4　データ分散

4.1　分散の種類

テンプレートの分散種別には以下のものがあり，アプリケーションによって適切なものを選択します．ここでは，N をテンプレートのサイズ，P をノード数とし，

!$xmp nodes nd($P$)

!$xmp template tp($N$)

と宣言されているとします．

4.1.1　Block 分散

!$xmp distribute tp(block) onto nd

最もよく使われます．差分法の計算など，近傍の要素の参照が多い場合に適します．各ノードに割り当てられる block 幅は，

$$W = \lceil \frac{N}{P} \rceil \quad (1)$$

と計算され，左詰めで配置されます．

#pragma xmp nodes nd(3)

#pragma xmp template tp(0:21)

#pragma xmp distribute tp(block) onto p

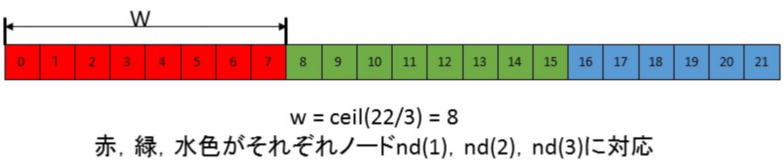

図 15: プログラム 13

4.1.2　Cyclic 分散

#pragma xmp distribute tp(cyclic) onto nd

計算負荷に偏りや不規則なばらつきがある場合に使われます．

#pragma xmp nodes nd(3)

#pragma xmp template tp(0:21)

#pragma xmp distribute tp(cyclic) onto p

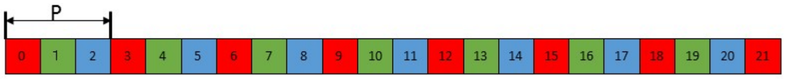

図 16: プログラム 14

4.1.3 Block-cyclic 分散

!$xmp distribute tp(cyclic( )) onto nd

block 分散と cyclic 分散の中間です．block 分散では負荷が不均等になるが，近傍要素の参照があるため cyclic では通信が増えてしまうような場合に使用します．ブロック幅 は利用者が指定します． のとき cyclic 分散と同じになり， のとき block 分散と同じになります．

#pragma xmp nodes nd(3)

#pragma xmp template tp(0:21)

#pragma xmp distribute tp(cyclic(3)) onto p

図 17: プログラム 15

4.1.4 不均等分散

!$xmp distribute tp(gblock( )) onto nd

はマッピング配列と呼ばれる大きさ のベクトルで， はノード nd( ) に割り当てる長さとなります．三角行列など，負荷の偏りが実行前に分かっているときに使用します．

#pragma xmp nodes nd(3)

#pragma xmp template tp(0:21)
int W[3]=6,11,5

#pragma xmp distribute tp(gblock(W)) onto p

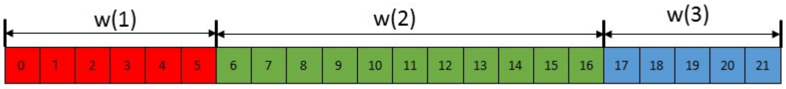

図 18: プログラム 16

4.2　データの整列と袖領域

4.2.1　データの整列

分散配列は，Align 指示文によってテンプレートへの整列を定義された配列変数です．分散したい配列の大きさに合わせてテンプレートの大きさが定義されることが多いですが，配列よりも大きいテンプレートを定義することもできます [5]．その場合，テンプレートと配列の位置関係は，インデックスのずれで表現されます．配列要素 b(1),b(2),... をテンプレート t(2),t(3),... にそれぞれ対応付けたいときには,

```
!$xmp align b(i) with t(i+1)
```

と表現します．テンプレートとのずれを表現することによって，変数どうし，および，変数とループの整列関係を宣言します．下の例は図 8 の例と似ていますが，変数 b の整列が 1 ずれたものになっています．そのため，同じ t(i) に整列する配列要素は a(i) と b(i-1) となります．

