Introduction to MPI

2021 HPC Workshop: Parallel Programming

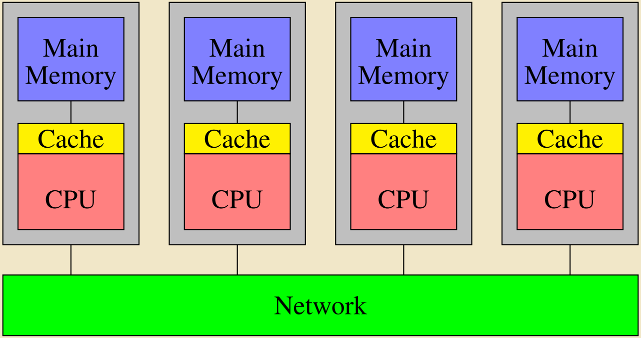
Alexander B. Pacheco

Research Computing

July 13 - 15, 2021

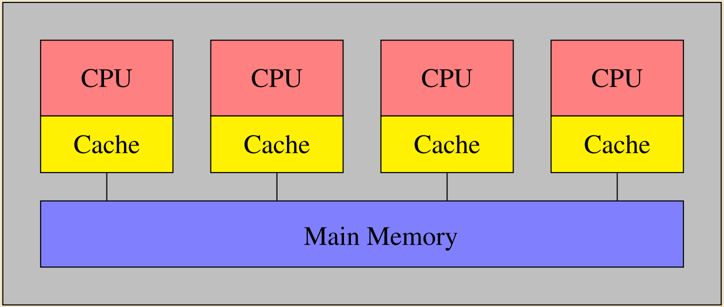
Distributed Memory Model

- Each process has its own address space
  - Data is local to each process
- Data sharing is achieved via explicit message passing
- Example
  - MPI



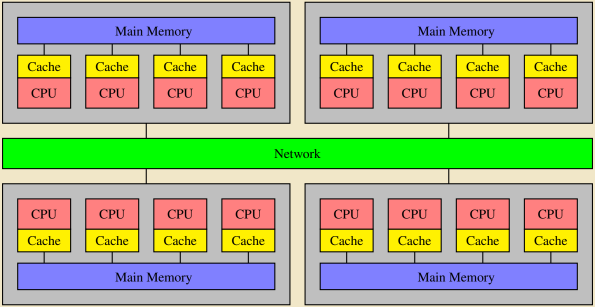
Shared Memory Model

- All threads can access the global memory space.
- Data sharing achieved via writing to/reading from the same memory location
- Example
  - OpenMP
  - Pthreads



Clusters of SMP nodes

- The shared memory model is most commonly represented by Symmetric Multi-Processing (SMP) systems
  - Identical processors
  - Equal access time to memory
- Large shared memory systems are rare, clusters of SMP nodes are popular.



Shared vs Distributed

Shared Memory

- Pros
  - Global address space is user friendly
  - Data sharing is fast
- Cons
  - Lack of scalability
  - Data conflict issues

Distributed Memory

- Pros
  - Memory scalable with number of processors
  - Easier and cheaper to build
- Cons
  - Difficult load balancing
  - Data sharing is slow

Why MPI?

- **Standardization**: MPI is the only message passing library that can be considered a standard. It is supported on virtually all HPC platforms.
- **Portability**: There is little or no need to modify your source code when you port your application to a different platform.
- **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.

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

What is MPI?

- MPI defines a standard API for message passing
  - The standard includes
    - What functions are available
    - The syntax of those functions
    - What the expected outcome is when calling those functions
  - The standard does NOT include
    - Implementation details (e.g. how the data transfer occurs)
    - Runtime details (e.g. how many processes the code run with etc.)
- MPI provides C/C++ and Fortran bindings

MPI Implementations

- There are two different types of MPI implementations commonly used.
  1. **MPICH**: Developed by Argonne National Laboratory.
     - used as a starting point for various commercial and open source MPI libraries
     - **MVAPICH2**: Developed by D. K. Panda with support for InfiniBand, iWARP, RoCE, and Intel Omni-Path. (default MPI on Sol),
     - **Intel MPI**: Intel's version of MPI. Part of Intel OneAPI HPC Toolkit
     - **IBM MPI**: for IBM BlueGene, and
     - **CRAY MPI**: for Cray systems.
  2. **OpenMPI**: A Free, Open Source implementation from merger of three well know MPI implementations. Can be used for commodity network as well as high speed network.
     - **FT-MPI** from the University of Tennessee,
     - **LA-MPI**: from Los Alamos National Laboratory,
     - **LAM/MPI**: from Indiana University

