Assignment one

Due Jan 28 by 10pm **Points** 10 **Available** after Jan 14 at 10am

Assignment 1 - System Performance

Due: Monday January 28, at 10 p.m.

Points: 10

Deadline to find your group partner and create your group on MarkUs: Wednesday, January 16th at 10pm. If you do not have a partner in MarkUs by this deadline, you will not be able to submit this assignment!

Intro to Scinet:

We will use Scinet for this lab. You should have already received an email with your Scinet login. In that email, you see a link to *The short intro to Scine*t document (the document is also uploaded in Quercus). You have to read that document before starting this lab. While you can use your own machine or a lab machine to debug the code, your final code has to be performance engineered and executed on Scinet. We will do all grading on Scinet. For each part, we do provide scripts to help with running your code on Scinet.

Overview

You have to work on groups of two. For this assignment, you will work on learning about the limitations of system memory performance. You will study some of the concepts discussed in lecture in a practical context, by performing performance measurements, profiling, and designing code meant to expose your system's boundaries. Additionally, you will experience first hand the impact of programming efficiently with your system's memory performance in mind.

You may work with a partner on this assignment. Please log into MarkUs as soon as possible to find your repository and invite your partner, and make sure that you can commit and push to your repo. For all assignments and labs, you will be given your repo URL on MarkUs. Make sure to use this repository (which should already be created for you), otherwise, MarkUs won't know about it and we won't be able to see your work.

The starter code is available on Scinet under */home/t/teachcsc367/CSC367Starter/assignments/assignment1/starter_code.tgz* so copy this into your repository and make sure you can do your first commit. Please make sure to read carefully over the code, including the licenses and the instructions from the comments.

Part 1 - Know your machine's memory system!

In this part, you will test the memory bandwidth and cache size of your machine.

A) Design an experiment to measure the memory bandwidth in your computer. In particular, you will measure the write bandwidth, by creating a program that accesses memory in a specific way, designed to test the limits of the memory bus.

Hint: You will want to write to memory as fast as possible; a naive approach like writing to memory byte by byte using a for-loop might not give you a reasonable estimate of memory bandwidth.

Note: This will help you reason about your system's performance limitations and scaling issues. This will be particularly useful later in the course in understanding the benefits and limitations of GPU devices.

B) Design an experiment to determine the number of levels in the CPU cache hierarchy, and to measure cache sizes and cache (write) latencies for each level, as well as the write latency of main memory. You must create a program that accesses memory in a particular manner, such that it allows you to calculate these parameters. Keep in mind how memory accesses are serviced within the memory hierarchy and design your code to expose this behaviour.

To measure the elapsed time of a piece of code, you can use the *clock_gettime()* function (with the CLOCK_MONOTONIC clock) to capture the time before and after, then subtract them using the *difftimespec()* helper function defined in *time_util.h* in the starter code, and convert the difference to time units of your choice using the helper functions in *time_util.h*.

Once your design your experiment, you must represent your results visually (with graphs) and analyze them in a short report (called *report.pdf*). You should describe your approach, analyze the findings and draw conclusions. More on this in a later section.

You must automate running the experiments, collecting the data and generating the graphs. You need to write a script (e.g. Bash or Python) that invokes your C program (*part1.c*) to performs all measurements, and produces all the graphs and other necessary data that you use to draw conclusions that your describe in your report. Invoke the script in the 'run' target of the part 1 makefile (see *part1/Makefile* in the starter code). You must also describe your data collection programs and scripts in your report.

Note: In your experiments, you can assume that the cache line size is known (usually 64 bytes, check file */proc/cpuinfo* on the machine you're using for experiments).

BONUS (10%): Design an experiment to measure the cache line size.

Hint: measuring memory write bandwidth is all about hitting the memory with heavy requests and measuring the performance of your code, while measuring cache sizes and latencies is all about hitting the memory with accesses that are increasingly unlikely to hit in the cache, until you see significant drops in performance. Particularly for cache measurements, you are encouraged to draw graphs to support your analysis.

Hint: when your program accesses a memory region for the first time after it is dynamically allocated using *malloc()*, or when it accesses a static global array for the first time, it might incur some overhead (you don't need to worry about what these are, but if you're taking CSC369 or just curious, this involves setting up the virtual to physical address translation - populating the page tables and the TLB). These are called cold

accesses. To avoid the negative impact of the initial (cold) access overheads on the accuracy of your measurements, you can "warm up" the data by touching (e.g. writing to) the respective memory before starting the measurements.

