An Introduction to MPI Programming

Xing Cai

Department of Informatics
University of Oslo

Contents

Parallel computing

Pursuit of shorter computation time and larger simulation size gives rise to parallel computing.

The essence is to divide the entire computation among collaborative processes.

Distribution/partition:

- Logical: work load

- Algorithm: domain decomposition

- Hardware: distributed memory

The message passing paradigm

Collaboration of separate processes to solve a global problem. Each process has an independent thread of control and separate address space.

Partition of the computational load is enforced by some kind of data partition or domain partition.

Collaboration is achieved by explicit message passing, which is data transfer plus synchronization.

Better parallel efficiency is usually achieved by explicit message passing.

MPI

MPI is a library specification for the *message passing interface*, proposed as a standard.
- Independent of hardware;
- not a language or compiler specification;
- not a specific implementation or product.

A message passing standard for portability and ease-of-use. Primarily for SPMD/MIMD.

Designed for high performance.

Widely available, many implementations.

Two phases:
- MPI-1: Traditional message-passing;
- MPI-2: Remote memory, parallel I/O, dynamic process, threads.

What's MPI-1?

Point-to-point message passing

Collective communication

Support for process groups

Support for communication contexts

Support for process topologies

Environmental inquiry routines

Profiling interface

The 6 most important MPI routines

`MPI_Init` - enter MPI

`MPI_Finalize` - exit MPI

`MPI_Comm_size` - how many processes participate in the present computation?

`MPI_Comm_rank` - which one am I? (A number between `0` and `size-1`.)

`MPI_Send` - send a message to a particular process

`MPI_Recv` - receive a message from a particular process

Communicator

A group of MPI processes with a name (context).

Mechanism to identify subset of processes.

Promotes modular design of parallel libraries.

The rank of any process is only meaningful within a particular communicator.

By default communicator `MPI_COMM_WORLD` contains all the MPI processes.

The first MPI program

Let every process write "Hello world" on the standard output.

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

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I'm No.%d of %d procs.\n",my_rank,size);
  MPI_Finalize ();
  return 0;
}
```


Compilation

```
> gcc -I/usr/lib/mpich/include -c hello.c
> gcc -o hello hello.o -L/usr/local/mpich/lib -lmpi
```

Or

```
> mpicc -c hello.c
> mpicc -o hello hello.o
```

Execution

```
> mpirun -np 4 ./hello
```

Example running result (non-deterministic)

```
Hello world, I'm No.2 of 4 procs.
Hello world, I'm No.1 of 4 procs.
Hello world, I'm No.3 of 4 procs.
Hello world, I'm No.0 of 4 procs.
```


Synchronization

Many algorithms require that none process proceeds before all the processes have reached the same state at certain points of a program.

Explicit synchronization

```
int MPI_Barrier (MPI_Comm comm)
```

Implicit synchronization
through use of e.g. `MPI_Send` and `MPI_Recv` pairs.

Ask yourself the following question: *"If process one progresses 100 times faster than process two, will the final result still be correct?"*

Example: ordered output

Explicit synchronization

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

int main (int nargs, char** args)
{
  int size, my_rank,i;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  for (i=0; i<size; i++) {
    MPI_Barrier (MPI_COMM_WORLD);
    if (i==my_rank) {
      printf("Hello world, I'm No.%d of %d procs.\n",
             my_rank, size);
      fflush (stdout);
    }
  }

  MPI_Finalize ();
  return 0;
}
```

Result

```
Hello world, I'm No.0 of 4 procs.
Hello world, I'm No.1 of 4 procs.
Hello world, I'm No.2 of 4 procs.
Hello world, I'm No.3 of 4 procs.
```

Point-to-point communication

Data: start address of the message buffer, counter of elements in the buffer, data type.

Envelope: source/destination process, message tag, communicator.

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);
```

This blocking send function returns when the data has been delivered to the system and the buffer can be reused. The message may not have been received by the destination process.

Data type:
- Predefined standard types
- Derived types

Point-to-point communication (cont'd)

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status *status);
```

Waits until a matching message is received from the system and the message buffer can be used.

Receiving fewer `datatype` elements than `count` is ok, but receiving more is an error.

`status` is a data structure that contains further information.

