Lab 1
Introduction to Performance Instrumentation and Report Writing

CS3210 – 2024/25 Semester 2

Learning Outcomes

1. Learning some measurement techniques for quantifying performance of programs.
2. Learning how to do basic performance instrumentation and profiling
3. Learn the fundamentals of technical report writing

You can obtain 2% of your grade in CS3210 by submitting your work at the end of the lab. Full marks can be obtained for submissions that have minor inaccuracies.

Logging in & Getting Started

For the lab and assignments, you are going to be running your code remotely on the machines in the Parallel and Distributed Computing Lab located in COM1-B1-02. **Use the following instructions to connect to the lab machines remotely over** `ssh`.

Follow: https://www.comp.nus.edu.sg/~cs3210/student-guide/accessing.

For this lab, **connect to one of the machines using the guide above, and start working on completing the tasks in the lab**. The lab files can be found here: https://www.comp.nus.edu.sg/~cs3210/L1_code.zip. You can use the command "`wget`" to download the code to the lab machine, and "`unzip`" to unzip the file.

Part 1: Performance Instrumentation

In this course, we care a lot about extracting the maximum performance from the programs that we write. In order to do this, we need to carefully analyze on the performance of the code during its runtime.

For Lab 1, we will be working with two programs that sum an array serially. The programs are `sum_stride_1.cpp` and `sum_stride_64.cpp`.

- Example to compile one of the programs in a terminal (console):

  ```
  > g++ -o sum_64 sum_stride_64.cpp
  ```

- Run the program in a terminal:

  ```
  > ./sum_64
  ```

The provided programs should each add all the numbers within a generated array and output the sum; the two differ in how the elements of the array are accessed.

Exercise 1

Study the two files and understand the difference of the array access patterns across the two programs. Compile and run both programs. Ensure that the outputs are the same.

time

One of the most basic forms of program performance analysis is measuring how long it runs. In most Linux system, there is a `time` utility program in `/usr/bin/` which gives the timing statistics on the program run.

- Get the timing statistics of the program you have compiled:

  ```
  > /usr/bin/time ./sum_64
  ```

- Get more interesting details by using the verbose (`-v`) flag:

  ```
  > /usr/bin/time -v ./sum_64
  ```

Exercise 2

Use `time` to see the difference between the runtime of the two programs. Which runs slower, and how can you be sure your run was not a random anomaly? What is the difference between **user** time and **system** time? Run `time` with the verbose option to see more specific details about the difference between the two programs. What do you observe? You may want to try running `man time` to read the documentation.

In some shells (e.g. bash, zsh), there might also a built-in `time` utility. This section of the lab concerns ourselves with `/usr/bin/time`, although you are free to use your shell's `time` utility in your work as well (though it may not provide as much information!)

perf stat

The `time` program provides us a good start in seeing how the program runs; however, it doesn't measure lower-level hardware performance statistics that other tools can do.

Due to the multiple layers of abstraction in modern high level programming, it is sometime hard to understand performance at the hardware level. For example, a single line of code e.g., `result[i][j] += a[i][k] * b[k][j];` typically translates into a quite a few machine instructions. In addition, this statement can take a wide range of execution times to finish depending on cache/memory behavior.

Hardware event counters are special registers **built into modern processors** that can be used to count low-level events in a system such as the number of instructions executed by a program, number of L1 cache misses, number of branch misses, etc. A modern processor such as a Core i5 or Core i7 supports a few hundred types of events.

In this section, we will learn how to read hardware events counters to measure the performance of a program using **perf**, a Linux OS utility. **perf** enables profiling of the entire execution of a program and produces a summary profile as output. We will use `perf` a lot!

- Use **perf stat** to produce a summary of program performance (collecting a default set of values)

```
$ perf stat -- ./sum_1
$ perf stat -- ./sum_64
```

- To count events of interest, you can specify exactly which events you wish to measure with the `-e` flag followed by a comma-delimited list of event names (this one focuses on L3 cache behavior):

```
$ perf stat -e LLC-loads,LLC-load-misses,cycles,instructions -- ./sum_1
```

- You can even ask `perf` to repeat measurements multiple times to see the variation (try it!):

```
$ perf stat -r 3 <command>
```

- You can list all events available on your platform with the command:

```
$ perf list
```

Perf Event Sampling

If you ask `perf` to sample too many events (i.e., too many arguments after `-e`), it will not give you an *exact* count for all of those events. Instead, it will *sample* the events and give you an estimate. This is because the hardware event counters are limited in number, so the hardware cannot count all events at the same time. The number of available counters depends on the CPU being used.

