Parallel Programming for Multi-Core, Distributed Systems, and GPUs
Exercises

Pierre-Yves Taunay
Research Computing and Cyberinfrastructure
224A Computer Building
The Pennsylvania State University
University Park
py.taunay@psu.edu

June 2014

1 Introduction

1.1 Compute cluster

The exercises will be executed on a compute cluster, either Lion-XV or Lion-GA. Each compute node on Lion-XV features two Ivybridge Xeon CPUs E5-2670, each of them with 10 cores, and 256 GB of memory. The domain name for Lion-XV is `lionxv.rcc.psu.edu`. Lion-GA is a GPU cluster and features 64 GPUs, spread across 8 compute nodes. The GPUs are either the Tesla M2070 or M2090 model. Each compute node on Lion-GA also features two Nehalem Xeon CPUs X5675, each of them with 6 cores, and 48 GB of memory. The domain name for Lion-GA is `lionga.rcc.psu.edu`.

To connect to a compute cluster, please follow the instructions corresponding to your Operating System on RCC's website.

1.2 Modules

1.2.1 Introduction

A large amount of software has already been installed on RCC systems. You can access them through **modules**. Each module lets you add software packages to your environment. Once the software is added to your environment, you can freely access it and perform computations. To list all available modules, you can use the command `module avail`. To load a module in your environment, you can use the command `module load <software>/<version>`. You can find more information on the modules section on our website.

Remark The examples below in the next section on how to use the `module` command are for illustrative purposes. At this time, do not try to type in the `module load` or `module switch` commands.

1.2.2 Modules examples

Available software List all software installed on the system

```
module avail
```

Modules loaded List all the software currently loaded by the user

```
module list
```

MPI Loading the recommended MPI version, OpenMPI 1.6.5:

```
module load openmpi/intel/1.6.5
```

GCC Loading the GNU compiler 4.8.2

```
module load gcc/4.8.2
```

CUDA Loading CUDA 5.0 for GPU computing (only on Lion-GA)

```
module load cuda/5.0
```

Switching Switching between two versions of GCC (from some version of GCC to 4.8.2):

```
module switch gcc gcc/4.8.2
```

1.3 PBS

1.3.1 Introduction

Direct connections to the clusters let you access a **login node**. Login nodes are used to perform simple tasks, such as compiling a program, or submitting a **job** through our resource manager and job scheduler, **PBS** and **Moab**, respectively. To be able to run compute-intensive commands such as R, Python, MPI, you will have to submit a job to the queue. You can either submit non-interactive batch jobs, or interactive jobs:

- Non-interactive batch jobs: a job script that contains PBS commands to request an overall amount of processor cores, memory, walltime, etc., and the subsequent command you would like to execute on a compute node. The job is submitted using `qsub`, and is run non-interactively when resources become available.
- Interactive jobs: the command `qsub` can also be used to gain direct access to one or more compute nodes. Any command you would like to execute are then executed interactively. Such jobs are great for debugging and for short runtimes.

You can find more information on the PBS section on our website. For the rest of the exercises, we will be using **interactive jobs**. To submit an interactive job, use the command `qsub`, with the `-I` argument.

Remark 1 Once your interactive job starts, you will have to reload any modules you previously loaded in your environment.

Remark 2 The examples below in the next section on how to use the `module` command are for illustrative purposes. At this time, do not try to type in the commands.

1.3.2 Interactive job examples

One processor core request

```
qsub -I -l nodes=1:ppn=1,walltime=01:00:00 -q astro-seminar
```

will request one processor core (**ppn=1**) on one node (**nodes=1**), for one hour (**walltime=01:00:00**). The `-q` argument specifies a queue, which contains nodes that are reserved for this seminar.

MPI If you want to run MPI jobs, you will need more than one processor, across multiple nodes. You can therefore adjust the argument `ppn` and `nodes` to better suit your needs. For example:

```
qsub -I -l nodes=2:ppn=2,walltime=01:00:00 -q astro-seminar
```

will request two processor cores per node, one two nodes, for a total of 4 processor cores, and a total runtime of 1 hour. You can therefore run 4 MPI processes at the most for that particular job.

OpenMP Same as MPI jobs; you will need more than one processor. However, OpenMP jobs can only run on one machine at the most. Therefore `nodes` has to be equal to 1. For example:

```
qsub -I -l nodes=1:ppn=6,walltime=01:30:00 -q astro-seminar
```

will request six processor cores on one node, and a total runtime of 1.5 hour. You can therefore run up to 6 OpenMP threads for that particular job.

