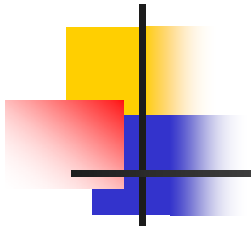


並列アルゴリズム

2005年後期　火曜　2限

青柳　睦

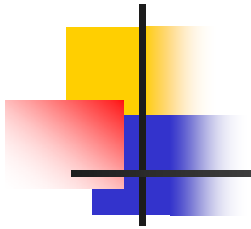
Aoyagi@cc.kyushu-u.ac.jp

http://server-500.cc.kyushu-u.ac.jp/

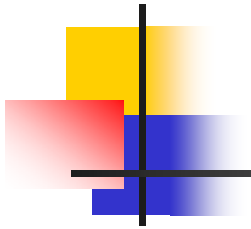
11月29（火）

7．集団通信（Collective Communication）

8．領域分割（Domain Decomposition）



もくじ



成績評価

出席点５割，レポート５割

aoyagi@cc.kyushu-u.ac.jpへメール

Subject: 並列アルゴリズム

学籍番号，氏名，専攻，

座席番号(A-1,A-2,・・C-3など)

7. 集団通信(Collective Communication)

前回までは，MPI_Send, MPI_Recv による1対1通信を紹介した。
ここでは，同じデータをコミュニケータ中のすべてのプロセスに送りたい場合に利用されるMPI_Bcast 関数を紹介する.

int MPI_Bcast(　/* 機能：バッファのデータをrootから全プロセスに送る． */

void*	message	/* 送(受)信バッファの開始アドレス(IN/OUT) */,
int	count	/* データの要素数(IN) */,
MPI_Datatype	datatype	/* データタイプ(IN) */,
int	root	/* 送信元(IN) */,
MPI_Comm	comm	/* コミュニケータ(IN) */)

MPI_Bcast 関数により、Rank=root(送信元)のデータが他の全プロセスに送信される。この際、「木構造通信」を使うなど、並列計算機のネットワーク結合網を考慮した集団通信手段が使われるため，一般にSend とRecv をp回繰り返す手法よりも，高速である。

