High Performance Computing

Homework #5 (Part A)

Due: Tuesday March 17 2015 by 11:59 PM (Midnight)

Email-based help Cutoff: 5:00 PM on Mon, March 16 2015

Points: 10

Submission Instructions

This homework assignment must be turned-in electronically via Niihka. Ensure that you save the supplied `HW5Report_PartA.docx` MS-Word document as a PDF document with the naming convention `MUid_HW5Report_PartA.pdf`. Once you have completed filling-in the necessary information in the report document, upload the duly filled-PDF version of the report back onto Niihka.

Objectives

The objectives of this homework are:

- Compare and contrast the semantic of two programming languages, namely Java and C/C++
- Understand ideas underlying the development of suitable benchmarks for multi-lingual (programming languages) comparisons.

Grading Rubric:

Scoring for this assignment will be determined as follows:

- 5 points: The programs were properly compiled, run, and the data was collated correctly as per the instructions. The data was properly tabulated in the report and the graphs were plotted correct as directed.
- 5 points: The inferences from the experiments are sufficiently detailed to explain the differences in performance by explicitly highlight the salient differences between the programming languages.

Homework Exercise:

Note that for all timing experiments you are expected to be running on an independent node (and not the head node) by requesting an interactive session on a compute node. This is important to ensure that your timings are consistent with the expected values for this homework exercise.

Don't forget to request interactive sessions for testing as indicated in the instructions for each part.

Background Semantic Gap:

In a HPDC context, the term "Semantic Gap" is often used to describe the disconnect or difference between the semantics of a high level programming language (such as: Java, C++, or C) and the underlying hardware architecture on which the program is actually executed. The term is also used to refer to the distinction between the conceptual view of the microprocessor from a programmer's perspective versus the actual operation and architecture of the microprocessor. For example, a programmer (developing in a high level language) may view the cache as a single continuous high-speed memory whose locations can be randomly accessed. However, the cache (May it be L1 or L2) could be operating in pages, each page being 16 to 32 bytes in size. That is, the cache does not actually operate in terms of bytes but in group of bytes. Such a difference in the view is attributed to the semantic gap between the microprocessors architecture and its conceptual view for programming. Note that such semantic gaps are critical to ensure the necessary "separation of concerns" between the programmer and the underlying architecture. Without such semantic gaps, the programmer will be overburdened with details, preventing rapid software development. Consequently, semantic gaps are necessary evils. However, a good balance is necessary to ensure that significant performance is not traded-off to provide a marginal improvement in ease programming.

Obviously, a high-level source code cannot be directly executed by a microprocessor (that requires instructions in machine language) and the source code needs to be compiled to machine code. As a consequence of compilation, the semantic gap between the high level source code and the microprocessor is reduced but not eliminated. Even though optimizing compilers have significantly improved, there still continues to remain a semantic gap between the ways a compiler can translate high level constructs to underlying microprocessor. Furthermore, additional operations may be necessary to implement the stipulated semantics of certain language constructs.

For example, the Java Virtual Machine (JVM) is a stack-based machine, i.e., it does not have the concept of registers at all. On the other hand, all microprocessors require the use of registers and are optimized for performing operations via registers. Consequently, when a Java program is compiled to target a JVM, the Java-compiler does not optimize programs for a register-based architecture. Instead, the JVM attempts to bridge the gap between the stack-based Java byte code and the underlying microprocessor. However, since the JVM operates at the byte code level, it cannot utilize high-level language constructs for optimization. Consequently, some semantic gaps continue to remain between the JVM and the underlying system architecture. Newer JVMs (Java 1.4 and above) now include dynamic, runtime optimizations that attempt to further narrow down the semantic gap.

On the other hand, programming languages such as C and C++ are designed with the underlying architecture in mind. They are relatively very close to the actual hardware. Consequently, compilers for the C language can effectively translate high-level object code to assembly code. The assembler further optimizes the code to suit the specific architecture of the microprocessor being targeted for execution of the program with purview of the high level source code. In addition, many optimizing C/C++ compilers perform further profile driven optimization. Such aggressive optimizations enable the language and compilers to significantly narrow the semantic gap.

In the case of High Performance Distributed Computing (HPDC), the semantic gap manifests itself in two different forms that are central to HPDC, namely: computation and communication. The computation form arises when high level languages are translated to machine code as discussed earlier. The communication form of semantic gap arises when HPDC programs attempt to communicate between two or more processes running on various compute nodes on a typical supercomputing cluster. Both computation and communication are crucial components that determine the overall performance that can be realized using HPDC. Semantic gaps in communication arise due to the approach adopted by high-level languages for performing Input-Output (I/O) operations.

