MPI Programming 1

Faculty Development Program

RSET, 17th August 2017

MPI

- The Message Passing Interface Standard (MPI) is a , vendor independent message passing library standard by MPI Forum, which has 40+ organizations.
- MPI is not an IEEE or ISO standard, but has become the "industry standard" for writing message passing programs on HPC platforms.

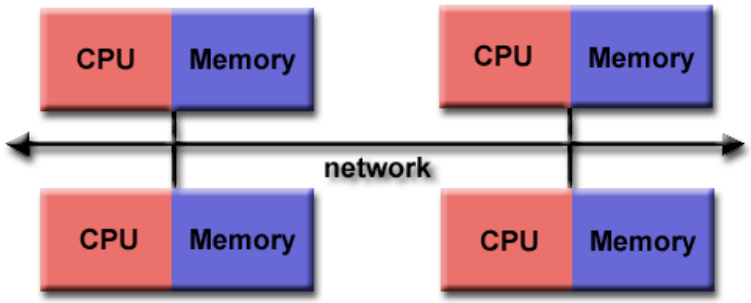
MPI

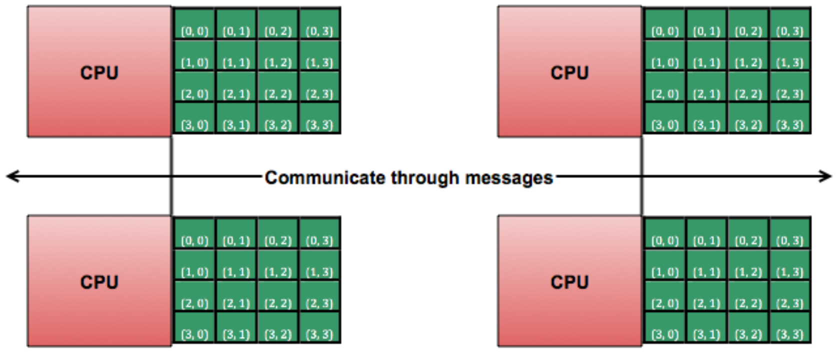
- If you want to scale your application beyond the maximum number of cores available on a node.
- There are different implementations of it.
- Popular implementations are: OpenMPI, MPICH, Intel MPI.
- Originally, MPI was designed for distributed memory architectures.

Distributed Memory Architecture

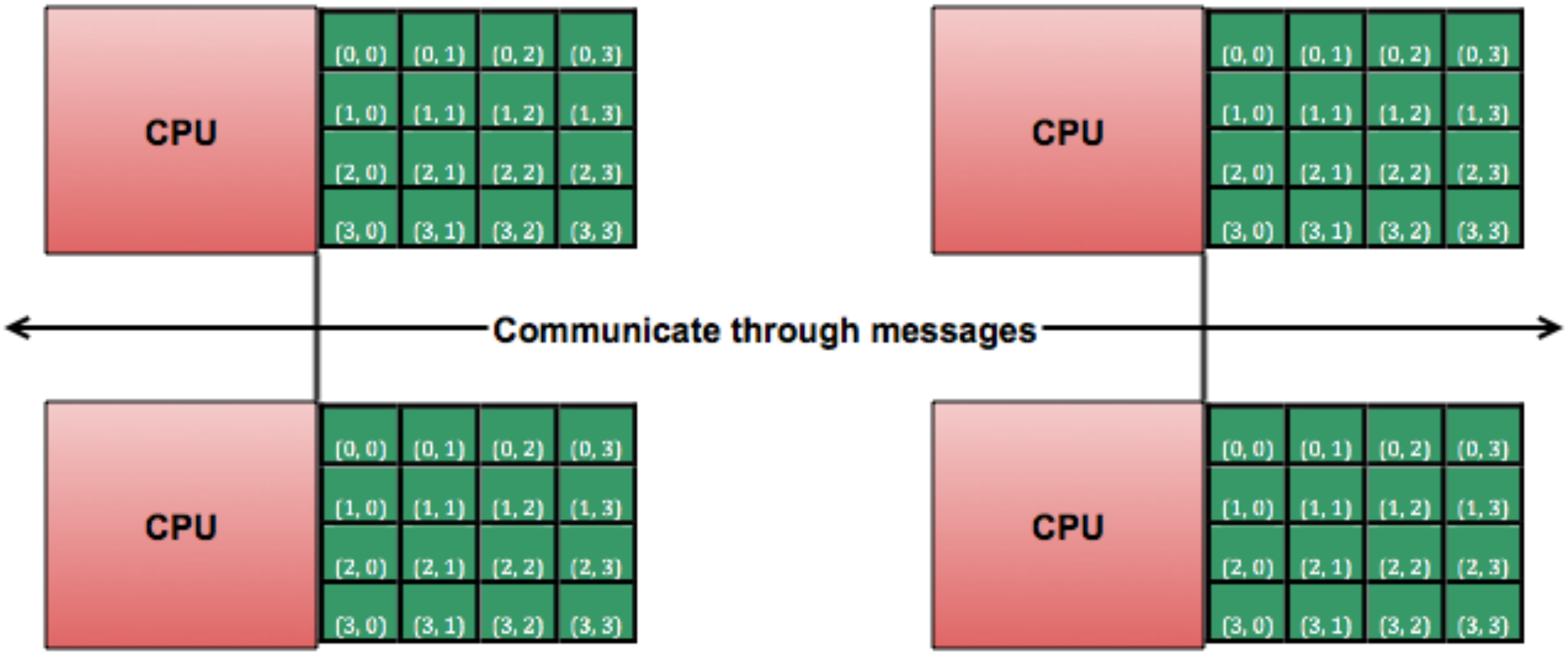
- Multiple-processor computer system in which each process has its own private memory.
- Computational tasks can only operate on local data.
- If remote data is required, the computational task must
- communicate with one or more remote processors.
- The most popular distribute memory programming paradigm is MPI.