GPU GPU jobs will be run on Lion-GA. You can specify the total number of GPUs you would like to use with the argument **gpus** in `qsub`:

```
qsub -I -l nodes=1:ppn=2:gpus=2,walltime=01:30:00 -q
                    astro-seminar
```

will request 2 processor cores one 1 node, along with 2 GPUs. Note that using more than 1 node on Lion-GA is invalid.

1.4 Exercises

1.4.1 Source files

All the exercises have been uploaded to a Git repository. You will have to clone the repository, and then use the exercises source files. To clone the repository:

1. Go into your home directory: `cd ~/`
2. Clone the repository:

   ```
   git clone
   https://github.com/pytaunay/bayes-computing-astro-data.git
   ```

   This will create a directory named `bayes-computing-astro-data`.
3. Go into the newly created directory

   ```
   cd bayes-computing-astro-data
   ```
4. List the available files

   ```
   ls .
   ```
5. Examples for each lesson covered can be found in their corresponding folder. Exercises are under the folder `Exercises`.

1.4.2 Data files

The data files will be made available in a temporary directory of Lion-XV: `/tmp/astro-seminar`. To copy them:

1. Go into your home directory: `cd ~/`
2. Create a directory for the data: `mkdir data`
3. Copy all the files:

```
cp /tmp/astro-seminar/* ~/data/
```

4. Check that files are in the data directory:

```
ls ~/data/
```

2 Scalability study

Notions covered

- *Lesson 1: scalability, weak scaling, strong scaling, speed–up*
- *Lesson 2: OpenMP, MPI*

2.1 Introduction

Login to Lion-XV: the domain name is `lionxv.rcc.psu.edu`

2.1.1 Submitting a job

Please remember that resources are limited ! If you are done with your job, just hit Ctrl+D to exit the job and free resources for other users. You will have to submit the sample job provided for OpenMP and MPI only once; once your job starts, you can then run all the cases covered in the exercises. You do not have to submit multiple jobs.

OpenMP To run the OpenMP program, submit a job with up to 4 processor cores:

```
qsub -I -l nodes=1:ppn=4,walltime=02:00:00 -q astro-seminar
```

MPI To run the MPI program, submit a job with up to 4 processor cores on multiple nodes:

```
qsub -I -l nodes=2:ppn=2,walltime=02:00:00 -q astro-seminar
```

2.1.2 Compiling programs

An OpenMP and MPI implementation of the log–likelihood calculation presented in Lesson 2 are provided. They are located in the folder

```
Exercises/Scalability/OpenMP
```

and

```
Exercises/Scalability/MPI
```

respectively. Each program will have to be compiled in order to be used. A Makefile has been provided for your convenience in both cases.

OpenMP To compile the OpenMP exercise:

1. Go in the OpenMP directory

   ```
   cd ~/bayes-computing-astro-data/Exercises/Scalability/OpenMP
   ```

2. Create directories for temporary object files and the binary:

   ```
   mkdir obj

   mkdir bin
   ```

3. Run the `make` command. The program should now be compiling. Once the compilation is done, the program will be found in the `bin/` directory, under the name `omp_example`.

The Intel MKL library has to be loaded in your environment in order to use the OpenMP program. Simply load the module `mkl` with the command `module load mkl` before starting to use the program.

MPI To compile the MPI exercise:

1. Go in the MPI directory

   ```
   cd ~/bayes-computing-astro-data/Exercises/Scalability/MPI
   ```

2. Create directories for temporary object files and the binary:

   ```
   mkdir obj

   mkdir bin
   ```

3. Load OpenMPI in your environment: `module load openmpi/intel/1.6.5`

4. Run the `make` command. The program should now be compiling. Once the compilation is done, the program will be found in the `bin/` directory, under the name `mpi_example`.

The Intel MKL library and OpenMPI have to be loaded in your environment in order to use the MPI program. Simply load the module `mkl` and `openmpi/intel/1.6.5` with the command `module load mkl` and `module load openmpi/intel/1.6.5`, respectively, before starting to use the program.

2.1.3 Running a program

OpenMP

1. First load the MKL module

```
module load mkl
```

2. Go inside the "bin/" directory:

```
cd bin/
```

3. You can now run the command to execute the OpenMP program:

```
./omp_example -sx 32 -nobs 10000 -data ~/data -ns 100
```

   where

   `-sx` is the size of the considered random variables (always 32).

   `-nobs` is the total number of observations (between 1 and 1000000).

   `-data` the location of the data.

   `-ns` the total number of runs to perform to get accurate timing results.

MPI

1. First load the MKL and the OpenMPI modules:

```
module load mkl

module load openmpi/intel/1.6.5
```

2. Go inside the "bin/" directory:

```
cd bin/
```

3. Once done, you can run the command to execute the MPI program:

```
mpirun -np 2 ./mpi_example -sx 32 -nobs 10000 -data ~/data
-ns 100
```

   where

   `-np` is the total number of MPI processes to run.

