HPC project report: Conway's Game Of Life

Michele Sanna

March 22, 2023

1 Introduction

In this work we're going to study the scalability and the parallelization design of a program that simulates the popular Conway's game of life. The choices concerning the parallelization design and the programming optimization are discussed in part 2 and 3. The code follows the specifications given and is tested against a variety of cases. In part 4 and 5 is discussed the scalability of this implementation.

2 Methodology

2.1 Static update: domain decomposition

The grid of the game is divided in rectangular subgrids along the y axis. Those grids are as many as the number of processes. Every process receives a subgrid and the two adjacent rows (the one immediately before, and the one immediately after). We're considering the grid as a torus, so the first row is considered adjacent to the last and viceversa. For a process the subgrid is the section of the grid to actually update, while the others two rows are necessary to check the number of alive neighbours for the cells of the first and last rows of the subgrid, but they belong to a section of the grid that will be updated by other processes.

2.2 Static update: message passing

In order to have a consistent update of the grid, the processes need to exchange the two rows not belonging to their subgrid (from now on we will call them "external rows") at every step, indeed they need to have them updated to correctly check the number of alive neighbours for the first and last rows's cells. For the i-th process, the area of the grid represented by the external rows is updated by the processes i-1 and i+1. For the process 0 they are updated by the process n-1 (where n is the number of processes) and process 1. For the process n-1 they are updated by process n-2 and 0.

2.3 Static update: snapshot saving

If we need to save a snapshot or in general to save the game grid into a .pgm file (ase requested by the project rules) we need to merge back the subgrids of the processes into the game grid. In order

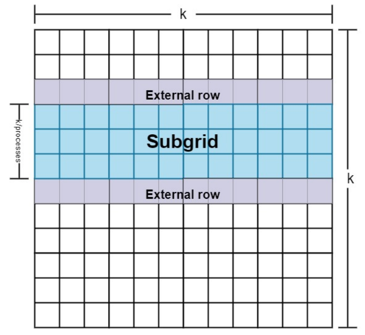


Figure 1: A subgrid and its external rows.

to do this all the processes will send their subgrids to the process 0, then the process 0 will merge all the subgrids into a consistent game grid and save it into a file. Eventually the process 0 will split the game grid and send back the subgrids to the processes in order to continue the computation.

2.4 Dynamic update

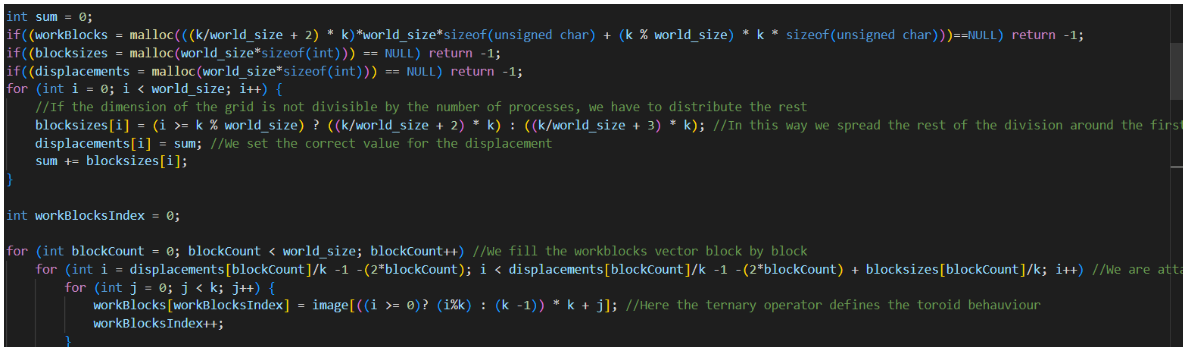
The dynamic update is inherently serial, i didn't find a way to parallelize it that doesn't give different results with respect to the serial version. Parallelizing the check for every single cell (for example: every thread checks the state of one different neighbour cell) doesn't lead to any improvement due to the overhead of the parallelization and the simplicity of the task.