Exercise 3

Use `perf stat` to see the differences between the execution of the two programs. What do you think is the main difference between the two programs in terms of hardware behavior? Experiment with different events to try to understand the behavior of the programs further. There's a very nice reference here https://perfwiki.github.io/main/tutorial/ for learning more about perf.

perf record → perf report

Often, we need to understand not only the performance of our program in hardware, but also *which parts of our code contributed to the hardware's behavior* (e.g., cycles, or cache misses). This brings us to the idea of profiling and dynamic instrumentation, where we use tools to measure performance at a finer level.

perf record records the performance of a program into a `perf.data` file, and **perf report** reads the file and produces a report indicating which parts of the program took the most time (or other things, depending on the event specified in perf record).

- So that later steps can give you better debugging info, recompile the program with the `-g` flag:

```
$ g++ -g -o sum_1 sum_stride_1.cpp
```

- To record the performance of a program, use `perf record` followed by the program name:

```
$ perf record -- ./sum_1
```

- This, by default, profiles your program by the number of **cycles** each line takes. To see the report, use `perf report`:

```
$ perf report
```

```
Samples: 12K of event 'cycles', Event count (approx.): 9032988712
Overhead  Command  Shared Object        Symbol
  49.34%  sum_1    libc.so.6            [.] __random
  11.09%  sum_1    sum_1                [.] std::__fill_a1<int*, int>
   8.40%  sum_1    sum_1                [.] sumArray
   7.60%  sum_1    libc.so.6            [.] __random_r
   5.24%  sum_1    sum_1                [.] generateLargeArray
   4.07%  sum_1    sum_1                [.] std::vector<int, std::allocator<int> >::size
   3.58%  sum_1    sum_1                [.] std::vector<int, std::allocator<int> >::operator[]
   3.23%  sum_1    sum_1                [.] std::vector<int, std::allocator<int> >::operator[]
   1.38%  sum_1    libc.so.6            [.] rand
   1.11%  sum_1    [kernel.kallsyms]    [k] __irqentry_text_end
   0.77%  sum_1    [kernel.kallsyms]    [k] error_entry
   0.33%  sum_1    [kernel.kallsyms]    [k] native_irq_return_iret
   0.25%  sum_1    [kernel.kallsyms]    [k] sync_regs
   0.23%  sum_1    [kernel.kallsyms]    [k] cgroup_rstat_updated
   0.18%  sum_1    [kernel.kallsyms]    [k] clear_page_erms
```

Figure 1: Example output of `perf record` and `perf report` on the `sum_1` program (yours may differ)

The report might look confusing at first, but it's much simpler than you expect. The numbers on the left are the percentage of all program cycles taken by each "symbol" of code (i.e., the functions in your program).

The "shared object" column tells you where the function call is from. `sum_1` indicates function calls from the code you wrote (including some C++ standard library calls), `libc.so` indicates calls from the C standard library, and `kernel.kallsyms` shows calls within the Linux kernel. Typically, you just have to look at the `sum_1` section, since that is the code you wrote. The "symbol" column shows the function names, and no, you don't have to know anything that's not outside your own program.

However, we can go deeper into the performance of the code, since we compiled with the `-g` flag. To see the performance of each line of code, you can use the arrow keys to highlight the `sumArray` line, and press ENTER twice to enter the "annotation" of the sumArray function.

