FOUNDATIONS OF HIGH PERFORMANCE COMPUTING ASSIGNEMENT

Alessandro Cesa

September 2023

Contents

Exercise 1

Introduction

The first exercise, defined in the pdf provided, asked to implement the Conway's Game of Life, parallelizing the work using a hybrid OpenMPI/MPI implementation on the Trieste's Area Science Park's cluster ORFEO.

The Game of Life had to be implemented in two different ways:

- Ordered Evolution: cells are evaluated and updated one after the other
- Static Evolution: cells are evaluated and updated all together

It was asked to measure the scalability of the code in three different ways:

- OpenMP scalability: fixing the size of the problem and number of MPI tasks and incrementing the number of OpenMP threads
- Strong MPI scalabilty: fizing the size of the problem and increasing the number of MPI tasks
- Weak MPI scalability: fixing the workload per MPI task, and increasing the number of tasks

Methodology

Game of Life

I implemented the Game of Life's grid as a vector of size $k * k$,with k equal to the number of cells in a row (or column), with each element being 0 for a dead cell and 1 for an alive cell.

In order to initialize a playground, i created a function that initializes a vector assigning random values between 0 and 1.

In order to evaluate if at a certain iteration a cell had to live or to die, i first spotted his neighbors (taking in consideration the conditions at the border), and then, given the fact that an alive neighbor was represented by a 1 and a dead one by a 0, i summed the values of the cells representing the neighbors in order to count the number of neighbors. At this point if the sum is equal to 2 or 3 the cell had to live, otherwise it had to die.

Ordered Evolution

In order to implement the ordered evolution method, the following loop was used:

```
for (int i*i = 0; i < size; i++) {
        update_cell(grid, size, i, j);
}
```

Where i is the position of the cell on the vector.

The evolution method is intrinsecally serial, since the state of a cell depends on the state of the previous one, so no parallel work has been made.

Static Evolution

In the static evolution, the grid has to be evaluated and eventually updated all together so some parallel work is possible.

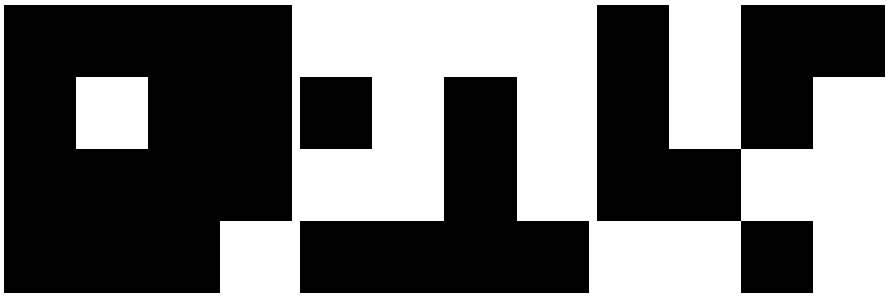
Given a number of MPI tasks, and a number of OpenMPI threads per MPI task, all MPI tasks had to store the whole grid. At this point, for each episode, each MPI task evaluated only a portion of the matrix, storing it into a new vector that was eventually shared to the other tasks which copied it into their complete grid.

It was actually possible to have each MPI task to store only a part pf the grid: the portion that they had to update plus the neighbors of that portion residing in iother task's portions. This would have meant a harder implementation, since it required to compute the neighbors of each portion of the grid as well as more MPI communication, so i opted for the easier but less efficient option described before.

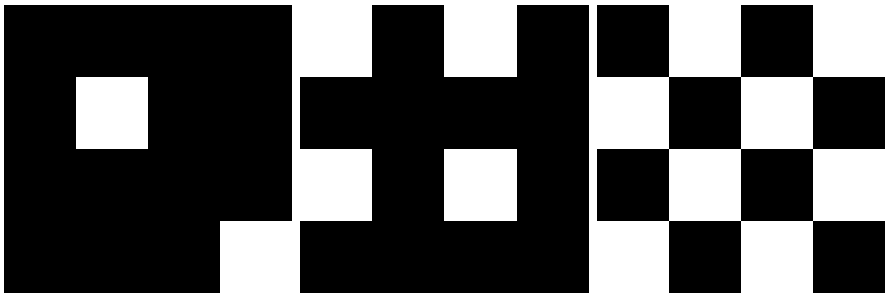
Implementation

Method validation

In order to be sure that my implementation of the Game of Life was correct, i ran a small 4x4 example both manually with pen and paper and with the functions that i wrote, and it turned out the the results were the same.



