Introduction to MPI

Ritu Arora

Email: rauta@tacc.utexas.edu

February 6th 2012

Course Objectives & Assumptions

- Objectives
  - Teach basics of MPI-Programming
  - Share information related to running MPI programs on Ranger & Lonestar

- Assumptions
  - The audience has the basic understanding of C programming
    - Fortran binding will be mentioned where necessary
  - The audience has access to MPI installation either locally or remotely

Content Overview

- Basic concepts related to MPI
- Environment Management MPI routines
- Compiling and running MPI programs
- Types of communication
  - Point-to-Point communication routines
  - Collective communication routines
- Examples
- Summary

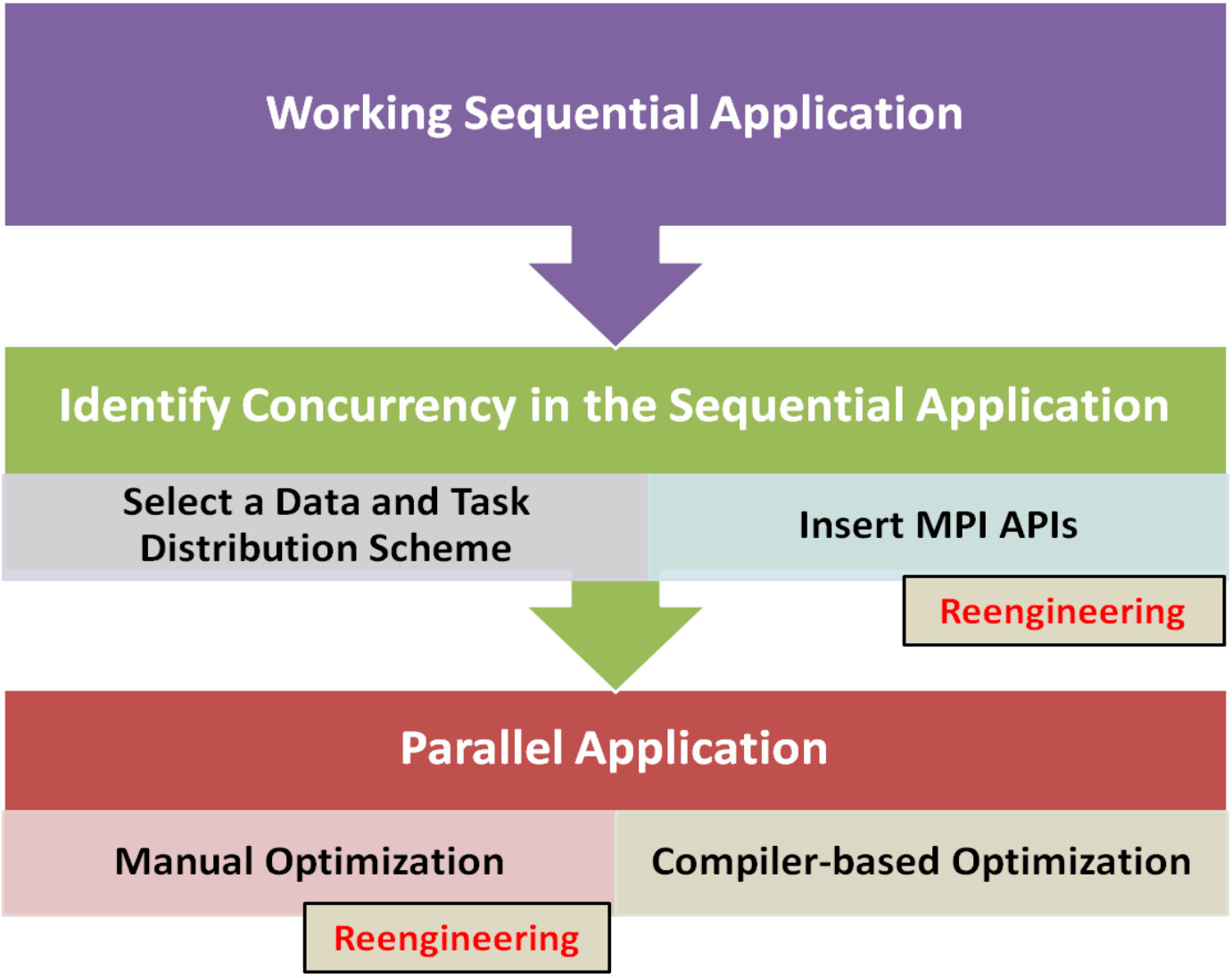
Message Passing Interface (MPI)

- MPI is a standard/specification for message passing library
  - Multiple vendor-specific implementations

