ICS632 Assignment #2

For this assignment turn in an archive of a single directory with your code in source files as specified in the exercise(s). Provide a report that answers specific questions in the exercises. For this report turn in a PDF file named `README.pdf` with both text and graphs (no points awarded for prettiness, only for readability and content).

Run your code on a **dedicated node of the Cray** (i.e., 20 cores on 1 node). Running the programs takes a bit of time. It's not a good idea to start running code right before the due date as you may not get your results in time (e.g., due to queue waiting time).

Of course, it is recommended to write scripts/makefiles/whatever to automate running the code and gathering performance numbers.
Warning: if you're testing/developing on a Mac OS X box, OpenMP may not work correctly unless you have gcc 4.8 or above (your program may abort with something like "Abort trap: 6"). You simply have to upgrade your compiler (often the default compiler that has a known bug with OpenMP). This issue may have been corrected, but I haven't checked.

Exercise #1 [30 pts]

In this exercise we parallelize the **i-k-j** matrix multiplication program from the lecture notes. Compile all code with the highest level of compiler optimization.

Questions:

1. [**15 pts**] Produce 3 versions of the program, where in each version you parallelize one of the three loops using an OpenMP pragma (taking care of declaring the correct variables private or shared). Name your source file `exercise1.c`.
    - Arrays should be of **integer** elements.
    - Each matrix multiplication should be a different run of an executable. C macros and the -D option of gcc are your friends if you want to avoid having 3 distinct copies of your code. If you have multiple copies, name them `exercise1_1.c`, `exercise1_2.c`, and `exercise1_3.c`.

- Fill in the `A` and `B` array with values $i + j$ for element (i, j), and initialize the `C` array to zero. Have you code print out the sum of the diagonal elements of array `C` so that you can double check that your parallel version print the same value as the sequential version for correctness.
- BEWARE: Parallelizing some of these loops with a simple OpenMP parallel for pragma may cause race conditions. You must prevent race conditions using **atomic** executions whenever necessary. State in your report which loop parallelizations produce race conditions and explain why.

2. [**10 pts**] Determine and use a value of N so that the sequential version (compiled with the highest level of optimization) runs is around 5 seconds. Plot the average (over 10 trials) elapsed time of all 3 versions versus the number of threads, all on the same figure (from 1 to 20 threads). Discuss what you observe and venture explanations (or guesses) as to what may be happening.
   - It is possible that some versions of the code produce very high wall-clock times. Explain, and feel free to have a time-out in your code or to interrupt the execution when it takes too long.
3. [**5 pts**] Which version is the fastest (and with how many threads?). Compare this fastest version to the purely sequential version. How much of a speedup do you observe? What is the parallel efficiency? If the parallel efficiency is far (e.g., more than just a few percent) from 100% why do you think that is?

Exercise #2 [50 pts]

Consider the program provided on the course page in the Experiential Learning section of this module, (`exercise2_startingpoint.c`), which applies a heat-transfer-like transformation to a square 2-D array over several iterations. The program takes two command-line arguments which are the number of iterations and a number of threads (which is unused in this program for now). Compile all code with the highest level of compiler optimization.

Questions:

1. [**5 pts**] Explain why using a simple parallel for pragma to parallelize the `i` or the `j` loop will not lead to correct executions.
2. [**15 pts**] Re-organize the computation (not the data) so that you can have a parallelizable loop. This can be done by changing the order in which elements of the matrix are evaluated (by adding an extra loop). Implement

this change in a source file named `exercise2.c` and briefly explain the general idea.

- *Hint*: think of the computation as a number of steps so that at each step a number of array values can be computed safely in parallel. It may help to draw a small example for a small 2-D array and proceed through the computational steps.
- *Hint*: it may be helpful to think of the computation as proceeding in two distinct phases (each phase computes half of the array values).

3. [**10 pts**] Determine and use a number of iterations that makes the original program run in about 10 seconds. Plot the performance of your OpenMP program vs. the number of threads (from 1 to 20 threads). On that same plot also plot the performance of the original sequential program (as a straight line). What do you observe? Any explanation? Should we be happy with this performance/efficiency?
   - You may want to use `perf` to understand the performance of the code.
4. [**15 pts**] Use the tiling optimization idea (i.e., the checkboard idea from Assignment #1 with square tiles) to try to accelerate your program in a view to executing it on multiple threads. The idea is that **each tile is processed sequentially by a thread, and that threads will be able to process tiles in parallel**. Write the corresponding code in a program called `exercise2_fast.c`. Use a reasonable tile size (you can determine a good tile size experimentally, but something in the 300's or 400's should be fine).
5. [**5 pts**] Add a curve for the performance of the tiled version to the plot you generated in question #3 above. Does tiling improve performance? What speedup and efficiency is achieved?

Beyond this assignment towards a project?

- Matrix multiplication and stencil applications are well-studied topics:
  - Do a brief review of (a subset of) the literature (and talk to me about it before)
  - Re-implement some of the algorithms
  - Compare performance to publicly available ”fast” implementations
  - Discover what works, what doesn't, and try to explain why
  - Go to distributed-memory settings (see upcoming lectures)