The Message Passing Interface (MPI): Parallelism on Multiple (Possibly Heterogeneous) CPUs

http://mpi-forum.org
https://www.open-mpi.org/

The Open MPI Consortium

Why Two URLs?

http://mpi-forum.org
This is the definitive reference for the MPI standard. Go here if you want to read the official specification, which, BTW, continues to evolve.

https://www.open-mpi.org/
This consortium formed later. This is the open source version of MPI. If you want to start using MPI, I recommend you look here.

https://www.open-mpi.org/doc/v4.0/
This URL is also really good – it is a link to all of the MPI man pages

MPI: The Basic Idea

Network

CPU

Note: Each CPU in the MPI “cluster” must be conditioned ahead of time by having the MPI server code installed on it. Each MPI CPU must also have an integer ID assigned to it (called the rank) and must be registered with the primary (“root”) MPI CPU.

Compiling and Running

Compiling:  
% mpicc -o program program.c
or
% mpic++ -o program program.cpp

Running:  
% mpiexec -np 64 program

# of processors to use

Setting Up and Finishing

#include <mpi.h>

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    ...  
    MPI_Finalize();
    return 0;
}
MPI follows a Single-Program-Multiple-Data (SPMD) Model

A communicator is a collection of CPUs that are capable of sending messages to each other.

Getting information about our place in the communicator:

```c
int numCPUs; // total # of cpus involved
int me; // which one I am
MPI_Comm_Size( MPI_COMM_WORLD, &numCPUs );
MPI_Comm_Rank( MPI_COMM_WORLD, &me );
```

Both the sender and receivers need to execute MPI_Bcast – there is no separate receive function.

```
MPI_Bcast( array, count, type, src, MPI_COMM_WORLD );
```

How Does this Work? Think Star Trek Wormholes!

This is our heat transfer equation from before. Clearly, every CPU will need to know this value.

```
\Delta T_i = \frac{k_i}{\rho c_i} \left( T_{i+1} - 2T_i + T_{i-1} \right) \Delta x
```

Sending Data from a Source CPU to Several Destination CPUs

```
MPI_Send( array, numToSend, type, dst, tag, MPI_COMM_WORLD );
```

Rules:
- One message from a specific src to a specific dst cannot overtake a previous message from the same src to the same dst.
- There are no guarantees on order from different src's.
- MPI_Send() blocks until the transfer is far enough along that the data can be destroyed or re-used.

Receiving Data in a Destination CPU from a Source CPU

```
MPI_Recv( array, maxCanReceive, type, src, tag, MPI_COMM_WORLD, &status );
```

Rules:
- The receiver blocks waiting for data that matches what it declares to be looking for
- One message from a specific src to a specific dst cannot overtake a previous message from the same src to the same dst
- There are no guarantees on the order from different src's
- The order from different src's could be implied in the tag
- status is type MPI_Status – the "&status" can be replaced with MPI_STATUS_IGNORE
### MPI status Values

<table>
<thead>
<tr>
<th>Status Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_COMM_NULL</td>
<td>No communication group</td>
</tr>
<tr>
<td>MPI_COMM_WORLD</td>
<td>All processes</td>
</tr>
<tr>
<td>MPI_COMM_LOCAL</td>
<td>Local processes</td>
</tr>
<tr>
<td>MPI_COMM_SELF</td>
<td>Self process</td>
</tr>
<tr>
<td>MPI_COMM_WORLD_SIZE</td>
<td>Total number of processes</td>
</tr>
<tr>
<td>MPI_COMM_WORLD_RANK</td>
<td>Rank of the process</td>
</tr>
</tbody>
</table>

