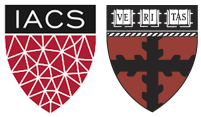
CS205, Fall 2012
Computing Foundations for Computational Science
Hanspeter Pfister, Cris Cecka



HARVARD
School of Engineering
and Applied Sciences

Homework 2 – Due October 12th, 2012 at 11:59pm

This week's homework covers Big-O concepts and more MapReduce-able problems, but implemented with MPI. This assignment will familiarize you with some of the MPI primitives and common MPI pitfalls.

Download the code for HW2 here.

or

`wget http://www.cs205.org/resources/hw2.zip`

Problem 1 - Introduction **2**
- mpi4py . . . 2
- Running MPI on Resonance Nodes . . . 2
- Job Queue and QSUB . . . 2
- Troubleshooting: . . . 3

Problem 2 - Ranking Functions [20%] **5**

Problem 3 - Hello MPI [20%] **6**
- Verify correctness and analyze speedup . . . 6
- Wait a sec... . . . 6
- Generalize and Improve . . . 6

Problem 4 - Tomography [20%] **7**
- MPI Send/Recv . . . 7
- MPI Scatter/Reduce . . . 7
- Analysis of Results . . . 7

Problem 5 - Master/Slave with MPI [40%] **9**
- Load Balancing . . . 9
- Load balancing with Master/Slave . . . 10
- Extra Credit . . . 10

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 on Resonance Nodes

The Resonance cluster is composed of 16 nodes where each node contains six hyperthreaded cores for an effective 12 cores per node. To log in to a Resonance node you can simply open the RESONANCE NODE terminal or type

```
$ gpu-login
```

from the RESONANCE HEADNODE terminal. If the login to the RESONANCE NODE fails, check if you have a hanging session by executing

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

on the RESONANCE HEADNODE where `###` is the number of the hanging job associated with your user name.

Students will be approximately equally distributed across the nodes of the cluster. Once logged in to RESONANCE NODE, you will be able to execute `mpirun` commands:

```
$ mpirun -n P EXECUTABLE
```

Using a RESONANCE 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, read the next section.

Job Queue and QSUB

Instead of logging in to a single node with only six cores, we can submit jobs to the 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 [1-32]
```

```
#$ -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 resonance cluster from the RESONANCE HEADNODE 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 RESONANCE HEADNODE 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 output dynamically. Only the `cs205hw.o###` output file is returned.

Troubleshooting:

The RESONANCE NODE terminals may also not be able to allocate you a slot into resonance. This is usually the case when you already have a slot that has not been killed. To determine if you have a already have an abandoned slot on resonance and kill it, open the RESONANCE HEADNODE and execute

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

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

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

```
$ module list
```

which should display an entry for `courses/cs205/2012`. If this is not the case, you can load it with the command

```
$ module load courses/cs205/2012
```

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

```
source /etc/bashrc
module load courses/cs205/2012
```

which forces Resonance to automatically load the module each time you log in.

Problem 2 - Ranking Functions [20%]

$\sqrt{n}$	n^2	$(\sqrt{2})^{\log_2 n}$	$\left(\frac{3}{2}\right)^n$
$4^{\log_2 n}$	$n \log_{10} n$	$(n^{1/6} + n^{1/12})(n^{1/6} + 1)$	1.001^n
2^n	12	$\log_2 n^2$	$\log_2 4^n$
e^n	$\log_2 n!$	$(\log_2 n)^2$	n^e

List the functions above in increasing $\mathcal{O}(\cdot)$ order by placing one function on each line of the table below, with the top line containing the slowest growing function and the bottom line containing the fastest growing function. If two functions are in the same group, then write those functions together on the same line.

In the next column, write the (simplest and tightest form of the) $\mathcal{O}(g(n))$ group that the functions on that line belong to. For example, the function $7n^2 - 9n$ belongs to the group $\mathcal{O}(n^2)$.

In this problem, we are not asking for proofs, just the complete table. You should assume that n is a positive integer in all cases.

$f(n)$	$\mathcal{O}(g(n))$	$g(8)$	$g(64)$
	(smallest to largest)		

Submission Requirements

- `P2.pdf`: Completed table.

Problem 3 - 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 RESONANCE 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

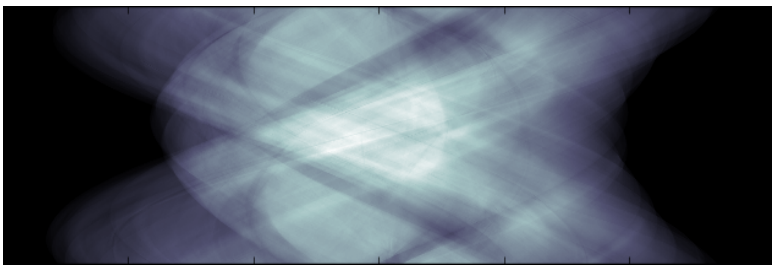
Fix the code in `P3.py` so that it works for any number of processes and any size input. Explain your approach.

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

Submission Requirements

- `P3.py`: Final implementation.
- `P3.pdf`: Explanations, plots/tables, and output.

Problem 4 - Tomography [20%]



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 n corresponds to the angle $\phi = n\pi/2048$.

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` 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

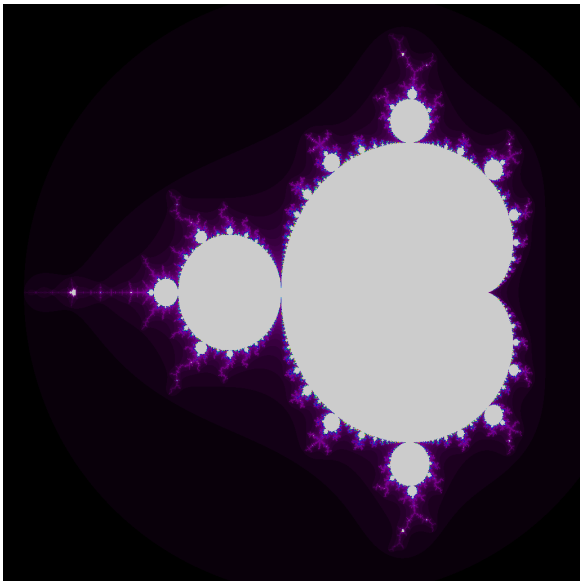
Time the scatter/reduce code with `MPI.Wtime()` counting only the communication and computation – ignore any IO, plotting, or precomputation. Compute the speedup and efficiency

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

- `P4A.py`: Completed Send/Recv implementation.
- `P4B.py`: Completed Scatter/Reduce implementation.
- `P4.pdf`: Explanations and/or plots.

Problem 5 - 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 32 processes and an image size of your choice. Submit the plots and a discussion of your results.

Submission Requirements

- `P5_Susie.py`: Completed Susie Mandelbrot implementation.
- `P5_Joe.py`: Completed Joe Mandelbrot implementation.
- `P5.py`: Completed master/slave Mandelbrot implementation.
- `P5.pdf`: Explanations and/or plots.