Distributed Memory Architecture





Distributed Memory Architecture



MPI

- As architecture trends changed, shared memory SMPs were combined over networks creating hybrid distributed memory / shared memory systems.
- Today, MPI (MPI.3.0) runs on virtually any hardware platform:
  - Distributed Memory
  - Shared Memory
  - Hybrid (MPI+...)

Reasons for Using MPI

- Standardization: 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 that supports MPI standards.
- Performance Opportunities - Vendor implementations should be able to exploit native hardware features to optimize performance.
- Functionality - There are over 430 routines defined in MPI-3

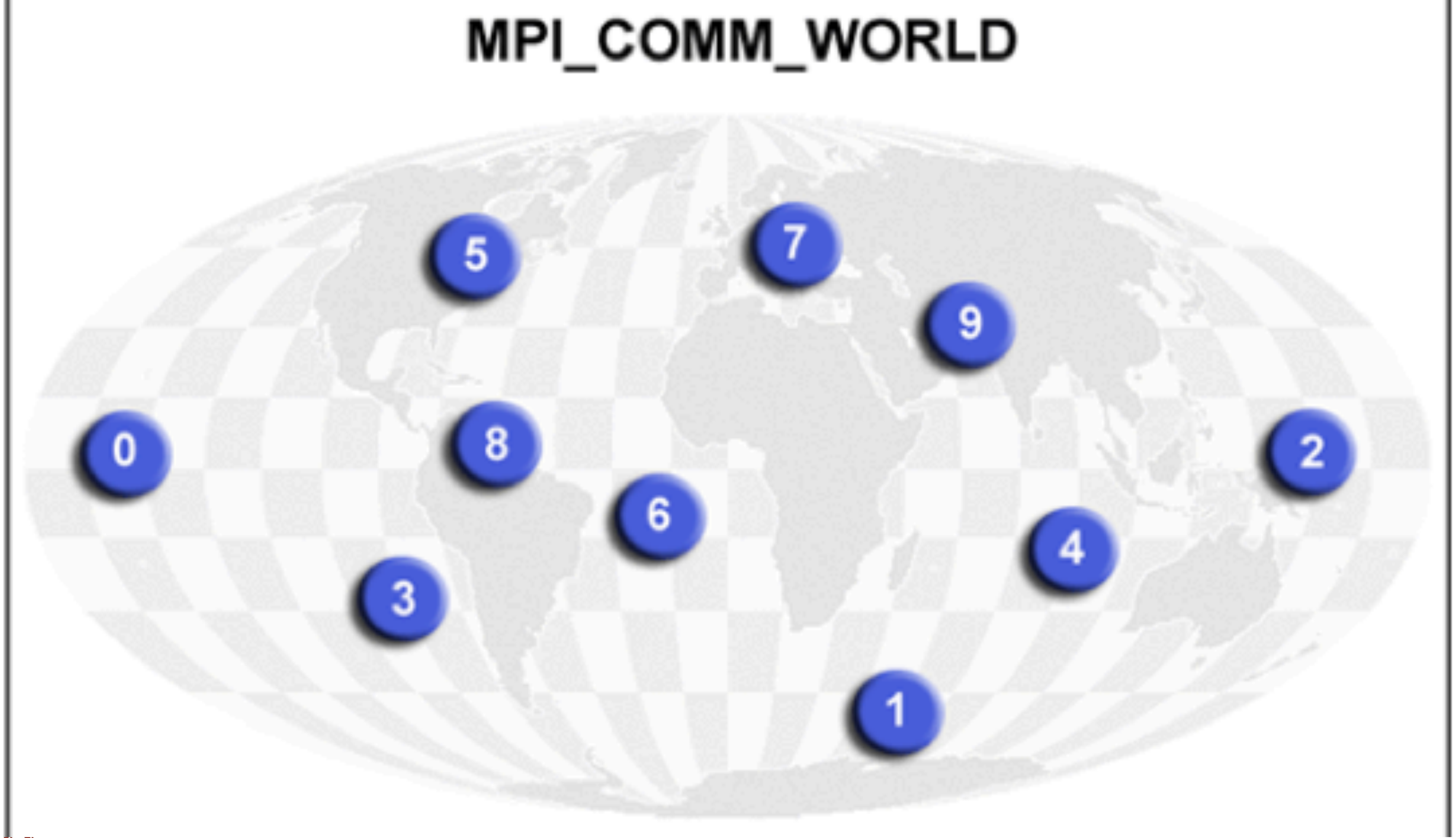
Compile & Run

- Compile:
  - Mpicc hello.c –o hello
- Run
  - mpirun –np *n* o hello
- Include:
  - #include <mpi.h>

