MPIを用いた並列化

「天体とスペースプラズマのシミュレーションサマースクール」第 4 日 (2003/9/11)9:30-10:30

野澤 恵 (茨城大学理学部) snozawa@env.sci.ibaraki.ac.jp

1 はじめに

ネットワークに接続された計算機群 (クラスター) を、単一の並列計算機として利用するライブラリーの一つとして、MPI がある。MPI は **M**essage **P**assing **I**nterface[2] の略である。MPI は仕様を定めているだけで、その実装には MPICH[3],LAM/MPI[4] などがある。

クラスターは、分散メモリー型並列計算機 (MIMD:Multiple Instruction Multipule Data) の一種で、個々の計算機のメモリーは共有されず独立し、Message Passing (メッセージパッシング) 方式により、ノード (MPI ではプロセス、ランク (rank)、PE(Processing Element) などともいう) 間で通信を行う。ここでメッセージとは、データと制御情報から成り立っている。

図 1 では、その簡単な模式図を示した。送信側のノード (A:Machine(CPU,OS,,) で動作するもの) は、送信側の計算機のメモリーから送ろうとするデータ (a:Memory にある x(i) という変数) を含むメッセージ (x(i)+α) を作り、受信側のノード (B:Machine(CPU,OS,,) で動作するもの) に渡す (network)。受け取った受信側のノードはメッセージからデータを取り出し、受信側の計算機のメモリー (b:Memory) に書き込む。よって、メッセージを送受信する時に、送信側 (a:Memory) または受信側のメモリー (b:Memory) を直接アクセスしないのが特徴である。

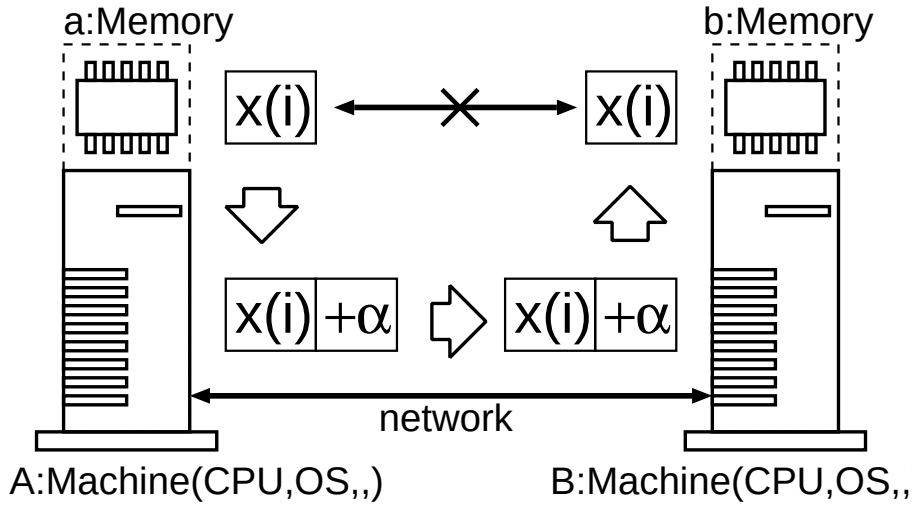

図 1: メッセージパッシングの模式図

この原理的のよると、必ずしもネットワークに接続されている必要はない。例えば単一の CPU の計算機であっても、複数のノードが動作する環境であれば、実際に数値計算やそのデバックなどを行なえる。しかし、メッセージの送受信によるオーバーヘッドが発生するために、並列化しない場合に比べて基本的に速度は低下する。

また、計算機やオペレーティングシステム (以下 OS) が異なると、エンディアン (メモリーの配置方法) の違いにより整数や浮動小数点数の表現が一般的に異なる。MPI はこの違いをソフトウエアで吸収する。また、メッセージを作り出すプロセスの起動や停止、及びネットワークでのデータのやりとりも自動化されている。

実装としては、C/C++ の関数または FORTRAN のサブルーチンとして、外部ライブラリーを提供している。具体的には、FORTRAN ではヘッダーファイルの宣言と MPI を呼び出すサブルーチンを記述し、コンパイル時にリンクする。

このため、既存のプログラムを「そのまま」では MPI 化することができない。そこで、既存のプログラムを「手動」で MPI 化するしかないのが現状である。

動作環境は、スーパーコンピューターから UNIX 環境、Windows でも動作している。インストールや設定、実際の起動の方法はここでは省き、プログラムだけに注目する。

世の中には共有メモリー型並列計算機 (SIMD: Single Instruction Multipule Data) もあり、スーパーコンピューターの一部や、パソコンでも SMP(Symmetric Multi Processor) と呼ばれるものである。その場合は MPI ではなくスレッド (thread) や OpenMP[5] などの並列化の方法があるが、ここでは触れない。