Note: Since memory and cache latency are typically very small, you want to avoid as much as possible time-consuming operations within the code being timed. For example, avoid division or modulo operations inside the timed code, if possible. You can also use bit shift operations if the denominator is a power of 2, instead of a division, etc. You might have to play a bit with such tweaks if your measurements do not seem right.

Part 2 - Performance and profiling

In this part, you will implement a piece of simple parallel code, and profile its performance. The profiling should guide you into how to optimize your code further.

You'll start with a program (part2.c in the part2/ subdirectory in the starter code) that computes the historic average grades for a set of courses. Profile your code (e.g., using the kcachegrind tool of valgrind or gprof), and then parallelize (in part2/part2-parallel.c) the piece(s) of code which take a considerable amount of execution time, and which are also parallelizable.

We provide you with a data generator (datagen.c) that is automatically invoked with default parameters by 'make' when you build the code (see comments in the makefile). This default dataset should be enough, but you can generate more data if you wish (please do not commit any data files!). Refer to the starter code for more details on the data generator. You can run the programs (part2*.c) on generated data by specifying the path to the data (the one that you gave to the generator) as an argument.

To profile your code, you have several options:

- Use gprof. For full documentation, see here **(https://sourceware.org/binutils/docs/gprof/)**.
  The most basic usage involves simply adding the "-pg" flag to the compilation and linking of your program, and then running your code. After you run your code, a "gmon.out" file will be generated. To inspect the gprof details, simply use:

```
$ gprof ./myprogram gmon.out > gprof_analysis.txt
```

You can then inspect the *gprof_analysis.txt* file for details on how much time is spent in each function.

- Use valgrind's callgrind **(http://valgrind.org/docs/manual/cl-manual.html)** tool and kcachegrind for a more visual representation. For example:

```
$ valgrind --tool=callgrind ./myprogram
```

- This will generate a trace file that starts with "callgrind". You can view this with a text editor, but this won't be very helpful since the trace can be quite cryptic. You can analyze visually the trace results using a tool like kcachegrind.
- Use a tool like oprofile. For more documentation, see here **(http://oprofile.sourceforge.net/docs/)**.

Once you have parallelized your code, you must use the perf tools to capture architectural performance counters that might help you optimize your code.

Hint: look at your cache misses in particular! You must then optimize your code accordingly and measure the improvements. Document your findings and discuss them in the report. You will implement the optimized version in part2/part2-parallel-opt.c.

If you have trouble getting stable measurements from perf, you can try building the code as follows:

EMBED_DATA=1 make (instead of a simple make invocation) This will embed the data in the compiled executable file (instead of loading it from a file explicitly by default; see the source code for details), and might help to make perf readings less "noisy". Please do not commit the generated .c files with embedded data (*part2_data.c* for example)!

Note: you ***must not*** add any gcc optimization options in your part2 makefile. This is important for the particular performance effects we want you to explore in this assignment.

BONUS (5%): study the effect of the gcc optimization level (-O0, -O1, -O2, -O3) on the performance issues that you encounter in your initial parallel implementation. Describe your findings in the report.

Testing

First read the *run-job-**.sh* files in the code folders.

1- Running Part 1: After you have logged in to Scinet and copied the code into your *$SCRATCH* folder, you can start working on the assignment. A script called *run-job-part1.sh* is provided that builds and runs your code on a Scinet compute node**.**

2- Running Part 2: Three scripts are provided called *run-job-serial.sh*, *run-job-parallel.sh* and *run-job-parallel-opt.sh* that build, generate data, and run in order the *part2.c*, *part2-parallel.c*, and *part2-parallel-opt.c* codes on a compute node. Also, script *run-gprof.sh* is provided to profile code on a compute node. If you want to use Valgrind write your own script similar to *run-gprof.sh* and remember to load the Valgrind module. You can change any of these scripts to test for different data based on the comments inside the scripts.

Report

You must write a report documenting your implementation, displaying your results in a meaningful way, and analyzing your findings, for each part of the assignment. In your report, you should present your ideas, the experimental setup and results. You should discuss what you noticed, draw conclusions and explain any optimization decisions, if any. The report should be written in a scientific manner (clear structure, clear description of your approach, results, findings, etc., and should use technical writing instead of colloquial terminology or phrases). Keep in mind that presenting your experimental findings and observations to a technical audience is an important skill to develop as a computer scientist.

