# XPFortran 入門

永井　亨

## Ⅰ．予備知識

名古屋大学情報連携基盤センター全国共同利用システムのスーパーコンピュータは Fujitsu PRIMEPOWERHPC2500 です。アプリケーションサーバも同一機種であるため，これらを一体的に運用しています。XPFortran は，HPC2500 上で複数の CPU を使って並列処理を行うために必要な命令及びデータの分割や，CPU 間のデータ通信の命令を記述するための言語で，Fortran に指示行を挿入する形式で表現します。並列処理プログラミングを説明する前に，HPC2500 及び XPFortran に関するいくつかの予備的な説明をしておくことにします。

### 1　ハードウェア構成

HPC2500 のハードウェア構成を図 1 に示します。システムは 24 ノードで構成される分散並列型スカラ計算機で，ノード間は 4 GB/ 秒の高速クロスバネットワークで結合されます。各ノードは共有メモリ型スカラ計算機で，24 ノードのうちの 22 ノードには 64 個の CPU，残りの 2 ノードには 128 個の CPU を搭載し，全体では 1664CPU になります。各 CPU は理論最大性能 8 Gflops[1] のスカラプロセッサです。ただし，これは乗算と加算が同時に実行された場合のピーク性能です。システム全体の理論最大性能は 12.5Tflops になります。各ノードは主記憶 512GB を搭載し，トータルで 12TB になります。磁気ディスク容量は 100TB です。24 ノードのうちの 1 つは学内 LAN（NICE）に接続され，バッチ処理に加えて会話型処理のサービスも行ないます。IO ノード以外のノードは演算処理専用で，入出力は IO ノードを経由して行ないます。

### 2　並列処理の実行イメージ

XPFortran で記述したプログラムの実行イメージを図 2 にしたがって説明します。3 つのプロセスを使って並列処理を行なうとして，図 2 の左半分が XPFortran による並列処理プログラムを，右半分が処理の流れをあらわしています。

図 2 のプログラム中の `!xocl` で始まる行が並列処理のための指示行です。実行の流れは以下のようになります。

1．プログラム（ロードモジュール）が 3 つのプロセスのいずれか（図 2 ではプロセス 1 ）にロー

1　1 秒間に $8 \times 10^9$ 回の浮動小数点演算ができることになります。

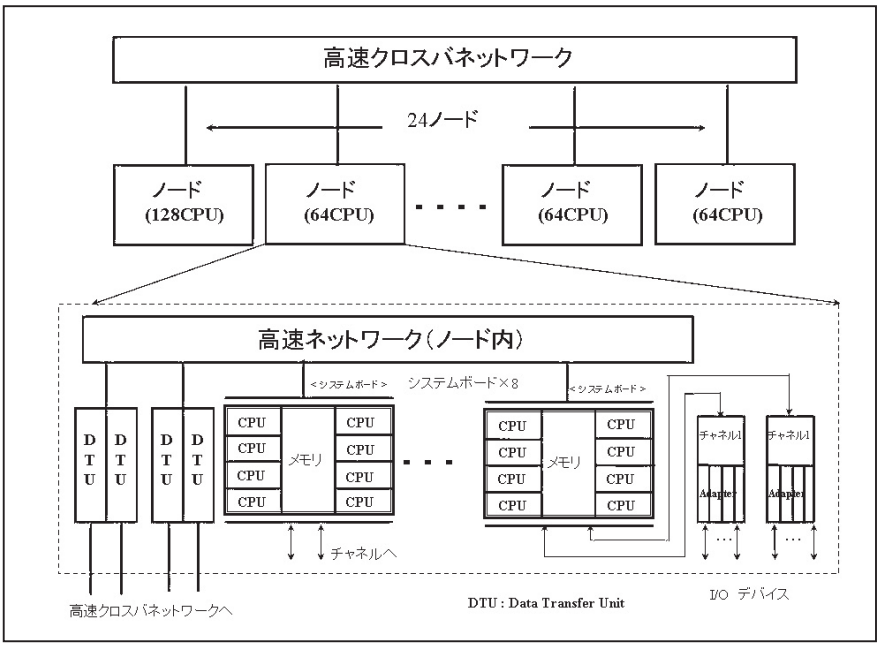


図１　HPC2500 のシステム構成

ドされて実行が始まります。したがって，開始直後はプロセス１だけが処理を行ないます（図２では A の部分。この部分には通常の Fortran が記述されていると考えてください）。この部分を逐次実行と呼びます。逐次実行を行なうプロセス（ここではプロセス１）をここでは親プロセスと呼び，それ以外のプロセス（プロセス２とプロセス３）を子プロセスと呼ぶことにします。

２．parallel region 文にさしかかったところでプログラムが他のプロセスに複写されて[2]並列処理が始まります。並列処理の開始直後はどのプロセスも同一の処理（図２では B の部分）を行なうことになります。この同一の処理を行なう部分を冗長実行と呼びます[3]。

３．spread do 文は do ループを分割する命令です。この部分（spread do 文から end spread do 文まで）では，それぞれのプロセスが自身に割り当てられた do ループの繰り返し範囲を処理します。図２では１から 300 の繰り返し範囲を３等分して処理することを示しています。

４．spread region 文から end spread region 文までのところはそれぞれのプロセスが行なう処理を個々に記述した部分です。図２ではプロセス１が処理 E を，プロセス２が処理 F を，プロセス３が処理 G をそれぞれ行ないます。

５．end parallel region 文にさしかかると，並列処理は終了します。図２ではプロセス２及びプロセス３ではプログラムの実行が終了し，プロセス１だけが実行を継続します（I の部分）。

2　これは fork 機能に似ています。

3　ちょっと考えると冗長実行は無駄な並列処理をしている部分のようにも思われますが，実際にプログラムを書いてみると冗長実行を行なわなければならない部分はどうしても出てくるようです。

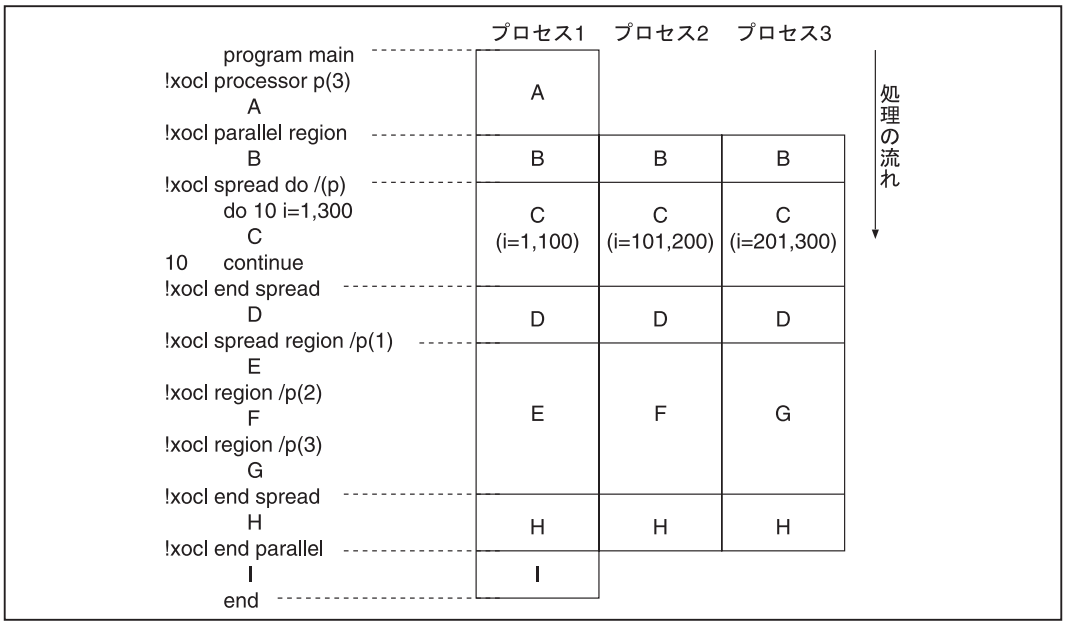


図２　並列処理の実行イメージ

６．end 文（または stop 文）の実行によりプログラムが終了します。

parallel region 文と end parallel region 文の間で実行されるプログラム部分や spread do 文（または spread region 文）と end spread do 文（または end spread region 文）の間で実行されるプログラム部分をリージョンと呼びます。特に，parallel region 文と end parallel region 文の間の部分をパラレルリージョンと呼びます。リージョンは入れ子になっていてもよくて，パラレルリージョンはすべてのリージョンの親リージョンとなります。これを図２で説明しますと，B から H までのそれぞれの部分がリージョンを形成していて，図の右半分はそれぞれのリージョンがどのプロセス上で実行されるかをあらわしていることになります[4]。また，すべてのプロセスが同一リージョンを実行することが冗長実行であることになります。

図２中の処理の流れの方向をあらわす矢印を時間軸としてみれば，各プロセスはそれぞれのリージョンの実行を同時に開始して，同時に終了していることになります。これは，プロセス間で自動的にバリア同期を取るためです。バリア同期とは，並列処理しているプロセスが同期を取るべき場所（バリアポイント）で実行を停止し，すべてのプロセスがバリアポイントに達したところで実行を再開する同期方式です。バリア同期を取る必要がなければ，それを指示することもできます。

4　正確に言うと，B，D 及び H は同一リージョン（パラレルリージョン）にあります。また，spread do 文から end spread do 文までの部分に関しては，do ループの繰り返しの１つ１つがそれぞれリージョンを形成するので 300 個のリージョンを三等分してそれぞれのプロセス上で実行することになります。

## 3　グローバル変数とローカル変数

通常，１つのプロセスは固有のメモリ空間をもち，他のプロセスからアクセスできませんが，XPFortran ではすべてのプロセスが共有するメモリ空間をもつことができます。この空間をグローバル空間と呼び，グローバル空間上に配置された変数（または配列）をグローバル変数（またはグローバル配列）と呼びます。図３に示すように，グローバル変数は DTU（data transfer unit）及びクロスバネットワークを介してすべてのプロセスからアクセスできます。また，グローバル変数はパラレルリージョンの内外でアクセスできます。

一方，グローバルとして宣言されなかった変数（または配列）をローカル変数（またはローカル配列）と呼び，図３に示すように各プロセス固有の空間（ローカル空間）上に割り付けられます。ローカル変数は他のプロセスからはアクセスできません。ローカル変数には２種類あって，複数のプロセスに分割して割り当てられるものを分割ローカル配列とよび，それぞれのプロセス上に分割されずに宣言どおり割り当てられるものを重複ローカル変数（または重複ローカル配列）と呼びます。重複ローカル変数はパラレルリージョンの外では親プロセス上の値だけが使え，パラレルリージョンにさしかかったところで親プロセス上の値が子プロセス上に複写されます。分割ローカル配列はパラレルリージョン内だけでアクセスできます。

プログラム表記上はグローバル変数の引用や定義は通常の変数（ローカル変数）の場合と同様ですが，実際にはプロセス間の通信が発生しますのでアクセス速度はローカル変数の場合に比べて遅くなります。