Another solution to "ordered output"

Passing semaphore from process to process

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

int main (int nargs, char** args)
{
  int size, my_rank, flag;
  MPI_Status status;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  if (my_rank>0)
    MPI_Recv (&flag, 1, MPI_INT,
              my_rank-1, 100, MPI_COMM_WORLD, &status);

  printf("Hello world, I'm No.%d of %d procs.\n",my_rank,size);

  if (my_rank<size-1)
    MPI_Send (&my_rank, 1, MPI_INT,
              my_rank+1, 100, MPI_COMM_WORLD);

  MPI_Finalize ();
  return 0;
}
```


Example: sum of random numbers

Write an MPI program in which each process generates a random number and then calculates the sum of these numbers.

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

int main (int nargs, char** args)
{
  int size, my_rank, i, a, sum;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  srand (7654321*(my_rank+1));
  a = rand()%100;
  printf("<%02d> a=%d\n",my_rank,a);
  fflush (stdout);

  if (my_rank==0) {
    MPI_Status status;
    sum = a;
    for (i=1; i<size; i++) {
      MPI_Recv (&a, 1, MPI_INT,
                i, 500, MPI_COMM_WORLD, &status);
      sum += a;
    }
    printf("<%02d> sum=%d\n",my_rank,sum);
  }
  else
    MPI_Send (&a, 1, MPI_INT, 0, 500, MPI_COMM_WORLD);

  MPI_Finalize ();
  return 0;
}
```

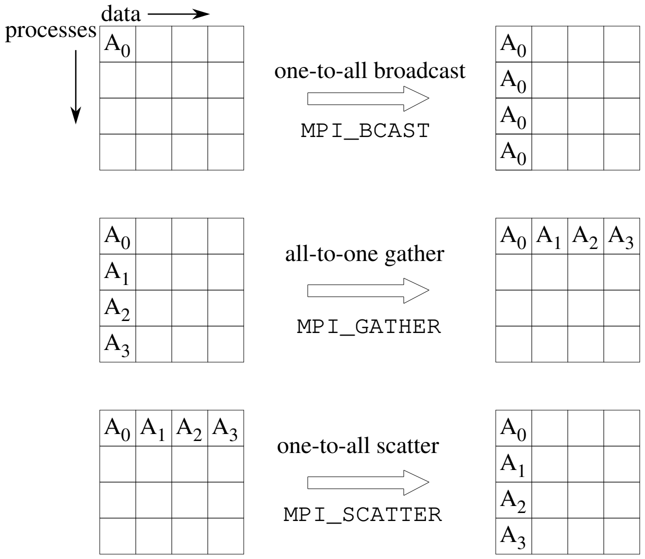
Alternative one

Use of `MPI_Probe`

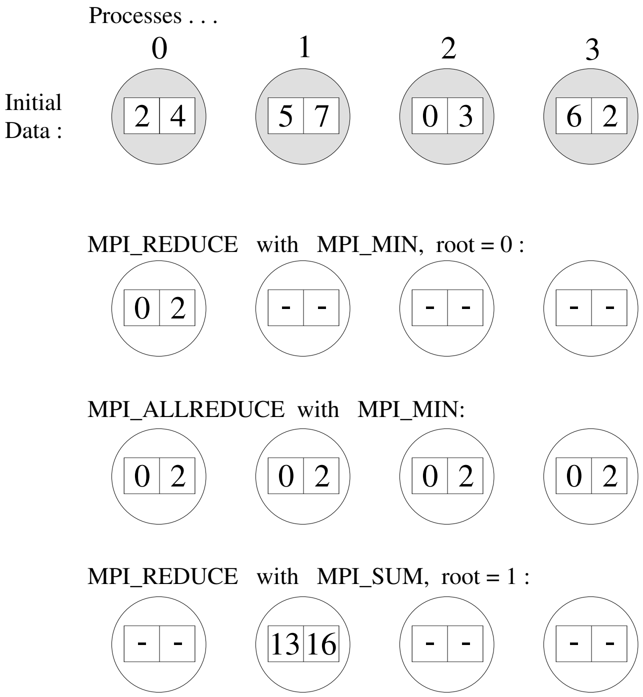
```
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank, i, a, sum;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  srand (7654321*(my_rank+1));
  a = rand()%100;
  printf("<%02d> a=%d\n",my_rank,a);
  fflush (stdout);

  if (my_rank==0) {
    MPI_Status status;
    sum = a;
    for (i=1; i<size; i++) {
      MPI_Probe (MPI_ANY_SOURCE,500,MPI_COMM_WORLD,&status);
      MPI_Recv (&a, 1, MPI_INT,
                status.MPI_SOURCE,500,MPI_COMM_WORLD,&status);
      sum += a;
    }
    printf("<%02d> sum=%d\n",my_rank,sum);
  }
  else
    MPI_Send (&a, 1, MPI_INT, 0, 500, MPI_COMM_WORLD);

  MPI_Finalize ();
  return 0;
}
```


