Homework 1

Assignments: All parts are to be solved individually except for the last problem, which is a group problem for groups of exactly three students. All parts are to be turned in electronically, written parts in ASCII text, NO Word, postscript, etc. permitted unless explicitly stated.

Please use the ARC cluster (Linux). All programs have to be written in C, translated with mpicc/gcc and turned in with a corresponding Makefile.

1. (0 points) Learn how to compile and execute an MPI program on the ARC cluster.
   - For this class you will be using the ARC cluster. This cluster is made up of hundreds of individual nodes including AMD Opteron, Intel Sandy/Ivy Bridge, Intel Broadwell, and a login node. Each node has multiple processors/cores. When you login to the ARC cluster you will be on the login node. You should immediately obtain access to a compute node (interactively) or schedule batch jobs. Do not execute any other commands on the entry node!

     You will be sharing the nodes on this cluster with many other users outside of the CSC548 class. Please be mindful of how many nodes you are using and for how long. If you are using one or more nodes, no one else will be able to use those nodes! For all work in this class, you should use the AMD Opteron nodes.

   - The ARC cluster may only be accessed from an NCSU IP address. If you're off campus, first log in to some NCSU machine

     ```
     ssh remote.eos.ncsu.edu -l [your-unity-username]
     ```

     Use your unity password.

   - The ARC cluster requires a keypair for authentication. How to generate a long pub/private key pair (for ARC cluster etc.):

     ```
     ssh-keygen -t rsa -b 4096
     ```

     Press ENTER on all questions to just take the defaults!!!
     You should have received an email with a link to upload your public key to the cluster. Only upload your public key! Do NOT share your private key!

   - Log in to the ARC cluster using your private key

     ```
     ssh arc.csc.ncsu.edu -l [your-unity-username] -i ~/.ssh/id_rsa
     ```

     You may also set up a config file in the .ssh directory to make accessing the ARC cluster with your username and keypair easier.

   - We will compile and run a simple "Hello World" mpi program. Download the program mpi_hello.c

- Login interactively to a compute node by using srun and compile/execute the program. (only works on compute nodes, not on the login node) For all work in this class, you should use the AMD Opteron nodes by specifying "-p opteron". Use "-N1" to request one node and "-n2" to request two processors.

  ```
  srun -N1 -n2 -p opteron --pty /bin/bash
  mpicc -O3 -o mpi_hello mpi_hello.c
  prun ./mpi_hello
  ```

  Release this compute node using "exit", then try again with a different number of processors.

  ```
  exit
  srun -N1 -n 4 -p opteron --pty /bin/bash
  prun ./mpi_hello
  exit
  srun -N1 -n 8 -p opteron --pty /bin/bash
  prun ./mpi_hello
  exit
  ```

  Repeat this process but this time request two nodes using "-N2". Notice how the number of processors is distributed across the two nodes you requested. Promptly release both of the compute nodes by using "exit".

- Use a batch script mpi_hello.batch to submit a batch job for 2, 4, 8 processors and 1, 2 nodes with the sbatch command. The advantage of batch scripts is that for long jobs, it will automatically release the node(s) when the job is finished so that you aren't hogging nodes on the cluster! You must edit the mpi_hello.batch file to request the different number of processors/nodes (only works on the login node, not on the compute nodes):

  ```
  sbatch mpi_hello.batch
  ```

  Notice: sbatch will output the job id assigned to your batch script.

- Monitor your jobs progress with

  ```
  squeue
  ```

  Enter the command repeatedly until the job is done. (Note: This job is very small and will execute very quickly. You may not see your job in the queue because it has already finished.) Then, inspect the output/error files.

- If you ever want to kill your job, issue

  ```
  scancel [jobid]
  ```

- See ARC web page for additional job submission parameters and scripts.

MPI Resources:

- MPI API

- MPICH
- A User's Guide to MPI by Peter Pacheco
- MPI Quick Reference by LAM MPI
- Debugging: Gdb does not work with MPI, but totalview or pgdbg (the latter only for Portland Group compiled binaries) could be used with some training.

Nothing to turn in, this is just a warm-up exercise.

2. (50 points) Write an MPI program that determines the point-to-point message latency for pairs of nodes. We will exchange messages with varying message sizes from 32B to 2M (32B, 64B, 128B, ..., 2M). The same program should iterate through each of the message sizes with 10 iterations for each. We will use 8 nodes (4 pairs). Remember to promptly release the nodes by exiting the compute node IMMEDIATELY after your code is finished or by using batch scripts.

