### 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

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

#### 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

[Logo of Open MPI Consortium]

#### MPI: The Basic Idea

<table>
<thead>
<tr>
<th>Network</th>
<th>Memory</th>
<th>CPU</th>
<th>...</th>
<th>Memory</th>
<th>CPU</th>
</tr>
</thead>
</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 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

% mpicc -o program program.c ...

or

% mpicc++ -o program program.cpp ...

% 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 (SPMD) Model

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

A communicator of deer!

A communicator of turkeys!

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 to go do it.

A Good Place to Start: MPI Broadcasting

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

A Good Place to Start: MPI Broadcasting

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

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

# elements

MPI_CHAR
MPI_INT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE

rank of the CPU doing the sending
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.

\[
\Delta T = \frac{k}{\rho C} \left( \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2} \right)
\]

int numCPUs;
int me;
float k_over_rho_c; // the ROOT node will know this value, the others won't (yet)

#define ROOT 0
MPI_Comm_Size(MPI_COMM_WORLD, &numCPUs); // how many are in this communicator
MPI_Comm_Rank(MPI_COMM_WORLD, &me); // which one am I?

if( me == ROOT )
{
    \text{read k_over_rho_c from the data file}
    MPI_Bcast( &k_over_rho_c, 1, MPI_FLOAT, ROOT, MPI_COMM_WORLD ); // send
}
else
{
    MPI_Bcast( &k_over_rho_c, 1, MPI_FLOAT, ROOT, MPI_COMM_WORLD ); // receive
}

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.
- 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

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
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 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
", in, me );
}
```

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

Example

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

```
MPI_CHAR
MPI_INT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE
```

Type = MPI_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

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 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
}
```

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.

```
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

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

- **# elements to send**
  - MPI_CHAR
  - MPI_INT
  - MPI_LONG
  - MPI_FLOAT
  - MPI_DOUBLE
  - ...

- **# elements to receive**
  - MPI_CHAR
  - MPI_INT
  - MPI_LONG
  - MPI_FLOAT
  - MPI_DOUBLE
  - ...

**Signifies “no data to send”**

It is typical that you scatter to yourself
MPI Gather

\[
\text{MPI} \_\text{Gather}( \text{snd} \_\text{array}, \text{snd} \_\text{count}, \text{snd} \_\text{type}, \text{rcv} \_\text{array}, \text{rcv} \_\text{count}, \text{rcv} \_\text{type}, \text{dst}, \text{MPI} \_\text{COMM} \_\text{WORLD})
\]

# elements to send
MPI_CHAR
MPI_INT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE

# elements to receive
MPI_CHAR
MPI_INT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE

A Full MPI Scatter / Gather Example

\[
\text{MPI} \_\text{Gather}( \text{myData}, \text{localSize}, \text{MPI} \_\text{FLOAT}, \text{ NULL}, \text{0}, \text{MPI} \_\text{FLOAT}, \text{ROOT}, \text{MPI} \_\text{COMM} \_\text{WORLD})
\]

Who is given the final answer
Both the sender and receivers need to execute MPI_Reduce -- there is no separate receive function

MPI Reduction

\[
\text{MPI} \_\text{Reduce}( \text{partialResult}, \text{globalResult}, \text{count}, \text{type}, \text{operator}, \text{dst}, \text{MPI} \_\text{COMM} \_\text{WORLD})
\]

Number of elements in the partial result
Where the partial result is stored on each CPU
Place to store the full result on the dst CPU

MPI Reduction Example

```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.;
MPI_Reduce(&partialSum, &globalSum, 1, MPI_FLOAT, MPI_SUM, ROOT, MPI_COMM_WORLD);
```

**MPI Barriers**

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

**MPI Timing**

```c
double MPI_Wtick();

double MPI_Wtime();
```

**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 MPI_POINT;
```

```c
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.
Suppose We Have This Setup

Welcome to Parallelism Jeopardy!

IN A MULTI-CPU SYSTEM, THIS IS THE TOTAL NUMBER OF DIFFERENT PARALLELISMS THAT WE COVERED THIS QUARTER

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

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

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

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