＜特集：炉物理研究への PC クラスタの利用・並列計算プログラミング超入門＞

並列計算プログラミング超入門

佐々木誠　(株)日本総合研究所　　sasaki.makoto@jri.co.jp

さて、ここまでの記事であなたの手元には PC クラスターが構築されているでしょう。ただ、そのままでは単なる PC をネットワークでつないだシステムにすぎません。これからこの上で「並列計算」を行なうソフトウェアを自ら構築するか、他所から導入するかするわけですが、そのためには並列処理を手助けするソフトウェアを導入する必要があります。

ここではそのようなソフトウェアとして MPI(Message Passing Interface)を導入し、あわせて簡単な例で MPI を用いて並列計算を行なうためにはどのようなプログラミングを行なうかを示します。

１. MPI の導入

MPI(Message Passing Interface)は並列処理を行なうプログラミングを行なうためのプログラミングインターフェイスの規格であり、米欧日を中心とした企業や機関を会員とする非営利機関である MPI Forum で議論され決定されています。現在、MPI-2.0 までの規格が決定されています(http://www.mpi-forum.org/)。

MPI Forum はプログラミングのためのインターフェイス、すなわち C 言語関数および FORTRAN のサブルーチンや関数を決めているだけで、それらの関数ライブラリの構築や、どのようにして並列計算を行なわせるかについては別途それらの実装者を想定しています。代表的な実装として MPICH （http://www-unix.mcs.anl.gov/mpi/mpich/）や LAM(http://www.lam-mpi.org/)があります。MPICH は各種メーカー製並列計算機用ライブラリーのベースにも採用されている等の実績があり利用者ベースが大きいと思われますので、ここでも MPICH を導入する事にして解説します。

●MPICH のコンパイルとインストール

クラスターの OS を Linux としたとき、RPM などのパッケージとして MPICH が提供されている場合もあると思いますが、ここではソースからコンパイルしてインストールする方法を解説します。まず MPICH の上記の WWW ページからソースをダウンロードします("Download MPICH"をクリックしてダウンロードのページに行きます)。ファイル名は mpich.tar.gz でこの記事の執筆中の最新バージョンは mpich-1.2.4 です。

さて mpich.tar.gz を入手したら適当な場所でこれを展開します；

```
tar zxvf mpich.tar.gz
```

mpich-1.2.4 という名前のディレクトリが作られているはずですのでそこに移動して configure コマンドを実行します。

```
cd mpich-1.2.4
./configure -prefix=/home/mpich
```

ここで"-prefix=/home/mpich"という「オプション」をつけたのは MPI がインストールされる先を現在の場所ではなくて違う場所にしたいときに有効です。特にそのインストール先が(今の場合は/home/mpich)クラスターを構成するマシン間で NFS などによるファイル共有されている場所であれば、クラスターの各マシンでいちいち MPICH のインストールを行なわなくてもどれかのマシンで一回だけインストールを行なえばよいということになります。

configure コマンドの実行で Makefile が作成されますのでコンパイルを行ないます。

```
make
```

コンパイルが終了したらインストールを行ないます(インストール先が root ユーザーでなければ書き込みが出来ないところなら su コマンドで root ユーザーになっておきます)。

```
make install
```

インストールが終了したら MPICH 関係のコマンドが実行出来るように環境変数 PATH(bash をログインシェルにしている場合)あるいはシェル変数 path(tcsh をログインシェルにしている場合)に MPICH をインストールしたディレクトリ以下の bin ディレクトリが含まれるようにする必要があります。ログインシェルに応じて以下の内容をログインシェルの初期化コマンドファイルに追加します。

・bash の場合： ファイル".bashrc"に以下を追加します。

```
PATH=${PATH}:/home/mpich/bin
export PATH
```

・tcsh の場合： ファイル".cshrc"もしくは".tcshrc"に以下を追加します。

```
path=($path /home/mpich/bin)
```

上記で/home/mpich/bin は適当に貴方のインストール先に読み変えて下さい。

●MPICH の設定変更

MPICH をインストールした後で設定を変えておいた方がよいことがあります。インストールした先のディレクトリに share というディレクトリがあり、その中に machines.LINUX というファイルがあるはずです。そしてたとえばインストールを行なった PC のマシン名が"hogehoge1"であった場合その内容に以下のような行があるはずです。

```
hogehoge1
hogehoge1
hogehoge1
hogehoge1
hogehoge1
```

このままでは並列計算コマンドで計算を実行するとき、計算を行なうマシンが hogehoge1 ばかりになり、複数マシン並列計算になりません。もし PC クラスターが hogehoge1 から hogehoge8 までの 8 台の PC で構成されている場合には以下のように書き換えておきます。

```
hogehoge1
hogehoge2
hogehoge3
hogehoge4
hogehoge5
hogehoge6
hogehoge7
hogehoge8
```

●rsh、rlogin が実行できる環境にする。

MPICH が動作するためには rsh(remote shell)と rlogin(remote login)がクラスタを構成する PC 間でパスワード無しで実行出来る環境でなければなりません。そのような設定を可能にするには /etc/hosts.equiv にマシン名のリストを記述します(root ユーザ権限が必要です)。

