MPI プログラミング入門

コンパックコンピュータ株式会社

ソリューション＆テクニカルサポート統括本部

MPI プログラミング入門　目次

１．並列プログラミングとは

なぜ並列プログラミングが必要か

近年、コンピューティングパワーはますます増大を続けています。しかしながら、そのパワーの増加以上にユーザの要求は大きくなっています。そして、単体でのコンピューティングパワーの限界を打ち破るために、並列プログラミングが考えられてきました。並列プログラミングの目的には、主に次の２つがあります。

・計算時間の短縮

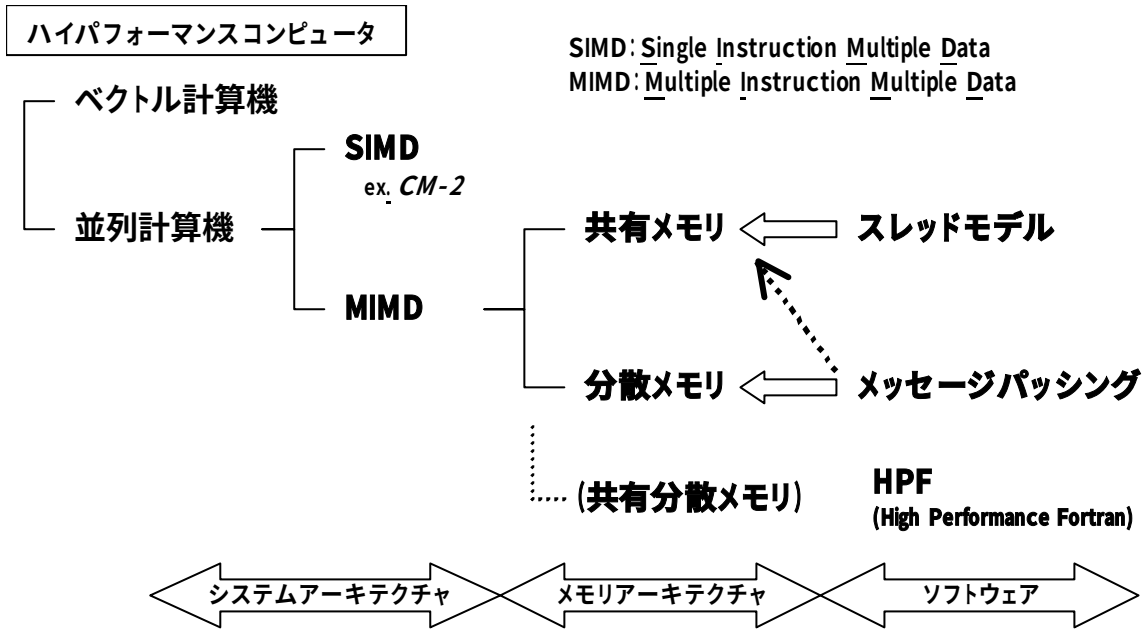
・大容量のメモリを必要とする計算をする

そして、並列化によって、

『N 並列にした場合、計算量を１／N に短縮する』

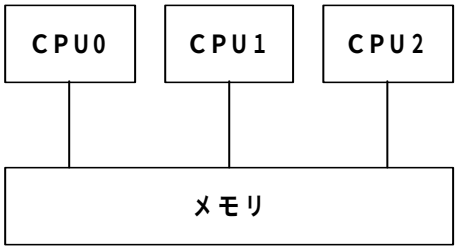
ことを理想とします。

並列計算機のアーキテクチャ



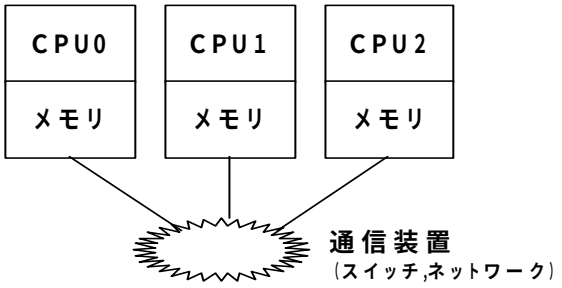
現在の並列計算機は、ほとんどが MIMD(Multiple Instruction Multiple Data)タイプであり、メモリへのアクセス方式によって、さらに共有メモリ型と分散メモリ型に分かれます。

共有メモリ・アーキテクチャ



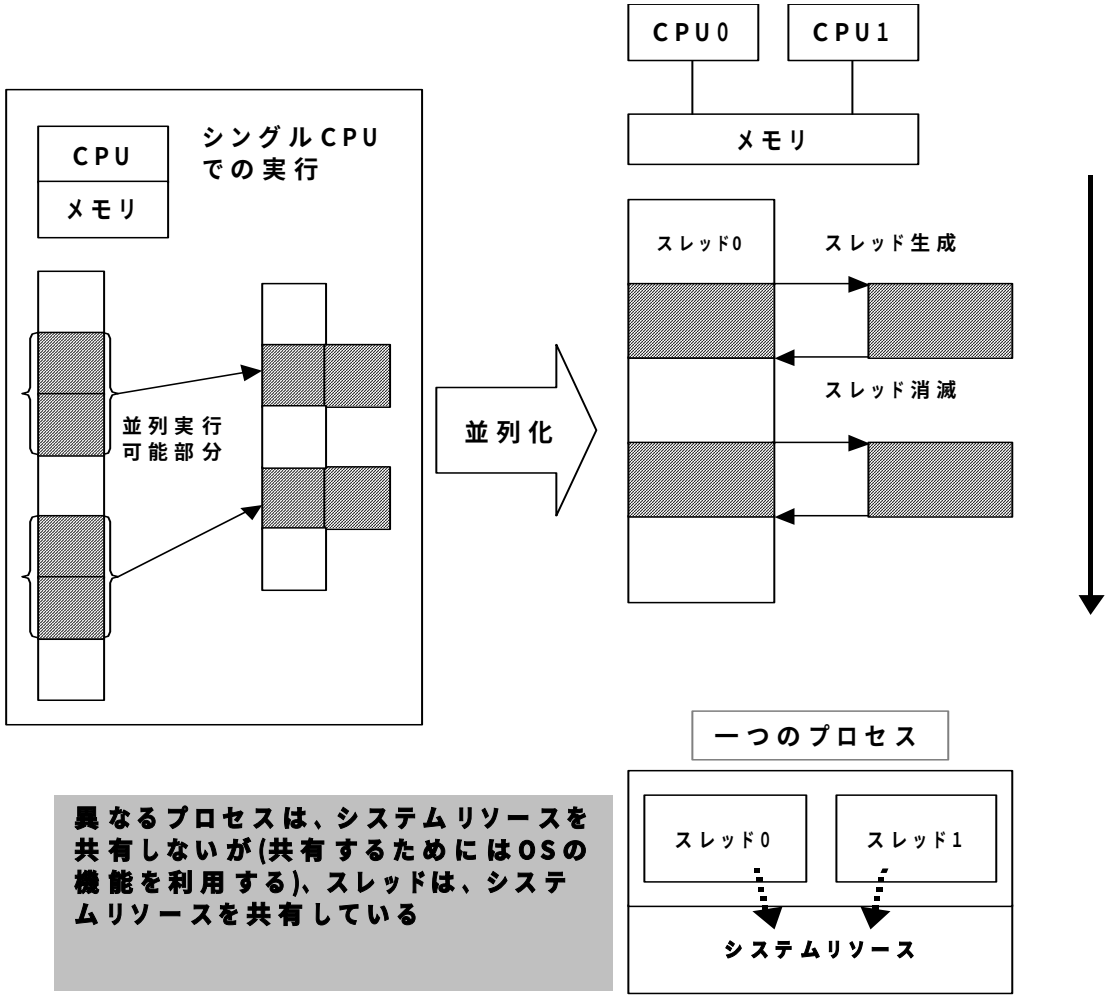
・複数のCPUがメモリを共有する
・異なるCPUが同じメモリ空間にアクセス可能であるため、排他制御を必要とする場合がある

分散メモリ・アーキテクチャ



・メモリは各CPUにローカルに接続されている
・他のCPUがもつ情報が必要な場合CPU間で明示的な情報交換を必要とする

◆ 共有メモリマシンでのプログラムの実行



異なるプロセスは、システムリソースを共有しないが(共有するためにはOSの機能を利用する)、スレッドは、システムリソースを共有している

プログラムの動き

１．マスタースレッド(スレッド０)が起動する

２．並列化された部分になると、スレッド０は『スレッド１』を生成する

３．スレッド０および生成されたスレッド１がそれぞれ処理を１／N ずつ行う

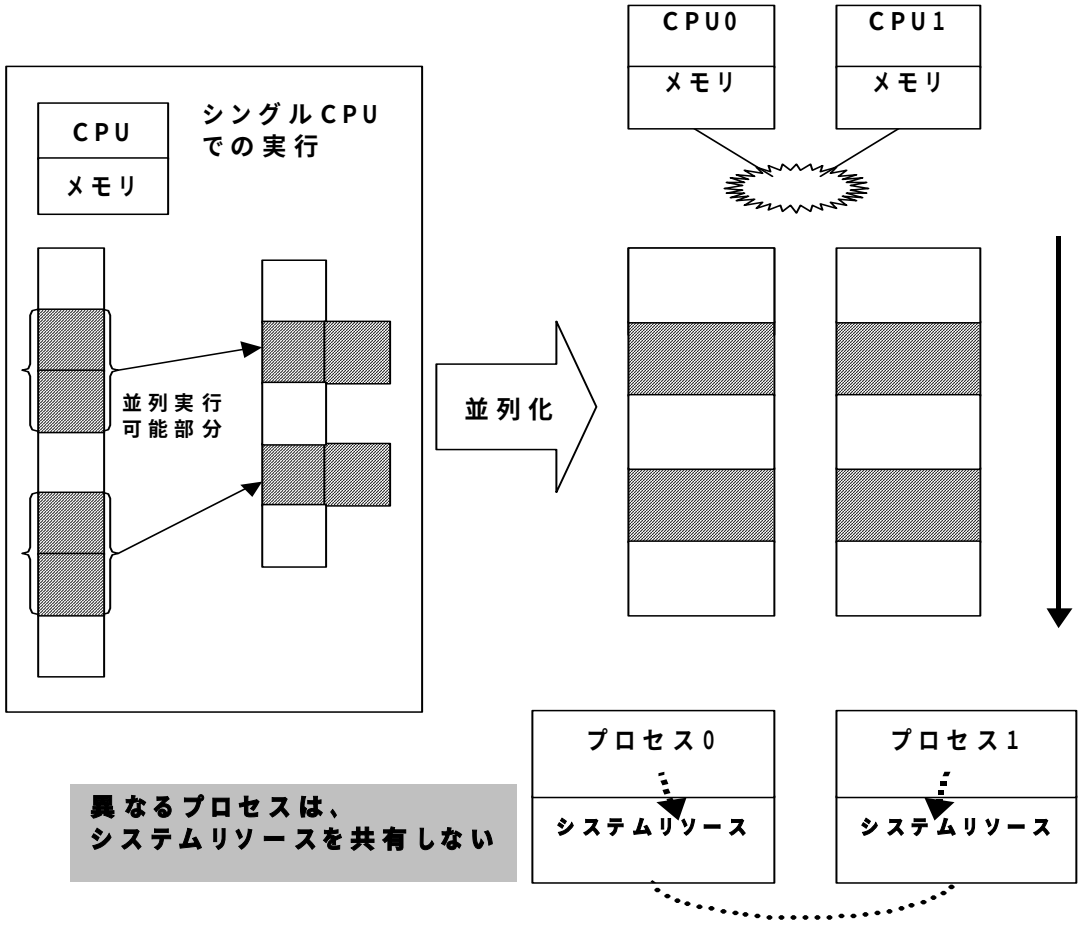
４．それぞれの処理終了後同期されて、スレッド１は消滅しスレッド０が実行を続ける

○生成されたスレッドはシステムリソース(メモリ，I／O)を共有する

○システムリソースへのアクセスの競合は、ユーザが制御しなければならない

特長 ・CPU 数を増やすには限界がある(～現在６４個程度)
・コンパイラによる自動並列化が進んでいる
→最高のパフォーマンスが得られるとは限らない
→メーカ独自のコンパイラ指示行を用いると他機種への移植性の問題が生じる
・OpenMP というコンパイラ指示文による並列化が業界標準となっている
↓
限られた時間でハイパフォーマンスなプログラムを開発するには有効である

◆ **分散メモリマシンでのプログラムの実行**



プログラムの動き

１．各 CPU で各プロセスが起動する

２．並列化された部分では、各プロセスがそれぞれ処理を１／N ずつ行う

○各プロセスは、システムリソースを共有しない

○他のプロセスのもつシステムリソースへのアクセスは、メッセージパッシングによって明示的に指定しなければならない。

特長 ・大規模並列化へのスケーラビリティが高い
・メッセージパッシングによるプログラミング
↓
手間をかければハイパフォーマンスが得られる
→『並列コンピューティングのアセンブリ言語』と呼ばれることもある

メッセージパッシングによるプログラミング
・分散メモリアーキテクチャのためのモデル
・共有メモリアーキテクチャでも実装することは可能
→プラットフォームに依存しない
・現在は、MPI(Message Passing Interface)が業界標準

◆ 並列プログラミングモデル

並列計算でのプログラミングモデルには、

・SPMD(Single Program Multiple Data)

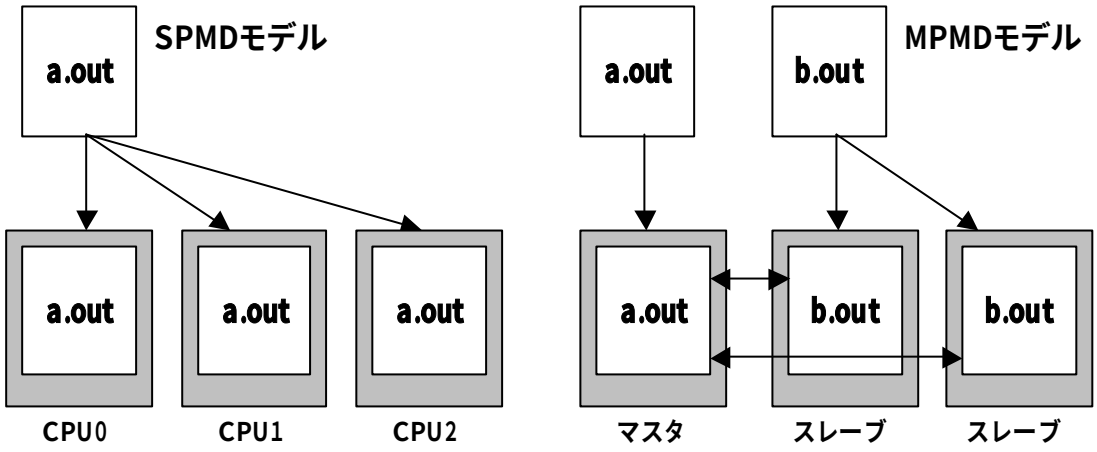
→ 一つのプログラムで並列処理を記述する

・MPMD(Multiple Program Multiple Data)

→ 複数のプログラムを協調的に動作させて並列処理を行う

(例えば、クライアント・サーバモデル)

の２つがありますが、科学技術計算の多くは SPMD モデルで並列化されます。



1.2. MPI プログラミングとは

MPI を利用して書いたプログラムはどういうものになるのでしょうか。
例として、次のような単純な一次元配列の計算を考えてみます。

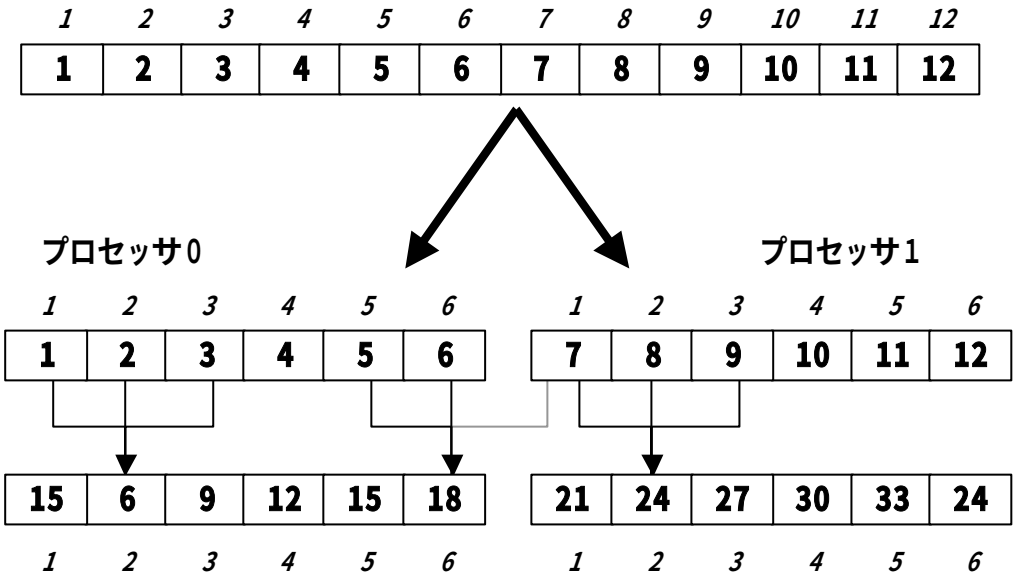
$$n'(i) = n(i-1) + n(i) + n(i+1); \quad \text{(周期的境界条件)}$$

シングルプロセッサでこの例を計算するプログラムを書いてみると、次のようになります。
境界部分の計算を見通しよくするために、“ のりしろ ” 部分をとって計算することにします。

```
parameter(nmax=12)
integer n(0:nmax+1)

n(0)=n(nmax)
n(nmax+1)=n(1)
do i=1,nmax
   n(i) = n(i-1)+ n(i)+n(i+1)
end do
end
```

周期的境界条件の計算ために、配列の最初と最後の部分に“ のりしろ ” 部分を余分に取っています

では、これを MPI を利用して２つのプロセッサに並列実行させるには、どう書いたらよいのでしょうか？
配列の前半部分をプロセッサ 0 が、後半部分をプロセッサ１が担当するとします。



```
include 'mpif.h'
parameter(nmax=12/2)
integer      n(0:nmax+1)

     ～
call mpi_?????
     ～
do i=1,nmax/2
   n(i)= n(i-1)+n(i)+n(i+1)
end do
call mpi_?????
     ～
```