MPI Compilers

- There is no MPI compiler available to compile programs nor is there is a compiler flag.
- Instead, you need to build the MPI scripts for a particular compiler.
- You can use MVAPICH2 and MPICH on Sol
- You should use MPICH only on Hawk.
- Each of these builds provide mpicc, mpicxx and mpif90 for compiling C, C++ and Fortran codes respectively that are wrapper for the underlying compilers

```
[alp514.sol](793): module load mvapich2
[alp514.sol](794): mpicc -show
/share/Apps/intel/2020/compilers_and_libraries_2020.3.275/linux/bin/intel64/icc -lmpi -I/share/Apps/lusoft/opt/spack/linux-
        centos8-haswell/intel-20.0.3/mvapich2/2.3.4-wguydha/include -L/share/Apps/lusoft/opt/spack/linux-centos8-haswell/intel-
        20.0.3/mvapich2/2.3.4-wguydha/lib -Wl,-rpath -Wl,/share/Apps/lusoft/opt/spack/linux-centos8-haswell/intel-20.0.3/
        mvapich2/2.3.4-wguydha/lib
[alp514.sol](795): module load mpich

Lmod is automatically replacing ”mvapich2/2.3.4” with ”mpich/3.3.2”.

[alp514.sol](796): mpif90 -show
/share/Apps/intel/2020/compilers_and_libraries_2020.3.275/linux/bin/intel64/ifort -L/share/Apps/lusoft/opt/spack/linux-centos8-
        haswell/intel-20.0.3/hwloc/2.2.0-rjrzfy7/lib -I/share/Apps/lusoft/opt/spack/linux-centos8-haswell/intel-20.0.3/mpich/3.
        3.2-n7f36fo/include -I/share/Apps/lusoft/opt/spack/linux-centos8-haswell/intel-20.0.3/mpich/3.3.2-n7f36fo/include -L/
        share/Apps/lusoft/opt/spack/linux-centos8-haswell/intel-20.0.3/mpich/3.3.2-n7f36fo/lib -lmpifort -Wl,-rpath -Wl,/share/
        Apps/lusoft/opt/spack/linux-centos8-haswell/intel-20.0.3/mpich/3.3.2-n7f36fo/lib -lmpi
```

Running MPI Applications

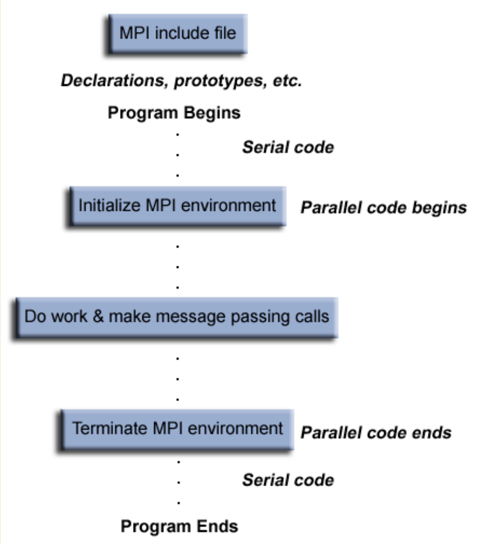
- To run MPI applications, you need to launch the application using
  - mpirun (OpenMPI),
  - mpirun_rsh (MPICH and MVAPICH2), or
  - mpiexec (OpenMPI, MPICH and MVAPICH2).
- mpirun, mpirun_rsh and mpiexec are schedulers for the MPI library.
- On clusters with SLURM scheduler, srun can be used to launched MPI applications
- The MPI scheduler needs to be given additional information to correctly run MPI applications

	mpiexec	mpirun_rsh	mpirun
# Processors	-n numprocs	-n numprocs	-np numprocs
Processors List	-hosts core1,core2,...	core1 core2 ...	-hosts core1,core2,...
Processor filelist	-f file	-hostfile file	-f/-hostfile file

Running MPI Applications

- Run an application myapp on 72 processors on a total of 3 nodes - node1, node2 and node3
  - mpirun: `mpirun -np 72 -f filename myapp`
  - mpirun_rsh: `mpirun_rsh -np 72 -hostfile filename myapp`
  - mpiexec: `mpiexec -n 72 -hosts node1,node2,node3 -ppn 24 myapp`
- The SLURM scheduler's srun launcher has information needed to run a mpi job
  - srun: `srun myapp`

MPI Program Structure



MPI Program Structure

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

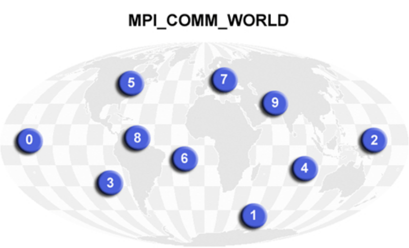
C	Fortran
`#include "mpi.h"`	`include 'mpif.h'`

- 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)`
Example	`CALL MPI_BSEND(buf,count,type,dest,tag,comm,ierr)`
Error code	Returned as "ierr" parameter. MPI_SUCCESS if successful

MPI Program Structure

- 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.
- `MPI_COMM_WORLD`: the default communicator contains all processes running a MPI program.



