Parallel Systems HW 2

December 2024

Introduction

For compilation, the makefile found in the initial directory was used, while the source code for each exercise is located under the corresponding directory **ask-X**. For each exercise there is a separate rule and therefore with **make askX** each exercise is compiled with its executable located in its directory. The measurements were made automatically with python scripts in which, to calculate the time of each program, 5 executions were made and their average was obtained. Their results are printed on the terminal in table format and saved in .csv files, which facilitated the process of measurements and production of graphical representations.

The measurements were made on the department's Ubuntu Linux machines, which have a CPU with 4 cores and an L1 cache-line of 64 bytes::

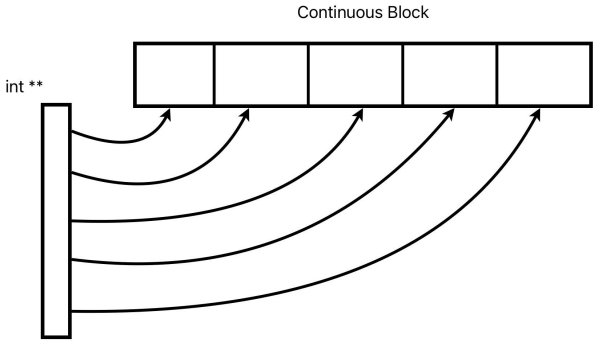
```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 94
model name      : Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz
... skip ...
cpu cores       : 4
... skip ...
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual

$ gcc -v
... skip ...
Target: x86_64-linux-gnu
... skip ...
Thread model: posix
gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.2)

$ cat /etc/os-release
NAME="Ubuntu"
VERSION="20.04.6 LTS (Focal Fossa)"
```

2D allocation

2D allocation Both of the questions in the paper handle two-dimensional arrays, it is important to note the way in which the allocation is done. Instead of the classic iterative allocation in a for loop, we call only one malloc at the beginning to allocate a large contiguous block of memory of size `n * n * sizeof(int)` and then we share the pointers for the rows on this block, schematically:



This way we only make one allocation and the array data is in contiguous memory locations, thus improving cache efficiency.

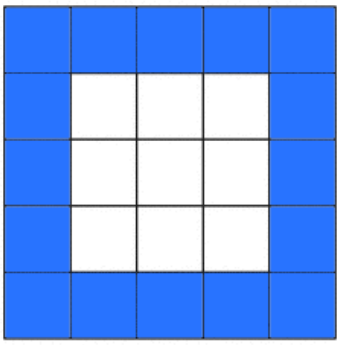
Excercise 2.1

Execution

To execute excercise 2.1, after the compilation run the command:
`./ask1/game_of_life <generations> <matrix_size> <algorithm> <threads>`
, where algorithm = 0 or 1 (0 for serial 1 for parallel).

General Implementation Comments

The basic idea of the algorithm is that to create a new generation, we sequentially examine all the cells of the array and apply the rules based on the values of the neighboring cells. The implementation of the algorithm for the Game of Life includes three nested for loops. The outer loop is responsible for sequentially processing the generations, while the two inner loops traverse all the cells of the two- dimensional array, examining the state of each one separately. For input matrix-size = n, we create a square matrix of dimension (n+2) x (n+2). The cells of the outline are initialized with 0, while the inner n x n cells randomly get 0 or 1. This approach allows us to avoid additional if statements that would be required to check if a cell is in the border, thus preventing access to neighboring cells that are outside the array boundaries, schematically for size = 3:



In the above diagram, the cells in blue are the cells of the border that we add and have a value of 0. The cells in white take on random values and constitute the basic grid of the game.

