Foundations of HPC
Final assignment

Andrej Cuber

December 4, 2023

Contents

1 Exercise 1

1.1 Introduction

The purpose of this exercise is to program a custom hybrid MPI + openMP application. Specifically we needed to implement the Conway's Game of life algorithm in parallel.

Game of life is a zero-sum game, which outcome is completely dependent on initial state of the grid cells, which are either dead or alive. We can represent the grid as an infinite two-dimensional grid. And at each iteration, neighbours of cell that we are interested in determine its next step status. There are three simple rules for the state of current cell:

- If there are exactly 2 or 3 cells that are alive in the neighbourhood of the cell we are interested in, it survives and at next evolution is alive.
- If there are less than 2 cells or more than 4 cells alive, then the cells dies.
- If there are exactly 3 live cells and the cell we are interested in is dead, it becomes alive.

1.2 Methodology

The way that I tackled the problem was to divide the original grid into smaller sub-grids and distribute those to each processor. I did that by assigning each processor m number of rows, and n number of columns (where original grid is sized xxn). I had to calculate for each processor how many rows will it get and in case of remainder (if the grid did not divide evenly by the number of processors), it got evenly spread across the first k processors (where k is the remainder). That way, sending and receiving elements from processor above and below was easy. Send the top row to process above you as bottom row and receive the bottom row as top row from processor above. For writting grids to files, I used serial code, that was provided to us. I have tried to parallelize writting and reading, with no luck, but I will write more about this later. I actually did not use the .pgm file format, but went with the .pbm instead, since I only run the grid with ones and zeros.

1.3 Implementation

First thing that I want to mention is that I developed the program completely on my own machine, prior to running it in ORFEO, that is why you'll find the first two lines with path to MPI and omp libraries. If you have M1 Mac, you can run the program by installing the libomp and open-mpi via homebrew package installer. After this we have some global variables.

In the following subsections, I will talk more in depth about the way that I coded functions. The name of the subsection is linked to the name of the function that you will find in the provided file.

1.3.1 Memory allocation

Crucial thing with parallelization is correctly allocating memory, because in case when we do not allocate it correctly, the sending and receiving of elements in MPI will be off, even if we correctly code up our messages that we want to send. In case the memory is not allocated contiguously, we can leave gaps in memory and thus sending messages with unknown sizes of gaps becomes hard. The way that I implemented allocation of the 2D array ensures contiguously allocated memory in row-major order, meaning that the first element of the bellow row is "in contact" with the last element of row above.

1.3.2 Generate initial image, write/read pgm image, write pbm parallel

The names of functions are pretty self explanatory. Generate initial image function takes grid dimensions and the address of allocated memory, which is then randomly populated with zeros and ones. Code is written in serial and is not parallelized, as the majority of times it is only ran once, to generate initial image. This could be parallelized and would be beneficial on big grids, with size above the 2.5 billion elements (grid size of 50 000 by 50 000). For example, to generate and write in serial an image of size 25 000 by 25 000 (625 million elements) took approximately 10s. Next two functions are copied from the provided material on the github. They read and write grid in serial. Write function does take 2D array without problem, while the read function return a one row element, that I needed to transform to 2D array and that takes some computation time when running the program. Next is my attempt at writting a parallel approach to writting elements of the grid to a file. Sadly, I failed (as of 25th of November 2023, the time of writting this section). The approach that I took was to use the MPI implemented functions to open and write to a file in parallelization, but I do not get the correct file out of this. There is for sure my data in it, but I am not sure that I get correct amount of elements and there is a bunch of weird elements in the saved file, plus it is almost double the size of the files coming from the provided functions.

1.3.3 Calculate workload

It is a function, that returns an integer array, where each position in the array, corresponds to the rank of a process. The values stored in the arrays are number of rows for each processor in the MPI world. First for loop stores the whole division while the second for loop evenly distributes the remainder. For example, if I am running program in MPI mode, where I allocate 30 processors for a 1000 by 1000 grid, each processor will get 33 rows, while the first process (rank 0) will get 34.

1.3.4 Game rules, evolve inner, evolve outer

All functions returns if the next generation changed (denoted by return changed = 1) or if it remained the same (return changed=0). In theory, you could have stopped the iteration sooner, if there was nothing changing, but, in a randomly filled grid as initial state, you get figures that oscillate. All three functions deal with static evolution of the game (ie, the cells are not dependent on the change of the neighbours cells).

Game rules changes the array allocated as next generation, based on the number of neighbours of a cell.

The evolution of the cells is split between outer ring and inner part. Inner part is easy task, have double for loop iterating through the grid, calculating the number of alive neighbours and make a call to the game rules function to actually evolute the next gen array based on the number of alive cells of a given cell. Outer ring is a bit more complicated. I split the outer ring on 8 parts. There are 2 rows (first and last) and to columns (again first and last). In both cases, I remove the first and last element (first element of the first row is shared with the first column, last element in first row is shared as first element in last column, etc ...). This is to easier coding of the for loops for rows and columns, because the corner elements use diagonal element which is harder to code in a for loop, that is why you will notice the top left, top right, bottom left, bottom right comments in the code.