## 4　プロセス間のデータ転送

グローバル変数をアクセスする場合にクロスバネットワークを介したプロセス間の通信が発生

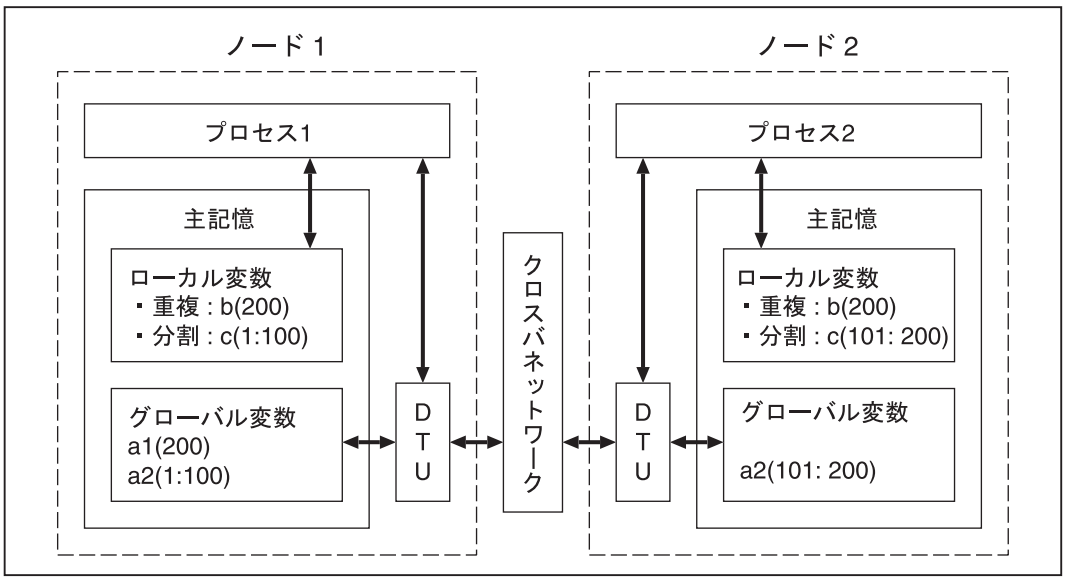


図３　グローバル変数とローカル変数

します。クロスバネットワークの最大データ転送速度は4 GB/秒です。この速度は現在ある並列計算機のプロセス間の通信速度としては速い方なのですが，それでも主記憶とCPUの間のデータ転送速度と比べると決して速くはありません。また，4 GB/秒の速度が常に出るわけではなくて，転送速度は転送するデータ量に依存します。実際の転送時間には転送のための初期化などに要するオーバーヘッドが含まれますのでデータ量が少ない転送を頻繁に行なうことは避け，できるだけまとめて転送するように心掛けましょう。

## Ⅱ．並列処理プログラムの記述法

先ほども少し説明しましたが，XPFortranではソースプログラムにXPFortran用拡張最適化制御行と呼ばれる並列処理のための指示行を挿入します。これは第1桁～第5桁が`!xocl`または`*xocl`であるもので（以下，xocl行と呼びます），第6桁は空白，そして第7桁以降に指示する内容（XPFortran構文と呼びます）を記述します[5]。指示する内容が複数行にまたがる場合には，行の最後に`&`を付けてつぎの行に継続します。xocl行は，記述される文の種類によってXPFortran宣言構文とXPFortran実行構文とに分類されます。xocl行で指定できる文を表1に示します。xocl行に関して以下のような制約があります。

- 表1に示した2種類の構文をFortranプログラム上で指定できる位置は，（Fortranの）宣言文及び実行文が指定できる位置と同じです。
- xocl行で定義される名前（後述するプロセッサグループ名など）はxocl行以外では使用できません。

表1　xocl行で指定できる文

| XPFortran宣言構文 | XPFortran実行構文 | |
|---|---|---|
| processor文 | parallel region文 | unify文 |
| proc alias文 | end parallel region文 | overlapfix文 |
| index partition文 | spread region文 | move wait文 |
| local文 | end spread region文 | broadcast文 |
| global文 | spread do文 | barrier文 |
| subprocessor文 | end spread do文 | lockon文 |
| resident文 | spread move文 | end lockon文 |
| commonid文 | end spread move文 | pass by local文 |
| | spread assignment文 | |
| | end spread assignment文 | |

### 1　並列処理プログラム例1

まず，簡単な並列処理プログラムを作ってみることにします。ここではパラメータや入力データだけを変えて，全く同じ計算を各プロセスで行なわせるという並列処理を考えます。これは最

5　xocl行の先頭は！または＊で始まりますから，他のFortran処理系では注釈行とみなされます。また，xocl行では小文字と大文字の区別はありませんので，本稿ではすべて小文字で記述してあります。

も単純な（最も粒度の粗い）並列化といえるでしょう。例題としてつぎのような２次元の拡散方程式を適当な初期条件及び境界条件の下で差分法で解くことにします。

$$\frac{\partial u}{\partial t} = \alpha\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right) \qquad (\alpha > 0) \qquad (1)$$

式（１）を，時間を前進差分，空間を中心差分でそれぞれ離散化します。$x=i\Delta x$，$y=j\Delta y$として，関数$u$（$x$，$y$，$t$）の時刻$t=k_{\Delta}t$における格子点（$i$，$j$）上の値を$u^k_{i,j}$であらわせば，差分方程式はつぎのようになります。

$$u^{k+1}_{i,j} = s_x u^k_{i+1,j}+(1-2s_x-2s_y)u^k_{i,j}+s_x u^k_{i-1,j}+s_y u^k_{i,j+1}+s_y u^k_{i,j-1} \qquad (2)$$

ただし，

$$s_x = \frac{\alpha\Delta t}{\Delta x^2}, \quad s_y = \frac{\alpha\Delta t}{\Delta y^2} \qquad (3)$$

です。

計算領域を$0\leq x\leq x_{max}$及び$0\leq y\leq y_{max}$として，$\alpha$と$\Delta t$の値の組をいくつか与え，それぞれの組について差分方程式（２）を解くことにします。$x_{max}$，$y_{max}$，$\Delta x$，$\Delta y$の値は固定します。並列処理プログラムを図４に示します。ただし，図４では初期条件及び境界条件をそれぞれ設定するサブルーチン `init` 及び `bound` は省略してあります。以下で，図４の並列処理プログラムについて説明します。

１行目　いくつかの整定数名を定義します。

２行目　複数のプロセスをまとめて扱う場合，そのまとまりをプロセッサグループと呼びます。processer 文はプログラムが使用するすべてのプロセスから成るプロセッサグループの名前と使用するプロセスの総数[6]を指定します。ここでは，`mproc` というプロセッサグループ名で５台のプロセスを使用することを宣言しています。

３行目　index partition 文は命令やデータをどのように分割するかを定義し，それを名前として宣言します。ここでは名前（分割指定名と呼びます）を `ind` と付けています。分割を指定する項目として，プロセッサグループ名，インデックス範囲（`index=` の部分），分割方法（`part=` の部分）等があります。インデックス範囲には，do ループを分割する場合には do 変数の初期値及び終値を，配列を分割する場合には寸法の下限及び上限を指定します[7]。分割方法には，均等（`band`），循環（`cyclic`），不均等の３つがあります。分割方法を省略した場合の標準値は `band` です。この index partition 文は 15 ～ 27 行目の do ループの分割を指定

6　正確にはプロセスの総数ではなくて形状と呼びます。配列の宣言と同様に多次元にすることもできるためです。図４の場合は１次元プロセッサグループということになります。

7　インデックス範囲の下限が１のときには `1:` の部分を省略できます。

```
01:       parameter (npe=5,ncase=10,imax= 100,jmax= 100,kstep= 50)
02: !xocl processor mproc(npe)
03: !xocl index partition ind=(mproc,index=1:ncase,part=cyclic)
04: c
05:       dimension u(0:imax,0:jmax),wk(0:imax,0:jmax)
06:       dimension alpha(ncase),dt(ncase)
07:       data xmax,ymax/1.0d0,1.0d0/
08:       dx=xmax/dble(imax)
09:       dy=ymax/dble(jmax)
10:       t=0.0d0
11:       read(1,500) (alpha(i),dt(i),i=1,ncase)
12: 500   format(2d10.2)
13: c
14: !xocl parallel region
15: !xocl spread do /ind
16:       do icase=1,ncase
17:         sx=alpha(icase)*dt(icase)/dx**2
18:         sy=alpha(icase)*dt(icase)/dy**2
19:         call init(imax,jmax,u,xmax,ymax,alpha(icase),t)
20:         do k=1,kstep
21:           call bound(imax,jmax,u,xmax,ymax,alpha(icase),t)
22:           call diffus(imax,jmax,u,sx,sy,wk)
23:           t=t+dt(icase)
24:         end do
25:         write(icase+10) alpha(icase),dt(icase),u
26:       end do
27: !xocl end spread do
28: !xocl end parallel region
29:       end
30: c
31:       subroutine diffus(imax,jmax,u,sx,sy,wk)
32:       dimension u(0:imax,0:jmax),wk(0:imax,0:jmax)
33:       do j=1,jmax-1
34:         do i=1,imax-1
35:           wk(i,j)=sx*u(i+1,j)+(1.0d0-2.0d0*sx-2.0d0*sy)*u(i,j)
36:      *     +sx*u(i-1,j)+sy*u(i,j+1)+sy*u(i,j-1)
37:         end do
38:       end do
39: c
40:       do j=1,jmax-1
41:        do i=1,imax-1
42:          u(i,j)=wk(i,j)
43:        end do
44:       end do
45:       end
```

⋮

図４　並列処理プログラム例１

するためのものですが，cyclic を指定すると，do ループの 1 回目をプロセス 1 で，2 回目をプロセス 2 で，...，5 回目をプロセス 5 で，6 回目をプロセス 1 で，...，というようにサイクリックに処理が割り当てられます。

5 ～ 12 行目　配列の宣言とパラメータの設定を行ないます。配列 wk はサブルーチン diffus で使う作業領域です。$\alpha$と$\Delta t$は装置参照番号 1 から読み込みます（11 行目）。この部分は逐次実行ですから親プロセスだけが処理を行ないます。

14 行目　parallel region 文にさしかかると並列処理が始まります。親プロセス上のプログラムが子プロセス上に複写され，並列処理の開始直後は各プロセスの変数や配列はそれぞれ同じ値を持ちます。だたし，この段階では配列 u などは未定義のままです。図 4 中にあらわれるすべての変数及び配列は各プロセス上に重複して割り当てられるので重複ローカル変数となります。

15 行目　spread do 文は do ループを分割する命令です。3 行目で宣言した分割指定 ind にしたがって，直後の do ループ（16 ～ 26 行目）を分割して処理します。

