CS3211 - Lab 2
Accuracy, and openMPI

February 5, 2017

In this tutorial you will be investigating accuracy, and then learn how to compile and run openMPI applications. We will use the lab PCs, the teaching cluster tembusu, and an MPI implementation called OpenMPI. The objectives of this lab are to (1) investigate basic accuracy properties in normal programming languages, and (2) get familiar with using MPI.

To remind you, the activities you are doing in these labs all contribute to your project, and so you should be recording (even writing up) the results you get from the experiments.

1 Accuracy examples

We have some small C programs that have somewhat surprising behaviour. The first one sets a variable `a` to the value `0`, and then adds 1 to it, many times. To try the program:

```
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ wget http://www.comp.nus.edu.sg/~hugh/cs3211/Lab2/fpadd1.c
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ gcc -o fpadd1 fpadd1.c
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ ./fpadd1
   Adding 1 to 0, lots of times...
   Adding 500000 1s to 0 gives this result:   500000.0
   ...
```

Task 1: Compile and run the `fpadd1` program. Observe the accuracy of the answers you get. Examine the code. Examine the output. You will see a peculiarity between 16500000 and 17000000 additions. Note that you would really expect 19000000 1s added together to have the value 19000000, but it is way way out! Your goal here is to make sure you understand why this mindlessly simple program acts the way it does.

Here is a similar small C program that sets a variable `a` to the value `0.3`, and then adds 1 to it, many times. To try the program:

```
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ wget http://www.comp.nus.edu.sg/~hugh/cs3211/Lab2/fpadd2.c
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ gcc -o fpadd2 fpadd2.c
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ ./fpadd2
   Adding 1 to 0.3, lots of times...
   Adding 500000 1s to 0.3 gives this result:   500000.3
   ...
```

Task 2: Compile and run the `fpadd2` program. Observe the accuracy of the answers you get. Examine the code. Examine the output. You will see peculiarities between 1000000 and 1500000, 4000000 and 4500000 additions. Your goal here is to make sure you understand why this mindlessly simple program acts the way it does.

We have another small C program which illustrates another peculiar accuracy anomoly. In this example, the program adds up twenty (pseudo) randomly generated numbers in one order (1..20), and then in another order (20..1). To try the program:

```
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ wget http://www.comp.nus.edu.sg/~hugh/cs3211/Lab2/fporder.c
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ gcc -o fporder fporder.c
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ ./fporder
   Adding 20 (pseudo)randomly generated floating point numbers for 1 to 20 and 20 down to 1
   Sum from 1 to 20 is 276.261200
   Sum from 20 to 1 is 276.261200
```

Task 3: Compile and run the `fporder` program. Run it multiple times - each time it will choose a different set of values to add. Observe the accuracy of the answers you get. Examine the code. Examine the output. Your goal here is to make sure you understand why this mindlessly simple program acts the way it does.

Finally, we have an openMP program, which partitions an "adding" problem into threads, but each thread always adds in the same order. The program repeats each add five times. To try the program:

```
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ wget http://www.comp.nus.edu.sg/~hugh/cs3211/Lab2/fpomp.c
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ gcc -fopenmp -o fpomp fpomp.c
NUSSTU\a0xxxxxx@soctf-pdc-0yy:~$ ./fpomp
   Adding 100000 randomly generated floating point numbers using 24 threads
   Final sum is 3057532.500000
   Final sum is 3057532.500000
   ...
```

Task 4: Compile and run the `fpomp` program. Run it multiple times - each time it will choose a different set of values to add. Observe the accuracy of the answers you get. Examine the code. Examine the output. Try running the program on tembusu (use nodes between xcnd0 and xcnd14 - they have been reserved for CS3211). Your goal here is to make sure you understand why this mindlessly simple program acts the way it does.

The tasks exhibit some of the issues with the use of floating point numbers, and you should develop a clear understanding of the underlying reasons for these issues.

2 MPI

Tembusu has several MPI implementations installed. We will use OpenMPI. To use the version of OpenMPI we specify, you need to patch your local profile settings to update the PATH of the MPI utilities. Append the file `prof_cfg.txt` to your local `.profile` file. To do this, log into some node (between `xcnd0` and `xcnd14`), and execute the following commands:

```
-bash-4.2$ wget http://www.comp.nus.edu.sg/~hugh/cs3211/Lab2/prof_cfg.txt
-bash-4.2$ cp ~/.profile ~/.profile.old
-bash-4.2$ cat prof_cfg.txt >> ~/.profile
-bash-4.2$
```

Log back out (`exit`) and log back in. You can test that everything is working OK:

```
-bash-4.2$ which mpicc
   /usr/lib64/openmpi/bin/mpicc
-bash-4.2$
```

When an MPI program runs on different machines in the cluster (`xcnd1`, `xcnd2` and so on), the same mechanism used for `ssh` remote login is used. This means that to use different processors, you need to have password-free access to all the nodes in the cluster. To do this, you must set up your login to an adjacent machine so you can login without using your password. A minimal set of commands to do this might be as follows:

```
-bash-4.2$ ssh-keygen -t rsa
Generating public/private rsa key pair...
...
-bash-4.2$ cd ~/.ssh
-bash-4.2$ echo "StrictHostKeyChecking no" > config
-bash-4.2$ chmod 644 config
-bash-4.2$ cp id_rsa.pub authorized_keys
-bash-4.2$ chmod 600 authorized_keys
```

Try logging in to another machine:

```
-bash-4.2$ ssh xcnd2
```

Once you can log in remotely using `ssh`, and without using your password, you can also transfer files (using `scp`), and MPI should be able to start up on other nodes correctly.

