Foundations of High Performance Computing

Final assignment

Flavia De Santis

June 25, 2024

AA 2022/2023

Contents

1 Exercise 1

1.1 Introduction

This section presents a discussion on a potential hybrid implementation of the Game of Life using both MPI and OpenMP.
Different types of evolution are considered, although here only static evolution has been implemented:

- **Ordered evolution**: the evolution of a cell depends on the evolution of its neighboring cells (serial).
- **Static evolution**: cells are evolved simultaneously based on their present neighboring cells.
- **Black-White (static) evolution**: variation of the static evolution, in which the evolution of black (alive) cells takes place before white (dead) cells evolution.

1.2 Approach

Before looking into the implementation of a parallel solution, it's important to identify areas within the problem where parallelism can be effectively employed. In this particular case, we're adopting a hybrid solution, requiring the distribution of data among both processes and threads.

1.2.1 Parallelism at the Process Level

In this specific implementation, the grid's evolution is divided among multiple processes. The partitioning involves breaking down the grid into groups of its rows, with each process focusing on a distinct group of rows. If the number of rows in the grid are not evenly divided among processes, the remaining rows are assigned to the final process.

1.2.2 Parallelism at the Thread Level

Subsequently, each process spawns threads tasked with independently performing operations on individual cells to determine the grid's evolution. Thread partitioning is accomplished using OpenMP and the parallel OMP `pragma for` directive.

Once the domain decomposition is complete, each thread proceeds to calculate the number of living neighbors for each cell in its designated portion of the grid and facilitates its evolution.

1.3 Implementation

The code is divided into 3 different `.c` files:

- `main.c`: It contains the functions needed to initialize the playground. It also contains the `main` function responsible for program execution, including the retrieval of input arguments and the invocation of appropriate functions based on these inputs.
- `evolution.c`: It contains the code that performs the static evolution, either in parallel or serially.
- `read_write_pgm_image.c`: It contains the functions `read pgm image()` and `write pgm image()`, which are used to manage the reading and writing operations, respectively, on `.pgm` images.

And 2 `.h` header files:

- `evolution.h`: It contains the headers needed in `evolution.c`.
- `read_write_pgm_image.h`: It contains the headers for the functions implemented in `read_write_pgm_image.c`.

The compilation and linking processes are facilitated by a `Makefile`, which is configured with the necessary compilation flags, such as `-fopenmp` to enable OpenMP thread support, and the appropriate MPI wrapper for MPI-related operations.

To determine the number of living neighbors, a threading approach is employed, requiring information about the states of all eight neighboring cells. However, cells located on the borders of sub-grids lack a complete neighborhood.

This limitation is overcome by modeling the grid as a torus, wrapping around at the top, bottom, and sides. Ghost rows and columns are introduced as copies of rows and columns on the opposite sides of the grid's edges. The challenge in computing ghost rows arises from the fact that the corresponding rows are stored in processors on other nodes or sockets, necessitating the exchange of messages.

1.3.1 Ghost rows

The MPI Send and MPI Receive commands facilitate the transmission of rows. Specifically, the first row is sent to the upper process, and the lower ghost row is received from the lower process. Similarly, the last row is sent to the lower process, while the upper ghost row is received from the upper process. The computation of ghost columns follows a similar process, but it is necessary to exchange ghost rows first, as they are crucial for determining corner values. Consequently, the right column becomes the left ghost column, and the left column becomes the right ghost column.

1.3.2 Ghost columns

The torus modeling strategy not only addresses the issue of incomplete neighborhoods but also underscores the necessity of message exchanges between processors to obtain upper and lower rows. Additionally, the computation of ghost columns is simplified, requiring only a straightforward calculation once each process has received the ghost rows. This approach ensures the computation of corners, with the last column representing the left ghost column and the first column representing the right ghost column.

1.3.3 Domain decomposition

Domain Decomposition among Processes For the domain decomposition phase, process 0 (master process) obtains the number of rows and columns of the grid from the initial `.pmg` image and sends these values to the other processes. Each process calculates the number of rows and columns it will work on. The master process reads the `.pmg` image into an array and sends chunks of rows to the other processes, keeping a portion for itself.

```
[...]

    int grid_size=rows*cols;    //dimension of the grid
    int local_rows=rows/size;   //number of rows on which each
       process will work
    int local_cols=cols;        //number of rows on which each
       process will work
```

```
    //check if the number of rows is divisible by number of processes
       , if not the last process will work on remaining rows
    int offset=rows%size;
    if (rank==size-1){
      local_rows=local_rows+offset;
    }

    int local_grid_size=local_rows*local_cols;      //size of the
       local grid of each process
```

```
        //process 0 reads image, send local grids to other processes
           and keep one for itself
    if (rank==0){
      int *full_grid_current=(int*) malloc(grid_size*sizeof(int));
      readin_array(full_grid_current, file_name, rows, cols);

      for (int i=0; i<local_grid_size; i++){
        local_grid_current[i]=full_grid_current[i];
      }

[...]

    //send local grids to other processes
      for (int i=1; i<size; i++){
        if (i<size-1){
          MPI_Send(&full_grid_current[i*local_grid_size],
             local_grid_size, MPI_INT, i, 0, MPI_COMM_WORLD);
        }else{
          MPI_Send(&full_grid_current[i*local_grid_size], (local_rows
             +offset)*local_cols, MPI_INT, i, 0, MPI_COMM_WORLD);
        }
      }

      free(full_grid_current);
    }
    //receive the local grids from process 0
    if (rank !=0){
      MPI_Recv(&local_grid_current[0], local_grid_size, MPI_INT, 0,
         0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
```