3 Implementation

The program is implemented in C and for the parallelization are used MPI processes, for implementig the multithreading it's used the openMP library.

3.1 Grid and workblock representation

At the moment of the creation of the grid, or when loaded from a file, the grid is represented as a flat char array where every entry represents a cell, in a row major order. In order to being sent to the processes this array is divided according to the subgrids described in the methodology section, and the external rows are actually sent attached to the subgrid, so instead of a subgrid of (#rows/#processes) rows, a workblock of (#rows/#processes + 2) rows is sent. In this way the process will seamleassy check all the cells, even the ones in the margins.



```c
int sum = 0;
if((workBlocks = malloc(((k/world_size + 2) * k)*world_size*sizeof(unsigned char) + (k % world_size) * k * sizeof(unsigned char)))==NULL) return -1;
if((blocksizes = malloc(world_size*sizeof(int))) == NULL) return -1;
if((displacements = malloc(world_size*sizeof(int))) == NULL) return -1;
for (int i = 0; i < world_size; i++) {
    //If the dimension of the grid is not divisible by the number of processes, we have to distribute the rest
    blocksizes[i] = (i >= k % world_size) ? ((k/world_size + 2) * k) : ((k/world_size + 3) * k); //In this way we spread the rest of the division around the first
    displacements[i] = sum; //We set the correct value for the displacement
    sum += blocksizes[i];
}

int workBlocksIndex = 0;

for (int blockCount = 0; blockCount < world_size; blockCount++) //We fill the workblocks vector block by block
    for (int i = displacements[blockCount]/k -1 -(2*blockCount); i < displacements[blockCount]/k -1 -(2*blockCount) + blocksizes[blockCount]/k; i++) //We are atta
        for (int j = 0; j < k; j++) {
            workBlocks[workBlocksIndex] = image[((i >= 0)? (i%k) : (k -1)) * k + j]; //Here the ternary operator defines the toroid behauviour
            workBlocksIndex++;
        }
```

Figure 2: How the workblocks are represented

3.2 Update of the subgrid

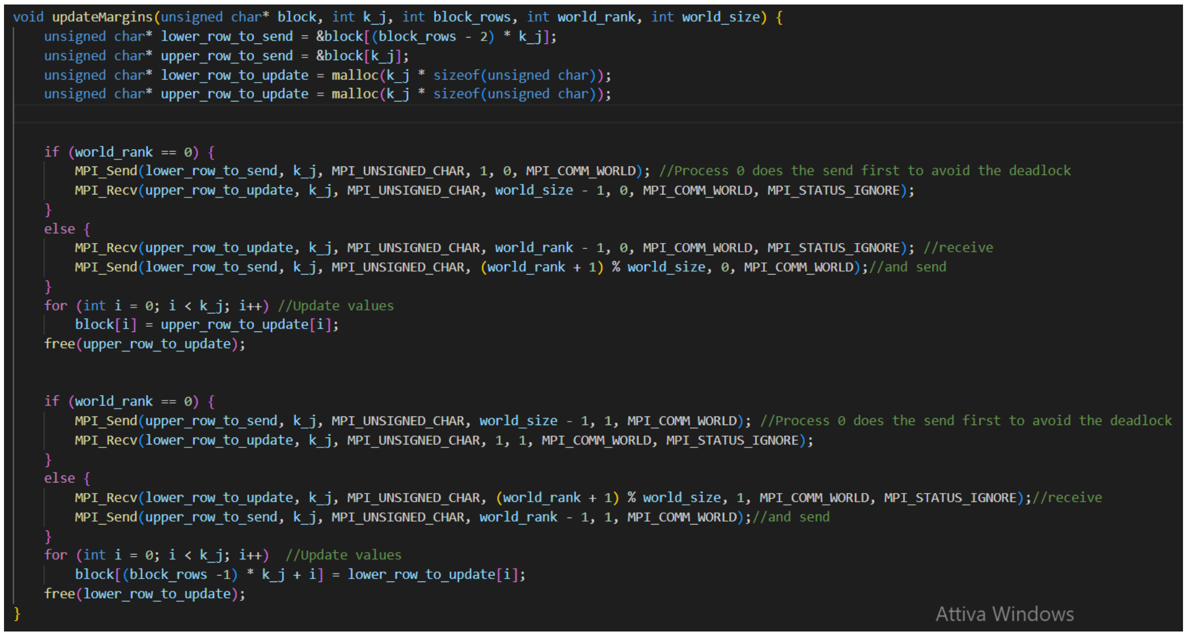
For the update of the subgrid every process spawns subthreads. The implementation is pretty straightforward because the domain decomposition into the for loop is managed by the omp construct. If every thread checks the neighbours and update a different cell with respect to the others, there is no data race going on. Ternary operators are used instead of if constructs in order to optimize (the code, by contrast, is a bit less readable)