Collective communication

Communication carried out by all processes in a communicator



Collective communication (Cont'd)



Alternative two to "random number sum"

Use of `MPI_Gather`

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

int main (int nargs, char** args)
{
  int size, my_rank, i, a, sum, *array;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  srand (7654321*(my_rank+1));
  a = rand()%100;
  printf("<%02d> a=%d\n",my_rank,a);
  fflush (stdout);

  if (my_rank==0)
    array = (int*) malloc(size*sizeof(int));

  MPI_Gather (&a, 1, MPI_INT, array, 1, MPI_INT,
      0, MPI_COMM_WORLD);

  if (my_rank==0) {
    sum = a;
    for (i=1; i<size; i++)
      sum += array[i];
    printf("<%02d> sum=%d\n",my_rank,sum);
    free (array);
  }

  MPI_Finalize ();
  return 0;
}
```

Alternative three

Use of MPI_Reduce (MPI_Allreduce)

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

int main (int nargs, char** args)
{
  int my_rank, a, sum;
  MPI_Init (&nargs, &args);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  srand (7654321*(my_rank+1));
  a = rand()%100;
  printf("<%02d> a=%d\n",my_rank,a);
  fflush (stdout);

  MPI_Reduce (&a,&sum,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);

  if (my_rank==0)
    printf("<%02d> sum=%d\n",my_rank,sum);

  MPI_Finalize ();
  return 0;
}
```


MPI timer

```
double MPI_Wtime(void)
```

The function returns a number representing the number of wall-clock seconds elasped since some time in the past.

Example:

```
double starttime, endtime;
starttime = MPI_Wtime();
 ....  stuff to be timed  ...
endtime   = MPI_Wtime();
printf("That took %f seconds\n",endtime-starttime);
```


Timing point-to-point communication

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

#define NUMBER_OF_TESTS 10

int main( int argc, char **argv )
{
  double       *buf;
  int          rank;
  int          n;
  double       t1, t2, tmin;
  int          i, j, k, nloop;
  MPI_Status   status;

  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  if (rank == 0)
    printf( "Kind\t\tn\ttime (sec)\tRate (MB/sec)\n" );

  for (n=1; n<1100000; n*=2) {
    nloop = 1000;

    buf = (double *) malloc( n * sizeof(double) );
    if (!buf) {
      fprintf( stderr,
       "Could not allocate send/recv buffer of size %d\n", n );
      MPI_Abort( MPI_COMM_WORLD, 1 );
    }
    tmin = 1000;
    for (k=0; k<NUMBER_OF_TESTS; k++) {
      if (rank == 0) {
        /* Make sure both processes are ready */
        MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 1, 14,
```



```
                      MPI_BOTTOM, 0, MPI_INT, 1, 14, MPI_COMM_WORLD,
                     &status );
        t1 = MPI_Wtime();
        for (j=0; j<nloop; j++) {
          MPI_Send( buf, n, MPI_DOUBLE, 1, k, MPI_COMM_WORLD );
          MPI_Recv( buf, n, MPI_DOUBLE, 1, k, MPI_COMM_WORLD,
                   &status );
        }
        t2 = (MPI_Wtime() - t1) / nloop;
        if (t2 < tmin) tmin = t2;
      }
      else if (rank == 1) {
        /* Make sure both processes are ready */
        MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 0, 14,
                      MPI_BOTTOM, 0, MPI_INT, 0, 14, MPI_COMM_WORLD,
                     &status );
        for (j=0; j<nloop; j++) {
          MPI_Recv( buf, n, MPI_DOUBLE, 0, k, MPI_COMM_WORLD,
                   &status );
          MPI_Send( buf, n, MPI_DOUBLE, 0, k, MPI_COMM_WORLD );
        }
      }
    }
    /* Convert to half the round-trip time */
    tmin = tmin / 2.0;
    if (rank == 0) {
      double rate;
      if (tmin > 0) rate = n * sizeof(double) * 1.0e-6 /tmin;
      else          rate = 0.0;
      printf( "Send/Recv\t%d\t%f\t%f\n", n, tmin, rate );
    }
    free( buf );
  }

