Report

Assignment 3

Ioannis Koutoulas and Alexandros Tsagkaropoulos

January 17, 2025

Exercise 3.1

Introduction

The Exercise studies the 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 [1, 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 Message-Passing Interface (MPI) library.

Methods

The Game of Life functionality is aggregated in the `gol` module. The interface was extended and updated compared with the solution provided in Assignment 2. Specifically, 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: (a) pointers to the input and output grids (borders inclusive); (b) the total (global) grid size; (c) the (local) grid size assigned to each process[2]; (d) MPI specific state for communications. This design choice enables the interchangeability of grid pointers to mitigate the necessity to swap them in each generation. In addition, both global and local grid sizes facilitate the iteration over the corresponding arrays[3]. For example, the process with rank 0 utilizes the global grid size to create, fill, distribute and gather the array on which Game of Life executes. Simultaneously, all processes operate on the part of the array that was assigned to them by respecting the local grid size. To exploit the C language's array semantics for performance, block partitioning of rows was used to split the work among processes.

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. Both function contain logic to distribute the grid to all participating processes. Each process receives its part —determined from the block partitioning policy— and the North and South neighboring rows of the grid. The semantics remain consistent with the implementation of Assignment 2, since all processes store the part of the grid they operate on and abstract their neighbors as being on the "border". This is a viable way of looking at the problem due to the fact that the "border" cells are not modified by the corresponding process, but are only needed to update the inner cells of the grid. The fifth function executes the Game of Life, for a number of generations, in a parallel manner, using the MPI library. The processing part of each generation is preceded by a communication part. In the latter part, each process sends updates for the neighbors (or "horizontal borders") of processes that handle neighboring grid parts and stores the updates that receives by them to its own grid borders. The former part consists of calculating the neighbors of each cell and enforcing the Game of Life rules. Before the next generation begins the processes must synchronize, by a barrier, which constitutes an unavoidable serialization point. The last function aggregates the computed values from all participating processes and prints them on the `stdin`.

The system on which tests were executed was part of the Linux Lab. The machines that constitute the Linux Lab display identical hardware and software characteristics. Specifically, the CPU model was Intel(R) Core(TM) i5-6500 at 3.20 GHz with 4 cores and the operating system was Ubuntu 20.04.6 LTS. The host names of the machines were ranging from `linux01` to `linux30`. The compiler used to create the binary executables was the GCC 9.4.0 with the optimization flag `-O2` set. To facilitate the compilation process the `mpicc` wrapper script was utilized. Lastly, each node was configured so as to host no more than four processes simultaneously, when oversubscribing is absent.

[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] It is assumed that the division of the grid to processes does not have any remainder. Therefore, all the processes would store a `gol` object with the same value for the local grid size.

[3] The terms "global" and "local" have no resemblance to the concepts of shared memory systems. The MPI library presupposed a distributed memory model. In that context, the whole grid is referred to as "global" and the part of the grid that is handled by a process as "local".

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: $128 \times 128,\ 1024 \times 1024,\ 8192 \times 8192$. The grid is initialized with uniformly-distributed data of possible values `0` (dead) or `1` (alive). The number of processes used where: $1,\ 4,\ 16,\ 64,\ 128$. It was observed that the MPI runtime was assigning processes to nodes in a block partitioning manner. 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.

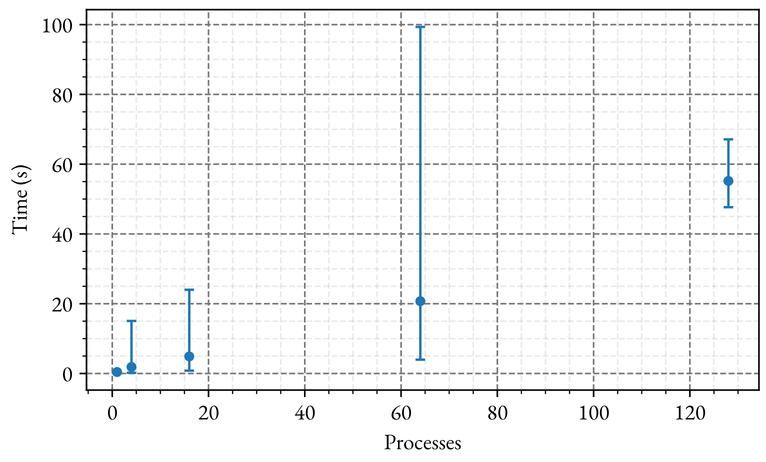
Processes	128 × 128			1024 × 1024			8192 × 8192		
	T	S	E	T	S	E	T	S	E
1	0.4	1.00	1.00	26.9	1.00	1.00	1755	1.0	1.00
4	1.9	0.23	0.06	14.6	1.84	0.46	622	2.8	0.71
16	4.9	0.09	0.01	6.1	4.39	0.27	175	10.0	0.63
64	20.7	0.02	0.00	14.4	1.88	0.03	64	27.2	0.43
128	55.2	0.01	0.00	58.5	0.46	0.00	122	14.3	0.11

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 execution with one process on a single host.

