Final assignment[1] FHPC course 2022/2023

Student: Michele De Petris IN2000137

GitHub Repository
https://github.com/mikedepetris/Foundations_of_HPC_2022_final_project

Table of Contents

Exercise 1

Introduction

Implementation of a parallel version of a variant of the famous Conway's "game of life"[23], that is cellular automaton which plays autonomously on an infinite 2D grid.[4]
The playground is implemented as a square grid with periodic boundary conditions at the edges. Four evolution options have been implemented, that lead to different results, and require different parallel implementation, with hybrid MPI + OpenMP code. In each evolution kind, a different order is used to update cells.

1. Ordered: classic row-major, from left to right, by lines, from top to bottom.
2. Static: updating a new matrix, keeping the original untouched until all cells are updated.
3. Wave: square wave propagation from a new random point at each step.
4. White-black: static update of "white" positions first, and then "black" like in a chessboard.

Methodology

Each cell occupies a position in the bidimensional playground that we consider as the "world" where the cells may be either dead or alive. The world may be represented by a data matrix in memory, using data structures available in C language. As the cells may only have two possible statuses, its status only requires one bit. In this work a full byte is used instead, as managing the matrix of bits would be quite complex and would not add interesting points to this exercise. Of course, using "unsigned char *world" for the matrix allocates eight times the needed memory, so this choice severely limits the dimension of the possible playgrounds. The same reasoning can be done about the files in which the playgrounds are saved. Given the choice of the memory structure to hold the values, the cell statuses are defined as:

#define **DEAD** 255
#define **ALIVE** 0

Most graphic programs will display alive cells with black colour, and dead cells in white colour.
Several executions have been done to test the program, to verify the correct implementation. The following checks have been used:

- Well known patterns have been processed using the classic static evolution, checking that their behaviour follows what is expected for them. See appendix.
- Some start patterns generate a periodic sequence of identical cell distributions. Tools have been used to identify duplicates, see appendix where also animated videos are created, by first deleting all dupes and keeping the initial frames of the sequence. See appendix.
- Binary comparison has been used to check that identical results are obtained, as output files, when the execution is done using a different number of processors, changing the number of MPI processes or OpenMP threads. See appendix.
- Debugging code has been added to print useful information about the execution, making it possible to check the correctness of the processing, and to identify the locations of possible problems. See appendix about preprocessing directives.

A "-q" option can be passed on the command line, to obtain a timing output suitable for a .csv file. In this way all other output of the program is suppressed. The obtained .csv files contain all

the information about the execution parameters and timing results. The files can then be directly used to draw graphs and calculate performance speed up.
Reading and writing to image files has been implemented in a way that can execute in parallel MPI processes.
Timing has been recorded. As most of the I/O operations happens in parallel, separate I/O time durations have been collected for the file reading and writing.
A long series of executions have been run on Orfeo nodes, on a range of world sizes, number of iterations, number of MPI processes, OpenMP threads, writing image file snapshots at each iteration or only the result.
Gathered data have been analysed computing the values average on multiple executions at the same conditions and finding a way to exclude "outliers" that may have occurred due to some system external perturbation. Standard deviation has been used to limit a range of acceptable values.
Speed up has been computed for the meaningful situations and graph charts made to show the scalability behaviours and compare them. Speed up is computed as Ts/Tnp, with Ts representing the time taken by a single process to complete the execution and Tnp is the time taken by n processes to complete the same task. On the graph the ideal speed up is drawn as a reference line, which represents the maximum theoretical achievement, that is a value equal to n, that is the number of processes.

The whole process to complete the exercise can be summarized as the following steps:

1. Design on paper with few simplified attempts, gathering documentation.
2. First version of the program that initializes a new random pattern.
3. Implementation of the ordered evolution.
4. Selection of well-known patterns, saved as files that can be read from the program.
5. Test of the ordered evolution on the known patterns.
6. Implementation of other evolutions.
7. Evolution runs of test cases with verification of the expected behaviour and output, using file binary comparison tools.
8. Execution with different levels of information logging, to spot problems and track flows.
9. Execution on different number of processors, with different parameters, to cover most common situations, and validation of the program output.
10. Design of the scalability studies.
11. Batch files implementation for the scalability execution.
12. Extensive batch executions on Orfeo.
13. Results analysis, graph charts drawing and report writing.

Implementation

The C language has been used to make a program that can build the needed data structure and process data for the give cases. A CLion project has been created to locally debug in a ubuntu linux environment. While developing the program has been tested on several machines with different processor, all with ubuntu installations, native or within the wls Windows Linux Subsystem. OpenMPI and OpenMP worked on the local systems even if the available cores in the CPUs was only 4. The local setup allowed a quick cycle of test and debug, to complete the

development before moving to Orfeo. Files were then transferred by ftp to update sources and output files.
The code is duplicated where possible for each kind of evolution, avoiding the use of reusable functions to pursue the best performance, but keeping a clear structure that allows to easily compare similar modules, spotting the differences by visual or automated comparison.
For the same reason long code lines are preferred, this allows a better visual file comparison when comparing code side by side or when analysing multiple modules search results to keep the modules aligned when making changes to code that is duplicated. It is in fact easy to reformat to the usual world wrapping in the code editor. When splitting code on multiple lines, commas or operators are kept at the line beginning to better understand the code is the prosecution of the previous line.

The project contains the following files:

Makefile	make file for mpicc
CMakeLists.txt	make file for CLion CMake
files_io.h	header with shared definitions for files module
files_io.c	file reading and writing
gameoflife.h	header with shared definitions
gameoflife.c	main code that parses the input and executes the operations
new_playground.c	initialization of a new playground with random values
evolution_ordered.c	-e0 ordered evolution
evolution_static.c	-e1 static evolution
evolution_wave.c	-e2 wave evolution
evolution_whiteblack.c	-e3 white-black evolution

Command line arguments parsing

Several switches can be passed to the program on the command line, calling the program omitting the arguments, or passing “-h” wil make the program print help informations:

```
Usage: /mnt/c/dev/HPC/gameoflife/cmake-build-debug/gameoflife.x [options]
      -i  initialize a playground
      -r  run a playground (needs -f)
      -k  <num> playground size (default value 10000, minimum value 100)
      -e  [0|1|2|3] evolution type (0: ordered, 1: static, 2: wave, 3: white-black
      -f  <string> filename to be written (new_playground) or read (run)
      -n  <num> number of steps to be iterated (default value 100, max 99999)
      -s  <num> number of steps between two dumps (default 0: print only last state)
      -h  print help about program usage
      -q  print csv output
      -D  print debug informations
      -D2 print advanced debug informations

Examples:
    to create initial conditions:
        gameoflife.x -i -k 10000 -f initial_condition
    this produces a file named initial_condition.pgm that contains a playground
    of size 10000x10000
    to run a play ground:
        gameoflife.x -r -f initial_condition.pgm -n 10000 -e 1 -s 1
    this evolves the initial status of the file initial_condition.pgm for 10000 steps
    with the static evolution mode, producing a snapshot at every time-step

Parallel execution:
    mpirun -np 1 gameoflife.x -i -k 100 -f pattern_random
    mpirun -np 4 gameoflife.x -r -f pattern_random -n 3 -e 1 -s 0
```

The input requirements and error situations are specifically managed by the code:

```
    if (world_size < MIN_WORLD_SIZE) {
        printf("Value %ld is too low to be passed as -k <num> "
                "playground size, the minimum value is %d\n"
                , world_size, MIN_WORLD_SIZE);
        perror("Value is too low to be passed as -k <num> "
                "playground size\n");
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }
```

The “-D” switch activates various debugging output levels, while on the opposite “-q” suppresses most of the output to enable the csv comma separated values output of the collected measurements.

The requirements about the file format specifies that the snapshot file name should be:

    snapshot_nnnnn

where nnnnn (with 5 digits, padded with zeros) is the time-step it refers to. To comply to it a check is made:

```
// max iterations, must print as nnnnn
#define MAX_NUMBER_OF_STEPS 99999
if (number_of_steps > MAX_NUMBER_OF_STEPS) {
     printf("Value %d is too big to be passed as -n <num> "
          "number of steps to be iterated, max admitted value "
          "is %d\n", number_of_steps, MAX_NUMBER_OF_STEPS);
     perror("Value is too big to be passed as -n <num> "
          "number of steps to be iterated\n");
     MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
}
```

Hybrid

The initializatin of the MPI library is done using the "funneled" mode, with a specific call in place of MPI_Init:

```
MPI_Status mpi_status;
MPI_Request mpi_request;
int mpi_provided_thread_level;
MPI_Init_thread(argc, argv, MPI_THREAD_FUNNELED
     , &mpi_provided_thread_level);
if (mpi_provided_thread_level < MPI_THREAD_FUNNELED) {
     perror("a problem occurred asking for MPI_THREAD_FUNNELED"
          level\n");
     MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
}
int mpi_rank, mpi_size;
MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
```

Broadcast file paths and names

When the master process concatenates strings to produce filenames and paths, the result is broadcasted to all other processes by MPI_Bcast, sending first the string length, followed by the actual text:

```
// Broadcast the string to all other processes
MPI_Bcast(&directoryname_len, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (mpi_rank > 0)
     directoryname = malloc(directoryname_len);
     MPI_Bcast(directoryname, directoryname_len, MPI_CHAR, 0
          , MPI_COMM_WORLD);
```

Time recording

Timing of the I/O operations is recorded inside functions using the MPI functions and returned to the calling code:

```
double t_time_point = MPI_Wtime();
...
return MPI_Wtime() - t_time_point;
```

Then the caller sums the timing of all the I/O operations, accumulating in a separate variable the time used by all the non-zero rank processes:

```
double t_io = 0; // total I/O time spent
// total I/O time spent by processes > 0
double t_io_accumulator = 0;
// start time t_io += file_chunk_merge(
double t_start = MPI_Wtime();
```

All the timing values are then written to the csv output:

```
printf("e2,%ld,%d,%d,%d,%d,%f,%f,%f,%f\n"
     , world_size
     , number_of_steps
     , number_of_steps_between_file_dumps
     , mpi_size
     , omp_get_max_threads()
     , MPI_Wtime() - t_start
     , t_io
     , t_io_accumulator
     , mpi_size == 1 ? 0 : t_io_accumulator / (mpi_size - 1));
```

Having obtained all the measurements, it can then be checked that the I/O time spent by rank zero process is always longer than the average of all other processes average, obtained dividing the accumulated sum by the number of those processes.

Matrix in memory

The status value of each cell of the world is kept in memory as an "unsigned char". The values are organized in an array. The C language specifications state that matrices are laid out in memory in a row-major order: the elements of the first row are laid out consecutively in memory, followed by the elements of the second row, and so on. Each row has the number of elements that is considered the "world size". The actual size of the square matrix is the power of two of the "world size".

Domain decomposition

Given the world size that is the square matrix columns and rows number, the chunk of data processed by each MPI process is computed dividing the matrix in rows slices, keeping the world size for the columns number, and obtaining the local size as number of rows, dividing the world size by the number of the MPI processes. As the world size is not always divisible by that number, some slices will have one row more, to manage the remainders of the divisions.

```
*local_size = world_size % mpi_size - mpi_rank <= 0
 ? (long) (world_size / mpi_size)
 : (long) (world_size / mpi_size) + 1;
```

Where needed the local size takes into account two additional "ghost rows" that are used to enable processing of the evolution by each process on its own slice of data, as the rules need to evaluate the status of the neighbouring even for the first and last row of cells, and those neighbour cells are located in the last row of the previous slice and the first row of the next slice.
Cells neighbours of the first column on the left and last column on the right are in the same slice, so each process has just to take in account to go to the other side of the world to manage this simplified infinity of the world.
Border line situations with small data side and many processors are avoided, as those are not interesting for the experiments:

```
if (mpi_size > world_size || local_size == 0) {
     perror("ERROR: wrong situation with more processes "
          "than domain decomposition slices\n");
     MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
}
```

Cell updating

At iteration step, the evolution rules are applied to each cell to determine its next state. The rules consider the states of the neighbouring of the cell, that are the eight cells all around.
At first, coordinates are calculated for the cell position, given the "i" index that iterates the matrix, by integer division to obtain the y as the vertical position (row), and taking the reminder as the x (column):

```
long x = i % world_size;
long y = i / world_size;
```

Given these cell coordinates, the eight neighbours of the actual cell have its coordinates that span from the previous cell to the next, both horizontally and vertically:

```
long x_prev = x - 1 >= 0 ? x - 1 : world_size - 1;
long x_next = x + 1 < world_size ? x + 1 : 0;
long y_prev = y - 1;
long y_next = y + 1;
```

Here we can see how infinity is managed on the vertical borders, when the x coordinate is zero, the cell is the first in the row, so the previous position is not "x – 1" but it's "world_size – 1" that is the coordinate of the last cell in the row, on the right, that we consider contiguous with the first one, on the left side.
Given all the coordinates, all the values of the neighbour cells are summed up to determine the next cell's status, dead or alive.

Reading and writing to chunk files and merging

Reading and writing operations to image files is done in parallel.

MPI I/O

The reading and writing operations to the image or chunk files is done using specific MPI calls:

```
MPI_File fh;
MPI_File_open(MPI_COMM_WORLD, image_filename, MPI_MODE_RDONLY
     , MPI_INFO_NULL, &fh);
unsigned int to_read_size =
     (*world_size) * (*local_size) * color_depth;
MPI_File_seek(fh
     , (int) number_of_read_chars + 1 + first_row * *world_size
     , MPI_SEEK_CUR);
if ((*world = (unsigned char *) malloc(
     (*local_size + 2) * (*world_size) * sizeof(unsigned char)))
      == NULL)
          <error handling>
unsigned char *v = *world;
MPI_File_read(fh, &v[*world_size], (int) to_read_size
     , MPI_UNSIGNED_CHAR, &mpi_status);
MPI_File_close(&fh);
```

The “file_chunk_merge” function implements the merging of the chunk files into the image file following the correct sequence:

```
f1 = fopen(filename1, "a");
if (f1 == NULL) {
     printf("Error joining file chunks: file=%s\n", filename1);
     perror("Error joining file chunks\n");
     MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
}
f2 = fopen(filename2, "r");
if (f2 == NULL) {
     printf("Error joining file chunks: file=%s\n", filename2);
     perror("Error joining file chunks\n");
     MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
}
size_t read_size;
unsigned char buffer[4096]; // 4k buffer
do {
     read_size = fread(buffer, 1, sizeof(buffer), f2);
     if (read_size <= 0) break;
          fwrite(buffer, read_size, 1, f1);
} while (read_size == sizeof buffer);
     // if it was a full buffer, loop again
fclose(f1);
fclose(f2);
```

File reading

When reading, each MPI task accesses it's portion of data. In the function *"file_pgm_read()"*, the master process reads the image file header that contains the image characteristics like the rows and columns sizes and colour depth. Those values are shared with all other MPI processes using the *"MPI_Bcast()"* function. At this point each process can calculate the file chunk that has to be read, points to that chunk using *"MPI_File_seek()"*, allocates memory and reads data:

```
unsigned int to_read_size = (*world_size) * (*local_size) * color_depth;
MPI_File_seek(fh, (int) number_of_read_chars + 1 + first_row * *world_size, MPI_SEEK_CUR);
if ((*world = (unsigned char *) malloc(
(*local_size + 2) * (*world_size) * sizeof(unsigned char))) == NULL)
        <error handling>
MPI_File_read(fh, &v[*world_size], (int) to_read_size, MPI_UNSIGNED_CHAR, &mpi_status);
```

There are two implementations of the function, one allocates space for two additional "ghost" rows that are used for most of the evolutions. Another function with a name suffix "_noghost" implements the same operations but does not allocate the additional rows, and is used by the wave evolution, that do not process data in separate slices.

File writing

Each process writes to a separate file with a specific naming that contains the chunk number, that is the MPI rank number and the iteration step number. After saving all the chunks for an iteration, all those files that have the ".pgmpart" extension are merged in the final ".pgm" image file.
All files written during an execution of an evolution run are saved to a new directory that contains the name of the input file, the name of the evolution type, the number of steps, the number of MPI processes used and a timestamp that identifies the time instant of the execution.

```
pattern_random1000.pgm_ordered_00003_001_2023-06-05_20_19_31\
pattern_random1000.pgm_ordered_00003_128_2023-06-04_10_03_48\
pattern_random.pgm_whiteblack_00003_128_2023-06-04_09_50_31\
pattern_random.pgm_static_00003_128_2023-06-04_09_50_05\
pattern_random.pgm_ordered_00003_128_2023-06-04_09_49_53\
pattern_random.pgm_whiteblack_00003_001_2023-06-04_09_41_12\
pattern_random.pgm_wave_00003_001_2023-06-04_09_41_07\
pattern_random.pgm_static_00003_001_2023-06-04_09_41_00\
pattern_random.pgm_ordered_00003_001_2023-06-04_09_40_51\
pattern_random.pgm_ordered_00003_001_2023-06-04_09_40_29\
patterns_random_pattern_random5.pgm_ordered_00011_128_2023-06-02_21_13_17\
patterns_random_pattern_random5.pgm_ordered_00011_006_2023-06-02_21_12_59\
patterns_random_pattern_random5.pgm_ordered_00011_005_2023-06-02_21_12_51\
```

There are corresponding writing functions with the "_noghost" suffix that do the same operations without the ghost row extra allocation, those functions are used by the wave evolution.

New playground with random values

The operation is done in parallel by MPI processes that manage each a slice of the data, as explained in the domain decomposition description. Each process then uses OpenMP static parallelization of the loop that generates random values assigned to the cells.

Each process saves to a partial file. MPI process of rank zero writes the first chunk with the image file header, then appends all the other files in the correct sequence to generate the desired output.

Ordered evolution

The cells are to be processed in classic row-major order, from left to right of the matrix, by lines, from top to bottom. Each MPI process manages a slice of the matrix with a size that is obtained by dividing the number of rows by the number of MPI processes as already described about the domain decomposition approach.
After the matrix slice is read as a chunk from the image file, before beginning the evolution iterations, the last process (rank mpi_size - 1) sends its last row to the first process (rank 0) that uses it as the needed first ghost row.
At the beginning of the iteration cycle, all processes except the first send their first row to the previous process by a non-blocking *MPI_Isend* call. Then the last process waits to receive its first ghost row from the previous process and its last ghost row from the first process (rank 0). In the same way the first process waits to receive its first ghost row from the last one, and its last ghost row from the first. All other processes, that are not the first or the last, wait to receive their first ghost row from the previous (rank - 1) and the last ghost row from next (rank + 1).
At this point each process will update the cell statuses in sequence, iterating over the matrix index, that means starting from the upper left position, by rows, to the bottom right. The sequence from the first process to the last one is determined by the send/receive mechanism, that makes every process wait for the previous one to complete its updating.
After the updating, to complete each iteration cycle, each process sends its last row to the next process, except for the last one, that completes the updating.

Static evolution

MPI parallelization is obtained by domain decomposition of the world matrix. Each process manages a slice of the matrix that is obtained by dividing the number of rows by the number of MPI processes as already described, plus two “ghost rows”.
A new array is allocated by each process, of the same size of its world slice. The processing uses the data of the world slice to update the new array, keeping the original data untouched until all cells are updated.
At the end of each iteration, each process sends its first row to the previous process that will use it as its last “ghost row”. In the same way the last row is sent to the next process. This means that each process must wait to receive the two ghost rows form next and previous processes before starting the evolution data processing.
The data is sent by the non-blocking MPI call, using unique tag numbers to identify the data, so that each process can receive the correct block of data even when other processes are completing their iterations at different speed. The first top and the last bottom chunks of data has to be processed in a separate way, as they are to be considered contiguous in the infinite world.

```
int tag_0 = 2 * iteration_step;
int tag_1 = 2 * iteration_step + 1;

MPI_Isend(&w[world_size], (int) world_size
     , MPI_UNSIGNED_CHAR, mpi_rank - 1, tag_0,...

MPI_Recv(&w [(local_size + 1) * world_size], (int) world_size
     , MPI_UNSIGNED_CHAR, mpi_rank + 1, tag_0,...
```

After each process has its data available for processing, a cell updating function is called for each of the cell. The iteration could follow any order, as the new state is computed by considering the previous state only, so in this kind of evolution it's better to choose the order that may maximize performance. The matrix values are stored in a row-major ordered array, so the iteration is implemented as a "for loop" that scans the array from first to last element, using OpenMP that further parallelizes operations inside each MPI process. The workload is balanced so the default static scheduling is used. After some experimenting with several chunk sizes, the default size of 1 is used, as no evidence of performance improvement appeared.

To process the new cell statuses of the next iteration, instead of copying back all values to the original array, a pointers swap is done to reuse allocated "world_local" and "world_local_next", that are the pointers to the original data chunk of the MPI process, and the new one, needed for the static evolution:

```
unsigned char *temp = world_local_actual;
world_local_actual = world_local_next;
world_local_next = temp;
```

At the end of the evolution iteration the final output must be written to a file. It must be considered that when the number of iteration steps is odd, the pointer is set to the new array, that is lost exiting the function, so all data must be copied back:

```
if (number_of_steps % 2 == 1)
    for (long long i = world_size; i < world_size * (local_size + 1); i++)
        world_local_next[i] = world_local_actual[i];
```

Wave evolution

This evolution is implemented as a square wave propagation from a new random point at each step. It is possible to force the program to use the same point at each iteration while debugging, to be able to reproduce the same experiment and compare the output to check for correctness. Iterations must be done by one single process so rank zero process gathers all the data from the others in the function "evolution_wave_parallel":

```
    if (mpi_rank != 0) {
         // other processes send matrix to process 0
         MPI_Isend(world_local, (int) (local_size * world_size)
              , MPI_UNSIGNED_CHAR, 0, 0, MPI_COMM_WORLD, mpi_request);
    } else {
         // process 0 allocates the full matrix
         // and receives data from all other processes
         world = (unsigned char *) malloc(
              world_size * world_size * sizeof(unsigned char));
         // copy values from process zero matrix
         for (long long i = 0; i < local_size * world_size; i++)
              world[i] = world_local[i];
         // receive all other matrix data
         for (int i = 1; i < mpi_size; i++) {
              size = world_size % mpi_size - i <= 0
                   ? (long) (world_size / mpi_size)
                   : (long) (world_size / mpi_size) + 1;
              long start = local_size * world_size
                   + (i - 1) * size * world_size;
              MPI_Recv(&world[start], (int) (size * world_size)
                   , MPI_UNSIGNED_CHAR, i, 0
                   , MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

After that, all processes iterate, but only rank zero process does the update and sends results to other processes, while processes with rank > 0 receive back the local chunk to save it to file when data is available.
At each iteration a random point of the world is chosen and passed to the function that does the cell updating:

```
    x = rand() % world_size;
    y = rand() % world_size;
    set_dead_or_alive_wave_single(world, world_size, iteration_step
         , x, y, debug_info);
```

The updating iterates enlarging the square wave starting from 1 cell to squares of size 3, 5, 7... up to world size.

```
for (long square_size = 1; square_size <= world_size
     ; square_size += 2) {
     // integer division: 1/2=0 3/2=1 5/2=2 7/2=3...
     long half_size = square_size / 2;
     // horizontal and vertical iterations
     // from -half_size to half_size
     // 1:1, 3:-1 0 1, 5: -2 -1 0 1 2,...
     // 1: start from upper left to upper right (constant y)
     // cross boundary if exceeding
     y = (startY + half_size) % world_size;
     // upper horizontal line from left to right
     for (long i = startX - half_size; i <= startX + half_size
          ; i++) {
          // actual cell x coordinate
          // , crossing boundary if exceeding
          x = (i + world_size) % world_size;
          // neighbours of actual cell
          x_prev = x - 1 >= 0 ? x - 1 : world_size - 1;
          x_next = x + 1 < world_size ? x + 1 : 0;
          y_prev = y - 1 >= 0 ? y - 1 : world_size - 1;
          y_next = y + 1 < world_size ? y + 1 : 0;
```

At this point the code can determine the number of dead neighbours and update the cells, following the four sides of the square. The code is quite long as the operations are repeated four times moving the coordinates counterclockwise over the four sides of the square.

White-black evolution

The order followed to update cells is similar to the static evolution, but with the update of "white" positions first, and then "black" like in a chessboard. Most of the magic is done inside the iteration loop by the following code, that simply repeats two time what is done for the static case, first for white, then for black cells, with the same pointer swapping to reuse the allocated arrays:

```
set_dead_or_alive_white_parallel(mpi_rank, mpi_size, mpi_status
     , mpi_request, world_local_actual, world_local_next
     , world_size, local_size, iteration_step);
// pointers swap to reuse allocated world_local
// and world_local_next for next iteration
temp = world_local_actual;
world_local_actual = world_local_next;
world_local_next = temp;
set_dead_or_alive_black_parallel(mpi_rank, mpi_size, mpi_status
     , mpi_request, world_local_actual, world_local_next
     , world_size, local_size, iteration_step);
temp = world_local_actual;
world_local_actual = world_local_next;
world_local_next = temp;
```

Inside the "set_dead_or_alive_*" functions, cells are updated skipping even or odd cells in each case:

```
// update even cells, white on the chessboard (cell 0,0 is white)
#pragma omp for
     for (long long i = world_size + 1
          ; i < world_size * (local_size + 1); i += 2) {
          // copy previous even white cell
          world_next[i - 1] = world_local[i - 1];
          set_dead_or_alive_cell(world_local, world_next
               , world_size, i);
```

Results & Discussion

A very long series of executions have been run on the Epyc and Thin Orfeo nodes to check the correctness of the program and the behaviour in the situations object of study. More than 500 files have been obtained, two for each execution, one log file and one csv data file, and around 100 of those files have been discarded, requiring corrections and re-executions.

At each stage, output has been analysed to understand if it was matching what was expected. In case of unexpected behaviour or output results, the program has been corrected and re-executed. In many cases the execution had to be repeated with different parameters or setting. Some nasty problems require long debugging sessions. The behaviour has been analysed mostly comparing the execution timings. A total execution time has been recorded, together with the time spent in I/O operations by the master MPI process (rank 0) and the overall I/O time of the other processes, saved in an accumulator, that also gives an average, dividing the sum by the number of processes. In most cases the overall time is much bigger then the I/O time, and I/O time of the master process bigger than the mean of other processes. The overhead of the code that manages the timers has not been taken in consideration, as it would only add knowledge about the exact timings, but not about the program behaviour.

Ordered evolution

In the ordered evolution the problem is serial, so there is no way to parallelize the workload of the cell processing and updating. Reading and writing concurrently to the image partial files lets the program operate on a matrix data structure bigger than what could be possible on a single node, as each MPI process can keep its own slice of data using all the memory available for its node. With the achievement of memory size scaling, the total size of the problem that can be managed by the program depends on the number of nodes available and is not limited by the memory size of the single nodes.
This is the only evolution that gives the classic and well know results for the game of life. Its results have been checked against the known and popular patterns to verify the program correctness.

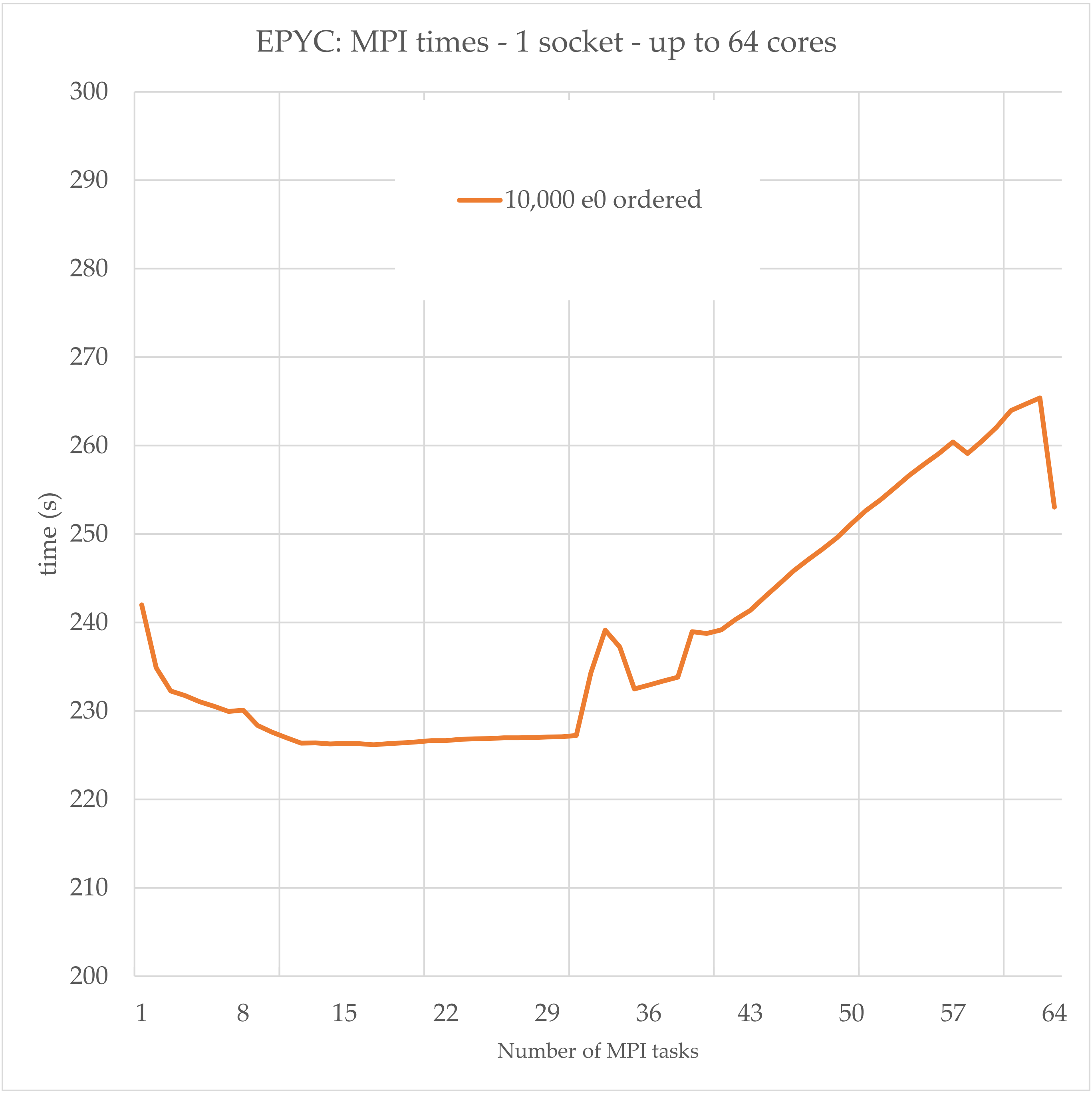


Figure 1: "Ordered" times on Epyc – 1 socket – up to 64 cores.

Static evolution

This is the most interesting evolution for the scalability studies, so executions have been done to study the OpenMP, strong and weak MPI scalability.

OpenMP scalability

To analyse the program behaviour with a different number of OpenMP threads, a bash shell script is used to set up a specific environment with a single MPI process on a socket. A cycle increases the number of threads from 1 up to the number of cores present on the socket:

```
export OMP_PLACES=cores
export OMP_PROC_BIND=close
for threads in {1..64}; do
export OMP_NUM_THREADS=$threads
mpirun -n 1 --map-by socket gameoflife.x -r ...
```

On the Epyc nodes the cycle goes from 1 to 64.
Sometimes to speed up the debugging and testing process, the reversed order has been used `{64..1}` as the computation goes faster if the program does scale, and few results were enough to quickly check the effect of the program corrections and modifications. In this way the batch execution could be stopped by "`scancel <id>`" and relaunched after adjustments, without waiting for the full execution cycle, discarding the partial results.
Running the program on more sockets, with a single MPI process on each socket, requires a different command line:

```
mpirun -n <number of sockets> --map-by node --bind-to socket
```

A maximum number of 6 sockets were available on the Epyc nodes and 4 on Thin.
A range of sizes as been used for the tests. Smaller sizes at the beginning, for the correctness validation. Then bigger sizes to analyse the scaling behaviour. As seen for the number of threads, sometimes it was more convenient to start from smaller sizes, for example to check if a bug was fixed or a new functionality worked as expected. Other times it was more interesting to have as soon as possible a measurement of the time taken processing bigger sizes. For this reason, the scrips were frequently adapted along the testing:

```
for SIZE in 100 1000; do
for SIZE in 10000 1000 100; do
for SIZE in 30000 25000 20000 15000 10000; do
```

The number of iteration step for the evolution is kept fixed at 100 in this way:

```
STEPS=100
SNAPAT=0
TYPE=1 # STATIC
gameoflife.x -r -f pattern_random$SIZE.pgm -n $STEPS -e "$TYPE" -s
"$SNAPAT" -q
```

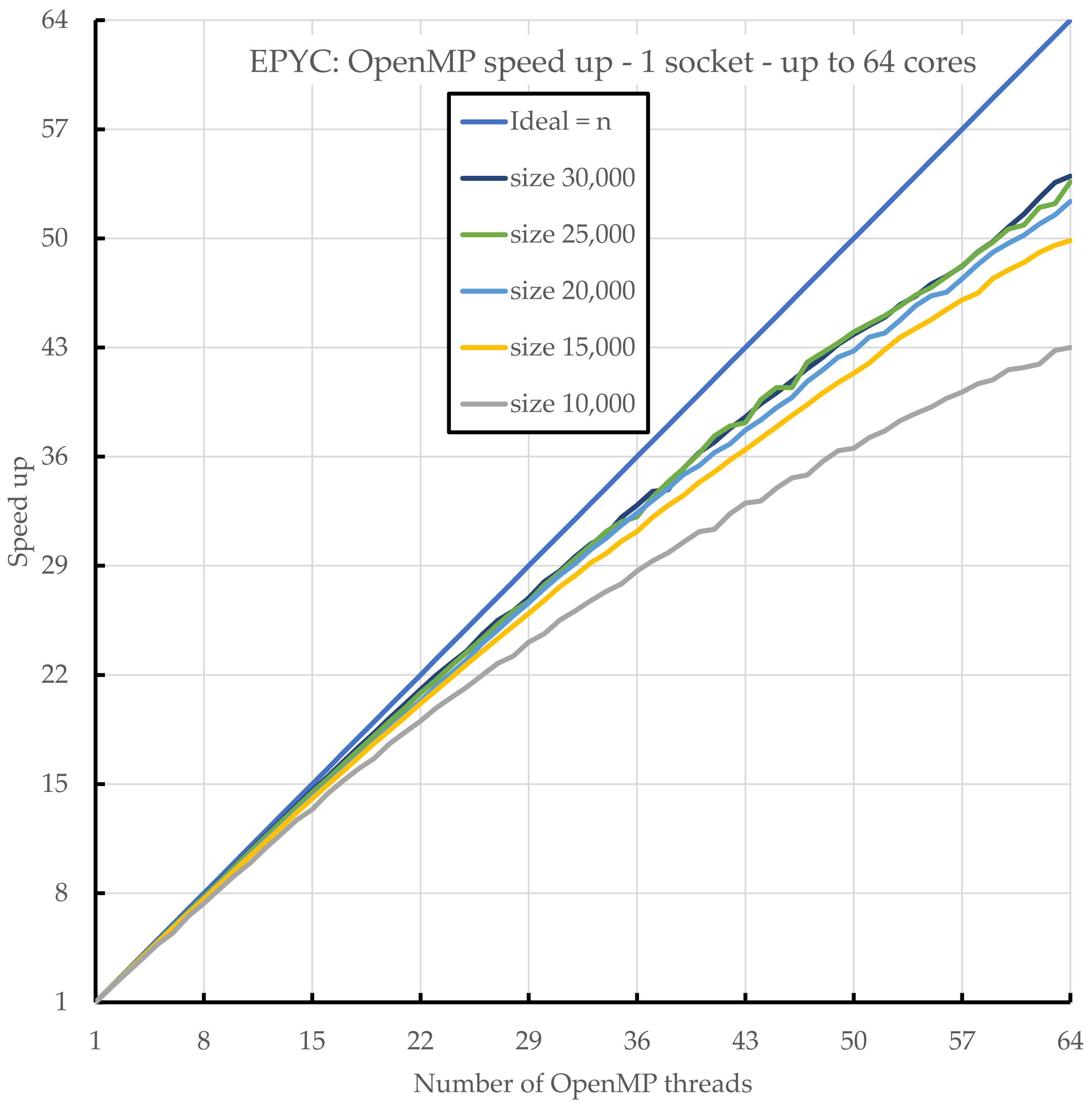


Figure 2: "Static" OpenMP scalability on Epyc – 1 socket – up to 64 cores.

The graph charts show that the program is scaling quite well, and better with bigger sizes of data. When the world is defined to be of size 10,000, it means that cells are positioned on a square grid of 10,000 rows by 10,000 columns. The worst behaviour is observed for size 10,000. Sizes from 15,000 to 30,000 have a similar plot that is quite less then ideal just for the bigger number of threads used. For example, using 60 threads, for the execution with the three bigger sizes, the speed up is close to 50.
The Thin nodes have a similar behaviour, with the range of OpenMP threads going from 1 to 12.

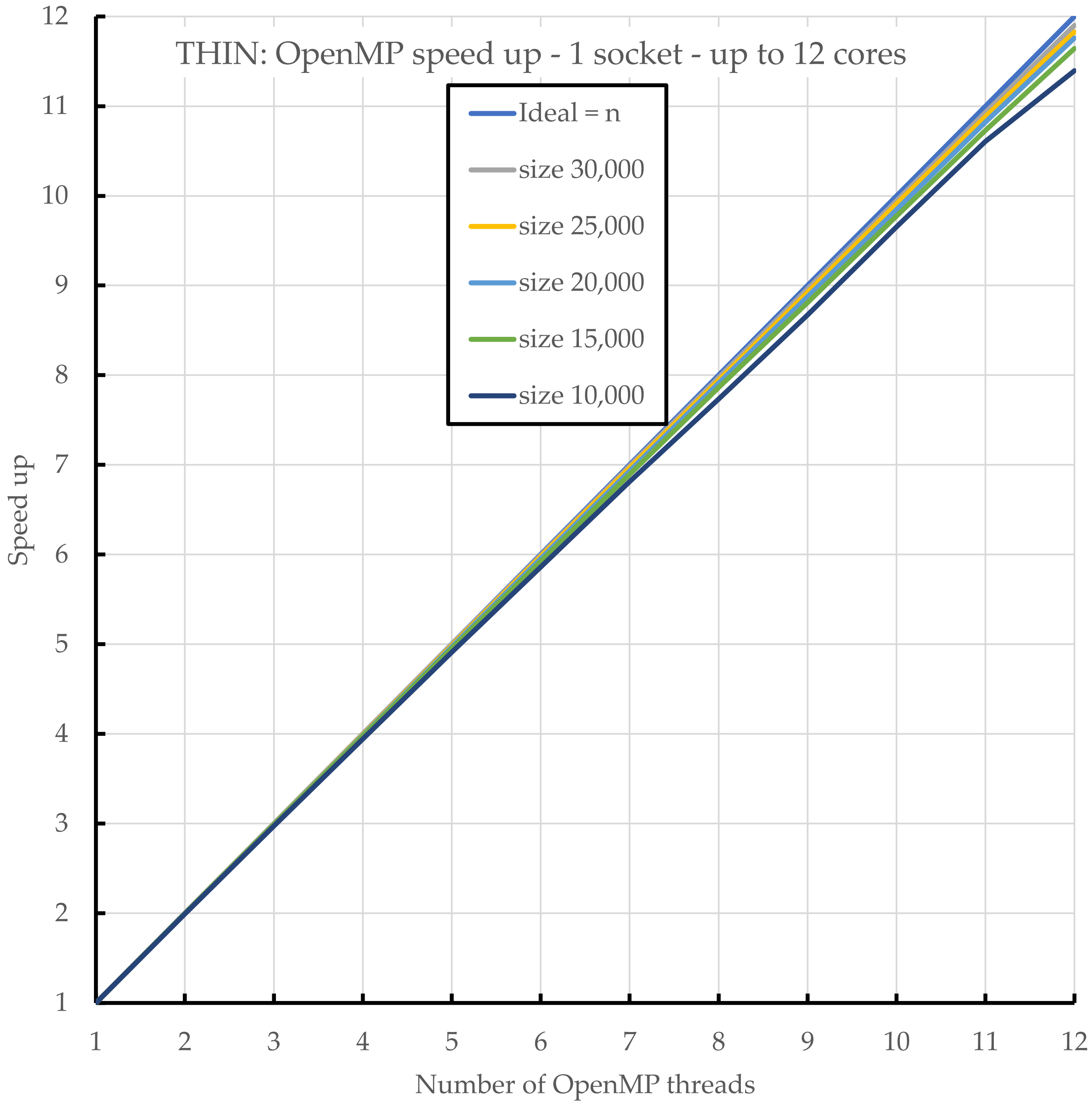


Figure 3: "Static" OpenMP scalability on Thin – 1 socket – up to 12 cores.

The plots seem closer to the ideal line, but this is related to the smaller number of maximum threads of 12 against 64 of Epyc. It can be observed that the size of 10,000 is scaling worse than bigger sizes, especially using more than 10 threads.

Strong MPI scalability

To analyse the program behaviour with an increasing number of MPI tasks, a bash shell script is used to set up a specific environment. A cycle increases the number of tasks from 1 up to the number of cores present on the socket:

```
for tasks in {1..64}; do
    echo rep "$REP" scalability -e"$TYPE" "$SIZE" "$tasks"
    mpirun -np "$ tasks" --map-by core --report-bindings gameoflife.x
     ...
```

The same strategy has been used as already described about the OpenMP scalability, about the many executions with reversed order of tasks or specific ranges of numbers, sizes, and repetitions.

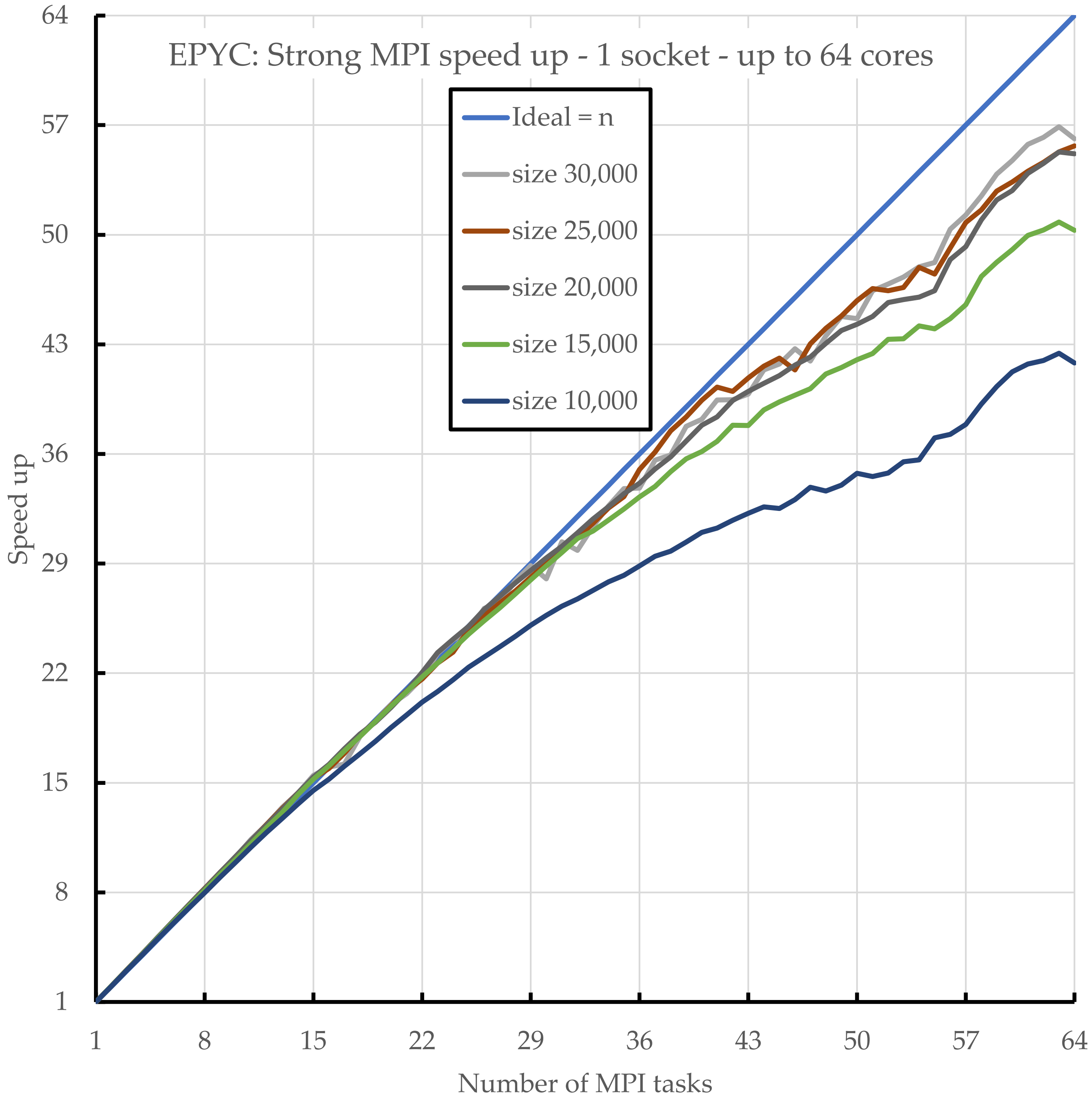


Figure 4: "Static" Strong MPI scalability on Epyc - 1 socket – up to 64 cores.

The program scale quite well, better for bigger sizes of the data workload. It has already been written about the OpenMP scalability that the 10,000x10,000 size has the worst behaviour. The performance is a bit better that that seen about OpenMP scalability, for example, using 60 threads, for the execution with the three bigger sizes, the speed up is well over 50.

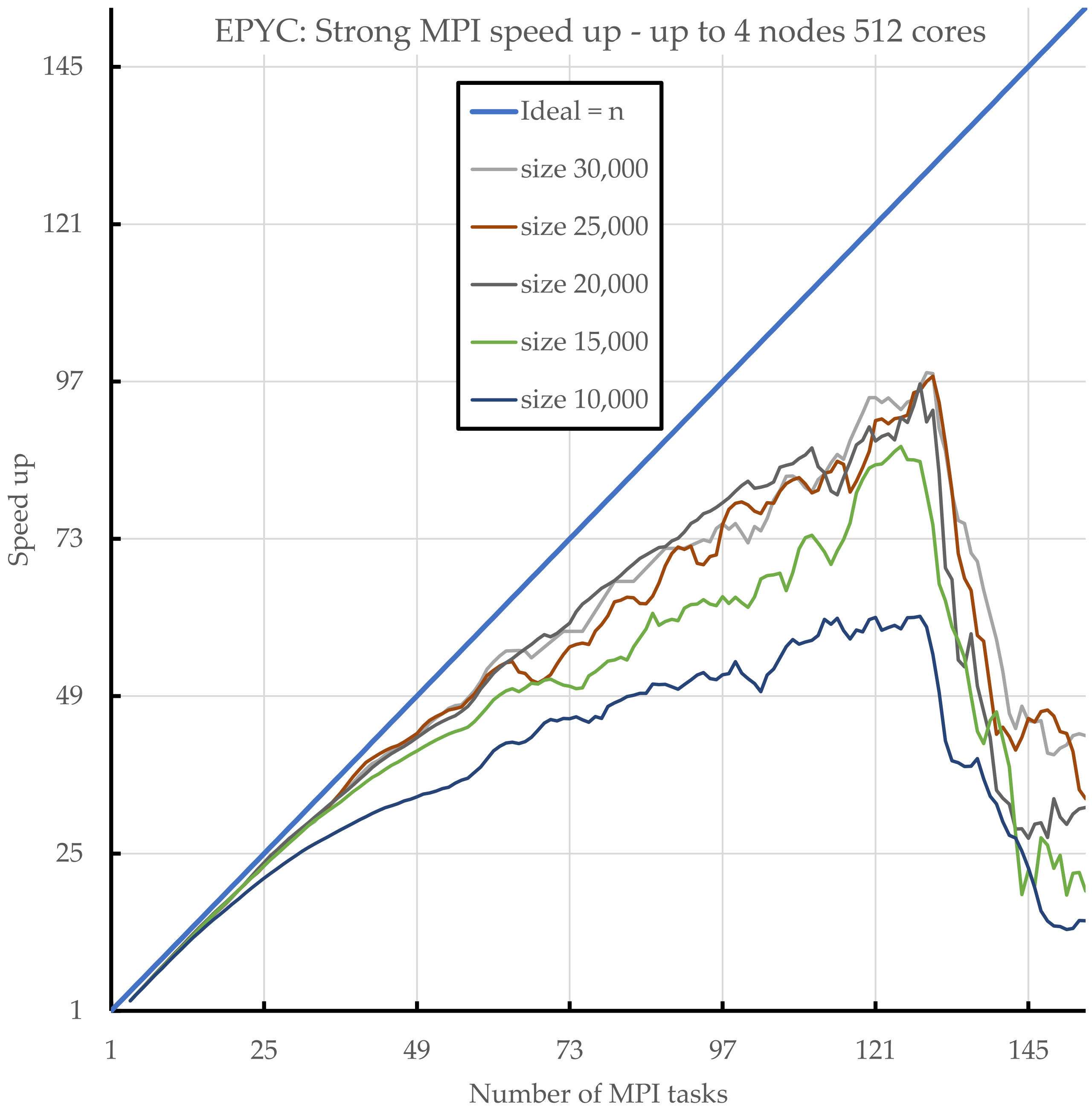


Figure 5: "Static" Strong MPI scalability on Epyc - up to 4 nodes 512 cores (cut at 150).

The program continues to scale even running on more cores, up to the 128 cores of a single Epyc node. When more nodes are used, a sudden decrease in performance is observed. The chart has been cut at 150 MPI tasks, as for bigger numbers the speedup remains almost constant.

The Thin nodes have a similar behaviour, with the range of OpenMP threads going from 1 to 12, the same considerations can be done as seen about the OpenMP scalability, noting the difference between scaling up to 12 tasks compare to 64.

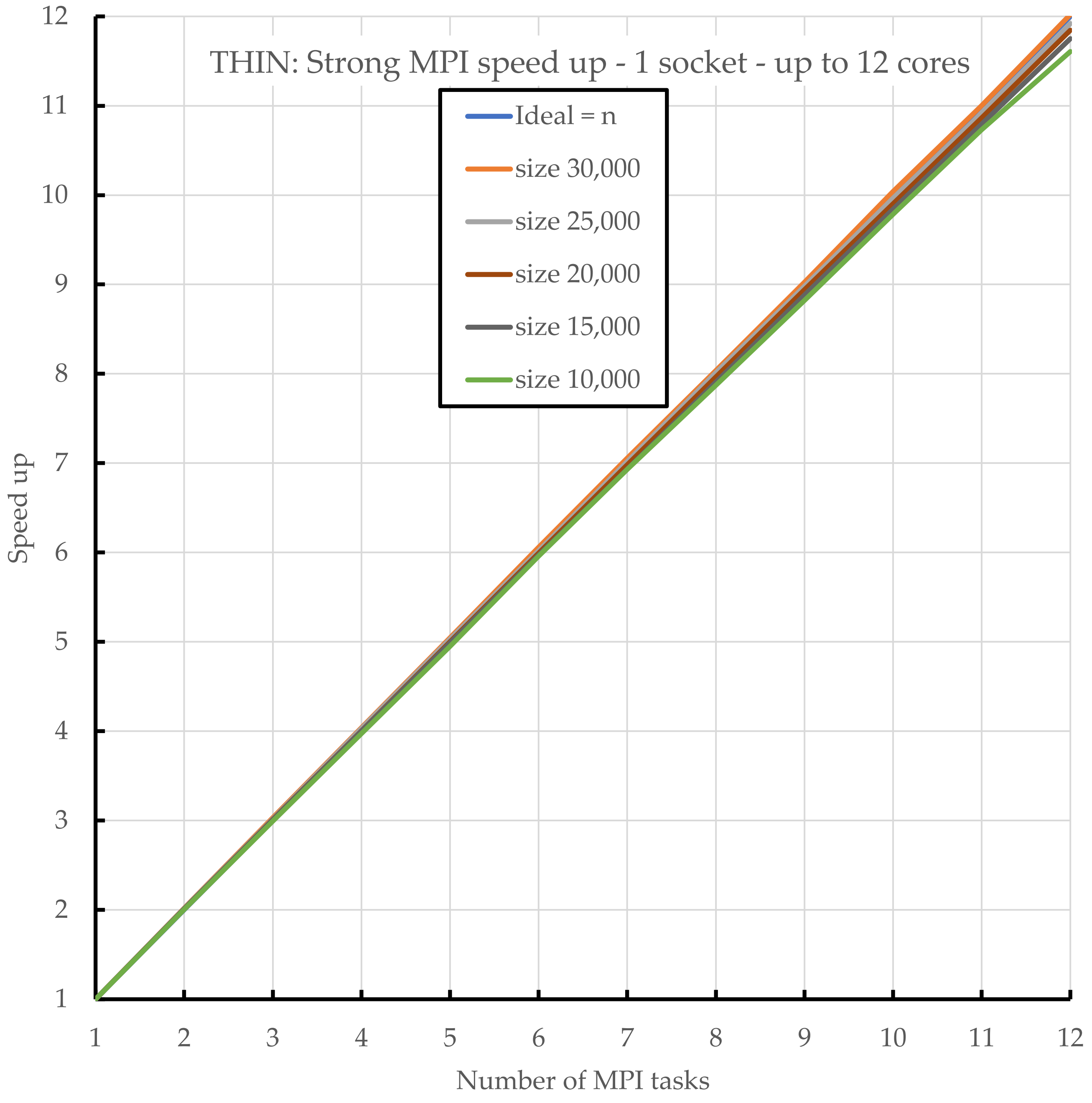


Figure 6: "Static" Strong MPI scalability on Thin - 1 socket – up to 12 cores.

A worse behaviour appears involving more nodes in the computation, scaling up to the maximum available of 4 nodes, each with 2 sockets with 12 cores, that means a total of 72 cores.

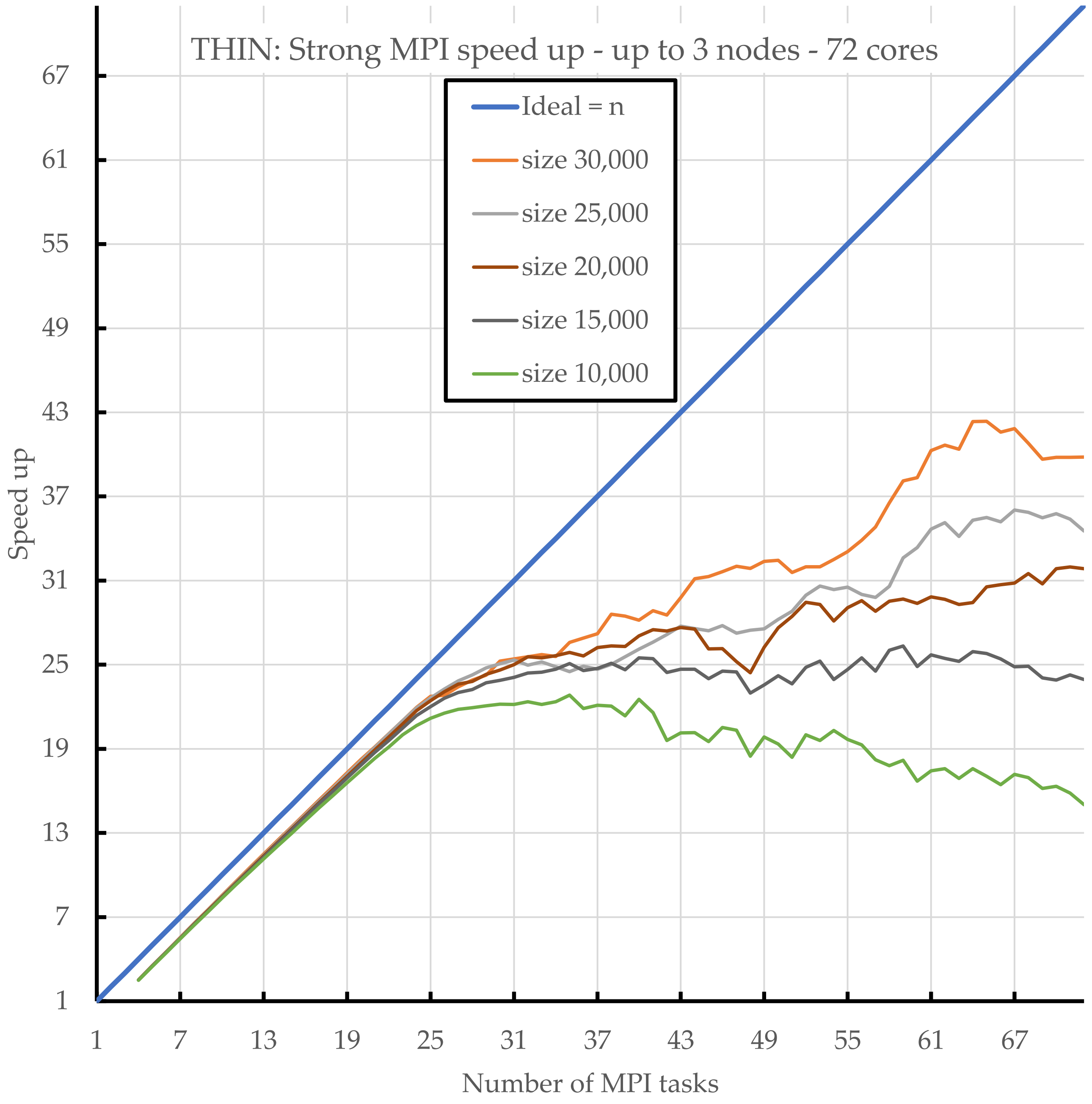


Figure 7: "Static" Strong MPI scalability on Thin – up to 3 nodes – 72 cores.

Strong scalability is almost linear on THIN nodes up to the 24 cores of a single node. Running on more than 24 tasks shows and more irregular improvement and limited to the two bigger sizes of 25,000 and 30,000 as number of rows and columns of the square matrix.

Weak MPI scalability

For this kind of study, the data size is increased together with the number of MPI tasks. To have a constant workload with different executions, the world size is increased in a way that the total size of the array in memory is proportional to the number of tasks, when using a double number of MPI tasks, the total size must double too. To achieve this situation a computation must be done to obtain the "world size" that will determine the total size as the square of that value. The “sbatch” script takes care of this, running over the sizes that double the total size, and the number of tasks is doubled accordingly as follows:

```
for SIZE in 4000 5657 8000 11314 16000 22627 32000; do
    echo rep "$REP" scalability -e"$TYPE" "$SIZE" "$tasks"
    mpirun -np "$tasks" --map-by core --report-bindings gameoflife.x -r
           -f pattern_random$SIZE.pgm -n $STEPS -e "$TYPE" -s "$SNAPAT" -q
    ((tasks = tasks * 2))
```

These are the sizes used, given the above script:

size	total size	tasks
4,000	16,000,000	1
5,657	32,001,649	2
8,000	64,000,000	4
11,314	128,006,596	8
16,000	256,000,000	16
22,627	511,981,129	32
32,000	1,024,000,000	64

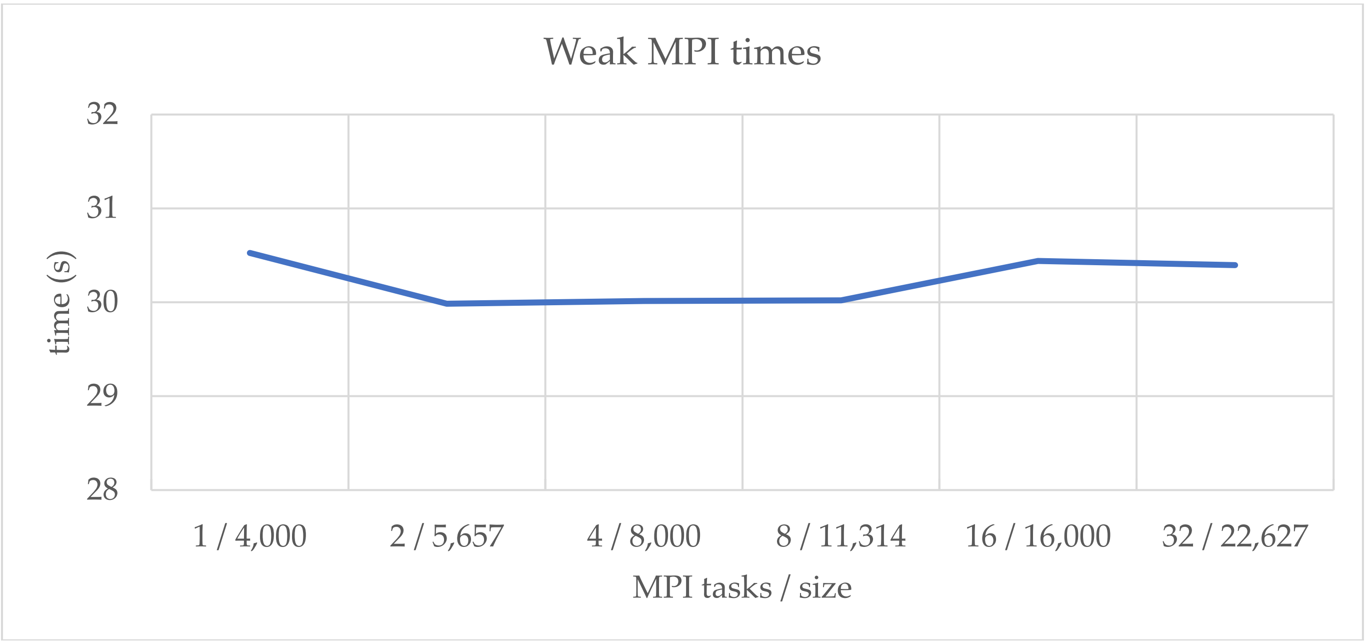


Figure 8: "Static" Weak MPI scalability on Epyc.

The total execution time does not change with the increasing number of MPI tasks processing proportionally bigger data sizes. The same has been observed running the program on THIN nodes or with different combinations of MPI and OpenMP settings.

Wave evolution

This evolution shows a similar trend to the ordered but is a bit slower, and this is expected as it has more computations to do.

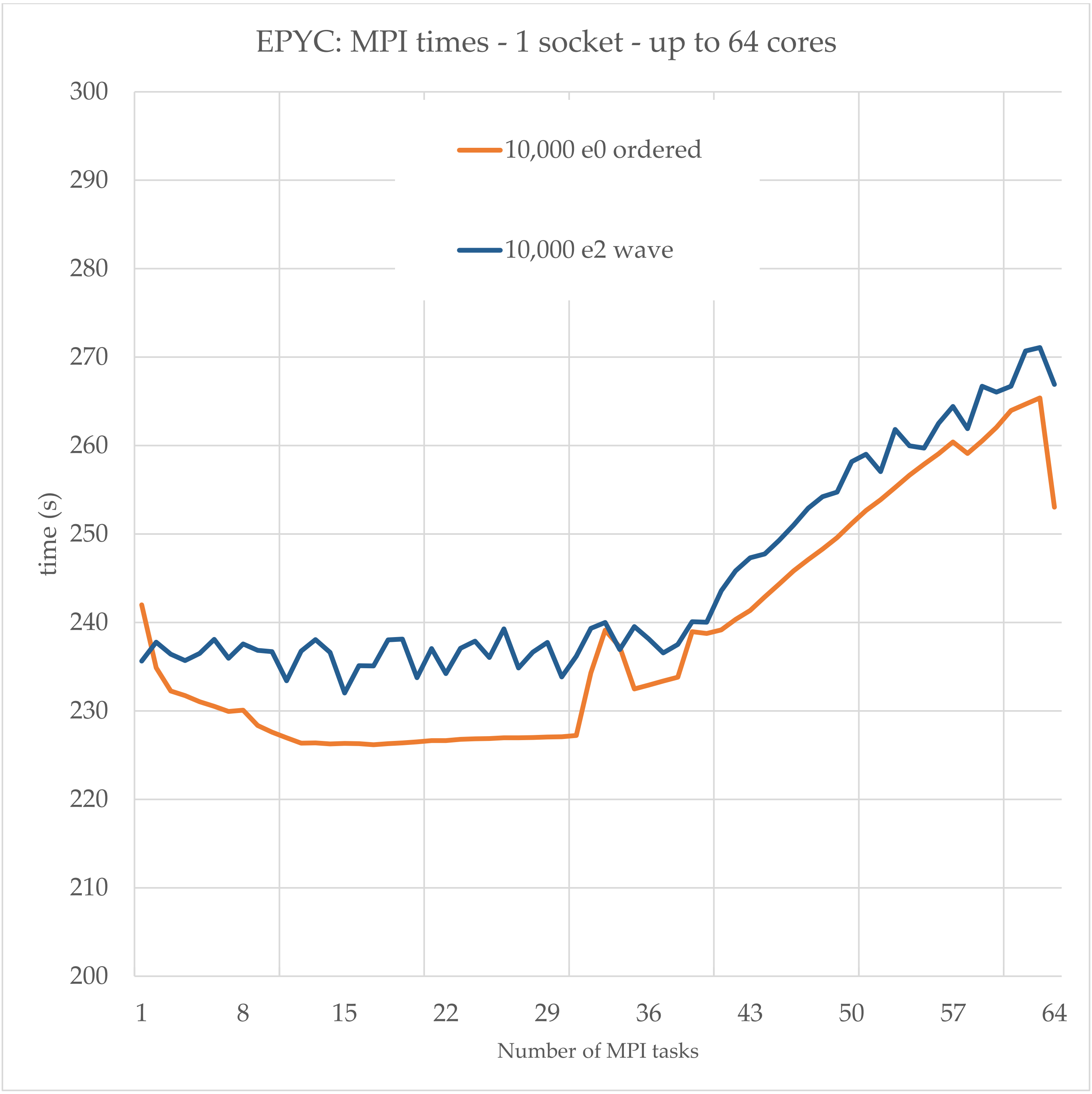


Figure 9: "Wave" times on Epyc – 1 socket – up to 64 cores.

White-black evolution

The program is slightly slower than the static evolution case, but the behaviour is very similar. A small increase in the measured elapsed time can be observed when using 16 cores and saving snapshots at each iteration step.

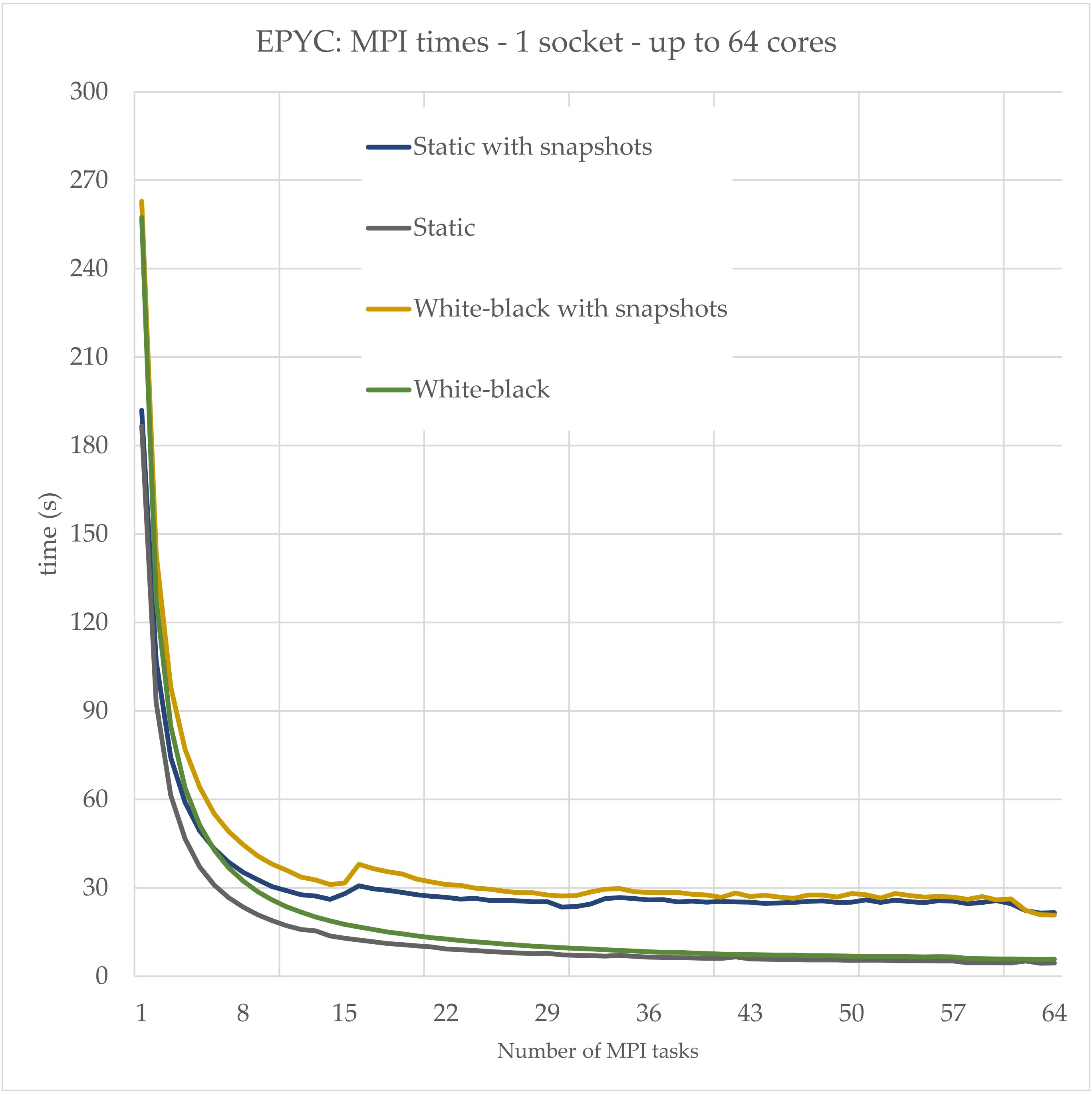


Figure 10: times for Static and White-black evolutions for a size of 10,000.

I/O performance

Analysing the measured total elapsed times of the I/O operations, it is close to zero when saving the final output only, and it is around 15 seconds when saving a snapshot at each step. Again, the behaviour is very similar for static and white-black evolutions.

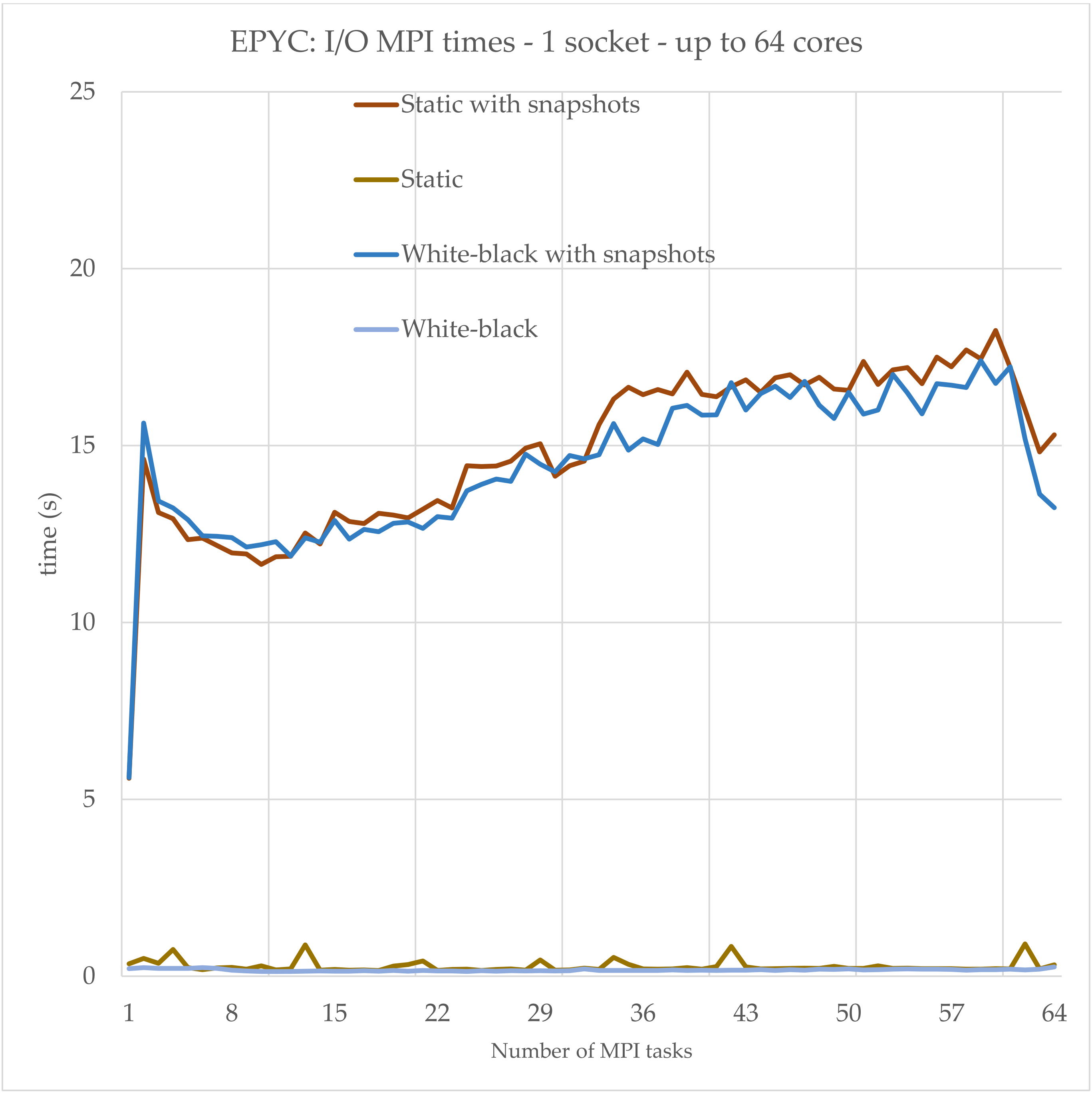


Figure 11: I/O times for Static and White-black evolutions.

Conclusions

The program performance scales when running the "Static" as well as the "White-Black" evolutions, within the given range of Epyc and Thin nodes, sockets, cores. "Ordered" and "Square wave" evolution have a serial behaviour and can just scale the use of the memory to process bigger problem sizes.
It should be possible to achieve better parallelization with the concurrent file reading and writing, even if no improvements show in the results. It may be investigated if the issue is related to the data size, the file system or simply hidden by all the MPI communications effort. The time taken

for writing to files is anyway much less than that taken by the computations, even when writing snapshots at each iteration step, this means that there is little space for improvement anyway. This is even more true about file reading, so even if it could be improved with specific I/O functions and OpenMP parallelization, it is not expected to improve the overall performance by a significant percentage.
The code can be extended in various aspects. The time recording could be refined subtracting the time overhead spent to manage the recording itself.
Another aspect that may be interesting for investigation is the compiler. Different compilers should be used to compare the compiled program performance. It is expected that dedicated compiler and compiler options could enhance the performance on specific hardware.
The code could be refined about the management of border line situations like limiting the minimum size to 3 to avoid the overlapping of border columns and rows, this would make the program more robust, but it may be out of interest as it is not conceived to be used massively by the crowds.
When using the MPI_Send() function it would be necessary to check that the size of the message does not exceeds the maximum size available using the INT data type.
In some cases the string memory is allocated to default values like "malloc(60)", it would be a bit more memory efficient to calculate the exact string length instead, even if it is quite clear that advantages would be minimal.
In the wave evolution some code has been implemented to always save the initial pattern as the first output step as "snapshot_00000" to help building a complete evolution animation. The code should be implemented for all other evolutions too, but it has been commented and left for future editing if the functionality would become necessary.
The code passes the "debug_info" level as a parameter to many functions, it should be removed to achieve the best possible performance. It has been taken care however to include all the debugging code between conditional preprocessor directives to be able to reduce any overhead when compiling the final executable. Some "#define" macro could be set by the makefile to better mange specific compilation for debugging or maximum performance executables.

Exercise 2

Introduction

Comparing MKL, OpenBLAS and BLIS on matrix-matrix multiplication.[5]
Performance comparison of some math libraries available on HPC: MKL, OpenBLAS and BLIS. The comparison is performed measuring the number of floating-point operations per second (FLOPS)[6]. The focus is on the level 3 BLAS function called gemm. Such function comes in different flavours, for double precision (dgemm) and single precision (sgemm). The given source code "*`gemm.c`*" has been adapted to enable dgemm and sgemm testing, enabling csv data output to measure performance within a range of problem sizes and execution settings.
The nature of matrix-matrix operations is such that, when the size of the problem grows, the number of floating-point operations to be performed grows faster than the number of memory transfers needed. This kind of operations are said to be CPU bounded instead of memory bounded.

Methodology

To compare the performances of the math libraries MKL, OpenBLAS and Blis, several executions have been run on the epyc and thin nodes. At each run all the useful data has been recorded and collected. To compare the performances the main information used is the number of single or double precision floating point operations per second, measured in GFLOPS.
Two main kinds of executions have been run, one scaling the size of the processed problem data and the other scaling the number of processors used, as cores that run OpenMP threads. The given code has been adapted to save all data as a csv comma separated file, that can be paired to a job output file that records all the parameters of the environment of each execution.
Eventually all data from paired files have been collected into spreadsheets to manage, filter, analyse them and produce graphic charts.
To reduce the error noise in the experiments multiple repetitions have been run with the same execution settings, keeping track of each.
Batch files have been used and iteratively refined to run the tests avoiding external perturbances, implementing a practical way to run with multiple possible combinations, between the different architectures (epyc and thin), floating point operations (single or double precision), size of the data, number of processors, processor binding policies, compiler optimization settings, choice of nodes from the given partitions, order of the sequence of the executions (running multiple times with the same settings before scaling to the next or scale to the full sequence and then repeat all of them, from the smaller to the biggest or vice versa).
For each single execution it has been also saved all the information regarding the batch job that run it with date and time. This has been useful to spot errors and anomalies.
Eventually the data have been plotted in graphic charts, adding the Theorical Peak Performance as a reference, adjusting scales and colours, to enable the analysis and conclusions.

Theoretical peak performance

The theoretical peak performance computation:

$$Tpp = \frac{(clock\_rate \times \#\_of\_FP\_operations \times \#\_of\_cores)}{sec}$$

The HPC environment of Orfeo offers some different hardware:[7]

CPU	Sockets	# of CPU Cores	Base Clock
EPYC Amd 7H12[8]	2	64	2.6 GHz
THIN Intel Xeon Gold 6126[9]	2	12	2.6 GHz

EPYC nodes

In EPYC nodes a single core can deliver up to 16 double precision floating point operations per cycle, since we have 64 cores in a socket and the maximum frequency is 2.6 GHz (base clock rate), the theoretical peak performance is close to 2.7 TFLOPS. The most common reported performance at benchmarks is 2.550 TFLOPS[10][11], that is reasonably close to the theoretical peak.

$$Tpp(socket) = \frac{(2.6\text{ GHz} \times 16 \times 64)}{sec} = 2.6624 \sim 2.7\ TFLOPS$$

It can be noticed that the product of 16 by 64 is 1,024, so it is quite easy to have a direct idea of the singles socket Tpp given the base clock frequency.
The peak performance of a node with two sockets or that of a single core can be computed in the same way:
One node with two sockets for a total of 64 cores Tpp(node) = 5.3 TFLOPS.
Single core Tpp(core) = 41.6 GFLOPS for double precision floating point operations.

THIN nodes

It must be noticed that "Intel no longer makes FLOPS (Floating Point Operations)[6] per cycle information available for Intel® processors. Instead, Intel publishes GFLOPS (Giga-FLOPS) and APP (Adjusted Peak Performance) information."[12] So the information can be found in the relative document:
APP Metrics for Intel Microprocessors Intel Xeon Processor Revision.05 Date: 11/16/2022

Processor Group	Processor Name	GFLOPS	APP
Intel® Xeon® Gold 6126 (19.25M Cache, 2.60 GHz)	6126T	652.8	0.19584

This means that in THIN nodes the performance of a single core must be computed dividing the given value of 652.8 GFLOPS by the number of cores, that is 12:

$$Tpp(core) = \frac{652.8}{12} = 54.4\ GFLOPS$$

Given the maximum frequency of 2.6 GHz (base clock rate) we can identify the maximum number of double precision floating point operations per cycle:

$$\#\_of\_FP\_operations = \frac{\text{Tpp(core)}}{clock\_rate} = \frac{54.4}{2.6} = 20.9231 \sim 20.9\ FLOP\ per\ cycle$$

This means that a single core can deliver up to 20.9 double precision floating point operations per cycle.

Curiously, there are not many reports about the 6126 performances at benchmarks. In "Performance of Devito on HPC-Optimised ARM Processors (Hermes Senger, Jaime F. de Souza, Edson S. Gomi, Fabio Luporini, Gerard J. Gorman)"[13] , these are the performance evaluations:

Max. # FP64 ops./cycle	32
Max. perf.(FP64) Gflop/s	883.2
Linpack (FP64) Gflop/s	695.0

These values are quite bigger then the declared peak value, so it would be interesting to have more data available.

In the course slides the values for the peak performance are 1.997 TFLOPS that for a single core would lead to a Tpp(core) = 1.997 TFLOPS / 24= 83.2 GFLOPS, that are much bigger values. We do not have details about how that number is calculated. After the experiments it has been noticed that these are the most reasonable values, so they will be used as the THIN nodes Tpp reference in the plots.
The Intel THIN nodes have two AVX-512 FMA Units. Intel Advanced Vector Extensions 512 (AVX-512), new instruction set extensions, delivering ultra-wide (512-bit) vector operations capabilities, with up to 2 FMAs (Fused Multiply Add instructions), to accelerate performance for the most demanding computational tasks. This may well explain the observed performance.

Scalability over size

Scalability over the data size have been analysed measuring the performance at fixed number of cores, 128 for Epyc, 24 for Thin, increasing the matrices size from 2,000x2,000 to 30,000x30,000.
At the beginning a 20,000x20,000 size have been used has the top limit, but it has been scaled up after the first experiments, noting that the bigger sizes would have been more appropriate and interesting for the analyses.

Scalability over the number of cores at fixed size

Scalability over the number of cores have been analysed measuring the performance at fixed size. After a first round of executions using at most 64 cores on epyc and 12 on thin, the maximum has been raised to 128 and 24 respectively. After a first analysis using matrix sizes of 20,000, all experiments have been repeated with a size of 30,000. Both changes have been done to inspect more interesting behaviour.

Double and single precision

All experiments have been run twice, one with a specifically C program that does the computations in double precision floating point operations, and another program in single precision. All data have been collected keeping the specific kind of program information.

Implementation

The code has been adapted to correctly output the data as csv, that means comma separated values. A CLion project has been created to write, check, and format the C code and the Makefile. The files have then been transferred to Orfeo by sftp for building the executables on the target platform.

Makefile
The makefile defines the locations and switches for the libraries that are under test:

```
### MKL libraries
MKL= -L${MKLROOT}/lib/intel64 -lmkl_intel_lp64 -lmkl_gnu_thread -
lmkl_core -lgomp -lpthread -lm -ldl
### OpenBLAS libraries
OPENBLASROOT=${OPENBLAS_ROOT}
### BLIS library
BLISROOT=/u/dssc/mdepet00/assignment/exercise2/blis
```

A separate executable is produced by compiling the same source with different options, for all the combinations of library used, single (sgemm) or double (dgemm) precision floating point operations and with compiler optimizations enabled. A specific name is used for each executable to uniquely identify it:

```
cpu: sgemm_mkl.x sgemm_oblas.x sgemm_blis.x
     dgemm_mkl.x dgemm_oblas.x dgemm_blis.x
     sgemm_mkl_optimized.x dgemm_mkl_optimized.x
     sgemm_oblas_optimized.x dgemm_oblas_optimized.x
     sgemm_blis_optimized.x dgemm_blis_optimized.x
```

For each of the compiled executable the corresponding switches are enabled and passed to the gcc compiler:

```
sgemm_mkl.x: gemm.c
  gcc -DUSE_FLOAT -DMKL $^ -m64 -I${MKLROOT}/include $(MKL) -o $@
...
dgemm_blis_optimized.x: gemm.c
  gcc -O3 -march=native -DUSE_DOUBLE -DBLIS $^ -m64
    -I${BLISROOT}/include/blis -L/${BLISROOT}/lib
    -o $@ -lpthread -lblis -fopenmp -lm
...
```

The code outputs the csv using the “printf” function:

```
printf("%i,%i,%i,", m, k, n);
...
printf("%d.%08d,"   , diff(begin, end).tv_sec
                    , diff(begin, end).tv_nsec);
printf("%lf,%lf\n", elapsed, gflops);
...
```

The run durations are computed in two different ways, that are then compared for correctness, given the bit of complexity necessary to get the correct measurements in the time units. In the previous “printf” statements a first output is done by the formatting string "%d.%08d" that is then filled by the separate seconds and nanoseconds values returned in the time data structure by the “diff” function. Then and “elapsed” variable is assigned by the actual summation of those values, taking in consideration the needed precision. The performance value is computed in GFLOPS:

```
clock_gettime(CLOCK_MONOTONIC, &begin);
GEMMCPU(CblasColMajor, CblasNoTrans, CblasNoTrans,
            m, n, k, alpha, A, m, B, k, beta, C, m);
clock_gettime(CLOCK_MONOTONIC, &end);
elapsed = (double) diff(begin, end).tv_sec
          + (double) diff(begin, end).tv_nsec / 1000000000.0;
double gflops = 2.0 * m * n * k;
gflops = gflops / elapsed * 1.0e-9;
```

The batch files loop the executions over all the combinations, for the relative scaling. They all start with the conventional slurm header declarations, specific for epyc and thin nodes:

```
#!/bin/bash
#SBATCH --no-requeue
# set the job name with "sbatch -J thejobname <script>.sh
[arguments]
#SBATCH --job-name="sgemm_epyc"
#SBATCH --get-user-env
#SBATCH --chdir=/u/dssc/mdepet00/assignment/exercise2
#SBATCH --partition=EPYC
#SBATCH --nodes=1
#SBATCH --exclusive
#SBATCH --ntasks-per-node 128
#SBATCH --mem=490G
#SBATCH --time=02:00:00
#SBATCH --output=sgemm_epyc_job_%j.out
```

Then the "module load" is used to set up needed environment. At some point Orfeo have been updated and some declarations had to be changed. For example, it is no more necessary to specify the architecture and a warning is returned. Some libraries and frameworks were updated too (OpenMPI not used for this exercise anyway):

```
22/06/23:
module load architecture/AMD
Lmod Warning:  The architecture selection module has been
     deprecated. The module system will now autonomously handle
     the proper selection of binaries.
While processing the following module(s):
Module fullname   Module Filename
---------------   ---------------
architecture/AMD  orfeo/opt/modules/profiles/architecture/AMD.lua

module load openMPI/4.1.4/gnu/12.2.1
Lmod has detected the following error:
The following module(s) are unknown: "openMPI/4.1.4/gnu/12.2.1"
```

So, the configuration has been updated to:

```
module load mkl
module load openBLAS/0.3.23-omp
export LD_LIBRARY_PATH
 =/u/dssc/mdepet00/assignment/exercise2/blis/lib:$LD_LIBRARY_PATH
```

To experiment with different policies of association of the threads to the available cores, the following alternative lines have been used:

```
export OMP_PLACES=cores
export OMP_PROC_BIND=close
...
export OMP_PLACES=cores
export OMP_PROC_BIND=spread
```

A timestamp is set to uniquely identify the output files names. The filename contains the sgemm or dgemm prefix for single or double precision, the name of the library under test, the architecture name epyc or thin, the hostname with the node number and the timestamp:

```
now=$(date +"%Y-%m-%d_%H-%M-%S")
csvname=sgemm_"$LIB"_epyc_$(hostname)_$now.csv
```

At first a loop over the tested libraries has been used to reduce the number of output files, but given the time limit assigned to the Orfeo account, specific job batch files have been created instead, one for each of the libraries and floating point operations precision. This has greatly increased the time needed to execute the jobs, collect, and manage the output.