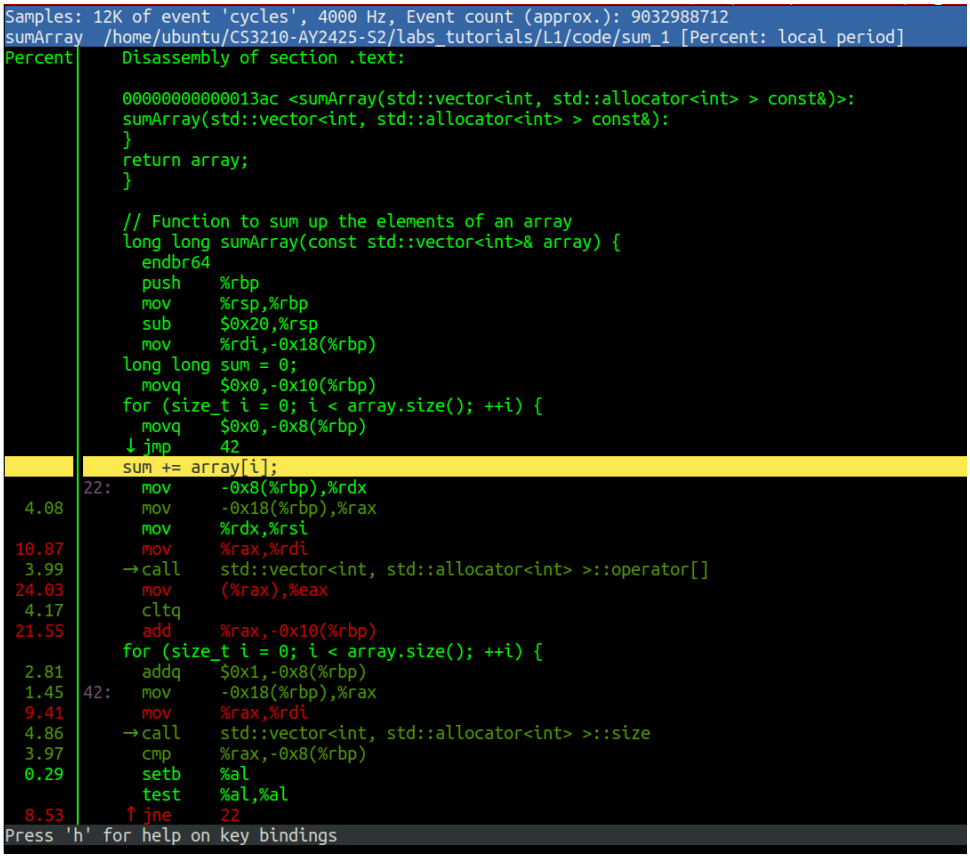

Figure 2: Example output of annotation on the `sum_1` program's `sumArray` function

This might seem even more daunting, but ignore the assembly instructions, and look just at the lines of C++ code. For example, for `sum += array`, look at the percent of cycles attributed to the assembly that makes up that source line, e.g., very broadly $4.08 + 10.87 + 3.99 + 24.03 + 4.17 + 21.55 = \mathbf{68.69\%}$ of the total cycles in `sumArray` are taken up by the actual summing operations in the loop. We don't need to be exact here – we just care broadly about what parts of our code contribute to our cycle count.

- Another incredibly powerful idea is: we don't just have to measure cycles – you can also use `perf record` to measure other things, like cache misses, and attribute them to lines of code. For example, to measure the number of last-level cache (LLC) load misses, use the following command:

```
$ perf record -e LLC-load-misses -- ./sum_1
```

Exercise 4

Use `perf record` and `perf report` to understand (a) which functions in the program take the most time, and (b) which functions in the program cause the most LLC cache misses. Why do you think so?

`perf` is an extremely powerful and deep tool. It's completely fine if you don't understand most of what is going on yet. Just remember that it's using performance counters in hardware to measure things about your program, and because of that, running perf doesn't slow down your program much at all. This is the main reason why it's used in so many contexts in performance measurement.

Flamegraphs

Flamegraphs are a way to visualize the performance of a program, by visualizing the call stack of a program, such that the width of each stack frame is proportional to the amount of time spent in that function. We have installed a tool to generate flamegraphs from your programs, called `flamegraph`.

- Make sure again that your program is compiled with the -g flag:

```
$ g++ -g -o sum_1 sum_stride_1.cpp
```

- Now, run the program with `flamegraph`:

```
$ flamegraph -- ./sum_1
```

- This generates a file called `flamegraph.svg` in the current folder. To view it, you need to copy the file to your local machine and open it in a web browser. Disconnect from the lab machine or open a new terminal session. Then, assuming that the svg file is inside `~/L1_code/flamegraph.svg` on `soctf-pdc-001`:

```
$ scp <username>@soctf-pdc-001.d1.comp.nus.edu.sg:~/L1_code/flamegraph.svg .
```

- Open the file in a web browser to view the flamegraph.