3.3 File writing

For writing into the file the workblocks have to be merged back. In order to retrive the game grid from the merged the workblocks efficiently, i used an array of pointers of dimension #rows. Every pointer points of the beginng of a row in the workblocks array, keeping attention to jump the repeated rows (that actually represents the external rows of every block)

3.4 Message passing

The external rows are passed with blocking calls, the order of the calls is designed to avoid deadlocks.



```c
void updateMargins(unsigned char* block, int k_j, int block_rows, int world_rank, int world_size) {
    unsigned char* lower_row_to_send = &block[(block_rows - 2) * k_j];
    unsigned char* upper_row_to_send = &block[k_j];
    unsigned char* lower_row_to_update = malloc(k_j * sizeof(unsigned char));
    unsigned char* upper_row_to_update = malloc(k_j * sizeof(unsigned char));


    if (world_rank == 0) {
        MPI_Send(lower_row_to_send, k_j, MPI_UNSIGNED_CHAR, 1, 0, MPI_COMM_WORLD); //Process 0 does the send first to avoid the deadlock
        MPI_Recv(upper_row_to_update, k_j, MPI_UNSIGNED_CHAR, world_size - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    else {
        MPI_Recv(upper_row_to_update, k_j, MPI_UNSIGNED_CHAR, world_rank - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); //receive
        MPI_Send(lower_row_to_send, k_j, MPI_UNSIGNED_CHAR, (world_rank + 1) % world_size, 0, MPI_COMM_WORLD);//and send
    }
    for (int i = 0; i < k_j; i++) //Update values
        block[i] = upper_row_to_update[i];
    free(upper_row_to_update);


    if (world_rank == 0) {
        MPI_Send(upper_row_to_send, k_j, MPI_UNSIGNED_CHAR, world_size - 1, 1, MPI_COMM_WORLD); //Process 0 does the send first to avoid the deadlock
        MPI_Recv(lower_row_to_update, k_j, MPI_UNSIGNED_CHAR, 1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    else {
        MPI_Recv(lower_row_to_update, k_j, MPI_UNSIGNED_CHAR, (world_rank + 1) % world_size, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);//receive
        MPI_Send(upper_row_to_send, k_j, MPI_UNSIGNED_CHAR, world_rank - 1, 1, MPI_COMM_WORLD);//and send
    }
    for (int i = 0; i < k_j; i++)  //Update values
        block[(block_rows -1) * k_j + i] = lower_row_to_update[i];
    free(lower_row_to_update);
}
```

Figure 3: The message passing design

4 OpenMP scalability

4.1 Fixed size of the problem, increasing the number of threads

Table 1: Execution time for different number of threads. All the tests are done with a grid of size 4096, 10000 steps of computation and 3 epyc nodes.

