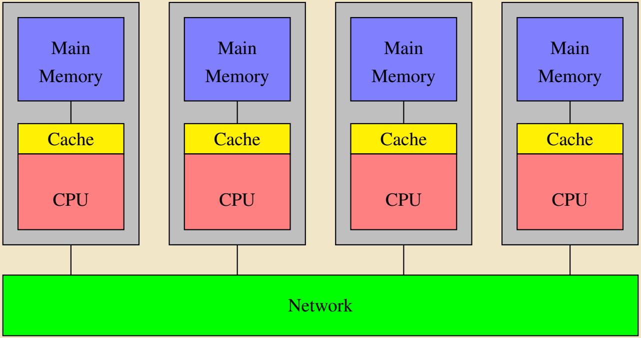
Introduction to MPI

2017 HPC Workshop: Parallel Programming

Alexander B. Pacheco
LTS Research Computing
May 31 - June 1, 2017

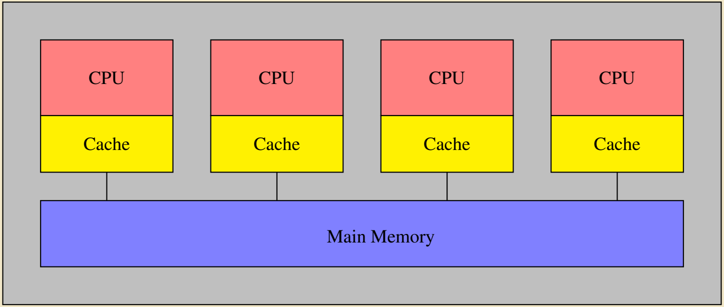
Distributed Memory Model

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



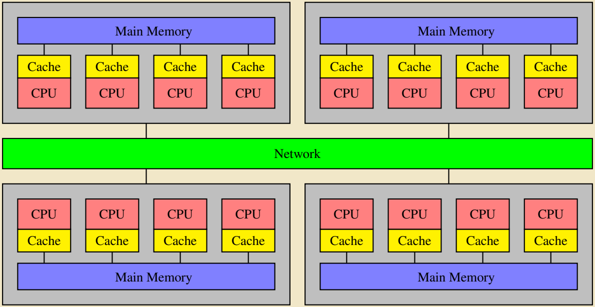
Shared Memory Model

- 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?

- There are already network communication libraries
- Optimized for performance
- Take advantage of faster network transport
  - Shared memory (within a node)
  - Faster cluster interconnects (e.g. InfiniBand)
  - TCP/IP if all else fails
- Enforces certain guarantees
  - Reliable messages
  - In-order message arrival
- Designed for multi-node technical computing

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

Various MPI Implementations

- OpenMPI: open source, portability and simple installation and config
- MPICH: open source, portable
- MVAPICH2: MPICH derivative - InfiniBand, iWARP and other RDMA-enabled interconnects (GPUs)
  - MPI implementation on Sol
- Intel MPI (IMPI): vendor-supported MPICH from Intel

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.
- On Sol, we have build MVAPICH2 version 2.1 using GCC 5.3 and 6.1, Intel 2016 and PGI 2016, and version 2.2 using GCC 7.1 and Intel 2017
- 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 mvapich2/2.2/intel-17.0.3
[alp514.sol](794): mpicc -show
icc -fPIC -I/share/Apps/mvapich2/2.2/intel-17.0.3/include -L/share/Apps/mvapich2/2.2/intel-17.0.3/lib -Wl,-rpath -
      Wl,/share/Apps/mvapich2/2.2/intel-17.0.3/lib -Wl,--enable-new-dtags -lmpi
[alp514.sol](795): mpicxx -show
icpc -fPIC -I/share/Apps/mvapich2/2.2/intel-17.0.3/include -L/share/Apps/mvapich2/2.2/intel-17.0.3/lib -lmpicxx -
      Wl,-rpath -Wl,/share/Apps/mvapich2/2.2/intel-17.0.3/lib -Wl,--enable-new-dtags -lmpi
[alp514.sol](796): mpif90 -show
ifort -fPIC -I/share/Apps/mvapich2/2.2/intel-17.0.3/include -I/share/Apps/mvapich2/2.2/intel-17.0.3/include -L/
      share/Apps/mvapich2/2.2/intel-17.0.3/lib -lmpifort -Wl,-rpath -Wl,/share/Apps/mvapich2/2.2/intel-17.0.3/lib
      -Wl,--enable-new-dtags -lmpi
```

Running MPI Applications I

- 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

- 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 24myapp`

MPI Program Outline

1. Initiate communication between processes

   MPI_INIT: initialize MPI environment
   MPI_COMM_SIZE : return total number of MPI processes
   MPI_COMM_RANK : return rank of calling process

2. Communicate data between processes

   MPI_SEND : send a message
   MPI_RECV : receive a message

3. Terminate the MPI environment using **MPI_FINALIZE**

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](1003): module load mvapich2/2.2/intel-17.0.3
[alp514.sol](1004): mpicc -o helloc hello.c
[alp514.sol](1005): mpif90 -o hellof hello.f90
[alp514.sol](1006): srun -p eng -n 4 ./helloc
Number of tasks= 4 My rank= 3 Running on sol-b110
Number of tasks= 4 My rank= 2 Running on sol-b110
Number of tasks= 4 My rank= 1 Running on sol-b110
Number of tasks= 4 My rank= 0 Running on sol-b110
[alp514.sol](1007): srun -p eng -n 4 ./hellof
Number of tasks= 4 My rank= 3 Running on sol-b110
Number of tasks= 4 My rank= 2 Running on sol-b110
Number of tasks= 4 My rank= 0 Running on sol-b110
Number of tasks= 4 My rank= 1 Running on sol-b110
```