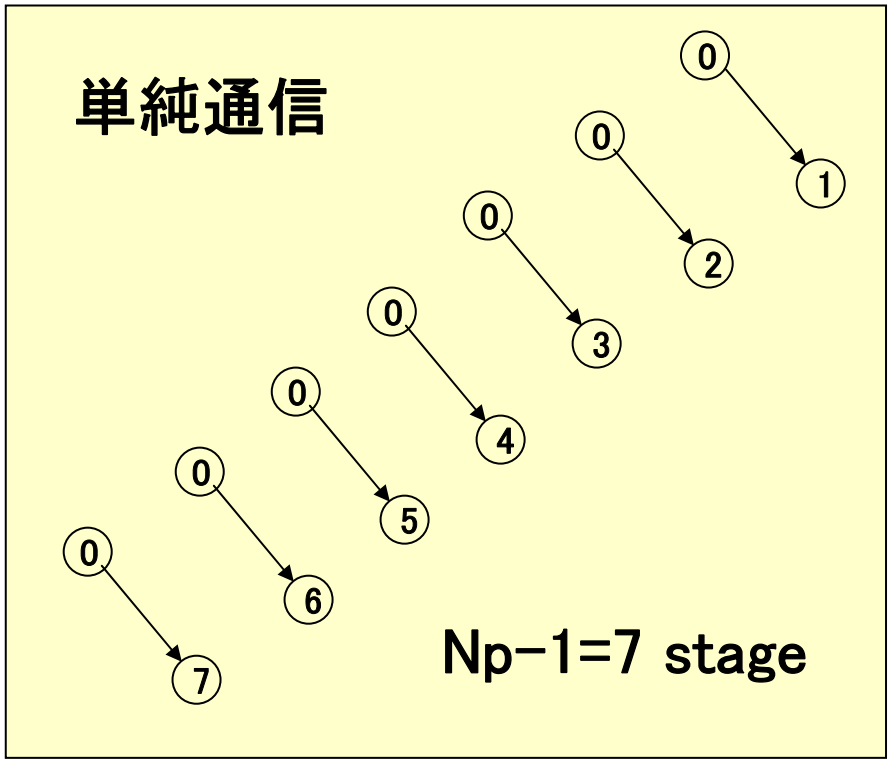
7.1 MPI_Bcast 関数の使用例

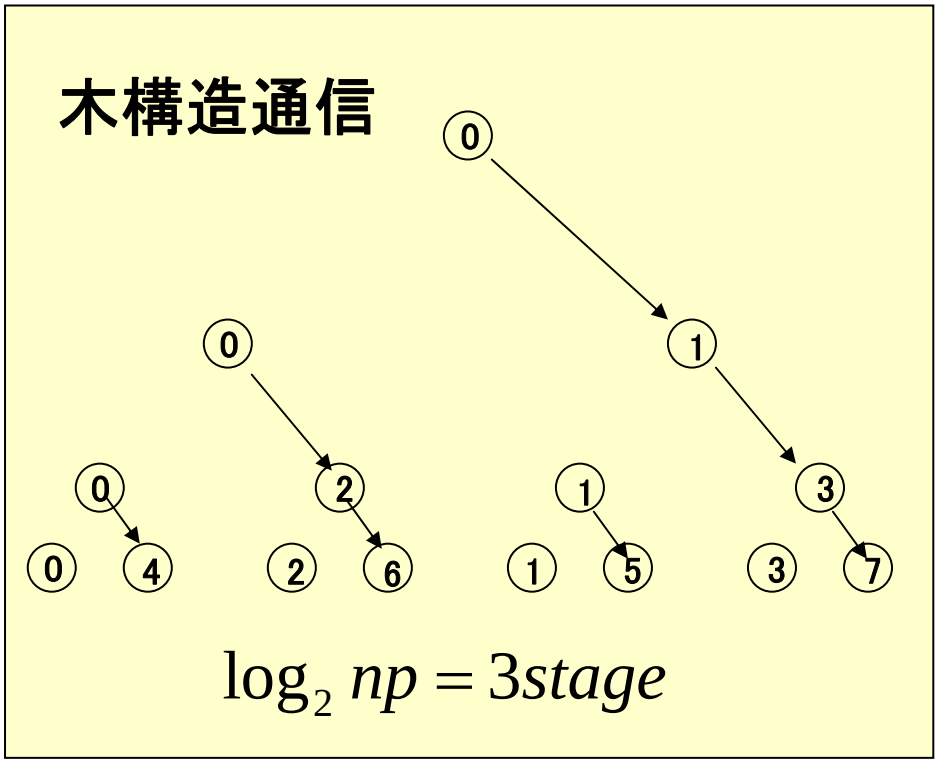
```
      :
      :

if (my_rank == 0) {
    loop=atoi(argv[1]);
    for (dest = 1; dest < numprocs; dest++) {
        MPI_Send(&loop, 1, MPI_INT, dest, tag,
            MPI_COMM_WORLD);
    }
} else {
        source=0;
        MPI_Recv(&loop, 1, MPI_INT, source, tag,
            MPI_COMM_WORLD, &status);
}

 width = 1.0 / loop;
 local_loop = loop / numprocs;

    :
    :
```

```
      :
      :

 if (my_rank == 0) {
     loop=atoi(argv[1]); }

MPI_Bcast(&loop, 1, MPI_INT, 0,
          MPI_COMM_WORLD);

  width = 1.0 / loop;
  local_loop = loop / numprocs;

     :
     :
```