Communicators and Groups

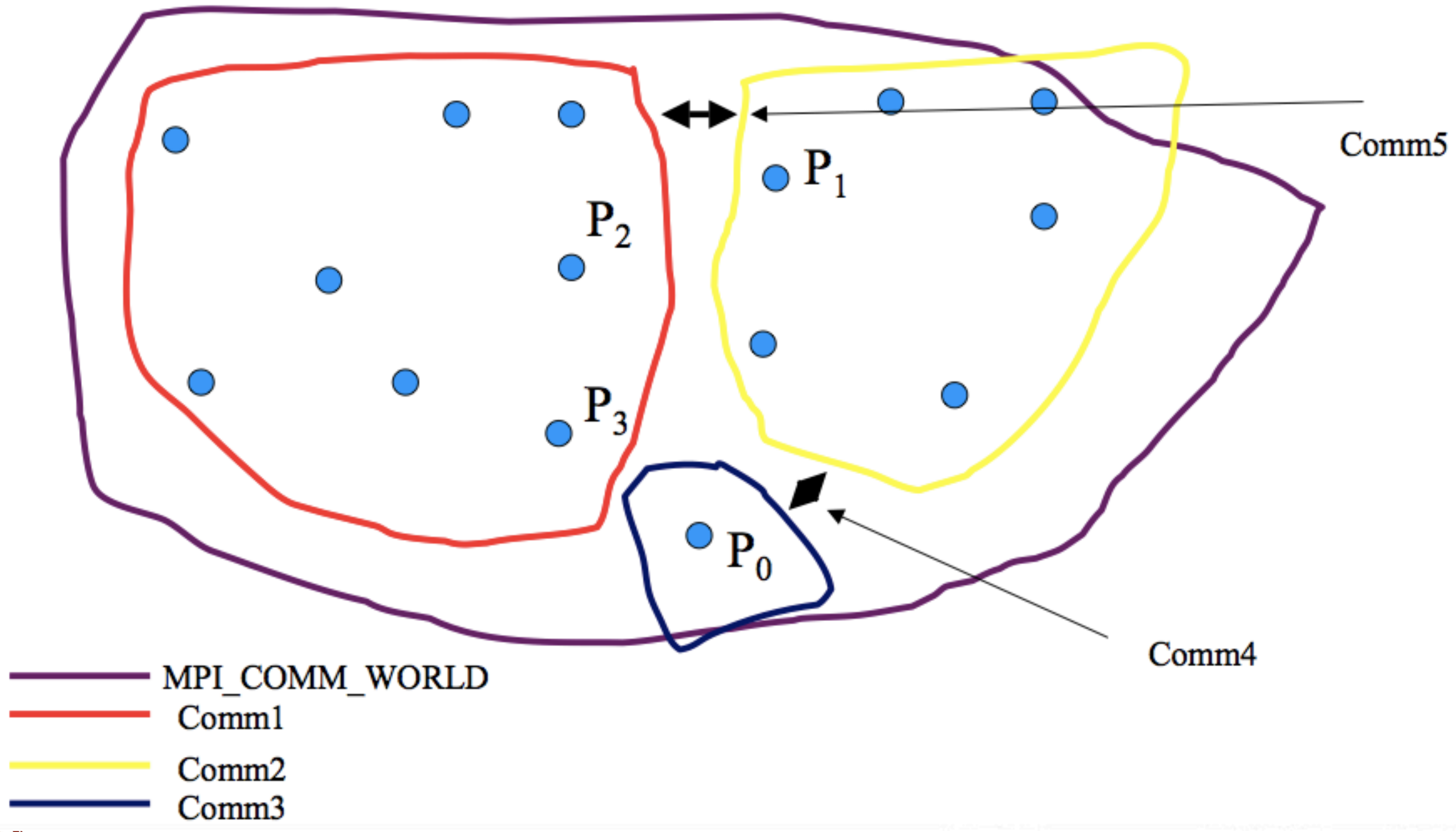
Generally, use MPI_COMM_WORLD whenever a communicator is required - it is the predefined communicator that includes all of your MPI processes.

MPI_COMM_WORLD



Communicators and Groups

Communicators and Groups(cont)



