# PCSE Lecture 7

## Introduction to the Message Passing Interface Specification

Cyrus Proctor
Victor Eijkhout

March 12, 2015

# What is MPI?

- Message Passing Interface (MPI for short) is a specification for developers and users of message passing libraries
- MPI is NOT a library but a specification of what a library should be
- The MPI has gone through a number of revisions over the years; the most recent version (2012) is MPI-3
- Specifications have been defined for C and Fortran90 language bindings
  - C++ bindings from MPI-1 are now removed in MPI-3
  - MPI-3 also provides support for Fortran 2003 and 2008 features

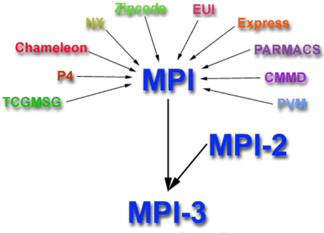
What is the goal of MPI?

- Provide a widely used standard for writing message passing programs
- The interface tries to be:
  - Practical
  - Portable
  - Efficient
  - Flexible
- Different library implementations differ in which version and/or features may be supported.

# Reasons for Using MPI

- Standardization – MPI is the only message passing library which 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 the MPI standard
- Performance – Vendor implementations should be able to exploit native hardware features to optimize performance (largely transparent to the users)
- Functionality – There are over 440 routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1
- Availability – A variety of implementations are available, both vendor supported and public domain libraries

# Evolution of MPI



- 80's → 90's – Distributed memory parallel computing develops along with many incompatible software tools; each with tradeoffs. Standard needed
- May 1992 – Workshop on standards for message passing with distributed memory; working group established
- Nov 1992 – MPI-1 draft proposal from ORNL presented. Created MPI forum comprised of about 175 individuals from 40 organizations
- Nov 1993 – MPI-1 draft presented at Supercomputing (SC); final version in May 1994
- 1996 – MPI-2 picked up where MPI-1 left off and went far beyond
- 2012 – MPI-3 standard was approved
- Documentation for all versions can be found at `http://www.mpi-forum.org/docs/`

# What's the Difference?

- Intentionally, the MPI-1 specification did not address several "difficult" issues. These issues were deferred, for expediency, to MPI-2
- MPI-2 had many areas of new functionality:
  - Dynamic Processes – extensions that remove the static process model of MPI. Provides routines to create new processes after job startup
  - One-Sided Communications – provides routines for one directional communications. Include shared memory operations (put/get) and remote accumulate operations
  - Extended Collective Operations – allows for the application of collective operations to inter-communicators
  - External Interfaces – defines routines that allow developers to layer on top of MPI, such as for debuggers and profilers
  - Additional Language Bindings – describes C++ bindings and discusses Fortran-90 issues
  - Parallel I/O – describes MPI support for parallel I/O

# What's the Difference?

- MPI-3 contains many extensions to MPI-1 and MPI-2 including:
  - Nonblocking Collective Operations – permits tasks in a collective to perform operations without blocking, possibly offering performance improvements
  - New One-sided Communication Operations – to better handle different memory models
  - Neighborhood Collectives – Extends the distributed graph and Cartesian process topologies with additional communication power
  - Fortran 2008 Bindings – expanded from Fortran90 bindings
  - MPIT Tool Interface – This new tool interface allows the MPI implementation to expose certain internal variables, counters, and other states to the user (most likely performance tools)
  - Matched Probe – Fixes an old bug in MPI-2 where one could not probe for messages in a multi-threaded environment

# Popular Implementations with MPI-3 Standard Compliance

- MPICH – Currently v3.1.4
  `http://www.mpich.org`
- MVAPICH2 – Currently v2.1*
  `http://mvapich.cse.ohio-state.edu`
- Intel MPI – Currently v5.0*
  `https://software.intel.com/en-us/intel-mpi-library`
- Open MPI – Currently v1.8.4
  `http://www.open-mpi.org`

*Versions likely to be released on Stampede some time this semester.