OMP_NUM_THREADS:	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
Time:	257.88	129.42	65.08	32.94	17.03	8.78	5.01	3.69

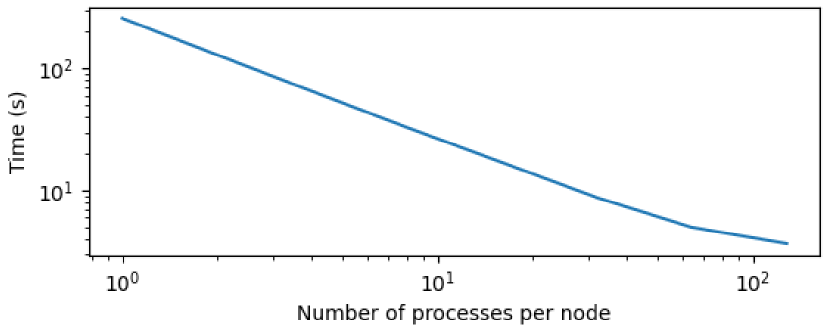


Figure 4: Behaviour of the execution time at the increasing of the number of threads, log scale

Table 2: Execution time for different number of threads

OMP_NUM_THREADS:	**1**	**4**	**9**	**16**	**25**	**36**	**49**	**64**	**81**	**100**	**121**
Time:	4.23	4.37	4.43	4.53	4.53	4.62	4.68	4.98	5.06	5.48	5.82

4.2 Fixed size of the problem per thread, increasing the number of threads

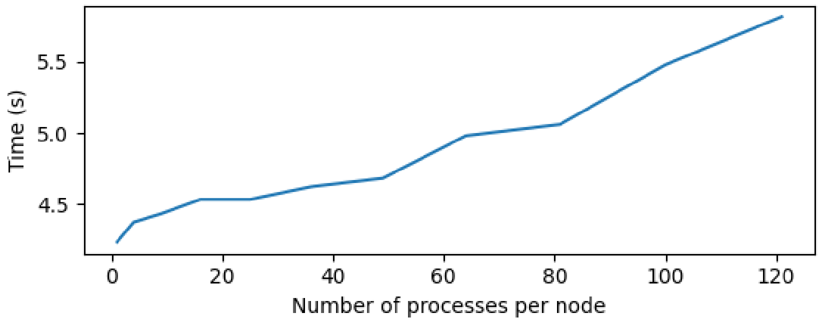


Figure 5: Behaviour of the execution time at the increasing of the number of threads, keeping the problem size per thread constant

It's clear that the program scales really well with the openMP threads, that's due to the simplicity of the problem and to the absence of race conditions. The amount of overhead required for coordinating the threads is therefore minimal. By the way exceeding the number of cores with the number of threads comes with a great penalty: with 130 threads (we have 128 cores available) the time of execution grows to 13.24 seconds, more than the time required for a 32 threads execution.

5 Strong MPI scalability

Table 3: Execution time for different number of processes. All the tests are done with a grid of size 4096, 10000 steps of computation and 4 epyc nodes.

MPI tasks per node:	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
Time:	193.15	96.97	49.01	25.56	13.69	8.58	8.30	43.83

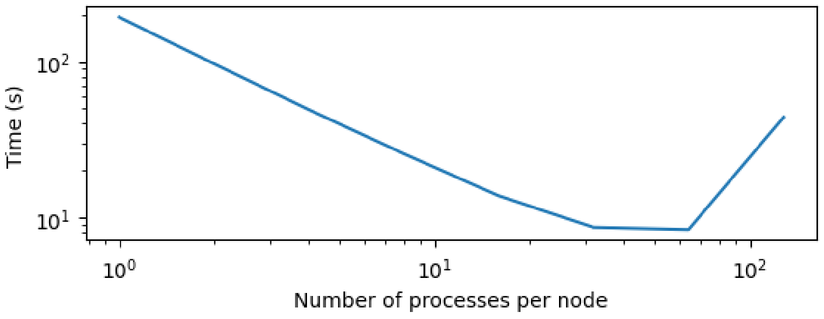


Figure 6: Behaviour of the execution time at the increasing of the number of processes, log scale.

For some reason increasing the MPI tasks over 32 doesn't come with any relevant execution time reduction, increasing the MPI tasks over 64 slows very much the execution. I initially thought that this strange behaviour was due to the limited size of the grid, but even after increasing it the results are pretty the same. Probably, due to the placement of the tasks, after a certain threshold the communication between them becomes a major bottleneck.