Initial playground Ordered iteration 1 Ordered iteration 2



Initial playground Static iteration 1 Static iteration 2

Nodes used

For all the exercise, i used the THIN nodes.

THIN nodes have 2 sockets with Intel Xeon Gold 6126 12-Core; 12 Cores per socket and 1 thread per core. They have 768 GB of memory, 24 instances of L1d Cache with 768KiB of memory, 24 instances of L2 Cache with 24MiB of memory and 2 instances of L3 Cache with 38.5MiB of memory.

I have chosen THIN nodes over EPYC nodes since at the time when i've done the exercise they were the less crowded.

There are 10 THIN nodes on the ORFEO cluster, but i haven't used more than 2 of them at the same time, in order to reduce the time waiting

for resources.

In order to use MPI, i have used intelMPI/2021.7.1 .

Timing

In order to measure the running times, i always ran the code for 5 times and i considered the mean.

Parallelization

The static evolution method was parallelized with MPI and OpenMP.

At first, all tasks store the complete grid. When the run_static function is called, MPI is initialized, and the following steps are made:

- Each tasks computes the sizes of each tasks' share of the grid: the grid is equally divided between, giving to each task a contiguous share of the grid; the remainder is distributed equally among the first tasks (if the remainder is equal to n, the first n tasks will have one more cell)
- Each task, based on the sizes, computes at which part of the grid each tasks' share starts
- Each task initializes a new grid, of the size of its share, to store its evaluations

Now, for each episode:

- Each task evaluates the cells of its share of the grid, and stores them in his new grid: this is done using a for loop which is parallelized with OpenMP: using a 'static' policy,each OpenMP thread computes the evaluations for a chunk of cells.
- An MPI_Barrier is set up in order to be sure that each MPI task has completed his job
- Using the function MPI_Allgatherv evaluations are shared between all tasks, wich store them in the original grid, that now will contain the new state of the playground
- If it's required, task 0 prints the playground as a pgm image

At the end of all episodes, MPI is finalized.

Only the task 0 prints the running time.

Mainly, parallelization is implemented through domain composition: the grid is decomposed and each MPI task and each OpenMP thread work on a different share of the grid. However, there is some kind of functional decomposition since only one task is responsible for printing the playgrounds and the running time.

OpenMP scalability

In order to measure OpenMP scalability, it was asked to fix the size of the problem, fix the number of MPI tasks to 1 per socket and increment the number of OpenMP threads from 1 up to the number of cores present in the socket.

I tested two different approaches:

- Using 1 MPI task, so that everything runs on a single socket
- Using 4 MPI tasks across 2 THIN nodes, setting 1 task per socket (2 per node) using the environmental variable I_MPI_PIN_DOMAIN=socket

I increased the number of OpenMP threads per MPI tasks from 1 to 12 (the number of cores present in the socket) using the environmental variable OMP_NUM_THREADS,while the size of the grid was fixed . The threads were distributed using a close policy, implemented with OMP_PROC_BIND=close, meaning that the threads had to run on cores as close as possible.

I compared results using a 10000x10000, a 15000x15000 and a 30000x30000 grid.

The running time is expected to scale down as fast as possible with the increase of threads.

Strong MPI scalability

In order to measure strong MPI scalabilty, it was asked to fix the size of the problem and increase the number of MPI tasks.

I used 2 THIN nodes and, using using the environmental variable I_MPI_PIN_DOMAIN=1, i placed 1 MPI task per core. I fixed the number of OpenMP threads to 1 per MPI task.

I increased the number of MPI tasks from 1 to 48 (the number of cores on the 2 nodes) while the grid size was fized. I used the environmental variable I_MPI_PIN_ORDER=compact. in order to make MPI threads run on sockets as close as possible.

I compared results using a 10000x10000, a 15000x15000 and a 25000x25000 grid.

The running time is expected to scale down as fast as possible with the increase of tasks.

Weak MPI scalability

In order to measure weak MPI scalability, it was asked to fix the workload per MPI task, and increasing the number of tasks up from 1 socket (saturated with OpenMP threads).

So it was necessary to give a measure of the workload of an MPI task. Given the fact that i distributed the cells of the grid as evenly as possible among MPI tasks, i measured the workload as $w = \frac{k^2}{n}$, with k equal to the size of a row or column of the grid (and so k^2 the total number of cells) and n the number of MPI tasks.

