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

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

Mike Bailey
mjb@cs.oregonstate.edu
Oregon State University

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.

The Open MPI Consortium

MPI: The Basic Idea

Network

<table>
<thead>
<tr>
<th>Memory</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>CPU</td>
</tr>
</tbody>
</table>

Programs on different CPUs coordinate computations by passing messages between each other.

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

Compiling and Running

C:\% mpiicc -o program program.c...
C:\% mpiic++ -o program program.cpp...
C:\% 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;
}

If you don’t need to process command line arguments, you can also call:
MPI_Init(NULL, NULL);
MPI follows a single-program-multiple-data 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:
- `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 );`

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

A good place to start: MPI broadcasting.

MPI follows a single-program-multiple-data model. This requires MPI server code getting installed on all those CPUs.

Broadcast

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

Broadcasting

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.

### MPI Broadcast Example

<table>
<thead>
<tr>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>One message from a specific src to a specific dst cannot overwrite a previous message from the same src to the same dst.</td>
</tr>
<tr>
<td><code>MPI_Send()</code> blocks until the transfer is far enough along that array can be destroyed or reused.</td>
</tr>
<tr>
<td>There are no guarantees on order from different src’s.</td>
</tr>
</tbody>
</table>

### Sending Data from a Source CPU to Several Destination CPUs

<table>
<thead>
<tr>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>One message from a specific src to a specific dst cannot overwrite a previous message from the same src to the same dst.</td>
</tr>
<tr>
<td><code>MPI_Send()</code> blocks until the transfer is far enough along that array can be destroyed or reused.</td>
</tr>
<tr>
<td>There are no guarantees on order from different src’s.</td>
</tr>
</tbody>
</table>

### Receiving Data in a Destination CPU from a Source CPU

<table>
<thead>
<tr>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>The receiver blocks waiting for data that matches what it declares to be looking for</td>
</tr>
<tr>
<td>One message from a specific src to a specific dst cannot overwrite a previous message from the same src to the same dst</td>
</tr>
<tr>
<td>There are no guarantees on the order from different src’s</td>
</tr>
<tr>
<td>The order from different src’s could be implied in the tag</td>
</tr>
<tr>
<td><code>status</code> is type <code>MPI_Status</code> – the “&amp;status” can be replaced with <code>MPI_STATUS_IGNORE</code></td>
</tr>
</tbody>
</table>

### Receiving Data

- `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 overwrite 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`
Example

Remember, this same code runs on all CPUs:

```c
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 master
{
    for (int dst = 0; dst < numCPUs; dst++)
    {
        char *InputData = "Hello, Beavers!";
        MPI_Send(InputData, strlen(InputData)+1, MPI_CHAR, dst, 0, MPI_COMM_WORLD);
    }
}
else // a slave
{
    MPI_Recv(myData, MYDATA_SIZE, MPI_CHAR, ROOT, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("'%s' from rank #%d\n", in, me);
}
```

Example

Remember, this same code runs on all CPUs:

```c
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 master
{
    for (int dst = 0; dst < numCPUs; dst++)
    {
        char *InputData = "Hello, Beavers!";
        MPI_Send(InputData, strlen(InputData)+1, MPI_CHAR, dst, 0, MPI_COMM_WORLD);
    }
}
else // a slave
{
    MPI_Recv(myData, MYDATA_SIZE, MPI_CHAR, ROOT, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("'%s' from rank #%d\n", in, me);
}
```

How Much Data Did I Actually Receive and Where Did I Get It?

```c
MPI_Recv(array, maxCanReceive, type, src, tag, MPI_COMM_WORLD, &status);
MPI_Status status;
Int actualCount;
MPI_Get_count(&status, type, &actualCount);
int src = status.MPI_SOURCE;
int tag = status.MPI_TAG;
```

Another Example

You typically don’t send the entire workload to each dst – you just send part of it, like this:

```c
#define NUMELEMENTS 7777
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 even
float *myData = new float[localSize];
if (me == ROOT) // the master
{
    float *InputData = new float[NUMELEMENTS];
    // read the full input data into InputData from disk
    for (int dst = 1; dst < numCPUs; dst++)
    {
        MPI_Send(&InputData[dst*localSize], localSize, MPI_FLOAT, dst, 0, MPI_COMM_WORLD);
    }
}
else // a slave
{
    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.

```
The Compute : Communicate Ratio still applies, except that it is even more important now because there is much more overhead in the Communicate portion.
```

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 MPI function calls.

```
MPI_Scatter
```

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

```
MPI_CHAR
MPI_SHORT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE
• • •
```

```
# elements to send
```

```
MPI_CHAR
MPI_SHORT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE
• • •
```

```
# elements to receive
```

```
MPI_CHAR
MPI_SHORT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE
• • •
```

This pattern still applies, except that it is even more important now because there is much more overhead in the Communicate portion.

```
```

```
```
```c
#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 )
{
  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
{
  MPI_Scatter( NULL,       0,                            MPI_FLOAT,  myData, localSize, MPI_FLOAT, ROOT, MPI_WORLD_COMM );
}

MPI_Gather( myData, localSize, MPI_FLOAT, NULL,        0,                          MPI_FLOAT, ROOT, MPI_WORLD_COMM );
```

---

**MPI Scatter Example**

**MPI Gather Example**

**A Full MPI Scatter / Gather Example**

**MPI Reduction**
```c
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.;
/* 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 Barriers**

All CPUs must execute a call to `MPI_Barrier()` before any of them can move past it.
Reminder: barriers are based on count, not location.

**MPI Derived Types**

Idea: In addition to types MPI_INT, MPI_FLOAT, etc., allow the creation of new MPI types so that you can transmit an "array of structures".

Reason: There is significant overhead with each transmission. Better to send one entire array of structures instead of sending several arrays separately.

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

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

MPI_datatype point_t;
int blocklengths[ ]     = { 1, 1, 1, 1 };
int displacements[ ]   = { 0, 4, 8, 12 };
MPI_type typetyped[ ]  = { MPI_INT, MPI_FLOAT, MPI_FLOAT, MPI_FLOAT };
MPI_Type_create_struct(4, blocklengths, displacements, types, &point_t);
```

You can now use point_t everywhere you could have used MPI_INT, MPI_FLOAT, etc.

**Suppose We Have This Setup**

```
Welcome to Count the Parallelisms
```

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