In both evolve inner and evolve outer function I used the #pragma omp command which basically hyperthreads the for loop on all available processors, if we have compiled the program with openmp flag and told to hyperthread. In case we did not, the program runs in regular for loop.

1.3.5 Ordered evo *

* means starting with ordered evo and ending with either game rules, evolve outer or evolve inner.

Similarly, we have separate but similar functions to evolve the game in the ordered evolution, meaning that the next cell status is dependent on the status of the neighbour cell and how the neighbour cell changed at the current step. If, for example cell at position x=3, y=4 has in static evolution 3 alive neighbours but the cell at position x=2, y=4 remains alive, in ordered evolution this brings up the number of alive neighbour to number 4, which "kills" the cell. In the static evolution, the cell would survive, but in ordered it does not. In this case I evolve the outer ring first, starting with top element, continuing with first row and first column, than the last element of the first row and last element of the first column and then the last row and last column. After I am done with that I evolve the inner part of the grid, again starting at the top left corner of the grid and working my way down to bottom right. Rules remain the same.

1.3.6 Execute __ evolution

Main part of the ordered or static evolutions.

We allocate a local grid for each processor and a gather grid that will serve as a place in memory, where processes will send the data so I can actually save it in a pbm file. I allocate the space for top row and bottom row, that I receive from the process above and send to process below. I also allocated some space for snapshots names.

Initial idea was to split the initial grid into smaller grids, but one issue with this approach is that it does not allow for custom sized grids, or better said, coding up the splitting in case the number of processes does not divide evenly the grid into smaller squares is a very though task when you want to ensure correct division and being sure, that the process will only get the access to memory address that it needs. This issue actually made me ask classmate how he implemented the sending and receiving, and I got the answer that he sends only first row and the last row. This, combined with the fact that I have allocated the memory contiguously made the solution easier. I used the MPI_Scatterv command to scatter the original grid across all processes. This is a blocking call, meaning that until all processes returned successful status, the program will not continue. Con of this method is time required to scatter original data, so it is not advisable to call it many times in the program. Looking back, and talking with a bit more with classmates, one approach could be to not scatter the grid across processes, but to set boundaries for each process and let the game evolve in this way, of course with the double arrays (one for current generation, that we are reading, and another for saving next step) in case of static evolution. In the for loop, that evolutes the states, I have used the MPI_Sendrecv, which is basically a blocking send and receive command (instead of writting two lines I could write only one). I used blocking type to ensure that I have top and bottom row prior to starting iterating the the game.

After this, if the condition is met to save a file, I used the MPI_Gatherv function, to collect and write the iteration as snapshot file. Again, this is locking call, until all data is collected and stored sequentially based on the way it was distributed across processors, the program will not continue with execution. What I have noticed is that the odd numbered iterations are a bit shifted in regards to even iterations. Looking at the sequence of even iterations, there is no indication, that there is something wrong with the way I compute evolutions, so I am confused a bit here, what has gone wrong. Maybe it is just saving problem? Not sure. Again, I save the file in serial, using the provided functions.

Maybe at this point I can touch a bit about performance. If I decide to save the file often, the barrier (at gathering) data is called and this reduces the performance of my program. Same goes for writting the file, since this is done in serial and slows down the program even more. What I have also noticed, you can speed up a lot, if you already have the file in the system and the program only opens up file and rewrites it, rather than creating a new file from scratch and write into it.

1.3.7 The main function

Before actually calling the execute _ evolution, I must address some choices that I have done in the main function. First of all, if I want to use the functions that were provided for reading and writting the files, after reading the file, I need to convert it into 2D grid, as what I get returned is 1D grid (single line). Next, I have separated the cases and how I initialize the MPI based on if the program is compiled with openMP flag or not. What follows is the standard initialization of the MPI world. After this, I calculate the workload of each processor and the displacement in memory for each processor and the index of the last row in local grid scheme. I used MPI_Broadcast to share the information of array across all processes (otherwise I got some strange errors). Another fundamental thing that I have done is find the neighbours of the processes. With top I denoted neighbour that is above random process and with bottom I have denoted the random process that we want to compute the neighbour for. Why I have not bothered to find the neighbour below? Because it is a cyclic operation. If I have n processes, for process 0 (or root), neighbour above is $(n-1)$-th process and can be denoted as top, while root gets denoted as below. Similarly, for process 1, neighbour above is 0, so we can denote root as top and 1 as bottom. That way, the top process sends bottom row to below and below sends the top row to top. And this goes in circle to send all the rows to the neighbours. After this comes the call for evolutions, based on what we have chosen, either ordered or static. Rest of the main function is then basically calculating the time for MPI computation and freeing memory.

1.4 Results and discussion

In the next few subchapters I will present results that I have obtained by running my implementation of code. This exercise was ran only on EPYC partition of the ORFEO.

