An Introduction to MPI for Computational Mechanics

P.K. Jimack and N. Touheed
Computational PDE Unit
School of Computer Studies
University of Leeds
Leeds LS2 9JT, UK

Abstract: This paper is aimed at people with little or no previous experience of using MPI but who wish to get off to a quick start in parallel programming. This is achieved by only introducing a small subset of the features of MPI: those which are the most useful for a large majority of computational mechanics applications. The paper begins with a brief overview of the distributed memory programming paradigm and a motivation of the need for message passing. It then goes on to introduce some of the main features of parallel algorithms and to discuss their implementation in MPI. These features are all illustrated through elementary example programs which culminate in a simple iterative finite difference solver for Poisson's equation in two dimensions. The paper concludes with a brief indication of some of the further features contained in MPI and a pointer to some more comprehensive introductory material.

1 Introduction

1.1 The distributed memory paradigm

There are two main architectural paradigms associated with parallel computing: distributed memory and shared memory. In the former each processing unit is assumed to have exclusive access to a particular partition of the memory associated with the parallel computing architecture. Typically this memory is located physically alongside the processing unit (although this is not necessarily the case). All data used by a processor on a distributed memory computer must be situated on that processor's partition of the memory. The shared memory paradigm on the other hand assumes that each processor has read and write access to all of the available memory on the parallel computer.

In this paper we assume that the reader wishes to develop their own parallel software under the distributed memory paradigm. Architectures which map directly onto this paradigm include tightly coupled distributed memory parallel computers (such as the Cray T3E or the IBM-SP2) and also networks of individual workstations which are configured to cooperate in a concurrent manner. This does not mean that the paper is not relevant to users of shared memory computers however since the distributed memory programming paradigm is, as the name implies, merely an abstraction of a distributed memory computer which provides a mechanism for writing parallel programs. Such programs may still be executed on shared memory architectures.

The Message Passing Interface (MPI), which is the subject of this paper, provides a standard set of definitions which allow parallel programs to be written under the distributed memory paradigm. These definitions describe a library of over 100 C and Fortran subprograms which are now supported by almost all parallel computer manufacturers. In addition, numerous commercial and public domain implementations of MPI exist which allow clusters of workstations to be used as a single parallel computing system. The effect of this standardization has been to ensure that, for the first time since the advent of parallel computers, software may be written that is truly portable between different parallel architectures.

1.2 Data Locality and Message Passing

As already stated, when working with a distributed memory computer it is necessary to ensure that each processor has its own copy of each data item that is required for each computation that is performed. Often, some or all of this data will depend upon the result of a previous computation which may have been made on one of the other processors. Clearly therefore some mechanism is required for the transfer of copies of data between processors. This is achieved through the use of message passing: whereby each processor is given the ability to send and receive copies of data to and from other processors. This requires the cooperation of each processor that is involved in the communication, whether as a sender or a receiver (or both).

Historically, as different hardware manufacturers developed their own distributed memory parallel computers they also developed and implemented their own message passing libraries. This effectively meant that the applications programmer was tied to a particular computer architecture unless they were prepared to invest the time and effort required to port their code from one to another. This lack of portability lead to a number of public domain message passing libraries being developed, each with their own software implementation built on top of the communications libraries provided by a variety of vendors. The natural progression from this therefore was the definition of a standard message passing library which would be agreeable to users and manufacturers alike. In 1992 an MPI Forum was established, encompassing hardware and software vendors and researchers from academia, industry and U.S. government, to address this issue. By 1994 an initial MPI standard was published [4] and since that time many efficient implementations have been released for all types of parallel architecture.

1.3 Overview of paper

Having introduced the notions of distributed memory computing and message passing the rest of the paper now concentrates entirely on introducing the Message Passing Interface (MPI) standard. Only certain aspects of this standard are described: these being those deemed by us to be the most useful to readers who may be starting out in parallel programming for applications in the general area of computational mechanics. Nevertheless, the final section of the paper is used to outline some further features of MPI and to provide references to more comprehensive guides than this.

All of the MPI subprograms that we describe are illustrated through the use of simple example programs which are included within the text. These range in difficulty from the summation of a series in parallel to the solution of a linear partial differential equation. Each of the programs is implemented in C and throughout the main body of the paper we describe only the C language binding for MPI. Fortran programmers should not be concerned by this however since the Fortran language binding is almost identical to that for C and the minor differences are clearly outlined in the paper's appendix. Readers who are interested in these differences should consult the relevant section of the appendix once they have completed that section of the paper itself.

Throughout all of our examples we will not make any assumptions on the number of parallel processors that are available to the user (other than that it is greater than or equal to one of course!) but we will assume that each processor being used is running its own copy of the same program. This *single program* model is not the only way in which to program using MPI (for example different processors could be running different programs which cooperate on a given task) but, in the opinion of the authors, it is usually the best. Finally, for the purposes of this paper, we will always assume that each processor has just one of our concurrent processes running on it.

2 Getting Started with MPI

In this section we see how to write a very simple program which uses some MPI functions but does not involve any explicit message passing between processes. As with each of the next five sections we begin with a program listing and then go on to explain the details.

2.1 Sample program

The following program simply enquires as to the number of processes which are running this job in parallel and finds out which process this copy is being run as. The copy running as process 0 then prints to standard output details of the number of processes being used.

