The Message Passing Interface (MPI):
Parallelism on Distributed CPUs

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

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

The Open MPI Consortium

MPI: The Basic Idea

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

Note: Each CPU in the MPI “cluster” must be prepared 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 its rank).
Compiling and Running

% mpicc -o program program.c ...

% mpicc -o program program.cpp ...

% mpiexec -np 64 program

# of processors to use

All distributed processors execute the same program at the same time

Setting Up and Finishing

#include <mpi.h>

int main( int argc, char *argv[] )
{
  ...
  MPI_Init( &argc, &argv );
  ...
  MPI_Finalize( );
  return 0;
}

You don’t need to process command line arguments if you don’t need to.
You can also call it as:
MPI_Init( NULL, NULL );

So, we have a group (a “communicator”) of distributed processors. How do they communicate about what work they are supposed to do?

Who am I?
Where am I?
What am I supposed to be doing?
Hello? Is anyone listening?

Example: You could coordinate the units of our DGX system using MPI

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

Oh, look, a communicator of deer!

Oh, look, a communicator of turkeys!

This requires MPI server code getting installed on all those CPUs. Only an administrator can do this.

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 );

Size, i.e., how many altogether?

Rank, i.e., which one am I?

It is then each CPU’s job to figure out what piece of the overall problem it is responsible for and then go do it.
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 );

Address of data to send from if you are the src node; Address of the data to receive into if you are not src node

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

A Good Place to Start:
MPI Broadcasting

Confused? Look at this Diagram

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.

$$\Delta T = \frac{k}{\rho C_p} \left( \frac{T_{in} - 2T + T_{out}}{\Delta x} \right)^\Delta t$$

$$\Delta T = \frac{k}{\rho C_p} \left( \frac{T_{in} - 2T + T_{out}}{\Delta x} \right) \Delta t$$

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

If \( m = \text{BOSS} \):

\[
\text{MPI_Bcast}(\text{k\_over\_rho\_c}, 1, \text{MPI\_FLOAT}, \text{BOSS}, \text{MPI\_COMM\_World});
\]

// send

I am the BOSS: this identifies this call as a send

I am not the BOSS: this identifies this call as a receive

Executable code

```
if( me == BOSS ) {
  << read \text{k\_over\_rho\_c} from the data file >>
  \text{MPI\_Bcast}(\text{k\_over\_rho\_c}, 1, \text{MPI\_FLOAT}, \text{BOSS}, \text{MPI\_COMM\_World});
  \text{\# send}
}
else {
  \text{MPI\_Bcast}(\text{k\_over\_rho\_c}, 1, \text{MPI\_FLOAT}, \text{BOSS}, \text{MPI\_COMM\_World});
  \text{\# receive}
}
```
### Sending Data from a Source CPU to a Destination CPU

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

- **address of data to send from**
- **MPI_CHAR**
- **MPI_INT**
- **MPI_LONG**
- **MPI_FLOAT**
- **MPI_DOUBLE**
- **rank of the CPU to send to**

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

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

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

- **address of data to receive into**
- **MPI_CHAR**
- **MPI_INT**
- **MPI_LONG**
- **MPI_FLOAT**
- **MPI_DOUBLE**
- **Rank of the CPU we are expecting to get a transmission from**
- **Type = MPI_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`.

### Example

Remember, this identical code runs on all CPUs:

```c
int numCPUs;
int me;

#define MYDATA_SIZE 128
char myData[MYDATA_SIZE];
#define BOSS 0
MPI_Comm_Size( MPI_COMM_WORLD, &numCPUs );
MPI_Comm_Rank( MPI_COMM_WORLD, &me );
if( me == BOSS ) // the primary
{
  for( int dst = 0; dst < numCPUs; dst++ )
  {
    if( dst != BOSS )
    {
      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, BOSS, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
  printf( " '%s' from rank # %d
", myData, me );
}
```

You are highly discouraged from sending to yourself. Because both the send and receive are capable of blocking, the result could be deadlock.
Look at this Diagram

<table>
<thead>
<tr>
<th>Executable code</th>
<th>Input Data</th>
</tr>
</thead>
</table>

**Source**

- If (dst == BOSS)
  - MPI_Send(InputData, localSize * dst, MPI_FLOAT, dst, 0, MPI_COMM_WORLD);

**Destinations**

- if (a secondary)
  - MPI_Recv(myData, localSize, MPI_FLOAT, BOSS, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

How Much Data Did I Actually Receive?

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

- # elements we can receive, at most
- Type = MPI_Status

- MPI_CHAR
- MPI_INT
- MPI_LONG
- MPI_FLOAT
- MPI_DOUBLE

- 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

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

```c
#define NUMELEMENTS ????
int numCPUs;
int me;
#define BOSS 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 == BOSS ) // 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 == BOSS )
    {
      MPI_Send( InputData, localSize, MPI_FLOAT, dst, 0, MPI_COMM_WORLD );
    }
  }
}
else // a secondary
{
  MPI_Recv( myData, localSize, MPI_FLOAT, BOSS, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
  // do something with this subset of the data
}
```
Another Example

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

```
if (dst != 0) {
    MPI_Ssend(scenario, &localSize, MPI_FLOAT, dst, MPI_COMM_WORLD);
}
```

And, what’s a good example of when you want to do this?