MPI Program Structure

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

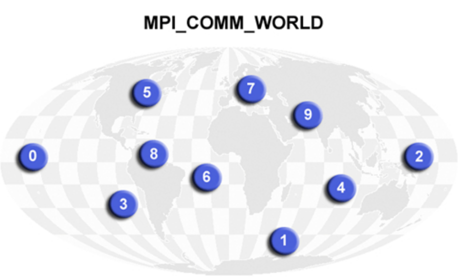
C	Fortran
`#include "mpi.h"`	`include 'mpif.h'`

- 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)` `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

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)`

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 split out errors about missing libraries

MPI Functions I

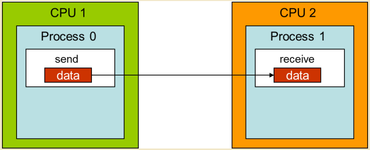
- Environment management functions
  1. MPI_INIT
  2. MPI_COMM_SIZE
  3. MPI_COMM_RANK
  4. MPI_ABORT: Terminates all MPI processes
  5. MPI_GET_PROCESSOR_NAME: Returns the processor name.
  6. MPI_GET_VERSION: Returns the version and subversion of the MPI standard
  7. MPI_INITIALIZED: Indicates whether MPI_Init has been called
  8. MPI_WTIME: Returns an elapsed wall clock time in seconds
  9. MPI_WTICK: Returns the resolution in seconds of MPI_WTIME
  10. MPI_FINALIZE

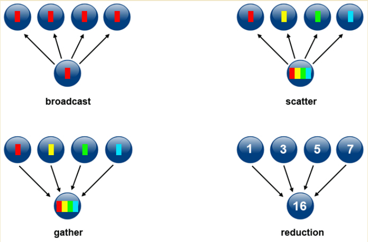
```
MPI_Init (&argc,&argv)
MPI_Comm_size (comm,&size)
MPI_Comm_rank (comm,&rank)
MPI_Abort (comm,errorcode)
MPI_Get_processor_name (&name,&resultlength)
MPI_Get_version (&version,&subversion)
MPI_Initialized (&flag)
MPI_Wtime ()
MPI_Wtick ()
MPI_Finalize ()
```

```
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)
```

MPI Functions II

- 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 I

- 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

Point-to-point Communication II

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.
- 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.

Point-to-point Communication III

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 IV

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 V

Synchronous send

- **MPI_Ssend**: Send a message
- will block until the application buffer in the sending task is free for reuse and the destination process has started to receive the message.

  ```
  MPI_Ssend (&buf,count,datatype,dest,tag,comm)
  MPI_SSEND (buf,count,datatype,dest,tag,comm,ierr)
  ```
- **MPI_Issend**: Non-blocking synchronous send

  ```
  MPI_Issend (&buf,count,datatype,dest,tag,comm,&request)
  MPI_ISSEND (buf,count,datatype,dest,tag,comm,request,ierr)
  ```

Blocking Message Passing Example I

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

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

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

  // task 0 sends to task 1 and waits to receive a return message
  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);
  }

  // task 1 waits for task 0 message then returns a message
  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);
  }

  // query recieve Stat variable and print message details
  MPI_Get_count(&Stat, MPI_CHAR, &count);
  printf("Task %d: Received %d char(s) from task %d with tag %d \n",
   rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);