### Example

```
int numCPUs;
int me;
#define MYDATA_SIZE 128
char mydata[MYDATA_SIZE];
#define ROOT 0
MPI_Comm_Size(MPI_COMM_WORLD, &numCPUs);
MPI_Comm_Rank(MPI_COMM_WORLD, &me);
if (me == ROOT) // the primary
{
  for (int dst = 0; dst < numCPUs; dst++)
  {
    if (dst != me)
    {
      char *InputData = "Hello, Beavers!";
      MPI_Send(InputData, strlen(InputData)+1, MPI_CHAR, dst, 0, MPI_COMM_WORLD);
    }
  }
}
else // a secondary
{
  MPI_Recv(myData, MYDATA_SIZE, MPI_CHAR, ROOT, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  printf("'%s' from rank # %d\n", In, me);
}
```

### Another Example

```
#define NUMELEMENTS 123456
int numCPUs;
int me;
#define ROOT 0
MPI_Comm_Size(MPI_COMM_WORLD, &numCPUs);
MPI_Comm_Rank(MPI_COMM_WORLD, &me);
int localSize = NUMELEMENTS / numCPUs; // assuming it comes out evenly
float *myData = new float[localSize];
if (me == ROOT) // the primary
{
  float *InputData = new float[NUMELEMENTS];
  // read the full input data into InputData from disk
  for (int dst = 0; dst < numCPUs; dst++)
  {
    if (dst != me)
    {
      MPI_Send(InputData[dst*localSize], localSize, MPI_FLOAT, dst, 0, MPI_COMM_WORLD);
    }
  }
}
else // a secondary
{
  MPI_Recv(myData, localSize, MPI_FLOAT, ROOT, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  // do something with this subset of the data
}
```

### How Much Data Did I Actually Receive?

```
MPI_Recv(array, maxCanReceive, type, src, tag, MPI_COMM_WORLD, &status);
MPI_Status status;
int actualCount; // in # of elements
MPI_Get_count(&status, type, &actualCount);
```

### How does MPI let the Sender perform an MPI_Send() even if the Receivers are not ready to MPI_Recv()?

MPI_Send() blocks until the transfer is far enough along that the array can be destroyed or re-used.

### Another Example

```
Matrix NUMELEMENTS = ????
int numCPUs;
int me;
#define ROOT 0
MPI_Comm_Size(MPI_COMM_WORLD, &numCPUs);
MPI_Comm_Rank(MPI_COMM_WORLD, &me);
int localSize = NUMELEMENTS / numCPUs; // assuming it comes out evenly
float *myData = new float[localSize];
if (me == ROOT) // the primary
{
  float *InputData = new float[NUMELEMENTS];
  // read the full input data into InputData from disk
  for (int dst = 0; dst < numCPUs; dst++)
  {
    if (dst != me)
    {
      MPI_Send(InputData[dst*localSize], localSize, MPI_FLOAT, dst, 0, MPI_COMM_WORLD);
    }
  }
}
else // a secondary
{
  MPI_Recv(myData, localSize, MPI_FLOAT, ROOT, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  // do something with this subset of the data
}
```

### Remember This? It's Baaaaaack.

This pattern of breaking a big problem up into pieces, sending them to different CPUs, computing on the pieces, and getting the results back is so common that it has its own name: **Scatter/Gather**, and has its own MPI function calls.
MPI Scatter

Take a data array, break it into equal portions, and send it to each CPU.

```
MPI_Scatter( snd_array, snd_count, snd_type, rcv_array, rcv_count, rcv_type, src, MPI_COMM_WORLD);
```

Both the sender and receivers need to execute MPI_Scatter -- there is no separate receive function.