I used 2 THIN nodes and, using the environmental variable I_MPI_PIN_DOMAIN=omp and setting 2 tasks per node, i placed 1 MPI task per socket; so i had to increase the number of MPI tasks from 1 to 4. Each MPI task had 12 OpenMP threads

Using the formula given above, i kept the workload constant by using these sizes:

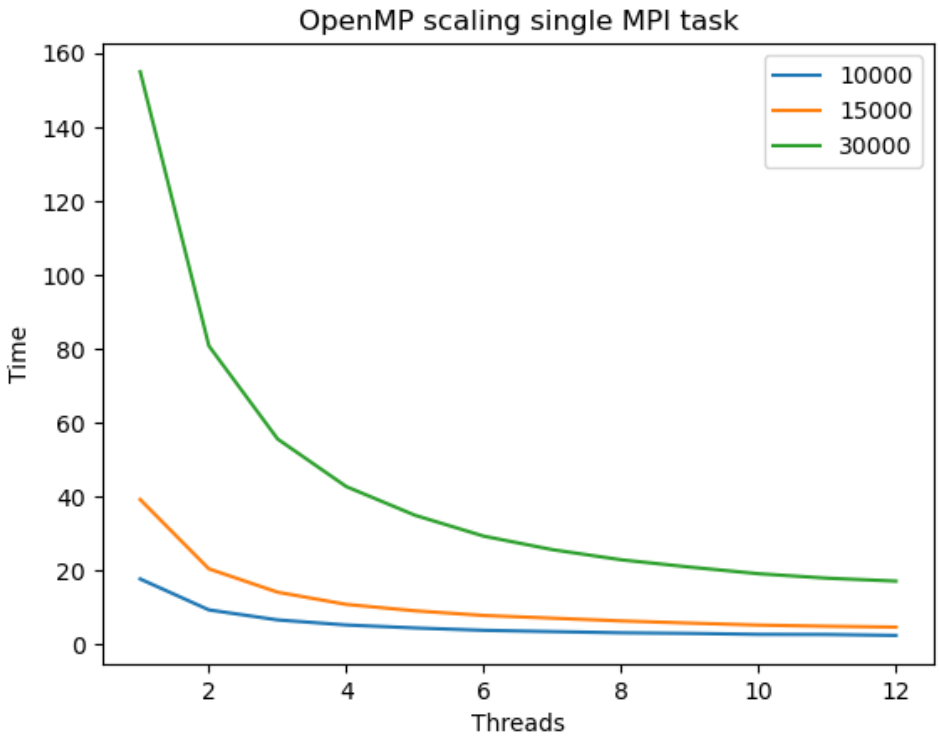
n	k
1	10000
2	14142
3	17321
4	20000

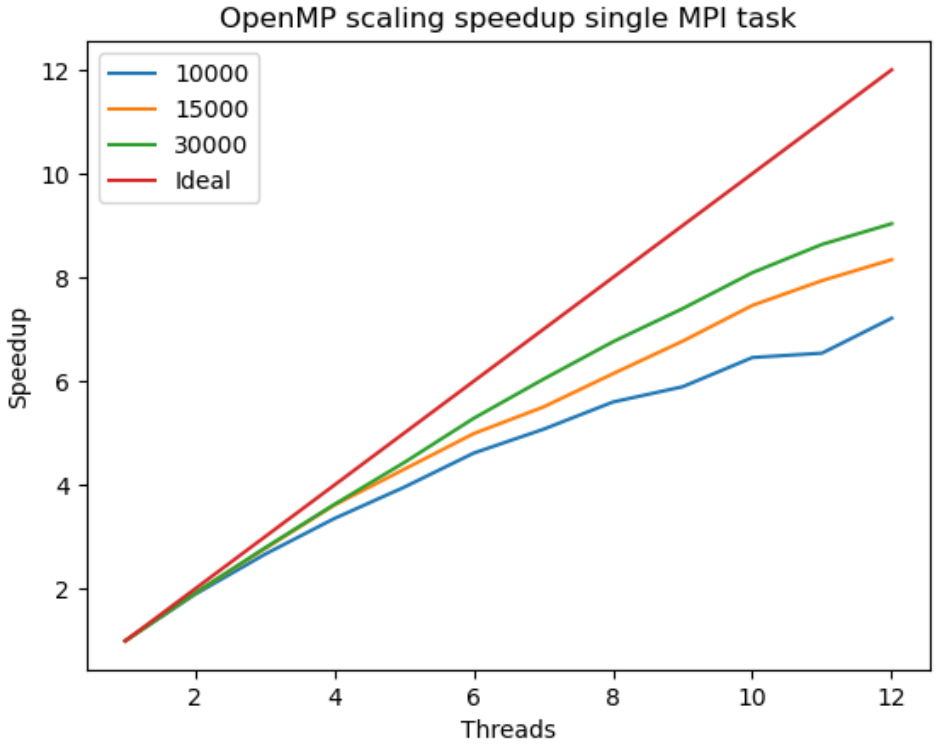
The running time is expected to increase as little as possible

