Lab Assignment
DV1567: Performance Optimization

David Holmqvist <daae19@student.bth.se>

Lab Description

The *DV1567* lab assignment consists of the following tasks:

- learn how to get useful information about your system with regard to performance and scalability
- learn how to use the performance monitoring tools available in Linux (`sar`, `top`, `ps`, `gprof`, `valgrind`)
- practice measuring and analyzing the performance and scalability of your system under different workloads

At the end of the lab, your group should produce a report describing their experience and findings (more instructions will follow).

To prepare yourself for the lab, read the following material:

Skimming and goal-oriented reading is necessary since the resources contain a lot of content

- the man pages for `sar`, `top`, `ps`, `gprof` and `valgrind`
- the user manual for `valgrind`
- the handbook for `kcachegrind`

You'll be asked to produce plots of your performance measuring reuslts, which means that you'll need to use some tool for that purpose; there is Excel (and Google Sheets), but you can use any tool you want as long as it produces plots that are easy to read and understand. One other such tool is `GNU plot` for example.

Task 1: Information about the SUT

SUT: System under Test

The goal of this task is to learn how to get information about the hardware characteristics of the system you'll be using as a testbed.

There are many Linux commands that allow you to get info about system characteristics; one of them is `lshw`. Answer the following questions:

1. What Linux commands did you use to obtain information about your SUT?
2. Which characteristics are useful to know about for the later tasks?[1]
3. Give a concise and relevant description about your SUT (omit technical information that you deem irrelevant with regard to performance.).

Task 2(a): General performance monitoring

Linux commands such as `sar`, `top` and `ps` allow you to monitor the system performance counters (or performance metrics), the state of the processes running on the system, and the number of resources used by the processes running on the system. Use the above-mentioned tools to answer the following questions (write your answers in the report and for each answer also provide an example documented with e.g., screenshots or textual output from Linux commands):

1. What are the performance metrics/counters that you can use to detect memory leaks?
   (a) What are the commands and related flags to enable the collection of those metrics/counters?
2. What are the performance metrics/counters that you can use to analyze the CPU utilization?
   (a) What are the commands and related flags to enable the collection of those metrics/counters?
3. What are the performance metrics/counters that you can use to analyze disk throughput?
   (a) What are the commands and related flags to enable the collection of those metrics/counters?
4. How can you sample performance counters each 2 seconds? Give an answer for all the three tools.
5. How can you automatically stop the sampling after 3 minutes? Give an answer for all the three tools.
6. Is it possible to save the sampled data into a file and load it again for further analysis? Give an answer for all the three tools; if a command option is not available, propose a solution.
7. How can you sample statistics for a specific processor?
8. How can you sample statistics for a process/thread or group of processes/threads?[2]

[1] Note that you might not know this information until you've completed the later assignments.
[2] The use of pipes and `grep` in combination with a monitoring command can turn useful.

Task 2(b): Performance monitoring of C++ programs

Linux commands such as `gprof` and `valgrind` allow you to monitor the performance of specific applications in a more fine-grained manner than is possible with the tools presented in task 2(a).

As part of the assignment is distributed a folder containing three C++ applications:

1. `log_analyzer`

   There are three variations of this application:

   (a) `analyze_log`

   (b) `analyze_charbuf`

   (c) `analyze_pm_hash`

2. `file_reader`

   There are four variations of this application:

   (a) `file_reader`

   (b) `file_reader_parsimonious`

   (c) `file_reader_sgetn`

   (d) `file_reader_read`

3. `matrix_multiplication`

   There are three variations of this application:

   (a) `naive`

   (b) `naive_parallel`

   (c) `strassen_parallel`

Begin by looking at the Makefiles and compiling the programs. Every program expects at least a command-line argument standing for the amount of times the application logic is to be run (*test_runs*).

For the `matrix_multiplication` program, the *dimension* of the matrices is always expected. The `naive_parallel` implementation however also expects the amount of *threads* to spawn. Running any of the programs with less than the expected command-line arguments prints a usage string.

Use either `gprof` or `valgrind`(possibly in combination with `kcachegrind`) to answer the following questions for each of the applications and their variations(write your answers in the report and for each answer also provide an example documented with e.g., screenshots or textual output from Linux commands):

1. Execute and monitor the application (all variations) with *test_rounds* set to 32; present your results.[3]

[3]Feel free to set *test_rounds* to values higher than that. For applications that complete relatively quickly (`log_analyzer` and `file_reader` for example) you are highly encouraged to do so. For heavy computations such as `matrix_multiplication`, going higher than 32 is tedious.

2. Present your hypothesis about what the hotspot of the application is.

3. Use the results of your monitoring activities to motivate your hypothesis.

4. Moving from one variation to the next, certain optimizations are performed to increase the performance of the application. For each step (e.g. *variation_1 – variation_2*) present your hypothesis about what kind of change could have caused an increase of the performance (better CPU utilization, less calls to the memory allocator, etc.).

5. Use the results of your monitoring activities to motivate your hypothesis.

In addition to the above, there are special instructions for the `matrix_multiplication` application and its variations:

1. For each of the three variations, use 64, 128, 256, 512, 1024, 2048 and 4096 as the *dimension* parameter; for the `naive` variation however, it is not mandatory to use more than 1024 since execution times become very long. Different values of *dimension* in this case simulate different *workloads* for the application.

2. For the `naive_parallel` variation, use 1, 2, 4, 8, 16 and 32 as the *threads* parameter.[4]

[4]Note that this means that you'll have to set *threads* to 1, 2, ... 16, and 32 for *dimension* set to 64, then for 128, then for ... etc.