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char** argv)
```

```
{
   int noprocs, nid;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &noprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &nid);

   if (nid == 0)
      printf("Hello from processor %i of %i \n",nid,noprocs);

   MPI_Finalize();
}
```

2.2 Further explanation

In order to execute any parallel program which uses MPI a number of essential features must be present in the source code.

- Before the start of the program the file "mpi.h" should be included. This contains a number of predefined constants and data types which are used by MPI functions, the most important of which will be introduced in this paper.

- In addition the program must make a call to the function MPI_Init. This should be the first MPI function that is called and may only be called once. In the C version of the MPI binding the two arguments that were passed to main, *argc* and *argv*, are also accepted by MPI_Init.

- Upon successful completion of all of the communication within the program the function MPI_Finalize should be called. This should be the last MPI function called by the program since no other MPI routines may be executed after this.

Notice that the above example provides no indication of how to begin the execution of the parallel program since this is outside of the scope of MPI and will vary from one hardware platform to another. The example program does introduce two further MPI functions and a predefined MPI constant however. We begin with a brief explanation of the constant MPI_COMM_WORLD.

In MPI the important notion of a "communicator" is widely used. A communicator may be thought of as a communication domain which, amongst other things, defines a set of processes which can communicate with each other. The MPI constant MPI_COMM_WORLD is a predefined communicator which consists of all of the processes which have been launched. As we will see, most MPI functions require a communicator as one of their parameters and for each of the examples in this paper that communicator will be MPI_COMM_WORLD.

From this brief description of communicators it is clear that associated with each communicator are a group of processes (which can communicate with each other). It is also the case that the processes in this group are ranked by MPI from zero up to *noprocs*-1 (where *noprocs* is the number of processes). The MPI functions MPI_Comm_size and MPI_Comm_rank may be used to determine the number of processes in a given group and the rank of each process within that group. Each of these functions has just two parameters: a communicator and a pointer to an integer. The second parameter returns the number of processes in the group associated with the communicator and the rank of the process within the group respectively. Note that since the communicator used in the above example program is MPI_COMM_WORLD, the function MPI_Comm_size determines the total number of processes whilst MPI_Comm_rank determines the overall rank of each process.

Before leaving this introductory program it is worth passing comment on the way in which i/o has been performed. The current definition of MPI does not provide any functions to support parallel i/o and so programmers must account for this themselves. When the amount of input and output is quite small, as in this example, it is usual to perform the i/o sequentially on a single processor by making use of MPI_Comm_rank. There are occasions when the quantity of i/o might make this sequential approach unattractive however: this is considered further in section 6 below.

3 One-to-All and All-to-One Communication

We now move on to look at two types of message passing that are available under MPI: where one process sends data to every other process in a communicator (a broadcast), and where each process in a communicator sends data to a single process (a reduction).

3.1 Sample program

This example program estimates the natural logarithm of 2 by calculating a partial sum of the series $\sum_{i=1}^{\infty}(-1)^{i-1}\frac{1}{i}$, which may also be written as $\sum_{i=0}^{\infty}(-1)^{i}\frac{1}{i+1}$.

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char** argv)
{
   int i, N, noprocs, nid;
   float  sum = 0, Gsum;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &nid);
   MPI_Comm_size(MPI_COMM_WORLD, &noprocs);

   if(nid == 0){
      printf("Please enter the number of terms N -> ");
      scanf("%d",&N);
   }
   MPI_Bcast(&N,1,MPI_INT,0,MPI_COMM_WORLD);
```

```
   for(i = nid; i < N; i += noprocs)
      if(i % 2)
         sum -= (float) 1 / (i + 1);
      else
         sum += (float) 1 / (i + 1);
   MPI_Reduce(&sum,&Gsum,1,MPI_FLOAT,MPI_SUM,0,MPI_COMM_WORLD);
   if(nid == 0)
      printf("An estimate of ln(2) is %f \n",Gsum);
   MPI_Finalize();
}
```

3.2 Further explanation

As in the previous section the program includes "mpi.h" and begins with calls to MPI_Init, MPI_Comm_rank and MPI_Comm_size. Also, all of the input and output is handled by the process whose rank is zero. Once this process has obtained from the user the number of terms required in the partial sum this value is then broadcast to all of the other processes involved in the task of calculating the sum. This is achieved by a call to the function MPI_Bcast which is made by every process.

The function MPI_Bcast has 5 parameters as follows:

1. the starting address of the memory location which stores the data being broadcast,

2. the number of items of data being broadcast,

3. the data type of each item of data being broadcast,

4. the rank of the processor within the communicator which is doing the broadcasting,

5. the communicator across which the broadcast is to be made.

In our example the first parameter is the address of the memory location which is used to store the number of terms in the partial sum and the second and third parameters are just 1 and MPI_INT respectively since only a single integer is being broadcast. It is the process with rank 0 in MPI_COMM_WORLD which is the root of the broadcast and so these are the fourth and fifth parameters respectively. Note that there is an MPI data type (parameter three) corresponding to all of the standard C data types: MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED and MPI_UNSIGNED_LONG. There is also a data type called MPI_PACKED which allows a number of different types to be packed together but this is beyond the scope of this paper.

Upon successful completion of the broadcast each process is aware of the number of terms in the partial sum and is then able to compute a contribution to this sum independently. In our example each process only adds every p^{th} term, where p is the number of processes (*noprocs* in the program), so as to ensure that the load is equally balanced across the processors. If each processor has the same performance then each process should complete the task of calculating its contribution to the sum at about the same time. These contributions (which are distributed across the processors) must then be added together in order to obtain the final answer. This is where the MPI function MPI_Reduce is called.

MPI_Reduce has seven parameters:

1. the address of the memory location storing the data being sent,
2. the address of the memory location storing the received data after the reduction operation (only relevant to one of the processors),
3. the number of data items being sent,
4. the type of each data item being sent,
5. the operation to be performed on the received data,
6. the rank of the processor which is to receive the data,
7. the communicator across which the reduction is to be made.