●RedHat7.2 での rsh、rlogin の問題

各 PC にインストールされた Linux が RedHat7.2 以降の場合、デフォルトでは rsh、rlogin が使えない設定になっています。これらのコマンドの実行を可能にするには /etc/xinetd.d/rsh と /etc/xinetd.d/rlogin に含まれる以下のような行を；

```
disable                 = yes
```

以下のように書き換えておきます；

```
disable                 = no
```

このあと xinetd デーモンの再起動を行ないます；

```
/etc/rc.d/init.d/xinetd restart
```

なおこれらの処理をおこなうには root ユーザー権限が必要です。
同様のケースが RedHat Linux 以外にもあるかも知れませんので注意してください。

２.MPI による並列計算プログラミング(1)

MPI では並列処理を行なうための様々な関数(サブルーティン)が用意されていて、並列計算プログラミングを行なう人はそれらを利用したプログラムをコーディングします。まず簡単な例を FORTRAN で書いたものを以下に示します。これは MPICH のサンプルとしてついてくるものと同様のものですが、数値積分によって円周率πの計算を行ないます。

```
C==============================================================
      implicit double precision (a-h,o-z)
      include 'mpif.h'
C
      call MPI_Init(ierr)
      call MPI_Comm_Size(MPI_COMM_WORLD,nproc,ierr)
      call MPI_Comm_Rank(MPI_COMM_WORLD,myrank,ierr)
C
      if( myrank.eq.0 ) then
         read(*,*) n
         write(*,*) 'Number of divisions ',n
      end if
C
      call MPI_Bcast(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
C
      t1 = MPI_Wtime()
C
      h = 1.0d0/n
      pisum = 0
      do i=myrank,n-1,nproc
         x  = (i+0.5)*h
         pisum = pisum + sqrt(1-x*x)
      end do
      pisum = 4.0*h*pisum
C
      call MPI_Reduce(pisum,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,
     &                MPI_COMM_WORLD,ierr)
C
      t2 = MPI_Wtime()
C
      if( myrank.eq.0 ) then
         write(*,'(a,1p,d18.10)') 'Calculated Pi is ',pi
         write(*,*) 'Calculattion time is ',t2-t1
      end if
C
      call MPI_Finalize(ierr)
C
      end
C==================================================================
```

このサンプルコードで使われている"MPI_"で始まるサブルーチンが MPI の FORTRAN サブルーティンです。また include 'mpif.h' があることに注意して下さい。FORTRAN の MPI サブルーチンの引数に MPI_INTEGER、MPI_COMM_WORLD といった定数が必要になりますが mpif.h はこれらを定義しているファイルで、一般にはこれをインクルードします(C 言語の場合にも同様に'mpi.h'をインクルードします)。

このプログラムをコンパイルするには MPICH でインストールされるコマンド mpif77 を用います。たとえば上記のプログラムが ex1.f というソースファイルにあるのなら；

```
mpif77 ex1.f
```

一般的には。

`mpif77` [*options*] *sources*

また C 言語のプログラム用にはコマンド mpicc、Fortran90 用には mpif90、C++用には mpiCC があります。

コンパイルで得られたバイナリファイルを a.out とすると、これを 4 つのプロセスで並列計算させるには mpirun コマンドを用います；

```
mpirun -np 4 a.out
```

一般には；

`mpirun -np` *number_of_process* *program* [*arguments*]

コンパイルで得られたバイナリファイルの存在するディレクトリとそれを実行するディレクトリが異なる場合には *program* はフルパス名で指定する必要があります。たとえば a.out のフルパス名が/home/mpitest/a.out で、mpirun を実行するディレクトリが /home/mpitest でない場合には以下のようにしなければなりません；

```
mpirun -np 4 /home/mpitest/a.out
```

さてプログラムソースに戻って、なぜこのプログラムが並列処理になっているのかを各

MPI サブルーチンの説明を通して解説します。

◎ MPI_Init(ierr)

これが呼ばれることによって MPI でのメッセージ交換が出来るようになります。実際には mpirun によるプログラムの実行の段階で同じプログラムがすでに複数動いているのですが、MPI_Init()を呼び出すことで、動いている各プログラム(これらを「プロセス」とよぶことにします)が MPI のプロセスとしてメッセージの交換などができるようになります。MPI での並列計算では必ず call します。

ierr はエラーコードで MPI の FORTRAN プログラムではほとんどのものがこれを最後の引数とします(C 言語の場合には関数の戻り値になります)。

◎ MPI_Comm_Size(communicator,nproc,ierr)

並列計算がいくつのプロセスで行なわれているかを知るために呼び出します。並列プロセス数をこれで与えるわけではありません。並列プロセス数は mpirun コマンド呼び出し時に決まっています。

communicator という引数がありますが、これは並列プロセスのまとまりに対して与えられた識別番号のようなもので、各種の MPI 呼び出し関数で指定されます。大抵の場合ここではデフォルトで決まっていて mpirun で生成する全てのプロセスを含んでいる MPI_COMM_WORLD を使用します。自分でプロセスの別の束ね方を指定して communicator を作るということもできますが、これはかなり凝った処理をする場合に必要でしょう。