1.4.1 Strong scaling

This is scaling where I allocate all available cores to MPI tasks. I have ran measurements both for ordered and static evolution.

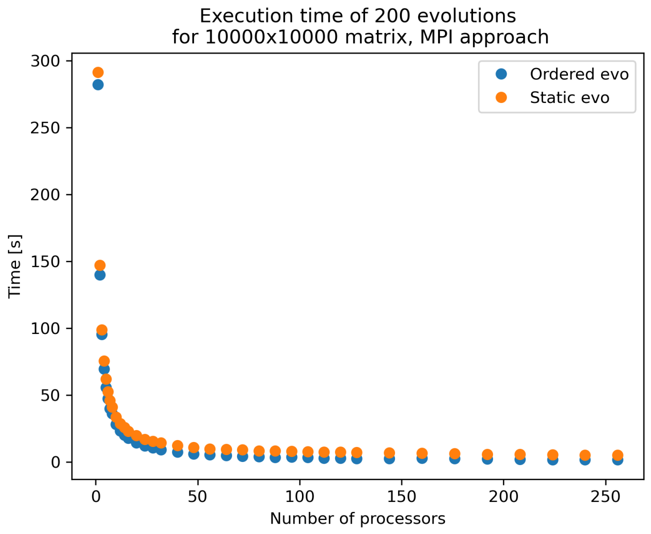

Figure 1: Shows time execution in seconds for ordered and static evolution of the game, ran on a grid of 10000x10000 and for 200 evolutions.

As you have may noticed, the ordered evolution is faster than static, as the way I have implemented the algorithm, it is done in place, instead of using a separate array for ensuring that I do not overwrite anything and in that way influence the game outcome in case of static evolution.

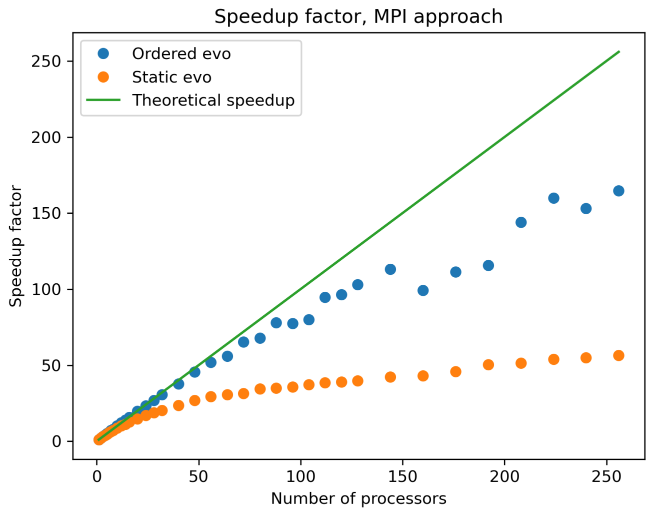

Figure 2: Shows the speed up in relation to increasing number of processors used for MPI tasks

Speed up figure shows measured speed up and the theoretical one. At the beginning we see that the scaling works close to perfectly, following theoretical speed up line, while at some point the curve flattens and we begin to have a diminishing returns on the speed.

1.4.2 Hybrid scaling

This is scaling, where I fix 1 MPI task per socket and increase the openMP threads. Again, I have ran measurements for both ordered and static evolution.

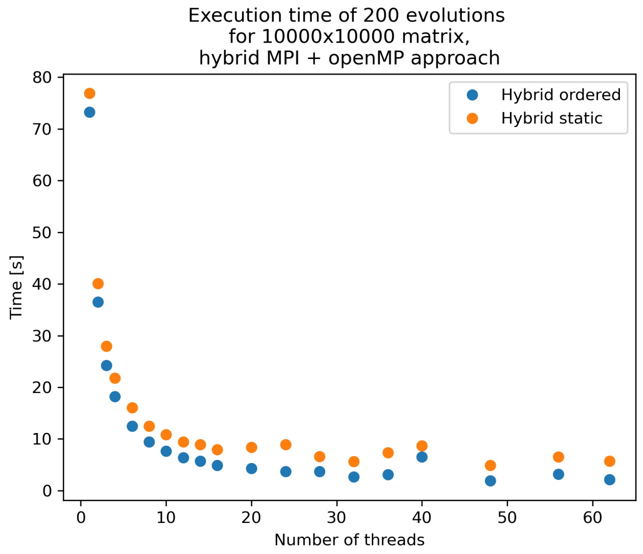

Figure 3: Shows the time execution in seconds for ordered and static evolution of the game, ran on grid of 10000x10000 and for 200 evolution

