Lecture 8: Intro to MPI

CMSE 822: Parallel Computing

Prof. Sean M. Couch

Puppy time

Oppenheimer

Brief MPI Tutorial

See https://computing.llnl.gov/tutorials/mpi/

also: http://www.mpi-forum.org/docs/

What is MPI?

An Interface Specification:

- **M P I** = **M**essage **P**assing **I**nterface

- MPI is a ***specification*** for the developers and users of message passing libraries. By itself, it is NOT a library - but rather the specification of what such a library should be.

- MPI primarily addresses the ***message-passing parallel programming model:*** data is moved from the address space of one process to that of another process through cooperative operations on each process.

- Simply stated, the goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs. The interface attempts to be:
  - Practical
  - Portable
  - Efficient
  - Flexible

- The MPI standard has gone through a number of revisions, with the most recent version being MPI-3.x

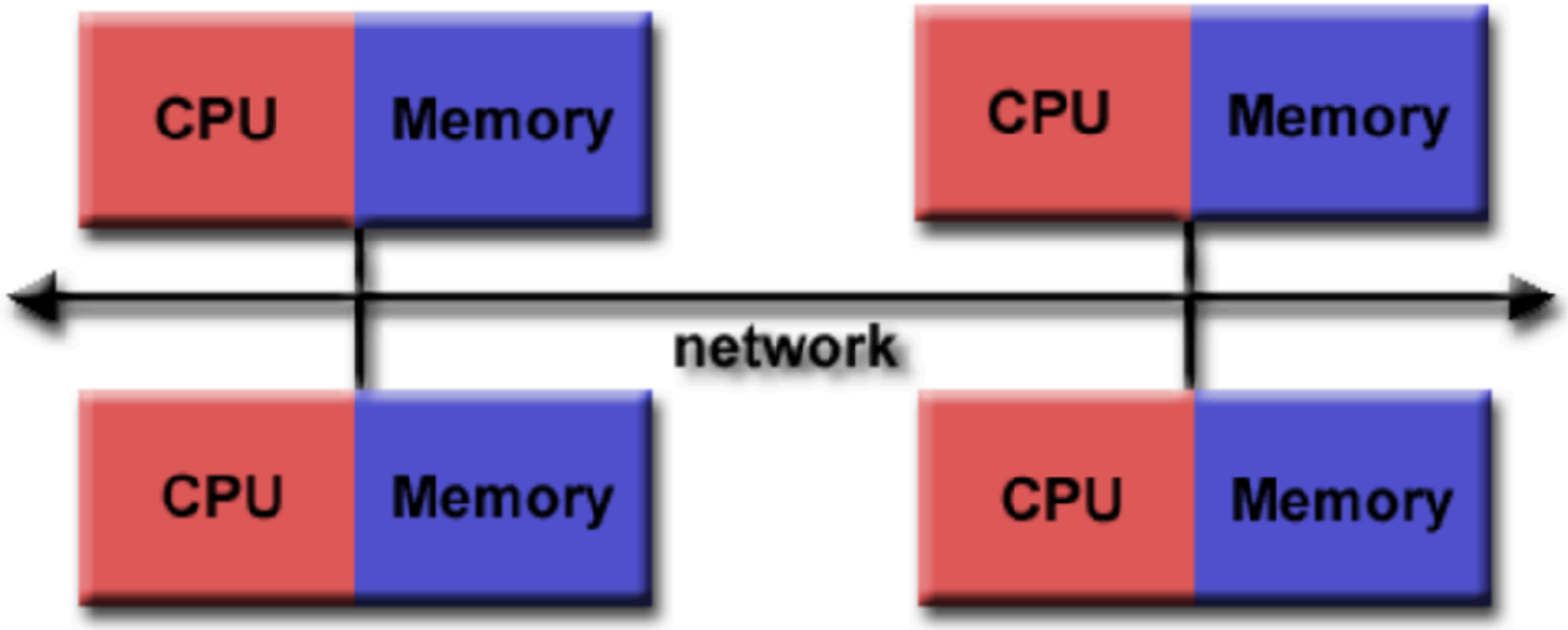
- Interface specifications have been defined for C and Fortran90 language bindings:
  - C++ bindings from MPI-1 are removed in MPI-3
  - MPI-3 also provides support for Fortran 2003 and 2008 features

- Actual MPI library implementations differ in which version and features of the MPI standard they support. Developers/users will need to be aware of this.