2　どのように並列化させるか

「天体とスペースプラズマのシミュレーションサマースクール」の「流体・磁気流体コース」で用いる方法は、主に格子を使った差分法のため、隣合う格子との計算を行う方法が多い。そこで、並列化は領域を分割する方法が単純でよく用いられている。例えば三次元の問題の場合では図 2 のように分割を一次元的にするのか、二次元、三次元的にする方法がある。一般的に分割する次元が多いほどプログラムは複雑化するが、並列計算の効果は大きくなる。

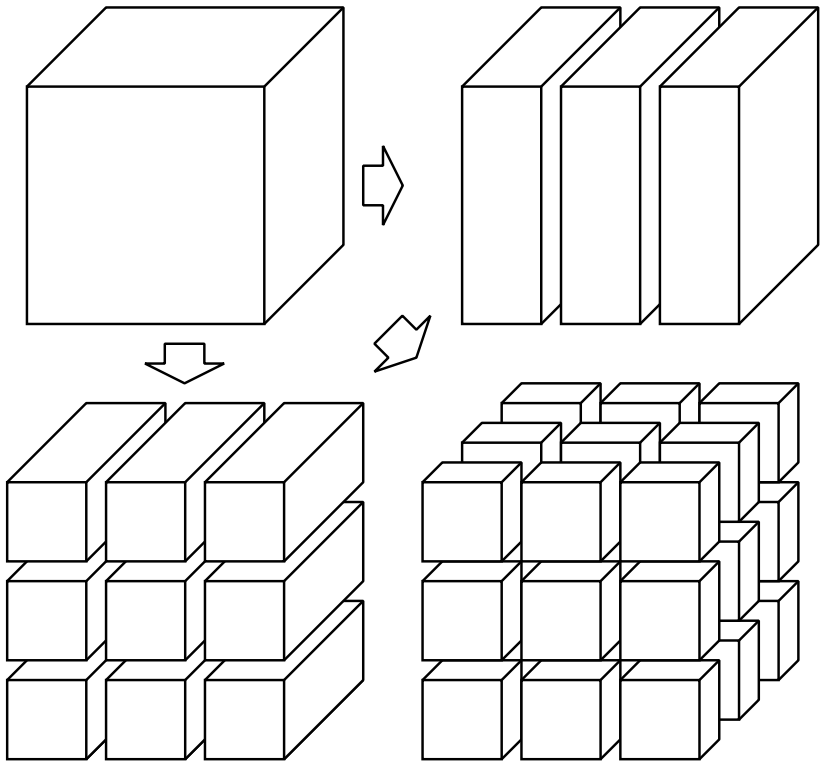

図 2: 三次元モデルの分割方法

簡単のために一方向の分割とした場合を図 3 に示した。初期値を設定し、分割された領域が受け持つ各ノードに分配し、各々のノード毎に閉じて計算を行う。しかし、流体計算では境界条件が発生するために、図 3 の「のりしろ」を考慮に入れて、通信を行い並列計算する必要がある。

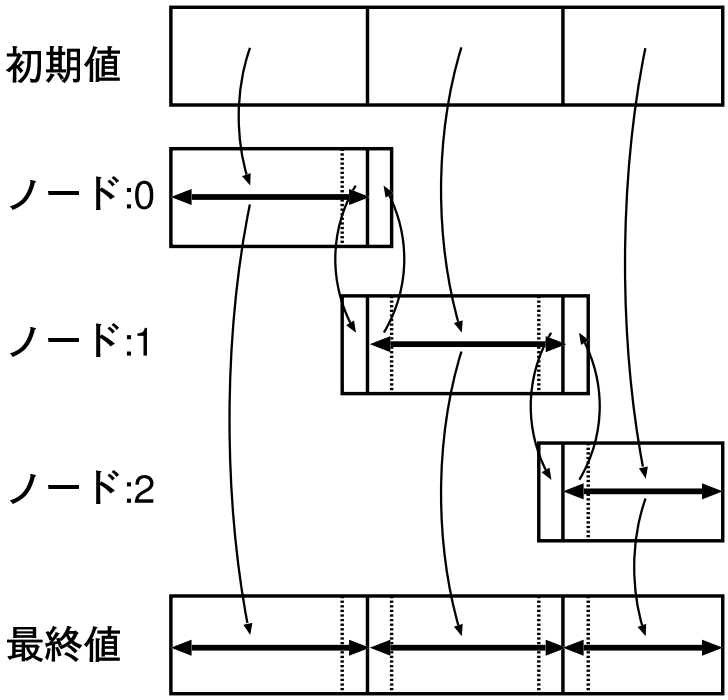

図 3: 領域分割の例、流体計算では各ノードに「のりしろ」が必要となる

また、計算後ノード毎の計算結果を、どう処理するのかも問題となる。そこで、以下は CANS の例を取り、MPI のサブルーチンを紹介する。以下では、特別に断らない限り、FORTRAN の暗黙の了解に従い、i-n で始まる変数は整数とする。