In our example the first parameter is the address of the contribution to the sum from each process and the second parameter is the address of the location for storing the overall sum. Only 1 item of type MPI_FLOAT is being summed across the processes and so these are the next two parameters. The fifth parameter is MPI_SUM to indicate the nature of the reduction operation (i.e. we wish to sum together the contributions found by each process) and the last two parameters indicate that we wish to accumulate the sum on the process whose rank is 0 in the communicator MPI_COMM_WORLD (i.e. we wish to sum over all of the processes).

It should be noted that MPI_Reduce may also be used to perform different reduction operations by altering the fifth parameter from MPI_SUM. Other possibilities include MPI_PROD, MPI_MAX, MPI_MIN and many more.

Returning to the example; because the sum was accumulated on the process whose rank is zero this must be the process which is used to output the final answer before MPI_Finalize is called. (In fact MPI_Finalize will be called on all processes except that with rank 0 at the same time as the final answer is being output.) Had it been necessary for each process to be aware of the final answer then MPI_Reduce would not have been the appropriate function to have called: instead there is a related function called MPI_Allreduce. This has the same action as MPI_Reduce except that the result appears in the memory address given by the second parameter for all of the processes in the communicator. Also, there is one less parameter in MPI_Allreduce since there is no need to indicate the rank of the process which receives the data (see section 5 for an example of the use of this function).

As a final note to this section it should be pointed out that the way in which the sum has been accumulated in the example program is highly susceptible to the effects of rounding errors due to the fact that the largest terms have been added first on each process. A better implementation would involve each process summing backwards rather than forwards, so as to minimize the effects of rounding error by adding the smallest terms first. Such an implementation is left as an exercise for the reader.

4 Point-to-Point Communication

Each of the communication functions introduced so far involve all of the processes in a communicator, as either senders or receivers. Often however it is necessary to pass data directly from one process to another without involving any of the other processes within the communication domain. This is referred to as point-to-point communication and, in its simplest form, is supported by the two functions MPI_Send and MPI_Recv.

4.1 Sample program

This program calculates the scalar product of two vectors which are contained in a data file. It is assumed that the length of the vectors is an exact multiple of the number of processes being used since this significantly simplifies the task of distributing the vectors evenly across the processes.

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char** argv)
{
   int noprocs, nid, i, n, size;
   float *a, *b, sum = 0.0, Gsum;
   FILE *fp;
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &nid);
   MPI_Comm_size(MPI_COMM_WORLD, &noprocs);
   if(nid == 0){
      fp = fopen("DotData.Txt","rt");
      fscanf(fp,"%d",&n);
      MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
      if(n % noprocs){
         printf("Number of processes is not a multiple of n.\n");
         MPI_Abort(MPI_COMM_WORLD,-1);
      }
      a = (float *) calloc(n,sizeof(float));
      b = (float *) calloc(n,sizeof(float));
      for(i = 0; i < n; i++)
```