◎ MPI_Comm_Rank(communicator,rank,ierr)

自分が communicator 中の何番目のプロセスであるかを得ます。番号は 0 番から順に付けられます。rank が 0 のプロセスは mpirun を実行したマシンのプロセスになるので、計算の入出力などを担当させることが多くなります。また計算の分担を決めるとき等にも rank を使用します。

◎ MPI_Bcast(buffer,number_of_data,data_type,source,communicator,ierr)

配列 buffer に含まれる number_of_data 個のデータを rank 値が source のプロセスから communicator の全プロセスに同報します。すなわち送り手のプロセス source の配列 buffer の値が、そのプロセス以外のすべてのプロセスの配列 buffer にコピーされます。例では積

分区間の分割数 n だけを同報していますので number_of_data は 1、また data_type は整数型なので MPI_INTEGER となります。データ型を指定する定数としては、他に MPI_REAL, MPI_DOUBLE_PRECISION などもあります(C 言語では MPI_INT、MPI_FLOAT、MPI_DOUBLE などとなります)。

◎ MPI_Reduce(buffer1,buffer2,number_of_data,data_type,operation,dest,
communicator,ierr)

配列 buffer1 に含まれる各データに対して operation で指定される「縮約」操作をおこない、それをランクが dest の配列 buffer2 に返します(buffer1 と buffer2 にメモリ上で重なりがあってはいけません)。例では operation が MPI_SUM となっていて総和計算を行なっています。operation には他に MPI_MAX, MPI_MIN(最大値、最小値)、MPI_PROD(総乗)などがあります。

ここで例にあげたプログラムでは pisum という変数の全プロセス総和を rank が 0 のプロセスの pi という変数に入れています。この pisum という変数の計算のされ方を見てみましょう。通常なら；

```
pisum = 0
do i=0,n-1
   ....
end do
```

と、n 分割された積分区間の全てを計算しないと答えになりませんが、例では；

```
pisum = 0
do i=myrank,n-1,nproc
   ....
end do
```

となっていて図 1 のようにプロセス毎にひとつずつずれた開始インデックスから nproc おきに計算していて、各プロセスの計算量は前の場合の 1/nproc になっています。そしてこれらの総和をとってはじめて積分が完了します。このため並列計算での各タスクの計算量が減って、全体としての計算時間が小さくなります。
もちろん、以下のような分割でも同じようなことになります；

```
nn = (n+nproc-1)/nproc
n1 = myrank*nn
n2 = n1+nn-1
if(myrank.eq.nproc-1) n2 = n-1
pisum = 0
do i=n1,n2
   ....
end do
```

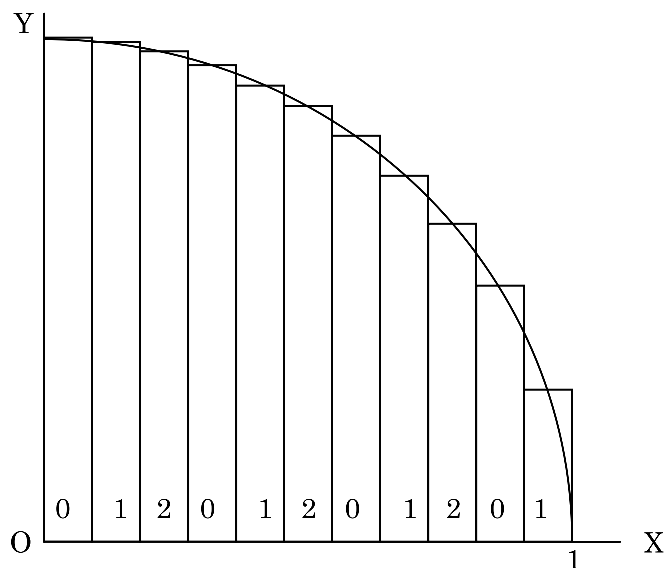

図１　積分区間のプロセスへの分解（３プロセスの場合）

各プロセスでの計算値の総和を行なうことで答えが得られるものには、数値積分以外に、放射線輸送モンテカルロ計算などがあります。

◎ MPI_Wtime()

経過時間(CPU ではなくていわゆる wall clock time)を返します。MPI_Wtime()は MPI のライブラリーでは例外的に FORTRAN でも関数呼び出しになっています。

◎ MPI_Finalize(ierr)

これが呼ばれることによって MPI でのメッセージ交換が終了します。これ以降は MPI_Reduce などの MPI 処理を行なうことはできません。

MPI のライブラリー関数(サブルーチン)はまだまだありますが、詳しくは MPI フォーラムのホームページから入手できるドキュメント、あるいは Web ページを参照して下さい。

３.MPI による並列計算プログラミング(2)