```
for LIB in oblas_optimized oblas mkl_optimized mkl blis_optimized
blis; do
```

The repetition of the executions to get an average of the output values is implemented with a loop:

```
for REP in {1..10}; do
```

The appropriate loop implements the scaling over data sizes or processor cores:

```
#increase the size of matrices size from 2000x2000 to 20000x20000
for SIZE in {30000..2000..500}; do

#increase the number of cores
for CORES in {128..1}; do
    export OMP_NUM_THREADS="$CORES"
    export BLIS_NUM_THREADS="$CORES"
```

The main execution is then run passing all the parameters, echoing all the relative data to the job output file:

```
echo dgemm_epyc_job_"$SLURM_JOB_ID".out$'
     \t'"$csvname"$'
     \t'"$(hostname)"$'
     \t'rep "$REP" size "$SIZE" OMP_PLACES=$OMP_PLACES
          OMP_PROC_BIND=$OMP_PROC_BIND
          OMP_NUM_THREADS="$OMP_NUM_THREADS"
          BLIS_NUM_THREADS="$BLIS_NUM_THREADS"
          srun -n1 --cpus-per-task="$CORES" ./dgemm_$LIB.x
               "$SIZE" "$SIZE" "$SIZE"'>>'"$csvname"

srun -n1 --cpus-per-task="$CORES" ./dgemm_$LIB.x
     "$SIZE" "$SIZE" "$SIZE" >>"$csvname"
```

A footer is written to the output file to help issue diagnosis:

```
now=$(date +"%Y-%m-%d_%H-%M-%S")
echo "$(hostname)" "$now" END
echo size scalability end
```

The results files have been collected in a local computer transferring them by sftp. The filenames identify them as specified:

```
dgemm_epyc_job_17365.out
dgemm_blis_optimized_epyc_epyc007_2023-07-11_14-15-34.csv
dgemm_thin_job_17371.out
dgemm_mkl_optimized_thin_thin007_2023-07-11_12-16-07.csv
sgemm_epyc_job_17363.out
sgemm_mkl_optimized_epyc_epyc005_2023-07-11_12-15-07.csv
```

dgemm_thin_job_17376.out:

```
Loading mkl version 2022.2.1
Loading tbb version 2021.7.1
Loading compiler-rt version 2022.2.1
size scalability begin
dgemm_thin_job_17376.out dgemm_oblas_optimized_thin_thin007_2023-
07-11_16-17-05.csv  thin007   rep 1 size 30000 OMP_PLACES=cores
OMP_PROC_BIND=close OMP_NUM_THREADS=24 BLIS_NUM_THREADS=24 srun -
n1 --cpus-per-task=24 ./dgemm_oblas_optimized.x 30000 30000
30000>>dgemm_oblas_optimized_thin_thin007_2023-07-11_16-17-05.csv
dgemm_thin_job_17376.out dgemm_oblas_optimized_thin_thin007_2023-
07-11_16-17-05.csv  thin007   rep 1 size 30000 OMP_PLACES=cores
OMP_PROC_BIND=close OMP_NUM_THREADS=23 BLIS_NUM_THREADS=23 srun -
n1 --cpus-per-task=23 ./dgemm_oblas_optimized.x 30000 30000
30000>>dgemm_oblas_optimized_thin_thin007_2023-07-11_16-17-05.csv
...
srun: error: Unable to create step for job 17376: Job/step
already completing or completed
thin007 2023-07-11_18-17-21 END
size scalability end
```

dgemm_oblas_optimized_thin_thin007_2023-07-11_16-17-05.csv:

```
m,n,k,elapsed1,elapsed2,GFLOPS
30000,30000,30000,38.637182160,38.637182,1397.617450
30000,30000,30000,39.791269344,39.791269,1357.081613
30000,30000,30000,43.461023096,43.461023,1242.492610
...
```

All the data have then been inserted into spreadsheet files. Pivot tables have been used to count the number of entries of each experiment, compute averages and eliminate outliers. The analysis led to new executions to explore interesting situations. By filtering, formula applications and macro code execution, the data have been prepared as values series for the graphic chart plots.

In the final days these were the numbers:

```
gemm_size
     38,596 records
     88 Distinct Count of out
     311 Distinct Count of csv
gemm_cores
     4,763 records
     53 Distinct Count of out
     53 Distinct Count of csv
```

Spreadsheets became quite heavy of data, tables, and charts:

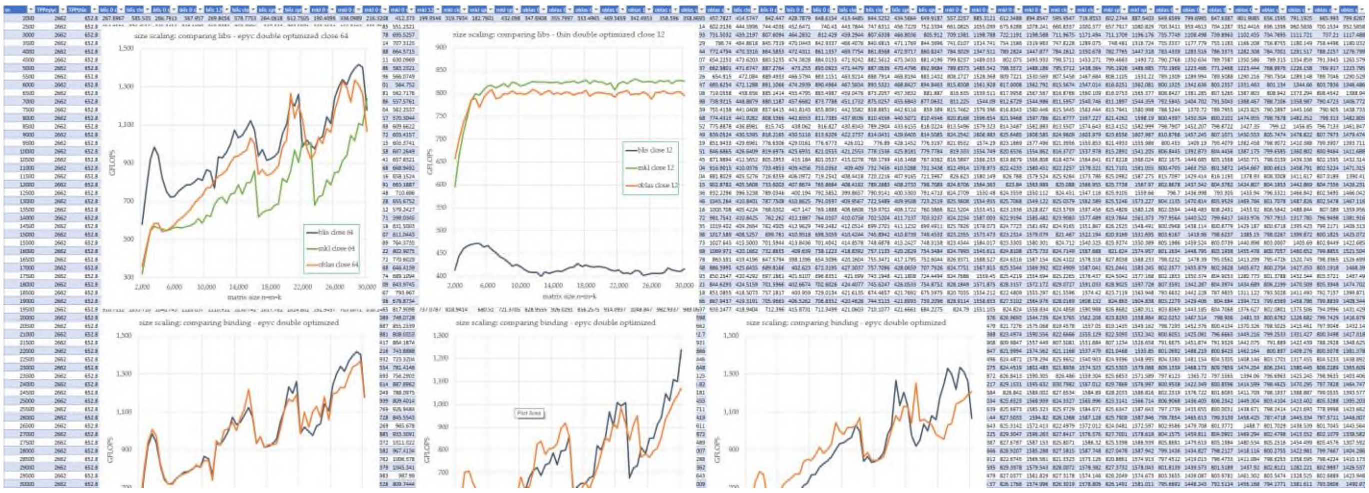

size scaling: comparing libs - epyc double optimized close 64
size scaling: comparing libs - thin double optimized close 12
size scaling: comparing binding - epyc double optimized
size scaling: comparing binding - epyc double optimized
size scaling: comparing binding - epyc double optimized
GFLOPS
matrix size m=n=k
blis close 64
mkl close 64
blis close 12
mkl close 12

Figure 12: spreadsheet data table with charts for size scaling

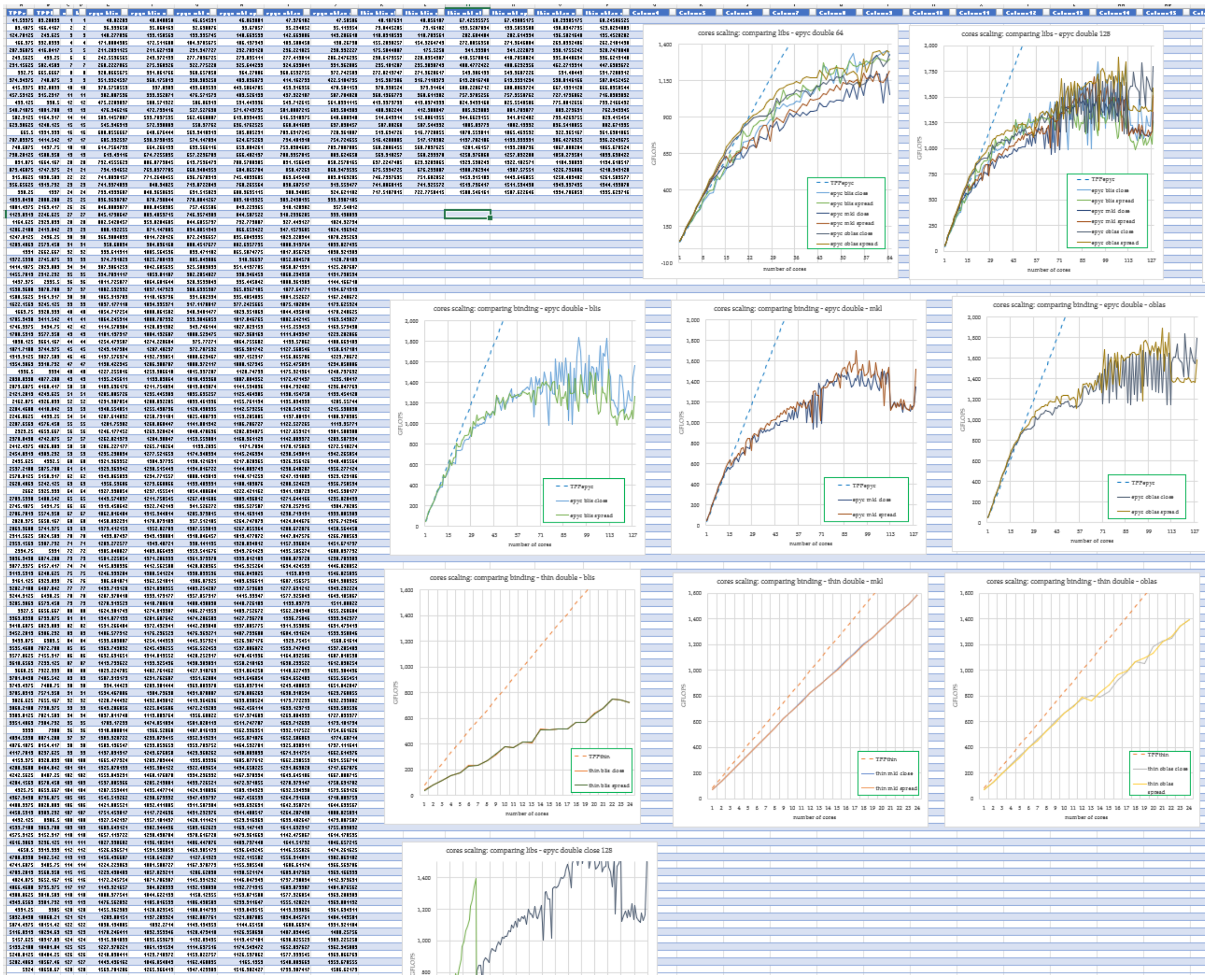

cores scaling: comparing libs - epyc double 64
cores scaling: comparing libs - epyc double 128
cores scaling: comparing binding - epyc double - blis
cores scaling: comparing binding - epyc double - mkl
cores scaling: comparing binding - epyc double - oblas
cores scaling: comparing binding - thin double - blis
cores scaling: comparing binding - thin double - mkl
cores scaling: comparing binding - thin double - oblas
cores scaling: comparing libs - epyc double close 128
GFLOPS
number of cores
TPPepyc
epyc blis close
epyc blis spread
epyc mkl close
epyc mkl spread
epyc oblas close
epyc oblas spread
TPPthin
thin blis close
thin blis spread
thin mkl close
thin mkl spread
thin oblas close
thin oblas spread

Figure 13: spreadsheet data table with charts for cores scaling

Results & Discussion

Some of the produced graphic charts are presented, not all of them as that would make this report too long. The spreadsheets are available on the GitHub repository[14] to see the omitted ones and possibly produce others using the available data series.

Data series have been chosen for each graph to make them readable while enabling a comparison of the behaviours enriched with all the elements that can help. The theoretical peak performance has been plotted as a dashed line where it was useful as a reference. It is a constant line related to the fixed number of cores used for size scalability and an increasing inclined line for the cores scaling.

Scalability over size

The charts show the performance values measured in GFLOPS at fixed number of cores. On the horizontal axes the data matrices size is increased from 2,000x2,000 to 30,000x30,000 in steps of 500.

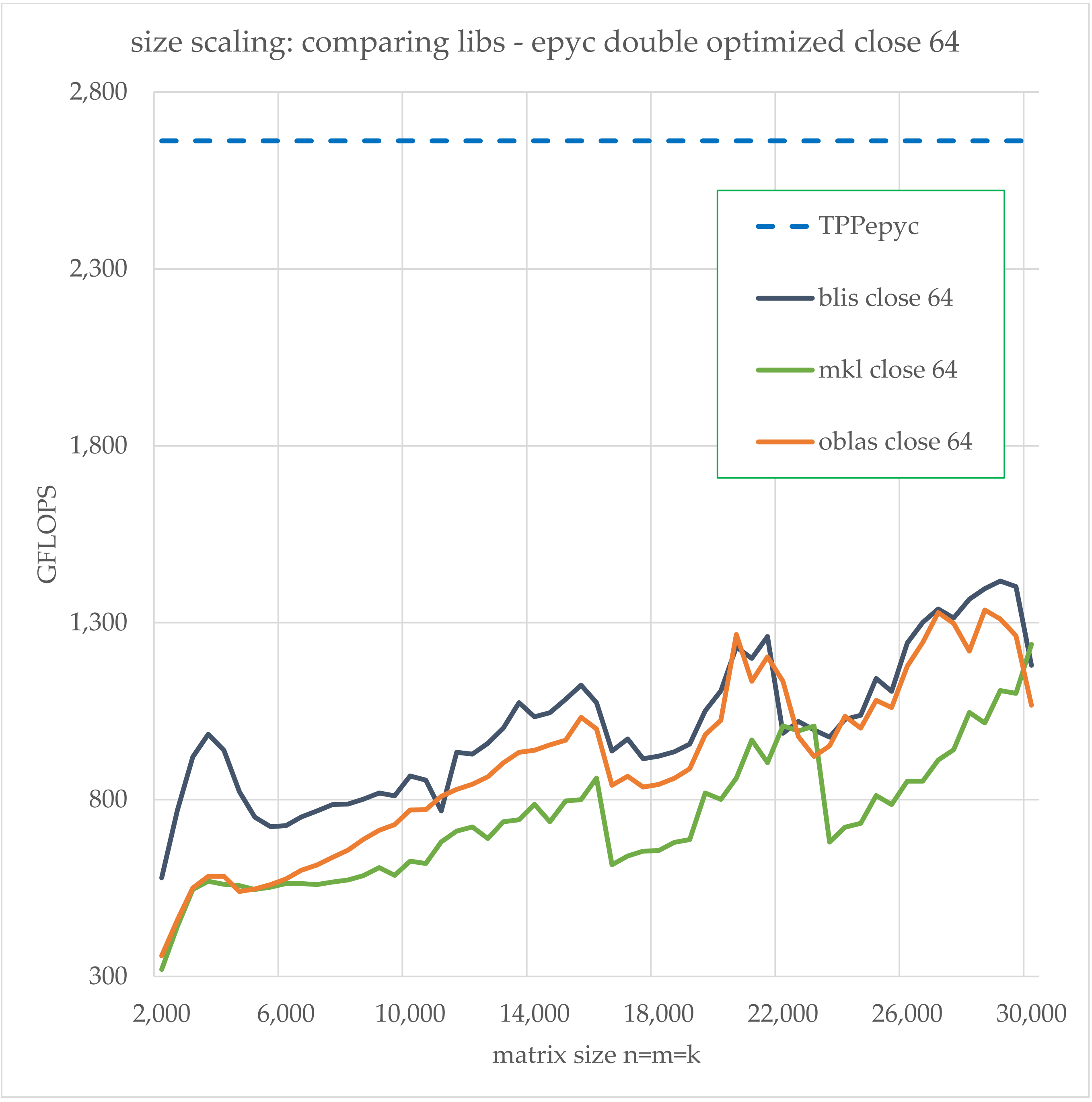


Figure 14: Scaling over matrix size, epyc nodes, double precision (dgemm), compiled with optimizations, binding policy "close", 64 cores. Comparing with the TPP Theoretical Peak Performance.

This first comparison shows that the TPP for epyc nodes is much bigger than the measured values. The TPP dashed line is omitted in following charts when it would require a useless enlargement of the values scale.
Libraries have a similar behaviour, tough blis is performing quite better.

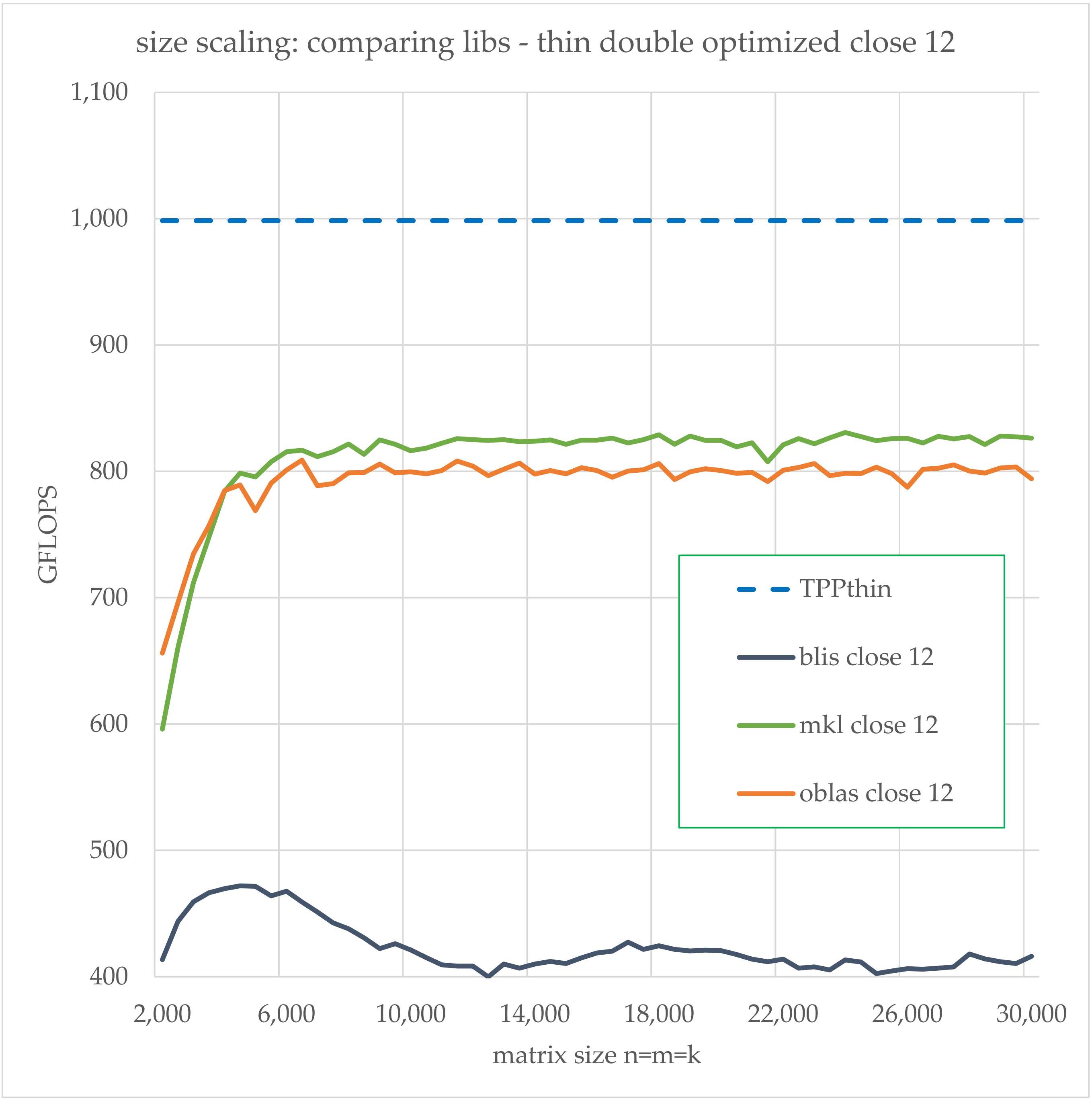


Figure 15: Scaling over matrix size, thin nodes, double precision (dgemm), compiled with optimizations, binding policy "close", 12 cores. Comparing with the TPP Theoretical Peak Performance.

For the thin nodes the best performance stabilizes at around 825GFLOPS, against a TPP of 998.5GFLOPS.
The mkl and oblas libraries have a similar trend, with mkl having slightly better numbers. The blis library shows much worse results, even worsening for sizes bigger then 6,000x6,000.

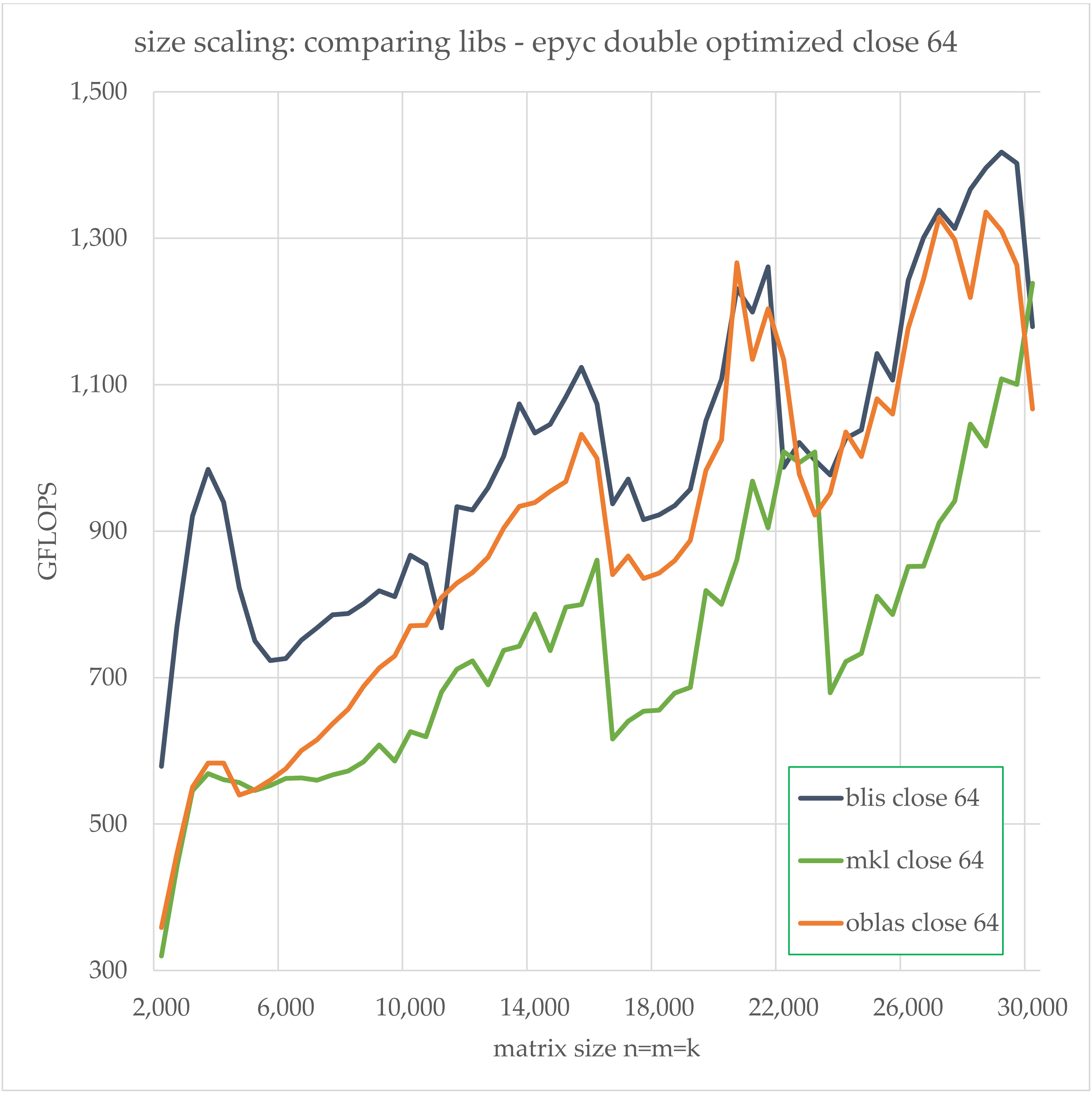


Figure 16: Scaling over matrix size, epyc nodes, double precision (dgemm), compiled with optimizations, binding policy "close", 64 cores.

With a restricted vertical scale range, it is easier to compare libraries performance differences. Some oscillation of values in the same size ranges can be observed for all libraries and must be related to the architecture. The values decrease suddenly after sizes 16,000x16,000 and 22,000x22,000 and then reprise. Those points are shifted to the right for the worst performing mkl library by 500 to 1,500.

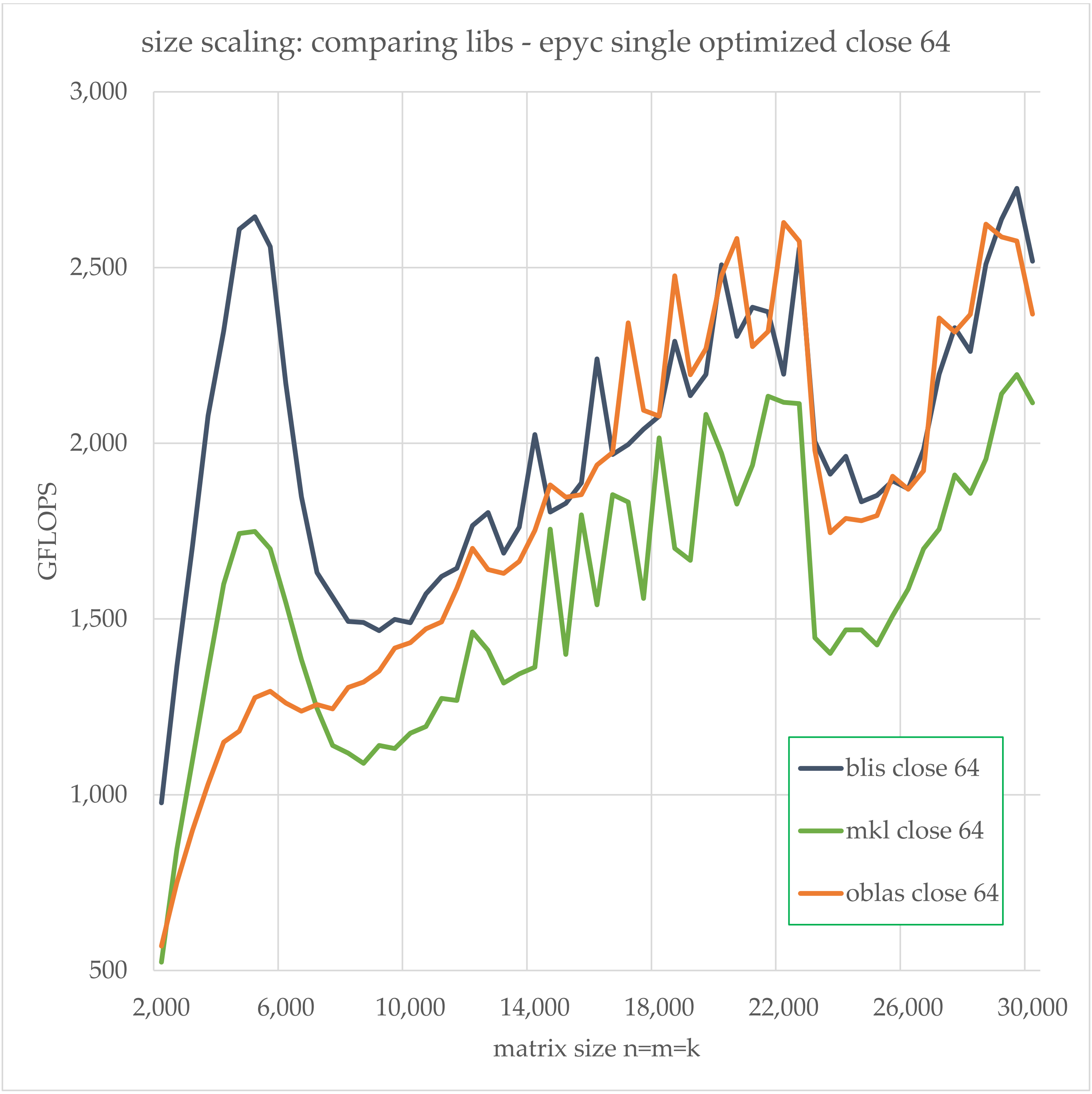


Figure 17: Scaling over matrix size, epyc nodes, single precision (sgemm), compiled with optimizations, binding policy "close", 64 cores.

The same comparison is done for the single precision floating point operations instead of double precision. The behaviour is similar with doubled values, as the processed data is half the size, 32 bits of single instead of 64 bits of double precision.

The trend shows an initial peak, like the double precision chart, but for bigger sizes there are more short “teeth” and a sudden decrease for size of 23,000x23,000 and then the performance increases again.

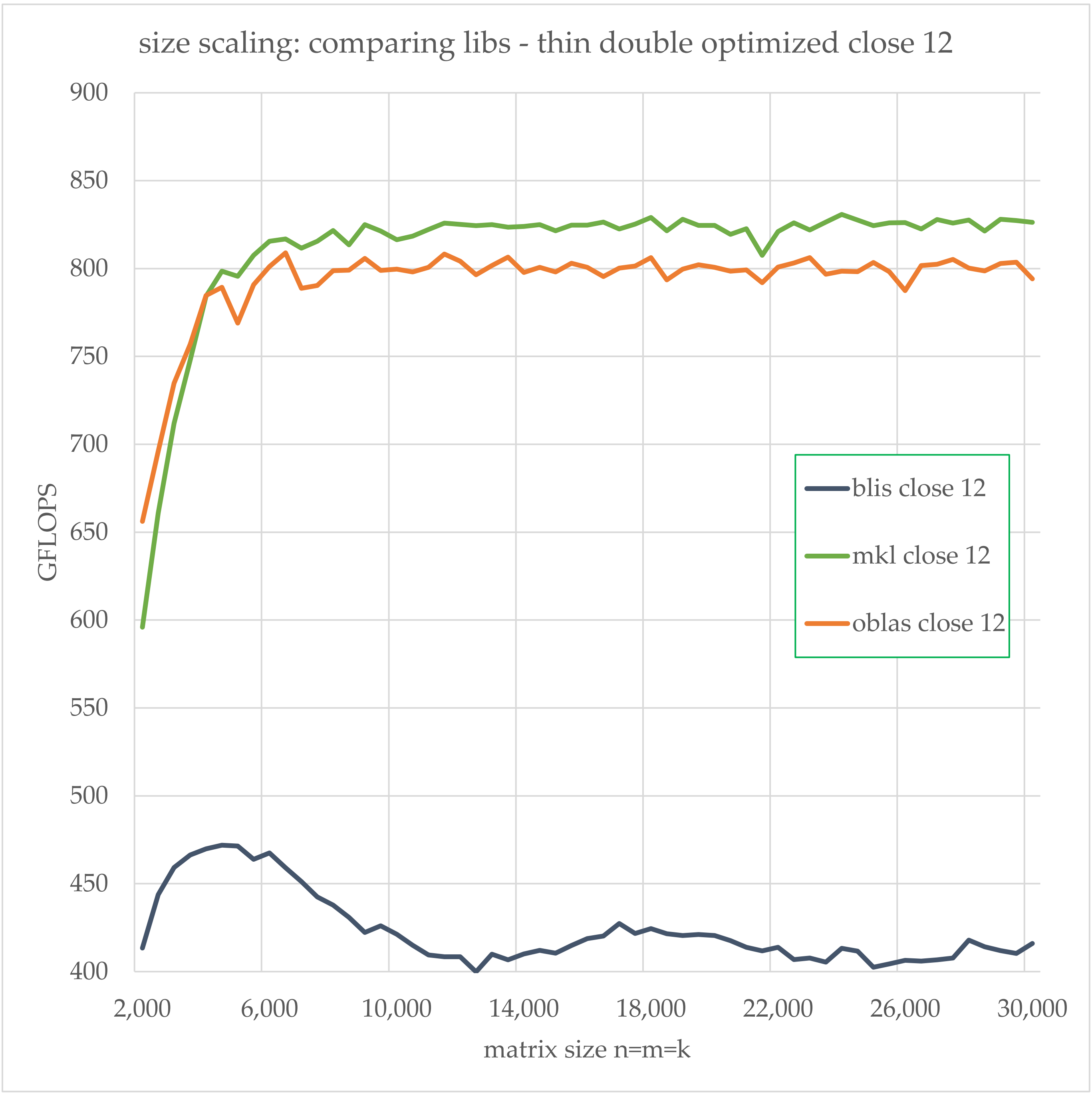


Figure 18: Scaling over matrix size, thin nodes, double precision (dgemm), compiled with optimizations, binding policy "close", 12 cores.

The blis library confirm its poor performance on epyc nodes, compared to the other two. On the other side mkl is again performing slightly better than oblas.

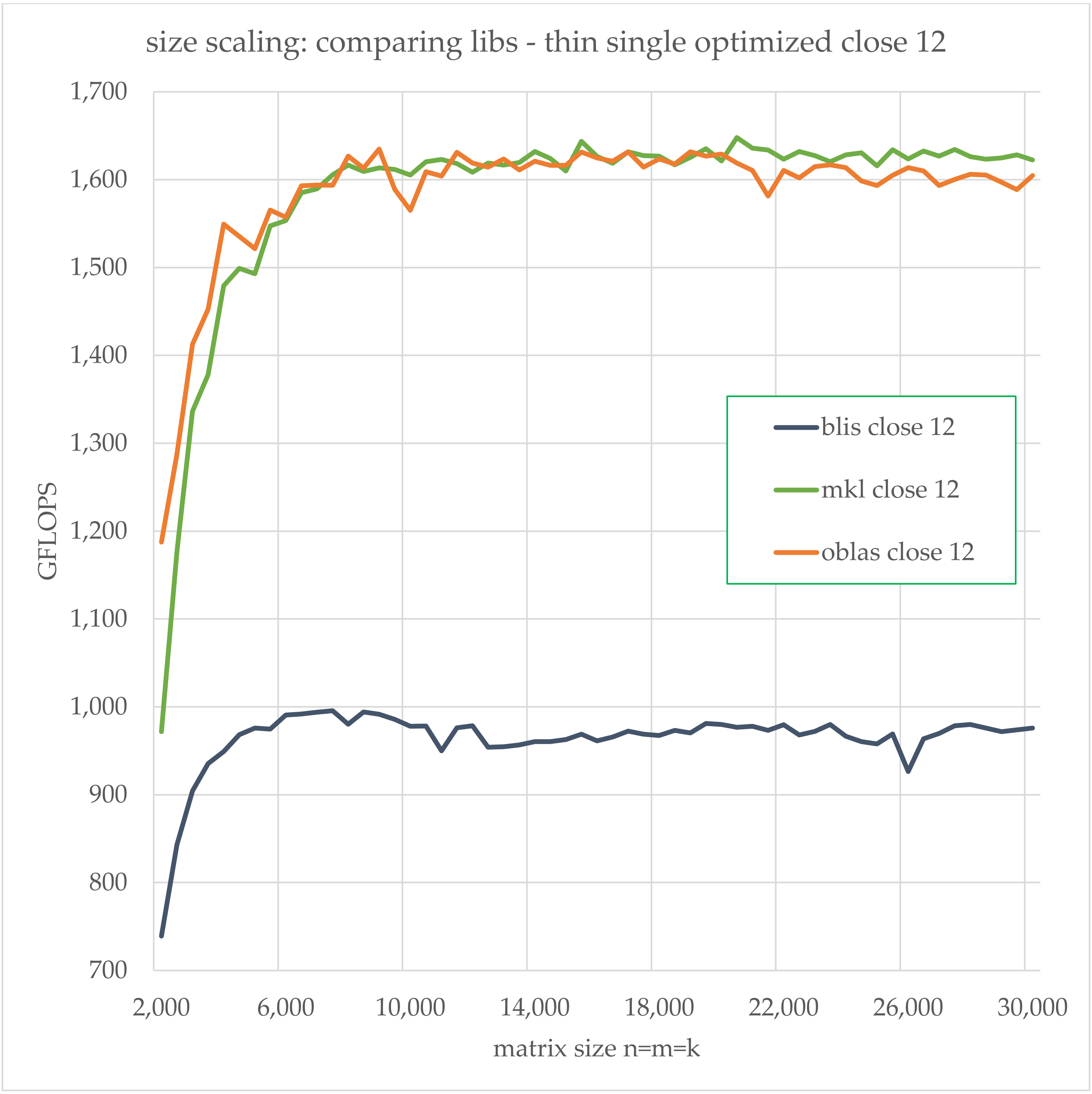


Figure 19: Scaling over matrix size, thin nodes, single precision (sgemm), compiled with optimizations, binding policy "close", 12 cores.

In the case of single precision, blis is still performing worse than the other libraries but its trend is more constant after the initial ramp.

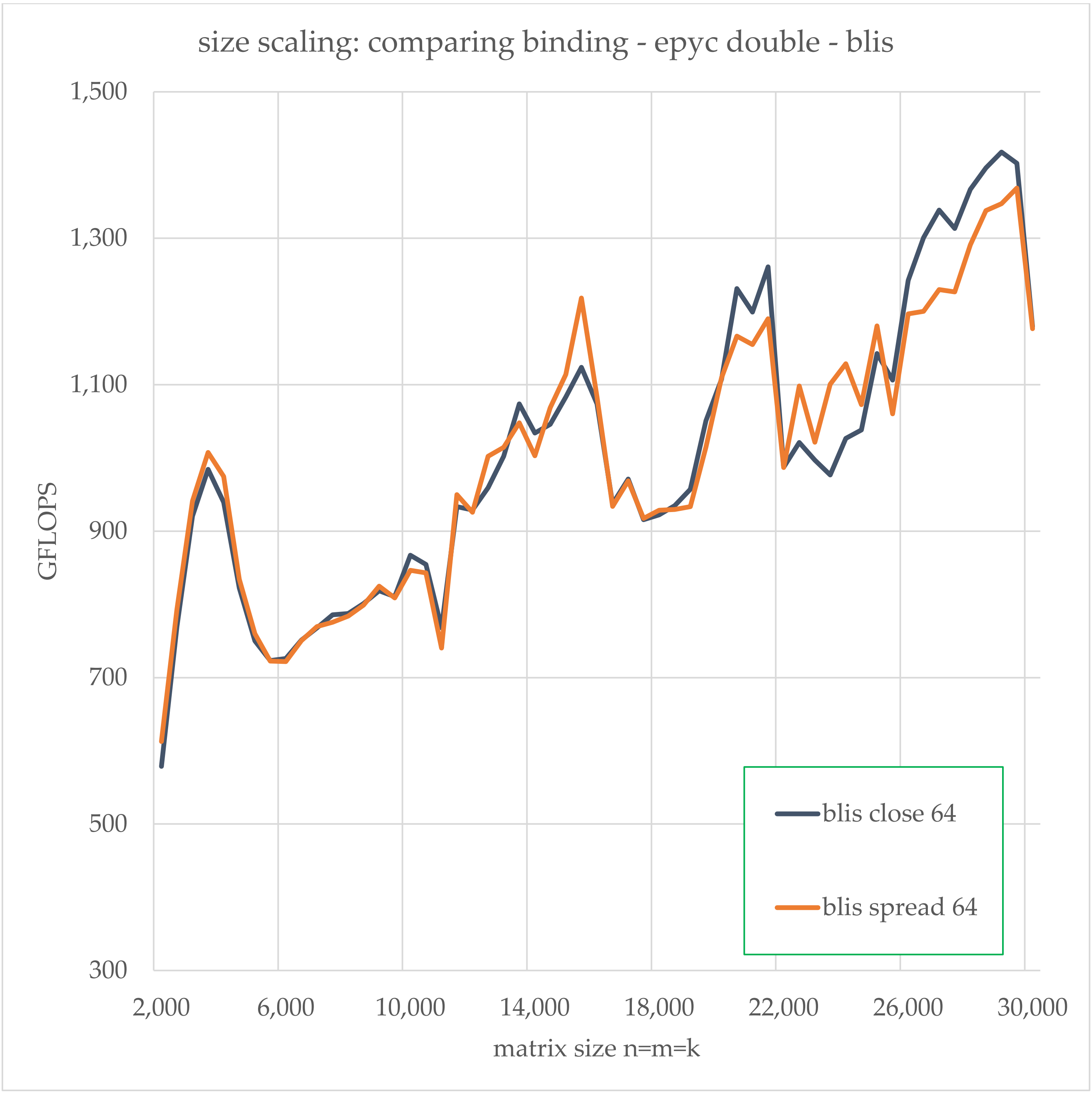


Figure 20: Scaling over matrix size, epyc nodes, double precision (dgemm), compiled with optimizations, 64 cores, blis library, comparing binding policy "close" to "spread".

Repeating the executions with the binding policy set to “spread” instead of “close” we cannot appreciate much difference.

