Parallel Computing for Science & Engineering Spring 2018: MPI introduction

Instructors:
Lars Koesterke, TACC
Charlie Dey, TACC

Outline

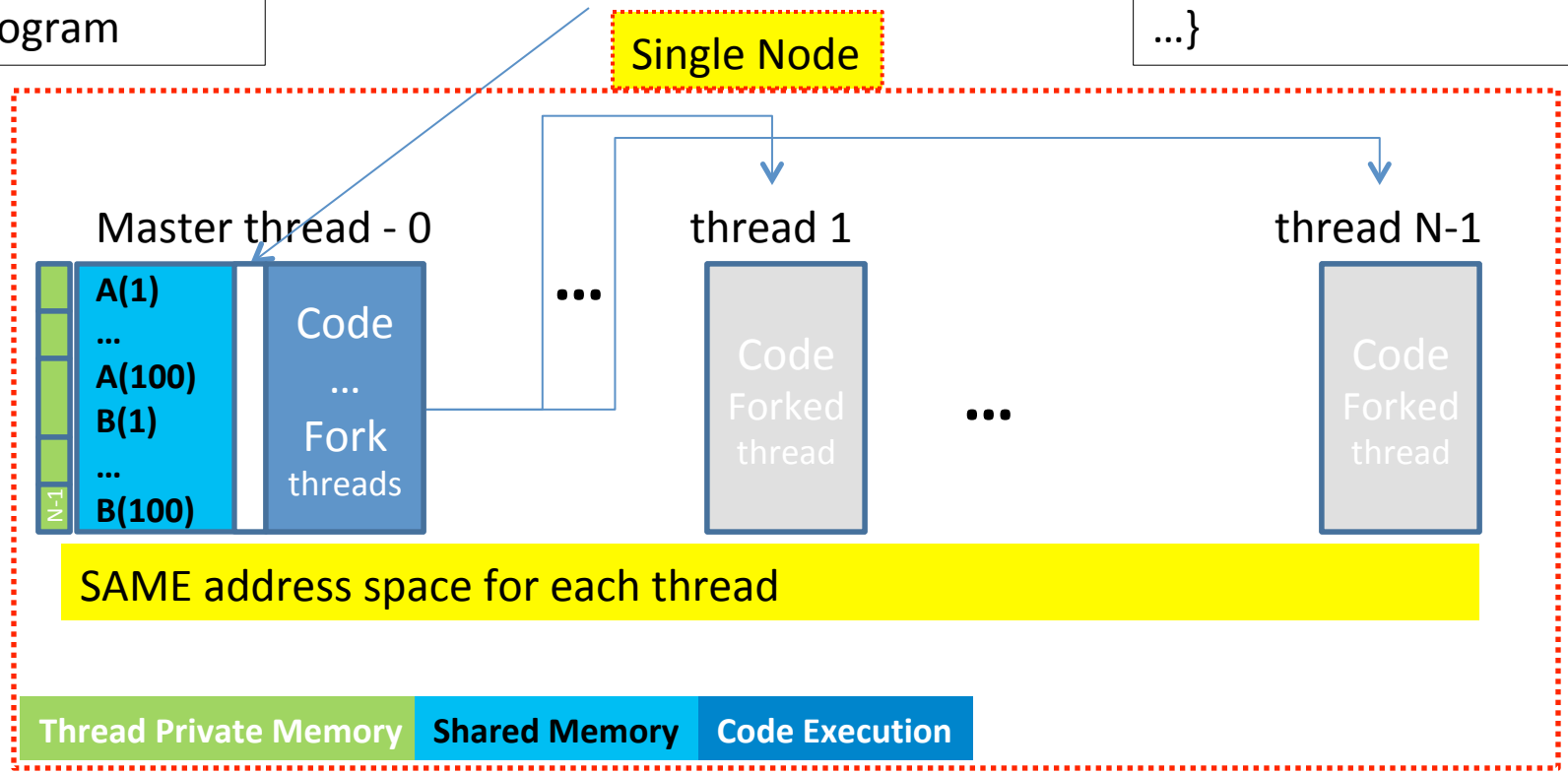
- Executing OpenMP and MPI
- Paradigm/Key Concepts/Advantages
- MPI History version 1 and 2, implementations
- Compiling, Running
- MPI Initialize, Finalize and task-id/task-count
- MPI Communicators