- Every process has its own unique, integer identifier assigned, called rank, by the system when the process initializes

Environment management functions

- **MPI_INIT**: Initialize the MPI environment
- **MPI_COMM_SIZE**: Return total number of MPI processes
- **MPI_COMM_RANK**: Return rank of calling process
- **MPI_ABORT**: Terminates all MPI processes
- **MPI_GET_PROCESSOR_NAME**: Returns the processor name.
- **MPI_GET_VERSION**: Returns the version and subversion of the MPI standard
- **MPI_INITIALIZED**: Indicates whether MPI_Init has been called
- **MPI_WTIME**: Returns an elapsed wall clock time in seconds
- **MPI_WTICK**: Returns the resolution in seconds of MPI_WTIME
- **MPI_FINALIZE**: Terminate the MPI environment

Environment management functions

C/C++

```
MPI_Init (&argc,&argv)
MPI_Comm_size (comm,&size)
MPI_Comm_rank (comm,&rank)
MPI_Abort (comm,errorcode)
MPI_Get_processor_name (&name,&
      resultlength)
MPI_Get_version (&version,&subversion)
MPI_Initialized (&flag)
MPI_Wtime ()
MPI_Wtick ()
MPI_Finalize ()
```

Fortran

```
MPI_INIT (ierr)
MPI_COMM_SIZE (comm,size,ierr)
MPI_COMM_RANK (comm,rank,ierr)
MPI_ABORT (comm,errorcode,ierr)
MPI_GET_PROCESSOR_NAME (name,resultlength
      ,ierr)
MPI_GET_VERSION (version,subversion,ierr)
MPI_INITIALIZED (flag,ierr)
MPI_WTIME ()
MPI_WTICK ()
MPI_FINALIZE (ierr)
```

Exercise: Writing your first MPI program

- Take the hello world code and add a few Environment Management functions
- Compile your code
- Run your code several different ways

- Examples to try out
  1. Print hostname
  2. Print mpi version
  3. Print hostname if your rank is odd and mpi version if rank is even

First MPI Program

C

```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);

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

Fortran

```fortran
program simple

  ! required MPI include file
  include 'mpif.h'

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

  ! initialize MPI
  call MPI_INIT(ierr)

  ! get number of tasks
  call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

  ! get my rank
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

  ! this one is obvious
  call MPI_GET_PROCESSOR_NAME(hostname, len, ierr)
  print '(a,i2,a,i2,a,a)', 'Number of tasks=',numtasks,'
          My rank=',rank,&
       ' Running on ',hostname

  ! done with MPI
  call MPI_FINALIZE(ierr)

end program simple
```

Compile & Run

```
[alp514.sol](1024): module load intel mpich

Lmod is automatically replacing ``mvapich2/2.3.4'' with ``mpich/3.3.2''.

