Assignment 3 - Introduction to Parallel Programming

Bernat Xandri Zaragoza

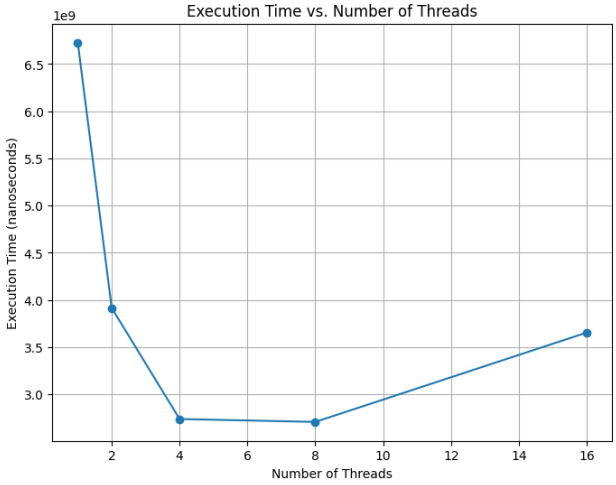
1. **Exercise 1**: Sieve of Eratosthenes (2 points)

a) 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.

For this implementation, I took the sequential implementation that I did in assignment 2 as a starting point, because the parallel implementation using threads that I handed in had clear problems in the logic of the algorithm, and also, I thought it was easier and clearer to adapt the sequential version.
For this new implementation using OpenMP, I omitted some of the parallelization instructions from the previous assignment that proved to be confusing (probably due to a miss-understanding of these instructions from my part) such as "*First sequentially compute primes up to √Max*", and rather do a more simple approach using Parallel regions and barriers as a synchronization tool.
This way, I kept the main structure of the code intact, just making the algorithm part of a Parallel region using the *"#pragma omp parallel num_threads(num_threads)"* so that the desired number of threads are each executing a copy of the code within the structured block. I kept the chunk of code from the first parallel implementation used to split the range of numbers we're working with among threads, but it could have also been done by using loop worksharing Constructs such as a for, in such a way that the splitting of regions among threads was taken care of by OpenMP. But rather than doing this, which I think would be a better way of implementing this code, I choose the other way in order to maintain as much code as possible untouched.

In order to get a representative speedup curve, we need to set a really high max number in order to compensate for the overhead of creating and managing the threads. The speedup curve that we obtain when running this program with different number of threads, and a max of 100.000.000 is as follows:

Number of Threads	Time (nanoseconds)
1	6727248700
2	3915773500
4	2739001900
8	2708207500
16	3654879200



Graph with python

As we can observe in this graph, the parallelization of the algorithm greatly improves the execution times. We observe that from 4 threads to 8 the difference is quite small, and when we jump to 16 it increases the time. This is probably due to the overhead of using so many threads, and in case of having a bigger max number, the curve would look different.

About the implementation using OpenMP, I would consider it to be a simpler implementation. The fact that some of the tools are already prewritten in a package makes it easier to comprehend and write, but also the code is easier to read.

2. **Exercise 2**: Conway's Game of Life (4 points in total)

a) 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.

For this implementation in C, we can see that there's multiple loops that can be parallelized. Some of these, such as the ones in functions like "*allocate_array(int N)*" or "*init_random(int ** array1, int ** array2, int N)*" would not benefit from parallelization since, as seen in previous exercises, the overhead of managing multiple threads can be significant, so parallelization is only beneficial for heavy computations. In the case of simple operations (and I think these functions can still be considered light operations computationally speaking), the overhead would make the execution slower.

Therefore, I decided to only parallelize the main loop of the Game of Life. To do this, we need to study which loops have iteration-dependency, and which can indeed be parallelized. We can see that the first loop "*for (t = 0 ; t < T ; t++) {*" acts as the tick between generations. This first loop, has time-dependency, since the next state depends on the previous ones. Therefore, it cannot be parallelized in an easy way.