# Building MPI Code

Several Popular "Stacks" available on Stampede:

| Compiler | MV2 Version | IMPI Version |
|---|---|---|
| gcc/4.7.1 | mvapich2/1.9a2 | None |
| intel/13.0.2.146 | mvapich2/1.9a2 | impi/4.1.0.030 |
| intel/14.0.1.106 | mvapich2/2.0b | impi/4.1.3.049 |

A non-portable and **not recommended** way with the default stack is:

```
icc -c hello_mpi_c.c -I/opt/apps/intel13/mvapich2/1.9/include
icc -o hello_mpi_c hello_mpi_c.o -L/opt/apps/intel13/mvapich2/1.9/lib -lmpich


ifort -c hello_mpi_f.F90 -I/opt/apps/intel13/mvapich2/1.9/include
ifort -o hello_mpi_f hello_mpi_f.o -L/opt/apps/intel13/mvapich2/1.9/lib -lmpich
```

The portable and **recommended** way with **ALL** stacks is:

```
mpicc -c hello_mpi_c.c
mpicc -o hello_mpi_c hello_mpi_c.o


mpif90 -c hello_mpi_f.F90
mpif90 -o hello_mpi_f hello_mpi_f.o
```

# Running MPI Code

- Typically, mpirun is used
- mpirun is disabled on purpose on Stampede
- Typically use either an interactive session
- Or a batch submission script

# Running MPI on Local Nix-like Computers

- MPI programs require some help to get started
  - What computers should I run on?
  - How do I access those computers?
- MPICH-like Style:

```
mpirun -np <num_MPI_tasks> -machinefile <machine_filename> ./a.out
```

where a representative machine file and host file may be

### Machine File

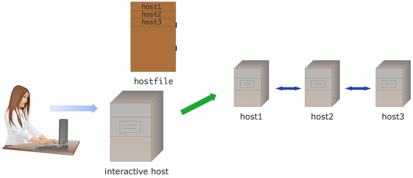
```
host1.foo.utexas.edu
host1.foo.utexas.edu
host2.foo.utexas.edu
host7.foo.utexas.edu
```

### /etc/hosts File

```
...snip
192.168.1.101 host1.foo.utexas.edu host1
192.168.1.102 host2.foo.utexas.edu host2
192.168.1.107 host7.foo.utexas.edu host7
```

For this particular example, num_MPI_tasks=4; two MPI tasks are run on host1, one MPI task on host2, and one MPI task on host7. These names are resolved from /etc/hosts file which then point to IP addresses.
Note: All this can be done with only one host!

# Running MPI Job Interactively on Stampede



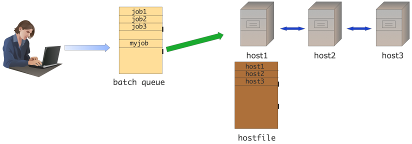
- Start an interactive session with “idev”
- May use command-line options -N and -n, e.g.:

```
idev -N <num_nodes> -n <num_MPI_tasks>
```

- Once on the master host, issue an ibrun command, e.g.:

```
ibrun tacc_affinity ./a.out
```

# Running MPI Job With Batch Submit on Stampede



## Example Batch Submit Script

```
#!/bin/bash
#SBATCH -J hello            # job name
#SBATCH -o hello.o.%j       # output filname
#SBATCH -e hello.e.%j       # error filename
#SBATCH -n 4                # mpi tasks requested
#SBATCH -N 2                # nodes requested
#SBATCH -p normal           # queue
#SBATCH -t 00:05:00         # run time (hh:mm:ss)
#SBATCH -A PCSE-2015        # account

ibrun tacc_affinity ./a.out
```

- Create batch submit script
- Slurm scheduler decides when and where
- Manages all jobs to efficiently utilize the whole machine

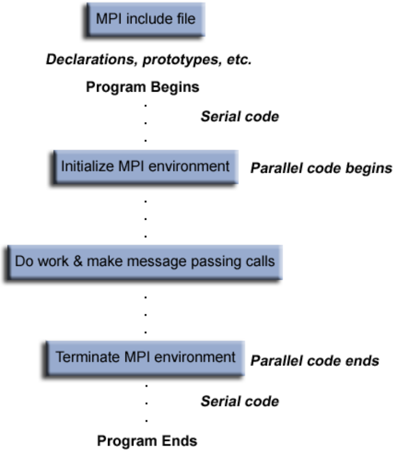
# General MPI Program Structure

- Parallel executables are nothing more than independent tasks/processes launched by ssh commands:

```
ssh <node_name> <environment> ./a.out
```

