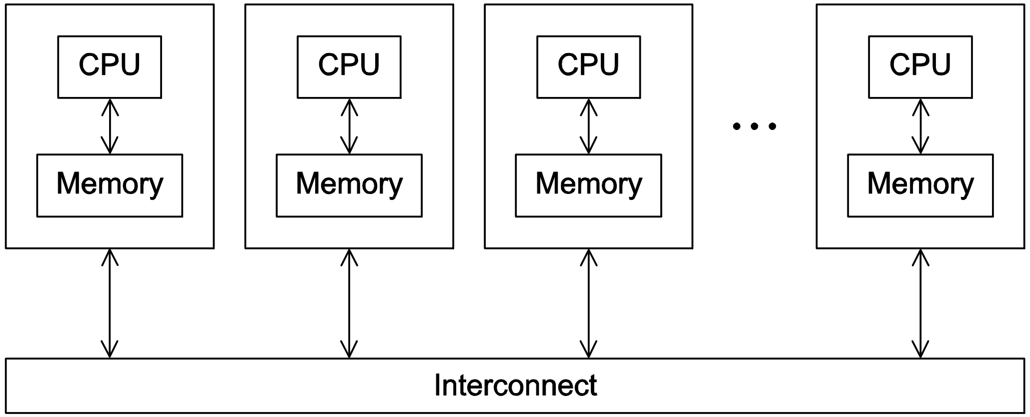
ECE 432/532
Programming for Parallel Processors

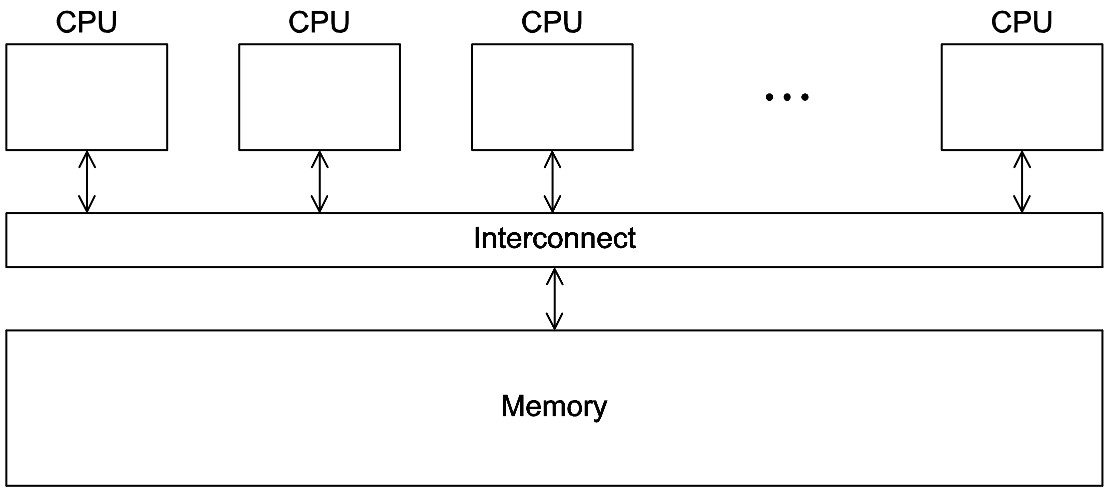
Roadmap

- Writing your first MPI program.
- Using the common MPI functions.
- The Trapezoidal Rule in MPI.
- Collective communication.
- MPI derived datatypes.
- Performance evaluation of MPI programs.
- Parallel sorting.
- Safety in MPI programs.

A distributed memory system



A shared memory system



What is MPI?