Message Passing and Ghost Columns Computation After domain decomposition, the processes exchange the required rows and compute the ghost columns using the functions shown in the snippet below. All data are saved in a matrix with dimensions of rows + 2 and columns + 2.

```
[...]

void exchange_ghost_rows(int *local_grid_ghost, int local_rows_ghost,
    int local_cols_ghost, int upper_rank, int lower_rank){
```

```
  MPI_Request request1, request2;
  MPI_Isend(&local_grid_ghost[local_cols_ghost], local_cols_ghost,
                  MPI_INT, upper_rank, 0, MPI_COMM_WORLD, &request1
     );
  MPI_Irecv(&local_grid_ghost[(local_rows_ghost-1)*local_cols_ghost],
      local_cols_ghost, MPI_INT, lower_rank, 0, MPI_COMM_WORLD, &
     request1);

  MPI_Isend(&local_grid_ghost[(local_rows_ghost-2)*local_cols_ghost],
      local_cols_ghost, MPI_INT, lower_rank, 1, MPI_COMM_WORLD, &
     request2);
  MPI_Irecv(&local_grid_ghost[0], local_cols_ghost, MPI_INT,
     upper_rank, 1, MPI_COMM_WORLD, &request2);
  MPI_Wait(&request1, MPI_STATUS_IGNORE);
  MPI_Wait(&request2, MPI_STATUS_IGNORE);
}

void compute_ghost_cols(int *local_grid_ghost, int local_rows_ghost,
    int local_cols_ghost){
  for (int i=0; i<local_rows_ghost; i++){
    local_grid_ghost[i*local_cols_ghost]=local_grid_ghost[(i+1)*
       local_cols_ghost-2];
    local_grid_ghost[(i+1)*local_cols_ghost-1]=local_grid_ghost[i*
       local_cols_ghost+1];
  }
}

void compute_ghost_rows(int *grid, int rows, int cols,  int
   rows_ghost, int cols_ghost){
  for (int i=1; i<=cols; i++){
    grid[i]=grid[cols_ghost*rows+i];
    grid[cols_ghost*(rows_ghost-1)+i]=grid[cols_ghost+i];
  }
}
```

```
//count the alive neighbours
int count_alive_neighbors(int *grid, int i, int j, int cols){
  int neighbours=grid[(i-1)*cols+(j-1)] + grid[(i-1)*cols+j] + grid[(
     i-1)*cols+(j+1)]+
                grid[i*cols+(j-1)] + grid[i*cols+(j+1)]+
                grid[(i+1)*cols+(j-1)] + grid[(i+1)*cols+j] + grid[(
                   i+1)*cols+(j+1)];

  return (8-neighbours/255);   //since dead=255 and there are 8
     neigbours, the number of alive neighbours will be 8-neighbours
     /255
}

//apply static evolution algorithm
void static_evo(int *grid, int *grid_next, int rows, int cols){
  #pragma omp parallel for schedule(static)
```

```
  for (int i=1; i<rows-1; i++){
    for (int j=1; j<cols-1; j++){
      int count=count_alive_neighbors(grid, i, j, cols);
      if (grid[i*cols+j]==ALIVE && (count==2 || count==3)){
        grid_next[i*cols+j]=ALIVE;
      }else if(grid[i*cols+j]==DEAD && count==3){
        grid_next[i*cols+j]=ALIVE;
      }else{
        grid_next[i*cols+j]=DEAD;
      }
    }
  }
}

[...]
```

After each evolution or once all evolutions are completed, the result is saved in a binary image in `.pgm` format.

1.4 Results

Tests were conducted to assess the performance of the code under different OpenMP/MPI configurations and grid sizes for static evolution. The evaluations was made on three scalabilities:

- OpenMP scalability.
- MPI weak scalability.
- MPI strong scalability.

The key metrics for performance comparison included execution time and Speedup S, defined as follows:

$$S_i = \frac{T_i}{T_p}$$

where T_i is the average execution time of a single process averaged for 5 iterations and T_p is the time due for p processes.

1.4.1 OpenMP Scalability

To analyze OpenMP scalability, a single task was assigned to a node, and the number of threads increased with each run. The tests used the OpenMP option `OMP PROC BIND = close` and the mpirun option `-map-by socket`. The grid size 5000 was employed, with each evolution iterating 100 times.