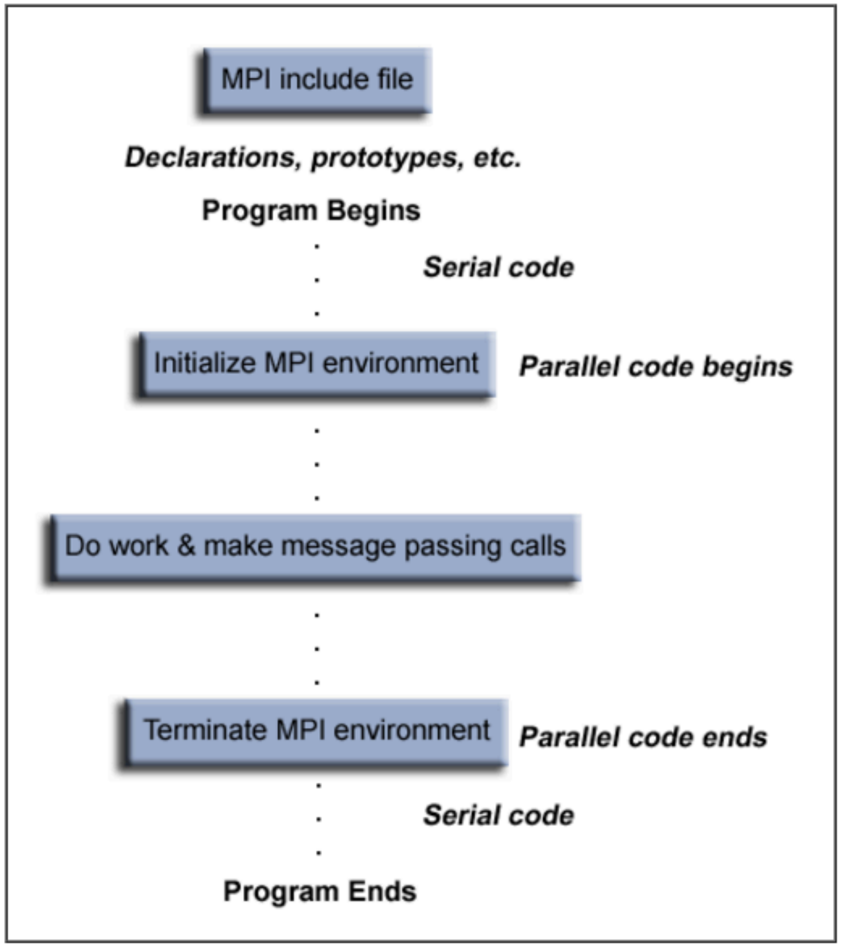
Communicators and Groups

- Refer to previous slide
- There are 4 distinct groups
- These are associated with intracommunicators MPI_COMM_WORLD, comm1, and comm2, and comm3

Communicators and Groups

- P3 is a member of 2 groups and may have different ranks in each group(say 3 & 4)
- If P2 wants to send a message to P1 it must use MPI_COMM_WORLD (intracommunicator) or comm5 (intercommunicator)
- If P2 wants to send a message to P3 it can use MPI_COMM_WORLD (send to rank 3) or comm1 (send to rank 4)
- P0 can broadcast a message to all processes associated with comm2 by using intercommunicator comm5

General MPI Program Structure


MPI include file
Declarations, prototypes, etc.
Program Begins
Serial code
Initialize MPI environment
Parallel code begins
Do work & make message passing calls
Terminate MPI environment
Parallel code ends
Serial code
Program Ends

6 Most Important Functions

- MPI_Init
- MPI_Finalize
- MPI_Comm_size
- MPI_Comm_rank
- MPI_Send
- MPI_Receive

MPI_Init

- Initializes the MPI execution environment.
- In C programs, MPI_Init may be used to pass the command line arguments to all processes, although this is not required by the standard and is implementation dependent.
- MPI_Init (&argc,&argv)

MPI_Finalize

- Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program - no other MPI routines may be called after it.
- **MPI_Finalize ()**

MPI_Comm_size

- Returns the total number of MPI processes in the specified communicator, such as MPI_COMM_WORLD.
- MPI_Comm_size (comm,&size)

MPI_Comm_rank

- Within a communicator, every process has its own unique, integer identifier assigned by the system when the process initializes.
- Ranks are contiguous and begin at zero.
- Used by the programmer to specify the source and destination of messages.
- Often used conditionally by the application to control program execution (if rank=0 do this / if rank=1 do that).

MPI_Comm_rank

- Returns the rank of the calling MPI process within the specified communicator. Initially, each process will be assigned a unique integer rank between 0 and number of tasks - 1 within the communicator MPI_COMM_WORLD.
- **MPI_Comm_rank (comm,&rank)**

MPI_Get_processor_name

- Returns the processor name. Also returns the length of the name. The buffer for "name" must be at least MPI_MAX_PROCESSOR_NAME characters in size.
- **MPI_Get_processor_name (&name,&resultlength)**

