Homework #4 Part C

Due: Tuesday March 3 2015 by 11:59 PM (Midnight)

Email-based help Cutoff: 5:00 PM on Mon, Mar 2 2015

Points (for Part C): 7

Note: If you are using your personal machine then prior to commencing work on this exercise, you may need to install XMing, Putty, and WinScp as illustrated in LinuxEnvironment.pdf (and shown in the videos in the Handouts folder).

You should save and rename this document using the naming convention MUid_Homework4_PartC.docx (example: `raodm_Homework4_PartC.docx`).

Objective: The objective of this exercise is to gain some familiarity with:

- Reserving time on a compute node to run interactive jobs
- Measure runtimes of programs.
- Compare performance of cache access patterns

Submission: Once you have completed this exercise, **save the file as a PDF document and upload the PDF** (with the necessary information) named with the convention *`MUid_Homework4_PartC`*`.pdf` (example: `raodm_Homework4_PartC.pdf`)

Fill in answers to all of the questions as directed. For some of the questions that require outputs to be indicated, you can simply copy-paste appropriate text from the shell/PuTTY window into this document. You are expected refer to `LinuxEnvironment.pdf` document available in Handouts folder off Niihka. You may discuss the questions with your instructor.

Name: **Yan Yu**

Preliminaries

1. Log onto the Linux server for this course via the following steps (that were covered in the previous exercises and as illustrated in the LinuxEnvironment.pdf):
   i. Run the X-Server `Xming`.
   ii. Use PuTTY to log into the Linux server `redhawk.hpc.muohio.edu`.
   iii. When you log onto the server, you will be presented with a shell (`$`) prompt. You need to perform various tasks by typing commands at the shell prompt and pressing the ENTER (↵) key.
   iv. Start `emacs` (in the background) and ensure you see the graphical screen for `emacs`.

Task 1: Request time on a compute node via `PBSpro`

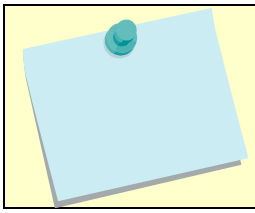
The computational infrastructure of RedHawk cluster is managed using a cluster management software system called `PBSpro`. `PBSpro` provides a suite of commands to run, manage, and remove jobs on the cluster such that each job is guaranteed to receive the requested computational infrastructure (in terms of number of processors, number of nodes, amount of memory, and run time). Refer to `LinuxEnvironemnt.doc` available off Niihka (under `Resources` → `Handouts` folder) for some additional details on `PBSpro`.

In this exercise we will request time on a compute node (1 processor/core) for performing interactive operations. This task is performed so that when time sensitive operations are performed, it is guaranteed that one core on a compute node is reserved solely for the program(s) you are running. Reservation of a CPU for jobs is critical to ensure that we obtain reasonably reliable timings for comparisons and measurements.

Accordingly, request reservation of a single compute node via `PBSpro` job management system by issuing the following command at the shell prompt:

```
$ qsub -I -X
```

In the above command `-I` option requests `PBSpro` to start an interactive session and the `-X` option instructs `PBSpro` to continue to honor forwarding of graphical windows started using the X-windows system. The `-X` option essentially ensures that `emacs` and other graphical tools will start and run properly.



In future whenever you are performing timing related experiments for performance comparison ensure you use the above command (`qsub -I -X`) to reserve a compute node for your use. Note that the process of reserving multiple compute nodes is a different and will be covered later on in this course.

Task 2: Download and review `MatrixColumnSum.cpp`

Download and save the supplied `MatrixColumnSum.cpp` program to your work folder. This program is based on the Example 2.5 discussed in your textbook in Chapter 2. This program is used to demonstrate how memory access patterns can have a significant impact on performance of program as they impact cache access, specifically cache misses.

The supplied C++ program computes the sum of all columns in the matrix in the following manner:

1. The `columnSum1` method provides the first (relatively inefficient) approach for computing sum of each column of the given matrix. The code is structured as:
    a. For each column in the matrix
        i. Compute sum of all rows in the column of the matrix

2. The `columnSum2` method provides the second (relatively more efficient) approach for computing sum of each column of the given matrix. The code is structured as:
    a. For each column in the matrix
        i. Update current sum for all columns in the current row

Task 3: Measure timings for `columnSum1` approach

1. Ensure that the `main()` function is calling `columnSum1()` function for testing. Note that the `main()` function calls the function-under-test hundreds of times. The function-under-test is repeatedly invoked a large number of times so that the overall runtime is at an acceptable (in few seconds) value so that the timing measurement is more reliable and less noisy. Noise in timings are introduced because of:
    a. The resolution of hardware clock and its interaction with the OS is usually in microseconds if not milliseconds. Consequently, if a program runs for a fraction of a second the timings will have considerable errors in them.
    b. Although a core is reserved for your job, still your task may get preempted for performing other OS related actives.
    c. Large number of repetitions ensures that such issues do not skew the timings.

2. Compile `MatrixColumnSum.cpp` (calling `columnSum1`), run, and note the program execution time (execution time is measured using Linux's `time` command; simply put `time` before any program you run to measure the time the program takes to run) using the commands indicated below:

```
$ qsub -I
$ g++ -std=c++11 -g -Wall -O3 MatrixColumnSum.cpp -o MatrixColumnSum
$ /usr/bin/time -v ./MatrixColumnSum
```

3. Run the program five times (later on we will review how to handle variance in run time appropriately) and record your observations below in milliseconds. Next, compute the average runtime using the five measurements and record it below.

Observation	Elapsed Time (milliseconds)
Run #1	7.50
Run #2	7.58
Run #3	7.86
Run #4	7.42
Run #5	7.36
Average #1 (of above 5 runs)	**7.544**

Task 4: Measure timings for `columnSum2` approach

1. Now, edit only the line in `main` method in the supplied `MatrixColumnSum.cpp` and change call to ~~`columnSum1`~~ to `columnSum2` (literally you should be changing only one character).
2. Ensure you recompile your program (as illustrated earlier) after making the changes.
3. Run the program five times (later on we will review how to handle variance in run time appropriately) and record your observations below in milliseconds. Next, compute the average runtime using the five measurements and record it below.

Observation	Elapsed Time (milliseconds)
Run #1	6.51
Run #2	6.52
Run #3	6.50
Run #4	6.53
Run #5	6.54
Average #2 (of above 5 runs)	**6.52**

Task 5: Record your inference

Using the average values computed in Task 3 and Task 4 to determine which one of the two approaches is faster and record your inference below. Provide a detailed (about 8-10 sentences) for the performance difference.

Observed from the two tables above, approach2 is faster than approach1 about 1 sec.
The reason why approach2 is faster is that in approach2 we manually removed the data dependence, since in approach1 every time when we add, we need the value from last time, but in approach2 it doesn't. So the compiler can optimize it by using SIMD feature to speed up the process to make the program execute faster.

Task 6: Submit files to Niihka

Once you have completed this exercise, save this MS-Word document (duly filled with the necessary information) **as a PDF** named with the convention *`MUid_Homework4_PartC`*`.pdf` (example: `raodm_Homework4_PartC.pdf`) and upload PDF to Niihka.