What is MPI?

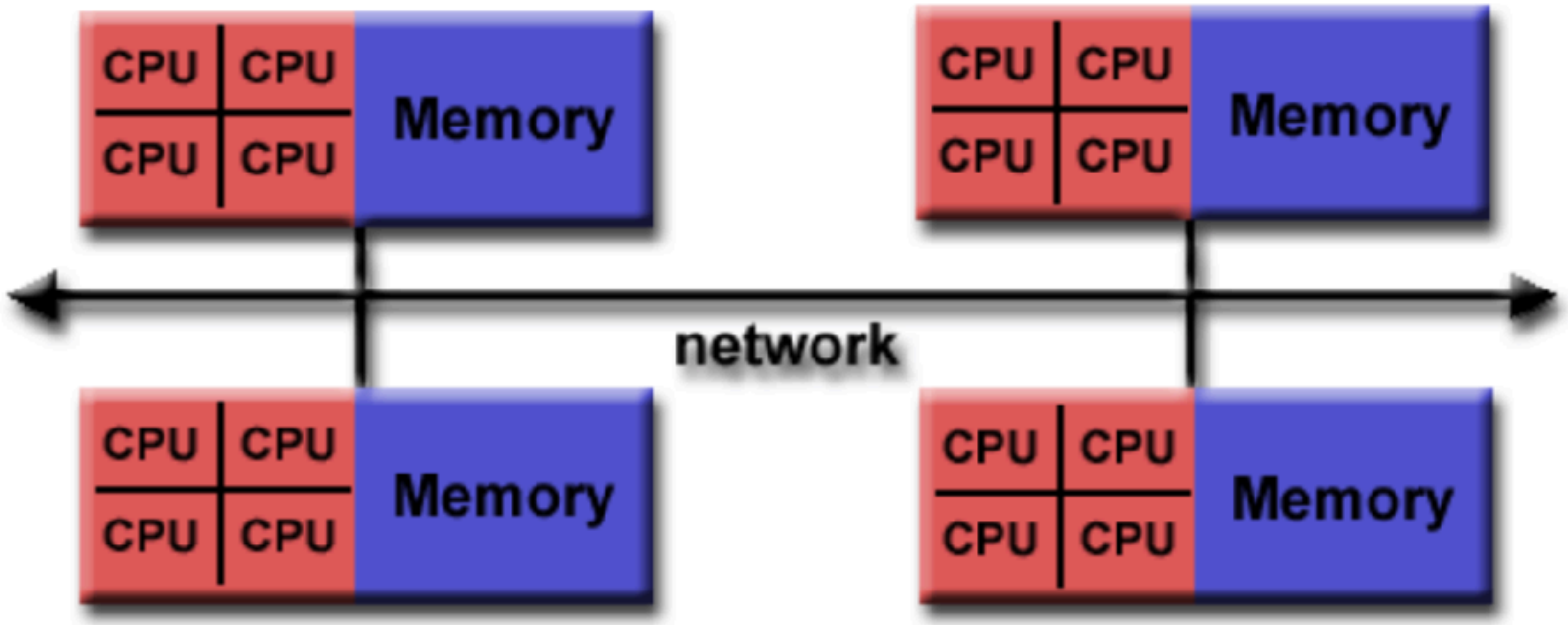
Programming Model:

- Originally, MPI was designed for distributed memory architectures, which were becoming increasingly popular at that time (1980s - early 1990s).



