The Message Passing Interface (MPI): Parallelism on Distributed 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

Computer Graphics
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 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

% mpicc -o program program.c ...

or

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

All distributed processors execute the *same* program at the same time

% mpiexec -np 64 program

# of processors to use
Setting Up and Finishing

You don’t need to process command line arguments if you don’t need to. You can also call it as:

```c
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. Only an administrator can do this.

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.
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
Both the sender and receivers need to execute **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
}
else
{
    MPI_Bcast( &k_over_rho_c, 1, MPI_FLOAT, BOSS, MPI_COMM_WORLD ); // receive
}
```

**MPI Broadcast Example**

This diagram illustrates the MPI broadcast process. The source node (src) sends a message to all other nodes (src nodes), which then receive the message.
Confused? Look at this Diagram

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

```c
if (me == BOSS) {
    // send
    MPI_Bcast(&k_over_rho_c, 1, MPI_FLOAT, BOSS, MPI_COMM_WORLD);
}
else {
    // receive
    MPI_Bcast(&k_over_rho_c, 1, MPI_FLOAT, BOSS, MPI_COMM_WORLD);
}
```
How Does this Work?
Think Star Trek Wormholes!
Sending Data from a Source CPU to a Destination CPU

```
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.

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
```

address of data to send from

# elements

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!)
Receiving Data in a Destination CPU from a Source 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

- **# elements we can receive, at most**
  - MPI_CHAR
  - MPI_INT
  - MPI_LONG
  - MPI_FLOAT
  - MPI_DOUBLE