OpenMP (shared memory)

```
program  myomp
real*8  a(100), b(100)
…
!$omp parallel
… end program
```

Compile→ a.out
Set OMP_NUM_THREADS to N
./a.out

```
int main () {
double a(100), b(100);
  …
#pragma omp parallel
…}
```

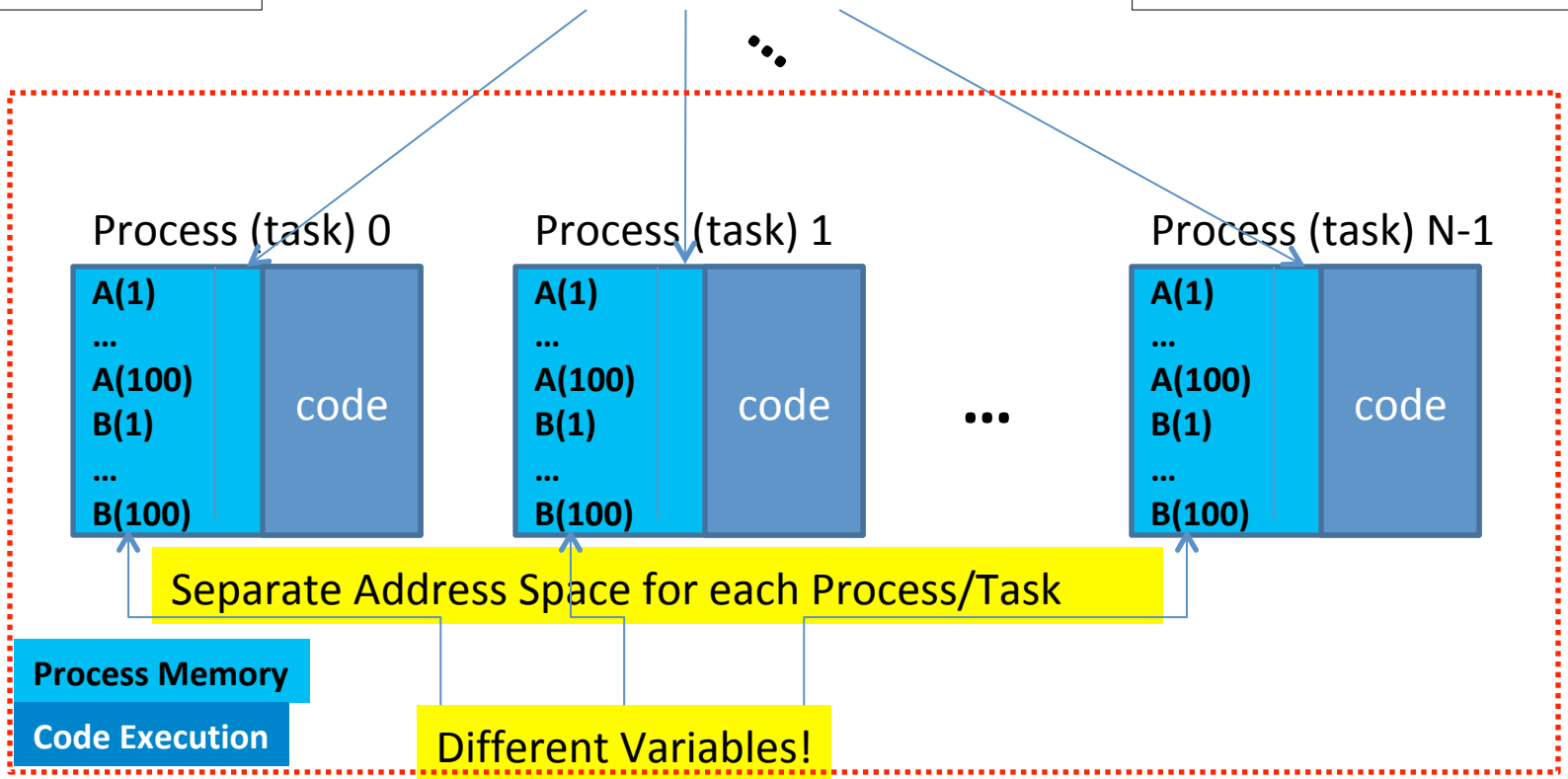


MPI (distributed memory)

```
program  mympi
real*8  a(100), b(100)
…
end program
```