```
export OMP_PLACES=cores
export OMP_PROC_BIND=close
...
export OMP_PLACES=cores
export OMP_PROC_BIND=spread
```

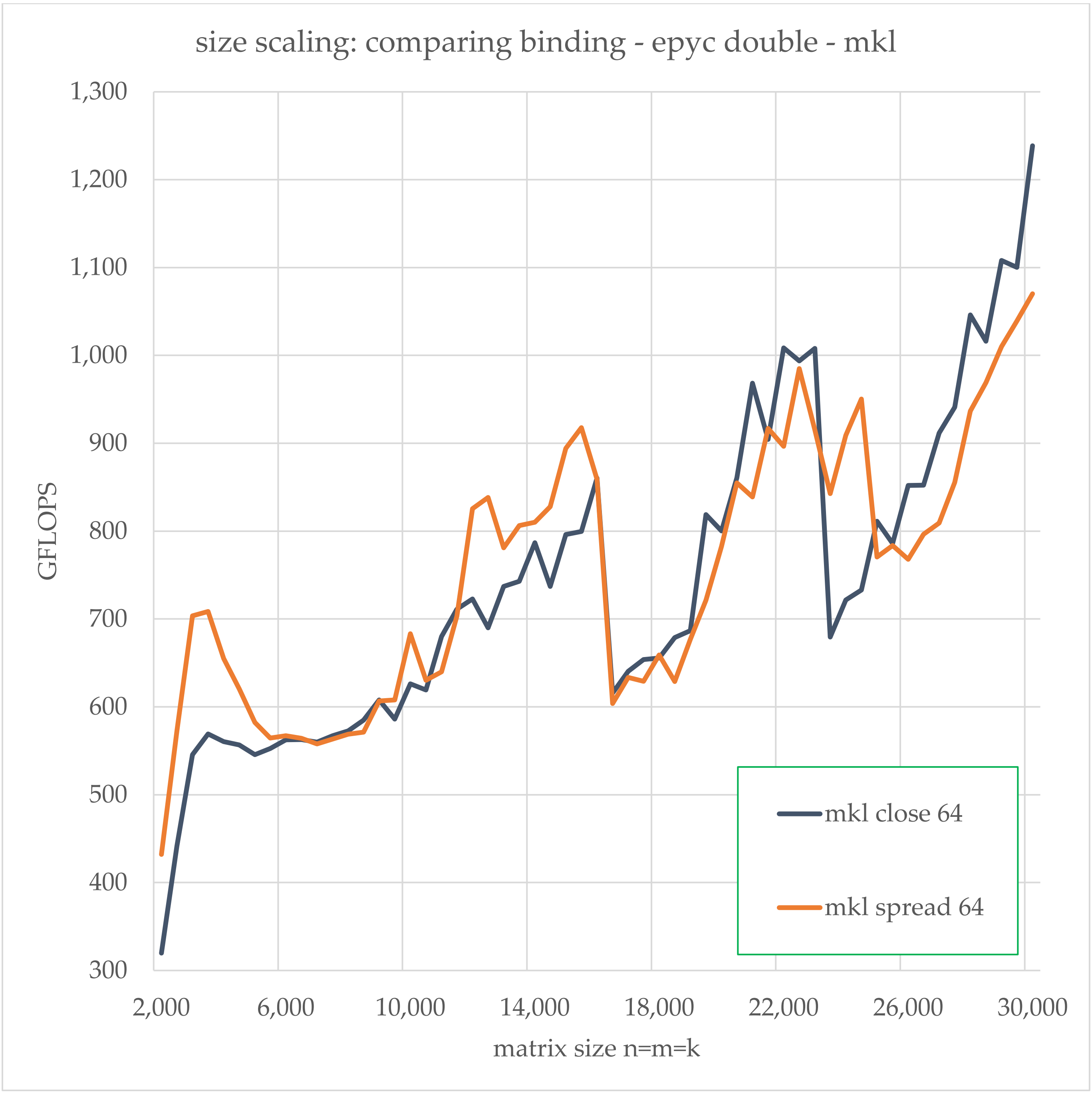


Figure 21: Scaling over matrix size, epyc nodes, double precision (dgemm), compiled with optimizations, 64 cores, mkl library, comparing binding policy "close" to "spread".

Some better performances are read for the lower range of the data sizes when using the “spread” binding.

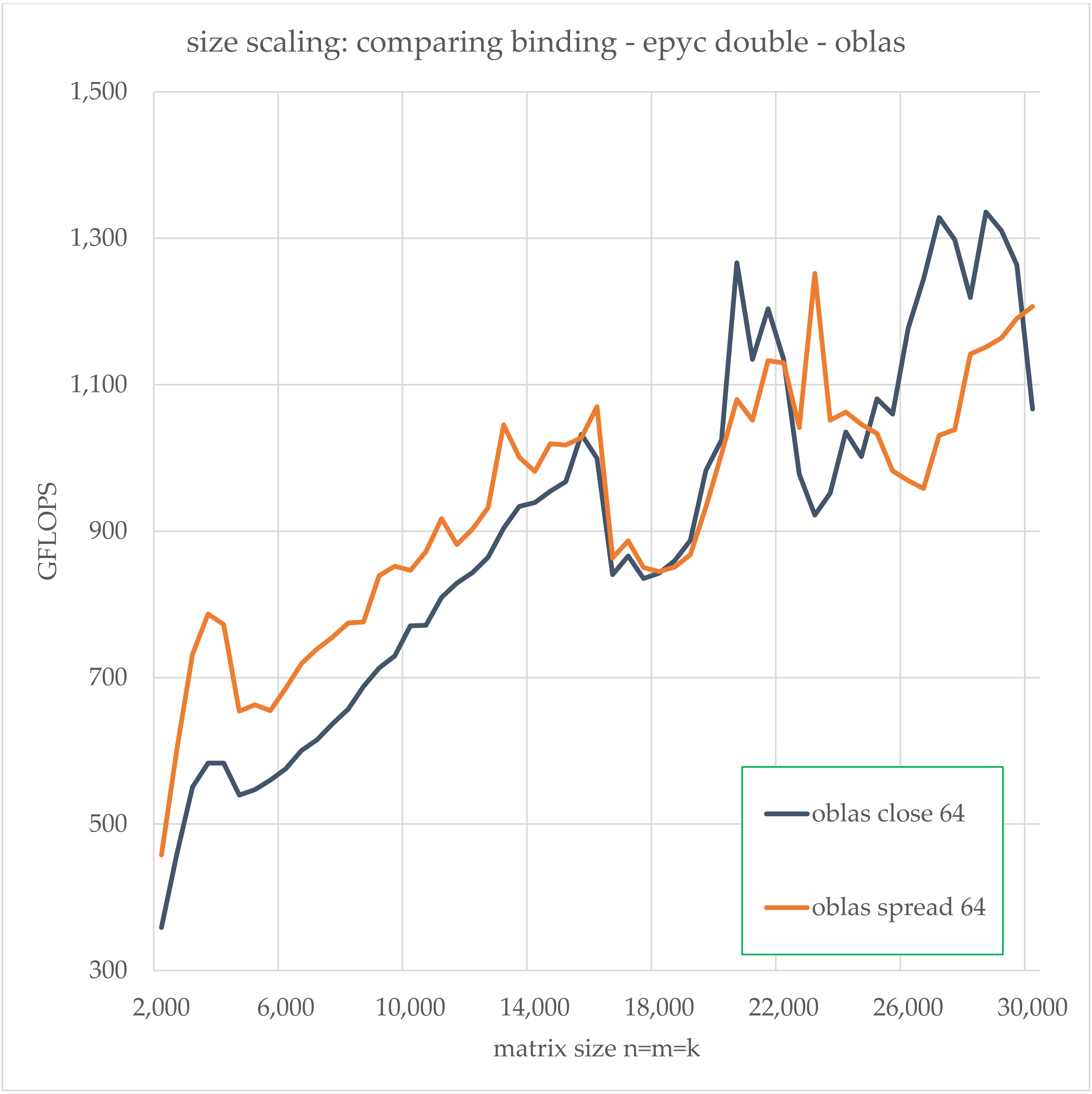


Figure 22: Scaling over matrix size, epyc nodes, double precision (dgemm), compiled with optimizations, 64 cores, oblas library, comparing binding policy "close" to "spread".

Again, for the lower range of the data sizes when using the “spread” binding the program is performing better. The difference is quite clear, so a different usage of the available hardware is helping the program perform better.

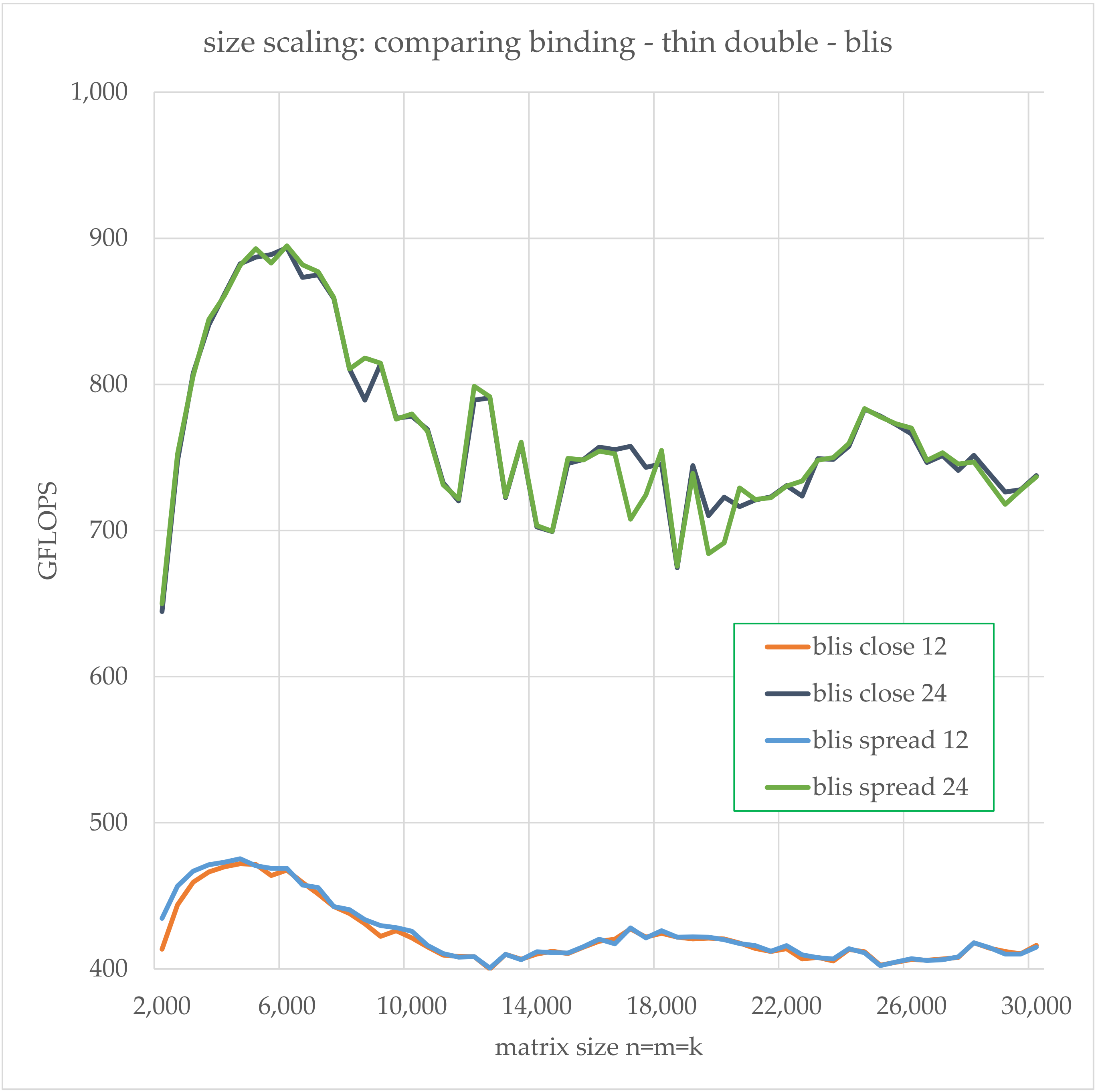


Figure 23: Scaling over matrix size, thin nodes, double precision (dgemm), compiled with optimizations, blis library, comparing binding policy "close" to "spread" and number of cores 12 to 24.

The performance can be compared using 12 or 24 cores on the thin nodes. Using more cores gives almost double performance values. The behaviour is similar but the “spread” policy does not give an advantage using 24 cores for the smaller sizes as it does with 12 only.
The scale must be taken in consideration as the blis library is performing with quite smaller values then the others.

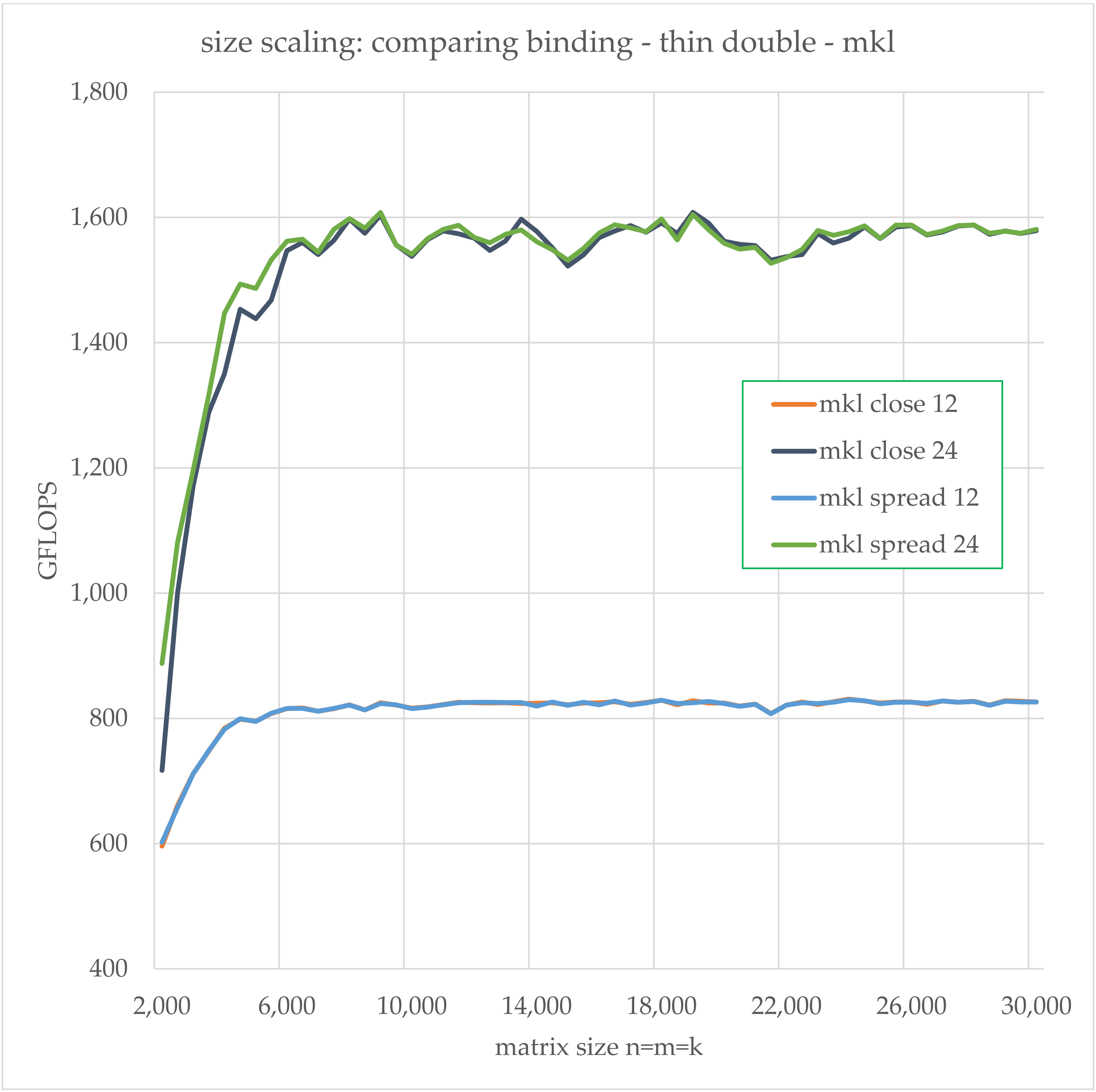


Figure 24: Scaling over matrix size, thin nodes, double precision (dgemm), compiled with optimizations, mkl library, comparing binding policy "close" to "spread" and number of cores 12 to 24.

The mkl library has the most constant behaviour after the initial ramp over the smaller size range.

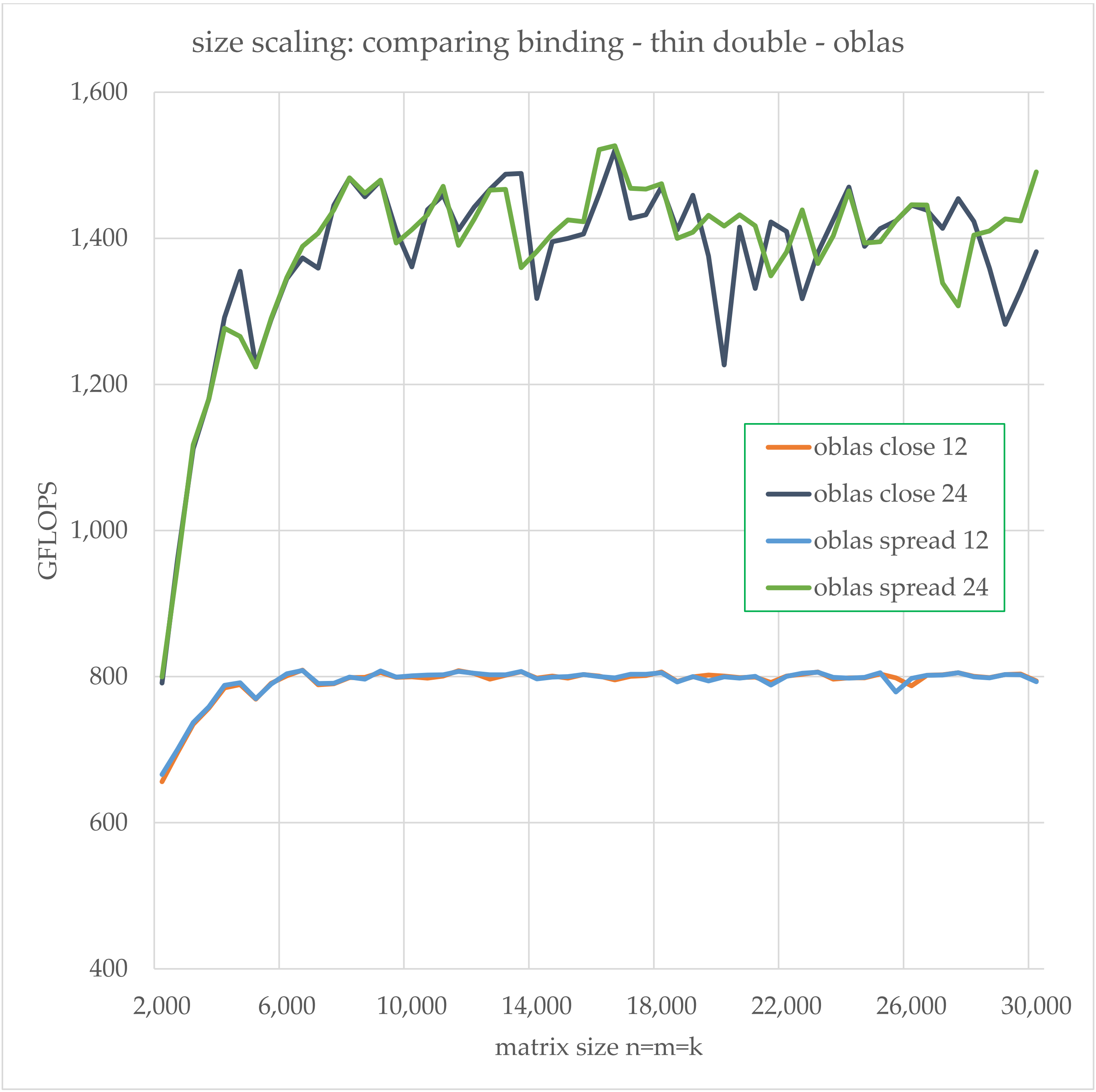


Figure 25: Scaling over matrix size, thin nodes, double precision (dgemm), compiled with optimizations, oblas library, comparing binding policy "close" to "spread" and number of cores 12 to 24.

The oblas library is performing with measured values that are lower by around 200GFLOPS relatively to mkl, and with a more irregular trend.

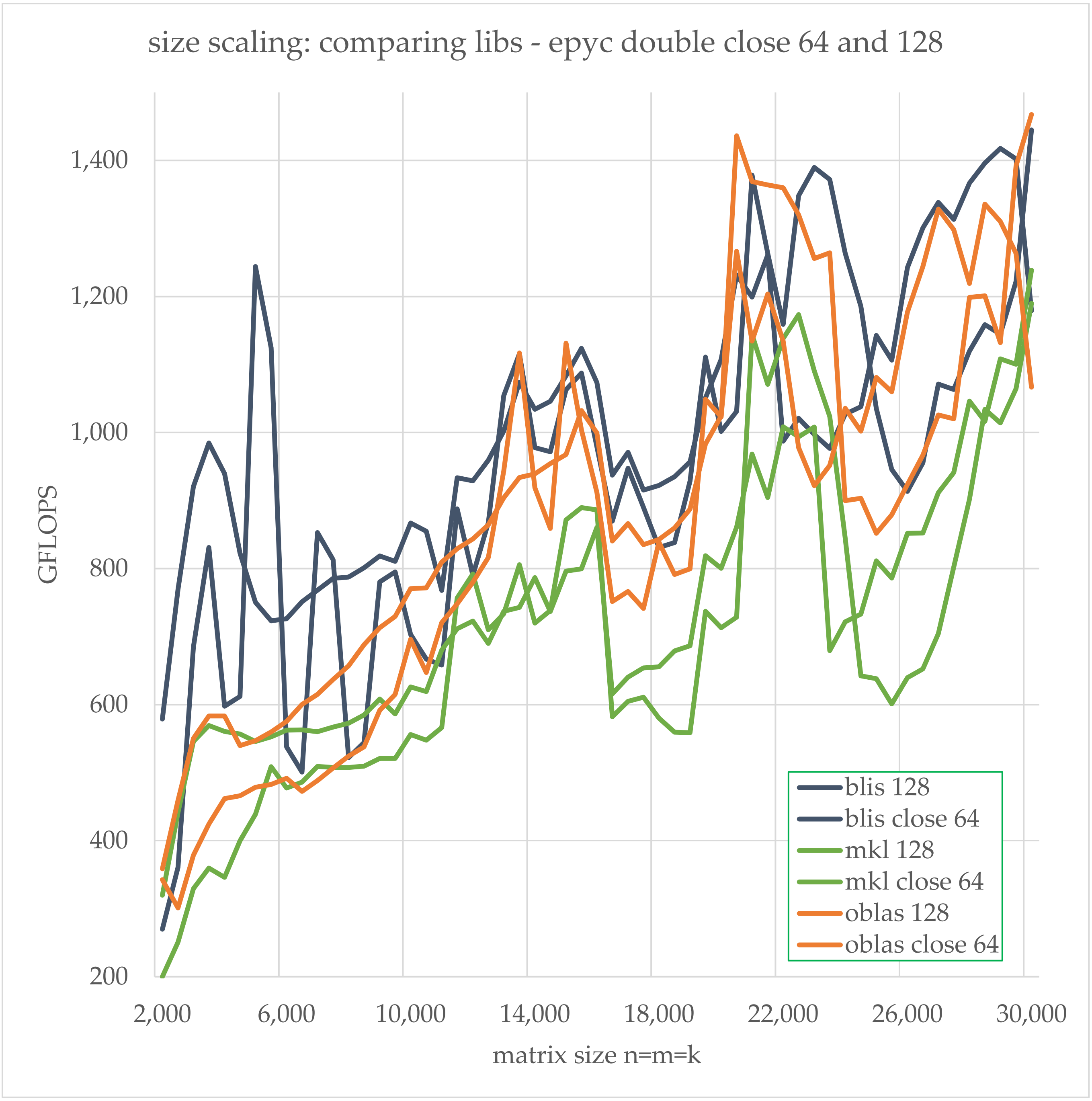


Figure 26: Scaling over matrix size, epyc nodes, double precision (dgemm), compiled with optimizations, binding policy "close", 64 and 128 cores.

Using a full epyc node with 128 cores produces more irregular plot lines, but the performance value range is similar.

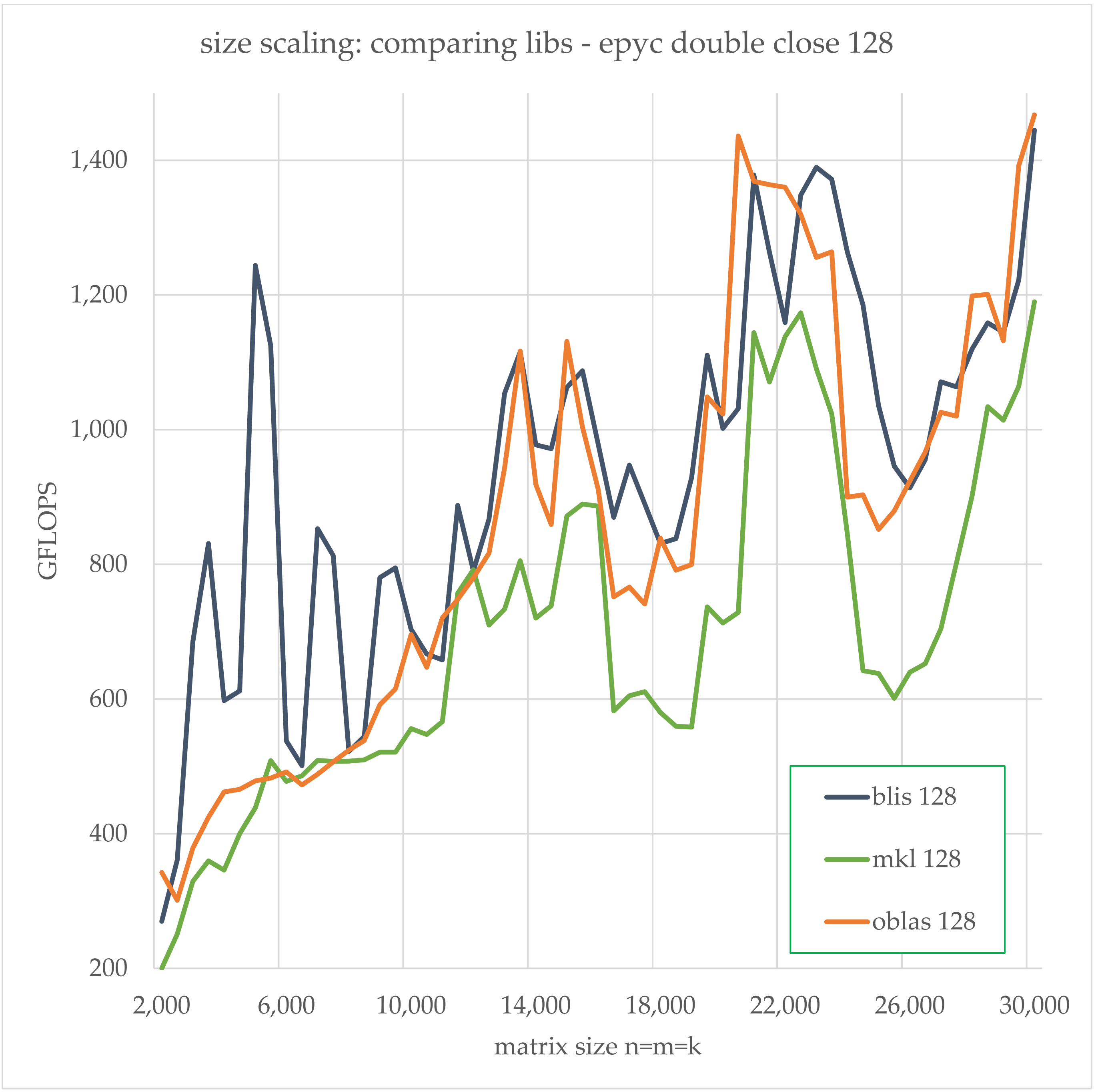


Figure 27: Scaling over matrix size, epyc nodes, double precision (dgemm), compiled with optimizations, binding policy "close", 128 cores.

Drawing only the values for the executions done with 128 cores it can be observed that the blis library has some peaks in the lower sizes range that can't be seen for the others. The following behaviour is similar, with blis and oblas alternating as the top performer.

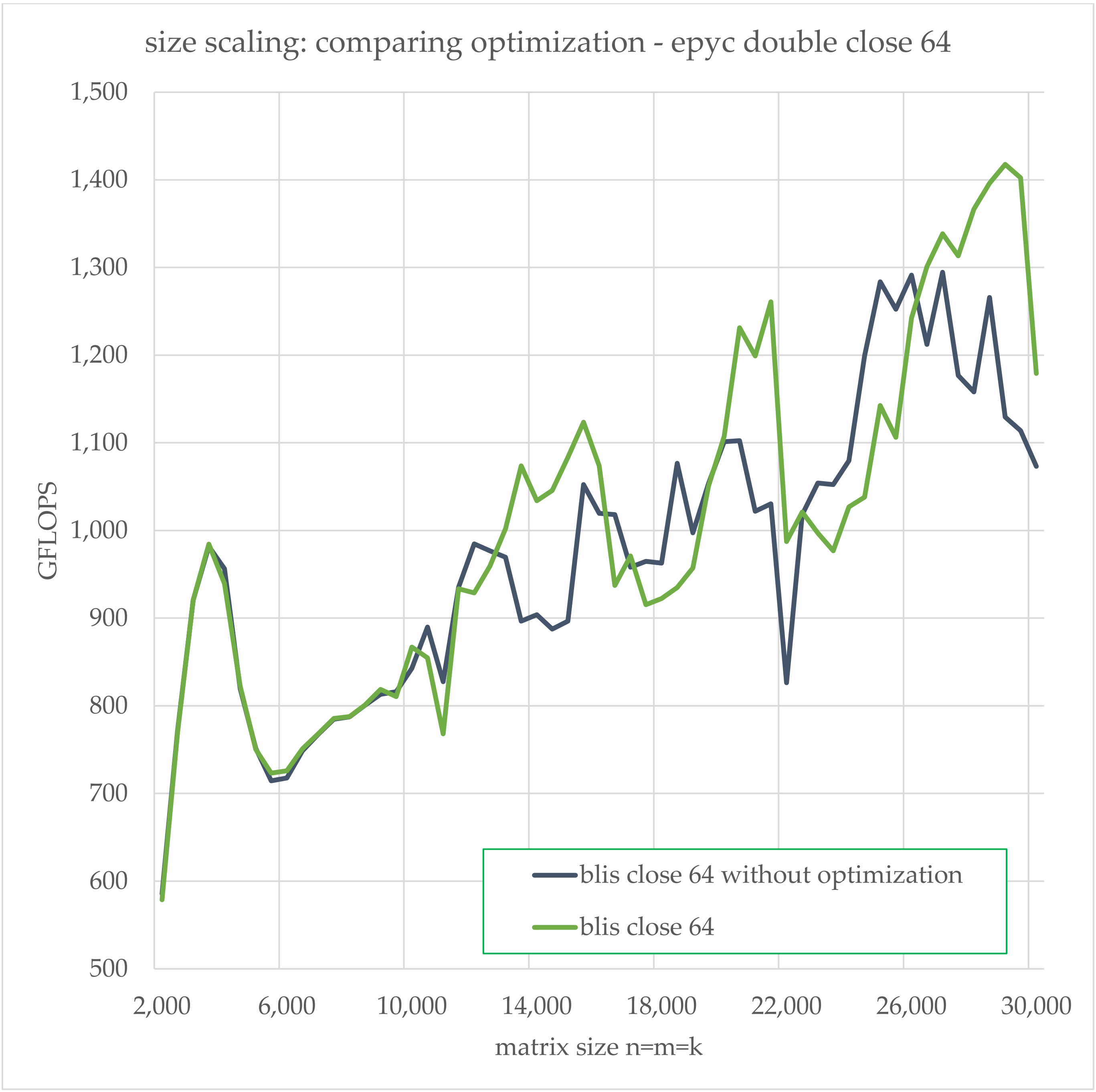


Figure 28: Scaling over matrix size, epyc nodes, double precision (dgemm), binding policy "close", 64 cores, blis library, comparing code compiled with or without optimizations.

Passing the optimization switches to the compiler enables the comparison of the performance of the program compiled with or without the optimizations. The performance is slightly better of the blis library when run on most of the biggest sizes range.

