

High Performance Computing

Homework #4 Part A

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 A): 7

Submission Instructions

This homework assignment must be turned-in electronically via Niihka. Ensure that you save this MS-Word document with the naming convention `MUId_Homework4_PartA.docx` prior to proceeding with this homework. Once you have completed filling-in the necessary information in this document, **save the document as a PDF file and upload the PDF** document back onto Niihka.

Objective

The objective of this homework is to measure the performance improvements gained by resolving some of the data hazards in a straightforward program.

Task 1: Log onto the RedHawk cluster

1. First run X-Ming our X-Server. If needed, refer to `LinuxEnvironemnt.doc` available off Niihka (under Course Documents → Handouts folder) for details.
2. Use a properly configured PuTTY session to log onto the head node of the cluster, namely: `redhawk.hpc.muohio.edu`.
3. Verify that you have completed the above two steps correctly by launching an `emacs` at the shell prompt and ensuring you see a graphical `emacs` window on your local machine.



If you did not manage to complete Step 1 successfully, then stop and get help!

Task 2: Request time on a compute node via PBSpro

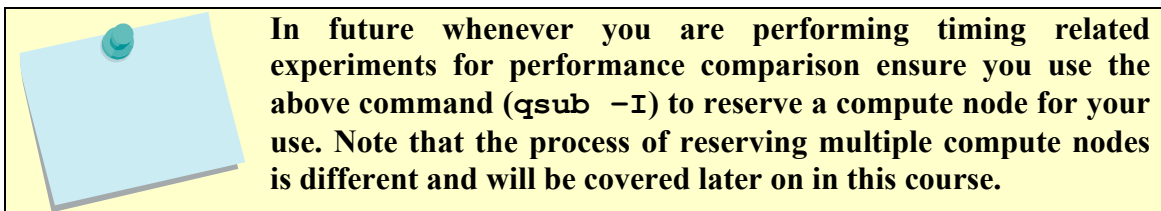
The computational infrastructure of RedHawk cluster is managed using a cluster management software system called PBSpro. PBSpro provides a suite of commands to run, manage, and remove jobs on the cluster such that each job is guaranteed to receive the requested computational infrastructure (in terms 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 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 jobs we 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
```

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



Task 3: Download and review store2load.c

Download and save the supplied `store2load.c` program in your work folder. This program was adapted from the software optimization guide provided by AMD. This program is used to demonstrate how resolving data hazards that arise due to Store-to-load dependencies can enhance performance of a program. The supplied `store2load.c` program uses a standard sequence of calculations involving 3 vectors (or arrays of values). Such transformations are commonly performed in graphical applications and many numerical simulations involving Ordinary Differential Equations (ODEs).

Open the `store2load.c` program in `emacs` and review function `approach1()`. This function provides the default/reference implementation which is typically used by almost all programmers to implement a sequence of vector operations. The exact nature of the operations being performed is not as important. What is important is to note that in the two `for`-loops in this function, there is a data dependency or **a data hazard** (specifically a Store-to-Load dependency) between `x[k]` and `x[k-1]` (yes, it does take a discerning eye to note something like that). Consequently, until `x[k-1]` is computed and written, `x[k]` cannot be computed. This introduces a stall in the pipeline operation due to a Store-to-Load dependency between `x[k]` and `x[k-1]`.

A store-to-load dependency exists when data is stored to memory, only to be read back shortly thereafter. Processors from Intel and AMD contain hardware to accelerate such store-to-load dependencies, allowing the load to obtain the store data before it has been written to memory through forwarding. However, it is still faster to avoid such dependencies altogether and keep the data in an internal register.

Avoiding store-to-load dependencies is especially important if they are part of a long dependency chain, as may occur in a recurrence computation. If the dependency occurs while operating on arrays, many compilers are unable to optimize the code in a way that avoids the store-to-load dependency. In some instances the language definition may prohibit the compiler from using code transformations that would remove the store-to-load dependency. Therefore, it is recommended that the programmer remove the dependency manually, for example, by introducing a temporary variable that can be kept in a register.

The `approach2()` function in `store2load.c` applies the above strategy to the body of `approach1()` to minimize the Load-to-Store dependency in the body of the `for`-loops by introducing a temporary variable `t`. Review `approach2()` function.

Task 4: Measure timings for approach1() in store2load.c

1. Ensure that the `main()` function in `store2load.c` is calling `approach1()` function for testing. Note that the `main()` function calls the function-under-test millions of times. The function-under-test is repeatedly invoked a large number of times so that the overall runtime is at an acceptable (in 10s of 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 activities.
 - c. Large number of repetitions ensures that caching issues do not skew the timings.
2. Compile `store2load.c` (calling `approach1()`), 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:

```
$ g++ -O2 -std=c++11 -g -Wall store2load.c -o store2load
$ /usr/bin/time -v ./store2load
```

3. Repeat the above command (that is `time ./store2load`) 5 times and note the time taken to run the same program five times in the following table:

<i>Run #</i>	<i>Elapsed Time (sec)</i>
1	12.35
2	12.15
3	11.98
4	11.96
5	12.11

4. Observe that the runtimes for exactly the same program vary with each run. This is a typical and expected behavior. However, the key question is what the correct runtime is and what value to report? The answer is that all of the runtimes are correct and each user may get a different runtime. However, to ensure that numbers are comparable you must report the mean (average) value along with 95% confidence interval (CI). The 95% CI value essentially defines a range around the average that essentially indicates that you are confident that 95% of the time, the observed execution time would lie within the range you report. In other words there is a 5% chance that the numbers you report are wrong. In practice researchers often perform 10 to 20 runs and report statistics to obtain better CI values. The confidence interval for the data must be computed using the following formulas:

Assume the five timings you have recorded are $t_1, t_2, t_3, t_4,$ and t_5 . First calculate the mean (μ) and standard deviation (σ) using the formulas:

$$\mu = \sum_{i=1}^n t_i / n \quad \text{and} \quad \sigma = \sqrt{\left(\sum_{i=1}^n (t_i - \mu)^2 \right) / n}$$

Where $n=5$ (in this specific exercise since we are using only 5 timing values). Now, from the student t-distribution tables, the 95% CI is computed using the formula:

$$95\% \text{ CI} = (\mu \pm 2.132 * \sigma) / \sqrt{n}$$

Note that the number 2.132 (aka the z-value) varies depending on the value of n . Refer to the following URL for a nice table containing various zed values:

http://en.wikipedia.org/wiki/Student's_t-distribution.

5. Compute your timing in the form $\mu \pm \text{CI}$ seconds and record it below (don't forget the unit):

Time for `approach1()`: 11.953 --- 12.342

Task 5: Measure timings for approach2() in store2load.c

1. Now, edit the `main()` function in `store2load.c` and ensure it is now calling `approach2()` function for testing.
2. Compile (using command line shown earlier with **-O2**), and repeatedly run the program (as described earlier) to obtain 5 runtimes and note the time taken in the following table:

<i>Run #</i>	<i>Elapsed Time (sec)</i>
1	6.22
2	5.97
3	5.72
4	5.88
5	5.69

3. Compute the average and 95% CI for running the program using `approach2()`. Record the timings below:

Time for `approach2()`: 5.701 --- 6.077

Task 6: Compare timings for approach1() and approach2() in store2load.c

Now compare the timings for `approach1()` (from Task 4) and `approach2()` (timings from Task 5). Based on the averages (and CI) draw inferences on the performance gained by using the two approaches by computing different timing values and record them in the table further below. Note that you must compute percentage gain using the following formulas and information provided below:

$$\% \text{Gain} = \frac{\text{Timing Difference}}{\text{Reference Timing}} * 100$$

In this exercise, the timing value for `approach1()` is the reference timing. The most performance is when `approach1()` is slowest (add CI value to `approach1()` timing) and `approach2()` is the fastest (subtract CI value from `approach2()` timing). For least performance gain use the converse of the above rationale. The average value is computed by simply using the average values.

Type of performance	Timing Difference	Percentage gain
Most performance gain observed:	<u>6.361</u>	<u>51%</u>
Average performance gain observed:	<u>6.192</u>	<u>51%</u>
Least performance gain observed:	<u>6.068</u>	<u>50%</u>

Task 7: Measuring timings with more compiler optimizations

In the previous tasks the performance of approach1 and approach2 was quantitatively measured and compared using compiler optimization level of **-O2**. This compiler optimization level is typically used for most programs as it provides a good balance between various competing factors and provides good performance. The g++ compiler is capable of performing more aggressive optimizations using the **-O3** flag. However, the **-O3** optimization may or may not improve performance (in some cases it has known to degrade performance) when compared to **-O2** optimization level.

It is time to determine if the **-O3** optimization performs better than **-O2** for the problem at hand in the following manner:

1. Edit the `main()` function in `store2load.c` and ensure it is now calling `approach1()` function for testing.
2. Compile (using command line shown earlier with **-O3** instead of **-O2**), and repeatedly run the program (as described earlier) to obtain 5 runtimes and note the time taken in the following table:

<i>Run #</i>	<i>Elapsed Time (sec)</i>
1	5.01
2	4.96
3	4.99
4	4.83
5	4.88

3. Compute the average and 95% CI for running the program and record the timing below:

Time for approach1 & -O3: 4.912 --- 5.001

4. Edit the `main()` function in `store2load.c` and ensure it is now calling `approach2()` function for testing.
5. Compile (using command line shown earlier with **-O3** instead of **-O2**), and repeatedly run the program (as described earlier) to obtain 5 runtimes and note the time taken in the following table:

<i>Run #</i>	<i>Elapsed Time (sec)</i>
1	4.88
2	4.74
3	5.02
4	5.00
5	4.99

6. Compute the average and 95% CI for running the program and record the timing below:

Time for approach2 & -O3: 4.671 --- 5.177

7. Discuss in reasonable detail (that means at least 6-9 sentences) your observations about the performance of approach1 and approach2 with -O2 and -O3 flags. In addition, provide a plausible explanation about the potential optimizations that the g++ compiler may be performing (or failing to perform) on the code.

When using -O2 and -O3 compiler flags, there is a big difference for the elapsed time between approach1 and approach2.

The reason why there is a big difference is that, in approach2, we manually removed the data dependence, so when use -O3 and -O2 flag to optimize the program, the compiler makes use of the SIMD feature to optimize the code to speed up the execution.

Task 8: Upload this document back to Niihka

Once you have successfully completed this exercise, computed and filled in all the necessary values into the document, **save this document as a PDF file and upload the PDF document** to Niihka using the appropriate submission links. Ensure you click the Submit button after you have uploaded all the necessary files for various parts of this homework.