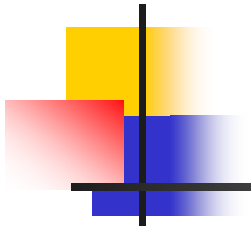
木構造通信

（例）np=8 の単純通信と木構造通信





一般に npの集団通信に，単純通信ではnp−1
木構造通信では $\log_2 np$ 回の通信が必要



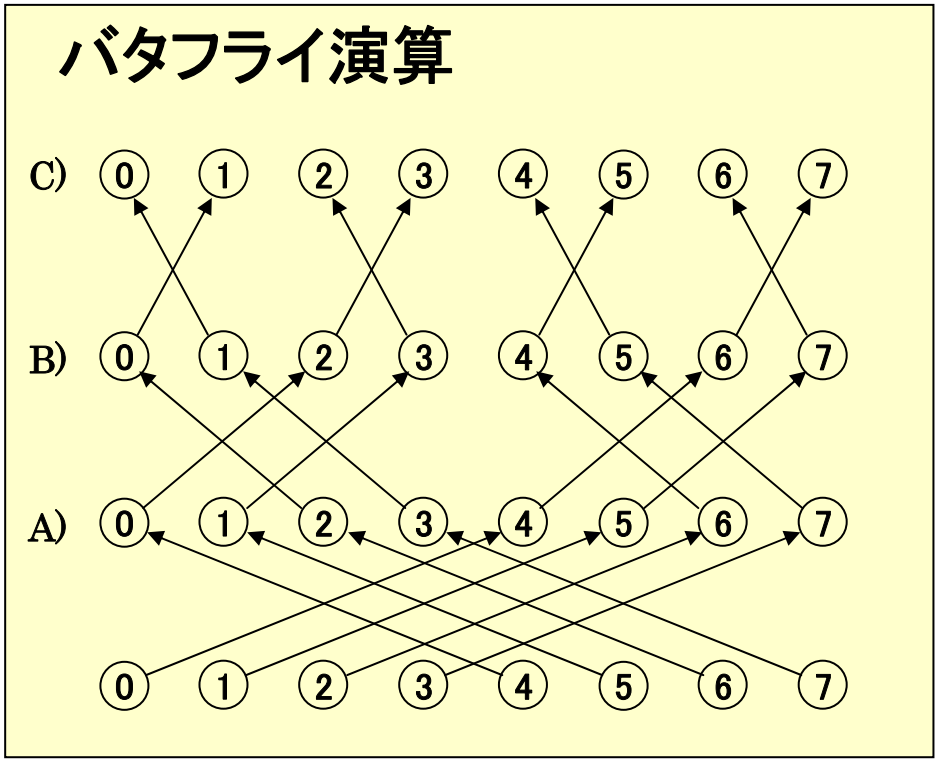
Reduction演算

同じデータをコミュニケータ中のすべてのプロセスで演算操作（例えば加算）し、その結果をすべてのプロセスに送りたい場合に利用されるMPI_Allreduce 関数を紹介する.

int MPI_Allreduce(　/* 機能：*/

void*	operand	/* Operand（演算される側）の開始アドレス(IN) */,
void*	result	/*演算結果の格納開始アドレス(OUT) */,
int	count	/* データの要素数(IN) */,
MPI_Datatype	datatype	/*データタイプ(IN) */,
MPI_Op	operator	/*演算操作タイプ(IN) */,
MPI_Comm	comm	/*コミュニケータ(IN) */,

オールレデュースの通信構造



プロセスiの最初のsumの値をS_iと書くと左のデータ交換・加算が行われた場合の各プロセスの sum の値の変化は

プロセス		A)	B)	C)
0	S_0	$\sum_{i=0,4} S_i$	$\sum_{i=0,2,4,6} S_i$	$\sum_{i=0}^{7} S_i$
1	S_1	$\sum_{i=1,5} S_i$	$\sum_{i=1,3,5,7} S_i$	$\sum_{i=0}^{7} S_i$
2	S_2	$\sum_{i=2,6} S_i$	$\sum_{i=0,2,4,6} S_i$	$\sum_{i=0}^{7} S_i$
3	S_3	$\rightarrow \sum_{i=3,7} S_i$	$\rightarrow \sum_{i=1,3,5,7} S_i$	$\rightarrow \sum_{i=0}^{7} S_i$
4	S_4	$\sum_{i=0,4} S_i$	$\sum_{i=0,2,4,6} S_i$	$\sum_{i=0}^{7} S_i$
5	S_5	$\sum_{i=1,5} S_i$	$\sum_{i=1,3,5,7} S_i$	$\sum_{i=0}^{7} S_i$
6	S_6	$\sum_{i=2,6} S_i$	$\sum_{i=0,2,4,6} S_i$	$\sum_{i=0}^{7} S_i$
7	S_7	$\sum_{i=3,7} S_i$	$\sum_{i=1,3,5,7} S_i$	$\sum_{i=0}^{7} S_i$