前項で挙げたものよりもう少し複雑な例を挙げてみます。この例では以下のようなポアッソン方程式を２次元の差分法で離散化し SOR 法で解くものです。

$$\nabla^2 \phi(\mathbf{r}) = S(\mathbf{r})$$

境界条件は図 2 のようにしています。２次元のメッシュは図 3 のように Y 方向に分割されます。

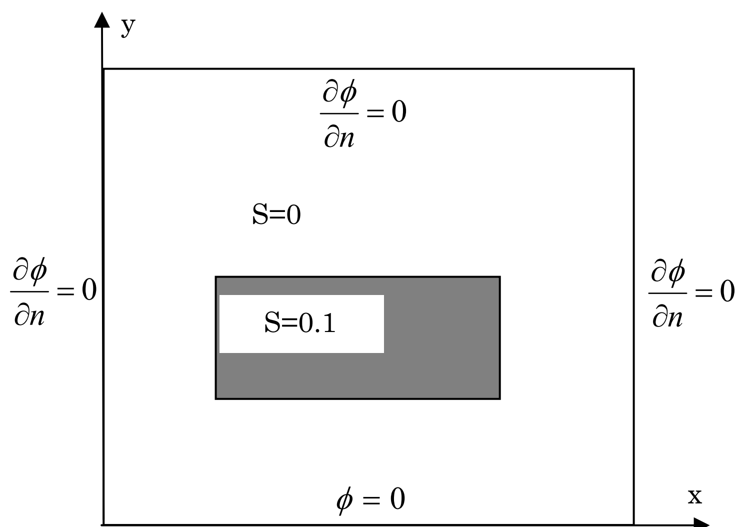

図２　例題の境界条件とソース分布

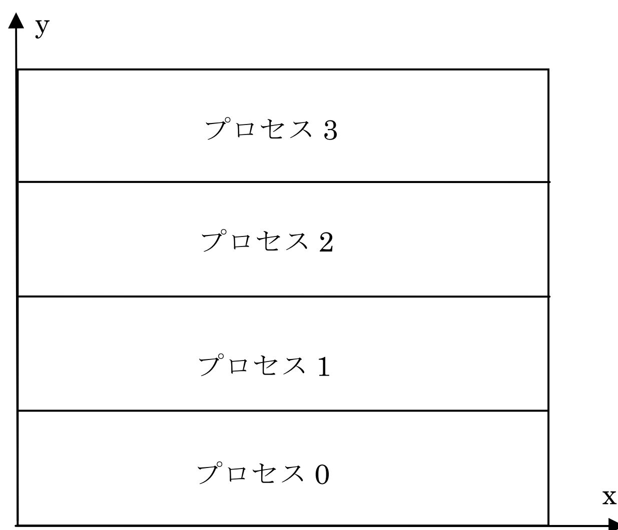

図３　並列プロセスへの分解（4 プロセスの場合）

まずはメインルーチンと計算結果の出力ルーチンです。

```
      implicit double precision (a-h,o-z)
      include 'mpif.h'
c
      parameter (memsize = 500000)
      common /marea/ a(memsize)
c
```

```
      call MPI_Init(ierr)
      call MPI_Comm_Size(MPI_COMM_WORLD,nproc,ierr)
      call MPI_Comm_Rank(MPI_COMM_WORLD,myrank,ierr)
C
      if( myrank.eq.0 ) then
        read(*,*) nx,ny
        write(*,'(a,2i10)') 'Number of divisions ',nx,ny
      end if
C
      call MPI_Bcast(nx,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
      call MPI_Bcast(ny,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
C
      nn = (ny+nproc-1)/nproc
      ny1 = myrank*nn + 1
      ny2 = ny1 + nn - 1
      if ( myrank.eq.nproc-1 ) ny2 = ny
C
      lphi = 1
      lphi2 = lphi + nx*(ny2-ny1+1+2)
      lmem = lphi2 + nx*(ny2-ny1+1)
      if ( lmem > memsize+1 ) then
        write(*,*) '== process ',myrank,' memory over ',lmem-1
        stop 1
      end if
C
      call MPI_Barrier(MPI_COMM_WORLD,ierr)
C
      call poisson( nx,ny,ny1,ny2,myrank,nproc,a(lphi),a(lphi2))
C
      nymax = nn
C
      lphi3 = lphi2
      lmem = lphi3 + nx*nymax
      if ( lmem > memsize+1 ) then
        write(*,*) '== process ',myrank,' memory over ',lmem-1
        stop 1
      end if
C
      call output(nx,ny,ny1,ny2,myrank,nproc,
     &          a(lphi),a(lphi3),nymax)
C
      call MPI_Finalize(ierr)
C
      stop
      end
C-----------------------------------------------------------------
      subroutine output(nx,ny,ny1,ny2,myrank,nproc,phi,phi3,nymax)
      include 'mpif.h'
      double precision phi(nx,ny1-1:ny2+1)
      double precision phi3(nx,nymax)
      integer mpistat(MPI_STATUS_SIZE)
      nn = (ny+nproc-1)/nproc
      do ir = 0,nproc-1
        iny1 = ir*nn + 1
        iny2 = iny1 + nn - 1
        if ( ir.eq.nproc-1 ) iny2 = ny
        kk = iny2-iny1+1
        if( ir.gt.0.and.ir.eq.myrank ) then
          call MPI_Send(phi(1,ny1),nx*kk,MPI_DOUBLE_PRECISION,0,
     &          123+ir,MPI_COMM_WORLD,ierr)
        end if
        if ( myrank.eq.0 ) then
```

```
          if ( ir.gt.0 ) then
             call MPI_Recv(phi3(1,1),nx*kk,MPI_DOUBLE_PRECISION,ir,
     &             123+ir,MPI_COMM_WORLD,mpistat,ierr)
          else
             do j=1,kk
                do i=1,nx
                   phi3(i,j) = phi(i,ny1+j-1)
                end do
             end do
          endif
          do j = 1,kk
             write(10,*) (phi3(i,j),i=1,nx)
          end do
       endif
      end do
      return
      end
```

