High Performance Computing

Homework #3

Due: Wednesday Sept 22 2021 Before 11:59 PM (Midnight)

Email-based help Cutoff: 5:00 PM on Tue, Sept 21 2021

Points: 20

Submission Instructions

This homework assignment must be turned-in electronically via Canvas. Ensure that you save the supplied `HW3Report.docx` MS-Word document with the naming convention `MUid_HW3Report.pdf`. Once you have completed filling-in the necessary information in the report document, upload the duly filled-PDF version of the report and the SLURM script back onto Canvas.

Objectives

The objectives of this homework are:

- Compare and contrast the performance of the following three different approaches to running programs:
    - Pre-compiled (microprocessor specific): C++
    - Just-In-Time (JIT) compile: Java
    - Interpreted: Python
- Understand ideas underlying the development of suitable micro-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.
- **2 points**: The SLURM script(s) used for generating timing information was correctly coded and submitted.
- **13 points**: The inferences from the experiments are sufficiently detailed, explain the various differences in performance / efficiency, and discuss applicability in context of different applications. **This part of the report will be critically graded and only comprehensive / complete reports will be assigned full score based on the following rubric:**
    1. The document was correctly submitted in PDF format
    2. The document contained a good description of the microbenchmark used for experiments and its significance (justify why are method calls important?)
    3. The document contained information on the experimental platform used for benchmarking

4. The document provided runtime statistics from 3 independent runs for the three languages along with average values
5. The runtime statistics for argument '15' are consistent with expectations -- that is: PGO is slightly faster than the C++ version (compiled with `-O3`).
6. The report included discussion/rationale as to why PGO is faster.
7. The report discusses trends (log, linear, polynomial, exponential) in both runtime and memory footprint for the 3 languages.
8. The report includes brief mention on the semantic gap arising due to compiled, JIT compiled, versus interpreted approaches
9. The report compares and contrasts the source code in the three languages with implications on complexity of developing the micro benchmarks.
10. The report includes any language-specific peculiarities (excludes use of SLURM script) involved in running the micro benchmarks such as special command-line arguments or special operations in the languages with implications on issues in developing/running general programs on servers/datacenters etc.
11. The report includes discussion on applicability of approaches for three domains of problems, possibly from 3 different fields, namely: CPU/runtime intensive tasks (gaming / visualization), memory intensive tasks (big data), and energy efficiency (smartphones, tablets, etc.)
12. General comments/thoughts/opinions of current industry practices of teaching and using Java and Python for many applications, given concerns for energy, pollution due to energy generation, and climate issues.

Required reading: Background on Semantic Gap

In a HPC 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 Python) 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 1.8 and above) now include dynamic, Profile Driven Optimization (PGO) that attempt to further narrow down the semantic gap.

On the other hand, programming languages such as 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.

Languages such as Python pose a challenge for pre-compiling (as in C++) or Just-In-Time (JIT) compiling (as in Java) due to several unique features of the language including: ① dynamic typing of variables and parameters – that is data type of parameters is not set until a method is called, ② the data types of variables can change in the body of the method, ③ new instance variables can be dynamically added to objects, and so on. Consequently, the most commonly used Python environment uses an interpreted version called `CPython`. The primary use of Python is to facilitate automation of seamlessly interconnecting C/C++ core methods. Unfortunately, the use of Python is not limited to such usage and many massive frameworks and environments have been built all in Python. Consequently, comparison with an interpreted version of Python, the commonly used approach, is warranted.

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.

Background about the Micro-Benchmark

In this part of the homework exercise you will be comparing the computational semantic gap of three programming languages, namely: Java, Python, and C++. For this task you will be using a simple benchmark that focuses on **overheads of method calls** – the most fundamental units in a program. If a language does not support efficient method calls then building complex programs and using them for large or complex problems does not make much sense.

Accordingly, a simple, highly recursive Ackermann's function (https://en.wikipedia.org/wiki/Ackermann_function) is supplied as the benchmark. Ackermann's function was chosen because of its highly recursive nature provides a short yet scalable benchmark involving millions of methods calls. Its stack-based approach provides excellent locality of reference enabling very efficient use of instruction and data caches minimizing influence of other variables on the machine.