```
      fscanf(fp,"%f %f",&a[i],&b[i]);
    fclose(fp);
    for(i = 1, size = n / noprocs; i < noprocs; i++){
      MPI_Send(&a[size*i],size,MPI_FLOAT,i,10,MPI_COMM_WORLD);
      MPI_Send(&b[size*i],size,MPI_FLOAT,i,20,MPI_COMM_WORLD);
    }
  }
  else{
    MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
    if(n % noprocs){
      printf("Number of processes is not a multiple of n.\n");
      MPI_Abort(MPI_COMM_WORLD,-1);
    }
    size =  n / noprocs;
    a = (float *) calloc(size,sizeof(float));
    b = (float *) calloc(size,sizeof(float));
    MPI_Recv(&a[0],size,MPI_FLOAT,0,10,MPI_COMM_WORLD,&status);
    MPI_Recv(&b[0],size,MPI_FLOAT,0,20,MPI_COMM_WORLD,&status);
  }
  for(i = 0; i < size; i++)
    sum += a[i] * b[i];
  MPI_Reduce(&sum,&Gsum,1,MPI_FLOAT,MPI_SUM,0,MPI_COMM_WORLD);
  if(nid == 0)
    printf("The inner product is %f \n",Gsum);
  MPI_Finalize();
}
```

4.2 Further explanation

It is intended that the above program be quite straightforward to follow. The process with rank zero reads in the data from a file, starting with the number of entries in each of the arrays. This number is then broadcast to all of the processes in MPI_COMM_WORLD so as to allow them to allocate a sufficient amount of memory to receive an equal share of each of the vectors. Once this is complete process 0, having read in and stored both vectors, distributes them equally across the processes. This is done by dividing each vector into blocks of length $n/noprocs$ and sending the corresponding block from each vector to a different process using the MPI function MPI_Send. The processes receive these blocks using the MPI function MPI_Recv. Each process then computes its own contribution to the scalar product concurrently before MPI_Reduce is used to sum these contributions in order to obtain a final value.

MPI_Send has six parameters:

1. the starting address of the data to be sent,
2. the number of items of data being sent,
3. the data type of each item of data being sent,

4. the rank of the destination process,
5. an integer tag to allow the message to be labeled,
6. a communicator.

On the other hand, MPI_Recv has seven parameters:

1. the starting address of the memory location storing the received data,
2. the maximum number of data items to be received,
3. the data type of each item of data being received,
4. the rank of the sending process,
5. the tag that should be associated with the incoming message,
6. a communicator,
7. a return status.

In the above example two messages are sent to each process and so a different tag has been used to allow them to be distinguished from each other. By referring to this tag explicitly in MPI_Recv the receiving process can be sure to store the right data in the right variable. Often however, a receiving process does not know nor care about the tag associated with an incoming message, in which case the MPI constant called MPI_ANY_TAG should be used as the fifth parameter. Similarly there is another predefined constant called MPI_ANY_RANK which may be used as the fourth parameter in MPI_Recv if the rank of the sending process is unimportant. The final parameter for this function is of the predefined type MPI_Status which is a structure with three fields: MPI_SOURCE, MPI_TAG and MPI_ERROR. The first two of these fields contain the actual source rank and tag for the received message (these may be useful if MPI_ANY_RANK and/or MPI_ANY_TAG have been used), whilst the third field contains an error code.

It is important to understand that the functions MPI_Send and MPI_Recv allow what is known as "blocked" communication. This notion of blocking ensures that any changes in the state of the calling process which result from this call must occur before the call returns. In practice this means that a process only returns from MPI_Send once all of the details of the message have been copied and successfully prepared for transmission (so that the variables used in the call may be safely overwritten), and a process will only return from MPI_Recv once the incoming message has been completely received and stored in the correct memory location.

Notice that in the example program in this section the work is distributed across the processes in blocks (i.e. each process receives a contiguous block of each of the two vectors). This is necessary since the functions MPI_Send and MPI_Recv are designed to pass data which lies in contiguous memory locations. A consequence of this is that, for simplicity, we have assumed that the size of the vectors is an exact multiple of the number of processes so as to ensure that the size of each block is identical. This should be contrasted with the example used in section 3 where the work of calculating the sum is distributed across the processes in a cyclic manner. In that example it does not matter if the total number of terms in the sum is not an exact multiple of the number of processes since the cyclic nature of the algorithm ensures that, to within one term, each process gets the same computational load.

In order to allow the premature termination of a parallel program MPI provides the function MPI_Abort. This has been used in the scalar product program when it is discovered that the length of the vectors, n, is not an exact multiple of the number of processes, $noprocs$. MPI_Abort has two parameters:

1. the communicator of the tasks to abort,

2. an error code to return to the invoking environment.

In practice this function should always be called with MPI_COMM_WORLD as its first parameter. It will then make a "best attempt" to abort all of the processes: for additional reliability however it is best to get each process in the group to abort where possible (as in our example program). How the error code is handled by the operating system is not defined by MPI and will depend upon that operating system.

Of course making a call to MPI_Abort just because the length of the input data is not a multiple of the number of processes is hardly satisfactory, and any practical parallel program should be considerably more robust than this example. There are numerous possible modifications to this code which will allow vectors of arbitrary length to be split into $noprocs$ blocks of about the same size. For example each vector could be padded with zeros until their length becomes a multiple of $noprocs$. Alternatively some of the blocks could be one item longer than the others but with their total length being equal to n. The implementation of one of these solutions is left as an exercise for the reader.

A final point to note from this section is that, in general, there is an overhead associated with sending each individual message. In practice this means that it is generally more efficient to send a single message of length $2 * size$ than two messages of length $size$. Hence a further improvement that the reader should consider making to our example program would be for process 0 to read in both vectors into a single array and then partition this array into blocks of length $2 * size$. Provided the entries of the two vectors are stored as alternating entries in the array the modified program should only require a single call to both MPI_Send and MPI_Recv.

5 Non-Blocking Communication and a Simple Poisson Equation Solver

The final example that we introduce in this short paper illustrates a parallel finite difference solver for Poisson's equation in two dimensions. Although not particularly complex this is comfortably the most demanding of the examples we consider and, amongst other things, introduces the reader to non-blocked communication and simple parallel i/o.

5.1 Sample program

The following program attempts to solve the equation

$$\nabla^2 u = f \qquad \text{in the domain } \Omega = (0,1) \times (0,1),$$

subject to the Dirichlet boundary condition $u = g$ on $\partial\Omega$. This is achieved by dividing Ω into $N \times N$ square cells and applying a central difference formula at each of the $(N-1) \times (N-1)$ interior vertices. For the vertex in the i^{th} row and the j^{th} column of the grid this gives the formula

$$u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - 4u[i][j] = h^2 f[i][j]$$

for i and j in $\{1, ..., N-1\}$, where h is the length of an edge of each cell and $f[i][j]$ is the value of f at this vertex. Application of Jacobi iteration to this system of equations yields the iterative formula

$$new[i][j] = (old[i+1][j] + old[i-1][j] + old[i][j+1] + old[i][j-1] - h^2 f[i][j])/4$$

for i and j in $\{1, ..., N-1\}$, where $new[i][j]$ is the approximation to $u[i][j]$ after the iteration and $old[i][j]$ is the approximation to $u[i][j]$ before the iteration. Note that whenever $old[i][j]$ is required for a value of i or j in $\{0, N\}$ then this is given by the boundary condition $u = g$.

For simplicity the following example code takes $f \equiv 0$ and $g \equiv 1$ but more general functions could easily be accommodated. In addition, the values of N and the convergence tolerance, Tol, are predefined. Parallel solution is achieved by partitioning the rows of vertices for i from 1 to $N-1$ inclusive across the processes in blocks of equal size (where possible). This is illustrated in figure 1 where $N = 9$ and the number of processes is 3. Hence each process is responsible for solving the iterative equations for exactly 3 rows of unknowns. As is also illustrated by this figure it is also necessary for each process to have access to the old solution values for the rows immediately above and below its own rows of unknowns. Hence at the end of every iteration some point-to-point communication is required.

- With the exception of the process solving for the top block of rows (i.e. with rank *noprocs*-1), each process must send a copy of the latest solution values for its top row (row *size*-1) to the process above it (i.e. with rank *nid*+1) and receive a copy of the latest solution values for the row above its top row (row *size*) from that process (i.e. with rank *nid*+1).

- With the exception of the process solving for the bottom block of rows (i.e. with rank 0), each process must send a copy of the latest solution values for its bottom row (row 1) to the process below it (i.e. with rank *nid*-1) and receive a copy of the latest solution values for the row below its bottom row (row 0) from that process (i.e. with rank *nid*-1).

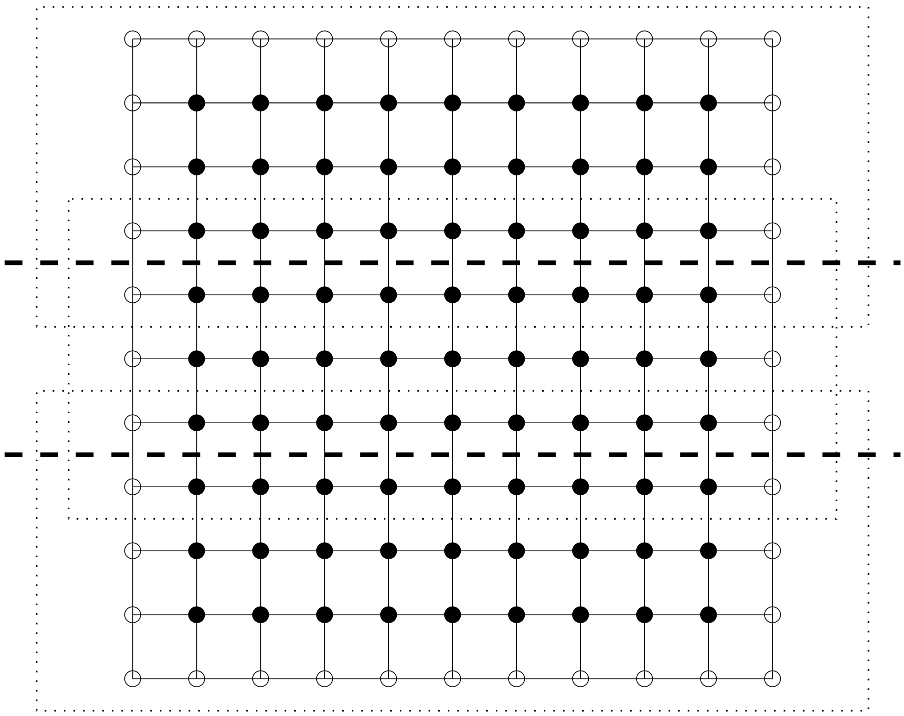

Figure 1: Partition of a finite difference mesh across 3 processes.

The processors solving for the top and bottom blocks have access to the rows immediately above and below their top and bottom rows respectively through the boundary condition $u = 1$.