[alp514.sol](1025): mpicc -o helloc hello.c
[alp514.sol](1026): mpif90 -o hellof hello.f90
[alp514.sol](1027): srun -p hawkgpu -n 4 -t 10 ./hellof
Number of tasks= 4 My rank= 0 Running on hawk-b624.cc.lehigh.edu
Number of tasks= 4 My rank= 1 Running on hawk-b624.cc.lehigh.edu
Number of tasks= 4 My rank= 3 Running on hawk-b624.cc.lehigh.edu
Number of tasks= 4 My rank= 2 Running on hawk-b624.cc.lehigh.edu
[alp514.sol](1028): srun -p hawkgpu -n 4 -t 10 ./helloc
Number of tasks= 4 My rank= 0 Running on hawk-b624.cc.lehigh.edu
Number of tasks= 4 My rank= 1 Running on hawk-b624.cc.lehigh.edu
Number of tasks= 4 My rank= 3 Running on hawk-b624.cc.lehigh.edu
Number of tasks= 4 My rank= 2 Running on hawk-b624.cc.lehigh.edu
[alp514.sol](1031): srun -p lts -n 4 -N 4 -t 10 ./helloc
Number of tasks= 4 My rank= 0 Running on sol-a105.cc.lehigh.edu
Number of tasks= 4 My rank= 2 Running on sol-a107.cc.lehigh.edu
Number of tasks= 4 My rank= 3 Running on sol-a108.cc.lehigh.edu
Number of tasks= 4 My rank= 1 Running on sol-a106.cc.lehigh.edu
```

Compiling MPI Programs

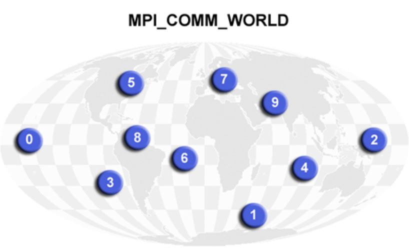
- Not a part of the standard
  - Could vary from platform to platform
  - Or even from implementation to implementation on the same platform
  - mpicc/mpicxx/mpif77/mpif90: wrappers to compile MPI code and auto link to startup and message passing libraries

Compiling MPI Programs

- Not a part of the standard
  - Could vary from platform to platform
  - Or even from implementation to implementation on the same platform
  - mpicc/mpicxx/mpif77/mpif90: wrappers to compile MPI code and auto link to startup and message passing libraries
- Unlike OpenMP and OpenACC, you cannot compile a MPI program for running in serial using the serial compiler
- The MPI program is not a standard C/C++/Fortran program and will spit out errors about missing libraries

Communicators

- A communicator is an identifier associated with a group of processes




```
MPI_Comm_size(MPI_COMM_WORLD,int &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,int &rank);

call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
```

Communicators

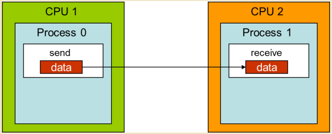
- A communicator is an identifier associated with a group of processes
  - Can be regarded as the name given to an ordered list of processes
  - Each process has a unique rank, which starts from 0 (usually referred to as "root")
  - It is the context of MPI communications and operations
    - For instance, when a function is called to send data to all processes, MPI needs to understand what "all"
  - `MPI_COMM_WORLD`: the default communicator contains all processes running a MPI program
  - There can be many communicators
    - e.g., `MPI_Comm_split(MPI_Commcomm, intcolor, int, kye, MPI_Comm* newcomm)`
  - A process can belong to multiple communicators
    - The rank is usually different

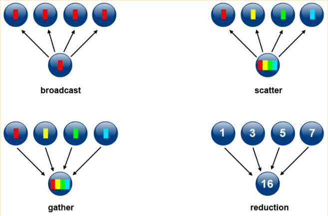
Communicator Information

- Rank: unique id of each process
  - C: `MPI_Comm_Rank(MPI_Comm comm, int *rank)`
  - Fortran: `MPI_COMM_RANK(COMM, RANK, ERR)`
- Get the size/processes of a communicator
  - C: `MPI_Comm_Size(MPI_Comm comm, int *size)`
  - Fortran: `MPI_COMM_SIZE(COMM, SIZE, ERR)`

Communicator Functions

- Point-to-point communication functions
  - Message transfer from one process to another
- Collective communication functions
  - Message transfer involving all processes in a communicator



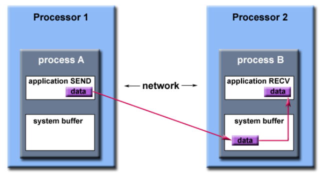


Point-to-point Communication

- MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks.
- One task is performing a send operation and the other task is performing a matching receive operation.
- There are different types of send and receive routines used for different purposes.
  1. Blocking send / blocking receive
  2. Non-blocking send / non-blocking receive
  3. Synchronous send

Buffering

- Ideally, every send operation would be perfectly synchronized with its matching receive.
- MPI implementation must be able to deal with storing data when the two tasks are out of sync.
- Consider the following two cases:
  1. A send operation occurs 5 seconds before the receive is ready - where is the message while the receive is pending?
  2. Multiple sends arrive at the same receiving task which can only accept one send at a time - what happens to the messages that are backing up?
- MPI implementation (not the MPI standard) decides what happens to data in these types of cases.
- Typically, a system buffer area is reserved to hold data in transit.



Path of a message buffered at the receiving process

System buffer space

- Opaque to the programmer and managed entirely by the MPI library
- A finite resource that can be easy to exhaust
- Often mysterious and not well documented
- Able to exist on the sending side, the receiving side, or both
- Something that may improve program performance because it allows send - receive operations to be asynchronous.

Blocking vs. Non-blocking

Blocking send / receive

- send will "return" after it is safe to modify the application buffer (your send data) for reuse
- send can be synchronous i.e. handshake with the receive task to confirm a safe send.
- send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.
- receive only "returns" after the data has arrived and is ready for use by the program.

LEHIGH UNIVERSITY

Non-blocking send / receive

- behave similarly - they will return almost immediately.
- do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
- operations simply "request" the MPI library to perform the operation when it is able.

  The user can not predict when that will happen.
- communications are primarily used to overlap computation with communication and exploit possible performance gains.

Blocking vs. Non-blocking

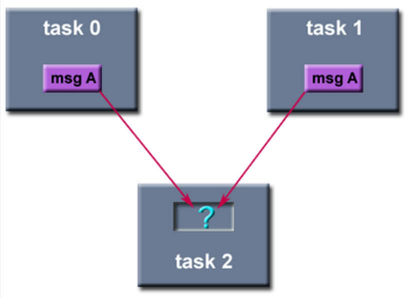
Order

- MPI guarantees that messages will not overtake each other.
- If a sender sends two messages (Message 1 and Message 2) in succession to the same destination, and both match the same receive, the receive operation will receive Message 1 before Message 2.
- If a receiver posts two receives (Receive 1 and Receive 2), in succession, and both are looking for the same message, Receive 1 will receive the message before Receive 2.
- Order rules do not apply if there are multiple threads participating in the communication operations.

Blocking vs. Non-blocking

Fairness

- MPI does not guarantee fairness - its up to the programmer to prevent operation starvation.



Point-to-point Communication Functions

Blocking send / receive

- **MPI_Send**: Basic blocking send operation
- Routine returns only after the application buffer in the sending task is free for reuse.