When using Tembusu for your CS3211 experiments, you can use any nodes from the range `xcnd1-xcnd14`. The node home directories are synchronized over the network, which means when you write a file in `xcnc33's` home directory, this file is automatically transferred to the home directory of all `xcnd1-xcnd14` nodes.

3 Compiling and Running MPI Programs

An MPI program consists of multiple cooperating processes. In an MPI program, a series of MPI routine calls allow different processes to exchange information and to work together. Each MPI process has a unique identifier called `rank`. Processes are grouped into sets called communicators. By default, all MPI processes in a program belong to a communicator called `MPI_COMM_WORLD`.

Download and run the hello world example:

```
-bash-4.2$ wget http://www.comp.nus.edu.sg/~hugh/cs3211/Lab2/hello.c
-bash-4.2$ mpicc -o hello hello.c
-bash-4.2$ mpirun -np 4 ./hello
   ...
```

`mpirun` is a script that receives at least two parameters: (1) the number of processes to be created, `-np <n>`, where `n` is an integer, and (2) the path to the program binary. In this case, `./hello`

The program binary is the last parameter of `mpirun`, and command line arguments come after the name of the program.

Task 5: Run the program with 1, 4 and 32 processes and observe the output. Notice the ordering of "hello world" messages.

Next, we compile another MPI program. We will run this program using two nodes:

Task 6: Compile the program called `loc`. If you compile the program on `xcnc41` then run the program on `xcnd1` and also the remote node `xcnd2`:

```
-bash-4.2$ wget http://www.comp.nus.edu.sg/~hugh/cs3211/Lab2/loc.c
-bash-4.2$ mpicc -o loc loc.c
-bash-4.2$ mpirun -np 4 ./loc
   ...
-bash-4.2$ mpirun -H xcnd2 -np 4 ./loc
   ...
```

The `-H xcnd2` option to `mpirun` forces the program to run on that machine. The program outputs the location of each process that is started together with the processor affinity mask. MPI allows us to specify exactly where we want the processes to execute. The easiest way to accomplish this is to use a machine configuration file. This file contains a list of hostnames of the nodes where your processes will run. For example, the file `machinefile.1` specifies that process 0 starts on node `xcnd1` and process 1 on `xcnd2`. If there are more than two processes, by default `mpirun` will place the rest of processes cycling between the listed nodes in a round-robin fashion.

```
-bash-4.2$ wget http://www.comp.nus.edu.sg/~hugh/cs3211/Lab2/machinefile.1
-bash-4.2$ mpirun -machinefile machinefile.1 -np 60 ./loc
   ...
```

A more precise control of the mapping can be achieved using a machine and rank file. The rank file lists each MPI rank number, starting with 0 and tells exactly which nodes, which sockets and which cores can be used by that rank. The rank file has the following syntax

```
rank <number>=<hostname> slot=<socket_range>:<core_range>
```

For example, the rank file `rankfile.1`:

```
rank 0=xcnd1 slot=0:0
rank 1=xcnd2 slot=0:0
rank 2=xcnd1 slot=0:0
rank 3=xcnd2 slot=0:0-2
rank 4=xcnd2 slot=0:0-2
rank 5=xcnd2 slot=0:0-2
```

specifies that process 0 is mapped to node `xcnd1` on core 0 of socket 0, then process 1 is mapped to node `xcnd2` socket 0 core 0, process 2 on node `xcnd1` socket 0 core 0 and then processes 3, 4 and 5 will be started on node `xcnd2` on socket 0, and are free to be executed on either of cores0, 1 and 2. As an example of using `rankfile.1`:

```
-bash-4.2$ wget http://www.comp.nus.edu.sg/~hugh/cs3211/Lab2/rankfile.1
-bash-4.2$ mpirun -machinefile machinefile.1 -rankfile rankfile.1 -np 6 ./loc
   Process 0 is on hostname xcnd1.comp.nus.edu.sg on processor mask 0x1001
   Process 1 is on hostname xcnd2.comp.nus.edu.sg on processor mask 0x1001
   ...
```

For more details on mapping between processes and cores, consult the `mpirun` manual.

Task 7: Compile the MPI matrix multiplication program, `mm-mpi.c`. The program attempts to measure communication and computation time for a matrix-multiply operation. Try running it with 8, 16, 24, 32, 40, 48, 56, 64 processes, using multiple machines.

```
-bash-4.2$ wget http://www.comp.nus.edu.sg/~hugh/cs3211/Lab2/mm-mpi.c
   ...
```

4 Writeup

Your first project will include a write-up, with the results for the tasks in the laboratories, which explore issues related to memory, processors, and accuracy. The document must contain clear answers to each question asked in each lab. For this lab, we will be looking primarily for your clear explanations for the behaviour you observed in tasks 1,2,3,4 and 7. For each task - why did you get the results you got? To get ahead on your project, you should work on this sooner rather than later.