High Performance Computing

Homework #3

Due: Wednesday Sept 21 2022 Before 11:59 PM (Midnight)

Email-based help Cutoff: 5:00 PM on Tue Sept 20 2022

Name: Joshua Crail

Description

The objective of this experiment is to determine the computational semantic gap between three programming languages (Java, C++, and Python) using micro-benchmarking on the Ohio Supercomputing Cluster. The design of the micro-benchmarks is intended to determine the effectiveness of each programming language when it comes to method call (and the overhead that they can create). For determining the overheads of method calls we will be using the Ackermann's function due to is highly recursive nature to give a function that has a high scalability allowing for the millions of method calls. Overhead in method calls is important as method calls could be considered one of the most basic and important utilities that programming provides and as such if the method calls are not efficient for a language the programming language will not be able to support problem solving on a large and sufficiently complex scale.

Experimental Platform

The experiments documented in this report were conducted on the following platform (**Note**: commands to obtain the information are provided in the template version of this report):

Component	***Details***
CPU Model	Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz
CPU/Core Speed	2.40GHz
Operating system used	Linux pitzer-login04.hpc.osc.edu 3.10.0-1160.71.1.el7.x86_64 #1 SMP Wed Jun 15 08:55:08 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
Name and version of C++ compiler	g++ (GCC) 10.3.0
Name and version of Java compiler	java version "11.0.8" 2020-07-14 LTS
Name and version of Python environment	Python 3.7.4

Runtime Observations

Record the runtime statistics collated from your experiments conducted using the specified command-line arguments below. Job information should be placed in the appendix at the end of this report.

Command-line Argument	C++ (-O3)		C++ (-O3 and PGO)		Java		Python	
	Elapsed time (sec)	Peak memory (KB)	Elapsed time (sec)	Peak memory (KB)	Elapsed time (sec)	Peak memory (KB)	Elapsed time (sec)	Peak memory (KB)
10	0.05	2300	0.04	2300	0.25	46128	7.59	14664
10	0.05	2304	0.04	2300	0.26	48420	7.74	14668
10	0.05	2304	0.04	2304	0.25	42524	7.82	14668
10 (Avg)	**0.05**	**2303**	**0.04**	**2301**	**0.25**	**45690**	**7.72**	**14667**
13	2.00	2640	1.81	2840	5.14	49752	650.64	64860
13	2.01	2644	1.80	2840	5.10	43536	634.09	65112
13	2.00	2644	1.80	2840	5.04	45452	638.47	65068
13 (Avg)	**2.00**	**2642**	**1.80**	**2840**	**5.09**	**46246**	**614.07**	**65013**
15	31.34	4348	29.70	5260	80.90	39036	15654	249455
15	32.16	4348	29.71	5256	80.08	68428	15712	249645
15	31.34	4348	29.70	5256	79.97	54808	15678	250970
15 (Avg)	**31.61**	**4348**	**29.70**	**5257**	**80.32**	**54090**	**15681.33**	**250023**
16	127.12	6628	124.56	8444	530.68	61792	Optional	Optional
16	126.92	6624	124.31	8444	530.05	55820	Optional	Optional
16	126.69	6624	124.34	8444	534.11	54736	Optional	Optional
16 (Avg)	**126.91**	**6625**	**124.40**	**8444**	**531.61**	**57449**	Optional	Optional

Inferences & Discussions

Now, using the data from tables and the discussion points from the project grading rubric, document your inferences in the space below. Ensure you address all of the 12 aspects mentioned in the grading rubric. The discussion will be critically graded and only the comprehensive / complete discussions will be assigned full score.

Preliminary Findings:

From the above results, we can see that C++ is clearly faster than Java and Python by quite a large margin. For the trial with the command line argument at 15 we can see that C++ is 2.7 times faster than Java and over 500 times faster than Python. With the reading that was provided this makes sense as C++ is the closest to the hardware in the sense that they are built with the underlying architecture in mind. C++ can interact with the underlying architecture much more efficiently than the others, which can be improved with PGO or profile-guided operation to give C++ even more efficiency than just -O3 optimization. It makes sense that PGO would make the operation of the Ackermann function faster because PGO allows for the program that is being run to be tested and allow for profiles to be generated that then can be used to enhance the optimization process even further.

Data Trends:

From the above results we can see that for the C++ with just -O3 the trend seems to indicate that the trend is exponential for both the run time and the memory footprint. For C++ with both -O3 and PGO it seems that it is again exponential, but you can see in the data that as the number of arguments increase the time is slightly less than just -O3 so we can reason that the exponential growth of -O3 with PGO is less than the one with just -O3. For the memory footprint that growth seems the same for both 10 and 13 but after that -O3 with PGO grows at a larger exponential rate than the trial with just -O3, but overall they are still close when compared to the memory footprint of the others (appendix B graph 1). For Java the results are the same exponential growth for the run time but at a much more aggressive rate than the other trials mentioned so far. One interesting note though is that the memory footprint for Java seems to be semi consistent which would make sense as all the memory is allocated at the start due to the fact that JVM has to allocate everything onto a stack before executing the function. For python is can be seen as the worst case its exponential growth is much more rapid than the other previously mentioned trials in its run time. Interestingly Python initially has a smaller memory footprint than Java as Java has to put everything onto the stack, but once the Ackermann function goes over 13, Pythons memory foot print jumps eclipses Java (appendix B graph 2) and we can assume if the trend holds it would exponentially increase again with Ack 16 as well. The high growth that was noted makes sense though as Python has so many issues that bog it down such as, dynamic typing meaning variables can change on a whim. To account for dynamic typing Python is not a compiled language and as such everything will have to be interpreted to usable code for the underlying architecture when being ran which massively effects performance, which graph 1 in the appendix clearly shows, even with the time scale being logarithmic, that the time python takes sticks out from the others.