- The executables (./a.out) need to organize information (initialize)
- The executables need to synchronize
- The program needs to know its ID and number of executables
- The executables need to “clean up” when they’re done

# General MPI Program Structure



MPI include file

*Declarations, prototypes, etc.*

**Program Begins**

*Serial code*

Initialize MPI environment

*Parallel code begins*

Do work & make message passing calls

Terminate MPI environment

*Parallel code ends*

*Serial code*

**Program Ends**

# MPI Header Files

- Required for all programs that make MPI library calls
- For C codes:

```
#include "mpi.h"
```

- For Fortran2008 + TR 29113 and later codes:

```
use mpi_f08 **
```

- For Fortran90 codes:

```
use mpi
```

- For Fortran77 codes (backwards compatibility only):

```
include 'mpif.h'
```

**Not yet supported on Stampede.

# Format of MPI Calls

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

**C Binding:**

| | |
|---|---|
| Format | `ierr = MPI_Xxxxx(parameter, ... )` |
| Example | `ierr = MPI_Bsend(&buf,count,type,dest,tag,comm)` |
| Error Code | Returned as "ierror". MPI_SUCCESS if succesful |

**Fortran Binding:**

| | |
|---|---|
| Format | `CALL MPI_XXXXX(parameter,..., ierr)` |
| Example | `call MPI_BSEND(buf,count,type,dest,tag,comm,ierr)` |
| Error Code | Returned as "ierror" parameter. MPI_SUCCESS if succesful |

# 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 tasks

# MPI Ranks

- 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 through the total number of tasks per communicator
- 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” slide 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. We may explore this later
- The types of errors displayed to the user are implementation dependent

# 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)
```

# 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)
```

# 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)
```

# 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)
```

# 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)
```

# MPI_Get_processor_name

- Returns the processor name. Also returns the length of the name. The buffer for “name” must be at least MPI_MAX_PROCESSOR_NAME characters in size. What is returned into “name” is implementation dependent - may not be the same as the output of the “hostname” or “host” shell commands.

```
MPI_Get_processor_name (&name,&resultlength)
MPI_GET_PROCESSOR_NAME (name,resultlength,ierr)
```

# MPI_Wtime

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

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

# A Basic Fortran Example

## Basic Fortran MPI Example

```fortran
program simple
   use mpi
   implicit none

   integer numtasks, rank, len, ierr, errorcode
   character(MPI_MAX_PROCESSOR_NAME) hostname

   call MPI_INIT(ierr)
   if (ierr .ne. MPI_SUCCESS) then
      print *,'Error starting MPI program. Terminating.'
      call MPI_ABORT(MPI_COMM_WORLD, errorcode, ierr)
   end if

   call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
   call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
   call MPI_GET_PROCESSOR_NAME(hostname, len, ierr)
   print *, 'Number of tasks=',numtasks,' My rank=',rank, &
  &         ' Running on=',hostname

! ****** do some work ******

   call MPI_FINALIZE(ierr)

end program simple
```

# A Basic C Example

## Basic C MPI Example

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

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

  ierr = MPI_Init(&argc,&argv);
  if (ierr != MPI_SUCCESS) {
    printf ("Error starting MPI program. Terminating.\n");
    MPI_Abort(MPI_COMM_WORLD, ierr);
    }

  MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Get_processor_name(hostname, &len);
  printf ("Number of tasks= %d My rank= %d Running on %s\n", numtasks,rank,hostname);

  /*******  do some work *******/

  MPI_Finalize();
}
```

# References

- Victor Eijkhout, “Introduction to High Performance Scientific Computing”
- Victor Eijkhout, “Parallel Computing for Science and Engineering”
- Blaise Barney, “MPI Tutorial”
- Mark Lubin, “Introduction into new features of MPI-3.0 Standard”
- “MPI: A Message-Passing Interface Standard Version-3.0”