**Compute Pi using Distributed Memory**
Blue Waters Institute 2017
Day 4 (Week 1, Thursday, June 1), 7:00pm-9:00pm
Lead Instructor: Colleen Heinemann

## Interactive Job Request

**qsub -I -l nodes=1:ppn=32:xe,walltime=02:00:00,advres=bajy<ENTER>**

Note: the **advres** does not appear to be working this evening.

When you get on the MOM node, run **screen<ENTER>**
Keep track of the hostname of the MOM node, so if you lose connection to it, you can SSH to it from an **h2ologin** node and run **screen -x<ENTER>**.
If you forget to keep track of the hostname of the MOM node, you can run **qstat -u your username -f<ENTER>** and look for the hostname near the bottom of the output.
If you are logged into a MOM node but not sure whether you got there through **qsub** or through **ssh**, run **env|grep JOB<ENTER>**. If it shows that the job ID is **STDIN**, it means you got there through **qsub**; if it shows nothing, it means you got there through **ssh**. Keep in mind you need to use **aprun** ONLY if you got there through **qsub**; DON'T use **aprun** if you got there through **ssh**.

## Goals

- Learn the concepts:
  - Analyze a simple scientific algorithm
  - Apply MPI and distributed memory concepts to the code
- Practice:
  - Connecting to Blue Waters
  - Transferring code to Blue Waters
  - Compiling code on Blue Waters
  - Writing MPI code

## Introduction

The code that we will be analyzing and modifying computes pi by approximating the area under the curve for **f(x) = 4 / (1+x*x)** between 0 and 1. To do such an integration numerically, the interval from 0 to 1 is divided into a given number of subintervals, **num_rect**. The area of the rectangles is then added together.

The larger the value of the **num_rect**, the more accurate the results will be.

The program asks the user to input a value for number of subintervals, computes the approximation for pi, and compares it to a more accurate approximate value of pi in the **<math.h>** library.

## Activity

1. Download the serial code for calculating pi and copy it to Blue Waters.

**wget https://shodor.org/media/content//petascale/materials/BW2017/compute_pi.zip<ENTER>**

2. Unzip the folder
   **unzip compute_pi.zip<ENTER>**
3. Change locations into the directory containing the code
   **cd compute_pi<ENTER>**
4. Compile the serialized version of the code:
   **cc compute_pi.c -o compute_pi.exe<ENTER>**
5. To run the serial version of the code, run the command
   aprun -n 1 **./compute_pi.exe<ENTER>**
6. Experiment with varying numbers of rectangles and take note of the runtimes for the different tests
7. Once you are comfortable with the serialized code for computing pi, refer to the following MPI commands and modify the code to run with MPI in distributed memory
   a. To compile your code with MPI, use the command
      **cc compute_pi.c -o mpi_pi.exe**
   b. To run your code with MPI, use the command
      **aprun -n   ./mpi_pi.exe**

***Once your code runs with MPI, run tests on varying numbers of ranks with varying problem sizes***

NOTE: Some of the commands in the following list have not been discussed yet, but are given as reference due to the fact that they can be very useful in this exercise

**MPI_Init**

USAGE: `int MPI_INIT(int *argc, char ***argv)`

EXAMPLE: `ierr = MPI_Init(&argc, &argv);`

**MPI_Init** is required to initialize the MPI execution environment; it can be in your code only once

**MPI_Finalize**

USAGE: `int MPI_Finalize()`

EXAMPLE: `MPI_Finalize(void)`

**MPI_Finalize** terminates the MPI execution environment; it can only be in your code once

**MPI_Comm_rank**

USAGE: `int MPI_Comm_rank(MPI_Comm comm, int *rank)`

EXAMPLE: `ierr = MPI_Comm_rank(MPI_COMM_WORLD, &processId);`

**MPI_Comm_rank** returns the rank of the calling process in the communicator

**MPI_Comm_size**

USAGE: `int MPI_Comm_size(MPI_Comm comm, int *size)`

EXAMPLE: `ierr = MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);`

**MPI_Comm_size** determines the size of the group, or how many processes, are associated with what you are doing

**`MPI_Send`**

USAGE: `int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`

EXAMPLE: `ierr = MPI_Send(&work, 0, MPI_Int, rank, DIETAG, MPI_COMM_WORLD);`

**`MPI_Send`** performs a safe send. It may block until the message is received by the destination process. The tag of the send call will need to match that of the receive call

PARAMETERS:

- **`buf`** initial address of the send buffer
- **`count`** number of elements in the send buffer; this must be a non-negative integer
- **`datatype`** datatype of each send buffer element
- **`dest`** is the destination rank; usually rank 0
- **`tag`** message tag; generally a unique identifier
- **`comm`** the communicator

**`MPI_Recv`**

USAGE: `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`

EXAMPLE: `ierr = MPI_Recv(&result, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);`

**`MPI_Recv`** is a receive command that is necessary to receive a message

OUTPUT PARAMETERS:

- **`buf`** initial address of the send buffer
- **`status`** a status object showing whether or not everything is running successfully

INPUT PARAMETERS:

- **`count`** max number of elements in the receive buffer; this is a non-negative integer
- **`datatype`** datatype of each receive buffer element
- **`source`** rank of the source destination
- **`tag`** message tag; generally a unique identifier
- **`comm`** the communicator

**`MPI_Wtime`**

USAGE: `int MPI_Wtime()`

EXAMPLE: `MPI_Wtime(void)`

**`MPI_Wtime`** provides the time in seconds since a given arbitrary time in the past

**`MPI_Bcast`**

USAGE: `int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`

EXAMPLE: `ierr = MPI_Bast(&send, 1, MPI_INT, 0, &status)`

**`MPI_Bcast`** broadcasts a message from the process with rank “root” to all other processes

OUTPUT PARAMETERS:

**buf** starting address of the buffer

INPUT PARAMETERS:

**count** number of entries in buffer

**datatype** data type of buffer

**root** broadcast root's rank

**comm** the communicator

**MPI_Reduce**

USAGE: `int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`

EXAMPLE: `ierr = MPI_Reduce(&send, &recv, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);`

**MPI_Reduce** reduces all of the values on the processes to a single value

INPUT PARAMETERS:

**sendbuf** address of send buffer

**count** number of elements in send buffer

**datatype** data type of elements in the send buffer

**op** reduce operation, such as add, subtract, etc.

**root** the root process's rank

**comm** the communicator

OUTPUT PARAMETERS:

**recvbuf** the address of the receive buffer

**MPI_Scatter**

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

EXAMPLE: `ierr = MPI_SCATTER(&send, 10, MPI_INT, &recv, 50, MPI_INT, 0, MPI_COMM_WORLD);`

INPUT PARAMETERS:

**sendbuf** address of send buffer

**sendcount** number of elements sent to each processes

**sendtype** data type of send buffer elements

**recvcount** number of elements in the receive buffer

**root** rank of sending processes

**comm** the communicator

OUTPUT PARAMETERS:

**recvbuf** address of receive buffer