4.2.2　袖領域の宣言

差分法などを使うアプリケーションでは，配列要素 b(i) の計算のために，b(i-1) や b(i+1) の値を参照することがよくあります．この参照は，ノードの担当範囲の境界付近の配列要素では，隣接するノードとの通信になりますが，担当範囲を少しだけ拡張して隣接するノードのデータのコピーを保持できるようにすれば，通信回数を減らすことができます．この拡張された領域を袖またはシャドウと呼びます．袖の宣言には Shadow 指示文を使います．次の例で，b は左右に 1 つずつ，c は左に 2 つと右に 3 つの袖を持つことを指示しています．

[5] XMP では，逆にテンプレートより大きな配列が許される場合があります．袖付きの配列をテンプレートに整列させるとき，袖の大きさまでに限ってテンプレートの上下限をはみ出すことができます．

```
main(){
#pragma xmp nodes p(2)
#pragma xmp template t(0:12)
#pragma xmp distribute t(block) onto p
          double a[10],b[10];
#pragma xmp align a[i] with t(i)
#pragma xmp align b[i] with t(i+1)
#pragma xmp loop on t(i)
for (i=1;i<10;i++){
        a[i]=a[i]+1.0;
        b[i-1]=a[i];
      }
for(i=0;i<10;i++)
{
 printf("%lf¥t%lf¥n",a[i],b[i]);
}
return 0;}
```

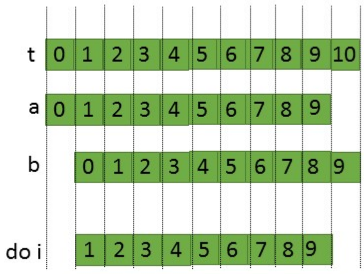

図 19: プログラム 17

```
main(){
#pragma xmp nodes p(3)
#pragma xmp template tp(0:21)
int W[3]={8,8,6};
#pragma xmp distribute tp(gblock(W)) onto p
     dimension a(0:21),b(0:21),c(0:21)
#pragma xmp align a[i] with tp(i)
#pragma xmp align b[i] with tp(i)
#pragma xmp shadow b(1)
#pragma xmp align c[i] with tp(i)
#pragma xmp shadow c(2:3)
return 0;}
```

