Parallel Programming with MPI

Meifeng Lin

mlin@bnl.gov

BROOKHAVEN NATIONAL LABORATORY

a passion for discovery

U.S. DEPARTMENT OF ENERGY | Office of Science

Plan of the Tutorial

- Morning
  - 9:00 -10:30 Introduction to MPI
    - Example 1: Hello World
    - Exercise 1: Running your first MPI program
  - 10:45 -12:00 Point-to-Point Communications
    - Example 2: Greetings
    - Exercise 2: Trapezoidal rule
- Afternoon
  - 13:00 – 15:30 Collective Communications
    - Example 3: Vector dot product
    - Exercise 3: Rewrite Trapezoidal rule
    - Example 4: Matrix Vector multiplication
    - Exercise 4: Matrix Matrix multiplication
  - 15:45 – 17:00 Other Topics in MPI

Disclaimer

- This is not a parallel algorithm course.
  - I will not talk about the best way to parallelize the problems.
- This is not a programming language course.
  - I will not talk about the best way to write the C programs.
- All the examples and exercises are for demonstration purposes only.
  - They do not necessarily use the most efficient implementations.
- I will focus on the basic MPI functions and how to use them in parallel programming.

Introduction to MPI

What is MPI?

- MPI = Message Passing Interface
  - It is a standard for implementing communication between compute nodes/processes in parallel programming.
  - It defines the syntax and semantics of a core of library routines.

- MPI is NOT a new programming language.
  - It merely consists of a library of definitions and functions that can be used in C (or Fortran, Python, …)

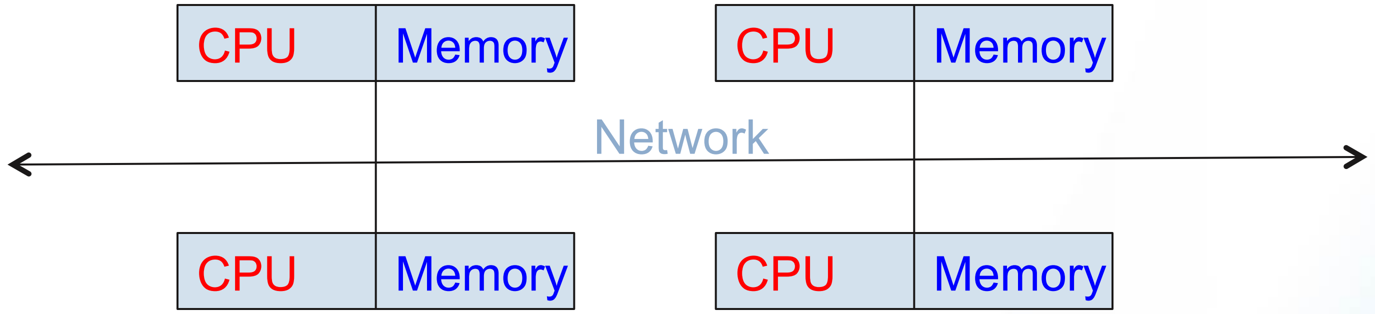
- MPI is designed to be efficient, flexible, practical and portable.

- The reports of the MPI standard can be found on MPI Forum:

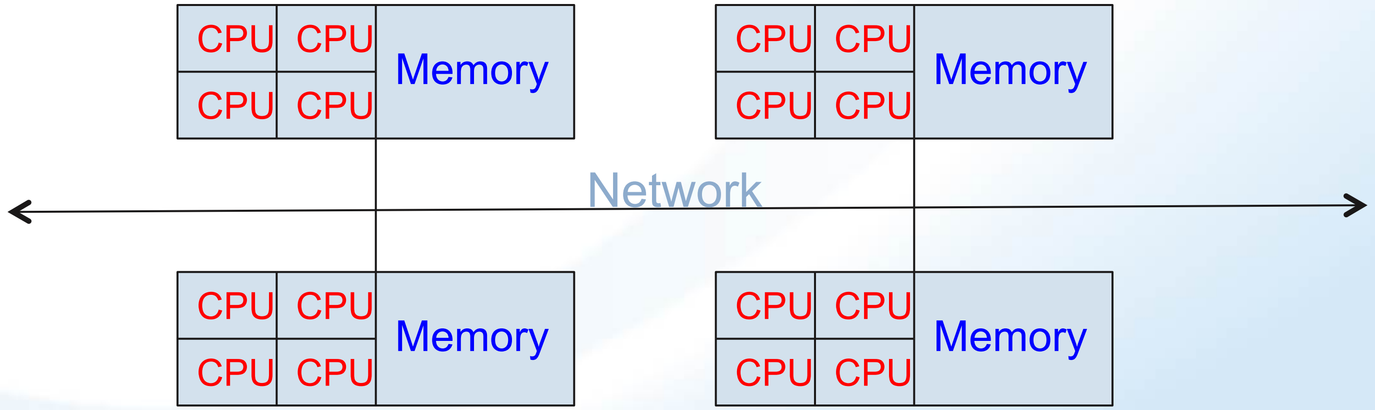
  http://www.mpi-forum.org/

Parallel Systems

- Distributed-memory architectures.



- Shared-memory/distributed-memory hybrid systems.



MPI in Shared-Memory and Hybrid Systems

- On shared-memory systems, MPI provides the internode message passing routines, e.g., **one MPI process per node**.

- On hybrid systems, there are two ways to configure MPI:

  ✓ Use MPI for both inter-node and intra-node communications, e.g. **one MPI process per core**.

  ✓ Use MPI for inter-node communication, and use OpenMP for the intra-node work-sharing. In this case, we will run **one MPI process per node**. (MPI/OpenMP hybrid programming.)

  ✓ Most current HPC systems are hybrid systems.

Evolution of MPI

- MPI-1.x:
  - The goal was to develop a widely used standard for writing message-passing programs.
  - Initially over 40 organizations participated in the discussion.
  - The final report of Version 1.0 was completed in 1994.

- MPI-2.x:
  - Contains corrections and extensions to MPI-1.x.
  - Focused on process creation and management, one-sided communications, extended collective communications, external interfaces and parallel IO.
  - Version 2.0 published in 1997.

- MPI-3.0:
  - Major extensions to MPI, including nonblocking collectives, new one-sided communication operations and Fortran 2008 bindings.
  - Published in 2012.

Evolution of MPI

Focus of this tutorial

- MPI-1.x:
  - The goal was to develop a widely used standard for writing message-passing programs.
  - Initially over 40 organizations participated in the discussion.
  - The final report of Version 1.0 was completed in 1994.
- MPI-2.x:
  - Contains corrections and extensions to MPI-1.x.
  - Focused on process creation and management, one-sided communications, extended collective communications, external interfaces and parallel IO.
  - Version 2.0 published in 1997.
- MPI-3.0:
  - Major extensions to MPI, including nonblocking collectives, new one-sided communication operations and Fortran 2008 bindings.
  - Published in 2012.

Different MPI Implementations

- Open-source Implementations (free to download):
  - MPICH www.mpich.org
    - MPICH, MPICH2
  - MVAPICH mvapich.cse.ohio-state.edu
    - MVAPICH, MVAPICH2
  - OpenMPI www.open-mpi.org

- Vendor-specific Implementations (usually not free):
  - Intel MPI
  - HP MPI
  - …

- You can compile your code with different MPI implementations, and compare the performance between them.

Syntax of MPI Functions

- MPI functions, data types and predefined constants all begin with MPI_
- MPI functions begin with a capital letter, followed by lower-case letters. Examples:
  - MPI_Init(int *argc, char **argv[]),
  - MPI_Finalize()
  - MPI_Comm_rank(MPI_Comm comm, int *rank)
- Predefined MPI constant names are all capitalized. Examples:
  - MPI_COMM_WORLD
  - MPI_DOUBLE

General MPI Program Structure

```
#include <mpi.h>
/* other include files */
#include …
int main(int argc, char *argv[]){
          /*serial code*/
          ……
          /* parallel MPI code begins*/
          MPI_Init(&argc, &argv);
          ……
          MPI_Finalize()
          /* parallel MPI code ends here */
          return 0;
}
```

MPI Communicators

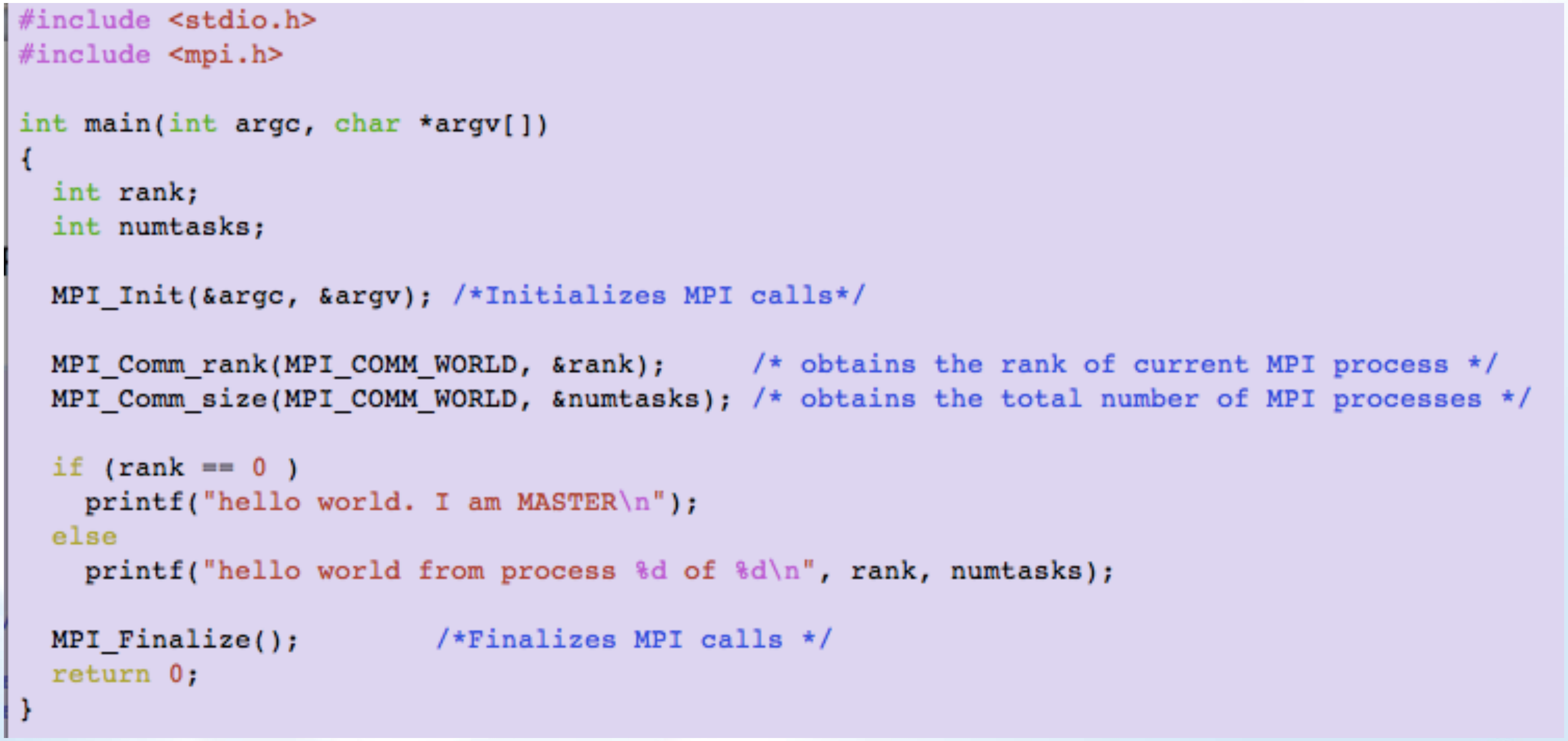
- A communicator is a collection of processes that can send messages to each other.
- MPI_COMM_WORLD is a predefined communicator which consists of all the processes running when the program begins execution.
- For this tutorial, we will only use MPI_COMM_WORLD.

Basic MPI Functions

- int MPI_Init(int *argc, char ***argv)
  - Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program.
- int MPI_Finalize(void)
  - Terminates the MPI execution environment. It must be the last MPI routine called in an MPI program. No other MPI routines may be called after it.
- int MPI_Comm_size(MPI_Comm comm, int *size)
  - Determines the total number of processes in the specified communicator, such as MPI_COMM_WORLD.
- int MPI_Comm_rank(MPI_Comm comm, int *rank)
  - Returns the rank of the calling MPI process with the specified communicator. Each MPI process is assigned a unique integer value between 0 and (number of processes – 1)
- int MPI_Abort(MPI_Comm comm, int code)
  - Terminates the MPI processes associated with the communicator *comm*, and returns the error code specified in *code.*

Hello World

- A sample hello world program with MPI
- Note: no communications yet.



