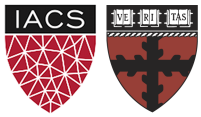
CS205, Fall 2013
Computing Foundations for Computational Science
Cris Cecka



HARVARD
School of Engineering
and Applied Sciences

Homework 2
Due October 11th, 2012 at 11:59pm

This assignment will familiarize you with some of the MPI primitives and common MPI pitfalls. Many of the problems remain MapReduce-able, but with MPI we will have more fine-grained control and be able to realize more reliable performance gains.

Download the material for HW2 here.

or

```
wget http://iacs-courses.seas.harvard.edu/courses/cs205/resources/hw2.zip
```

Problem 1 - Introduction

mpi4py

Although the documentation for `mpi4py` is sparse, all of the methods and submodules can be found here. For syntax and MPI primitives, see the listing for `mpi4py.MPI.Comm` here.

Running MPI interactively on CS205/Resonance Nodes

The `resonance.seas` cluster is composed of 16 compute nodes where each node contains six hyperthreaded cores for an effective 12 cores per node. The head node is called `resonance` and the compute nodes are called `cuda-1-XX`.

The `cs205.seas` cluster is composed of 4 compute nodes where each node contains 64 cores. We recommend using the `cs205` cluster nodes for this homework. The head node is called `cs205` and the compute nodes are called `cs205-compute-X`

To log in to a CS205 node you can simply open the CS205 NODE terminal. The terminal icon is a shortcut for `ssh`-ing into the cluster's headnode:

```
$ ssh -AY cs205.seas.harvard.edu
```

and then logging into a compute node with:

```
$ qlogin
```

To load the software required for this course into the cluster, execute:

```
$ module load courses/cs205/2013
```

This command can be run automatically each time you log-in if it is appended to the file `~/.bashrc` on one of the cluster or compute nodes.

Students will be approximately equally distributed across the compute nodes of the cluster. Once logged into a compute node, you will be able to execute `mpiexec` commands interactively:

```
$ mpiexec -n P EXECUTABLE
```

For example, to run a python script named `hello_mpi.py` with 8 processes:

```
$ mpiexec -n 8 python hello_mpi.py
```

Using a compute node will give you an interactive session where you can view your output and plot from your python scripts as you normally would. This can be useful for debugging, but to use even more cores we will need to use the entire cluster rather than a single node. We introduce how to do so in the next section.

Job Queue and QSUB on Head Nodes

Instead of logging in to a compute single node with `qlogin`, we can submit jobs to the head node's job queue to be run across the entire cluster. Included in the provided files is a script that tells the queue how many processes to launch and makes sure each process has the proper compute environment. View this script with

```
$ cat runscript
```

Note that some of the lines of the runscript are commented. This is intentional, these commented commands are passed to `qsub` to configure the launch of the job. The important lines that should be changed throughout this homework are the number of processes to launch:

```
# The number of processes to launch
#$ -pe orte 2
```

and the name of the python script:

```
# Launch the job
mpiexec -n $NSLOTS python my_python.py
```

Once you believe these are correct, you can submit a job to the `cs205.seas` cluster from the head node by executing

```
$ qsub runscript
```

which queues your job for launch. If you are fast, you can execute

```
$ qstat
```

to view your job on the queue. The status of the job is indicated by one or more of the following characters

```
r - running
t - transferring to a node
qw - waiting in the queue
d - marked for deletion
R - marked for restart
```

The job will return a file `cs205hw.o###` which contains the standard output and any errors. If your job appears to be locked up or you decide to abandon the run, you can delete the job from the queue with

```
$ qdel ###
```

where `###` is the job number it is assigned from `qstat`.

Using the head node and submitting jobs to the queue prevents any interactive output, which means that you will not be able to produce plots (but you will be able to save images) or view textual output dynamically. Only the `cs205hw.o###` output file is returned.

Troubleshooting

The clusters may also not be able to allocate you a slot into a compute node. This is usually the case when you already have an interactive slot that has not been killed. To determine if you have a already have an abandoned slot and kill it, login to the head node and execute

```
$ qstat
$ qdel ###
```

To avoid this case, always exit a terminal with the `exit` command.

If running your code on results in a module error, check that you have the correct modules loaded with:

```
$ module list
```

which should display an entry for `courses/cs205/2013`. If this is not the case, you can load it with the command

```
$ module load courses/cs205/2013
```

or edit your ~/`.bashrc` script to include the lines

```
source /etc/bashrc
module load courses/cs205/2013
```

which forces the cluster to automatically load the module each time you login.

