University of Liège

High performance Scientific computing

Project 1 - Report

Master in Data Science & Engineering

Author:
Tom Crasset

Professors :
C. Geuzaine

Academic year 2019-2020

1 Introduction

The aim of this project is to familiarize the student to high performance computing and to parallel programing. This report is going to study the influence of the density of conducting fibers in a composite material. A percolation model will be used to test the electrical conductivity using the C programming language and various libraries, in particular the OpenMP library. This library will be used to parallelize the computation and its effects on the computation time will be studied too.

2 Implementation

2.1 Randomization of conducting fibers

To fill the grid with a certain density of conducting fibers, which were 3 cell long conducting strands, one first had to randomly choose the starting cell. After this, one only had to expand the conductivity to the upper/lower or the left/right neighbours to have 3 conducting cells in a row. The starting cell was chosen by creating an array of N^2 indices, randomly shuffling that array and then placing a conducting cell on the first `nbConducting` indices of the grid, `nbConducting` being the number of conducting fibers needed to have a desired density of `d`. Note that this is the number of conducting fibers, not cells. Ideally, there would be 3 times more conducting cells if no fibers overlap. In this implementation, there is never a complete overlap as the starting cells are each unique. For each fiber, there is at most a partial overlap of 1 cell, meaning that is 2 fibers overlap, there are 5 conducting cells.

2.2 Pathfinding

To test the conductivity of the originially non-conducting matrix after introducing a certain density of conducting fibers, the pathfinding algorithm was applied on every cell of the leftmost column. This mimicks an electric input on the left side of the composite material. Once all the pathfinding algorithms have finished, the rightmost column of cells is checked to see if they are conducting and connected. If one of them is, it means that a path of conducting cells from the right side to the left side exists. This is similar to checking if a current has passed through the composite material.

This pathfinding is not a real pathfinding algorithm, like Dijkstra's algorithm. Rather, it boils down to applying a flood fill algorithm and then checking if both sides have a connected conducting cell. The flood fill algorithm works in the same fashion as a depth-first search approach as it searches as far as possible along each branch before backtracking.

At first, a recursive approach was chosen. However, the recursive call stack limit of the operating system was reached rather quickly at grid sizes of around 2000. One solution could have been to do horizontal pixel runs rather than checking the 4 neighbours at each iteration. However, this implementation is not trivial.

The chosen solution is to simply forgo the recursive aspect and use a stack based approach by maintaining a stack of points that need to be reviewed. This stack was built using linked lists and could handle up to grid sizes of 20000. However, the time complexity is still bounded by $\mathcal{O}(N^2)$, N being the square grid size and thus the grids can not be too big either to finish in a reasonable time.

2.3 Parallelism

While there are many possibilities to parallelize the code (pathfinding, filling the grid with conducting fibers, etc...), the one that has the most effect is parallelising the Monte Carlo simulation. Using

simple OpenMP declaration such as `#pragma omp parallel` and `#pragma omp for`, the speedup was substantial as will be shown in the following sections.

3 Strong scaling and effect of loop scheduling

3.1 Strong scaling

Investigating the strong scaling of an implementation is measuring how efficient an application is when using increasing numbers of parallel processing elements. Strong scaling measures cpu-bound applications whereas weak scaling is for testing memory-bound ones.

In the following (unless mentioned), the number of threads used were powers of two, up to 8 threads. The runtime was averaged over all the values for parameters `N` in [10, 50, 100, 200, 500], `M` in [10, 100, 1000, 10000], and density `d` from 0 to 1 in steps of 0.02.

Figure (1) shows the results of the strong scaling measurements of the application alongside a perfect scaling. On the left, the actual average time is plotted and on the right plot, the speedup factor with respect to a single threaded application is depicted. The right graph shows clearly the linear scaling of the application, albeit a little sub-optimal. Using the formula (1) for strong scaling, we get a strong scaling efficiency of 81.04%.