Results & Discussion

OpenMP scalability

Single MPI task



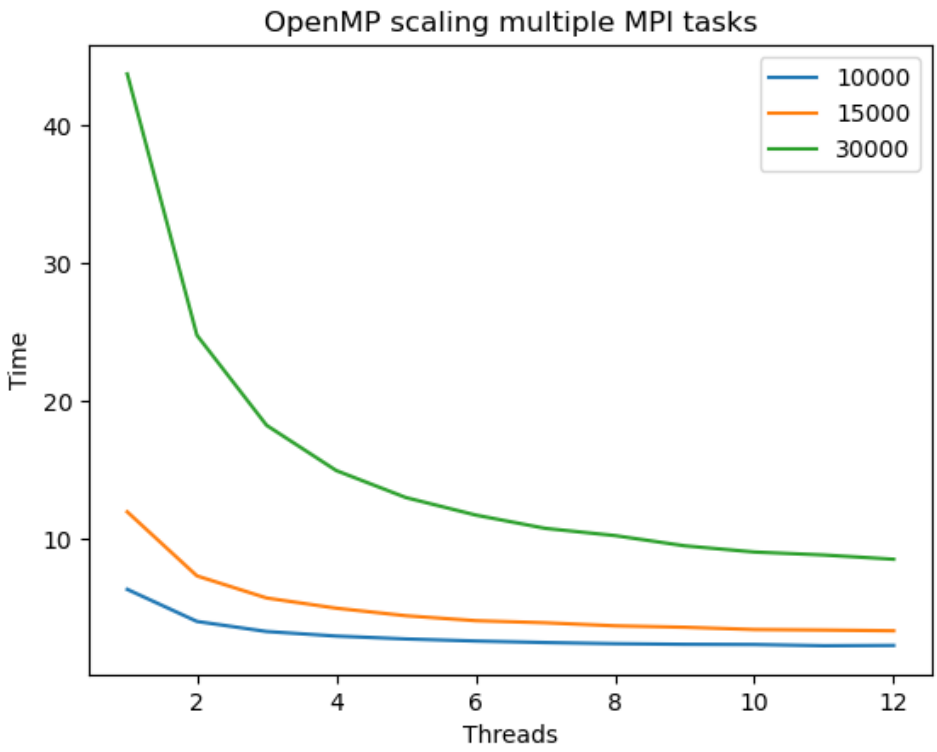


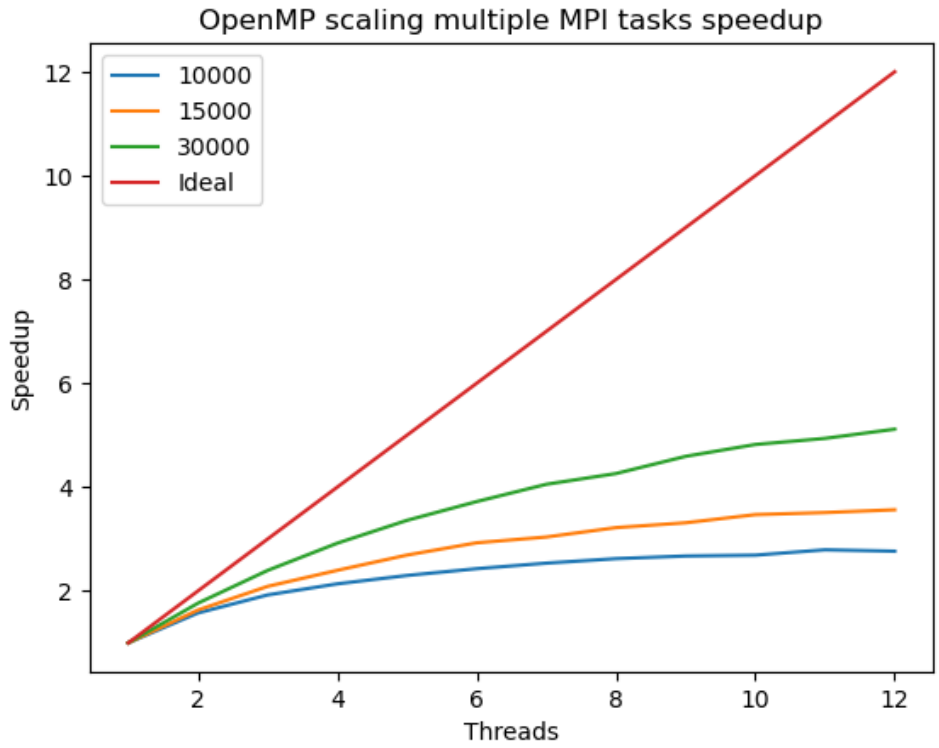
We can see that the running time drops from the serial execution to the 2-threads one, and keeps shrinking as the number of threads grows; however after 6/8 (depending on the size) threads it variation is really low.