ここで新しい関数 MPI_Barrier が登場しました。

◎ MPI_Barrier(communicator,ierr)

communicator に所属するすべてのプロセスがこの関数を実行するまで待ち合わせを行います。すべてのプロセスが同時に何かの処理を始めなければいけないような場合に利用します。ここでは後続の poisson ルーチン内で実行時間測定を行うために、poisson ルーチンの開始がすべてのプロセスで同期するようにするために使用しています。

そしてソースを与える関数とポアッソン方程式を SOR 法で解く本体サブルーチン poisson()です。

```
C---------------------------------------------------------------
      double precision function S(i,j,nx,ny)
C ... source
      S = 0
      if ( i.ge.nx/4.and.i.le.nx/2
     &    .and. j.ge.ny/4.and.j.le.3*ny/4) then
        S = 0.1
      end if
      return
      end
C---------------------------------------------------------------
      subroutine poisson( nx,ny,ny1,ny2,myrank,nproc,phi,phi2)
      implicit double precision (a-h,o-z)
      include 'mpif.h'
      double precision phi(nx,ny1-1:ny2+1)
      double precision phi2(nx,ny1:ny2)
      integer mpistat(MPI_STATUS_SIZE)
C    ... SOR acceleration factor ....
      w  = 1.9d0
      w1 = 1.0d0 - w
C
      t1 = MPI_WTime()
      lcount = 0
```

```
  100 continue
      lcount = lcount + 1
C ..... send/recieve process boundary values
      if ( ny1.ne.1 ) then
         call MPI_Send(phi(1,ny1),nx,MPI_DOUBLE_PRECISION,myrank-1,
     &            1,MPI_COMM_WORLD,ierr)
      end if
      if ( ny2.ne.ny ) then
         call MPI_Send(phi(1,ny2),nx,MPI_DOUBLE_PRECISION,myrank+1,
     &            2,MPI_COMM_WORLD,ierr)
      end if

      if ( ny1.ne.1 ) then
         call MPI_Recv(phi(1,ny1-1),nx,MPI_DOUBLE_PRECISION,myrank-1,
     &            2,MPI_COMM_WORLD,mpistat,ierr)
      end if
      if ( ny2.ne.ny ) then
         call MPI_Recv(phi(1,ny2+1),nx,MPI_DOUBLE_PRECISION,myrank+1,
     &            1,MPI_COMM_WORLD,mpistat,ierr)
      end if
C
      do j=ny1,ny2
         do i=1,nx
            phi2(i,j) = phi(i,j)
         end do
      end do
C
      if ( ny1.eq.1 ) then
C      .... Dirichlet condition.
         do i=1,nx
           phi(i,1) = 0.0
         end do
      else
         phi(1,ny1) = w1*phi(1,ny1) +
     &      w*(phi(2,ny1) + phi(1,ny1-1) + phi(1,ny1+1)+
     &            S(1,ny1,nx,ny))/3d0
         do i=2,nx-1
            phi(i,ny1) = w1*phi(i,ny1) +
     &         w*(phi(i-1,ny1) + phi(i+1,ny1)
     &            +phi(i,ny1-1) + phi(i,ny1+1)+
     &            S(i,ny1,nx,ny))/4d0
         end do
         phi(nx,ny1) = w1*phi(nx,ny1) +
     &      w*(phi(nx-1,ny1) + phi(nx,ny1-1) + phi(nx,ny1+1) +
     &            S(nx,ny1,nx,ny))/3d0
      end if
C
      do j = ny1+1,ny2-1
         do i=1,nx
C
            if ( i.ne.1.and.i.ne.nx ) then
               phi(i,j) = w1*phi(i,j) +
     &            w *(phi(i-1,j) + phi(i+1,j)
     &               +phi(i,j-1) + phi(i,j+1) +
     &                  S(i,j,nx,ny))/4d0
C
C     x = 1 : neumann condition
C     x = nx : neumann condition
            else if(i.eq.1) then
               phi(1,j) = w1*phi(1,j) +
     &           w*(phi(2,j)+phi(1,j-1)+phi(1,j+1)+
     &              S(1,j,nx,ny))/3d0
```

```
     &
            else if(i.eq.nx) then
              phi(nx,j) = w1*phi(nx,j) +
     &          w*(phi(nx-1,j)+phi(nx,j-1)+phi(nx,j+1)+
     &             S(nx,j,nx,ny))/3d0
            end if
          end do
        end do
C
        if ( ny2.eq.ny ) then
          phi(1,ny) = w1*phi(1,ny) +
     &       w*(phi(2,ny)+phi(1,ny-1)+S(1,ny,nx,ny))/2d0
          do i=2,nx-1
            phi(i,ny) = w1*phi(i,ny) +
     &          w*(phi(i-1,ny)+phi(i+1,ny)+phi(i,ny-1)+
     &             S(i,ny,nx,ny))/3d0
          end do
          phi(nx,ny) = w1*phi(nx,ny) +
     &       w*(phi(nx-1,ny)+phi(nx,ny-1)+S(nx,ny,nx,ny))/2d0
        else
          phi(1,ny2) = w1*phi(1,ny2) +
     &       w*(phi(2,ny2) +phi(1,ny2-1) + phi(1,ny2+1)+
     &          S(1,ny2,nx,ny))/3d0
          do i=2,nx-1
            phi(i,ny2) = w1*phi(i,ny2) +
     &          w*(phi(i-1,ny2) + phi(i+1,ny2)
     &            +phi(i,ny2-1) + phi(i,ny2+1)+
     &               S(i,ny2,nx,ny))/4d0
          end do
          phi(nx,ny2) = w1*phi(nx,ny2) +
     &       w*(phi(nx-1,ny2)+phi(nx,ny2-1)+phi(nx,ny2+1)+
     &          S(nx,ny2,nx,ny))/3d0
        end if
C
        devmax = -1e30
        do j=ny1,ny2
          do i=1,nx
            d = abs(phi2(i,j) - phi(i,j))
            if ( d .gt. devmax ) devmax = d
          end do
        end do
C
        call MPI_Allreduce(devmax,dd,1,MPI_DOUBLE_PRECISION,MPI_MAX,
     &                  MPI_COMM_WORLD, ierr)
C
        if ( dd.gt.1.0e-7.and.lcount.lt.100000 ) go to 100
C
        t2 = MPI_WTime()
        if ( myrank.eq.0 ) then
          write(*,*) '== Converged : deviation max. ',devmax,
     &              ' iteration  ',lcount
          write(*,*) '== Calculation time ',t2-t1
        end if
        return
        end
```