$$\frac{t_1}{nt_n} * 100 \tag{1}$$

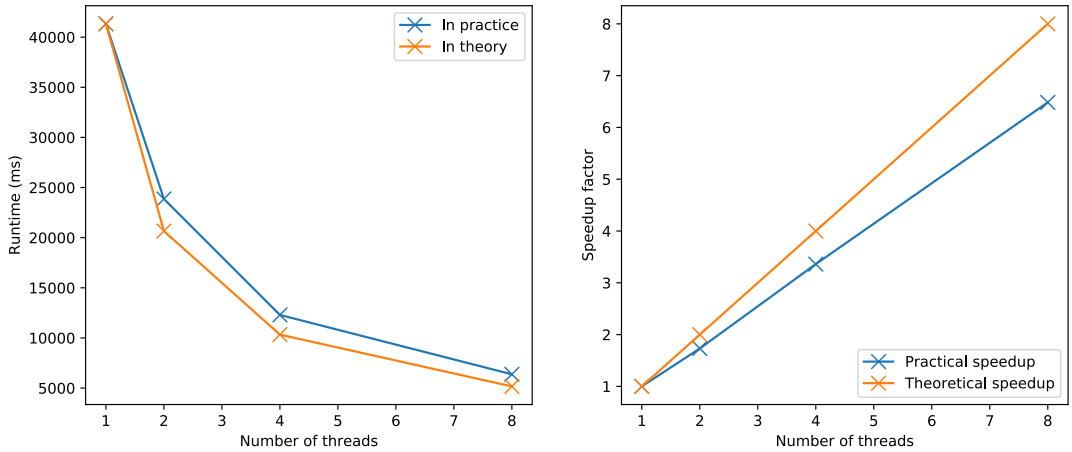

Figure 1: Left) Average runtime with respect to the number of threads, both in theory and applied on our problem. Right) Average speedup with respect to the number of threads, both in theory and applied on our problem.

A more detailed graph on the left of Figure (2) shows the runtime for different gridsizes `N` with different values of the parameter `M`. One can see that the runtime is linear for all gridsizes with respect to the number of Monte Carlo simulations, except for the smallest graph, where the difference between 1 and 10 iterations is negligible. Note that the x- and y-axis are logarithmic, however both variables are still linear with respect to each other. They do not represent a power law, or more exactly, the exponent in the power law of type $y = kx^a$ is 1.

Moreover, on the right of Figure (2), one can see the relationship between average runtime and the gridsize `N`. One can clearly see that the time complexity of the application is $\mathcal{O}(N^2)$. No attention

should be given to the exact values of the runtime as it is an average over all parameters as stated at the beginning of this section.

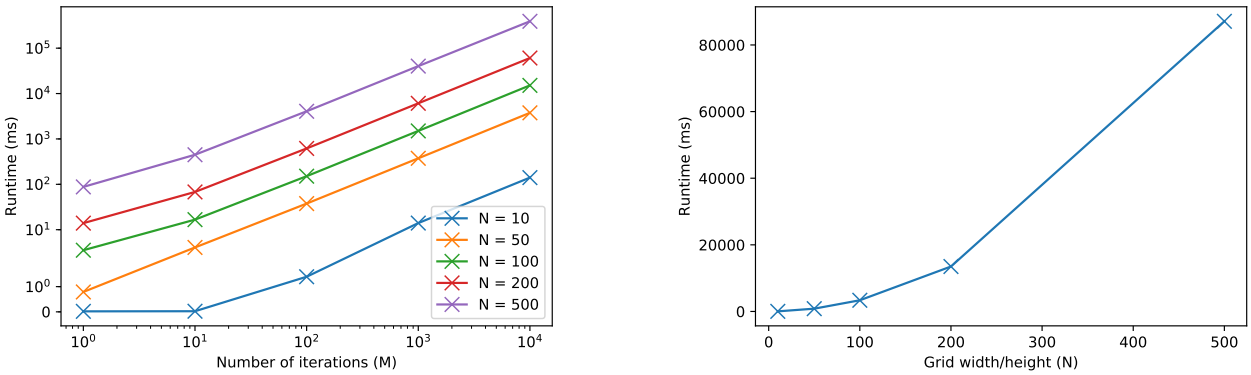

Figure 2: Left) Average runtime with respect to the grid size `N` for different numbers of simulations `M`. Right) Average runtime with respect to the grid size `N`.

3.2 Loop scheduling

Loop scheduling is the task of measuring the effect of different thread scheduling options on the parallel implementation. The different scheduling options used were `auto, dynamic, static, guided`. Again, the average was done over various parameters, mostly the density `d`. `N` was set to 100, `M` to 1000 and the density `d` was varied from 0 to 1 in steps of 0.1. Figure (3) shows a quick overview of the different scheduling methods on the runtime of the threaded application. On average (over 10 runs), one can see that the schedule type has little influence on the runtime of the application.