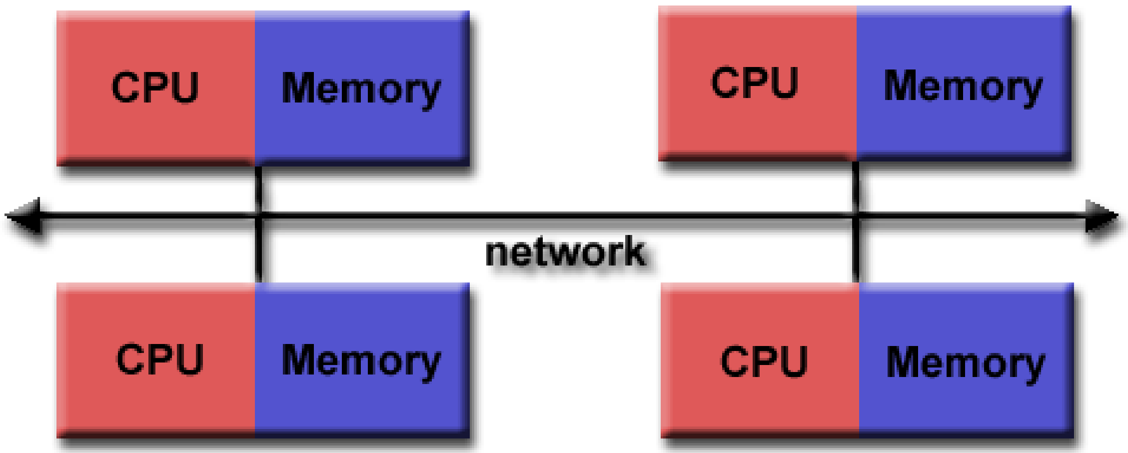
- An Interface Specification → M P I = Message Passing Interface
- MPI is NOT a library - but rather the specification of what such a library should be
- MPI primarily addresses the message-passing parallel programming model:
  - data is moved from the address space of one process to that of another process through cooperative operations on each process.
- Simply stated, the goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs. The interface attempts to be:
  - Practical
  - Portable
  - Efficient
  - Flexible

What is MPI?

- The MPI standard has gone through a number of revisions, with the most recent version being MPI-3.x
- Interface specifications have been defined for C and Fortran90 language bindings:
  - C++ bindings from MPI-1 are removed in MPI-3
  - MPI-3 also provides support for Fortran 2003 and 2008 features
- Actual MPI library implementations differ in which version and features of the MPI standard they support.
  - Developers/users will need to be aware of this.

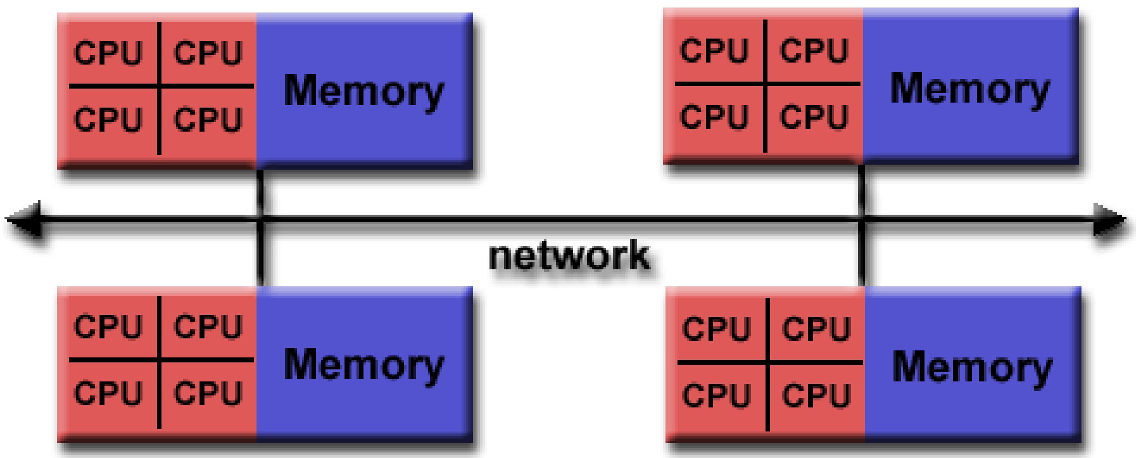
Programming model

- Originally, MPI was designed for distributed memory architectures,



Programming model

- As architecture trends changed, shared memory SMPs were combined over networks creating hybrid distributed memory / shared memory systems.