HPC project report: OpenBLAS and mkl

Michele Sanna

March 22, 2023

1 Scalability on the matrix size

The scaling with respect to the matrix dimension for the oBlas and MKL benchmarks is not linear. This behaviour was expected, being the matrix moltiplication an $O(n^3)$ problem. It's noticeable that the float version of the benchmarks gives a major GFlops output.

For what concerns the binding configurations of OMP, it's clear that the worst configuration are those that use OMP_PROC_BIND=master (almost an order of magnitude worst), where the others are pretty much equivalent, with the one that has OMP_PLACES=cores and OMP_PLACES=spread performing a little bit better.

1.1 Using float: mkl

Table 1: Execution time for different matrices dimensions

Matrix size:	1000	1500	2000	2500	3000	3500	4000	4500	5000
Time:	0.024	0.036	0.053	0.09	0.11	0.16	0.22	0.32	0.44
GFLOPS:	81.49	184.9	297	346.3	466.5	522	573	562	556

Table 2: Execution time for different matrices dimensions

Matrix size:	6000	7000	8000	9000	10000	12000	14000	16000	18000	20000
Time:	0.74	1.22	1.76	2.49	3.29	5.7	8.8	13.3	18.5	25.7
GFLOPS:	582	558	579	584	607	606	619	616	628	620

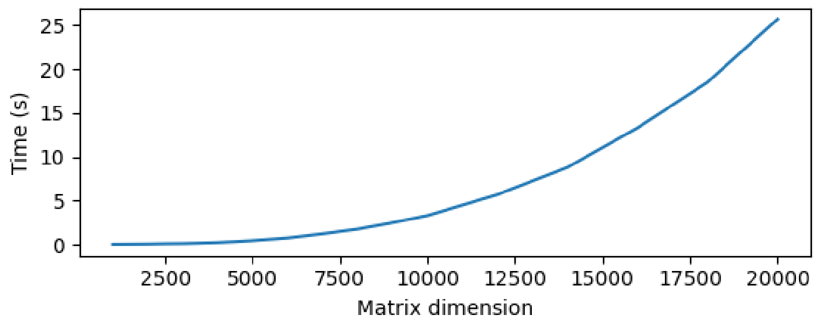


Figure 1: Mkl scaling with floats

1.2 Using float: oBlas

Table 3: Execution time for different matrices dimensions

Matrix size:	1000	1500	2000	2500	3000	3500	4000	4500	5000
Time:	0.042	0.16	0.177	0.21	0.27	0.37	0.41	0.55	0.64
GFLOPS:	47.1	40.9	89.9	147.2	196.4	227.6	309.7	326.9	389.1

Table 4: Execution time for different matrices dimensions

Matrix size:	6000	7000	8000	9000	10000	12000	14000	16000	18000	20000
Time:	0.83	1.26	1.74	2.54	3.25	4.98	8.1	10.6	13	18.7
GFLOPS:	518	545	587	572	616	692	670	770	891	851

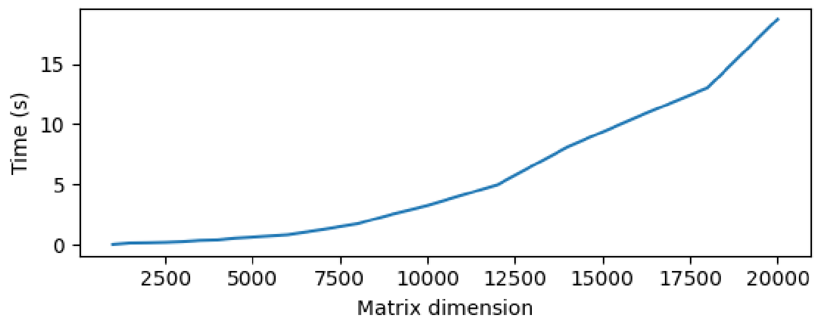


Figure 2: Mkl scaling with doubles

1.3 Using double: mkl

Table 5: Execution time for different matrices dimensions

Matrix size:	1000	1500	2000	2500	3000	3500	4000	4500	5000
Time:	0.031	0.053	0.089	0.125	0.19	0.28	0.39	0.53	0.71
GFLOPS:	63.02	125.31	179.4	249.33	282.96	304.9	325.5	340.4	352.3