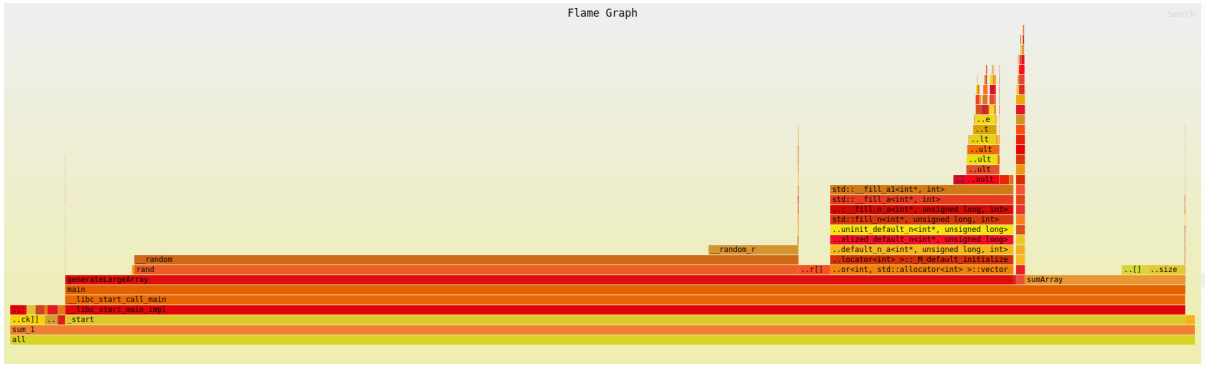

Figure 3: Example output of a flamegraph on the `sum_1` program

The flamegraph is a very powerful tool to understand the performance of your program, and it's used in many contexts in performance measurement. As mentioned, the width of each rectangle is proportional to the amount of time spent in that function, and the vertical axis shows the call stack (the order in which functions call other functions, etc). The colors are just there to differentiate between functions. You can even use `Control + F` to search for and highlight functions in the flamegraph.

Exercise 5

Use `flamegraph` to understand the performance of your program. What do you see as the main performance limitation of the program, time-wise? Is that what you expected?

Putting it all together

Exercise 6

Now that you've learnt perf stat, perf record, perf report, and flamegraphs, study the two sum programs we provided. Use the tools you have learnt to understand the performance of the two programs in depth. Discuss with your TA (if time permits) what you think the bottlenecks are in each program, and how you might optimize them.

Other (Potentially Useful) Perf Events

- `L1-dcache-load-misses`: L1 data cache load misses
- `L1-dcache-loads`: L1 data cache loads
- `L1-dcache-stores`: L1 data cache stores
- Advanced: run `perf list metricgroups` to find common groups of events (CPU specific) that are monitored together. Use `perf stat -M <groupname> -- <command>` to run the command and monitor the events in the metric group. It might take some web searches to figure out what all the metrics mean.

Useful Further Tools and Resources

- Perf Wiki Tutorial
- Flame Graphs
- magic-trace by Jane Street

Part 2: Technical Report Writing

Writing technical reports forms a significant amount of work done in this course – each of our three assignments requires you to write a report. The ability to produce good technical writing is distinct from that of writing essays or research papers, although there are some transferrable skills. In this part of the lab, we will be going through some aspects of technical writing which we expect to see in your reports throughout the course of the module.

There are several questions / requirements which are commonly needed in the reports of CS3210 assignments, and the following sections outline how these requirements can be tackled.

Code Explanation

The *technical* part in technical writing refers to how you will be explaining technical concepts to your audience. For this course, this mostly refers to your code. Always keep in mind the audience you are writing the report for, as it determines the granularity as well as the complexity of your explanation.

For this course, you can think of the readers of your reports as having the technical expertise of someone who has a good grasp on the basics of parallel computing, in addition to having programming skills; and they are interested in reading the approach that you take to solve a problem. As such, **write your report in a manner which explains the high-level overview of your code, alongside any important details which you think is needed for the grader to understand your report**. Avoid an excessive amount of implementation details in your report, which would negatively impact its clarity, direction, and conciseness.

