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
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 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

% 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 );
```
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 );
```

This requires MPI server code getting installed on all those CPUs.

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

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

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

Both the sender and receivers need to execute \text{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.

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

```c
int numCPUs;
int me;
float k_over_rho_c; // the 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 )
{
    << 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.
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.

---

**MPI_Recv**( array, maxCanReceive, type, src, tag, MPI_COMM_WORLD, &status );

- **array**
  - Address of data to receive into
  - MPI_CHAR
  - MPI_INT
  - MPI_LONG
  - MPI_FLOAT
  - MPI_DOUBLE

- **maxCanReceive**
  - # elements we can receive, at most

- **type**
  - Type = MPI_Status

- **src**
  - 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!)

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

- **status**
  - Type = MPI_Status
  - # elements we can receive, at most
  - Address of data to receive into

---
# MPI status Values

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_SUCCESS</td>
<td>0</td>
<td>Successful return code.</td>
</tr>
<tr>
<td>MPI_ERR_BUFFER</td>
<td>1</td>
<td>Invalid buffer pointer.</td>
</tr>
<tr>
<td>MPI_ERR_COUNT</td>
<td>2</td>
<td>Invalid count argument.</td>
</tr>
<tr>
<td>MPI_ERR_TYPE</td>
<td>3</td>
<td>Invalid datatype argument.</td>
</tr>
<tr>
<td>MPI_ERR_TAG</td>
<td>4</td>
<td>Invalid tag argument.</td>
</tr>
<tr>
<td>MPI_ERR_COMM</td>
<td>5</td>
<td>Invalid communicator.</td>
</tr>
<tr>
<td>MPI_ERR_RANK</td>
<td>6</td>
<td>Invalid rank.</td>
</tr>
<tr>
<td>MPI_ERR_REQUEST</td>
<td>7</td>
<td>Invalid MPI Request handle.</td>
</tr>
<tr>
<td>MPI_ERR_ROOT</td>
<td>8</td>
<td>Invalid root.</td>
</tr>
<tr>
<td>MPI_ERR_GROUP</td>
<td>9</td>
<td>Null group passed to function.</td>
</tr>
<tr>
<td>MPI_ERR_OP</td>
<td>10</td>
<td>Invalid operation.</td>
</tr>
<tr>
<td>MPI_ERR_TOPOLOGY</td>
<td>11</td>
<td>Invalid topology.</td>
</tr>
<tr>
<td>MPI_ERR_DIMS</td>
<td>12</td>
<td>Illegal dimension argument.</td>
</tr>
<tr>
<td>MPI_ERR_ARG</td>
<td>13</td>
<td>Invalid argument.</td>
</tr>
<tr>
<td>MPI_ERR_UNKNOWN</td>
<td>14</td>
<td>Unknown error.</td>
</tr>
<tr>
<td>MPI_ERR_TRUNCATE</td>
<td>15</td>
<td>Message truncated on receive.</td>
</tr>
<tr>
<td>MPI_ERR_Other</td>
<td>16</td>
<td>Other error; use Error_string.</td>
</tr>
<tr>
<td>MPI_ERR_INTERN</td>
<td>17</td>
<td>Internal error code.</td>
</tr>
<tr>
<td>MPI_ERR_IN_STATUS</td>
<td>18</td>
<td>Lock in status for error value.</td>
</tr>
<tr>
<td>MPI_ERR_PENDING</td>
<td>19</td>
<td>Pending request.</td>
</tr>
<tr>
<td>MPI_ERR_ACCESS</td>
<td>20</td>
<td>Permission denied.</td>
</tr>
<tr>
<td>MPI_ERR_AMODE</td>
<td>21</td>
<td>Unsupported amode passed to open.</td>
</tr>
<tr>
<td>MPI_ERR_ASSERT</td>
<td>22</td>
<td>Invalid assertion.</td>
</tr>
<tr>
<td>MPI_ERR_BN_FILE</td>
<td>23</td>
<td>Invalid file name (for example, path name too long).</td>
</tr>
<tr>
<td>MPI_ERR_BASE</td>
<td>24</td>
<td>Invalid base.</td>
</tr>
<tr>
<td>MPI_ERR_CONV</td>
<td>25</td>
<td>An error occurred in a user-supplied data-conversion function.</td>
</tr>
<tr>
<td>MPI_ERR_DISP</td>
<td>26</td>
<td>Invalid displacement.</td>
</tr>
<tr>
<td>MPI_ERR_DUP_DATAREP</td>
<td>27</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_FILE_EXISTS</td>
<td>28</td>
<td>File exists.</td>
</tr>
<tr>
<td>MPI_ERR_FILE_IN_USE</td>
<td>29</td>
<td>File operation could not be completed, as the file is currently open by some process.</td>
</tr>
<tr>
<td>MPI_ERR_FILE</td>
<td>30</td>
<td>Invalid open file handle.</td>
</tr>
<tr>
<td>MPI_ERR_INFO_KEY</td>
<td>31</td>
<td>Illegal info key.</td>
</tr>
<tr>
<td>MPI_ERR_INFO_NOKKEY</td>
<td>32</td>
<td>No such key.</td>
</tr>
<tr>