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

int main(int argc, char *argv[])
{
  int rank;
  int numtasks;

  MPI_Init(&argc, &argv); /*Initializes MPI calls*/

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);      /* obtains the rank of current MPI process */
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks); /* obtains the total number of MPI processes */

  if (rank == 0 )
    printf("hello world. I am MASTER\n");
  else
    printf("hello world from process %d of %d\n", rank, numtasks);

  MPI_Finalize();          /*Finalizes MPI calls */
  return 0;
}
```

SPMD Parallel Programming

- Note that in the previous **Hello World** example, while we do not send different programs to different processes, the code segments different processes execute are actually slightly different.
- We differentiate the data with the process ranks.
- This technique is called Single Program Multiple Data (SPMD) parallel programming.
- It is a common parallel programming technique in MIMD (Multiple Instruction Multiple Data) systems.

Compiling and Running MPI Programs

- **mpicc** is a shell script that sets the environment for compiling an MPI program using GNU-C compiler
- **mpirun** or **mpiexec** executes the MPI program

- On hpc set environment variables using **module**
  - **module avail** (modules available on the system)
  - **module list** (modules currently loaded for you)
  - **module load <module name>** (load a new module)
- We will use mvapich2 implementation. At terminal prompt, run
  - **module load mvapich2**

- Check we have the **mpicc** and **mpirun** in the path:
  - which mpicc
  - which mpirun

Compiling and Running MPI Programs

- To compile an MPI program program.c
  - mpicc program.c -o program.x
- To run an MPI program
  - mpirun -np 8 program.x
  - This will run program.x with 8 processes on the head node.
- *Note: different systems may be configured differently. Be sure to check the system documentation or user guide for MPI commands and options in your own work.*

Exercise 1: Hello World

- Obtain the source code helloWorld.c, Makefile and PBS batch script from GitHub:
  - On hpc, run

```
> git clone https://github.com/meifeng/MPI.git
```

  - This will create a directory **MPI** in your current work directory. You should have **1_HelloWorld** subdirectory as well.

- Compile the HelloWorld program using the provided Makefile

```
> make
```

- Run the executable:

```
> mpirun -np <number of nodes> helloWorld.x
```

Running Batch Jobs

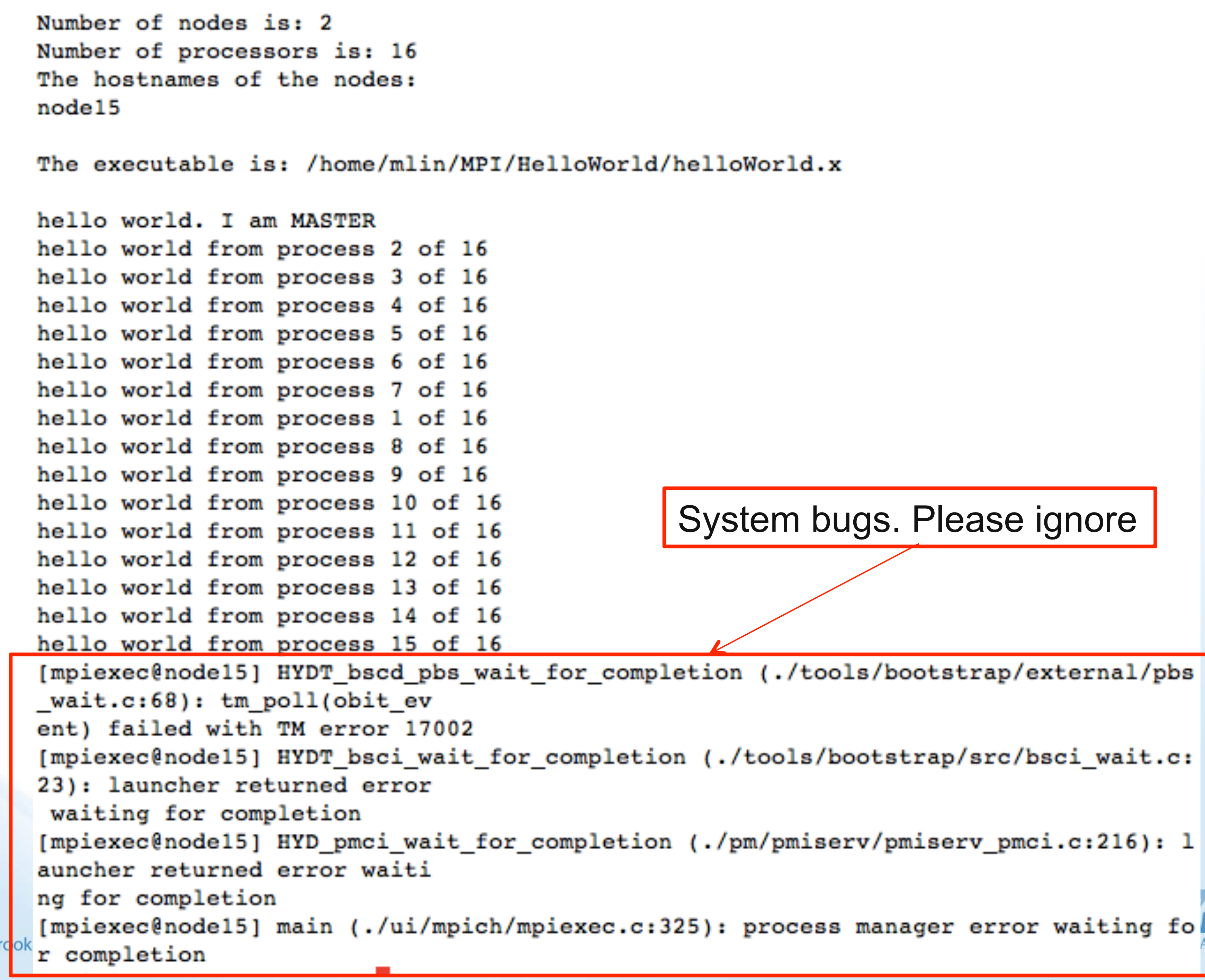
- Executing “mpirun -np 8 program.x” directly from your terminal will run the program on the host node.
- PBS scheduler can automatically distribute the jobs to the available nodes.
- PBS batch jobs are controlled by PBS scripts:
- Example:

`#PBS -m e`	➔Notify when job ends (optional)
`#PBS -M mlin@bnl.gov`	➔Email address to get the notifications (optional)
`#PBS -S /bin/bash`	➔Default shell
`#PBS -l nodes=2:ppn=8`	➔Number of nodes and processors per node requested
`#PBS -l walltime=00:05:00`	➔Wall-clock time requested
`#PBS -j oe`	➔Joining stdout and stderr in the output (optional)
`#PBS -N mpi`	➔Name of the job
`#PBS -o $PBS_JOBID.out`	➔Name of the output file

```
NODECOUNT=`sort -u ${PBS_NODEFILE} | wc -l`
PROCCOUNT=`sort ${PBS_NODEFILE} | wc -w`
WORK=${PBS_O_WORKDIR}
EXE=${WORK}/helloWorld.x
mpirun -np ${PROCCOUNT} $EXE
```

Running Batch Jobs

- Now use batch.example in the directory.
- Change the scripts to suit your needs
- Run
  - qsub batch.example
- This will submit a job to the PBS queue. Your job will be executed when there are sufficient resources available.
- You can check the status of your jobs by running
  - qstat
- Running “**qstat –n**” will show you the nodes you are currently running on.



```
Number of nodes is: 2
Number of processors is: 16
The hostnames of the nodes:
node15

The executable is: /home/mlin/MPI/HelloWorld/helloWorld.x

hello world. I am MASTER
hello world from process 2 of 16
hello world from process 3 of 16
hello world from process 4 of 16
hello world from process 5 of 16
hello world from process 6 of 16
hello world from process 7 of 16
hello world from process 1 of 16
hello world from process 8 of 16
hello world from process 9 of 16
hello world from process 10 of 16
hello world from process 11 of 16
hello world from process 12 of 16
hello world from process 13 of 16
hello world from process 14 of 16
hello world from process 15 of 16
[mpiexec@node15] HYDT_bscd_pbs_wait_for_completion (./tools/bootstrap/external/pbs
_wait.c:68): tm_poll(obit_ev
ent) failed with TM error 17002
[mpiexec@node15] HYDT_bsci_wait_for_completion (./tools/bootstrap/src/bsci_wait.c:
23): launcher returned error
 waiting for completion
[mpiexec@node15] HYD_pmci_wait_for_completion (./pm/pmiserv/pmiserv_pmci.c:216): l
auncher returned error waiti
ng for completion
[mpiexec@node15] main (./ui/mpich/mpiexec.c:325): process manager error waiting fo
r completion
```

System bugs. Please ignore

Point-to-Point Communications

Point-to-Point Communications

- MPI point-to-point operations typically involve message passing between two, and only two, different MPI processes. One process is performing a send operation, and the other process is performing a matching receive operation.

- There are different types of send and receive routines used for different purposes. For example:
  - Synchronous send
  - Blocking send/blocking receive
  - Non-blocking send/non-blocking receive
  - Buffered send
  - “Ready” send

Basic Blocking Point-to-Point Message Passing

```
int MPI_Send (void *          buffer,         /* pointer to data sent        */
              int             count,          /* length of data              */
              MPI_Datatype    datatype,       /* message data type           */
              int             destination,    /* destination  rank           */
              int             tag,            /* non-negative integer        */
              MPI_Comm        communicator)   /* communicator involved       */
```

- Routine returns only after the application buffer in the sending process is free for reuse.

```
int MPI_Recv (void *          buffer,         /* pointer to data received    */
              int             count,          /* length of data              */
              MPI_Datatype    datatype,       /* data type                   */
              int             source,         /* source rank                 */
              int             tag,            /* non-negative integer        */
              MPI_Comm        communicator,   /* communicator involved       */
              MPI_Status *    status)         /* status of receive */
```

- Receive a message and block until the requested data is available in the application buffer in the receiving process.

MPI Datatypes

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed 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	

Note: MPI also allows user-defined data types.

Exercise 2: Greetings

- Let’s modify the Hello World program so that Process i > 0 sends its rank to Process 0.
- Use MPI_Send and MPI_Recv.
- Compile and run on hpc.csc.bnl.gov.

Exercise 2: Example Code

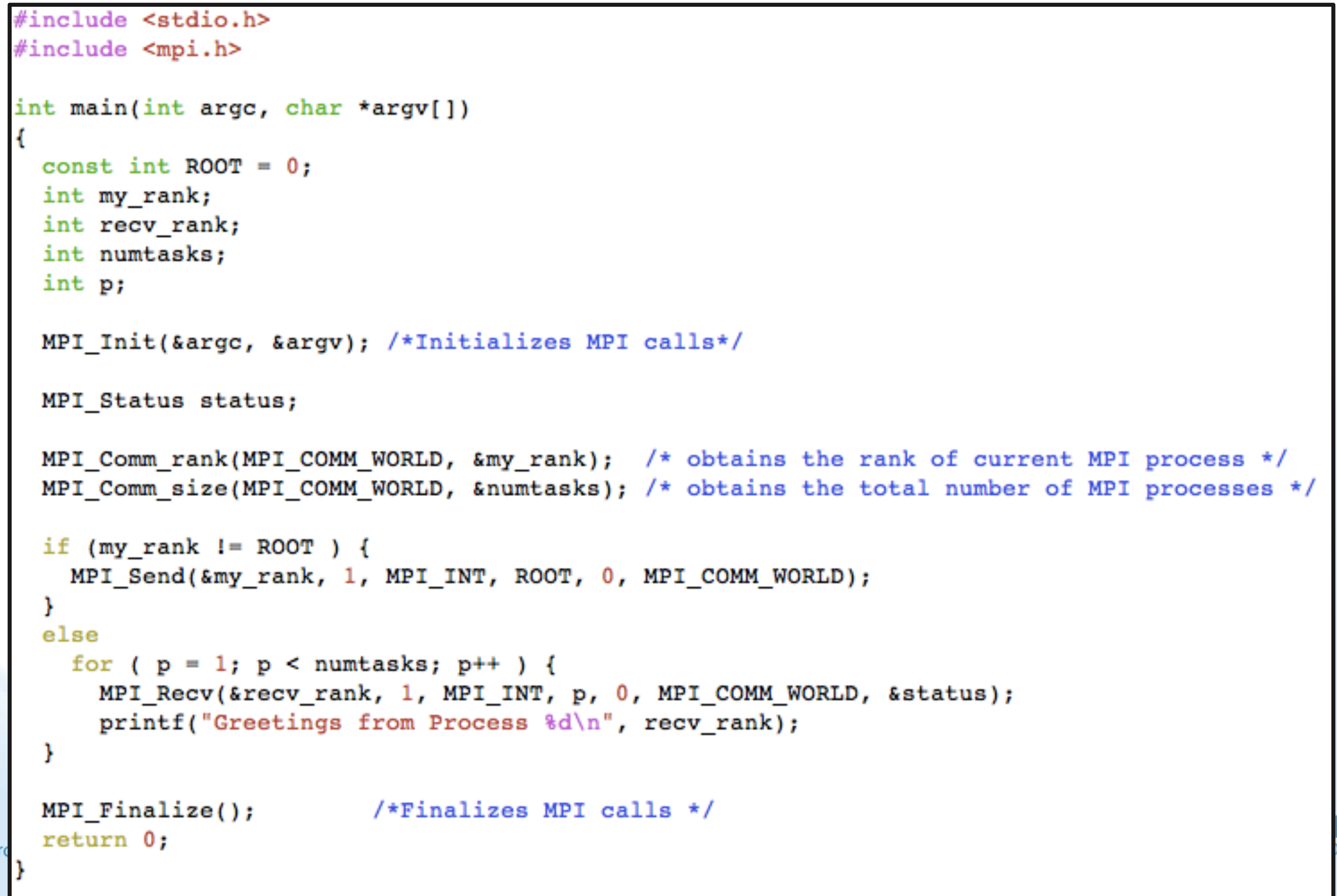
- Obtain the example code **2_Greetings** from GitHub
  - From your MPI directory, run

    ```
    > git pull
    ```

  - Or point your browser to
    http://github.com/meifeng/MPI

- Compile and run the code either in interactive mode, or in batch mode.

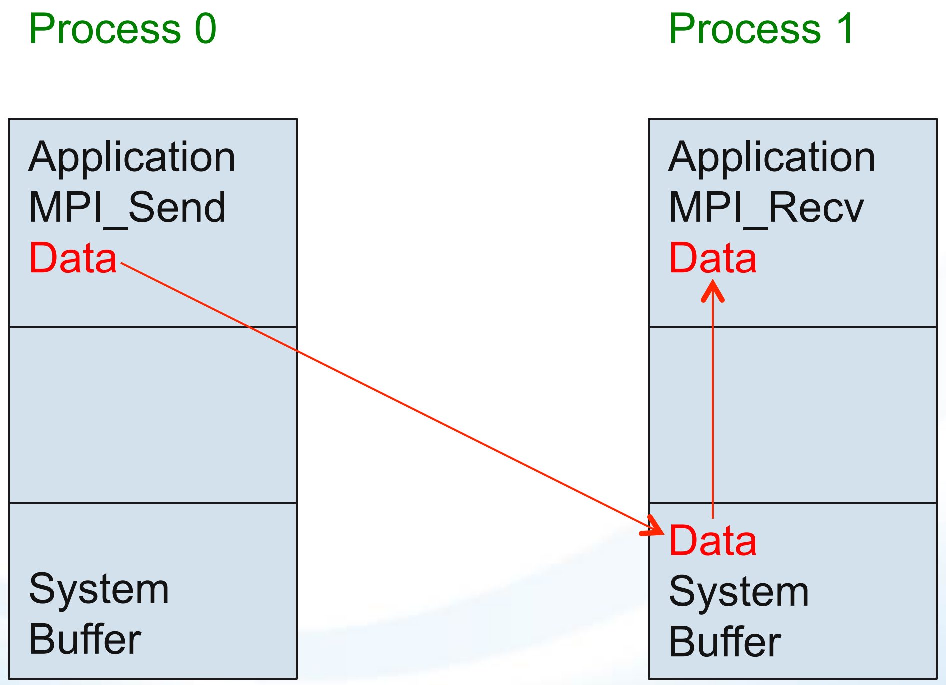
Message Passing in Greetings



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

int main(int argc, char *argv[])
{
  const int ROOT = 0;
  int my_rank;
  int recv_rank;
  int numtasks;
  int p;

  MPI_Init(&argc, &argv); /*Initializes MPI calls*/

  MPI_Status status;

  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);  /* obtains the rank of current MPI process */
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks); /* obtains the total number of MPI processes */

  if (my_rank != ROOT ) {
    MPI_Send(&my_rank, 1, MPI_INT, ROOT, 0, MPI_COMM_WORLD);
  }
  else
    for ( p = 1; p < numtasks; p++ ) {
      MPI_Recv(&recv_rank, 1, MPI_INT, p, 0, MPI_COMM_WORLD, &status);
      printf("Greetings from Process %d\n", recv_rank);
  }

  MPI_Finalize();         /*Finalizes MPI calls */
  return 0;
}
```