3 CANS1D の並列化の部分

3.1 main.f

`cans(or cans-current)/cans1d/mdp_shktb` にある `main.f` の MPI に関するサブルーチンを取り出してみる。

```
:
      include "mpif.h"
.
c------
c   for MPI
      call mpi_init(merr)
      call mpi_comm_size(mpi_comm_world,npe   ,merr)
      call mpi_comm_rank(mpi_comm_world,myrank,merr)
:
         call mpi_allreduce(dt,dtg,1,mpi_double_precision,mpi_min
     &                      ,mpi_comm_world,merr)
:
      call mpi_finalize(merr)
:
```

たったこれだけで、`main.f` の並列化が行なわれる (CANS は並列化が最初から考えられいるため、それほど並列化は難しくない)。ここで、最初の部分の

```
      include "mpif.h"
:
      call mpi_init(merr)
```

は初期化である。最初の `include "mpif.h"` は `mpif.h` というヘッダー (インクルード)

ファイルを使うということで、MPI を使用するには必ず必要なものである[*1]。

次の mpi_init は、今から MPI を始めるという宣言部である。merr は一つの整数変数で、返り値が入る。終了には

```
      call mpi_finalize(merr)
```

とする必要がある。mpi_init と同じように merr は一つの整数変数で、返り値が入る。他の MPI のサブルーチンにも同じように使用されている。そして、この間に MPI のサブルーチンを記述して、並列化のプログラムを作成する必要がある。上のプログラムでは、

```
      call mpi_comm_size(mpi_comm_world,npe   ,merr)
      call mpi_comm_rank(mpi_comm_world,myrank,merr)
:
         call mpi_allreduce(dt,dtg,1,mpi_double_precision,mpi_min
     &                      ,mpi_comm_world,merr)
```

がある。最初の mpi_comm_size 全ノード数を与えるもので npe にその値が入る。そして、mpi_comm_rank は実行している自分のノード番号を与えるもので myrank にその値が入る。ここで mpi_comm_world は、変更の必要はない。npe,myrank は、それぞれ一つの整数変数である。また myrank は 0,1,2,,, と 0 から始まるので注意が必要である。

次の mpi_allreduce は図 4 にあるように、各々のノードで異なる dt から最小値を見つけ (mip_min)、その値を再び各々のノードに dtg として分配するものである。

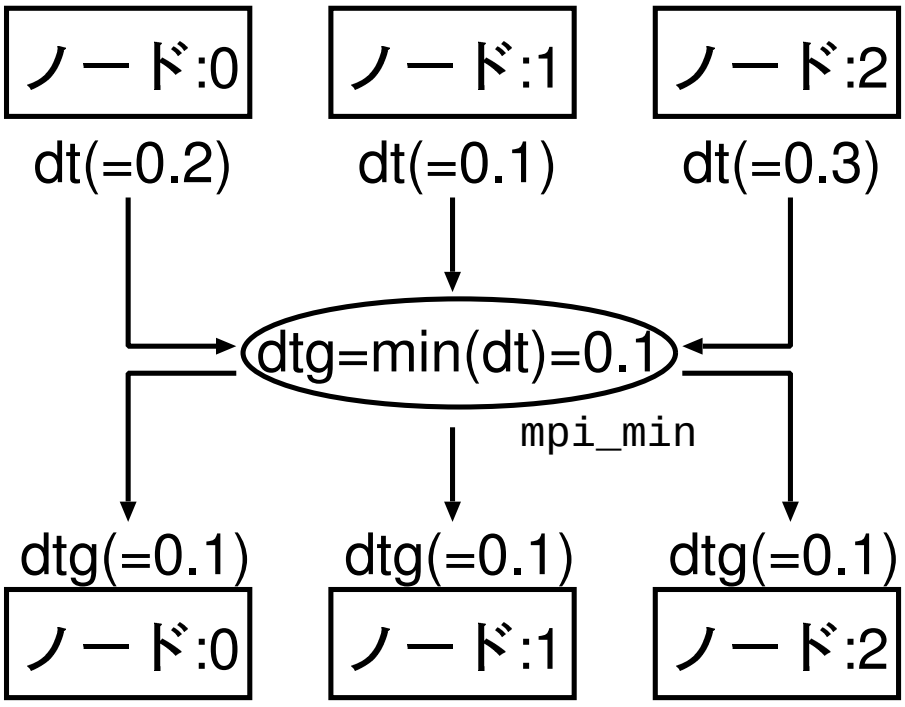

図 4: mpi_allreduce は各ノードの異った dt から最小値 dtg を求め (mpi_min)、配布する

この理由は、領域を分割する並列化を行なったために、ノードが受け持つ領域はそれぞれ物理

[*1] ヘッダーファイルにパスが通っていない場合は include "/opt/usr/local/mpi/mpif.h" のように絶対アドレスを用いて書く必要がある

