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
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
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

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

submit-c 142% module load slurm
submit-c 143% module load openmpi/3.1

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

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 from the Command Line

% mpicc -o program program.c . . .

or

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

% mpiexec -mca btl self,tcp -np 4 program

All distributed processors execute the same program at the same time

# of processors to use

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:

```bash
#!/bin/bash
#SBATCH -J Heat
#SBATCH -A cs475-575
#SBATCH -p class
#SBATCH -N 8       # number of nodes
#SBATCH -n 8       # number of tasks
#SBATCH -o heat.out
#SBATCH -e heat.err
#SBATCH --mail-type=END,FAIL
#SBATCH --mail-user=joeparallel@cs.oregonstate.edu
module load openmpi/3.1
mpic++ heat.cpp -o heat -lm
mpiexec -mca btl self,tcp -np 4 heat
```

**submit-c 143% sbatch submit.bash**
Submitted batch job 258759
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 slurms’s `scancel` if your Job Needs to Be Killed

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

```
submit-c 144%  scancel 258759
```
#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 );
A **communicator** is a collection of CPUs that are capable of sending messages to each other.

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 );
```

**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.

This requires MPI server code getting installed on all those CPUs. Only an administrator can do this.
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\n", BOSS, numCPUs );
    else
        fprintf( stderr, "Welcome from Rank %d\n", me );

    MPI_Finalize( );
    return 0;
}
```
submit-c 165% mpiexec -np 16 ./first
Welcome from Rank 13
Welcome from Rank 15
Welcome from Rank 3
Welcome from Rank 7
Welcome from Rank 5
Welcome from Rank 8
Welcome from Rank 9
Welcome from Rank 11
Welcome from Rank 1
Rank 0 says that we have a Communicator of size 16
Welcome from Rank 1
Welcome from Rank 12
Welcome from Rank 14
Welcome from Rank 6
Welcome from Rank 10
Welcome from Rank 4

submit-c 166% mpiexec -np 16 ./first
Welcome from Rank 13
Welcome from Rank 15
Welcome from Rank 7
Welcome from Rank 9
Welcome from Rank 11
Welcome from Rank 13
Welcome from Rank 15
Rank 0 says that we have a Communicator of size 16
Welcome from Rank 2
Welcome from Rank 3
Welcome from Rank 4
Welcome from Rank 6
Welcome from Rank 8
Welcome from Rank 12
Welcome from Rank 14
Welcome from Rank 10

submit-c 167% mpiexec -np 16 ./first
Welcome from Rank 9
Welcome from Rank 11
Welcome from Rank 13
Welcome from Rank 7
Welcome from Rank 1
Welcome from Rank 3
Welcome from Rank 10
Welcome from Rank 15
Welcome from Rank 4
Welcome from Rank 5
Rank 0 says that we have a Communicator of size 16
Welcome from Rank 2
Welcome from Rank 6
Welcome from Rank 8
Welcome from Rank 14
Welcome from Rank 12

submit-c 168% mpiexec -np 16 ./first
Welcome from Rank 13
Welcome from Rank 15
Welcome from Rank 7
Welcome from Rank 3
Welcome from Rank 5
Welcome from Rank 9
Welcome from Rank 11
Welcome from Rank 1
Welcome from Rank 12
Welcome from Rank 14
Welcome from Rank 4
Welcome from Rank 2
Rank 0 says that we have a Communicator of size 16
Welcome from Rank 8
Welcome from Rank 10
Welcome from Rank 6
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

\[ \text{MPI\_Bcast}( \text{array}, \text{count}, \text{type}, \text{src}, \text{MPI\_COMM\_WORLD} ); \]

- Address of data to send from if you are the src node;
- Address of the data to receive into if you are not
- # elements
- MPI\_CHAR
- MPI\_INT
- MPI\_LONG
- MPI\_FLOAT
- MPI\_DOUBLE

Both the sender and receivers need to execute \text{MPI\_Bcast} – there is no separate receive function
This is our heat transfer equation from before. Clearly, every CPU will need to know this value.

