Performance Engineering Coursework 1: System Profiling

Revision 5

Objective

The goal of this exercise is to practice your performance engineering analysis and tuning skills on a large, complex system supporting an unfamiliar application. We do not focus on a microarchitectural analysis but on the identification of pieces of code that are important for performance at an application level and the extraction of a meaningful microbenchmark for a Java application. This vastly increases the number of analysis tools at your disposal: VTune in general exploration mode, VisualVM, JProfiler, Java Mission Control and many more. With the help of your favorite analysis tools, you should identify the most performance-critical pieces of code of the application and focus your attention on those.

We use the Apache Flink data processing framework as our target.

Getting Started

Log in to your favorite lab machine and run the instructions there.

Cloning your labts repository

You will be submitting the various artifacts through labts, so please check out that repository. In addition to code, you shall also submit your report as text in the `Report.txt` file in the root of the LabTS repository.

The length of the report must not exceed 500 words

Setting up the java environment

Flink requires Java 8 to build. To make sure that you are running java 8 add the following lines to your ~/.profile file:

```
export JAVA_HOME=/usr/lib/jvm/oracle-java8-jdk-amd64
export PATH=$JAVA_HOME/bin:$PATH
```

Afterward, you should log out and log in again. To ensure that you are running the right version run

```
java -version
```

You should get this output

```
java version "1.8.0_271"
Java(TM) SE Runtime Environment (build 1.8.0_271-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.271-b09, mixed mode)
```

Building Flink

- pick a lab machine (and stick to it)
- if you run into weird problems check if someone else is working on the same machine: `ps auxw | grep java`

```
# download flink
mkdir -p /data/$USER/
cd /data/$USER/
curl -L https://apache.mirrors.nublue.co.uk/flink/flink-1.12.1/flink-1.12.1-src.tgz | tar xz

# pulling flink dependencies requires more space than your allotted quota, store in /data
mkdir -p /data/$USER/m2
rm -rf ~/.m2
ln -s /data/$USER/m2 ~/.m2

# build flink
cd flink-1.12.1
mvn clean install -DskipTests -Dfast -Dskip.npm
```

Running the Flink server

Check that nobody else is running on the machine:

```
pgrep java
```

should be empty

```
cd /data/$USER/flink-1.12.1/flink-dist/target/flink-1.12.1-bin/flink-1.12.1
./bin/start-cluster.sh
```

Check that it succeeded

```
pgrep java
```

should show two processes

Running the Flink benchmark

As subject of our benchmark, we use one of the example programs that come with the flink distribution. It implements the K-Means clustering algorithm (see Wikipedia if you are interested but it might be more fun if you learn about the algorithm by profiling the program). To run the program, you need to perform two steps

- Step 1: Get the data I have pre-created a dataset for you to download

```
cd /data/$USER/
curl -sL http://www.doc.ic.ac.uk/~hlgr/data.tar.gz | tar xvz
```

- Step 2: Run the program

```
cd /data/$USER/flink-1.12.1/flink-dist/target/flink-1.12.1-bin/flink-1.12.1
bin/flink run examples/batch/KMeans.jar --points /data/$USER/points.ssv --centroids /data/$USER/centroids.ssv --output
↪  $(mktemp -u)
```

Task 1: Exploration to focus optimization effort

Flink is a complex processing system and, therefore, has significant potential for performance improvement in many sections of the code. In the course of this exercise, we want to restrict ourselves to "local" improvements and treat the rest of the system as a black box.

To identify sections of code that are critical to the provided KMeans program, spin up a profiler of your choice and attach it to the flink worker process[1]. **Be careful what process you attach to:** Flink has a co-ordinator process running as well as a worker process (both run as process 'java'). We are interested in the worker process, the one running `org.apache.flink.runtime.taskexecutor.TaskManagerRunner`. To make sure you profile the right process in vtune, select "Attach to Process" in the "WHAT" section of the "Configure Analysis" dialog, select "PID" and press the "SELECT" button. In the displayed list, you search for "TaskManagerRunner" and double-click the process. That should fill the "PID" field.

After you start/attach the trace, run the k-means program from a terminal and stop the profiling run after the process on the terminal has finished (you could run it a couple of times for good measure).