Compile→ a.out
Launch on N cores
ibrun ./a.out

```
int main () {
double a(100), b(100);
  …
}
```



MPI Parallel Code

- The programmer is responsible for:
  - Executing independent work (just like OpenMP)
  - Partitioning (independent) Data for use on each task.
  - Moving Data between tasks

Message Passing Paradigm

- A Parallel MPI Program is launched as separate processes (tasks), each with their own address space.
  - Requires partitioning data across tasks.
- Data is explicitly moved from task to task
  - A task accesses the data of another task through a transaction called "message passing" in which a copy of the data (message) is transferred (passed) from one task to another.
- There are two classes of message passing (transfers)
  - Point-to-Point messages involve only two tasks
  - Collective messages involve a set of tasks
- Access to subsets of complex data structures is simplified
  - A data subset is described as a single Data Type entity
- Transfers use synchronous or asynchronous protocols
- Messaging distribution be arranged into efficient topologies

MPI Key Concepts-- Summary

- Used to create parallel SPMD programs on distributed-memory machines with explicit message passing
- Routines available for
  - Point-to-Point Communication
  - Collective Communication
    - 1-to-many
    - many-to-1
    - many-to-many
  - Synchronization (barriers, non-blocking MP)
  - Data Types
  - Parallel IO
  - Topologies

Advantages of Message Passing

- Universality
  - Message passing model works on separate processors connected by any network (and even on shared memory systems)
  - matches the hardware of most of today's parallel supercomputers as well as ad hoc networks of computers
- Performance/Scalability
  - Scalability (memory & compute) is the most compelling reason why message passing will remain a permanent component of HPC (High Performance Computing)
  - As modern systems increase core counts, management of the memory hierarchy (including distributed memory) is the key to extracting the highest performance
  - Each message passing task only directly uses its "process data", avoiding complexities of process-shared data, and allowing compilers and cache management hardware to function without contention.

MPI-1

- MPI-1 - Message Passing Interface (v. 1.2)
  - Library
  - Specification: defined by committee of vendors, implementers, and parallel programmers
  - Designed with SPMD (single program, multiple data) technique in mind.
- Available on almost all parallel machines in C/C++ and Fortran
- About 125 routines
  - 6 basic routines
  - the rest are extensions that can simplify algorithm implementation and optimize performance

MPI-1

Web

www-unix.mcs.anl.gov/mpi/
www.mcs.anl.gov/research/projects/mpich2/
www.mpi-forum.org/

Books

Using MPI, by Gropp, Lusk, and Skjellum
MPI Annotated Reference Manual, by Marc Snir, *et al*
Parallel Programming with MPI, by Peter Pacheco
Using MPI-2, by Gropp, Lusk and Thakur

Getting Started

www.mcs.anl.gov/research/projects/mpi/tutorial/gropp/talk.html
http://ci-tutor.ncsa.illinois.edu/
www.nersc.gov/nusers/help/tutorials/mpi/intro/ (simple, direct)
https://computing.llnl.gov/?set=training&page=index
Advanced: www.mcs.anl.gov/research/projects/mpi/tutorial/

Standard

www.mpi-forum.org/docs/

MPI Implementations

- MPICH is from Argonne Natl. Lab. It is the basis for IBM, Intel, Cray, Microsoft, Myricom and OSU MVAPICH version, etc. (The CH designation refers to the Chameleon library developed by W. Gropp.)
  - Hardware vendors: IBM, Oracle*, Cray, HP, SGI, Intel
  - Interconnect vendors: Myricomm, Quadrics, Mellanox/QLogic* , Intel **
    * InfiniBand: open source drivers/university MPI collaboration
      https://www.openfabrics.org
      http://mvapich.cse.ohio-state.edu/ MVAPICH
    **
      http://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-architecture-fabric-overview.html
  - Software vendors: MicroSoft, MPI/Pro, Platform MPI (was Scali MPI), etc.
  - OpenSource: OpenMPI (formerly LAM/MPI) http://www.open-mpi.org/