- As architecture trends changed, shared memory SMPs were combined over networks creating hybrid distributed memory / shared memory systems.
- MPI implementors adapted their libraries to handle both types of underlying memory architectures seamlessly. They also adapted/developed ways of handling different interconnects and protocols.

What is MPI?



- Today, MPI runs on virtually any hardware platform:
  - Distributed Memory
  - Shared Memory
  - Hybrid

- The programming model <u>clearly remains a distributed memory model</u> however, regardless of the underlying physical architecture of the machine.

- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.

What is MPI?

- **Reasons for Using MPI:**
  - **Standardization** - MPI is the only message passing library that can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.
  - **Portability** - There is little or no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.
  - **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance. Any implementation is free to develop optimized algorithms.
  - **Functionality** - There are ***over 430*** routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.

    Note: Most MPI programs can be written using a dozen or less routines
  - **Availability** - A variety of implementations are available, both vendor and public domain.

MPI Implementations

- Although the MPI programming interface has been standardized, actual library implementations will differ.

- For example, just a few considerations of many:
  - Which version of the MPI standard is supported?
  - Are all of the features in a particular MPI version supported?
  - Have any new features been added?
  - What network interfaces are supported?
  - How are MPI applications compiled?
  - How are MPI jobs launched?
  - Runtime environment variable controls?

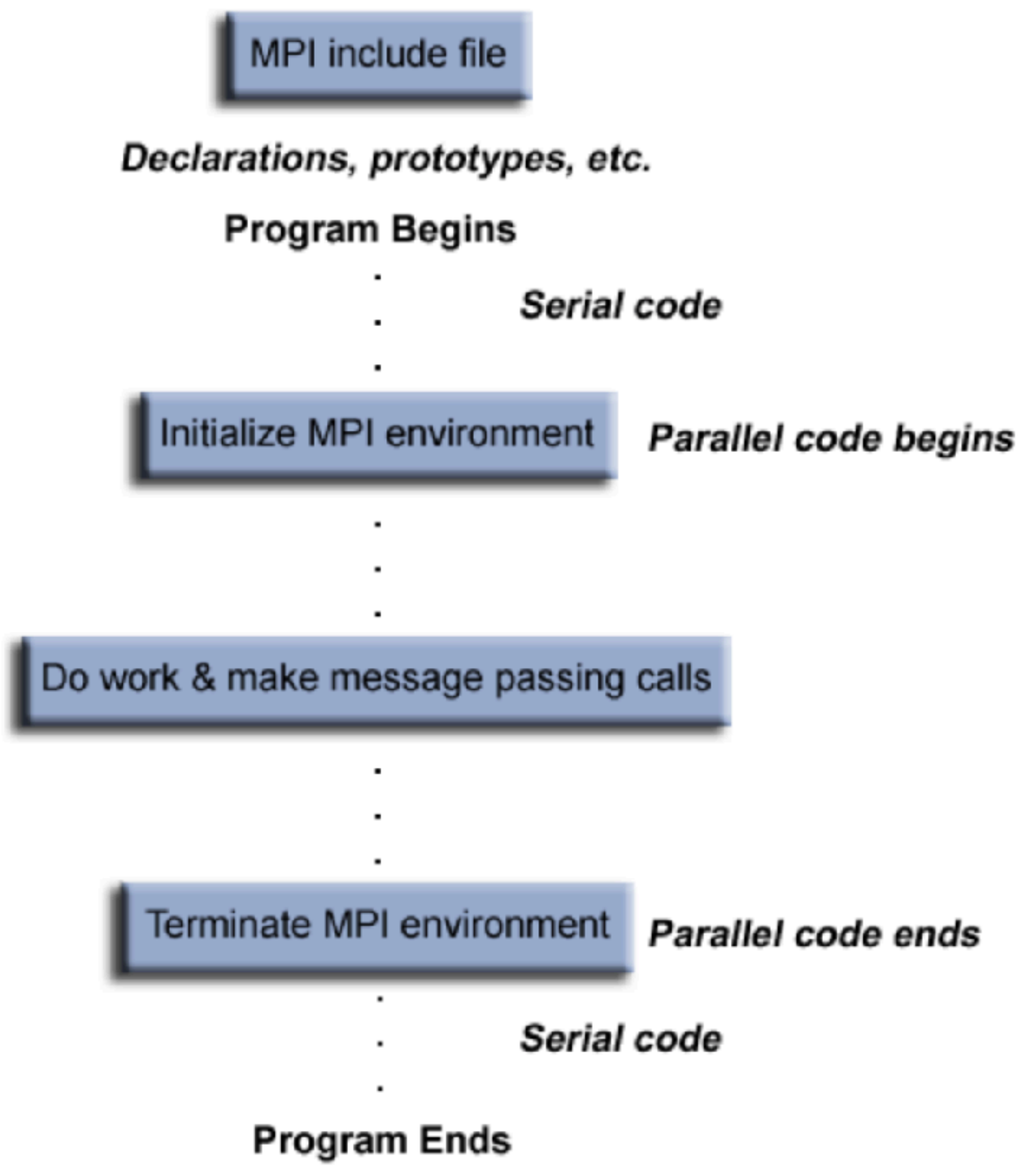
- MPI library implementations on LC systems vary, as do the compilers they are built for. These are summarized in the table below:

MPI Library	Where?	Compilers
MVAPICH	Linux clusters	GNU, Intel, PGI, Clang
Open MPI	Linux clusters	GNU, Intel, PGI, Clang
Intel MPI	Linux clusters	Intel, GNU
IBM Spectrum MPI	Coral Early Access and Sierra clusters	IBM, GNU, PGI, Clang

- Each MPI library is briefly discussed in the following sections, including links to additional detailed information.

Getting Started with MPI

General MPI Program Structure:




```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }

else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

MPI_Finalize();
}
```

Getting Started

- **Header File:**
  - Required for all programs that make MPI library calls.

C include file	Fortran include file
`#include "mpi.h"`	`include 'mpif.h'`

  - With MPI-3 Fortran, the **`USE mpi_f08`** module is preferred over using the include file shown above.

Getting Started

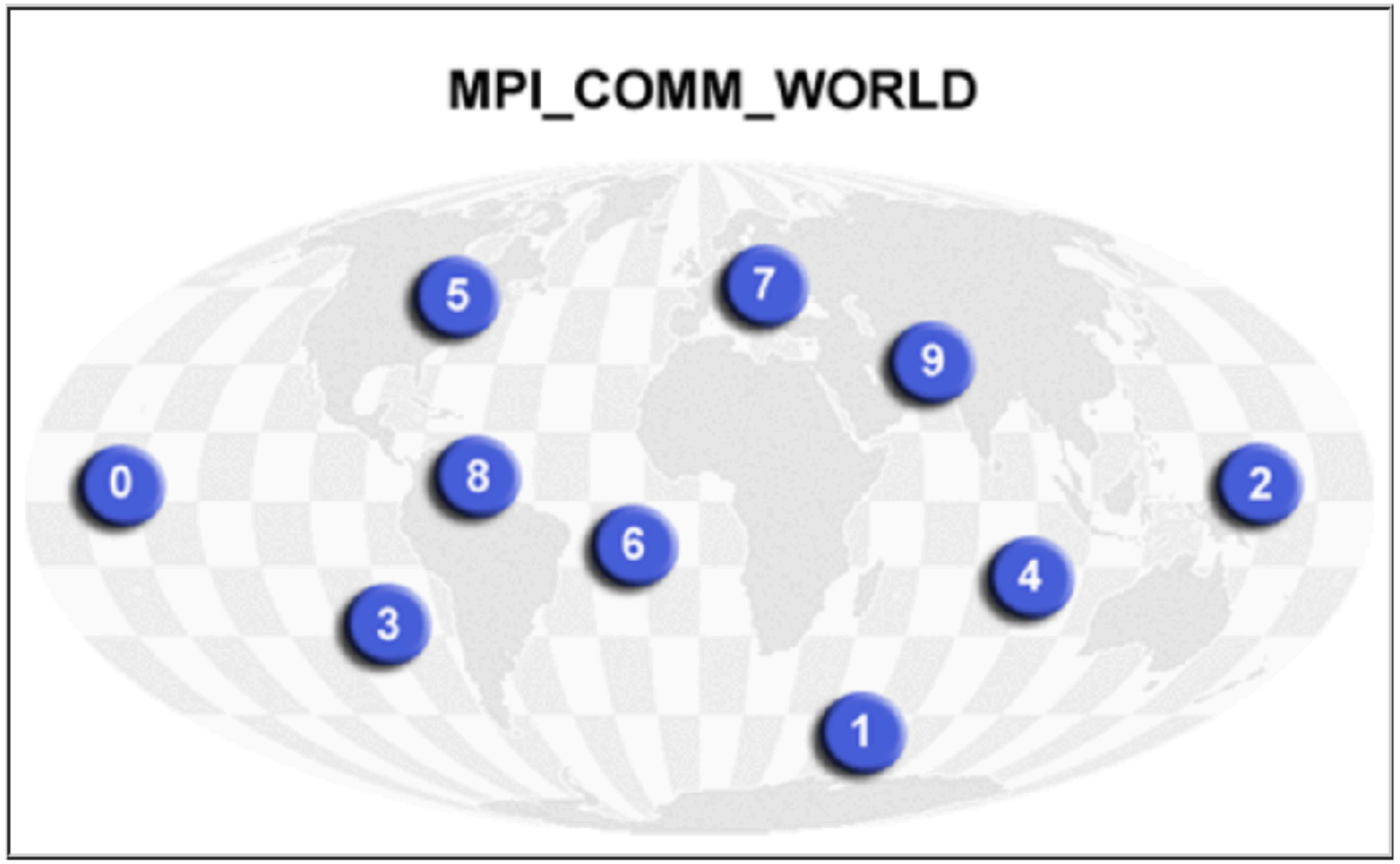
Format of MPI Calls:

- C names are case sensitive; Fortran names are not.
- Programs must not declare variables or functions with names beginning with the prefix MPI_ or PMPI_ (profiling interface).

C Binding	
Format:	`rc = MPI_Xxxxx(parameter, ... )`
Example:	`rc = MPI_Bsend(&buf,count,type,dest,tag,comm)`
Error code:	Returned as "rc". MPI_SUCCESS if successful
Fortran Binding	
Format:	`CALL MPI_XXXXX(parameter,..., ierr)` `call mpi_xxxxx(parameter,..., ierr)`
Example:	`CALL MPI_BSEND(buf,count,type,dest,tag,comm,ierr)`
Error code:	Returned as "ierr" parameter. MPI_SUCCESS if successful

Getting Started

- **Communicators and Groups:**
  - MPI uses objects called communicators and groups to define which collection of processes may communicate with each other.
  - Most MPI routines require you to specify a communicator as an argument.
  - Communicators and groups will be covered in more detail later. For now, simply use **MPI_COMM_WORLD** whenever a communicator is required - it is the predefined communicator that includes all of your MPI processes.