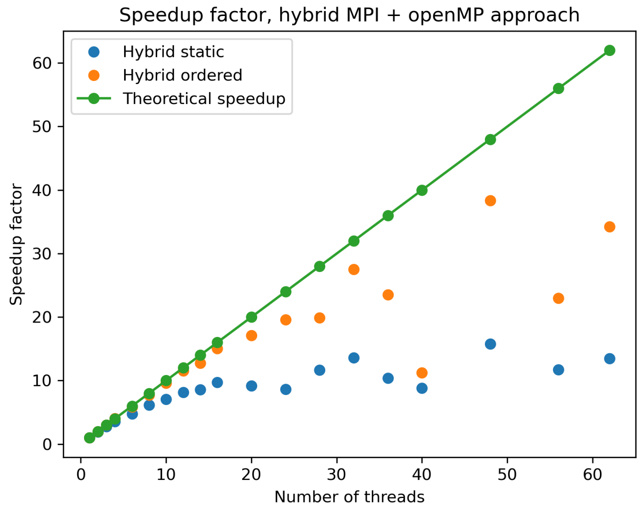

Figure 4: Shows the speed up in relation to increasing number of threads in hybrid approach.

1.4.3 Weak scaling

This is scaling, where I increase number of sockets which are saturated with openMP threads. I also increase grid size such that I have roughly the same workload for each socket.

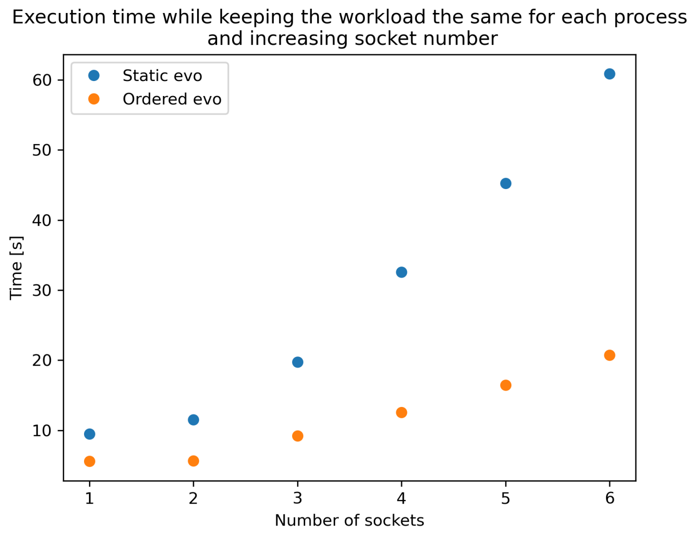

Figure 5: Execution time in seconds for increasing number of sockets

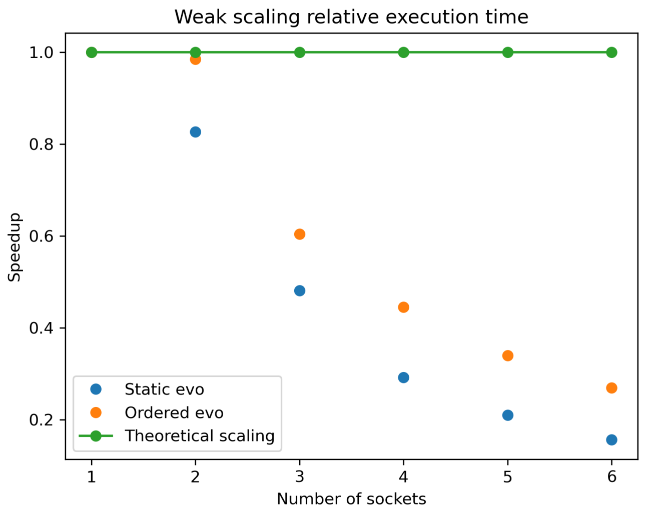

Figure 6: Shows relative scaling when I increase the number of sockets.

This result left me perplexed as I expected to see a fall-off, but not at this rate. If the ordered evolution kinda follows theoretical expected results for the second measurements and then starts to slowly fall-off, the static one does not follow it a tiny bit. I was also very careful at increasing size, adding roughly 100 million elements (as I started with 10000x10000 grid) for each added socket yet the performance does not remain kind off same. There are plenty reasons: diving square matrix into smaller blocks is heavier than dividing matrix that has 10000 rows added (if I fixed matrix to 10000 columns), sending longer rows between tasks takes longer time, etc ...

1.4.4 Some addition to the ordered evolution

First definition of alogrithm was done without pragma call (that starts multi-threaded for loop) to ensure that first eleemnt is actually influencing the whole development of the grid and when I measured execution time, it happily sit at around 110s, for a grid of 4000x4000 and 2000 iterations. In the graphs that you are seeing in this report, I have done it with openMP threading while ensuring that first element (top left) of local grids has changed. Here I have done a small mistake too, because after I have changed the first element, I should have evoluted the top row and then send it to the process above me as bottom row. I have not done that step.

1.5 Conclusion

My implementation is on the safer side, I think, as I truly ensured that all processors get necessary data before actually doing anything. One downside to this approach is having slower algorithm but on the other hand it helps with understanding what the program does. Maybe with more experience with MPI and openMP, it would allow me to be bolder, use the non blocking send and receive which should speed things up (due to less latency and waiting between processes). Additionally implementing in place evolution would speed things up too (as shown in comparison between ordered and static evolution).