この例ではプロセス間の一対一通信を行なうルーティンが使用されています。

◎ MPI_Send(buffer,number_of_data,data_type,target,tag,communicator,ierr)

配列 buffer に含まれる number_of_data 個のデータを rank が target のプロセスに送ります。tag はメッセージを識別するのに使用するタグで、同じプロセスに複数のメッセージ通信を続けて行う場合に各メッセージを受け側で対応するメッセージを正しく受け取れるように異なった値の tag を与えます。

◎ MPI_Recv(buffer,number_of_data,data_type,source,tag,communicator,
mpistatus,ierr)

配列 buffer に含まれる number_of_data 個のデータを rank が source のプロセスから受けとります。tag はメッセージを識別するのに使用するタグで、対応する MPI_Send と同じ値を指定します。mpistatus は FORTRAN では長さ MPI_STATUS_SIZE の配列で、正常にデータを受け取れたかどうかなどを示します。

◎ MPI_Allreduce(buffer1,buffer2,number_of_data,data_type,operation,
communicator,ierr)

配列 buffer1 に含まれる各データに対して operation で指定される「縮約」操作をおこない配列 buffer2 に格納します。communicator の全てのプロセスが縮約計算の答えを受けとるのが MPI_Reduce と異なる点です。この例では oepration を MPI_MAX としてプロセス間の最大値を求めています。

MPI_Send と MPI_Recv はプロセス毎に分解されたメッシュ群の境界で値を交換するのに使用されています。各プロセスは y が ny1 から ny2 までの計算を担当します。phi(nx,ny1-1:ny2+1)のように y 方向にひとつづつ担当分メッシュより多いメッシュをもっていて、隣のプロセスの境界値を受け取れるようにしています。

ここで対応する MPI_Send と MPI_Recv において、MPI_Recv の方を先に call したら何が起こるでしょうか？MPI_Recv はデータを受け取り終えるまで待たなければならないので、２つの隣接するプロセス間で相手のデータがやってくるのをずっと待ち続ける状態になります。このような状態を「デッド・ロック」(dead lock)とよびます。並列処理ルーチンを使用する際にはデッド・ロックのような状態が発生するのを注意深く避ける必要があります。この例の場合ではこのような問題に気を使わないですむ MPI_Sendrecv というデータ交換専門のルーチンもありますので、それを使うのも手でしょう。せっかくですのでこの例での計算時間の例を表 1 に示します。この計算は Pentium4/2GHz の PC を

100Base-TX のネットワークボードとスイッチングハブで接続したクラスターによるものです。メッシュ数は300×300です。

表１　例題の Poisson ソルバーの計算時間

プロセス数	時間（秒）	収束までの繰り返し数	繰り返し1万回あたりの計算時間
1	368.96	54577	67.6
2	219.01	56200	38.97
3	172.70	57810	29.87
4	147.64	59416	24.84