Instructions:

Your task is to conduct a number of performance measurement experiments, tabulate your findings, and draw conclusion on which one of the programming languages provides high performance as it has a smaller semantic gap. The experiments for this part of the homework must be conducted using the following steps:

1. You are not going to be developing any programs. The microbenchmarks are already given to you. You will be compiling programs from the terminal.
2. Nevertheless, it would be handy to use VS-Code to view and modify the benchmarks and view runtime results. In addition, you will be using the VS-Code terminal to submitting jobs.
3. Copy the microbenchmarks for this project as suggested below:

```
$ # First change to your workspace directory
$ cd ~/cse443
$ # Use ls to check if workspace.code-workspace file is in pwd
$ # Copy the microbenchmarks
$ cp -r /fs/ess/PMIU0184/cse443/homeworks/homework3 homework3
```

4. Now, briefly study the supplied Ackermann's function programs in the three different languages (of course, you can open them in VS code).
5. In order to provide a more up to date comparisons, first load the most recent version of the language tools available on the cluster:

```
$ module load gcc-compatibility/10.3.0 java/11.0.8
python/3.7-2019.10
```

6. Then use the commands for compiling the 3 microbenchmarks are shown below:
   Compile standard C++ version with optimizations

```
$ g++ -g -Wall -O3 -std=c++17 ackermann.cpp -o ackermann
```

Generate profile and create PGO C++ version:

```
$ g++ -fprofile-generate -g -Wall -O3 -std=c++17 ackermann.cpp -o
ackermann_pgo
$ ./ackermann_pgo 7      These 3 runs are used to generate a
$ ./ackermann_pgo 10     cumulative profile for the program
$ ./ackermann_pgo 13
$ g++ -fprofile-use -g -Wall -O3 -std=c++11 ackermann.cpp -o
ackermann_pgo
```

Compile and run Java version (you may have to change stack size if needed)

```
$ javac ackermann.java
```

! Note that for the Java benchmark you may have to adjust the default stack sizes to suitable values using the `-Xss` JVM option in the SLRUM script. In the above command 4m (is for 4 MB) can be changed to 8m or 12m depending on the stack size required for your program. This influences the peak memory usage and finding a suitable setting can be important.

The python version does not need to be compiled.

```
$ # No compilation needed for Python
```

☛ Note that for the Python benchmark you may have to adjust the default stack sizes to suitable values by modifying the following settings in the source code: `setrecursionlimit`.

7. You are given a starter SLURM script to run the benchmarks. You are expected to suitably modify the script. You may create separate ones for each benchmark if you find that easier (or your comment/uncomment commands). Run the benchmarks on the OSC cluster using a suitable SLURM scripts using non-interactive jobs as suggested below:

```
$ sbatch ackermann_slurm.sh
```

8. Download and save the attached `HW3Report.docx` to your local computer. You should save/rename this document using the naming convention *`MUid_HW3Report.docx`* (example: `raodm_HW3Report.docx`). Using the data from the experiments, complete the report.

Discussions (in report)

Finally, using all the results from various experiments, summarize your findings in the report. Use the discussion points in the rubric (at the beginning of this document) to develop your report.

Avoid common fallacies (recollect from CSE 262 Technology, Ethics, and Global Society):
A logical fallacy is a flaw in reasoning. Logical fallacies are like tricks or illusions of thought, and sometimes they lead to incorrect conclusions to be drawn. See for https://yourlogicalfallacyis.com/ for full list, examples, and discussions on common fallacies. Some of the common fallacies that I have observed in the past are:

- **Bandwagon**: Millions of people use Java and Python. They all can't be wrong – The fallacy is along the lines – "Millions of people smoke and do drugs. Therefore smoking and doing drugs can't be unhealthy."
- **Personal Incredulity**: I find Java and Python to be so easy. So clearly these languages are good – The fallacy is along the lines – "I find Chinese or Indian languages to be hard. Consequently, nobody in the word would be using them"
- **Anecdotal / Appeal to authority**: Architects, CTOs, and other business experts who are way above Dr. Rao's pay grade have assured me that Java and Python are best. Consequently, there must be something wrong with the data/statistics we have collected.

Submission

Once you have completed filling-in the following necessary information in the report document submit the following via Canvas:

1. Save the report document as a PDF document with the naming convention `MUid_HW3Report.pdf`. Upload the duly filled-PDF version of the report onto Canvas.
2. Upload the SLURM script that you developed to run the experiments and collect the data.