<td>MPI_ERR_INFO_VALUE</td>
<td>33</td>
<td>Illegal info value.</td>
</tr>
<tr>
<td>MPI_ERR_INFO</td>
<td>34</td>
<td>Invalid info object.</td>
</tr>
<tr>
<td>MPI_ERR_IO</td>
<td>35</td>
<td>I/O error.</td>
</tr>
<tr>
<td>MPI_ERR_RECV</td>
<td>36</td>
<td>Illegal key value.</td>
</tr>
<tr>
<td>MPI_ERR_RECVVAL</td>
<td>37</td>
<td>Invalid recv type.</td>
</tr>
<tr>
<td>MPI_ERR_NAME</td>
<td>38</td>
<td>Name not found.</td>
</tr>
<tr>
<td>MPI_ERR_NO_MEM</td>
<td>39</td>
<td>Memory exhausted.</td>
</tr>
<tr>
<td>MPI_ERR_NOT_SAME</td>
<td>40</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_INVALID</td>
<td>41</td>
<td>Not enough space.</td>
</tr>
<tr>
<td>MPI_ERR_NO_SUCH_FILE</td>
<td>42</td>
<td>File (or directory) does not exist.</td>
</tr>
<tr>
<td>MPI_ERR_PORT</td>
<td>43</td>
<td>Invalid port.</td>
</tr>
<tr>
<td>MPI_ERR_QUOTA</td>
<td>44</td>
<td>Quota exceeded.</td>
</tr>
<tr>
<td>MPI_ERR_READ_ONLY</td>
<td>45</td>
<td>Read-only file system.</td>
</tr>
<tr>
<td>MPI_ERR_RMA_CONFLICT</td>
<td>46</td>
<td>Conflicting accesses to window.</td>
</tr>
<tr>
<td>MPI_ERR_RMA_SYNC</td>
<td>47</td>
<td>Error of RMA synchronization.</td>
</tr>
<tr>
<td>MPI_ERR_SERVICE</td>
<td>48</td>
<td>Invalid publish/unpublish.</td>
</tr>
<tr>
<td>MPI_ERR_SIZE</td>
<td>49</td>
<td>Invalid size.</td>
</tr>
<tr>
<td>MPI_ERR.Trim</td>
<td>50</td>
<td>Error spawning.</td>
</tr>
<tr>
<td>MPI_ERR_UNSUPPORTED_DATAREP</td>
<td>51</td>
<td>Unsupported datatype passed to MPI_FILE_set_view.</td>
</tr>
<tr>
<td>MPI_ERR_UNSUPPORTED_OPERATION</td>
<td>52</td>
<td>Unsupported operation, such as seeking on a file that supports only sequential access.</td>
</tr>
<tr>
<td>MPI_ERR_WIN</td>
<td>53</td>
<td>Invalid window.</td>
</tr>
<tr>
<td>MPI_T_ERR_MEMORY</td>
<td>54</td>
<td>Out of memory.</td>
</tr>
<tr>
<td>MPI_T_ERR_NOT_INITIALIZED</td>
<td>55</td>
<td>Interface not initialized.</td>
</tr>
<tr>
<td>MPI_T_ERR_CANNOT_INIT</td>
<td>56</td>
<td>Interface not in the state to be initialized.</td>
</tr>
<tr>
<td>MPI_T_ERR_INVALID_INDEX</td>
<td>57</td>
<td>The enumeration index is invalid.</td>
</tr>
<tr>
<td>MPI_T_ERR_INVALID_ITEM</td>
<td>58</td>
<td>The item index queried is out of range.</td>
</tr>
<tr>
<td>MPI_T_ERR_INVALIDHANDLE</td>
<td>59</td>
<td>The handle is invalid.</td>
</tr>
<tr>
<td>MPI_T_ERR_OUT_OF_HANDLES</td>
<td>60</td>
<td>No more handles available.</td>
</tr>
<tr>
<td>MPI_T_ERR_OUT_OF_SESSIONS</td>
<td>61</td>
<td>No more sessions available.</td>
</tr>
<tr>
<td>MPI_T_ERR_INVALID_SESSION</td>
<td>62</td>
<td>Session argument is not a valid session.</td>
</tr>
<tr>
<td>MPI_T_ERR_CVAR_SET_NOT</td>
<td>63</td>
<td>Variable cannot be set at this moment.</td>
</tr>
<tr>
<td>MPI_T_ERR_CVAR_SET_NEVER</td>
<td>64</td>
<td>Variable cannot be set until end of execution.</td>
</tr>
<tr>
<td>MPI_T_ERR_PVAR_NO_STARTSTOP</td>
<td>65</td>
<td>Variable cannot be started or stopped.</td>
</tr>
<tr>
<td>MPI_T_ERR_PVAR_NO_WRITE</td>
<td>66</td>
<td>Variable cannot be written or reset.</td>
</tr>
<tr>
<td>MPI_T_ERR_PVAR_NO_ATOMIC</td>
<td>67</td>
<td>Variable cannot be read and written atomically.</td>
</tr>
<tr>
<td>MPI_ERR_RMA</td>
<td>68</td>
<td>Target memory is not part of the window (in the case of a window created with MPI_WIN_CREATE_DYNAMIC, target memory is not attached).</td>
</tr>
<tr>
<td>MPI_ERR_RMA_ATTACH</td>
<td>69</td>
<td>Memory cannot be attached (e.g., because of resource exhaustion).</td>
</tr>
<tr>
<td>MPI_ERR_RMA_FLAVOR</td>
<td>70</td>
<td>Passed window has the wrong flavor for the called function.</td>
</tr>
<tr>
<td>MPI_ERR_RMA_SHARED</td>
<td>71</td>
<td>Memory cannot be shared (e.g., some process in the group of the specified communicator cannot expose shared memory).</td>
</tr>
<tr>
<td>MPI_T_ERR_INVALID</td>
<td>72</td>
<td>Invalid use of the interface or bad parameter values(s).</td>
</tr>
<tr>
<td>MPI_T_ERR_INVALID_NAME</td>
<td>73</td>
<td>The variable or category name is invalid.</td>
</tr>
<tr>
<td>MPI_ERR_LASTCODE</td>
<td>93</td>
<td>Last error code.</td>
</tr>
</tbody>
</table>
**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\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.
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`
- `...`