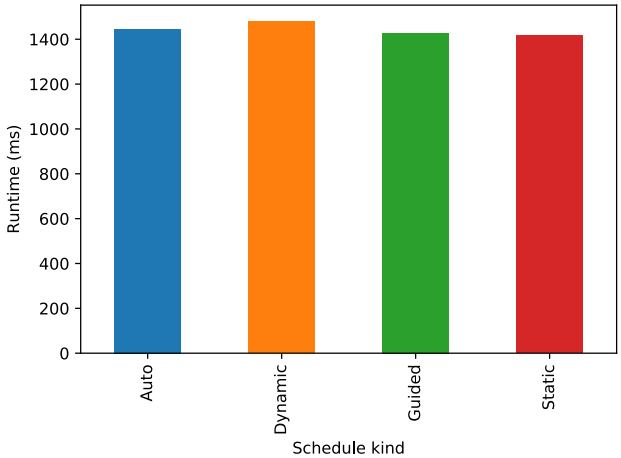

Figure 3: Average runtime for different scheduling types

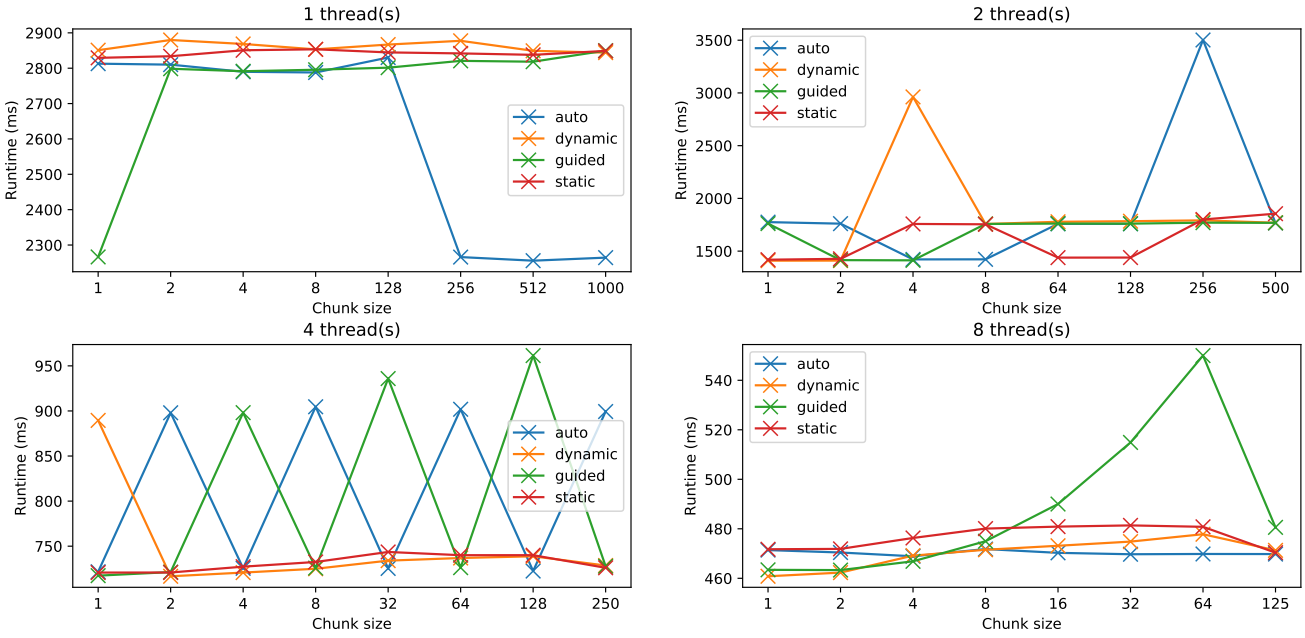

Figure 4: Average runtime for different threads, scheduling types and chunk sizes

A more detailed assessment of the scheduling was done by varying the chunk size on Figure (4). Some thread sizes were dependent on the number of threads while others were not. The different ones were $[1000/nbThreads,\ 512/nbThreads,\ 256/nbThreads,\ 128/nbThreads,\ 8, 4, 2, 1]$. As one can see, for 1 and 2 threads, the runtime with respect to the schedule type and the chunksizes are similar, ignoring the 2-3 slip-ups. For 4 threads, a kind of saw-tooth pattern appears for the guided and auto schedule type, which cannot be easily explained. However, with 8 threads, one can see that all the schedule type behave similary except for guided, which has a constant growth up to the second to last chunk size. That can be explained by the fact that specifying a big (minimum) chunk size goes against the purpose of guided, which tries to reduce the number of chunks per thread at the end of the computation. For chunksizes of 125 and 8 threads, each thread gets its share of iterations scheduled once, thus not continuing the upwards trend seen before.