MPI-2

- Includes features left out of MPI-1
  - One-sided communications
  - Dynamic process control
  - More complicated collectives
  - MPI-IO
- Implementations
  - not quickly undertaken after the standard document was released (in 1997)
  - Now MPICH (and its descendants), OpenMPI, and the vendor implementations are MPI-2 compliant, and are including MPI-3 features.

Compiling MPI Programs

- Generally an MPI compiler wrapper is used.
  - not defined by the standard
  - consult your implementation
  - what it handles: MPI "External compiler thingies"; correct include path, library path, and libraries
- MPICH-style (the most common)
  - C

```
mpicc  -o mycexe mycode.c
```

  - Fortran

```
mpif90 -o myfexe mycode.f
```

Running MPI Programs

- MPI programs require some help to get started
  - what computers (compute nodes) should I run on?
  - how do I access them?
- MPICH-style

```
mpirun -np 10 -machinefile mach ./a.out
```

- When batch systems are involved, all bets are off

  @TACC Lonestar/Stampede, … (via a job script)

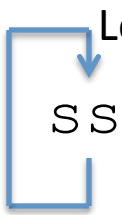
```
ibrun tacc_affinity ./a.out
```

  - SLURM batch utility handles the rest

tacc_affinity is not required

What is MPI Parallel Code

- An executable with MPI library calls – executed multiple times as independent processes (tasks) launched by ssh commands:




```
ssh <nodename> <environment> executable
```

  - Tasks need to organization/synchronize (initialize).
  - Tasks need to know their task id and total tasks.
  - Tasks need to clean up at end.

Minimal MPI program

- Every MPI program needs these…
  - C version

```
#include <mpi.h>
...
ierr=MPI_Init(&argc, &argv);
ierr=MPI_Comm_size(MPI_COMM_WORLD,&npes);
ierr=MPI_Comm_rank(MPI_COMM_WORLD,&iam);
...
ierr=MPI_Finalize();
```

Include params, etc.

Fire up MPI runtime

^ Get rank & # of tasks

Finish MPI messaging.

In C MPI routines are functions which return the error value

Minimal MPI program

- Every MPI program needs these…
  - Fortran version

```
 include 'mpif.h'          or     use mpi
...
call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD, npes,ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, iam, ierr)
...
call MPI_Finalize(ierr)
```

In Fortran, MPI routines are subroutines with the last parameter as the error value

Include files

- The MPI include file
  - C:

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

  - Fortran

```
#include "mpif.h"
use MPI                    ! Best to use "use"
```

- Defines constants required by MPI routines
  - In C: defines the interfaces for the functions, macros
  - In C++: the interfaces are different, so be careful
  - In F90, module defines interface for subroutines, parameters
- Compilers know where to find the include files
  - regular compilers are usually called through mpif90/mpicc wrapper scripts

MPI Initialization & Termination

- All processes must initialize and finalize MPI (each is a `collective call*`).
  - `MPI_Init` : starts up the MPI runtime environment
  - `MPI_Finalize` : shuts down the MPI runtime environment
- Must include header files – provides basic MPI definitions and types.
  - Header File

Fortran 77	Fortran 90	C/C++
include 'mpif.h'	use mpi	#include <mpi.h>

  - Format of MPI calls

Fortran 77/90 binding (upper or lower case)	C/C++ binding
CALL MPI_XYYY(parameters…, ierr)	ierr = MPI_Xyyy(parameters…)

`* Means the entire group of tasks must execute this call.`

Run Parameters

- `MPI_Comm_size` : gets the **number of processes** (*NP*) in a run
  Integer
  (typically called just after `MPI_Init`).
- `MPI_Comm_rank` : gets the process ID (rank) of the current process,
  integer between 0 and *NP*-1 inclusive
  (typically called just after `MPI_Init`).