```
#include <stdio.h>
#include "mpi.h"
#define N  100
#define Tol  0.00001

float **matrix(int m, int n)
{   int i;
    float **ptr;
    ptr = (float **)calloc(m, sizeof(float *));
    for(i = 0; i < m ;i++)
       ptr[i]=(float *)calloc(n, sizeof(float));
    return (ptr);
}

float iteration(float **old, float **new, int start, int finish)
{   float diff, maxerr = 0;
    int i, j;
    for(i = start; i < finish; i++)
       for(j = 1; j < N; j++){
          new[i][j] = 0.25*(old[i+1][j] + old[i-1][j] +
                            old[i][j+1] + old[i][j-1]);
```

```
         diff = new[i][j] - old[i][j];
         if(diff < 0)
            diff = -diff;
         if(maxerr < diff)
            maxerr = diff;
      }
   return (maxerr);
}

main(int argc, char** argv)
{  float **new, **old, **tmp, maxerr, err, maxerrG;
   int noprocs, nid, remainder, size, i, j;
   char str[20];
   FILE *fp;
   MPI_Status status;
   MPI_Request req_send10, req_send20, req_recv10, req_recv20;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &nid);
   MPI_Comm_size(MPI_COMM_WORLD, &noprocs);
   remainder = (N - 1) % noprocs;
   size = (N - 1 - remainder)/noprocs;
   if(nid < remainder)
      size = size + 2;
   else
      size = size + 1;
   new = matrix(size+1,N+1);
   old = matrix(size+1,N+1);
   for(i = 0; i < size + 1; i++)
      new[i][0] = new[i][N] = old[i][0] = old[i][N] = 1;
   if(nid == 0)
      for(j = 1; j < N; j++)
         new[0][j] = old[0][j] = 1;
   if(nid == noprocs - 1)
      for(j = 1; j < N; j++)
         new[size][j] = old[size][j] = 1;
   maxerr = iteration(old,new,1,size);
   MPI_Allreduce(&maxerr,&maxerrG,1,MPI_FLOAT,MPI_MAX,
                 MPI_COMM_WORLD);
   while(maxerrG > Tol){
      tmp = new;
      new = old;
      old = tmp;
      req_send10 = req_recv20 = MPI_REQUEST_NULL;
      if(nid < noprocs-1){
         MPI_Isend(&old[size-1][1],N-1,MPI_FLOAT,nid+1,10,
                   MPI_COMM_WORLD,&req_send10);
```