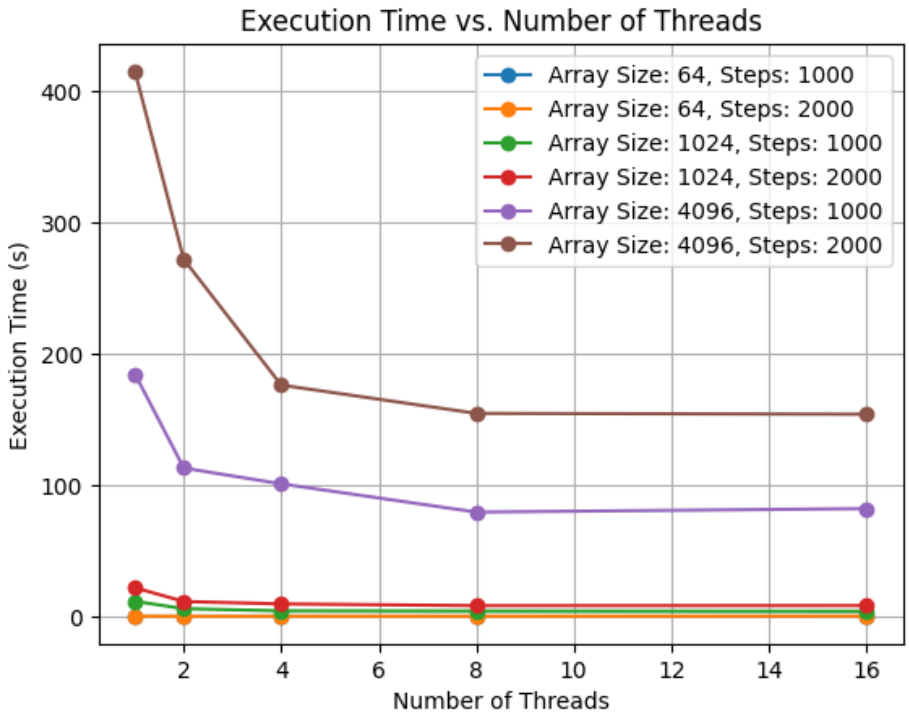
The following two loops are iterating over the array to apply the rules of the game. In this case, and because in the rules of the game there's dependencies between contiguous cells, we can only parallelize the second loop, but not the third one.

After doing this parallelization, we obtain the following speedup curves:

Array Size	Steps	Number of Threads	Execution Time (s)
64	1000	1	0.048022
64	1000	2	0.022647
64	1000	4	0.023675
64	1000	8	0.017181
64	1000	16	0.073329
64	2000	1	0.082397
64	2000	2	0.045321
64	2000	4	0.045180
64	2000	8	0.034758
64	2000	16	0.137614
1024	1000	1	11.466726
1024	1000	2	5.825789
1024	1000	4	4.200682
1024	1000	8	4.023235
1024	1000	16	3.855603
1024	2000	1	21.797438
1024	2000	2	11.289313
1024	2000	4	9.487902
1024	2000	8	8.151251
1024	2000	16	8.188374

4096	1000	1	184.351241
4096	1000	2	112.887442
4096	1000	4	100.846385
4096	1000	8	79.352491
4096	1000	16	82.014812
4096	2000	1	414.841980
4096	2000	2	271.714326
4096	2000	4	176.020564
4096	2000	8	154.511950
4096	2000	16	153.851473

When studying this results in a graph we obtain:



Graph with python

We can observe a clear effect due to the parallelitzation.

3. **Exercise 3**: Matrix Multiplication in OpenMP (3 points in total)

a) Does 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:

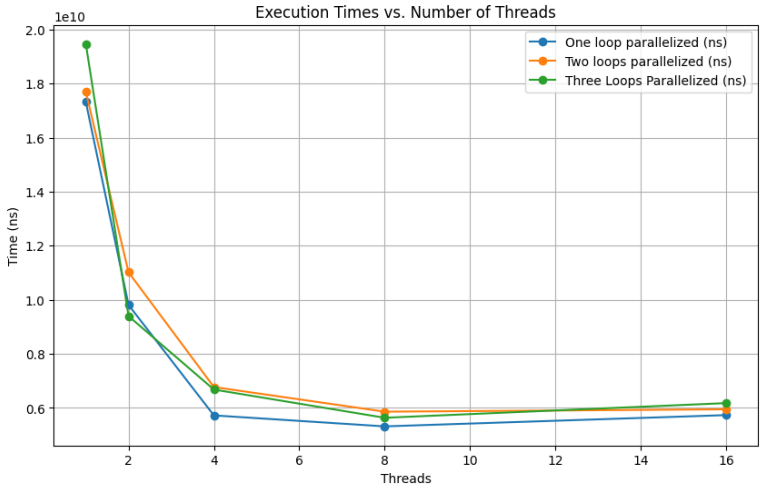
1. Only the outermost loop is parallelized;
2. The outer two loops are parallelized;
3. All three loops are parallelized.

What is the observed result from these three cases? Submit your code and a brief report with your experiments and comments?