```
    gcc -O3 -march=native ...
```

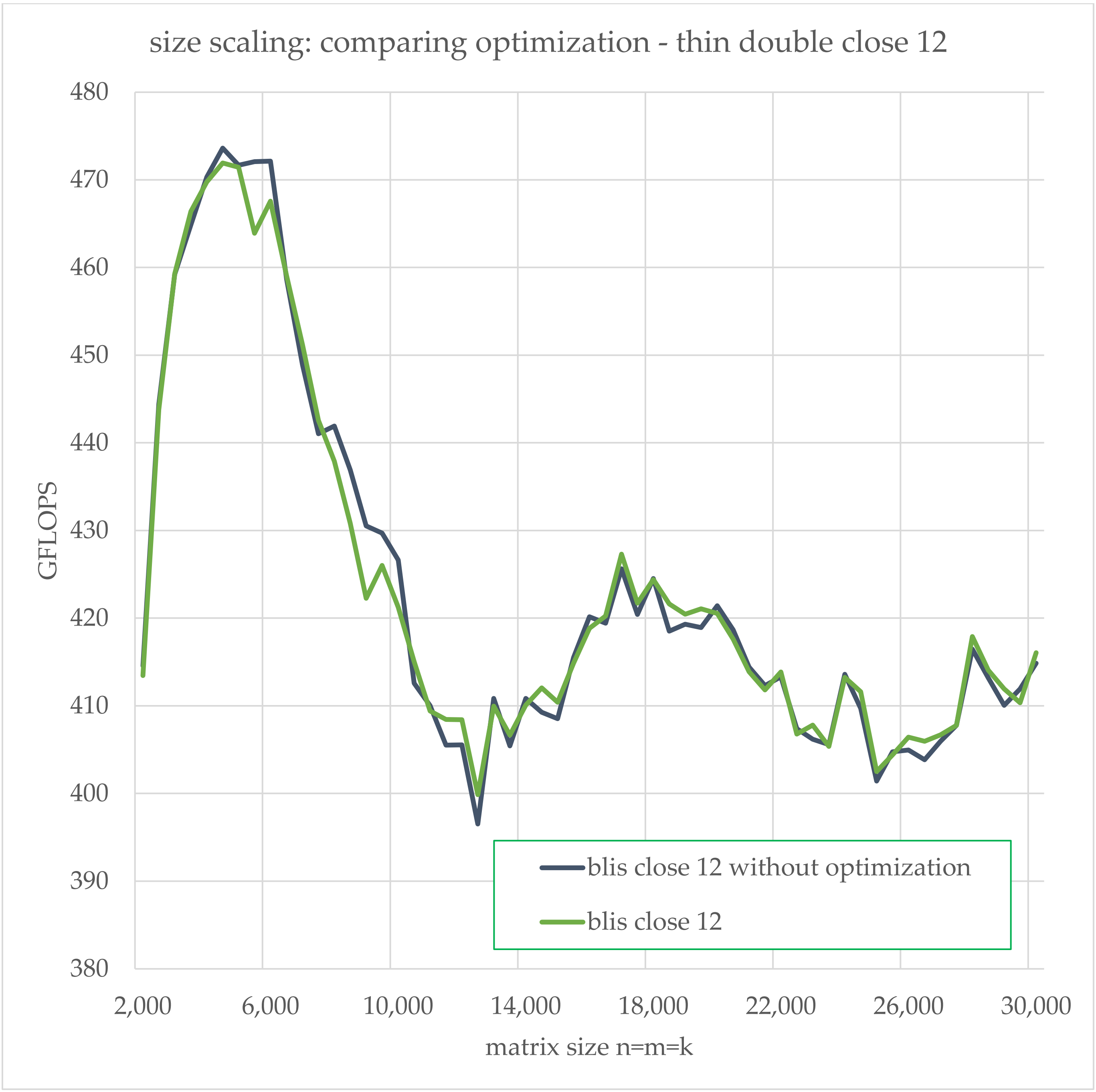


Figure 29: Scaling over matrix size, thin nodes, double precision (dgemm), binding policy "close", 12 cores, blis library, comparing code compiled with or without optimizations.

The activation of the compiler optimizations does not show benefits when the executions are run on the thin nodes.

Scalability over the number of cores at fixed size

The charts show the performance values measured in GFLOPS at fixed matrix size of 30,000. On the horizontal axes the number of cores is increased from 1 up to 128 on epyc and 24 on thin nodes.

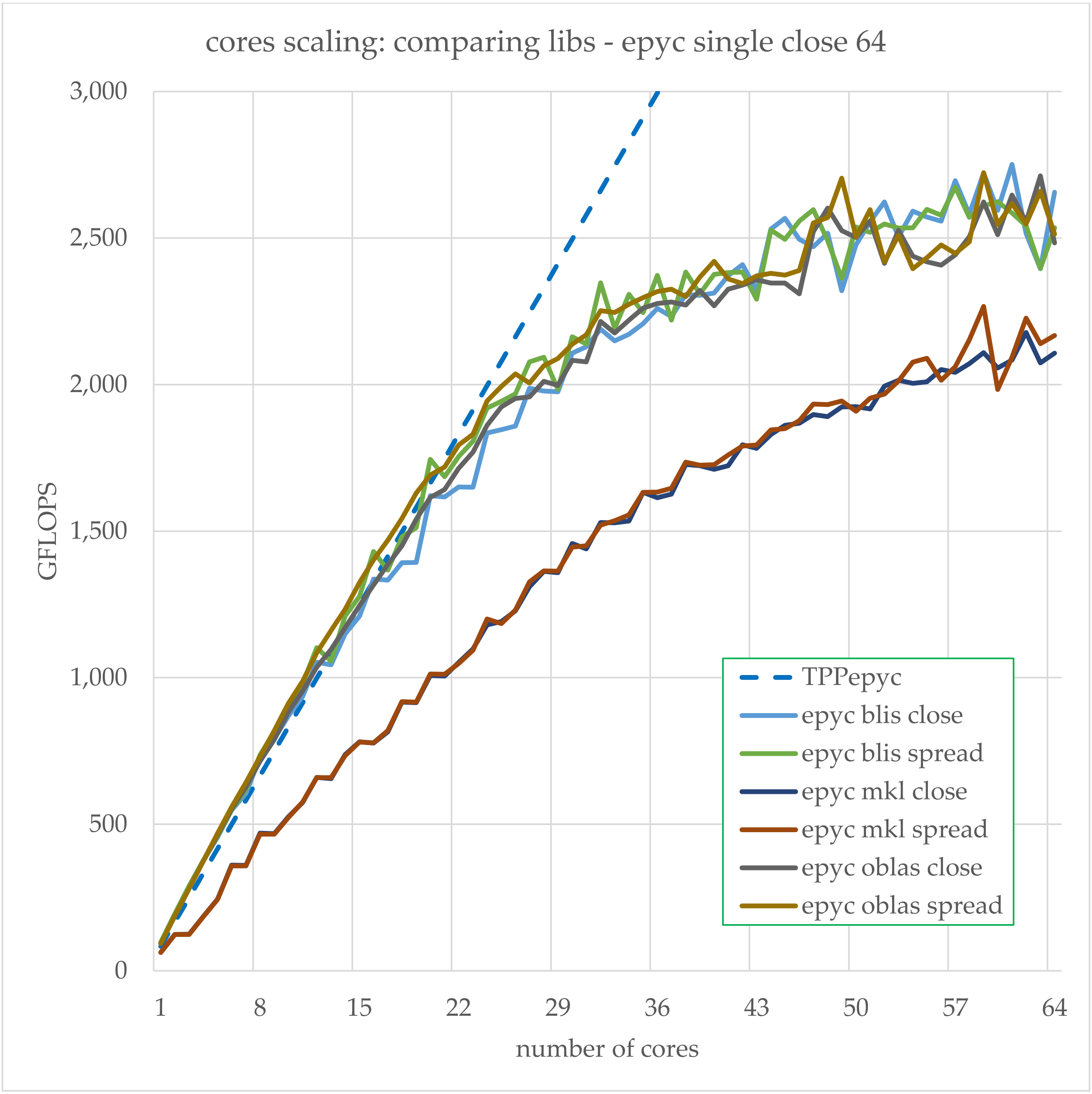


Figure 30: Scaling over number of cores up to 64, epyc nodes, single precision (sgemm), binding policy "close" and “spread”, comparing libraries to TPP Theoretical Peak Performance.

Scaling over the number of cores the performance is close to the Theoretical Peak Performance for blis and oblas libraries, where mkl gives lower values, almost closing the gap when using all of the 64 cores, as it has been seen when scaling over the data size.

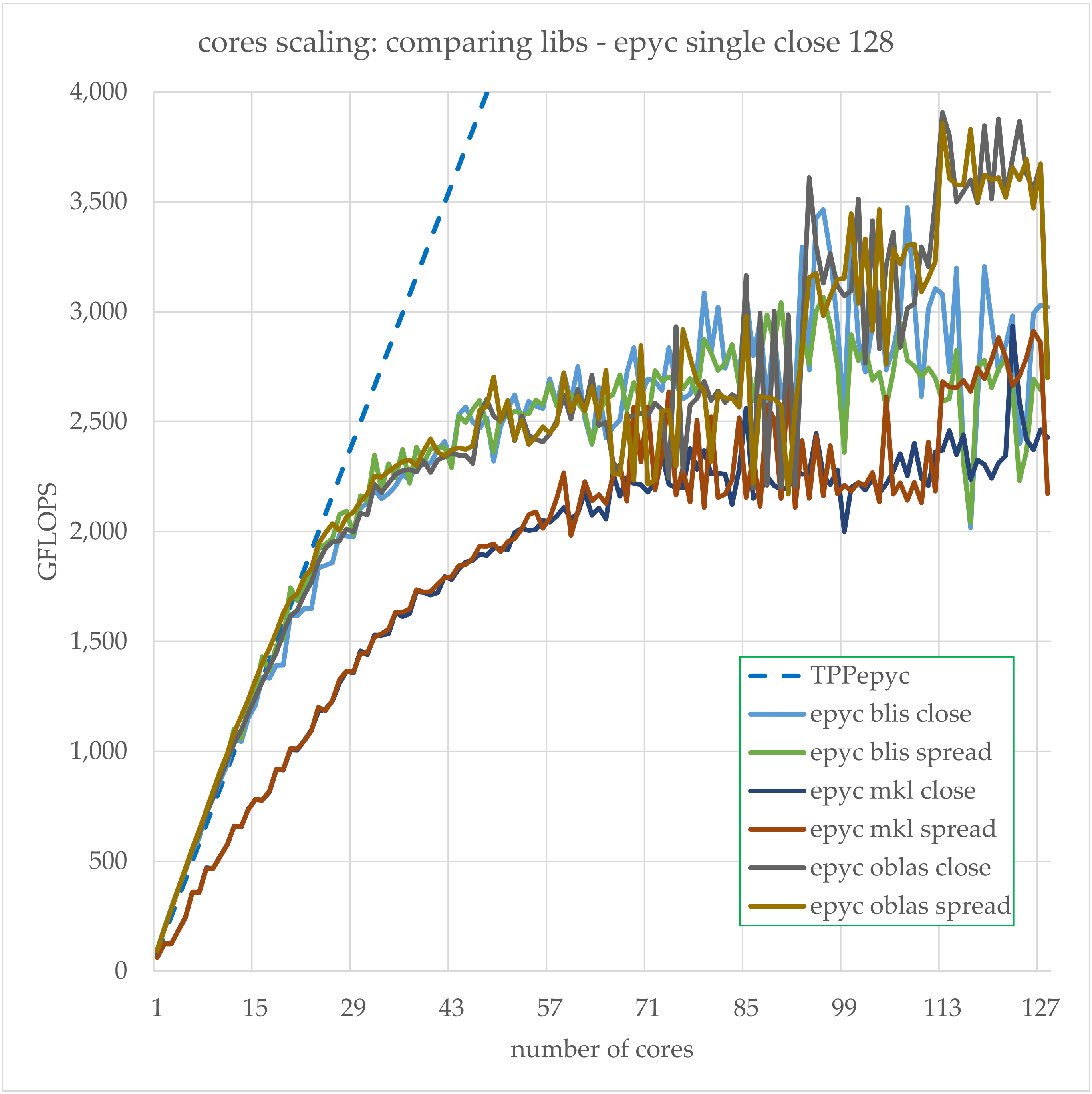


Figure 31: Scaling over number of cores up to 128, epyc nodes, single precision (sgemm), binding policy "close" and “spread”, comparing libraries to TPP Theoretical Peak Performance.

When scaling is extended to all the 128 cores of an epyc node, it can be observed that using more than 64 cores results in more irregular measured values. In this situation the oblas library gives better results in the higher range.

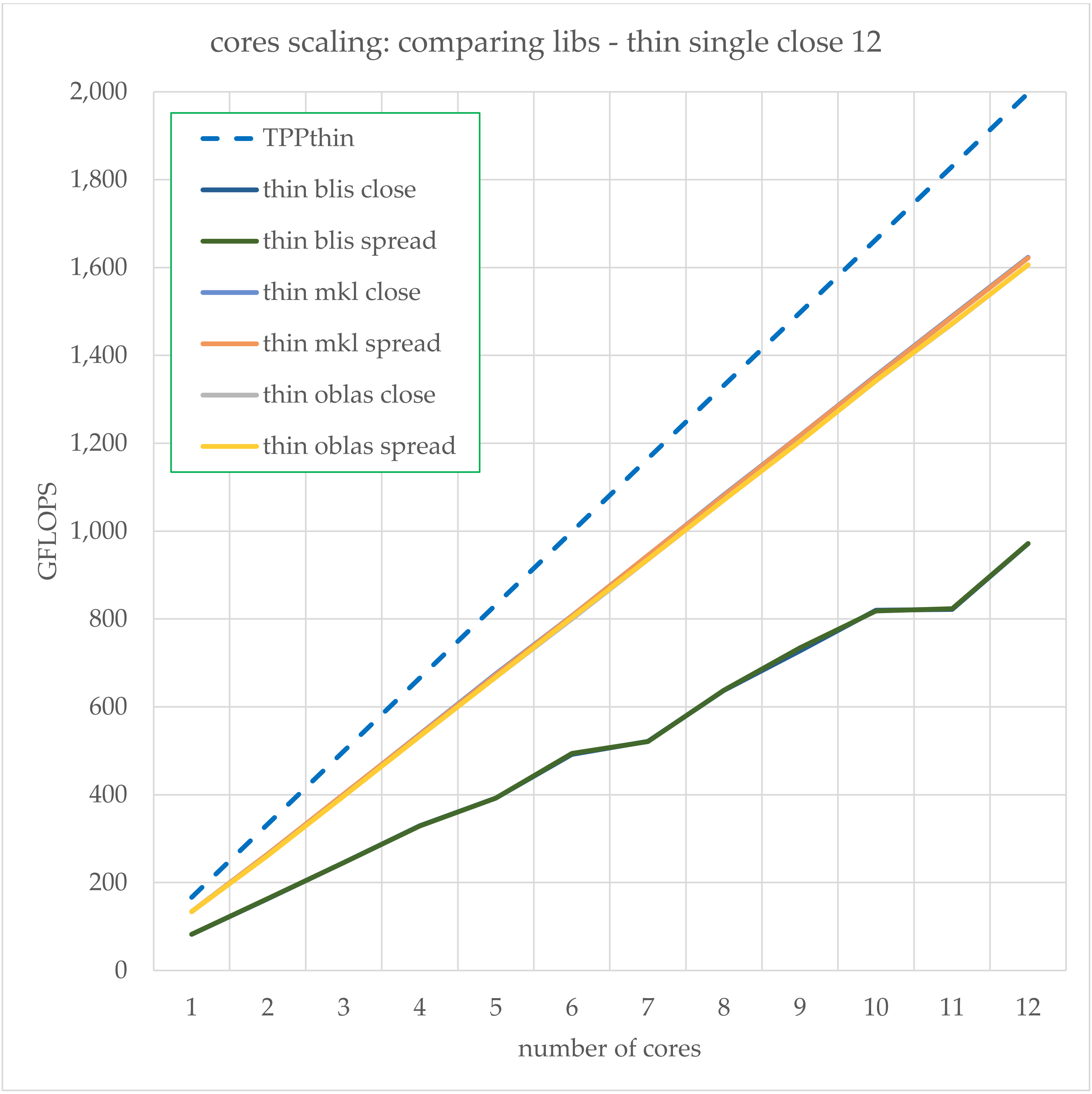


Figure 32: Scaling over number of cores up to 12, thin nodes, single precision (sgemm), binding policy "close" and "spread", comparing libraries to TPP Theoretical Peak Performance.

On the thin nodes there is almost no difference using the "spread" instead of the "close" binding policy. The blis library confirms it worse performance on thin nodes while the other two are almost identical.

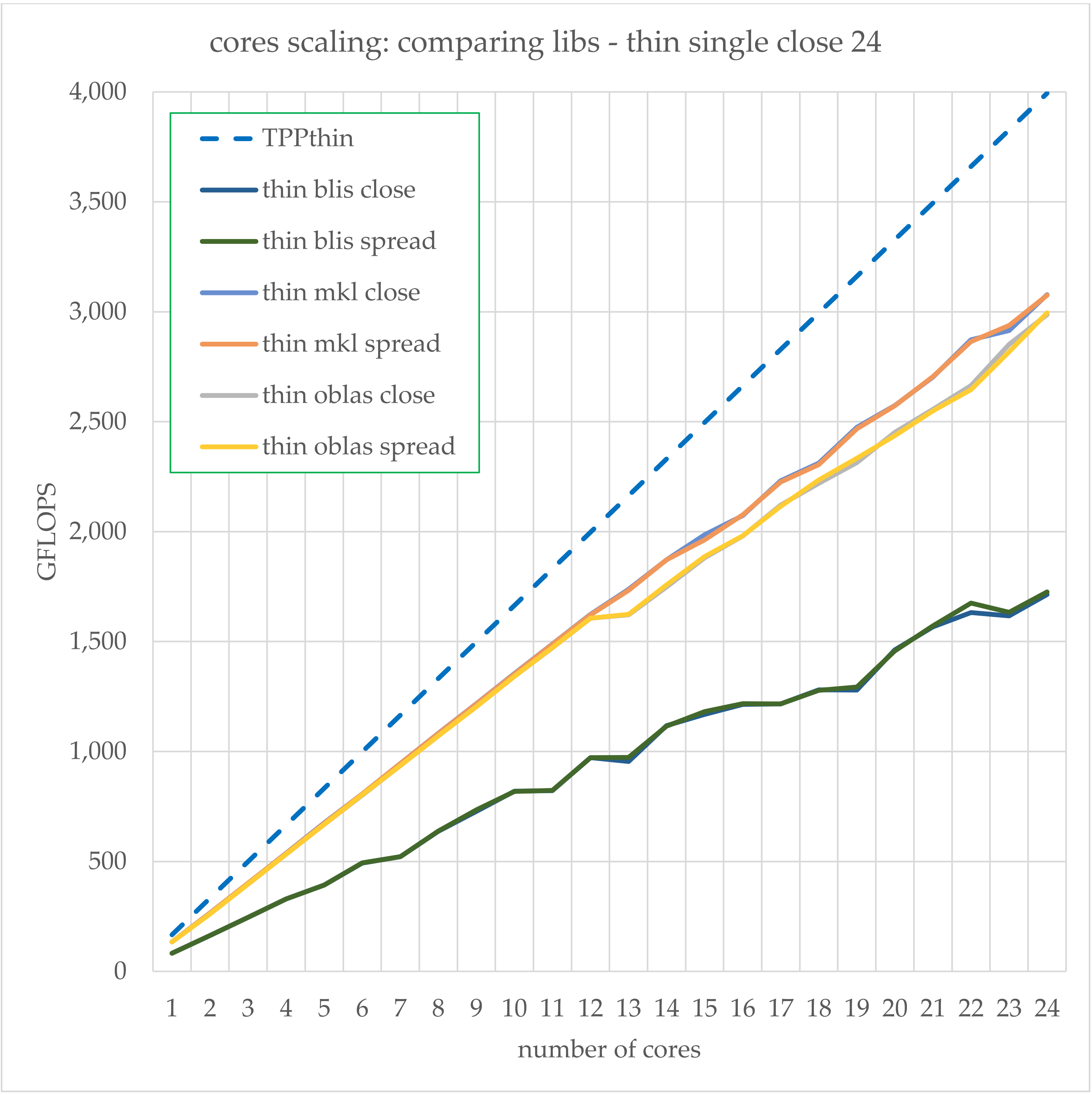


Figure 33: Scaling over number of cores up to 24, thin nodes, single precision (sgemm), binding policy "close" and “spread”, comparing libraries to TPP Theoretical Peak Performance.

When scaling is extended to all the 24 cores of a thin node, it can be observed that using more than 12 cores results in more irregular measured values. The mkl library shows an almost linear behaviour performing better then oblas in the range 13-24.

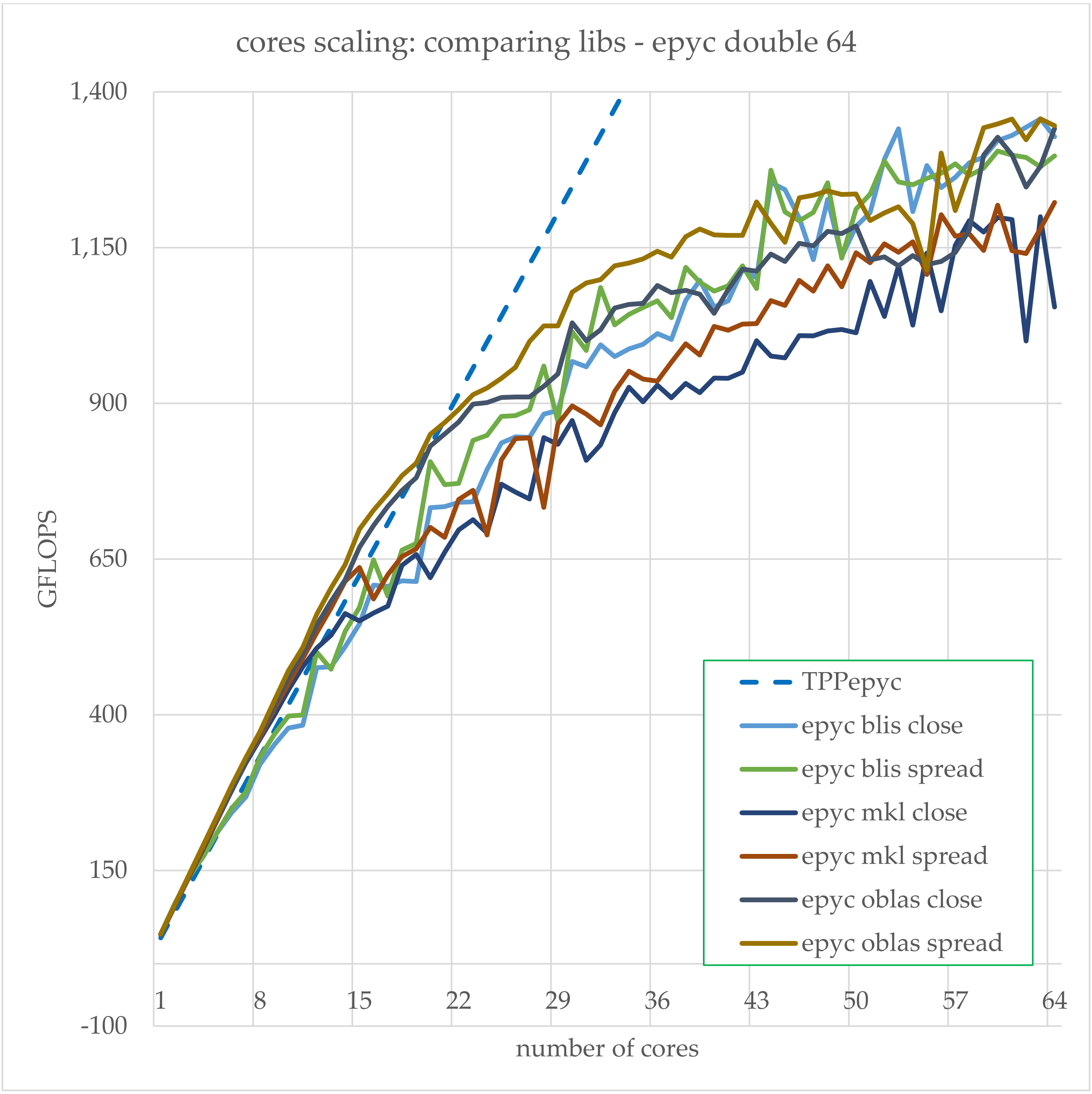


Figure 34: Scaling over number of cores up to 64, epyc nodes, double precision (dgemm), binding policy "close" and "spread", comparing libraries to TPP Theoretical Peak Performance.

Repeating the experiment for the double precision program a similar behaviour is observed as for single precision with some more irregular lines.

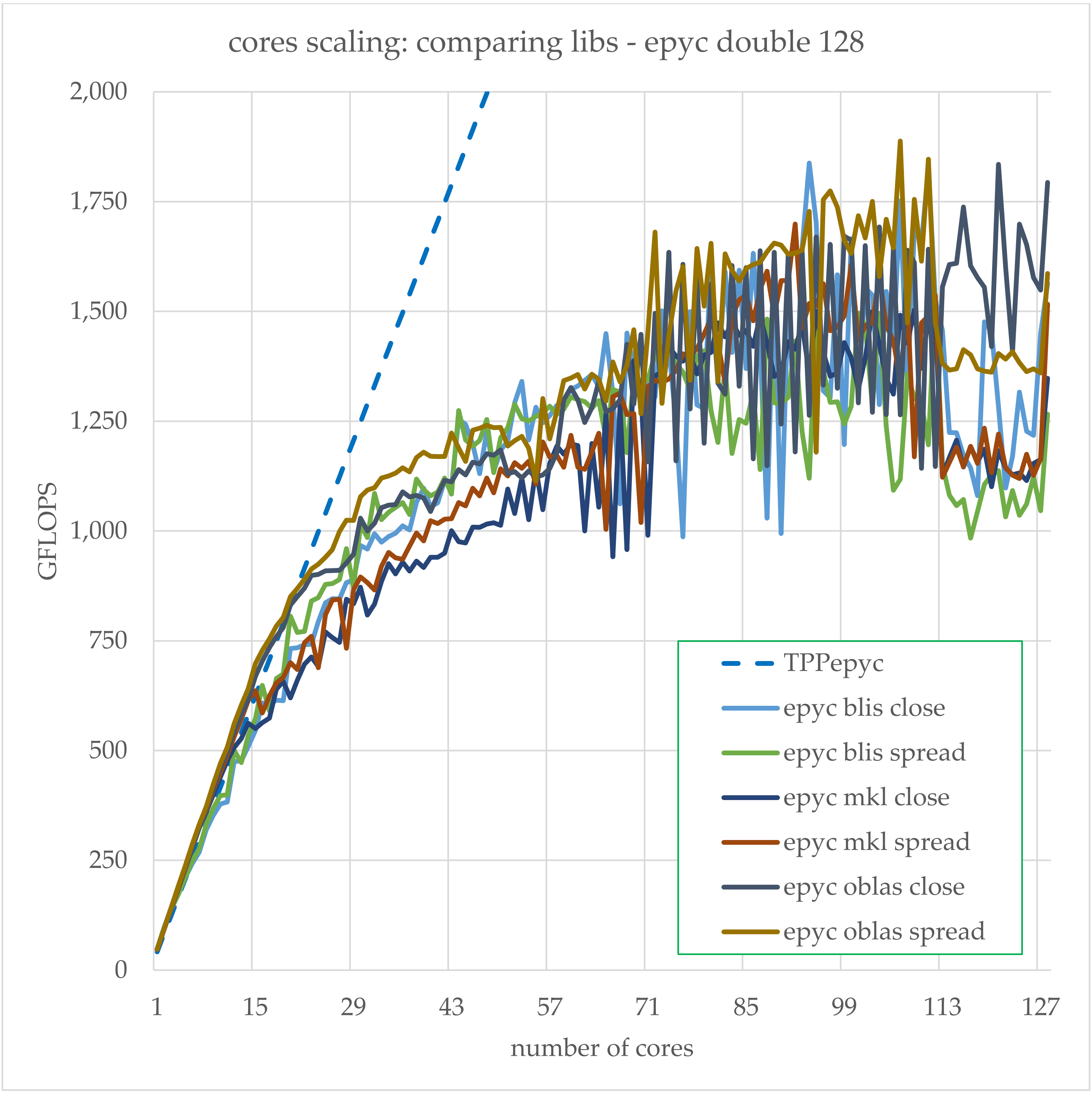


Figure 35: Scaling over number of cores up to 128, epyc nodes, double precision (dgemm), binding policy "close" and “spread”, comparing libraries to TPP Theoretical Peak Performance.

Extending to 128 cores shows a much more irregular plot in the higher range. Some more charts are added later to make clearer the behaviour of single libraries avoiding the overlapping observed in this one.

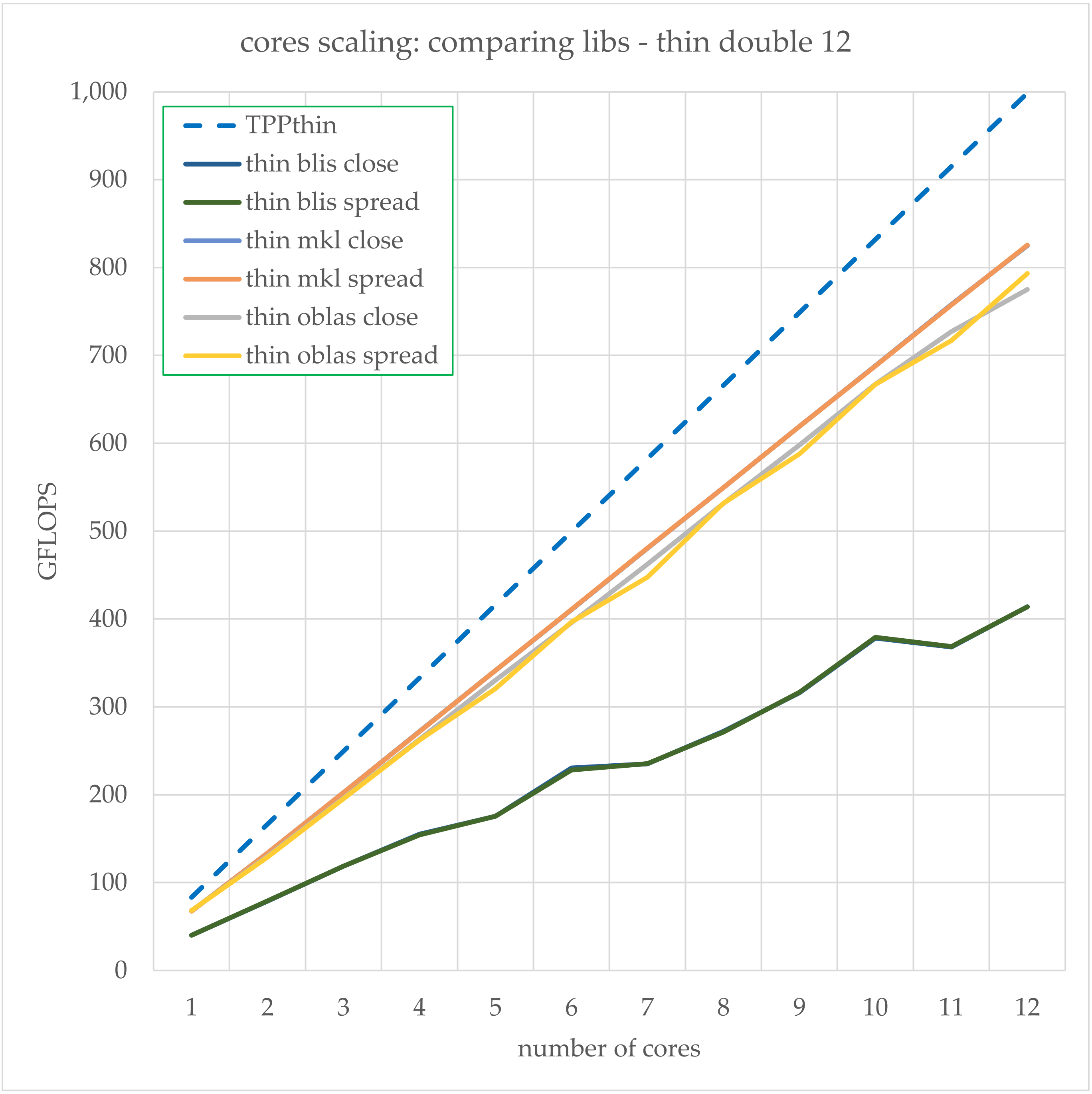


Figure 36: Scaling over number of cores up to 12, thin nodes, double precision (dgemm), binding policy "close" and “spread”, comparing libraries to TPP Theoretical Peak Performance.

On the thin nodes the behaviour for double precision is almost the same.

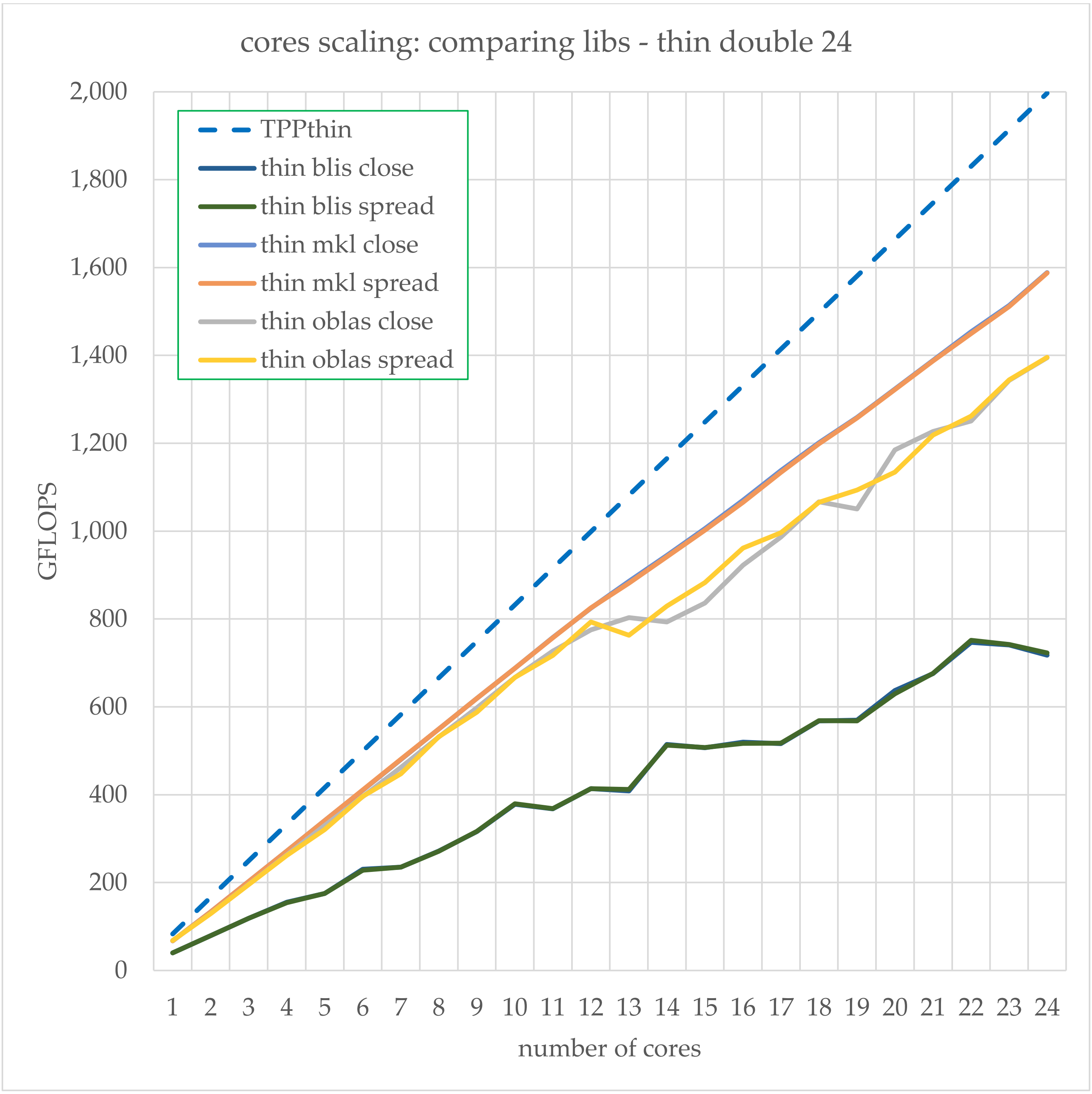


Figure 37: Scaling over number of cores up to 24, thin nodes, double precision (dgemm), binding policy "close" and “spread”, comparing libraries to TPP Theoretical Peak Performance.

In double precision it is more remarked the difference between mkl and oblas with the latter showing a lower performance when using more then 12 cores as observed for single precision.

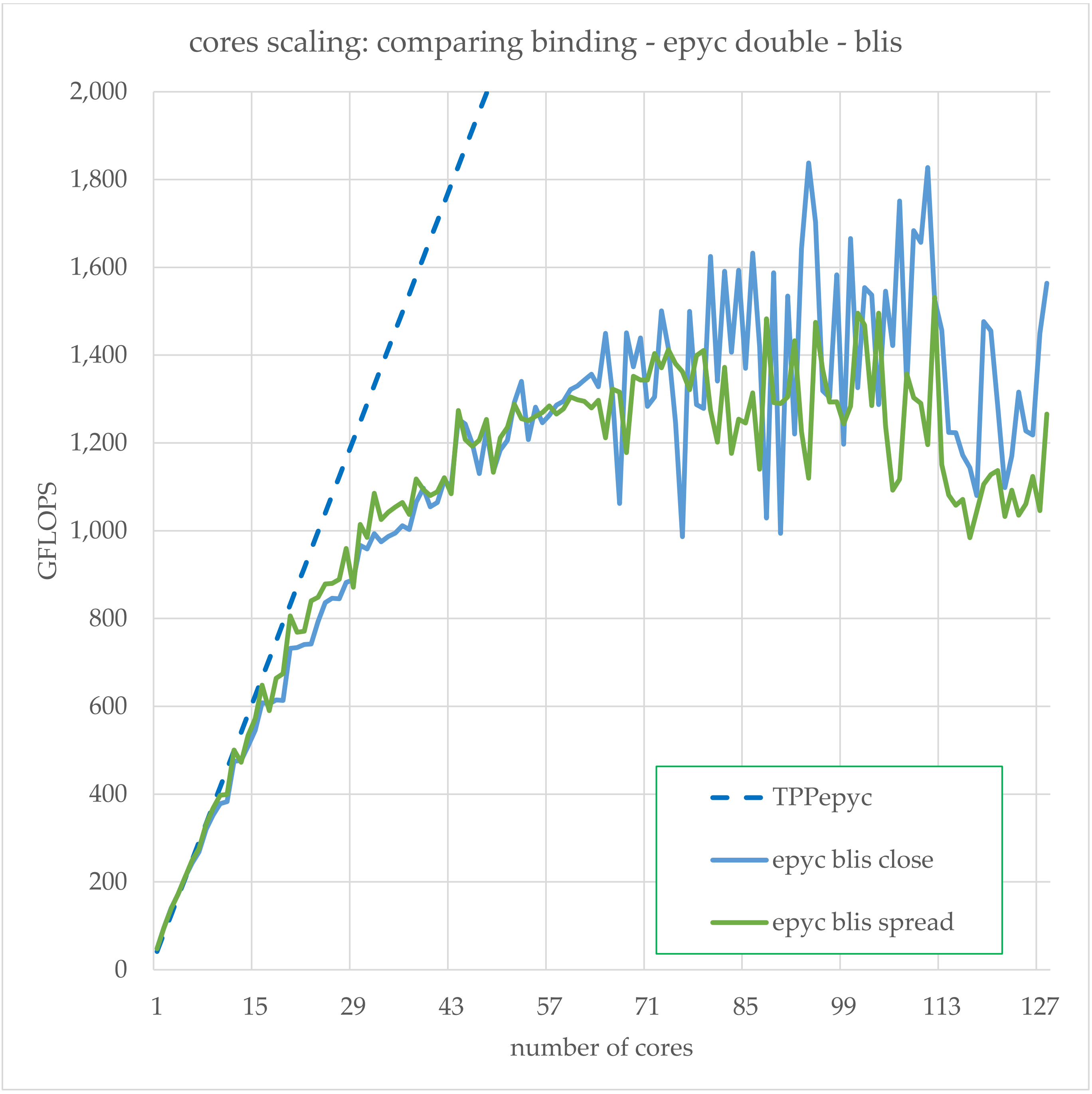


Figure 38: Scaling over number of cores up to 128, epyc nodes, double precision (dgemm), comparing binding policy "close" and “spread”, blis library.

In these plots each library is shown by itself to help distinguish the drawing of irregular values.

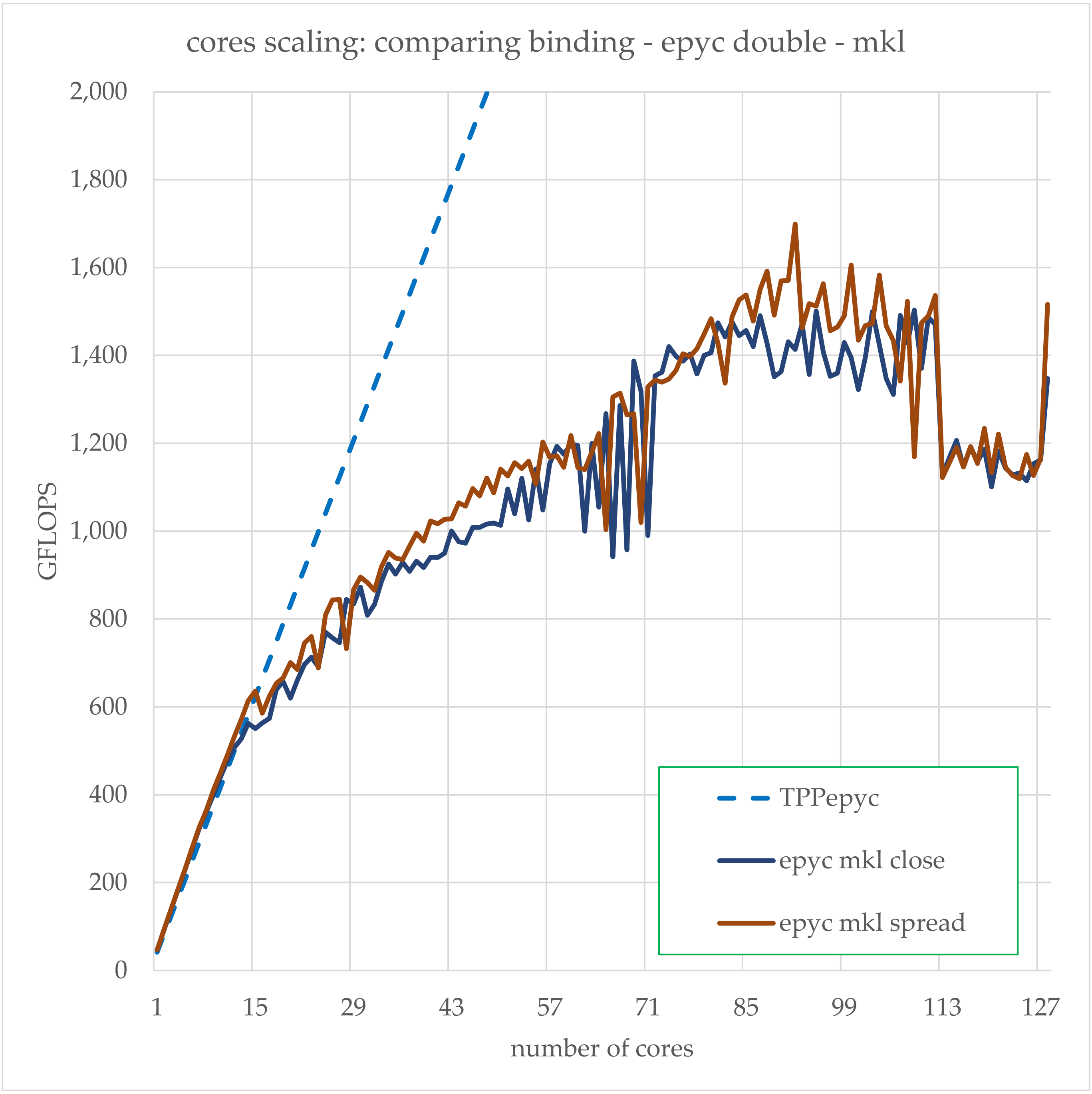


Figure 39: Scaling over number of cores up to 128, epyc nodes, double precision (dgemm), comparing binding policy "close" and "spread", mkl library.

The mkl library shows a more irregular trend in the range 60-70 cores and a drop after 112 cores.

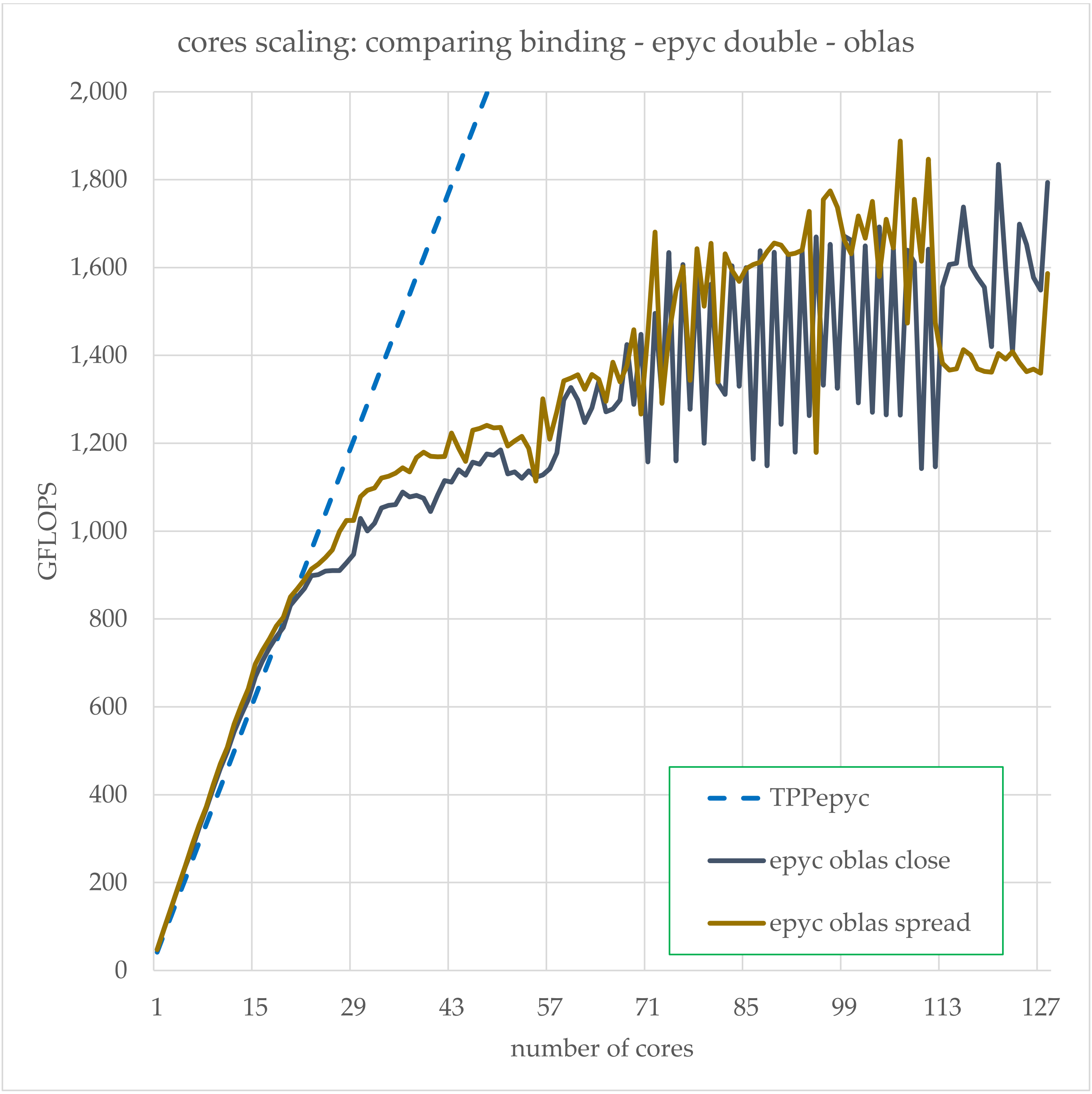


Figure 40: Scaling over number of cores up to 128, epyc nodes, double precision (dgemm), comparing binding policy "close" and “spread”, oblas library.

The oblas library is more irregular with more then 70 cores used, with a difference of behaviour when using the “spread” policy that gives better results in the range 24-54 and worse over 112.

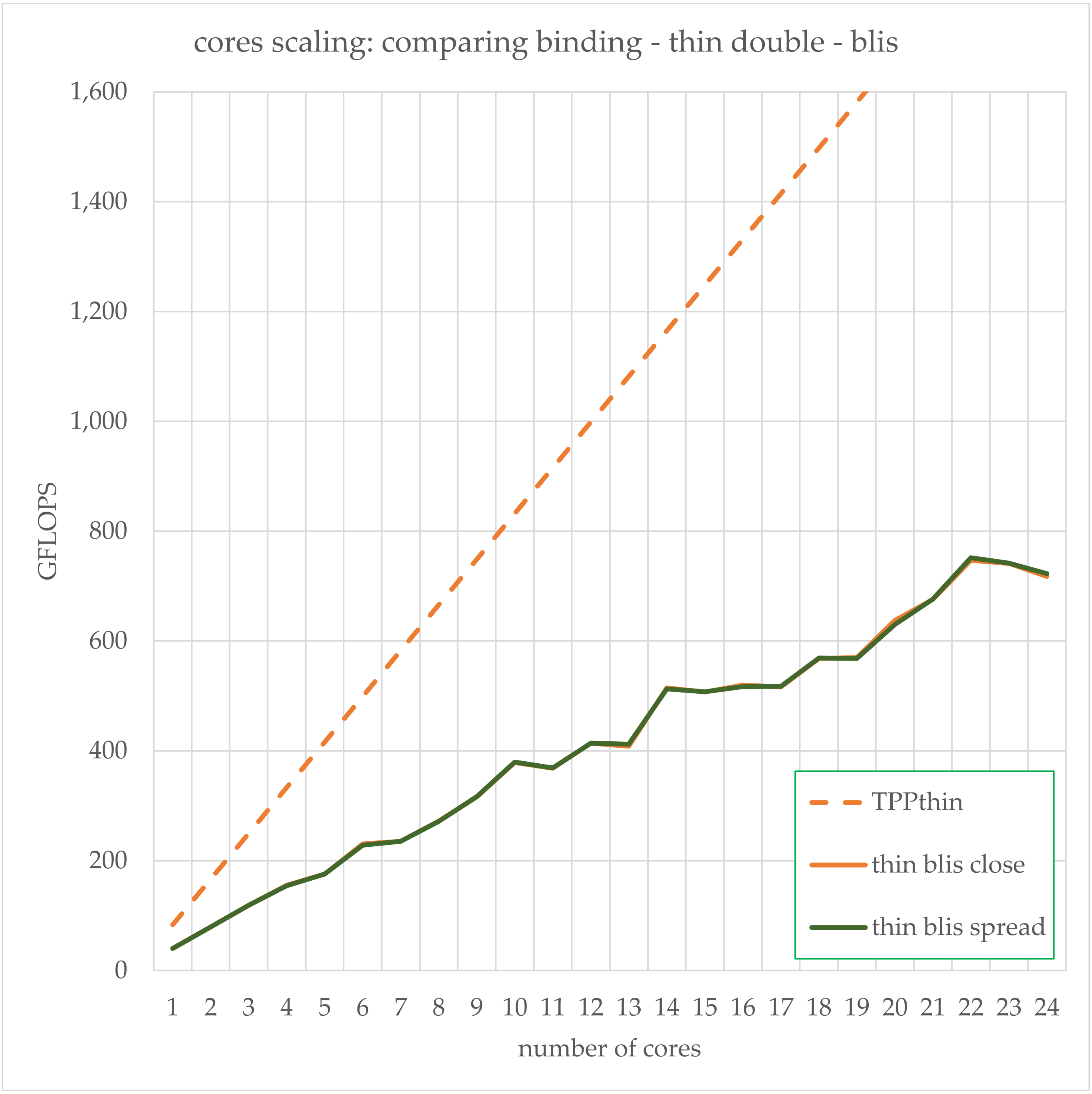


Figure 41: Scaling over number of cores up to 24, thin nodes, double precision (dgemm), comparing binding policy "close" and "spread", blis library.

The plot confirms that the binding policy is not effective on thin nodes.

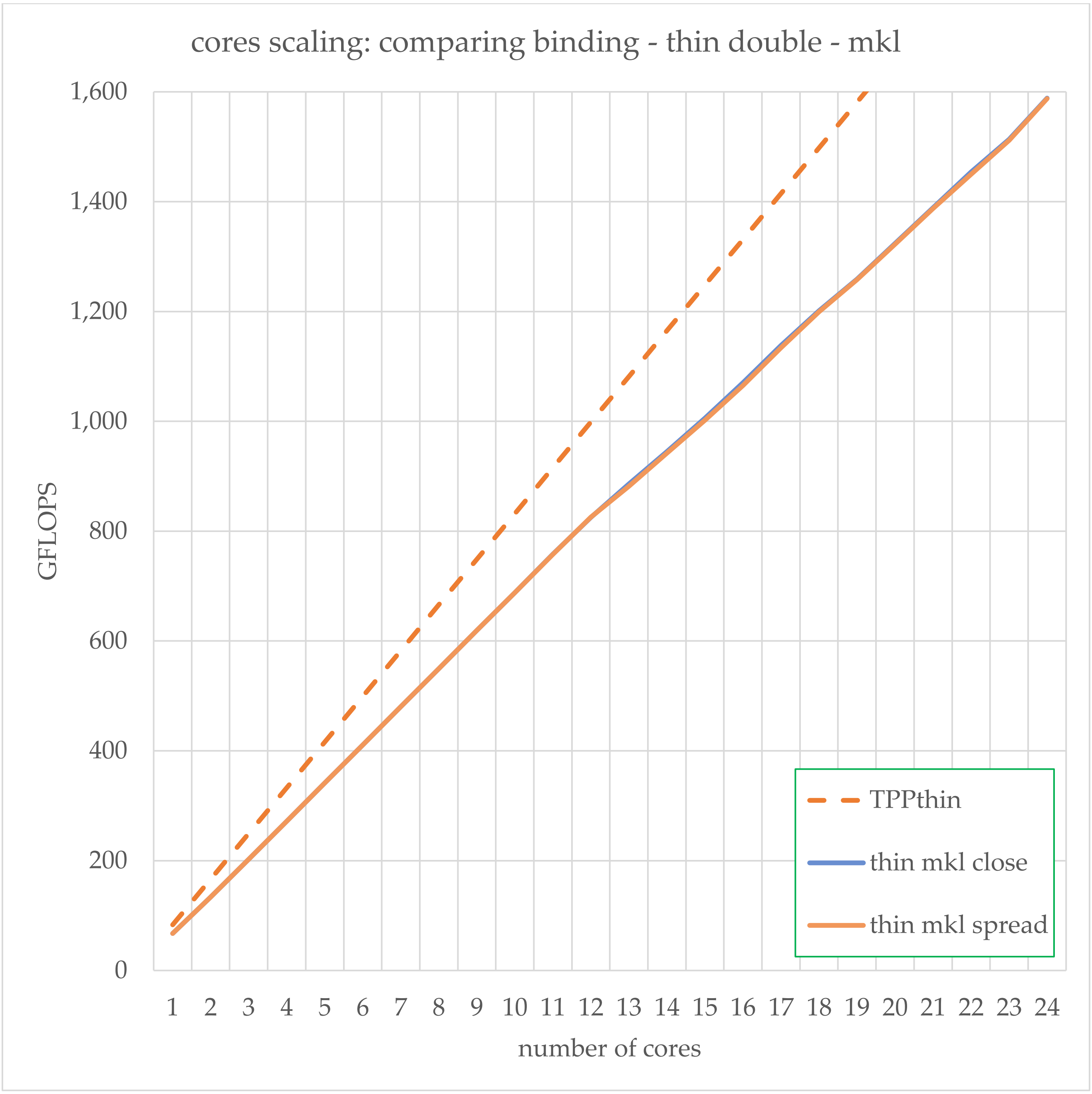


Figure 42: Scaling over number of cores up to 24, thin nodes, double precision (dgemm), comparing binding policy "close" and “spread”, mkl library.