Also, the speedup is satisfying at the beginning, but becomes worst with a larger number of threads.

Overall, the scaling improves as the size of the problem grows.

Multiple MPI tasks



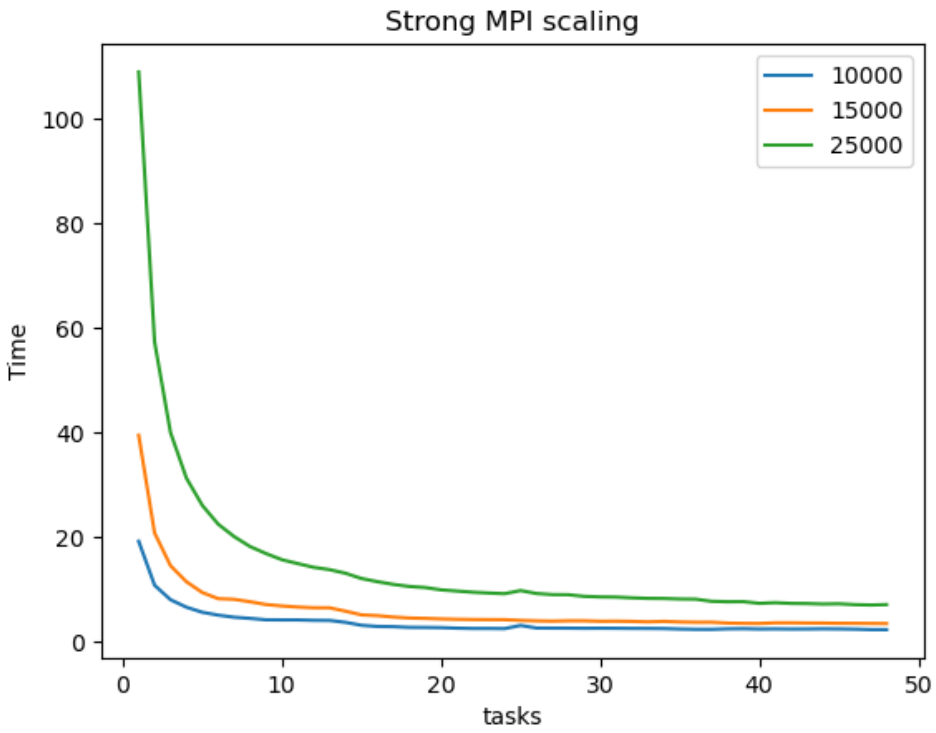


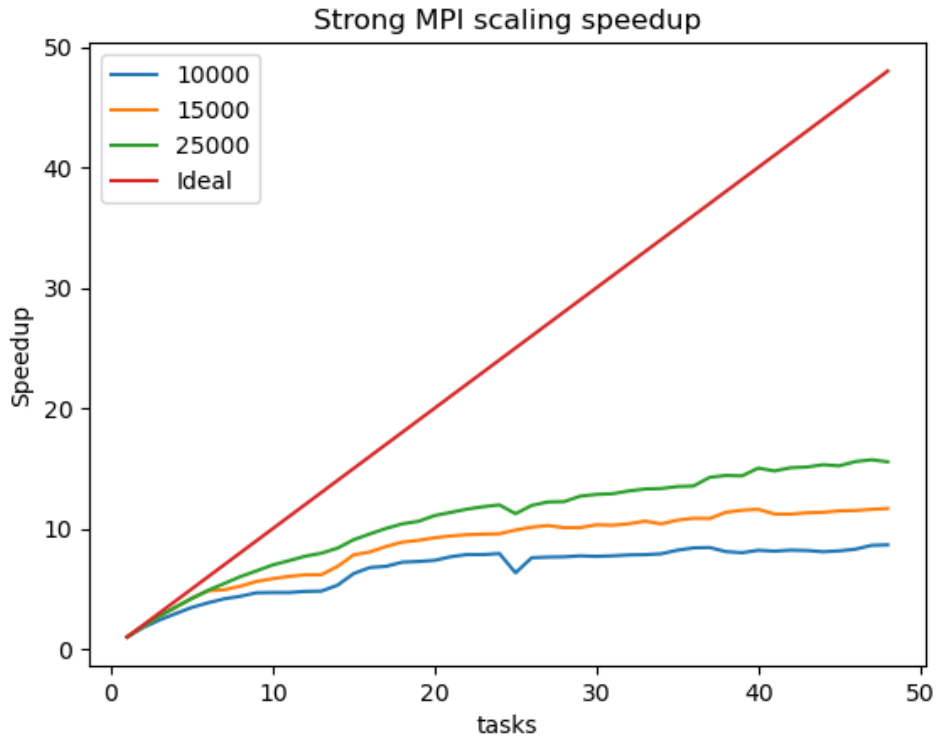
As in the previous case, the running time drops from the serial execution to the 2-threads one, and keeps shrinking as the number of threads grows; however after 6/8 (depending on the size) threads it variation is really low.

However, the speedup is much worse, going far away from the ideal linear speedup already at 3 threads.

Probably in this case the MPI communications between different tasks placed on different sockets give a large overhead and prevents from a better speedup

Strong MPI scalability



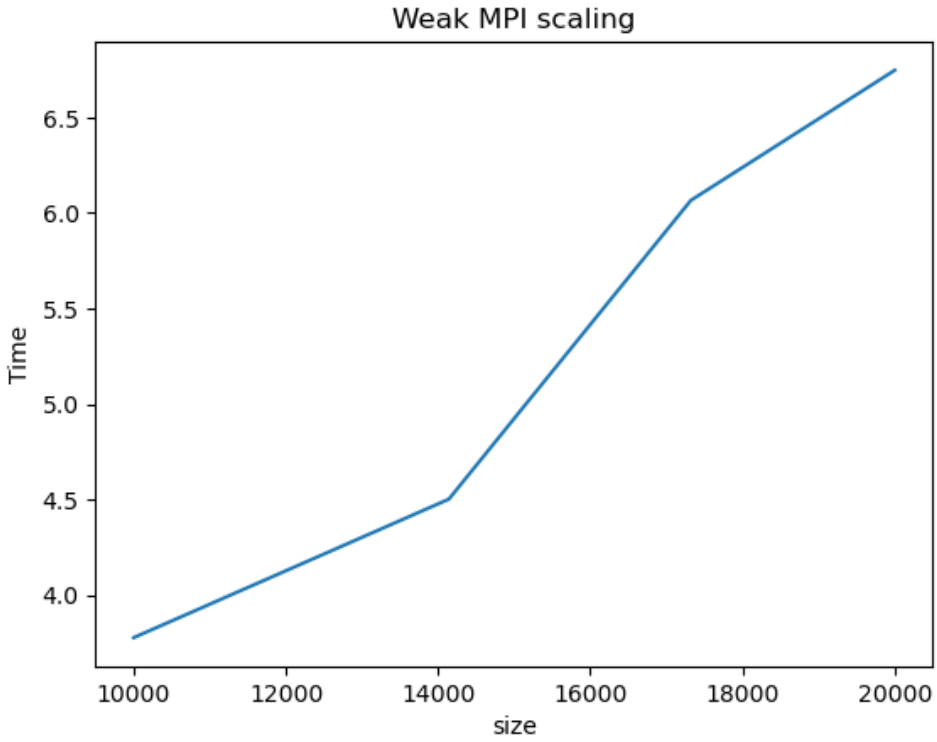


Also here, with the first couple of increases in the number of tasks it scales well, then the speedup stops improving, and so we go far away from the ideal situation.

We can an increase in the running time when we pass to 12 tasks and to 25 taks: these are the points when we go from the first to the second socket and from the first to the second node. It's clear that the communication between different sockets is slower than inside the same socket, and the communication between differen nodes is even worse.

As in the previous case, the scaling improves as the size of the problem grows

Weak MPI scalability



size=size of a row of the matrix

The running time, that should increase as little as possible has a considerable increase. However, if we look at the numbers, on a 300% increase in the problem size, the running time increases by just the 78%.