  MPI_Finalize( );
  return 0;
}
```

Measurements of `pingpong.c`

```
Kind n time (sec) Rate (MB/sec)
Send/Recv 1 0.000154 0.051985
Send/Recv 2 0.000155 0.103559
Send/Recv 4 0.000158 0.202938
Send/Recv 8 0.000162 0.394915
Send/Recv 16 0.000173 0.739092
Send/Recv 32 0.000193 1.323439
Send/Recv 64 0.000244 2.097787
Send/Recv 128 0.000339 3.018741
Send/Recv 256 0.000473 4.329810
Send/Recv 512 0.000671 6.104322
Send/Recv 1024 0.001056 7.757576
Send/Recv 2048 0.001797 9.114882
Send/Recv 4096 0.003232 10.137046
Send/Recv 8192 0.006121 10.706747
Send/Recv 16384 0.012293 10.662762
Send/Recv 32768 0.024315 10.781164
Send/Recv 65536 0.048755 10.753412
Send/Recv 131072 0.097074 10.801766
Send/Recv 262144 0.194003 10.809867
Send/Recv 524288 0.386721 10.845800
Send/Recv 1048576 0.771487 10.873298
```


Example: inner-product

Write an MPI program that calculates inner-product between two vectors $(u_1, u_2, \ldots, u_m)$ and $(v_1, v_2, \ldots, v_m)$, where

$$u_k = x_k, \quad v_k = x_k^2, \quad x_k = \frac{k}{m}.$$

Partition each of the vectors into P segments (load balancing):

$$m_i = \begin{cases} \lfloor \frac{m}{P} \rfloor + 1 & 1 \le i \le \text{mod}(m, P) \\ \lfloor \frac{m}{P} \rfloor & \text{mod}(m, P) + 1 \le i \le P \end{cases}$$

First find the local result.

Then compute the global result.


```c
#include <stdio.h>
#include <mpi.h>
#include <malloc.h>

int main (int nargs, char** args)
{
  int size, my_rank, i, m = 1000, lm, lower_bound, res;
  double l_sum, g_sum, time, x, dx, *u, *v;

  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  if (my_rank==0 && nargs>1)
    m = atoi(args[1]);

  MPI_Bcast (&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
  dx = 1.0/m;

  time = MPI_Wtime();

  /* data partition & load balancing */
  lm = m/size;
  res = m%size;
  lower_bound = my_rank*lm;
  lower_bound += (my_rank<=res) ? my_rank : res;
  if (my_rank+1<=res)
    ++lm;

  /* allocation of data storage */
  u = (double*) malloc (lm*sizeof(double));
  v = (double*) malloc (lm*sizeof(double));

  /* fill out the u og v-vectors */
  x = lower_bound*dx;
  for (i=0; i<lm; i++) {
```



```c
    x += dx;
    u[i] = x;
    v[i] = x*x;
  }

  /* calculate the local result */
  l_sum = 0.;
  for (i=0; i<lm; i++)
    l_sum += u[i]*v[i];

  MPI_Allreduce (&l_sum,&g_sum,1,MPI_DOUBLE,
                 MPI_SUM,MPI_COMM_WORLD);

  time = MPI_Wtime()-time;

  /* output the global result */
  printf("<%d>m=%d, lower_bound=%d, lm=%d, u*v=%g, time=%g\n",
         my_rank, m, lower_bound, lm, g_sum*dx, time);

  free (u);
  free (v);

  MPI_Finalize ();
  return 0;
}
```

Message passing programming

Keep in mind that:
Multiple processes with own memory co-exist. They need to work collaboratively to solve a common global problem. The collaboration is done by synchronization and data exchange in form of passing messages (with tags) between neighbors.

There are certain non-deterministic features of a parallel simulation:
- The order of incoming messages from neighbors may vary from execution to execution.
- Progress of one process may lag behind others.

Source of deadlocks

Send a large message from one process to another
- If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive).

Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

Unsafe because it depends on the availability of system buffers.

Solutions:
- order the operations more carefully
- Use non-blocking operations

Use of non-blocking and completion routines allow computation and communication to be overlapped. (Not guaranteed, though.)

Non-blocking send and receive

Non-blocking send: returns "immediately"; message buffer should not be written to after return; must check for local completion.

Non-blocking receive: returns "immediately"; message buffer should not be read from after return; must check for local completion.

Two basic ways of checking on non-blocking sends and receives:

- `MPI_Wait` blocks until the communication is complete.