MPI_Recv がデータを受け取り終えるまで待たなければならないことについて書きましたが、データ受け取り終了を「待たない」MPI_Irecv というサブルーチンもあります。これはデータ受け取りの終了を待つ MPI_Wait というサブルーチンと対になって使用されます。

◎ MPI_Irecv(buffer,number_of_data,data_type,source,tag,communicator,
mpistatus,request,ierr)

MPI_Recv と同様にデータを受け取るルーチンですが、データ受け取り終了を待たないですぐにサブルーティンを抜け出す点が異なります。request は request handle と呼ばれるもので MPI_Irecv によって与えられます。

◎ MPI_Wait(request,mpistatus,ierr)

MPI_Irecv(または MPI_Isend)の終了を待ちます。request は対応する MPI_Irecv などで設定されたハンドルです。

これらを使用した poisson ルーチンを以下に示します。

```
C-----------------------------------------------------------------
      subroutine poisson( nx,ny,ny1,ny2,myrank,nproc,phi,phi2)
      implicit double precision (a-h,o-z)
      include 'mpif.h'
      double precision phi(nx,ny1-1:ny2+1)
      double precision phi2(nx,ny1:ny2)
      integer mpistat(MPI_STATUS_SIZE)
C    ... SOR acceleration factor ....
      w  = 1.9d0
      w1 = 1.0d0 - w
C
```

```
      t1 = MPI_WTime()
      lcount = 0
  100 continue
      lcount = lcount + 1
C ..... send/recieve process boundary values
      if ( ny1.ne.1 ) then
          call MPI_Irecv(phi(1,ny1-1),nx,MPI_DOUBLE_PRECISION,
     &          myrank-1,2,MPI_COMM_WORLD,mpistat,ireq1,ierr)
      end if
      if ( ny2.ne.ny ) then
         call MPI_Irecv(phi(1,ny2+1),nx,MPI_DOUBLE_PRECISION,
     &          myrank+1,1,MPI_COMM_WORLD,mpistat,ireq2,ierr)
      end if

      if ( ny1.ne.1 ) then
         call MPI_Send(phi(1,ny1),nx,MPI_DOUBLE_PRECISION,myrank-1,
     &           1,MPI_COMM_WORLD,ierr)
      end if
      if ( ny2.ne.ny ) then
         call MPI_Send(phi(1,ny2),nx,MPI_DOUBLE_PRECISION,myrank+1,
     &           2,MPI_COMM_WORLD,ierr)
      end if
C
      do j=ny1,ny2
         do i=1,nx
            phi2(i,j) = phi(i,j)
         end do
      end do
C
C
      if ( ny1.eq.1 ) then
C     .... Dirichlet condition.
         do i=1,nx
           phi(i,1) = 0.0
         end do
      end if
C
      do j = ny1+1,ny2-1
         do i=1,nx
C
            if ( i.ne.1.and.i.ne.nx ) then
               phi(i,j) = w1*phi(i,j) +
     &           w *(phi(i-1,j) + phi(i+1,j)
     &              +phi(i,j-1) + phi(i,j+1) +
     &                 S(i,j,nx,ny))/4d0
C
C     x = 1 : neumann condition
C     x = nx : neumann condition
            else if(i.eq.1) then
               phi(1,j) = w1*phi(1,j) +
     &          w*(phi(2,j)+phi(1,j-1)+phi(1,j+1)+
     &             S(1,j,nx,ny))/3d0
     &
            else if(i.eq.nx) then
               phi(nx,j) = w1*phi(nx,j) +
     &          w*(phi(nx-1,j)+phi(nx,j-1)+phi(nx,j+1)+
     &             S(nx,j,nx,ny))/3d0
            end if
         end do
      end do
C
      if ( ny2.eq.ny ) then
```

```
         phi(1,ny) = w1*phi(1,ny) +
     &      w*(phi(2,ny)+phi(1,ny-1)+S(1,ny,nx,ny))/2d0
         do i=2,nx-1
           phi(i,ny) = w1*phi(i,ny) +
     &         w*(phi(i-1,ny)+phi(i+1,ny)+phi(i,ny-1)+
     &            S(i,ny,nx,ny))/3d0
         end do
         phi(nx,ny) = w1*phi(nx,ny) +
     &      w*(phi(nx-1,ny)+phi(nx,ny-1)+S(nx,ny,nx,ny))/2d0
      end if
C
      if( ny1.ne.1 ) call MPI_Wait(ireq1,mpistat,ierr)
      if( ny2.ne.ny ) call MPI_Wait(ireq2,mpistat,ierr)
C
      if( ny1.ne.1 ) then
         phi(1,ny1) = w1*phi(1,ny1) +
     &      w*(phi(2,ny1) + phi(1,ny1-1) + phi(1,ny1+1)+
     &            S(1,ny1,nx,ny))/3d0
         do i=2,nx-1
            phi(i,ny1) = w1*phi(i,ny1) +
     &         w*(phi(i-1,ny1) + phi(i+1,ny1)
     &            +phi(i,ny1-1) + phi(i,ny1+1)+
     &            S(i,ny1,nx,ny))/4d0
         end do
         phi(nx,ny1) = w1*phi(nx,ny1) +
     &      w*(phi(nx-1,ny1) + phi(nx,ny1-1) + phi(nx,ny1+1) +
     &            S(nx,ny1,nx,ny))/3d0
      end if
C
      if ( ny2.ne.ny ) then
         phi(1,ny2) = w1*phi(1,ny2) +
     &      w*(phi(2,ny2) +phi(1,ny2-1) + phi(1,ny2+1)+
     &         S(1,ny2,nx,ny))/3d0
         do i=2,nx-1
            phi(i,ny2) = w1*phi(i,ny2) +
     &         w*(phi(i-1,ny2) + phi(i+1,ny2)
     &           +phi(i,ny2-1) + phi(i,ny2+1)+
     &              S(i,ny2,nx,ny))/4d0
         end do
         phi(nx,ny2) = w1*phi(nx,ny2) +
     &      w*(phi(nx-1,ny2)+phi(nx,ny2-1)+phi(nx,ny2+1)+
     &         S(nx,ny2,nx,ny))/3d0
      end if
C
      devmax = -1e30
      do j=ny1,ny2
         do i=1,nx
            d = abs(phi2(i,j) - phi(i,j))
            if ( d .gt. devmax ) devmax = d
         end do
      end do
C
      call MPI_Allreduce(devmax,dd,1,MPI_DOUBLE_PRECISION,MPI_MAX,
     &                  MPI_COMM_WORLD, ierr)
C
      if ( dd.gt.1.0e-7.and.lcount.lt.100000 ) go to 100
C
      t2 = MPI_WTime()
      if ( myrank.eq.0 ) then
         write(*,*) '== Converged : deviation max. ',devmax,
     &               ' iteration  ',lcount
         write(*,*) '== Calculation time ',t2-t1
```

```
end if
return
end
```