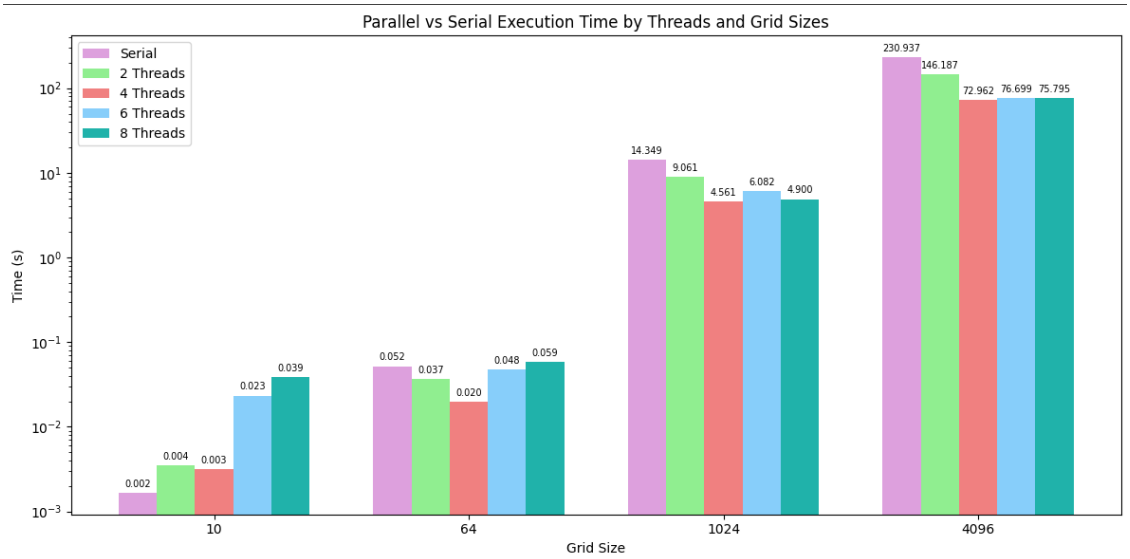
In our implementation we use two square arrays: grid and `grid_copy`. The table grid stores the current state of the grid, while `grid_copy` is used to record the new values of the cells in each generation. After the calculation for all cells is completed, the two tables are swapped, so that the grid is updated with the new values without the need to copy data (memcpy).
To calculate the alive neighbors of a cell, we sum the values of the eight neighboring cells, which can be either 0 or 1. The result is stored in the variable `alive_neighbours`.
Updating the grid is done simply by exchanging the pointers of the two arrays. This way, we avoid copying the entire array, which significantly improves the performance of the program.

Parallel Algorithm

In the parallel algorithm, we chose to parallelize only the outer for loop, which iterates over the rows of the array. Instead of using the collapse(2) directive to parallelize both for loops, we preferred to have each thread handle an entire, contiguous row.
This approach optimizes data locality, as the elements of a row are stored in consecutive memory locations. This way, we reduce cache misses and achieve better performance.
Finally, in the parallel algorithm we did not need synchronization between threads. This happens because each thread exclusively updates a specific position of the table, which is not used by other threads. Therefore, there is no risk of race conditions, as there is no simultaneous access to the same memory location by multiple threads.

Running the program on the laboratory machines we obtain the following results:

Generations	Grid Size	Algorithm	Threads	Time (s)
1,000	10	Serial	-	0.0017
1,000	10	Parallel	2	0.0035
1,000	10	Parallel	4	0.0032
1,000	10	Parallel	6	0.0234
1,000	10	Parallel	8	0.039
1,000	64	Serial	-	0.052
1,000	64	Parallel	2	0.0368
1,000	64	Parallel	4	0.0197
1,000	64	Parallel	6	0.0476
1,000	64	Parallel	8	0.059
1,000	1,024	Serial	-	14.3492
1,000	1,024	Parallel	2	9.0608
1,000	1,024	Parallel	4	4.5606
1,000	1,024	Parallel	6	6.0818
1,000	1,024	Parallel	8	4.8999
1,000	4,096	Serial	-	230.9378
1,000	4,096	Parallel	2	146.1874
1,000	4,096	Parallel	4	72.9625
1,000	4,096	Parallel	6	76.6991
1,000	4,096	Parallel	8	75.7956



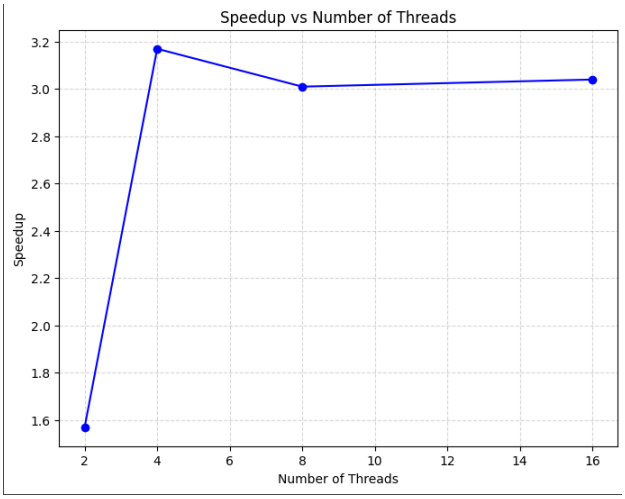
Συμπεράσματα

Observing and comparing the times of the serial program with the parallel one, we see a clear improvement in performance, and a speedup close to ideal, as we would expect from such a readily parallelizable problem. We see that for each doubling of threads, the time almost halves, while by oversaturating the number of threads, i.e. by adding more threads than the available cores (4 on the lab machines) we observe that no further improvement is noted, in fact the execution time increases slightly, which may be due to the additional overhead of creating a thread.
We observe that for very small arrays (e.g., size = 10), the serial algorithm achieves slightly better execution