   - Plot the average round-trip time (rtt) as a function of message size, as well as the standard deviation (stddev) as error bars.

   In a README file, explain your plots. Do some pairs consistently take longer than others? In particular, could your results be indicative of the underlying network configuration? Explain. Also discuss message size as it relates to latency; are there any odd data points in your graphs?

   Hints:

   - To ensure only one process per node, make sure that the "-n" and "-N" values are equal
   - When calculating average rtt, skip the first message exchange -- why?
   - Use your favorite plotting program

   Note: Use gettimeofday to time the round-trip rather than MPI_Wtime since MPI_Wtime doesn't have the kind of precision we need for small message sizes.

   Sample output (of not necessarily realistic numbers): read as follows [message size] [pair_1_average] [pair_1_stddev]...

   ```
   32 8.276531e-06 5.960464e-08...
   64 9.534331e-06 4.882124e-08...
   128 12.12341e-06 5.1234264e-08...
   ...
   ```

   Example output: example output graph for some message sizes (you have more)here.

   Turn in the files p1.c, p1.Makefile, p1.png, and p1.README

3. (50 points, Group Problem)
   We will implement differentiating a function and plotting it in a parallel scheme:

Download the serial code for this problem: p2.c and p2_func.c

This code implements the serial version of the numerical techniques described below. You will update this code to run in parallel using MPI.

Given some function, the code will numerically compute *derivatives* at all grid points, vary slice sizes, and plot the function and it's derivative using gnuplot.

The serial code uses the function in p2_func.c which is x^2 in the example code. The program should work with all functions whose derivative can be calculated between the given intervals. It also takes as input the number of slices from the command line. For a list of functions, and their derivatives, see here.

Write a Makefile to compile the code with:

```
mpicc -lm -O3 -o p2 p2.c p2_func.c
```

Then run the code as: ./p2 {number_of_slices}

The program will output a file **fn-{number_of_slices}.dat** which gives:

- All the grid points the program considers;
- The corresponding value of the function at that grid point;
- The differential at that grid point.

Test the program to include more grid points, different functions, etc.

Finally, plot the function and it's derivative using the gnuplot file here with the command:

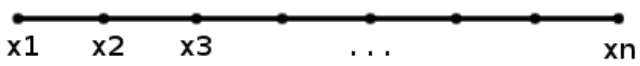
gnuplot -e "filename='fn-{number_of_slices}'" p2.gnu

Remember to replace the {number_of_slices} with the actual number of slices.

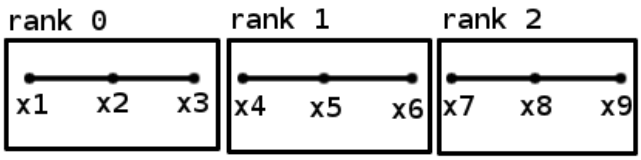
Data Decomposition

A common approach to implementing a parallel program is data decomposition. The data the program will run with is split up into chunks and a copy of the program runs using each individual chunk.

Consider the array of N grid points x_1 to x_n:



For our program, we will split this grid into evenly spaced "slices" by processor rank:



IMPORTANT: Be sure that the number of grid points is evenly divisible by the number of processors, *or* make sure your final code handles the case where the number of processors does not evenly divide the number of grid zones.

Computing Derivatives: Finite Differencing

It is often the case that we cannot analytically calculate the derivative of some data. However, we can approximate it using a finite differencing scheme.

The method comes from calculus. Recall the definition of a derivative of the function f(x):

$$\frac{d}{dx}f(x)=\lim_{h\to 0}\frac{f(x+h)-f(x)}{h}$$

We can assert, with reasonable justification, that if h is small enough, a good approximation to the derivative is given by:

$$\frac{d}{dx}f(x)\approx\frac{f(x+h)-f(x)}{h}$$

This is a good approximation, but a better one is to use the symmetric form of the above:

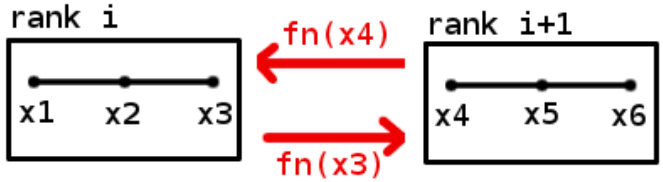
$$\frac{d}{dx}f(x)\approx\frac{f(x+h)-f(x-h)}{2h}$$