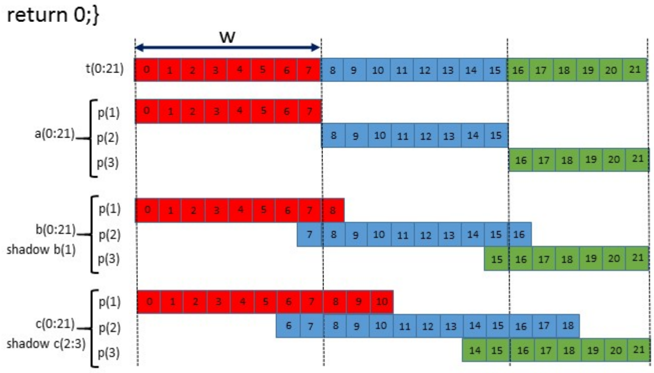

図 20: プログラム 18

袖領域は，他の部分と同じように，それを持つノードから自由にアクセスすることができます．袖に値を設定する方法については，5 章で紹介します．

4.3 多次元配列の分散

多次元配列の分散を使ったプログラムを見ていきます．次の例は，行列積を計算するサブルーチンです．[m,l] 行列 a，[l,n] 行列 b とサイズ l,m,n を引数として与え，結果を [m,n] 行列 c に格納しています．配列 a と c についての Align 指示文は，2 次元目で block 分散することを指示しています．配列 c については Align 指示がないので，重複配列となります．Loop 指示文も同じテンプレートに対して記述されています．

```
#include <stdio.h>
#include "xmp.h"

void sub(double a[2][2],double b[2][2], double c[2][2],int m,int l,int n)
{
#pragma xmp nodes p(2)
#pragma xmp template t(0:1)
#pragma xmp distribute t(block) onto p
#pragma xmp align c[*][i] with t(i)
#pragma xmp align b[*][i] with t(i)
int i,j,k;
#pragma xmp loop on t(j)
       for (j=0; j<n; j++)
          for(i=0; i<m; i++)
           for(k=0; k<l;k++)
             c[i][j] = c[i][j]+a[i][k]*b[k][j];
if(xmp_node_num()==1)
printf("%lf\n",c[0][0]);
if(xmp_node_num()==2)
printf("%lf\n",c[0][1]);
if(xmp_node_num()==1)
printf("%lf\n",c[1][0]);
if(xmp_node_num()==2)
printf("%lf\n",c[1][1]);
}


int main(){
#pragma xmp nodes p(2)
#pragma xmp template t(0:1)
#pragma xmp distribute t(block) onto p
double a[2][2],b[2][2],c[2][2];
#pragma xmp align c[*][i] with t(i)
#pragma xmp align b[*][i] with t(i)


a[0][0]=1.0;
a[0][1]=2.0;
a[1][0]=3.0;
a[1][1]=4.0;

if(xmp_node_num()==1)
b[0][0]=1.0;
if(xmp_node_num()==2)
b[0][1]=2.0;
if(xmp_node_num()==1)
b[1][0]=3.0;
if(xmp_node_num()==2)
b[1][1]=4.0;

if(xmp_node_num()==1)
c[0][0]=0.0;
if(xmp_node_num()==2)
c[0][1]=0.0;
if(xmp_node_num()==1)
c[1][0]=0.0;
if(xmp_node_num()==2)
c[1][1]=0.0;
sub(a,b,c,2,2,2);
return 0;
}
```

図 21: プログラム 19

この分散と配列アクセスのパターンを図 22 に示します．同図で配列を縦に切っている線は，4 並列の場合のデータ分散の様子を示し，あるプロセッサ（2 番目のプロセッサ）から見て読むだけの配列要素は薄い青で，読んで書く配列要素は赤で示しています．分散配列 b と c については，データ分散と

読み書きする配列要素がぴったり一致しているので，通信は生じません．重複配列 a については，どのノードも自分の計算のために全配列要素を参照します．

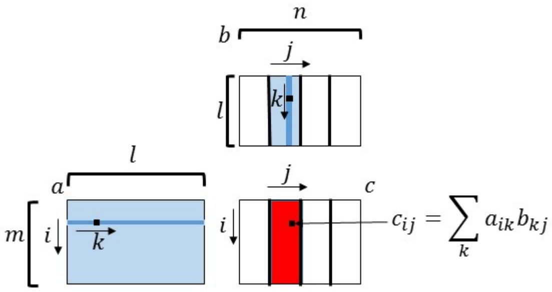

図 22:

4.4 重複配列の使い方

図 21 の配列 a のように，読むだけのデータであれば，重複変数とすることで通信をなくすことができます．分散の指示を書かなければ，重複変数になります．重複配列の特徴をまとめると，以下のように言えます．

- 読み出しは高速です．なぜなら通信が不要だからです．
- 書き込みは分散配列より時間がかかります．重複配列への書き込みは，冗長実行で書き込むか，あるノードで書き込んだ後で全ノードにブロードキャストをする必要があります．分散配列への書き込みのように並列化できません．逐次実行と同じ速さが上限となります．
- 分散配列よりメモリを消費します．ノード数が増えても，ノード当りのメモリ消費量が減っていきません．

つまり，読み出しが主のデータが適しています．一方で，並列効果が出ないことと，メモリの浪費には気をつける必要があります．

4.5 分散次元の選択

多次元配列は，分散する次元を選択できます．例えば 2 次元配列なら，下図に示すようにテンプレートに整列させる次元を選ぶことにより，分散次元を選びます．図 21 と同じ結果を行列積のプログラムですが，下に示す例は，書き込み先の c を重複配列としています．a と b は，それぞれ 2 次元目と 1

```
int main()
{
#pragma xmp nodes p(4)
    real a(100,200),b(100,200)
#pragma xmp template t1(100)
#pragma xmp distribute t1(block) onto p
#pragma xmp align b[i][*] with t1(i)
#pragma xmp template t2(200)
#pragma xmp distribute t2(block) onto p
#pragma xmp align a[*][j] with t2(j)
return 0;
}
```