Semantic Gap:

There is a clear semantic gap for all of these languages more for some like python or less for ones like C++. The semantic gap present in C++ is so low because it has the underlying architecture in mind and as such programs are made to be the most optimized with smooth integration with the architecture. The efficiency of C++ comes from that face that the compilers will take the code and with turn that into assembly which is optimized by the assembler to turn out assembly code that fits the underlying architecture very efficiently, this efficiency or narrowing of the semantic gap can be increased even more with PGO allowing for profile to be given before hand to better optimize the program before execution. Java uses a JVM to allocate everything to a stack before executing (appendix B graph 2 Java has a consistent stack size) which is great for Java's portability but it causes issues with the semantic gap. With the JVM being stacked based registers that microprocessors use so well are not accounted for in Java, as such the semantic gap needs to be filled and that is done by Java compiler, but that filling takes time and resources to do. Python with its dynamic typing results in it being not a compiled language that needs everything to be interpreted to usable information for the underlying architecture at run time, resulting in interpretation when it is needed meaning that stuff has to be constantly loaded in and potentially even changed if there is a type change meaning that it is very slow to run (and causes a large memory footprint) for the underlying architecture to use.

Source code and language quirks :

Starting with C++ the only necessary file is the Ackermann.cpp file and to run with just -O3 there are no real restrictions in place. When trying to use PGO with C++ there is the restriction that a profile is needed, and the program has to be run a few times so a profile can be generated. In both cases of C++ there is no real quirk of the language than needed to be over come to run it. Java is a bit more restrictive in how it is used especially given that it needs a .class file and need to compile it so that it can run. There is one quirk that being that you need to set the stack size manually when doing the slurm and that can be mostly trial and error that is dependent on what is trying to be ran, which could lead to issues if this needs to be run on a server as the ideal stack size will have to be found. Python has one of the odder quirks as in the python file the system recursion limit has to be changed so that the python application can actually run. One peculiarity that Java and Python share is that before doing the slurm the versions of the software to be checked to make sure it is on the latest version whereas C++ does not have that issues as the version is set when the file is compiled with -std=c++17. Overall, the source code is the same for the three languages with any differences being down to language syntax (except for having to set the recursion limit in Python).

Applicability :

When looking as the applicability of these approaches they should be able to be used effectively and efficiently in real world domains such as smartphones, big data problems, and AI/ML CPU intensive tasks. The biggest issue is that I would personally ague that C++ is the best for all of these applications it consistently uses the underlying architecture most efficiently with a relatively small memory footprint. If I wanted to train a machine learning algorithm for facial recognition, I would want C++ for training such an algorithm as it is intense on the CPU and it can take a long time and other languages would only compound that time making it take far too

long and use up far too many resources which is the same issue in big data applications. Smartphones are unique as they have a very limited set of resources and have a limited operating time due to the fact that they are battery powered as such why would I want a language that will take all of my resources and take forever to work? C++ even in a smartphone application seems like a good idea to me, as the alternatives have only shown the number of resources they hog and the excessive time they take to reach the same solution as C++.

Appendix A

Copy-paste the SLRUM job script that you used to collect the runtime statistics in the space below:

SLURM job script to collect run time stats (C++ and Java):

```
#!/bin/bash

#SBATCH --account=PMIU0184
#SBATCH --job-name=hw3_cpp_java
#SBATCH --time=03:00:00
#SBATCH --mem=4GB
#SBATCH --nodes=1
#SBATCH --tasks-per-node=1

# To keep the benchmarks more upto date, we load the more recent
# versions of the tools available on the cluster
module load gcc-compatibility/10.3.0 java/11.0.8 python/3.7-2019.10

# Function to run given command 3 times
function run3() {
    cmd="$*"
    for arg in 10 13 15 16; do
        echo "-----------------------------------------"
        echo "Running 3 reps of ${cmd} ${arg}"
        for rep in `seq 1 3`; do
           /usr/bin/time -v ${cmd} ${arg}
        done
    done
}

# This script assumes the benchmarks have been precompiled.

echo "================================================"
run3 ./ackermann

echo "================================================"
run3 ./ackermann_pgo
```

```
echo "============================================="
run3 java -Xss24m ackermann
# End of script
```

SLURM job script to collect run time stats (Python Only):

```
#!/bin/bash

#SBATCH --account=PMIU0184
#SBATCH --job-name=hw3_cpp_java
#SBATCH --time=06:30:00
#SBATCH --mem=4GB
#SBATCH --nodes=1
#SBATCH --tasks-per-node=1

# To keep the benchmarks more upto date, we load the more recent
# versions of the tools available on the cluster
module load gcc-compatibility/10.3.0 java/11.0.8 python/3.7-2019.10

# Function to run given command 3 times
function run3() {
    cmd="$*"
    # Removed 16 because it takes too long and would have to be ran separately
    for arg in 10 13 15; do
        echo "-------------------------------------------"
        echo "Running 3 reps of ${cmd} ${arg}"
        for rep in `seq 1 3`; do
           /usr/bin/time -v ${cmd} ${arg}
        done
    done
}

echo "============================================="
run3 python3 ackermann.py
# End of script
```

Appendix B

Supplemental Graphs for better visualization