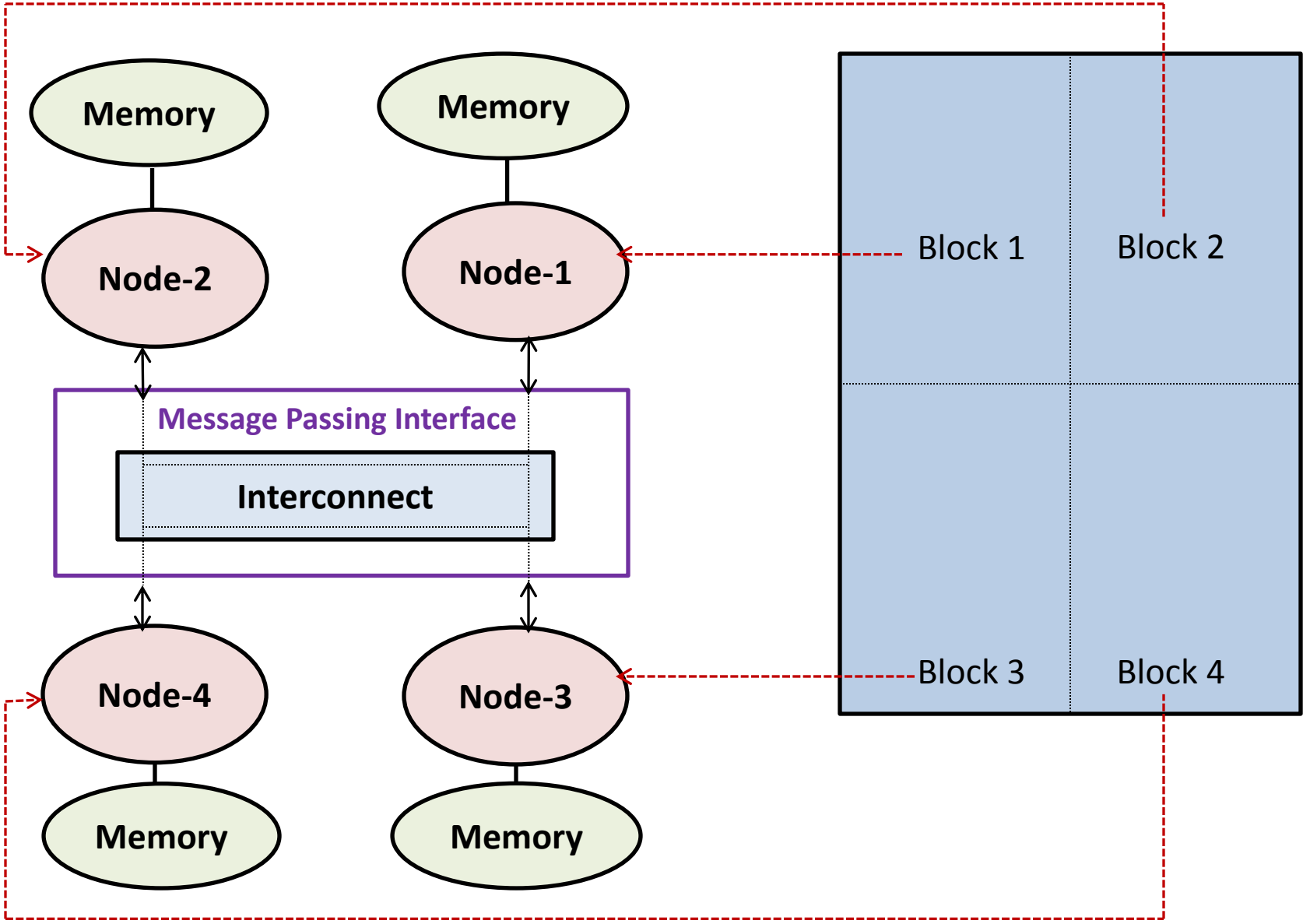
- Mainly used for programming systems with distributed memory
  - Where each process has a different address space
  - Processes need to communicate with each other
    - Synchronization
    - Data Exchange
  - Can also be used for shared memory and hybrid architectures

- MPI specifications have been defined for C, C++ and Fortran programs
    - MPI-1 versus MPI-2

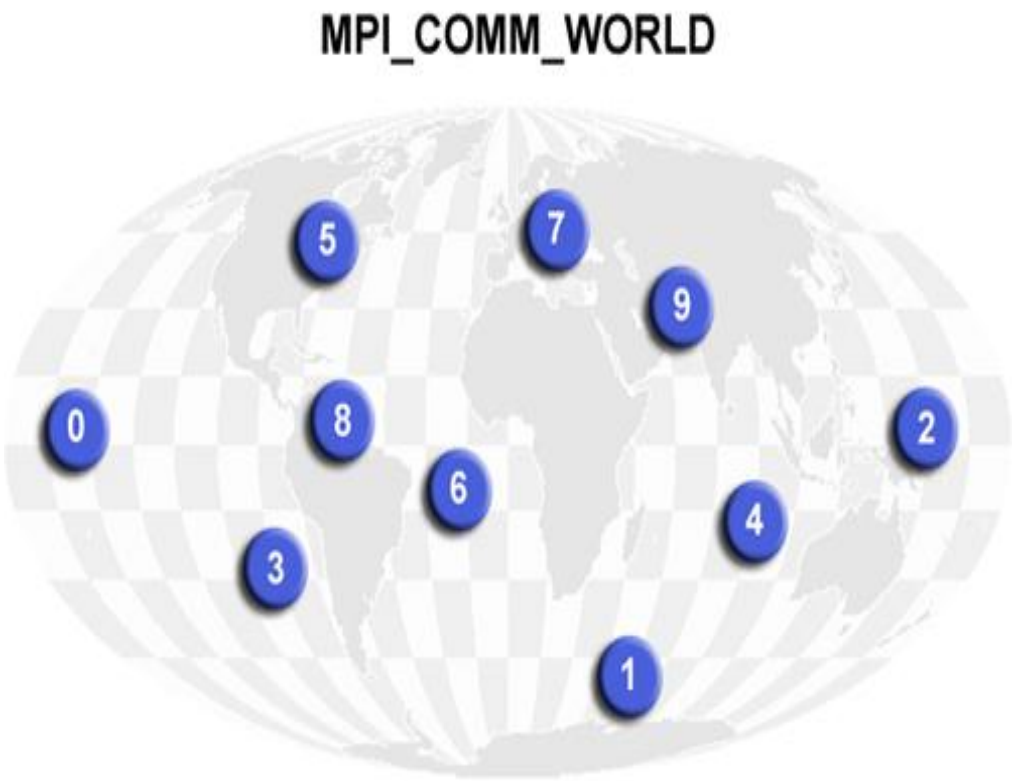
Explicit Parallelization with MPI (traditional way)