```
MPI_Wait(MPI_Request *request, MPI_Status *status)
```

- `MPI_Test` returns "immediately", and sets flag to true is the communication is complete.

```
MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```


Four different modes of MPI send routines

- standard mode - a send may be initiated even if a matching receive has not been initiated.
- buffered mode - a send may be initiated only if a matching receive has been initiated.
- synchronous mode - The same as standard mode, except the send will not complete until message delivery is guaranteed.
- ready send - Similar to standard mode, but completion is always independent of matching receive, and message may be buffered to ensure this.

Two-point boundary value problem

$$-u''(x) = f(x), \quad 0 < x < 1, \;\; x(0) = x(1) = 0.$$

Uniform 1D grid: $x_0, x_1, \ldots, x_{n+1}$, $\Delta x = \frac{1}{n+1}$.

Finite difference discretization

$$\frac{-u_{i-1} + 2u_i - u_{i+1}}{\Delta x^2} = f_i, \quad 1 \le i \le n.$$

$$\begin{bmatrix} 2 & -1 & & 0 \\ -1 & 2 & -1 & \\ & \ddots & \ddots & \ddots \\ 0 & & -1 & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} = \Delta x^2 \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix}$$

Jacobi iteration

$$u_i^k = \left(\Delta x^2 f_i + u_{n-1}^{k-1} + u_{n+1}^{k-1}\right)/2.$$

Partition of work load

Divide the interior points $x_1, x_2, \ldots, x_n$ among the P processors:

$$x_{i,1}, x_{i,2}, \ldots, x_{i,n_i}.$$

We need two "ghost" boundary nodes $x_{i,0}$ and x_{i,n_i+1}. Those two boundary nodes need to be updated after each Jacobi iteration by receiving data from the neighbors.


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

extern void jacobi_iteration (int my_rank, int size, int n_i,
      double* rhs, double* x_k1, double* x_k);
extern void calc_residual (int my_rank, int size, int n_i,
                           double* rhs, double* x, double* res);
extern double norm (int n_i, double* x);

int main (int nargs, char** args)
{
  int size, my_rank, i, j, n = 99999, n_i, lower_bound;
  double time, base, r_norm, x, dx, *rhs, *res, *x_k, *x_k1;

  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  if (my_rank==0 && nargs>1)
    n = atoi(args[1]);

  MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  dx = 1.0/(n+1);

  time = MPI_Wtime();

  /* data partition & load balancing */
  n_i = n/size;
  i = n%size;
  lower_bound = my_rank*n_i;
  lower_bound += (my_rank<=i) ? my_rank : i;
  if (my_rank+1<=i)
    ++n_i;

  /* allocation of data storage, plus 2 (ghost) boundary points */
```



```
  x_k = (double*) malloc ((n_i+2)*sizeof(double));
  x_k1= (double*) malloc ((n_i+2)*sizeof(double));
  rhs = (double*) malloc ((n_i+2)*sizeof(double));
  res = (double*) malloc ((n_i+2)*sizeof(double));

  /* fill out the rhs og x_k-vectors */
  x = lower_bound*dx;
  for (i=1; i<=n_i; i++) {
    x += dx;
    rhs[i] = dx*dx*M_PI*M_PI*sin(M_PI*x);
    x_k[i] = x_k1[i] = 0.;
  }
  x_k[0]=x_k[n_i+1]=x_k1[0]=x_k1[n_i+1]=0.;

  base = norm(n_i,rhs);
  r_norm = 1.0;
  i = 0;
  while (i<1000 && (r_norm/base)>1.0e-4) {
    ++i;
    jacobi_iteration(my_rank,size,n_i,rhs,x_k1,x_k);
    calc_residual(my_rank,size,n_i,rhs,x_k,res);
    r_norm = norm(n_i,res);
    for (j=0; j<n_i+2; j++)
      x_k1[j] = x_k[j];
  }

  time = MPI_Wtime()-time;

  printf("<%d> n+1=%d, iters=%d, time=%g\n",my_rank,n+1,i,time);

  free (x_k); free (x_k1); free (rhs); free (res);