少し上の例について考えてみます。
例えば、配列の６番目の要素を計算するためには、５，６，７番目配列の要素が必要になりますが、７番目の要素は隣のプロセッサがローカルに持っているデータなので、隣のプロセッサから何らかの方法でこの要素の値を教えて貰わなければなりません。このように処理を並列化することによりプロセッサ間でデータのやり取りが発生します。ここでは詳細は述べませんがこれをどうやって実現するのでしょうか？ また、この例では配列ｎに適当なデータがすでに正しく入っているものと勝手に考えていますが、異なるプロセッサに正しくデータを読み込ませるにはどうしたらよいのでしょうか？ このように処理を並列化することによって、並列化に伴う不足分を補ってやらなければなりません。
次章からこのような並列プログラミングをどのようにして書いていくかについて説明します。

2. MPI プログラミングをはじめる前に

並列プログラミングの説明に入る前に、この章では本書で説明する MPI について紹介します。また、プログラムのコンパイルと実行の方法について簡単に説明します。コンパイルと実行の詳細については、システムの利用の手引きもご参照ください。

2.1. MPI とは

MPI とは、Message-Passing Interface の略称で、現在では業界標準のメッセージ交換用のライブラリです。前の章で少し触れたように、プログラミングの並列化に伴い、シングルプロセッサでは必要のなかった処理を加えなければ正しい答を導き出せなくなってしまいます。この不足分を補うために必要最小限度、通信(ネットワークあるいはスイッチなど)を介してプロセッサにローカルなメモリにあるデータのやり取りをします。この通信の機能とその呼び出し方法を規定しているのが MPI です。MPI は規格であり、MPI フォーラムという会議によって規格されています。

MPI フォーラムでは、１９９４年に MPI の規格第一版を発表し、発表とともにその実装サンプルである MPICH を配布しました。現在では第二版が発表されていますが、第二版のすべての規格を実装しているものはそれ程多くありません。Compaq では Argonne National Laboratories の MPICH V1.1.1 をもとに、Alpha アーキテクチャ用に最適化を加えた Compaq MPI を提供しています。

MPI は、ハードウェアやオペレーティングシステム、コンパイラに依存しません。Compaq MPI を利用して Alpha Server 上で作成された MPI プログラムは、他のマシンに移植しても再コンパイルすることによって動作させることができます。

2.2. プログラムを動かしてみよう　―　コンパイル

C 言語の入門書に倣って、”Hello World”を表示する最も簡単な MPI のプログラムをコンパイル&実行してみることにします。このプログラムは、次章からの説明を読むと簡単に理解できますので、プログラムの説明については次章にまかせることにします。

MPI プログラム例：test.f

```
  include 'mpif.h'

  call mpi_init(ierr)
  print *, 'Hello World!'
  call mpi_finalize(ierr)
  end
```

Compaq MPI プログラムのコンパイルは、通常のプログラムのコンパイルと同じですが、コンパイルオプション『 –lfmpi –lmpi –lrt –lpthread 』を付加する必要があります。

○ Fortran の場合

f90 –o test test.f90 **–lfmpi -lmpi –lrt –pthread**

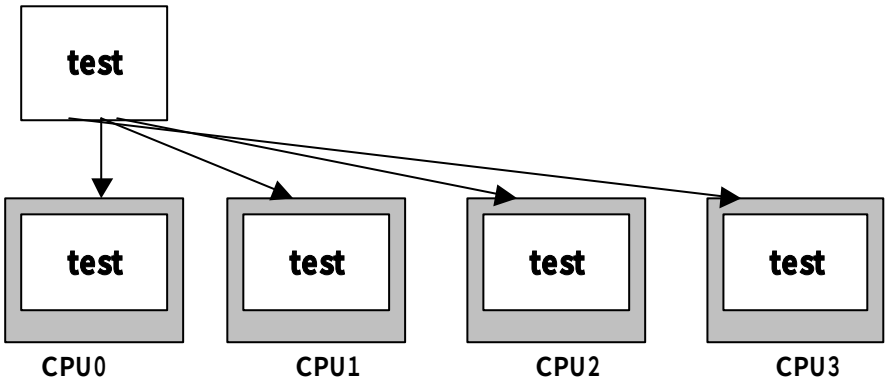
○ C 言語の場合

cc –o test test.c **–lmpi –lrt –pthread**

2.3. プログラムを動かしてみよう　―　実行

上のプログラムを４つのプロセッサで並列実行させてみることにします。
MPI プログラムを実行させるには dmpirun コマンドを使用します。dmpirun コマンドでは、いくつのプロセッサで MPI プログラムを実行させるかを–np オプションの直後にその数を指定することにより、複数プロセッサでプログラムが実行されます。–np オプションを省略するとシングルプロセッサでの実行になります。

```
alpha> dmpirun  –np 4  test
Hello World!
Hello World!
Hello World!
Hello World!
alpha>
```



注意	正確に言うと複数プロセッサではなく複数プロセスです。複数のプロセッサを搭載してあるマシンでは、OS が各々のプロセスを複数のプロセッサに分配して実行させます。ところが UNIX システムでは、シングルプロセッサのマシンでも複数プロセスが実行できます。つまり、パフォーマンスを全く問題にしない場合、シングルプロセッサのマシンでも MPI の並列プログラムを実行することは可能です。

このプログラムは SPMD 方式のため各プロセッサで同じプログラムが一斉に実行されます。そのため、各４つのプロセッサが”Hello World!”を各々独立に表示します。各プロセッサが別々に同じことをするだけでは、並列処理の目的である「計算量を１／N に縮小させる」という目的は果たせません。

それでは、処理を分割して並列処理させるにはどうしたらよいのでしょうか？

↓

同じプログラムで各プロセスに異なる動作をさせるには、

ランク(RANK)

という情報が必要になります。

次の章からは、ランクを利用した並列プログラミングを実際にはじめてみます。

３．MPI プログラミング～はじめの一歩

この章では、次のような簡単な配列の総和

$$isum = \sum_{i=1}^{N} n(i) \qquad \cdots式①$$

を求める MPI プログラムの例からはじめることにします。
次に、１章で紹介した一次元配列の演算を MPI でプログラミングする方法について説明します。

$$n'(i) = n(i-1) + n(i) + n(i+1) \qquad \text{[周期的境界条件]} \qquad \cdots式②$$

[固定境界条件] …式③

これらの演算をするためには、MPI プログラミングにおいてなくてはならないプロセス間のデータ通信を行うことが必要になります。

３.１. MPI プログラミングとは

MPI サブルーチンには、次のような種類があります。

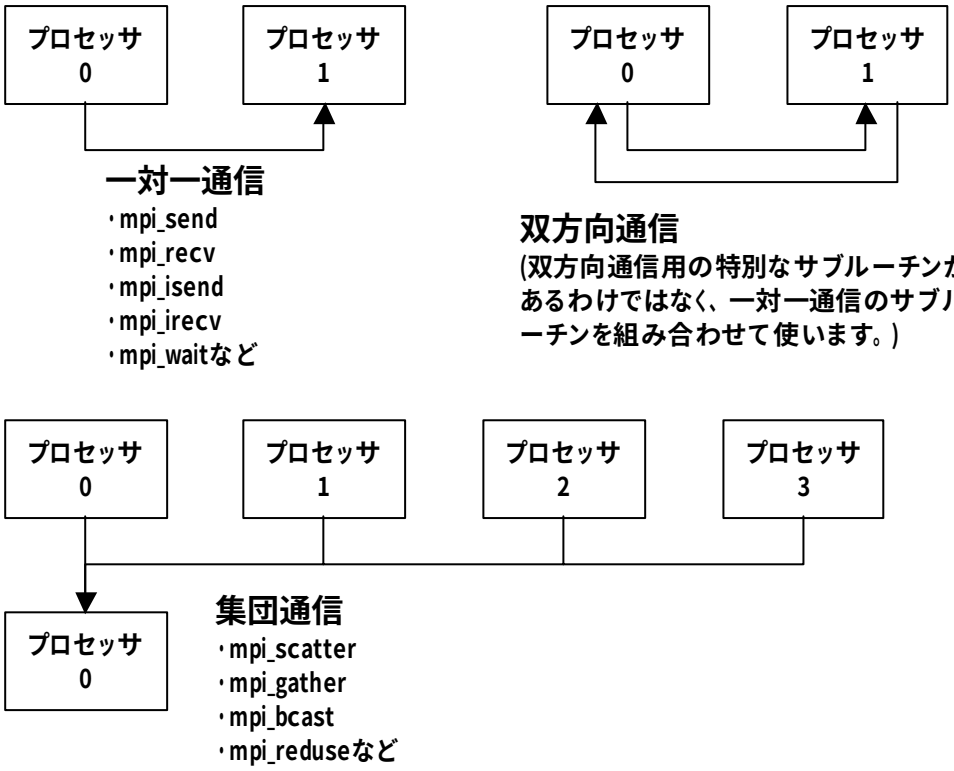
- MPI 環境制御サブルーチン
- 一対一通信サブルーチン
- 集団通信サブルーチン
- 派生データタイプ
- コミュニケータ
- プロセストポロジ

すべて合わせると百個以上のサブルーチンがありますが、数値計算で使用するサブルーチンは約４０個です。その中でも特によく使う重要なサブルーチンは１０個程度です。

本章ではこの約１０個のサブルーチンについて、それらの使い方を説明していきます。

- MPI 環境制御サブルーチン
  MPI プログラミングをはじめるための、初期化や終了処理などを行うサブルーチンです。またランク情報の取り出しなどを行うサブルーチンもあります。
- MPI 一対一通信サブルーチン
  あるプロセスから別のプロセスへデータなどを送るためのサブルーチンです。
  また、一対一通信を組合わせた双方向による通信(これ以降本章では、便宜上双方向通信を呼ぶことにします)もあります。一対一通信の組合わせですが、双方向通信には特別の注意が必要となります。
- MPI 集団通信
  グループに属するプロセスすべてにデータを送受信するためのサブルーチンです。
  データを送受信するだけでなく、データ受信時に簡単な演算を加えることも可能です。

これらのサブルーチン、特に通信関連サブルーチンの使い方が分かれば MPI プログラミングをはじめることができるようになります。



3.2. MPI 環境制御ルーチン

◆ 同じプログラムで各プロセスに異なる動作をさせる

各プロセスに処理を分散させるためには、各プロセスに固有の ID があればその ID に従って処理を分けることが可能になります。MPI ではシステムから各プロセスに

ランク(RANK)

という ID が付与されます。それではランクを取り出すプログラムを見てみましょう。

```
include 'mpif.h'

call mpi_init(ierr)
call mpi_comm_size(MPI_COMM_WORLD,isize,ierr)
call mpi_comm_rank(MPI_COMM_WORLD,irank,ierr)
print *, 'isize=', isize, ' irank=', irank

call mpi_finalize(ierr)
end
```

mpif.h を必ずインクルードする！

MPI 初期化

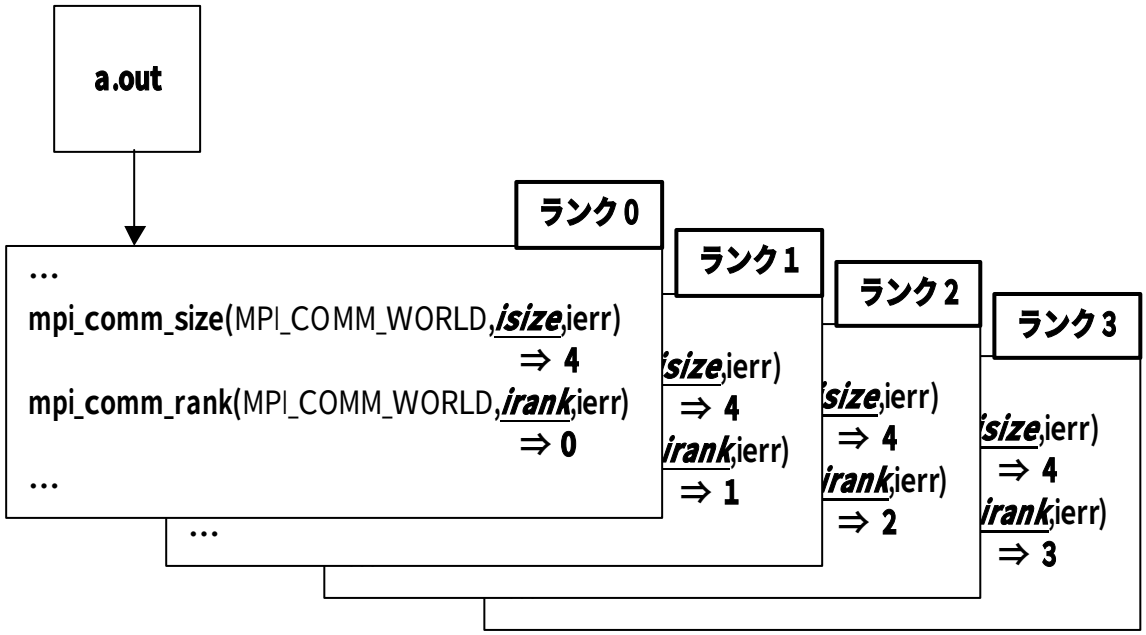
プロセスサイズを取出す

ランクを取出す！

MPI 終了処理

このプログラムを、４つのプロセッサで並列に実行させると次のようになります。

```
alpha> dmpirun  –np 4  a.out
isize=4  irank=0
isize=4  irank=1
isize=4  irank=2
isize=4  irank=3
```



実行時の－np オプションに４を指定したので、このプログラムは４プロセスで実行されます。そして、isize にはそのプロセス総数が返され、irank が求めるランクの ID になります。このようにして取出されたランクの番号は、必ず０からはじまる続き番号になります。ランク番号はあるプロセスに固有の ID だと考えることができます(ただし、オペレーティングシステムの管理するプロセス ID そのものではありません)。
次はこのランクを利用して計算を分割させる方法について考えることにします。

mpi_init(ierr)			
機能	MPI 環境の初期化処理を行う		
引数	型	入出力	
ierr	Integer	出力	終了コード

mpi_finalize(ierr)			
機能	MPI 環境の終了処理を行う		
引数	型	入出力	
ierr	Integer	出力	終了コード

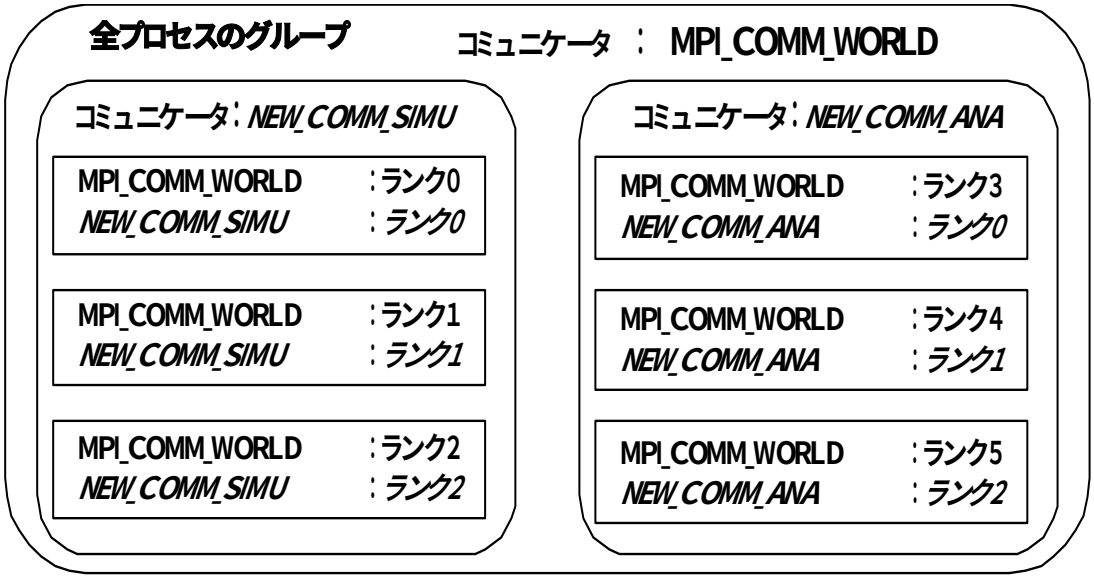
mpi_comm_size(communicator, size, ierr)			
機能	コミュニケータグループに属するプロセスの数を返します		
引数	型	入出力	
communicator	Integer	入力	コミュニケータを指定
size	Integer	出力	指定されたコミュニケータ内のプロセスの数
ierr	Integer	出力	終了コード

mpi_comm_rank (communicator, rank, ierr)			
機能	コミュニケータグループに属するランクの値を返します		
引数	型	入出力	
communicator	Integer	入力	コミュニケータを指定
rank	Integer	出力	コールしたプロセス(自分)のランク
ierr	Integer	出力	終了コード

ちょっと寄り道

本書の範囲ではコミュニケータについて意識する必要はありません。本書では、MPI によってあらかじめ定義されているコミュニケータ **MPI_COMM_WORLD** のみ取り扱います。コミュニケータは、それに属するプロセスのグループを代表する ID のようなものです。コミュニケータ **MPI_COMM_WORLD** によって指定されるグループは、プログラム実行時に生成される全プロセスからなるグループです。

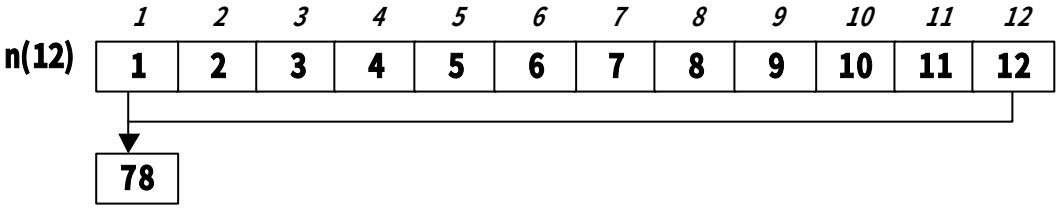
そして、ランクはコミュニケータごとに付けられます。例えば２つのコミュニケータがあった場合、それぞれのコミュニケータごとにランクが０，１，２，…と割当てられます。



◆ **ランクを利用して計算領域を分割させる**

この章のはじめに述べた一次元配列の総和を求める式①の例について考えることにします。

$$isum = \sum_{i=1}^{N} n(i) \qquad \cdots式①$$

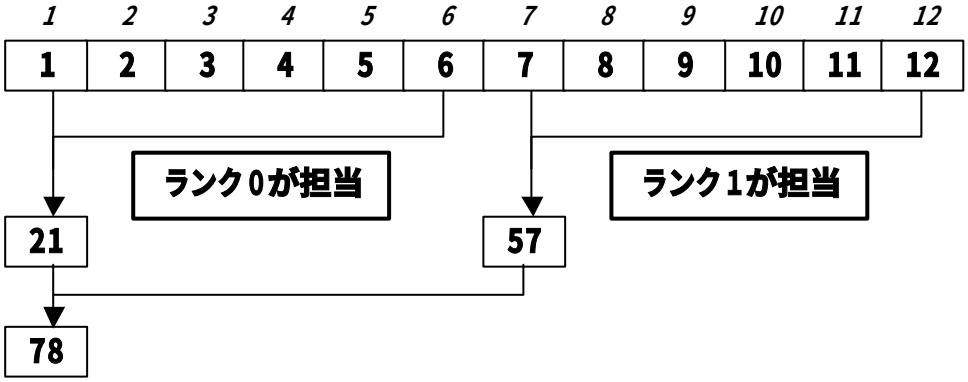


シングルプロセッサの場合のプログラムは次のようになります。

```
parameter(nmax=12)
integer n(nmax)                    シングルプロセッサの場合
isum=0
do i=1,nmax
   isum = isum + n(i)
end do
end
```