Divide & Conquer



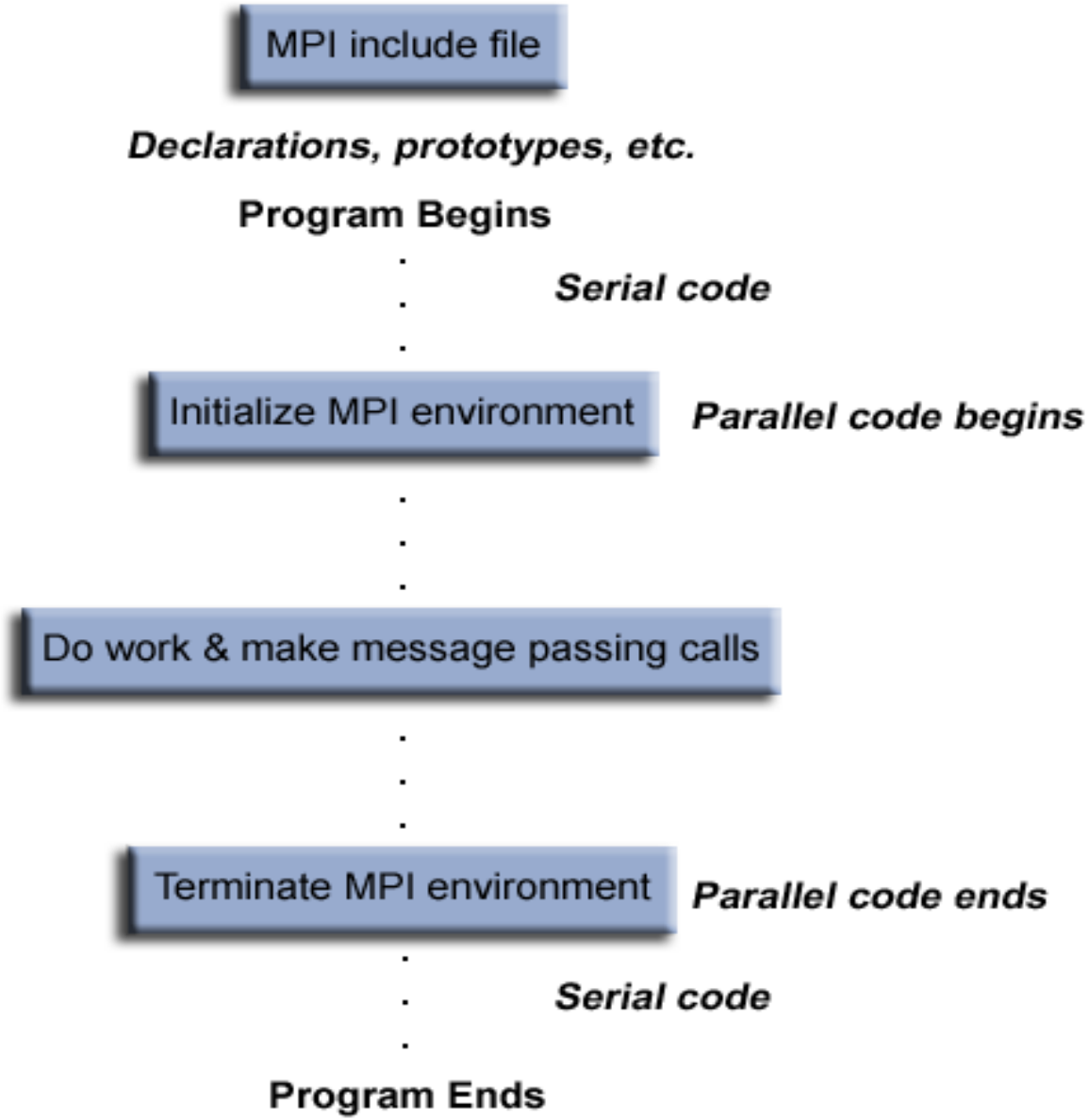
Concept of Communicators and Groups



- Communicators and groups are objects that are used to define which collection of processes may communicate with each other
- Most MPI routines require a communicator as an argument
- **`MPI_COMM_WORLD`** is the predefined communicator that includes all MPI processes
- Multiple communicators and groups can be defined

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

General Structure of MPI Programs



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

Arguments for MPI Routine (buffer, data count, data type, destination)

