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

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

Oregon State University
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. This is the MPI that the COE systems use

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

The Open MPI Consortium

Computer Graphics

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).
This paradigm is how modern supercomputers work!

The Texas Advanced Computing Center's Frontera supercomputer

How to SSH to the COE MPI Cluster

`ssh` over to an MPI submission machine -- `submit-a` and `submit-b` will also work

*flip3 151% ssh submit-c@hpc.engr.oregonstate.edu*

`submit-c` 142% module load slurm

`submit-c` 143% module load openmpi

Type these two lines right away to set your paths correctly

BTW, you can find out more about the COE cluster here:
https://it.engineering.oregonstate.edu/hpc

“The College of Engineering HPC cluster is a heterogeneous mix of 202 servers providing over 3600 CPU cores, over 130 GPUs, and over 31 TB total RAM. The systems are connected via gigabit ethernet, and most of the latest servers also utilize a Mellanox EDR InfiniBand network connection. The cluster also has access to 100TB global scratch from the College of Engineering’s Dell/EMC Isilon enterprise storage.”

Compiling and Running

C

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

C++

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

# of processors to use

All distributed processors execute the same program at the same time

Warning – use `mpic++` and `mpiexec` !

Don’t use g++ and don’t run by just typing the name of the executable!

Running with a `bash` Batch Script

```
submit.bash:
#!/bin/bash
#SBATCH -J AutoCorr
#SBATCH -A cs475-575
#SBATCH -p class
#SBATCH -N 4      # number of nodes
#SBATCH -n 4      # number of tasks
#SBATCH --constraint=ib
#SBATCH -o autocorr.out
#SBATCH -e autocorr.err
#SBATCH --mail-type=END,FAIL
#SBATCH --mail-user=joeparallel@cs.oregonstate.edu
module load openmpi
mpic++ autocorr.cpp -o autocorr -lm
mpiexec -mca btl self.tcp -np 4 ./autocorr
```

```
submit-c 143% sbatch submit.bash
Submitted batch job 258759
```
Auto-Notifications via Email

#SBATCH --mail-user=joeparallel@oregonstate.edu

You don't have to ask for email notification, but if you do, please, please, please be sure you get your email address right!

The IT people are getting real tired of fielding the bounced emails when people spell their own email address wrong.

Use slurm's scancel if your Job Needs to Be Killed

submit-c 143% sbatch submit.bash
Submitted batch job 258759

submit-c 144% scancel 258759

Setting Up and Finishing

#include <mpi.h>

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );

    /*
    * Get information about our place in the communicator:
    *
    * Size, i.e., how many altogether?
    * Rank, i.e., which one am I?
    */
    MPI_Comm_size( MPI_COMM_WORLD, &numCPUs);
    MPI_Comm_rank( MPI_COMM_WORLD, &me);

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

    // Get the communicator of deer!
    // Get the communicator of turkeys!

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.

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.

Oh, look, a communicator of deer!
Oh, look, a communicator of turkeys!
A First Test of MPI

```c
#include <stdio.h>
#include <math.h>
#include <mpi.h>
#define BOSS 0

int main( int argc, char* argv[] )
{
    MPI_Init( &argc, &argv );
    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 );
    if( me == BOSS )
        fprintf( stderr, "Rank %d says that we have a Communicator of size %d
", BOSS, numCPUs );
    else
        fprintf( stderr, "Welcome from Rank %d
", me );
    MPI_Finalize( );
    return 0;
}
```

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 Good Place to Start: MPI Broadcasting

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

Both the sender and receivers need to execute `MPI_Bcast` — there is no separate receive function.
int numCPUs;
int me;
float k_over_rho_c;  // the BOSS node will know this value, the others won't (yet)

#define BOSS 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 == BOSS )
{
    // read k_over_rho_c from the data file
}

MPI_Bcast( &k_over_rho_c, 1, MPI_FLOAT, BOSS, MPI_COMM_WORLD ); // send if BOSS, and receive if not

This is our heat transfer equation
\[ \Delta T = \frac{k}{\rho c} \left( \frac{T_{i+1} - 2T_i + T_{i-1}}{(\Delta x)^2} \right) \Delta t \]

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

Node #BOSS:

Executable code
k_over_rho_c (set)

All Nodes that are not #BOSS:

Executable code
k_over_rho_c (being set)

Sending Data from One Source CPU to One Destination CPU

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

address of data to send from

# elements
(note: this is the number of elements, not the number of bytes)

MPI_CHAR
MPI_INT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE

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.

How Does this Work?

Think Star Trek Wormholes!

Confused? Look at this Diagram

Sending Data from One Source CPU to One Destination 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

Receiving Data in a Destination CPU from a Source CPU

src node ———— dst node

Look at this Diagram

Executable code | Input Data

How does MPI let the Sender perform an MPI_Send() even if the Receivers are not ready to MPI_Recv()?