Overall, for somebody that was not used to C language and to parallelization world in programming, I am content with obtained results.

2 Exercise 2

2.1 Introduction

In this exercise we had to measure the performance of matrix multiplication using three different math libraries (MKL, openBLAS, BLIS) and two different precisions (single, double). Tasks:

- Measure scalability over the size of matrix at fixed number of cores (12 for THIN, 64 for EPYC)
  - Increase the size of matrices from 2000 x 2000 to 20000 x 20000 and analyse scaling of GEMM calculation for at least MKL and openBLAS libraries using single and double precision.
  - Compare results with the peak performance of the processor.
  - Play with different thread allocations and compare the results.
- Measure scalability over number of cores at fixed size of matrices. Choose an intermediate size and do:
  - Increase the number of cores and analyse the scaling of GEMM calculation for (at least) MKL and openBLAS libraries using both single and double precision.

2.2 Methodology

This exercise from the point of approach was rather simple. I was given a file, which I had to benchmark on ORFEO.

One thing that I did was to modify the output of gemm.c file, where I commented out most of the print functions, as I am only interested in timings, reported GFLOPS and which precision I am using. The dimension of matrices was printed in slurm output file via shell script.

Results files were then taken to a python script where I modified and plotted the results, as I am most familiar with the plotting tools of python.

2.2.1 Peak performance

One important part in comparisons in this exercise is comparing the results against theoretical peak power. This is obtained by a simple calculation:
$GFLOPS = frequency$ x $operations$ x $processors$. Both EPYC and THIN has same base clock of CPU set at 2.6GHz, while the turbo mode is 3.7GHZ for THIN and 3.3GHZ for EPYC. Since the latter two values are reached only in special cases, I chose as my theoretical peak performance the base clock speed of processor. Assuming this, we get for EPYC at double precision: 41.6GFLOPS, at single precision: 83.2GFLOPS per core, while for THIN at double precision: 83.2GFLOPS and 166.4GFLOPS for single precision.

2.3 Implementation

As I said previously, the gemm.c file was modified to output only information that I am interested in, to speed up the process of analysis and plotting the results.

To analyse the performance of GEMM I chose the EPYC node, so I compiled the BLIS library for AMD first, modify the make file to include the path to the BLIS library and ran once to output the compiled runnable applications for double precision (files are denoted as *double.x) and then I modified the makefile again, to use the single precision (files are denoted as *float.x) (* means that the files are ending with yy.x, where yy is either float or double). There was no further modifications of the make file.

I submitted jobs to cluster via SLURM sbatch option, which needs shell scripts that are then used to instruct the slurm what it needs to do. All shell scripts are found on github.

When I had to choose the size of the matrices I decided on using a 9000x9000 matrix. Even this size was for some processes too large in sbatch script, so I tested it in iterative mode and program compiled it no problem. So what I did was for the missing values from the sbatch script, I ran it manually in iterative mode and inserted it into table.

2.4 Results and discussion

2.4.1 Increasing the number of threads

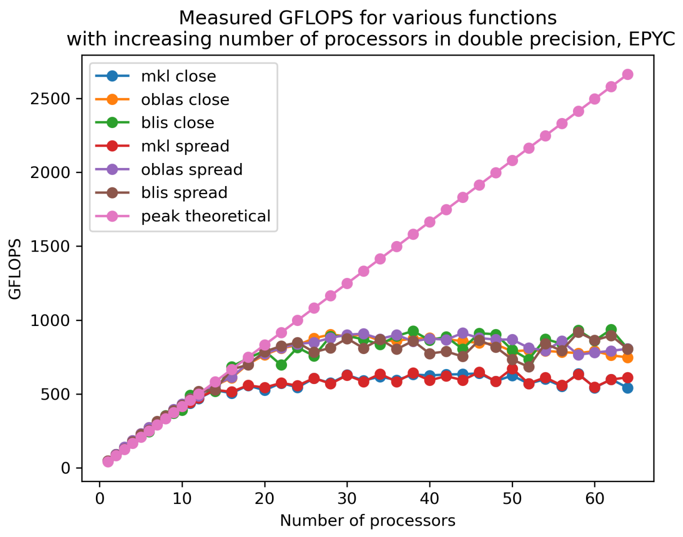

Figure 7: Measured GFLOPS for functions with increasing number of processors in double precision on EPYC

