# Assignment 3

Introduction to Parallel Programming

due **Saturday 11 October 2025, 23:59** (hard deadline)

## Instructions

You should register in a group in Studium and submit there. Your group can be different than previous ones, but, once again, we suggest that you find another student to work together in a pair. You can share the coding and testing effort this way. Contact the assistant responsible for this assignment (xiaoyue.chen@it.uu.se) for questions about the assignment and for clarifications.

**Submission checklist:**

- Submissions must clearly show your name(s).
- Submit a **single** PDF report, as well as a **.zip** file with all source code for your programs.
- Solutions must be in C11 or C++11 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 is not a good idea to leave this part of the assignment for too close to the deadline!

## Some OpenMP Tutorials

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:

- The "Hands-on Introduction to OpenMP" video tutorial in 27 parts (www.openmp.org/resources/) by Tim Mattson, which also comes with exercises (www.openmp.org/resources/tutorials-articles/).
- OpenMP (https://computing.llnl.gov/tutorials/openMP/).
- Guide into OpenMP: Easy multithreading programming for C++ (https://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. Still, we suggest you study them a bit and experiment with some OpenMP annotations and runtime functions before you start on the exercises of this assignment.

## 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 (4 points in total)

Conway's Game of Life 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.
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 $64 \times 64$, $1024 \times 1024$, and $4096 \times 4096$ 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: Matrix Multiplication in OpenMP (3 points in total)

Consider the matrix-matrix product code given in slide 16 of Lecture 7 (`07-OpenMP.pdf`). Your task is to implement it and test it. Use the `OMP_NUM_THREADS` environment variable to control the number of threads and plot the performance with varying numbers of threads. Consider three cases in which (i) only the outermost loop is parallelized; (ii) the outer two loops are parallelized; and (iii) all three loops are parallelized. What is the observed result from these three cases? Submit your code and a brief report with your experiments and comments.

## Exercise 4: Gaussian Elimination in OpenMP (3 points in total)

When we solve a large linear system, we often use *Gaussian elimination* followed by *backward substitution*. Gaussian elimination converts an $n \times n$ linear system into an upper triangular linear system by using the "row operations." as follows: (i) add a multiple of one row to another row; (ii) swap two rows; and (iii) multiply one row by a non-zero constant.
An upper triangular system has zeroes below the "diagonal" extending from the upper left-hand corner to the lower right-hand corner.
For example, the linear system

$$\begin{aligned} 2x_0 - 3x_1 &= 3 \\ 4x_0 - 5x_1 + x_2 &= 7 \\ 2x_0 - x_1 - 3x_2 &= 5 \end{aligned}$$

can be reduced to the upper triangular form

$$\begin{aligned} 2x_0 - 3x_1 &= 3 \\ x_1 + x_2 &= 1 \\ -5x_2 &= 0 \end{aligned}$$

and this system can be easily solved by first finding $x_2$ using the last equation, then finding $x_1$ using the second equation, and finally finding $x_0$ using the first equation.
We can devise a couple of serial algorithms for back substitution. The "row-oriented" version is

```
for (row = n-1; row >= 0; row--) {
    x[row] = b[row];
    for (col = row+1; col < n; col++)
        x[row] -= A[row][col] * x[col];
    x[row] /= A[row][row];
}
```

Here the "right-hand side" of the system is stored in array `b`, the two-dimensional array of coefficients is stored in array `A`, and the solutions are stored in array `x`. An alternative is the following "column-oriented" algorithm:

```
for (row = 0; row < n; row++)
    x[row] = b[row];
for (col = n-1; col >= 0; col--) {
    x[col] /= A[col][col];
    for (row = 0; row < col; row++)
        x[row] -= A[row][col] * x[col];
 }
```

For starters, determine whether the outer loop and/or the inner loop of the row-oriented algorithm can be parallelized. Similarly, determine whether the (second) outer and/or the inner loop loop of the column-oriented algorithm can be parallelized. Your tasks are:

- Write one OpenMP program for each of the loops that you determined could be parallelized. You may find the `single` directive useful—when a block of code is being executed in parallel and a sub-block should be executed by only one thread, the sub-block can be modified by a `#pragma omp single` directive. The threads in the executing team will block at the end of the directive until all of the threads have completed it.
- Modify your parallel loop with a `schedule(runtime)` clause and test the program with various schedules. If your upper triangular system has 42 000 variables, which schedule gives the best performance?

As in the previous exercises, submit your code and a brief report with your experiments and comments.

**Good luck!**