Getting Started

- **Rank:**
  - Within a communicator, every process has its own unique, integer identifier assigned by the system when the process initializes. A rank is sometimes also called a "task ID". Ranks are contiguous and begin at zero.
  - Used by the programmer to specify the source and destination of messages. Often used conditionally by the application to control program execution (if rank=0 do this / if rank=1 do that).
- **Error Handling:**
  - Most MPI routines include a return/error code parameter, as described in the "Format of MPI Calls" section above.
  - However, according to the MPI standard, the default behavior of an MPI call is to abort if there is an error. This means you will probably not be able to capture a return/error code other than MPI_SUCCESS (zero).
  - The standard does provide a means to override this default error handler. A discussion on how to do this is available HERE. You can also consult the error handling section of the relevant MPI Standard documentation located at http://www.mpi-forum.org/docs/.
  - The types of errors displayed to the user are implementation dependent.

Environment Management Routines

MPI_Init

Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program. For C programs, MPI_Init may be used to pass the command line arguments to all processes, although this is not required by the standard and is implementation dependent.

```
MPI_Init (&argc,&argv)
MPI_INIT (ierr)
```

Environment Management Routines

MPI_Comm_size

Returns the total number of MPI processes in the specified communicator, such as MPI_COMM_WORLD. If the communicator is MPI_COMM_WORLD, then it represents the number of MPI tasks available to your application.

```
MPI_Comm_size (comm,&size)
MPI_COMM_SIZE (comm,size,ierr)
```

Environment Management Routines

MPI_Comm_rank

Returns the rank of the calling MPI process within the specified communicator. Initially, each process will be assigned a unique integer rank between 0 and number of tasks - 1 within the communicator MPI_COMM_WORLD. This rank is often referred to as a task ID. If a process becomes associated with other communicators, it will have a unique rank within each of these as well.

```
MPI_Comm_rank (comm,&rank)
MPI_COMM_RANK (comm,rank,ierr)
```

Environment Management Routines

MPI_Abort

Terminates all MPI processes associated with the communicator. In most MPI implementations it terminates ALL processes regardless of the communicator specified.

```
MPI_Abort (comm,errorcode)
MPI_ABORT (comm,errorcode,ierr)
```

Environment Management Routines

MPI_Initialized

Indicates whether MPI_Init has been called - returns flag as either logical true (1) or false(0). MPI requires that MPI_Init be called once and only once by each process. This may pose a problem for modules that want to use MPI and are prepared to call MPI_Init if necessary. MPI_Initialized solves this problem.

```
MPI_Initialized (&flag)
MPI_INITIALIZED (flag,ierr)
```

Environment Management Routines

MPI_Wtime

Returns an elapsed wall clock time in seconds (double precision) on the calling processor.

```
MPI_Wtime ()
MPI_WTIME ()
```

MPI_Wtick

Returns the resolution in seconds (double precision) of MPI_Wtime.

```
MPI_Wtick ()
MPI_WTICK ()
```

Environment Management Routines

MPI_Finalize

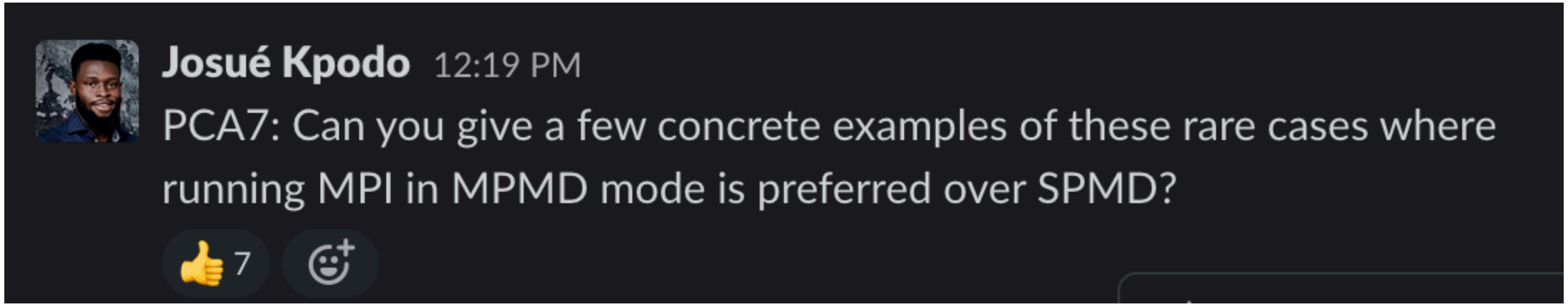
Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program - no other MPI routines may be called after it.

```
MPI_Finalize ()
MPI_FINALIZE (ierr)
```