Table 6: Execution time for different matrices dimensions

Matrix size:	6000	7000	8000	9000	10000	12000	14000	16000	18000	20000
Time:	1.15	1.79	2.57	3.58	4.85	8.1	12.6	18.78	25.94	35.56
GFLOPS:	376.13	382.3	398.02	407.5	412.2	429	434	436	449	449.9

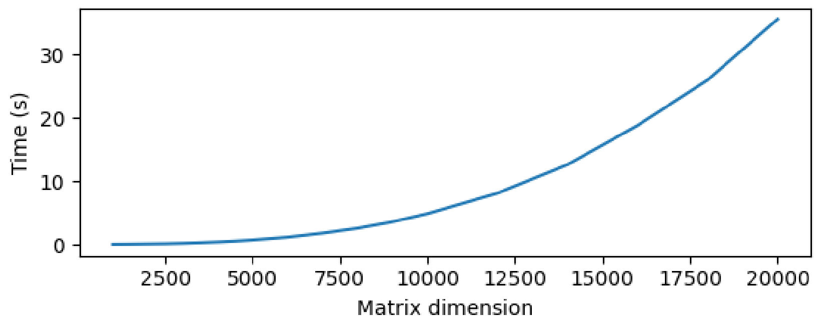


Figure 3: Mkl scaling with doubles

1.4 Using double: oBlas

Table 7: Execution time for different matrices dimensions

Matrix size:	1000	1500	2000	2500	3000	3500	4000	4500	5000
Time:	0.059	0.17	0.23	0.27	0.32	0.4	0.69	0.8	1.0
GFLOPS:	33.6	39	67	98.9	134.8	186	225.2	250	268

Table 8: Execution time for different matrices dimensions

Matrix size:	6000	7000	8000	9000	10000	12000	14000	16000	18000	20000
Time:	1.6	2.6	3.6	4.3	5.3	9.0	13.7	18.9	27.3	35.9
GFLOPS:	279	278	338	374.2	381.7	401	433	426	445	450

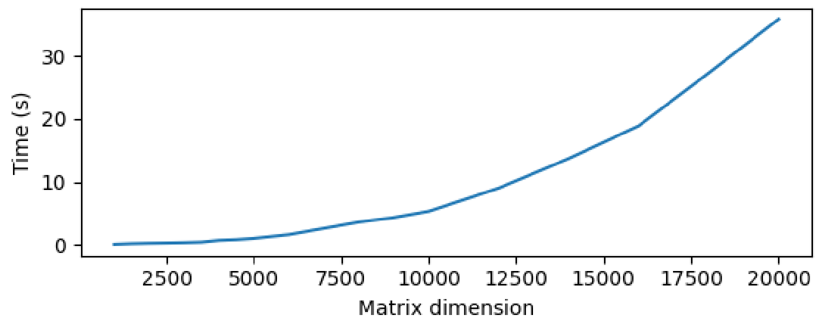


Figure 4: Oblas scaling with doubles

2 Scalability on openMP threads

For some reason these benchmarks don't scale well whit OpenMP threads. After reaching 32 threads per task, they tend to stall with performance or even worsen the times.

2.1 Using floats

Table 9: Execution time for different number of threads in Mkl using float. Log scale.

OMP_NUM_THREADS:	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
Time:	31.75	16.51	11.01	4.39	4.69	3.49	3.63	3.53

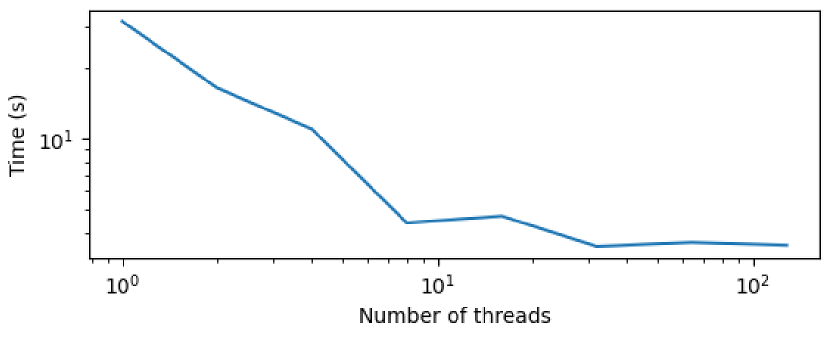


Figure 5: Mkl scaling with number of threads and using floats. Log scale

Table 10: Execution time for different number of threads in oBlas using float.