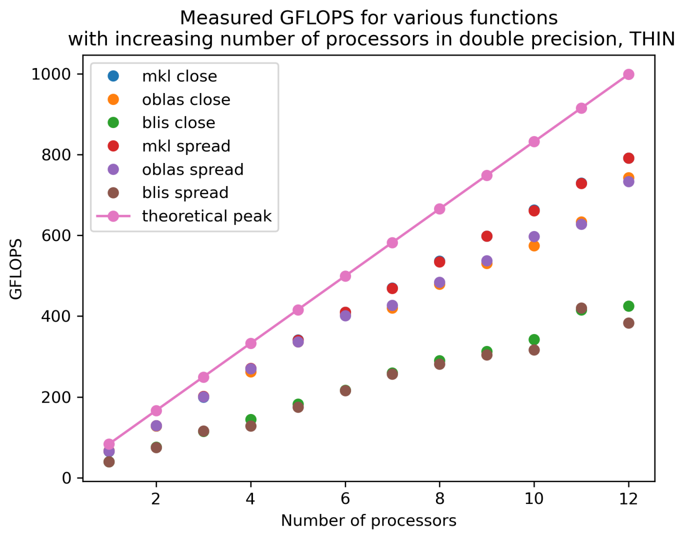

Figure 8: Measured GFLOPS for functions with increasing number of processors in double precision on THIN

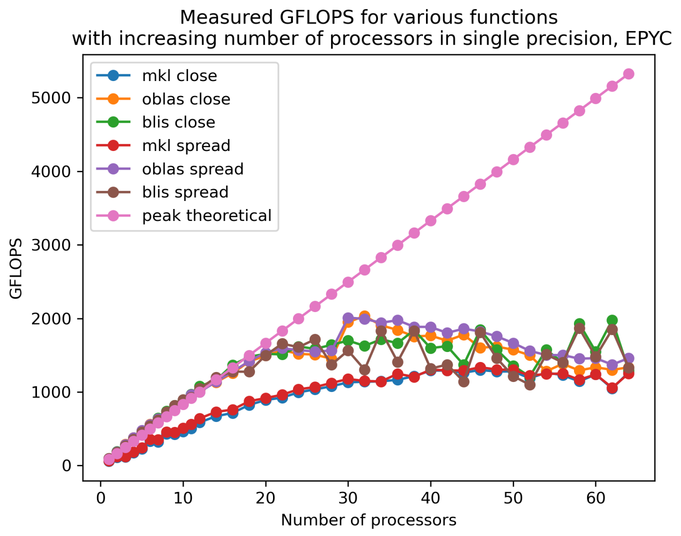

Figure 9: Measured GFLOPS for functions with increasing number of processors in single precision on EPYC

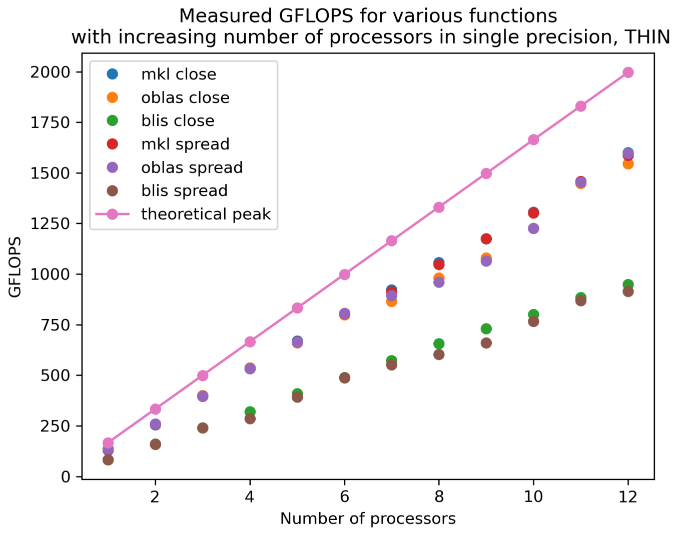

Figure 10: Measured GFLOPS for functions with increasing number of processors in single precision on THIN

What can be observed in the above figures is that some functions report higher GFLOPS than the theoretical one. The thing is, since I do not know the CPU clock under sustained load, I need to do an approximation. One way would be to use the boosted reported clocks, but those aer not sustainable and are only relevant in certain cases, for example reading the documentation and specification of Intel Xeon processor that is used in THIN nodes I read that the 3.7GHz value (boosted) is relevant only for single core applications and it is not sustained. The approximation that I ended up doing is, taking the base clock value and do the calculations for each process. What we can observe is that the processor under full load was at around 2.6GHz. The staling after a certain point around 20 processors is due to the nature of scaling. This is better shown on the next four images, where we can observe another thing, that the THIN nodes had smaller fall-of in comparison to the EPYC.

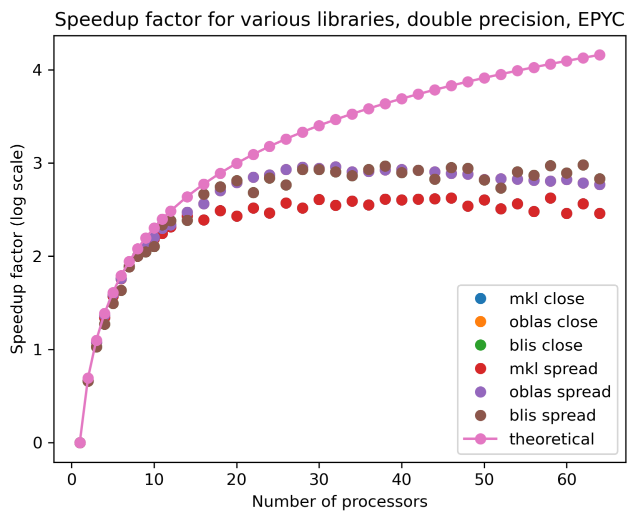

Figure 11: Shows speed up factor when increasing number of processors using double precision on EPYC nodes