4 Effect of the parameters on the probability of conduction

In addition to the strong scaling and scheduling measurements, more graphs were created to get a better insight on the influence of the parameters on the application, especially the density `d`.

On Figure (5), one sees that as the grid size increases, the probability of conduction increases more abruptly. For a grid-size of 10, the slope is smaller than for $N = 500$. A closeup on the region of interest is shown more clearly on the left of Figure (6) and one can see that the fiber density threshold for the conduction is around 21%. On the right of that Figure, one can also see that the smaller the density, the smaller the runtime, with a big increase right around the conductivity threshold. The size of the grid has no influence apart from it taking longer because the grid is bigger.

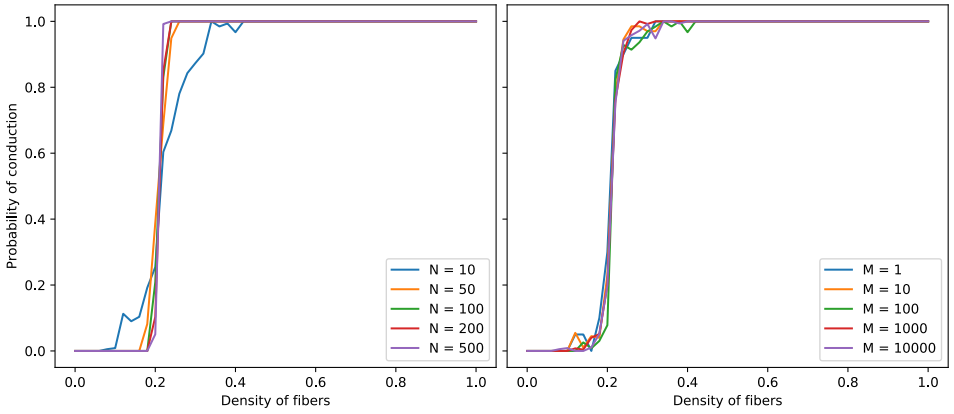

Figure 5: Average probability of conduction with respect to the density of fibers, for different values of gridsize `N` (on the left) and number of simulations `M` (on the right)

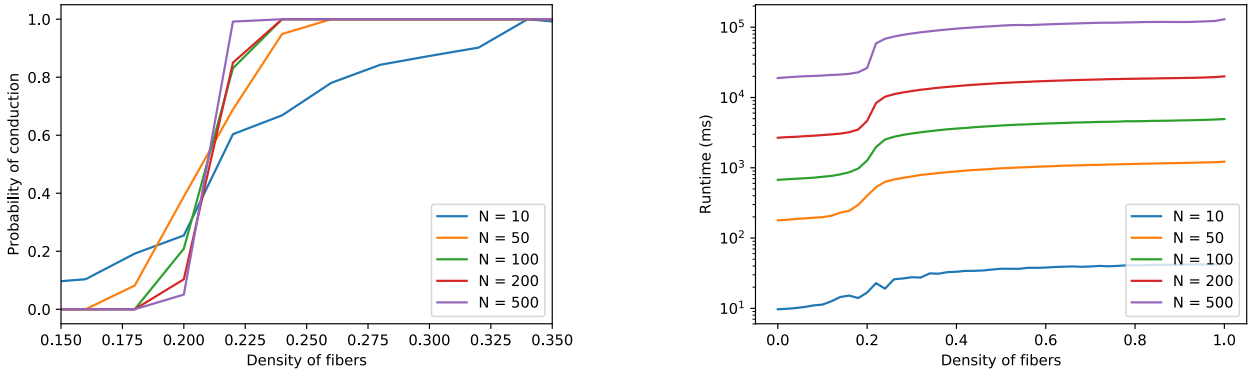

Figure 6: Left) Closeup around the region of interest of the average runtime with respect to the grid size `N`. Right) Average runtime with respect to the density of fibers in the grid.

5 Conclusion

During this project, one found that the speedup of an application can be achieved using parallel programming using OpenMP on multi-core/multi-threaded machines. The application presented a scaling efficiency of 81.04 % and the schedule type and/or the chunk sizes had little effect on the runtime. Moreover, one found that the probability of conduction changes abruptly after a certain threshold value. The abruptness of the change also increases with increasing grid sizes.