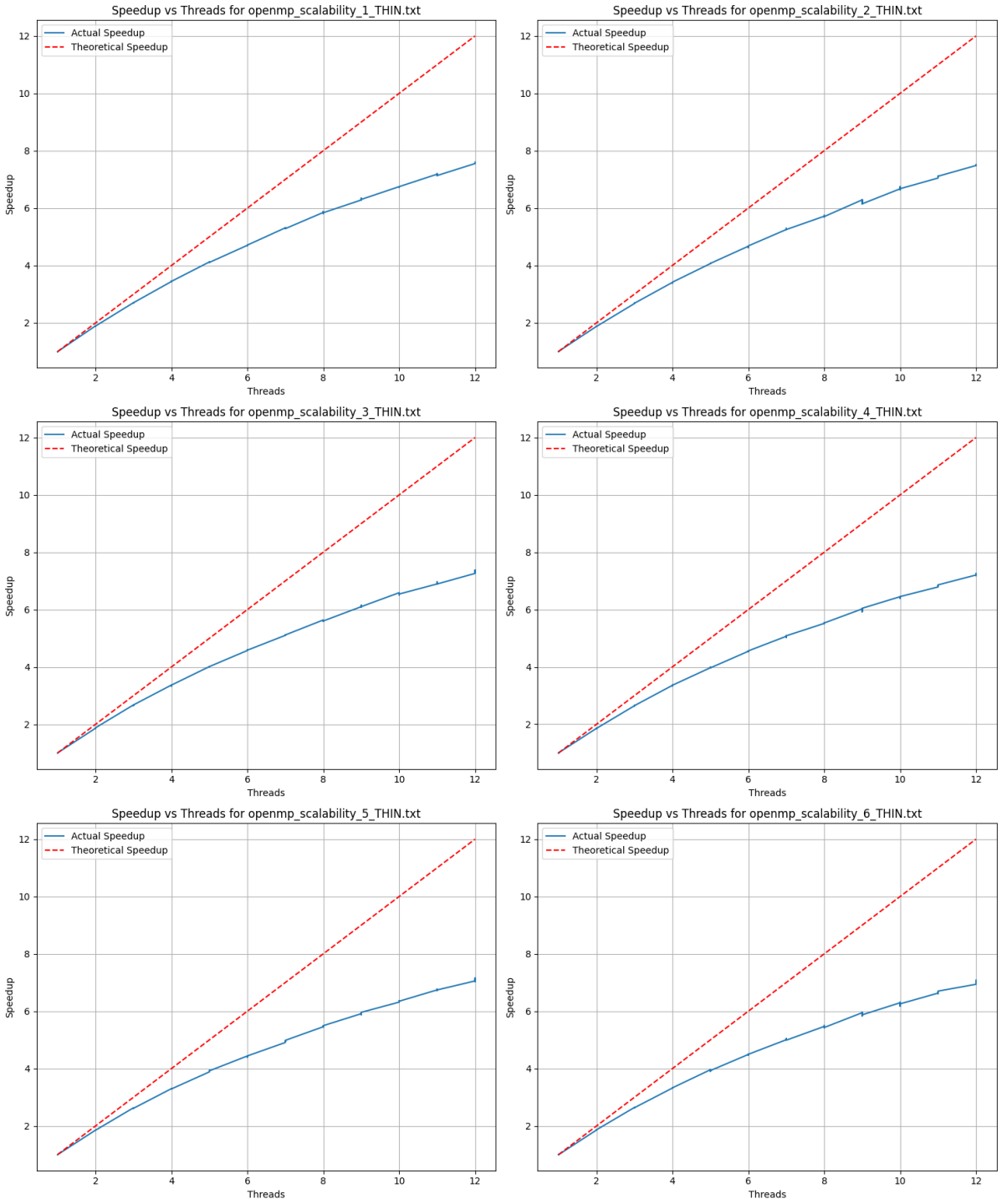


Figure 1: OpenMP Scalability from 1 to 6 threads on THIN nodes

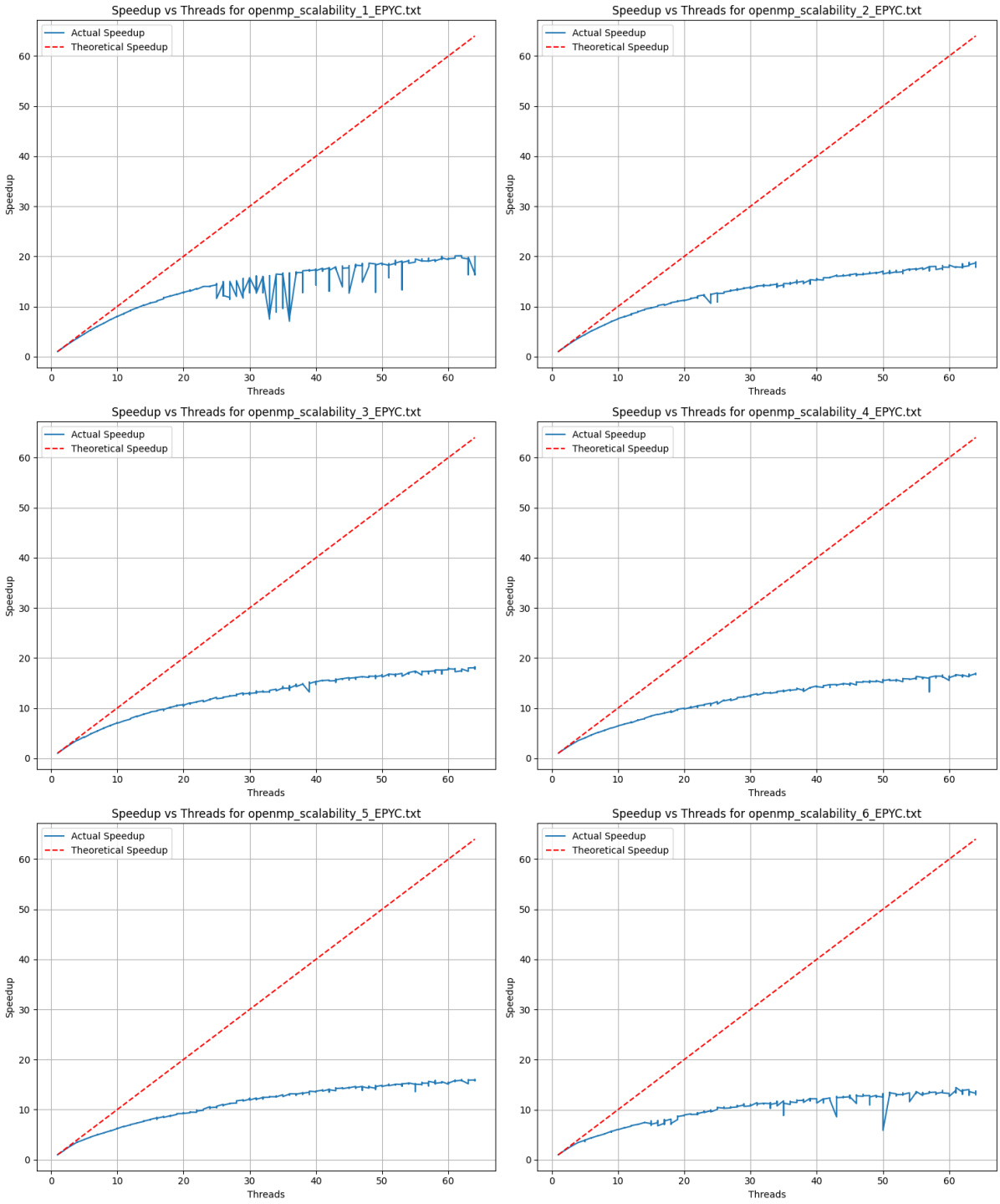


Figure 2: OpenMP Scalability from 1 to 6 threads on EPYC nodes

Figure 1 and 2 illustrate the test results.

1.4.2 MPI Weak Scalability

For MPI weak scalability, the workload is kept constant for each task. Starting from an initial grid size, the number of cells doubles for each task, following the following table:

Tasks	Size
1	5000
2	7071
3	8660
4	10000
5	11180
6	12247

Table 1: Tasks and their corresponding Grid sizes

The test was run with 64 threads for each process and then with just one thread. Ideally, the runtime would remain constant, but as shown in Figures 3 and 4, it slightly increases with the number of tasks in both cases. This is likely due to an increase in overhead as the number of tasks rises.