   `-sx` is the size of the considered random variables (always 32).

   `-nobs` is the total number of observations (between 1 and 1000000).

   `-data` the location of the data.

   `-ns` the total number of runs to perform to get accurate timing results.

2.2 Questions

Question 1 Study the scalability for OpenMP and MPI programs provided, for various number of processor cores (1,2, and 4), and number of observations (10, 100, 1000, 10000, 100000, and 1000000). Note that the total number of observations have to be dividable by the total number of processor cores. For example, it is not possible to run 10 observations and 4 processor cores.

The following quantities should be evaluated:

a. Speed-up,

b. Strong scaling, and

c. Weak scaling.

Reminders

- The environment variable `OMP_NUM_THREADS` controls the total number of OpenMP threads.

Question 2 Discuss the effect of the overall number of cores on the total runtime for OpenMP and MPI.

3 GPU

Notions covered

- *Lesson 3: GPU architecture, GPU programming*

3.1 Introduction

Disconnect from Lion-XV, then login to Lion-GA. The domain name for Lion-GA is `lionga.rcc.psu.edu`

3.1.1 Submitting a job

Please remember that resources are limited ! If you are done with your job, just hit Ctrl+D to exit the job and free resources for other users. You will have to submit the sample job provided for the GPU only once; once your job starts, you can then run all the cases covered in the exercise. You do not have to submit multiple jobs.

To run the GPU program, submit a job with up to 1 processor core and 1 GPU:

```
qsub -I -l nodes=1:ppn=1:gpus=1,walltime=02:00:00 -q
                    astro-seminar
```

3.1.2 Compiling the program

A GPU implementation of the log–likelihood calculation presented in Lesson 3 is provided. It is located in the folder

```
Exercises/GPU
```

The program will have to be compiled in order to be used. A Makefile has been provided for your convenience. To compile the GPU exercise:

1. Go in the GPU directory

   ```
   cd ~/bayes-computing-astro-data/Exercises/GPU
   ```

2. Create directories for temporary object files and the binary:

   ```
   mkdir obj

   mkdir bin
   ```

3. Load CUDA 5.5 in your environment: `module load cuda/5.5`

4. Run the `make` command. The program should now be compiling. Once the compilation is done, the program will be found in the `bin/` directory, under the name `cuda_example`.

The Intel MKL library and CUDA have to be loaded in your environment in order to use the MPI program. Simply load the module `mkl` and `cuda/5.5` with the command `module load mkl` and `module load openmpi/intel/1.6.5`, respectively, before starting to use the program.

3.1.3 Running the program

1. First load the CUDA module:

   ```
   module load cuda/5.5
   ```

2. Go inside the “bin/” directory:

   ```
   cd bin/
   ```

3. Once done, you can run the command to execute the CUDA program:

   ```
   ./cuda_example -sx 32 -nobs 10000 -data ~/data -ns 100
   ```

   where

   `-sx` is the size of the considered random variables.

   `-nobs` is the total number of observations.

   `-data` the location of the data.

   `-ns` the total number of runs to perform to get accurate timing results.

3.2 Questions

Question 1 Study the runtime of the provided log–likelihood program, for various number of observations (10, 100, 1000, 10000, 100000, and 1000000). Compare these results to the Scalability Study section.

4 OpenMP

Notions covered

- *Lesson 1: Resource contention*
- *Lesson 2: OpenMP, shared memory, race conditions*

4.1 Introduction

Login to Lion-XV: the domain name is `lionxv.rcc.psu.edu`

4.1.1 Submitting a job

Please remember that resources are limited ! If you are done with your job, just hit Ctrl+D to exit the job and free resources for other users. You will have to submit the sample job provided for OpenMP only once; once your job starts, you can then run all the cases covered in the exercises. You do not have to submit multiple jobs.

To run the OpenMP code samples, submit a job with up to 4 processor cores:

```
qsub -I -l nodes=1:ppn=4,walltime=02:00:00 -q astro-seminar
```

4.1.2 Compiling programs

1. Go into the `OpenMP` directory:

   ```
   cd ~/bayes-computing-astro-data/Exercises/OpenMP/
   ```

2. Two sample code will be run with OpenMP. They are located at

   ```
   Code1
   ```

   and

   ```
   Code2
   ```

   respectively. To go inside one of these directories, just use the command `cd`, e.g. `cd Code1`

3. To compile an OpenMP program, please use the following command:

   ```
   gcc -std=c99 -fopenmp main.c
   ```

   This will create an executable named “a.out”.