これを２つの CPU に分割させて並列実行させてみることにします。

配列の１から６まではランク０が担当し、配列の７から１２まではランク１が担当することにします。プログラムの動作の概要は下図のようになります。




```
include 'mpif.h'
parameter(nmax=12)
integer n(nmax)

call mpi_init(ierr)
call mpi_comm_size(MPI_COMM_WORLD,isize,ierr)
call mpi_comm_rank(MPI_COMM_WORLD,irank,ierr)

ista = irank*(nmax/isize)+1
iend = ista+(nmax/isize -1)
do ii = 1,nmax
   n(i) = I
```

```
    end do

    do i = ista,iend
       isum = isum + n(i)
    end do
    print *,'ista=', ista, 'iend=', iend, 'isum=', isum
    call mpi_finalize(ierr)
    end
```

このプログラムを実行すると、次のような結果が得られます。

```
          ista   iend    isum
-----+------+------+-------
rank0： 1      6       21
rank1： 7      12      57
```

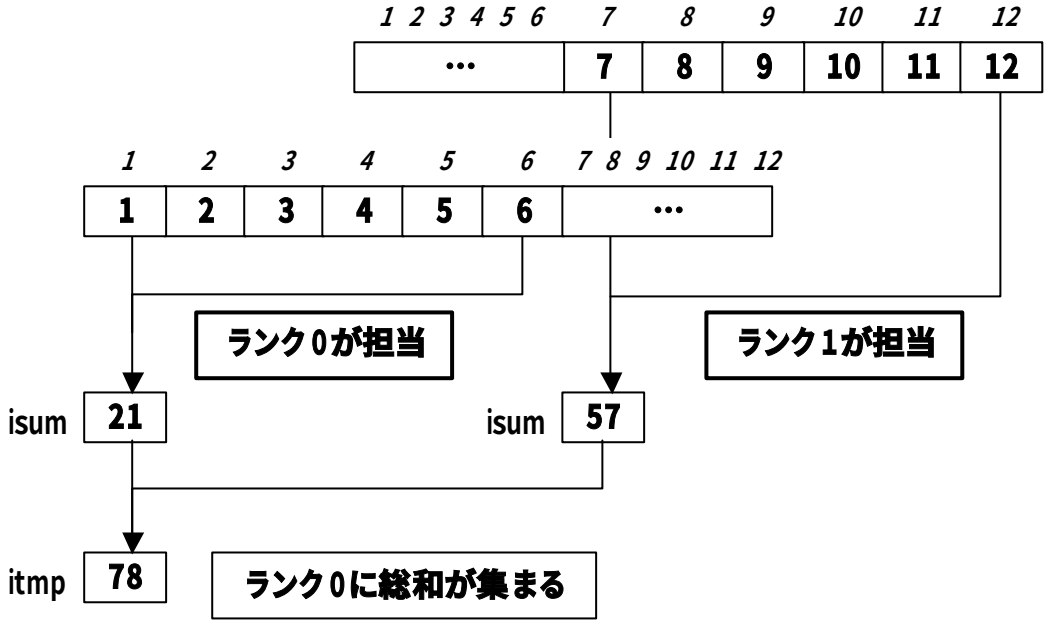
この結果のように、単に配列を分割しただけでは全体の総和は得ることができません。はじめに書いたように、このような並列化に伴う不足分を補ってやらなければなりません。
では、配列を全部足し合わせるにはどうしたらよいでしょうか？

3.3. MPI 集団通信

この節では、MPI のサブルーチンを用いて『配列を全部足し合わせる』例を示します。

◆ **集団通信(mpi_reduce)を使う**

上の例では、各ランクが担当した部分についての和はすでに求められています。それらの和をさらに足し合わせれば全体の総和が得られます。全体に総和を得るには、**mpi_reduce** という集団通信サブルーチンを使用します。



```
      include 'mpif.h'
      parameter(nmax=12)
      integer n(nmax)

      call mpi_init(ierr)
      call mpi_comm_size(MPI_COMM_WORLD,isize,ierr)
      call mpi_comm_rank(MPI_COMM_WORLD,irank,ierr)
      ista=irank*(nmax/isize)+1
      iend=ista+(nmax/isize-1)
      do i=1,nmax
         n(i) = i
      end do

      isum=0
      do i=ista,iend
         isum = isum + n(i)
      end do
      call mpi_reduce(isum,itmp,1,MPI_INTEGER,MPI_SUM, &
                      0, MPI_COMM_WORLD,ierr)

      if(irank == 0) then
         isum=itmp
         print *,'isum=', isum
      endif
      call mpi_finalize(ierr)
      end
```

各ランクの sum の値をランク０に集めながら、それらの総和を求める

このように MPI には通信しながら演算を行うサブルーチンがあります。**mpi_reduce** はそのサブルーチンの一つです。**mpi_reduce** の演算には和を求める以外にも、積、最大・最小を求めるなどの演算があります。

mpi_reduce (sendbuf, recvbuf, count, datatype, op, root, comm, ierr)			
機能	コミュニケータグループに属する全プロセスからの送信データが、宛先プロセス(root)の受信バッファに格納されます。格納時には演算子 op で定義された演算が施されます。		
引数	型	入出力	
sendbuf	アドレス	入力	送信バッファのアドレス
recvbuf	アドレス	出力	受信バッファのアドレス(root のみ意味をもつ)
count	Integer	入力	データタイプで指定された要素の個数
datatype	Integer	入力	送信データのデータタイプ[＊1]
op	Integer	入力	受信データに施される演算の種類[＊2]
root	Integer	入力	宛先プロセスのランク
comm	Integer	入力	属するプロセスグループのコミュニケータ
ierr	Integer	出力	終了コード

＊１：MPI で定義されているデータタイプ

Fortran でのデータタイプ	(byte 数)	MPI でのデータタイプ
integer, integer*4	4	MPI_INTEGER
real, real*4	4	MPI_REAL
double precision, real*8	8	MPI_DOUBLE_PRECISION, MPI_REAL8
complex	8	MPI_COMPLEX
double complex, complex*16	16	MPI_COMPLEX16
character	1	MPI_CHARACTER
byte	1	MPI_BYTE
logical, logical*4	4	MPI_LOGICAL

＊２：MPI で提供されている演算の種類

演算		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	〃

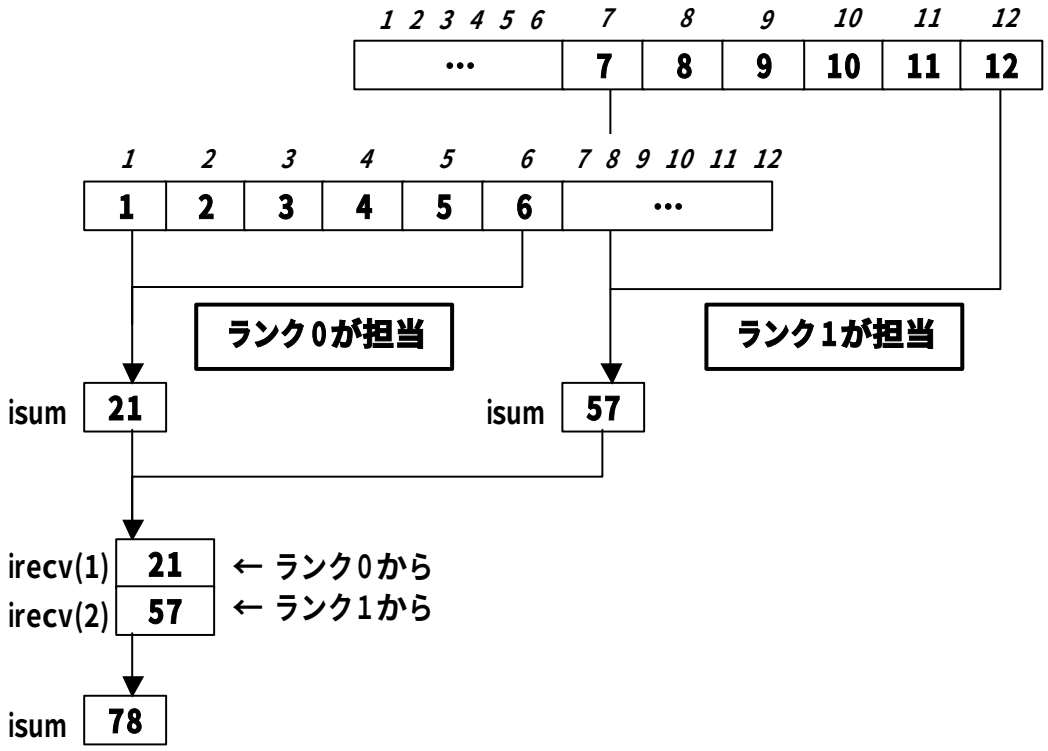
補足 MPI_MAXLOC と MPI_MINLOC は他とは少しタイプが異なっています。これらは大域的な最大値・最小値とその位置するインデックスの両方を求めます。これらで演算可能なデータタイプは、Fortran と C 言語で異なっています。Fortran では２つの同じデータタイプの組を使いますが、C 言語ではインデックスの方のタイプはすべて“ int ”です。これらの演算はデータタイプなどに十分に注意しご使用ください。

ちょっと寄り道

ここで少し寄り道することにします。本章の最終目的である配列の計算からは少しはずれてしまいますので、ここを飛ばして３．４節へ進んでもかまいません。
別の集団通信サブルーチンを使用して同じように配列を足し合わせる例を紹介します。

◆ 集団通信(mpi_gather)を使う

mpi_reduce では定義された演算以外のものについては、MPI_OP_CREATE というサブルーチンを使用することによりユーザが新たに演算を定義することができます。あるいは、次の例のように **mpi_gather** というサブルーチンを使いデータを一度ランク０に集めてしまい、その後自ら演算を行うということも可能です。




```
include 'mpif.h'
parameter(nmax=12)
integer n(nmax)
integer irecv(2)

     ～
do i=ista,iend
   isum = isum + n(i)
end do
```

```
    call mpi_gather(isum,1,MPI_INTEGER,irecv,1, &
                    MPI_INTEGER,0,MPI_COMM_WORLD,ierr)

    if(irank == 0) then
       isum = 0
       do i=1,2
          isum = isum + irecv(i)
       end do
       print *,'isum=', isum
    end if
    call mpi_finalize(ierr)
    end
```

各ランクの isum の値をランク０に集めて、配列 irecv に格納する

このように、配列 irecv に各ランクの isum の値を収集だけして、その後に和を求めることもできます。各ランクからデータを収集するルーチンには mpi_gather 以外にも mpi_gatherv など幾つかあります。また逆に、データを各ランクに分配する mpi_scatter というサブルーチンもあります。

注意 上記プログラムにおいて、配列 irecv の和を求める部分は、このままではランク０以外の他のランクでも実行されてしまいます。もしこの部分に割り算などの演算が含まれていると、ランク０以外の他のランクでは配列 irecv の中身は不定なので、０除算などの演算によりエラーになる場合があります。このような状況を回避するには、ランク番号を利用してこの演算をランク０のみに実行させるようにします。あるいは、ランク０の配列の内容を他のランクにコピーする mpi_bcast などのルーチンを使用して回避することもできます。

まとめ 集団通信には、収集と分配を行うルーチンがいくつかあります。

収集ルーチン：

- mpi_gather
- mpi_gatherv
- mpi_allgather
- mpi_allgatherv

分配ルーチン：

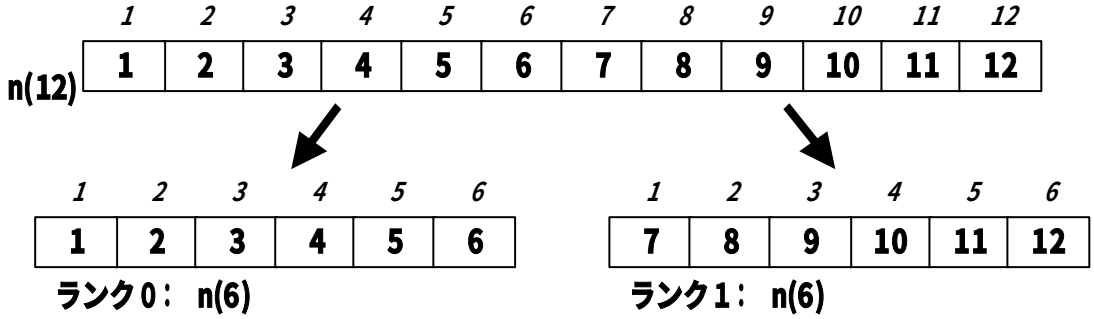
- mpi_scatter
- mpi_scatterv

３.４. MPI 一対一通信

この節では『配列を領域分割して足し合わせる』ことを目的とします。
計算を実行してみる前に、一対一通信と双方向通信のしくみについて説明します。

◆ 一対一通信を使う

前の例では、計算処理は各 CPU に分割して実行しましたが、各ランクがそれぞれに配列ｎをメモリ上にもっていました。大規模な計算を行う場合、メモリ不足のためにそれぞれが同じ配列をメモリ上にもてない場合があります。このような場合データ領域を分割して各 CPU に割当てれば、メモリは分割した分だけ増やせることになります。



ここで、今までの例と同じく

$$isum = \sum_{i=1}^{N} n(i)$$

という計算をする限りは、単純にデータ領域を分割してもプログラムに大きな修正を加えなくてもかまいませんので、この章のはじめに紹介した次のような例を考えることにします。

$$n'(i) = n(i-1) + n(i) + n(i+1) \quad \text{[周期的境界条件]} \qquad \cdots式②$$

この場合、

```
        ～
do i=ista,iend
      n(i) = n(i-1)+ n(i)+n(i+1)
end do
        ～
```

という計算に修正しただけでは、正しい答を得ることはできません。
例えばランク０の６番目の配列の計算を行う場合、ランク１に存在する隣の７番目のデータが必要になります。それではランク１からデータを取ってくるにはどうしたらよいでしょうか？

このような場合のために、MPI には一対一通信というサブルーチン群が用意されています。実際に上記の計算を行う前に、一対一通信と双方向通信の簡単な説明をします。

◆ 一対一通信のしくみ

まずは一対一通信の説明からはじめます。一対一通信には、ブロッキング通信とノンブロッキング通信があります。

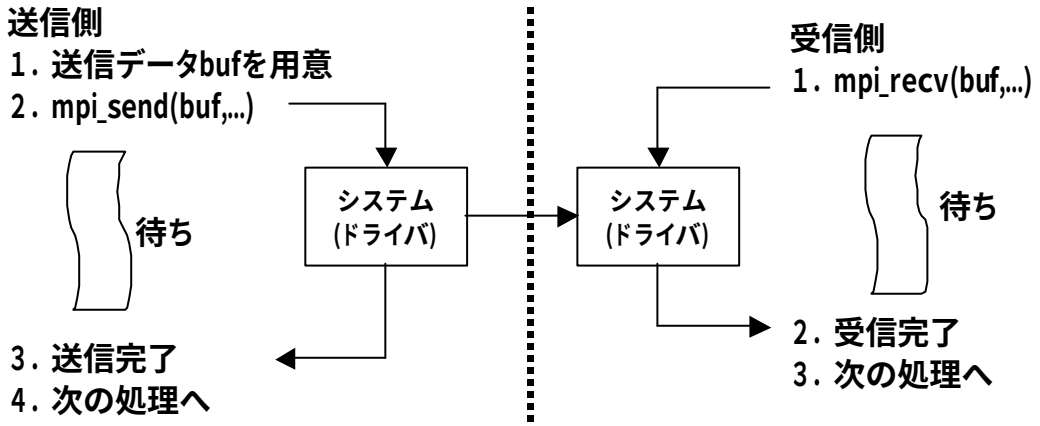
・ブロッキング通信とは、送信もしくは受信の指示があった場合、その処理が完了する(完了するとは相手側がすべてのデータを受信することではなく、送信側で送信バッファの使用が完了した状態のことを指します)まで次の処理に進まない(処理をブロックする)通信のことです。
・ノンブロッキング通信とは、送信もしくは受信処理の完了を待たずに「送れ」と指示を出したら次の処理へ進みます。プログラム中で送れという指示だけ出せばそれでよいという状況はほとんどありませんので、どこかで送信処理が完了したことを確認しなくてはなりません。この確認のために「待つ(wait)」処理とペアで使います。

MPI でよく使われる一対一通信ルーチンを以下に示します。

	送信	受信	
ブロッキング	mpi_send	mpi_recv	
ノンブロッキング	mpi_isend	mpi_irecv	mpi_wait

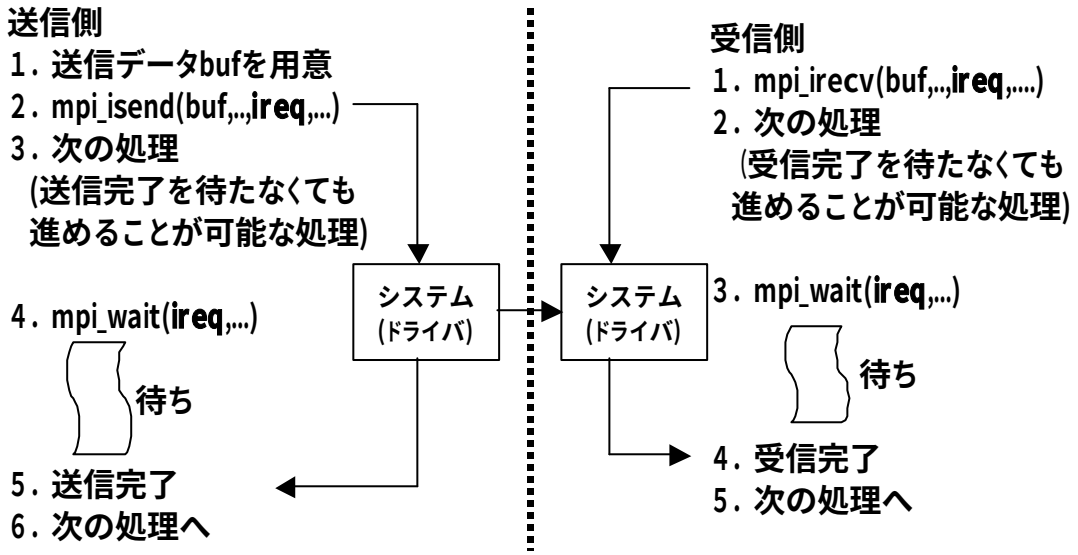
※『i』は、immediate の略称

○mpi_send と mpi_recv の動作について



※送信データは、システム(ドライバ)側へ渡された後、相手先への送信処理が行われます。
※受信データは、システム側で一旦受け取り、受信側バッファへ転送処理されます。