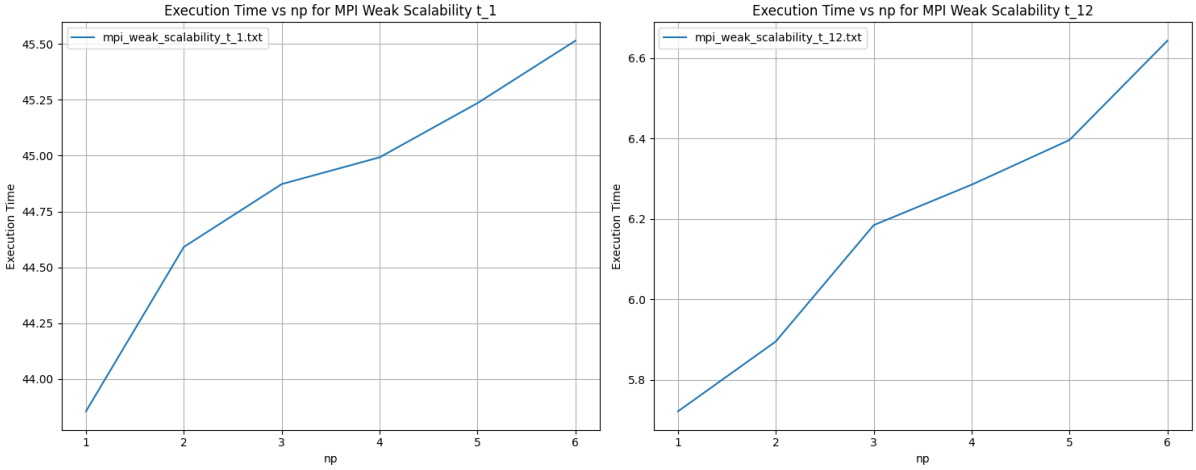


Figure 3: MPI Weak Scalability on THIN nodes, for 1 and 12 threads

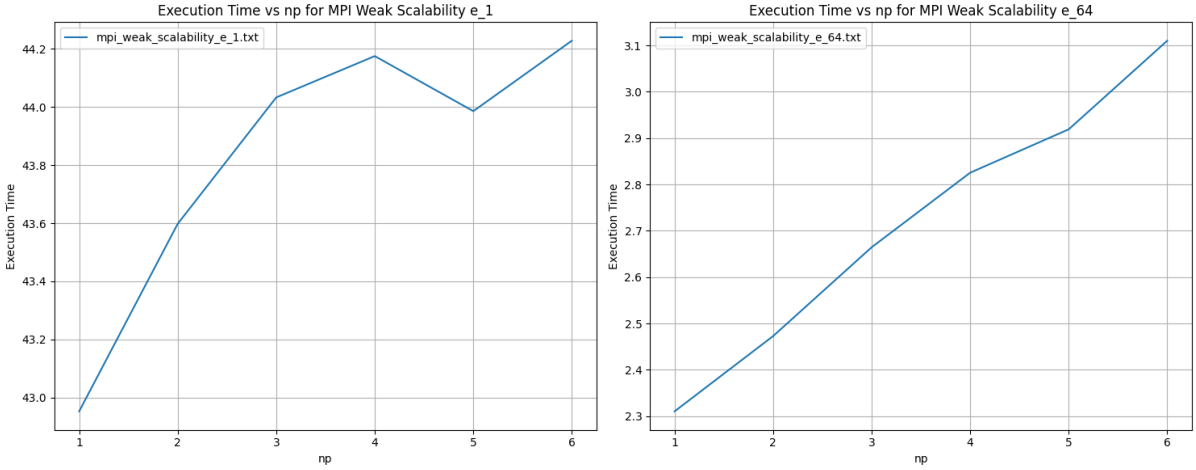


Figure 4: MPI Weak Scalability on EPYC nodes, for 1 and 64 threads

Figures 3: MPI weak scaling on THIN nodes using 1 thread and 12 threads, and Figure 4: MPI weak scaling on EPYC nodes using 1 thread and 64 threads display the outcomes.

1.4.3 MPI Strong Scalability

To examine MPI strong scalability, the grid size is kept constant, and the number of tasks increases with each run. Tests were conducted on various grid sizes, and on both THIN and EPYC nodes, using respectively 12 threads for THIN and 64 threads for EPYC, and then just one thread for both. In both cases, there is no increase in speedup with the growing size of the grids, likely due to an increase in message overhead.

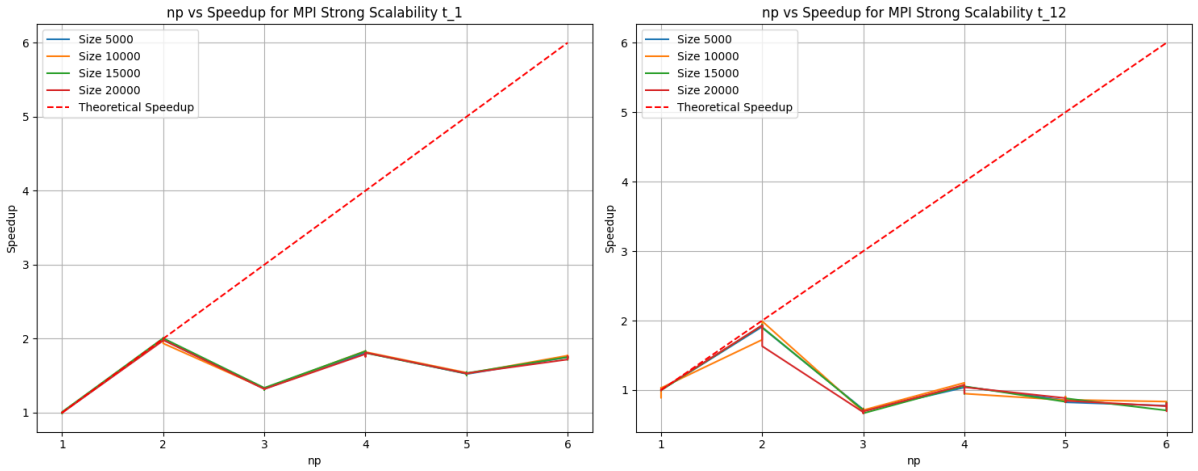


Figure 5: MPI Strong Scalability on THIN nodes, for different sizes of the initial grid

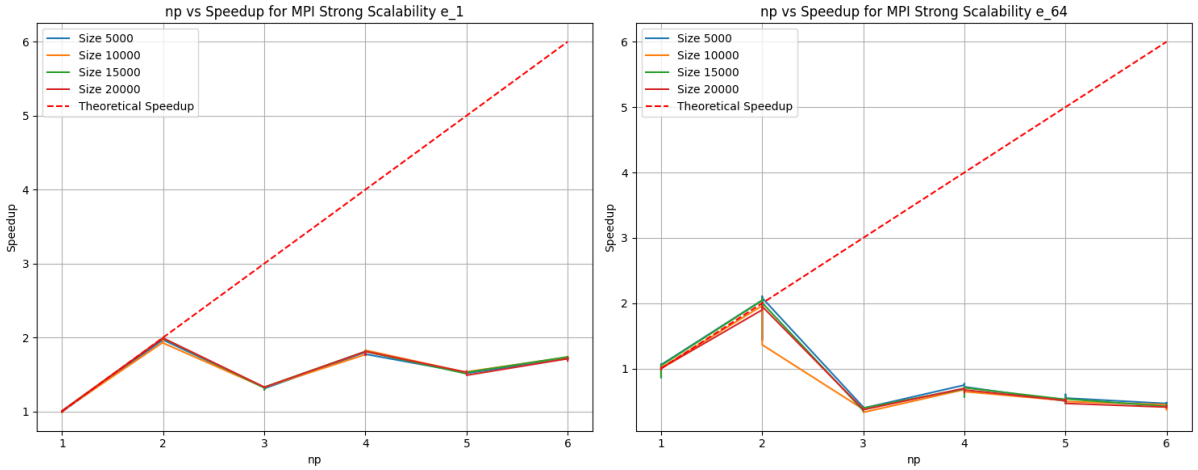


Figure 6: MPI Strong Scalability on EPYC nodes, for different sizes of the initial grid

Figures 5 displays MPI strong scaling using 1 thread and Figure 6 shows MPI strong scaling using 64 threads.