16 ～ 26 行目　入力した$\alpha$と$\Delta t$の各組について，指定した計算ステップ数（kstep の値）まで計算して結果をファイルに出力します。この部分は各プロセスが与えられた$\alpha$と$\Delta t$の値を使って同一の処理を行ないます。

27 行目　end spread do 文[8]は 15 行目の spread do 文に対応していて，分割した do ループの直後に置きます。

28 行目　end parallel region 文[9]は並列処理の終了を指示します。

31 ～ 45 行目　サブルーチン diffus は差分式（2）を計算する部分です。diffus には xocl 行は挿入されていません[10]。

図 4 に示したように，パラメータや入力データだけを変えて各プロセスで同一の処理を行なうならば，xocl 行を挿入する箇所も少なく，比較的容易に並列化できます。また，プロセス間の通信もほとんど発生しませんから処理速度は使用するプロセス数にほぼ比例して伸びていくことが期待できます。

## 2　配列の分割

つぎに，配列を分割して並列処理する場合のプログラミングを簡単な例を使って説明します。まず，寸法 30 の 1 次元配列 a，b，c をローカル配列として宣言し，それぞれの配列を 3 等分して 3 台のプロセスに分割して割り当てることを考えます。プログラムはつぎのようになります。

8　end spread do の do は省略可能です。

9　end parallel region の region は省略可能です。

10　図 4 では省略されているサブルーチン init 及び bound にも xocl 行は挿入されていません。

```
!xocl processor p(3)
!xocl index partition ind=(p,index=1:30,part=band)
      dimension a(30),b(30),c(30)
!xocl local a(/ind),b(/ind),c(/ind)
                ⋮
!xocl spread do /ind
      do i=1,30
        c(i)=a(i)+b(i)
      end do
!xocl end spread do
```

local 文はローカル変数であることを宣言します[11]。ローカル配列を分割する場合には（分割ローカル配列）分割指定 ind を使って，a(/ind) のような形式で指定します。分割方法は band ですから，インデックス範囲を均等に分割します。したがって，図５に示すようにプロセス１上には a(1) ～ a(10) が，プロセス２上には a(11) ～ a(20) が，プロセス３上には a(21) ～ a(30) がそれぞれ配置されます。配列 b 及び c も同様に分割して配置されます。後に続く do ループの分割でも spread do 文で分割指定 ind を使っています。この分割によって，プロセス１上では do 変数 i の値が 1 ～ 10 の範囲を，プロセス２上では 11 ～ 20 の範囲を，プロセス３上では 21 ～ 30 の範囲をそれぞれ処理することになります。

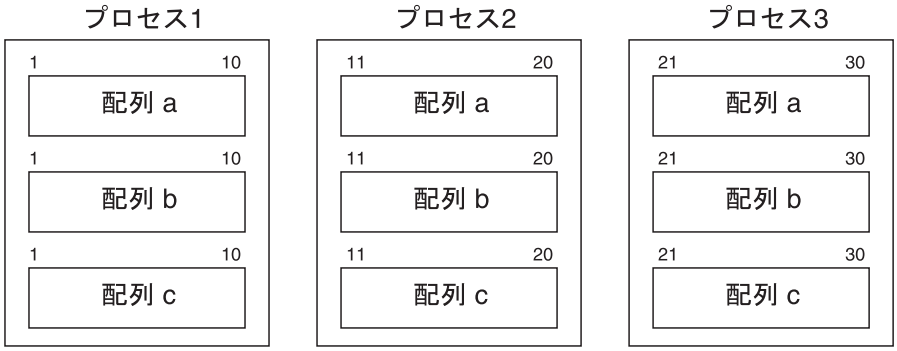


図５　ローカル配列の分割

この例では，a, b, c はローカル配列ですから，例えば，プロセス１から a(11) ～ a(30) のデータをアクセスすることはできません。もし，他のプロセス上にあるデータをアクセスする必要があるならば，上のプログラムの宣言部分をつぎのように書き換えます。

11　local 文にも global 文（後述）にもあらわれない変数や配列はすべて重複ローカル変数になります。

```
!xocl processor p(3)
!xocl index partition ind=(p,index=1:30,part=band)
      dimension a(30),b(30),c(30)
      dimension ag(30),bg(30),cg(30)
!xocl local a(/ind),b(/ind),c(/ind)
!xocl global ag,bg,cg
      equivalence(a,ag),(b,bg),(c,cg)
```

global 文はグローバル変数であることを宣言します[12]。よって, ag はグローバル配列になります。また，equivalence 文によってグローバル配列 ag はローカル配列 a と同一の記憶領域を共有することになります。したがって，global 文には分割指定がありませんが[13]，図６に示すように配列 ag は３等分されて各プロセス上に配置されます。配列 bg, cg も同様です。このようにすると，例えば，プロセス１上の計算において

```
c(1)=ag(11)+bg(30)
```

といった記述が可能になります[14]。しかし，ローカル変数と同一の記憶領域を共有していても ag, bg, cg はグローバル変数ですから, アクセス速度はローカル変数に比べて遅くなります。よって, できる限りローカル変数へのアクセスだけで計算を済ませるように配慮する必要があります。

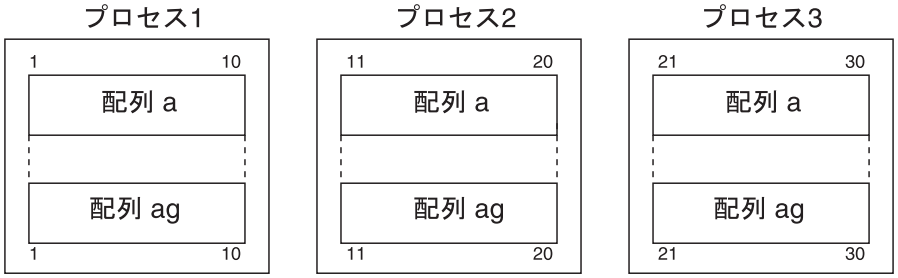


図６　ローカル配列とグローバル並列の記憶領域の共有

つぎに，下のような do ループを３等分して，３台のプロセスを使って並列処理することを考えます。

12　global 文に指定できる変数の型は基本整数型，８バイト整数型，基本論理型，８バイト論理型，基本（単精度）実数型, 倍精度実数型, 4 倍精度実数型, 基本（単精度）複素数型, 倍精度複素数型，４倍精度複素数型です。

13　!xocl global ag(/ind), bg(/ind), cg(/ind) のように書くこともできます。

14　ag(11) 及び bg(30) はそれぞれ a(11) 及び b(30) と同じ記憶領域を占めています。しかし，c(1)=a(11)+b(30) という記述は許されません。

```
dimension a(30),b(30)
        ⋮
do i=2,29
b(i)=a(i-1)+a(i)+a(i+1)
end do
```

配列 a 及び b は先ほどと同様に 3 等分して 3 台のプロセス上に分割ローカル配列として配置します。また，do ループは，プロセス 1 上では do 変数 i の値が 2 ～ 10 の範囲を，プロセス 2 上では 11 ～ 20 の範囲を，プロセス 3 上では 21 ～ 29 の範囲をそれぞれ分割して処理することになります。ところが，変数 i の値が 10 のときの計算では a(i+1) すなわち a(11) はプロセス 2 上にありますから，プロセス 1 からはアクセスできません。これは変数 i の値が 11，20，21 のときにも同様なことが起こります。このように配列を複数のプロセス上に分割することによって「泣き別れ」になる部分のことを袖と呼びます。袖を含む計算では，先ほどのように equivalnce 文を使ってグローバル配列と結合してやればよいのですが，これに加えてもう一工夫します。

上の例に xocl 行を挿入したプログラムは図 7 のようになります。以下で，図 7 について説明します。

```
01: !xocl processor p(3)
02: !xocl index partition ind1=(p,index=1:30,part=band,overlap=(1,1))
03: !xocl index partition ind2=ind1(overlap=(0,0))
04:       dimension a(30),ag(30),b(30)
05: !xocl local a(/ind1),b(/ind2)
06: !xocl global ag
07:       equivalence (a,ag)
            ⋮
08: !xocl overlapfix(a) (id)
09: !xocl move wait (id)
10: !xocl spread do /ind2
11:       do i=2,29
12:         b(i)=a(i-1)+a(i)+a(i+1)
13:       end do
14: !xocl end spread do
```

図 7　袖付きローカル配列の分割例

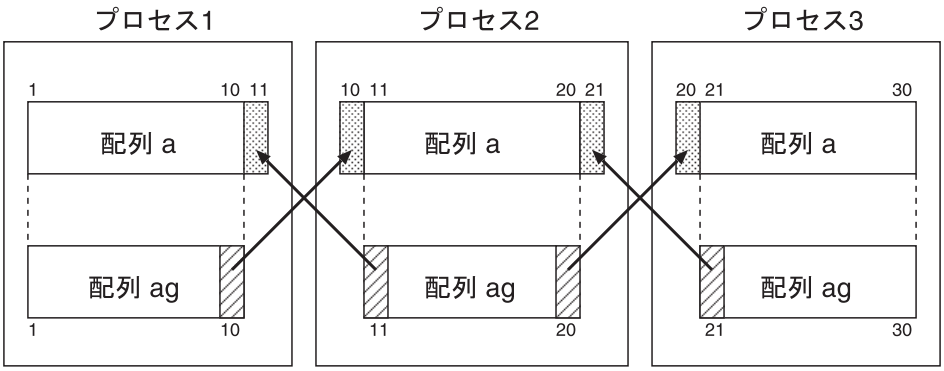


図 8　ローカル配列 a とグローバル配列 ag の分割のイメージ

2行目　　分割指定 ind1 の項目の中の overlap= の部分が袖をどれだけ伸ばすかを指定するものです[15]。overlap=（*l*, *r*）は，インデックス範囲を指定された分割方法[16]で分割した結果に，インデックスが減少する方向へ *l* 個，また，インデックスが増加する方向へ *r* 個伸ばした範囲を割り当てることをあらわしています。よって，図８に示すように，ローカル配列 a はプロセス１上には a(1) ～ a(11) が，プロセス２上には a(10) ～ a(21) が，プロセス３上には a(20) ～ a(30) がそれぞれ割り当てられます。袖部分は隣接するプロセス上に重複して割り当てられることになります。グローバル配列 ag は袖付きの分割ローカル配列 a と equivalence 文で結合しています（７行目）が，図８のように ag は袖を持ちません。図８中の各プロセス上の配列 a の網掛けをした部分が袖になります。また，各プロセス上の配列 a と ag を点線で結んだ部分が互いに共有する記憶領域になります。

3行目　　すでに定義した分割指定（ここでは ind1）の一部を変更して新たな分割指定（ここでは ind2)を宣言する方法です。分割指定 ind2 は袖なしの均等分割になります。