```
         MPI_Irecv(&old[size][1],N-1,MPI_FLOAT,nid+1,20,
                   MPI_COMM_WORLD,&req_recv20);
      }
      req_send20 = req_recv10 = MPI_REQUEST_NULL;
      if(nid > 0){
         MPI_Isend(&old[1][1],N-1,MPI_FLOAT,nid-1,20,
                   MPI_COMM_WORLD,&req_send20);
         MPI_Irecv(&old[0][1],N-1,MPI_FLOAT,nid-1,10,
                   MPI_COMM_WORLD,&req_recv10);
      }
      maxerr = iteration(old,new,2,size-1);
      if(nid < noprocs-1)
         MPI_Wait(&req_recv20,&status);
      err = iteration(old,new,size-1,size);
      if(err > maxerr)
         maxerr = err;
      if(nid > 0)
         MPI_Wait(&req_recv10,&status);
      err = iteration(old,new,1,2);
      if(err > maxerr)
         maxerr = err;
      MPI_Allreduce(&maxerr,&maxerrG,1,MPI_FLOAT,MPI_MAX,
                    MPI_COMM_WORLD);
   }
   sprintf(str,"Solution%d.Txt",nid);
   fp = fopen(str,"wt");
   if(nid == 0)
      for(j = 0; j < N + 1; j++)
         fprintf(fp,"%6.4f\n",new[0][j]);
   for(i = 1; i < size; i++)
      for(j = 0; j < N + 1; j++)
         fprintf(fp,"%6.4f\n",new[i][j]);
   if(nid == noprocs - 1)
      for(j = 0; j < N + 1; j++)
         fprintf(fp,"%6.4f\n",new[size][j]);
   fclose(fp);
   MPI_Finalize();
}
```

5.2 Further explanation

The function "matrix" is used by each process to allocate some local memory in order to store an $m \times n$ array of floats. Note that the use of "calloc" ensures that each entry of this array is initialized to zero. The function "iteration" is used by each process to apply the Jacobi iteration formula to rows $start$ through to $finish$-1 of its block of rows. It returns the value of the maximum absolute difference between the new and the old estimates of the solution over each point that has been updated.

The first thing that the main program has to compute is the size of the block of rows that are to be dealt with by that process. If the total number of rows of unknowns (N-1) is a multiple of the number of processes (*noprocs*) then each process will have the same number of rows of unknowns in its block, otherwise some processes will have one more row of unknowns than others. In either case the total number of rows of unknowns on a given process is equal to *size*-1 (with each row containing N-1 unknowns). Hence two arrays of dimension *size*+1 by N+1 are dynamically allocated (since the Jacobi iteration formula requires extra rows and columns around the block of unknowns). The boundary condition is then used to allocate values to the 0^{th} and N^{th} columns of each block and to the 0^{th} row of the bottom block (process 0) and to row *size* of the top block (process *noprocs*-1). Following this an initial Jacobi iteration is then taken for each unknown in the block.

In this example the convergence criterion that is used is quite typical: that the maximum difference between the solution estimates before and after the last iteration over every single point in the domain should be less than some tolerance. In order to determine this maximum difference a global communication is required since each process only calculates the maximum difference over its own block of rows. MPI_Allreduce is the appropriate MPI function to use here since, on return, each process needs to know the overall maximum difference in order to determine whether or not the iteration has converged (see section 3 for an introduction to MPI_Allreduce).

Assuming that convergence has not yet occurred the program then enters the while loop in order to take a further Jacobi iteration. This loop begins by copying the contents of the array *new*, which saves the latest solution estimates, into the array *old*, in preparation for the iteration. In C this is most efficiently achieved by swapping pointers to the arrays rather than actually copying the contents of one to another. We have not yet completely updated the array *old* however: for processes with ranks 0 through to *noprocs*-2 the latest solution estimates for row *size* are required, and for processes with ranks 1 through to *noprocs*-1 the latest solution estimates for row 0 are required. These are obtained through the use of point-to-point communication.

In principal one could use the blocking communication routines MPI_Send and MPI_Recv to send and receive copies of these rows of values. However there are two possible drawbacks with this.

- The first drawback is quite subtle but also very important. The implementation of a blocking send may take one of two forms. Either the send will block until a corresponding receive has been posted so that the message can be delivered to the receiver or, more usually, the message to be sent will be placed into a temporary buffer to allow later delivery and the blocking send to complete. The latter option is clearly preferable since the send operation does not necessarily need to block until a corresponding receive has been posted. Nevertheless, if the amount of available buffer space is not sufficient to store a copy of the outgoing message then MPI_Send will be forced to block until either buffer space becomes available or the corresponding MPI_Recv is posted. Now, if every process in a task wishes to send a message to the process ranked above it and to receive a message from the process ranked below it then there is a danger of deadlock occurring if their is insufficient buffer space available when blocking communication is attempted. This is because it is conceivable that every process could end up blocking on its own send because no other process has yet returned from its blocking send in order to post its receive! In this rather small example such an occurrence of deadlock would be extremely unlikely (unless the implementation of MPI was very poor) but it is always an important issue to consider when designing parallel programs.

- The second drawback relates to the issue of parallel efficiency. When using MPI_Recv the processor running the process will be essentially idle whilst the message is being received (or its arrival is being awaited). This is clearly undesirable from an efficiency point of view if there is useful work that the process could be getting on with whilst it is waiting for an incoming message and whilst this incoming message is being buffered. In our finite difference example it is only the top and bottom row of unknowns in each block that require data to be received from other processes in order to take the next Jacobi iteration. All of the other rows in each block may be updated independently of this information. Hence they may be updated at the same time as the point-to-point communications are taking place. This idea is known as "overlapping" communication with computation and may be achieved through the use of non-blocking receives.

Note that the first of these two points motivates the use of non-blocking sends in our example and the second motivates the use of non-blocking receives. This is what has been achieved through the use of the MPI functions MPI_Isend and MPI_Irecv respectively. Each of these functions returns before their respective send and receive operation has been completed, thus allowing the program to continue. (As we will see this means that additional MPI functions are also required to test whether or not these functions have completed at some later point in the code.)

The function MPI_Isend has seven parameters:

1. the starting address of the data to be sent,
2. the number of items of data being sent,
3. the data type of each item of data being sent,
4. the rank of the destination process,
5. an integer tag to allow the message to be labeled,
6. a communicator,
7. a request handle to identify the send request so that it may be checked for completion at a later time.

The function MPI_Irecv also has seven parameters:

1. the starting address of the memory location storing the received data,
2. the maximum number of data items to be received,
3. the data type of each item of data being received,