```
MPI_Send (&buf,count,datatype,dest,tag,comm)
MPI_SEND (buf,count,datatype,dest,tag,comm,ierr)
```

- **MPI_Recv**: Receive a message
- will block until the requested data is available in the application buffer in the receiving task.

```
MPI_Recv (&buf,count,datatype,source,tag,comm,&status)
MPI_RECV (buf,count,datatype,source,tag,comm,status,ierr)
```

Point-to-point Communication Functions

Non-blocking send / receive

- **MPI_Isend**: Identifies an area in memory to serve as a send buffer.
- Processing continues immediately without waiting for the message to be copied out from the application buffer

```
MPI_Isend (&buf,count,datatype,dest,tag,comm,&request)
MPI_ISEND (buf,count,datatype,dest,tag,comm,request,ierr)
```

- **MPI_Irecv**: Identifies an area in memory to serve as a receive buffer
- Processing continues immediately without actually waiting for the message to be received and copied into the the application buffer

```
MPI_Irecv (&buf,count,datatype,source,tag,comm,&request)
MPI_IRECV (buf,count,datatype,source,tag,comm,request,ierr)
```

- **MPI_WAIT** and **MPI_TEST**: Functions required by nonblocking send and receive use to determine when the non-blocking receive operation completes and the requested message is available in the application buffer.

Point-to-Point Communication Function Arguments

- `buf`: address space that references the data that is to be sent or received

  In most cases, variable name that is to be sent/received

  C programs: this argument is passed by reference and usually must be prepended with an ampersand: &var1
- `count`: number of data elements of a particular type to be sent
- `datatype`: MPI predefines its elementary data types
- `dest`: indicates the process where a message should be delivered

  specified as the rank of the receiving process
- `source`: indicates the originating process of the message.

  specified as the rank of the sending process
- `tag`: arbitrary non-negative integer assigned by the programmer to uniquely identify a message

  send and receive operations should match message tags

Point-to-Point Communication Function Arguments

for a receive operation, the wild card `MPI_ANY_TAG` can be used to receive any message regardless of its tag.

- `comm`: indicates the communication context, or set of processes for which the source or destination fields are valid

  unless the programmer is explicitly creating new communicators, the predefined communicator `MPI_COMM_WORLD` is usually used

- `status`: for a receive operation, indicates the source of the message and the tag of the message.

  C: pointer to a predefined structure `MPI_Status`

  Fortran: integer array of size `MPI_STATUS_SIZE`

- `request`: used by non-blocking send and receive operations

  since non-blocking operations may return before the requested system buffer space is obtained, the system issues a unique request number

  programmer uses this system assigned handle later determine completion of the non-blocking operation

Point-to-Point Communication Function Arguments

C: a pointer to a predefined structure `MPI_Request`.

Fortran: an integer

Blocking Message Passing Funtions

- **MPI_Send**: Basic blocking send operation
- **MPI_Recv**: Receive a message and block until the requested data is available in the application buffer in the receiving task.
- **MPI_Ssend**: Synchronous blocking send: Send a message and block until the application buffer in the sending task is free for reuse and the destination process has started to receive the message
- **MPI_Sendrecv**: Send a message and post a receive before blocking. Will block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message
- **MPI_Wait[any,all,some]**: MPI_Wait blocks until a specified non-blocking send or receive operation has completed. For multiple non-blocking operations, the programmer can specify any, all or some completions.

Blocking Message Passing Funtions

- **MPI_Probe**: Performs a blocking test for a message. The wildcards MPI_ANY_SOURCE and MPI_ANY_TAG may be used to test for a message from any source or with any tag. For the C routine, the actual source and tag will be returned in the status structure as status.MPI_SOURCE and status. MPI_TAG. For the Fortran routine, they will be returned in the integer array status(MPI_SOURCE) and status(MPI_TAG).
- **MPI_Get_Count**: Returns the source, tag and number of elements of datatype received. Can be used with both blocking and non-blocking receive operations. For the C routine, the actual source and tag will be returned in the status structure as status.MPI_SOURCE and status. MPI_TAG. For the Fortran routine, they will be returned in the integer array status(MPI_SOURCE) and status(MPI_TAG).

Blocking Message Passing Funtions