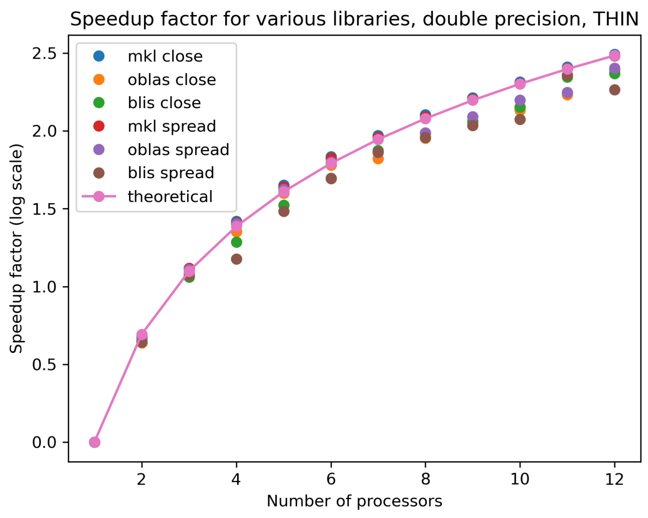

Figure 12: Shows speed up factor when increasing number of processors using double precision on THIN nodes

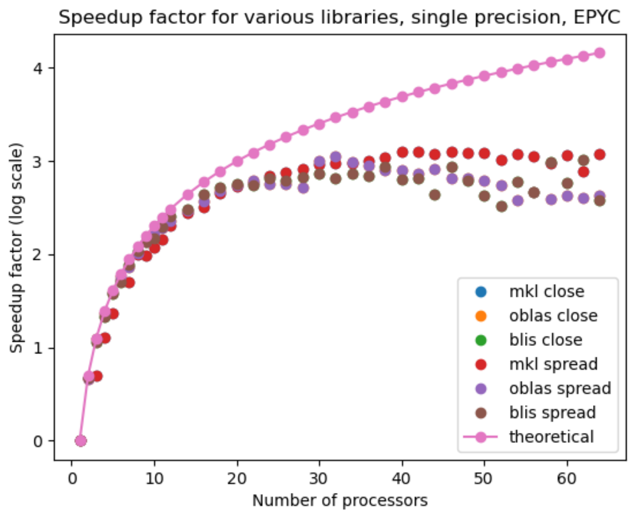

Figure 13: Shows speed up factor when increasing number of processors using single precision on EPYC nodes

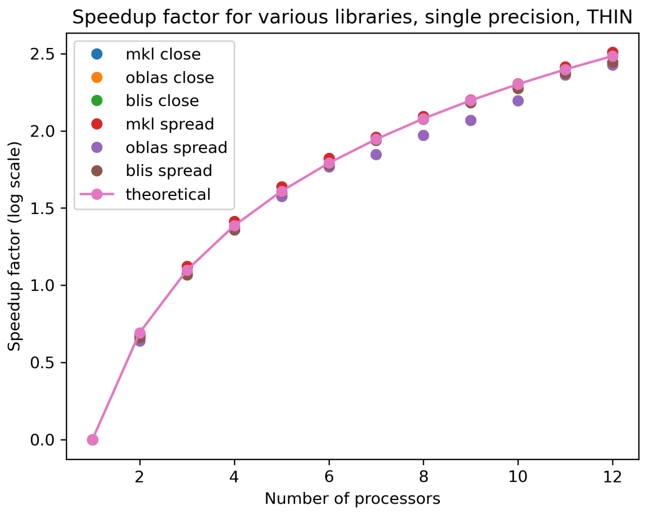

Figure 14: Shows speed up factor when increasing number of processors using single precision on THIN nodes

2.4.2 Increasing matrix size

In this case we had to keep the number of processors untouched and change the dimension of the matrix from 2000 to 20000.