For this implementation, we only needed to add all the code wrapping the algorithm of matrix-matrix product given in the lecture. For this, we added a few functions to generate randomly filled matrixes of N dim, and also the basic code to input the number of threads and the dimension of the matrix.
Finally, we implemented the 3 cases that are required in this exercise (the first one was the provided case) using the OpenMP collapse clause. This way, we can indicate how many loops to be parallelized, counting from the outside.
When studying the speedup curves for these different implementations, we observe the following;

Threads	Dimension	One loop parallelized (ns)	Two loops parallelized (ns)	Three Loops Parallelized (ns)
1	1000	17321600500	17697566500	19468341200
2	1000	9811608900	11019646100	9396991400
4	1000	5722683400	6772138300	6679327200
8	1000	5316824200	5860898400	5635350700
16	1000	5733746100	5949090800	6178150000

When studying this results in a graph we obtain:



Graph with python

We don't observe a very noticeable difference, probably due to using a dimension too small. In cases where more computation is required, the difference would be more noticeable.

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

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 columnoriented algorithm can be parallelized. Your tasks are:

1. 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.
2. 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?

In the **row-oriented algorithm**, we can observe that the first loop "*for (row = n-1; row >= 0; row--)*", is iteration-dependant, since in order to calculate the value of one row, we need to have calculated the value of all the rows before that one (it starts from the bottom and moves upwards until the first equation), and therefore cannot be parallelized. We will always need to have solved all previous rows before being able to solve a new one.

The second loop "*for (col = row+1; col < n; col++)*", on the other hand, can be parallelized, since its using elements of the rows already calculated to solve different elements of the row in which we are iterating. Therefore, since it's not order-dependant, we can have multiple threads working on different elements of the row at the same time.

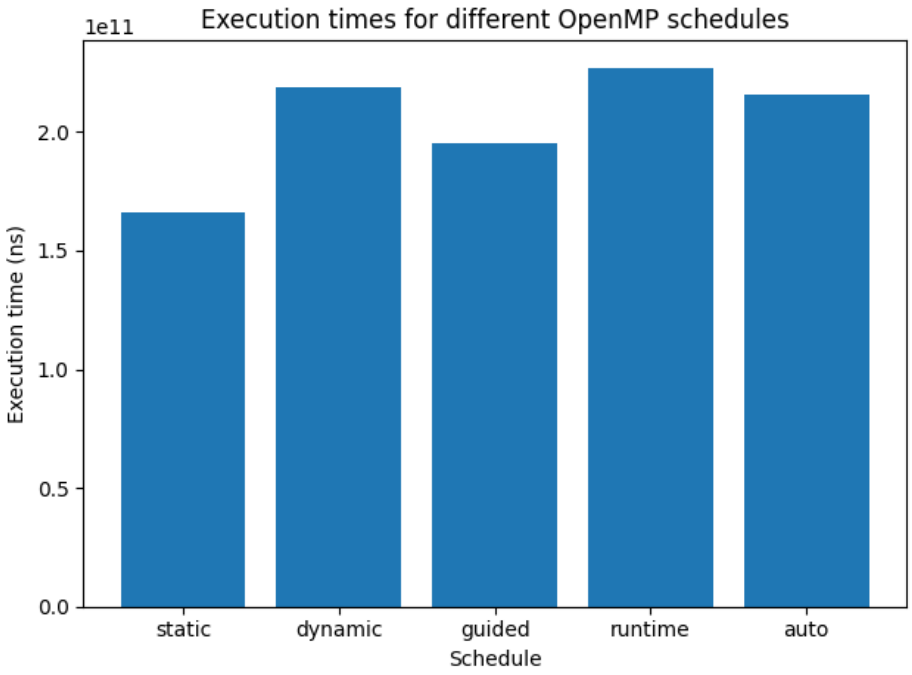
In the **column-oriented algorithm**, we can observe that the independent loop can be parallelized since it's not a nested loop.

For the other nested loops, we observe the same situation as in the first algorithm, where the outer loop is iteration-dependant, and can not be parallelized in a easy way, but the inner one, since in this case it's iterating over the different elements of a column, can be parallelized without problems.

For this scheduler comparation, we use 8threads and 42000 variables, and we obtain the following results:

Schedule clause	Execution time (ns)
static	166398346300
dynamic	218838518200
guided	195018027300
runtime	227119292100
auto	215675050500

When studying this results in a graph we obtain:



Graph with python

We can observe that the best performance is the static schedule, and the worst one is actually the runtime.