Clusters on Amazon EC2 with StarCluster

StarCluster is an open-source toolkit designed to automate and simplify the process of building, configuring, and managing clusters of virtual machines on Amazon EC2. Watch the 10-minute screencast **here** to get a sense of what StarCluster is capable of. To get started, install StarCluster in your Ubuntu VM

```
$ sudo apt-get install python-setuptools
$ sudo easy_install StarCluster
```

Now run the `help` command:

```
$ starcluster help
```

When it asks you if you want to create a config file, select option 2.

A config file has now been created in ~/.starcluster/config. The `help` command and the well-documented config file are great resources for understanding the various StarCluster options.

Now that we have created a config file, we need to add our AWS credentials. Open the config file with your favorite editor and fill in the `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`,

and `AWS_USER_ID` with your credentials.

Finally, we need to create a keypair that we will use to login to the machines in our cluster. StarCluster will conveniently create a keypair for us and upload it to Amazon so that it's available on all our machines.

```
$ starcluster createkey mykey -o $HOME/.ssh/mykey.rsa
```

Now that you have StarCluster installed and configured, try it out by creating your own cluster.

```
$ starcluster start -s 2 -i m1.small -k mykey hw2
```

This tells StarCluster to create a cluster called `hw2` with 2 small instance type nodes and to use `mykey` as the key pair for logging into the cluster. This setup corresponds to the default cluster template defined in the StarCluster config file. We can launch the same cluster using the name of the cluster template instead.

```
$ starcluster start -c smallcluster hw2
```

Once your cluster is up and running, try logging into it.

```
$ starcluster sshmaster -u sgeadmin hw2
```

This logs you in as the default user sgeadmin. Once logged in, you will be able to `qlogin` to compute nodes, run `mpiexec` commands, and submit jobs to the queue using `qsub` just like our in-house clusters.

You can send files to and from your cluster using the `put` and `get` commands.

```
$ starcluster put -u sgeadmin hw2 </path/to/local/file/or/dir> </path/on/cluster>
$ starcluster get -u sgeadmin hw2 </path/on/cluster> </path/to/local/file>
```

Additionally, the `/home` directory is NFS shared across all the nodes in the cluster so any file saved in that directory by a node will immediately be visible to all the other nodes in the cluster.

When you are done remember to terminate any clusters you have running or Amazon will continue charging your account. First find out which clusters are running.

```
$ starcluster listclusters
```

Then terminate each one.

```
$ starcluster terminate <name_of_your_cluster>
```

Problem 2 - Hello MPI [20%]

We wish to compute the inner-product:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{k=0}^{K-1} a_k b_k$$

where $\mathbf{a}$ and $\mathbf{b}$ are vectors and the dimension K is quite large.

Verify correctness and analyze speedup

Run the code on a `cs205.seas` compute node with $P = 1, 2, 4, 6, 8, 12$, and 16 processes, verify that it is correct, and record the running times. Plot the speedup and efficiency as a function of the number of processes. Explain the results.

Wait a sec...

Run the code with $P = 5$ processes. Show and explain the results.

Generalize and Improve

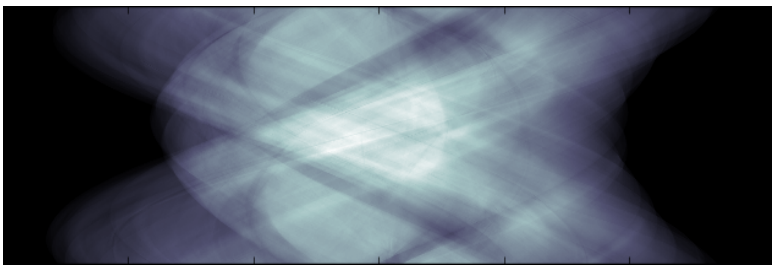
Fix the code in `P3.py` so that it works for any number of processes and any size input. Explain your approach.

What other improvements could be made to the code? Explain your reasoning.

Submission Requirements

- `P2.py`: Final implementation.
- `P2.pdf`: Explanations, plots/tables, and output.

Problem 3 - Tomography [40%]



The image above is called a tomographic projection. It is a rendering of the data in the file `TomographicData.bin`, which consists of 2048 rows of 6144 double-precision (64-bit) floating points samples each. One row of data is the projection of an image at an angle ϕ, where row k corresponds to the angle $\phi = -k\pi/2048$.

You're working at a medical office and your x-ray tomography machine sends you data files like this as it circles around a patient's head and images a single slice of tissue at different angles. Each line of data is the attenuation curve of the slice taken at a different angle. You are tasked with creating the highest resolution image from this data as quickly as possible.

The python code in `P4_serial.py` reads this data and reconstructs the original image in serial.

MPI Send/Recv

Write a parallel version in which the root process reads the input and uses only MPI `send` and `recv` (or `Send` and `Recv`) to distribute the data and perform the computation in parallel. Show how you verify your program is correct. Plotting the image at each step is not required. You may assume the number of processes is a power of two.