- **Buffer:** the name of a variable (including arrays and structures) that is to be sent/received. For C programs, this argument is passed by reference and usually must be prepended with an ampersand: **`&var1`**

- **Data Count:** number of data elements to be sent or received

- **Data Type:** could be elementary data type or derived

- **Destination:** the process where a message is to be delivered

Arguments for MPI Routine (source, tag, status, request)

- **Source:** indicates the process from which the message originated
- **Tag:** non-negative integer value to uniquely identify a message
- **Status:** for a receive operation, indicates the source and tag of a message
- **Request:** a unique "request number" issued by the system that can be used to check if a particular category of operation has completed or not (**more on this later**)

MPI Execution

- Each process runs a copy of the executable:
  Single **P**rogram, **M**ultiple **D**ata (SPMD)
- Each process picks the portion of the work according to its rank
- Each process works independent of the other processes, except when communicating

Content Overview

- Basic concepts related to MPI
- Environment Management MPI routines
- Compiling and running MPI programs
- Types of communication
  - Point-to-Point communication routines
  - Collective communication routines
- Examples
- Summary

Every MPI Program…

- Includes the MPI header file (`mpi.h`)
- Has a routine to initialize the MPI environment (`MPI_Init`)
- Has a routine to terminate the MPI environment (`MPI_Finalize`)

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

`MPI_Xxx(. . .);`	`CALL MPI_XXX(. . ., ierr)`
	
	`Call mpi_xxx(. . ., ierr)`

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

`MPI_Finalize()`	`MPI_FINALIZE(ierr)`

Environment Management Routines (1)

- `MPI_Init` initializes the MPI execution environment, must be called before any other MPI routine is called, and is invoked only once in an MPI program

- `MPI_Finalize` terminates the MPI execution environment and must be called in the last

- `MPI_Comm_size` determines the number of processes that are associated with a communicator (size is N, if N total processors are participating in a program run)
  - **`C: MPI_Comm_size (comm,&size)`**
  - **`Fortran: MPI_COMM_SIZE (comm,size,ierr)`**

Environment Management Routines (2)

- `MPI_Comm_rank` determines the number of processes within a communicator, ranges from 0 to N-1
  - **`C: MPI_Comm_rank (comm,&rank)`**
  - **`Fortran: MPI_COMM_RANK (comm,rank,ierr)`**

- `MPI_Wtime` is a timer routine that returns elapsed wall clock time in seconds
  - **`C: MPI_Wtime()`**
  - **`Fortran: MPI_WTIME()`**

Serial Program: `example1.c`

```
#include <stdio.h>

int main(){

  printf("Wonderful Class!\n");

  return(0);

}
```

Compiling:

```
login3$ icc -o example1 example1.c
```

Running:

```
login3$ ./example1
Wonderful Class!
```

Serial to Parallel: `example1.c` to `mpiExample1.c`

```
#include <stdio.h>

#include "mpi.h"

int main(){

  printf("Wonderful Class!\n");

  return(0);
}
```

Include the header file "mpi.h"

Serial to Parallel: `example1.c` to `mpiExample1.c`

```
#include <stdio.h>

#include "mpi.h"

int main(){

  MPI_Init(NULL, NULL);

  printf("Wonderful Class!\n");

  return(0);
}
```

Include the header file "mpi.h"

Start up MPI

Notice the NULL value being passed to `MPI_Init`. We will come back to this later.

Serial to Parallel: `example1.c` to `mpiExample1.c`

```
#include <stdio.h>

#include "mpi.h"

int main(){

  MPI_Init(NULL, NULL);

  printf("Wonderful Class!\n");

  MPI_Finalize();

  return(0);
}
```

Include the header file "mpi.h"

Start up MPI

Shut down MPI

Passing `NULL` to `MPI_Init`

- In MPI-1.1, an implementation is allowed to require that the arguments `argc` and `argv` that are passed to main, be also passed to `MPI_Init`

- In MPI-2 , conforming implementations are required to allow applications to pass NULL for both the `argc` and `argv` arguments of main

Content Overview

- Basic concepts related to MPI
- Environment Management MPI routines
- Compiling and running MPI programs
- Types of communication
  - Point-to-Point communication routines
  - Collective communication routines
- Examples
- Summary

Compiling `mpiExample1.c` on Ranger & Lonestar

- Compiling the program on Ranger
  - Both MPI-1 and MPI-2 available
  - You could either use Intel or the PGI compiler or gcc
  - Use module commands to choose the appropriate compiler/MPI stack

    Example:

```
login3$ module avail
login3$ module list
login3$ module swap pgi intel
login3$ module swap mvapich mvapich2
```

- Compiling the program on Lonestar
  - Only MPI-2 is available
  - You could either use Intel or gcc
  - Use module commands to choose the appropriate compiler/MPI stack

Compiling `mpiExample1.c` on Ranger & Lonestar

- Compiling the example program

```
login3$ mpicc -o mpiExample1 mpiExample1.c
```

Compiler	Program	File Extension
`mpicc`	C	.c
`mpicxx`	C++	Intel: .C/c/cc/cpp/cxx/c++ PGI: .C/c/cc/cpp/cxx/
`mpif90`	F77/F90	.f, .for, .ftn, .f90, .f95, .fpp

