Assignment 3

Introduction to Parallel Programming

due **Thueday 17 October 2017, 18:00** (hard deadline)

Instructions

As in the previous assignments, you will need to register in a group on Studentportalen. The group can be different than your previous ones, but, once again, we suggest that you find another student to work together and form a group. In case of trouble, please contact Stephan (stephan.brandauer@it.uu.se).

Submission checklist:

- Submissions must clearly show your name(s).
- Submit a **single** PDF report, as well as all source code related to the exercises in Studentportalen.
- Solutions must be in C or C++ with OpenMP, as specified in the exercises.
- All source code must compile and run on the **Linux lab machines running Ubuntu**. For the list of the Linux servers that you can use, refer to https://www.it.uu.se/datordrift/maskinpark/linux.
- **Provide instructions for compilation and running, preferably by including Makefile(s).**
- No source code modifications should be required for reproducing your results.
- Your report must describe the theoretical concepts used as well as all relevant details of your solution.

In case you do not reach a working solution, describe the main challenges and proposals to address them. Please keep your answers short and concise, but clear and complete.
Also, note that some of the exercises ask you to do benchmarking on the Lab machines and this requires 1) using the machine at a time when it is lightly loaded, and 2) taking multiple measurements to see if there is any variation. Thus, it's not a good idea to leave this part of the assignment for too close to the deadline!

Exercise 0: "Hello, OpenMP World!" (0 points; no need to submit)

In the tar file (`assignment3_code.tar`) that this assignment comes with on Studentportalen, you can find a `Hello_OpenMP.c` file that shows you how one can embed OpenMP directives and runtime functions into a C program and do something (admittedly very boring) with the threads that are created.

a) Open an editor and study this program. One or two minutes will suffice.
Find out how you can compile it using `gcc`. (Hint: you simply need to supply an appropriate `-f` command line argument. Feel free to try to guess it before you look it up on the Web. Perhaps it's your lucky day!)
Run the resulting executable and notice its output. Run it again a couple of times more and notice that its output is not always the same. What does this tell you and what can you learn from it?

b) The Web contains a myriad of tutorials for OpenMP, in many languages and for many languages! Some of them in English for C and C++ are:

- OpenMP (https://computing.llnl.gov/tutorials/openMP/).
- A "Hands-on" Introduction to OpenMP (www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf), which also comes with code exercises (see www.openmp.org/resources/tutorials-articles/).
- Guide into OpenMP: Easy multithreading programming for C++ (bisqwit.iki.fi/story/howto/openmp/).
- OpenMP FAQ (http://www.openmp.org/about/openmp-faq/).

There are many more. Be aware that these resources contain much more information than you will need for this assignment, but on the other hand to do the assignment you will need to know more about OpenMP than what the Hello World! program showed you. So, we suggest you study them a bit and experiment with some OpenMP annotations and runtime functions before you start on the exercises below. Also feel free to modify the Hello, OpenMP World! program by adding some more code to it and running it on the lab's machines.

Exercise 1: Sieve of Eratosthenes (2 points)

The previous assignment asked you to use Posix threads to implement a parallel version of the Sieve of Eratosthenes algorithm for finding prime numbers and also suggested a strategy to parallelize your program. This exercise asks you to use OpenMP instead of Posix threads for the parallelization of your solution. If you choose to work in the same (one or two person) group as before, you need to use the program of your previous submission as a basis for this one. If you decide to form a *new* group with another student for this assignment, you can choose *one* of your previous submissions as basis (state which one you used in your report). If you have *not* submitted a program for this exercise of assignment 2, you can of course write an OpenMP solution from scratch.
Besides your code, you need to provide a short section in your report that explains how you modified your solution and reports the speedup curve you get as the number of cores is increased. Was the OpenMP version easier or more difficult to write? Are the speedups you get the same better or worse (and why)? Refer to the second assignment for more information about how to benchmark your program.

Exercise 2: Conway's Game of Life (3.5 points in total)

Conway's Game of Life[1] takes place in a two dimensional array of cells. Each cell can be empty (dead) or full (alive) representing the existence of a living organism in it, that can switch from one state to the other one *once* during some particular time interval. In every such time period (called a *generation* or a *step*), each cell examines its own state and that of its neighbours (right, left, up, down and the neighbouring cells in its two diagonals) and follows the following rules to update its state:

- If a cell has fewer than two live neighbours it dies of loneliness.
- If a cell has two or three live neighbours it lives on to the next generation.
- If a cell has more than three live neighbours it dies of overpopulation.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The initial array constitutes the *seed* of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed—births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a *tick*. The rules continue to be applied repeatedly to create further generations.

[1] Try to search for it in Google Chrome and see what happens!

File `Game_of_Life.c` contains a sequential implementation of the game; it takes as arguments the size of the array and the number of generations (steps) in the game.
The exercise asks you to convert this program into a parallel one using OpenMP and conduct experiments to measure the performance of your program as the number of cores increases for array sizes 64×64, 1024×1024, and 4096×4096 using 1000 and 2000 steps.
Submit your code, your speedup curves (x axis should be the number of cores/threads, y axis the speedup you get) for your measurements and a brief report with your findings and comments.
As you can easily discover, the Web contains plenty of OpenMP implementations of Conway's Game of Life. We have many of them, but *we want your own!*

Exercise 3: Merge Sort Using OpenMP Tasks (6.5 points in total)

Study the slides of the OpenMP tasks lecture; see the steps in the `nfib` code discussed in that lecture.
The starting point of this exercise is the `Tasks` directory of the tar ball accompanying this assignment. Spend some time studying it.

a) Time the sequential code (0.5 points). Start by running the sequential version on a lightly-loaded machine of the lab. Run it a few times, and get a median run time for this. You can test your code by running it with a small input size, e.g., `Merge_sort_debug 100`. For all timing experiments run your codes with the default (no explicit size on the command line) size of $100,000,000$.

b) Make it parallel (6 points; 2 points each). Copy the sequential `Merge_sort.c` code into a file called `Merge_sort_parallel.c` and make this code parallel using OpenMP *task parallelism*. Spawn two new tasks, one for each recursive `mergesort` call. Adjust the `Makefile` appropriately.
Keep in mind: the OpenMP pragma `parallel` should only be executed once. Also, do not merge the left and right halves of the array until they are sorted, i.e. you need a `taskwait`.

First parallel version: no pruning. Do not prune the task tree yet. Call this code `Merge_sortNP` (for not pruned). Run the parallel code for $1, \ldots N$ threads (N: number of logical cores that your machine has i.e., physical cores times the hypethreads each allows). See how the run times compare to the sequential version. Why do you think you are getting the results you are? Think about how many tasks are spawned. Bring this to the discussion on this topic (later).

Second parallel version: pruning. Now prune your code by only spawning sufficiently large tasks. Call this code `Merge_sortP`. Run the parallel code for $1, \ldots N$ threads. Experiment with tasks sizes and create a speedup curve for your preferred task size. See how the run times compare to the sequential and unpruned versions. Think about how many tasks are spawned now. Bring your observations to the discussion on this topic.

Third parallel version: spawning only one task. Now change your code to spawn only one task for one of the recursive calls, and let the other recursive call be done by the parent task. Call this code `Merge_sortS`. Experiment with it. What are your observations? Summarize your observations to take to the discussion on OpenMP Tasks.

You should submit your code and your report.

Good luck!