量が異なり、結果的に CFL 条件である時間の刻み幅も異ってしまうからである。そこで、毎ステップに時間の刻み幅を各領域で同じ値になるように、各領域毎に最小値である時間の刻み幅を求めてから、その領域毎の時間の刻み幅の最小値を求める必要がある。そして求められた時間の刻み幅を各領域に再び転送する。このように領域分割した場合は同期を図る必要がある。

3.2 exc_h.f

次に `cans(or cans-current)/cans1d/commonmpi` にある`/exc_h.f` の MPI に関するサブルーチンを取り出す。これは差分法のために境界条件が発生し、隣との領域でその境界の値をやりとりするものである。いわゆる「のりしろ」部分の処理である。

```
c=========
      subroutine exc_h(margin,ro,pr,vx,ix,myrank,npe)
c=========
c---------
c  from PE(myrank) to PE(myrank+1) for new da(1)
c---------
      mright=  myrank+1
      mleft =  myrank-1
      if (myrank.eq.npe-1) mright  = mpi_proc_null
      if (myrank.eq.0    ) mleft   = mpi_proc_null
      do i=1,margin
        bufsnd(i,1)=ro(ix-2*margin+i)
        bufsnd(i,2)=pr(ix-2*margin+i)
        bufsnd(i,3)=vx(ix-2*margin+i)
      enddo
      call mpi_sendrecv
     &    (bufsnd,mmx,mpi_double_precision,mright ,1
     &    ,bufrcv,mmx,mpi_double_precision,mleft,1
     &    ,mpi_comm_world,mstatus,merr)
      if (myrank.ne.0) then
      do i=1,margin
        ro(i)=bufrcv(i,1)
        pr(i)=bufrcv(i,2)
        vx(i)=bufrcv(i,3)
      enddo
      endif
:
```

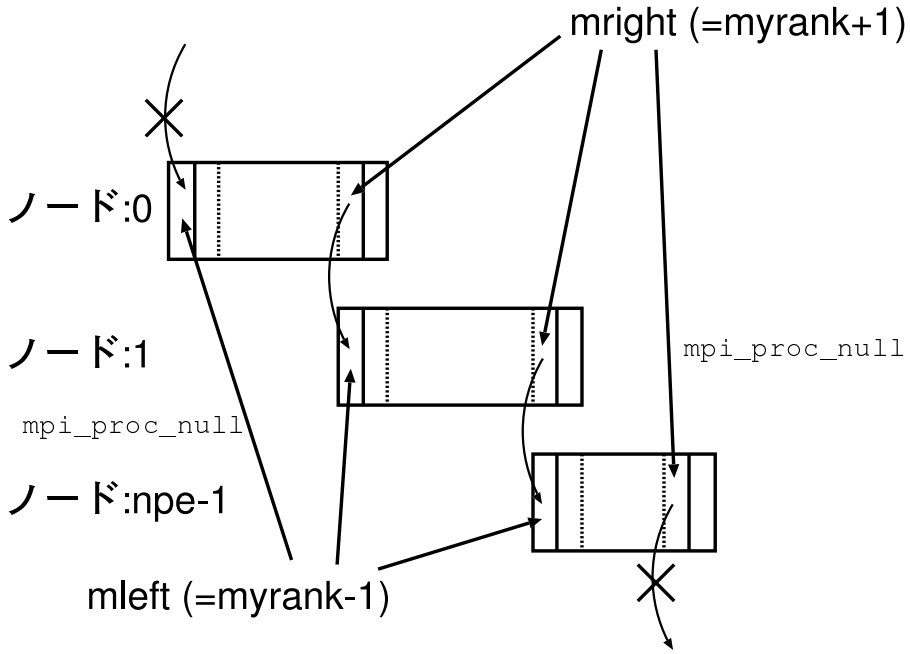

図 5: `mpi_sendrecv` はここでは「のりしろ」の送受信を行なう

図 5 のあるノードは、隣り合うノードから境界条件として物理量の転送が必要なため、

```
      mright=  myrank+1
      mleft =  myrank-1
      if (myrank.eq.npe-1) mright  = mpi_proc_null
      if (myrank.eq.0    ) mleft   = mpi_proc_null
```