Running mpiExample1.c

- To run your application on TACC resources
  - Please consult the userguide and write a job script (**`myJob.sh`**)

    http://www.tacc.utexas.edu/user-services/user-guides/ranger-user-guide

    http://www.tacc.utexas.edu/user-services/user-guides/lonestar-user-guide

  - Pay attention to the "wayness"

    `-pe <TpN>way <NoN x 16>`

  - Submit the job to the SGE queue

    `login3$ qsub myJob.sh`

  - Remember that Ranger has 16 cores per node and Lonestar has 12 cores per node

Job Script for Ranger: `myJob.sh`

```
#!/bin/bash
#$ -V                          #Inherit the submission environment
#$ -cwd                        # Start job in submission directory
#$ -N myMPI                    # Job Name
#$ -j y                        # Combine stderr and stdout
#$ -o $JOB_NAME.o$JOB_ID       # Name of the output file
#$ -pe 16way 16                # Requests 16 tasks/node, 16 cores total
#$ -q normal                   # Queue name normal
#$ -l h_rt=01:30:00            # Run time (hh:mm:ss) - 1.5 hours
#$ -A xxxxx                    # Mention your account name (xxxxx)
set -x                         # Echo commands
ibrun ./mpiExample1            # Run the MPI executable
```

Note 1: On Lonestar you can request cores in multiples of 12 instead of 16

Note 2: `ibrun` is the wrapper for `mpirun/mpiexec` that is exclusive to TACC resources

Output from `mpiExample1.c`

```
login3$ cat myMPI.o2339942
…
TACC: Starting up job 2339942
TACC: Setting up parallel environment for MVAPICH ssh-based mpirun.
TACC: Setup complete. Running job script.
TACC: starting parallel tasks...
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
Wonderful Class!
TACC: Shutting down parallel environment.
TACC: Cleaning up after job: 2339942
TACC: Done.
```

Using Communicator: mpiExample2.c

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

3.  int main(int argc, char* argv[]){

4.     int  rank, size;

5.     MPI_Init(&argc, &argv);

6.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);

7.     MPI_Comm_size(MPI_COMM_WORLD, &size);

8.     printf("Hello MPI World from process %d!", rank);

9.     MPI_Finalize();

10.    return 0;
11. }
```

Extend the variable declaration section

Note argc and argv

Find process rank

Find out number of processes

Using rank

Output from `mpiExample2.c`

TACC: Setup complete. Running job script.
TACC: starting parallel tasks...
Hello MPI World from process 5!
Hello MPI World from process 1!
Hello MPI World from process 0!
Hello MPI World from process 6!
Hello MPI World from process 7!
Hello MPI World from process 2!
Hello MPI World from process 3!
Hello MPI World from process 4!
TACC: Shutting down parallel environment.
TACC: Shutdown complete. Exiting.
TACC: Cleaning up after job: 2340827
TACC: Done

Content Overview

- Basic concepts related to MPI
- Environment Management MPI routines
- Compiling and running MPI programs
- Types of communication
  - Point-to-Point communication routines
  - Collective communication routines
- Examples
- Summary

Modes of Communication

- Point-to-Point
  - Blocking
  - Non-Blocking
  - Synchronous
  - Buffered
  - Combined
- Collective

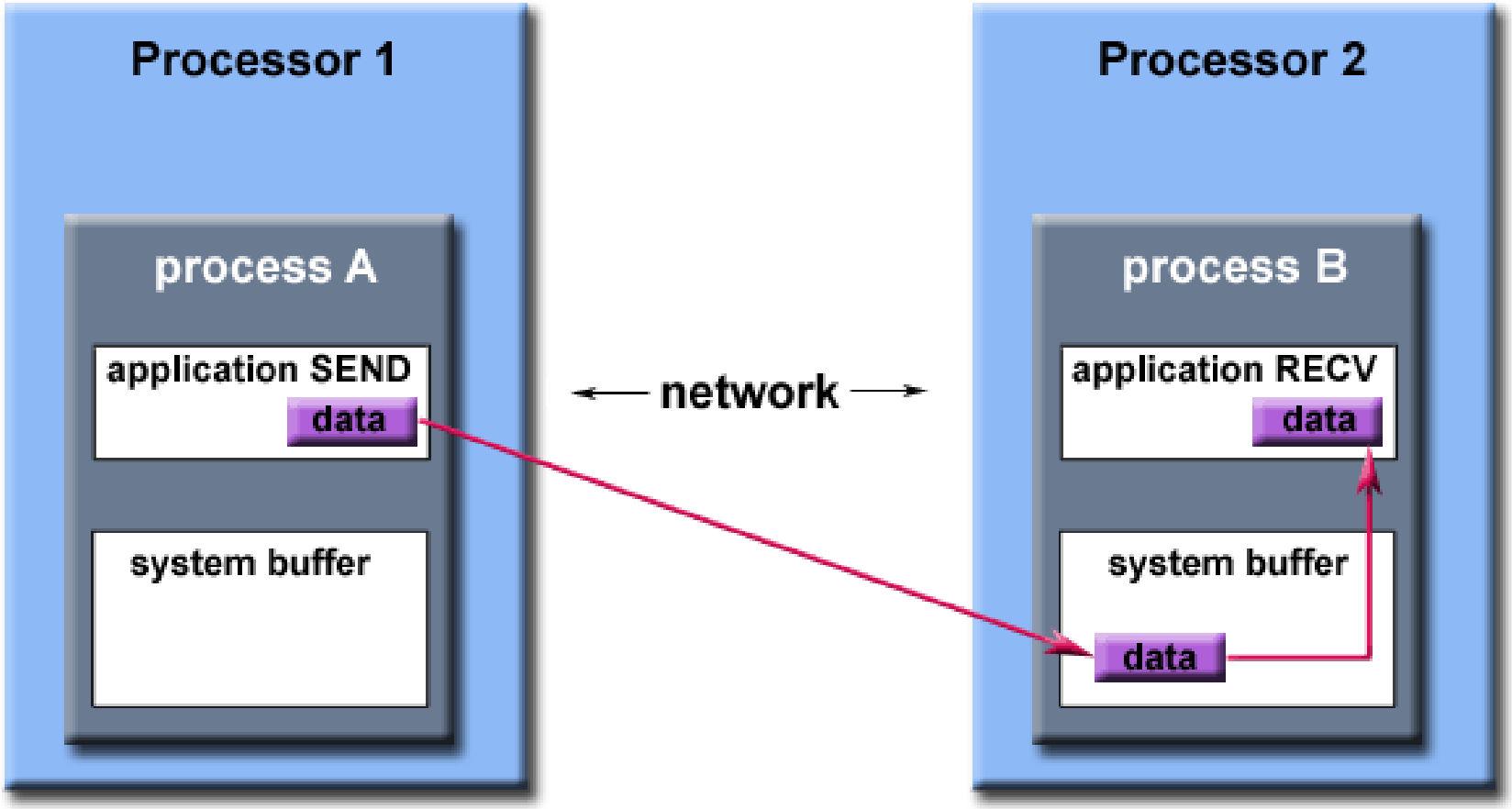
Content Overview

- Basic concepts related to MPI
- Environment Management MPI routines
- Compiling and running MPI programs
- Types of communication
  - Point-to-Point communication routines
  - Collective communication routines
  - Examples
- Summary

Point-to-Point Communication

- Involve message passing between two different MPI processes
- One process performs a send operation and the other task performs a matching receive operation
- There should be a matching receive routine for every send routine
  - If a send is not paired with a matching receive then the code will have a deadlock

Buffering



Path of a message buffered at the receiving process

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

Point-to-Point Communication (blocking versus non-blocking)

- Blocking:
  - A blocking receive only "returns" after the data has arrived and is ready for use by the program
  - A blocking send routine returns after it is safe to modify the application buffer for reuse
  - A blocking send can be either synchronous or asynchronous

- Non-blocking:
  - Non-blocking send and receive routines will return almost immediately
  - It is unsafe to modify the application buffer until you know for a fact that the requested non-blocking operation was actually performed

Point-to-Point Communication (blocking send)