2 Exercise 2

2.1 Introduction

The goal of this exercise was to compare the performance of three libraries for the purpose of matrix-matrix multiplication. The libraries are available on HPC: MKL, OpenBLAS and BLIS. The performance had to be evaluated focusing the comparison on the on the level 3 BLAS function called `gemm`, that was implemented in single and double-point precision. The code provided `gemm.c` is a standard `gemm` code, where 3 matrices A, B and C are allocated, A and B are filled and the BLAS routine calculates the matrix-matrix product $C = A * B$. For the exercise, only square matrices were considered.

The comparison of the three different libraries had to take into consideration:

- Different architectures:
  - THIN
  - EPYC.
- Different thread allocation policies:
  - spread cores
  - close cores.
- Different precision:
  - single-point
  - double-point.

2.2 Approach

In order to compare the performance of the three different libraries, two different scalability studies were performed:

- Size scaling
- Core scaling.

Different measures had to be performed, and then the results were then compared to the Theoretical Peak Performance. Two files were previously provided to perform this experiment:

- The C code containing the `gemm` function (`gemm.c`) for the matrix-matrix multiplication with single or double precision and with either one of the different libraries.
- The `Makefile` to compile the `gemm.c`.

While MKL and OpenBLAS libraries were available on ORFEO's module's list, the BLIS library had to be downloaded and compiled by the user for every configuration needed.

Performances were measured by keeping track of the time and the number of GFLOPs required to complete the matrix-matrix multiplication for each configuration.

2.3 Implementation

Files `gemm.c` and `Makefile` that were previously made availabale were left unchanged, and the scalability study was performed on ORFEO architecture making use of some bash scripts that were modified at need. At the beginning, needed resources were allocated and `Makefile` was used to compiled the `gemm.c` file and create three different executables for each configuration: `gemm_oblas.x`, `gemm_mkl.x`, and `gemm_blis.x`. The compilation phase was repeated for each different configuration and also for single and double precision, modifing the `Makefile` by replacing the command `-DUSE_FLOAT` for single precision with `-DUSE_DOUBLE` for double precision.

Once all the executable files were available, corresponding output files were generated (`.csv`).

The output `.csv` files contained a line used for specifying the headers that were: $\$parameter$, $\$ExecutionTime$ and $\$GFLOPs$. To generate them, some batch scripts were written: half of them were written for the experiments with increasing matrix size, while the others contained instructions for operations with varying number of cores.

Files that were created in this way had this general structure: $\$scalability\_\$library.csv$ and to be differentiated, they were put into their corresponding folder at the time of creation. Folders followed this structure: $\$parameter(size/cores)/\$partition/\$\#threads/\$policy/\$precision/$.

The measurement process was conducted as follows. First, the BLIS library had to be configured and installed, a task repeated for each new node partition and core count setup considered. To analyze different core counts, we started with an EPYC node with 128 cores. The BLIS library was configured to match this setup, and all necessary measurements were carried out accordingly. Next, the library was then re-installed with a different setup (THIN 24 cores), and the measurements were repeated.

For the analysis of different sizes, we used EPYC 64 cores and THIN 12 cores architectures.

All measurements were then repeated leaving the `OMP_PLACES = cores` and changing the policy `OMP_PROC_BIND` from `close` to `spread`.

The reason for recompiling the BLIS library with various settings lies in its ability to produce optimized code specific to the hardware architecture during the compilation process. Hence, recompiling with different configurations for various architectures was deemed vital. To conduct the measurements with different setups, the batch script were modified. Once all the procedures were completed, the results of the measurements were available for the final analysis.

2.4 Results

Results have been plotted with the help of `Matplotlib` in Python, in such a way to compare the performances of the different libraries on the same task. The results have been visually compared to the Theoretical Peak Performances of each considered configuration on each processor. The Theoretical Peak Performance is measured in gflops as well and is calculate via the formula:

$$TPP = \#cores \cdot \frac{cycles}{s} \cdot \frac{\#FLOPS}{cycle}$$

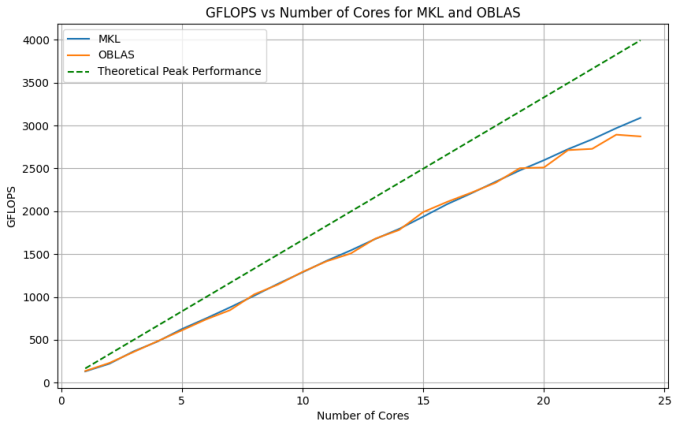
2.4.1 Strong Scalability (Cores changing)

The scalability over the increasing number of cores used for the calculation was verified for a maximum of 128 cores on EPYC partition and 24 cores for the THIN partition on ORFEO. The size of the matrix was fixed at 5000 x 5000. Each measure again was repeated for a couple of iterations and then the mean value was calculated for each considered core. The `OMP_PROC_BIND` policy was first set to `close` and then the experiment was repeated for `spread`.