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

```c
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
```

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

MPI Broadcast Example
Confused? Look at this Diagram

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

Executable code: `k_over_rho_c (set)`

**Node #BOSS:**

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

**All Nodes that are not #BOSS:**

<table>
<thead>
<tr>
<th>Executable code</th>
<th>k_over_rho_c (being set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executable code</td>
<td>k_over_rho_c (being set)</td>
</tr>
<tr>
<td>Executable code</td>
<td>k_over_rho_c (being set)</td>
</tr>
<tr>
<td>Executable code</td>
<td>k_over_rho_c (being set)</td>
</tr>
<tr>
<td>Executable code</td>
<td>k_over_rho_c (being set)</td>
</tr>
</tbody>
</table>
How Does this Work?
Think Star Trek Wormholes!
MPI_Send( array, numToSend, type, dst, tag, MPI_COMM_WORLD );

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

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.

src node  dst node
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

 MPI_CHAR
 MPI_INT
 MPI_LONG
 MPI_FLOAT
 MPI_DOUBLE

# elements we can receive, at most

Type = MPI\_Status

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!). I like to use chars.

### 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

---

src node \[\text{[ ]}\] \rightarrow \text{dst node}
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 );
        }
    }
}
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.
Look at this Diagram

```c
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}` in, me ");
}
```
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 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

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

```c
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
}
```
In Distributed Computing, You Often Hear About These Design Patterns

Broadcast

Scatter

Gather
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

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

The total large array to split up

# elements to send per-processor

The total large array to split up

# elements to send per-processor

Local array to store this processor's piece in

# elements to receive per-processor

This is who is doing the sending – everyone else is receiving

Both the sender and receivers need to execute MPI_Scatter.
There is no separate receive function
MPI_Gather( snd_array, snd_count, snd_type, rcv_array, rcv_count, rcv_type, dst, MPI_COMM_WORLD );

The total large array to put the pieces back into

# elements to return per-processor

Local array that this processor is sending back

M_PI_CHAR
M_PI_INT
M_PI_LONG
M_PI_FLOAT
M_PI_DOUBLE

# elements to send back per-processor

M_PI_CHAR
M_PI_INT
M_PI_LONG
M_PI_FLOAT
M_PI_DOUBLE

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

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 very common. That’s why MPI has its own scatter and gather functions.
```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!
// K / (RHO*C) = 5.33x10^-6 m^2/sec

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
    PPTemps  = new float [PPSize];       // all processors now have this uninitialized Local array
    NextTemps = new float [PPSize];      // all processors now have this uninitialized local array too

    // 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, PPTemps, PPSize, MPI_FLOAT, BOSS, MPI_COMM_WORLD );
all the PPTemps 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( PPTemps, PPSize, MPI_FLOAT, TempData, PPSize, MPI_FLOAT,
                BOSS, MPI_COMM_WORLD );
    #endif
}

    #ifndef WANT_EACH_TIME_STEPS_DATA
    MPI_Gather( PPTemps, PPSize, MPI_FLOAT, TempData, 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\n", 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\n", me, me-1 );
    }

    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\n", me, me+1 );
    }
}
```c
float left = 0.;
float right = 0.;