4. the rank of the sending process,
5. the tag that should be associated with the incoming message,
6. a communicator,
7. a request handle to identify the receive request so that it may be checked for completion at a later time.

Note that in the code a different tag has been used to identify when the top row of a block is being sent (tag is 10) from when the bottom row is being sent (tag is 20). Upon receipt therefore a tag of 10 represents the row below the bottom row of unknowns in the block (row 0) whilst a tag of 20 represents the row above the top row of unknowns in the block (row $size$). For both the MPI_Isend and MPI_Irecv the final parameter is a request handle that is set by the function (and so must be a pointer) to give that communication request a unique identifier. The value of this parameter on input to these functions is unimportant but we choose to ensure that its value on entry is always equal to the predefined MPI constant MPI_REQUEST_NULL which is of type MPI_Request.

Having initiated the non-blocking communications required by each process the example program then goes on to complete all of the computations which are independent of these communications. This involves performing a Jacobi iteration for each unknown in rows 2 to $size$-2 of the block (if $size$ is less than 4 then $maxerr$ is merely set to zero). Before the top row of the block (row $size$-1) can be updated it is necessary to check that the row above this has been successfully received from the process above (unless $nid == noprocs - 1$ in which case the row above is the top boundary). This is achieved through the use of the MPI function MPI_Wait. This function simply blocks until the message in question has been successfully received. Similarly, before the bottom row of the block (row 1) can be updated MPI_Wait is used to ensure that the row below it has been successfully received (unless $nid == 0$ in which case the row below is the bottom boundary). MPI_Wait has just two parameters:

1. a request handle to identify the request to be checked,
2. a return status.

On successful exit the request handle is given the value MPI_REQUEST_NULL and the return status contains information on the completed operation.

Once all of the rows have been successfully updated MPI_Allreduce is again used to calculate the maximum difference between the old and new values at any point on the entire finite difference grid. Once this is less than the convergence tolerance the calculation is complete and it is time to output the computed solution. Unlike in the previous examples this program attempts to perform parallel output by getting each process to write its own contribution to the solution to its own unique file concurrently. The rank of the process is used to ensure that the output file has a unique name. It should be noted however that whether or not this output is actually completed in parallel (as opposed to one process at a time) will depend upon the particular hardware platform that is being used.

We now finish this section with a couple of further comments. Firstly, as well as the MPI function MPI_Wait, there is another MPI function called MPI_Test which can be used to identify when a non-blocking communication request is complete. This does not block until the request has completed successfully; instead it returns a parameter which is true if this is the case and false otherwise. This means that it is possible to enquire as to whether a non-blocking communication has completed without actually having to wait for its completion. MPI_Test has three parameters:

1. a request handle to identify the request to be checked,
2. a flag which is true on exit if the operation has completed,
3. a return status.

Finally it is worth noting that the way in which our example program has been written it will work just as happily with one process as with many. In addition it will also work properly if there are less than three rows of unknowns per block. Whilst this may seem to be an obvious requirement for any parallel program it is always good practice to design your code so that this is achieved without resorting to the use of any special conditional statements.

6 Further Features of MPI

The purpose of this short paper has been to introduce the inexperienced reader to the fundamentals of parallel programming using MPI. Only a very small proportion of the MPI library has been discussed: we have described only 13 functions out of well over 100. These functions are MPI_Abort, MPI_Allreduce, MPI_Bcast, MPI_Comm_rank, MPI_Comm_size, MPI_Finalize, MPI_Init, MPI_Irecv, MPI_Isend, MPI_Recv, MPI_Send, MPI_Test and MPI_Wait. Clearly there is a very large amount more to MPI than this and so we strongly recommend further reading once a command of these basic concepts has been acquired. A full and complete definition of the MPI standard, along with a number of detailed examples and explanations, may be found in [5]. Alternatively, [2] is a more introductory text which is considerably more comprehensive than this paper.

Some of the main features of MPI that may be found in these texts but which are not presented here include the ability to create your own groups of processes and inter- or intra-group communication domains (communicators). This is particularly useful when writing programs to implement divide-and-conquer or multilevel algorithms since groups can be made to divide into subgroups or merge together into larger groups as required. Another important MPI feature is the ability to create user-defined data types and to pack data items of different types together to be passed as a single message. Timing routines are also provided.

As ones parallel programs become more and more sophisticated a wider variety of MPI functions are likely to be of use. It is worth ending this paper therefore with a couple more references, this time to introductory books on the design of parallel algorithms: [1, 3]. Message passing in general, and MPI in particular, are merely tools for the implementation of parallel algorithms. It follows therefore, that efficiency of any parallel programs that one writes depends just as much on the quality of the underlying algorithm as on the quality of the message passing implementation.

References

[1] I. Foster, *"Designing and Building Parallel Programs"*, Addison-Wesley, 1994.

[2] W. Gropp, E. Lusk, A. Skjellum, *"Using MPI: Portable Parallel Programming with the Message-Passing Interface"*, MIT Press, Cambridge, Massachusetts, 1994.

[3] V. Kumar, A. Grama, A. Gupta and G. Karypis, *"Introduction to Parallel Computing"*, Benjamin/Cummings, 1994.

[4] Message Passing Interface Forum, *"MPI: A Message-Passing Interface Standard"*, International Journal of Supercomputer Applications, Vol. 8, No. 3/4, 1994.

[5] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, J. Dongarra, *"MPI The Complete Reference"*, MIT Press, Cambridge, Massachusetts, 1996.

A Appendix for Fortran Programmers

A.1 Introduction

In this appendix we outline the differences between the Fortran language binding for MPI and the C language binding used for illustration in the main text. As will be seen, these differences are quite small and they are handled in a natural and consistent manner. Each subsection below corresponds directly to a numbered section in the main paper and it is intended that it should be read immediately after that numbered section. Please note that throughout this appendix we use the term "parameter" to refer to the arguments that are passed to a subprogram (as opposed to the alternative meaning in Fortran where a parameter is a constant value). We also assume that the Fortran compiler accepts identifiers that are longer than six characters in length and of either upper or lower case.

A.2 Getting started with MPI

The first difference when using Fortran rather than C is that instead of including the file "mpi.h", the file "mpif.h" should be included. The syntax for this is to place the line