MPI_Op（演算操作のタイプ）

OP	演算	可能なデータタイプ
MPI_SUM	合計	MPI_INTEGER, MPI_REAL, MPI_REAL8, MPI_COMPLEX
MPI_PROD	積	〃
MPI_MAX	最大	MPI_INTEGER, MPI_REAL, MPI_REAL8
MPI_MIN	最小	〃
MPI_MAXLOC	最大と位置	MPI_2INTEGER, MPI_2REAL, MPI_2DOUBLE_PRECISION
MPI_MINLOC	最小と位置	〃
MPI_LAND	論理AND	MPI_LOGICAL
MPI_LOR	論理OR	〃
MPI_LXOR	論理XOR	〃
MPI_BAND	ビットAND	MPI_INTEGER, MPI_BYTE
MPI_BOR	ビットOR	〃
MPI_BXOR	ビットXOR	〃

(*) MPI_OP_CREATE 関数により、ユーザ定義の演算を登録し利用することができる

ベクトルの内積（dot product）

二つのベクトル $\mathbf{x} = (x_0, x_1, \cdots;, x_{n-1})^T$ $\mathbf{y} = (y_0, y_1, \cdots, y_{n-1})^T$ の内積

$$\mathbf{x} \cdot \mathbf{y} = x_0 y_0 + x_1 y_1 + \cdots x_{n-1} y_{n-1}$$

```
float Serial_dot(
    float x[] /* 入力 */,
    float y[] /* 入力 */,
    int n     /* 入力 */) {

    int i;
    float sum = 0.0;

    for (i =  0; i < n; i++)
        sum = sum + x[i]*y[i];
    return sum;
} /* Serial_dot */
```

```
void Read_vector(
        char*  prompt  /* in  */,
        float  v[]     /* out */,
        int    n       /* in  */) {
    int i;

    printf("Enter %s¥n", prompt);
    for (i = 0; i < n; i++)
        scanf("%f", &v[i]);
}  /* Read_vector */
```

ベクトルの内積（逐次プログラム）

```
#include <stdio.h>

#define MAX_ORDER 100

main() {
    float  x[MAX_ORDER];
    float  y[MAX_ORDER];
    int    n;
    float  dot;

    void Read_vector(char* prompt, float v[], int n);
    float Serial_dot(float x[], float y[], int n);

    printf("Enter the order of the vectors¥n");
    scanf("%d", &n);
    Read_vector("the first vector", x, n);
    Read_vector("the second vector", y, n);
    dot = Serial_dot(x, y, n);
    printf("The dot product is %f¥n", dot);
}  /* main */
```

内積の並列プログラム例

```
float Parallel_dot(
   float local_x[]    /* 入力 */,
   float local_y[]    /* 入力 */,
   int n_bar /* 入力 */) {

   float local_dot;
   float dot = 0.0;
   float Serial_dot(float x[], float y[], int m);

   local_dot = Serial_dot(local_x, local_y, n_bar);
   MPI_Reduce(&local_dot,&dot,1,MPI_FLOAT,
      MPI_SUM,0,MPI_COMM_WORLD);
   return dot;
} /* Parallel_dot */
```

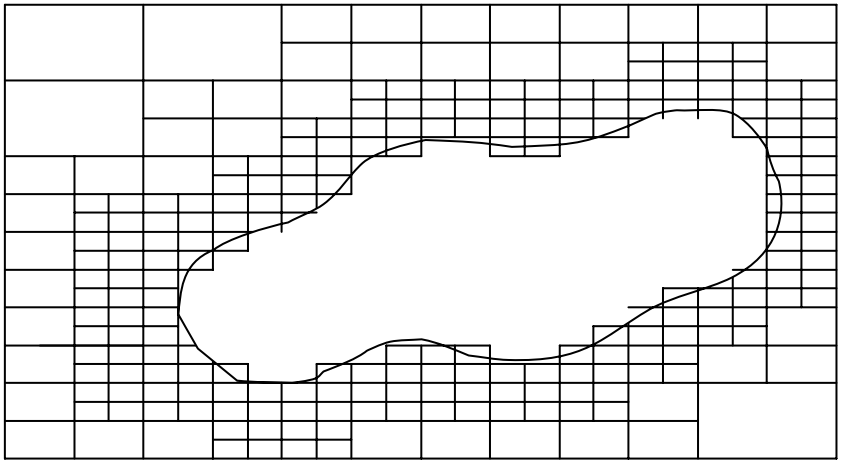
次回、並列のDOTプログラムを完成させてPPTに書く
Reduceの引数

内積の並列プログラム例

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

#define MAX_LOCAL_ORDER 100

main(int argc, char* argv[]) {
   float  local_x[MAX_LOCAL_ORDER];
   float  local_y[MAX_LOCAL_ORDER];
   int    n;
   int    n_bar;  /* = n/p */
   float  dot;
   int    p;
   int    my_rank;

   float Parallel_dot(float local_x[], float local_y[], int n_bar);