- **address of data to receive into**

- **Type = MPI_Status**

- **src node** ➔ **dst node**
## MPI status Values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_SUCCESS</td>
<td>Successful return code.</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_COMM_RANK</td>
<td>Invalid rank.</td>
</tr>
<tr>
<td>MPI_ERR_REQUEST</td>
<td>Invalid MPI Request handle.</td>
</tr>
<tr>
<td>MPI_ERR_ROOT</td>
<td>Invalid root.</td>
</tr>
<tr>
<td>MPI_ERR_GROUP</td>
<td>Null group passed to function.</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>Illegal dimension argument.</td>
</tr>
<tr>
<td>MPI_ERR_ARG</td>
<td>Invalid argument.</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>Other error; use Error_string.</td>
</tr>
<tr>
<td>MPI_ERR_INTERN</td>
<td>Internal error code.</td>
</tr>
<tr>
<td>MPI_ERR_STATUS</td>
<td>lock in status for error value.</td>
</tr>
<tr>
<td>MPI_ERR_PENDING</td>
<td>Pending request.</td>
</tr>
<tr>
<td>MPI_ERR_ACCESS</td>
<td>Permission denied.</td>
</tr>
<tr>
<td>MPI_ERR_AMODE</td>
<td>Unsupported amode passed to open.</td>
</tr>
<tr>
<td>MPI_ERR_ASSERT</td>
<td>Invalid assert.</td>
</tr>
<tr>
<td>MPI_ERR_BASE</td>
<td>Invalid base.</td>
</tr>
<tr>
<td>MPI_ERR_CONV</td>
<td>An error occurred in a user-supplied data-conversion function.</td>
</tr>
<tr>
<td>MPI_ERR_DISP</td>
<td>Invalid displacement.</td>
</tr>
<tr>
<td>MPI_ERR_DUP_DATAREP</td>
<td>Conversion functions could not be registered because a data representation identifier was already defined was passed to MPI_REGISTER_DATAREP.</td>
</tr>
<tr>
<td>MPI_ERR_FILE_EXISTS</td>
<td>File exists.</td>
</tr>
<tr>
<td>MPI_ERR_FILE_IN_USE</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>Invalid file handle.</td>
</tr>
<tr>
<td>MPI_ERR_INFO_KEY</td>
<td>Illegal info key.</td>
</tr>
<tr>
<td>MPI_ERR_INFO_NONE</td>
<td>No such key.</td>
</tr>
<tr>
<td>MPI_ERR_INFO_VALUE</td>
<td>Invalid info value.</td>
</tr>
<tr>
<td>MPI_ERR_INFO_OBJECT</td>
<td>Invalid info object.</td>
</tr>
<tr>
<td>MPI_ERR_IO</td>
<td>I/O error.</td>
</tr>
<tr>
<td>MPI_ERR_IEEXEC</td>
<td>Illegal key value.</td>
</tr>
<tr>
<td>MPI_ERR_LOCMPART</td>
<td>Not found.</td>
</tr>
<tr>
<td>MPI_ERR_NAME</td>
<td>Memory exhausted.</td>
</tr>
<tr>
<td>MPI_ERR_NO_MEM</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_NOTSAME</td>
<td>For use only by implementation.</td>
</tr>
<tr>
<td>MPI_ERR_NO_SPACE</td>
<td>Not enough space.</td>
</tr>
<tr>
<td>MPI_ERR_NOT_SUCH_FILE</td>
<td>File (or directory) does not exist.</td>
</tr>
<tr>
<td>MPI_ERR_PORT</td>
<td>Invalid port.</td>
</tr>
<tr>
<td>MPI_ERR_QUOTA</td>
<td>Quota exceeded.</td>
</tr>
<tr>
<td>MPI_ERR_READ_ONLY</td>
<td>Read-only file system.</td>
</tr>
<tr>
<td>MPI_ERR_RMA_CONFLICT</td>
<td>Conflicting access to window.</td>
</tr>
<tr>
<td>MPI_ERR_RMA_SYNC</td>
<td>Error in RMA synchronization.</td>
</tr>
<tr>
<td>MPI_ERR_RMA_SERVICE</td>
<td>Invalid publish/unpublish.</td>
</tr>
<tr>
<td>MPI_ERR_RMA_SIZE</td>
<td>Invalid size.</td>
</tr>
<tr>
<td>MPI_ERR_RMA_SHARED</td>
<td>Error in RMA shared.</td>
</tr>
<tr>
<td>MPI_ERR_UNSUPPLIED_DATAREP</td>
<td>Unsupported datarep passed to MPI_File_set_view.</td>
</tr>
<tr>
<td>MPI_ERR_UNSUPPLIED_OPERATION</td>
<td>Unsupported operation, such as seeking on a file that supports only sequential access.</td>
</tr>
<tr>
<td>MPI_ERR_WIN</td>
<td>Invalid window.</td>
</tr>
<tr>
<td>MPI_ERR_MEMORY</td>
<td>Out of memory.</td>
</tr>
<tr>
<td>MPI_ERR_NOT_INITIALIZED</td>
<td>Interface not initialized.</td>
</tr>
<tr>
<td>MPI_ERR_CANNOT_INIT</td>
<td>Interface not in the state to be initialized.</td>
</tr>
<tr>
<td>MPI_ERR_INVALID_INDEX</td>
<td>The enumeration index is invalid.</td>
</tr>
<tr>
<td>MPI_ERR_INVALID_INDEX</td>
<td>The item index queried is out of range.</td>
</tr>
<tr>
<td>MPI_ERR_INVALID_HANDLE</td>
<td>The handle is invalid.</td>
</tr>
<tr>
<td>MPI_ERR_OUT_OF_HANDLES</td>
<td>No more handles available.</td>
</tr>
<tr>
<td>MPI_ERR_OUT_OF_SESSIONS</td>
<td>No more sessions available.</td>
</tr>
<tr>
<td>MPI_ERR_INVALID_SESSION</td>
<td>Session argument is not a valid session.</td>
</tr>
<tr>
<td>MPI_ERR_CVAR_SET_NOT</td>
<td>Variable cannot be set at this moment.</td>
</tr>
<tr>
<td>(MPI_ERR_CVAR_SET_NOT EVER</td>
<td>Variable cannot be set until end of execution.</td>
</tr>
<tr>
<td>MPI_ERR_PVAR_NO_STARTSTOP</td>
<td>Variable cannot be started or stopped.</td>
</tr>
<tr>
<td>MPI_ERR_PVAR_NO_WRITE</td>
<td>Variable cannot be written or read.</td>
</tr>
<tr>
<td>MPI_ERR_PVAR_NO_ATOMIC</td>
<td>Variable cannot be read and written atomically.</td>
</tr>
<tr>
<td>MPI_ERR_PVAR_ATOMIC</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_PVAR_ATOMIC</td>
<td>Memory cannot be attached (e.g., because of resource exhaustion).</td>
</tr>
<tr>
<td>MPI_ERR_PVAR_ATOMIC</td>
<td>Passed window has the wrong flavor for the called function.</td>
</tr>
<tr>
<td>MPI_ERR_PVAR_ATOMIC</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_ERR_PVAR_ATOMIC</td>
<td>Invalid use of the interface or bad parameters: values,</td>
</tr>
<tr>
<td>(MPI_ERR_PVAR_ATOMIC</td>
<td>The variable or category name is invalid.</td>
</tr>
<tr>
<td>MPI_ERR_UNKNOWN</td>
<td>Last error code.</td>
</tr>
</tbody>
</table>
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, 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", 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\n", in, me );
}
```
How Much Data Did I Actually Receive?

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

- `maxCanReceive`: Number of elements we can receive, at most
- `type`: Type of elements
  - `MPI_CHAR`
  - `MPI_INT`
  - `MPI_LONG`
  - `MPI_FLOAT`
  - `MPI_DOUBLE`
- `actualCount`: Number of elements received

```c
MPI_Status status;
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_Recv( )?