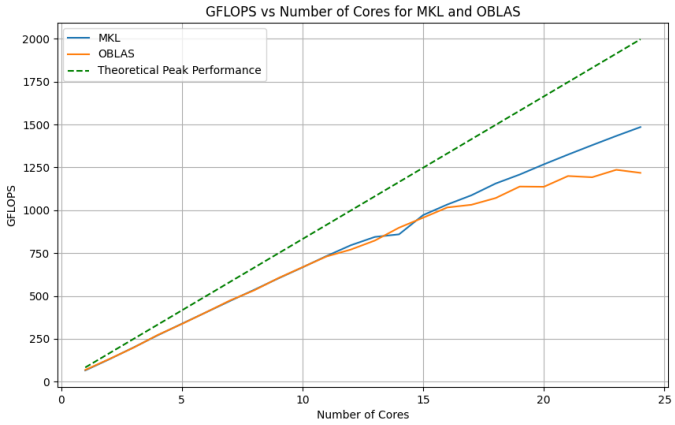
THIN

Single precision The analysis has been carried out using `OMP_PLACES = cores` and `OMP_NUM_THREADS = 24`, and for both `OMP_PROC_BIND = close` and `OMP_PROC_BIND = spread`, respectively.

Figure 7 shows the GFLOPS vs Number of Cores for MKL and OBLAS libraries, along with the theoretical peak performance.



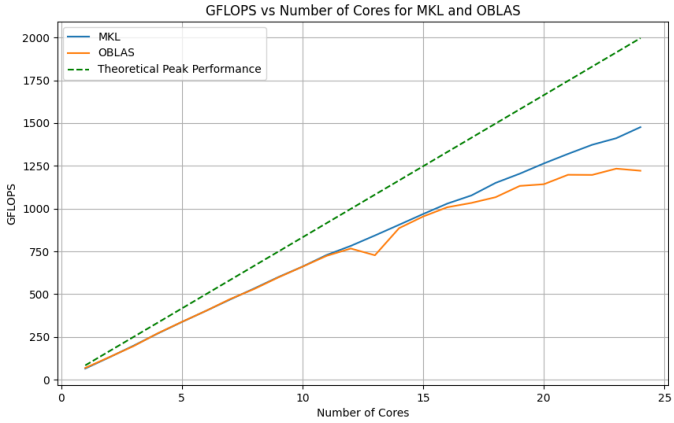
(a) Float precision, close policy



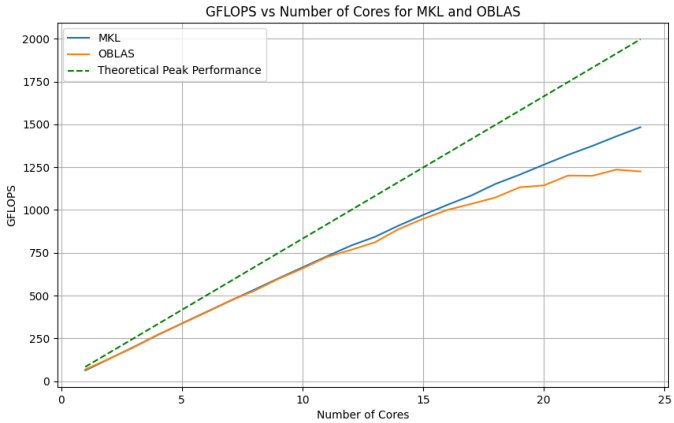
(b) Float precision, spread policy

Figure 7: Comparison of performance graphs for Core scaling, float precision, THIN 24 cores

Double precision Also in this case the parameters were set using `OMP_PLACES = cores` and `OMP_NUM_THREADS` = 24, and for both `OMP_PROC_BIND = close` and `OMP_PROC_BIND = spread`.



(a) Double precision, close policy



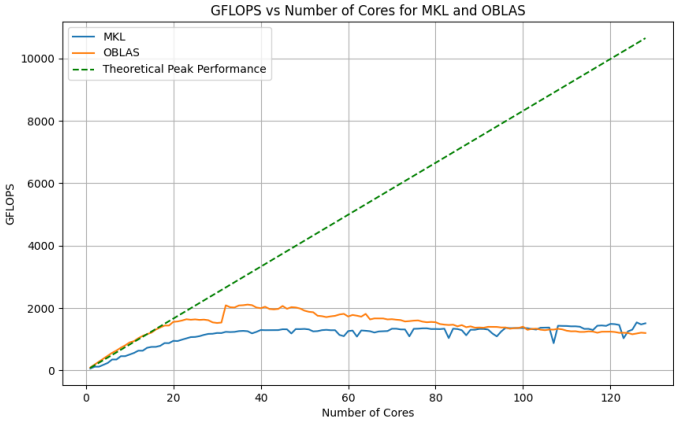
(b) Double precision, spread policy

Figure 8: Comparison of performance graphs for Core scaling, double precision, THIN 24 cores

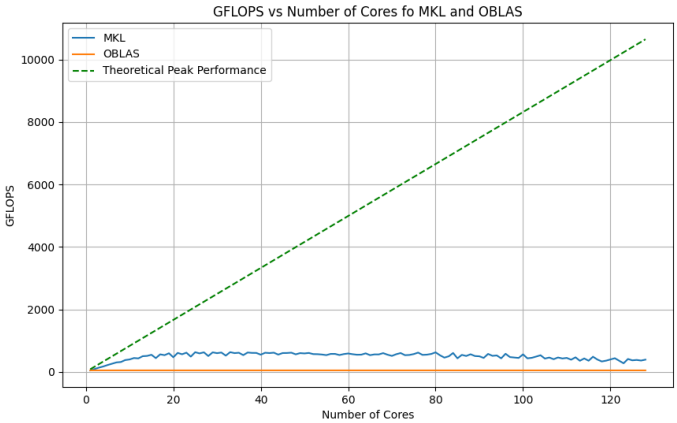
EPYC For what concerns the AMD EPYC 7H12 nodes, the base frequency is 2.6 GHz as well.

Single precision The analysis has been carried out using `OMP_PLACES = cores` and `OMP_NUM_THREADS = 128`, and for both `OMP_PROC_BIND = close` and `OMP_PROC_BIND = spread`, respectively.

Figure 9 shows the GFLOPS vs Number of Cores for MKL and OBLAS libraries, along with the theoretical peak performance.