  MPI_Finalize ();
  return 0;
}
```

```
#include <mpi.h>

void jacobi_iteration (int my_rank, int size, int n_i,
       double* rhs, double* x_k1, double* x_k)
{
  MPI_Status status;
  int i;

  for (i=1; i<=n_i; i++)
    x_k[i] = 0.5*(rhs[i]+x_k1[i-1]+x_k1[i+1]);

  if (my_rank>0) {
    /* receive from left neighbor */
    MPI_Recv (&(x_k[0]),1,MPI_DOUBLE,
      my_rank-1,501,MPI_COMM_WORLD,&status);
    /* send left neighbor */
    MPI_Send (&(x_k[1]),1,MPI_DOUBLE,
      my_rank-1,502,MPI_COMM_WORLD);
  }

  if (my_rank<size-1) {
    /* send to right neighbor */
    MPI_Send (&(x_k[n_i]),1,MPI_DOUBLE,
      my_rank+1,501,MPI_COMM_WORLD);
    /* receive from right neighbor */
    MPI_Recv (&(x_k[n_i+1]),1,MPI_DOUBLE,
      my_rank+1,502,MPI_COMM_WORLD,&status);
  }
}
```



```
#include <mpi.h>

void calc_residual (int my_rank, int size, int n_i,
    double* rhs, double* x, double* res)
{
  /* no communication is necessary */

  int i;
  for (i=1; i<=n_i; i++)
    res[i] = rhs[i]-2.0*x[i]+x[i-1]+x[i+1];
}
```

Note that no communication is necessary, because the latest solution vector is already correctly duplicated between neighboring processes after `jacobi_iteration`.


```
#include <mpi.h>
#include <math.h>

double norm (int n_i, double* x)
{
  double l_sum=0., g_sum;
  int i;

  for (i=1; i<=n_i; i++)
    l_sum += x[i]*x[i];

  MPI_Allreduce (&l_sum,&g_sum,1,MPI_DOUBLE,
                 MPI_SUM,MPI_COMM_WORLD);

  return sqrt(g_sum);
}
```


Using non-blocking send & receive

```
#include <mpi.h>

extern MPI_Request* req;
extern MPI_Status* status;

void jacobi_iteration_nb (int my_rank, int size, int n_i,
  double* rhs, double* x_k1, double* x_k)
{
  int i;

  /* calculate values on the end-points */
  x_k[1] = 0.5*(rhs[1]+x_k1[0]+x_k1[2]);
  x_k[n_i] = 0.5*(rhs[n_i]+x_k1[n_i-1]+x_k1[n_i+1]);

  if (my_rank>0) {
    /* receive from left neighbor */
    MPI_Irecv (&(x_k[0]),1,MPI_DOUBLE,
      my_rank-1,501,MPI_COMM_WORLD,&req[0]);
    /* send left neighbor */
    MPI_Isend (&(x_k[1]),1,MPI_DOUBLE,
      my_rank-1,502,MPI_COMM_WORLD,&req[1]);
  }

  if (my_rank<size-1) {
    /* send to right neighbor */
    MPI_Isend (&(x_k[n_i]),1,MPI_DOUBLE,
      my_rank+1,501,MPI_COMM_WORLD,&req[2]);
    /* receive from right neighbor */
    MPI_Irecv (&(x_k[n_i+1]),1,MPI_DOUBLE,
      my_rank+1,502,MPI_COMM_WORLD,&req[3]);
  }

  /* calculate values on the inner-points */
  for (i=2; i<n_i; i++)
```

```
    x_k[i] = 0.5*(rhs[i]+x_k1[i-1]+x_k1[i+1]);

  if (my_rank>0) {
    MPI_Wait (&req[0],&status[0]);
    MPI_Wait (&req[1],&status[1]);
  }

  if (my_rank<size-1) {
    MPI_Wait (&req[2],&status[2]);
    MPI_Wait (&req[3],&status[3]);
  }
}
```


Some tips on MPI programming

Use predefined MPI reduction operations;

Avoid `MPI_Barrier` if possible;

Make use of local disk to different processors when saving a large amount of data to files;

When computing a global maximum/minimum entity, remember to call `MPI_Reduce` or `MPI_Allreduce` after finding the local maximum/minimum value.

Related MPI web pages

http://www-unix.mcs.anl.gov/mpi/tutorial/

http://www.mpi.nd.edu/mpi_tutorials/

http://www.nas.nasa.gov/Groups/SciCon/Tutorials/MPIintro/toc.html

http://www-unix.mcs.anl.gov/mpi/tutorial/snir/mpi/mpi.htm

http://www.epm.ornl.gov/~walker/mpitutorial/

http://www-unix.mcs.anl.gov/dbpp/