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 name: Scatter/Gather, and has its own MPI function calls.

In Distributed Computing, You Often Hear About These Design Patterns

Scatter and Gather Usually Go Together

Note surprisingly, this is referred to as Scatter/Gather :-)
**MPI Scatter**

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

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

- `# elements to send`
- `# elements to receive`

Both the sender and receivers need to execute `MPI_Scatter`. There is no separate receive function.

**MPI Scatter Example**

```c
#define NUMELEMENTS ????
int numCPUs;
int me;
#define BOSS 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 == BOSS)
// this is the sender
{
    float *InputData = new float [NUMELEMENTS];
    // read the full input data into InputData from file
    MPI_Scatter(InputData, NUMELEMENTS, MPI_FLOAT, myData, localSize, MPI_FLOAT, BOSS, MPI_WORLD_COMM);
}
else
// this is the receiver
{
    MPI_Scatter(NULL, 0, MPI_FLOAT, myData, localSize, MPI_FLOAT, BOSS, MPI_WORLD_COMM);
}
```

**MPI Gather**

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

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

- `# elements to send`
- `# elements to receive`

Both the sender and receivers need to execute `MPI_Gather`. There is no separate receive function.

**MPI Gather Example**

```c
#define NUMELEMENTS ????
int numCPUs;
int me;
#define BOSS 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 == BOSS)
// this is the receiver
{
    float *InputData = new float [NUMELEMENTS];
    // write data from Array to disk
    MPI_Gather(myData, localSize, MPI_FLOAT, InputData, NUMELEMENTS, MPI_FLOAT, BOSS, MPI_WORLD_COMM);
}
else
// this is the sender
{
    MPI_Gather(NULL, 0, MPI_FLOAT, myData, localSize, MPI_FLOAT, BOSS, MPI_WORLD_COMM);
}
```
A Full MPI Scatter / Gather Example

```c
#define NUMELEMENTS ????
int numCPUs;
int me;
define BOSS 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 == BOSS )
    // send over, 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, BOSS, MPI_WORLD_COMM );
        <<< do some computing on myData[] >>
        MPI_Gather( myData, localSize, MPI_FLOAT, InputData, NUMELEMENTS, MPI_FLOAT, BOSS, MPI_WORLD_COMM );
        <<< write data from InputData[] to disk >>
    }
else
    // receive, then send back
    {
        MPI_Scatter( NULL, 0, MPI_FLOAT, myData, localSize, MPI_FLOAT, BOSS, MPI_WORLD_COMM );
        <<< do some computing on myData[] >>
        MPI_Gather( myData, localSize, MPI_FLOAT, NULL, 0, MPI_FLOAT, BOSS, MPI_WORLD_COMM );
    }
```

A Full MPI Scatter / Gather Example

MPI Reduction Example

```c
int numCPUs;
int me;
float globalSum;
define BOSS 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, BOSS, MPI_COMM_WORLD );
```

MPI Reduction Example

MPI Reduction

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

MPI Reduction

MPI Barriers

```c
MPI_Barrier( MPI_COMM_WORLD );
```

MPI Barriers

Distributed Processors:

```
0 1 2 3 4 5
```

Barrier

All CPUs must execute a call to MPI_Barrier( ) before any of the CPUs can move past it.
That is, each CPU's MPI_Barrier( ) blocks until all CPUs execute a call to MPI_Barrier( ).
**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
define struct point {
    int pointSize;
    float x, y, z;
};
```

```c
MPI_Datatype MPI_POINT;
int blocklengths[] = {1, 1, 1, 1};
int displacements[] = {0, 4, 8, 12};
MPI_type types[] = {MPI_INT, MPI_FLOAT, MPI_FLOAT, MPI_FLOAT};
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 Timing**

- `double MPI_Wtick();`
  Returns the resolution of the clock, in seconds.

- `double MPI_Wtime();`
  Returns the time, in seconds, since “some time in the past”.

*The clocks on the different CPUs are not guaranteed to be synchronized.*

---

**And now: Parallelism Conclusion**

Reviewing what you know!

---

**Suppose We Have This Setup**

Diagram showing network, CPU, memory, and GPU, with different core counts and SSE support.
Welcome to Parallelism Jeopardy!

I'll take CS 475/575 for $800, Alex.

In a multi-CPU distributed system, this is the total number of different kinds of parallelisms that we covered this quarter.

1. Multicore OpenMP
2. CPU SIMD
3. GPU
What is “4”?

This is how modern supercomputers work!
And you now know something about using all 4 – congratulations!

1. Multicore OpenMP
2. CPU SIMD
3. GPU
4. MPI

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