ECE 459: Programming for Performance
Assignment 2

Patrick Lam & Jeff Zarnett

January 24, 2018 (Due: February 27, 2017)

Important Notes:

- **Make sure you run your program on** `ece-tesla0.uwaterloo.ca`**.**
- **Use the command "**`OMP_NUM_THREADS=14;export OMP_NUM_THREADS`**" to set 14 threads.**
- **Run "**`make report`**" and push your fork of the** `a2` **directory.**

The repository will be created for you and initialized with starter files at `git.uwaterloo.ca`; look for your assignment 2 repo 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 significantly optimize the computation of the sequential raytrace. Report the speedup due to the compiler and speculate 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. Determine what why it won't parallelize as is, and make any changes necessary. Preserve behaviour and make all your changes to `raytrace_auto.c`.

Solaris Studio 12.3 is available on `ece-tesla0`. 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
```

Justify each change you make and explain why:

- the existing code does not parallelize;
- your changes improve parallelization and preserve the behaviour of the sequential version
- your changes adversely impact maintainability

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 question, you will apply OpenMP tasks to the n-queens problem[1]. Benchmark the provided sequential version with a number that executes in approximately 15 seconds under `-O2` (14 on `ece-tesla0` takes roughly 13.4s, but 15 takes about 1m 25s. So choose 14 in this scenario.).

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. 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. 5) Be sure to check the output of the OpenMP program for a given input against the non-OpenMP program to be sure that your results are consistent.

- **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 plan is to use Heun's method from ECE 204A (if you took that course).

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 input file is in csv format, first column is whether it is a proton (indicated by `p`) or electron (indicated by `e`), second is x coordinate, third is y coordinate, fourth is z coordinate. The input values are floating point numbers (of `float` type) and they have a precision of 6 digits and are written using scientific notation such as `3.14159e-05`.

The program produces as output the new positions of the electrons and protons (the protons should not move). The output file format should be the same as the input file format.

Some physics facts you may need to know to get this done:

- Coulomb's law calculates the force of attraction as $\frac{k(q_1 q_2)}{r^2}$. Where k is Coulomb's constant, q the charge, and r the distance between the two points.
- The value for k is Coulomb's constant $8.99 \times 10^9 Nm^2C^{-2}$.
- The charge of an electron is the same as the charge of a proton which is $1.60217662 \times 10^{-19}$ Coulombs. Electrons have negative charge and protons have positive charge.
- The mass of an electron is $9.10938356 \times 10^{-31}$ kg.

The simulation algorithm is:

1. The initial vector y_0 contains the positions of the electrons and protons.
2. Calculate a vector k_0 being the sum on each electron (and zero force on the protons), and use that to approximate the new positions vector $y_1 = y_0 + h \times k_0$.

   To calculate new positions, remember $F = m \times a$; acceleration will be considered constant in the range we're talking about (h represents a very small unit of time, or it will after the value of h has been divided multiple times after we find the error has been too large).

   With the acceleration then we compute velocity v for any point as $h \times a$; with the velocity in hand, the position is updated as simply $h \times v$ (again worth noting that the times are very small here so we don't have wild swings in v).

   Or to break it down in math notation: $F = ma \rightarrow \frac{F}{m} = a \rightarrow \frac{hF}{m} = v \rightarrow \frac{h^2F}{m} = d$
3. Calculate a second vector k_1 being the sum of the forces on each point at the new position y_1, again having zero forces on the protons. You don't have to calculate the next set of positions y_2 because it is not needed for the next step.

4. Using the two force vectors, compute z_1 as $y_0 + h \times \frac{(k_0 + k_1)}{2}$ (the average of the two forces). This produces a second, more accurate position vector.
5. If $||z_1 - y_1|| > e$ at any position (i.e., the error at any one position is larger than the tolerance), then the simulation is too coarse, and we need to go back to step 2, this time with h divided by 2.
6. Otherwise, you are ready to produce the output file. The output file should be in the same format as the input file, and the positions z_1 are the final results that should be put in the output. The output file name should be the same as the input file name, with the extension changed to `.out`. So if the input file to the program is `testcase1.in` then the output file name should be `testcase1.out`.

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.

To produce the data we want for the report, the easiest way is just to take notes about what about OpenMP directives you have used. Each time you add some OpenMP directive(s), note down what it was and what effect it had, if any. By writing down what was successful and what was not, as well as what made a big difference, you will have at hand all the data you need.

You should also try to achieve the maximum speedup you can while preserving behaviour. The usage of the compiled output is: `./protons h e inputfile`. So a sample call might be: `./protons 1 1.000e-05 example.in` ; this behaviour needs to be preserved for your solution to be tested.

Submit the final OpenMP-annotated version of your code. Your report will contain the impact of various OpenMP directives, walking the reader through the process of applying them and testing out their effectiveness.

- **Minimum expected speedup:** 8x
- **Initial solution speedup:** 12x

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):

- 35 marks for the single-threaded implementation.
- 10 marks for the use of OpenMP pragmas and minor code changes to parallelize the code and get speedup.
- 10 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.