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   :
```

8. 領域分割（Domain Decomposition）

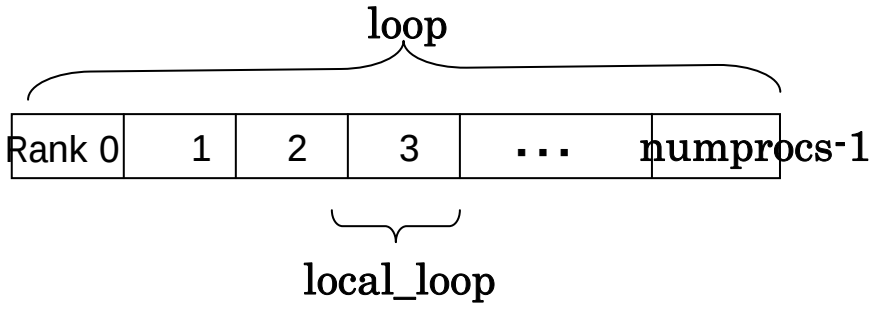
- Introduction（1次元ループ）
- ブロック分割
- サイクリック分割
- ブロック・サイクリック分割
- 数値計算例



領域分割のIntro（1次元ループを例に・・）

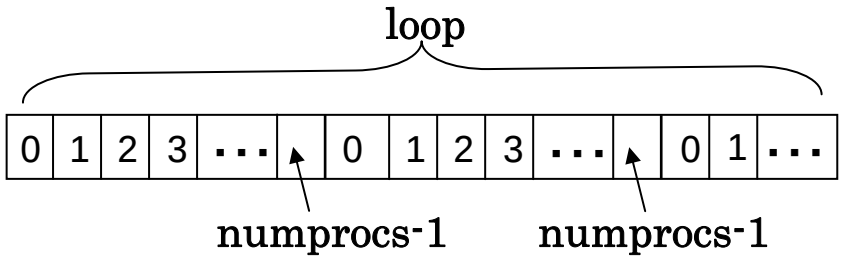
領域をプロセス(rank)に分割

```
 :
 width = 1.0 / loop;
 local_loop = loop / numprocs;

 sum = 0.0;
 for (i = my_rank*local_loop;
     i < (my_rank+1)*local_loop; i++) {
     x = (i + 0.5) * width;
     sum += 4.0 / (1.0 + x*x);
 }
:
```



要素演算をプロセス(rank)に再帰的に分割

```
 :
 width = 1.0 / loop;

 sum = 0.0;
 for (i = my_rank; i < loop;
         i+=numprocs ) {
     x = (i + 0.5) * width;
     sum += 4.0 / (1.0 + x*x);
 }
:
```



（注）領域には複数の要素演算 （例では，sum += 4.0 / (1.0 + x*x); ）を含むとする

領域分割のIntro(2)

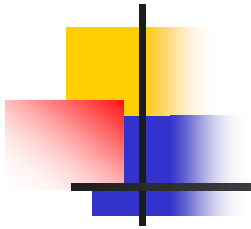
台形公式によるπの計算例では要素演算は，均一の計算負荷であるから，どちらの分割方法を用いてもrankごとのロードバランスに不均衡は生じないが・・