MPI_Send( ) blocks until the transfer is far enough along that the array can be destroyed or re-used.
```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:
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
}
```

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.
In Distributed Computing, You Often Hear About These Design Patterns

- **Broadcast**
  - [Diagram of Broadcast]

- **Scatter**
  - [Diagram of Scatter]

- **Gather**
  - [Diagram of Gather]
Scatter and Gather Usually Go Together

Note surprisingly, this is referred to as Scatter/Gather :-)

Oregon State University
Computer Graphics

mjb – April 23, 2020
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.
#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 )
// 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, BOSS, MPI_WORLD_COMM );
}
else
// this is the receiver
{
  MPI_Scatter( NULL, 0, MPI_FLOAT, myData, localSize, MPI_FLOAT, BOSS, MPI_WORLD_COMM );
}

This is all of the global data to be sent
This is each dst's local data to be received
Signifies "no data to send"
It is typical that you scatter to yourself
MPI Gather

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

# elements to send
- MPI_CHAR
- MPI_INT
- MPI_LONG
- MPI_FLOAT
- MPI_DOUBLE
- \...\n
# elements to receive
- MPI_CHAR
- MPI_INT
- MPI_LONG
- MPI_FLOAT
- MPI_DOUBLE
- \...\n
Gather
```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 )
    // this is the receiver
    {
        float *InputData = new float [ NUMELEMENTS ];
        ...
        MPI_Gather( myData, localSize, MPI_FLOAT, InputData, NUMELEMENTS, MPI_FLOAT, BOSS, MPI_WORLD_COMM );
        // << write data from Array to disk >>
    }
else
    // this is the sender
    {
        MPI_Gather( myData, localSize, MPI_FLOAT, NULL, 0, MPI_FLOAT, BOSS, MPI_WORLD_COMM );
    }
```

This is all of the global data to be received

Signifies “no data to receive”

This is each dst’s local data to be sent
A Full MPI Scatter / Gather Example

```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 )
    // send over, 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, BOSS, MPI_WORLD_COMM );

        << do some computing on mydata[ ] >>

        MPI_Gather( myData, localSize, MPI_FLOAT, InputData, NUMELEMENTS, MPI_FLOAT, BOSS, MPI_WORLD_COMM );
        << write data from InputData[ ] to disk >>
    }
else
    // receive, then send back
    {
        MPI_Scatter( NULL, 0, MPI_FLOAT, myData, localSize, MPI_FLOAT, BOSS, MPI_WORLD_COMM );
        << do some computing on mydata[ ] >>

        MPI_Gather( myData, localSize, MPI_FLOAT, NULL, 0, MPI_FLOAT, BOSS, MPI_WORLD_COMM );
    }
```

Oregon State University
Computer Graphics

mjb – April 23, 2020
MPI Reduction

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

- Where the partial result is stored on each CPU
- 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_CHAR
  - MPI_INT
  - MPI_LONG
  - MPI_FLOAT
  - MPI_DOUBLE
  - ...
- MPI_MIN
  - MPI_MAX
  - MPI_SUM
  - MPI_PROD
  - MPI_MINLOC
  - MPI_MAXLOC
  - MPI_BAND
  - MPI_BOR
  - MPI_LAND
  - MPI_LOR
  - MPI_LXOR
  - MPI_BXOR
- Who is given the final answer

This really should be called Scatter/Gather/Reduction

Both the sender and receivers need to execute MPI_Reduce. 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 Barriers

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.

\[
\text{MPI_Type_create_struct}( \text{count}, \text{blocklengths}, \text{displacements}, \text{types}, \text{datatype} );
\]

```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

```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.*
And now: Parallelism Conclusion

Reviewing what you know!
Suppose We Have This Setup
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
1. Multicore OpenMP
2. CPU SIMD
1. Multicore OpenMP
2. CPU SIMD
3. GPU
What is “4”?  
*This is how modern supercomputers work!*

And you now know something about using all 4 – congratulations!

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

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