Report

Assignment 2

Ioannis Koutoulas and Alexandros Tsagkaropoulos

January 7, 2025

Exercise 2.1

Introduction

The Exercise studies the sequential and parallel implementation of Conway's *Game of Life*. The Game of Life consists of a square grid of cells The cell's state depends on its eight neighbors[1], one in each cardinal and intercardinal direction: North, North-East, East, South-East, South, South-West, West and North-West [5]. The rules that govern the state transitions [2, 4] are the following:

1. Any alive cell with fewer than two alive neighbors dies, as if by underpopulation.
2. Any alive cell with two or three alive neighbors lives on to the next generation.
3. Any alive cell with more than three alive neighbors dies, as if by overpopulation.
4. Any dead cell with exactly three alive neighbors becomes a alive cell, as if by reproduction.

The state of all grid cells changes simultaneously, in discrete steps; the generations. This property of Game of Life classifies it to the *embarrassingly parallel* problems. The parallel implementation of the Game of Life utilizes the OpenMP library.

Methods

The Game of Life functionality is aggregated in the `gol` module. The `gol` module consists of an opaque data type and six functions that operate on it. The opaque data type hides a C `struct` which contains pointers to the input and output grids (C arrays) and the grid side size. This design choice enables the interchangeability of grid pointers to mitigate the necessity to swap them in each generation. Also, the last parameter of the `struct` stores the grid size so as to iterate over the grid arrays by respecting their bounds. As for the functions, the two of them are responsible for the creation/initialization and destruction of the `gol` object, respectively. The other two are related to the configuration of the initial state of the grid. Specifically, the first takes an input from a file and the second initializes the grid with uniformly distributed dead and alive cells. The latter two execute the Game of Life, for a number of generations, in a sequential or parallel manner. The common logic includes the calculation of the neighbors belonging to a cell and, subsequently, the enforcement of the game's rules to a cell[2]. The parallel implementation utilizes the OpenMP library. A thread pool is created and the iteration over the grid cells is done in parallel. Each thread is responsible for a number of lines of the grid[3]. The scheduling of the iteration is `static` with the default `chunksize`, since the work for each grid cell is the same. The swapping of the input and output arrays at the end of each generation is performed by a single thread and constitutes an unavoidable serialization point.

The system on which tests were executed was part of the Linux Lab. The host name of the system was `linux30` and its operating system was Ubuntu 20.04.6 LTS. The CPU model was Intel(R) Core(TM) i5-6500 at 3.20 GHz with 4 cores. Lastly, the compiler used to create the binary executables was the GCC 9.4.0 with the optimization flag `-O2` set.

Results

All the experiments were automated by the script attached to the deliverable. Each experiment was run ten times and simulated 1000 generations with grid sizes: 64×64, 1024×1024, 4096×4096. The grid is initialized with uniformly-distributed data of possible values `0` (dead) or `1` (alive). For the parallel program the number of threads varied from one to sixteen with exponential step, namely: $1, 2, 4, 8, 16$. The results for timing measurements are depicted on Figure 1. In addition, the timing measurements and the speedup, efficiency metrics are gathered on the Table 1.

[1]The cells that reside on the the edges of the square grid have less that eight neighbors. However, to reason coherently about all the cells, it can be assumed that the neighbors which get out of bounds are always dead.

[2]Despite that their logic is relatively common, it could not be separated in subroutines in order to facilitate reusability. OpenMP resulted in noticeable speedup only when the use of subroutines was avoided. We could not find references that support this behavior.

[3]Another option was to use the `collapse(2)` OpenMP directive to assign contiguous cells of the grid to each thread that do not necessarily belong to the same line. Both methods ensure that each thread will fill up whole cache lines (64 B each). This way false sharing would be avoided when the update of any cell's state is performed and would be restricted to only when a thread gathers information from neighboring cells that are altered by other threads.

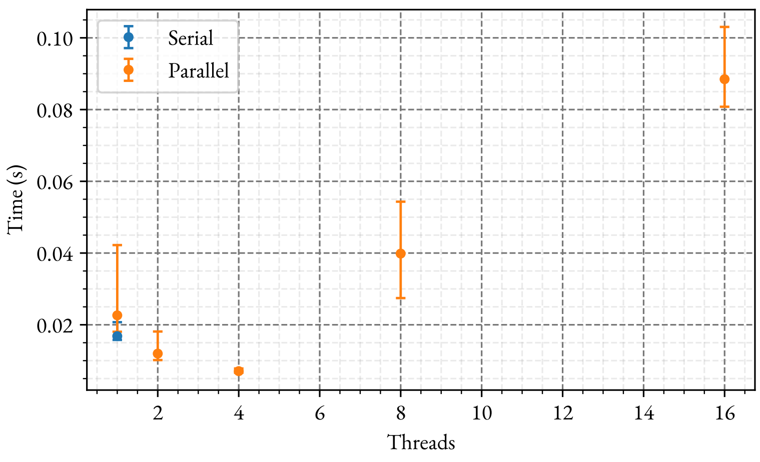
Threads	64 × 64			1024 × 1024			4096 × 4096		
	T	S	E	T	S	E	T	S	E
1	0.017	1.00	1.00	4.59	1.00	1.00	73.62	1.00	1.00
1	0.023	0.74	0.74	4.67	0.98	0.98	75.09	0.98	0.98
2	0.012	1.41	0.70	2.42	1.90	0.95	38.92	1.89	0.95
4	0.007	2.38	0.59	1.29	3.56	0.89	20.70	3.56	0.89
8	0.040	0.42	0.05	1.73	2.65	0.33	21.86	3.37	0.42
16	0.088	0.19	0.01	1.85	2.49	0.16	21.45	3.43	0.21

Table 1. Results from experiments performed with 1000 generations. Each value in the time (T) columns represents the mean of ten runs with uniformly-distributed data and is measured in seconds. The speedup (S) and efficiency (E) metrics are calculated with respect to the first data row, which corresponds to the sequential execution.

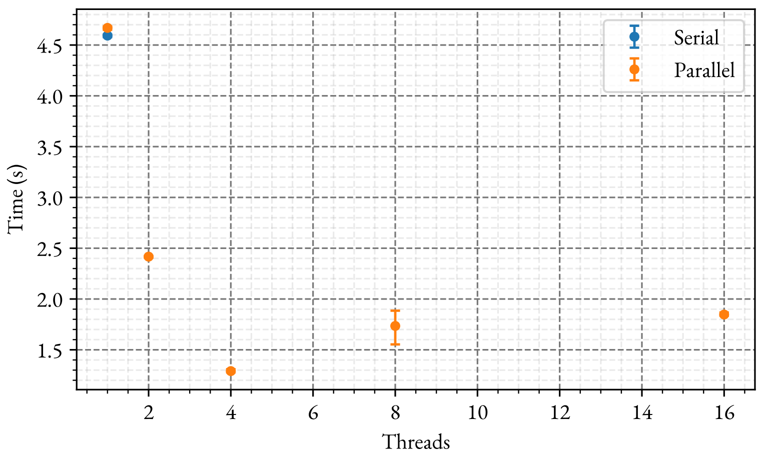
Discussions

The Game of Life rules render it embarrassingly parallel, since by simple diving the work among threads the algorithm can be parallelized. The results in Figure 1 confirm that for every grid size tested there is a notable speedup when the system is not oversubscribed. In the oversubscribed territory, where the number of threads outgrows the available cores, the overhead of scheduling is enough to degrade the execution time. The size of the error bars indicate that the execution times vary more in smaller grid sizes. This is due to the fact that false sharing and the serialization part of the algorithm are more pronounced in that case. Specifically, when a thread operates on the first or last line of the grid's part that is assigned to it, it necessarily fetches the neighboring cell's state that reside outside its part. Those cells are processed by other threads and, therefore, the cached values get invalidated at every generation. As for the serialization part, it presupposes that all threads have finished processing their part when one of them will exchange the input and output array pointers. The implicit barrier at the end cannot be omitted, since all threads should start the next generation's computations with the array pointers exchanged. In addition, the serial implementation of the algorithm achieves smaller execution time from the parallel one using one thread. This behavior is expected since the OpenMP library introduces an overhead comparing to the "bare-bones" implementation.

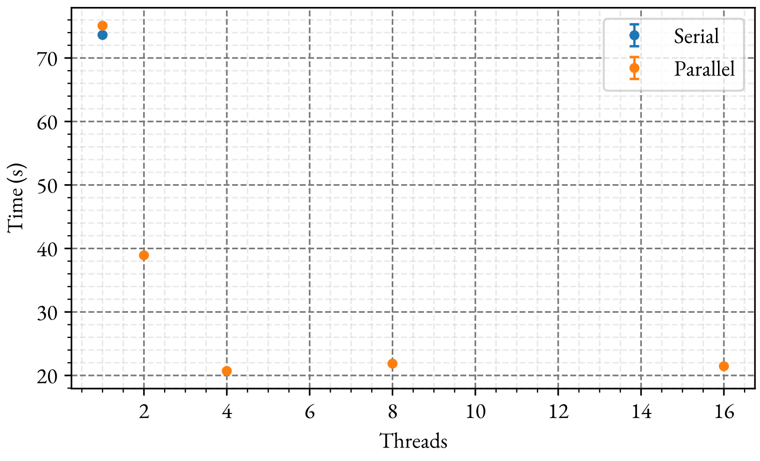
The speedup and efficiency metrics are summed up in Table 1. Both metrics have statistically the same values, when the grid size is 1024 × 1024 or 4096 × 4096, since their deviation is less that 0.5%. They deviate from the optimal values no less that 11%, which is acceptable value due to the innate serialization part of the algorithm. The values for the speedup and efficiency is worse with the grid size 64 × 64. This remark indicates that the problem might be weakly scalable, but more experimental results are needed for this categorization to be made.



(a) Grid size of: 64×64.



(b) Grid size of: 1024×1024.



(c) Grid size of: 4096×4096.

Figure 1. Results from experiments performed with 1000 generations. Each mark-point in the graphs represents the mean of ten runs with uniformly-distributed input data. Error bars indicate the minimum and maximum values obtained across all runs.

Exercise 2.2

Introduction

This exercise studies the parallelization of two algorithms to solve linear systems of n equations and n unknown variables. The system of n equations is represented from a coefficient matrix A of size $n \times n$, that multiplies a vector of the unknown variables x of size $n \times 1$ and the result is equal to another $n \times 1$ vector b that represents the zero-order coefficients of the equations. Matrix A can be transformed to an upper-triangular matrix using the Gaussian Elimination Method [6] and then it can be solved producing unique solution to the linear system, other than the trivial zero solution. The method to find the solutions after the matrix A is transformed into an upper-triangular matrix is called back-substitution [7].

In this study two back substitution algorithms will be used to produce solutions and to study their parallelization properties. The one method is *back substitution by row* and the other is *back substitution by column*. These two algorithms are presented in Listing 1 and Listing 2, respectively. The back substitution by row method produces the first unknown variable from the single element of the last row of matrix A along with the corresponding zero-order coefficient from vector b and then uses it to compute the next unknown variable, from the second from last row of matrix A, that contains two elements, and so on and so forth. The outer `for` loop in line 1 of Listing 1 traverses the matrix by row and the inner `for` loop in line 4, goes through the elements of each row. Because the matrix is upper triangular it means that this `for` loop will start from one single iteration and it will increment by one with each new iteration of the outer `for` loop, eventually reaching n iterations. In the back substitution by column method the algorithm begins with a `for` loop that initializes all the unknown variables to the value of the zero-order coefficients of vector b, as shown on line 1 of Listing 2. Then, moving from the leftmost to the rightmost columns of matrix A (`for` loop in line 4), it computes partial multiplication and accumulations for all the rows in each column (`for` loop in line 6) until eventually, after traversing all the columns of the matrix, it computes the correct unknown variable values.

```
1 for (row = n-1; row >= 0; row--)
2     x[row] = b[row]
3
4     for (col = row+1; col < n; col++)
5         x[row] -= A[row][col] * x[col]
6
7     x[row] /= A[row][row]
```

Listing 1. Back substitution by row algorithm pseudo-code.

```
1 for (row = 0; row < n; row++)
2     x[row] = b[row]
3
4 for (col = n-1; col >= 0; col--)
5     x[col] /= A[col][col]
6     for (row = 0; row < col; row++)
7         x[row] -= A[row][col] * x[col]
```

Listing 2. Back substitution by column algorithm pseudo-code.

The goal of this exercise is to create at first a simple sequential version of the two algorithms based on the pseudo-code shown in Listings 1 and 2 and then try to modify them into parallel execution multi-threaded versions, utilizing the OpenMP library. We assume that the equation system is represented by an already initialized upper triangular matrix and an already initialized zero order coefficient vector, so that the two algorithms can be applied immediately to produce solutions for the unknown variables. The parallel code versions' execution will be compared to the sequential versions' for different problem sizes and number of threads, as well as various OpenMP schedules.

Methods

The attempt to parallelize the two algorithms will be based on creating data parallelism versions of the sequential code by parallelizing the execution of the `for` loops, wherever this is possible. This is achieved by the `omp parallel for` directive provided by the OpenMP library. By calling this directive before a `for` loop the execution of different iterations of the loop is assigned to different threads. If the system has a multi-core CPU true data parallelism can be achieved. It is important though that the iterations of the `for` loop do not have dependencies. Thus not all these loops of the two algorithms can be parallelized.

In the case of the back substitution by row algorithm the outer `for` loop (line 1 in Listing 1) cannot be parallelized because to compute `x[row]` with the multiplication and subtraction operation on the inner for loop (line 4) previously computed values of vector x elements with larger index are needed, as was described in the Introduction. Therefore the outer for loop must execute in a sequential matter, incrementing the iterations of the inner `for` loop with every new iteration. However, the inner `for` loop presents no internal dependencies. There is no particular order that the elements of the row of matrix A, along with the previously computed unknown variable values of vector x, should be combined to finally produce the new value of the unknown variable `x[row]`. This operation in line 7 of Listing 2 can be performed independently by different threads, for various values of `column`. In the end, they should accumulate in the final value of `x[row]`. This is a reduction operation of addition (subtraction to be more precise) and OpenMP provides the reduction clause that automatically deals with the critical section of the memory space of the final variable `x[row]` and also accumulates the various values in the most effective way (depending on the OpenMP implementation). Finally the parallel version of the code consists of the same code as the sequential with a `omp parallel for` directive before the `for` loop in line 4, with a reduction clause on a local variable to store the value of `x[row]` of the pseudo-code.

Back substitution by column algorithm contains three `for` loops, from which the first one (line 1 in Listing 2) is completely independent from the other two. It presents no internal dependencies, since it only assigns as initial values to the unknown variables the independent values of the zero-order coefficients. This loop can be parallelized with the `omp parallel for` directive. The outer `for` loop in line 4 cannot be parallelized. The way the code is structured, the operation in line 5 assumes that in the previous iteration the correct value has accumulated in the `x[column]` value from the inner `for` loop in line 4. If this happens out of order the algorithm is incorrect because the division by `A[col][col]` operation might happen prematurely. The `for` loop in line 6 however can and will be parallelized. In every iteration of this loop an independent value `x[row]` from those of the other iterations is computed, so the iterations can be safely executed by different threads, sharing the value of `x[col]`. The `omp parallel for` directive will, once again, be used before this loop.

The sequential and parallel versions of the algorithms have the form of C functions, that are called by the main program. The main program takes as command line arguments which algorithm to execute and wether to execute the sequential or the parallel version. For the sequential versions it will measure and print the execution time, and it will also compare the results to those of the other algorithm. For the parallel versions it will measure and print the execution time on the console, but it will also measure and print the speedup of the parallel algorithm. Speedup is defined as the ratio of the execution time of the fastest sequential algorithm –in this case this is the provided sequential versions– to that of the parallel algorithm. It will also compare and print the relative error of the results of the sequential and parallel version of the algorithm. The main program also takes as command line argument the number of threads, to specify them in the `omp parallel` directives, as well as the size of the system, to initialize the matrix A as an upper triangular matrix of random values and the coefficient vector b with random values too. For random value generation the standard library function `rand()` was used, with the same seed for all the executions. The timing measurements do not take into account the initialization of variables and structures of any kind prior to the execution, following the [3]. The timing measurements utilized the `timer.h` header file that was given in Assignment's 1 supplementary material [1].

The executables are built from C code using the make build system and their build and execution are performed by a dedicated python script. The goal is to compare the performance of the parallel versions of the two algorithms for different numbers of threads and system sizes. Other python scripts automate this process by running the two algorithms for varying arguments ten times over, so as to capture their speedups, produce their average and store it in appropriate files. This scripts also assures that the relative error of the parallel methods compared to the sequential ones does not become too big, or else the results are rejected.

In OpenMP the user can also specify the scheduling of the parallel execution of a `for` loop; that is how the iterations of the loop will be grouped together for execution on a thread. This is achieved by the `schedule` directive. In this report `schedule(runtime)` is used so that different scheduling can be specified in the runtime, by setting up the `OMP_SCHEDULE` environmental variable, to see how they affect the speedup of the parallel versions. This is done by the python scripts that produce the final results.

The system on which tests were executed was part of the Linux Lab. The host name of the system was `linux29` and its operating system was Ubuntu 20.04.6 LTS. The CPU model was Intel(R) Core(TM) i5-6500 at 3.20 GHz with 4 cores. Lastly, the compiler used to create the binary executables was the GCC 9.4.0 with the optimization flag `-O0` set.

Results

The results of the back substitution by row method are summarized in Table 2 and the results of back substitution by column in Table 3. These tables contain the average speedup values over ten executions, for varying OpenMP schedules, problem size and number of threads. Speedup is considered linear and optimal whenever its value approaches the number of threads used.

All the available OpenMP schedules have been tested. For each schedule there are runs with different chunk sizes. For `static` schedule the default chunk size is the number of `for` loop iterations over the number of threads, while for `dynamic` and `guided` the default chunk size is 1. For `static` scheduling the default chunk size and chunk size 1 constitute, in a way, the boundary cases for all the possible useful chunk sizes, one being the largest that would make sense to use while the latter being the smallest. For `dynamic` scheduling the opposite of the default would be to use as big of a chunk size as possible, which is denoted as maxchunk on the Tables and it is precisely the number of `for` loop iterations over the number of threads. The `guided` schedule decreases the chunk size as a chunk completes and in the default case it starts form maxchunk value and it decreases it down to chunk size 1. So the opposite case is to not decrease it at all by specifying `guided, maxchunk`, where maxchunk is the number of `for` loop iterations over the number of threads again. In a way this almost identical to `dynamic, maxchunk`.

The "Size" columns in the Tables denotes the problem size, which is the number of equations of the linear system and, consequently, the number of unknown variables. Smaller problem sizes showed no speedup over the sequential execution or had wildly varying results. Larger problem sizes did not execute because the programs resulted in stack overloads, due to the large size of the matrices and vectors that the programs used.

Finally the number of threads that where passed as arguments to the main program ranges from 1 to 4, since the tests where ran on a system with a 4 core CPU. Additionally, for each execution, there is a run for 8 cores to study the effects of thread thrashing. It is important to note that for 1 thread the speedup is the execution time of the sequential algorithm over the execution time of the parallel algorithm run with only one thread, making it effectively a sequential algorithm too.

Discussions

Both back substitution algorithms' results present some similarities, that are worth discussing before the intricacies and nuances of each one. To begin with, both present speedup for most cases, especially as the size of the problem rises. For the smallest problem size of 2500 equations there are cases where the speedup is slightly smaller than one, which basically means there is a slowdown with the parallel execution. This is due to thread creation overhead in every iteration of the outer `for` loop for both algorithms (line 1 for Listing 1 and line 4 for Listing 2). Furthermore, for 8 threads, the effects of thrashing are so severe that speedup is either a slowdown (values smaller than 1) or it is a bit more than 1, deviating a lot from linear speedup performance. Results for 8 threads will not be included further in the following discussions.

Interestingly some cases present super-linear speedup. Most of them are for the single-threaded execution. This is not as strange as it seems, because most of these executions take a long time in a system that already runs some other processes, so it is statistically possible that the sequential execution got delayed sometimes, compared to the single-threaded one, due to context switching of the Linux system. For small problem sizes there is always a slowdown for the single threaded execution, due to the thread creation overhead. This overhead does not have an important effect for large systems, where it is possible that for some runs the single-threaded execution has better results than the fastest sequential algorithm. In any case the super-linear speedups are never more than 14% of the expected linear speedup.

Back substitution by row method presents exceptionally good results very close to linear speedup for large problem sizes. For every scheduling type the speedup increases with the problem size and at all cases the best performance is for the largest size at 40000 equations. All the schedules have very similar results and no schedule has consistently better results than the other. This is expected due to the fact that even though the iterations of the inner `for` loop (line 4 in Listing 1) increase with every new iteration of the outer `for` loop (line 1 in Listing 1) the work of every iteration of the inner loop is the same. Furthermore the most super-linear speedup cases are for a large number of iterations. This implies that for large problem sizes the sequential program might suffer from cache misses. These do not happen so much in the parallel version, because each core has local caches, and because the program data break down to groups for different threads to execute their size is smaller, and thus the cache misses reduce.

Back substitution by column method presents increasing speedup as the problem size rises. The biggest speedup is for problem size of 10000 equations and then the speedup drops slightly and remains steady for larger problem sizes. This happens due to a combination of many factors that plague the parallel execution of this method. One is the effect of *false sharing*. Many elements in consecutive iterations of the inner `for` loop (line 6 in Listing 2) are neighboring elements of the x vector. The same is true for the first `for` loop in line 1 of Listing 2 which is parallelized too; in this there is also false sharing for the elements of vector b. Large parts or even the entirety of x and / or b vector could fit inside a single cache line and as the vector is updated there is a large number of cache line invalidations, that result in sharing of the cache lines between the threads. This did not happen in the back substitution by rows method because in the parallel `for` loop of line 4 in Listing 1 only local variables where updated, that combined in the end via the reduction clause. In this algorithm consecutive neighboring elements of x are updated and there is no elegant mechanism or workaround to avoid false sharing. However, as was seen in the execution of the back substitution by row method, cache memory can have a positive effect in the parallel code with the use of local caches. Unfortunately the code for back substitution by columns cannot benefit from local caches. This is because the inner `for` loop in line 6 in Listing 2, which is parallelized, traverses the elements of matrix A by row, as it is seen in line 7 of Listing 2. This is an intricacy of the C programming language and of its compilers; elements of two dimensional arrays are stored in row major order. This means that cache lines will most likely contain neighboring elements of a row of matrix A. The operation in line 7 of Listing 2 skips entire rows, which results in cache misses, especially for larger matrix sizes. All the different schedules exhibit similar speedups once again, because the inner `for` loop of the algorithm in line 6 in Listing 2 has the same work for all the iterations, which is just a multiply accumulate operation of line 7 in Listing 2. Also no schedule offers any particular advantage over the cache coherence problems discussed above. To conclude the parallel version of the code does provide some speedup but for large problem sizes cache coherence problems arise, while for small sizes there is thread creation overhead and false sharing. The best balance is struck in problem size of around 10000 where speedup approaches optimal values (linear speedup).

Schedule	Size	1 Thread	2 Threads	3 Threads	4 Threads	8 Threads
static	2500	0.804	0.942	1.077	1.141	0.118
	5000	0.910	1.491	1.490	1.680	0.262
	10000	0.977	1.549	2.021	2.336	0.491
	20000	1.208	2.110	2.855	3.423	0.984
	40000	1.245	2.267	3.174	3.966	1.625
static, 1	2500	0.805	0.933	1.089	1.142	0.116
	5000	0.913	1.493	1.528	1.677	0.263
	10000	0.977	1.543	2.021	2.331	0.491
	20000	1.211	2.097	2.861	3.436	0.975
	40000	1.243	2.260	3.197	3.966	1.568
dynamic	2500	0.806	0.937	1.085	1.144	0.124
	5000	0.912	1.485	1.526	1.676	0.260
	10000	0.977	1.547	2.020	2.333	0.469
	20000	1.211	2.099	2.833	3.456	0.989
	40000	1.248	2.268	3.177	3.985	1.619
dynamic, maxchunk	2500	0.805	0.934	1.086	1.142	0.115
	5000	0.911	1.492	1.526	1.684	0.257
	10000	0.978	1.546	2.018	2.331	0.469
	20000	1.219	2.117	2.830	3.462	1.022
	40000	1.238	2.265	3.161	3.960	1.578
guided	2500	0.807	0.931	1.086	1.161	0.133
	5000	0.912	1.500	1.519	1.667	0.246
	10000	0.975	1.546	2.025	2.332	0.467
	20000	1.210	2.123	2.839	3.467	1.030
	40000	1.241	2.270	3.188	3.967	1.582
guided, maxchunk	2500	0.807	0.938	1.086	1.102	0.127
	5000	0.913	1.489	1.527	1.679	0.254
	10000	0.977	1.547	2.024	2.312	0.473
	20000	1.223	2.099	2.828	3.416	1.004
	40000	1.238	2.265	3.176	3.947	1.577

Table 2. Average speedup results for back-substitution by row method.

Schedule	Size	1 Thread	2 Threads	3 Threads	4 Threads	8 Threads
static	2500	0.982	1.615	2.041	2.227	0.327
	5000	0.985	1.852	2.645	3.214	0.694
	10000	0.995	1.931	2.899	3.917	1.269
	20000	1.012	1.877	2.636	3.347	2.439
	40000	1.034	1.887	2.596	3.329	2.621
static, 1	2500	0.992	1.622	2.034	2.198	0.354
	5000	0.989	1.861	2.666	3.241	0.756
	10000	0.998	1.937	2.908	3.898	1.286
	20000	1.012	1.890	2.647	3.342	2.409
	40000	1.030	1.875	2.645	3.325	2.584
dynamic	2500	0.993	1.612	2.028	2.200	0.352
	5000	0.993	1.871	2.679	3.133	0.742
	10000	0.996	1.927	2.922	3.970	1.261
	20000	1.012	1.882	2.646	3.338	2.448
	40000	1.033	1.872	2.656	3.337	2.617
dynamic, maxchunk	2500	0.994	1.622	2.037	2.219	0.363
	5000	0.991	1.863	2.664	3.175	0.789
	10000	0.999	1.929	2.924	3.942	1.259
	20000	1.012	1.892	2.637	3.386	2.440
	40000	1.036	1.882	2.672	3.362	2.646
guided	2500	0.996	1.626	2.026	2.216	0.332
	5000	0.990	1.864	2.668	3.148	0.765
	10000	0.999	1.929	2.898	3.879	1.237
	20000	1.012	1.878	2.647	3.387	2.431
	40000	1.035	1.874	2.670	3.380	2.633
guided, maxchunk	2500	0.995	1.632	2.020	2.223	0.319
	5000	0.992	1.859	2.675	3.193	0.742
	10000	0.999	1.921	2.899	3.869	1.277
	20000	1.012	1.879	2.634	3.377	2.447
	40000	1.033	1.888	2.663	3.373	2.595

Table 3. Average speedup results for back-substitution by column method.

References

[1] Vasileios Karakostas. *Assignment 1 (Mandatory) – Programming with Pthreads*.

[2] Vasileios Karakostas. *Assignment 2 (Mandatory) – Programming with OpenMP*.

[3] Peter Pacheco, Matthew Malensek. *An Introduction to Parallel Programming*. 2nd Edition.

[4] *Wikipedia - Game of Life*. https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life.

[5] *Wikipedia - Cardinal Direction*. https://en.wikipedia.org/wiki/Cardinal_direction.

[6] *Wikipedia - Gaussian Elimination*. https://en.wikipedia.org/wiki/Gaussian_elimination.

[7] *Mathwords - Back-Substitution*. https://www.mathwords.com/b/back_substitution.htm.