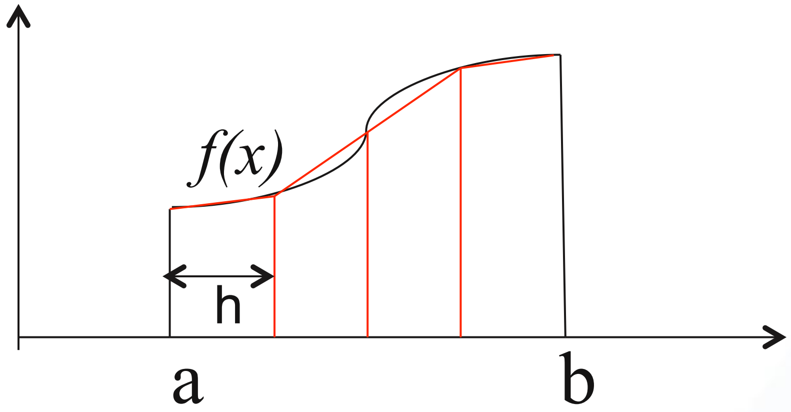
Buffering

- While every process is sending, there is no one receiving. Where do the data go? (Think ball throwing).
- Typically a chunk of memory is reserved as system buffer.
- The message resides in the system buffer while waiting for the receiving node to retrieve.



Numerical Integration: Trapezoidal Rule

- A finite integral in the region [a,b] $\int_a^b f(x)\,dx$ is essentially the area enclosed by the function curve and the boundaries.

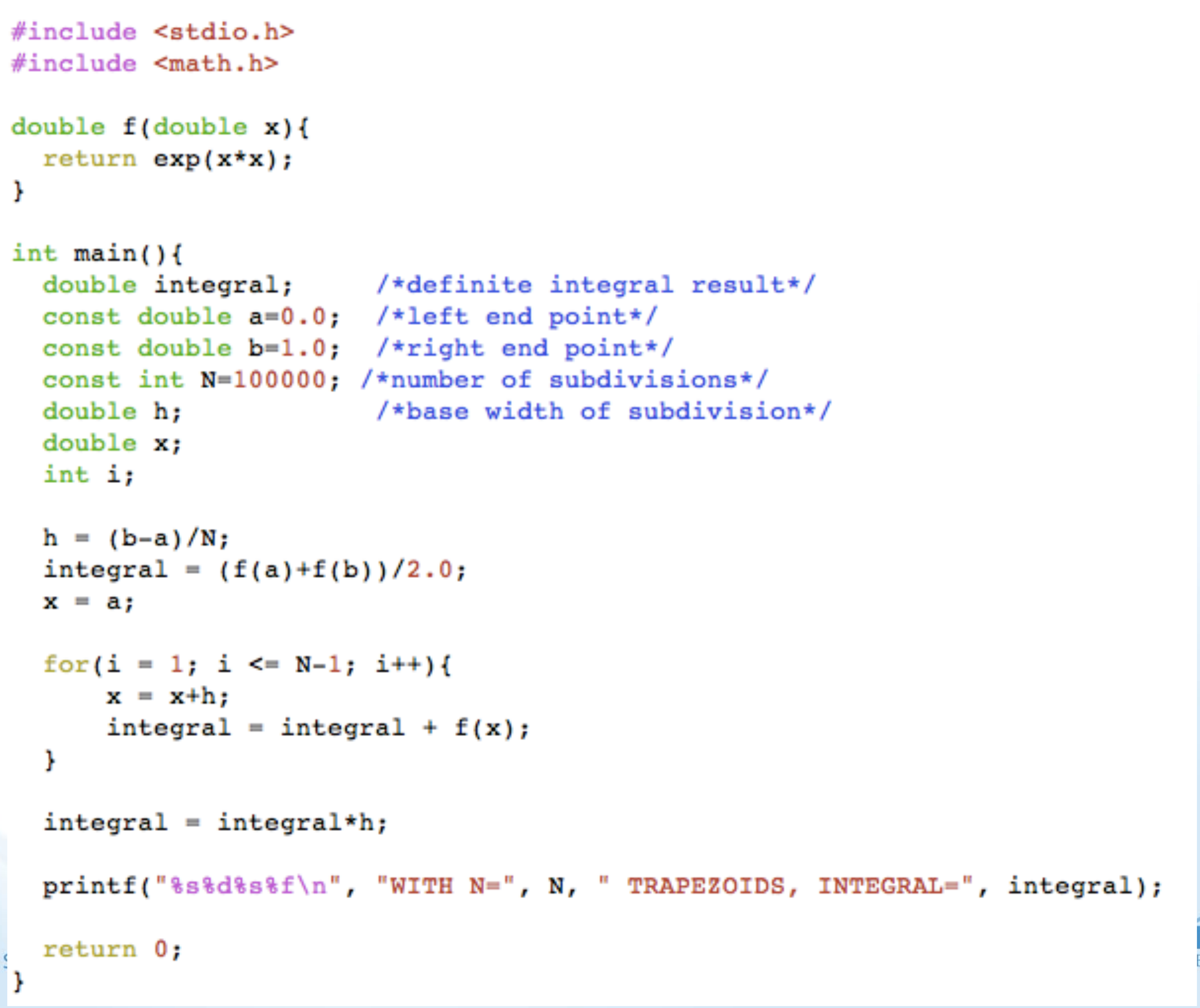


- It can be approximated by calculating the areas of the trapezoids that evenly divide the whole area.

$$\int_a^b f(x)dx \approx \frac{h}{2}\big(f(a)+f(a+h)\big)+\frac{h}{2}\big(f(a+h)+f(a+2h)\big)+\ldots+\frac{h}{2}\big(f(a+(N-1)h)+f(b)\big)$$

$$=\left[\frac{1}{2}\big(f(a)+f(b)\big)+\sum_{j=1}^{N-1} f(a+jh)\right]\cdot h$$

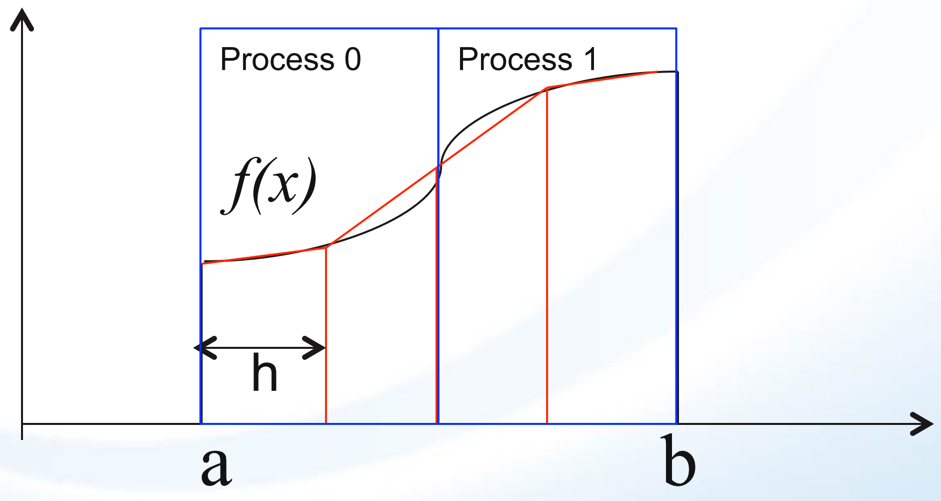
Serial Trapezoidal Rule



```c
#include <stdio.h>
#include <math.h>

double f(double x){
  return exp(x*x);
}

int main(){
  double integral;      /*definite integral result*/
  const double a=0.0;   /*left end point*/
  const double b=1.0;   /*right end point*/
  const int N=100000; /*number of subdivisions*/
  double h;             /*base width of subdivision*/
  double x;
  int i;

  h = (b-a)/N;
  integral = (f(a)+f(b))/2.0;
  x = a;

  for(i = 1; i <= N-1; i++){
      x = x+h;
      integral = integral + f(x);
  }

  integral = integral*h;

  printf("%s%d%s%f\n", "WITH N=", N, " TRAPEZOIDS, INTEGRAL=", integral);

  return 0;
}
```

Parallel Trapezoidal Rule

- Before we can write a parallel version of the trapezoidal rule, we need to decide how we want to share the workload among different processes. ➔ Data mapping.
- The simplest approach is perhaps to distribute the data evenly among the processes, and each process runs essentially the same program on its share of the data. ➔Data-parallel programming.
- We can
  - Assign a subinterval of [a, b] to each process.
  - Have that process estimate the integral of *f* over the subinterval.
  - Add up the local processes' calculations to get the final integral.



Exercise 3: Trapezoid

- Parallelize the Trapezoidal rule program using MPI_Send and MPI_Recv, assuming N can be divided by the number of processes evenly.
- Send the final result to Process 0 and print out the result at stdout.
- Compile and run the program on hpc.csc.bnl.gov
- Hint:
  - Each process is essentially calculating its own integral, with different integration lower and upper bounds
  - Your main job is to use the process ranks to determine the bounds and integration steps for each process.