○mpi_isend と mpi_irecv の動作について



上の図を見ると mpi_isend，mpi_irecv，mpi_wait のルーチンを使用した方が、処理を効率的に進められそうに思われるかもしれません。しかしながら、上位のプログラムからは見えませんが、通信処理も OS つまりはソフトウェアによって実行されている部分が大半です。そのため「３番の次の処理」を進められるという効果は、場合によりますがそれ程大きくはなりません。

補足	mpi_send あるいは「mpi_isend＋mpi_wait」のどちらを用いて送信した場合でも、mpi_recv あるいは「mpi_irecv＋mpi_wait」のどちらを用いてでも受信することが可能です。 例えば、mpi_send で送信して、mpi_irecv＋mpi_wait で受信することができます。

注意	mpi_isend や mpi_irecv の後に mpi_wait をコールしなくても、計算結果が正しくなる場合があります。特に mpi_isend と mpi_wait をコールすべき位置の間の処理が十分に長ければそうなる場合があります。 ただし、十分に長いという時間がどれぐらいかは、システムの状態にも影響されてしまいます。このようにある種偶然に左右されるようなプログラムでは、再現性のないエラーが発生することがあります。再現性のないエラーをデバッグするのは非常に困難な作業です。このようなことのないよう mpi_wait を忘れずにコールしましょう。

◆ **一対一通信**

自分のランクを相手におくる一対一通信の例を以下に示します。SPMD モデルではプログラムは一つなので、一つのプログラム中に送信と受信を記述します。

```
      include 'mpif.h'
      integer mstatus(MPI_STATUS_SIZE)
      integer bno

      call mpi_init(ierr)
      call mpi_comm_rank(MPI_COMM_WORLD,irank,ierr)

      bno = irank
      if(irank == 0) then
         call mpi_send(bno,1,MPI_INTEGER,1,1, &
                              MPI_COMM_WORLD,ierr)
      else if(irank == 1) then
         call mpi_recv(bno,1,MPI_INTEGER,0,1, &
                        MPI_COMM_WORLD,mstatus,ierr)
      endif

      print *,'I am rank ', irank,'. Hello rank ', bno,' !'
      call mpi_finalize(ierr)
      end
```

```
      alpha> a.out
      I am rank 0. Hello rank 0!
      I am rank 1. Hello rank 0!
      alpha>
```

mpi_send (buf, count, datatype, dest, tag, comm, ierr)			
機能	送信バッファのメッセージを宛先ランクに送信する		
引数	型	入出力	
buf	アドレス	入力	送信バッファのアドレス
count	Integer	入力	データタイプで指定された要素の個数
datatype	Integer	入力	送信データのデータタイプ
dest	Integer	入力	送信したい相手先のランク
tag	Integer	入力	メッセージのタグ＊1
comm	Integer	入力	属するプロセスグループのコミュニケータ
ierr	Integer	出力	終了コード

＊１：タグは、送信するメッセージを受信相手に区別させたい場合に使用します。
特に区別する必要がない場合は、“ １ ”などの適当な値を指定します。

mpi_recv (buf, count, datatype, source, tag, comm, status, ierr)			
機能	送信元から送られたメッセージを受信バッファに受信する		
引数	型	入出力	
buf	アドレス	入力	受信バッファのアドレス
count	Integer	入力	データタイプで指定された要素の個数
datatype	Integer	入力	受信データのデータタイプ
source	Integer	入力	送信元のプロセスのランク
tag	Integer	入力	メッセージのタグ＊1
comm	Integer	入力	属するプロセスグループのコミュニケータ
status	アドレス	出力	通信結果の状態＊2
ierr	Integer	出力	終了コード

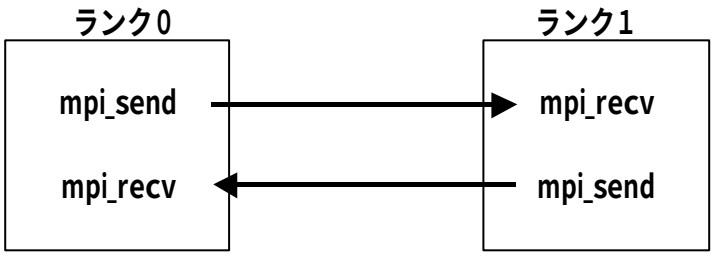
＊１：タグは、受信したいメッセージに付けられているタグの値を指定に使用します。通常送信元プロセスが送信時につけたタグを指定します。任意のタグ番号のメッセージを受信することも可能ですがここでは説明しません。

＊２：status には受信結果の状態が納められます。これらの値を通常使用することはありませんが、これらの値の格納用に MPI_STATUS_SIZE の大きさの配列を用意しておかなければばなりません。

◆ 双方向通信

この章のはじめに説明したように、双方向での通信のための特別なサブルーチンが存在するわけではありません。プログラム上同じ位置で、A から B への送信と B から A への送信が同時に必要になった場合、一対一通信を組み合わせて使用します。ただし、この組み合わせには注意が必要です。そのため、ここでは「双方向通信」として分けて説明します。
(ただし、mpi_sendrecv というサブルーチンが一つあります)

２つのランク間で双方向通信を行うことを考えます。一対一通信の例では、ランク０からランク１へランク番号を送っただけなので、双方でランクを送りあうことにします。



```
      include 'mpif.h'
      integer mstatus(MPI_STATUS_SIZE)
      integer bsnd, brcv

      call mpi_init(ierr)
      call mpi_comm_rank(MPI_COMM_WORLD,irank,ierr)

      bsnd = irank
      if(irank == 0) then
         call mpi_send(bsnd,1,MPI_INTEGER,1,1, &
                       MPI_COMM_WORLD,ierr)
         call mpi_recv(brcv,1,MPI_INTEGER,1,1, &
                       MPI_COMM_WORLD,mstatus,ierr)
      else if(irank == 1) then
         call mpi_recv(brcv,1,MPI_INTEGER,0,1, &
                       MPI_COMM_WORLD,mstatus,ierr)
         call mpi_send(bsnd,1,MPI_INTEGER,0,1, &
                        MPI_COMM_WORLD,ierr)
      endif

      print *,'I am rank ', irank,'. Hello rank ', brcv,' !'
      call mpi_finalize(ierr)
      end
```

rank 1 へ送信

rank 1 から受信

rank 0 から受信

rank 0 へ送信

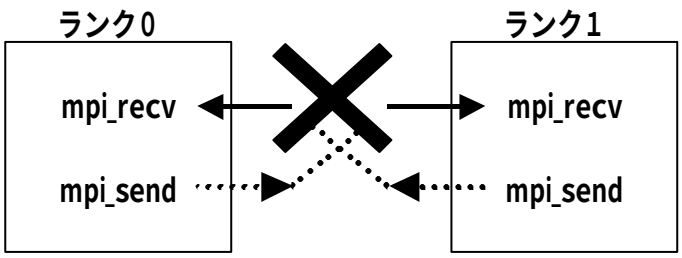
```
      alpha> a.out
      I am rank 0. Hello rank 1!
      I am rank 1. Hello rank 0!
      alpha>
```

ここで、ランク０とランク１で mpi_send と mpi_recv をコールする順番が逆になっていることに注意してください。双方向通信ではこのようにプログラムを記述します。このように記述しない場合は、その双方向通信の場所で双方のプログラムが次の処理へ進めなくなる『デッドロック』という現象が発生することがあります。

注意	mpi_send と mpi_recv サブルーチンでは、引数の数が違うことに注意してください。mpi_recv サブルーチンには、受信完了のステータスを格納するために mstatus という配列(配列の数は、MPI_STATUS_SIZE)の引数が一つ多く必要になります。これを忘れた場合、デッドロックのような現象が発生することがあります。

○デッドロックの例 － その１

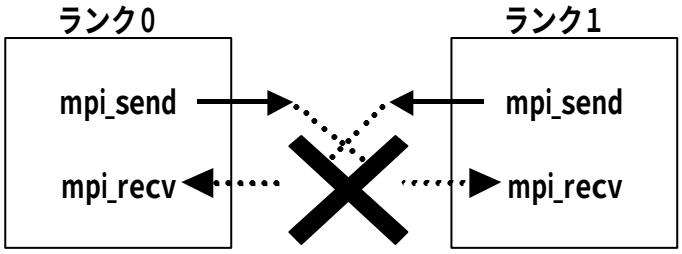
```
if(irank == 0) then
    call mpi_recv(.....)
    call mpi_send(.....)
else if(irank == 1) then
    call mpi_recv(.....)
    call mpi_send(.....)
end if
```



この例では、はじめに双方でブロッキングの受信待ちをコールしていますので、互いに受信待ちのまま次の処理へ進めなくなってしまいます。

○デッドロックの例 ― その２

```
if(irank == 0) then
    call mpi_send(.....)
    call mpi_recv(.....)
else if(irank == 1) then
    call mpi_send(.....)
    call mpi_recv(.....)
end if
```



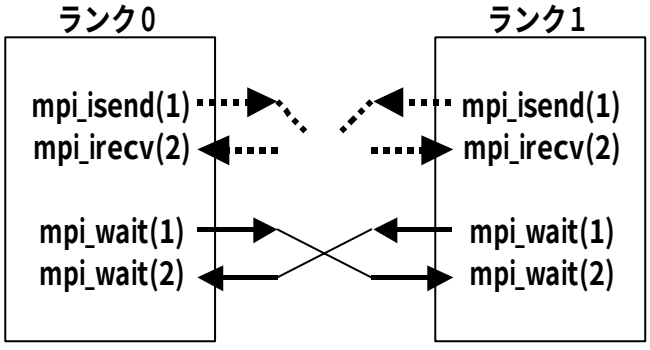
この例は、送受信されるデータの大きさによっては動作することもありますが、偶然に左右されてしまうかもしれないようなプログラムを記述することは賢明なことではありません。

この現象の発生する原因を簡単に考えてみます。送信されるデータがある程度大きくなると、実際に転送を受持つドライバは、mpi_send より渡されたデータを一度に相手先ランクへ送信できない場合、複数回に分けて送信処理します。相手側でもこちら側へ送信処理をしようとしていれば、双方で受信の処理へ進めずデッドロックとなります。しかし、場合によっては送信バッファの使用が完了して受信処理へ進めることがあり([注]完了するとは相手側がすべてのデータを受信することではありません)、デッドロックとはならないことがあります。

○お手軽な双方向通信

ここでは、お手軽に使える双方向通信の例を紹介します。ノンブロッキング通信を利用しているためデッドロックに注意しなくてよくなります。

```
call mpi_isend(..., ireq1,...)
call mpi_irecv(..., ireq2,...)
call mpi_wait(ireq1,..)
call mpi_wait(ireq2,..)
```



双方向通信のはじめに示した例では、
ランク１側で受信処理が完了しないうちはランク１は送信作業に移れませんが、このプログラムでは、ランク１側でも送信要求を先にすることができます。そして mpi_wait(2)がコールされた後すべての送受信が完了します。このお手軽な双方向通信の方法を使って次の項の問題を解いてみることにします。

補足　次の例もデッドロックしないお手軽な双方向通信です。

1．call **mpi_isend**(..., ***ireq1***,...)
　　call **mpi_recv**(.....)
　　call **mpi_wait**(***ireq1***,...)

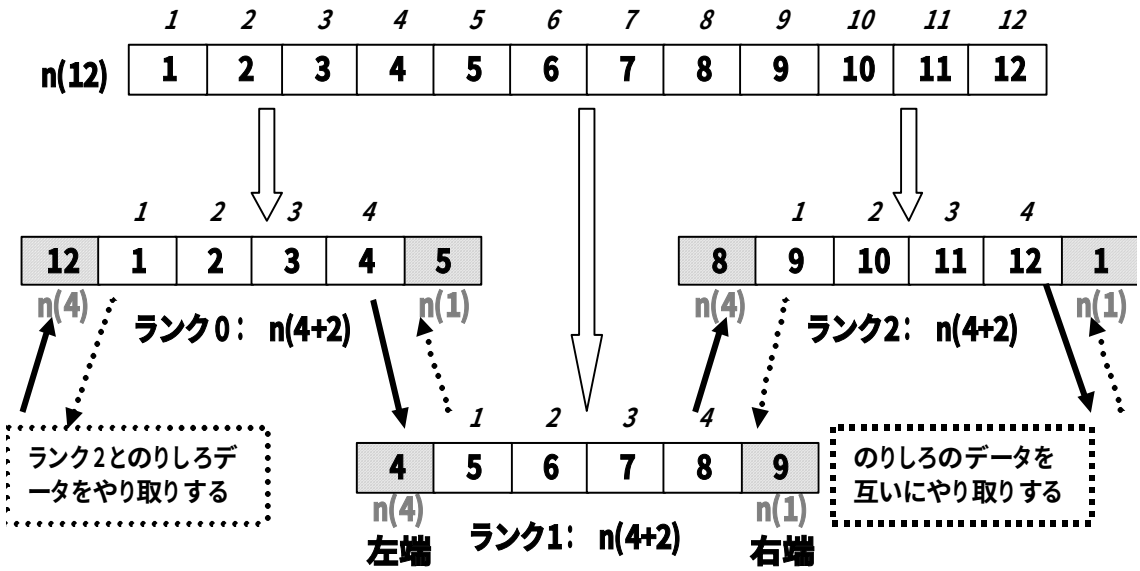
2．call **mpi_irecv**(..., ***ireq1***,...)
　　call **mpi_send**(.....)
　　call **mpi_wait**(***ireq1***,...)

◆ 配列を領域分割して足し合わせる－その１[周期的境界条件]

やっとこの章のはじめに例として挙げた計算を行う準備ができました。

$$n'(i) = n(i-1) + n(i) + n(i+1) \quad \text{[周期的境界条件]}$$

配列を領域分割し双方方向通信を利用して上記の計算を行う例を以下に示します。
ここでは、配列を３つに分割して計算する例について考えることにします。双方向通信のために **mpi_isend**, **mpi_irecv**, **mpi_wait** サブルーチンを組合わせて使うこととします。



各ランクは、“ のりしろ ”データを隣のランクから受信すると同時に、それとは反対側にあるランクへ相手先ランクの“ のりしろ ”データを送信しなければなりません。
例えばランク１では左端側の“ のりしろ ”について、ランク０から n(4)のデータを送信してもらい自身の“ のりしろ ”n(0)へ格納するとともに、ランク２の n(0)の“ のりしろ ”のために、n(4)のデータをランク２へ送信しなければなりません。また右端側の“ のりしろ ”についても同様の操作を行います。

```
      include 'mpif.h'
      parameter(nprocs=3,nmax=12/nprocs)
      integer n(0:nmax+1),na(0:nmax+1)
      integer mstatus(MPI_STATUS_SIZE)

      call mpi_init(ierr)
      call mpi_comm_size(MPI_COMM_WORLD,isize,ierr)
      call mpi_comm_rank(MPI_COMM_WORLD,irank,ierr)
      do i = 1, nmax
         n(i) = irank*nmax + i
      enddo

      iLF = irank-1
      if(irank == 0) iLF=isize-1
      iRT = irank+1
      if(irank == isize-1) iRT=0

      call mpi_isend(n(nmax),1,MPI_INTEGER,iRT,1, &
                     MPI_COMM_WORLD,ireq1,ierr)
      call mpi_irecv(n(0),1,MPI_INTEGER,iLF,1, &
                     MPI_COMM_WORLD,ireq2,ierr)
      call mpi_wait(ireq1,mstatus,ierr)
      call mpi_wait(ireq2,mstatus,ierr)

      call mpi_isend(n(1),1,MPI_INTEGER,iLF,1, &
                     MPI_COMM_WORLD,ireq1,ierr)
      call mpi_irecv(n(nmax+1),1,MPI_INTEGER,iRT,1, &
                    MPI_COMM_WORLD,ireq2,ierr)
      call mpi_wait(ireq1,mstatus,ierr)
      call mpi_wait(ireq2,mstatus,ierr)

      do i = 1, nmax
         na(i) = n(i-1) + n(i) + n(i+1)
      enddo

      print *, 'rank[',irank,'] : ',(na(i),i=1,nmax)
      call mpi_finalize(ierr)
      end
```

MPI 初期化

配列ｎに初期値代入

左端“のりしろ”のランク番号を計算

右端“のりしろ”のランク番号を計算

左端“のりしろ”へのりしろデータを送信

左端“のりしろ”用のデータを受信

右端“のりしろ”へのりしろデータを送信

右端“のりしろ”用のデータを受信

主計算

各ランクの“のりしろ”データの送信先と受信元となるランクの対応表を以下に示します。

	左端“のりしろ”　（上図実線矢印－＞の方向）	
	相手先の左端のりしろとなるデータ送信	自分の左端のりしろデータを受信
ランク０	ランク１へ n(4)のデータ送信	ランク２から n(0)用のデータを受信
ランク１	ランク２へ n(4)のデータ送信	ランク０から n(0)用のデータを受信
ランク２	ランク０へ n(4)のデータ送信	ランク１から n(0)用のデータを受信

	右端“のりしろ”　（上図点線矢印…＞の方向）	
	自分の右端のりしろデータ受信	相手先の右端のりしろとなるデータ送信
ランク０	ランク１から n(5)用のデータを受信	ランク２へ n(1)のデータを送信
ランク１	ランク２から n(5)用のデータを受信	ランク０へ n(1)のデータを送信
ランク２	ランク０から n(5)用のデータを受信	ランク１へ n(1)のデータを送信