Communications

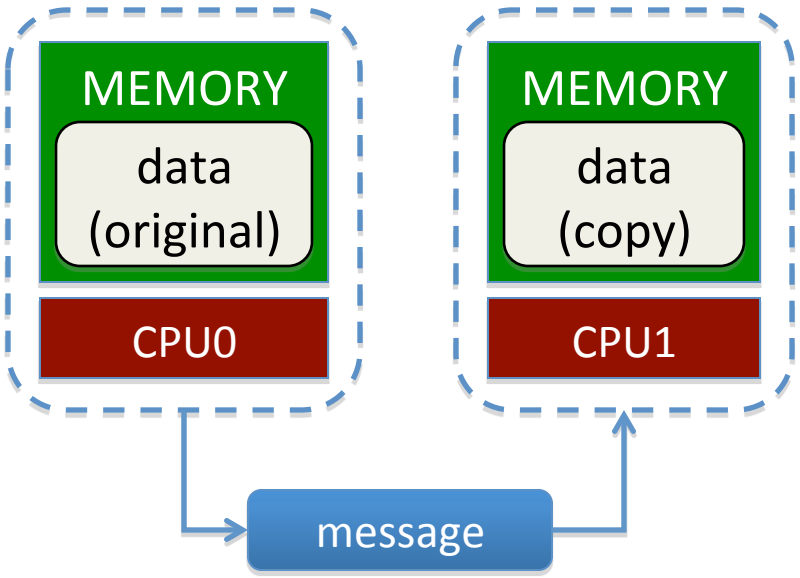
- Nee an API=library routine
- Need a “channel” to communicate on
- Need to indicate what to send/rec
  - How many “elements”, and
  - size of an element
- Need a destination/source
- Need to know if successful.

Channels of Communication

- Communicator (think of as a channel to communicate)
  - MPI uses a communicator objects (and groups) to identify a set of processes which communicate only within their set.
  - MPI_COMM_WORLD is defined in the MPI include file as all processes (ranks) of your initial launch
  - Required parameter for most MPI calls
  - You can create subsets of MPI_COMM_WORLD
- Rank
  - Unique *process ID* within a communicator
  - Assigned by launcher and available after initialization
  - Processors within a communicator are assigned numbers 0 to np-1 (C/F90)
  - An identifier: Used to specify sources and destinations of messages, process specific indexing and operations.

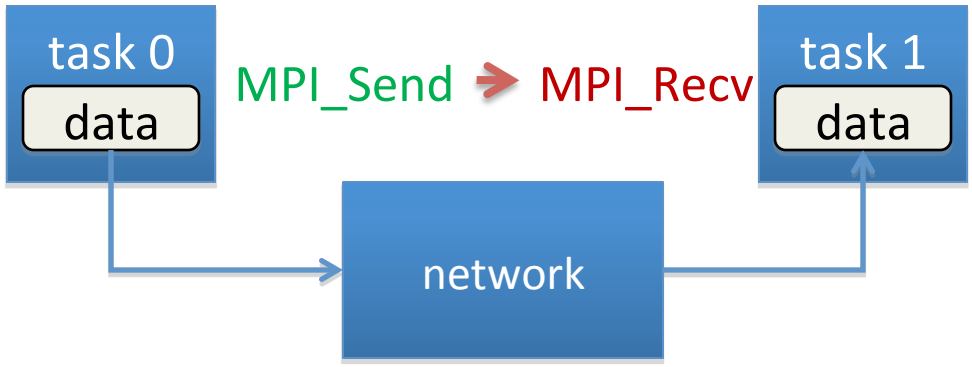
Communicating in Code

- Tasks (independent processes executing anywhere) send and receive “messages” to exchange data.
- Data transfer requires cooperative operation to be performed by each process (point to point communications).