Now you want to shut down the flink server

```
cd /data/$USER/flink-1.12.1/flink-dist/target/flink-1.12.1-bin/flink-1.12.1
./bin/stop-cluster.sh
```

To complete the task, analyze the profile to in your profiler to identify the hot sections of code (the ones comprises the majority of the runtime). Report your findings in the `Report.txt` file. State what the hot section is and (briefly) how you determined it.

30 percent of the marks are awarded for Task 1

Task 2: Extracting a microbenchmark

Your second task is to extract a microbenchmark for the hot path of non-framework code (as mentioned, we are interested in code that can be optimized without affecting the entire system). If you are in doubt if you have identified the right function, the containing class's name has 19 characters and the functions return type 22 (including generics and spaces.)

You shall implement the microbenchmark in the *Java Microbenchmark Harness* (JMH) framework. JMH is similar to google benchmark. You can find a good tutorial at `http://tutorials.jenkov.com/java-performance/jmh.html` while the official page is `https://openjdk.java.net/projects/code-tools/jmh/`. For your convenience, I have pre-generated a benchmark project in `java-benchmark` in your LabTS repository. You can run it by installing maven (and java), `cd` ing into the directory and running (JMH has a lot of options to influence how it runs the benchmark but this combination works well for our case):

```
mvn install && java -jar target/benchmarks.jar -w 1 -wi 1 -r 1 -f 1 -i 5 -bm avgt -tu ns
```

}To create a meaningful microbenchmark, feel free to copy pieces of the relevant code into your microbenchmark program (the point of this part of the task is not for you to be creative). I have added flink as a dependency to the JMH project so you can just import things using, e.g., IntelliJ.

[1] If you stick to VTune, you might want to run it from your laptop. Other profilers might be best run directly on the lab machine you are working on

Evaluate the microbenchmark and note down your findings in `Report.txt`. Establish a hypothesis (a simple characteristic equation) describing the impact a parameter might have on the execution time. Develop a microbenchmark that sweeps meaningful values for your parameter [2]. To validate or refute your hypothesis. Note that it is fine if you end up refuting your hypothesis. It is better the refute an interesting hypothesis than validate a boring one.

30 percent of the marks are awarded for Task 2

Task 3: Optimization through native code

Your next task is to contrast the performance of the Java implementation of the critical section to one in C++ (or C). For that purpose, your LabTS project contains a `cpp-benchmark` project. Build the code as usual:

```
mkdir Release
cd Release
cmake .. -DCMAKE_BUILD_TYPE=Release
cmake --build .
```

Run it like this

```
./Benchmark
```

To complete the task, reimplement the java benchmark in the file `Benchmark/Benchmarks.cpp` using the google benchmark framework you know (and love: `https://github.com/google/benchmark`). Place your the code you are going to benchmark (i.e., the re-implementation of the flink code) in `Source/Implementation.hpp` and/or `Source/Implementation.cpp` (similar to separating tests and implementation). In this section, you may lose marks for bad code (you do not have to adhere to any C/C++ coding guideline but you want to avoid things that are bad in any language: unclear or spaghetti code, uninitialized variables, poor structure, etc. Remember: there are people on the other side that have to read your code and you do not want to upset them. Note that even in C, you should structure data in `structs`.

Evaluate the microbenchmark and note down your findings in `Report.txt`. Focus on a comparison of the performance of the Java and C++ implementation.

40 percent of the marks are awarded for Task 3

Bonus Task: Testing

To ensure correctness of your implementation you should also write at least one meaningful test for your C++ implementation in `Test/Tests.cpp`.

You can earn up to 20 percent bonus marks are awarded for this task that can help you make up for marks lost in other tasks (naturally, you cannot achieve more than 100% of the marks in total).

Submission

The entire submission, including the Report is performed through LabTS.

[2] see `https://github.com/openjdk/jmh/blob/master/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_27_Params.java` for an example on how to sweep parameters

The entire length of the report (all three tasks) must not exceed 500 words You may note that our labts script truncates the report after 500 words. We include the rest in the report as well but you will lose marks (and goodwill) if you exceed the limit.

Happy coding!