```
   :
sum = 0.0;
   for (i =  ・・・)
   {
       x = (i + 0.5) * width;
       sum += f(x);
   }
  :
/* 被積分関数を定義 */
double f( double x) {
  return  4.0 / ( 1.0 + x*x );
}
```

例えば，$f(x)=\dfrac{4}{1+x^2}$ の代わりに，

$$f(x)=\begin{cases}\dfrac{54}{5}x\cdots\left(0\le x\le 1/3\right)\\ \dfrac{4}{1+x^2}\cdots\left(1/3\prec x\le 2/3\right)\\ \dfrac{81}{13}x^2\cdots\left(2/3\prec x\le 1\right)\end{cases}$$

の様にxの範囲で演算内容が異なっている場合には演算負荷に不均衡が生じる.



並列処理の課題（2回目の講義ノート再掲）

- 並列化不可能な部分が有る → アムダール則
- 負荷のアンバランス → 領域分割を工夫
- 並列化によるオーバーヘッド
  - 通信のオーバーヘッド
  - 並列アルゴリズム

→
- プロセス間のロードバランスをなるべく均等にする
- プロセス間の通信コストをなるべく少なくする

（スカラ並列の場合，キャッシュを効果的に使う）

ブロック分割

プロセス数がN個の場合、各プロセスに1／Nの領域を割当てる方法を**ブロック分割**という.

1／N の領域を分割するとしてもその方法は一つだけではなく，数値計算の並列化によって発生する通信コストを最小にする様に“問題に適した”分割方法を用いる.

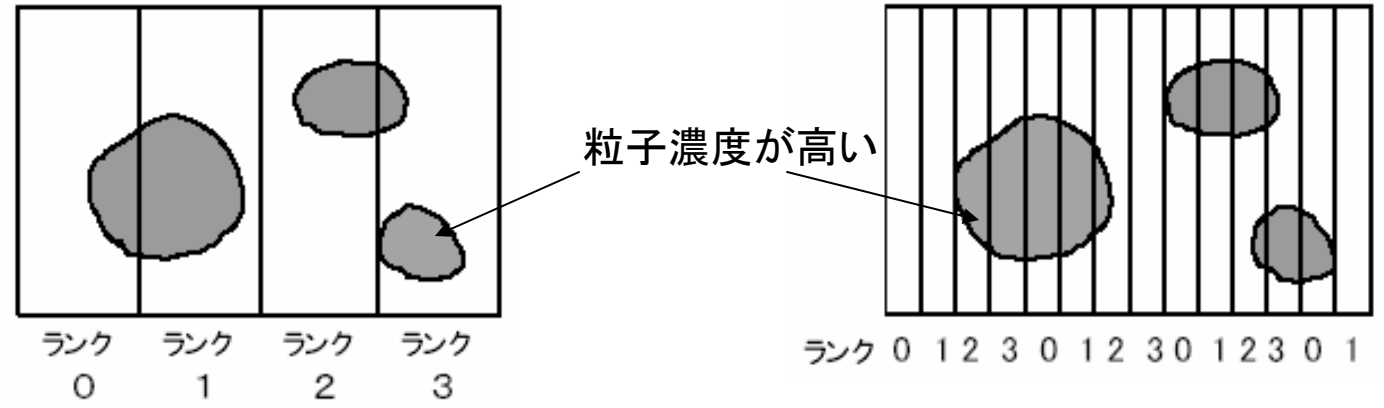
例えば2次元配列の場合、以下に示すように行で分割するか、列で分割するか、行と列の両方で分割するか等，行列演算の内容に依存する.

領域ごとの計算量が同じであるならばプロセスのロードバランスは均等になるが，キャッシュのヒット・ミスヒットを考慮する必要がある.

サイクリック分割

例として，２次元空間をある運動方程式に従い運動する複数粒子の運動を解く場合を考える．運動領域（平面）を均等にメッシュに分割しP(xi, yj)，それぞれの小領域をプロセスに割り当てるとする．

このとき，単純なブロック分割を行った場合には，粒子がある部分領域に集まってきた際、領域ごとの粒子数の差に応じ，ロードバランスが不均等になってしまう．

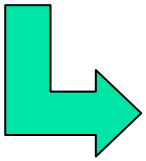


ブロック分割　　サイクリック分割

この様な場合には，領域または演算要素そのものを再帰的に分割する**サイクリック分割**が有効である．また、LU分解の並列化（次ページ参照）の様に、計算の進行とともに演算領域が変化する場合にもサイクリック分割が有効に働く．

LU分解の並列プログラム

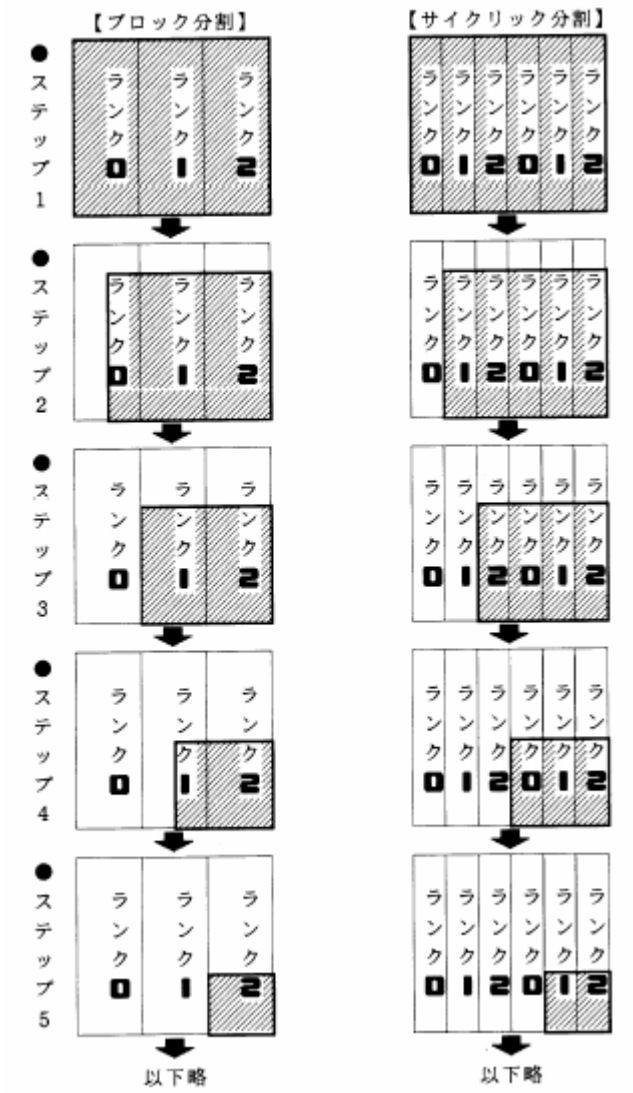
```
      :
      :