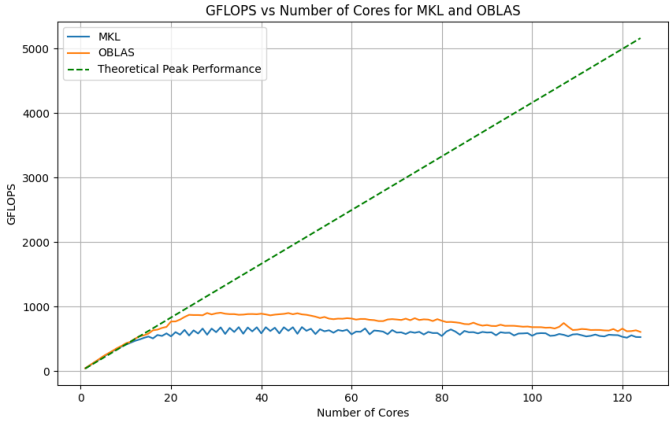
(a) Float precision, close policy



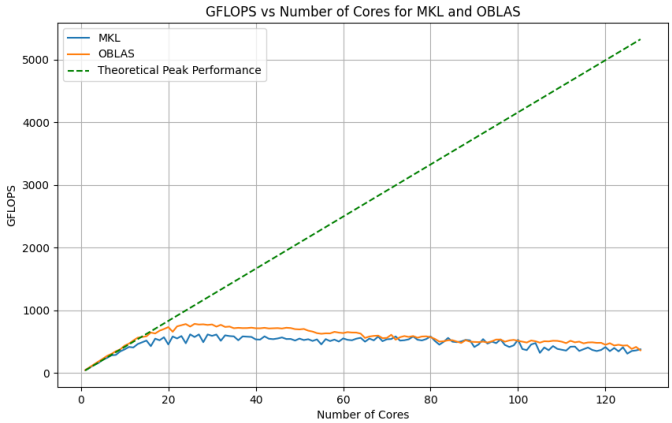
(b) Float precision, spread policy

Figure 9: Comparison of performance graphs for Core scaling, float precision, EPYC 128 cores

Double precision Also in this case the parameters were set using `OMP_PLACES = cores` and `OMP_NUM_THREADS =` 128, and for both `OMP_PROC_BIND = close` and `OMP_PROC_BIND = spread`.



(a) Double precision, close policy



(b) Double precision, spread policy

Figure 10: Comparison of performance graphs for Core scaling, double precision, EPYC 128 cores

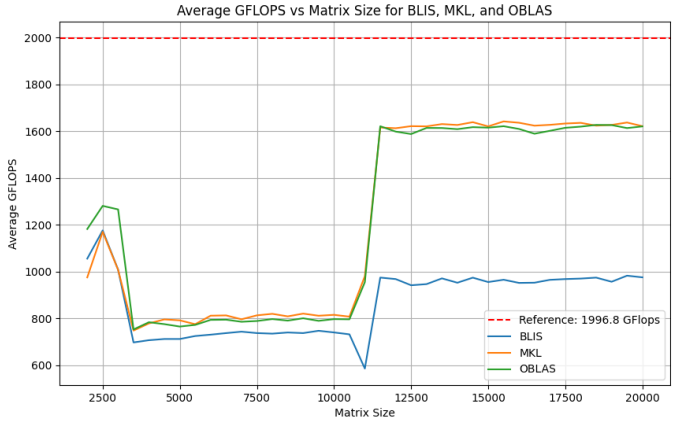
2.4.2 Weak Scalability (Size changing)

The scalability over the size of the calculated matrix was verified for 64 cores on EPYC partition and 12 cores for the THIN partition on ORFEO. The size was increased from 2000 x 2000 to 20000 x 20000 by 500. Each measure was repeated for a couple of iterations and then the mean value was calculated for each size. The `OMP_PROC_BIND` policy was first set to `close` and then the experiment was repeated for `spread`. For this kind of scalability, measures over BLIS library have also been considered.

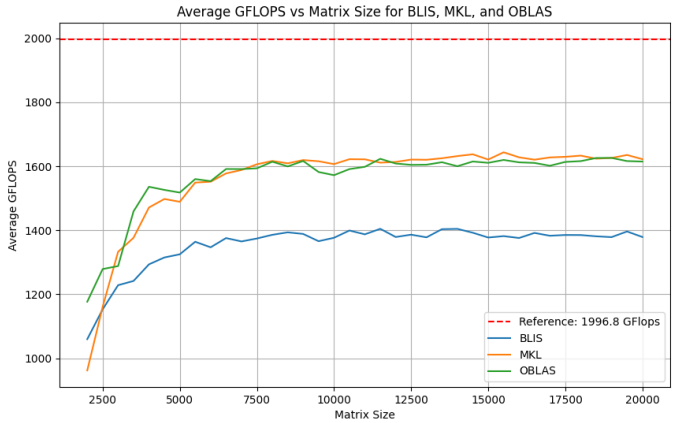
THIN For what concerns the Intel Xeon Gold 6126 THIN nodes, the base frequency is 2.6 GHz.

Single precision The TPP in this case is equal to 1996.8 GFLOPs per second , calculated $TPP = 12 \cdot 2.6GHz \cdot 64$. The analysis has been carried out using `OMP_PLACES = cores` and `OMP_NUM_THREADS = 12`, and for both `OMP_PROC_BIND = close` and `OMP_PROC_BIND = spread`, respectively.

Figure 11 shows the GFLOPS vs Number of Cores for BLIS, MKL, and OBLAS libraries, along with the theoretical peak performance.



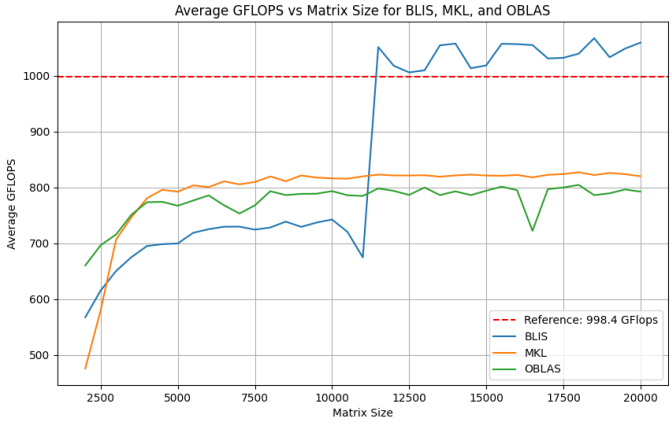
(a) Float precision, close policy



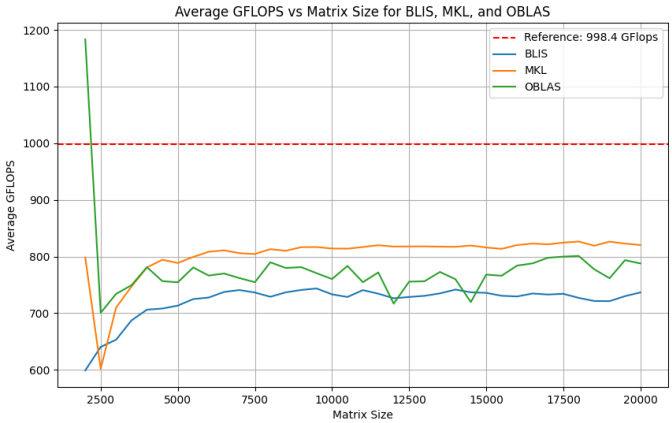
(b) Float precision, spread policy

Figure 11: Comparison of performance graph for Size scaling, float precision, THIN 12 cores