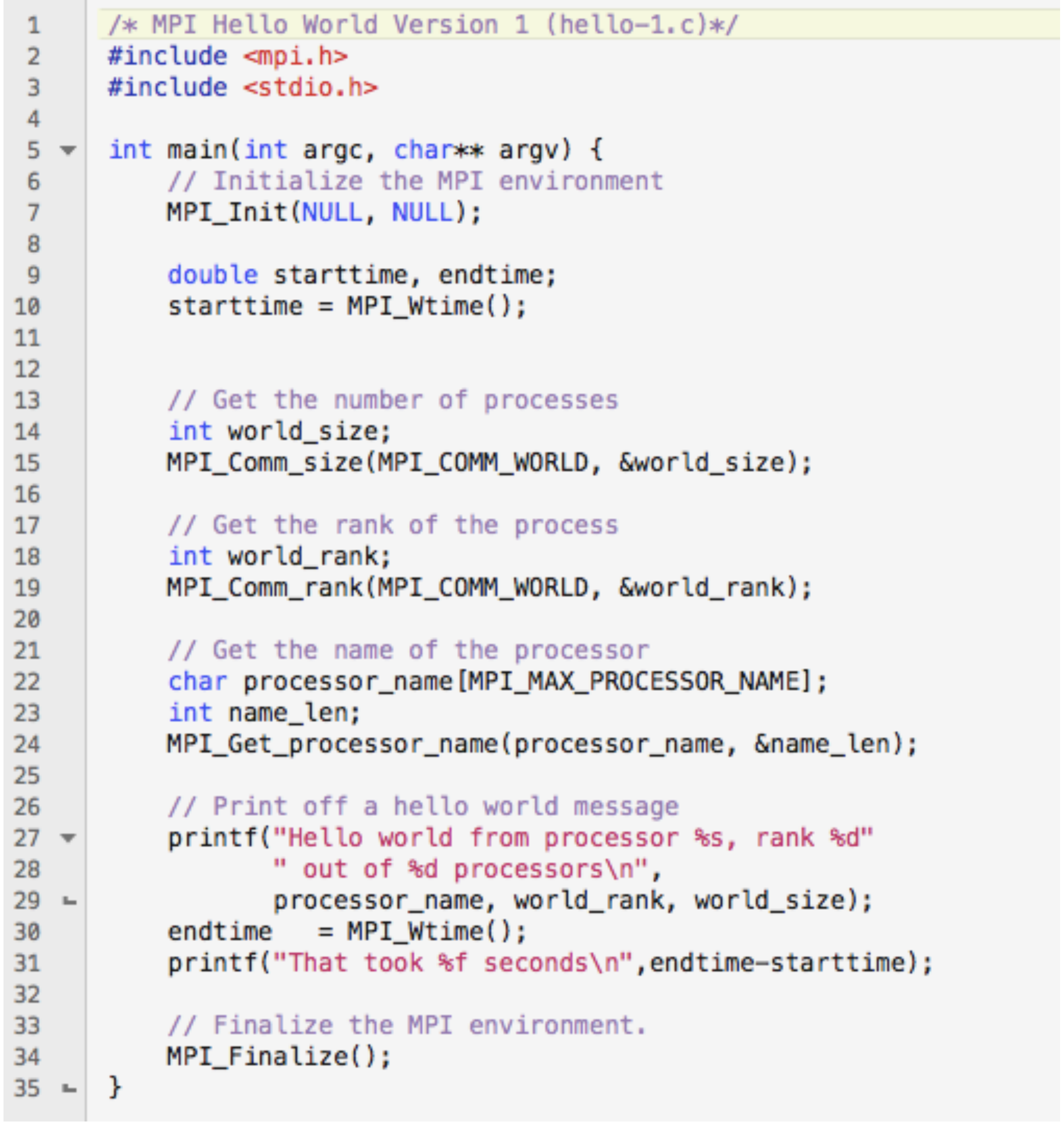
MPI_Wtime()

- Returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past.
- The times returned are local to the node that called them.
- There is no requirement that different nodes return ``the same time.''

MPI_Wtime()

```
{
        double starttime, endtime;
        starttime = MPI_Wtime();
         ....  stuff to be timed  ...
        endtime   = MPI_Wtime();
        printf("That took %f seconds\n",endtime-
starttime);
}
```

HELLO WORLD



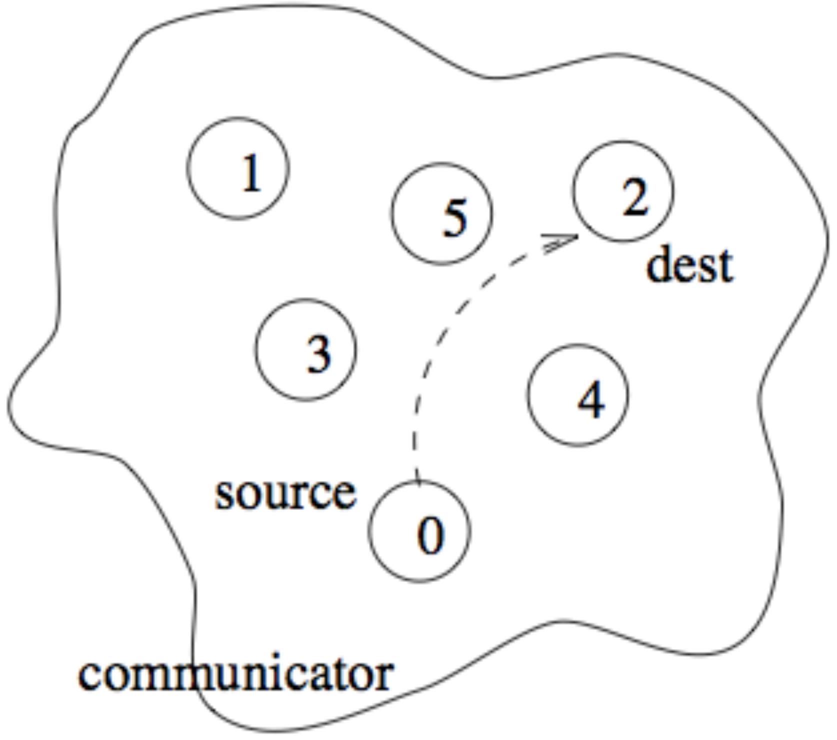
```c
/* MPI Hello World Version 1 (hello-1.c)*/
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    double starttime, endtime;
    starttime = MPI_Wtime();


    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d"
           " out of %d processors\n",
           processor_name, world_rank, world_size);
    endtime   = MPI_Wtime();
    printf("That took %f seconds\n",endtime-starttime);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

MPI Programming

Point-to-point Communications

Point to Point Communication

- In point-to-point communication a process sends a message to another specific process



Point to Point Communication

- To send a message, a source process makes an MPI call which specifies a destination process in terms of its rank in the appropriate communicator (e.g. MPI_COMM_WORLD).
- The destination process also has to make an MPI call if it is to receive the message.

MPI Send