Conclusions

From the results we can conclude that this parallel implementation of Conway's game of life, despite being still an improvement from a serial one, doesn't scale really well, especially on the MPI side.

A possible explanation for this is the aforementioned fact that each MPI task stores the whole grid, including the parts that he doesn't need, and each tasks sends the updates of his part of the grid to all other tasks, including the one that don't need it. So there are a lot of actually useless MPI messages

that increase a lot with the increasing of tasks: this creates a situation where the time needed for these messages compensate the advantage of the parallelization.

So, a possible improvement would be to make each task store only the part of the grid on which he has to work on, along with all neighboring cells, and send the informations about his updates only to the tasks that actually needs them (the tasks that work on the neighboring cell).

A further improvement would be to parallelize the process of writing the grid to a pgm image.

It would also be interesting to try to implement the project with the the EPYC nodes instead of THIN, and using OpenMPI instead of IntelMPI (given the fact that for the THIN nodes IntelMPI is probably the best choice, since it is highly optimized for Intel architectures).

Also, i haven't parallelized the ordered evolution, because of his serial nature. But one could divide the work among MPI tasks on different nodes in order to handle larger matrices

Exercise 2

Introduction

The second exercise, defined in the github repository of the course, asked to compare the MLK and OpenBLAS math libraries on the level 3 BLAS function called gemm.

This function allocates 2 matrices of a given size and computes their product using a given library.

It was asked to compare running time and GFlops in two different ways:

- Increase the size of the matrix at fixed number of cores
- Increase the number of cores at fixed matrix sie

Both the comparisons have to be made for both single precision and double precision floating point numbers.

Methodology

section*Size scaling

The number of cores was fixed at 12, and the size of the matrices was increased from 2000x2000 to 20000x20000.

I tried 2 different thread allocation policies for the threads: 'close', where the threads are put as close as possible between each other, and 'spread' where the threads are put as distant as possible.

section*Cores scaling

The size of the matrix was fixed at 12000x12000, while the number of cores was increased from 1 to 12, in order to fill up the socket.

I tried 2 different thread allocation policies for the threads: 'close', where the threads are put as close as possible between each other, and 'spread' where the threads are put as distant as possible.

Implementation

In order to run the exercises, i have modified the provided Makefile, in order to make it produce all the needed executables at once; i have also modified te dgemm.c code, in order to make it easier to manage the output.

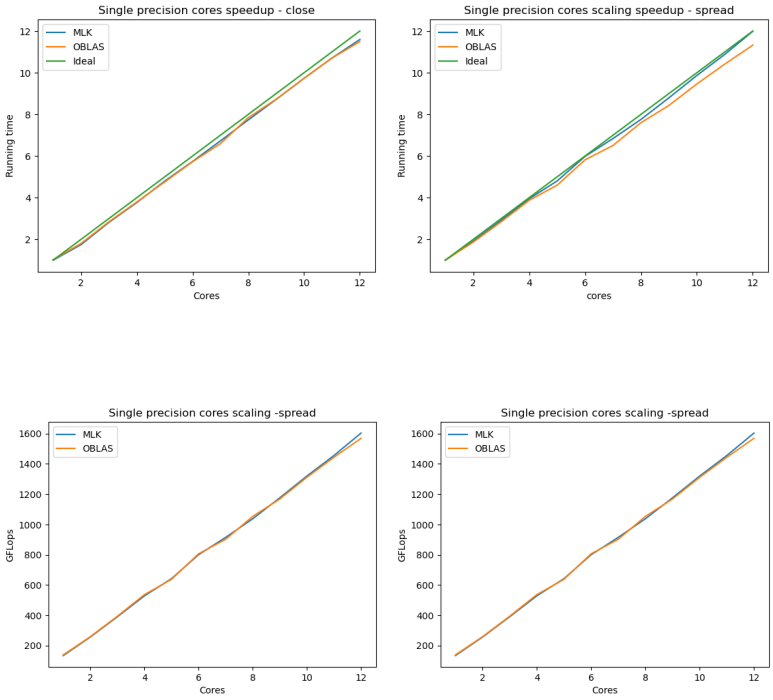
The whole exercise was runned on a THIN node (whose properties are described in Exercise 1),which have a theoretical peak performance of 1,997 TFlops.

In order to measure the running times, i always ran the code for 5 times and i considered the mean.