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\n", me, me-1 );
}

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\n", me, me+1 );
}
```
Sharing Values Across the Boundaries

1 sent 'L' to 0
1 sent 'R' to 2
2 sent 'L' to 1
2 sent 'R' to 3
2 received 'R' from 1
0 sent 'R' to 1
0 received 'L' from 1
1 received 'R' from 0
1 received 'L' from 2
3 sent 'L' to 2
3 received 'R' from 2
2 received 'L' from 3
1D Compute-to-Communicate Ratio

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 Elements

Number of Elements

Number of Processors

Oregon State University
Computer Graphics
Low Dataset-Size MPI Performance

Mega-Elements Computed Per Second vs. Number of Elements

Number of Elements

Mega-Elements Computed Per Second

Number of Processors
MPI Performance

Mega-Elements Computed Per Second vs. Number of Processors

- Number of Processors
- Number of Elements

Mega-Elements Computed Per Second

- 1024
- 8192
- 65536
- 524288
- 4194304
- 33554432

Oregon State University
Computer Graphics
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 Straka, Melvyn Shapiro, Alan Norton, Thomas Galarneau, 
*Petascale WRF Simulation of Hurricane Sandy.*

![Graph showing performance metrics for WRF Hurricane Sandy Simulation on Cray XE6 Blue Waters.](image)
MPI Reduction

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

Where the partial result is stored on each CPU

Place to store the full result on the dst CPU

Number of elements in the partial result

MPI_MIN
MPI_MAX
MPI_SUM
MPI_PROD
MPI_MINLOC
MPI_MAXLOC
MPI_LAND
MPI_BAND
MPI_LOR
MPI_BOR
MPI_LXOR
MPI_BXOR

This really should be called Scatter/Gather/Reduction

Both the sender and receivers need to execute MPI_Reduce. There is no separate receive function
MPI Reduction Example

// gratuitous use of a reduce -- average all the temperatures:

float partialSum = 0.;
for( int 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 );
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()`.
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);
```

```c
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”.

Warning: the clocks on the different CPUs are not guaranteed to be synchronized!
MPI Status-Checking

Some MPI calls have a \texttt{&status} in their argument list.

The \texttt{status} argument is declared to be of type \texttt{MPI_Status}, which is defined like this:

\begin{verbatim}
typedef struct _MPI_Status
{
   int MPI_SOURCE;
   int MPI_TAG;
   int MPI_ERROR;
} MPI_Status;
\end{verbatim}

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

\textbf{Example:}
\begin{verbatim}
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 );
\end{verbatim}
### MPI Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_SUCCESS</td>
<td>No error</td>
</tr>
<tr>
<td>MPI_ERR_BUFFER</td>
<td>Invalid buffer pointer</td>
</tr>
<tr>
<td>MPI_ERR_COUNT</td>
<td>Invalid count argument</td>
</tr>
<tr>
<td>MPI_ERR_TYPE</td>
<td>Invalid datatype argument</td>
</tr>
<tr>
<td>MPI_ERR_TAG</td>
<td>Invalid tag argument</td>
</tr>
<tr>
<td>MPI_ERR_COMM</td>
<td>Invalid communicator</td>
</tr>
<tr>
<td>MPI_ERR_RANK</td>
<td>Invalid rank</td>
</tr>
<tr>
<td>MPI_ERR_REQUEST</td>
<td>Invalid request (handle)</td>
</tr>
<tr>
<td>MPI_ERR_ROOT</td>
<td>Invalid root</td>
</tr>
<tr>
<td>MPI_ERR_GROUP</td>
<td>Invalid group</td>
</tr>
<tr>
<td>MPI_ERR_OP</td>
<td>Invalid operation</td>
</tr>
<tr>
<td>MPI_ERR_TOPOLOGY</td>
<td>Invalid topology</td>
</tr>
<tr>
<td>MPI_ERR_DIMS</td>