times compared to the parallel one. This is due to the fact that, for small workloads, the cost required to create and manage threads, as well as to distribute the workload, exceeds the benefits of parallel execution. In such cases, serial execution is more efficient as the additional operational cost of thread management is avoided.

Speedup for array with dimension 4096:

Threads	T.Serial	T.Parallel	Speedup
2	230.9378	146.1874	1.57
4	230.9378	72.9625	3.17
6	230.9378	76.6991	3.01
8	230.9378	75.7956	3.04



Excercise 2.2

Execution

To execute exercise 2.2, after the compilation run the command:
`./ask2/back_sub <size> <par/ser> <row/col> <thread_num>`.

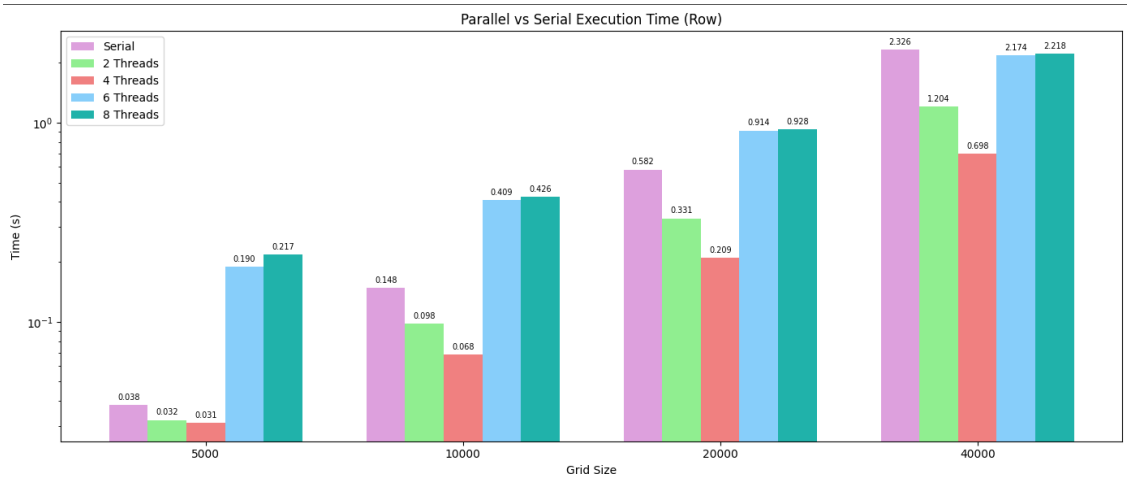
General Implementation Comments

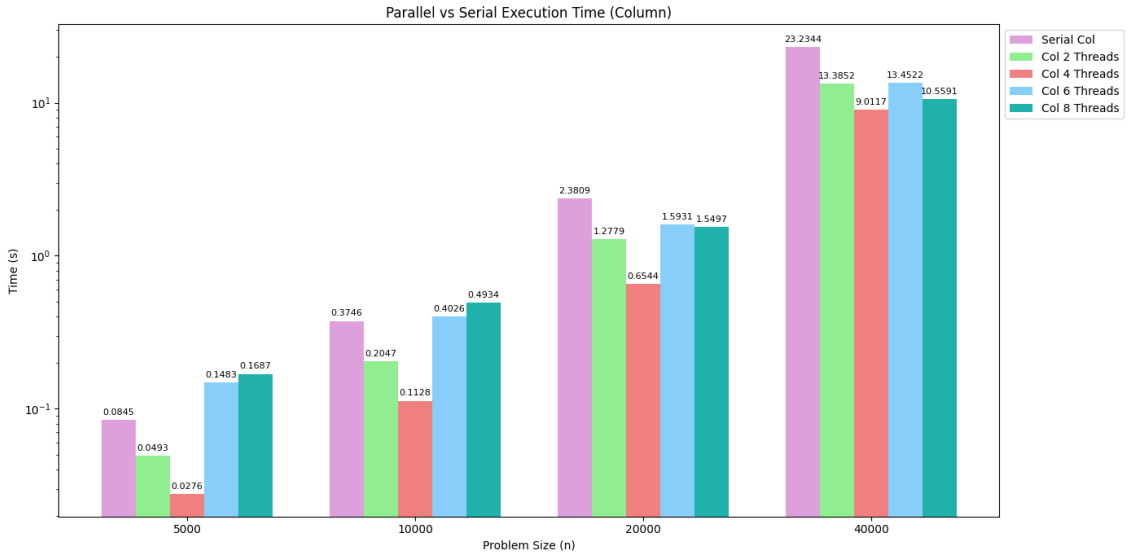
For the algorithms we followed the code given in the lecture, leaving it without modifications in the serial approach. As for the parallel version, the main limitation arises from the data dependency between the values of x. Specifically, to calculate x_{n-1}, the value of x_n is required. This limitation makes the parallel execution of the outer loops that traverse the values of the vector impossible.
So, in both cases, only the inner loop is parallelized, while the outer loop, since we don't add any directive, remains serial. We note that the entire algorithm is declared as a parallel section, with the threads receiving work through the corresponding omp for directives. In this way, we avoid creating new threads at each iteration of the outer loop, thus reducing overhead and improving efficiency.
In the "row-wise" algorithm, because the threads update the same element of the vector `x[row]`, synchronization is required using the reduction directive on a new `tmp` ariable, which is added at the end of the inner loop to compute `x[row]`. In the "column-wise" algorithm, both the first initialization loop and the inner loop that traverses the rows are parallelized. Here, each thread updates different positions of x, so synchronization is not necessary. To allocate the upper triangular matrix A, we used the technique described above. We then filled the diagonal elements and those above the diagonal with random values in the range $[-50, 50]^*$, and we did the same for vector b.
From our measurements, we observed that matrix initialization is the most time-consuming part of the program. For example, for size 40,000, initialization takes approximately 9.432 seconds, while solving the system takes only 2.32 seconds (serial row). For this reason, we attempted to parallelize the initialization. The parallel version, using 4 threads, reduced the initialization time to just 2.92 seconds, a significant improvement. Therefore, we use parallel initialization to save time. The measurements only include the