Submission

You must keep the same structure as the starter code: part1 code under the "part1/" directory, part2 code under the "part2/" directory. You must submit all the files required to build and run all your programs (including any header files and the makefiles, etc.). Make sure your code compiles and runs correctly on the teaching lab machines.

- For part1, you may keep all your C code in *part1.c*, but if you add any other source files, make sure that you update the makefile accordingly. You must also submit the scripts for collecting data and generating graphs, and create a target called "run" in the part1 makefile that invokes the script(s) to collect all the necessary data and generate all the graphs that you use in your report.
- For part2, you shouldn't need to add any new files to the starter code, but if you do, make sure that you update the makefile accordingly.

Be sure to make it clear how to run your code with various configurations, in your report! Do not submit any executables or object files! (do a "make clean" before making your final commit)

Aside from your code, you must submit the report (named *report.pdf*, in the top directory of the assignment) documenting your approach, presenting the results, and discussing your findings. When discussing your approach, feel free to also describe any problems encountered and workarounds, what isn't fully implemented (or doesn't work fully), any special design decisions you've taken or optimizations you made (as long as they conform to the assignment specs!), etc.

Additionally, you must submit an *INFO.txt* file, which contains as the first 2 lines the following:

- your name(s)
- your UtorID(s)

If you want us to grade an earlier revision of your assignment for whatever reason (for example, for saving some grace tokens if you had a stable submission before the deadline, tried to add new functionality after the deadline but broke your submission irreparably), then you may specify the git hash for the earlier revision you want marked.

As a general rule, by default we will always take the last revision before the deadline (or last one after the deadline, up to your remaining unused grace tokens), so you should **not** be including a line with the git commit hash, except in the exceptional circumstances where it makes sense. So in general, please avoid using this option and just make sure that the last revision (either before the deadline if you submit on time, or up to a subset of your remaining grace tokens if you submit late) is the one you want graded.

Finally, whether you work individually or in pairs with a partner, you **must** submit a *plagiarism.txt* file (in the top directory of the assignment), with the following statement:

"All members of this group reviewed all the code being submitted and have a good understanding of it. All members of this group declare that no code other than their own has been submitted. We both acknowledge that not understanding our own work will result in a zero on this assignment, and that if the code is detected to be plagiarised, severe academic penalties will be applied when the case is brought forward to the Dean of Arts and Science."

A missing *INFO.txt* file will result in a 10% deduction (on top of an inherent penalty if we do not end

upgrading the revision you expect). **Any missing code files or makefile will result in a 0 on this assignment**! Please reserve enough time before the deadline to ensure correct submission of your files. No remark requests will be addressed due to an incomplete or incorrect submission!

Again, make sure your code compiles without any errors or warnings.

Code that does not compile will receive zero marks!

Marking scheme

We will be marking based on correctness (90%), and coding style (10%). Make sure to write legible code, properly indented, and to include comments where appropriate (excessive comments are just as bad as not providing enough comments). Code structure and clarity will be marked strictly!

Once again: code that does not compile will receive 0 marks! More details on the marking scheme:

- Part 1: 40%
- Part 2: 30%
- Report: 20% (12% for Part 1 + 8% for Part 2)
- (BONUS) Cache line size measurement: 10% (5% for the implementation, 5% for explaining the rationale clearly in the report, justifying your approach)
- (BONUS) Studying the effect of compiler optimizations for Part2: 5%
- Code style and organization: 10% - code design/organization (modularity (if applicable), code readability, reasonable variable names, avoid code duplication, appropriate comments where necessary, proper indentation and spacing, etc.)
- **Negative deductions (please be careful about these!):**
  - **Code does not compile: -100%** for *any* mistake, for example missing source file necessary for building your code (including makefile, header files, etc.), typos, any compilation error, etc
  - **No plagiarism.txt file: -100%** (we will assume that your code is plagiarised and wish to withdraw your submission if this file is missing)
  - **Missing or incorrect INFO.txt: -10%**
  - **Warnings: -10%**
  - **Extra output: -20%** (for any output other than what is required in the handout)
  - **Code placed in other subdirectories than indicated: -20%**