Programming model

- Today, MPI runs on virtually any hardware platform:
  - Distributed Memory
  - Shared Memory
  - Hybrid
- BUT, the programming model <u>clearly remains a distributed memory model</u>
- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.

Reasons for using 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 that supports (and is compliant with) the MPI standard

Reasons for using MPI

- <u>Performance Opportunities</u>
  - Vendor implementations should be able to exploit native hardware features to optimize performance. Any implementation is free to develop optimized algorithms.

- <u>Functionality</u>
  - 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

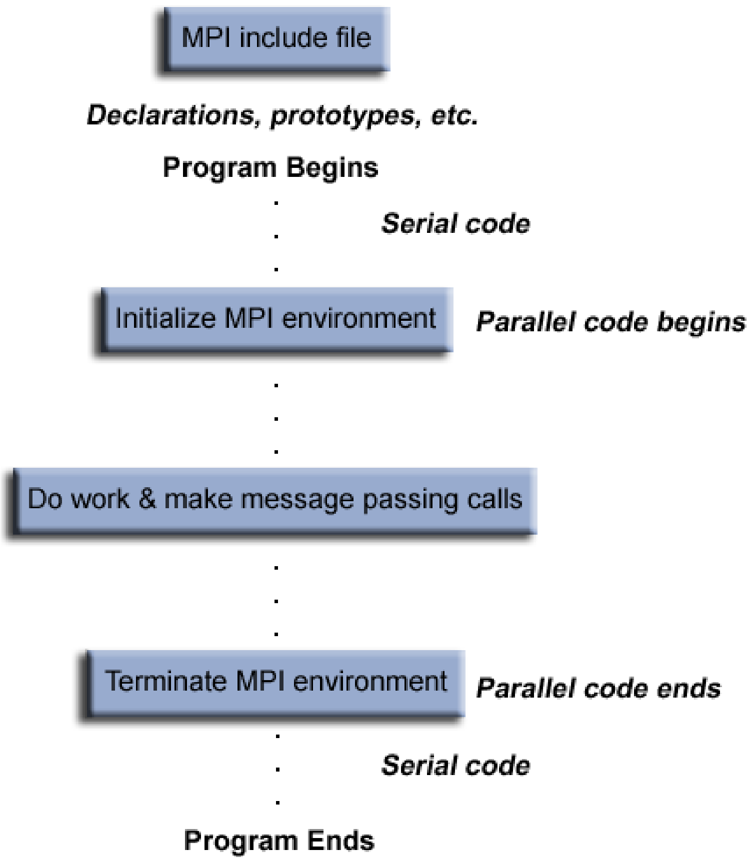
Implementations and Compilers

- Although the MPI programming interface has been standardized, actual library implementations will differ

- The way MPI programs are compiled and run on different platforms may also vary

- Currently, three different MPI implementations are supported:
  - MVAPICH - Linux clusters
  - Open MPI - Linux clusters
  - IBM MPI - BG/Q clusters

Getting Started



- **Format of MPI Calls:**
  - C names are case sensitive
  - 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

Getting Started

- Communicators and Groups
  - MPI uses objects called communicators and groups to define which collection of processes may communicate with each other
  - Most MPI routines requires a communicator as an argument
  - MPI_COMM_WORLD is the predefined communicator that includes all of your MPI processes

- Rank
  - Within a communicator, every process has its own unique, integer identifier assigned by the system when the process initializes. A rank is sometimes also called a "task ID". Ranks are contiguous and begin at zero.
  - Used by the programmer to specify the source and destination of messages. (if rank=0 do this / if rank=1 do that)

Getting Started

- Error Handling
  - Most MPI routines include a return/error code parameter
  - However, according to the MPI standard, the default behavior of an MPI call is to abort if there is an error
  - You will probably not be able to capture a return/error code other than MPI_SUCCESS (zero)
  - The standard does provide a means to override this default error handler.
  - The types of errors displayed to the user are implementation dependent.