`mright,mleft` では自分の右側と左側のノード番号の指定を自分のノード番号から `+1,-1` とすることで指定を行なう。しかし、`myrank,h=0,npe-1` の両端では送受信が発生しない (周期境界は除く) ため、`mpi_proc_null` という値を入れて、送受信の抑制を行なう。それが

```
       call mpi_sendrecv
     &    (bufsnd,mmx,mpi_double_precision,mright ,1
     &    ,bufrcv,mmx,mpi_double_precision,mleft,1
     &    ,mpi_comm_world,mstatus,merr)
```

という形となる。実際には `bufsnd,bufrcv` を用いているのは、図 6 に示すように複数の物理量を、一つの配列に直し、`mpi_sendrecv` で一気に送受信を行ない、その後、分配を行う。

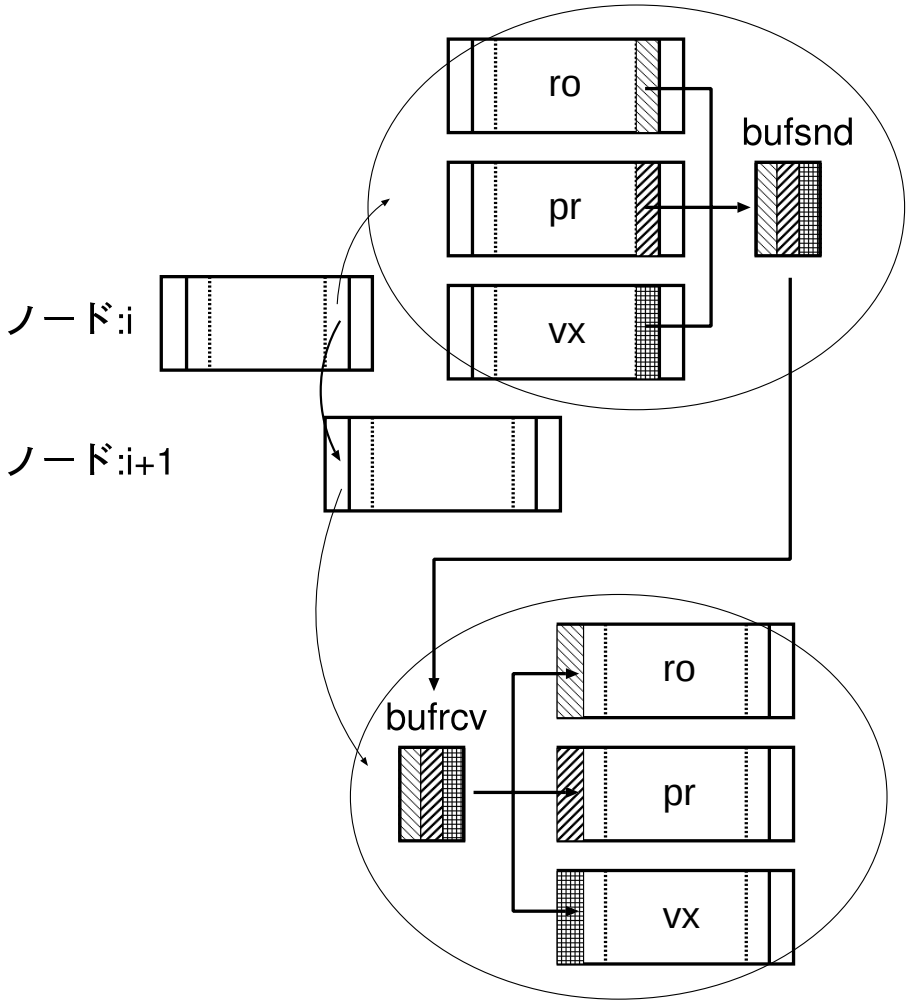

図 6: 各物理量の「のりしろ」の部分を一度一つ配列にして送受信、配布

4　MPI の主なサブルーチンの使い方

MPI のサブルーチンとして主なものを紹介する。これ以外にもあるので是非とも調べて、使ってほしい。

4.1 mpi_comm_size : 全ノード数を求める

mpi_comm_size(communicator,size,ierr)

引数	型	入出力	役割
communicator	integer	入力	通常は `mpi_comm_world`
size	integer	出力	全ノード数を返す
merr	integer	出力	終了コード

ex. `call mpi_comm_size(mpi_comm_world,npe,merr)`

4.2 mpi_comm_rank : 自分のノード番号を求める

mpi_comm_rank(communicator,rank,ierr) : $0 \leq$ rank $\leq$ size-1

引数	型	入出力	役割
communicator	integer	入力	通常は `mpi_comm_world`
rank	integer	出力	自分のノード番号を返す
merr	integer	出力	終了コード

ex.`call mpi_comm_rank(mpi_comm_world,myrank,merr)`

4.3 mpi_send : ブロッキング送信

mpi_send(buf,count,datatype,dest,tag,comm,ierr)