OMP_NUM_THREADS:	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
Time:	20.44	10.66	5.30	2.7	2.7	2.0	2.4	3.31

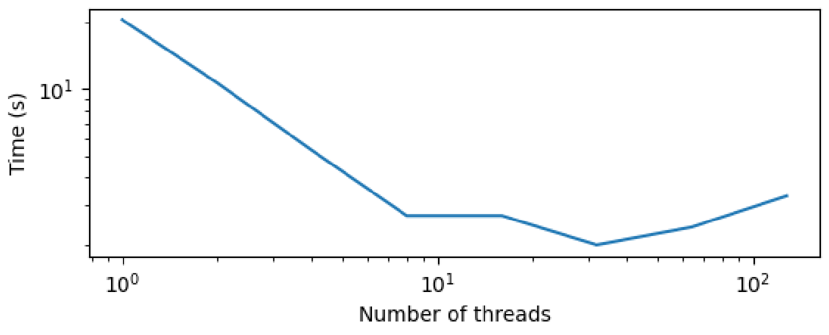


Figure 6: Oblas scaling with number of threads and using floats. Log scale

2.2 Using doubles

Table 11: Execution time for different number of threads in Mkl using double. Log scale.

OMP_NUM_THREADS:	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
Time:	43.1	21.9	11.16	5.7	6.18	4.99	5.32	5.55

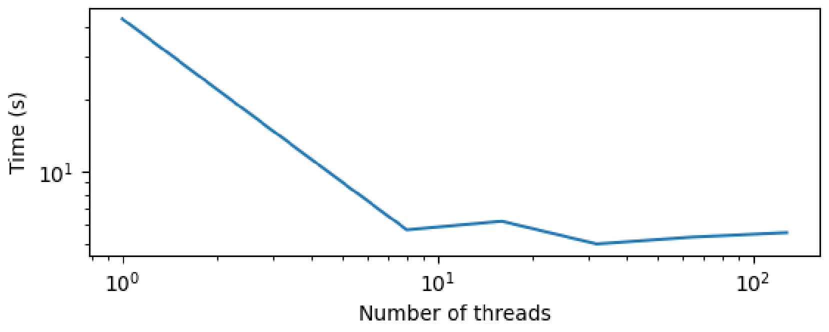


Figure 7: Mkl scaling with number of threads and using doubles. Log scale

Table 12: Execution time for different number of threads in oBlas using double. Log scale.

OMP_NUM_THREADS:	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
Time:	42.19	21.55	10.64	5.43	5.53	4.44	5.32	6.68

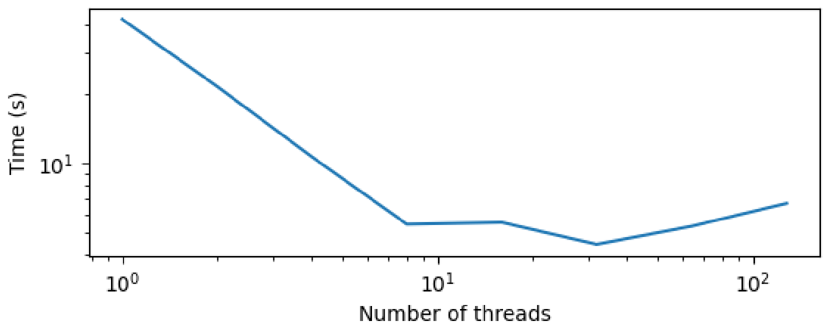


Figure 8: Oblas scaling with number of threads and using doubles. Log scale