Exercise 3: Example Code

- Obtain the example code **3_Trapezoid** from GitHub
  - From your MPI directory, run

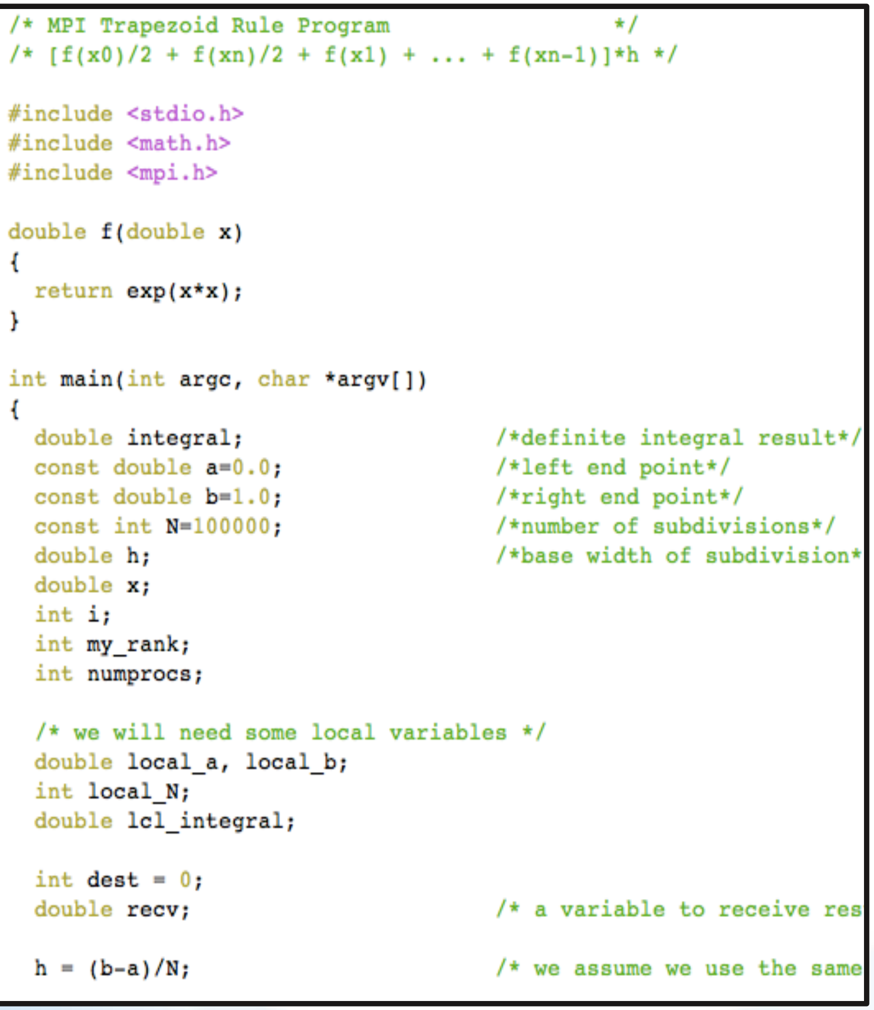
    ```
    > git pull
    ```

  - Or point your browser to
    http://github.com/meifeng/MPI

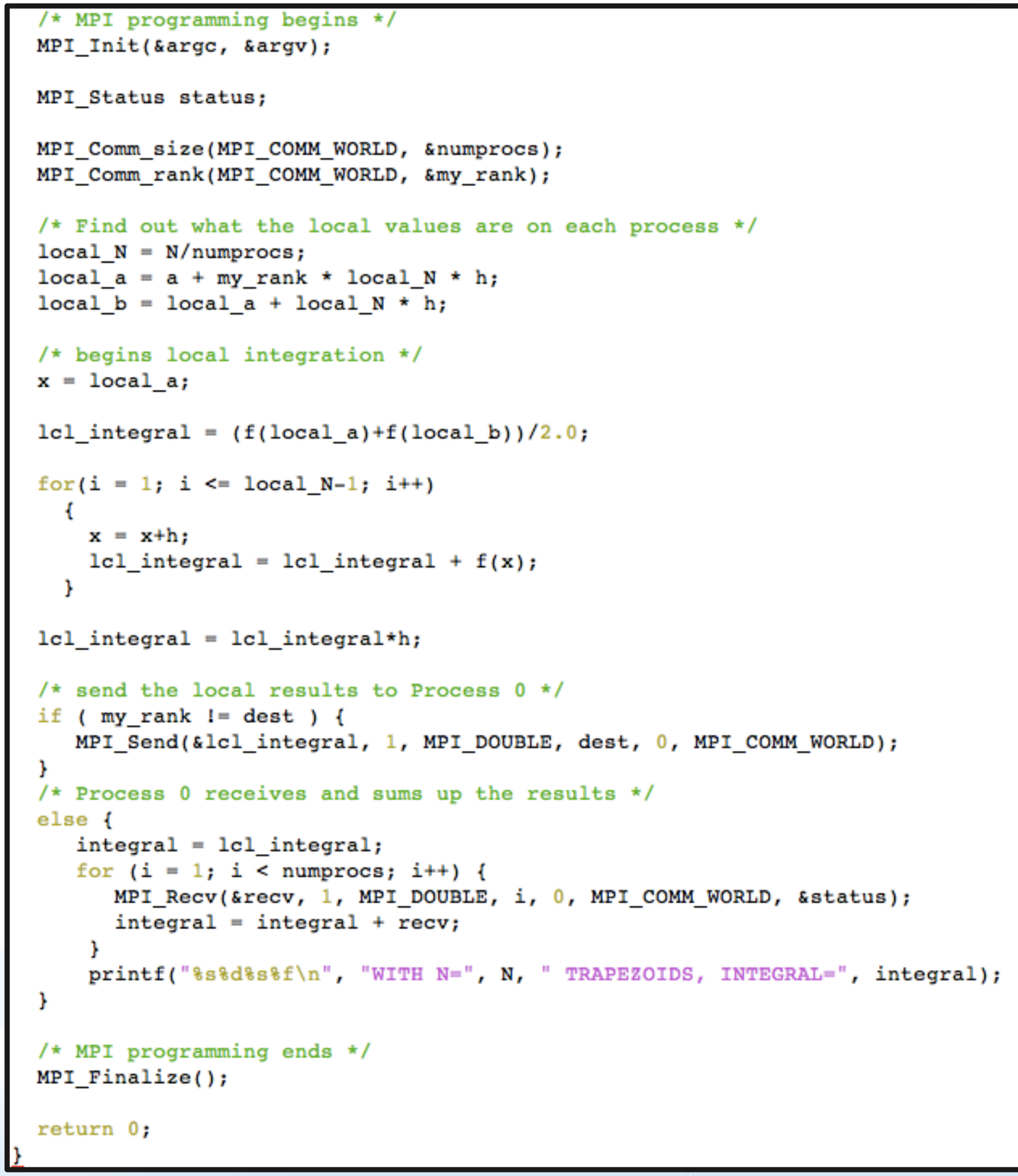
- Compile and run the code either in interactive mode, or in batch mode.
  - You can compile the serial version of the code using

    ```
    > make serial
    ```

Parallel Trapezoidal Rule



```
/* MPI Trapezoid Rule Program                    */
/* [f(x0)/2 + f(xn)/2 + f(x1) + ... + f(xn-1)]*h */

#include <stdio.h>
#include <math.h>
#include <mpi.h>

double f(double x)
{
  return exp(x*x);
}

int main(int argc, char *argv[])
{
  double integral;                  /*definite integral result*/
  const double a=0.0;               /*left end point*/
  const double b=1.0;               /*right end point*/
  const int N=100000;               /*number of subdivisions*/
  double h;                         /*base width of subdivision*
  double x;
  int i;
  int my_rank;
  int numprocs;

  /* we will need some local variables */
  double local_a, local_b;
  int local_N;
  double lcl_integral;

  int dest = 0;
  double recv;                      /* a variable to receive res

  h = (b-a)/N;                      /* we assume we use the same
```



```
 /* MPI programming begins */
 MPI_Init(&argc, &argv);

 MPI_Status status;

 MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
 MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

 /* Find out what the local values are on each process */
 local_N = N/numprocs;
 local_a = a + my_rank * local_N * h;
 local_b = local_a + local_N * h;

 /* begins local integration */
 x = local_a;

 lcl_integral = (f(local_a)+f(local_b))/2.0;

 for(i = 1; i <= local_N-1; i++)
   {
     x = x+h;
     lcl_integral = lcl_integral + f(x);
   }

 lcl_integral = lcl_integral*h;

 /* send the local results to Process 0 */
 if ( my_rank != dest ) {
   MPI_Send(&lcl_integral, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
 }
 /* Process 0 receives and sums up the results */
 else {
   integral = lcl_integral;
   for (i = 1; i < numprocs; i++) {
     MPI_Recv(&recv, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);
     integral = integral + recv;
   }
   printf("%s%d%s%f\n", "WITH N=", N, " TRAPEZOIDS, INTEGRAL=", integral);
 }

 /* MPI programming ends */
 MPI_Finalize();

 return 0;
}
```

Other Blocking Message Passing Routines

- **MPI_Ssend**(…)
  - Synchronous blocking send.
  - In synchronous blocking mode, a send won't complete until a matching receive has been posted and the matching receive has begun reception of the data.
- **MPI_Bsend**(…)
  - Buffered blocking send, with user-provided buffering.
- **MPI_Rsend(…)**
  - Blocking ready send.
  - Send may be started *only if* the matching receive has begun.
- **MPI_Sendrecv(…)**
  - Combines sending of a message to one destination, and receiving of a message from another process.

Non-Blocking Message Passing

- Non-blocking message passing **returns immediately**, regardless of the state of the message sending or retrieval.
- It merely sends a request to the MPI library to perform the operation when it is able to.
- It is unsafe to modify the user variable space (the application buffer) until you are sure that MPI has completed the requested operation. Can use **MPI_Wait** or **MPI_Test** to check.
- Non-blocking message passing is very useful in applications with overlapping communication and computation to enhance the program performance.

Non-Blocking Message Passing

MPI_Isend(void *data, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)

- Starts a nonblocking send and returns immediately.

MPI_Irecv(void *data, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Request *request)

- Starts a nonblocking receive and returns immediately.

MPI_Wait(MPI_Request *request, MPI_Status *status)

- Completes any nonblocking operation.
- The *request* parameter corresponds to the *request* parameter in MPI_Isend or MPI_Irecv.
- It blocks until the operation identified by *request* completes.

Collective Communications

Collective Communications

- A communication pattern that involves all the processes in a communicator is a collective communication.

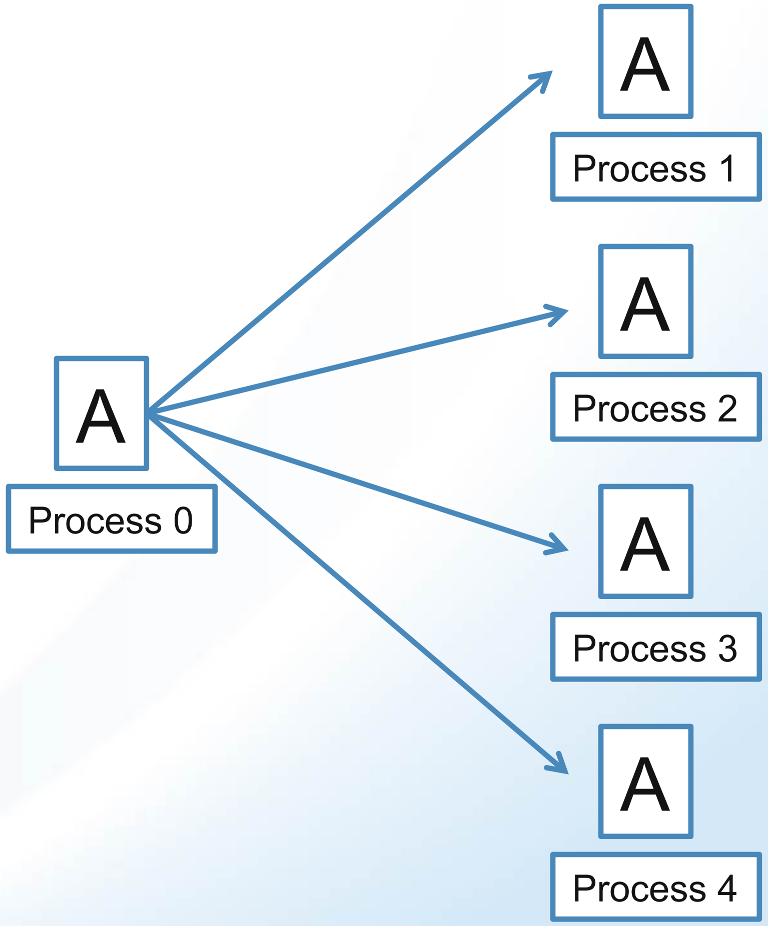
- A collective communication usually involves more than two processes.

- Types of collective communications
  - Broadcast
  - Reduce
  - Allreduce
  - Gather and Scatter
  - Allgather

Broadcast

```
int MPI_Bcast (void*              message,
               int                count,
               MPI_Datatype       type,
               int                root,
               MPI_Comm           comm)
```

- Sends *message* from *root* to all the processes in communicator *comm*.
- Must be called by all processes in the communicator *comm* with the same *root.*
- *count* and *type* should be the same on all processes in the communicator.
- A sequence of collective communications on distinct processes will be matched in the order in which they're executed.