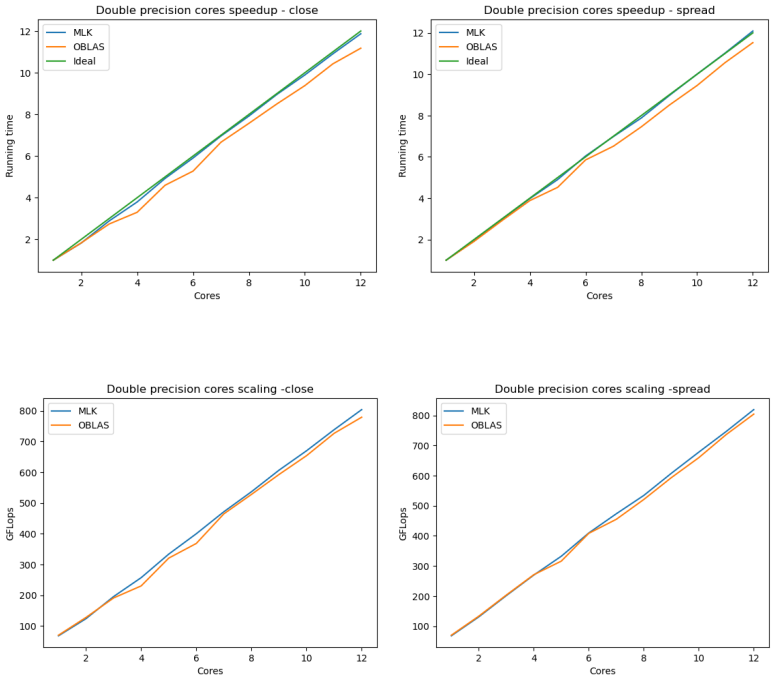
Results & Discussion

Core scaling

Single precision

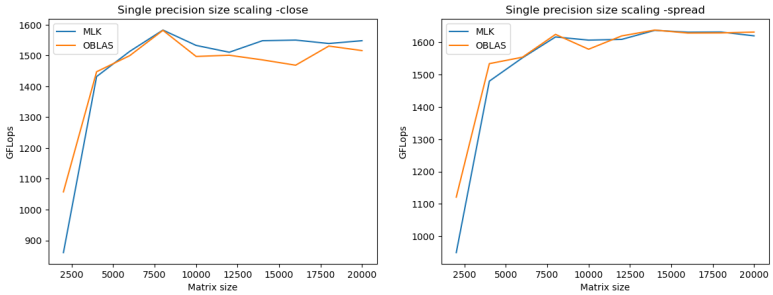


Double precision



Size scaling

Single precision

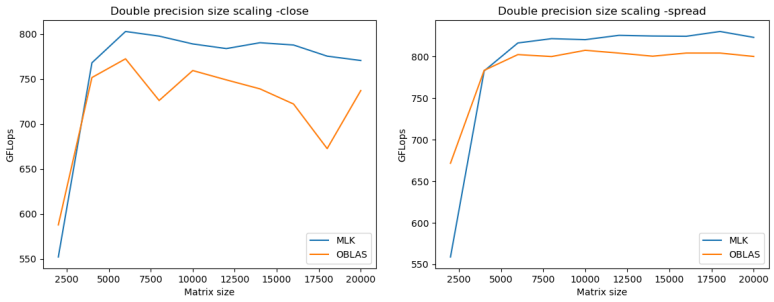


size=size of a row of the matrix

Both libraries scale very well with respect to the number of cores. The speedup is near to the ideal one and the increase in GFlops si almost linear

with the increasing of cores. However we can see how MLK has better performances, situation which is enhanced with the close policy.

Double precision



size=size of a row of the matrix

We can see that in general there is a rapid increase in GFlops until matrices around 3000x3000, after that number GFlops increase at a slower path or they tend to stabilize or even decrease.

As intiuivetely obvious the GFlops hare much higher in single precision than in double precision. OBLAS has better performances on smalle matrices, but when the matrix gets bigger MKL works with moreGFlops. This difference tends to be smaller for single precision and for spread allocation and larger for double precision and spread allocation.

For some reason for matrices of size 18000x18000 , double precision and close thread allocation, OBLAS has a steady drop in performances, but it recovers with larger matrices.

Conclusions

We can conclude that both libraries perform well, but MKL looks better than OBLAS. For both libraries the scaling is better on the number of cores with the matrix fixed than on the size of the matrix with the number of cores fixed. In our measurements, none of the libraries have reached their theoretical peak performanca, the maximum reached is for both around 1600 GFlops.