Hello World in C

```
#include <stdio.h>

int main(void) {
    printf("hello, world\n");

    return 0;
}
```

- Compile: gcc –g –Wall hello.c –o hello
- Execution: ./hello

Hello World- MPI

```c
#include <stdio.h>
#include <string.h>  /* For strlen              */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
   char       greeting[MAX_STRING];
   int        comm_sz;  /* Number of processes */
   int        my_rank;  /* My process rank     */

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank != 0) {
      sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
      MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
            MPI_COMM_WORLD);
   } else {
      printf("Greetings from process %d of %d!\n", my_rank,
           comm_sz);
      for (int q = 1; q < comm_sz; q++) {
         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("%s\n", greeting);
      }
   }

   MPI_Finalize();
   return 0;
}  /* main */
```

Hello World- MPI

```c
#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
   char       greeting[MAX_STRING];
   int        comm_sz;  /* Number of processes */
   int        my_rank;  /* My process rank     */

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank != 0) {
      sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
      MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
            MPI_COMM_WORLD);
   } else {
      printf("Greetings from process %d of %d!\n", my_rank,
          comm_sz);
      for (int q = 1; q < comm_sz; q++) {
         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("%s\n", greeting);
      }
   }

   MPI_Finalize();
   return 0;
}  /* main */
```

Compilation

wrapper script to compile

source file

```
mpicc  -g  -Wall  -o  mpi_hello  mpi_hello.c
```

produce debugging information

create this executable file name (as opposed to default a.out)

turns on all warnings

Execution

mpiexec -n <number of processes> <executable>

mpiexec -n 1 ./mpi_hello

run with 1 process

mpiexec -n 4 ./mpi_hello

run with 4 processes

Execution

```
mpiexec  -n  1  ./mpi_hello
```

Greetings from process 0 of 1 !

```
mpiexec  -n  4  ./mpi_hello
```

Greetings from process 0 of 4 !

Greetings from process 1 of 4 !

Greetings from process 2 of 4 !

Greetings from process 3 of 4 !

MPI Programs

- Written in C.
  - Has main.
  - Uses stdio.h, string.h, etc.
- Need to add mpi.h header file.
- Identifiers defined by MPI start with "MPI_".
- First letter following underscore is uppercase.
  - For function names and MPI-defined types.
  - Helps to avoid confusion.

Hello World- MPI

```c
#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
   char       greeting[MAX_STRING];
   int        comm_sz;  /* Number of processes */
   int        my_rank;  /* My process rank     */

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank != 0) {
      sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
      MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
            MPI_COMM_WORLD);
   } else {
      printf("Greetings from process %d of %d!\n", my_rank,
         comm_sz);
      for (int q = 1; q < comm_sz; q++) {
         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("%s\n", greeting);
      }
   }

   MPI_Finalize();
   return 0;
}  /* main */
```

MPI_Init:

- Tells MPI to do all the necessary setup
- No other MPI functions should be called before the program calls MPI_Init.

```c
int MPI_Init(
      int*      argc_p  /* in/out */,
      char***   argv_p  /* in/out */);
```

- The arguments, argc_p and argv_p, are pointers to the arguments to main, argc, and argv
- When a program doesn't use these arguments, just pass NULL for both

Hello World- MPI

```c
#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
   char       greeting[MAX_STRING];
   int        comm_sz;  /* Number of processes */
   int        my_rank;  /* My process rank     */

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank != 0) {
      sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
      MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
            MPI_COMM_WORLD);
   } else {
      printf("Greetings from process %d of %d!\n", my_rank,
         comm_sz);
      for (int q = 1; q < comm_sz; q++) {
         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("%s\n", greeting);
      }
   }

   MPI_Finalize();
   return 0;
}  /* main */
```

MPI_Finalize:

- Tells MPI we're done, so clean up anything allocated for this program

```c
int MPI_Finalize(void);
```

- In general, no MPI functions should be called after the call to MPI_Finalize

```c
#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
   char       greeting[MAX_STRING];
   int        comm_sz;  /* Number of processes */
   int        my_rank;  /* My process rank     */

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank != 0) {
      sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
      MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
            MPI_COMM_WORLD);
   } else {
      printf("Greetings from process %d of %d!\n", my_rank,
           comm_sz);
      for (int q = 1; q < comm_sz; q++) {
         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("%s\n", greeting);
      }
   }

   MPI_Finalize();
   return 0;
}  /* main */
```

Hello World- MPI

MPI_Finalize:

- Tells MPI we're done, so clean up anything allocated for this program

```c
int MPI_Finalize(void);
```

- In general, no MPI functions should be called after the call to MPI_Finalize

It's not necessary that the calls to MPI_Init and MPI_Finalize be in main

Communicators

- A collection of processes that can send messages to each other.
- MPI_Init defines a communicator that consists of all the processes created when the program is started.