では何のために MPI_Irecv と MPI_Wait を用いたのでしょうか？この例では MPI_Irecv と MPI_Wait の間に、プロセス間境界メッシュ以外でのメッシュでの処理がおかれていることが分かります。つまり MPI_Irecv で指定されたデータ通信と、その通信にかかわるデータを参照しない計算を「同時に」行なうことで MPI_Recv を使用した場合に比べて計算時間を短縮できるということになります。計算時間を表２に示します。表１の計算と同じ環境によるものです。このような通信の方法を「非同期通信」または「ノンブロッキング通信」とよびます。100Mb Ether ボードとスイッチングハブといった安価な通信手段を用いた PC クラスターの場合、通信にかかる時間の負荷は並列処理専用に設計された計算機や Myrinet などの高速の通信手段を用いた場合に比べて大きなものになり、あまり並列計算による高速化の効果が得られない場合があります。しかし非同期通信のような技法で計算時間短縮が行なえる場合もありますので、可能ならば使用してみるべきでしょう。

今回挙げた SOR 法による計算の例では繰り返し計算のメッシュスイープの順序が変わってしまってプロセス数ごとに収束までの繰り返し数が変化してしまい、あまり時間短縮効果はありませんでしたが、単位繰り返しあたりの秒数を見ると効果があることはわかるでしょう。

この例のように並列プロセス数が変わると収束までの繰り返し数が変わってしまうような繰り返し方法は並列化を行う際に問題となります。解法を SOR 法でなくヤコビ法にすれば繰り返し回数はプロセス数と無関係になり、もっと並列化の効果が上がったかもしれません。実際にヤコビ法を用いたベンチマーク計算もあります（http://w3cic.riken.go.jp/HPC/HimenoBMT/）。

表２　例題の Poisson ソルバーの計算時間（非同期通信を用いた場合）

プロセス数	計算時間（秒）	収束までの繰り返し数	繰り返し1万回あたりの計算時間
1	367.57	54577	67.35
2	213.76	59498	35.93
3	161.47	64311	25.11
4	133.79	69068	19.37

４．おわりに

並列計算の入門ということで MPI を用いたプログラミングを紹介しました。並列プログラミングに慣れていない方にはなにやら奇妙で面倒臭いプログラムの書き方に思えたかもしれません。並列処理専用に設計された計算機の場合にはいろいろ並列プログラミングの手段があるでしょうが、PC クラスターでは現在のところ MPI あるいは同様のメッセージ通信ルーチンによる PVM などを用いた並列計算が当分は主流であると思われます。この文章が PC クラスター利用の一助となれば幸いです。

参考文献、ウエッブサイトなど

(1) 湯浅太一、他編 「はじめての並列プログラミング」(共立出版)
(2) MPI Forum： http://www.mpi-forum.org
(3) MPICH A Portable MPI Implementation：
http://www-unix.mcs.anl.gov/mpi/mpich/
(4) 青山幸也 「並列プログラミング虎の巻 MPI 版」（日本 IBM、非売品）
(5) MPI Primer/Developing with LAM：
http://www.lam-mpi.org/download/files/lam61.nol.doc.pdf