8行目　　プロセス１上の配列要素 a(11)（これは袖です）とプロセス２上の a(11)（これは配列要素 ag(11) と記憶領域を共有しています）は異なるプロセス上に配置されていますから，計算の途中のある時点ではどちらか一方が未定義であったり，互いに異なる値を持ったりする可能性があります[17]。この不整合を解消するための命令が overlapfix 文です。overlapfix 文は，指定された袖付き分割ローカル配列（ここでは配列 a）の袖部分の値を，equivalence 文で結合されたグローバル配列（ここでは配列 ag）の同じ添字を持つ要素の値に一致させるためにデータ転送の開始を指示します[18]。これを図８を使って説明しますと，配列 ag の斜線を引いた部分（要素）の値を矢印で結んだ先の配列 a の袖部分に複写することになります。これによって，例えばプロセス１上の a(11) の値はプロセス２上の ag(11) の値に置き換わります。

9行目　　overlapfix 文は袖部分の値の置き換えの開始を指示する命令ですから，データ転送の終了を待たずに後続の文の実行に移ります。したがって，overlapfix 文の直後に袖の値を参照する文を書いてもまだデータ転送が終了していない可能性があります。データ転送が完了したことを確認する命令が move wait 文です[19]。overlapfix 文とこれに対応する move wait 文を識別するために，２つの文に同じ

15　袖付きの分割は，local 文，global 文，spread do 文，spread move 文，spread assignment 文で指定できます。

16　part=cyclic とした場合には袖は指定できません。

17　これは実際に起こります。

18　当然，データ転送を開始する時点では equivalence 文で結合されたグローバル配列の持つ値が正しい値であることを前提にしています。

19　move wait 文の位置でバリア同期がとられます。

プロセッサ間データ転送識別名（ここでは id）を付けます[20]。

10 ～ 14 行目　spread do 文では分割指定 ind2 を指定します（ind1 を指定するのは誤りです）。do ループは分割されて，プロセス 1 上では do 変数 i の値が 2 ～ 10 の範囲を，プロセス 2 上では 11 ～ 20 の範囲を，プロセス 3 上では 21 ～ 29 の範囲をそれぞれ処理することになります[21]。

## 3　並列処理プログラム例 2

配列を分割した並列処理プログラムの例題として，先ほどの 2 次元拡散方程式（1）を差分法で解くことを考えます。ただし，入力となるパラメータは 1 組だけにして，計算領域を分割して処理します。並列処理プログラムの主プログラム部分を図 9 に，サブルーチン diffus の部分を図 10 にそれぞれ示します。

以下で，図 9 及び図 10 の並列処理プログラムについて説明します。

4 ～ 5 行目　この index partition 文は配列 u の分割を指定します。4 行目の最後の & はこの xocl 行がつぎの行に継続することをあらわしています。インデックス範囲を index=0:jmax としていますから，配列 u の 2 次元目（*y* 軸方向）を分割します。分割方法は band ですから均等分割になりますが，配列 u の 2 次元目の寸法は 101 なのでプロセス数の 5 で割ると端数がでます。この場合には，商に 1 を足した要素数を順に割り当てていって，最後に残りの要素数をそのつぎのプロセスに割り当てます。101 を 5 で割ったときの商は 20 ですから，最初の 4 プロセスではそれぞれ 21 要素，最後の 1 プロセスでは 17 要素となるように分割されます[22]。この分割指定では袖も指定していますが，差分式（2）から袖の指定が必要になることはお分かり頂けると思います。

7 行目　分割する配列が多次元の場合，この local 文にあるように分割しない次元（この場合 1 次元目）にはコロン（:）を置きます。

9 行目　ローカル配列 u とグローバル配列 ug を equivalence 文で結合します。

10 行目　グローバル配列と equivalence 文で結合されたローカル配列は common 文に指定できないので，グローバル配列 ug を指定してプログラム単位間のデータの結合を行ないます[23]。

---

20　データ転送の開始と終了の確認が 2 つの文に分かれていますから，2 つの文の間にデータ転送とは無関係な処理を行なう命令を書き込むこともできます。ただし，通常は overlapfix 文と movewait 文は同じプログラム単位内になければなりません。

21　spread do 文に分割指定 ind1 を指定すると，各プロセス上で袖を含む繰り返し範囲を処理することになります。

22　4 × (20+1)+17=101 です。

23　一般には，common 文にはローカル変数，グローバル変数のいずれも指定できます。ただし，重複ローカル変数，分割ローカル配列，グローバル変数が 1 つの共通ブロックに混在することはできません。

```
01:       parameter(kstep= 50)
02:       parameter (npe=5,imax= 100,jmax= 100)
03: !xocl processor mproc(npe)
04: !xocl index partition ind1=(mproc,index=0:jmax, &
05: !xocl        part=band,overlap=(1,1))
06:       dimension u(0:imax,0:jmax),ug(0:imax,0:jmax)
07: !xocl local u(:,/ind1)
08: !xocl global ug
09:       equivalence (u,ug)
10:       common /data/ ug
11: c
12:       data xmax,ymax/1.0d0,1.0d0/
13:       dx=xmax/dble(imax)
14:       dy=ymax/dble(jmax)
15:       dt=2.0d0
16:       alpha=1.0d-5
17:       sx=alpha*dt/dx**2
18:       sy=alpha*dt/dy**2
19:       t=0.0d0
20: c
21: !xocl parallel region
22:       call init(xmax,ymax,alpha,t)
23:       do k=1,kstep
24:         call bound(xmax,ymax,alpha,t)
25: c
26: !xocl overlapfix(u) (id)
27: !xocl move wait (id)
28: c
29:         call diffus(sx,sy)
30:         t=t+dt
31:        end do
32: c
33: !xocl end parallel region
34:       write(1) ug
35:       end
```

図９　並列処理プログラム例２－１（主プログラム）

```
36:       subroutine diffus(sx,sy)
37:       parameter (npe=5,imax= 100,jmax= 100)
38: !xocl processor mproc(npe)
39: !xocl subprocessor sproc=mproc(1:npe)
40: !xocl index partition ind1=(sproc,index=0:jmax, &
41: !xocl         part=band,overlap=(1,1))
42:       dimension u(0:imax,0:jmax),ug(0:imax,0:jmax)
43: !xocl local u(:,/ind1)
44: !xocl global ug
45:       equivalence (u,ug)
46:       common /data/ ug
47: c
48: !xocl index partition ind2=(sproc,index=0:jmax,part=band)
49:       dimension wk(0:imax,0:jmax)
50: !xocl local wk(:,/ind2)
51: c
52: !xocl spread do /ind2
53:       do j=1,jmax-1
54:         do i=1,imax-1
55:           wk(i,j)=sx*u(i+1,j)+(1.0d0-2.0d0*sx-2.0d0*sy)*u(i,j)
56:      *      +sx*u(i-1,j)+sy*u(i,j+1)+sy*u(i,j-1)
57:         end do
58:       end do
59: !xocl end spread do
60: c
61: !xocl spread do /ind2
62:       do j=1,jmax-1
63:        do i=1,imax-1
64:          u(i,j)=wk(i,j)
65:        end do
66:       end do
67: !xocl end spread do
68:       end
```

図 10　並列処理プログラム例２－２（サブルーチン diffus）

12 ～ 19 行目　パラメータの設定。

21 行目　パラレルリージョンの始まり。

26 ～ 27 行目　計算ステップごとに，サブルーチン diffus を呼び出す前に分割ローカル配列 u の袖部分の値をグローバル配列 ug の対応する要素の値に一致させます。

29 行目　サブルーチン diffus を呼び出します。配列 u（正確には ug）は共通ブロックでサブルーチン側に引き渡されます[24]ので，図 4 の並列処理プログラム例 1 の中の diffus と比べて引数の並びが異なっています。

33 行目　パラレルリージョンの終り。

34 行目　計算結果をファイルに出力します。write 文にグローバル配列 ug を指定しています。グローバル配列はパラレルリージョンの外側（逐次実行時）でもアクセスできます。

36 行目　サブルーチン diffus の始まり（ここから図 10 を御覧ください）。図 4 の並列処理プログラム例 1 の場合と異なり，サブルーチン内部も分割して処理するので xocl 行の挿入が必要になります。

37 ～ 46 行目　この宣言部分は主プログラムの宣言部分（図 9 の 2 ～ 10 行目）とよく似ていますが，39 行目に主プログラムにはない subprocessor 文があります。実引数と仮引数の関係に類似して，プロセッサグループには実プロセッサグループと仮プロセッサグループがあります。subprocessor 文は仮プロセッサグループの名前とその形状を宣言します。39 行目では，仮プロセッサグループ sproc は実プロセッサグループ mproc の 1 ～ 5 番目のプロセスと結合することをあらわしています[25]。このサブルーチン内では，40 ～ 41 行目や 48 行目の index partition 文にあるようにプロセッサグループには仮プロセッサグループ名を使います。

48 ～ 50 行目　作業領域 wk を（袖なしの）分割ローカル配列として宣言しています。

52 ～ 59 行目　spread do 文の直後の do ループ（ここでは do 変数 j に関するループ）が分割されます。分割指定は ind1 ではなくて ind2 を指定していることに注意してください。

図 9 及び図 10 には示されていませんが，サブルーチン init 及び bound にも diffus と同

---

24　共通ブロックを使わずに，分割ローカル配列やグローバル配列を引数に指定することも可能です。ただし，この場合には分割ローカル配列 u をサブルーチン側で equivalence 文を使ってグローバル配列と結合させることはできなくなります。Fortran の規約上，仮引数に指定した変数名や配列名を equivalence 文に指定することができないからです。

25　38 行目の processor 文にあるように mproc はプログラムで使用するすべて（5 つ）のプロセスで構成されていますから，結果的に mproc というプロセッサグループ名に sproc という別名を付けたことにもなります。このような指定の仕方をプロセッサグループ固定手続と呼びます。他にプロセッサグループ大きさ固定手続とプロセッサグループ大きさ引継ぎ手続があります。

様な xocl 行が挿入されています[26]。

この例題では２次元配列 u の２次元目を分割していますが，多次元配列の分割の仕方は全体の処理速度に影響を与えることがあります。プロセス間の通信を少なくすることと，各プロセスの処理性能を引き出すことを考えて命令及び配列を分割することが基本でしょう。

## ４　グローバル変数・ローカル変数間のデータ転送

すでに説明したようにグローバル変数はどのプロセスからもアクセスできますが，プロセス間の通信が発生しますからアクセス速度はローカル変数に比べて遅くなります。アクセスするグローバル変数が同じノードの主記憶上にある場合にはクロスバネットワークを介した通信は起きませんが，通常の主記憶へのアクセスとは異なる[27]ためやはりアクセス速度は遅くなります。このためグローバル変数を使って計算などの処理を行なう場合には，グローバル変数を一旦ローカル変数に転送し，そのローカル変数を使って処理を行なう方が効率的です[28]。グローバル変数とローカル変数との間のデータ転送には spread move 文を使います[29]。

### ４．１　データ転送例１

２次元のグローバル配列をローカル配列に複写するにはつぎのようにします。

```
!xocl processor p(2)
!xocl index partition ind=(p,index=1:100,part=band)
      dimension a(100,100),b(100,100)
!xocl global a(:,/ind)
!xocl local b(:,/ind)
        ⋮
!xocl spread move /ind,:
      do j=1,100
        do i=1,100
          b(i,j)=a(i,j)
        end do
      end do
!xocl end spread move
```