Exercise 7

The following excerpt comes from a Producer-Consumer problem scenario (see Lab 0), and tries to explain the general overview of the solution they wrote. Are there any changes, additions, or deletions that you would make to this to improve the report?

"In our solution, we manage the synchronization of 2 producer processes and 2 consumer processes through the usage of a circular buffer and 2 named semaphores. The circular buffer is implemented through using `std::array` and 2 pointers for the start and the end of the buffer. The two named semaphores used are named "empty" and "full" – the former to indicate whether the buffer is empty and the latter to indicate if the buffer is full.
The "empty" semaphore is initialized to 0 at the start of the program as the buffer is initially empty. After the production of each item, we will signal this semaphore; and before the consumption of each item, we will wait on this semaphore. This ensures that no consumers will read from an empty buffer. Likewise, the "full" semaphore will be initialized to the size of the buffer and conversely prevent a producer from producing to a full buffer."

The example above provides a decent level of granularity for the reports that we are looking for in the module; being high-level enough to outline the idea of the solution, while providing sufficient details to the readers.

Testing and Optimizing

Another part of your technical writing will focus on testing and optimizing your code. On your first iteration of a project, rarely do you get a fully-optimized solution. The first section of this tutorial aims to equip you with enough tools for you to instrument and profile your code in order to find bottlenecks on your programs such that you can make optimizations to your code.

Bear in mind that should you improve your code, you need to convince the readers of your report that your code truly improves and the performance improvement neither happens through either sheer luck nor hyper-optimizing to certain parameters. Both of these can be shown by doing the following:

1. Perform **repeated measurements** of your experiment both before and after each optimizations. Having enough data in this fashion would eliminate the chance that the improvement is due to lucky deviations.
2. Perform your testing with **different parameters and conditions**. This shows that the optimizations generalize, and is not specific to just one specific set-up.
3. Crucially, **collect data from metrics** which support the fact that you have optimized your code. For example, if you have changed the code to improve cache hits, collect the data from e.g., L1/L2/L3 cache misses in addition to other metrics (e.g., time) that you want to show.

A simple before-after comparison with 2 data points will *not* be sufficient to show that your optimizations have worked. This unfortunately means you have to collect quite some measurement data, which is the reason that we always recommend people to start early for assignments :)

Be careful about whether your measurements are significant enough (e.g., you run your code for long enough, the sizes of things in your code are large enough) to outweigh the noise from other irrelevant sources.

Modifying parameters, Observing trends, and Making hypotheses + analysis

This section is related to the previous section. Aside from showing that the optimization that you have done is generalizable to various situations, sometimes you want to see how the (initial or final) program behaves under different parameters.

There are various kinds of parameters which you can change, some of which include:

- Input parameter(s)
- Type of machine(s)
- Configuration of machines, e.g., number of threads

There are many parameters that you can explore. In your reports, we prefer **thorough explorations of a few parameters** rather than trying out every combinations of parameters accompanied with shallow explanations. Aside from collecting the data needed, you also need to perform analysis on the data. If the report is only surface level, this would be e.g., "the time taken for the program increases proportionally to the input size". This style is insufficient for a technical report at this level; you need to come up with hypotheses on the **underlying cause of the observation.**

Not all parameters are created equal – some would lead to more interesting analyses than others. Taking the `sum_stride_64.cpp` as an example, there are two 'parameters' that can be tweaked: the size of the array and the stride length. In this case, the stride length would likely lead to a more interesting analysis than the size of the array; in this particular example, there are more things to be said, e.g. about the access patterns.

Exercise 8

Run `perf stat` to collect some performance metric measurements on the `sum` program used in Part 1 of the tutorial, changing the stride between 1 to 128 (inclusive). Think about what metrics you should collect, as well as what values you should use for the stride. Make sure to run each measurement at least 5 times. Consider automating the process as well as the data collection – the pipe (`|`) and output redirection operation (`>`) might come in handy.

Data and Visualization

Having good data visualization is paramount to having a clear report, as it allows you to paint a picture of what you are trying to convey in a concise manner. Conversely, bad data visualisation can also weaken your report, or even show flaws within your data collection processes.