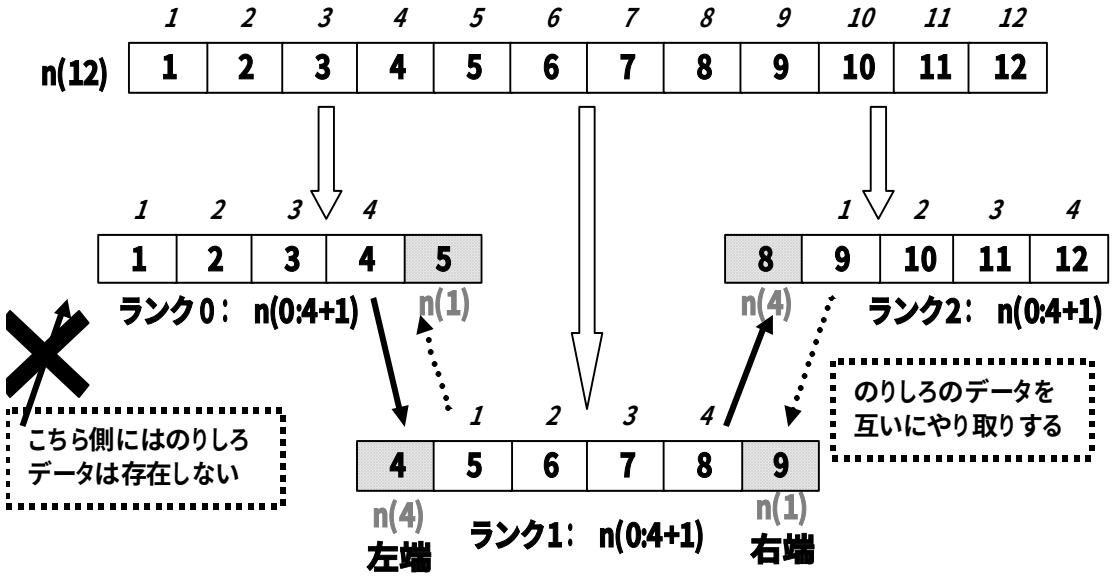
この例では、片端の“のりしろ”データの送受信のために４つのサブルーチンを呼び出しましたが、次の例では一回で送受信できるサブルーチンを紹介します。

◆ 配列を領域分割して足し合わせる－その２[固定境界条件]

前の例では **mpi_isend**, **mpi_irecv**, **mpi_wait** サブルーチンを組合わせたお手軽な双方向通信の方法を利用して計算を行いましたが、ここでは双方向通信専用ともいえる **mpi_sendrecv** サブルーチンを用いた例について説明します。
また前の例では、すべてのランクが右側と左側の両方に“のりしろ”を持っていましたが、境界条件を周期的ではなく固定境界条件にした場合など、問題の種類によっては境界部分にあたるランクは片側に“のりしろ”を持たない場合があります。この場合、境界条件にあたるランクは“のりしろ”データの送受信を行わないようにしなければなりません。

$$n'(i) = n(i-1) + n(i) + n(i+1) \quad \text{[固定境界条件]} \qquad \cdots式③$$

このような場合、ランクごとに送受信そのものの処理を分けなくてもよい方法があります。
『送り先、あるいは送り元のランク(プロセス)がない場合には、送受信を行うな』
という指定を“のりしろ”のない方にすることによって、そちら側の送受信だけを行わないようにすることができます。“**MPI_PROC_NULL**”という値(インクルードファイル mpif.h 中で定義されています)を相手先ランク番号として MPI サブルーチンに渡すと、そのランクへの送受信は行われません。



上の例ではランク１には両端に“ のりしろ ”データが存在しますが、ランク０とランク２には“ のりしろ ”データは、片端にしか存在しません。
“ のりしろ ”データの存在しない方に“ **MPI_PROC_NULL** ”という値を代入することにより、その方向へのデータの送受信は行なわれなくなります。

```
      include 'mpif.h'
      parameter(nprocs=3)
      parameter(nmax=12/nprocs)
      integer n(0:nmax+1),na(0:nmax+1)
      integer mstatus(MPI_STATUS_SIZE)

      call mpi_init(ierr)
      call mpi_comm_rank(MPI_COMM_WORLD,irank,ierr)
      call mpi_comm_size(MPI_COMM_WORLD,isize,ierr)
      if(isize .ne. nprocs) then
         stop
      endif
      do i = 1,nmax
         n(i) = irank*nmax + i
      enddo
①
      iLF = irank-1
②     if(irank == 0) iLF=MPI_PROC_NULL
      iRT = irank+1
      if(irank == isize -1) iRT=MPI_PROC_NULL
③
      call mpi_sendrecv( &
            n(nmax),1,MPI_INTEGER,iRT,1, &
            n(0)   ,1,MPI_INTEGER,iLF,1, &
            MPI_COMM_WORLD,mstatus,ierr)

④     call mpi_sendrecv( &
            n(1)    ,1,MPI_INTEGER,iLF,1,&
            n(nmax+1),1,MPI_INTEGER,iRT,1,&
            MPI_COMM_WORLD,mstatus,ierr)

      do i = 1, nmax
         na(i) = n(i-1) + n(i) + n(i+1)
      enddo

      print *, 'rank[',irank,'] : ',(na(i),i=1,nmax)
      call mpi_finalize(ierr)
      end
```

MPI 初期化

ランク０では、左端に“ のりしろ ”は存在しない

ランク２では、右端に“ のりしろ ”は存在しない

左端“ のりしろ ”データを送受信
iRT：送り先ランク
iLF：このランクから受取

右端“ のりしろ ”データを送受信
iLF：送り先ランク
iRT：このランクから受取

主計算

各ランクの“ のりしろ ”データの送信先と受信元となるランクの関係は以下の通りです。

	左端“ のりしろ ”　（上図実線矢印－＞の方向）	
	相手先の左端のりしろとなるデータ送信	自分の左端のりしろデータを受信
ランク０	ランク１へ n(4)のデータ送信	×
ランク１	ランク２へ n(4)のデータ送信	ランク０から n(0)用のデータを受信
ランク２	×	ランク１から n(0)用のデータを受信

	右端“ のりしろ ”　（上図点線矢印…＞の方向）	
	自分の右端のりしろデータ受信	相手先の右端のりしろとなるデータ送信
ランク０	ランク１から n(5)用のデータを受信	×
ランク１	ランク２から n(5)用のデータを受信	ランク０へ n(1)のデータを送信
ランク２	×	ランク１へ n(1)のデータを送信

①：ランク０では左端に“ のりしろ ”データが存在しませんので、左のランク(iLF)からのデータ受信(③で実行)を行わないようにするため、iLF に MPI_PROC_NULL をセットします。また、左にランクが存在しませんので、左のランクの右端の“ のりしろ ”へのデータ送信(④で実行)を行わないようにしなければなりません。

②：ランク２では右端に“ のりしろ ”データが存在しませんので、右のランク(iRT)からのデータ受信(③で実行)を行なわないようにするため、iRT に MPI_PROC_NULL をセットします。また、右にランクが存在しませんので、右のランクの左端の“ のりしろ ”へのデータ送信(④で実行)を行わないようにしなければなりません。

補足 ここまで一対一通信、双方向通信の例を挙げてきましたが、同じような働きのサブルーチンがいくつか出てきました。

mpi_send / mpi_recv
mpi_isend / mpi_irecv / mpi_wait
mpi_sendrecv

はじめて MPI プログラミングをされる方は、いったいどれを使ったらいいのか悩まれるかもしれません。少し考えてノンブロッキング通信である mpi_isend/irecv を使うのがよさそうに思えるかもしれません。例えるならば mpi_send/recv はトランシーバーのように片方づつしか話ができないのに対して、mpi_isend/irecv の方は電話のように同時に双方向で話ができて効率がよく、プログラムも書き易そうに思えるからです。

しかしながら、現時点ではこれは正しいとは言えません。現状多くの MPI サブルーチンは MPICH という実装を元に作成されています。Compaq MPI もそうです。この MPICH では、効率よくノンブロッキング通信をするようにはプログラミングされていません。そのため、大量のデータを送受信する時には mpi_send/recv や、mpi_sendrecv のブロッキング通信を使う方が少し高速なプログラムになります。

例えば、実のところすべての集団通信は一対一通信の組合わせにより実現できます。にもかかわらず MPI に多くのサブルーチンが用意されているのは、MPI ライブラリを実装する側に最適化の自由度を与えるためです。低機能のサブルーチンを組合わせて使うよりも、高機能のものを使った方が最適化されているか、あるいは将来最適化されるかもしれません。Compaq MPI には常に改良が加えられています。

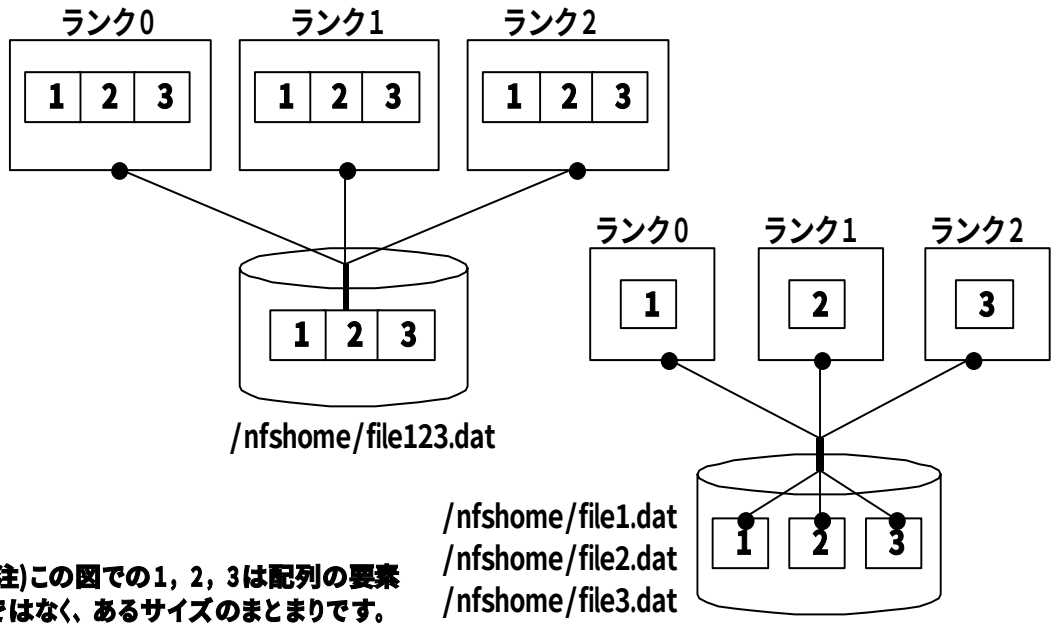
３.５. ファイルの入出力

この節では、並列プログラムにおけるデータファイルの入出力について説明します。
どのようにファイルを入出力すればよいかは、並列化の目的により異なります。
例えば、データはそれ程大きくはないけれども計算量そのものが多いために並列化するのか、
データが大きくそれを分割して並列化するのかによって、どうデータファイルを入出力するかが異なってきます。この節では、Beowulf 型＋NFS システムでの３つの代表的な例を示します。

・共有ディスクを利用してファイルを入出力する
・あるランクが代表してファイルを入出力する
・各ランクがローカルディスクからそれぞれのファイルを入出力する

◆ すべてのランクがそれぞれ共有ディスクからファイルを入出力する

小規模な並列システムや、入出力するファイルサイズが小さい場合には、共有ディスクからファイルを入出力するのに利用します。



●すべてのランクが同じファイルを入力する場合：(上図左側の例)

計算の初期データファイルを読込む場合など、主にサイズの小さなファイルの入力に使います。ただし、読込みデータに対して各ランクが計算処理を行った結果をそのまま共有ディスクへ書込んではいけません。そのまま書込むと他のランクの書込んだ結果を破壊してしまいます。各ランクの担当した部分のデータを集めて一つのファイルとして書込むためには、あるランクに代表させてデータの収集を行わなければなりません。それには、次項「ランク０が代表してファイルを入出力する」の例を参照してください。

```
include 'mpif.h'
integer data(12)

call mpi_init(ierr)

open(8,FILE='/nfshome/file123.dat',FORM='unformatted')
read(8) data
close(8)

call mpi_barrier(MPI_COMM_WORLD,ierr)

  次の処理

call mpi_finalize(ierr)
end
```

次の処理のために待ち合わせが必要な場合、mpi_barrier を使う

※例えばファイルを読込んだ後、次の処理のために同期をとらなければならない場合には、上の例のように、**mpi_barrier** サブルーチン(4.8 節を参照ください)を使用します。

●各ランクがそれぞれのファイルを入力する場合：(前ページ右側の例)

これはデータサイズの小さなファイルの入出力に使います。初期条件ファイルなどを読込む場合には適していますが、大きなデータサイズのファイルの入出力に利用してはいけません。特に大きなファイルの書込み時には、トラブルの元になりかねません。また、はじめにそれぞれのランク用にデータを分割してファイルに格納しておかなければなりません。

```
include 'mpif.h'
integer data(12)

call mpi_init(ierr)
call mpi_comm_size(MPI_COMM_WORLD,irank,ierr)

if(irank == 0) then
   open(8,FILE='/nfshome/file1.dat',FORM='unformatted')
elseif(irank == 1) then
   open(8,FILE='/nfshome/file2.dat',FORM='unformatted')
elseif(irank == 2) then
   open(8,FILE='/nfshome/file3.dat',FORM='unformatted')
endif
read(8) data
close(8)

call mpi_finalize(ierr)
end
```

各ランクごとに読込むファイル名を指定する

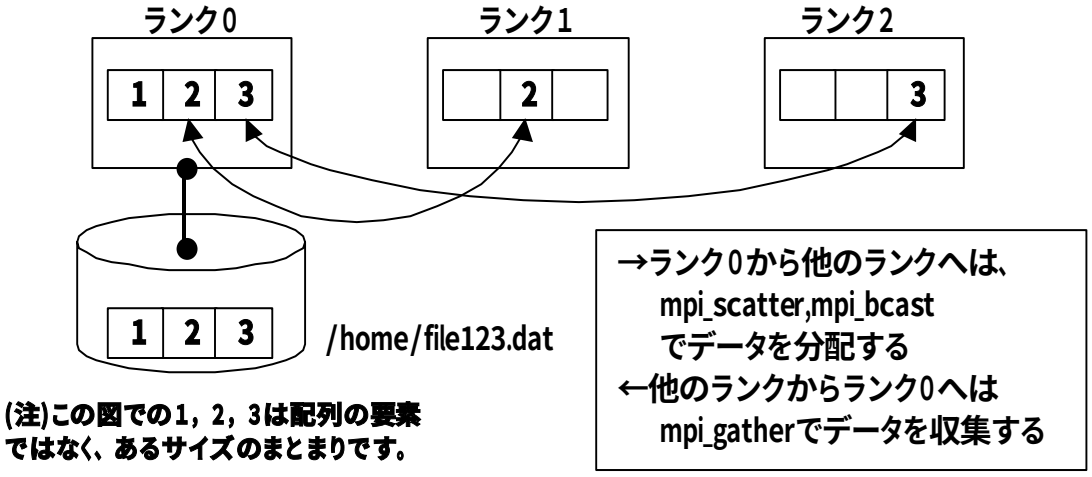
注意 ランク(CPU)の数が増えたり、大規模なデータの入出力を行うと資源(ファイル、ネットワーク)の競合が起こります。資源の競合が起こるとファイルの入出力にかかる時間が非常に長くなります。

↓

小規模な並列システム、あるいはファイルサイズが小さい初期化ファイルを読込む場合などには有効ですが、それ以外の場合はトラブル発生の元になりかねません。

◆ ランク０が代表してファイルを入出力する

中規模程度ぐらいまでのデータサイズのファイルを入出力する場合に利用します。特に各ランク間が高速なネットワークで接続されている場合は、各ランクへのデータの転送が高速に行えます。




```
      include 'mpif.h'
      integer data(4),dbuf(12)

      call mpi_init(ierr)
      call mpi_comm_size(MPI_COMM_WORLD,isize,ierr)
      call mpi_comm_rank(MPI_COMM_WORLD,irank,ierr)

      if(irank == 0) then
         open(8,FILE='file123.dat',FORM='unformatted')
         read(8) dbuf
         close(8)
      endif

      call mpi_scatter(dbuf,4,MPI_INTEGER, &
①                      data,4,MPI_INTEGER, &
                       0,MPI_COMM_WORLD,ierr)

      do i = 1,4
         data(i) = data(i) + irank
      enddo

      call mpi_gather(data,4,MPI_INTEGER,
                       dbuf(irank*4+1),4,MPI_INTEGER,
                       0,MPI_COMM_WORLD,ierr)

      if(irank == 0) then
         print '(I2,4X,12I3)',irank,(dbuf(i),i=1,12)
      endif
      call mpi_finalize(ierr)
      end
```

ランク０が代表してファイルを読込む

各ランクが計算を担当する部分をそれぞれ分配する

計算処理

各ランクが担当していた部分を収集して配列 dbuf(...)に格納する

①：**mpi_scatter** では各ランクにそれぞれが計算を担当する部分のデータを分配することができます。初期値などすべてのランクに同じデータを送る場合など、データサイズが大きくなくパフォーマンスにそれ程大きな影響を与えないような場合には **mpi_bcast** というサブルーチンを利用して各ランクに同じデータを送信するということも可能です。

上記プログラムの①の部分を **mpi_bcast** に変更すると次のようになります。

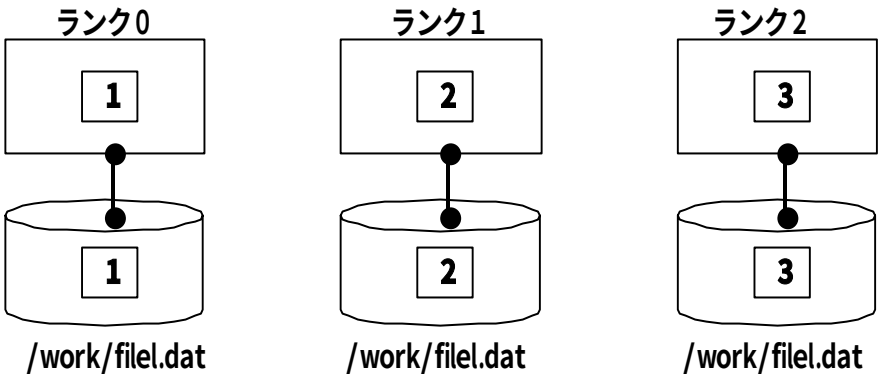
```
    ～

if(irank == 0) then
    open(8,FILE='file1.dat',FORM='unformatted')
    read(8) data
    close(8)
endif

call mpi_bcast(data,12,MPI_INTEGER,
                0,MPI_COMM_WORLD,ierr)
    ～
```

ランク０が代表して
ファイルを読込む

配列 data を他のすべ
てのランクにばらまく

◆ 各ランクがローカルディスクからそれぞれファイルを入出力する

配列を領域分割して計算を行う場合など、各ランクがそれぞれの担当するデータを処理する場合は、それぞれがローカルディスクから入出力する方がパフォーマンスが向上します。
(ただし、AlphaServerSC シリーズの pfs ファイルシステムなど、大規模並列システム用のファイルシステムが用意されている場合は、この限りではありません)
またローカルディスクの領域が一時的なものである場合は、一時保管の期限が切れる前後でホームディレクトリからファイルの転送処理を行わなければなりません。