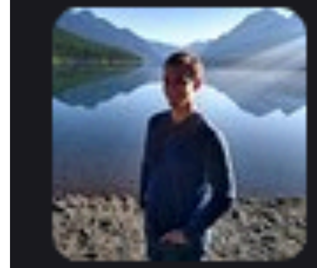

C Language - Environment Management Routines

```c
// required MPI include file
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
int  numtasks, rank, len, rc;
char hostname[MPI_MAX_PROCESSOR_NAME];

// initialize MPI
MPI_Init(&argc,&argv);

// get number of tasks
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);

// get my rank
MPI_Comm_rank(MPI_COMM_WORLD,&rank);

// this one is obvious
MPI_Get_processor_name(hostname, &len);
printf ("Number of tasks= %d My rank= %d Running on %s\n", numtasks,rank,hostname);


     // do some work with message passing


// done with MPI
MPI_Finalize();
}
```

PCA Questions



PCA Questions

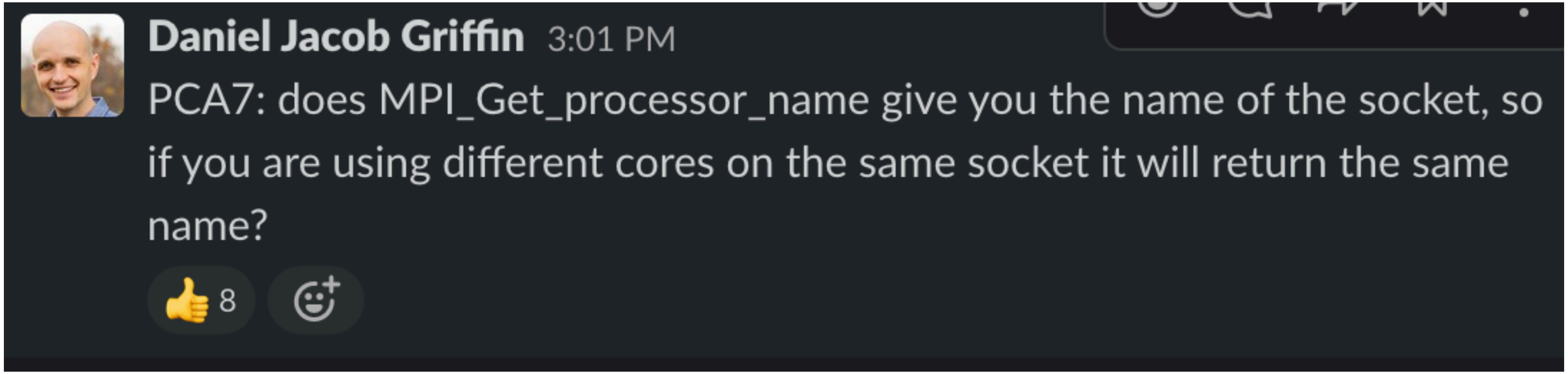


Stephen White 1:32 PM

PCA7: "However, the error codes are hardly ever useful, and there is not much your program can do to recover from an error." (PC 19)
This strikes me as pretty strange. What's the alternative to using error codes, how does one debug otherwise blackbox method calls like these? Why was MPI written in such a way that errors are so opaque? (edited)

👍 12

PCA Questions



The name returned should identify a particular piece of hardware; the exact format is implementation defined. This name may or may not be the same as might be returned by gethostname , uname , or sysinfo .