- Called MPI_COMM_WORLD.

Hello World- MPI

```c
#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
   char       greeting[MAX_STRING];
   int        comm_sz;  /* Number of processes */
   int        my_rank;  /* My process rank     */

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank != 0) {
      sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
      MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
            MPI_COMM_WORLD);
   } else {
      printf("Greetings from process %d of %d!\n", my_rank,
           comm_sz);
      for (int q = 1; q < comm_sz; q++) {
         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("%s\n", greeting);
      }
   }

   MPI_Finalize();
   return 0;
}  /* main */
```

MPI_Finalize :

- It is the predefined communicator that includes all of your MPI processes.

```c
#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
   char       greeting[MAX_STRING];
   int        comm_sz;  /* Number of processes */
   int        my_rank;  /* My process rank     */

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank != 0) {
      sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
      MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
            MPI_COMM_WORLD);
   } else {
      printf("Greetings from process %d of %d!\n", my_rank,
           comm_sz);
      for (int q = 1; q < comm_sz; q++) {
         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("%s\n", greeting);
      }
   }

   MPI_Finalize();
   return 0;
}  /* main */
```

Hello World- MPI

MPI_Comm_size:

- Returns in its second argument the number of processes in the communicator

```c
int MPI_Comm_size(
      MPI_Comm    comm       /* in   */,
      int*        comm_sz_p  /* out  */);
```

number of processes in the communicator

Hello World- MPI

```c
#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
   char       greeting[MAX_STRING];
   int        comm_sz;  /* Number of processes */
   int        my_rank;  /* My process rank     */

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank != 0) {
      sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
      MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
            MPI_COMM_WORLD);
   } else {
      printf("Greetings from process %d of %d!\n", my_rank,
         comm_sz);
      for (int q = 1; q < comm_sz; q++) {
         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("%s\n", greeting);
      }
   }

   MPI_Finalize();
   return 0;
}  /* main */
```

MPI_Comm_size:

- Returns in its second argument the number of processes in the communicator

```c
int MPI_Comm_size(
      MPI_Comm  comm       /* in  */,
      int*      comm_sz_p  /* out */);
```

number of processes in the communicator

MPI_Comm_rank:

- Returns the calling process' rank in the communicator

```c
int MPI_Comm_rank(
      MPI_Comm  comm       /* in  */,
      int*      my_rank_p  /* out */);
```

my rank (the process making this call)

SPMD

- Single-Program Multiple-Data
- We compile <u>one</u> program.
- Process 0 does something different.
  - Receives messages and prints them while the other processes do the work.

- The if-else construct makes our program SPMD.

Communication

- In Lines 17 and 18, each process, other than process 0, creates a message it will send to process 0
- Lines 19–20 actually send the message to process 0
- Process 0, on the other hand, simply prints its message using printf, and then uses a **for** loop to receive and print the messages sent by other processes

Hello World- MPI

```c
#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
   char       greeting[MAX_STRING];
   int        comm_sz;  /* Number of processes */
   int        my_rank;  /* My process rank     */

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank != 0) {
      sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
      MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
            MPI_COMM_WORLD);
   } else {
      printf("Greetings from process %d of %d!\n", my_rank,
          comm_sz);
      for (int q = 1; q < comm_sz; q++) {
         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("%s\n", greeting);
      }
   }

   MPI_Finalize();
   return 0;
}  /* main */
```