```c
MPI_Status status;
int actualCount; // in # of elements
```

```c
MPI_Get_count( &status, type, &actualCount );
```

Type = MPI_Status
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

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

```c
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 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 )  // this is the sender
{
    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
{  // this is the receiver
    MPI_Scatter( NULL, 0, MPI_FLOAT, myData, localSize, MPI_FLOAT, ROOT, MPI_WORLD_COMM );
}

Signifies “no data to send”

It is typical that you scatter to yourself
```
MPI_Gather( snd_array, snd_count, snd_type, rcv_array, rcv_count, rcv_type, dst, MPI_COMM_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
• • •
```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 ) // this is the receiver
{
    float *InputData = new float[NUMELEMENTS];
    ...
    MPI_Gather( myData, localSize, MPI_FLOAT, InputData, NUMELEMENTS, MPI_FLOAT, ROOT, MPI_WORLD_COMM );
    << write data from Array to disk >>
}
else // this is the sender
{
    MPI_Gather( myData, localSize, MPI_FLOAT, NULL, 0, MPI_FLOAT, ROOT, MPI_WORLD_COMM );
}
```

**Gather Example**
#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 ) // send, then receive
{
    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 );
    << 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
{
    // receive then send
    MPI_Scatter( NULL, 0, MPI_FLOAT, myData, localSize, MPI_FLOAT, ROOT, MPI_WORLD_COMM );
    << do some computing on mydata[] >>
    MPI_Gather( myData, localSize, MPI_FLOAT, NULL, 0, MPI_FLOAT, ROOT, MPI_WORLD_COMM );
}
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
- MPILAND
- MPI_BAND
- MPI_LOR
- MPI_BOR
- MPI_LXOR
- MPI_BXOR
- MPI_CHAR
- MPI_INT
- MPI_LONG
- MPI_FLOAT
- MPI_DOUBLE

Both the sender and receivers need to execute MPI_Reduce — there is no separate receive function
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.;
<< compute this CPUs partialSum by, perhaps, adding up a local array >>

MPI_Reduce( &partialSum, &globalSum, 1, MPI_FLOAT, MPI_SUM, ROOT, MPI_COMM_WORLD );
MPI Barriers

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

All CPUs must execute a 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().
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
dtype 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**

```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”.

The clocks on the different CPUs are not guaranteed to be synchronized.
Suppose We Have This Setup
Welcome to *Parallelism Jeopardy!*

I’ll take CS 475/575 for $800, Alex.

**IN A MULTI-CPU SYSTEM, THIS IS THE TOTAL NUMBER OF DIFFERENT PARALLELISMS THAT WE COVERED THIS QUARTER**
1. Multicore OpenMP
1. Multicore OpenMP
2. CPU SIMD
1. Multicore OpenMP
2. CPU SIMD
3. GPU
1. Multicore OpenMP
2. CPU SIMD
3. GPU
4. MPI

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