  MPI_Finalize();
}
```

```fortran
program ping
  include 'mpif.h'

  integer :: numtasks, rank, dest, source, count, tag, ierr
  integer :: stat(MPI_STATUS_SIZE)   ! required variable for receive
          routines
  character :: inmsg, outmsg
  outmsg = 'x'
  tag = 1

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

  ! task 0 sends to task 1 and waits to receive a return message
  if (rank .eq. 0) then
     dest = 1
     source = 1
     call MPI_SEND(outmsg, 1, MPI_CHARACTER, dest, tag, MPI_COMM_WORLD,
              ierr)
     call MPI_RECV(inmsg, 1, MPI_CHARACTER, source, tag, MPI_COMM_WORLD,
              stat, ierr)

     ! task 1 waits for task 0 message then returns a message
  else if (rank .eq. 1) then
     dest = 0
     source = 0
     call MPI_RECV(inmsg, 1, MPI_CHARACTER, source, tag, MPI_COMM_WORLD,
              stat, err)
     call MPI_SEND(outmsg, 1, MPI_CHARACTER, dest, tag, MPI_COMM_WORLD,
              err)
  endif

  ! query recieve Stat variable and print message details
  call MPI_GET_COUNT(stat, MPI_CHARACTER, count, ierr)
  print *, 'Task ',rank,': Received', count, 'char(s) from task', &
       stat(MPI_SOURCE), 'with tag',stat(MPI_TAG)

  call MPI_FINALIZE(ierr)

end program ping
```

Blocking Message Passing Example II

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

main(int argc, char *argv[])  {
  int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
  MPI_Request reqs[4];   // required variable for non-blocking calls
  MPI_Status stats[4];   // required variable for Waitall routine

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

  // determine left and right neighbors
  prev = rank-1;
  next = rank+1;
  if (rank == 0)  prev = numtasks - 1;
  if (rank == (numtasks - 1))  next = 0;

  // post non-blocking receives and sends for neighbors
  MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
  MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

  MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
  MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

  // wait for all non-blocking operations to complete
  MPI_Waitall(4, reqs, stats);

  printf("Task %d: Received from task %d with tag %d and from task %d
          with tag %d\n",
   rank, prev, tag1, next, tag2);
  printf("Task %d: Send to task %d with tag %d and to task %d with tag %d
          \n",
   rank, prev, tag2, next, tag1);

  MPI_Finalize();
}
```

```fortran
program ringtopo
  include 'mpif.h'

  integer numtasks, rank, next, prev, buf(2), tag1, tag2, ierr, count
  integer reqs(4)   ! required variable for non-blocking calls
  integer stats(MPI_STATUS_SIZE,4)   ! required variable for WAITALL
          routine
  tag1 = 1
  tag2 = 2

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

  ! determine left and right neighbors
  prev = rank - 1
  next = rank + 1
  if (rank .eq. 0) then
     prev = numtasks - 1
  endif
  if (rank .eq. numtasks - 1) then
     next = 0
  endif

  ! post non-blocking receives and sends for neighbors
  ! Receive 1 from left and 2 from right
  call MPI_IRECV(buf(1), 1, MPI_INTEGER, prev, tag1, MPI_COMM_WORLD, reqs
          (1), ierr)
  call MPI_IRECV(buf(2), 1, MPI_INTEGER, next, tag2, MPI_COMM_WORLD, reqs
          (2), ierr)
  ! Send 1 to right and 2 to legft
  call MPI_ISEND(rank, 1, MPI_INTEGER, prev, tag2, MPI_COMM_WORLD, reqs(3
          ), ierr)
  call MPI_ISEND(rank, 1, MPI_INTEGER, next, tag1, MPI_COMM_WORLD, reqs(4
          ), ierr)

  ! wait for all non-blocking operations to complete
  call MPI_WAITALL(4, reqs, stats, ierr);

  print '(5(a,i2))',  'Task ',rank,': Received from task', prev, ' with
          tag',tag1, &
      ' and from task', next, ' with tag',tag2
  print '(5(a,i2))',  'Task ',rank,': Send to task', prev, ' with tag',
          tag2, &
      ' and to task', next, ' with tag',tag1

  ! continue - do more work

  call MPI_FINALIZE(ierr)

end program ringtopo
```

Blocking Message Passing Example III

```
[alp514.sol](1110): mpicc -o ringc ring.c
[alp514.sol](1113): srun -p eng -n 4 ./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 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
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
```

```
[alp514.sol](1111): mpif90 -o ringf ring.f90
[alp514.sol](1114): srun -p eng -n 4 ./ringf
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  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  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
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
```