引数	型	入出力	役割
buf	変数	入力	変数、配列
count	integer	入力	要素の個数
datatype	integer	入力	変数、配列のデータタイプ
dest	integer	入力	受信する相手のノード番号
tag	integer	入力	メッセージのタグ
comm	integer	入力	通常は `mpi_comm_world`
ierr	integer	出力	終了コード

ex. `call mpi_send( x,1,mpi_real,0,itag,mpi_comm_world,merr)`

4.4 mpi_recv : ブロッキング受信

mpi_recv(buf,count,datatype,dest,tag,comm,status,ierr) : mpi_send に比べ status が必要

引数	型	入出力	役割
buf	変数	入力	変数、配列
count	integer	入力	要素の個数
datatype	integer	入力	変数、配列のデータタイプ
dest	integer	入力	送信された相手のノード番号
tag	integer	入力	メッセージのタグ
comm	integer	入力	通常は `mpi_comm_world`
status	integer	出力	通信結果
ierr	integer	出力	終了コード

ex. `call mpi_recv( x,1,mpi_real,i,itag,mpi_comm_world,mstatus,merr)`

4.5 mpi_isend : ノンブロッキング送信

`mpi_send` はブロッキング送信といわれ、デッドロックを起こす可能性がある。そこで、デッドロックを起さないノンブロッキング送信が別に `mpi_isend` として用意されている。しかし同期を行うために、後に `mpi_wait` を用いる必要がある[*2]。

mpi_isend(buf,count,datatype,dest,tag,comm,ireq,ierr) : mpi_send に比べ ireq が増加

引数	型	入出力	役割
buf	変数	入力	変数、配列
count	integer	入力	要素の個数
datatype	integer	入力	変数、配列のデータタイプ
dest	integer	入力	送信する相手のノード番号
tag	integer	入力	メッセージのタグ
comm	integer	入力	通常は `mpi_comm_world`
ireq	integer	入力	メッセージのリクエスト
ierr	integer	出力	終了コード

ex. `call mpi_isend( x,1,mpi_real,0,itag,mpi_comm_world,ireq,merr)`

4.6 mpi_irecv : ノンブロッキング受信

`mpi_recv` はブロッキング送信といわれ、デッドロックを起こす可能性がある。そこで、デッドロックを起さないノンブロッキング送信が別に `mpi_irecv` として用意されている。しかし同期を行うために、後に `mpi_wait` を用いる必要がある[*3]。

mpi_irecv(buf,count,datatype,dest,tag,comm,ireq,ierr) : mpi_recv の status が ireq に変更

引数	型	入出力	役割
buf	変数	入力	変数、配列
count	integer	入力	要素の個数
datatype	integer	入力	変数、配列のデータタイプ
dest	integer	入力	送信する相手のノード番号
tag	integer	入力	メッセージのタグ
comm	integer	入力	通常は `mpi_comm_world`
ireq	integer	入力	メッセージのリクエスト
ierr	integer	出力	終了コード

ex. `call mpi_irecv( x,1,mpi_real,i,itag, mpi_comm_world,ireq,merr)`

4.7 mpi_wait : ノンブロッキング通信終了

mpi_wait(ireq,status,ierr) : ireq,status が必要

引数	型	入出力	役割
ireq	integer	入力	メッセージのリクエスト
status	integer	出力	通信結果
ierr	integer	出力	終了コード

ex. `call mpi_wait(ireq, mstatus, merr)`

[*2] 送信に `isend` を使用したら `irecv` で受信する必要はなく `recv` で受信しても良いが、`mpi_wait` は必要である

[*3] `isend` と同様に、受信に `irecv` を使用したら `isend` で送信する必要はなく `send` で送信しても良いが、`mpi_wait` は必要である

4.8 mpi_sendrecv : ブロッキング送受信

送信と受信を一緒にするためにデッドロックが発生しない。

mpi_sendrecv(buf1,count1,datatype1,dest1,tag1,buf2,count2,datatype2,dest2,tag2,comm,status,ierr)

引数	型	入出力	役割
buf1	変数	入力	送信する変数、配列
count1	integer	入力	要素の個数
datatype1	integer	入力	送信する変数、配列のデータタイプ
dest1	integer	入力	送信する相手のノード番号
tag1	integer	入力	メッセージのタグ
buf2	変数	入力	受信する変数、配列
count2	integer	入力	要素の個数
datatype2	integer	入力	受信する変数、配列のデータタイプ
dest2	integer	入力	受信する相手のノード番号
tag2	integer	入力	メッセージのタグ
comm	integer	入力	通常は `mpi_comm_world`
status	integer	出力	通信結果
ierr	integer	出力	終了コード

ex. `call mpi_sendrecv(x,1,mpi_real,0,1,x0,1,mpi_real,i,1,mpi_comm_world,mstatus,merr)`