As such, make sure that your data presentation works within the flow of your report to strengthen it. There are several choices for data visualization that you can do, but *most* of the time in this course, a line graph works well. Conversely, while putting all your data in a table is arguably "visualization", it's not great at telling the reader the story you intend. Similarly, don't include direct perf screenshots as your visualization.

Regardless of your choice of data visualization, we also require you to include the **raw measurement data** that you got during your experiments, so that we can verify your report. This should go in the appendix, and serves to ensure that what you have done is verifiable and repeatable when the teaching team run it. **Do not put other important information that you want us to mark in the appendix**. The appendix is only for supporting information at best.

Graph requirements

Whenever you make a graph, make sure that you minimally have the following:

1. Title
2. Axis labels
3. Axis marking / numbering
4. Clear data points

There are other elements which you should try to add when applicable:

1. Gridlines: nearly always a good idea to have, unless it clutters the graph exccessively.

2. Error bars: if you do repeated measurements, you should be able to make error bars in your graph. This can come in the form of maximum / minimum value, standard deviation, or other measures of variance. Do note somewhere what the error bars indicate.

3. Trendlines: Due to the nature of the experiments that we do, most of the time several line segments connecting the data points would be adequate at showing the data clearly. However, there are cases where you would want to do a line / curve of best-fit; in that case, take care to not make your graph deceptive. It is also a good idea to include the R^2 value if you are doing line/curve fitting.

Aside from the above, there are also some other general guidelines on making graphs that you should consider:

- Your axes need to make sense. Some (bad) examples that we have received in past assignments:
  - Axes values decreasing towards the right
  - Axes ticks are spaced equally with values not spaced equally (1, 2, 3, 5, 10)
  - Axes label does not include units and axes values' units are inconsistent
- Consider using a logarithmic scale for your axis if you find that it facilitates your data visualization. If you do use a log scale for the axis, do note that it you do not necessarily need to use log scale for both axes. Additionally, make it clear (either through the axes values or through the axes labels) that the scale is logarithmic.
- Have more than 3 data points. Trying to derive a trend with 3 or less points is at worst impossible and at best misleading. Note that these 3 data points do not include repetitions.
- Axes values should not be misleading. If your optimization causes a 10% decrease in time taken, it should not look like you made it 10x faster.
- Be careful of mixing different datsets in one graph. Improper inclusion of different dataset might clutter up the graph and muddle the purpose of the graph.

Exercise 9

Create a graph to visualise the data you have collected in the previous exercise, along with a description of the data and a short analysis (max. 2 sentences) of the data you have collected. Discuss with your TA if time permits.

Part 3: Submission Exercise (graded)

Exercise 10

For Submission (see deadline below)

Study the given `asdf.cpp` program. Write a report of not more than 2 pages, with reasonable font size and margins on A4-sized paper. Your report must have visually distinct headers for each of the following content items:

- An overview of what the code does.
- Description, visualization, and data on the execution of the given program, by exploring how and why the program's performance changes based on parameter(s) within the program.
- Try to optimize the program (there's no one right answer) such that it runs faster than the original version, for array sizes that are sufficiently large. Justify the size of array you use.

  Describe ONE performance optimization you tried, including supporting measurements and hypotheses on it worked or did not work. Do not change the random number generation process. Also, you do not need to try to parallelize the program. Please don't include a minor optimization like removing print statements, but we are also not looking for the most optimal solution. Just try to apply any optimization that works to make the program faster, and prove it to us.

Include an appendix (does not count towards the two pages of report, but don't just dump random information in there – an appendix supplements a reader's understanding but is not essential), which should contain:

- Details on exactly how to reproduce your results, e.g. which node you ran on, inputs, execution time measurement, etc. You must get the results from a CS3210 lab machine.
- Relevant performance measurement, either in table form or linking to an external document (e.g., Google Sheets).

Note that you do not necessarily need to use everything we taught you in Part 1 and 2 – focus on making a good case for what you write, not on using every possible tool on hand.

For further context - what we're asking here is very close to what we ask for assignments, so developing a good report here will help you in the very near future.

Lab sheet (2% of your final grade):
Submit the lab report in **PDF** form with file name format A0123456X.pdf (**Do not zip your file!**) via Canvas before **Wednesday, 5th February, 2pm** (note: not 11.59 pm in this course!).