4.1.3 Running programs

In both cases, you can simply execute the program "a.out" with the command, while being in either the `Code1` or `Code2` directory.

```
./a.out
```

Reminders

- The environment variable `OMP_NUM_THREADS` controls the total number of OpenMP threads.

4.2 Code sample 1

The result printed on the screen is intended to be equal to the one plus the total number of threads running.

Question 1 Compile the code in the `Code1` folder. Run the executable multiple times, for various number of threads (1 to 8). What happens ?

Question 2 How can you fix the code and still get the correct result ? Write as many solutions as you can.

Question 3 Run the original code through Valgrind with the helgrind tool to check for thread errors. Observe the different errors.

Note To run Valgrind with Helgrind on your "a.out" executable:

```
valgrind -tool=helgrind ./a.out
```

Remark We set `OMP_NUM_THREADS` up to 8, but have only 4 processor cores available ?! The operating system is smart enough to do "context switching", where one thread is paused while another thread is executed, if there are more threads than cores available. This allows for multiple threads to be executed on a single core. However, that method is usually detrimental to performance, since the state of a thread has to be saved, and since switching takes time. Do not do that on production code ! We are doing it here since the samples are very simple, and since our total number of cores reserved for the seminar does not allow for everyone to submit a job with 8 processor cores.

4.3 Code sample 2

This code is supposed to shift the content by 1 of a sequence of integers stored in an array. For example, the sequence $[0, 1, 2, 3]$ will be shifted to $[-1, 0, 1, 2]$. This behavior is achieved by assigning the content of the array at index $i-1$ to the index i: $X_i \leftarrow X_{i-1}$.

Question 1 Compile the code in the `Code2` folder. Run the executable multiple times, for various number of threads (1 to 8). What happens ?

Question 2 The OpenMP clause **ordered** “specifies a structured block in a loop region that will be executed in the order of the loop iterations.” The **ordered** clause allows for sequential execution of the iterations. Use the **ordered** clause to fix the issue. The output should be the same as the serial application (i.e. one thread).

5 Input / Output (optional)

Notions covered

- *Lesson 1: I/O*

It is possible in Unix to monitor the total number of voluntary and involuntary context switches, using files located `/proc/<PID>` directory. Specifically, the file `/proc/<PID>/sched` contains the lines `nr_voluntary_switches` and `nr_involuntary_switches` to monitor these quantities.

Question 1 Edit the C code with the following modifications:

- Remove all the log–likelihood calculations. Keep only the command line parser, and the data reader function.
- Add a `for` loop of size `nsample` around the data reader routine.
- Add a `sleep()` statement in the `for` loop, after the data has been parsed. The `sleep()` function takes a total number of seconds as its only argument. It results in the process going into a sleep state. You will have to include the file `sys/unistd.h` to use the `sleep()` function.

Once the edits are in, compile the code.

Question 2 Run the C code with a large number of samples, and observe the total amount of context switches.

Unix Reminders To get the process ID of a running process, you can use `top -u your_username`. The left-most column will give you the PID. Once you have the PID, you can watch the growth of the total number of context switches by outputting the content of the file `/proc/<PID>/sched`. You can do so every N seconds using the command `watch`, e.g. `watch -n 1 /proc/<PID>/sched`.

Question 3 Comment out the line that parses the data, and the `sleep` statement in the for loop. Add a long sleep statement (e.g. 20 or 30 seconds) **before** the `for` loop so you have time to get the PID of the application. Monitor the total number of context switches, and compare to the case with I/O.

6 MPI (optional)

Notions covered

- *Lesson 2: MPI, collective calls*

Problem statement The goal of this section is to both implement a matrix-vector multiplication operation (GEMV), and to obtain the L_2 norm of the resulting vector, using MPI collective routines. A code skeleton is provided in the MPI folder. The total number of processes used to run the program should always divide the number of rows of the matrix, `MSIDE`.

A possible approach is to divide the the matrix M into submatrices S of size `WORKLOAD`×`MSIDE` amongst a team of processes, where `WORKLOAD` is the total number of rows divided by the total number of processes. Each process then performs a GEMV on a smaller matrix S. Finally, results are concatenated together in a single vector.

Question 1 Divide the matrix M into submatrices S using an MPI collective call.

Question 2 Since all processes perform the product $S \times V$, send the vector V to all the processes, using an MPI collective call.

Question 3 Each process now has a subvector, result of $S \times V$. Using an MPI collective call, retrieve the data from each process, and assemble it into the result vector `mpi_res`.

Question 4 Finally, we want the L_2 norm of the resulting vector V. Partial square and sum of squares have already been provided, and stored in `sv_res`. Using an MPI collective call, calculate the overall sum of the square.

Question 5 Add timers to compare the serial solution and the MPI solution. Discuss.