```
MPI_Ssend (&buf,count,datatype,dest,tag,comm)
MPI_Sendrecv (&sendbuf,sendcount,sendtype,dest,sendtag,
  &recvbuf,recvcount,recvtype,source,recvtag,
  comm,&status)
MPI_Wait (&request,&status)
MPI_Waitany (count,&array_of_requests,&index,&status)
MPI_Waitall (count,&array_of_requests,&array_of_statuses)
MPI_Waitsome (incount,&array_of_requests,&outcount,
  &array_of_offsets, &array_of_statuses)
MPI_Probe (source,tag,comm,&status)
MPI_Get_count (&status,datatype,&count)
```

```
MPI_SSEND (buf,count,datatype,dest,tag,comm,ierr)
MPI_SENDRECV (sendbuf,sendcount,sendtype,dest,sendtag,&
   recvbuf,recvcount,recvtype,source,recvtag,&
   comm,status,ierr)
MPI_WAIT (request,status,ierr)
MPI_WAITANY (count,array_of_requests,index,status,ierr)
MPI_WAITALL (count,array_of_requests,array_of_statuses,ierr)
MPI_WAITSOME (incount,array_of_requests,outcount,&
   array_of_offsets, array_of_statuses,ierr)
MPI_PROBE (source,tag,comm,status,ierr)
MPI_GET_COUNT (status,datatype,count,ierr)
```

Exercise: Ping Pong

- Modify the `pingpong.c` or `pingpong.f90` example to do a blocking send and recieve.
- Task 0 send a ping to task 1 and awaits return ping
- This example only requires two MPI processes
- What happens if you run on more than 2 cpus

Exercise: Ping Pong