4.9 mpi_reduce : 集団通信により演算の結果をあるノードに送信

mpi_reduce(buf1,buf2,count,datatype,op,dest,comm,ierr) : op は演算

引数	型	入出力	役割
buf1	変数	入力	送信する変数、配列
buf2	変数	入力	受信する変数、配列
count	integer	入力	要素の個数
datatype	integer	入力	変数、配列のデータタイプ
op	integer	入力	演算
dest	integer	入力	送信する相手のノード番号
comm	integer	入力	通常は `mpi_comm_world`
ierr	integer	出力	終了コード

ex. `call mpi_reduce(dt,dtg,1,mpi_double_precision,mpi_min,0,mpi_comm_world,merr)`

4.10 mpi_allreduce : 集団通信により演算の結果を全ノードに送信

mpi_allreduce(buf1,buf2,count,datatype,op,comm,ierr) : op は演算、mpi_reduce に比べ dest が無い

引数	型	入出力	役割
buf1	変数	入力	送信する変数、配列
buf2	変数	入力	受信する変数、配列
count	integer	入力	要素の個数
datatype	integer	入力	変数、配列のデータタイプ
op	integer	入力	演算
comm	integer	入力	通常は `mpi_comm_world`
ierr	integer	出力	終了コード

ex. `call mpi_allreduce(dt,dtg,1,mpi_double_precision,mpi_min,mpi_comm_world,merr)`

4.11 mpi_bcast : 集団通信により、あるノードの値を全てのノードに送信

mpi_bcast(buf,count,datatype,dest,comm,ierr)

引数	型	入出力	役割
buf	変数	入力	変数、配列
count	integer	入力	要素の個数
datatype	integer	入力	変数、配列のデータタイプ
dest	integer	入力	送信するノード番号
comm	integer	入力	通常は `mpi_comm_world`
ierr	integer	出力	終了コード

ex. `call mpi_bcast(x,1,mpi_real,0,mpi_comm_world,merr)`

4.12 mpi_gather : 集団通信 全てのノードからあるノードに収集

この仲間に mpi_gatherv,mpi_allgather,mpi_allgatherv がある。

mpi_gather(buf1,count1,datatype1,buf2,count2,datatype2,dest,comm,ierr)

引数	型	入出力	役割
buf1	変数	入力	送信する変数、配列
count1	integer	入力	要素の個数
datatype1	integer	入力	送信する変数、配列のデータタイプ
buf2	変数	入力	受信する変数、配列
count2	integer	入力	要素の個数
datatype2	integer	入力	受信する変数、配列のデータタイプ
dest	integer	入力	受信する相手のノード番号
comm	integer	入力	通常は `mpi_comm_world`
ierr	integer	出力	終了コード

ex. `call mpi_gather(x,1,mpi_real,x0,1,0,mpi_comm_world,merr)`

4.13 mpi_scatter : 集団通信 あるノードから全てノードに分配

mpi_bcast と違う点は、ノード毎に違うものを転送できることである。この仲間に mpi_scatterv がある。

mpi_scatter(buf1,count1,datatype1,buf2,count2,datatype2,dest,comm,ierr)

引数	型	入出力	役割
buf1	変数	入力	送信する変数、配列
count1	integer	入力	要素の個数
datatype1	integer	入力	送信する変数、配列のデータタイプ
buf2	変数	入力	受信する変数、配列
count2	integer	入力	要素の個数
datatype2	integer	入力	受信する変数、配列のデータタイプ
dest	integer	入力	送信するノード番号
comm	integer	入力	通常は `mpi_comm_world`
ierr	integer	出力	終了コード