MPI Scatter Example

```
#define NUMELEMENTS ????
int numCPUs;
int me;
#define ROOT 0
MPI_Comm_Size( MPI_COMM_WORLD, &numCPUs );
MPI_Comm_Rank( MPI_COMM_WORLD, &me );
Int localSize = NUMELEMENTS / numCPUs; // assuming it comes out evenly
float *myData = new float[localSize];
if( me == ROOT ) // this is the sender
{
    float *InputData = new float[NUMELEMENTS];
    // read the full input data into InputData from disk
    MPI_Scatter( InputData, NUMELEMENTS, MPI_FLOAT, myData, localSize, MPI_FLOAT, ROOT, MPI_WORLD_COMM );
}
else // this is the receiver
{
    MPI_Scatter( NULL, 0, MPI_FLOAT, myData, localSize, MPI_FLOAT, ROOT, MPI_WORLD_COMM );
}
```

It is typical that you scatter to yourself.

MPI Gather

```
MPI_Gather( snd_array, snd_count, snd_type, rcv_array, rcv_count, rcv_type, dst, MPI_COMM_WORLD);
```

A Full MPI Scatter / Gather Example

```
#define NUMELEMENTS ????
int numCPUs;
int me;
#define ROOT 0
MPI_Comm_Size( MPI_COMM_WORLD, &numCPUs );
MPI_Comm_Rank( MPI_COMM_WORLD, &me );
Int localSize = NUMELEMENTS / numCPUs; // assuming it comes out evenly
float *myData = new float[localSize];
if( me == ROOT ) // send, then receive
{
    float *InputData = new float[NUMELEMENTS];
    // read the full input data into InputData from disk
    MPI_Scatter( InputData, NUMELEMENTS, MPI_FLOAT, myData, localSize, MPI_FLOAT, ROOT, MPI_WORLD_COMM );
    // do some computing on myData[
    MPI_Gather( myData, localSize, MPI_FLOAT, InputData, NUMELEMENTS, MPI_FLOAT, ROOT, MPI_WORLD_COMM );
    // write data from InputData[] to disk
}
else // receive then send
{
    MPI_Scatter( NULL, 0, MPI_FLOAT, myData, localSize, MPI_FLOAT, ROOT, MPI_WORLD_COMM );
    // do some computing on myData[
    MPI_Gather( myData, localSize, MPI_FLOAT, NULL, 0, MPI_FLOAT, ROOT, MPI_WORLD_COMM );
}
```

A Full MPI Scatter / Gather Example

```
MPI_Reduce( partialResult, globalResult, count, type, operator, dst, MPI_COMM_WORLD );
```

MPI Reduction

```
MPI_Reduce( partialResult, globalResult, count, type, operator, dst, MPI_COMM_WORLD );
```

Both the sender and receivers need to execute MPI_Reduce -- there is no separate receive function.
int numCPUs;
int me;
float globalSum;

#define ROOT 0

MPI_Comm_Size(MPI_COMM_WORLD, &numCPUs);
MPI_Comm_Rank(MPI_COMM_WORLD, &me);

float partialSum = 0.0;
<< compute this CPUs partialSum by, perhaps, adding up a local array >>
MPI_Reduce(&partialSum, &globalSum, 1, MPI_FLOAT, MPI_SUM, ROOT, MPI_COMM_WORLD);

MPI_Barrier(MPI_COMM_WORLD);

MPI_Type_create_struct(4, blocklengths, displacements, types, &MPI_POINT);
You can now use MPI_POINT everywhere you could have used MPI_INT, MPI_FLOAT etc.

MPI_Type_create_struct( count, blocklengths, displacements, types, datatype );

struct point {
  int pointSize;
  float x, y, z;
};

MPI_Datatype MPI_POINT;

Suppose We Have This Setup

Welcome to Parallelism Jeopardy!

IN A MULTI-CPU SYSTEM, THIS IS THE TOTAL NUMBER OF DIFFERENT PARALLELSMS THAT WE COVERED THIS QUARTER
<table>
<thead>
<tr>
<th>1. Multicore OpenMP</th>
<th>2. CPU SIMD</th>
<th>3. GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Multicore OpenMP</td>
<td>2. CPU SIMD</td>
<td>3. GPU</td>
</tr>
</tbody>
</table>

What is “4”?

and, they can all be functioning within the same application!