```
[alp514.sol](1132): mpicc —o pingpongc pingpong.c
[alp514.sol](1133): mpif90 —o pingpongf pingpong.f90
[alp514.sol](1134): srun —n 2 —p hawkgpu —t 5 ./pingpongc
Task 1: Received 1 char(s) from task 0 with tag 1
Task 0: Received 1 char(s) from task 1 with tag 1
[alp514.sol](1135): srun —n 2 —p hawkgpu —t 5 ./pingpongf
Task   1 : Received  1  char(s) from task  0 with tag  1
Task   0 : Received  1  char(s) from task  1 with tag  1
[2021—07—12 18:03.21] ~/Workshop/sum2017/src/mpi
[alp514.sol](1136): srun —n 4 —p hawkgpu —t 5 ./pingpongf
Task   1 : Received  1  char(s) from task  0 with tag  1
Task   3 : Received  0  char(s) from task  0 with tag  0
Task   2 : Received  0  char(s) from task  0 with tag  0
Task   0 : Received  1  char(s) from task  1 with tag  1
[alp514.sol](1137): srun —n 4 —p hawkgpu —t 5 ./pingpongc
Task 0: Received 1 char(s) from task 1 with tag 1
Task 1: Received 1 char(s) from task 0 with tag 1
Task 2: Received —32766 char(s) from task 2496 with tag —1075053569
Task 3: Received —32766 char(s) from task 1668810496 with tag 32588
```

Non Blocking Message Passing Functions

- **MPI_ISend**: Identifies an area in memory to serve as a send buffer. Processing continues immediately without waiting for the message to be copied out from the application buffer. A communication request handle is returned for handling the pending message status. The program should not modify the application buffer until subsequent calls to MPI_Wait or MPI_Test indicate that the non-blocking send has completed.
- **MPI_Irecv**: Identifies an area in memory to serve as a receive buffer. Processing continues immediately without actually waiting for the message to be received and copied into the the application buffer. A communication request handle is returned for handling the pending message status. The program must use calls to MPI_Wait or MPI_Test to determine when the non-blocking receive operation completes and the requested message is available in the application buffer.

Non Blocking Message Passing Functions

- **MPI_Issend**: Non-blocking synchronous send. Similar to MPI_Isend(), except MPI_Wait() or MPI_Test() indicates when the destination process has received the message.
- **MPI_Test[any,all,some]**: MPI_Test checks the status of a specified non-blocking send or receive operation. The flag parameter is returned logical true (1) if the operation has completed, and logical false (0) if not. For multiple non-blocking operations, the programmer can specify any, all or some completions.
- **MPI_Iprobe**: Performs a non-blocking test for a message. The wildcards MPI_ANY_SOURCE and MPI_ANY_TAG may be used to test for a message from any source or with any tag. The integer flag parameter is returned logical true (1) if a message has arrived, and logical false (0) if not. For the C routine, the actual source and tag will be returned in the status structure as status.MPI_SOURCE and status.MPI_TAG. For the Fortran routine, they will be returned in the integer array status(MPI_SOURCE) and status(MPI_TAG).

Non Blocking Message Passing Functions

```
MPI_Issend (&buf,count,datatype,dest,tag,comm,&request)
MPI_Test (&request,&flag,&status)
MPI_Testany (count,&array_of_requests,&index,&flag,&status)
MPI_Testall (count,&array_of_requests,&flag,&array_of_statuses)
MPI_Testsome (incount,&array_of_requests,&outcount,
  &array_of_offsets, &array_of_statuses)
MPI_Iprobe (source,tag,comm,&flag,&status)
```

```
MPI_ISSEND (buf,count,datatype,dest,tag,comm,request,ierr)
MPI_TEST (request,flag,status,ierr)
MPI_TESTANY (count,array_of_requests,index,flag,status,ierr)
MPI_TESTALL (count,array_of_requests,flag,array_of_statuses,ierr)
MPI_TESTSOME (incount,array_of_requests,outcount, &
   array_of_offsets, array_of_statuses,ierr)
MPI_IPROBE (source,tag,comm,flag,status,ierr)
```

Exercise: Ring

- Modify the `ring.c` or `ring.f90` example to do a non blocking send and recieve.
- Each process sends 1 to the left and 2 to the right

Exercise: Ring

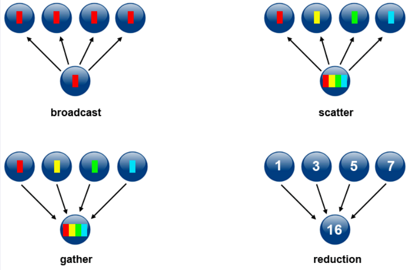
```
[alp514.sol](1160): mpicc —o ringc ring.c
[alp514.sol](1161): mpif90 —o ringf ring.f90
[alp514.sol](1162): srun —n 4 —p hawkgpu —t 10 ./ringc
Task 0: Received from task 3 with tag 1 and from task 1 with tag 2
Task 0: Send to task 3 with tag 2 and to task 1 with tag 1
Task 3: Received from task 2 with tag 1 and from task 0 with tag 2
Task 3: Send to task 2 with tag 2 and to task 0 with tag 1
Task 2: Received from task 1 with tag 1 and from task 3 with tag 2
Task 2: Send to task 1 with tag 2 and to task 3 with tag 1
Task 1: Received from task 0 with tag 1 and from task 2 with tag 2
Task 1: Send to task 0 with tag 2 and to task 2 with tag 1
[alp514.sol](1163): srun —n 4 —p hawkgpu —t 10 ./ringf
Task  0: Received from task 3 with tag 1 and from task 1 with tag 2
Task  0: Send to task 3 with tag 2 and to task 1 with tag 1
Task  2: Received from task 1 with tag 1 and from task 3 with tag 2
Task  2: Send to task 1 with tag 2 and to task 3 with tag 1
Task  3: Received from task 2 with tag 1 and from task 0 with tag 2
Task  3: Send to task 2 with tag 2 and to task 0 with tag 1
Task  1: Received from task 0 with tag 1 and from task 2 with tag 2
Task  1: Send to task 0 with tag 2 and to task 2 with tag 1
```

Collective Communications

- Collective communication routines must involve all processes within the scope of a communicator.
- All processes are by default, members in the communicator `MPI_COMM_WORLD`.
- Unexpected behavior, including program failure, can occur if even one task in the communicator doesnt participate.
- It is the programmers responsibility to ensure that all processes within a communicator participate in any collective operations.
- Collective communication routines do not take message tag arguments.

Types of Collective Operations

- **Synchronization**: processes wait until all members of the group have reached the synchronization point.
- **Data Movement**: broadcast, scatter/gather, all to all.
- **Collective Computation (reductions)**: one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.



Collective Communication Functions

- **MPI_Barrier**: Creates a barrier synchronization in a group
- **MPI_Bcast**: Broadcasts (sends) a message from the process with rank root to all other processes in the group
- **MPI_Scatter**: Distributes distinct messages from a single source task to each task in the group
- **MPI_Gather**: Gathers distinct messages from each task in the group to a single destination task
- **MPI_Allgather**: Concatenation of data to all tasks in a group. Each task in the group, in effect, performs a one-to-all broadcasting operation within the group
- **MPI_Reduce**: Applies a reduction operation on all tasks in the group and places the result in one task
- **MPI_Allreduce**: equivalent to an `MPI_Reduce` followed by an `MPI_Bcast`

Collective Communication Functions

- **MPI_Reduce_scatter** equivalent to an `MPI_Reduce` followed by an `MPI_Scatter operation`
- **MPI_Alltoall**: Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index
- **MPI_Scan**: Performs a scan operation with respect to a reduction operation across a task group

Collective Communication Functions