Point-to-Point Communication

- Sending data from one point (process/task) to another point (process/task)
- One task sends while another receives



Basic Communications in MPI

- Standard MPI_Send/MPI_Recv routines
  - Used for basic messaging

Modes of Operation

- Blocking
  - Call does not return until the data area is safe to use
- Non-blocking
  - Initiates send or receive operation, returns immediately
  - Can check or wait for completion of the operation
  - Data area is not safe to use until completion.
- Synchronous and Buffered (later)

Some Common Data Types (basics)

- Data Types (think of it as a data type identifier)
  - Specifies the data type and size in of a variable
  - Predefined MPI types correspond to base language types

Representation	MPI Type Fortran	Fortran	MPI Type C	C
32-bit floating point	MPI_REAL	REAL	MPI_FLOAT	float
64-bit floating point	MPI_DOUBLE_PRECISION	DOUBLE_PRECISION	MPI_DOUBLE	double
32-bit integer	MPI_INTEGER	INTEGER	MPI_INT	int

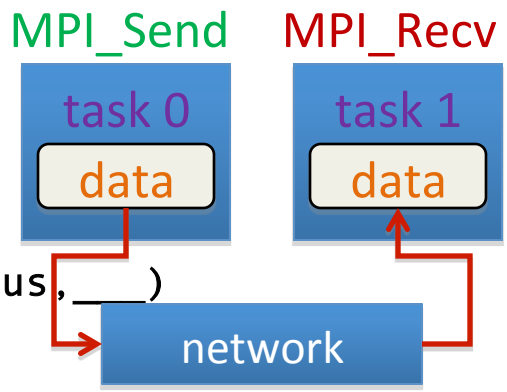
- User-defined Data Types
  - Simple (just a combination of normal data types)
  - Advanced (describes data layout, in essence it is a map)

Communications

- Need an API=library routine
- Need a "channel" to communicate on
- Need to indicate what to send/rec
  - How many "elements", and
  - size of an element
- Need a destination/source
- Need to know if successful.
- Anything else?

✓ MPI_Send & MPI_Recv
✓ MPI_WORLD_COMM
✓ my_array
✓ #
✓ MPI_INTEGER, MPI_Int, etc.
✓ Use Rank
✓More on error in a bit
2 More things—tags and status

Blocking Send/Receive



Generic Syntax

- `MPI_Send(buf, count, datatype, dest, tag, comm,___)`
- `MPI_Recv(buf, count, datatype, source, tag, comm, status,___)`
- **When MPI sends a message, it doesn't just send the contents; it also sends an *envelope* describing the contents:**

Argument	Description
buf	initial address of send/receive buffer (reference)
count	number of items to send/receive (integer)
datatype	MPI data type of items to send/receive
dest	MPI rank of task receiving the data (integer)
source	MPI rank of task sending the data (integer)
tag	message ID (integer)
comm	MPI communicator (set of exchange processors)
status	returns information on the message received

Parts of a P-2-P Communication:

Data

Send to/Recv from

Message ID

Details

buffer	data (address in C, name of array/variable in Fortran)
count	Length of source array (in elements, 1 for scalars)
datatype	Data Type: e.g. `MPI_INT` (C), `MPI_INTEGER` (F90), `MPI_DOUBLE_PRECISION` (F90), `MPI_DOUBLE` (C), etc.
source/dest.	Rank (proc #) of source/destination in communicator group
tag	Message identifier (arbitrary integer)
communicator	Group of processes
status	Information about message
ierr	Error, additional, last argument in Fortran, returned value in C

	C declarations	Fortran declarations
status	`MPI_Status mystat;`	`integer mystat(MPI_STATUS_SIZE)`
datatype	`MPI_Datatype mytype;`	`integer mytype`
comm.	`MPI_Comm mycomm;`	`integer mycomm`

Language Example

- C

```
ierr=MPI_Send(&a[0],cnt,type,dest,tag,com);
```

- F

```
call MPI_Send( a,   cnt,type,dest,tag,com,ierr)
```

- C

```
ierr=MPI_Recv(&b[0],cnt,type, src,tag,com,&status);
```

- F

```
call MPI_Recv( b,   cnt,type, src,tag,com, status,ierr)
```

- MPI_Send blocks until data of *a* has been sent or copied to a buffer.
- MPI_Recv blocks until data is in *b*.