```
MPI_SEND(void *start, int
   count,MPI_DATATYPE datatype, int dest,
   int tag, MPI_COMM comm)
```

- The message buffer is described by (start, count, datatype).
- dest is the rank of the target process in the defined communicator.
- tag is the message identification number.

MPI Send

- MPI_SEND (buf, count, datatype, dest, tag, comm)
  - **buf** is the address of the data to be sent.
  - **count** is the number of elements of the MPI datatype which buf contains.
  - **datatype** is the MPI datatype.
  - **dest** is the destination process for the message. This is specified by the rank of the destination process within the group associated with the communicator comm.

MPI Send

- **tag** is a marker used by the sender to distinguish between different types of messages. Tags are used by the programmer to distinguish between different sorts of message.
- **comm** is the communicator shared by the sending and receiving processes. Only processes which have the same communicator can communicate.

MPI Recv

```
MPI_RECV(void *start, int count,
MPI_DATATYPE datatype, int source, int tag,
MPI_COMM comm, MPI_STATUS *status)
```

- Source is the rank of the sender in the communicator.

- The receiver can specify a wildcard value for souce (MPI_ANY_SOURCE) and/or a wildcard value for tag (MPI_ANY_TAG), indicating that any source and/or tag are acceptable

MPI Staus

- Status is used for exrtra information about the received message if a wildcard receive mode is used. It can query:
  - The rank of the sender.
  - The tag of the message.
  - The length of the message.
- status.MPI_SOURCE, status.MPI_TAG and status.MPI_ERROR contain the source, tag, and error code, respectively, of the received message.

MPI Recv

- MPI_RECV (buf, count, datatype, source, tag, comm, status)
  - **buf** is the address where the data should be placed once received (the receive buffer). For the communication to succeed, the receive buffer must be large enough to hold the message without truncation — if it is not, behavior is undefined. The buffer may however be longer than the data received.

MPI Recv

- **count** is the number of elements of a certain MPI datatype which buf can con- tain. The number of data elements actually received may be less than this.
- **datatype** is the MPI datatype for the message. This must match the MPI da- tatype specified in the send routine.

MPI Recv

- **source** is the rank of the source of the message in the group associated with the communicator comm. Instead of prescribing the source, messages can be received from one of a number of sources by specifying a *wildcard*, MPI_ANY_SOURCE, for this argument.