MPI Scatter/Reduce

Instead of `send` and `recv` use only MPI `scatter` and `reduce` to write a (hopefully) more simple version. Plotting the image at each step is not required. You may assume the number of processes is a power of two.

HINT: The `mpi4py scatter` works along the rows of an array: each row is sent to a unique process.

Analysis of Results

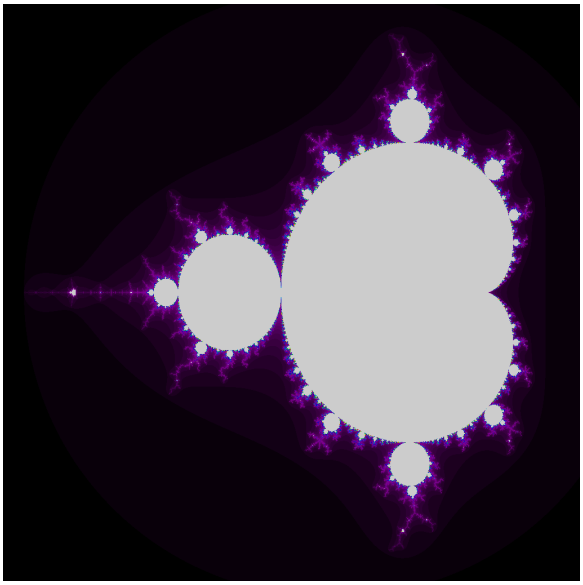
Time the scatter/reduce code with `MPI.Wtime()` counting only the communication and computation – ignore any IO, plotting, or precomputation. Compute the speedup and efficiency

of the scatter/reduce version with `ImageSize = 512, 1024, and 2048` and `P = 1, 2, 4,` and `8`. Tabulate these values. Does the efficiency increase when `ImageSize` is increased? Explain.

Submission Requirements

- `P3A.py`: Completed Send/Recv implementation.
- `P3B.py`: Completed Scatter/Reduce implementation.
- `P3.pdf`: Explanations and/or plots.

Problem 4 - Master/Slave with MPI [40%]



Because computing the Mandelbrot set above is embarrassingly parallel – no pixel of the image depends on any other pixel – we can distribute the work across many MPI processes in many ways. In this problem, we will conceptually think about load balancing this computation and implement a general strategy that works for a wide rage of problems (though doesn't scale to very very large problems).

A pixel of the Mandelbrot image located at the coordinates (x, y) can be computed with the `mandelbrot` function:

```
def mandelbrot(x, y):
    z = c = complex(x,y)
    it, maxit = 0, 511
    while abs(z) < 2 and it < maxit:
        z = z*z + c
        it += 1
    return it
```

One issue with parallelizing this is that the `mandelbrot` function can require anywhere from 0 to 511 iterations to return.

Load Balancing

Intern Susie implements the above computation with P MPI processes. Her strategy is to make process p compute all of the (valid) rows $p + nP$ for $n = 0, 1, 2, \ldots$ and then use an MPI `gather` operation to collect all of the values to the root process.

Intern Joe implements the above computation with P MPI processes as well. His strategy

is to make process p compute all of the (valid) rows $pN, pN+1, pN+2, \ldots, pN+(N-1)$ where $N = \lceil height/P \rceil$ and then use an MPI `gather` operation to collect all of the values to the root process.

Which do you think is better? Why? Which intern do you promote?

HINT: Run `P5_serial.py` and watch carefully.

Load balancing with Master/Slave

In general, you may not know beforehand how best to distribute the tasks you need to compute. In the worst case, you could get a list of jobs that makes any specific distribution the worst possible. Another option when the number of jobs is much greater than the number of processes is to let each process request a job whenever it has finished the last one it was given. This is the master/slave model.

The master is responsible for giving each process a unit of work, receiving the result from any slave that completes its job, and sending slaves new units of work.

A slave process is responsible for receiving a unit of work, completing the unit of work, sending the result back to the master, and repeating until there is no work left.

Implement the Mandelbrot image computation using a master/slave MPI strategy where a job is defined as computing a row of the image. Communicate as little as possible. See `P5.py` for starter code.

HINT: Use `MPI.ANY_SOURCE`, `MPI.ANY_TAG`, and the `MPI.Status` object as shown in class.

Extra Credit

Implement Joe and Susie's approaches from the first part of Problem 5 and analyze the speedup and efficiency of all Joe's, Susie's, and the master/slave approach against the provided serial version. Use up to 192 processes and an image size of your choice. Submit the plots and a discussion of your results.

Submission Requirements

- `P4.py`: Completed master/slave Mandelbrot implementation.
- `P4.pdf`: Explanations and/or plots.
- `P4_Susie.py`: Extra – Completed Susie Mandelbrot implementation.
- `P4_Joe.py`: Extra – Completed Joe Mandelbrot implementation.