```
MPI_Bcast (&buffer,count,datatype,root,comm)
MPI_Scatter (&sendbuf,sendcnt,sendtype,&recvbuf,recvcnt,recvtype,root,comm)
MPI_Gather (&sendbuf,sendcnt,sendtype,&recvbuf,recvcount,recvtype,root,comm)
MPI_Allgather (&sendbuf,sendcount,sendtype,&recvbuf,recvcount,recvtype,comm)
MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)
MPI_Allreduce (&sendbuf,&recvbuf,count,datatype,op,comm)
MPI_Reduce_scatter (&sendbuf,&recvbuf,recvcount,datatype,op,comm)
MPI_Alltoall (&sendbuf,sendcount,sendtype,&recvbuf,recvcnt,recvtype,comm)
MPI_Scan (&sendbuf,&recvbuf,count,datatype,op,comm)
```

```
MPI_BCAST (buffer,count,datatype,root,comm,ierr)
MPI_SCATTER (sendbuf,sendcnt,sendtype,recvbuf,recvcnt,recvtype,root,comm,ierr)
     MPI_GATHER (sendbuf,sendcnt,sendtype,recvbuf,recvcount,recvtype,root,comm,
     ierr)
MPI_ALLGATHER (sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,comm,info)
MPI_REDUCE (sendbuf,recvbuf,count,datatype,op,root,comm,ierr)
MPI_ALLREDUCE (sendbuf,recvbuf,count,datatype,op,comm,ierr)
MPI_REDUCE_SCATTER (sendbuf,recvbuf,recvcount,datatype,op,comm,ierr)
MPI_ALLTOALL (sendbuf,sendcount,sendtype,recvbuf,recvcnt,recvtype,comm,ierr)
MPI_SCAN (sendbuf,recvbuf,count,datatype,op,comm,ierr)
```

Exercise: Calculation of Pi

- Parallelize the `pi_mpi.f90` or `pi_mpi.c` making use of collective communication functions?

Further Reading

- Tutorials
  1. MPI: `https://computing.llnl.gov/tutorials/mpi/`
  2. Advanced MPI: `https://hpc.llnl.gov/sites/default/files/DavidCronkSlides.pdf`
  3. CITutor: `https://www.citutor.org/`
  4. XSEDE HPC Monthly Workshop Series: `https://psc.edu/xsede-hpc-series-all-workshops`
  5. MPI Tutorial: `http://mpitutorial.com/`
- Books
  1. Beginning MPI (An Introduction in C) by Wesley Kendall
  2. Parallel Programming with MPI by Peter Pacheco (No relation)
  3. Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering Computation) by William Gropp
  4. Parallel Programming in C with MPI and Openmp by Michael J. Quinn
  5. MPI: The Complete Reference by Marc Snir *et. al.*