system-solving time, so they are not affected. It is worth noting here the value of optimizing memory allocation. We had concerns about using this technique because if we parallelized the algorithms by rows, since the rows are stored one after the other, false sharing could occur. If we had done things differently, in the classic way, we would have had 40,000 malloc calls, with the data being more sparse. However, this would increase the initialization time from 9.5s to around 40s, so in the end, this method is more efficient.
Running the program on the lab machines, we get the following results:

Size (n)	Algorithm	Row/Col	Threads	Time (s)
5,000	Serial	Row	-	0.0381
5,000	Parallel	Row	2	0.0321
5,000	Parallel	Row	4	0.0311
5,000	Parallel	Row	6	0.1896
5,000	Parallel	Row	8	0.2169
5,000	Serial	Col	-	0.0845
5,000	Parallel	Col	2	0.0493
5,000	Parallel	Col	4	0.0276
5,000	Parallel	Col	6	0.1483
5,000	Parallel	Col	8	0.1687
10,000	Serial	Row	-	0.1478
10,000	Parallel	Row	2	0.0978
10,000	Parallel	Row	4	0.0683
10,000	Parallel	Row	6	0.4088
10,000	Parallel	Row	8	0.4257
10,000	Serial	Col	-	0.3746
10,000	Parallel	Col	2	0.2047
10,000	Parallel	Col	4	0.1128
10,000	Parallel	Col	6	0.4026
10,000	Parallel	Col	8	0.4934

Size (n)	Algorithm	Row/Col	Threads	Time (s)
20,000	Serial	Row	-	0.5822
20,000	Parallel	Row	2	0.3311
20,000	Parallel	Row	4	0.2092
20,000	Parallel	Row	6	0.9138
20,000	Parallel	Row	8	0.9283
20,000	Serial	Col	-	2.3809
20,000	Parallel	Col	2	1.2779
20,000	Parallel	Col	4	0.6544
20,000	Parallel	Col	6	1.5931
20,000	Parallel	Col	8	1.5497
40,000	Serial	Row	-	2.3257
40,000	Parallel	Row	2	1.2044
40,000	Parallel	Row	4	0.698
40,000	Parallel	Row	6	2.1737
40,000	Parallel	Row	8	2.2184
40,000	Serial	Col	-	23.2344
40,000	Parallel	Col	2	13.3852
40,000	Parallel	Col	4	9.0117
40,000	Parallel	Col	6	13.4522
40,000	Parallel	Col	8	10.5591





Conclusions

Comparing the two algorithms, beyond parallelization, even in their serial versions we observe that the row-oriented algorithm performs significantly better than the column-oriented one. For small matrix dimensions, the difference isn't very noticeable, but for dimension 40,000, the row-oriented version is almost 10 times faster. This is because in the row-oriented algorithm we have sequential memory access, meaning we request consecutively adjacent memory locations, which clearly has better cache efficiency|a fact reflected in the runtime.

