# ECE 459: Programming for Performance
# Assignment 2

Patrick Lam & Jeff Zarnett
With acknowledgement and thanks to Douglas Harder and Stephen Li

January 21, 2019 (Due: February 15, 2019)

Important Notes:

- **Make sure you run your program on `ecetesla0.uwaterloo.ca`.**
- **Use the command "`OMP_NUM_THREADS=14;export OMP_NUM_THREADS`" to set 14 threads.**
- **Run "`make report`" and push your code to gitlab.**

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 `ecetesla0`. 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. Speedup is calculated using the unix `time` utility and comparing the `real` values.

- **Minimum expected speedup:** 10x over `bin/raytrace`, 5x over `bin/raytrace_opt`
- **(my) initial solution speedup (real time):** 12.6x over `bin/raytrace`, 7.6x over `bin/raytrace_opt`

**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 `ecetesla0` 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:** 4x with n=13, 1.75x with n=14
- **Initial solution speedup:** 5x with n=13, 2x with n=14

[1] `https://en.wikipedia.org/wiki/Eight\_queens\_puzzle`

# 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 inputs.

Many web applications use JSON Web Token (JWT) for user authentication. A JWT is composed of three components: a header, a payload, and a signature. All three components are encoded using the Base64URL algorithm and concatenated together with dots. The header and payload components are JSON objects specific for each application but can mostly be ignored for the purpose of this assignment. The signature component that authenticates the information in the header and payload is generated using HMAC-SHA256 algorithm as shown below.

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

The “secret” is what prevents malicious actors from generating fake JWT. Your task is to write a program that brute forces a JWT’s secret. Your program will be given three inputs on the command line: (1) the JWT, (2) maximum length of the secret, and (3) an alphabet representing the set of possible characters in the JWT’s secret. For example, the alphabet “abcdefghijklmnopqrstuvwxyz0123456789” represents the set of lowercase letters and numeric digits. Once you found the secret, your program should print it out in `stdout`.

Note that in practice, a JWT secret should be 32 bytes long, making it impractical to brute force. For the purpose of this assignment, we will be using small secrets that can be brute forced within a reasonable timeframe. Please don’t try to use this code on UW hardware to crack anything other than some test data for this exercise. The test data is sufficiently small that you can crack it in a few minutes; if it’s taking much longer than that, something is wrong.

Once the sequential version is written, you will apply OpenMP directive. It’s recommended to add them one at a time, or a related group at a time, to see what works and what doesn’t.

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: `./jwtcracker jwt maxsecretlen alphabet`. 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:** 2
- **Initial solution speedup:** 2.4

## 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 and why. 3 marks for clarity.