```
MPI_Send(void *buf, int count, MPI_Datatype
   dType, int dest, int tag, MPI_Comm comm)
```

Argument	Description
buf	Initial address of the send buffer
count	Number of items to send
dType	MPI data type of items to send
dest	MPI rank or task that would receive the data
tag	Message ID
comm	MPI communicator where the exchange occurs

Some elementary data types: MPI_CHAR , MPI_SHORT , MPI_INT, MPI_LONG, MPI_FLOAT , MPI_DOUBLE , …

Point-to-Point Communication (blocking receive)

```
MPI_Recv(void *buf, int count, MPI_Datatype
  dType, int source, int tag, MPI_Comm comm,
  MPI_Status *status)
```

Argument	Description
buf	Initial address of receive buffer
count	Number of items to receive
dType	MPI data type of items to receive
source	MPI rank of task sending the data
tag	Message ID
comm	MPI communicator where the exchange occurs
status	Returns information on the message received, indicates the source of message and tag of the message

Point-to-Point Communication (blocking send and receive in code)

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

Point-to-Point Communication (this code will deadlock)

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

Point-to-Point Communication (blocking but combined send & receive)

- Send and Receive stages use the same communicator, but have distinct tags
- Useful for communication patterns where each node both sends and receives messages (two-way communication)

```
MPI_SendRecv(sendbuf, sendcount, sendtype,
   dest, sendtag,
   recvbuf, recvcount, recvtype, source,
   recvtag, comm, status);
```

- **Send arguments**
- **Receive arguments**
- **Common to both send and receive**

Point-to-Point Communication (non-blocking send & receive)

Non-blocking send	**`MPI_Isend(buffer,count,type,dest,tag,comm,request)`**
Non-blocking receive	`MPI_Irecv(buffer,count,type,source,tag,comm,request)`

- **`MPI_Request`** objects are used by non-blocking send & receive calls
  - In C, this argument is a pointer to a predefined structure named `MPI_Request`
  - The programmer uses this system assigned "handle" later (in a WAIT type routine) to determine completion of the non-blocking operation

Point-to-Point Communication (MPI_Wait)

- `MPI_Wait` is a blocking routine