- **tag** is used by the receiving process to prescribe that it should receive only a message with a certain tag. Instead of prescribing the tag, the wildcard MPI_ANY_TAG can be specified for this argument.

MPI Recv

- **comm** is the communicator specified by both the sending and receiving process. There is no wildcard option for this argument.
- **status:** If the receiving process has specified wildcards for both or either of source or tag, then the corresponding information from the message that was actually received may be required. This information is returned in status, and can be queried using different routines.

MPI Message

- The data message which is sent or received is described by a triple (address, count, datatype).
  - MPI_SEND (buf, count, datatype, dest, tag, comm)
- An MPI message is an array of elements of a particular MPI datatype.

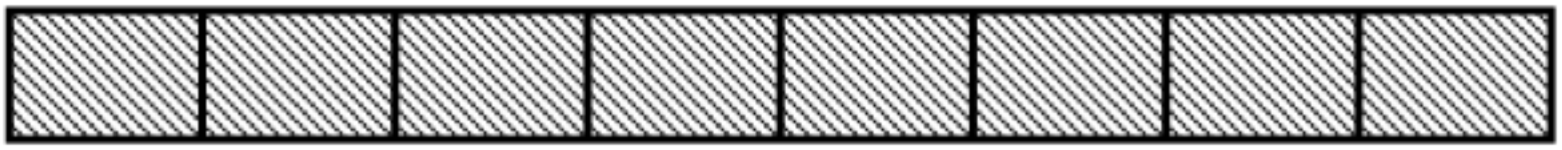
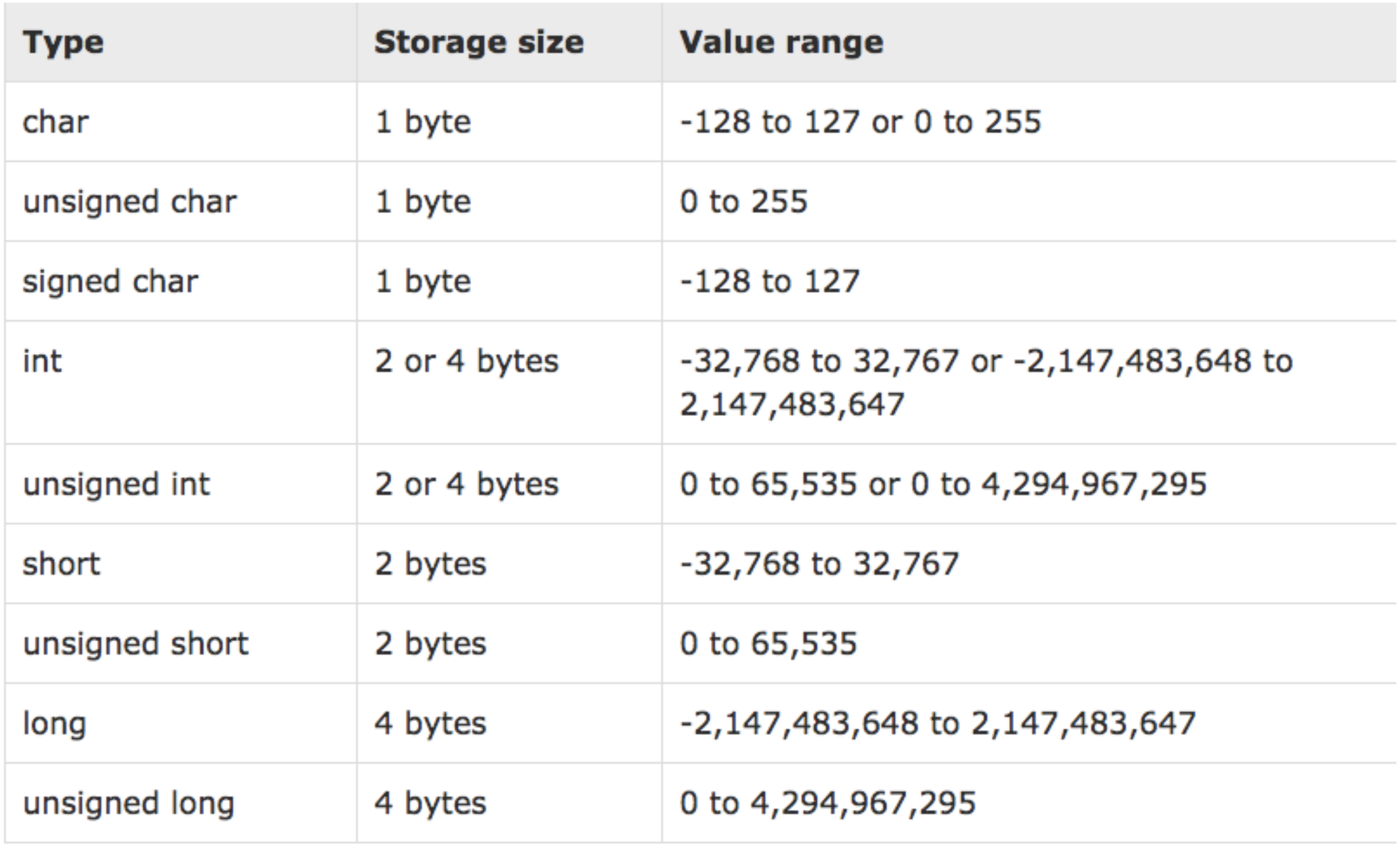


Figure 2: An MPI message.

C Data Types



Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

MPI Message

- All MPI messages are typed in the sense that the type of the contents must be specified in the send and receive.
- Why defining the datatypes during the send of a message?
  - Because communications take place between heterogeneous machines. Which may have different data representation and length in the memory.

Basic MPI types

<u>**MPI datatype**</u>	<u>**C datatype**</u>
MPI_CHAR	signed char
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_SHORT	signed short
MPI_UNSIGNED_SHORT	unsigned short
MPI_INT	signed int
MPI_UNSIGNED	unsigned int
MPI_LONG	signed long
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

MPI Message

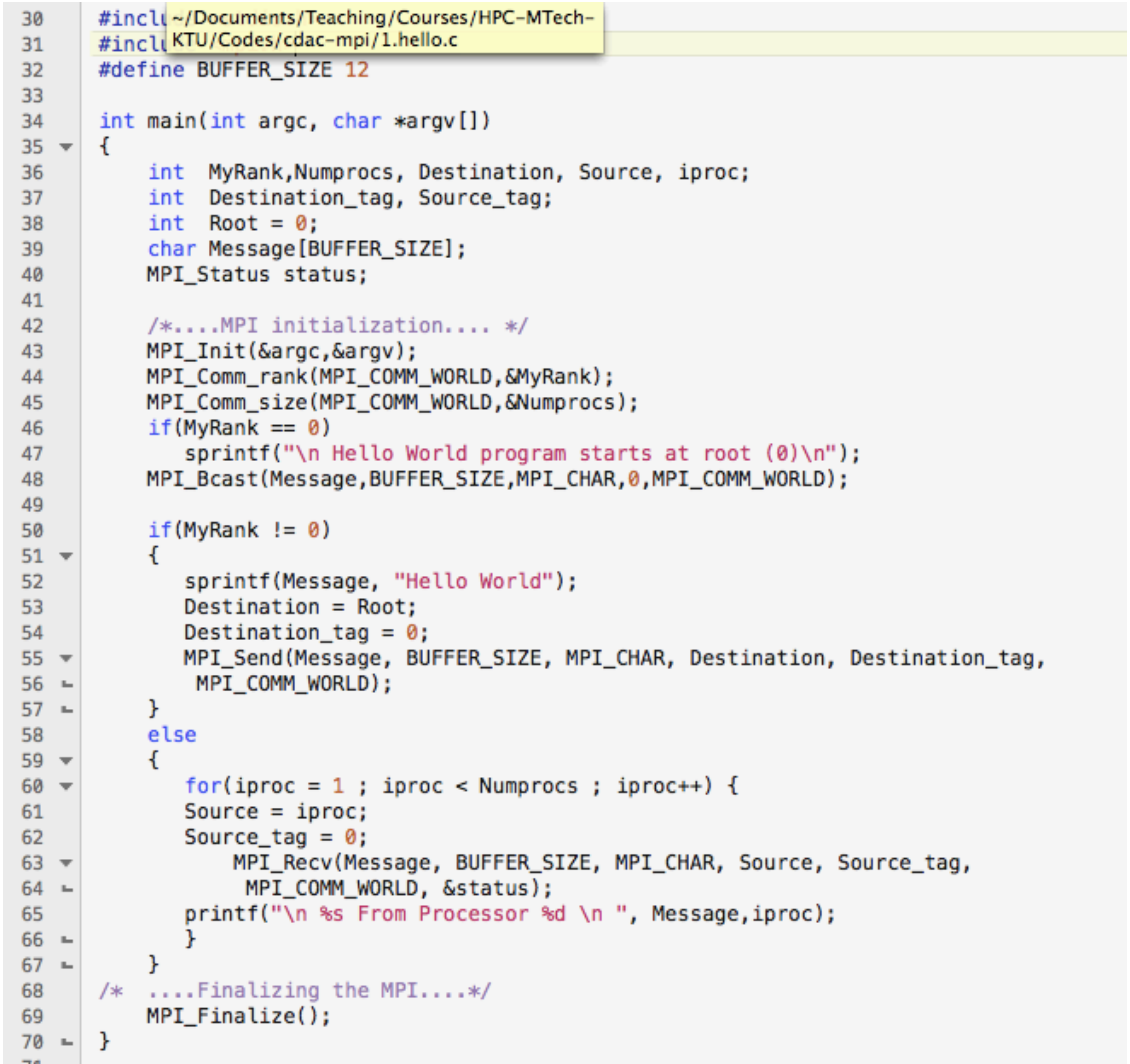
- Usually, the datatype specified in the receive must match the datatype specified in the send.
  - The great advantage of this is that MPI can support heterogeneous parallel architectures
  - i.e. parallel machines built from different processors, because type conversion can be performed when necessary.
  - Thus two processors may represent, say, an integer in different ways, but MPI processes on these processors can use MPI to send integer messages without being aware of the heterogeneity

Derived DataTypes

- More complex datatypes can be constructed at run-time. These are called derived datatypes and are built from the basic datatypes.
- They can be used for sending strided vectors, C structs etc.

SEND -RECV

HELLO WORLD



~/Documents/Teaching/Courses/HPC-MTech-KTU/Codes/cdac-mpi/1.hello.c