As for parallelization, we observe that the row-wise algorithm scales better, since as shown in the graph, its speedup is almost linear. The column-wise algorithm doesn't benefit as much from parallelization, possibly due to how consecutive rows are shared in the cache. Consecutive rows in the same cache mean that the next element of the column requested by a thread is closer in the cache hierarchy. We also observe that the speedup of the row-oriented algorithm drops dramatically after CPU thread oversaturation, with runtimes approaching the serial version. This may be caused by the additional overhead of synchronizing more threads. On the other hand, the column-oriented algorithm doesn't require synchronization, so after 4 threads, its speedup appears more stable.

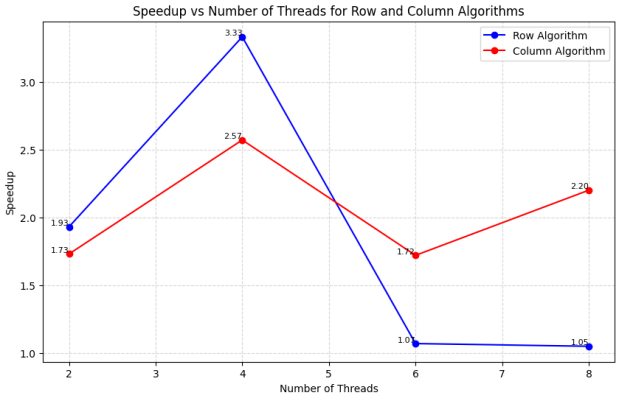
We present the speedup for a matrix of dimension 40000 below:

Row Speedup

Threads	T.Serial	T.Parallel	Speedup
2	2.3257	1.2044	1.93
4	2.3257	0.6979	3.33
6	2.3257	2.1736	1.07
8	2.3257	2.2183	1.05

Column Speedup

Threads	T.Serial	T.Parallel	Speedup
2	23.2344	13.3852	1.73
4	23.2344	9.0117	2.57
6	23.2344	13.4522	1.72
8	23.2344	10.5591	2.2



Scheduling

To experiment with different options, we set the parallelized loops with `schedule(runtime)`, where by changing the value of the environment variable `OMP_SCHEDULE` we can choose different options for distributing loop iterations across threads. To make the differences between scheduling options more noticeable, we selected the largest input size 40.000.
Below is the table of execution times with the various scheduling values, for the row-wise algorithm:

Row Time(s)

Threads	Static	Guided	Dynamic-1	Dynamic-100
2	1.2103	1.2222	20.73	1.3857
4	0.7117	0.7667	21.18	0.8836
6	2.158	2.085	22.742	2.1359
8	2.256	2.0014	23.158	2.1359

Observations

In general, we see that the standout options are guided and static scheduling modes. Specifically, in guided mode, we observe that it slightly underperforms in terms of execution time compared to static. This small difference is likely due to the fact that in static scheduling, there is no overhead for thread scheduling. Time is essentially saved because the number of iterations each thread will execute is predetermined, in contrast to guided, where this is calculated at runtime. We notice that both options yield execution times that are quite close to those in the earlier tables (where no specific scheduling option was selected), suggesting that the default might be either static or guided with a different chunk size. As for dynamic scheduling, the column with chunk size = 1 was included by mistake, as we initially overlooked the chunk factor, but it led to interesting results. We observe a huge increase in execution time, which gets worse as the number of threads increases. This makes sense, since each thread executes just one iteration before being assigned a new one, so most of the time is wasted in context switching and the overhead involved in distributing iterations. This was improved by setting a larger chunk size | now threads execute significantly more work before requesting a new chunk, so less time is wasted on scheduling overhead. Across all scheduling options, we observe | as before | a rise in execution time once the threads become oversaturated, likely for the reasons previously mentioned. Nevertheless, static scheduling managed to reduce this performance drop to a significant extent. Possibly, in static mode, more iterations that are close in memory were randomly distributed across threads running on the same core.