(注)この図での1，2，3は配列の要素ではなく、あるサイズのまとまりです。

1．ホームディレクトリからファイルを並列マシンの各ノードへ転送します。
例えば rcp コマンドを利用したばらまき用のシェルプログラム(nodecpto.sh)を作成します。/work ディレクトリは、ローカルディスク上のディレクトリとします。

```
alpha> cat nodecpto.sh
rcp file_1  node01:/work/filel.dat
rcp file_2  node02:/work/filel.dat
rcp file_3  node03:/work/filel.dat

alpha>  nodecpto.sh
```

2．MPI プログラムを実行します。
ファイルを読込み計算を行った後、データをローカルのファイルに書込みます。

```
include 'mpif.h'
integer data(4)
call mpi_init(ierr)

open(8, FILE='/work/filel.dat')
read(8) data

    ～
  計算部分
    ～

write(8) data
close(8)
call mpi_finalize(ierr)
end
```

3．ファイルの保管期限が切れる前に、並列マシンの各ノードからホームディレクトリへファイルを転送します。
例えば１とは逆の収集用のシェルプログラム(nodefrom.sh)を作成します。

```
alpha> cat nodefrom.sh
rcp node01:/work/filel.dat file_1
rcp node02:/work/filel.dat file_2
rcp node03:/work/filel.dat file_3

alpha>  nodefrom.sh
```

補足 **pfs** 等のファイルシステムの利用方法などについては、本書の説明の範囲を超えていますのでここでの説明は省略します。それぞれの利用の手引き等をご参照ください

ちょっと寄り道

章の最後に少し寄り道をすることにします。本章の MPI プログラミングの話からはずれてしまいますので、ここを飛ばして４章へ進んでもかまいません。

上のプログラム例では、ファイルから読込まれた配列 data(4)のインデックスはすべてのランクで、１～４と共通の値です。Fortran90 の新機能であるメモリの動的割当て(C 言語での malloc に相当)を利用すると、配列のインデックスを連続させることができ、配列を縮小した場合のインデックスの取扱いが比較的容易になります。

```
      include 'mpif.h'
      integer, ALLOCATABLE :: N(:)

      call mpi_init(ierr)
      call mpi_comm_rank(MPI_COMM_WORLD,irank,ierr)
①     ista = irank*4+1
      iend = irank*4+4

      ALLOCATE( N(ista:iend) )
②     If(irank == 0) then
         open(8, FILE='/work/file1.dat')
      else if(irank == 1) then
         open(8, FILE='/work/file2.dat')
      end if
      read(8) N
      close(8)
      do i = ista, iend
         N(i) = N(i) + i
      enddo
      DEALLOCATE(N)
      call mpi_finalize(ierr)
      end
```

動的割当て用の配列を定義する

各ランクが担当する配列の ista～iend の値を設定する

ファイルの読込み

①:ここでは配列のインデックスの値(ista,iend)を配列の全体のサイズと分割数から計算していますが、ファイルの中に事前に格納してある値を読込んでも構いません。

②：上の例では２つに分割していますが、例えば、配列の全体のサイズが N(12)で３つに分割した場合、各ランクでは次のように配列が割当てられます。

ランク０	ランク１	ランク２
N(1:4)	N(5:8)	N(9:12)

これでこの章はおしまいです。すぐにでも MPI プログラミングを始めたい方は、この章で説明した MPI サブルーチンをもとに、MPI プログラミングをはじめることができます。ただ、パフォーマンスのよい MPI プログラムを作成するためにはかなり注意が必要ですし、MPI には他に知っているともう少し楽に並列化プログラミングできるためのサブルーチンがいくつかあります。次章ではそのようなサブルーチンの紹介と共に、どう並列化するかについて説明していきます。

４．MPI プログラミング～次の一歩

前章までで、並列化プログラミングをはじめるための準備はできました。しかしながら、実際に並列化を行うことは、それ程やさしいことではありません。並列化ができたとしても、並列化によってパフォーマンスが向上しなければ並列化の意義はありません。うまくプログラミングされていない並列化プログラムでは、CPU の数を増やしてもある程度のところでパフォーマンスの向上が見られなくなります。並列化の方針については章の最後で少し詳しく説明しますが、並列化によってパフォーマンスを向上させるためには、次の３点に注意しなければなりません。

１．　並列化可能な部分をなるべく大きくする

例えば、差分法で使用される SOR 法はそのままでは並列化できません。
これをレッドブラック SOR 法という数値計算方法に変更すると並列化が可能になります。このように計算方法そのものを変更することが有効なことが多くあります。

２．　プロセス間の通信コストをなるべく減少させる

計算のコストに比較すると通信のコストは非常に高くつきます。
なるべく通信の回数を少なくし、そして通信量自体も少なくする工夫が必要です。
→このためには、領域分割の方法を工夫することなどが有効となります。

３．　プロセス間のロードバランスを均等にする

領域の分割の仕方と計算方法によっては、ある領域部分の計算だけが早く終わってしまい、CPU が遊んでしまうことがあります。分割の仕方を工夫してロードバランスを均等にすることが重要です。

並列計算向きの数値計算方法にすることも重要ですが、どう領域分割するかも同じくらいに重要となります。この章では、領域分割の仕方を中心により良い並列化の仕方について説明します。ただし、本章ではどういう分割方法があるかの紹介が目的ですので、問題に即したハイパフォーマンスな分割方法がどういうものかについては、第一義には考慮しないこととします。

４.１. データ領域を分割する

通信コストの削減とロードバランスの均等のためには、領域分割の仕方が重要となります。数値計算の解法とそれに適した領域分割を行うことによってプログラムのパフォーマンスは向上します。ここでは、２次元配列を例として次の代表的な３つの分割方法を示します。

１．ブロック分割
２．サイクリック分割
３．ブロック・サイクリック分割

◆ 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い、よく使われます。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ありません。
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、数値計算の並列化によって発生する通信コストを少なくするように領域を分割しなければなりません。

A. 列で分割
(列をプロセス数分に
分けて分割する)

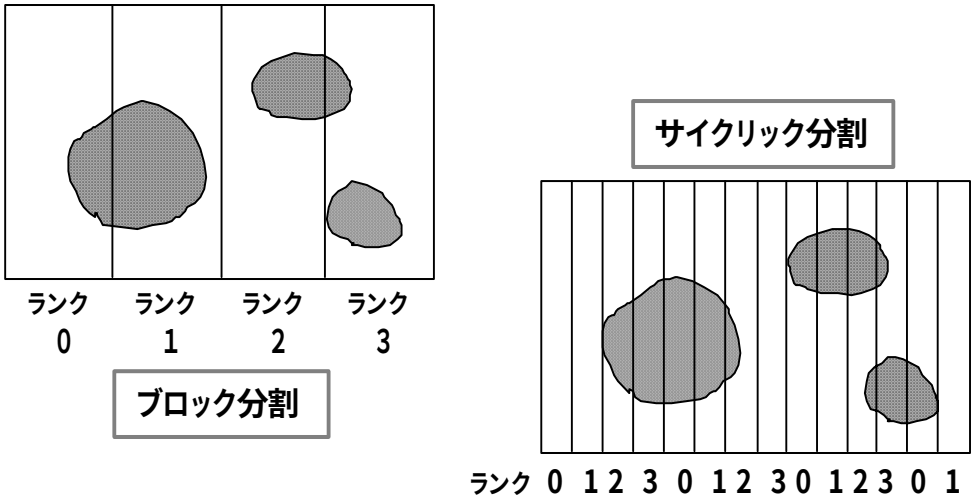
B. 行で分割
(行をプロセス数分に
分けて分割する)

C. 行と列の両方で分割
(行と列をプロセス数分に
分けて分割する)

補足 例えば、A(m,n)という２次元配列について何らかの行列計算をするとします。Fortranでは、行のインデックスmが大きくなる方向へメモリ領域が連続的に取られますので、一見、Aの列で分割するのがよいかと思われます。これは確かに正解である場合も多いのですが、配列のサイズによっては常に正しいとは言えなくなります。例えば、差分法など上下左右の４方向のデータが必要になる場合、列で分割すると、行数が大きくなった場合キャッシュのヒットミスが多発し､性能が低下してしまうことがあります。このような場合行と列の両方に分割すると、多くの場合性能が向上します。配列をどう分割するかは、配列のサイズやどういう計算を行うかによって決まり、残念ながらこれが唯一の正解というという決まり手はありません。

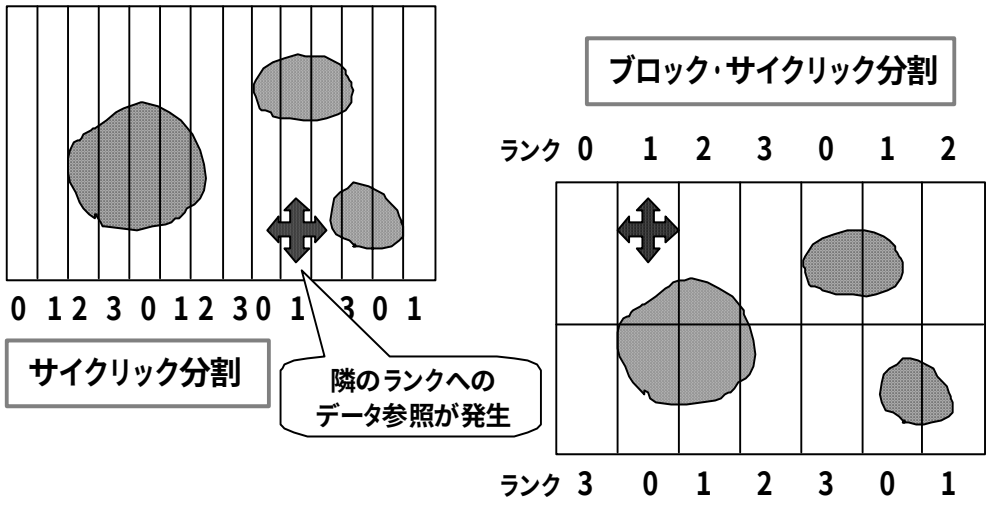
◆ **サイクリック分割**

計算領域を N 個にブロック分割した場合、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ります。ところが２次元空間中に粒子をバラまき、ある方程式の元でその粒子の運動を計算する場合、２次元空間を均等にブロック分割させて計算を行うと、粒子がある部分領域に集まってきた時、領域によっては粒子の数に著しい差が現れロードバランスが不均等になってしまいます。
このような場合、サイクリック分割にするとロードバランスはほぼ均等になります。



◆ **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コストが上昇し、逆にパフォーマンスの低下をまねく場合があります。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・サイクリック分割にします。

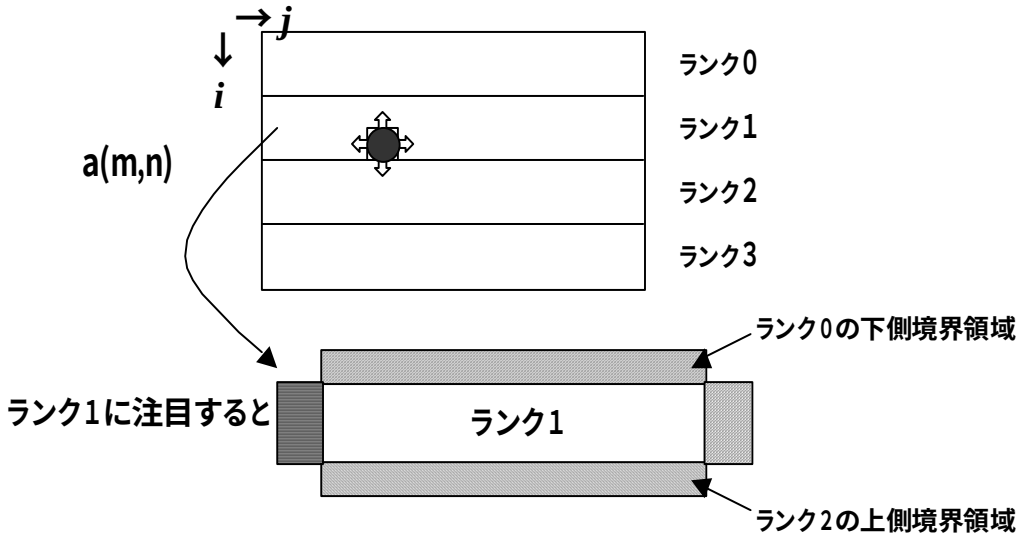


4.2. ブロック分割－行分割による差分法の計算－その準備

３章では１次元の差分法モドキを考えました。本章では、２次元配列をブロック分割して次のような差分法モドキの例を考えることにします。

$$n'(i, j) = n(i-1, j) + n(i, j-1) + n(i, j+1) + n(i+1, j) \qquad \cdots式④$$

２次元配列を行で分割することにします。



ここでランク１に注目します。ランク１において式④の計算を行うためには、ランク０とランク２の境界部分のデータが必要になります(上図参照)。
この境界部分のデータは通信によって転送して貰わなければなりませんが、例えば境界が、

a(5,1), a(5,2), a(5,3),…, a(5,n-1), a(5,n)

の部分だとすると、これらはメモリ上に連続して取られていません。このような場合、これらのデータ列を **mpi_send/mpi_recv** のサブルーチンを利用して一度に転送することはできません。一度に転送するためには、利用者自ら１次元の転送用の配列を用意し、一旦その配列に転送用データをコピーしてから転送するという方法もありますが、このようなデータ転送のために MPI には派生データタイプというサブルーチンが用意されています。

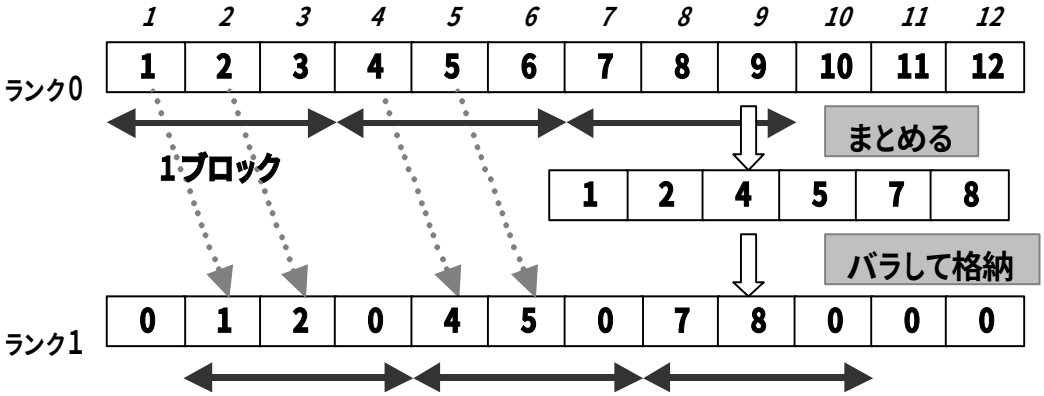
補足	この差分法モドキでは４方向の近接する格子点のデータが必要になります。メモリアクセスへのストライド(メモリ上でのデータアクセスの間隔)が小さい程キャッシュヒットミスが少なくなることから考えると、行で分割した場合、左右のデータへのアクセス時のストライドが小さくなるため、ある程度のパフォーマンスの良さが期待されます。ただし、以降で述べるように“のりしろ”データの転送のために派生データタイプを用いた場合、派生データタイプ生成のためのオーバーヘッドが加わりますので、そのオーバーヘッドとキャッシュヒットミスのトレードオフによりパフォーマンスが変化します。これは配列のサイズに依存するので、一概にどう分割したらよいという決まり手はありません。

４.３. 派生データタイプ

メモリ上で連続していない境界部分のデータなど、とびとびのデータをやり取りしたい場合、何度も送受信を繰り返すと通信コストが高くなりパフォーマンスが低下してしまいます。このような場合、任意形状のデータセットを集めて新しいタイプのデータセットにまとめ、一度に送受信できるようにするための機能があります。それを派生データタイプといいます。

◆ mpi_type_vector を使って派生データタイプを作成する

４.２節の２次元配列の境界部分に対して派生データタイプの機能を適用してみる前に、以下のような一次元配列中のとびとびのデータを送信する簡単な例を考えることにします。
1 と 2、4 と 5、7 と 8 でそれぞれ mpi_send/recv を使って送信すると 3 回の通信が発生することになりますが、mpi_type_vector というサブルーチンを利用すると一回の通信でランク０からランク１へデータを送信することができます。




```
include 'mpif.h'
integer mstatus(MPI_STATUS_SIZE)
integer n(12)

call mpi_init(ierr)
call mpi_comm_size(MPI_COMM_WORLD,isize,ierr)
call mpi_comm_rank(MPI_COMM_WORLD,irank,ierr)
if(irank == 0) then
   do i = 1,12
      n(i) = i
   enddo
endif

call mpi_type_vector(3,2,3,MPI_INTEGER,INEWVEC,ierr)
call mpi_type_commit(INEWVEC,ierr)

if(irank == 0) then
   call mpi_send(n(1),1,INEWVEC,1,1,MPI_COMM_WORLD,ierr)
elseif(irank == 1) then
   call mpi_recv(n(2),1,INEWVEC,0,1,MPI_COMM_WORLD,mstatus,ierr)
endif
```

・全ブロック数は３つ
・一つのブロックに含まれる
　要素数は、２つ
・各ブロックのストライド(間隔)
　は、MPI_INTEGER３要素分

mpi_type_commit を必ずコールすること！

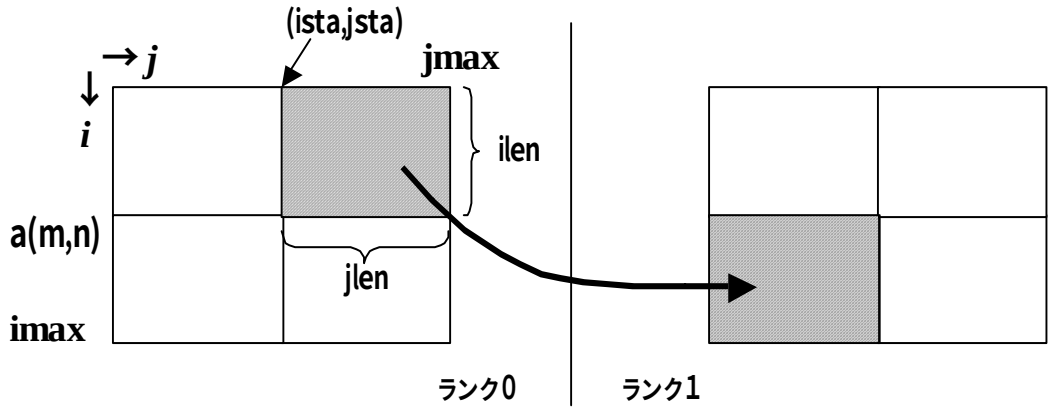
```
      print '(A,I2,2X,12I3)','rank:',irank,(n(i),i=1,12)
      call mpi_type_free(INEWVEC,ierr)
      call mpi_finalize(ierr)
      end
```