for (K = 1; K < N; K++) {

   for (I=K+1; I<N+1; I++) {
         A(I,K) = A(I,K) / A(K,K)
    }

    for (J=K+1; J<N+1; J++) {
       for ( I=K+1; I<N+1; I++)
       {   A(I,J) = A(I,J) – A(I,K) * A(K,J)  }
    }
}
    :
```



並列化

```
      :
/* Start up MPI */
   MPI_Init(&argc, &argv);

   /* Find out process rank  */
   MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

/* Find out number of processes */
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
      :
/* Block or Cyclic mapping */
 for (I=1; I<N+1; I++) {
    MAP(I)=mod(I-1, nprocs)
 }

/* Start LU decomposition */
  for (K = 1; K < N; K++) {
      if (MAP(K) == my_rank) then
            for (I=K+1; I<N+1; I++) {
              A(I,K) = A(I,K) / A(K,K)    }
      endif

(次ページへ続く)
```

LU分解の並列化例 （ブロック分割とサイクリック分割）

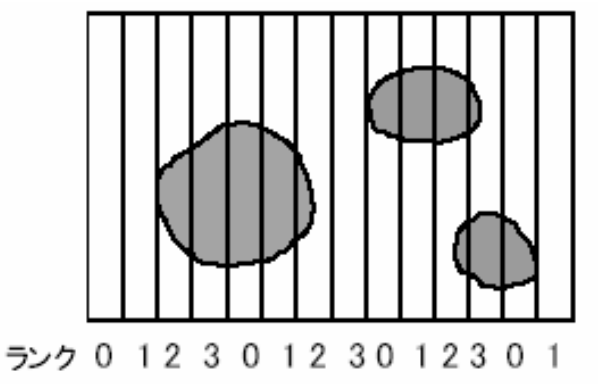


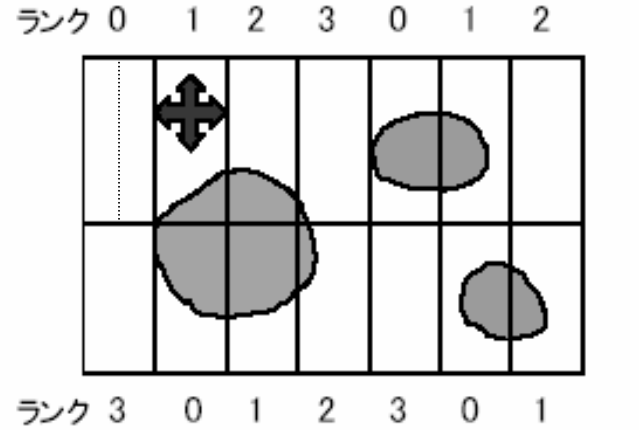

