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.

The Open MPI Consortium

MPI: The Basic Idea

Network

Memory

CPU

Memory

CPU

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

C

C++

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_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 );
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.

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:

```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 );
```

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

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

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:

```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 );
```

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

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

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

A Good Place to Start: MPI Broadcasting

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

Confused? Look at this Diagram

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

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

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

A Good Place to Start: MPI Broadcasting

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

Confused? Look at this Diagram

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

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

```c
MPI_Bcast( array, count, type, src, MPI_COMM_WORLD );
```
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 reused.
- There are no guarantees on order from different src's.

Example

```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\n", myData, me);
}
```

MPI Status

- Integer to differentiate this transmission from any other transmission (be sure this is unique).
- Rank of the CPU to send to.
- Type = MPI_Status

Example

```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
{
  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\n", myData, me);
}
```

Look at this Diagram

- Source
- Destinations
- # elements we can receive, at most

How Much Data Did I Actually Receive?

```c
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_Rcv()?

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[dst*localSize], 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:

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 :-)
It is typical that you scatter to yourself else // receive, then send back

\[
\text{if( me == BOSS )}
\]

// send over, then receive

\[
\text{float *myData = new float [ localSize ];}
\]

Int localSize = NUMELEMENTS / numCPUs; // assuming it comes out evenly

MPI_Comm_Rank( MPI_COMM_WORLD, &me);

MPI_Comm_Size( MPI_COMM_WORLD, &numCPUs);

#define BOSS 0

int me;

int numCPUs;

#define NUMELEMENTS ??????

This is each dst's local data to be received

This is all of the global data to be received

This is all of the global data to be sent

This is each dst's local data to be sent

Signifies "no data to send"

Signifies "no data to receive"

Both the sender and receivers need to execute MPI_Scatter. There is no separate receive function
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_Barrier( MPI_COMM_WORLD );

MPI Type_create_struct( count, blocklengths, displacements, types, datatype );
struct point
{
    int pointSize;
    float x, y, z;
};
MPI_Datatype
MPI_Point;
int blocklengths[ ]    = { 1, 1, 1, 1 };
int displacements[ ] = { 0, 4, 8, 12 };
int 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

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!
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
4. MPI

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

This is how modern supercomputers work! And you now know something about using all 4 – congratulations! and, they can all be active within the same application!