Reduce

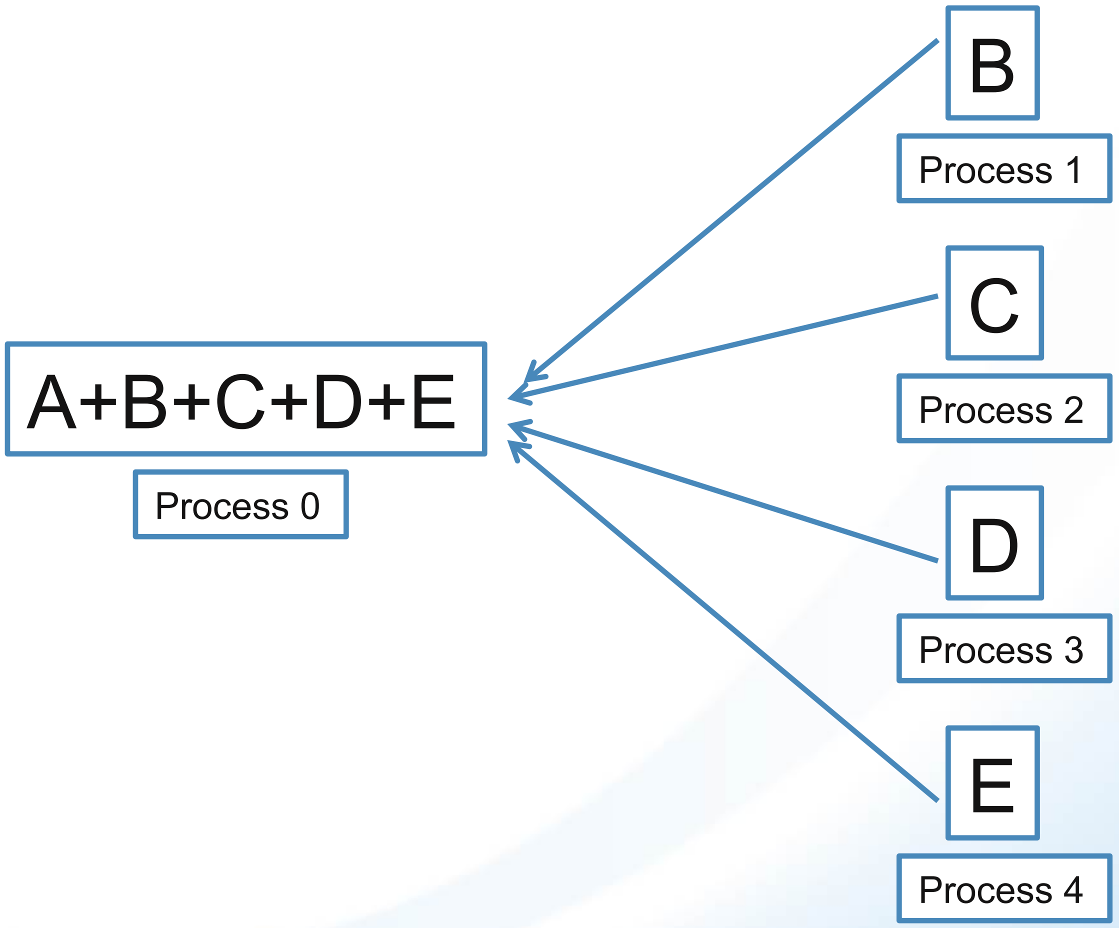
```
int MPI_Reduce( void*          operand,
                void*          result,
                int            count,
                MPI_Datatype   type,
                MPI_Op         operator,
                int            root,
                MPI_Comm       comm)
```

- Performs binary operation with *operator* on *operand*, and sends the *result* to *root.*
- It is run on all processes in the communicator *comm*.
- *Count, type, operator* and *root* must be the same on each process.

MPI_Op	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and its location
MPI_MINLOC	Minimum and its location

Reduce

- For example, if MPI_Op is MPI_SUM



Example 4: Vector Dot Product

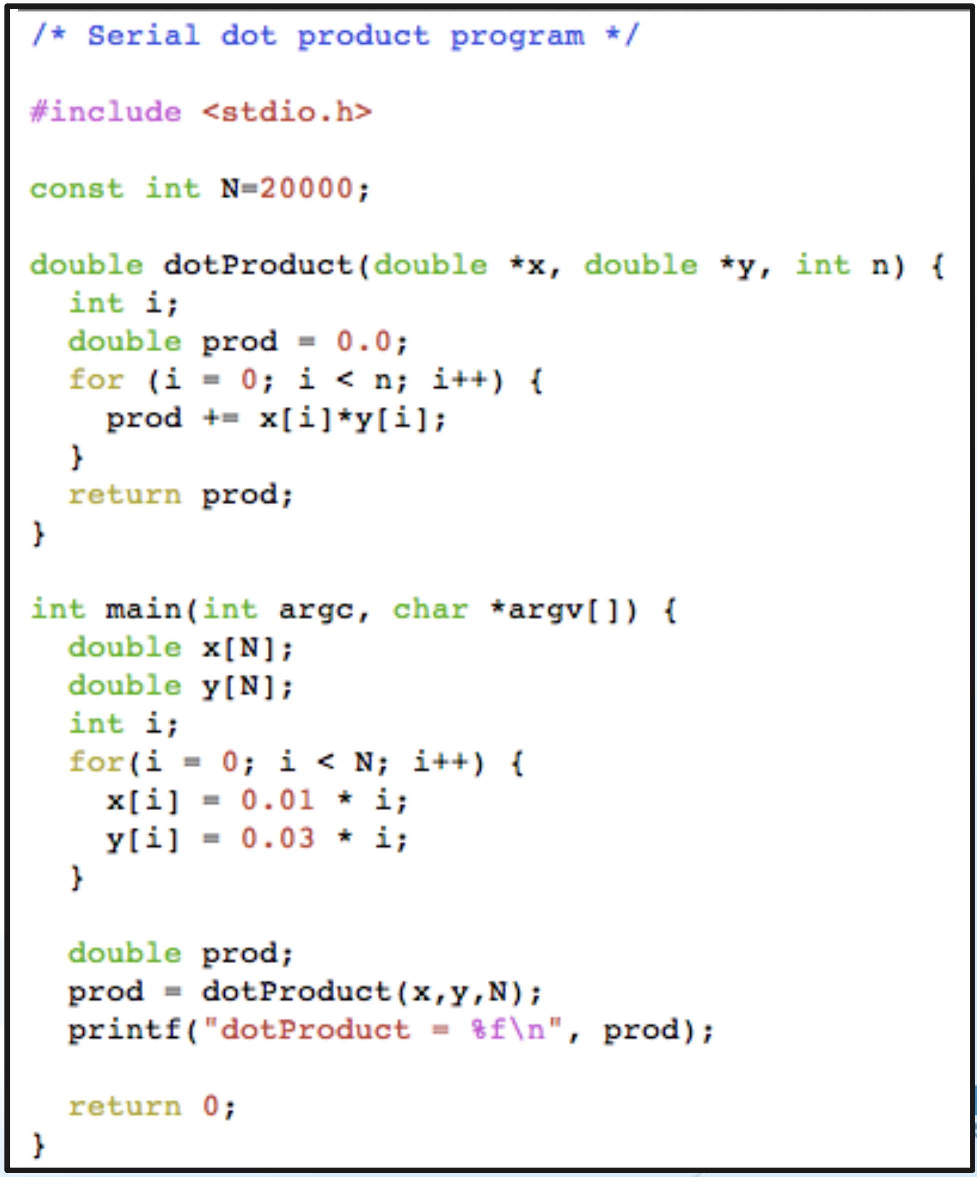
- The dot product of two vectors **x** and **y** is defined as

$$x \cdot y = \sum_i x_i y_i$$

- On the right is the serial version.

- How do we parallelize it?

4_DotProduct available on GitHub: http://github.com/meifeng/MPI



```
/* Serial dot product program */

#include <stdio.h>

const int N=20000;

double dotProduct(double *x, double *y, int n) {
  int i;
  double prod = 0.0;
  for (i = 0; i < n; i++) {
    prod += x[i]*y[i];
  }
  return prod;
}

int main(int argc, char *argv[]) {
  double x[N];
  double y[N];
  int i;
  for(i = 0; i < N; i++) {
    x[i] = 0.01 * i;
    y[i] = 0.03 * i;
  }

  double prod;
  prod = dotProduct(x,y,N);
  printf("dotProduct = %f\n", prod);

  return 0;
}
```

Parallel Vector Dot Product

- First we need to consider how to map the vectors onto different processors.
- It is natural to divide the vectors into **blocks** of data, with each process taking one block.

	x	y
P0	x_0 x_1	y_0 y_1
P1	x_2 x_3	y_2 y_3
	…	…
Pk-1	x_{n-2} x_{n-1}	y_{n-2} y_{n-1}

```c
/* Parallel dot product */
#include <mpi.h>
#include <stdio.h>

const int N=20000;

double dotProduct(double *x, double *y, int n) {
  int i;
  double prod = 0.0;
  for (i = 0; i < n; i++) {
    prod += x[i]*y[i];
  }
  return prod;
}

int main(int argc, char *argv[]) {
  int i;
  double prod;
  int my_rank;
  int num_procs;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  int local_N = N / num_procs; //assuming N is totally divisible by num_procs
  double local_x[local_N];
  double local_y[local_N];
  for(i = 0; i < local_N; i++) {
    local_x[i] = 0.01 * (i + my_rank * local_N);
    local_y[i] = 0.03 * (i + my_rank * local_N);
  }
  double local_prod;
  local_prod = dotProduct(local_x,local_y,local_N);
  MPI_Reduce(&local_prod, &prod, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
  if (my_rank == 0) {
    printf("dotProduct = %f\n", prod);
  }
  MPI_Finalize();

  return 0;
}
```

Dealing with extras

Some processes have to do extra work if N is not divisible by the number of processes

```c
/* Parallel dot product */
#include <mpi.h>
#include <stdio.h>

const int N=2000;

double dotProduct(double *x, double *y, int n) {
  int i;
  double prod = 0.0;
  for (i = 0; i < n; i++) {
    prod += x[i]*y[i];
  }
  return prod;
}

int main(int argc, char *argv[]) {
  int i;
  double prod;
  int my_rank;
  int num_procs;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  int local_N = N / num_procs;
  int extra = N % num_procs;
  double local_prod;
```

```c
  if (my_rank == (num_procs - 1)) {
    int new_N = local_N + extra;
    double local_x[new_N];
    double local_y[new_N];
    for(i = 0; i < new_N; i++) {
      local_x[i] = 0.01 * (i + my_rank * local_N);
      local_y[i] = 0.03 * (i + my_rank * local_N);
    }
    local_prod = dotProduct(local_x,local_y,new_N);
  }
  else {
    double local_x[local_N];
    double local_y[local_N];
    for(i = 0; i < local_N; i++) {
      local_x[i] = 0.01 * (i + my_rank * local_N);
      local_y[i] = 0.03 * (i + my_rank * local_N);
    }
    local_prod = dotProduct(local_x,local_y,local_N);
  }

  MPI_Reduce(&local_prod, &prod, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
  if (my_rank == 0) {
    printf("dotProduct = %f\n", prod);
  }
  MPI_Finalize();

  return 0;
}
```

Exercise 5: TrapezoidCollective

- Rewrite the parallel trapezoidal rule program so that
  - Process 0 takes the input values for a, b and N, and propagates these values to all processes using **MPI_Bcast**.
  - Use **MPI_Reduce** to sum up the local results from different processes to Process 0
  - Print the final result on stdout from all processes.

Exercise 5: Example Code

- Obtain the example code **5_TrapezoidCollective** from GitHub
    - From your MPI directory, run

      ```
      > git pull
      ```

    - Or point your browser to http://github.com/meifeng/MPI

- Compile and run the code in interactive mode (since you need to input the parameters).

Output

```
Enter integral lower bound, upper bound and
total integration steps:
0 1 10000
Process 0, WITH N=10000 TRAPEZOIDS,
INTEGRAL=1.462651
Process 1, WITH N=10000 TRAPEZOIDS,
INTEGRAL=0.000000
Process 2, WITH N=10000 TRAPEZOIDS,
INTEGRAL=0.000000
Process 3, WITH N=10000 TRAPEZOIDS,
INTEGRAL=0.000000
Process 4, WITH N=10000 TRAPEZOIDS,
INTEGRAL=0.000000
Process 5, WITH N=10000 TRAPEZOIDS,
INTEGRAL=0.000000
Process 6, WITH N=10000 TRAPEZOIDS,
INTEGRAL=0.000000
Process 7, WITH N=10000 TRAPEZOIDS,
INTEGRAL=0.000000
```

Notice that only process 0 gives a non-zero value.