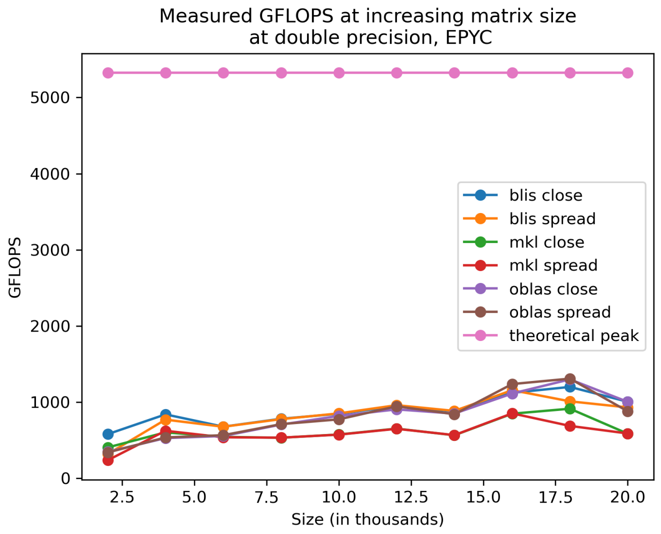

Figure 15: Measured GFLOPS for increasing matrix size using double precision on EPYC nodes

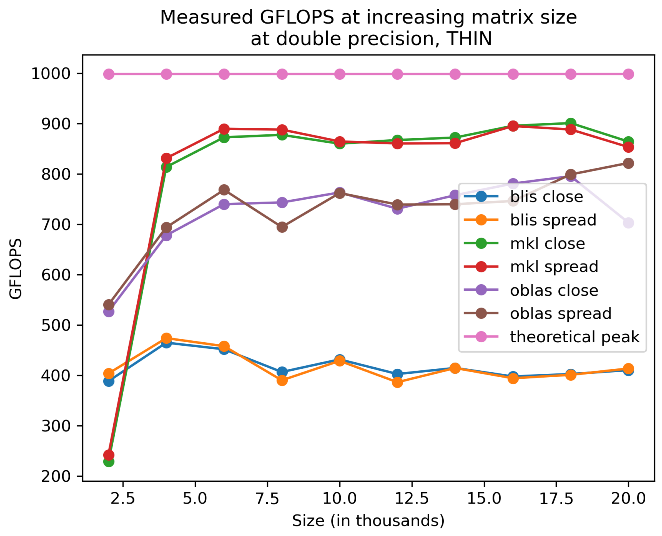

Figure 16: Measured GFLOPS for increasing matrix size using double precision on THIN nodes

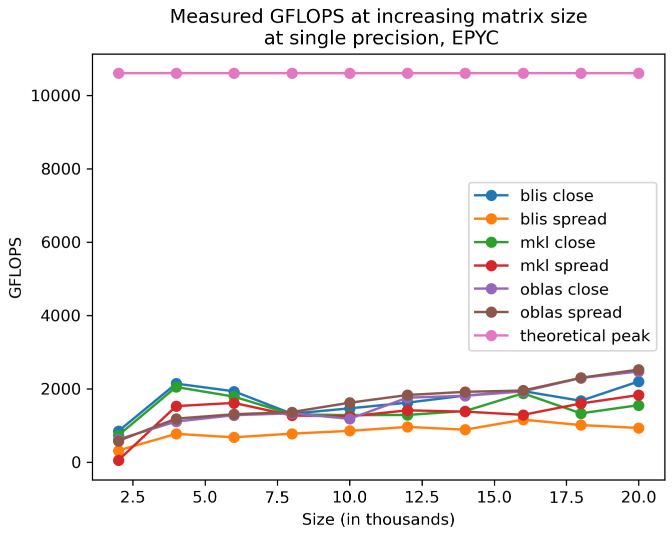

Figure 17: Measured GFLOPS for increasing matrix size using single precision on EPYC nodes

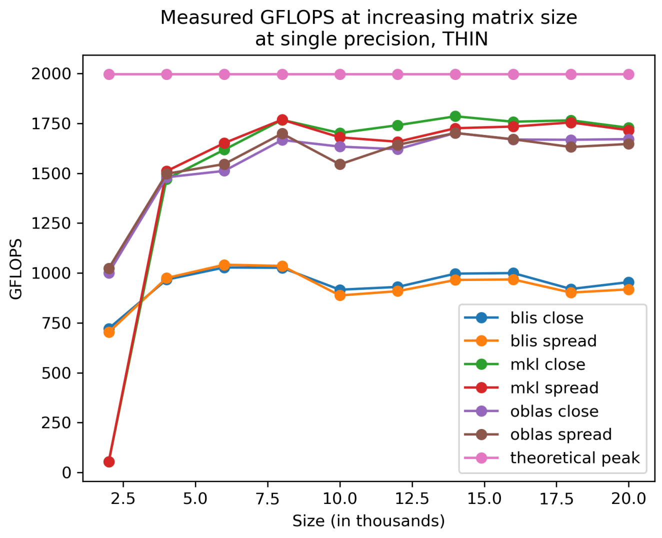

Figure 18: Measured GFLOPS for increasing matrix size using single precision on THIN nodes

While EPYC nodes were closer together, the THIN nodes were more apart in performance and closer to the theoretical peak performance. No fall-off in performance was observed.

2.5 Conclusion

I am a bit puzzled because there is close to no difference in spread or close policy of the position for cores. This goes for both subparts of the exercise. I was expecting a bigger difference between the two, at least observable one. Maybe I have done mistake when I called OMP thread position function in my code. Generally, we can see the performance fall-off clearly with EPYC nodes, for THIN nodes we would need a bit more cores on CPU.