8: Using MPI

John Burkardt
Information Technology Department
Virginia Tech

.....

FDI Summer Track V:
Parallel Programming

.....

https://people.sc.fsu.edu/~jburkardt/presentations/
mpi_part2_2008_vt.pdf

10-12 June 2008

Using MPI

- Your First Six Words in MPI
- How Messages Are Sent and Received
- Prime Sum in C+MPI
- Matrix*Vector in Fortran77+MPI
- Conclusion

Your First Six "Words" in MPI

You can write useful programs using the six fundamental routines:

- **MPI_Init**
- **MPI_Finalize**
- **MPI_Comm_Rank**
- **MPI_Comm_Size**
- **MPI_Send**
- **MPI_Recv**

MPI Language Lesson: MPI_Init

MPI_Init (&argc, &argv)

- **&argc**, the address of the program argument counter;
- **&argv**, the address of the program argument list

Must be the first MPI routine called.

MPI Language Lesson: MPI_Finalize

MPI_Finalize ( )

Must be the last MPI routine called.

MPI Language Lesson: MPI_Comm_Rank

MPI_Comm_Rank (communicator, &id)

- **communicator**, set this to **MPI_COMM_WORLD**;
- **&id**, returns the MPI ID of this process.

This is how a processor figures out its ID.

MPI Language Lesson: MPI_Comm_Size

MPI_Comm_Size (communicator, &p)

- **communicator**, set this to **MPI_COMM_WORLD**;
- **&p**, returns the number of processors available.

This is how a processor finds out how many other processors there are.

MPI Language Lesson: MPI_Send

MPI_Send (data, count, type, to, tag, communicator)

- **data**, the address of the data;
- **count**, the number of data items;
- **type**, the data type (**MPI_INT**, **MPI_FLOAT**...);
- **to**, the processor ID to which data is sent;
- **tag**, a message identifier ("0", "1", "1492" etc);
- **communicator**, set this to **MPI_COMM_WORLD**;

MPI Language Lesson: MPI_Recv

MPI_Recv (data, count, type, from, tag, communicator, status)

- **data**, the address of the data;
- **count**, number of data items;
- **type**, the data type (must match what is sent);
- **from**, the processor ID from which data is received (must match the sender, or if don't care, **MPI_ANY_SOURCE**;
- **tag**, the message identifier (must match what is sent, or, if don't care, **MPI_ANY_TAG**);
- **communicator**, (must match what is sent);
- **status**, (auxilliary diagnostic information).

How Messages Are Sent and Received

The main feature of MPI is the use of messages to send data between processors.

There is a family of routines for sending messages, but the simplest is the pair **MPI_Send** and **MPI_Recv**.

Two processors must be in a common "communicator group" in order to communicate. This is simply a way for the user to organize processors into sub-groups. All processors can communicate in the shared group known as **MP_COMM_WORLD**.

In order for data to be transferred by a message, there must be a sending program that wants to send the data, and a receiving program that expects to receive it.

How Messages Are Sent and Received

The sender calls **MPI_Send**, specifying the data, as well as an identifier for the message, and the name of the communicator group it is using.

On executing the call to **MPI_Send**, the sending program pauses, the message is transferred to a buffer on the receiving computer system and the MPI system there prepares to deliver it to the receiving program.

The receiving program must be expecting to receive a message, that is, it must execute a call to **MPI_Recv** and be waiting for a response. The message it receives must correspond in size, arithmetic precision, message identifier, and communicator group.

Once the message is received, the receiving process can proceed.

The sending process gets a response that the message was received, and it can proceed as well.

How Messages Are Sent and Received

If an error occurs during the message transfer, both the sender and receiver return a nonzero flag value, either as the function value (in C and C++) or in the final **ierr** argument in the FORTRAN version of the MPI routines.

When the receiving program finishes the call to **MPI_Recv**, the extra parameter **status** includes information about the message transfer.

The status variable is not usually of interest with simple **Send/Recv** pairs, but for other kinds of message transfers, it can contain important information

How Messages Are Sent and Received

1. The sender program pauses at **MPI_SEND**;
2. The message goes into a buffer on the receiver machine;
3. The receiver program does not receive the message until it reaches the corresponding **MPI_RECV**.
4. The receiver program pauses at **MPI_RECV** until the message has arrived.
5. Once the message has been received, the sender and receiver resume execution

Excessive idle time caused while waiting to receive a message, or to get confirmation that the message was received, can strongly affect the performance of an MPI program.

