ECE 459: Programming for Performance
Assignment 2

Patrick Lam

January 26, 2015 (Due: February 12, 2015)

Important Notes:

- **Make sure you run your program on** `ece459-1.uwaterloo.ca`**.**
- **Use the command "**`OMP_NUM_THREADS=4;export OMP_NUM_THREADS`**" to set 4 threads.**
- **Run "**`make report`**" and push your fork of the** `a2` **directory.**
- **Make sure you don't change the behaviour of the program.**

Fork the provided git repository at `git@ecegit.uwaterloo.ca:ece459/1161/a2`:

```
ssh git@ecegit.uwaterloo.ca fork ece459/1161/a2 ece459/1161/USERNAME/a2
```

and then clone the provided files.

Grading will be done by running `make`, running your programs, looking at the source code and reading the report.

Automatic Parallelization (40 marks)

Ray tracing is, in principle, easy to automatically parallelize. You do a separate computation for each point. In this part, you will convince a parallelizing compiler (I recommend Oracle's Solaris Studio) to parallelize a simple raytracing computation.

For this question, you will work with `raytrace_simple.c` and `raytrace_auto.c` in the `q1` directory. I've bumped up the height of the image to 60000 pixels so that the compiler will find it profitable to parallelize. Benchmark the sequential (`raytrace`) and optimized sequential (`raytrace_opt`) versions. Note that the compiler does manage to optimize the computation of the sequential raytrace quite a bit. Report the speedup due to the compiler and speculate about why that is the case. Compare all subsequent numbers to the optimized version.

Your first programming task is to modify your program so you can take advantage of automatic parallelization. Figure out why it won't parallelize as is, and make any changes necessary. Preserve behaviour and make all your changes to `raytrace_auto.c`.

I put Solaris Studio 12.3 on `ece459-1`. The provided `Makefile` calls that compiler with the relevant flags. Your compiler output should look something like the following (the line numbers don't have to match, but you **must** parallelize the critical loop):

```
Compiling Part 1 Automatic Parallelization
/opt/oracle/solarisstudio12.3/bin/cc  -fast -xautopar -xloopinfo -xreduction -xbuiltin -xO4 \
    src/raytrace_auto.c -o bin/raytrace_auto
"raytrace_auto.c", line 217: PARALLELIZED
"raytrace_auto.c", line 218: not parallelized, not profitable
"raytrace_auto.c", line 233: not parallelized, loop has multiple exits
"raytrace_auto.c", line 241: not parallelized, not a recognized for loop
"raytrace_auto.c", line 264: not parallelized, not a recognized for loop
```

Clearly and concisely explain your changes. Explain why the code would not parallelize initially, why your changes are correct but bad for maintainability, and why they preserve the behaviour of the sequential version. Run your benchmark again and calculate your speedup. Speculate about why you got your speedup.

- **Minimum expected speedup:** 1.1
- **(my) initial solution speedup:** 1.1

Totally unrelated hints. Consider this page:

`http://stackoverflow.com/questions/321143/good-programming-practices-for-macro-definitions-define-in-c`

Also, let's say that you want a macro to return a struct of type `struct foo` with two fields. You can create such a struct on-the-fly like so: `(struct foo){1,2}`.

Manual Parallelization with OpenMP (30 marks)

I replaced the old Q2 with a new program that does edge detection. Compile that program and run it like `bin/canny_edge 0496_rocks.pgm 0.5 0.8 0.2`. Either try to parallelize this new program, or the old Q2 benchmark, with Oracle's Solaris Studio (not going to get far) and write a paragraph about why that doesn't work.

Your task is to provide OpenMP pragmas and structure changes (if needed) to parallelize this implementation. Provide real and user timings for the original version as well as with OpenMP directives (report the mean over 3 runs). Calculate speedups, if you managed to get them. Describe how well your parallelization worked. Figure out whether there is a serial portion that is slowing you down and explain how you know. Be clear and concise.

In an afternoon, I was not able to get speedup with OpenMP for this new program, although I think that should be possible. You can therefore earn full marks by handing in something that safely parallelizes the program without necessarily making it faster. I do expect a discussion of the effects of your changes. You can also earn bonus marks by showing a speedup. I suspect that a 20% speedup should be possible. You'll get 10 more marks for a properly-executed sped-up version (no race conditions). You can change anything you want as long as you preserve output. Use `md5sum` to compare outputs. (You can also display this file much more easily than that from part 1.)

Notes: Use `er_src` to get more detail about what the Oracle Solaris Studio compiler did. You may change the Makefile's compilation flags if needed. Report speedups over the compiler-optimized sequential version. You can use any compiler, but say which one you used. OpenMP tips:

`www.viva64.com/en/a/0054/`

- **Minimum expected speedup:** n/a
- **(my) initial solution speedup:** n/a

Using OpenMP Tasks (30 marks)

We saw briefly how OpenMP tasks allow us to easily express some parallelism. In this part, we apply OpenMP tasks to the n-queens problem[1]. Benchmark the sequential version with a number that executes in approximately 15 seconds under `-O2` (14 on `ece459-1`).

Modify the code to use OpenMP tasks. Benchmark your modified program and calculate the speedup. Clearly and concisely explain why your changes improved performance. Write a couple of sentences explaining how you could further improve performance.

Hints: 1) Be sure to get the right variable scoping, or you'll get race conditions. 2) Just adding the task annotation is going to make your code way slower. 3) You will have to implement a cutoff to get speedup. See, for instance, the Google results for "openmp fibonacci tasks". 4) My solution includes 4 annotations and some cutting-and-pasting of code.

- **Minimum expected speedup:** 1.5
- **Initial solution speedup:** 1.7

Rubric

The general principle is that correct solutions earn full marks. However, it is your responsibility to demonstrate to the TA that your solution is correct. Well-designed, clean solutions are therefore more likely to be recognized as correct.

Solutions that do not compile will earn at most 39% of the available marks for that part. Segfaulting or otherwise crashing solutions earn at most 49%.

Part 1, Automatic Parallelization (40 marks): (20 marks for implementation) A correct solution must:

- preserve the behaviour (10 points); and
- enable additional parallelization (10 points).

(20 marks for report) 16 marks for including the necessary information (describing the experiments and results, reasonably speculating about the cause, and explaining why you preserve behaviour); 4 marks for clarity of exposition.

Part 2, Manual Parallelization (30 marks): (20 marks for implementation) A correct solution must properly use OpenMP pragmas and minor code changes to parallelize the code (16 points).

(10 marks for report) 7 marks for including speedup data and explaining why your parallelization helped. 3 marks for clarity.

Part 3, OpenMP Tasks (30 marks): (20 marks for implementation) A correct solution must:

- properly use OpenMP tasks to get a speedup;
- be free of obvious race conditions.

(10 marks for report) 7 marks for analyzing the performance of the provided version, describing the speedup due to your changes, explaining why your changes improved performance, and speculating reasonably about further changes. 3 marks for clarity.

[1] `http://jsomers.com/nqueen_demo/nqueens.html`