P-2-P Example

```
#include <mpi.h>
int main(int argc, char* argv[]){
MPI_Comm comm=MPI_COMM_WORLD;

int nranks,rank=-1,ierr;
ierr=MPI_Init(&argc, &argv);
ierr=MPI_Comm_size(comm,&nranks);
ierr=MPI_Comm_rank(comm, &rank);




ierr=MPI_Finalize();

printf("iam=%d\n",iam);
}
```

P-2-P Example

```
#include <mpi.h>
int main(int argc, char* argv[]){
MPI_Comm   comm=MPI_COMM_WORLD;
MPI_Status status;
int nranks,rank=-1,ierr,irec=-1;
ierr=MPI_Init(&argc, &argv);
ierr=MPI_Comm_size(comm,&nranks);
ierr=MPI_Comm_rank(comm, &rank );

if(rank==0)
        ierr=MPI_Send(&rank,1,MPI_INT, 1,9, comm);
if(rank==1)
        ierr=MPI_Recv(&irec,1,MPI_INT, 0,9, comm,&status);
ierr=MPI_Finalize();

printf("iam=%d, received=%d\n",rank,irec);
}
```

P-2-P Example

```
program main
use mpi
integer ::  comm=MPI_COMM_WORLD
integer ::status(MPI_STATUS_SIZE),nranks,rank,ierr,irec=-1
call MPI_INIT(ierr)
call MPI_COMM_SIZE(comm, nranks,ierr)
call MPI_COMM_RANK(comm,  rank, ierr)

if(rank==0)&
call MPI_SEND( rank, 1,MPI_INTEGER, 1,9, comm,ierr)
if(rank==1)&
call MPI_RECV( irec, 1,MPI_INTEGER, 0,9, comm,status,ierr)
call MPI_FINALIZE(ierr);

print*,"iam=",rank," received=",irec
end program
```

More on Status

- C
  - **status** (type `MPI_Status`) is a structure which contains three fields **`MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`**
  - `status.MPI_SOURCE`, `status.MPI_TAG`, and `status.MPI_ERROR` contain the source, tag, and error code respectively of the received message
- Fortran
  - **status** is an array of `INTEGER`s of length `MPI_STATUS_SIZE`, and the 3 constants **`MPI_SOURCE`, `MPI_TAG`, `MPI_ERROR`** are the indices of the entries that store the source, tag, & error
  - `status(MPI_SOURCE)`, `status(MPI_TAG)`, `status(MPI_ERROR)` contain respectively the source, the tag, and the error code of the received message.

For production codes you can use MPI_STATUS_IGNORE in place of status–
status fields are not to be filled in; hence no storage is required.

F90

Summary: The 6 Basic MPI Call

- MPI is used to create parallel programs based on message passing
- Usually the same program is run on multiple processors
- The 6 basic calls in MPI are

```
MPI_INIT(ierr)
MPI_COMM_RANK(MPI_COMM_WORLD, myid,ierr)
MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs,ierr)
MPI_SEND(buffer,count,MPI_TYPE, dest,tag,MPI_COMM_WORLD,ierr)
MPI_RECV(buffer,count,MPI_TYPE, src, tag,MPI_COMM_WORLD,stat,ierr)
MPI_FINALIZE(ierr)
```

MPI_TYPE is an MPI Parameter or User Data Type

c/c++ Summary: The 6 Basic MPI Call

- MPI is used to create parallel programs based on message passing
- Usually the same program is run on multiple processors
- The 6 basic calls in MPI are

```
MPI_Init(&argc, &argv);
MPI_Comm_Rank(MPI_COMM_WORLD,&myid);
MPI_Comm_Size(MPI_COMM_WORLD,&numprocs);
MPI_Send(buffer,count,MPI_TYPE, dest,tag,MPI_COMM_WORLD);
MPI_Recv(buffer,count,MPI_TYPE, src, tag,MPI_COMM_WORLD,&stat);
MPI_Finalize();
```

MPI_TYPE is an MPI Parameter or User Data Type
buffer is passed by reference

Excercise: Basic MPI Calls

```
if ( /* I am process A */ ) {
  MPI_Send( /* to: */ B ..... );
  MPI_Recv( /* from: */ B ... );
} else if ( /* I am process B */ ) {
  MPI_Recv( /* from: */ A ... );
  MPI_Send( /* to: */ A ..... );
}
```