26　特に宣言部分（図 10 では 37 ～ 46 行目）は各サブルーチンで共通になりますから，include 文を使うと便利です。

27　グローバル変数へのアクセスでは DTU を経由します。

28　ここではグローバル変数とローカル変数とを equivalence 文で結合することができない場合や結合できても不都合が生じる場合を考えています。

29　spread move 文を使うとデータは一括して転送されます。

グローバル配列 a 及びローカル配列 b はいずれも 2 次元目が分割されていますから，図 11 に示すように 2 つのプロセス上に配置されます。これを図 11 のようにグローバル配列 a(1:100,1:50) の値をプロセス 1 上の分割ローカル配列 b(1:100,1:50) に，a(1:100,51:100) の値をプロセス 2 上の b(1:100,51:100) にそれぞれ転送するプログラムです。

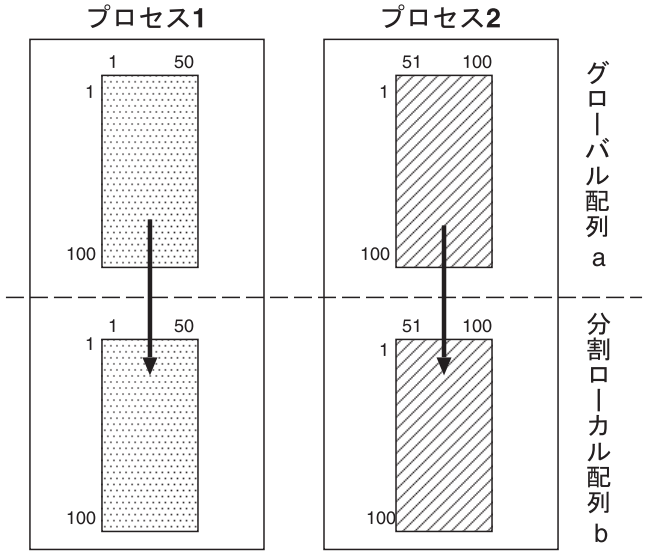


図 11　spread move 文による転送（その 1）

spread move 文から end spread move 文[30] までの間にある do ループのそれぞれについてどのように分割するかを spread move 文で指定します[31]。この例では do 変数 j のループを分割し，do 変数 i のループは分割しませんから，外側にある do ループから順番に対応させて /ind,: と指定します。分割指定と分割指定の間はコンマ（,）で区切ります。コロンを指定すると対応する do ループは分割されません。これによって例えば，プロセス 1 上では do 変数 j が 1 ～ 50 の範囲，i が 1 ～ 100 の範囲を処理することになります[32]。

上のプログラムの最後の end spread move 文を

```
!xocl end spread move(id)
!xocl move wait(id)
```

30　end spread move の move は省略できます。

31　spread do 文は直後の do ループだけを分割します。

32　この例ではグローバル配列 a とローカル配列 b の分割の方法は同じですからプロセス間の通信は起こらず，同じプロセス上でのデータ転送になります。

と記述することもできます。これはoverlapfix文と同様にspread move文で転送の開始を指示し，転送の完了はend spread move文で指定したプロセッサ間データ転送識別名（この場合id）と同じ名前が指定されたmove wait文によって確認することを指定します[33]。上のプログラムのようにend spread move文で（id）を省略すると，対応するmove wait文はend spread move文の直後にあるものとみなされます。

ローカル配列bからグローバル配列aへデータ転送する場合には上の例の代入文の左辺と右辺を入れ換えて，

```
a(i,j)=b(i,j)
```

とします。

配列代入を使って表現する場合には，上のプログラムのspread move文からend spread move文までを，

```
!xocl spread move :,/ind
      b=a
!xocl end spread move
```

とします。spread move文の分割指定の:,/indが上のプログラムと反対になっていることに注意してください。これはローカル配列bの分割指定に合わせているためです。

### ４．２　データ転送例２

この例も２次元のグローバル配列をローカル配列に複写する例です。ただし，グローバル配列aは２次元目が，ローカル配列bは１次元目がそれぞれ分割されています。

```
!xocl processor p(2)
!xocl index partition ind=(p,index=1:100,part=band)
      dimension a(100,100),b(100,100)
!xocl global a(:,/ind)
!xocl local b(/ind,:)
        ⋮
!xocl spread move :,/ind
      do j=1,100
        do i=1,100
          b(i,j)=a(i,j)
```

33　end spread move文とmove wait文の間にデータ転送とは無関係な処理を行なう命令を書き込むこともできます。

```
        end do
      end do
!xocl end spread move
```

この例では図12のような転送が行われます。例えば，プロセス1上ではdo変数jが1～100の範囲，iが1～50の範囲を処理することになります[34]。この例と前の例とを比べるとお分かりいただけるかと思いますが，処理を割り当てられたプロセスが，処理する範囲にあるすべての配列要素にアクセスできる必要がありますから，ローカル配列の分割指定に合わせてspread move文の分割指定をすることになります。

## 4．3　データ転送例３

グローバル配列を重複ローカル配列に複写する場合にはつぎのようになります。

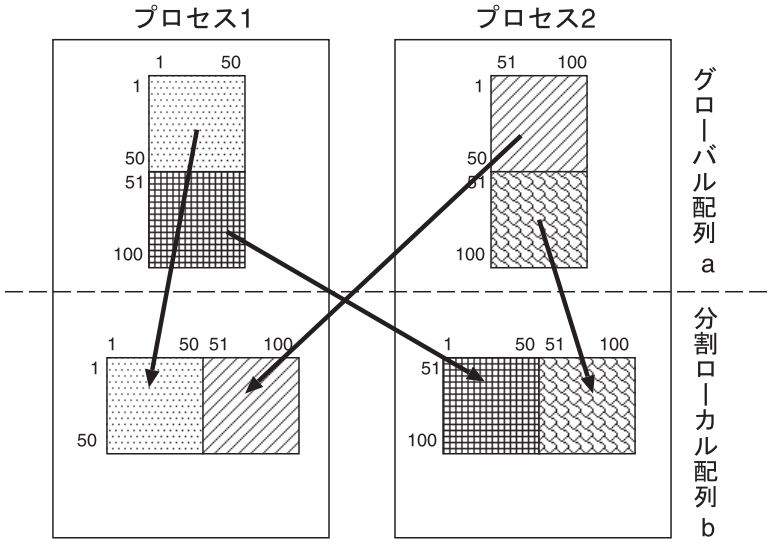


図12　spread move文による転送（その２）

```
!xocl processor p(2)
!xocl index partition ind=(p,index=1:100,part=band)
      dimension a(100,100),b(100,100)
!xocl global a(:,/ind)
        ⋮
!xocl spread move
      do j=1,100
```

34　この場合にはプロセス間の通信が発生します。

```
        do i=1,100
          b(i,j)=a(i,j)
        end do
      end do
!xocl end spread move
```

この例では図 13 のような転送が行われます。a はグローバル配列，b は重複ローカル配列です。この場合のように spread move 文に分割指定が省略されている場合にはすべてのプロセスがすべての do ループの範囲を処理します[35]。

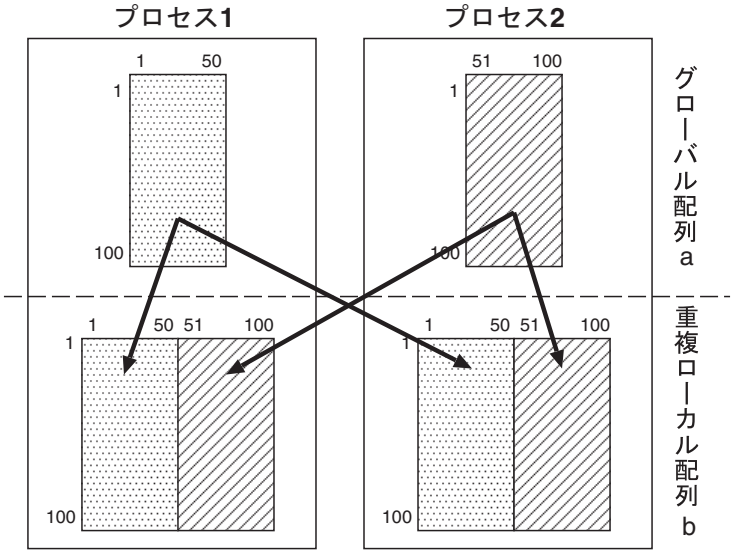


図 13　spread move 文による転送（その３）

一般に，spread move 文と end spread move 文の間にあらわれる do ループ中の代入文は

$$a(i_1,i_2,\ldots,i_m,\ldots)=b(j_1,j_2,\ldots,j_n,\ldots)$$

の形で，以下のような制約があります。

1．配列 a 及び b は同じ型の配列でなければなりません。

2．添字 $i_m$，$j_n$ はつぎの３つの形式のいずれかでなければなりません。

- $b$（do 変数を含まない整数式）
- ［+ または −］［$a$*］j［+$b$ または − $b$］（do 変数を含む整数式）
- list(j)（リストベクトル）

ただし，j は do 変数，$a$ 及び $b$ は do 変数を含まない整定数，整数型変数または整数式のい

35　この場合にはすべてのプロセス上で do 変数 j が 1 ～ 100 の範囲，i が 1 ～ 100 の範囲を処理します。

ずれかです。list は基本整数型の 1 次元重複ローカル配列です。［と］で囲まれた部分は省略できます。do 変数を含む整数式では，例えば do 変数を i として，2*i+1 は正しいですが，1+2*i は誤りです。

3．添字の並び（$i_1,i_2,\ldots,i_m,\ldots$）に同じ do 変数を 2 回以上指定することはできません。（$j_1,j_2,\ldots,j_n,\ldots$）も同様です。

4．添字の並び（$i_1,i_2,\ldots,i_m,\ldots$）にはすべての do 変数があらわれなければなりません。（$j_1,j_2,\ldots,j_n,\ldots$）も同様です。

## 5　グローバル変数の resident 指定

resident 指定はグローバル変数をローカル変数と同様に（高速に）アクセスできるようにする機能です。resident 指定されたグローバル変数はその変数をアクセスするプロセス上に割り当てられている必要があります。すでにでてきた図 7 を resident 指定で書き換えた図 14 を使って以下で説明します。

```
01: !xocl processor p(3)
02: !xocl index partition ind1=(p,index=1:30,part=band,overlap=(1,1))
03: !xocl index partition ind2=ind1(overlap=(0,0))
04:       dimension ag(30),b(30)
05: !xocl local b(/ind2)
06: !xocl global ag(/ind1)
              ⋮
07: !xocl overlapfix(ag) (id)
08: !xocl move wait (id)
09: !xocl spread do resident(ag) /ind2
10:       do i=2,29
11:         b(i)=ag(i-1)+ag(i)+ag(i+1)
12:       end do
13: !xocl end spread do
```