注意 **mpi_type_vector** でデータタイプを宣言しただけではいけません。**mpi_type_commit** をコールしてはじめて新しいデータタイプの登録が完了します。登録が完了した後でなければ実際に利用することはできません。

mpi_type_vector (count, blocklength, stride, oldtype, newtype, ierr)			
機能	一定間隔におかれたブロックをまとめて新しいデータタイプを作成する		
引数	型	入出力	
count	Integer	入力	ブロック数
blocklength	Integer	入力	各ブロックに含まれるの要素数(oldtype の数分)
stride	Integer	入力	各ブロック間の oldtype の数でのストライド (１ブロックの oldtype でのサイズ)
oldtype	Integer	入力	古いデータタイプ
newtype	Integer	出力	作成された新しいデータタイプ
ierr	Integer	出力	終了コード

mpi_type_commit (dataype, ierr)			
機能	新しいデータタイプを登録する		
引数	型	入出力	
datatype	Integer	入力	登録したい新しいデータタイプ (mpi_type_vector 等で作成したデータタイプ)
ierr	Integer	出力	終了コード

次に、行列中のある部分領域を転送する例を示します。



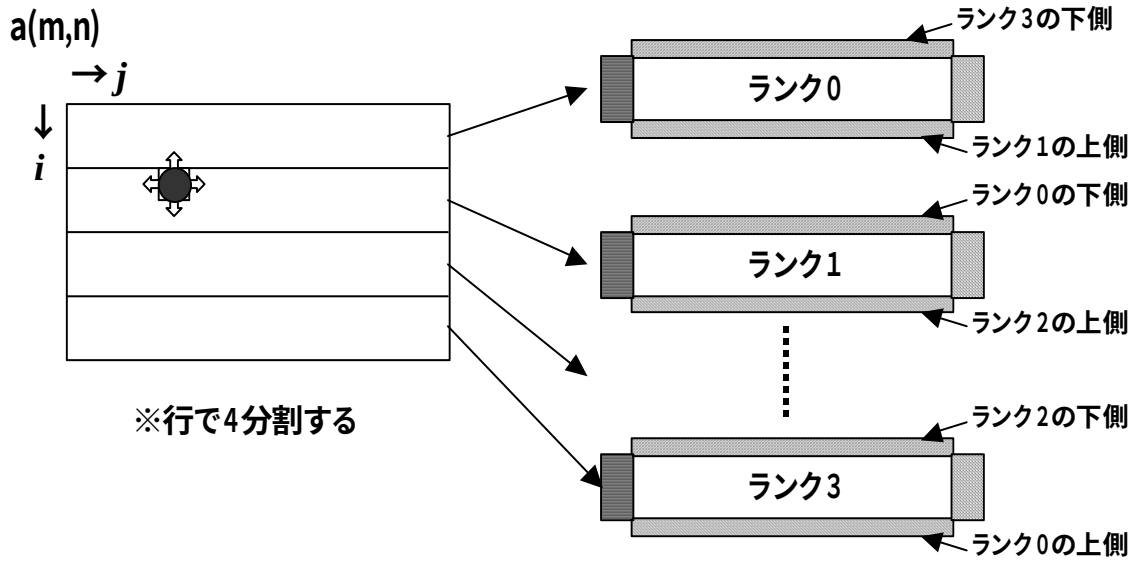
```
      include 'mpif.h'
      parameter(imax=8, jmax=8)
      integer mstatus(MPI_STATUS_SIZE)
      integer n(imax,jmax)

      call mpi_init(ierr)
      call mpi_comm_rank(MPI_COMM_WORLD,irank,ierr)

      ilen = 4
      jlen = 4
      call mpi_type_vector(jlen,ilen,imax,MPI_INTEGER,ISUBMTX,ierr)
      call mpi_type_commit(ISUBMTX,ierr)

      if(irank == 0) then
         call mpi_send(n(1,5),1,ISUBMTX,1,1,MPI_COMM_WORLD,ierr)
      elseif(irank == 1) then
         call mpi_recv(n(5,1),1,ISUBMTX,0,1,MPI_COMM_WORLD,mstatus,ierr)
      endif

      print '(A,I2)','rank:',irank
      print '(8I3)',((n(i,j),j=1,8),i=1,8)
      call mpi_type_free(ISUBMTX,ierr)
      call mpi_finalize(ierr)
      end
```

・全ブロック数は **4**(=jlen)
・一ブロックに含まれる要素数は、**4**(=jlen)
・ブロックのサイズは、MPI_INTEGER**8** 要素分

(注)配列 n には、適当な値を入れておく必要があります。

４.４. 行分割により差分法モドキを計算する

４.２節で示した式④を計算するための準備が整いました。
行列を４つに行分割して計算する例を以下に示します。



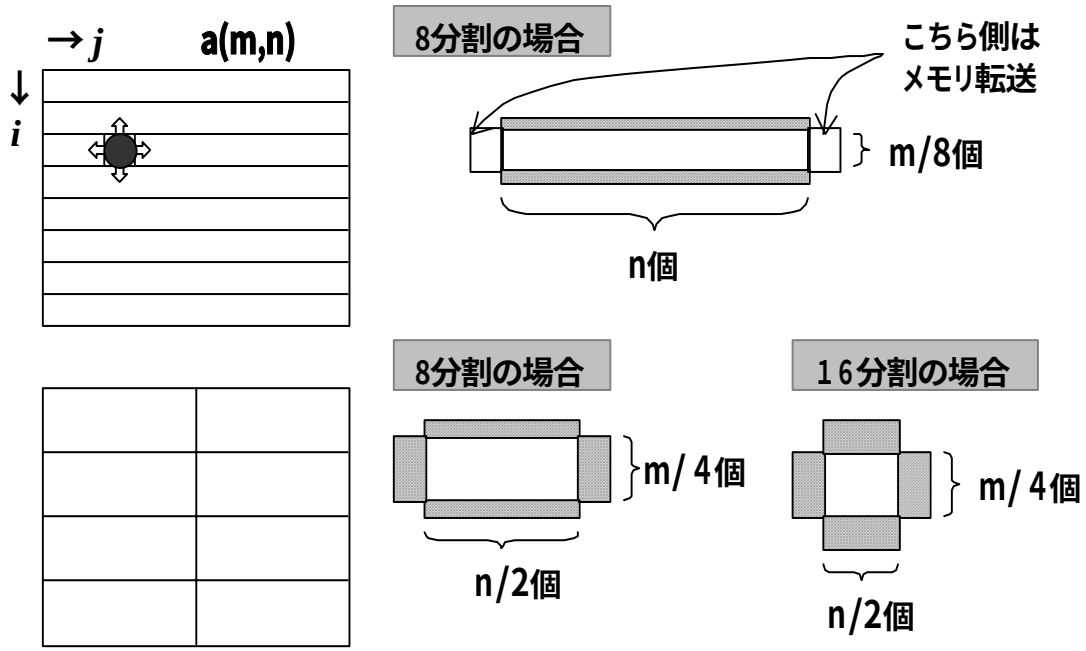
```
      include 'mpif.h'
      parameter(MMAX=16, NMAX=16, MM=MMAX/4)
      integer mstatus(MPI_STATUS_SIZE)
      integer n(0:MM+1,0:NMAX+1),m(0:MM+1,0:NMAX+1)

      call mpi_init(ierr)
      call mpi_comm_size(MPI_COMM_WORLD,isize,ierr)
      call mpi_comm_rank(MPI_COMM_WORLD,irank,ierr)
      do j = 1,NMAX
         do i = 1,MM
            n(i,j) = i + j*100
         enddo
      enddo

      call mpi_type_vector(NMAX,1,MM+2,MPI_INTEGER,INEWVEC,ierr)
      call mpi_type_commit(INEWVEC,ierr)

      iUp = irank-1
      if(irank == 0) iUp = 3
      iDw = irank+1
      if(irank == isize -1) iDw = 0
      call mpi_sendrecv(n(MM,1),1,INEWVEC,iUp,1,
                        n(0,1) ,1,INEWVEC,iDw,1,
                        MPI_COMM_WORLD,mstatus,ierr)

      call mpi_sendrecv(n(1,1)   ,1,INEWVEC,iUp,1,
                        n(MM+1,1),1,INEWVEC,iDw,1,
                        MPI_COMM_WORLD,mstatus,ierr)
      do i = 1,MM
         n(i,0)   = n(i,NMAX)
         n(i,NMAX+1) = n(i,1)
      enddo

      do j = 1,NMAX
         do i = 1,MM
            m(i,j) = n(i-1,j)+n(i,j-1)+n(i,j+1)+n(i+1,j)
         enddo
      enddo

      print '(A,I2)','rank:',irank
      print '(16I5)',((m(i,j),j=1,NMAX),i=1,MM)
      call mpi_type_free(INEWVEC,ierr)
      call mpi_finalize(ierr)
      end
```

・全ブロック数は、**NMAX**
・一ブロックに含まれる要素数は、**1**
・ブロックのサイズは、MPI_INTEGER **MM+2** 要素分(4+2=6 要素)

補足 ここまで何の説明もなく **mpi_type_free** というサブルーチンを使ってきましたが、作作成した派生データタイプはプログラミングの作法として **mpi_type_free** で解放しなければなりません。**mpi_type_commit** により作成した型は、**free** するまで何度でも使うことができます。

４.５．ブロック分割－行と列両方での分割による差分法の計算－その準備

４.４節では行分割によるブロック分割の例を示しました。配列のサイズがそれ程大きくないうちは、行あるいは列での１次元的な分割でもパフォーマンス的にはそれ程大きな問題とはならないかもしれません。しかし、配列のサイズが非常に大きくなった場合、問題の種類にもよりますが、行あるいは列分割のような１次元的な分割をするよりは、行と列の両方での２次元的な分割を行った方が通信コストが低くなりパフォーマンスが向上します。



上図を例に“のりしろ”の境界データの転送量がどれくらいか見積もりしてみます。

[行分割の場合]

８CPU　→　2n * 8　= 16n

16CPU　→　2n * 16 = 32n

[行と列両方での分割の場合]

８CPU　→　(n/2*2 + m/4*2)* 8　= 8n+4m　－(正方行列なら)→ 12n

16CPU　→　(n/4*2 + m/4*2)*16　= 8n+8m　－(正方行列なら)→ 16n

正方行列であるとするなら、行分割の場合と行と列両方での分割を比べると、16CPU の場合、データの転送量は、半分に減少します。

行と列両方で分割したプログラム例へ進む前に、もう一つ MPI の便利なサブルーチンを紹介します。CPU 数 P 個の並列マシンがあるとし、それに合わせてデータ領域を行と列それぞれ M 個と N 個に分割して(P≧M*N)計算させるとします。プログラムの試作時には P_1 個、本番実行時には P_2 個で計算させるなど、多くの場合ある配列サイズの場合は P_1 個、もっと

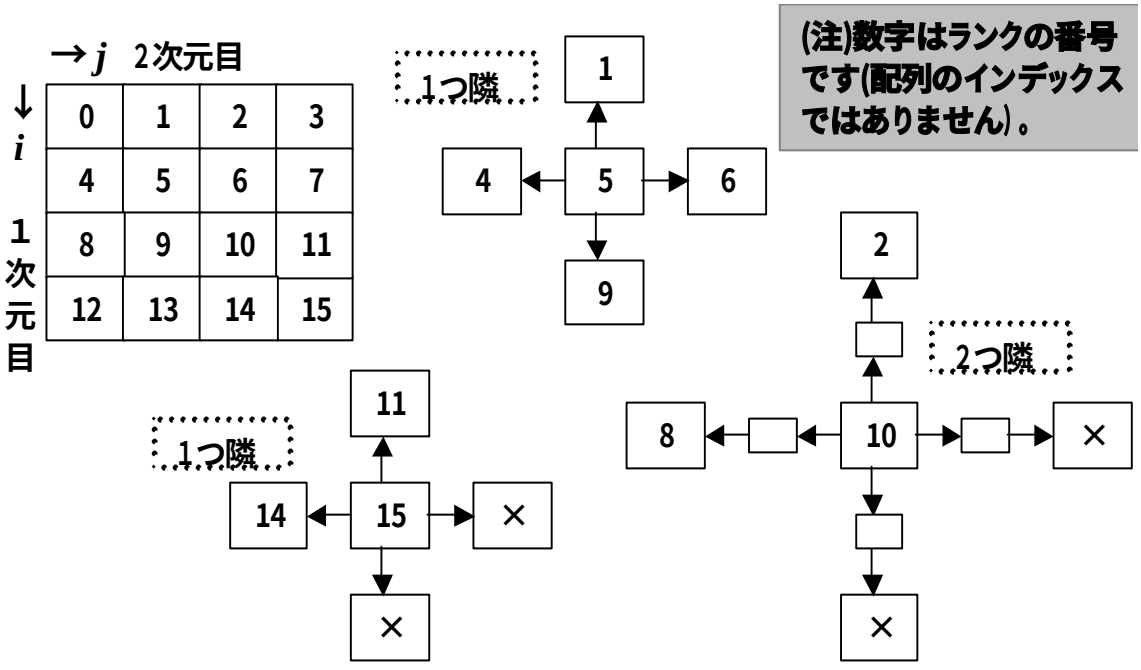
大きいサイズの場合は P_2 個と計算の目的に合わせて CPU 数を増やしていくケースは少なくありません。この時、上下左右の“ のりしろ ”データを転送するために、CPU 数固定の前提条件のもとで、あるランクの上下左右のランク番号を固定的に決めていると、CPU 数を変更した場合にはプログラムに大きな修正を加えなくてはならない場合があります(このようにハードウェアに依存する等、ある条件に強く依存するようなプログラムを作成することを「ハードコーディング」といいます)。

MPI には CPU 数が変更された場合にも対応しやすいように、あるプロセス(ランク)とその周り(上下左右、上下左右前後など)のプロセスのトポロジ(位置関係)を自動的に計算してくれるサブルーチン群があります。これらをプロセス・トポロジサブルーチンといいます。

４.６. プロセス・トポロジ

プロセス・トポロジサブルーチンには、カーテシアン・トポロジとグラフ・トポロジの２つがありますが、本書では、カーテシアン・トポロジのみについて説明します。

それでは、プロセス・トポロジサブルーチンとはどういうものか簡単な例を見てみましょう。ここでは固定周期条件で 2 次元配列を４×４＝１６分割した例を考えることにします。



例えば、ランク５の左右のランクはそれぞれ４と６であり、上下のランクは１と９です。ランク１５の場合、右側と下側のランクは存在しません。**mpi_cart_create**,**mpi_cart_shift** サブルーチンを利用すると、これらの値を計算して返してくれます。

```
      include 'mpif.h'
      logical period(2)
      integer idivid(2),iUD(2),iLR(2),idisp(2),narg,nainfo(3)
      character*16 argv

      call mpi_init(ierr)
      call mpi_comm_rank(MPI_COMM_WORLD,irank,ierr)

      narg = iargc()
      if(narg == 3) then
         do i = 1, narg
            call getarg(i, argv)
            read(argv,'(I)') nainfo(i)
         enddo
      else
         print *,'input error: COMMAND  rank i j'
         stop
      endif

      idivid(1) = 4
      idivid(2) = 4
      period(1) = .FALSE.
      period(2) = .FALSE.
      call mpi_cart_create(MPI_COMM_WORLD,2,idivid,period, &
                           .FALSE.,iptbl,ierr)

      idisp(1) = nainfo(2)
      idisp(2) = nainfo(3)
      call mpi_cart_shift(iptbl,0,idisp(1),iUD(1),iUD(2),ierr)
      call mpi_cart_shift(iptbl,1,idisp(2),iLR(1),iLR(2),ierr)

      if(irank == nainfo(1)) then
         print '(3X,I3)',iUD(1)
         print '(3I3)'  ,iLR(1),irank,iLR(2)
         print '(3X,I3)',iUD(2)
      endif
      call mpi_comm_free(iptbl,ierr)
      call mpi_finalize(ierr)
      end
```

コマンドラインからの入力を読込み、その値を nainfo にセットします。

４×４のプロセス座標テーブルを作成します。境界条件は、固定境界を指定します。

１次元目の方向(i 方向)のランクの値を取出します

２次元目の方向(j 方向)のランクの値を取出します

コマンドラインの引数として、トポロジを「表示させたいランクの番号」と、そのランクからの「i 方向と j 方向への変位」の 3 つを入力します。

```
alpha>  dmpirun –np 16 a.out 5 1 1
     1
  4  5  6
     9
alpha>  dmpirun –np 16 a.out 15 1 1
    11
 14 15  -1
    -1
alpha>  dmpirun –np 16 a.out 10 2 2
     2
 8 10 -1
    -1
alpha>
```

my rank = 5
i 方向への変位=1
j 方向への変位=1

my rank = 15
i 方向への変位=1
j 方向への変位=1

my rank = 10
i 方向への変位=2
j 方向への変位=2

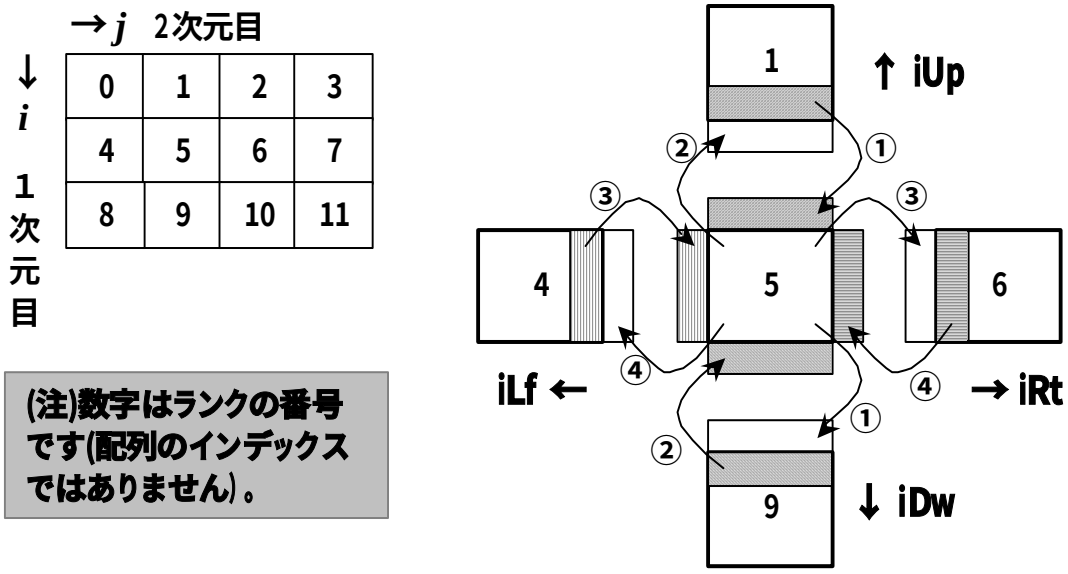
※前ページの実行例にあるように、存在しないランクには“－１”という値を返します。これは Compaq MPI での MPI_PROC_NULL の定義です。つまり **mpi_cart_shift** の返したランクの番号をいつでも **mpi_send/recv** の相手先の引数として使うことができます。

mpi_cart_create (comm, ndims, dims, periods, reorder, comm_cart, ierr)			
機能	プロセス座標グループを作成する		
引数	型	入出力	
comm	Integer	入力	所属するコミュニケータを指定
ndims	Integer	入力	作成するプロセス座標グループの次元数
dims	整数配列	入力	ｉ次元目のプロセス数
periods	整数配列	入力	ｉ次元目の境界条件 “.TRUE.”は周期境界、”.FALSE”で固定境界
reorder	Integer	入力	“.FALSE.”を指定
comm_cart	Integer	出力	プロセス座標グループのコミュニケータ
ierr	Integer	出力	終了コード

mpi_cart_shift (comm_cart, direction, disp, rank_source, rank_dest, ierr)			
機能	指定した方向のプロセスのランクの値を返す		
引数	型	入出力	
comm_cart	Integer	入力	プロセス座標グループのコミュニケータ
direction	Integer	入力	取出したいプロセス座標グループの次元 ｉ次元目の方向の場合『ｉ－１』を指定する
disp	Integer	入力	ｉ次元目方向でのランクの変位(離れた距離)
rank_source	Integer	出力	ｉ次元目で－disp だけ離れた位置にあるランク
rank_dest	Integer	出力	ｉ次元目で＋disp だけ離れた位置にあるランク
ierr	Integer	出力	終了コード