ex. `call mpi_scatter(x,1,mpi_real,x0,1,0,mpi_comm_world,merr)`

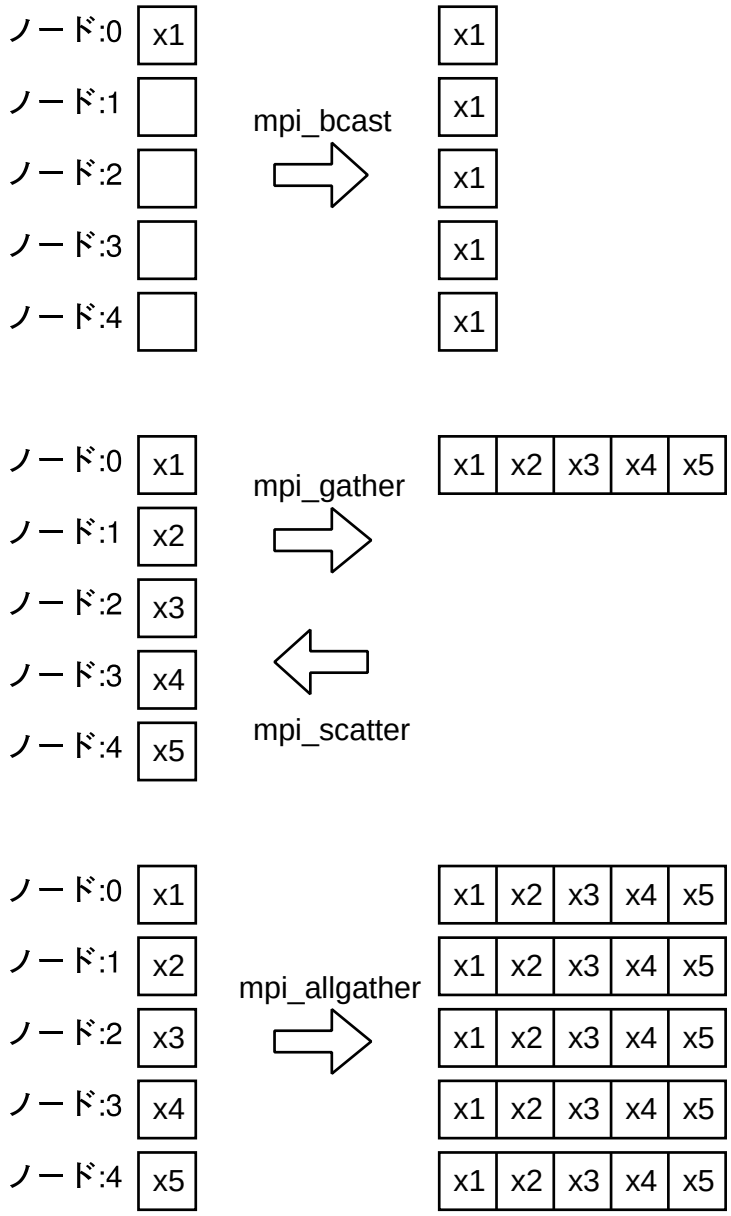

図 7: `mpi_bcast`, `mpi_gather`, `mpi_scatter`, `mpi_allgather`

4.14 変数、配列のデータタイプ

datatype に用いることができるもの以下がある。

データタイプ	byte 数	MPI 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_real8, mpi_double_precision`
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`

4.15 MPI で提供される演算 (op)

mpi_reduce,mpi_allreduce 等で使用できる演算に以下がある。また、mpi_op_create で独自の演算を作ることもできる。

演算のタイプ	演算	可能なのデータタイプ
mpi_sum	和	mpi_integer,mpi_real,mpi_real8,mpi_complex
mpi_pro	積	mpi_integer,mpi_real,mpi_real8,mpi_complex
mpi_max	最大	mpi_integer,mpi_real,mpi_real8
mpi_min	最小	mpi_integer,mpi_real,mpi_real8
mpi_maxloc	最大と位置	mpi_2integer,mpi_2real,mpi_2double_precision
mpi_minloc	最小と位置	mpi_2integer,mpi_2real,mpi_2double_precision
mpi_land	論理積	mpi_logical
mpi_lor	論理和	mpi_logical
mpi_lxor	xor(排他的論理和)	mpi_logical
mpi_band	ビット論理積	mpi_integer,mpi_byte
mpi_bor	ビット論理和	mpi_integer,mpi_byte
mpi_bxor	ビット xor	mpi_integer,mpi_byte

5 最後に

実行として、例えば Unix に MPI 環境を構築しているならば、

```
mpif77 (or mpif90) test.f
```

でコンパイルすることができる。実行は

```
mpirun -np n(ここにノード数が入る) a.out (ex. mpirun -np 2 ./a.out)
```

とする。ただし、デッドロックが起った場合は、シェルに戻らないので、C-c 等で強制的に実行を中止させ、ps でプロセスを確認し、

```
kill -9 (プロセス番号) or killall -9 a.out
```

などとプロセスを無くすようにしないと、残ったプロセスが次の実行に影響を与えることがあるので注意する。また以下に

```
http://www.env.sci.ibaraki.ac.jp/~snozawa/mpi/
```

この文書と、例題として簡単なプログラムと解説を載せている。参考になれば幸いである。

6 参考文献

[1] http://www.env.sci.ibaraki.ac.jp/~snozawa/mpi/summer03/
[2] http://www.mpi-forum.org/
[3] http://www-unix.mcs.anl.gov/mpi/mpich/
[4] http://www.lam-mpi.org/
[5] http://www.openmp.org/