図 14　グローバル変数の resident 指定

4～5 行目　　図 7 では分割ローカル配列 a を宣言しましたが，ここでは使用しません。

6 行目　　ag を袖付きのグローバル配列として宣言します。

7 行目　　グローバル配列 ag の袖部分の転送を行います。図 15 に示すように ag の袖（図

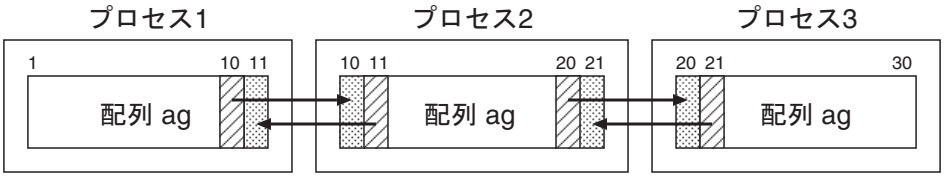


図 15　グローバル配列 ag の袖の転送

15 の網掛け部分）を対応する袖でない部分（図 15 の斜線部分）で置き換えます。例えば，プロセス 2 上の ag(10) の値はプロセス 1 上の ag(10) で置き換られます。

9 ～ 13 行目　9 行目の spread do 文に resident 指定をしています。これにより 10 ～ 12 行目の do ループにあらわれる配列 ag はローカル変数と同様の方法でアクセスすることができます。

resident 指定すると分割ローカル配列と equivalence 文で結合する必要がなくなるので，並列プログラムへの書き換えの作業量が軽減される場合があります。resident 指定は parallel region 文，spread region 文，spread do 文，spread move 文，spread assignment 文に指定できます。また，resident 文を使って宣言するとプログラム単位全体で有効となります。

## 6　重複ローカル配列の転送

同一の重複ローカル配列に対して，各プロセス上で別々の部分を定義したものを各プロセスからすべてのプロセスに転送するには unify 文を使います。簡単な使用例を以下に示します。

```
!xocl processor p(2)
!xocl index partition ind= (p,index=1:10,part=band)
      dimension a(10,10)
              ⋮
!xocl spread do /ind
      do j=1,10
        do i=1,10
          a(i,j)=i*j
        end do
      end do
!xocl end spread do
!xocl unify(a(:,/ind))(id)
!xocl move wait(id)
```

この例では，重複ローカル配列 a に関して，do ループにおいて a(1:10,1:5) の部分をプロセス 1 で，a(1:10,6:10) の部分をプロセス 2 でそれぞれ定義しています。これは図 16 中の転送前の状態になります。つぎに，各プロセスで定義した部分を他のプロセスに転送するように分割指定をして，unify 文を実行します。unify 文も転送の開始を指示する命令ですから，move wait 文によってデータ転送の完了を確認する必要があります。これによって，配列 a の全要素は各プロセス上ですべて同じ値を持つことになります（図 16 中の転送後の状態）。

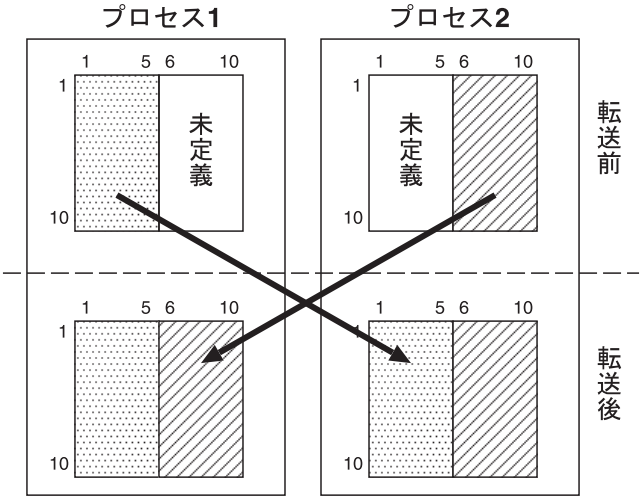


図 16　unify 文による転送

## 7　プログラムの分割

プログラム中のある部分の処理を 1 つのプロセス（または複数のプロセス）上で行ない，プログラムの別の部分の処理を別のプロセス上で行なう場合には spread region 文を使います。つまり，spread region 文はプログラムをいくつかのリージョンに分割し，それぞれのリージョンを 1 つまたは複数のプロセス [36] に割り当てる機能を持つ文です。使い方は以下のようです。

```
!xocl processor p(10)
              ⋮
!xocl spread region /p(1)
              ⋮  (処理 A)
!xocl end spread region
```

リージョンの分割は spread region 文で始まり，end spread region 文 [37] で終ります。この例では処理 A をプロセッサグループ p の 1 番目のプロセス上で行なうことを指定しています。リージョンを割り当てられなかったプロセス（この場合にはプロセス 2 ～プロセス 10）はプロセス 1 が処理 A を終えるまで待ちます [38]。つぎの例では処理 A, B, C を並列処理するために 3 つのリージョンに分割しています。

36　分割したリージョンの 1 つをさらに複数のリージョンに分割する場合などにはそのリージョンに複数のプロセスを割り当てる必要があります。

37　`end spread region` の `region` は省略できます。

38　バリア同期を取ります。

```
!xocl processor p(10)
!xocl proc alias p456=p(4:6)
              ⋮
!xocl spread region /p(1)
              ⋮ （処理 A）
!xocl region /p(2:3)
              ⋮ （処理 B）
!xocl region /p456
              ⋮ （処理 C）
!xocl end spread region
```

２つ目以降のリージョンの指定には region 文を使います。プロセス１上では処理 A を，プロセス２及びプロセス３上では処理 B を，プロセス４～プロセス６上では処理 C を実行します。ただし，proc alias 文を使ってプロセッサグループｐの４～６台目の３つのプロセスにプロセッサグループ p456 という別名を付けています。

## ８　配列式の並列化

単純 where 文，配列式を使った代入文を並列処理するには spread assignment 文を使います。簡単な使用例を以下に示します。

```
!xocl processor p(10)
!xocl index partition ind=(p,index=1:100,part=band)
      dimension a(100),b(100),c(100)
!xocl local a(/ind),b(/ind),c(/ind)
              ⋮
!xocl spread assignment /ind
      where(a.ge.0.0)b=...
      where(a.lt.0.0)b=...
      c=a*b
!xocl end spread assignment
```

この例のように spread assignment 文と end spread assignment 文[39] の間に並列処理したい単純 where 文や配列式を使った代入文を記述します。

39　end spread assignment の assignment は省略できます。

## 9　グローバル関数

do ループを使って総和や最大値・最小値等を求めることはしばしば起こります。その do ループが分割されている場合に利用するのがグローバル関数です。グローバル関数には sum（総和），max（最大値），min（最小値），and（論理積），or（論理和），last（最終値）[40] の 6 つがあります。使い方はつぎのようになります。

```
!xocl spread do /ind
      do i=1,n
              ⋮
        arg =...
              ⋮
      end do
!xocl end spread do g-func(arg)[,g-func(arg)...]
```

*g-func* の位置に 6 つのグローバル関数のいずれかを指定します。また，コンマで区切って複数のグローバル関数を指定することもできます。引数 *arg* には変数名または配列名が指定できますが，いずれも重複ローカル変数でなければなりません。各プロセスに割り当てられた do ループの繰り返し範囲の処理が終了した時点でグローバル関数の引数の値がプロセス間で評価され，引数の値はその結果に置き換えられます。引数が配列の場合には要素ごとに評価されて要素の値が結果に置き換えられます。引数として指定できる型は，sum が整数型，実数型及び複素数型，max と min が整数型及び実数型，and と or が論理型，last が整数型，実数型，複素数型及び論理型です。例えば，分割ローカル配列 z（寸法 1000 の 1 次元配列）の要素の総和を求めるにはつぎのようにします。

```
      s=0.0d0
!xocl spread do /ind
      do i=1,1000
        s=s+z(i)
      end do
!xocl end spread do sum(s)
```

この do ループを実行すると重複ローカル変数 s の値は配列 z の要素の総和になります。

グローバル関数 max 及び min に引数として変数名を指定する場合には，最大 8 個までの引数

40　last の引数として指定した重複ローカル変数について，最後のリージョンにおける値を返します。例えば，ある定数と等しい値を持つ配列要素の添字の最大のものを探すときに使います。

を指定することができます。ただし，第２引数以下は整数型でなければなりません。第２引数以下に関しては，プロセスの中で第１引数が最大値（または最小値）を与えたプロセス上の第２引数以下の値がそのまま他のプロセス上の対応する引数に複写されます。例えば，分割ローカル配列 z の要素の中の最大値と最小値及びその要素番号を求めるにはつぎのようにします。

```
      zmax=-1.0d5
      zmin= 1.0d5
!xocl spread do /ind
      do i=1,1000
        if(z(i).gt.zmax)then
          zmax=z(i)
          nmax=i
        endif
        if(z(i).lt.zmin)then
          zmin=z(i)
          nmin=i
        endif
      end do
!xocl end spread do max(zmax,nmax),min(zmin,nmin)
```

この場合には，各プロセスに割り当てられた do ループの繰り返し範囲の処理が終了した時点でグローバル関数 max の第１引数（ここでは変数 zmax）の値がプロセス間で比較され，その最大値を与えたプロセス上のすべての引数（ここでは変数 zmax 及び nmax）の値が他のプロセス上の対応する引数に複写されます。min に関しても同様です。

つぎの例はグローバル関数 sum を使って重複ローカル配列を転送するプログラムです。

```
!xocl processor p(2)
!xocl index partition ind=(p,index=1:10,part=band)
      dimension a(10,10)
              ⋮
      do j=1,10
        do i=1,10
          a(i,j)=0.0
        end do
      end do
!xocl spread do /ind
      do j=1,10
        do i=1,10
```

```
        a(i,j)=i*j
      end do
    end do
!xocl end spread do sum(a)
```

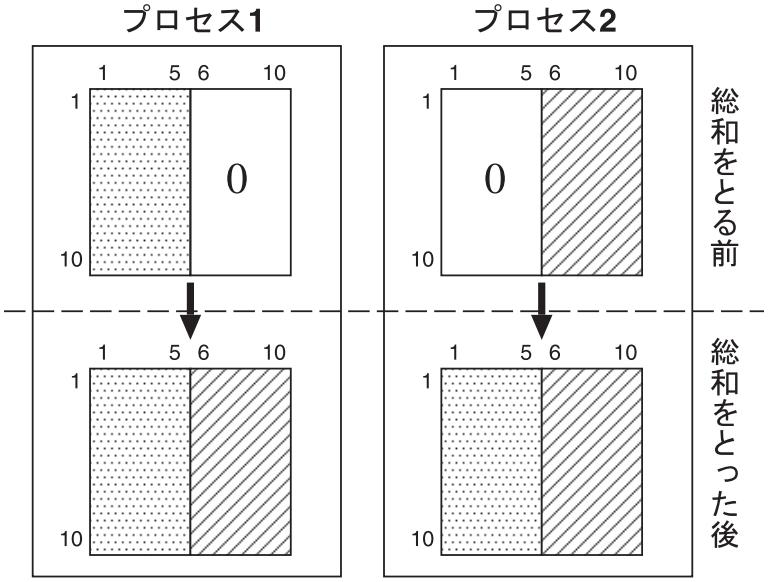


図 17　グローバル関数 sum による重複ローカル配列の転送

冗長実行で重複ローカル配列 a を 0 に初期化した後，spread do 文を使って a(1:10,1:5) の部分をプロセス 1 で，a(1:10,6:10) の部分をプロセス 2 でそれぞれ定義します（図 17 の総和をとる前の状態）。つぎに sum の引数に配列 a を指定して，a の要素ごとにプロセス間で総和をとります。結果として，配列 a の全要素は各プロセス上で同じ値を持つことになります（図 17 の総和をとった後の状態）。

