ECE 459: Programming for Performance
Assignment 2

Patrick Lam & Jeff Zarnett

October 19, 2017 (Due: February 27, 2017)

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:ece459/1181/a2`:

```
ssh git@ecegit.uwaterloo.ca fork ece459/1181/a2 ece459/1181/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.

1 Automatic Parallelization (15 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}`.

2 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`).

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/`

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

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

3 Manual Parallelization with OpenMP (55 marks)

This time rather than just apply OpenMP directives to an existing program, you will write the program according to what is written below and verify its correctness with some provided sample files.

The program does a simulation of Coulomb's Law: there are proton and electron particles (that have the standard masses and charges). The protons are kept fixed in place via mechanical forces, but the electrons will move. Electrons move according to classical physics: they are attracted to protons and repelled by other electrons. The program will perform just one step of the simulation.

The program takes parameters:

1. h – initial size of simulation step (a measure of time)
2. e – epsilon, the amount of error allowed
3. An input file of initial positions (comma separated value file).

The program produces as output the new positions of the electrons and protons (the protons should not move).

The simulation algorithm is:

1. The initial vector y_0 contains the positions of the electrons and protons.
2. Calculate the sum of forces K on each electron, and using that, the new positions vector y_1 as $y_0 + h \times K$.
3. Calculate a second vector L which is the sum of the forces on each point at y_1 (but their new positions y_2 are not needed).
4. For each point, compute z_1 as $y_0 + h \times \frac{(K+L)}{2}$ (the average of the two forces).
5. If $|z_1 - y_1| > e$ at any position (e.g., the error at any one position is larger than the tolerance), then the simulation has too much error and we need to go back to step 2, this time with h cut in half.

Once the sequential version is written, you will apply OpenMP directives, one at a time (or in a tightly integrated group) and judge their impact on the runtime of your program as a way to assess what areas benefit most from parallelization and the impact of various OpenMP directives. Note down for your report what is effective and what is ineffective, and what produces invalid results.

You should also try to achieve the maximum speedup you can while preserving behaviour.

Submit the final OpenMP-annotated version of your code. Your report will contain the impact of various OpenMP directives, trying some out individually to see which ones are the most important.

- **Minimum expected speedup:** n/a
- **Initial solution speedup:** n/a

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 (15 marks):

- 10 marks for implementation: A correct solution must:
  - preserve the behaviour (5 points); and
  - enable additional parallelization (5 points).
- 5 marks for report: include the necessary information (describing the experiments and results, reasonably speculating about the cause, and explaining why you preserve behaviour)

Part 2, 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.

Part 3, Manual Parallelization (55 marks):

- 20 marks for the single-threaded implementation.
- 20 marks for the use of OpenMP pragmas and minor code changes to parallelize the code and get speedup.
- 15 marks for report: Explain which OpenMP directives helped. Try them out individually and determine the impact of each, and identify which ones work synergistically with others. 3 marks for clarity.