Double precision The TPP in this case is half of that for double precision, equal to 998.4 GFLOPs per second ($TPP = 12 \cdot 2.6GHz \cdot 32$). Also in this case the parameters were set using `OMP_PLACES = cores` and `OMP_NUM_THREADS =` 12, and for both `OMP_PROC_BIND = close` and `OMP_PROC_BIND = spread`.



(a) Double precision, close policy



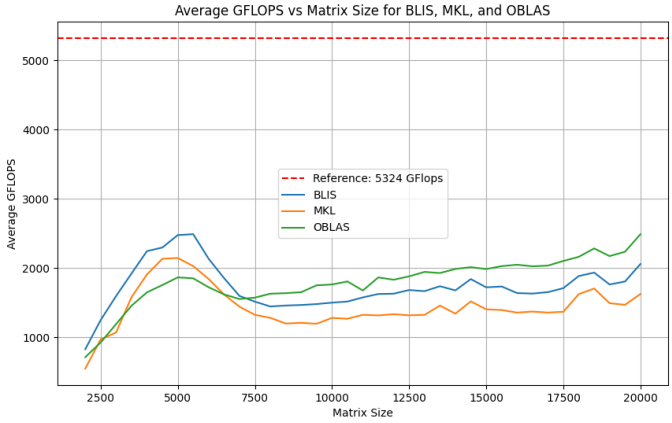
(b) Double precision, spread policy

Figure 12: Comparison of performance graphs for Size scaling, double precision, THIN 12 cores

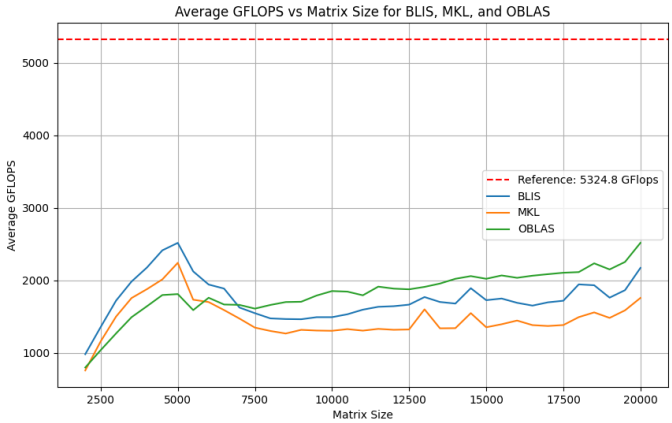
EPYC For what concerns the AMD EPYC 7H12 nodes, the base frequency is 2.6 GHz as well.

Single precision The TPP in this case is equal to 5324.8 GFLOPs per second ($TPP = 64 \cdot 2.6GHz \cdot 32$). The analysis has been carried out using `OMP_PLACES = cores` and `OMP_NUM_THREADS = 64`, and for both `OMP_PROC_BIND = close` and `OMP_PROC_BIND = spread`, respectively.

Figure **??** shows the GFLOPS vs Number of Cores for BLIS, MKL, and OBLAS libraries, along with the theoretical peak performance.



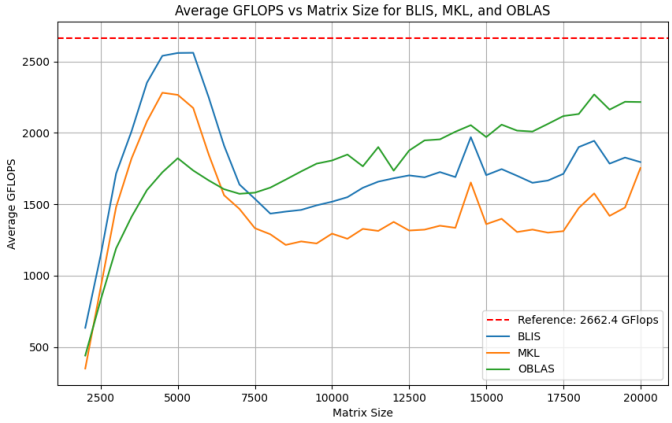
(a) Float precision, close policy



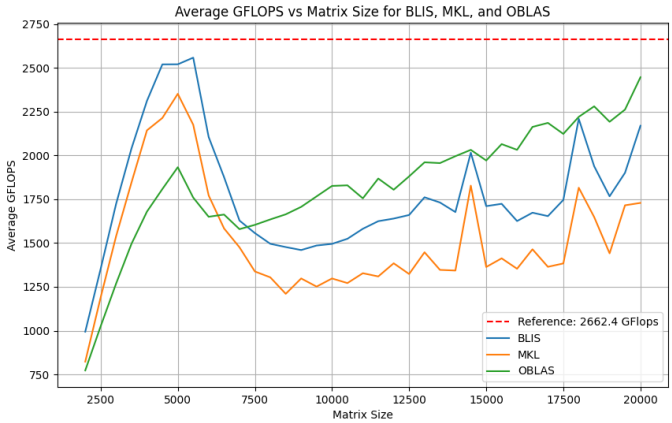
(b) Float precision, spread policy

Figure 13: Comparison of performance graphs for Size scaling, float precision, EPYC 64 cores

Double precision The TPP in this case is half of that for double precision, equal to 2662.4 GFLOPs per second ($TPP = 64 \cdot 2.6GHz \cdot 16$). Also in this case the parameters were set using `OMP_PLACES = cores` and `OMP_NUM_THREADS =` 64, and for both `OMP_PROC_BIND = close` and `OMP_PROC_BIND = spread`.



(a) Double precision, close policy



(b) Double precision, spread policy

Figure 14: Comparison of performance graphs for Size scaling, double precision, EPYC 64 cores

3 Bibliography