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

double f(double x){
  return exp(x*x);
}

int main(int argc, char *argv[]){
  const int ROOT = 0;
  double integral;          /*definite integral result*/
  double a;                 /*left end point*/
  double b;                 /*right end point*/
  int    N;                 /*number of subdivisions*/
  double h;                 /*base width of subdivision*/
  double x;
  int i;
  int my_rank;
  int numprocs;

  /* we will need some local variables */
  double local_a, local_b;
  int local_N;
  double lcl_integral;

  /* MPI programming begins */
  MPI_Init(&argc, &argv);
  MPI_Status status;
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  if(my_rank == ROOT) {
    printf("Enter integral lower bound, upper bound and total integration steps: \n");
    scanf("%lf %lf %d", &a, &b, &N);
  }

  MPI_Bcast(&a, 1, MPI_DOUBLE, ROOT, MPI_COMM_WORLD);
  MPI_Bcast(&b, 1, MPI_DOUBLE, ROOT, MPI_COMM_WORLD);
  MPI_Bcast(&N, 1, MPI_INT,    ROOT, MPI_COMM_WORLD);

  h = (b-a)/N;   /* we assume we use the same integration step on all processes */

  /* Find out what the local values are on each process */
  local_N = N / numprocs;
  local_a = a + my_rank * local_N * h;
  local_b = local_a + local_N * h;

  /* begins local integration */
  x = local_a;
  lcl_integral = (f(local_a)+f(local_b))/2.0;

  for(i = 1; i <= local_N-1; i++) {
      x = local_a+i*h;
      lcl_integral = lcl_integral + f(x);
  }
  lcl_integral = lcl_integral*h;

  /* Reduce and send result to ROOT */
  MPI_Reduce(&lcl_integral, &integral, 1, MPI_DOUBLE, MPI_SUM, ROOT, MPI_COMM_WORLD);
  //MPI_Allreduce(&lcl_integral, &integral, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

  printf("Process %d, WITH N=%d TRAPEZOIDS, INTEGRAL=%f\n", my_rank, N, integral);
  /* MPI programming ends */
  MPI_Finalize();

  return 0;
}
```

Allreduce

```
int MPI_Allreduce ( void*          operand,
                    void*          result,
                    int            count,
                    MPI_Datatype   type,
                    MPI_Op         operator,
                    MPI_Comm       comm)
```

- Similar to MPI_Reduce.
- But all the processes get the reduced result, hence there is no parameter for a root process.

- This is necessary if the calculation requires the reduced result to proceed.

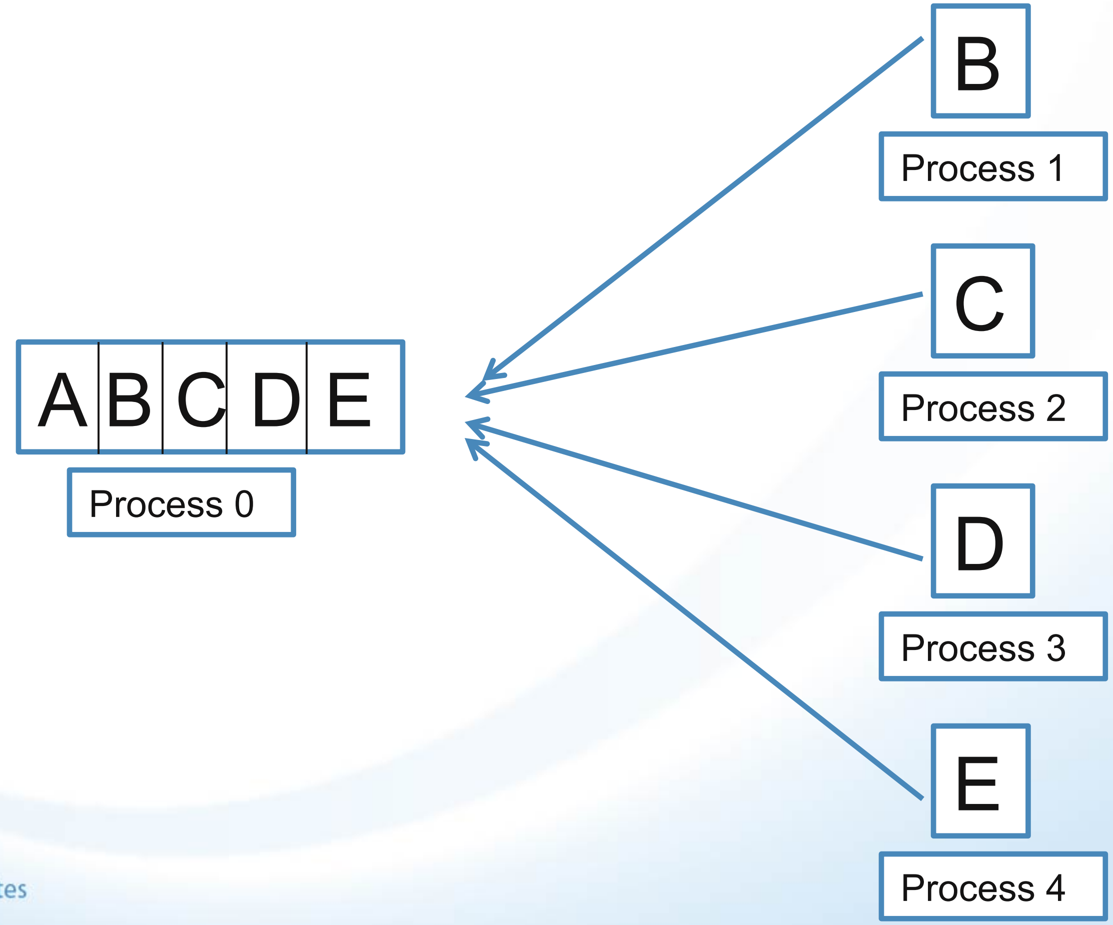
Exercise:

- Rewrite the parallel trapezoidal rule program so that
  - Process 0 takes the input values for a, b and N, and propagates these values to all processes.
  - Use **MPI_Allreduce** to sum up the local results from different processes and print the final result on stdout from all processes.
  - Login to hpc.csc.bnl.gov
  - Compile and run the new program, and see how the output is different.

Gather

Collects values from a group of processes in a communicator to root

```
int MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int recvcount,   MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```



Allgather

Collects data from all processes and makes them available on each process

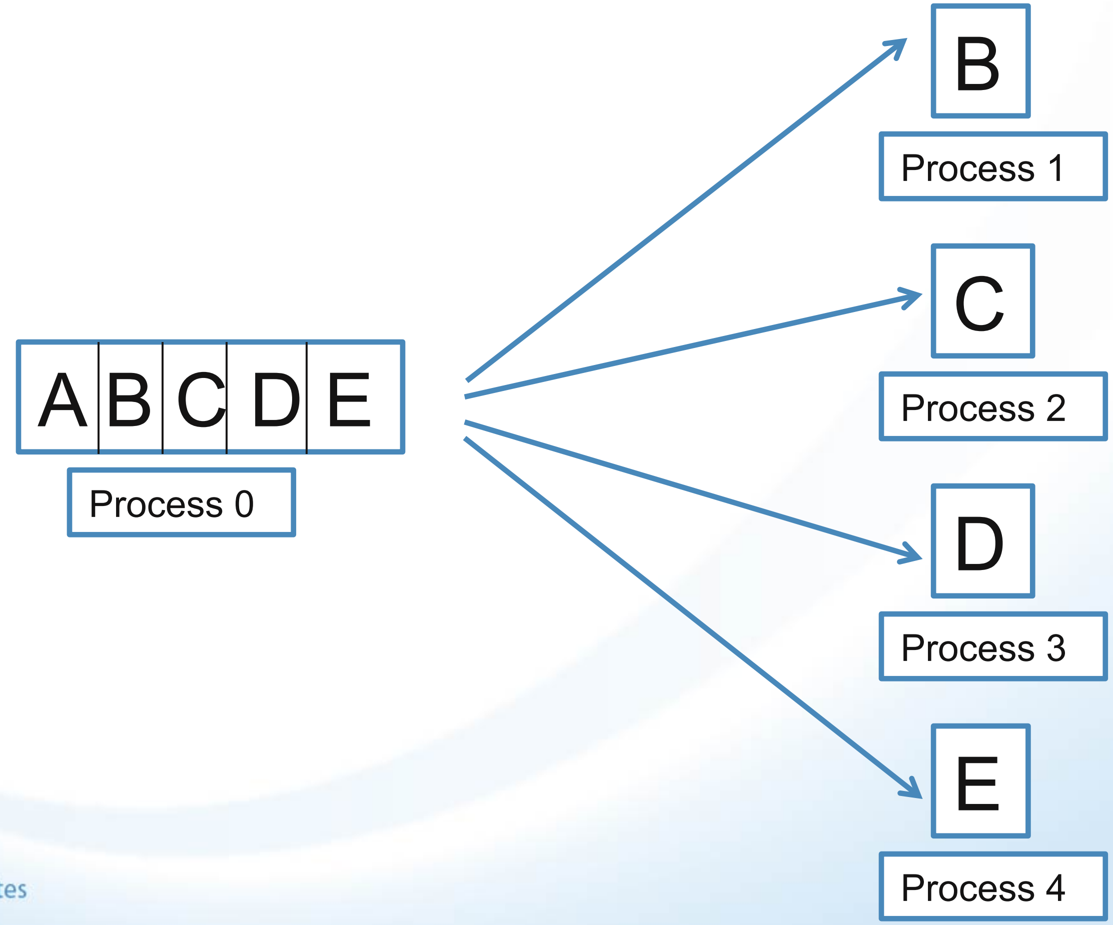
```
int MPI_Allgather (void *sendbuf, int sendcount, MPI_Datatype sendtype,
                   void *recvbuf, int recvcount,   MPI_Datatype recvtype,
                   MPI_Comm comm)
```

- Similar to MPI_Gather, but now all the processes get the collected data.
- Note in both MPI_Gather and MPI_Allgather, both *sendcount* and *recvcount* refer to the number of data blocks **per process**.

Scatter

Sends data from one process to other processes in a communicator

```
int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount,   MPI_Datatype recvtype,
                 int root, MPI_Comm comm)
```



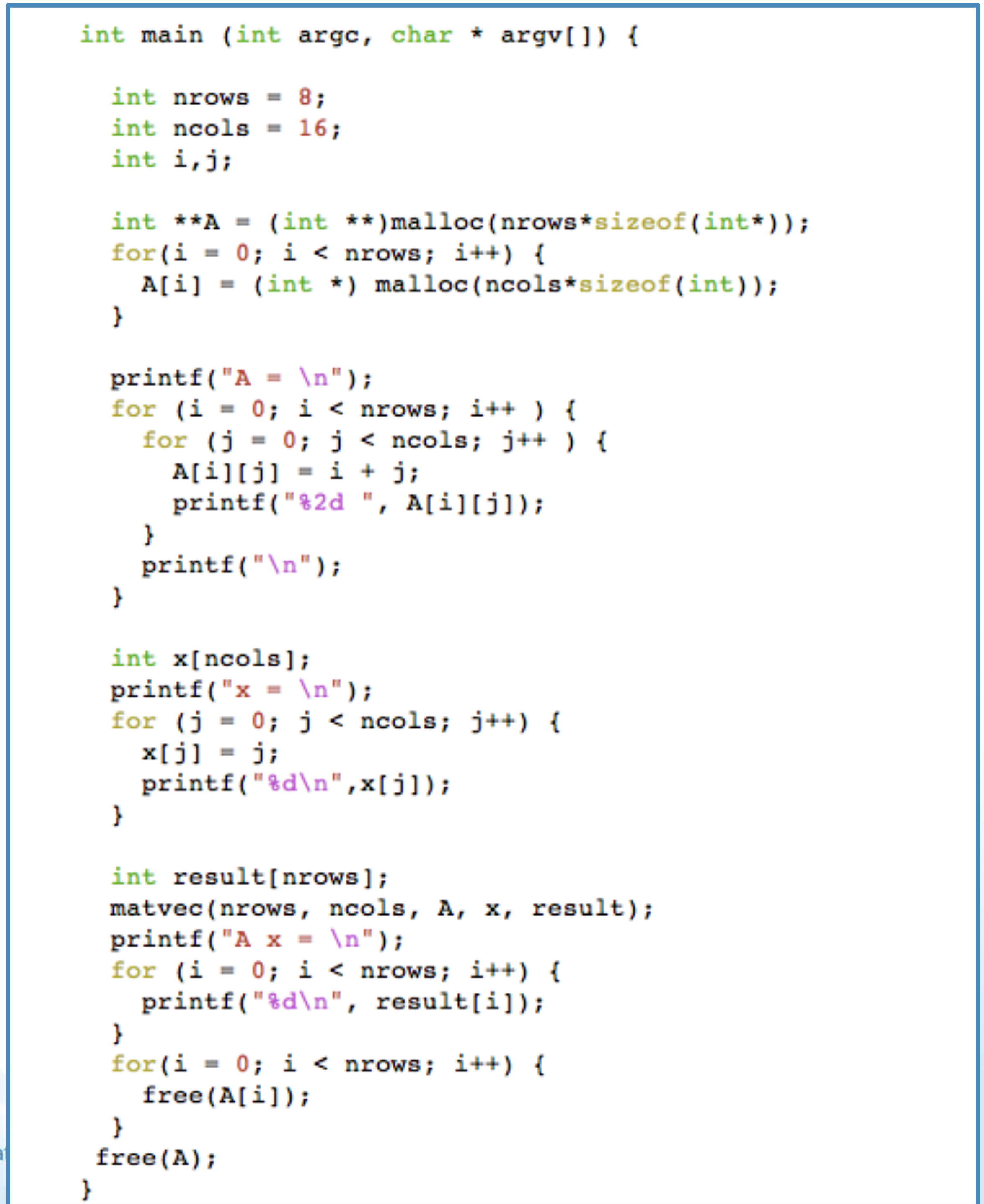
Example: Matrix-Vector Multiplication

- Multiplication of a matrix **A** of dimensions N x M and a vector **x** of size N is computed as

$$y_i = (\mathbf{Ax})_i = \sum_{j=0}^{N-1} A_{ij} x_j$$

- Matvec function implementation:

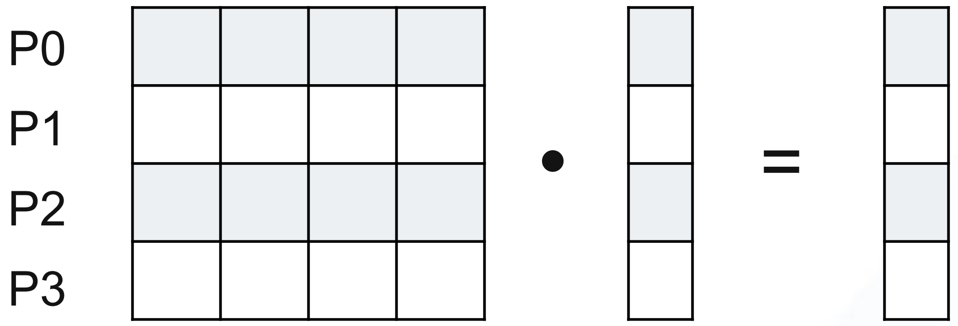
```
void matvec (int M, int N, int ** mat, int * vec, int * result) {
  int i, j;
  for ( i = 0; i < M; i++ ) {
    result[i] = 0.0;
    for ( j = 0; j < N; j++ ) {
      result[i] += mat[i][j] * vec[j];
    }
  }
}
```



```c
int main (int argc, char * argv[]) {

  int nrows = 8;
  int ncols = 16;
  int i,j;

  int **A = (int **)malloc(nrows*sizeof(int*));
  for(i = 0; i < nrows; i++) {
    A[i] = (int *) malloc(ncols*sizeof(int));
  }

  printf("A = \n");
  for (i = 0; i < nrows; i++ ) {
    for (j = 0; j < ncols; j++ ) {
      A[i][j] = i + j;
      printf("%2d ", A[i][j]);
    }
    printf("\n");
  }

  int x[ncols];
  printf("x = \n");
  for (j = 0; j < ncols; j++) {
    x[j] = j;
    printf("%d\n",x[j]);
  }

  int result[nrows];
  matvec(nrows, ncols, A, x, result);
  printf("A x = \n");
  for (i = 0; i < nrows; i++) {
    printf("%d\n", result[i]);
  }
  for(i = 0; i < nrows; i++) {
    free(A[i]);
  }
 free(A);
}
```

Parallelizing Matrix-Vector Multiplication

- In order to parallelize the Matrix-Vector multiplication, again we need to decide how the data will be mapped onto different processors.
- We could use "block-row" or "panel" mapping.