How Messages Are Sent and Received

```
MPI_Send ( data, count, type, to,   tag, comm )
                    |      |         |     |
MPI_Recv ( data, count, type, from, tag, comm, status )
```

The **MPI_SEND** and **MPI_RECV** must match:

1. **count**, the number of data items, must match;
2. **type**, the type of the data, must match;
3. **from**, must be the process id of the sender, or the receiver may specify **MPI_ANY_SOURCE**.
4. **tag**, a user-chosen ID for the message, must match, or the receiver may specify **MPI_ANY_TAG**.
5. **comm**, the name of the communicator, must match (for us, this will always be **MPI_COMM_WORLD**

How Messages Are Sent and Received

By the way, if the **MPI_RECV** allows a "wildcard" source by specifying **MPI_ANY_SOURCE** or a wildcard tab by specifying **MPI_ANY_TAG**, then the actual value of the tag or source is included in the **status** variable, and can be retrieved there.

```
source = status(MPI_SOURCE)          FORTRAN
tag = status(MPI_TAG)

source = status.(MPI_SOURCE);        C
tag = status.MPI_TAG);

source = status.Get_source ( );      C++
tag = status.Get_tag ( );
```

The Prime Sum Example in MPI

Let's do the PRIME SUM problem in MPI. Here we want to add up the prime numbers from 2 to N.

Each of **P** processors will simply take about 1/**P** of the range of numbers to check, and add up the primes it finds locally.

When it's done, it will send the partial result to processor 0.

So processors 1 to **P** send a single message (simple) and processor 0 has to expect any of **P**-1 messages total.

Prime Sum Example: Page 1

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

int main ( int argc, char *argv[] )
{
  int i, id, j, master = 0, n = 1000, n_hi, n_lo;
  int p, prime, total, total_local;
  MPI_Status status;
  double wtime;

  MPI_Init ( &argc, &argv );
  MPI_Comm_size ( MPI_COMM_WORLD, &p );
  MPI_Comm_rank ( MPI_COMM_WORLD, &id );
```

Prime Sum Example: Page 2

```
n_lo = ( ( p - id     ) * 1 + ( id     ) * n ) / p + 1;
n_hi = ( ( p - id - 1 ) * 1 + ( id + 1 ) * n ) / p;

wtime = MPI_Wtime ( );

total_local = 0.0;
for ( i = n_lo; i <= n_hi; i++ ) {
  prime = 1;
  for ( j = 2; j < i; j++ ) {
    if ( i % j == 0 ) {
      prime = 0;
      break; } }
  if ( prime == 1 ) total_local = total_local + i;
}
wtime = MPI_Wtime ( ) - wtime;
```

Prime Sum Example Page 3

```
  if ( id != master ) {
    MPI_Send ( &total_local, 1, MPI_INT, master, 1,
      MPI_COMM_WORLD ); }
  else {
    total = total_local;
    for ( i = 1; i < p; i++ ) {
      MPI_Recv ( &total_local, 1, MPI_INT, MPI_ANY_SOURCE,
        1, MPI_COMM_WORLD, &status );
      total = total + total_local; } }
  if ( id == master ) printf ( "  Total is %d\n", total );
  MPI_Finalize ( );
  return 0;
}
```

Prime Sum Example: Output

```
n825(0): PRIME_SUM - Master process:
n825(0):   Add up the prime numbers from 2 to 1000.
n825(0):   Compiled on Apr 21 2008 at 14:44:07.
n825(0):
n825(0): The number of processes available is 4.
n825(0):
n825(0): P0 [   2,  250] Total =  5830 Time = 0.000137
n826(2): P2 [ 501,  750] Total = 23147 Time = 0.000507
n826(2): P3 [ 751, 1000] Total = 31444 Time = 0.000708
n825(0): P1 [ 251,  500] Total = 15706 Time = 0.000367
n825(0):
n825(0):        The total sum is 76127

All nodes terminated successfully.
```

The Prime Sum Example in MPI

Having all the processors compute partial results, which then have to be collected together is another example of a reduction operation.

Just as with OpenMP, MPI recognizes this common operation, and has a special function call which can replace all the sending and receiving code we just saw.

Prime Sum Example Page 3 REVISED

```
MPI_Reduce ( &total_local, &total, 1, MPI_INT, MPI_SUM,
  master, MPI_COMM_WORLD );