Sender

MPI_Send() | Destination

Receiver

MPI_Recv() | Source

MyData

Be sure the receiving tag matches the sending tag

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, 'B', MPI_COMM_WORLD );
    }
  } // end for
}
else // a secondary
{
  MPI_Recv( myData, MYDATA_SIZE, MPI_CHAR, BOSS, 'B', MPI_COMM_WORLD, MPI_STATUS_IGNORE );
  printf( " '%s' from rank # %d\n", 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 Diagram
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 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 sender
{
    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 receiver
{
    MPI_Recv( myData, localSize, MPI_FLOAT, BOSS, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
    // do something with this subset of the data
}
```

Another Example

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

\[ \text{MPI\_Scatter}( \text{snd\_array}, \text{snd\_count}, \text{snd\_type}, \text{rcv\_array}, \text{rcv\_count}, \text{rcv\_type}, \text{src}, \text{MPI\_COMM\_WORLD} ); \]

- The total large array to split up
- Local array to store this processor’s piece in
- This is who is doing the sending – everyone else is receiving
- \# elements to send per-processor

Both the sender and receivers need to execute MPI\_Scatter. There is no separate receive function

MPI Gather

The total large array to put the pieces back into

\[ \text{MPI\_Gather}( \text{snd\_array}, \text{snd\_count}, \text{snd\_type}, \text{rcv\_array}, \text{rcv\_count}, \text{rcv\_type}, \text{dst}, \text{MPI\_COMM\_WORLD} ); \]

- Local array that this processor is sending back
- \# elements to return per-processor
- This is who is doing the receiving – everyone else is sending

Both the sender and receivers need to execute MPI\_Gather. There is no separate receive function

Remember This? It’s Baaaaaack as a complete Scatter/Gather Example

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 very common. That’s why MPI has its own scatter and gather functions.

heat.cpp

```c
#include <stdio.h>
#include <math.h>
#include <mpi.h>

const float RHO = 8050.;
const float C   = 0.466;
const float K   = 20.;
float k_over_rho_c = K / (RHO*C); // units of m^2/sec  NOTE: this cannot be a const!
const float DX    = 1.0;
const float DT    = 1.0;
#define BOSS 0
#define NUMELEMENTS (8*1024*1024)
define NUM\_TIME\_STEPS 4
define DEBUG false

float * NextTemps; // per-processor array to hold computer next-values
int NumCpus; // total # of cpus involved
int PPSize; // per-processor local array size
float * PPTemps; // per-processor local array temperature data
float * TempData; // the overall NUMELEMENTS-big temperature data

void DoOneTimeStep( int );
```
```c
int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    int me; // which one I am
    MPI_Comm_size( MPI_COMM_WORLD, &NumCpus );
    MPI_Comm_rank( MPI_COMM_WORLD, &me );
    // decide how much data to send to each processor:
    PPSize = NUMELEMENTS / NumCpus; // assuming it comes out evenly
    NextTemps = new float[PPSize]; // all processors now have this uninitialized local array
    // broadcast the constant:
    MPI_Bcast( (void *)&k_over_rho_c, 1, MPI_FLOAT, BOSS, MPI_COMM_WORLD );

    if( me == BOSS ) // this is the data-creator
    {
        TempData = new float[NUMELEMENTS];
        for( int i = 0; i < NUMELEMENTS; i++ )
            TempData[i] = 0.;
        TempData[NUMELEMENTS/2] = 100.;
    }
    MPI_Scatter( TempData, PPSIZE, MPI_FLOAT, NextTemps, PPSIZE, MPI_FLOAT, BOSS, MPI_COMM_WORLD );

    // all the NextTemps arrays have now been filled
    // do the time steps:
    double time0 = MPI_Wtime();
    for( int steps = 0; steps < NUM_TIME_STEPS; steps++ )
    {
        // do the computation for one time step:
        DoOneTimeStep( me );
        // ask for all the data:
        #ifdef WANT_EACH_TIME_STEPS_DATA
            MPI_Gather( NextTemps, PPSize, MPI_FLOAT, TempData, PPSIZE, MPI_FLOAT, BOSS, MPI_COMM_WORLD );
        #endif
        #ifdef WANT_EACH_TIME_STEPS_DATA
            MPI_Gather( TempData, PPSIZE, MPI_FLOAT, NextTemps, PPSIZE, MPI_FLOAT, BOSS, MPI_COMM_WORLD );
        #endif
        double time1 = MPI_Wtime();
    }