- We need to either gather different elements of the vectors, or scatter the corresponding rows of the matrix in order to do the multiplication.

- You can get the example code at GitHub either by running "git pull" from your MPI directory, or point your browser to

  http://github.com/meifeng/MPI

Gather in Matrix-Vector Multiplication

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

void matvec (int M, int N, int ** mat, int * vec, int * result) {
  int i, j;
  for ( i = 0; i < M; i++ ) {
    result[i] = 0.0;
    for ( j = 0; j < N; j++ ) {
      result[i] += mat[i][j] * vec[j];
    }
  }
}


int main (int argc, char * argv[]) {
  int nrows = 8;
  int ncols = 16;
  int i,j;
  int my_rank;
  int nprocs;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  //assuming nrows and ncols are divisible by nprocs
  int local_nrows = nrows / nprocs;
  int vsize = ncols;
  int local_vsize = vsize / nprocs;

  int **localA = (int **)malloc(local_nrows*sizeof(int*));
  for(i = 0; i < local_nrows; i++) {
    localA[i] = (int *) malloc(ncols*sizeof(int));
  }

  for (i = 0; i < local_nrows; i++ ) {
    for (j = 0; j < ncols; j++ ) {
      localA[i][j] = i + (j+my_rank*local_nrows);
    }
  }
```

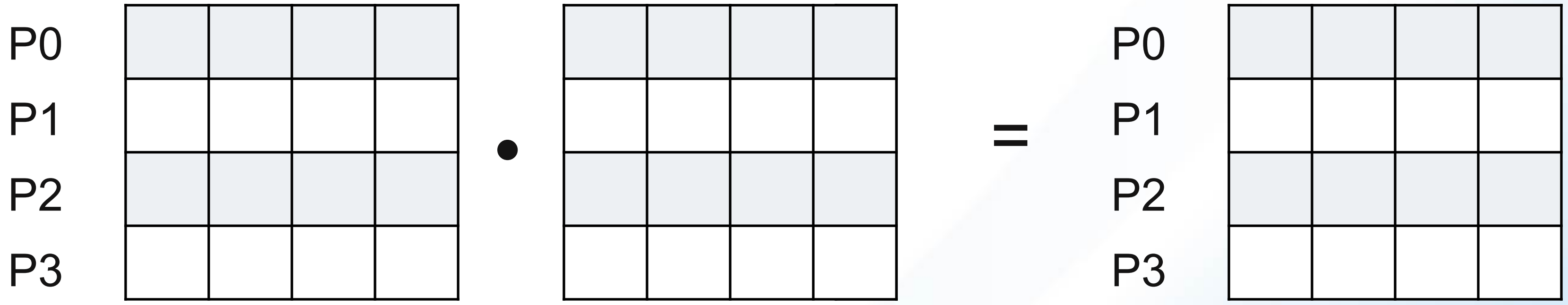
```c
  int x[vsize]; //global vector
  int local_x[local_vsize]; //local vector
  for (j = 0; j < local_vsize; j++) {
    local_x[j] = j + my_rank * local_vsize;
  }

  //Gather the local vectors into a full vector
  //and make it available on all processes
  MPI_Allgather(local_x, local_vsize, MPI_INT,
                x, local_vsize, MPI_INT, MPI_COMM_WORLD);

  int result[nrows];
  int local_result[local_nrows];

  //Do the local multiplication
  matvec(local_nrows, ncols, localA, x, local_result);

  //Gather the final result to Process 0
  MPI_Gather(local_result, local_nrows, MPI_INT,
             result, local_nrows, MPI_INT,
             0, MPI_COMM_WORLD);

  if (my_rank == 0) {
    printf("A x = \n");
    for (i = 0; i < nrows; i++) {
      printf("%d\n", result[i]);
    }
  }
  for(i = 0; i < local_nrows; i++) {
    free(localA[i]);
  }
 free(localA);
 MPI_Finalize();
}
```

Exercise 7: Matrix Multiplication

- Multiplication of two square matrices **A** and **B** of dimensions N x N is computed as

$$(\mathbf{AB})_{ij} = \sum_{k=0}^{N-1} A_{ik} B_{kj}$$

- Use “block-row” mapping to parallelize.



- Use MPI_Gather to collect the final result and print on Process 0.
- Hint: use one-dimensional array to represent the matrices.

Exercise 7: Example Code

- Obtain the example code **7_MatMult** from GitHub
  - From your MPI directory, run
  - ➢ `git pull`

  - Or point your browser to http://github.com/meifeng/MPI

- Compile and run the code either in interactive mode, or in batch mode.
  - You can compile the serial version of the code using
  - ➢ `make serial`

```c
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
void printMat(int R, int C, double *A) {
  int i,j;
  for (i = 0; i < R; i++) {
    for (j = 0; j < C; j++) {
      printf("%8.1f ",A[i*C+j]);
    }
    printf("\n");
  }
}

int main(int argc, char *argv[]) {
  int N = 8;
  int i,j,k;
  int my_rank;
  int nprocs;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  int local_nrows = N / nprocs;
  if (N % nprocs != 0) {
    printf("Matrix size is not divisible by the number of processes\n");
    MPI_Abort(MPI_COMM_WORLD,10);
  }

  int local_vsize = local_nrows;

  double *localA, *localB, *localC;
  double *local_v, *v;
  double *globalC;

  localA = (double *) malloc(local_nrows * N * sizeof(double));
  localB = (double *) malloc(local_nrows * N * sizeof(double));
  localC = (double *) malloc(local_nrows * N * sizeof(double));

  globalC = (double *) malloc(N * N * sizeof(double));

  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      globalC[i*N+j] = 0.0;
   }
  }
  local_v = (double *) malloc(local_vsize * sizeof(double));
  v       = (double *) malloc(N * sizeof(double));

  for (i = 0; i < local_nrows; i++) {
    for (j = 0; j < N; j++) {
      localA[i*N+j] = (i + my_rank * local_nrows) + j;
      localB[i*N+j] = (i + my_rank * local_nrows) - j;
    }
  }
```

```c
  if (my_rank == 0) {
    printf("localA=\n"); printMat(local_nrows, N, localA);
    printf("localB=\n"); printMat(local_nrows, N, localB);
  }

  for (j = 0; j < N; j++) {
    for (i = 0; i < local_vsize; i++) {
      local_v[i] = localB[i*N+j];
    }

    MPI_Allgather(local_v, local_vsize, MPI_DOUBLE,
                  v,       local_vsize, MPI_DOUBLE,
                  MPI_COMM_WORLD);
    for (i = 0; i < local_nrows; i++) {
      localC[i*N+j] = 0.0;
      for (k = 0; k < N; k++) {
        localC[i*N+j] += localA[i*N+k] * v[k];
      }
    }
  }

  if (my_rank == 0) {
    printf("localA=\n"); printMat(local_nrows, N, localA);
    printf("localB=\n"); printMat(local_nrows, N, localB);
  }

  if (my_rank==0) {
    printf("localC=\n"); printMat(local_nrows,N,localC);
  }

  MPI_Gather(localC , N*local_nrows, MPI_DOUBLE,
             globalC, N*local_nrows, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  if (my_rank == 0) {
    printf("globalC = \n"); printMat(N, N, globalC);
  }

  free(v);
  free(local_v);
  free(globalC);
  free(localC);
  free(localB);
  free(localA);
  MPI_Finalize();
}
```

Other Topics in MPI

Other Topics

- Grouping data for communications
- Parallel program performance
- MPI topologies

Grouping Data for Communication

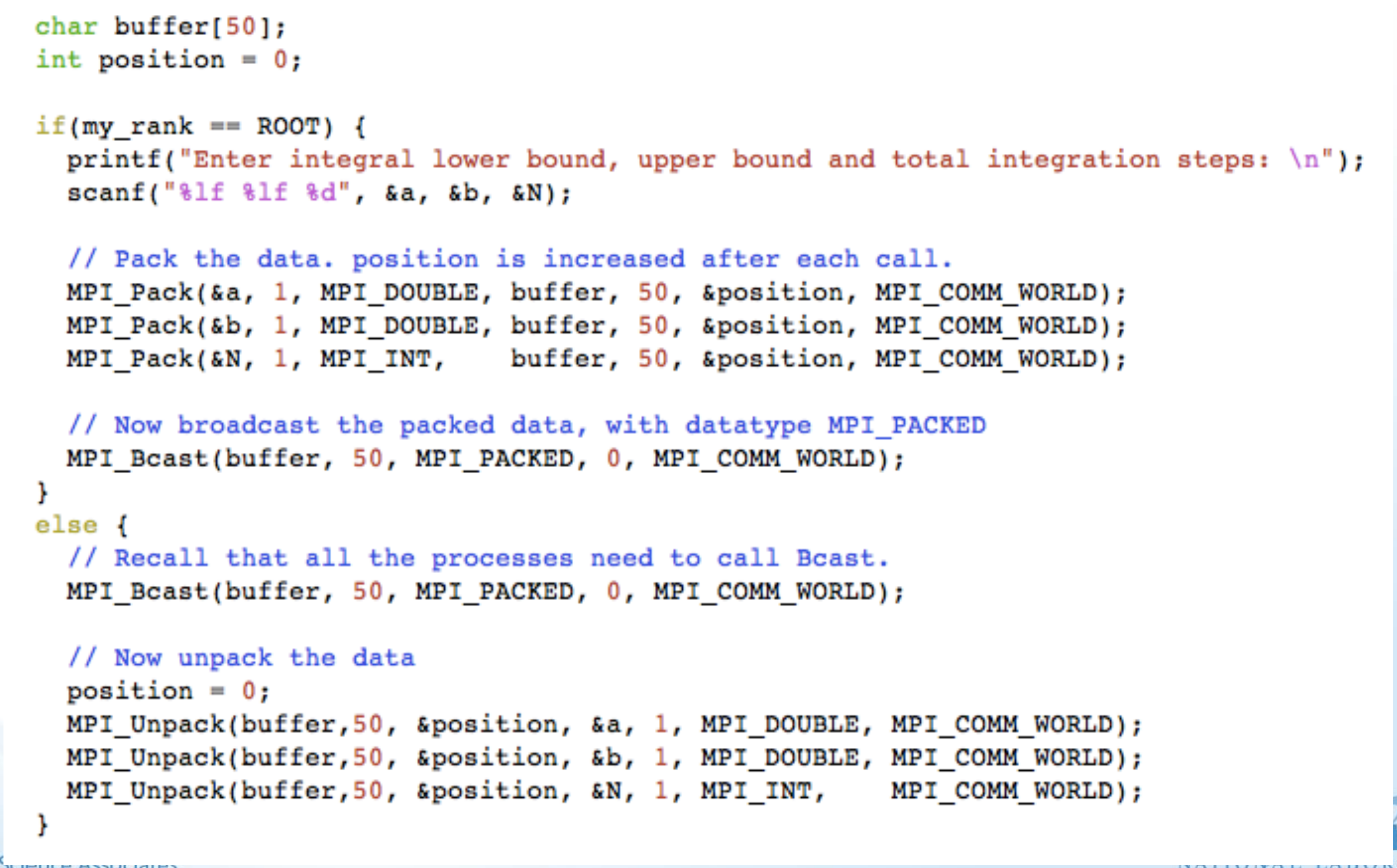
- Since communication is substantially slower than local computation, we will want to minimize the use of communication routines.
- We can achieve this by grouping data into a single message.
- One way to do this is to pack the data before communication, and unpack after.

```
MPI_Pack  (void *data,   int count, MPI_Datatype datatype,
           void *buffer, int buffer_size, int *position,
           MPI_Comm comm)

MPI_Unpack (void *buffer, int buffer_size, int *position,
            void *data,   int count, MPI_Datatype datatype,
            MPI_Comm comm)
```

➢ Note: here buffer_size is the number of bytes in buffer.

Example 8: TrapezoidPack

- In the Trapezoidal rule example, when we take inputs for a, b and N, we send three messages. We can pack them into one message.