MPI_Send:

```c
int MPI_Send(
    void*          msg_buf_p     /* in */,
    int            msg_size      /* in */,
    MPI_Datatype   msg_type      /* in */,
    int            dest          /* in */,
    int            tag           /* in */,
    MPI_Comm       communicator  /* in */);
```

- msg_but_p, is a pointer to the block of memory containing the contents of the message
- msg_size argument is the number of characters in the message
- The msg_type argument is MPI_CHAR
- dest, specifies the rank of the process that should receive the message
- tag, is a nonnegative **int**. It can be used to distinguish messages that are otherwise identical
- The final argument is a communicator. All MPI functions that involve communication have a communicator argument

```c
#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
   char       greeting[MAX_STRING];
   int        comm_sz;  /* Number of processes */
   int        my_rank;  /* My process rank     */

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank != 0) {
      sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
      MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
            MPI_COMM_WORLD);
   } else {
      printf("Greetings from process %d of %d!\n", my_rank,
           comm_sz);
      for (int q = 1; q < comm_sz; q++) {
         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("%s\n", greeting);
      }
   }

   MPI_Finalize();
   return 0;
}  /* main */
```

Hello World- MPI

MPI_Send:

```c
int MPI_Send(

   void*          msg_buf_p     /* in */,
   int            msg_size      /* in */,
   MPI_Datatype   msg_type      /* in */,
   int            dest          /* in */,
   int            tag           /* in */,
   MPI_Comm       communicator  /* in */);
```

- The msg_type argument is MPI_CHAR

C types (int, char, and soon.) can't be passed as arguments to functions

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Hello World- MPI

```c
#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
   char       greeting[MAX_STRING];
   int        comm_sz;  /* Number of processes */
   int        my_rank;  /* My process rank     */

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank != 0) {
      sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
      MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
            MPI_COMM_WORLD);
   } else {
      printf("Greetings from process %d of %d!\n", my_rank,
         comm_sz);
      for (int q = 1; q < comm_sz; q++) {
         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("%s\n", greeting);
      }
   }

   MPI_Finalize();
   return 0;
}  /* main */
```

MPI_Recv:

```c
int MPI_Recv(
      void*          msg_buf_p     /* out */,
      int            buf_size      /* in  */,
      MPI_Datatype   buf_type      /* in  */,
      int            source        /* in  */,
      int            tag           /* in  */,
      MPI_Comm       communicator  /* in  */,
      MPI_Status*    status_p      /* out */);
```

- msg_buf_p points to the block of memory
- buf_size determines the number of objects that can be stored in the block
- buf_type indicates the type of the objects
- The source specifies the process from which the message should be received
- The tag argument should match the tag argument of the message being sent
- The communicator argument must match the communicator used by the sending process
- status_p is pointer to an MPI structure

Message matching

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
         send_comm);
```

r

MPI_Send

src = q

MPI_Recv

dest = r

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

q

Message matching

- The message sent by *q* with the call to MPI_Send can be received by *r* with the call to MPI_Recv if:
  - recv_comm = send_comm
  - recv_tag = send_tag
  - dest = r and
  - src = q
- Most of the time, the following rule will suffice
  - If recv_type = send_type and recv_buf_sz ≥ send_buf_sz, then the message sent by q can be successfully received by r

Message matching

- Until now, one process is receiving messages from multiple processes

and

- the receiving process doesn’t know the order in which the other processes will send the messages

- However, if the work assigned to each process takes an unpredictable amount of time, then 0 has no way of knowing the order in which the processes will finish
  - it could happen that process a process could sit and wait for the other processes to finish

- In order to avoid this problem, MPI provides a special constant MPI_ANY_SOURCE that can be passed to MPI_Recv

Message matching

- For the previous example:

```
for (i = 1; i < comm_sz; i++) {
   MPI_Recv(result, result_sz, result_type, MPI_ANY_SOURCE,
         result_tag, comm, MPI_STATUS_IGNORE);
   Process_result(result);
}
```

- Similarly, it’s possible that one process can be receiving multiple messages with different tags from another process
- The receiving process doesn’t know the order in which the messages will be sent.
- For this circumstance, MPI provides the special constant MPI_ANY_TAG that can be passed to the tag argument of MPI_Recv

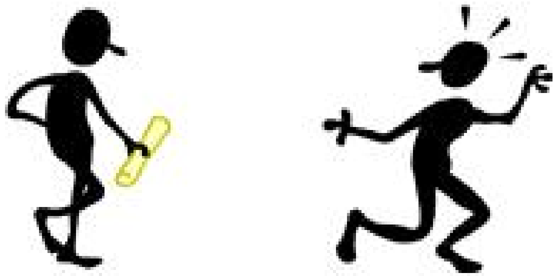
Message matching - remarks

- Only a receiver can use a wildcard argument. Senders must specify a process rank and a nonnegative tag. Thus, MPI uses a “push” communication mechanism rather than a “pull” mechanism

- There is no wildcard for communicator arguments; both senders and receivers must always specify communicators.

Receiving messages

- A receiver can get a message without knowing:
  - the amount of data in the message,
  - the sender of the message, (MPI_ANY_SOURCE)
  - or the tag of the message (MPI_ANY_TAG)



Receiving messages

- A receiver can get a message without knowing:
    - the amount of data in the message,
    - the sender of the message, (MPI_ANY_SOURCE)
    - or the tag of the message (MPI_ANY_TAG)

- So how can the receiver find out these values?

status_p argument

How receiver finds out the sender, tag if they are not needed by the receiver

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

MPI_Status*

MPI_Status* status;

status.MPI_SOURCE

status.MPI_TAG

MPI_SOURCE

MPI_TAG

MPI_ERROR

status_p argument

- The MPI type MPI_Status is a struct with at least the three members MPI_SOURCE, MPI_TAG, and MPI_ERROR
- Suppose our program contains the definition

```
MPI_Status status;
```

- Then, after a call to MPI_Recv in which &status is passed as the last argument, we can determine the sender and tag by examining the two members

```
status.MPI_SOURCE
status.MPI_TAG
```

status_p argument

BUT the amount of data that’s been received isn’t stored in a field that’s directly accessible to the application program

However, it can be retrieved with a call to MPI_Get_count

How much data am I receiving?

```
int MPI_Get_count(
        MPI_Status*   status_p  /* in  */,
        MPI_Datatype  type      /* in  */,
        int*          count_p   /* out */);
```

How much data am I receiving?

- For example, suppose that in our call to MPI_Recv, the type of the receive buffer is recv_type and, once again, we passed in &status

- Then the call

```
MPI_Get_count(&status, recv_type, &count)
```

will return the number of elements received in the count argument

Issues with send and receive

- Exact behavior is determined by the MPI implementation.
- MPI_Send (**locally blocking with buffer** copied to internal storage or **block** starts transmission)
  - Behave differently with regard to buffer size, cutoffs and blocking.
- MPI_Recv always blocks until a matching message is received.
  - Non-blocking MPI_Isend and MPI_Irecv, immediate return.
  - Know your implementation; don't make assumptions!

Issues with send and receive

- Exact behavior is determined by the MPI implementation.
- MPI_Send (**locally blocking with buffer** copied to internal storage or **block** starts transmission)
  - Behave differently with regard to buffer size, cutoffs and blocking.

- MPI_Recv always blocks until a matching message is received.
  - Non-blocking MPI_Isend and MPI_Irecv, immediate return.
  - Know your implementation; don't make assumptions!
- MPI requires that messages be **nonovertaking** → if process *q* sends two messages to process *r*, then the first message sent by *q* must be available to *r* before the second one

Pitfalls

- If a process tries to receive a message (MPI_Recv) and there's no matching send, then the process will block forever
  - The process will **hang**

- Developers:
  - need to be sure that every receive has a matching send
  - need to be very careful that there are no inadvertent mistakes in calls to MPI_Send and MPI_Recv

Pitfalls

- Examples:
  - tags don’t match
  - the rank of the destination process is the same as the rank of the source process
  - call to MPI_Send blocks and there’s no matching receive → sending process hangs
  - call to MPI_Send is buffered and there’s no matching receive → message is lost