### 10　拡張 spread do 構文

拡張 spread do 構文を使うと do ループの繰り返しの 1 回ずつと任意のブロックをリージョンとして並列処理することができます。使用例を図 18 に示します。これはすでにでてきた並列処理プログラム例 2 のサブルーチン bound を，拡張 spread do 構文を使って記述したものです。以下で，図 18 について説明します。

```
01:       subroutine bound(xmax,ymax,alpha,t)
02:       parameter (npe=5,imax= 100,jmax= 100)
03: !xocl processor mproc(npe)
04: !xocl subprocessor sproc=mproc(1:npe)
05: !xocl index partition ind1=(sproc,index=0:jmax, &
06: !xocl         part=band,overlap=(1,1))
07:       dimension u(0:imax,0:jmax),ug(0:imax,0:jmax)
08: !xocl local u(:,/ind1)
09: !xocl global ug
10:       equivalence (u,ug)
11:       common /data/ ug
12: !xocl index partition ind2=(sproc,index=0:jmax,part=band)
               ⋮
13: !xocl spread do /ind2
14: !xocl index 0
15:       do i=0,imax
16:         u(i,0)=…
17:       end do
18: !xocl index
19:       do j=1,jmax-1
20:         u(0,j)=…
21:         u(imax,j)=…
22:       end do
23: !xocl index jmax
24:       do i=0,imax
25:         u(i,jmax)=…
26:       end do
27: !xocl end spread do
28:       end
```

図 18　拡張 spread do 構文の例（サブルーチン bound）

1 ～ 12 行目　この部分は図 10 のサブルーチン diffus の宣言部分と同じです。

13 行目　拡張 spread do 構文の始まり。

14 ～ 17 行目　この部分は do 変数 j の値が 0 のときの処理を行います。このリージョンは 1 番目のプロセスが処理します。

18 ～ 22 行目　この部分は do 変数 j の値に対応して割り当てられたそれぞれのプロセスが並列処理します。

23 ～ 26 行目　この部分は do 変数 j の値が jmax のときの処理を 5 番目（最後）のプロセスが行います。

## 11　ブロードキャスト

あるプロセス上の重複ローカル変数の値を他のすべてのプロセスに転送するには broadcast 文を使います。例えば，3番目のプロセス上で重複ローカル配列 a の各要素の値が定義されていて，この値を他のプロセス上の配列 a に転送する場合[41]には以下のようにします。

```
!xocl processor p(5)
      dimension a(100)
              ⋮
!xocl spread region /p(3)
      do i=1,100
        a(i)=...
      end do
!xocl end spread region
!xocl broadcast(a)(idvproc().eq.3)
```

サービス関数 idvproc は無引数の関数で，関数値として idvproc を呼び出したプロセスのプロセッサ識別番号を返します[42]。

## 12　排他制御

複数のリージョンで同じグローバル変数の値を更新する場合などには排他制御が必要になりますが，これには lockon 文を使います。つぎの例では2つのリージョンでグローバル変数 k の値をそれぞれ更新しています。

```
!xocl processor p(2)
!xocl global k
      k=0
!xocl parallel region
!xocl spread region /p(1)
!xocl   lockon 1
          k=k+1
!xocl   end lockon
!xocl region /p(2)
```

41　3番目のプロセス上の配列 a の値を他のプロセス上の配列 a にコピーします。

42　プロセッサ識別番号は1からプログラム開始時に割り当てられたプロセス数までのいずれかの値を取ります。親プロセスのプロセッサ識別番号は1です。

```
!xocl   lockon 1
          k=k+1
!xocl   end lockon
!xocl end spread region
!xocl end parallel region
      write(6,*)k
```

同じ排他制御番号[43]を持つ lockon 文と end lockon 文の間にある部分の実行はリージョン間で排他制御されるので最後の write 文で出力される k の値は（正しく）２になります。このプログラムから lockon 文と end lockon 文を取り去ると write 文で出力される値は不定になります。lockon 文はリージョン間の実行順序の制御はできません。

## 13　入出力

XPFortran の入出力文で扱うファイルには共有ファイルと専有ファイルの２種類があります。共有ファイルはパラレルリージョン外でオープンされたファイルで，リージョンの内外を問わずプログラムのどこからでもアクセスできます。専有ファイルはパラレルリージョンの中でオープンされたファイルで，オープンしたリージョンの中だけ[44]で入出力が可能です。ただし，ファイルのオープンは open 文，read 文，write 文，backspace 文，endfile 文，print 文のいずれかが最初に実行されたときに行なわれます。標準入力ファイル（装置参照番号５），標準出力ファイル（装置参照番号６），標準エラー出力ファイルは共有ファイルです。

共有ファイルを異なるリージョンからアクセスする場合，入出力文の実行順序は不定になるので注意してください。

```
!xocl processor p(2)
      open(10,file='a.data')
              ⋮
!xocl spread region /p(1)
      write(10,*)i,j
!xocl region /p(2)
      write(10,*)x,y
!xocl end spread region
```

43　排他制御番号は 1 から 100 までの整数です。この例では 1 です。

44　ただし，そのリージョンの中でさらに子リージョンが生成される場合には子リージョンの部分は除きます。

この例ではファイル a.data を共有ファイルとして装置参照番号 10 でオープンし，このファイルにプロセス 1 上では変数 i 及び j の値を，プロセス 2 上では変数 x 及び y の値をそれぞれ出力しています。入出力の結果は入出力文単位で保証されますからこの 2 組の数値が 1 つのレコードに混在して出力されることはありませんが，2 つの write 文のどちらが先に実行されるかは不定です。

異なるリージョンにある入出力文の実行順序を制御したい場合にはサービスサブルーチン postevt 及び waitevt を使ってつぎのようにします。

```
!xocl processor p(2)
      open(10,file='a.data')
              ⋮
!xocl spread region /p(1)
      write(10,*)i,j
      call postevt(1)
!xocl region /p(2)
      call waitevt(1)
      write(10,*)x,y
!xocl end spread region
```

postevt と waitevt は互いに同じ事象識別番号[45]を引数にします。postevt が呼び出されると事象識別番号に対応する事象が発生したことが他のプロセスに通知されます。一方，waitevt を呼び出したリージョンに属するすべてのプロセス（この場合はプロセス 2）では事象識別番号に対応する事象が発生したという通知を受けとるまで待ち合わせします。したがって，この例ではプロセス 1 上の write 文が先に実行されます。postevt と waitevt の挿入位置に注意してください。

冗長実行では通常はすべてのプロセス上で同一の処理が行なわれますが，入出力文に関しては親プロセスのみが入出力処理を行ない，必要ならばプロセス間で通信を行なってその入出力処理の結果が子プロセス上でも同一になるようにします。例えば，冗長実行時に read 文を実行すると親プロセスが入力処理を行ない，入力した値は子プロセスへ転送されてすべてのプロセス上で同じ値になります。また，プロセスによって異なる値を持つ重複ローカル変数を出力する場合でも，親プロセス上の変数の値のみが出力されるので注意してください。

入出力文の入出力並びに関する制約として，1 つの入力文の入力並びの中にローカル変数とグローバル変数を混在させることはできません。また，入力並び及び出力並びのいずれにも分割ローカル配列の配列名を指定することはできません[46]。

45　事象識別番号は 1 から 100 までの整数です。この例では 1 です。

46　分割ローカル配列の配列要素名を指定することはできます。

## 14　プログラムミスの事例

これまでセンターの利用者の間で実際に起った XPFortran のプログラムミスの事例を以下で紹介します。

### 14. 1　事例 1

つぎのような差分式の計算の並列化を考えます。

```
dimension a(30),b(30)
      ⋮
do i=2,29
  b(i)=a(i-1)+a(i)+a(i+1)
end do
```

配列を均等分割して 3 台のプロセスを使って計算することにすると，図 19 のようなプログラムになります。配列 a を袖付き分割ローカル配列，配列 b を袖なしの分割ローカル配列としてそれぞれ宣言し，a はグローバル配列 ag と equivalence 文で結合します。overlapfix 文を使って配列 a の袖部分を転送した後，spread do 文を使って差分計算を分割して行ないます。

```
!xocl processor p(3)
!xocl index partition ind1=(p,index=1:30,part=band,overlap=(1,1))
!xocl index partition ind2=ind1(overlap=(0,0))
      dimension a(30),ag(30),b(30)
!xocl local a(/ind1),b(/ind2)
!xocl global ag
      equivalence (a,ag)
                ⋮
!xocl overlapfix(a) (id)
!xocl movewait (id)
!xocl spread do /ind1
      do i=2,29
        b(i)=a(i-1)+a(i)+a(i+1)
      end do
!xocl end spread do
```

図 19　差分計算を誤って並列化したプログラム

ところが，このプログラムを走らせてみると翻訳は正常終了するのですが，実行中に異常終了したり，実行は正常終了しても正しい計算結果が得られなかったりします。実は，このプログ

ラムには spread do 文の分割指定に誤りがあります。spread do 文の分割指定に ind1（袖付き）を指定すると，do ループは 1 番目のプロセスでは do 変数 i の値が 2 ～ 11 の範囲を，2 番目のプロセスでは 10 ～ 21 の範囲を，3 番目のプロセスでは 20 ～ 29 の範囲をそれぞれ計算しますから，配列 a，b 共に宣言した範囲以外の部分をアクセスすることになります。spread do 文には袖なしの分割指定 ind2 を指定してください。

### 14. 2　事例 2

下のような分割された配列中の特定の要素にアクセスするプログラムを考えます。

```
      parameter(npe=16,imax=35)
!xocl processor p(npe)
!xocl index partition ind=(p,index=1:imax,part=band)
      dimension a(imax)
!xocl local a(/ind)
            ⋮
!xocl spread region /p(1)
        a(1)=...
!xocl region /p(npe)
        a(imax)=...
!xocl end spread region
```