```
char buffer[50];
int position = 0;

if(my_rank == ROOT) {
  printf("Enter integral lower bound, upper bound and total integration steps: \n");
  scanf("%lf %lf %d", &a, &b, &N);

  // Pack the data. position is increased after each call.
  MPI_Pack(&a, 1, MPI_DOUBLE, buffer, 50, &position, MPI_COMM_WORLD);
  MPI_Pack(&b, 1, MPI_DOUBLE, buffer, 50, &position, MPI_COMM_WORLD);
  MPI_Pack(&N, 1, MPI_INT,    buffer, 50, &position, MPI_COMM_WORLD);

  // Now broadcast the packed data, with datatype MPI_PACKED
  MPI_Bcast(buffer, 50, MPI_PACKED, 0, MPI_COMM_WORLD);
}
else {
  // Recall that all the processes need to call Bcast.
  MPI_Bcast(buffer, 50, MPI_PACKED, 0, MPI_COMM_WORLD);

  // Now unpack the data
  position = 0;
  MPI_Unpack(buffer,50, &position, &a, 1, MPI_DOUBLE, MPI_COMM_WORLD);
  MPI_Unpack(buffer,50, &position, &b, 1, MPI_DOUBLE, MPI_COMM_WORLD);
  MPI_Unpack(buffer,50, &position, &N, 1, MPI_INT,    MPI_COMM_WORLD);
}
```

Parallel Program Performance

- It is obviously important to be able to measure the performance of the parallel problems.
- MPI provides some functions to do this.

int MPI_Barrier(MPI_Comm comm)

- ➢ It blocks until all the processes have successfully executed it.
- ➢ It essentially synchronizes the processes in the communicator.

double MPI_Wtime(void)

- ➢ It returns a double precision value that represents the number of seconds that have elapsed since some point in the past.
- ➢ Needs matching function calls at the start and finish of the code segment being timed.

Timing

```
double start, finish;

/* some calculations */

MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();

/* some more calculations */

MPI_Barrier(MPI_COMM_WORLD);
finish = MPI_Wtime();

if(my_rank == 0)
  printf(“Time elapsed = %e seconds\n”, finish-start);
```

Exercise 9: Timing

- Put in some timings in your existing trapezoidal rule programs with point-to-point communications, collective communications and with MPI_Pack.

- Compare timings in different versions.

- How does the timing change with different numbers of processes?

Strong and Weak Scaling

- Ideally, as we increase the number of processors while keeping the problem fixed, the time it takes should decrease proportionally.
- Equivalently, if we keep the local problem size fixed, increasing the number of processors should allow us to solve a bigger problem within a given time.

- In reality, communication and I/O latency will cause some performance loss. And such loss is usually bigger as we spread the problem over a larger number of processors.
- We measure the "scalability" of our program using strong scaling and weak scaling.

- **Strong scaling** is defined as how the solution time changes with the number of processors for a *fixed total problem size*.
- **Weak scaling** is defined as how the solution time changes with the number of processors for a fixed problem size *per processor*.

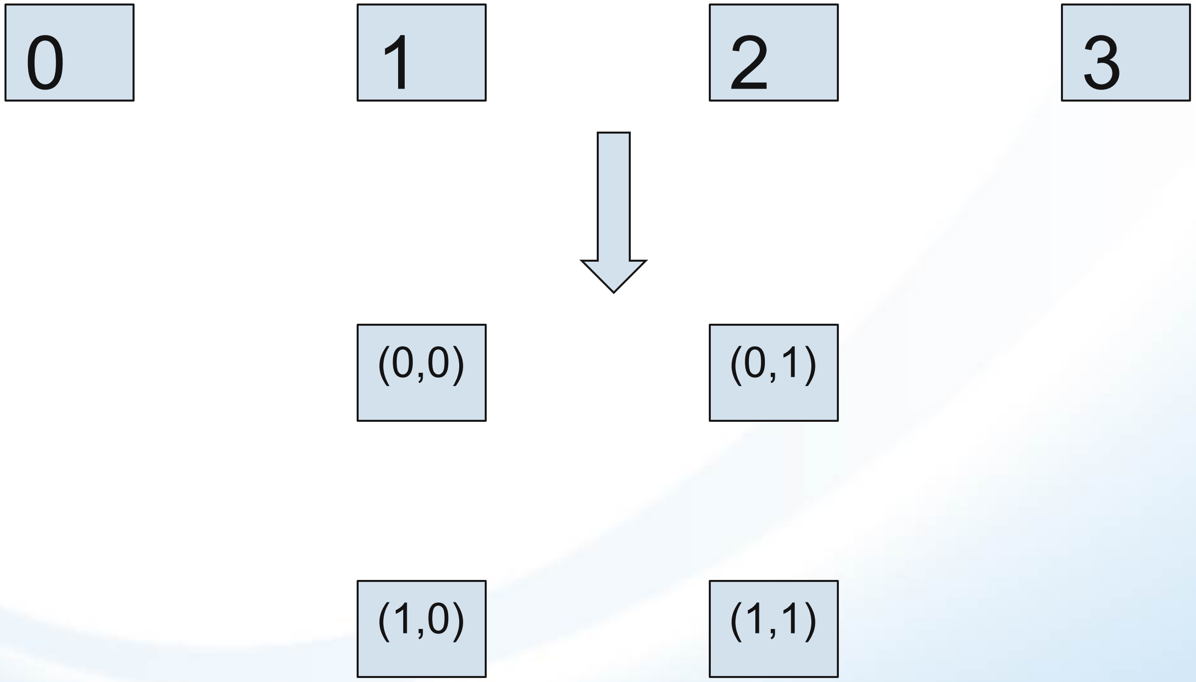
Homework

1. With the TrapezoidCollective example, measure the timing with 1, 2, 4, and 8 processes while keeping N fixed. [Strong Scaling]

2. Choose a starting value for N, then with 1, 2, 4 and 8 processes, increase the value of N by 1, 2, 4 and 8 respectively. Measure the timings in this scenario. [Weak Scaling]

Communicators and Topologies

- So far we have been using the MPI ranks of the processes to identify the source and destination of the communication.
- Sometimes it is more convenient to define a virtual topology of the processes.



Create an MPI Topology

- There are two types of virtual topologies that can be created in MPI: a **Cartesian** or grid topology, and a **graph** topology.
- Cartesian topology is the most widely used. We will need to specify
  - The number of dimensions in the grid.
  - The size of each dimension.
  - Periodicity of each dimension.
  - Whether to allow the system to optimize the mapping of the grid to the underlying physical processors by reordering.
- The mapping will define a new communicator.

```
MPI_Cart_create( MPI_Comm old_comm,
                 int ndims, int dim_sizes[], int wrap_around[],
                 int reorder, MPI_Comm *new_comm)
```

Cartesian Topology Example

```
MPI_Comm new_comm;
int dim_sizes[2];
int wrap_around[2];
int reorder = 1;

dim_sizes[0] = dim_sizes[1] = 2;
wrap_around[0] = wrap_around[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dim_sizes,
                wrap_around, reorder, &new_comm);
```

Using the new MPI Topology

- After the new grid topology is created, we can determine the coordinates of each process by calling

MPI_Cart_coords(MPI_Comm new_comm, int new_rank,
int ndims, int coordinates[])

- Given the coordinates, we can obtain the rank of the process

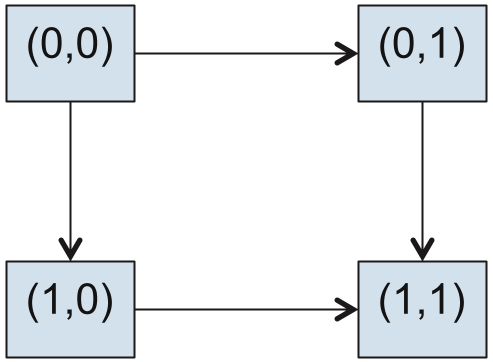
MPI_Cart_rank (MPI_Comm new_comm, int coords[], int *rank)

- Example:

```
int coordinates[2];
int my_grid_rank;
MPI_Comm_rank(new_comm, &my_grid_rank);
MPI_Cart_coords(new_comm, my_grid_rank, 2,
                coordinates);
```

Exercise 10: Greetings2D

- Rewrite the Greetings program so that a greeting is sent to the nearest neighbors in the plus directions within a two-dimensional Cartesian topology.



$$\begin{tikzcd}
(0,0) \arrow[r] \arrow[d] & (0,1) \arrow[d] \\
(1,0) \arrow[r] & (1,1)
\end{tikzcd}$$

- You will need to know the ranks of your four neighbors in the new Cartesian topology.

Cheat Sheet

```
MPI_Comm new_comm;
int dim_sizes[2];
int wrap_around[2];
int reorder = 1;

dim_sizes[0] = dim_sizes[1] = 2;
wrap_around[0] = wrap_around[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dim_sizes,
                wrap_around, reorder, &new_comm);
```

```
int coordinates[2];
int my_grid_rank;
MPI_Comm_rank(new_comm, &my_grid_rank);
MPI_Cart_coords(new_comm, my_grid_rank, 2,
                coordinates);
```

```
int neighbor_rank;
int neighbor_coords[2];
MPI_Cart_rank(new_comm,neighbor_coords,&neighbor_rank);
```

Exercise 10: Example Code

Get the code from GitHub:

➢ `git pull` or http://github.com/meifeng/MPI

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

int main(int argc, char *argv[])
{
  const int ROOT = 0;
  int my_rank;
  int recv_rank;
  int numtasks;
  int p;

  MPI_Init(&argc, &argv); /*Initializes MPI calls*/

  MPI_Status status;

  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);  /* obtains the rank of current MPI process */
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks); /* obtains the total number of MPI processes */

  int size = (int)sqrt(numtasks);
  MPI_Comm new_comm;
  int dim_sizes[2];
  int wrap_around[2] = {1,1};
  int reorder = 1;

  dim_sizes[0] = size;
  dim_sizes[1] = numtasks/size;

  MPI_Cart_create(MPI_COMM_WORLD, 2, dim_sizes, wrap_around, reorder, &new_comm);

  int coords[2];
  int my_grid_rank;
  MPI_Comm_rank(new_comm, &my_grid_rank);
  MPI_Cart_coords(new_comm, my_grid_rank, 2, coords);

  char message[100];
  sprintf(message, "Greetings from process (%d, %d)",coords[0], coords[1]);
```

```c
  int xp_neighbor_rank, xm_neighbor_rank;
  int yp_neighbor_rank, ym_neighbor_rank;

  int xp_neighbor_coords[2], xm_neighbor_coords[2];
  int yp_neighbor_coords[2], ym_neighbor_coords[2];

  xp_neighbor_coords[0] = (coords[0] + 1)%dim_sizes[0];
  xp_neighbor_coords[1] = coords[1];

  xm_neighbor_coords[0] = (coords[0] - 1 + dim_sizes[0])%dim_sizes[0];
  xm_neighbor_coords[1] = coords[1];

  yp_neighbor_coords[0] = coords[0];
  yp_neighbor_coords[1] = (coords[1] + 1)%dim_sizes[1];

  ym_neighbor_coords[0] = coords[0];
  ym_neighbor_coords[1] = (coords[1] - 1 + dim_sizes[1])%dim_sizes[1];

  MPI_Cart_rank(new_comm, xp_neighbor_coords, &xp_neighbor_rank);
  MPI_Cart_rank(new_comm, xm_neighbor_coords, &xm_neighbor_rank);

  MPI_Cart_rank(new_comm, yp_neighbor_coords, &yp_neighbor_rank);
  MPI_Cart_rank(new_comm, ym_neighbor_coords, &ym_neighbor_rank);

  MPI_Send(message, 100, MPI_CHAR, xp_neighbor_rank, 0, new_comm);
  MPI_Send(message, 100, MPI_CHAR, yp_neighbor_rank, 1, new_comm);

  char recv_messagex[100];
  char recv_messagey[100];
  MPI_Recv(recv_messagex, 100, MPI_CHAR, xm_neighbor_rank, 0, new_comm, &status);
  MPI_Recv(recv_messagey, 100, MPI_CHAR, ym_neighbor_rank, 1, new_comm, &status);

    printf("My Cartesian coordinates: (%d, %d)\n", coords[0], coords[1]);
    printf("Message received:\n %s \n %s\n", recv_messagex, recv_messagey);

  MPI_Finalize();        /*Finalizes MPI calls */
  return 0;
}
```

Wrapping Up…

- There are still many features in MPI that haven’t been covered.
  - User-defined data types.
  - MPI groups, contexts and user-defined communicators.
  - MPI I/O.
  - MPI/OpenMP hybrid programming.
  - …
- But what you learn today will allow you to write basic MPI programs.

References for further reading

- "**Parallel Programming with MPI**", Peter S. Pacheco, Morgan Kaufmann Publishers, 1997. [Pedagogical]
- "**Introduction to Parallel Computing**" (2nd Ed.), Ananth Grama, Anshul Gupta, George Karypis and Vipin Kumar, Pearson Education, 2003. [Technical]
- "**The Art of Multiprocessor Programming**", Maurice Herlihy and Nir Shavit, Morgan Kaufmann Publishers, 2008. [Technical]