<td>Invalid dimension argument</td>
</tr>
<tr>
<td>MPI_ERR_ARG</td>
<td>Invalid argument of some other kind</td>
</tr>
<tr>
<td>MPI_ERR_UNKNOWN</td>
<td>Unknown error</td>
</tr>
<tr>
<td>MPI_ERR_TRUNCATE</td>
<td>Message truncated on receive</td>
</tr>
<tr>
<td>MPI_ERR_OTHER</td>
<td>Known error not in this list</td>
</tr>
<tr>
<td>MPI_ERR_INTERNAL</td>
<td>Internal MPI (implementation) error</td>
</tr>
<tr>
<td>MPI_ERR_IN_STATUS</td>
<td>Error code is in status</td>
</tr>
<tr>
<td>MPI_ERR_PENDING</td>
<td>Pending request</td>
</tr>
<tr>
<td>MPI_ERR_FILE</td>
<td>Invalid file handle</td>
</tr>
<tr>
<td>MPI_ERR_BAD_FILE</td>
<td>Collective argument not identical on all processes, or collective routines called in a different order by different processes</td>
</tr>
<tr>
<td>MPI_ERR_ACCESS</td>
<td>Error related to the amode passed to MPI_FILE_OPEN</td>
</tr>
<tr>
<td>MPI_ERR_NO_SPACE</td>
<td>Unsupported datarep passed to MPI_FILE_SET_VIEW</td>
</tr>
<tr>
<td>MPI_ERR_QUOTA</td>
<td>Unsupported operation, such as seeking on a file which supports sequential access only</td>
</tr>
<tr>
<td>MPI_ERR_READ_ONLY</td>
<td>File does not exist</td>
</tr>
<tr>
<td>MPI_ERR_FILE_EXISTS</td>
<td>File exists</td>
</tr>
<tr>
<td>MPI_ERR_FILE_IN_USE</td>
<td>Invalid file name (e.g., path name too long)</td>
</tr>
<tr>
<td>MPI_ERR_CONV_CONFLICT</td>
<td>Permission denied</td>
</tr>
<tr>
<td>MPI_ERR_CONV_NO_SPACE</td>
<td>Not enough space</td>
</tr>
<tr>
<td>MPI_ERR_CONV_QUOTA_EXCEEDED</td>
<td>Quota exceeded</td>
</tr>
<tr>
<td>MPI_ERR_CONV_READONLY</td>
<td>Read-only file or file system</td>
</tr>
<tr>
<td>MPI_ERR_CONV_OPERATION_DENIED</td>
<td>File operation could not be completed, as the file is currently open by some process</td>
</tr>
<tr>
<td>MPI_ERR_CONV_CONVERSION</td>
<td>Conversion functions could not be registered because a data representation identifier that was already defined was passed to MPI_REGISTER_DATAREP</td>
</tr>
<tr>
<td>MPI_ERR_CONV_ERROR</td>
<td>An error occurred in a user supplied data conversion function</td>
</tr>
<tr>
<td>MPI_ERR_CONV_IO</td>
<td>Other I/O error</td>
</tr>
<tr>
<td>MPI_ERR_CONV_LASTCODE</td>
<td>Last error code</td>
</tr>
</tbody>
</table>

*Invalid keyval has been passed*

- MPI_ALLOC_MEM failed because memory is exhausted
- Key longer than MPI_MAX_INFO_KEY
- Value longer than MPI_MAX_INFO_VALUE
- Invalid key passed to MPI_INFO_DELETE

*Error in spawning processes*

- Invalid port name passed to MPI_COMM_CONNECT
- Invalid service name passed to MPI_UNPUBLISH_NAME
- Invalid service name passed to MPI_LOOKUP_NAME
- Invalid win argument
- Invalid size argument
- Invalid disp argument
- Invalid info argument
- Invalid locktype argument
- Invalid assert argument

*Conflicting accesses to window*

- Wrong synchronization of RMA calls
Example: Autocorrelation

The Original Signal
Autocorrelation – More than Just a Scatter

NUMELEMENTS

Divide NUMELEMENTS into pieces for the NumCpus (this is what MPI_Scatter does)

NUMELEMENTS
NumCpus

NUMELEMENTS
NumCpus

NUMELEMENTS
NumCpus

NUMELEMENTS
NumCpus

NUMELEMENTS
NumCpus

NUMELEMENTS
NumCpus

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

NUMELEMENTS
NumCpus
Autocorrelation – How the Shifting Works

Shift = 0

NUMELEMENTS
NumCpus

MAXSHIFTS

Shift = 1

NUMELEMENTS
NumCpus

MAXSHIFTS

Shift = 2

NUMELEMENTS
NumCpus

MAXSHIFTS

Shift = 3

NUMELEMENTS
NumCpus

MAXSHIFTS

Shift = MAXSHIFTS-1

NUMELEMENTS
NumCpus

MAXSHIFTS