```
        :

MPI_Bcast(A(K+1,K),N-K, MPI_REAL, MAP(K),
    MPI_COMM_WORLD);

     for (J=K+1; J<N+1; J++) {
        if ( MAP(J) == my_rank ) then
            for ( I=K+1; I<N+1; I++)
              {   A(I,J) = A(I,J) – A(I,K) * A(K,J)  }
        endif
      } /* end of J-loop */

} /* end of K-loop */
        :
MPI_Finalize();
```

ブロック分割の場合、計算ステップが進むにつれて演算負荷の低いランク(Rank)が生じる.

ブロック・サイクリック分割

単純にサイクリック分割した場合に、領域間で数値計算のためのデータ参照が多く発生する時には、領域間のデータ通信の負荷が上昇し、逆にパフォーマンスの低下をまねく場合がある.　このような場合、ブロック分割とサイクリック分割を組合わせたブロック・サイクリック分割が有効な場合もある.



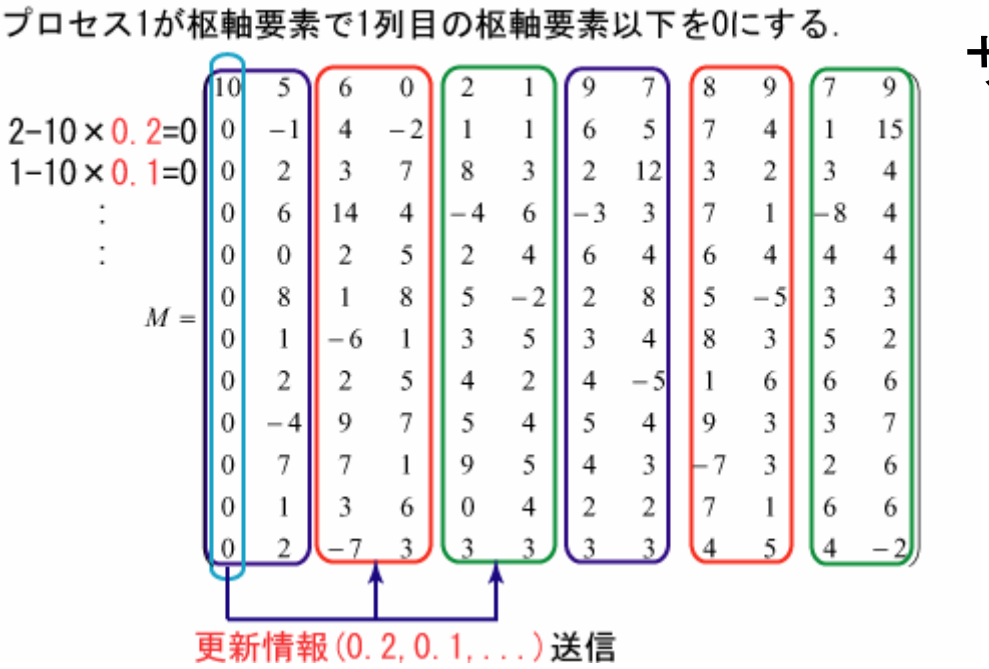


（単純な）サイクリック分割　　ブロック・サイクリック分割

（例）行列演算 （並列 LU分解）

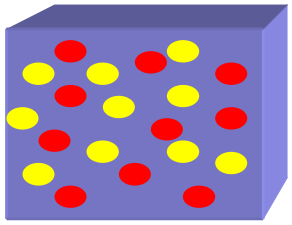
ブロック分割

サイクリック分割



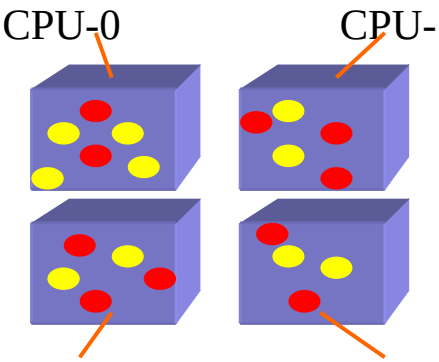
（例）粒子分割と領域分割　分子動力学シミュレーション

粒子分割法



赤　→　CPU-0
黄色　→　CPU-1

領域分割法



（「のりしろ」を活用）