```c
#inclu
#inclu
#define BUFFER_SIZE 12

int main(int argc, char *argv[])
{
    int  MyRank,Numprocs, Destination, Source, iproc;
    int  Destination_tag, Source_tag;
    int  Root = 0;
    char Message[BUFFER_SIZE];
    MPI_Status status;

    /*....MPI initialization.... */
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&MyRank);
    MPI_Comm_size(MPI_COMM_WORLD,&Numprocs);
    if(MyRank == 0)
       sprintf("\n Hello World program starts at root (0)\n");
    MPI_Bcast(Message,BUFFER_SIZE,MPI_CHAR,0,MPI_COMM_WORLD);

    if(MyRank != 0)
    {
       sprintf(Message, "Hello World");
       Destination = Root;
       Destination_tag = 0;
       MPI_Send(Message, BUFFER_SIZE, MPI_CHAR, Destination, Destination_tag,
        MPI_COMM_WORLD);
    }
    else
    {
       for(iproc = 1 ; iproc < Numprocs ; iproc++) {
       Source = iproc;
       Source_tag = 0;
           MPI_Recv(Message, BUFFER_SIZE, MPI_CHAR, Source, Source_tag,
            MPI_COMM_WORLD, &status);
       printf("\n %s From Processor %d \n ", Message,iproc);
       }
    }
/*  ....Finalizing the MPI....*/
    MPI_Finalize();
}
```

Example Code

MATRIX VECTOR MULTIPLIER

Matrix Vector Multiplier

$$\mathbf{Ax} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix}.$$

Matrix Vector Multiplier

$$\begin{aligned} A\mathbf{x} &= \begin{bmatrix} 1 & -1 & 2 \\ 0 & -3 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} 2\cdot 1 - 1\cdot 1 + 0\cdot 2 \\ 2\cdot 0 - 1\cdot 3 + 0\cdot 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ -3 \end{bmatrix}. \end{aligned}$$

Thank you

LEARN • SERVE • EXCEL
RAJAGIRI