```
MPI_Wait (&request, &status)
```

- It blocks until a specified non-blocking send or receive operation has completed

- Also check `MPI_Waitany, MPI_Waitall`

Content Overview

- Basic concepts related to MPI
- Environment Management MPI routines
- Compiling and running MPI programs
- Types of communication
  - Point-to-Point communication routines
  - Collective communication routines
- Examples
- Summary

Collective Communication

- Defined as communication between > 2 processes
  - One-to-many, many-to-one, many-to-many
- All processes within the communicator group call the same collective communication function with matching arguments
- Collective communication routines are blocking
- The size of data sent must exactly match the size of data received

Collective Communication (Synchronization)

- `MPI_Barrier` creates a barrier synchronization in a group
  - Each task, when reaching the MPI_Barrier call, blocks until all tasks in the group reach the same MPI_Barrier call

```
MPI_Barrier (comm)
```

Collective Communication (Data Movement)

- `MPI_Bcast` broadcasts (sends) a message from the process designated as "root" to all other processes in the group

  `MPI_Bcast(&buffer,count,datatype,root,comm)`

- `MPI_Scatter` distributes distinct messages from a single source task to each task in the group

  `MPI_Scatter(&sendbuf,sendcnt,sendtype,`
  `&recvbuf,recvcnt,recvtype,root,comm)`

Collective Communication (Data Movement)

- `MPI_Gather` is reverse of `MPI_Scatter` and gathers distinct messages from each task in the group to a single destination task

```
MPI_Gather(&sendbuf,sendcnt,sendtype,
   &recvbuf,recvcount,recvtype,root,comm)
```

- `MPI_Allgather` gathers data from all tasks in a group and distributes to all tasks

```
MPI_Allgather(&sendbuf,sendcount,
sendtype,&recvbuf, recvcount,recvtype,comm)
```

Collective Communication (collective computation)

- `MPI_Reduce` applies a reduction operation on all tasks in a group and places the result in one task

```
MPI_Reduce (&sendbuf,&recvbuf, count,
   datatype, mpi_red_operation, root, comm)
```

MPI Reduction Operation	Description
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product

Collective Communication (collective computation)

- `MPI_Allreduce` applies a reduction operation on all tasks in a group and passes the result to all tasks

```
MPI_Allreduce (&sendbuf,&recvbuf, count,
   datatype, mpi_red_operation, comm)
```

- Many more functions that the audience might want to explore on their own, example, `MPI_Reduce_scatter`, `MPI_All_to_all`, `MPI_Scatterv`, `MPI_Gatherv`, …

Content Overview

- Basic concepts related to MPI
- Environment Management MPI routines
- Compiling and running MPI programs
- Types of communication
  - Point-to-Point communication routines
  - Collective communication routines
- Examples
- Summary

Sequential Program with a For-Loop: example4.c

```
1.   #include <stdio.h>
2.   int main(int argc,char *argv[]){

3.       int i, sum, upToVal;
4.       upToVal = 10000;
5.       sum = 0;
6.
7.       for(i=1; i<= upToVal; i++){
8.            sum = sum +i;
9.       }

10.      printf("\nSum is %d\n", sum);

11.      return 0;

12.  }
```

For-Loop & MPI_Reduce: mpiExample4.c (1)

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

3.  int main(int argc,char *argv[]){
4.      int i, sum, sumTotal, upToVal;
5.      int start, end, size, rank;

6.      upToVal = 10000;

7.      MPI_Init(&argc,&argv);
8.      MPI_Comm_size(MPI_COMM_WORLD, &size);
9.      MPI_Comm_rank(MPI_COMM_WORLD, &rank);

10.     start = rank*(upToVal/size) + 1;
11.     if(rank==(size-1)){
12.         end = upToVal;
13.     }else{
14.         end = start + (upToVal/size)-1;
15.     }
```

For-Loop & MPI_Reduce: mpiExample4.c (2)

```
16.   sum = 0;
17.   sumTotal=0;
18.   for(i=start; i<= end; i++){
19.      sum = sum +i;
20.   }
21.   MPI_Reduce (&sum, &sumTotal, 1, MPI_INT, MPI_SUM, 0,
                                              MPI_COMM_WORLD );

22.   printf("\nRank: %d, sum: %d, sumTotal: %d\n", rank,
                                              sum, sumTotal);

23.   MPI_Finalize();
24.   return 0;
25. }
```

Output from `mpiExample4.c`

Rank: 6, sum: 10156875, sumTotal: 0

Rank: 4, sum: 7031875, sumTotal: 0

Rank: 7, sum: 11719375, sumTotal: 0

Rank: 5, sum: 8594375, sumTotal: 0

Rank: 3, sum: 5469375, sumTotal: 0

Rank: 2, sum: 3906875, sumTotal: 0

Rank: 1, sum: 2344375, sumTotal: 0

Rank: 0, sum: 781875, sumTotal: 50005000

MPI_Bcast Example: mpiExample7.c (1)

```
1.  #include <stdio.h>
2.  #include <mpi.h>

3.  int main(int argc, char *argv[]){
4.      int i,rank,size;
5.      int root,count;
6.      int buffer[4];

7.      MPI_Status status;
8.      MPI_Request request;

9.      MPI_Init(&argc,&argv);
10.     MPI_Comm_size(MPI_COMM_WORLD,&size);
11.     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
12.     root=0;
13.     count=4;
```