In our program, let **x[i]** be our grid points, and let **y[i]** be the value of our function at grid point **x[i]**. Using the above, we can approximate the derivative at the grid point **x[i]** as:

```
dy[i] = (y[i+1] - y[i-1]) / (2.0 * dx)
```

We should notice two things: (i) We need to know the *boundary*

conditions of our full grid domain - that is, what the value of the function will be at the left and right edge of the grid, and (ii) For our decomposed grid, we will need to communicate the boundary values of our function to the processors to the "left" and "right". The diagram below represents one side of this process. The process of *rank i* will receive the value **fn(x_4)** (**fn(x)** represents the value of our function at point **x**) from the process of *rank i+1* send the value of the function **fn(x_3)** to the process of *rank i+1*



Problem procedure

Using the provided serial code, the assignment is as follows:

- Decompose the grid into slices based on processor rank.
- Calculate the derivative of the functions using a finite difference method. Properly calculating the derivative at the domain boundaries requires communication across the boundaries. Perform this communication using (a) blocking point-to-point communication and (b) non-blocking point-to-point communication.
- Using MPI_Gather get all values at a single rank and write them out to a file called fn-{number_of_slices}.dat.
- Plot the sin(x) function and it's derivative for various grid sizes (100, 1000, 10000).

Some guidelines:

- Your program should produce accurate results; be sure to compare the output of the serial code to the parallel code.
- Your program should be robust enough to do all operations with a wide range of input parameters (e.g. number of grid points, number of processors used).
- Run your program with several different functions; your code will be tested with a unique function to check it runs correctly.
- Develop and debug your code in interactive mode on only one node with multiple processors.
- Test your code using batch scripts and no more than 8 nodes.

Compare the performance (using MPI_Wtime) for long-running inputs (large number of grid points), and each communication method (blocking/non-blocking point-to-point, single call/manual gather) with batch jobs (to ensure low contention). Show your results and comment on the outcome in the README file.

There will be 2 versions for point-to-point communication:

- **Blocking - 0 `**
- **Non-Blocking - 1**

There will be a total of 2 versions for gathering the final results:

- **Using MPI_Gather - 0**
- **Manual gather - 1**

The numbers 0 & 1 indicate the command line parameter passed to the program to use the corresponding version.

There will be a total of 4 versions:

- **0,0: The border values will be communicated using blocking MPI calls, and the final results will be gathered using MPI_Gather/GatherV/etc.**
- **0,1: The border values will be communicated using blocking MPI calls, and the final results will be gathered manually using blocking MPI point-to-point communications.**
- **1,0: The border values will be commuincated using non-blocking MPI calls, and the final results will be gathered using MPI_Gather/GatherV/etc.**
- **1,1: The border values will be communicated using non-blocking MPI calls, and the final results will be gathered manually using non-blocking MPI point-to-point communications**

For manual gather you should use the point-to-point version specified in the command line.

Your program will be executed as:

prun ./p2 {number_of_gridpoints} {point-to-point_type} {gather_type}

Comment on how the accuracy of the plot change for (i) the number of grid points and (ii) the function in question. The sample output of **fn-100.dat** is here and the corresponding **fn-100.png** is here.
NOTE: The above output was created using the provided **p2.c**

Turn in the files p2_mpi.c, p2_func.c, fn-100.png, fn-1000.png, fn-10000.png, p2.Makefile, p2.README

4. Peer evaluation: Each group members has to submit a peer evaluation form.

What to turn in for programming assignments:

- commented program(s) as source code, comments count 15% of the points (see class policy on guidelines on comments)
- Makefiles (if required)
- test programs as source (and input files, if required)

- README (documentation to outline solution and list commands to install/execute)
- in each file, include the following information as a comment at the top of the file, where "username" is your unity login name and the single author is the person who wrote this file:

```
Single Author info:

username FirstName MiddleInitial LastName

Group info:

username FirstName MiddleInitial LastName

username FirstName MiddleInitial LastName

username FirstName MiddleInitial LastName
```

How to turn in:

Submit your assignments on Moodle.

Remember: If you submit a file for the second time, it will overwrite the original file.

Additional references:

- **Cliff Notes for GDB** - the very bare bones of GDB (by Dr. McKinley of University of Massachusetts)
- **Gnu manuals (make, gdb and many more)**