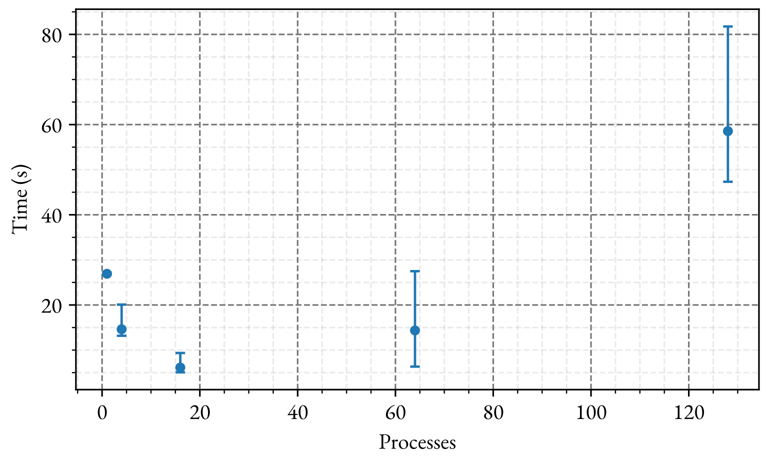
Discussions

The Game of Life implementation with the MPI library is weakly scalable. Specifically, the Figure 1 depicts that in order to achieve notable speedup by increasing the number of processes, the grid size should be increased as well. Two factors result in this behavior. First, when the grid size is smaller the communication time between processes becomes relevant and might even surpass the computation time. Second, each generation requires synchronization of processes before proceeding to the next, thus adding extra overhead. In addition, when the grid size is smaller, the variation of results is larger. This phenomenon could be caused by two factors. First, there is an inherent randomness in network latencies which are amplified when the total execution times are smaller. Second, the smaller grid sizes where executed earlier in day when the Lab's computers where almost fully occupied. Consequently, the scheduling policy of the OS could have affected the experimental data. When the grid size is 8192×8192, as illustrated on Figure 1c, the speedup is evident for process numbers ranging from 1 to 64. Nevertheless, when the process number is 128, a noticeable deterioration of the results is observed. This is due to the fact that the Linux Lab gets oversubscribed, since it can only handle up to 120 processes on its hosts. Consequently, the execution time when using 128 processes is visibly worse in every case.

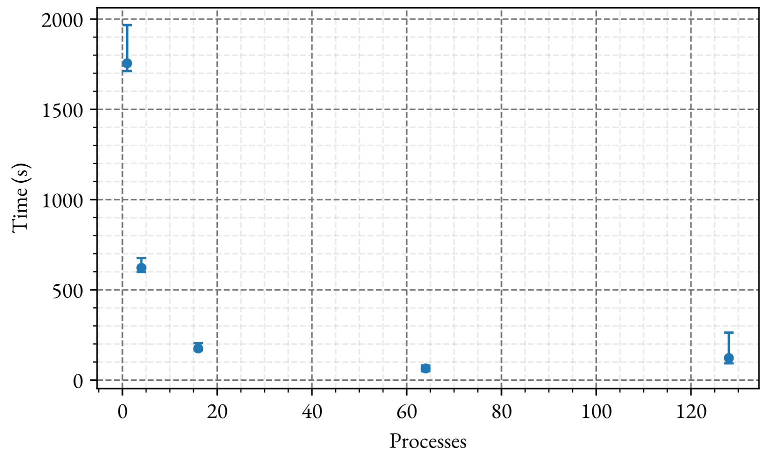
The Table 1 confirms and enhances the observations made on the experimental result in the previous paragraph. Specifically, the speedup and efficiency metrics are calculated to provide a more robust and complete view of the results. In the smallest grid size tested, the increase in processes count results in slowdown and degradation of the efficiency. The speedup is more promising in the case of 1024×1024 grid size. There is measurable speedup when using 4 or 16 processes. When the problem is partitioned to a larger number of processes the speedup and efficiency are worsened. The problem with grid size 8192×8192 receives noteworthy speedup when the system is not oversubscribed. However, the speedup and efficiency metrics are far from ideal. This behavior, that spans all grid sizes, is declarative of the communication and synchronization overhead.



(a) Grid size of: 128×128.



(b) Grid size of: 1024×1024.



(c) Grid size of: 8192×8192.

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 3.2

Introduction

This exercise involves a matrix vector multiplication program. In this a square matrix A of size $n \times n$ will be multiplied with a vector x of size $n \times 1$ to produce a result vector y of size $n \times 1$. This will be a multi-process application that uses the Message-Passing Interface (MPI) standard. The goal is to run the program in a network consisting of PCs set up as MPI machines. Each MPI process will run on a node, which can be a process running on one of the cores of a PC or to a core of another PC of the network. The basis of the parallel execution is the partitioning of the data of the sequential problem to create a *data-parallel* version of it. Each MPI process will compute a partial solution using a block of the original problem data. In the end all this partial solutions must combine to produce the final result.

To be more specific, in this implementation the matrix A will undergo a partition by columns. This means that each process will work with a *sub-matrix* block of data from the initial matrix, consisting of some whole columns of it. It is a requirement of the exercise that the dimension n of the matrix be user specified and so must be the number of MPI processes, which will be denoted as `comm_sz`. Furthermore, the number of processes must divide the dimension n of the matrix exactly. This way, all processes will have a block of data of the same size. Each one precisely needs a sub-matrix of size $n \times (n/\texttt{comm\_z})$. The implication of this is that the vector x must also be partitioned into sub-vectors of size $(n/\texttt{comm\_z}) \times 1$. Each process will perform a local multiplication of the sub-matrix block with the sub-vector block to produce a local partial result, which will be a vector of size $n \times 1$. In the end, all this local vectors must add up to the final result vector y, completing the matrix-vector multiplication.

All the initial program data of the matrix A and the vector x must be initialized by a single process, which will also gather the final result. This process will have the responsibility to send the partitioned data to the other processes and must send only the data they need. In this report the execution of the sequential program will be compared to that of the parallel program, by examining the *speedup* and *efficiency* metrics. There will also be a brief discussion on data sharing time versus execution time in MPI systems.

Methods

In this Section the methods of the implementation are discussed. The submitted code contains the `source` folder that has two C programs; the one is the sequential implementation and the other is the parallel implementation of the problem. Both programs utilize the custom library `linalgebra.h` which provides functions to initialize, print and operate on vectors and matrices (more details of the functions are documented on their contracts).

The sequential program receives command line arguments and after the execution it prints the execution time of the matrix-vector multiplication. If `#define VERBOSE` is set then it also prints the contents of the initial vector x and matrix A as well as the result vector y. For timing it uses the `timer.h` header file, provided by the accompanying material of [3]. The matrix A and the vector x are initialized with random values by using the `rand()` function with a fixed seed. It is important to note that all the following programs of this implementation that use these random-value initialization function will have the same seed, for consistency of results throughout different executions.

The parallel program implements the matrix-vector multiplication with the partitioning by-columns method described in the Introduction. The execution flow passes through four functions called by the `main` function, which is also responsible to perform checks on the user input of the problem size and the number of processes. Inside those four function some code parts branch-out based on the rank of the process; this is how the initialization logic from the process with rank 0 is achieved. The aforementioned functions are listed below with a brief explanation of their purpose. More details on their operation and their arguments can be found on their contracts.

- `problem_size_partitioning` : Calculates the size of the sub-matrix block and the sub-vector block that each process will use to compute its local result.
- `local_memory_allocations` : Allocates enough memory to store the local data blocks and results. The rank 0 process will need to allocate additional memory in order to store the initial matrix A and vector x as well as the result vector y.
- `parallel_matrix_vector_multiplication` : This function performs the local matrix-vector multiplications on each MPI process. At first it distributes the data that each process needs and then performs the local operations. Then it reduces the local results into the final result vector in process 0, by summing them.

- `local_memory_deallocations` : this function deallocates all the dynamically allocated memory on every MPI process.

The most important part of the parallel code is the execution of the `parallel_matrix_vector_multiplication` function. The way the local process data are shared from process 0 to all the other processes is with the use of custom MPI datatypes. Process 0 has allocated and initialized the matrix A and the vector x. In every process it sends a part of the matrix and the vector x. To be more precise it keeps for itself the first $n/$`comm_sz` columns of the matrix, it sends the next $n/$`comm_sz` to process 1, and so on. Similarly, it keeps for it self the first $n/$`comm_sz` elements of vector x, it sends the next $n/$`comm_sz` elements to process 1 and so on. To find which part of the data to share a custom datatype is created for every process which, beginning from the first element of matrix A it calculates the offsets of the elements that the sub-matrix and sub-vector block must contain. In this project this is implemented by an MPI custom datatype array that is initialized by the function `build_mpi_type`. This function uses the built-in API of MPI to create and commit types. After the types are created the process 0 calls `MPI_Send` for every other process, sending to each one the corresponding datatype that was created for it. After the datatypes are used to copy data to the local buffers of each process, they are deallocated. All processes, other than process 0, call `MPI_Recv` to receive these data. More details on the implementation can be found in the extensively commented code of these functions.

Inside the `parallel_matrix_vector_multiplication` function the timing of the parallel program is also performed. This is achieved by taking timing snapshots using the `MPI_Wtime` function. Before taking the initial snapshot `MPI_Barrier` is called, so that all processes start the timed code block approximately at the same time. Inside the function there are two timed code blocks. The first one is the part where the process with rank 0 shares the local data that each process needs. The second timed block includes the local sub-matrix with sub-vector multiplication and the final addition reduction operation to store the final result in process 0, which happen after the necessary local data have been received by all the processes. This second code block will be regarded as the parallel's program execution time, that will be used to produce the speedup and efficiency metrics. The first code block is considered as the MPI data sharing time. All this is in accordance with the MPI timing examples provided by [3].

The executables for the sequential and the parallel program are built from the C source code using the make build system. The sequential program uses the gcc compiler, while the parallel program uses the mpicc compiler wrapper provided by the used MPI implementation of mpich. In either case the optimization flag was set to `-O0`. The build and execution of the programs, along with the insertion of the correct command line arguments is performed by dedicated python scripts. Other scripts perform executions over various problem sizes and different number of processes and store the timing results in files. From these files other scripts compute the results that will be presented in the following Section.

The system on which the programs ran was the PC network of the DI Linux Lab. A list of setup MPI machines was provided for the exercise. This file contained all the 30 PCs of the lab and also specified that up to 4 MPI processes can run on each one. The operating system of the PCs was Ubuntu 20.04.6 LTS. Their CPU model was Intel(R) Core(TM) i5-6500 at 3.20 GHz with 4 cores.

Results

The sequential and parallel program where executed for various problem sizes and different number of MPI processes. The problem size is the dimension n of the square matrix A and the number of elements of vectors x and y. The number of processes is only valid for the parallel program. In the previous Section the system of the DI Linux Lab was discussed and how the PCs are configured as MPI machines. Each one can execute up to 4 processes and they schedule these processes in a *round-robin* fashion. This means that if the user inputs 8 processes as an argument to the parallel program then the first two PCs will execute 4 processes each. Since it is a prerequisite of the assignment to have the number of processes divide exactly the problem size, it was decided to use problem sizes in the powers of 2. Also it was decided to use processes that are multiples of 4, so that there is load balancing in the different PCs of the DI Linux Lab.

Both programs print to the terminal the time it took for the matrix-vector multiplication. The parallel program also print the time it took to share the local data to all the processes. From the execution times the speedup and efficiency metrics can be computed. Speedup is the ratio of the sequential program's execution time to the parallel program's execution time. Efficiency is the ratio of the speedup of the parallel program to the number of processes it used. For the current program the measured speedup is reported in Table 2, while the efficiency is reported in Table 3. These results are the average values over ten executions with the given program inputs.

There are also Tables that present timing results. Table 4 showcases the execution time of the matrix-vector multiplication operation on the parallel program. Table 5 on the other hand presents the time it took to transmit the local data from process 0 to all the other processes. These two tables are provided so that there is a discussion on the execution times versus the data sharing time in the following Section.

Size	MPI Processes Number				
	1	4	16	64	128
8192	0.979	3.690	12.956	32.913	3.184
16 384	0.972	3.666	13.964	41.319	8.636
32 768	0.953	3.048	13.913	39.097	16.194

Table 2. Average speedup results of the parallel program.

Size	MPI Processes Number				
	1	4	16	64	128
8192	0.979	0.923	0.810	0.514	0.025
16 384	0.972	0.916	0.873	0.646	0.067
32 768	0.953	0.762	0.870	0.611	0.127

Table 3. Average efficiency results of the parallel program.

Size	MPI Processes Number				
	1	4	16	64	128
8192	0.195	0.052	0.015	0.006	0.060
16 384	0.789	0.209	0.055	0.019	0.089
32 768	3.215	1.005	0.220	0.078	0.189

Table 4. Average execution times of the parallel program, measured in seconds.

Discussions

The parallel program exhibits speedup for all the different numbers of processes used, as it is made clear from the results in Table 2. Of course this does not include the execution for one process; in this case only an MPI process with rank 0 was created, which presented some overhead, resulting in slightly worse execution time than the purely sequential program. For a given problem size the speedup remains relatively stable. The maximum speedup is obtained with 64 processes. This might seem counter-intuitive, because more processes would mean larger partition of the program and more workers to execute the final result. But one has to bear in mind the limitations of the system. The DI Linux Lab has 30 PCs setup as MPI machines that can run up to 4 processes each. This is the optimal configuration, since every PC has a 4-core CPU. This means that the largest number of processes that would run alone on every core of every PC would be $30 \cdot 4 = 120$. Every time the parallel program ran for 128 processes it meant that there were 8 processes more than what can run solely on a single core. As a consequence some PCs ran more than four processes, which resulted in some processes sharing the same core taking turns to execute in their allocated timeslice in a round-robbin fashion. It is only reasonable that those machines took longer to execute all those processes, creating a bottleneck to the whole execution. Therefore the results for 128 processes present much worse speedup than the results for 64 processes, even if there are more workers.

The efficiency of a parallel program is regarded optimal when its value is close to unity. Programs with efficiency equal to 1 are programs that present linear speedup. The current program does not present linear speedups, since the efficiency is always smaller than 1, with the only exception being the single-process execution, which is expected to have efficiency close to 1 anyway. More specifically, the efficiency drops as the number of MPI processes increase. This means that the program is not scalable; the speedup increases when one uses more MPI processes, but it does not increase proportionally to the number of processes. A possible reason is that as the number of processes increase so does the intercommunication overhead. The

Size	MPI Processes Number				
	1	4	16	64	128
8192	0.372	0.311	3.935	4.844	7.897
16 384	1.486	1.240	15.733	19.367	27.465
32 768	13.282	10.742	62.933	77.486	105.412

Table 5. Average data share times of the parallel program, measured in seconds.

part of the timed code block that involves the most communications between processes is the final addition reduction operation of the local sub-vectors to compute the final value of the result vector in process 0. Even if there are more processes to compute the partial sub-matrix with sub-vector local multiplication the reduction operation is a part were global communications slow the execution more than the local operations speed it up. In any case, the efficiency stays relatively steady regardless of the problem size, exactly like the speedup does. One could possibly comment that the problem scales as its size increases, but not by increasing the number of processes.

Tables 4 and 5 provide timing results of the parallel program. Table 4 provides the execution time, which is not very informative by its own; the conclusions on the speed of the program have already been drawn in the previous paragraphs where the speedup and the efficiency were discussed. However, it is important to compare the results of the two timing Tables. The results in Table 4 show that the execution time increases as the problem size increases. But as the number of processes increases, for a fixed problem size, the execution time decreases. That is the reason why speedup is larger than 1 and increasing with the number of processes (at least up to 64 processes). On the other hand, the data sharing time not only rises dramatically with the problem size but also with the increase in the number of processes. This is because the local data have a bigger size with the increase in problem size. If, additionally, the number of MPI processes increases then the process with rank 0, which initializes the matrix and the vector to be multiplied, has to send partitioned data packages to more processes. In this implementation `MPI_Send` is called iteratively in a `for` loop with as many iterations as the number of all the processes other than the sender process with rank 0. `MPI_Send` is implemented differently with different versions of MPI. In some versions it might be a blocking function, while in others it copies the data to the MPI buffers and waits to send them whenever the network is able. On the other hand, `MPI_Recv` is blocking and there is a possibility that one process that takes too long to receive its data blocks the entire execution. In any case this implementation focused on ensuring that each process only receives the data it needs and no redundant data. It did not focus in an optimal fast data-sharing mechanism. MPI provides other global communication functions that would probably optimize the data sharing, like `MPI_Scatter` or `MPI_Bcast`. But it seems inevitable that these functions would have to send data that some processes would not need. This is also due to the nature of the problem, or more specifically, the partitioning of the matrix by columns instead of rows. The data in C-Style matrices are stored in a row-major order in the system memory, which results in the sub-matrix blocks of these partition to not be continuous data in the memory. Therefore custom data types with offsets were used and this limited the data share options to `MPI_Send` and `MPI_Recv` which are not the most optimized solution.

Acronyms

API	Application Programming Interface.
CPU	Central Processing Unit.
MPI	Message-Passing Interface.
PC	Personal Computer.

References

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

[2] Vasileios Karakostas. *Assignment 3 (Mandatory) – Programming with MPI*.

[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.