４.７. ブロック分割－行と列両方での分割による差分法モドキの計算

これで準備は完了です。４.２節の差分法モドキの式④を行と列両方で分割して計算してみることにします。ＣＰＵ数 12 個、周期的境界条件で計算する例を以下に示します。




```
      include 'mpif.h'
      parameter(Mdiv=3,Ndiv=4)
      parameter(MM=12,NN=16)
      parameter(Mmax=MM/3,Nmax=NN/4)
      logical period(2)
      integer istatus(MPI_STATUS_SIZE)
      integer idivid(2),iUp,iDw,iLf,iRt
      integer a(0:Mmax+1,0:Nmax+1), b(0:Mmax+1,0:Nmax+1)

      call mpi_init(ierr)
      call mpi_comm_rank(MPI_COMM_WORLD,irank,ierr)

      idivid(1) = Mdiv
      idivid(2) = Ndiv
      period(1) = .TRUE.
      period(2) = .TRUE.
      call mpi_cart_create(MPI_COMM_WORLD,2,idivid,period, &
                                        .FALSE.,iptbl,ierr)

      call mpi_cart_shift(iptbl,0,1,iUp,iDw,ierr)
      call mpi_cart_shift(iptbl,1,1,iLf,iRt,ierr)

      call mpi_type_vector(Mmax,1,Nmax+2,MPI_INTEGER,inewvec,ierr)
      call mpi_type_commit(inewvec,ierr)

      do j = 1, Nmax
         do i = 1, Mmax
            a(i,j) = i + j*100
```

４×３のプロセス座標テーブルを作成します。境界条件は、周期境界条件です。

“のりしろ”データ転送用の派生データタイプを作成します。

```
      end do
    end do

①   call mpi_sendrecv(a(Mmax,1),  1,inewvec,iDw,1, &
                       a(0,1),     1,inewvec,iUp,1, &
                       MPI_COMM_WORLD,istatus,ierr)

②   call mpi_sendrecv(a(1,1),      1,inewvec,iUp,1, &
                       a(Nmax+1,1),1,inewvec,iDw,1, &
                       MPI_COMM_WORLD,istatus,ierr)

③   call mpi_sendrecv(a(1,Nmax),Mmax,MPI_INTEGER,iRt,1,&
                       a(1,0),   Mmax,MPI_INTEGER,iLf,1,&
                       MPI_COMM_WORLD,istatus,ierr)

④   call mpi_sendrecv(a(1,1)     ,Mmax,MPI_INTEGER,iLf,1, &
                       a(1,Nmax+1),Mmax,MPI_INTEGER,iRt,1, &
                      MPI_COMM_WORLD,istatus,ierr)

    do j = 1, Nmax
       do i = 1, Mmax
          b(i,j) = a(i-1,j) + a(i,j-1) + a(i,j+1) + a(i+1,j)
       end do
    end do

    call mpi_comm_free(iptbl,ierr)
    call mpi_type_free(inewvec,ierr)
    call mpi_finalize(ierr)
    end
```

上側の“のりしろ”データ転送①

下側の“のりしろ”データ転送②

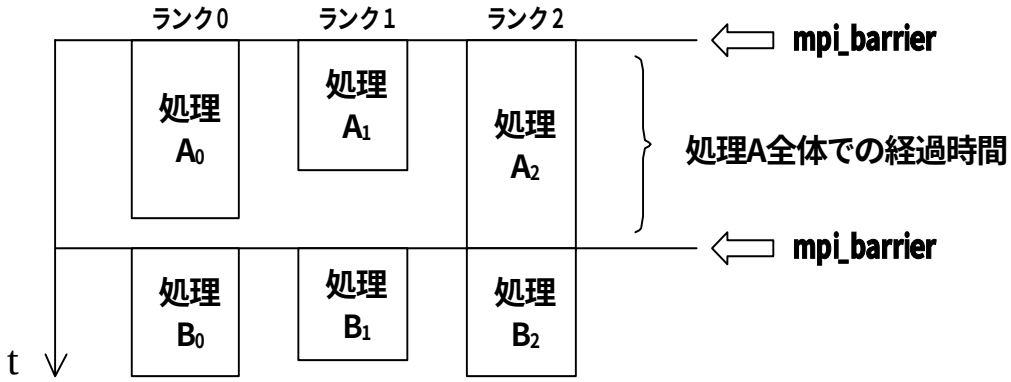
左側の“のりしろ”データ転送②

右側の“のりしろ”データ転送②

４.８. 処理時間を測定する

パフォーマンスを計測するためなど、プログラム中のあるサブルーチンや特定のループ部分などの処理時間を測定したい場合があります。しかし、並列プログラムでは各々のプロセスによってプログラムの経過時間が異なる場合があります。例えば下図のように処理 A_i の実行時間が各プロセスによって異なる場合、処理 A 全体でかかる時間を計測するには、すべてのプロセスで処理 A_2 の終了を待ってから計測を行わなければなりません。当然ながら開始時間も揃えなくてはなりません。

このように、コミュニケータ内の全プロセスの間で同期を取りたい場合、**mpi_barrier** というサブルーチンを使用します。また、経過時間の計測には Fortran の組込み関数として用意されている dtime や etime を使用することもできますが、MPI には **mpi_wtime** という使いやすい関数が用意されています。**mpi_wtime** はシステムのある時間からの経過時間を浮動小数点形式の秒数で返します。




```
include 'mpif.h'
read*8 tsta, tend
     ～
call mpi_barrier(MPI_COMM_WORLD,ierr)                    …①
tsta = mpi_wtime()
     ～
   計算部分
     ～
call mpi_barrier(MPI_COMM_WORLD,ierr)                    …②
tend = mpi_wtime()

print *, (tend-tsta)
     ～
```

※①の **mpi_barrier** のみ挿入し②の方を入れない場合には、各ランクそれぞれの処理時間を計測することになります。

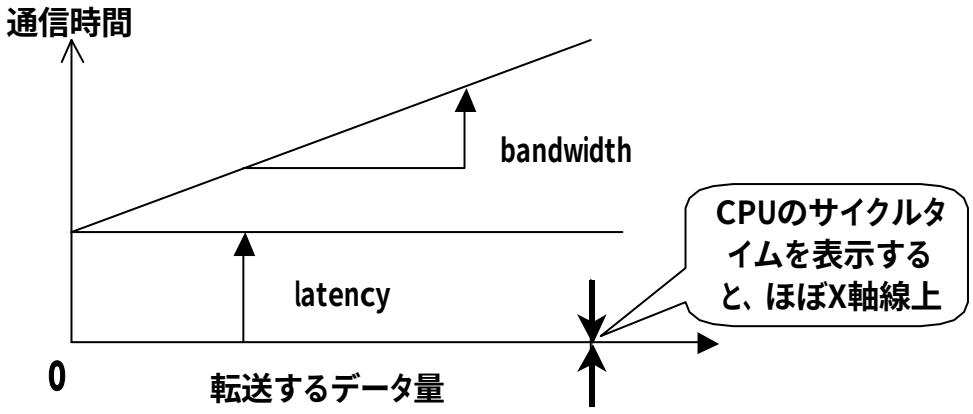
４.９. 並列プログラミング中級者への道のり

◆ 通信コストを考える

MPI プログラミングにおいて通信コストを減少させることが大切であると本章のはじめに書きました。通信はメモリ上にあるデータへアクセスするよりも、桁違いにコストがかかります。どのぐらいコストがかかるのか簡単に見積もってみることにします。
通信の性能を表す値には、実効転送速度(bandwidth)とレイテンシ(latency：遅延時間)の二つがあります。AlphaServer SC で使われているインターコネクト装置 QSW、Myrinet 2000、および Gigabit Ethernet の性能値を以下に示します。

	bandwidth	latency
QSW	200MB/s	5.5μs(MPI での計測)
Myrinet	150MB/s	15μs(MPI での計測)
Gigabit Ethernet	100MB/s	85μs(MPI での計測)

レイテンシから考えてみることにします。他のランクにデータを送れという命令(mpi_send 等)を出してから、“ 最初 ”のデータが実際に相手に届くまでにかかる時間がレイテンシです。簡単のために例えばレイテンシが５[μs]の装置があるとします。５[μs]という時間は速いように思われるかもしれませんが、現在の CPU の動作クロックは１GHｚのレベルです。
１GHｚの動作クロックとは、10^{-9} [s] = 1 [ns]に一度 CPU の命令が実行されることを示します(より正確には、Alpha CPU は、スーパースカラ CPU ですから１サイクルに複数の命令を実行できます)。５[μs]はこの 5000 倍です。他ランクのデータが送られてくるまでの間、何らかの命令を実行できないならば、CPU は 5000 サイクル以上も遊んでいることになってしまいます。つまり、他のランクへのデータの転送要求が頻繁に発生するようなプログラムでは、CPU はほとんど遊んでいる状態になります。
次に実効転送速度について考えてみます。例えば実行転送速度が 100[MB/s]であるとすると、倍精度(8byte)の配列の一部を転送する場合、倍精度データを一つ転送するのにかかる時間は 8byte／100[MB/s] = 80[ns]となります。100 個の配列要素を転送する場合、8[μs]の時間がかかりますので、この時も数千サイクルにわたって CPU が遊んでしまうことになります.
これらをまとめると次の図のようになります。



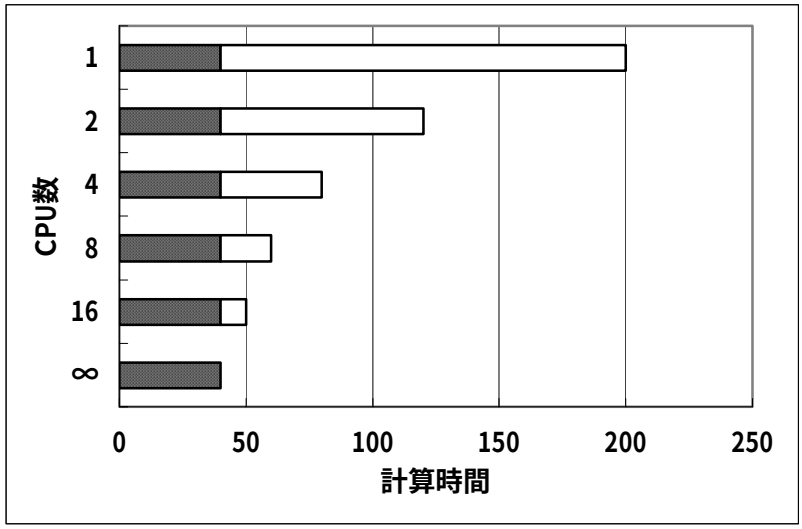
以上の通信の特性から、次の 2 点に注意しなければならないことが分かります。

1．なるべく通信量を減らす　→ 実行転送速度対策
2．(通信量が同じなら)通信回数を減らす　→ レイテンシ対策

◆ アムダールの法則

これまで MPI プログラミングの例をいろいろと見てきました。単純な例ではありますが、これから MPI プログラミングをはじめるために必要なサブルーチンはだいたい紹介されています。さあ、MPI プログラミングをはじめてみましょう！
といいたいところなのですが、その前にもう１度考慮しておかなければならない点があります。この章のはじめに並列化プログラミングにおいてパフォーマンスを向上させるための３つの注意点をあげました。これらに注意しなければ、せっかく並列化したのにシングル CPU の時より遅くなるという現象すら起こりかねません。また、手間をかけた割にはパフォーマンスが上がらず、KAP などの自動並列化や OpenMP などの SMP プログラミングで割合と手軽に並列化を行ったものとパフォーマンスが大して違わなかったということもありえます。ここでは並列化プログラミングの注意点について説明します。

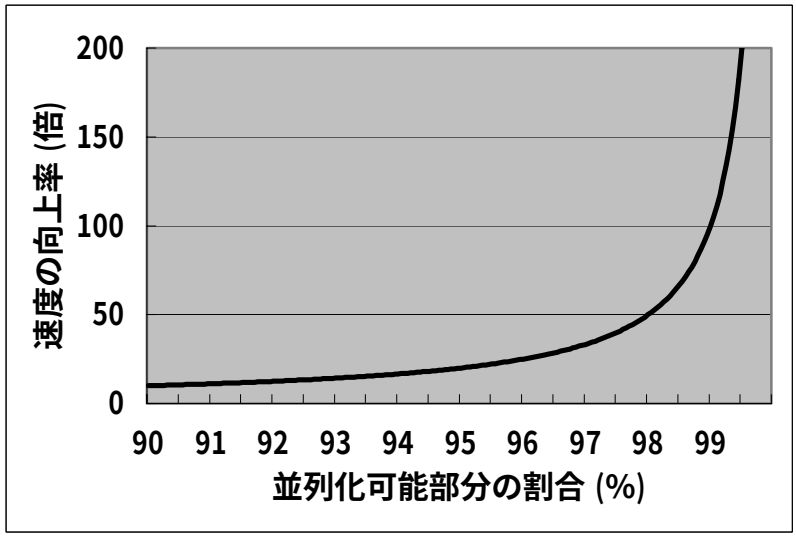
例えば、１CPU のマシンで計算に 200 分かかるプログラムがあるとします。
このプログラムを解析したところプログラム中最も計算コストのかかるループ部分を並列化できることがわかりました。このループ部分の計算時間は 160 分であったとすると、このプログラムを複数の CPU で並列実行させた場合、全体の計算時間は以下のようになります。



このように全体の 80%しか並列化できないプログラムでは、たとえ CPU を∞個用意しても 5 倍以上高速に実行することはできません。これを一般化していうと、

全体のｐ%の部分を並列化可能なプログラムの速度向上率は、
最大 100／(100－p) である

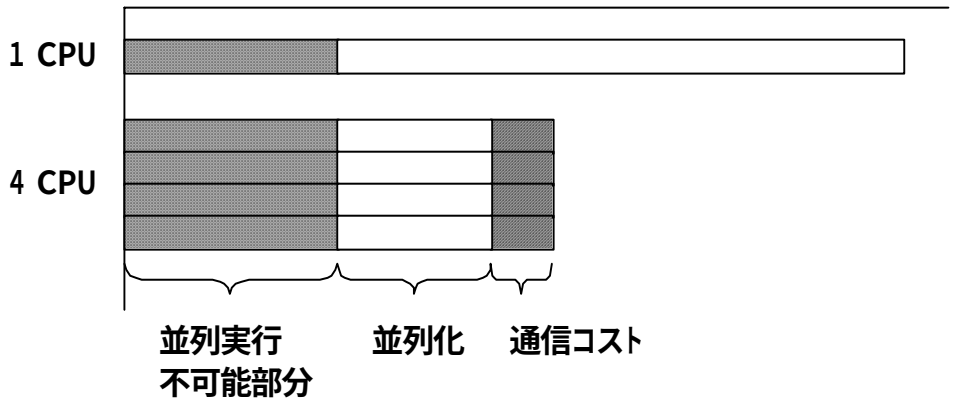
となります。これをアムダールの法則といいます。



上図はアムダールの法則をグラフにしたものです。これから並列化した場合の効果を推測することができます。例えば、並列化可能部分が 95%のプログラムがあるとします。このプログラムの最大の速度向上率は、20 倍です。このプログラムの実行時間が 200 分だとすると、理想的な条件の元で並列実行された場合、

8 CPU	→	34 分
16 CPU	→	22 分
32 CPU	→	16 分
64 CPU	→	13 分
128 CPU	→	11.5 分

となり、95%程度の並列化率では、CPU を 30 台程度以上に増やしても目に見える程には並列化の効果が現れなくなってしまします。逆に 64CPU、128CPU の並列コンピュータで並列化の効果をあげるためには、並列化可能部分がおそよ 99%以上なくてはなりません。
並列化可能な部分が多くないプログラムをいくら手間をかけて並列化しても、効果は頭打ちになり並列化のための努力は報われなくなってしまいます。その上、並列化に伴って生じる不足分を補うための通信のコストが新たに発生してしまいます。通信コストもある種並列実行不可能な部分と見なせますので、これもまた並列化の効率を低下させてしまします。



◆ おしまいに

並列化の手法は、本書で主に説明したデータ領域の分割だけではありません。例えば、パラメータを変えながら何度もプログラムを実行し最適なパラメータを探したい場合などは、パラメータごとにプロセスを分けて並列実行させることも可能です。また、プログラム中のサブルーチンに依存性がなく並列実行可能な場合は、サブルーチンごとに並列化することもできます。また、どうしても大規模な並列化に向かないプログラムがあった場合、そのようなプログラムに手間をかけて計算アルゴリズムを一から見直すよりも、手軽な SMP プログラミングでほどほどの効果をあげたり、シングル CPU で単体チューニングを行うことも一つの手段です。その方がプログラムの作成時間を含めたトータルとしての時間が削減され、目的の結果を得るまでの時間が短くなる場合もあります。

最後にもう一度、MPI プログラミングでハイパフォーマンスなプログラムを作成するための注意点を列記します。

1．並列可能な部分を大きくする

2．プロセス間の通信コストを減少させる
- **通信量を減らす**
- **通信回数を減らす**

3．プロセス間のロードバランスを均等にする

以上で MPI プログラミングの入門編はおしまいです。どうぞ MPI プログラミングをお楽しみください。本書がみなさんの MPI プログラミングの入門の手助けになれば幸いです。