    if( me == BOSS )
    {
        double seconds = time1 - time0;
        double performance = (double)NUM_TIME_STEPS * (double)NUMELEMENTS / seconds / 1000000.; // mega-elements computed per second
        fprintf( stderr, "%3d, %10d, %8.2lf
", NumCpus, NUMELEMENTS, performance );
    }
    MPI_Finalize();
    return 0;
}
```
void DoOneTimeStep( int me )
{
    MPI_Status status;
    // send out the left and right end values:
    // (the tag is from the point of view of the sender)
    if( me != 0 ) // i.e., if I'm not the first group on the left
        // send my PPTemps[0] to me-1 using tag 'L'
        MPI_Send( &PPTemps[0], 1, MPI_FLOAT, me-1, 'L', MPI_COMM_WORLD );
        if( DEBUG )   fprintf( stderr, "%3d sent 'L' to %3d
",
    if( me != NumCpus-1 ) // i.e., not the last group on the right
        // send my PPTemps[PPSize-1] to me+1 using tag 'R'
        MPI_Send( &PPTemps[PPSize-1], 1, MPI_FLOAT, me+1, 'R', MPI_COMM_WORLD );
        if( DEBUG )  fprintf( stderr, "%3d sent 'R' to %3d
",
    if( me != 0 ) // i.e., if I'm not the first group on the left
        // receive my "left" from me-1 using tag 'R'
        MPI_Recv( &left, 1, MPI_FLOAT, me-1, 'R', MPI_COMM_WORLD, &status );
        if( DEBUG )   fprintf( stderr, "%3d received 'R' from %3d
",
    if( me != NumCpus-1 ) // i.e., not the last group on the right
        // receive my "right" from me+1 using tag 'L'
        MPI_Recv( &right, 1, MPI_FLOAT, me+1, 'L', MPI_COMM_WORLD, &status );
        if( DEBUG )    fprintf( stderr, "%3d received 'L' from %3d
",

1D Compute-to-Communicate Ratio

Compute : Communicate ratio = N : 2
where N is the number of compute cells per processor

In the above drawing, Compute : Communicate is 4 : 2
// first element on the left (0):
{
  float dtemp = ( k_over_rho_c * 
                  ( left - 2.*PPTemps[0] + PPTemps[1] ) / ( DX*DX ) ) * DT;
  NextTemps[0] = PPTemps[0] + dtemp;
}

// all the nodes in the middle:
for( int i = 1; i < PPSize-1; i++ )
{
  float dtemp = ( k_over_rho_c * 
                  ( PPTemps[i-1] - 2.*PPTemps[i] + PPTemps[i+1] ) / ( DX*DX ) ) * DT;
  NextTemps[i] = PPTemps[i] + dtemp;
}

// last element on the right (PPSize-1):
{
  float dtemp = ( k_over_rho_c * 
                  ( PPTemps[PPSize-2] - 2.*PPTemps[PPSize-1] + right ) / ( DX*DX ) ) * DT;
  NextTemps[PPSize-1] = PPTemps[PPSize-1] + dtemp;
}

// update the local dataset:
for( int i = 0; i < PPSize; i++ )
{
  PPTemps[i] = NextTemps[i];
}
**MPI Performance**

Mega-Elements Computed Per Second vs. Number of Processors

- Number of Processors
- Number of Elements

**Using MPI and OpenMP on 13,680 nodes (437,760 cores) of the Cray XE6 at NCSA at the University of Illinois**

From: Peter Johnsen, Mark Steka, Melvyn Shapiro, Alan Norton, Thomas Galarneau, Petascale WRF Simulation of Hurricane Sandy.

**MPI Reduction**

MPI Reduction Example

// gratuitous use of a reduce -- average all the temperatures:
float partialSum = 0.;
tfor (i = 0; i < PPSize; i++)
partialSum += PPTemps[i];
float globalSum = 0.;
MPI_Reduce (&partialSum, &globalSum, 1, MPI_FLOAT, MPI_SUM, BOSS, MPI_COMM_WORLD);
if (me == BOSS)
fprintf(stderr, "Average temperature = %f\n", globalSum/(float)NUMELEMENTS);
MPI Barriers

`MPI_Barrier(MPI_COMM_WORLD);`

All CPUs must execute the 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().

Distributed Processors:

0 1 2 3 4 5

Barrier

MPI Timing

```c
double MPI_Wtick();
```

Returns the resolution of the clock, in seconds.

```c
double MPI_Wtime();
```

Returns the time, in seconds, since "some time in the past".

*Warning: the clocks on the different CPUs are not guaranteed to be synchronized!*

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;
};
```

```c
MPI_Datatype MPI_POINT;
```

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

```c
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 Status-Checking

Some MPI calls have a `&status` in their argument list. The `status` argument is declared to be of type `MPI_Status`, which is defined like this:

```c
typedef struct _MPI_Status {
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
} MPI_Status;
```

- `MPI_SOURCE` is the rank of the node who sent this
- `MPI_TAG` is the tag used during the send
- `MPI_ERROR` is the error number that occurred

Example:

```c
MPI_Status status;
MPI_Recv(myData, MYDATA_SIZE, MPI_CHAR, BOSS, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
fprintf(stderr, "Tag = %d\n", status.MPI_TAG );
```
Example: Autocorrelation

**Autocorrelation – More than Just a Scatter**

<table>
<thead>
<tr>
<th>NUMELEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
</tr>
<tr>
<td>50</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

Divide NUMELEMENTS into pieces for the NumCPus (This is what MPI_Scatter does)

But, in the Autocorrelation case, we need MAXSHIFTS more data values for each CPU.

**Autocorrelation – How the Shifting Works**

<table>
<thead>
<tr>
<th>Shift</th>
<th>NUMELEMENTS</th>
<th>MAXSHIFTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>20</td>
</tr>
</tbody>
</table>