Languages like Java heavily rely on the use of stream oriented I/O. All I/O operation is performed via streams that are suitably mapped to the underlying hardware devices. Such stream-oriented abstractions make overall program developed easier. However, some of the semantic overheads of streams are realized at the expense of reduced performance or throughput from the underlying device. For example, in the Java programming language, a stream permits bytes of data to be written to a network card and the bytes may be delivered to the hardware in intermittent rates depending on the behavior of the application program. However, the underlying network card may require the data to be delivered as a packet for optimal performance. Consequently, the number of bytes and pattern of writing may introduce additional I/O operations between the program, the OS, and the underlying hardware. On the other hand, in programming languages like C, the I/O libraries expose many details of the underlying hardware to the programmer providing an opportunity to optimize communication. Furthermore, the programmer can suitably package the data to be compatible with the underlying hardware device. Such enhancements along with reduced computational semantic gaps can boost overall performance of HPDC programs developed to run on typical supercomputing clusters of today and tomorrow.

Computational Semantic Gap

In this part of the homework exercise you will be comparing the computational semantic gap of two programming languages, namely: Java and C/C++. For this task you will be using a standard matrix multiplication program developed in the two languages. Identical implementation of matrix multiplication in the two different languages is already supplied to you along with this homework. Your task is to conduct a number of performance measurement experiments, tabulate your findings, and draw conclusion on which one of the programming languages has a smaller semantic gap. The experiments for this part of the homework must be conducted using the following steps:

1. Download and save the attached `HW5Report.docx` to your local computer. You should save/rename this document using the naming convention *`MUid`*`_HW5Report.docx` (example: `raodm_HW5Report.doc`).
2. Next fill in the first part of the report documenting the experimental platform.
3. Now, download and save `MatMul.cpp` (C++/C language source code for matrix multiplication) and `MatMul.java` (Java source code for matrix multiplication) to your work directory on `RedHawk`.
4. In order to compile and run the Java version of the program on RedHawk you will need to use the following commands (all at the shell prompt):

a. In order to use Java on RedHawk you must issue the command:

```
$ module load java
```

You need to execute the command once, each time you log onto the cluster or issue the qsub command to obtain a session on a compute node.

b. To compile your Java source code, use the following command line (where MatMul.java is the name of the source file. If the source file is different, the you need to suitably change the name of the Java source file in the following command):

```
$ javac MatMul.java
```

c. To run the Java program and measure its run time, use the following command (where MatMul is the name of the class file obtained by compiling your source using the command shown in 4.b. If the source file is different, the you need to suitably change the name of the Java source file in the following command):

```
$ /usr/bin/time -v java -Xmx2g MatMul
```

5. Request time on a compute node to make all of the following timing measurements using the command line:

```
$ qsub -I
```

6. Now, vary MATRIX_SIZE named constant in the Java source from 500 to 3000 according to the predefined values shown in the report document. However, the REPETITIONS value must be 5 in all cases. For each MATRIX_SIZE determine the program (MatMul.java) execution timings (report just mean and 95% CI values and not the five raw timing values) and record the values in Table 1.

Don't forget to recompile your source code after each change. Be concious about the version of source you are running to ensure your measurements are consistent with the values shown in Table 1.

7. Now to measure the performance of the C/C++ version of matrix multiplication. Use the standard icpc command line shown below to compile the C version of matrix multiplication.

```
$ icpc -O2 -std=c++11 -g -Wall MatMul.cpp -o MatMul
```

8. Now, vary MATRIX_SIZE named constant in the C source from 500 to 3000 according to the predefined values shown in the report document. However, the REPETITIONS value must not be 5 in all cases. For each MATRIX_SIZE determine the program (MatMul.cpp) execution timings (report just mean and 95% CI values and not the five raw

timing values) and record the values in Table 2. In addition, compute the TPO value as described in Step 7 (earlier).

9. Using the data from Table 1, plot a line graph comparing the raw execution timings of the matrix multiplication program written in Java and C. Your chart must include trend lines along with trend formulas and regression value (R^2). **Ensure the R^2 value for your trend line is greater than 0.9**. Refer to the following online tutorial from Microsoft for plotting trendline: https://support.office.com/en-sg/article/Add-change-or-remove-a-trendline-in-a-chart-072d130b-c60c-4458-9391-3c6e4b5c5812

Discussion of Semantic Gap (in report)

Finally, using all the results from various experiments, summarize your findings in the report. Then briefly comment on the computational semantic gaps in Java and C++ in the report document. Finally, draw a conclusion on which programming language would be more suitable for HPDC programs (see additional direction in the report document).

Submission

Once you have completed filling-in the necessary information in the report document, save it as a PDF document with the naming convention `MUid_HW5Report_PartA.pdf`. Upload the duly filled-PDF version of the report onto Niihka.