if ( id == master ) printf ( "  Total is %d\n", total );
MPI_Finalize ( );
return 0;
```

MPI Language Lesson: MPI_REDUCE

MPI_Reduce (local_data, reduced_value, count, type, operation, to, communicator)

- **local_data**, the address of the local data;
- **reduced_value**, the address of the variable to hold the result;
- **count**, number of data items;
- **type**, the data type;
- **operation**, the reduction operation **MPI_SUM, MPI_PROD, MPI_MAX...**;
- **to**, the processor ID which collects the local data into the reduced data;
- **communicator**;

Matrix * Vector Example

We will now consider an example in which matrix multiplication is carried out using MPI.

This is an artificial example, so don't worry about **why** we're going to divide the task up. Concentrate on **how** we do it.

We are going to compute $A * x = b$.

We start with the entire matrix **A** and vector **X** sitting on the "master processor" (whichever processor has lucky number 0).

We need to send *some* of this data to other processors, they carry out their part of the task, and processor 0 collects the results back

Matrix * Vector Example

Because one processor will be special, directing the work, this program will be an example of the "master-workers" model.

Entry b_i is the dot product of row i of the matrix with x:

$$b_i = \sum_{j=1}^{N} A_{ij} x_j$$

So if there were **N** workers available, then each one could do one entry of b.

There are only **P** processors available, and only **P-1** can be workers, so we'll have to do the job in batches.

Matrix * Vector Example

Give all the workers a copy of x.

Then send row i of A to processor i.

When processor i returns b_i, send the next available row of A,

The way we are setting up this algorithm allows processors to finish their work in any order. This approach is flexible.

In consequence, the master process doesn't know which processor will be sending a response. It has to keep careful track of what data comes in, and when everything is done.

Matrix * Vector Example

In a master-worker model, you can really see how an MPI program, which is supposed to be a single program running on all machines, can end up looking more like *two* programs.

Matrix * Vector: Master Pseudocode

```
If I am the master:
  SEND N to all workers.
  SEND X to all workers.
  SEND out first batch of rows.
  While ( any entries of B not returned )
    RECEIVE message, entry ? of B, from processor ?.
    If ( any rows of A not sent )
      SEND row ? of A to processor ?.
    else
      SEND "FINALIZE" message to processor ?.
    end
  end
  FINALIZE
```

Matrix * Vector: Worker Pseudocode

```
else if I am a worker

  RECEIVE N.
  RECEIVE X.
  do
    RECEIVE message.

    if ( message is "FINALIZE" ) then
      FINALIZE
    else
      it's a row of A, so compute dot product with X.
      SEND result to master.
    end
  end
end
```

Matrix * Vector: Using BROADCAST

In some cases, the communication that is to be carried out doesn't involve a pair of processors talking to each other, but rather one processor "announcing" some information to all the others.

This is often the case when the program is written using the *master/worker* model, in which case one processor, (usually the one with ID 0) is put in charge. It takes care of interacting with the user, doing I/O, collecting results from the other processors, handling reduction operations and so on.

There is a "broadcast" function in MPI that makes it easy for the master process to send information to all other processors.

In this case, the same function is used both for the sending and receiving!

MPI Language Lesson: MPI_Bcast

MPI_Bcast (data, count, type, from, communicator)

- **data**, the address of the data;
- **count**, number of data items;
- **type**, the data type;
- **from**, the processor ID which sends the data;
- **communicator**;

Matrix * Vector: An example algorithm

Compute $A * x = b$.

- a "task" is to multiply one row of A times x;
- we can assign one task to each processor. Whenever a processor is done, give it another task.
- each processor needs a copy of x at all times; for each task, it needs a copy of the corresponding row of A.
- processor 0 will do no tasks; instead, it will pass out tasks and accept results.

Matrix * Vector in FORTRAN77 (Page 1)

```
      if ( my_id == master )

        numsent = 0
c
c  BROADCAST X to all the workers.
c
        call MPI_BCAST ( x, cols, MPI_DOUBLE_PRECISION, master,
     &    MPI_COMM_WORLD, ierr )

c
c  SEND row I to worker process I; tag the message with the row number.
c
        do i = 1, min ( num_procs-1, rows )

          do j = 1, cols
            buffer(j) = a(i,j)
          end do

          call MPI_SEND ( buffer, cols, MPI_DOUBLE_PRECISION, i,
     &      i, MPI_COMM_WORLD, ierr )

          numsent = numsent + 1

        end do
```

Matrix * Vector in FORTRAN77 (Page 2)

```
c
c  Wait to receive a result back from any processor;
c  If more rows to do, send the next one back to that processor.
c
      do i = 1, rows

        call MPI_RECV ( ans, 1, MPI_DOUBLE_PRECISION,
     &      MPI_ANY_SOURCE, MPI_ANY_TAG,
     &      MPI_COMM_WORLD, status, ierr )

        sender = status(MPI_SOURCE)
        anstype = status(MPI_TAG)
        b(anstype) = ans

        if ( numsent .lt. rows ) then

          numsent = numsent + 1

          do j = 1, cols
            buffer(j) = a(numsent,j)
          end do

          call MPI_SEND ( buffer, cols, MPI_DOUBLE_PRECISION,
     &      sender, numsent, MPI_COMM_WORLD, ierr )

        else

          call MPI_SEND ( MPI_BOTTOM, 0, MPI_DOUBLE_PRECISION,
     &      sender, 0, MPI_COMM_WORLD, ierr )

        end if

      end do
```

Matrix * Vector in FORTRAN77 (Page 3)