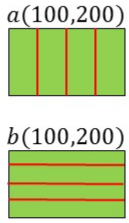

図 23: プログラム 20

次元目で分散しています．ループ交換により並列化される k のループを一番外側に移動し，通信の発生回数を削減しています．このプログラムの並列化では，以下の点が特徴的です．

- Loop 指示文の On 節は，代入文の左辺でなく右辺の分散に整合させています．アクセスパターンは図 25 に示すように，使用されるデータ（薄い水色部分）が各ノードの担当範囲となり，定義されるデータ（赤色の部分）が全体配列となります．

- 配列変数 c を集計変数として，配列の集計演算を行います．

同じ行列積のプログラムに対して，(1) 配列 a を重複配列とする方法（図 21）と，(2) 配列 c を重複配列とする方法（図 24）を見ました．比較すると，(1) の方が性能が良いと言えます．なぜなら，(2) では，重複配列 c の全配列要素について，ノード間で総和を取る集計演算を行うため，これが (1) と比べたオーバーヘッドになります．ただし，配列の分散はこのサブルーチンだけで決定することはできません．プログラムの別の部分にとっては，(1) のような分散方法より (2) のような分散方法の方がよいかもしれません．また，重複配列はメモリを多く必要としますので，行列 a,b,c の大きさも考慮に入れる必要があります．

```
#include <stdio.h>
#include "xmp.h"

void sub(double a[2][2],double b[2][2], double c[2][2],int m,int l,int n)
{
#pragma xmp nodes p(2)
#pragma xmp template t(0:1)
#pragma xmp distribute t(block) onto p
#pragma xmp align a[*][i] with t(i)
#pragma xmp align b[i][*] with t(i)
int i,j,k;
#pragma xmp loop on t(k) reduction(+:c)
       for (k=0; k<n; k++)
          for(j=0; j<m; j++)
           for(i=0; i<l; i++)
             c[i][j] = c[i][j]+a[i][k]*b[k][j];
if(xmp_node_num()==1){
printf("%lf\n",c[0][0]);
printf("%lf\n",c[0][1]);
printf("%lf\n",c[1][0]);
printf("%lf\n",c[1][1]);
}
}


int main(){
#pragma xmp nodes p(2)
#pragma xmp template t(0:1)
#pragma xmp distribute t(block) onto p
double a[2][2],b[2][2],c[2][2];
#pragma xmp align a[*][i] with t(i)
#pragma xmp align b[i][*] with t(i)


if(xmp_node_num()==1)
a[0][0]=1.0;
if(xmp_node_num()==2)
a[0][1]=2.0;
if(xmp_node_num()==1)
a[1][0]=3.0;
if(xmp_node_num()==2)
a[1][1]=4.0;

if(xmp_node_num()==1)
b[0][0]=1.0;
if(xmp_node_num()==1)
b[0][1]=2.0;
if(xmp_node_num()==2)
b[1][0]=3.0;
if(xmp_node_num()==2)
b[1][1]=4.0;

c[0][0]=0.0;
c[0][1]=0.0;
c[1][0]=0.0;
c[1][1]=0.0;
sub(a,b,c,2,2,2);
return 0;
}
```

図 24: プログラム 21

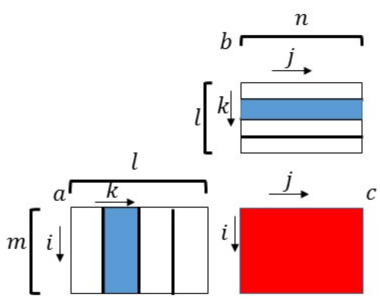

図 25: