FOUNDATION OF HIGH PERFORMANCE COMPUTING

FINAL PROJECT

REPORT OF THE PROJECT

Gianmarco Sarnelli

Student ID: SM3500549

Data Science and Scientific Computing

University of Trieste

May 2024

Contents

1 Exercise 1: Game of Life

1.1 Introduction

In this exercise we were asked to implement Conway's game of life using a hybrid MPI+OpenMP code. The general rules are that:

- Each playground is a grid of size $k \times k$;
- The state of each cell of the grid can be either alive (value 1) or dead (value 0);
- The evolution of each cell depends on it's current state and the number of its alive neighbouring cells:
  - If the cell is alive and has 2 or 3 alive neighbours, it stays alive;
  - If the cell is dead and has 3 alive neighbours, then the cell becomes alive;
  - In all the other cases, the cell becomes/stays dead;
- The playground must be evolved for n steps and an image of the system must be saved every s steps;

We have two methods to evolve the playground

- Ordered: evolve one cell after another in row major order. In this case, the status of each cell depends on the previous cells, so this operation $seems$ not parallelizable.
- Static: evolve each cell at the same time. This is the classic version of the game and is easily parallelizable.

For both methods, we are asked to study three kinds of scalability:

- OpenMP strong scalability: with one MPI process per socket, we are asked to increase the number of OMP threads from 1 up to the maximum number of cores in the socket (12 for THIN, 64 for EPYC); in this case the size of the matrix remains fixed for the whole evolution.
- MPI Strong scalability: keeping the matrix size fixed, we are asked to increase the number of MPI processes from 1 up to the maximum number of sockets available. Both cases of strong scalability should follow Amdahl's law of scalability.
- MPI Weak scalability: Here we increase the number of MPI processes while keeping the workload per process fixed. To achieve this, the number of cells in the grid must be proportional to the number of MPI processes N. Since the size of the grid is $k \cdot k$, than k must be proportional to $\sqrt{N}$. This weak scalability should instead follow Gustafson's law of scaling.

The final program should be able to accept some options and to:

- Create the initial conditions. Meaning that it can create a random playground of size $k \times k$ and saving it as a pgm file.
- Evolve the playground taking a pgm file as the initial condition.

Those options should be passed using inline commands when running the executable file:

- `-i` or `-r`: initialise a new playground or run an existing one.
- `-k`: the size of the playground.
- `-f <filename>`: the name of the file for the initialization/evolution.
- `-e <number>`: if it is followed by 0, the program will use the ordered method for the evolution, while if it is followed by 1 the static evolution will be used.
- `-n <number>`: the number of steps of the evolution.
- `-s <number>`: it specifies every how many steps a snapshot will be taken. If this value is 0 than a snapshot should be taken only at the end of the evolution.

1.2 Code implementation

1.2.1 General implementation

The two implementations of Game of Life differ substantially between them, but there are some elements of their implementation that can be valid for both of them.
In both cases the evolution function is called by a main function (*GoL_parallel_main.c*) that reads the initial file, starts the MPI communication, and distributes the playground among the processes using the *MPI_Scatterv* function. The main function then starts the timing and calls either *static_evolution* or *ordered_evolution*.
Both functions implement OMP and MPI parallelization and in particular in both cases the MPI processes share information by sending the first and last rows to other processes (using *ghost rows*). The functions implement some common features to increase their efficiency:

- The use of *char* instead of integer type for the cells to save space. This was done while also using bitwise operations inside each *char* to reduce the need to use more memory space. In particular in *static_evolution* this allowed to use a single grid for the current and the next generation of the playground.
- The use of pre-computed variables inside cycles to reduce the total amount of computation. For example the use of auxiliary variables *left_move , right_move*

to find the neighbours of a cell. Adding this variables to the current position also allows us to avoid using *if* statements in the border cells.

- Reducing the number of *if* statements. This is useful to reduce branch predictions and to increase the speed of the evolution. Most of the times *if* statements are substituted by boolean logic. In particular, no *if* statements are used in the *static_evolution* (except for the printing the snapshots).

1.2.2 Static evolution

The central idea of the static evolution is to use a single grid to store both the current and next generation of the playground. This is achieved using using the variables *current* and *next* that have a single 1 in the first/second position that alternates during the evolution.

```
current = gen % 2 + 1;
next = 2 - gen % 2;
```

Each element of the grid is then update as follows:

```
left_move = -1 + (xsize * (x == 0));
right_move = +1 - (xsize * (x == xsize-1));
pos = y*xsize + x;   //Current position
nei = 0;

nei += my_grid[pos + up_move + left_move] & current;
nei += my_grid[pos + up_move] & current;
nei += my_grid[pos + up_move + right_move] & current;
nei += my_grid[pos + left_move] & current;
nei += my_grid[pos + right_move] & current;
nei += my_grid[pos + down_move + left_move] & current;
nei += my_grid[pos + down_move] & current;
nei += my_grid[pos + down_move + right_move] & current;
// rescaling the value of nei
nei >>= (current -1);

my_current = current & my_grid[pos];
my_grid[pos] = my_current +
next*( (!(my_current)&&(nei==3)) || (my_current&&(nei==2||nei==3)) );
```

This function computes the number of neighbours *nei* by reading the bit that stores the current state of the cell and adding it to the variable. The value of *nei* is then shifted to be in stored starting from the first bit.
The function then computes the next value of the cell using boolean logic and adds it in the position of the next generation.

To efficiently parallelize the program using MPI the idea was to work on the first/last row first and then immediately sending them as they complete to minimize communication overhead. The OMP parallelization happens differently in the border and central rows. The first and last row are parallelized using a OMP for loop that divides the row in chunk of size 128. This number (called *stride*) is twice the length of the cache line, this is done to avoid working on the same cache line and avoid false sharing.
The parallelization on the central rows is done by a OMP for loop on the rows, such that each thread receives at least three different rows (to avoid working on the same cache lines).
The static evolution can be divided in this steps:

1. Sending/receiving all the ghost rows to start the communication.
2. Starting the for loop of the evolution.
3. Waiting for the first row to be sent and for the top ghost row to arrive.
4. Updating the first row.
5. Using non blocking communications to send the first row and to receive the next top ghost row.
6. Waiting for the last row to be sent and the for the bottom ghost row to arrive.
7. Updating the last row.
8. Using non blocking communications to send the last row and to receive the next bottom ghost row
9. Updating the central rows.
10. Ending the for loop, printing the snapshots and ending the communications.

1.2.3 Ordered evolution

In my implementation of the ordered evolution I managed to efficiently parallelize the work using OMP threads, while the parallelization over MPI is just added as an exercise and doesn't give a speedup. To explain the idea of this algorithm I need first to explain how the ordered evolution works in details and why it's thought to be non-parallelizable.

If we only consider the cells that are not in the border, then the evolution of the cell depends on the evolution of the three upper cells and the cell on the left. Since we cannot determine in advance how these cells will evolve we *cannot* evolve the current cell and so we cannot distribute the work for the parallelization.

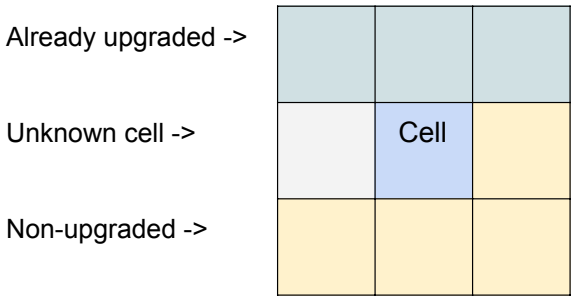


Figure 1: Evolution order of a central cell

But we could consider a simpler case: the one where we are working on a row and the row above is already evolved. This is simpler because we only have the cell on the left that is still unknown, as in Fig 1.
In this case there are some configurations of the 9 cells that allow to evolve the cell in the middle even if the previous cell is still not evolved. The simpler case is when we know that the number of alive neighbours (nei) is 0, because after the evolution of the previous cell we could get either nei $= 0$ or nei $= 1$.
Another possible configuration is obtained when the the cell is alive, the previous cell is dead, and nei = 2 (we count the number of alive neighbour in total, including the previous cell). In this configuration it's guaranteed that the cell in the middle will remain alive, because nei can only increase or stay the same, so we would get nei $= 2$ or nei $= 3$, and in both cases the cell stays alive. All these configuration are summarized in the table 2.
This set of cells will be called **line-independent points**, because their evolution doesn't depend on the evolution of the other cells in the same line. This means that the evolution could start from one of these points and continue from there without knowing the result on the previous elements of the line. If we find all this points in a row (possibly separated by a stride of 128) we can distribute the work on a single line between different OMP threads by making them work on fragments of the row separated by line independent points. This is the basic idea of the OMP parallalelization.

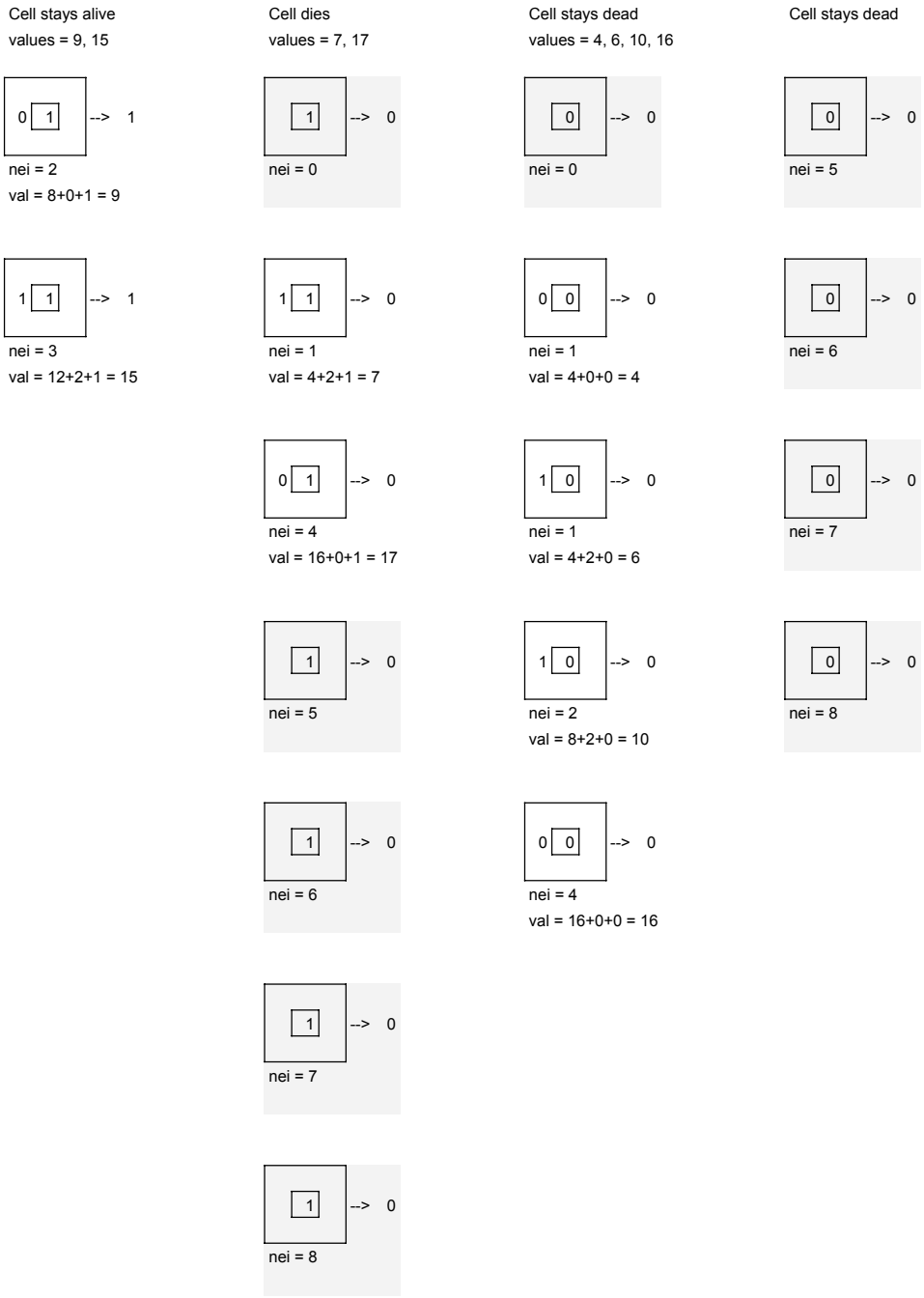


Figure 2: Table of the configurations of the line-independent cells

But there's a problem. To find these line-independent points we need to work serially and to find the value of all the neighbouring cells, and this takes a lot of time and this could ruin the speedup gained by the parallelization.
To solve this problem I completely modified how the grid works. All the important values of the cell (its state, the value *prev* of the previous cell, and *nei*) are stored in different bits of the *char*, this allows to find if a cell is a line-independent point by just looking at the value of the *char*, greatly decreasing the serial time.
This also modifies substantially the evolution of the cells, because instead of having to look at the value of the neighbouring cells, we need to already have the important information inside the cell. To do this, all the cells will modify the value of nei and prev in the neighbouring cells as soon as their state evolves. For example, if a cell is born it will increase by 1 the value of nei in the near cell and increase by 1 the value of prev in the next cell. The single cell will have a structure like in Figure 3.

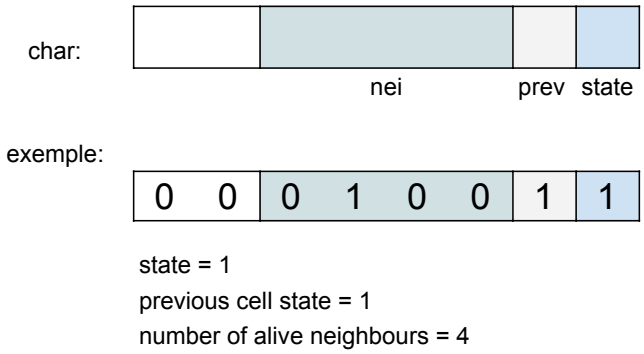


Figure 3: Structure of the byte of each cell

This structure allows us to use a simple line of code to find out if a cell is line-independent. As we can see in Table 2, all the line-independent points have a specific value *val* of the char associated to them:

- All the cells highlighted in grey have either $\text{nei} = 0$ or $\text{nei} \geq 5$. This means that the value of the char is either $\text{val} < 4$ or $\text{val} > 19$ (since $\text{val} = 4 \cdot \text{nei} + 2 \cdot \text{nei} + \text{state}$).
- All the other cells must have value: $4, 6, 7, 9, 10, 15, 16$ or 17 as in Table 2.

All of this can be expressed in the following code for the function *l_ind*. It returns count (the number of l_ind points) and creates the array *l_ind_pos* and *l_ind_dist* that store the position of the points and the size of the fragments between them.

```
char val, check;
int i = stride -1; // starting position
int dist = stride; // starting distance
int count = 1;
l_ind_pos[0] = 0; // makes sure that the first l_ind cell is the first one
while (i<xsize){
        val = my_grid[(y+1)*xsize + i];
        // Checking if the cell is a l_ind point
        check = (val<4) || (val>19) || (val==9) || (val==15) || (val==7) ||
        (val==17) || (val==4) || (val==6) || (val==10) || (val==16);
        if (check){
                l_ind_pos[count] = i;
                l_ind_dist[count-1] = dist;
                i+=stride;
                dist = stride;
                count++;
        }else{
                i++;
                dist++;
        }
}
// The final element of l_ind_dist will be the dist from the last l_ind cell
// to the end of the row + 1
l_ind_dist[count-1] = xsize - l_ind_pos[count-1];
return count;
```

The whole algorithm assumes that it's working on a grid structured like in Fig 3, so the first thing to do is to initialize the grid by filling the cells with values of nei and prev. Then the MPI communication can start and the evolution begins.
The evolution of the central lines is done like in the code in appendix A. There are some technical details that make the code work, but the three main steps are:

1. Updating the first element of the fragment (either a line-independent point or the first element of the row).
2. Update the other elements.
3. Update the value of nei in the last element of each fragment.

4. Find the new line-independent points in the following line.

The general structure of ordered evolution with its MPI communications can be divided in this steps:

1. Initialize the grid to the correct values.
2. The last process sends its last row, while all the process after the first send their first row. This allows the evolution to start.
3. Starting the for loop of the evolution.
4. Receiving the top ghost row (blocking).
5. Waiting for the first row to be sent.
6. Receiving the bottom ghost row (non-blocking).
7. Updating the first row without parallelization.
8. Sending the first row (non-blocking).
9. Updating the central rows using OMP parallelization.
10. Waiting for the bottom ghost row to arrive.
11. Waiting for the last row to be sent.
12. Updating the last row without parallelization.
13. Waiting for the first row to be sent.
14. Sending the last row to make the following MPI process start (non-blocking).
15. Ending the for loop, checking for errors (optional), printing the snapshots and ending the communications.

1.2.4 Correctness of the code

The correctness of the parallel version of Game of Life is verified both through a function call at each generation and by checking the printed snapshots. By using the same initial grid and by printing a snapshot at each generation, I was able to control if and where there was an error. This proved really important to modify and correct the code. The files used for the checking can be found in the folder EX1/Snapshots_check in this same repository.

1.3 Results

The scalability of the program is measured with the speedup, that is calculated as

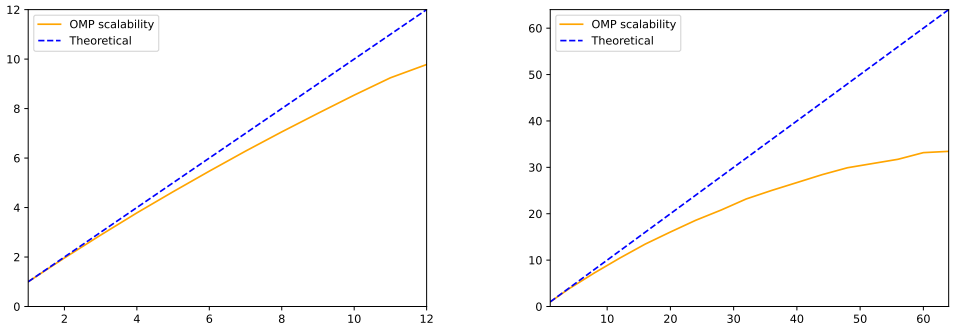
$$S = \frac{T_{serial}}{T_{parallel}} \tag{1}$$

In this section I will show the results of the scalability of the program and confront it with the theoretical speedup. In particular:

- For the OpenMP strong scalability the theoretical speedup grows linearly with the number of cores, but in reality we expect it to follow Amdahl's law and to be limited by the serial part of the code.
- For the MPI Strong scalability the behaviour should be the same, but we don't expect any speedup for the ordered evolution, since it doesn't parallelize on MPI processes.
- For MPI Weak scalability we expect the static evolution to follow Gustafson's law of scaling and be a constant as we increase the number of socket. For the ordered evolution we expect a speedup inversely proportional to the size of the matrix.

All the results shown are obtained by dividing the execution time by the number of generations in the evolution. This is done to minimize the effect of the initialization of the grid as we increase the number of generations.

1.3.1 OpenMP scalability



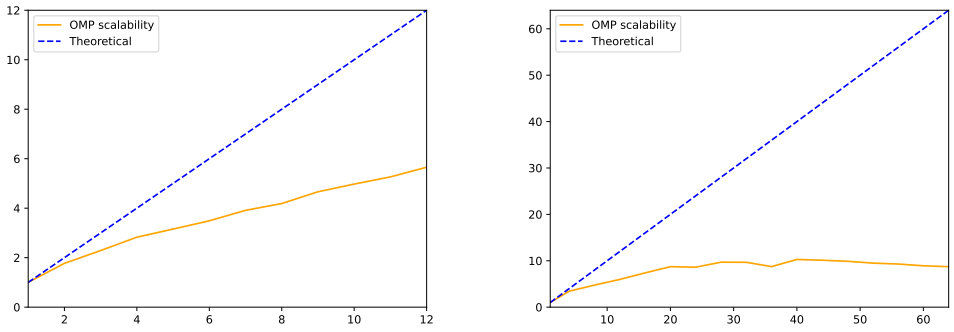
(a) Speedup on THIN node

(b) Speedup on EPYC node

Figure 4: OpenMP strong scalability for the static evolution

As we can see in Fig 4, the OMP scalability behaves as expected for both the THIN and the EPYC nodes and confirm the Amdhal's law. Both graphs show a linear growth that slowly reaches the asymptotic behaviour. Since the EPYC node has more cores, the asymptotic behaviour will be more noticeable.

In Fig 5 we can see that the program achieves a maximum speedup of around 10 at 30 cores and then stops growing. By the Amdal's law this means that the serial portion of the code takes about 10% of the total time. A speedup of 10 times is a great improvement over an algorithms thought to be non parallelizable. This also shows that some improvements are



(a) Speedup on THIN node

(b) Speedup on EPYC node

Figure 5: OpenMP strong scalability for the ordered evolution

still possible and highlights the importance of a fast algorithm for finding line-independent points like we did in 1.2.3.
The biggest limitation of this algorithm is that each process doesn't work on a fixed part of the grid, but instead must work on different parts of it depending on where the line-independent points are found. There could be some ways to fix this but for now this is unavoidable for this kind of algorithm. The efficiency of the algorithm is also greatly influenced by the parameter *stride* that for this implementation was set to 128. Increasing this parameter could reduce the serial part of the program an increase its speedup, but a stride too big wouldn't be able to divide the the rows in enough fragments to be used by the threads. Changing both this parameter and the size of the matrix, I showed that the speedup can still increase (Fig 6) and a new possible improvement could be the search for the best value of stride depending on the architecture and on the size of the grid.
Another way of increasing the speedup would be to remove the function *l_ind* and make it a code segment inside the program. This would remove the function call overhead, but would reduce the code readability.

1.3.2 MPI strong scalability

In Figure 7 we can see that the speedup has a linear trend, but it's clearly better on the THIN nodes. This could mean that the communication between EPYC nodes is not as efficient as the one on the THIN nodes. Another reason for the non-linear trend in the EPYC nodes could be connected to the fact that actually they are faster than THIN nodes on the parallel portion of the code (since I set the maximum number of OMP threads and EPYC has more cores), this would make the serial part of the code more influential and this would lead to a non-linear behaviour. We can also see that the speedup increases if we increase the size of

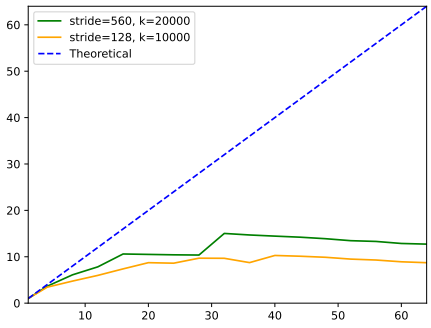
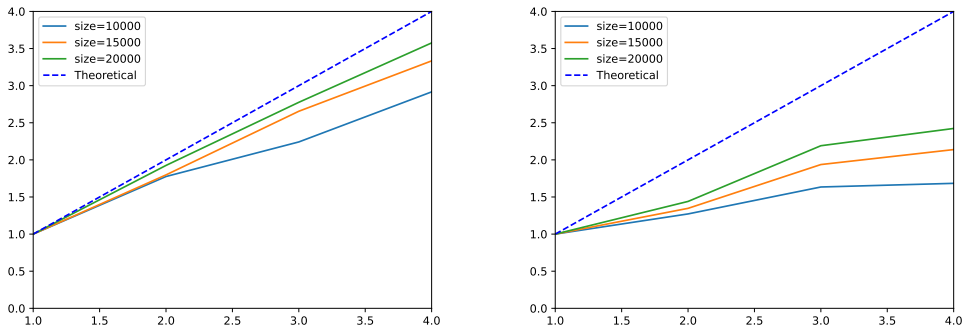


Figure 6: Speedup in ordered with different parameters



(a) Speedup on THIN node

(b) Speedup on EPYC node

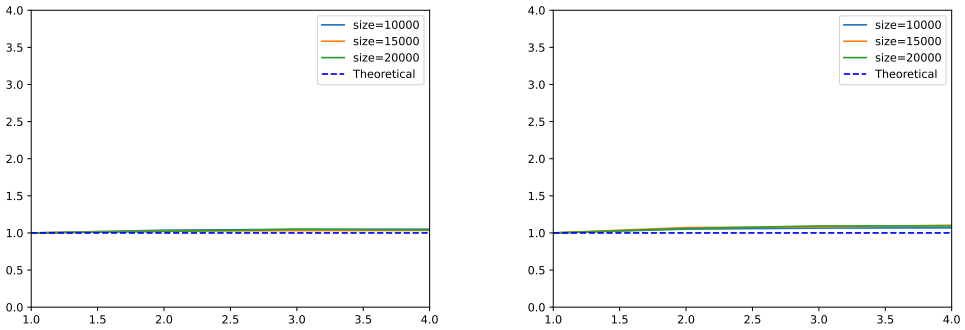
Figure 7: MPI strong scalability for the static evolution

the grid, this is consistent with the fact that the serial part increases linearly with the side of the grid, while the overall time increases quadratically.

In Figure 8 we can see that the ordered evolution has almost constant speedup, but the weird part is that it's slightly increasing as we increase the number of MPI processes. This is against our expectations and could be caused by a less efficient evolution in the case with 1 process. This could be caused by a less efficient implementation of MPI communications when we have only one process. This could make sense since MPI doesn't expect the message to be sent to itself and could have a non-optimal behaviour.

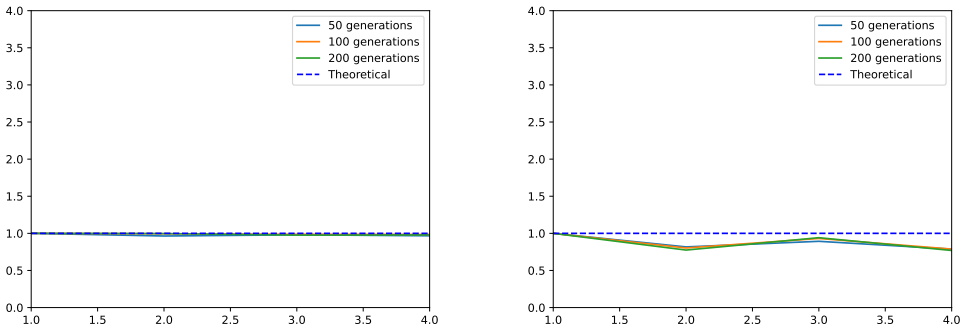
1.3.3 MPI weak scalability

In Fig 9 we can see that the speedup of weak scalabiility is almost constant as expected by Gustafson's law. There is a slightly more chaotic behaviour on the EPYC nodes that could be caused, as said before, by the fact that the serial part of the algorithm is more influential since the parallel part takes less time.



(a) Speedup on THIN node (b) Speedup on EPYC node

Figure 8: MPI strong scalability for the ordered evolution

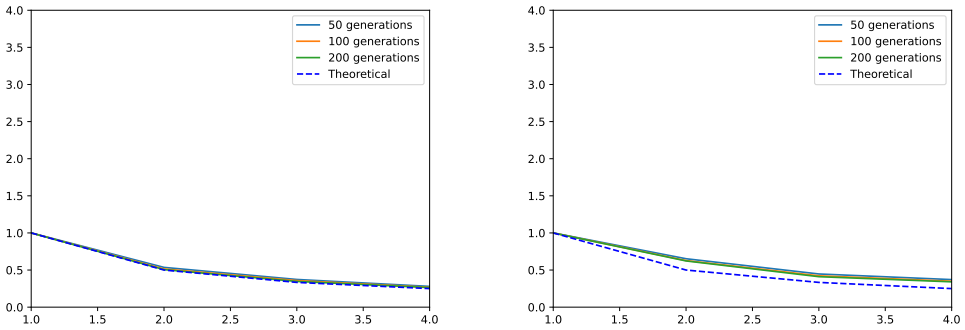


(a) Speedup on THIN node (b) Speedup on EPYC node

Figure 9: MPI weak scalability for the static evolution

For the ordered evolution, as we can see in Fig 10, we note that the speedup is almost inversely proportional to the size of the grid (and the number of MPI processes). This is as expected since the MPI implementation doesn't give any speedup on the ordered evolution. We note still an unexpected behaviour on the EPYC nodes that could be explained as before by the MPI implementation for a single process.

In both the Static and the Ordered evolution I tested the influence of the time to initialize the grid over the total time. I did it by increasing the number of generations and see the effect on the speedup. My idea was that the initialization of the grid in the ordered evolution could take a lot of time and could be technically parallelizable using MPI (since all the process would initialize a smaller grid) and this would make the speedup in Fig 10 greater than the expected one. But since increasing the generations doesn't change the behaviour of the speedup, I concluded that the initialization doesn't influence much the behaviour.



(a) Speedup on THIN node

(b) Speedup on EPYC node

Figure 10: MPI weak scalability for the ordered evolution

2 Exercise 2: Math libraries performances

2.1 Introduction

In this exercise the goal is to compare the performance, measured in Gflops/s, of matrix-matrix multiplications of three math libraries. We measure both the performance when the size of the matrix is increased and when the number of cores is increased. Then we repeat the measurement with different parameters. The three libraries that we are given to compare are

- MKL
- OpenBlas
- Blis

We then vary the measurement by choosing between one of the given options whenever we run the program:

- Library: MKL, OpenBlas or Blis
- Partition: THIN or EPYC
- Precision: Double or Float
- Threads affinity policy: spread or close
- Fixed quantity: cores or matrix dimension

2.2 Code implementation

The folder of the exercise is composed of:

- Folders for each specific instance of the exercise. They will contain the executable files during the execution of the job and will have only the csv file of the results at the end. The folders will iteratively divide into:
  - EPYC or THIN
  - fixed_cores or fixed_size
  - close or spread

  The file results.csv at the end will contain the results for all the libraries at both float or double precision.

- gemm.c: it is the original code that was given to us along with the assignment. I tried to do as few modifications as possible to the files to run the exercise.
  The program accepts command line arguments to specify the dimensions of the two matrices that we want to multiply and prints some information on the code execution, in particular it prints the Gflops/sec of the program.
  It can also be modified by some preprocessor directives that allow the code to:
  - Choose the library to use between **MKL**, **OPENBLAS** and **BLIS**.
  - Choose the desired precision between **USE_DOUBLE** and **USE_FLOAT**.

- Makefile: it is used to compile and run the gemm.c program for all the three libraries. In particular it will create the executable file inside the correct folder so that multiple jobs can be run simultaneously without having any conflicts. It can receive in input the location of the desired folder and the library path for the correct BLIS folder. At the end of the job, make clean will remove all the unwanted files.

- Sbatcher.sh: a bash script that runs the same sbatch file with different options so that I don't have to sbatch each job individually.
  The options are:
  - Name of the job that will be displayed in the queue
  - Partition that will be used
  - Policy used: close or spread
  - Number of threads for each task: –cpus-per-task

- my_job.sh: The job that will run on ORFEO. I made it so that this single bash script can be used for each option needed for the exercise. It runs gemm.c four times and prints the average on the results.csv file.

2.3 Results

2.3.1 Theoretical peak performance

If we have all the specifications of the hardware we are working on, the theoretical peak performances for each core can be calculated as

$$TPP = \text{cores} \times \text{frequency} \times \text{flops per cycle} \tag{2}$$

In our case we can find both the frequency and the number of cores here, but the number of flops per second is not as easily found, so I used references to obtain the Theoretical Peak Performance for both nodes. For the EPYC nodes I used the information reported here, while for the THIN nodes the course slides reported the theoretical peak performance for the whole node and the number of cores per node. At the end the theoretical peak performances that I used are:

- EPYC: 5324.8Gflops$/$s
- THIN: 1997Gflops$/$s

While the performance for double precision floats are half of these.

2.3.2 Size scalability

Here I examine the relation between the Gflops obtained by increasing the matrix dimension, keeping the number of cores constant. The number of cores were fixed to 12 for the THIN nodes and to 64 for the EPYC nodes while the matrix dimensions varied from 2000 to 20000. All the results are obtained by doing an average over 4 run of the same program.

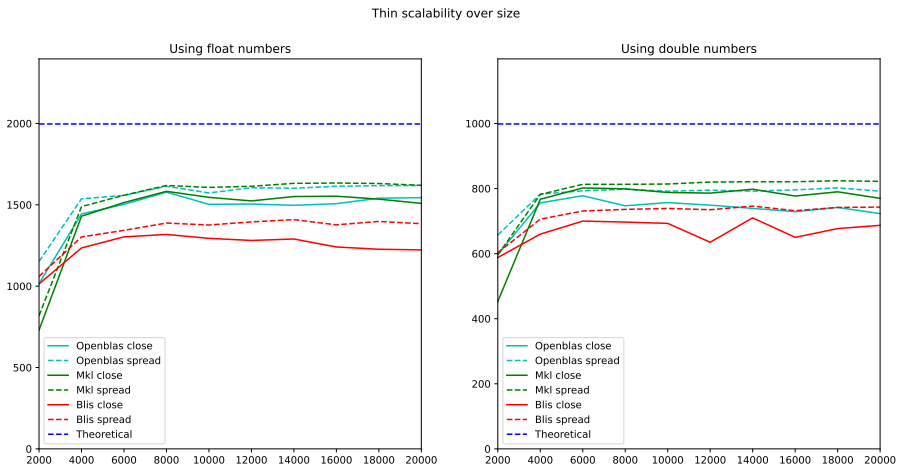


Figure 11: Gfops/s over the matrix size on THIN

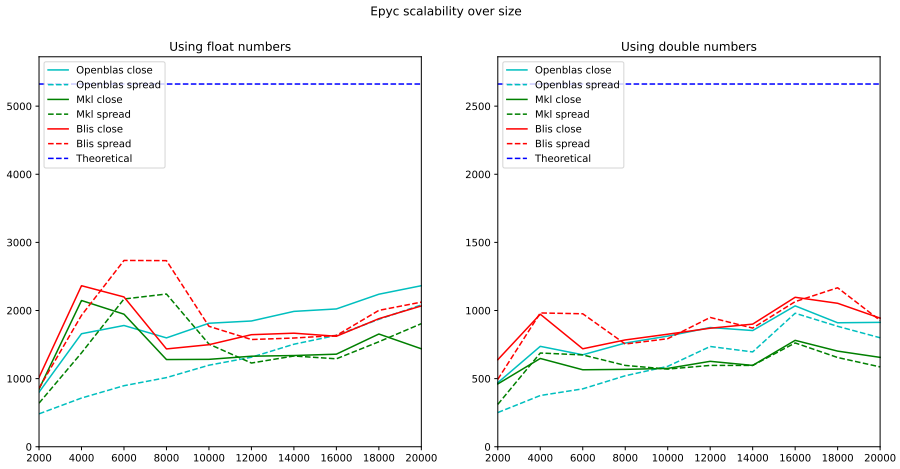


Figure 12: Gfops/s over the matrix size on EPYC

Ideally, the performance should be constant, since the number of cores is fixed, but as we can see from Fig 11 and 12, the measured performance has a growing trend and is much lower than the theoretical one, with the THIN nodes being closer to TPP than the EPYC ones. Generally, all the libraries and thread policy have the same behaviour, with some noticeable disadvantage of Blis on THIN nodes, while the behaviour reverses on EPYC nodes, and Blis is the best here. Generally, Openblas seems a solid library for both nodes, as its performances stay among the best. We can also notice that the graphs of the two nodes present two different trends: when using the THIN nodes, the performance increases until the matrix dimension is more or less 6000, then it stabilises. When using the EPYC nodes instead, the performance increases rapidly for small matrix dimensions,reaching a peak between 4000 and 8000, then it comes back down and starts a slow growth again.

2.3.3 Cores scalability

Here I examine the relation between the Gflops obtained by increasing the number of cores while keeping the size of the matrix constant. The size of the matrix was fixed to 10000 while the number of cores increased from 1 to the maximum per socket. Once again all the results are obtained by doing an average over 4 run of the same program.

While in Fig 13 we can see a clear linear trend (as in the best case for scalability), the trend in Fig 12 starts to grow and then suddenly stabilizes. This could be explained by the fact that EPYC has a lot more cores per socket, and actually until 16 cores the trend seems totally linear, while around 32 cores it stabilizes. Once again Blis performs pretty bad on the THIN nodes while is among the best on the EPYC node. The situation than reverses if we consider MKL. The best among the three must be Openblas, since it performs well on

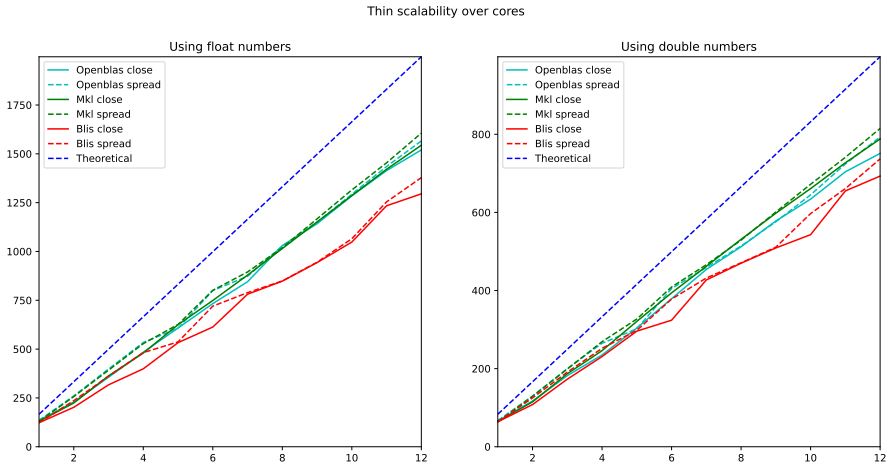


Figure 13: Gfops/s over the number of cores on THIN

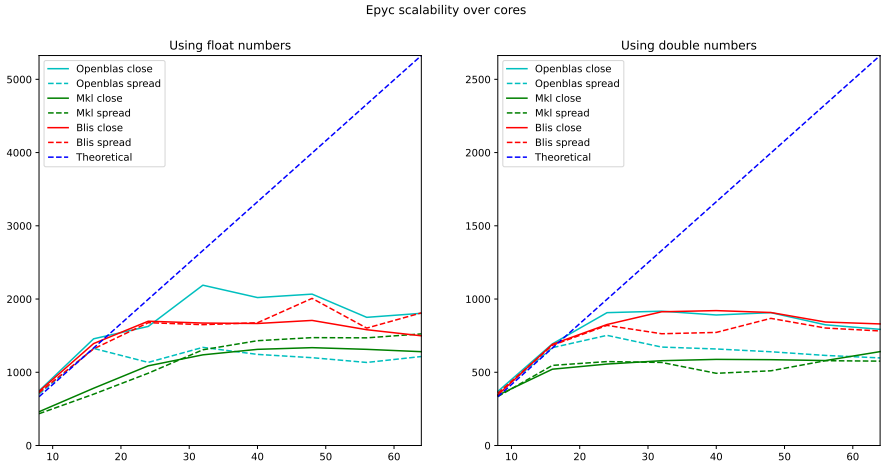


Figure 14: Gfops/s over the number of cores on EPYC

both nodes, and in particular it works best for the *close* policy.

In Fig 12, the performance of Openblas and Blis seem to exceed the theoretical bound. My explanation for this is that the theoretical bound might be harder to compute because it might fluctuate for many reasons.

In conclusion It's important to note that the performances on double precision seem to match perfectly the ones on floating point accuracy, with the latter being its double.

A GoL ordered parallelization

```
// Updating the central lines ( PARALLELIZATION )

for (int y = 1; y < my_chunk-1; y++){
    #pragma omp parallel for schedule( static, 1 ) private(pos, nei, left_move, right_move, prev, my_current, my_new, val, diff)
    for (int i = 0; i<count; i++){
        // Updating the first element

        pos = y*xsize + l_ind_pos[i];
        left_move = -1 + (xsize * ((pos%xsize) == 0));
        right_move = +1 - (xsize * ((pos%xsize) == (xsize-1)));
        val = my_grid[pos]; // Value of the grid in pos
        nei = val>>2; // The number of neighbour is stored starting from the third bit on the char
        prev = val & 2; // The value of the previous cell is stored in the second bit. This is prev*2
        if (i != 0){
            // Here the first element is a l_ind point
            my_current = val & 1;
            // if the first element is a l_ind point, it will become 1 only if its value is either 9 or 15 (see image)
            my_new = ((val==9) || (val==15)) ? 1 : 0;
        }else{
            // Here the first element is the first cell of the row
            my_current = val & 1;
            my_new = (!(my_current) && (nei == 3))  ||  (my_current && (nei == 2 || nei == 3));
        }
        my_grid[pos] = (nei*4) + prev + my_new; // Updating the cell
        diff = my_new - my_current;
        // Updating the value of prev in the next cell
        my_grid[pos + 1] += diff*2;
        // Updating the value of nei in the near cells
        diff *= 4;
        my_grid[pos + up_move + left_move]    += diff;  // diff now stores 4*(my_new - my_current)
        my_grid[pos + up_move]                += diff;
        my_grid[pos + up_move + right_move]   += diff;
        my_grid[pos + left_move]              += diff * (i==0);     // We pass backward this information only if this is the first
                                                                    // element of the row. Sending the this information backward
                                                                    // would create some irregularities, so the right way is to
                                                                    // modify the last element of each fragment at the end of the fragments
        my_grid[pos + right_move]             += diff;
        my_grid[pos + down_move + left_move]  += diff;
        my_grid[pos + down_move]              += diff;
        my_grid[pos + down_move + right_move] += diff;

        for(int j = 1; j<l_ind_dist[i]; j++){
            pos++;
            nei = 0;
            left_move = -1 + (xsize * ((pos%xsize) == 0));
            right_move = +1 - (xsize * ((pos%xsize) == (xsize-1)));
            val = my_grid[pos];
            my_current = val & 1;
            nei = val>>2;
            prev = val & 2; // This is prev * 2
            my_new = (!(my_current) && (nei == 3))  ||  (my_current && (nei == 2 || nei == 3));
            my_grid[pos] = (nei*4) + prev + my_new; // Updating the cell
            diff = my_new - my_current;
            // Updating the value of prev in the next cell
            my_grid[pos + 1] += diff*2;
            // Updating the value of nei in the near cells
            diff *=4;
            my_grid[pos + up_move + left_move]    += diff;
            my_grid[pos + up_move]                += diff;
            my_grid[pos + up_move + right_move]   += diff;
            my_grid[pos + left_move]              += diff;
            my_grid[pos + right_move]             += diff;
            my_grid[pos + down_move + left_move]  += diff;
            my_grid[pos + down_move]              += diff;
            my_grid[pos + down_move + right_move] += diff;
        }// End of work on the fragment
    }// End of the omp parallel
```

```
    // Modifying the value of nei in the last element of each fragment

    for (int i = 0; i<count; i++){
        pos = y*xsize + l_ind_pos[i] + l_ind_dist[i] - 1;
        left_move = -1 + (xsize * ((pos%xsize) == 0));
        right_move = +1 - (xsize * ((pos%xsize) == xsize-1));
        nei = 0;
        nei+=my_grid[pos + up_move + left_move] & 1;
        nei+=my_grid[pos + up_move] & 1;
        nei+=my_grid[pos + up_move + right_move] & 1;
        nei+=my_grid[pos + left_move] & 1;
        nei+=my_grid[pos + right_move] & 1;
        nei+=my_grid[pos + down_move + left_move] & 1;
        nei+=my_grid[pos + down_move] & 1;
        nei+=my_grid[pos + down_move + right_move] & 1;
        my_grid[pos] = (nei*4) + (my_grid[pos] & 3); // The first two bits stay the same
    }

    // Creating the next arrays of the line_independent points
    count = l_ind(my_grid, y, xsize, stride, l_ind_pos, l_ind_dist);

}// End of iteration on central line
```