```
c
c  Workers receive X, then compute dot products until
c  done message received
c
      else

         call MPI_BCAST ( x, cols, MPI_DOUBLE_PRECISION, master,
     &     MPI_COMM_WORLD, ierr )

 90      continue

         call MPI_RECV ( buffer, cols, MPI_DOUBLE_PRECISION, master,
     &     MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr )

         if ( status(MPI_TAG) .eq. 0 ) then
           go to 200
         end if

         row = status(MPI_TAG)

         ans = 0.0
         do i = 1, cols
           ans = ans + buffer(i) * x(i)
         end do

         call MPI_SEND ( ans, 1, MPI_DOUBLE_PRECISION, master,
     &      row, MPI_COMM_WORLD, ierr )

         go to 90

 200     continue

      end if
```

Matrix * Vector: An example algorithm

Compute $A * x = b$.

- a "task" is to multiply one row of A times x;
- we can assign one task to each processor. Whenever a processor is done, give it another task.
- each processor needs a copy of x at all times; for each task, it needs a copy of the corresponding row of A.
- processor 0 will do no tasks; instead, it will pass out tasks and accept results.

Non-Blocking Message Passing

Using **MPI_Send** and **MPI_Recv** forces the sender and receiver to pause until the message has been sent and received.

In some cases, you may be able to improve efficiency by letting the sender send the message and proceed immediately to more computations.

On the receiver side, you might also want to declare the receiver ready, but then go immediately to computation while waiting to actually receive.

The non-blocking **MPI_Isend** and **MPI_Irecv** allow you to do this. However, the sending routine must not change the data in the array being sent until the data has actually been successfully transmitted. The receiver cannot try to use the data until it has been received.

This is done by calling **MPI_Test** or **MPI_Wait**.

Nonblocking Send/Receive Pseudocode

```
if I am the boss
{
  Isend ( X(  1:100) to worker 1, req1 )
  Isend ( X(101:200) to worker 2, req2 )
  Isend ( X(201:300) to worker 3, req3 )
  Irecv ( fx1 from worker1, req4 )
  Irecv ( fx2 from worker2, req5 )
  Irecv ( fx3 from worker3, req6 )

  while ( 1 ) {
    if (  Test ( req1 ) && Test ( req2 ) &&
          Test ( req3 ) && Test ( req4 ) &&
          Test ( req5 ) && Test ( req6 ) )
      break
  }
}
```

Nonblocking Send/Receive Pseudocode

```
else if I am a worker
{
  Irecv ( X, from boss, req ) <-- Ready to receive

  set up tables               <-- work while waiting

  Wait ( req )                <-- pause til data here.

  Compute fx = fun(X)         <-- X here, go to it.

  Isend ( fx to boss, req )
}
```

MPI Language Lesson: MPI_Irecv

MPI_Irecv (data, count, type, from, tag, comm, req)

- **data**, the address of the data;
- **count**, number of data items;
- **type**, the data type;
- **from**, the processor ID from which data is received;
- **tag**, the message identifier;
- **comm**, the communicator;
- **req**, the request array or structure.

MPI Language Lesson: MPI_Test

MPI_Test (req, flag, status)

MPI_Test reports whether the message associated with **req** has been sent and received.

- **req**, the address of the data;
- **flag**, is returned as TRUE if the sent message was received;
- **status**, the status array or structure.

MPI Language Lesson: MPI_Wait

MPI_Wait (req, status)

MPI_Wait waits until the message associated with **req** has been sent and received.

- **req**, the address of the data;
- **status**, the status array or structure.

Conclusion

One of MPI's strongest features is that it is well suited to modern clusters of 100 or 1,000 processors.

In most cases, an MPI implementation of an algorithm is quite different from the serial implementation.

In MPI, communication is explicit, and you have to take care of it. This means you have more control; you also have new kinds of errors and inefficiencies to watch out for.

MPI can be difficult to use when you want tasks of different kinds to be going on.

MPI and OpenMP can be used together; for instance, on a cluster of multicore servers.

References: Books

- Gropp, **Using MPI**;
- **Mascagni, Srinavasan**, *Algorithm 806: SPRNG: a scalable library for pseudorandom number generation*, ACM Transactions on Mathematical Software
- Openshaw, **High Performance Computing**;
- Pacheco, **Parallel Programming with MPI** ;
- Petersen, **Introduction to Parallel Computing**;
- Quinn, **Parallel Programming in C with MPI and OpenMP**;
- Snir, **MPI: The Complete Reference**;