MPI_Bcast Example: mpiExample7.c (2)

```
14.   if(rank == root){
15.       for(i=0; i<count; i++){
16.         buffer[i]=i;
17.       }
18.   }
19.   MPI_Bcast(buffer,count,MPI_INT,root,MPI_COMM_WORLD);

20.   printf("Rank is: %d, Value at buffer[%d] is: %d \n",
                          rank, count-1, buffer[count-1]);

21.   printf("\n");
22.   MPI_Finalize();
23.   return 0;
24. }
```

Output from `mpiExample7.c`

```
TACC: starting parallel tasks...
Rank is: 0, Value at buffer[4] is: 4

Rank is: 1, Value at buffer[4] is: 4

Rank is: 2, Value at buffer[4] is: 4
 Rank is: 3, Value at buffer[4] is: 4



Rank is: 6, Value at buffer[4] is: 4
 Rank is: 4, Value at buffer[4] is: 4
 Rank is: 7, Value at buffer[4] is: 4
 Rank is: 5, Value at buffer[4] is: 4

TACC: Shutting down parallel environment.
```

Note: Do not expect the output to be printed in any particular order. You might see jumbled up output.

Content Overview

- Basic concepts related to MPI
- Environment Management MPI routines
- Compiling and running MPI programs
- Types of communication
  - Point-to-Point communication routines
  - Collective communication routines
- Examples
- Summary

Summary of Key MPI Routines

C	Fortran
`MPI_Init (&argc,&argv)`	`MPI_INIT (ierr)`
`MPI_Comm_size (comm,&size)`	`MPI_COMM_SIZE (comm,size,ierr)`
`MPI_Comm_rank (comm,&rank)`	`MPI_COMM_RANK (comm,rank,ierr)`
`MPI_Finalize ()`	`MPI_FINALIZE (ierr)`
`MPI_Send(&buf,count,datatype,…)`	`MPI_SEND (buf,count,datatype,…)`
`MPI_Recv(&buf,count,datatype,…)`	`MPI_RECV(&buf,count,datatype,…)`
`MPI_Wtime()`	`MPI_WTIME()`

Words of Caution!

- Not all applications can be parallelized
  - Analyze and understand the data dependencies in your application

- Not all parallelization result in speed-up (parallel slowdown)
  - Too much communication could be an overkill!

Note:

Total Execution Time = Computation Time + Communication Time + I/O time

References

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

http://www.mpi-forum.org

http://www.cs.usfca.edu/~peter/ppmpi/

http://www.mcs.anl.gov/research/projects/mpi/usingmpi/

http://geco.mines.edu/workshop/class2/examples/mpi/index.html

For Fortran Users

Sample MPI code (F90)

```
program samplempi
        use mpi
        [other includes]

        integer :: ierr, np, rank
        [other declarations]

        call mpi_init(ierr)
        call mpi_comm_size(MPI_COMM_WORLD, np, ierr)
        call mpi_comm_rank(MPI_COMM_WORLD, rank, ierr)
                        :
        [actual work goes here]
                        :
        call mpi_finalize(ierr)
end program
```

Send/Recv Pairs in Code

- **Blocking Send & Blocking Recv**

```
IF (rank==0) THEN
  CALL MPI_SEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ierr)
ELSEIF (rank==1) THEN
  CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)
ENDIF
```

- **Non-blocking Send & Blocking Recv**

```
IF (rank==0) THEN
  CALL MPI_ISEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req,ierr)
ELSEIF (rank==1) THEN
  CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)
ENDIF
CALL MPI_WAIT(req, wait_status)
```

Deadlock Example

```
! The following code contains a deadlock... can you spot it?
IF (rank==0) THEN
      CALL MPI_RECV(recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ierr)
      CALL MPI_SEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ierr)
ELSEIF (rank==1) THEN
      CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)
      CALL MPI_SEND(sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ierr)
ENDIF

! Solution
IF (rank==0) THEN
      CALL MPI_SEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ierr)
      CALL MPI_RECV(recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ierr)
ELSEIF (rank==1) THEN
      CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)
      CALL MPI_SEND(sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ierr)
ENDIF
```

Alternative Deadlock Solutions

```
! Solution using sendrecv
IF (rank==0) THEN
      CALL MPI_SENDRECV(sendbuf, count, MPI_REAL, 1, sendtag,
                        recvbuf, count, MPI_REAL, 1, recvtag,
                        MPI_COMM_WORLD, status, ierr)
ELSEIF (rank==1) THEN
      CALL MPI_SENDRECV(sendbuf, count, MPI_REAL, 0, sendtag,
                        recvbuf, count, MPI_REAL, 0, recvtag,
                        MPI_COMM_WORLD, status, ierr)
ENDIF
```

```
! Another possible solution (using all non-blocking calls)
IF (rank==0) THEN
      CALL MPI_ISEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req1,ierr)
      CALL MPI_IRECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req2,ierr)
ELSEIF (rank==1) THEN
      CALL MPI_ISEND(sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req1,ierr)
      CALL MPI_IRECV(recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req2,ierr)
ENDIF
CALL MPI_WAIT(req1, wait_status, ierr)
CALL MPI_WAIT(req2, wait_status, ierr)
```

Additional Information

If you do not have access to a cluster…

- To compile and run MPI programs on your PC or Laptop
  - Download and install the right MPI package

http://www.mcs.anl.gov/research/projects/mpich2/downloads/index.php?s=downloads

  - Download the C/C++/Fortran Compiler

  - You might also want to download an IDE like Eclipse