを書いてみる。
2. 主要変数を見つけて、分散を決める。
   - 分散する次元を決める。
   - 分散形式はblockに仮決めでもよい。
     いつでも変更できる。(ここがHPF！)
3. 並列化の対象と、逐次手続を分ける。
   - 主要変数が引数かCOMMONで渡る手続が、
     並列化の対象
   - 並列ループから呼ばれる手続は、
     逐次手続 (ローカル手続)
4. 方針は決まった。後はチューニング。
   - 主要変数以外の変数の分散
   - 計算マッピングが自動で不十分なら、INDEPENDENT、REDUCTON、
     ON HOME、・・・



その2の内容　おしまい

- 多次元配列と重複割付け (4.3)
  - 行列積のプログラム例
  - 重複配列の使い方
  - 多次元配列の分散方法
- 手続呼出しの種類 (5.1)
- グローバル手続の呼出し (5.2)
  - 手続間のデータ渡し
  - 自動再マッピング
  - 手続を跨ぐデータの分散
- ローカル手続の呼出し (5.3)
  - EXTRINSIC接頭辞
- 分散配列に関する制限事項 (5.4)
- HPFによる並列化の方法 (5.5)