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.

The Open MPI Consortium

http://www.open-mpi.org

Network

CPU

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 conditioned ahead of time by having the MPI server code installed on it. Each secondary MPI CPU must also have an integer ID assigned to it (called the rank) and must be registered with the primary MPI CPU.
Compiling and Running

```
% mpicc -o program program.c...
```

or

```
% mpic++ -o program program.cpp...
```

```
% mpiexec --np 64 program
```

# of processors to use

---

Setting Up and Finishing

```c
#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.

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.

---

A Good Place to Start: MPI Broadcasting

```
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

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

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

Send the value of \( k/\rho C \) from the data file.

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 );
MPI_Comm_Rank( MPI_COMM_WORLD, &me );

if( me == ROOT ) 
{ 
  // read \( k/\rho C \) from the data file
  MPI_Bcast( &k_over_rho_c, 1, MPI_FLOAT, ROOT, MPI_COMM_WORLD );
}
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

\[ \text{MPI Send( array, numToSend, type, dst, tag, MPI_COMM_WORLD );} \]

address of data to send from 
number of elements 

rank of the CPU to send to 

An integer to differentiate this transmission from any other transmission (be sure this is unique!)

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

\[ \text{MPI Recv( array, maxCanReceive, type, src, tag, MPI_COMM_WORLD, &status );} \]

address of data to receive into 

number of elements we can receive, at most 

MPI_CHAR MPI_INT MPI_FLOAT MPI_DOUBLE ***

Rank of the CPU we are expecting to get a transmission from 

An integer to differentiate what transmission we are looking for with this call (be sure this matches what the sender is sending!)

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.
Computer Graphics

### 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++) {
        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);
```

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

```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;
```

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

- **Type** = MPI_Status
  - If elements we can receive, at most
    - MPI_CHAR
    - MPI_INT
    - MPI_LONG
    - MPI_FLOAT
    - MPI_DOUBLE

Another Example

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

```
#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 = 1; dst < numCPUs; dst++) {
        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 term: Scatter/Gather, and has its own MPI function calls.

MPI Scatter Example

```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

```c
MPI_Gather( snd_array, snd_count, snd_type, rcv_array, rcv_count, rcv_type, dst, MPI_COMM_WORLD );
```
```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_Gather(myData, localSize, MPI_FLOAT, InputData, NUMELEMENTS, 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 {
  MPI_Scatter(NULL, 0, MPI_FLOAT, myData, localSize, MPI_FLOAT, ROOT, MPI_WORLD_COMM);
  // do some computing on mydata[]
  MPI_Scatter(NULL, 0, MPI_FLOAT, myData, localSize, MPI_FLOAT, ROOT, MPI_WORLD_COMM);
  // write data from InputData[] to disk
}
```

**MPI Reduction Example**

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

Number of elements in the partial result

- MPI_CHAR
- MPI_INT
- MPI_LONG
- MPI_FLOAT
- MPI_DOUBLE
- MPI_MAX
- MPI_MIN
- MPI_PROD
- MPI_MINLOC
- MPI_MAXLOC

Who is given the final answer

- MPI_CHAR
- MPI_INT
- MPI_LONG
- MPI_FLOAT
- MPI_DOUBLE
- MPI_MAX
- MPI_MIN
- MPI_PROD
- MPI_MINLOC
- MPI_MAXLOC

Where the partial result is stored on each CPU

Place to store the full result on the dst CPU

- MPI_CHAR
- MPI_INT
- MPI_LONG
- MPI_FLOAT
- MPI_DOUBLE
- MPI_MAX
- MPI_MIN
- MPI_PROD
- MPI_MINLOC
- MPI_MAXLOC

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

**MPI Reduction**

**MPI Gather Example**

```c
enemySize = NUMELEMENTS / numCPUs;
float *myData = new float [enemySize];

if (me == ROOT) {
  float *InputData = new float [NUMELEMENTS];
  // read the full input data into InputData[] from disk
  MPI_Gather(myData, enemySize, MPI_FLOAT, InputData, NUMELEMENTS, MPI_FLOAT, ROOT, MPI_WORLD_COMM);
  // write data from Array to disk
} else {
  MPI_Gather(myData, enemySize, MPI_FLOAT, NULL, 0, MPI_FLOAT, ROOT, MPI_WORLD_COMM);
}
```

**A Full MPI Scatter / Gather Example**

```c
MPI_Scatter(InputData, NUMELEMENTS, MPI_FLOAT, myData, localSize, MPI_FLOAT, ROOT, MPI_WORLD_COMM);
```

**MPI Reduction**

```c
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
MPI Barriers

MPI_Barrier( MPI_COMM_WORLD );

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.

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

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 types[ ] = { 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 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.
What is “4”? and, they can all be functioning within the same application!