プロセスを 16 台使用して寸法 35 の 1 次元ローカル配列 a を均等分割し，spread region 文を使って 1 番目のプロセス上で a(1) に，16 番目のプロセス上で a(35) にそれぞれ値を代入しています。このように配列の端の部分だけにアクセスすることは境界条件を設定する場合などに必要となります。このプログラムも一見誤りはないようにみえるのですが，実際に走らせてみると正しい結果は得られません。理由は a(35) に正しい値が代入されないためです。

配列 a の寸法 35 をプロセス数 16 で割ると商が 2，余りが 3 となります。割り切れない時の均等分割では，商に 1 を足した要素数を順にプロセスに割り当てていって，最後に残りの要素数をそのつぎのプロセスに割り当てます。この場合には，1 番目のプロセスに a(1) ～ a(3)，2 番目のプロセスに a(4) ～ a(6)，...，11 番目のプロセスに a(31) ～ a(33)，12 番目のプロセスに a(34) 及び a(35)，と割り当てられます。したがって，13 番目から 16 番目までのプロセスには配列は割り当てられず，実質上「空回り」するだけになります。分割する配列の次元の寸法とプロセス数との関係によっては必ずしも最後のプロセスに配列の最後の要素が割り当てられるわけではない，というわけです。同様な現象は上のプログラムで npe の値を 13，14，15 としたときにも起こります。npe の値が 12 以下ならば正しく計算されますから，npe の値によってうまくいったりいかなかったりしてわけがわからなくなります。

一般的には使用するプロセス数に比べて分割する配列の次元の寸法は十分大きい場合が多いので，このようなことが頻繁に起こるとは思われませんが，図 20 に示したように，spread region 文の代わりに spread do 文と if 文を使えばプロセス数や配列の寸法を意識せずに記述できます。

```
      parameter (npe=16,imax=35)
!xocl processor p(npe)
!xocl index partition ind=(p,index=1:imax,part=band)
      dimension a(imax)
!xocl local a(/ind)
              ⋮
!xocl spread do /ind
      do i=1,imax
        if(i.eq.1) then
          a(1)=…
        elseif(i.eq.imax) then
          a(imax)=…
        endif
      end do
!xocl end spread do
```

図 20　分割された配列中の特定の要素にアクセスするプログラム

### 14. 3　事例 3

下のように冗長実行中にグローバル変数にアクセスするプログラムを考えます。

```
!xocl processor p(3)
      dimension nn(3)
!xocl global nn
!xocl parallel region
      nn(idvproc())=idvproc()
!xocl end parallel region
      write(6,*)(nn(i),i=1,3)
```

プロセスを 3 台使用し，寸法 3 の 1 次元グローバル配列 nn を用意します。パラレルリージョンに入ったところ，すなわち，冗長実行においてサービス関数 idvproc（呼び出したプロセスのプロセッサ識別番号を関数値として返します）を使って配列 nn に値を代入し，その値を出力します。関数値にしたがって，1 番目のプロセスでは nn(1) に 1 を，2 番目のプロセスでは

nn(2) に２を，３番目のプロセスでは nn(3) に３をそれぞれ代入するように思われますが，残念ながら write 文の出力は，

```
 1  0  0
```

となって期待どおりには動きません。このプログラムの代入文を if 文を使ってつぎのように，

```
if(idvproc().eq.1)then
  nn(1)=1
elseif(idvproc().eq.2)then
  nn(2)=2
else
  nn(3)=3
endif
```

と書き換えてもやはりうまくいきません。これは，冗長実行では通常はすべてのプロセス上で同一の処理が行なわれますが，グローバル変数へのアクセスに関しては動作が異なるからです。すなわち，冗長実行ではグローバル変数の値を定義する（代入文の左辺にある）場合には親プロセスのみがその処理を行ないます[47]。よって，各プロセスでグローバル変数に値を代入する場合には spread do 文または spread region 文を使う必要があります。例えば，spread do 文を使ってプログラムを修正すると以下のようになります。

```
!xocl processor p(3)
      dimension nn(3)
!xocl global nn
!xocl parallel region
!xocl spread do /(p)
      do i=1,3
        nn(idvproc())=idvproc()
      end do
!xocl end spread do
!xocl end parallel region
      write(6,*)(nn(i),i=1,3)
```

47　冗長実行でグローバル変数を引用する（代入文の右辺にある）場合には，すべてのプロセスがグローバル変数を引用したのと同じ結果になります。これは冗長実行中の入出力文の処理とよく似ています。

### 14. 4　事例４

下のような spread do 文を使って分割ローカル配列に代入するプログラムを考えます。

```
!xocl processor p(4)
!xocl index partition ind=(p,index=1:100,part=band)
      dimension a(10000,100),b(10000,100)
!xocl local a(:,/ind),b(:,/ind)
            ⋮
      do i=1,10000
!xocl spread do /ind
      do j=1,100
        b(i,j)=a(i,j)
      end do
!xocl end spread do
      end do
```

このプログラムには spread do 文の外側に do 変数 i に関するループがありますから spread do 文を 10000 回実行することになります。すでに説明したように spread do 文と end spread do 文の位置でバリア同期をとりますから，合計で 20000 回のバリア同期が発生します。この例のようにバリア同期が多発するプログラムを実行するとそのプログラムの経過時間や CPU 時間が非常に長くなる場合があります。実行そのものは正常に終了しますからプログラムミスとはいえないかもしれませんが，あまりタチの良くないプログラムといえます。

このプログラムの場合には spread do 文と end spread do 文の位置でバリア同期をとる必要はありませんから，

```
      do i=1,10000
!xocl spread nobarrier do /ind
      do j=1,100
        b(i,j)=a(i,j)
      end do
!xocl end spread nobarrier do
      end do
```

のように spread do 文と end spread do 文に nobarrier 指定をすると経過時間や CPU 時間が長くなる現象を回避できます。もちろん，このプログラムの場合には

```
!xocl spread do /ind
      do j=1,100
        do i=1,10000
          b(i,j)=a(i,j)
        end do
      end do
!xocl end spread do
```

のようにしてdo変数iに関するループを内側に入れてしまえばバリア同期が多発することはありません。

Ⅲ．その他

1. commonid文，barrier文，pass by local文

この3つのXPFortran構文については本稿では使い方を紹介しませんでした。以下に簡単な説明を加えておきますが，具体的な使い方に関しては文献［1］をご覧ください。

- commonid文

  overlapfix文，move wait文などに指定されるプロセッサ間データ転送識別名は，通常1つのプログラム単位の中だけで有効ですが，commonid文は2つのプログラム単位にまたがって1つのプロセッサ間データ転送識別名が有効であることを宣言します。

- barrier文

  barrier文が置かれた位置でバリア同期をとることを指定します。

- pass by local文

  pass by local文の直後に置かれたcall文中のサブルーチンの実引数としてあらわれるグローバル変数をローカル変数と同様の方法で引数渡しすることを指定します。

2. 組込み関数

組込み関数の実引数としてローカル変数，グローバル変数のいずれも用いることができます。ただし，実引数としてグローバル変数を用いる場合には，通常アクセスに時間がかかるので処理速度は遅くなります[48]。

3. サービスサブルーチン等

計時用サービスサブルーチン`clock`，`clockm`，`gettod`などは呼び出したプロセス上での値を返します。プロセスの問い合わせに関するサービス関数には`novproc`[49]，`idvproc`[50]など

48　resident指定やpass by local文の指定がある場合にはローカル変数と変わりません。

49　関数値としてプログラムの開始時に割り当てられたCPU数を返します。このプロセス数はprocessor文で宣言したプロセス数ではなく，ジョブの依頼時に割り当てたCPU数になります。計算機資源を有効に利用するため，ジョブの依頼時に割り当てたCPU数とprocessor文で宣言したプロセス数とが必ず一致するように指定してください。

50　関数値としてidvprocを呼び出したプロセスのプロセッサ識別番号を返します。プロセッサ識別番号は1からnovprocが返す値までのいずれかの値を取ります。親プロセスのプロセッサ識別番号は1です。

があります。

4. エラーメッセージ等

コンパイル時や実行時にシステムから出力されるエラーメッセージ，診断メッセージなどには以下のようなものがあります。なお，文献［1，2］はセンター図書室で貸し出しをおこなっています。また，センターの全国共同利用システムの利用者はセンターホームページ（URL は http://www2.itc.nagoya-u.ac.jp）からオンラインマニュアルをみることもできます。

- 並列化に関するコンパイルメッセージ

  メッセージ id が `joo` で始まるものです。メッセージの説明は文献［1］を御覧ください。
- その他のコンパイルメッセージ

  メッセージ id が `jwd` で始まるものです。メッセージの説明は文献［2］を御覧ください。
- 実行時のメッセージ

  メッセージ id が `jwe` で始まるものです。メッセージの説明は文献［2］を御覧ください。

5. 利用法等に関する文献

本稿では紹介しなかった HPC2500 の利用法等に関しては，以下の文献を参照してください。

- HPC2500 の利用法

  「スーパーコンピュータ及びアプリケーションサーバ利用の手引」をセンターホームページから参照してください。
- XPFortran プログラムのデバッグ支援・チューニング支援ツール

  文献［1，3］を御覧ください。文献［3］もセンター図書室で貸し出しをおこなっています。また，センターの全国共同利用システムの利用者はセンターホームページからオンラインマニュアルをみることもできます。

6. 役に立つコンパイルオプション

- 分割ローカル配列に関して，定義された領域以外をアクセスしていないかのチェックにはコンパイルオプション `-Wx` のサブオプション `-Cf` または `-Cr` を使います。これらのオプションをコンパイル時に指定することにより，実行時に領域外アクセスをチェックできます。`-Cf` と `-Cr` の違いは `-Cr` の方が荒いチェックとなるため実行時のオーバーヘッドが `-Cf` の場合に比べて削減されることです。いずれにしても実行に長時間を要するプログラムでは実行時間が短くなるようにしてから指定した方がよいでしょう。以下に使用例を示します。

```
hpc% xpfrt -Wx,-Cr -o sample sample.f
```

  分割ローカル配列に加えて重複ローカル配列のチェックをする場合にはサブオプションを `-Cfd` または `-Crd` とします。redident 指定したグローバル配列のチェックにはサブオプションを `-Cfg` または `-Crg` とします。
- プログラム中の parameter 文で定義された定数の値を，プログラムを直接書き換えることなく，コンパイルオプションによって変更することができます。使い方は以下のようです。

  `xpfrt -Wx,-P`***name=value*** ...

*name* には parameter 文中の定数名，*value* には変更したい値を指定します。例えば，プログラム中の parameter 文で使用プロセス数を `npe=2` と指定してあるとして，この値を 4 に変更して翻訳するには以下のようにします。

```
hpc% xpfrt -Wx,-Pnpe=4 -o sample sample.f
```

### 参考文献


［1］富士通（2002）XPFortran 使用手引書

［2］富士通（2004）Fortran 翻訳時メッセージ /Fortran 実行時メッセージ

［3］富士通（2004）プログラミング支援ツール使用手引書



（ながい　とおる：名古屋大学情報連携基盤センター大規模計算支援環境研究部門）