```
      include 'mpif.h'
```

immediately after the program statement at the beginning of your code.

For each MPI function in C there is a corresponding subroutine in Fortran. This subroutine has exactly the same name as the C function and a corresponding parameter list. It should be noted that the last parameter passed to every MPI subroutine is an additional integer parameter which is an error flag. This replaces the integer function value that is returned by most functions in the C language binding. For example the two lines of C

```
MPI_Comm_size(MPI_COMM_WORLD, &noprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &nid);
```

are replaced by the two lines of Fortran

```
      call MPI_Comm_size(MPI_COMM_WORLD, noprocs, ierr)
      call MPI_Comm_rank(MPI_COMM_WORLD, nid, ierr)
```

where *noprocs*, *nid* and *ierr* are integer variables.

The final difference that arises in this section of the paper is in the initialization routine MPI_Init. Since Fortran does not permit command line arguments the two parameters *argc* and *argv* are not present in the call to this subroutine. Hence this correct Fortran syntax is:

```
      call MPI_Init(ierr)
```

where *ierr* is an integer variable.

A.3 One-to-all and all-to-one communication

There is little extra in this section that differs under Fortran. As with all parameters which return values from C functions via pointers these pointers are simply replaced by variable names in the corresponding Fortran subroutines. For example allowable subroutine calls include

```
      call MPI_Bcast(N,1,MPI_INT,0,MPI_COMM_WORLD,ierr)
      call MPI_Reduce(Sum,Gsum,1,MPI_REAL,MPI_SUM,0,
     &                MPI_COMM_WORLD,ierr)
```

where N and *ierr* are integer variables and *Sum* and *GSum* are variables of type real. Note that there are fewer MPI data types with the Fortran binding than with C and that some of the names are different (e.g. MPI_REAL rather than MPI_FLOAT). The full list of Fortarn MPI data types is: MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_COMPLEX, MPI_LOGICAL and MPI_CHARACTER. In addition there are two extra data types (as there are for the C binding): MPI_BYTE and MPI_PACKED.

A.4 Point-to-point communication

Since Fortran does not permit dynamic memory allocation the example program in section 4 could not be directly translated into Fortran without altering the arrays a and b to be staticly declared. Other than this, the only significant difference is with the variable *status*. In C this is a structure of type MPI_Status whereas in Fortran (because structures are not permitted) *status* is an array of integers of length MPI_STATUS_SIZE. The three constants MPI_SOURCE, MPI_TAG and MPI_ERROR are the indices of this array which point to the source, tag and error fields respectively.

A.5 Non-blocking communication and a simple Poisson equation solver

There are a few more differences between Fortran and C that arise in this section. One of the most obvious is that two dimensional arrays in Fortran are stored by column rather than by row. Hence when partitioning the finite difference grid for a Fortran code it makes much better sense to partion the columns rather than the rows. This will ensure that when an entire column of solution values is passed to a neighbouring process the data will be stored in contiguous memory locations. Clearly, for a square mesh, this will have no significant effect of the efficiency of the algorithm.

As in the previous examples we again note that with Fortran dynamic memory allocation is not possible and so all arrays would need to be declared staticly. In addition to this it would be necessary to explicitly overwrite the matrix *old* by the values stored in *new* rather than just swapping the pointers (as done in the C program).

There are two specific differences in MPI for this example with the Fortran binding. The first is that the variables which form the final parameter to MPI_Isend and MPI_Irecv (which are the penultimate parameters in the Fortran binding of course) are of type integer (rather than MPI_Request). Also, the second parameter to the function MPI_Test (not actually used in the example program) is of type logical rather than integer.

Finally, the parallel output to separate files that is attempted in the example program in this section can also be performed in Fortran. For example, a suitable fragment of code for up to 100 processes would be

```
call MPI_Comm_rank(MPI_COMM_WORLD, nid, ierr)
write(suffix,'(i2)')nid
if (nid.le.9) then
   suffix(1:1)='0'
end if
open(10,file='Solution'//suffix//'.Txt',form='formatted')
```

where $suffix$ is a variable of type character*2 and nid and $ierr$ are integer variables.