The same considerations hold than what has been written about Figure 37.

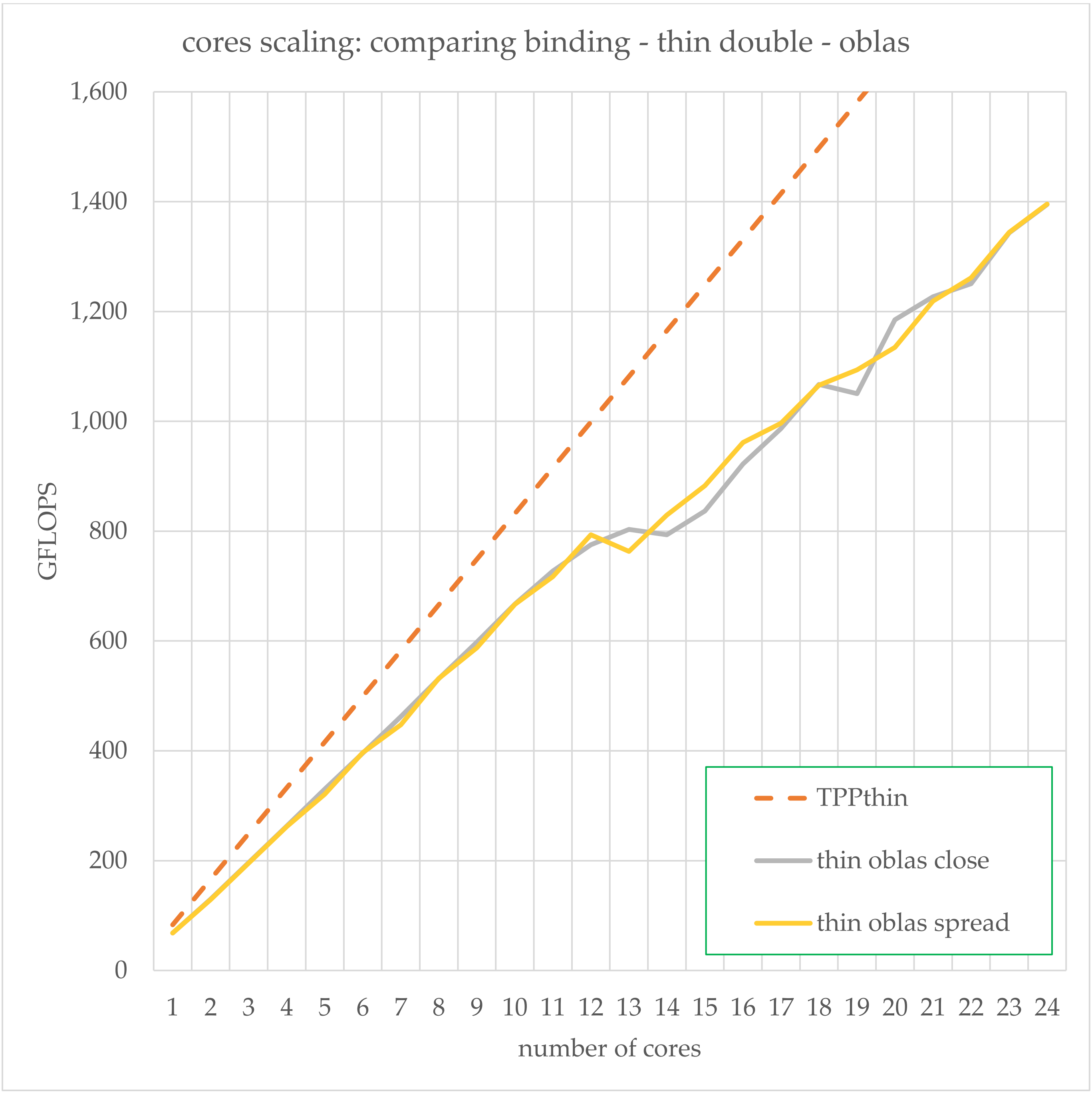


Figure 43: Scaling over number of cores up to 24, thin nodes, double precision (dgemm), comparing binding policy "close" and "spread", oblas library.

The same considerations hold than what has been written about Figure 37.

Conclusions

In the graphic charts, the first comparisons suggested that probably the TPP for epyc nodes has been computed in a way that gives a value that is the double of the real one. It has been considered that this may be due to wrong values of the number of operations per cycles or the number of cores used. When scaling over the number of cores it appeared instead that the TPP may be

correct, but 64 cores is already such a big number for which the performance gap is wide. The gap spreads out when using more then 20 cores. This may also explain the fact that the results on thin nodes do not show the same decrease in performance, as at most 24 cores are used.
On epyc the performance is close to the TPP when the number of cores is increased up to 16, this may be related to the hardware configuration. One epyc socket, called package, contains 64 CPUs assigned to 4 numa regions, each with 16 cores, in groups of 4 within the same ccx, sharing L3 cache. To better inspect the behaviour, some experiment should be caried out pinning the CPUs to specific locations, verifying the thread associations to the CPUs. Having 16 channels to simultaneously access the data memory could explain the situation if the threads are pinned in a way where each of them uses a separate channel. With an higher number of cores the channels and the caches have to be shared more and more by the cores, decreasing the overall performance. It is also known that the performance of the cores is reduced by design by a given percentage proportionally to the total number of active cores, this may also reduce the overall performance increasing the total number.
When scaling is extended to all the 128 cores of an epyc node, it can be observed that using more than 64 cores results in more irregular measured values and the performance is not increasing. This may be explained with the fact that using more then 64 cores implies the use of cores located on two different sockets. Cores on different sockets are more distant, so any data communication and arbitration on cache management takes longer and less predictable time.
The thin node architecture does not show the same behaviour, scaling quite well in all situations, maybe due to a less complex hardware configuration.
Repeating the executions with the binding policy set to "spread" instead of "close" we cannot appreciate much difference. In some case, for the lower range of the data sizes when using the "spread" binding the program is performing better. The difference is quite clear, so a different usage of the available hardware is helping the program perform better, probably associating thread to cores that can have a dedicated access to caches and memory. It should be investigated if the situation can be better controlled specifically pinning threads to cores. Obviously when this is useless when using most of the available cores as they will necessarily be anyway closer and sharing the same resources, increasing the overhead to manage the use of caches.
There are no great differences about the performance between the libraries on the epyc nodes, but still the comparison shows there is a distinct behaviour that can lead to the choice of the best one. On the thin nodes for sure the blis library is performing much worse, showing some incompatibility with the hardware setup, or it may be better to say it may have been optimized for other kind of architectures like epyc where it performs very well.
Scaling over the data size showed that bigger matrices are needed to exploit the processing power. Again, the scaling is quite good in the lower sizes range, probably exploiting caches and dedicated memory channels, but with more cores involved the performance stabilizes at values much lower than the TPP for epyc nodes, and quite better for epic, but with a simpler architecture. It may have sense to try to scale over the data size using a lower number of cores, especially on epyc, to find a configuration that exploits its computational power, maybe to process data in a reasonable time being more energy efficient.
The time recording could be refined subtracting the time overhead spent to manage the recording itself.

Different compilers should be used to compare the compiled program performance. It is expected that dedicated compiler and compiler options could enhance the performance on specific hardware.

Appendix 1: sbatch files

Several batch files have been created for the various executions.

scale_weak.sh
scale_omp.sh
scale_mpi.sh
merge_images.sh
strong_dgemm_thin24.sh
strong_sgemm_thin24.sh
strong_dgemm_epyc128.sh
strong_sgemm_epyc128.sh
strong_sgemm_thin12.sh
strong_dgemm_thin12.sh
strong_dgemm_epyc64.sh
strong_sgemm_epyc64.sh
sgemm_epyc64.sh
dgemm_thin12.sh
sgemm_thin24.sh
dgemm_epyc64.sh
sgemm_thin12.sh
sgemm_epyc128.sh
dgemm_thin24.sh
dgemm_epyc128.sh

To contain the number of batch files, some strategies have been adopted, like passing parameters to reuse the same file for different settings:

```
TYPE="i"
STEPS=100
SNAPAT=0
if [ $# == 1 ]; then
  TYPE="$1"
fi
if [ $# == 2 ]; then
  TYPE="$1"
  SNAPAT="$2"
fi
if [ $# == 3 ]; then
  TYPE="$1"
  SNAPAT="$2"
  SIZE="$3"
fi
echo "Selected type of execution: $TYPE"
...
```

To set the job name in a better way, that helps to keep track of the executions, the batch is called with "sbatch -J thejobname <script>.sh [arguments].

Variables are used to minimize repetitions and keep the commands as clear as possible:

```
#SBATCH --export=ALL,MPI_MODULE=openMPI/4.1.5/gnu/12.2.1
     ,EXECUTABLE=./gameoflife.x
...
module load "${MPI_MODULE}"
...
export MPIRUN_OPTIONS="--bind-to core --map-by socket:PE=$threads
-report-bindings"
export OMP_NUM_THREADS=$threads
...
startexe="mpirun -n ${SLURM_NTASKS} ${MPIRUN_OPTIONS}
${EXECUTABLE} -r -f pattern_random$SIZE.pgm -n $STEPS -e $TYPE -s
$SNAPAT -q"
```

The job output file name is set using the job ID as a suffix:

```
#SBATCH --output=scale_omp_epyc_2_job_%j.out
```

Appendix 2: averaging and outliers

Multiple executions are done to produce results data store together with all the settings that generated them. After a validation process averages have been calculated in the spreadsheets and outliers identified and discarded where possible. To identify the outliers the standard deviation of the measurements has been computed to define a range of values marked as “Pick”. The discarded values are lower than the average less two times the standard deviation or bigger than the sum of the average plus two times the standard deviation. With many measurements available it is advisable to avoid multiplying the standard deviation by two and restrict the range of picked values. On the contrary, having few numbers, it is convenient to widen the range and have some manual selection of values to pick or discard.

Appendix 3: periodicity detection

Some start patterns generate a periodic sequence of identical cell distributions. Tools have been used to identify duplicates. Deleting all dupes and keeping the initial frames of the sequence is convenient when animated videos are created by joining the frames, as described in the next appendix, to minimize the workload.
For example a sequence can be generated using “`-e1`” to use the static evolution, starting from any of the patterns, provided in the proper project directory:

```
mpirun -np 4 gameoflife.x -r -f
patterns/spaceships/pattern_ship32.pgm -n10000 -e1 -s1
```

10,000 files are written, as “`-s1`” is specified to tell the program to write a file at each evolution step, together with “`-n10000`” that asks for 10,000 evolution steps to be performed. Then the “`fdupes`” command can be used to identify duplicates and delete them, keeping the beginning of the sequence. This is done by steps:

```
cd patterns_spaceships_pattern_ship32.pgm_static_10000_004_2023-
06-29_20_40_48
mv final.pgm z_final.pgm
fdupes -d -N --order=name .
```

The "`cd`" command is given to set the path to the directory where all the snapshots have been written. Then the final output file "`final.pgm`" is renamed to "`z_final.pgm`" to be the last of the sequence in alphabetic order. Then "`fdupes`" is called with "`--order=name`" to instruct the program to treat the files sorted by file name. First by hash check, then by binary comparison, the program identifies that most of the files are duplicates, in the given example only the first 65 files are the unique sequence that is then repeated. The "`-d`" option is the command to delete the duplicates. To preserve the first instance of a file in every set of duplicate files, ignoring the prompt that asks interactively for a selection, the "`-N`" switch is included.

Appendix 4: animation videos

When running the static evolutions, the image files output sequence can be joined together, and the result can be played as a video animation. This is useful to check that well known patterns generate the correct sequence of evolution steps, with each step being represented as a video frame. It is advisable to proceed to delete duplicated parts of the sequence, as seen in the previous appendix, to minimize the video file size, and make it reproduceable in a loop to display the repetitive periodic sequence.
Three methods have been used to create the video animations:
GIMP interactive image editor:
From GIMP go to File -> Open as Layers to open all the files on their own layer. From here you can perform edits on the layers and, once done, go to File -> Export As. From the dialog be sure to set the file type to GIF. From there you will go to the GIF export options. Tick the 'As Animation' option and set the parameters as required. To change the delay between frames, modify the name of the layers, and include the delay in milliseconds between parenthesises, like this: (1500ms). To preview the animation before exporting, click "Filters" menu, then "Animation", then "Animation Playback".[15]

Imagemagick command line tool:
The tool can be installed as a package, and then run specifying the delay between frames in milliseconds, instructing to repeat the animation in a loop, taking all the "`pgm`" snapshot files as input, and writing to an output "ship.gif" animated gif:

```
sudo apt install imagemagick-6.q16
convert -delay 20 -loop 0 *.pgm ship.gif
```

ffmpeg:
ffmpeg tool can be installed and run, this time producing an MP4 video file. Using a video file codec, the result is a video with soft borders, that may give a more appreciable display effect, but that can hide the actual information about the alive cells.[16]
A slideshow video with one image per second can be produced by:

```
ffmpeg -framerate 1 -pattern_type glob -i '*.pgm' -c:v libx264 -r
30 -pix_fmt yuv420p out.mp4
```

PGM image sequence format:
Information has been searched about the PGM format, and it has been found that a video PGM sequence has been designed, but probably never implemented in software tools. By specifications, a PGM file consists of a sequence of one or more PGM images. There are no data, delimiters, or padding before, after, or between images. Before July 2000, there could be at most one image in a PGM file. As a result, most tools to process PGM files ignore (and don't read) any data after the first image.[17][18]

Appendix 5: CPU info EPYC

To obtain CPU information about the EPYC nodes the following commands can be used, that report identical output text for each of the 128 cores.

```
salloc -n 128 -N1 -p EPYC --time=2:0:0
srun lscpu>epyc.txt
exit
```

Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Address sizes	43 bits physical, 48 bits virtual
Byte Order	Little Endian
CPU(s)	128
On-line CPU(s) list	0-127
Vendor ID	AuthenticAMD
Model name	AMD EPYC 7H12 64-Core Processor
CPU family	23
Model	49
Thread(s) per core	1
Core(s) per socket	64
Socket(s)	2
Stepping	0
Frequency boost	enabled
CPU(s) scaling MHz	127%
CPU max MHz	2600.0000
CPU min MHz	1500.0000
BogoMIPS	5190.23
Flags	fpu vme de pse tsc msr pae mce cx8 ... **avx avx2** ...
Virtualization	AMD-V
L1d cache	4 MiB (128 instances)
L1i cache	4 MiB (128 instances)
L2 cache	64 MiB (128 instances)
L3 cache	512 MiB (32 instances)

NUMA node(s)	8
NUMA node0 CPU(s)	0-15
NUMA node1 CPU(s)	16-31
NUMA node2 CPU(s)	32-47
NUMA node3 CPU(s)	48-63
NUMA node4 CPU(s)	64-79
NUMA node5 CPU(s)	80-95
NUMA node6 CPU(s)	96-111
NUMA node7 CPU(s)	112-127
Vulnerability Itlb multihit	Not affected
Vulnerability L1tf	Not affected
Vulnerability Mds	Not affected
Vulnerability Meltdown	Not affected
Vulnerability Mmio stale data	Not affected
Vulnerability Retbleed	Mitigation; untrained return thunk; SMT disabled
Vulnerability Spec store bypass	Mitigation; Speculative Store Bypass disabled via prctl
Vulnerability Spectre v1	Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2	Mitigation; Retpolines, IBPB conditional, STIBP disabled, RSB filling, PBRSB-eIBRS Not affected
Vulnerability Srbds	Not affected
Vulnerability Tsx async abort	Not affected

Appendix 6: CPU info THIN

To obtain CPU information about the THIN nodes the following commands can be used, that report identical output text for each of the 24 cores.

```
salloc -n24 -t 1:00:00 -p THIN --time=2:0:0
srun lscpu>thin.txt
exit
```

Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Address sizes	46 bits physical, 48 bits virtual
Byte Order	Little Endian
CPU(s)	24
On-line CPU(s) list	0-23
Vendor ID	GenuineIntel
Model name	Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz
CPU family	6
Model	85
Thread(s) per core	1
Core(s) per socket	12

Socket(s)	2
Stepping	4
CPU(s) scaling MHz	42%
CPU max MHz	3700.0000
CPU min MHz	1000.0000
BogoMIPS	5200.00
Flags	fpu vme de pse tsc msr pae mce cx8 ... **avx avx2 avx512f avx512dq avx512cd avx512bw avx512vl** ... (omitted)
L1d cache	768 KiB (24 instances)
L1i cache	768 KiB (24 instances)
L2 cache	24 MiB (24 instances)
L3 cache	38.5 MiB (2 instances)
NUMA node(s)	2
NUMA node0 CPU(s)	0,2,4,6,8,10,12,14,16,18,20,22
NUMA node1 CPU(s)	1,3,5,7,9,11,13,15,17,19,21,23
Vulnerability Itlb multihit	KVM
Mitigation	VMX unsupported
Vulnerability L1tf	Mitigation; PTE Inversion
Vulnerability Mds	Mitigation; Clear CPU buffers; SMT disabled
Vulnerability Meltdown	Mitigation; PTI
Vulnerability Mmio stale data	Mitigation; Clear CPU buffers; SMT disabled
Vulnerability Retbleed	Mitigation; IBRS
Vulnerability Spec store bypass	Mitigation; Speculative Store Bypass disabled via prctl
Vulnerability Spectre v1	Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2	Mitigation; IBRS, IBPB conditional, RSB filling, PBRSB-eIBRS Not affected
Vulnerability Srbds	Not affected
Vulnerability Tsx async abort	Mitigation; Clear CPU buffers; SMT disabled

Appendix 7: useful commands

```
history|grep sinfo
history|grep -- -k (to display past invocations with "-k" parameter)
!<id> (to recall command line from history)
salloc -n 128 -N1 -p EPYC --time=2:0:0
salloc -n 24 -N1 -p THIN --time=2:0:0
exit
watch '<command>'
sinfo -N --format="%.15N %.6D %.10P %.11T %.4c %.10z %.8m %.10e %.9O %.15C"'
squeue --format="%.6i %.9P %.22j %.10u %.10M %.6D %.5C %.7m %.93R %.8T %l %L"
```

Appendix 8: preprocessor directives

A lot of code has been added to the gameoflife sources to aid complex debugging. This may make the code quite unreadable unless a good code editor is used that can hide or collapsed the added sections. A tool like https://coan2.sourceforge.net/ can be used to strip out the additional section, undefining the debugging macros. When the debugging macros are defined, the debugging level can be set from the command line using the "-D<n>" switch.

gameoflife.h

```
    // activate debug code
    //#define DEBUGOMP
    //#define DEBUG1
    //#define DEBUG2
    //#define DEBUG_ADVANCED_MALLOC_FREE
    //#define DEBUG_ADVANCED_B
    //#define DEBUG_ADVANCED_COORDINATES
```

gameoflife.c

```
    int debug_info = 0;
    ...
        // debug level set on the command line
        case 'D':
            debug_info = 1;
            if (optarg)
                debug_info = atoi(optarg);
```

evolution_ordered.c

```
    void evolution_ordered(..., int debug_info) {
    ...
        #ifdef DEBUG1
            if (debug_info > 0)
                printf("DEBUG1 - evolution_ordered ...
        #endif
```

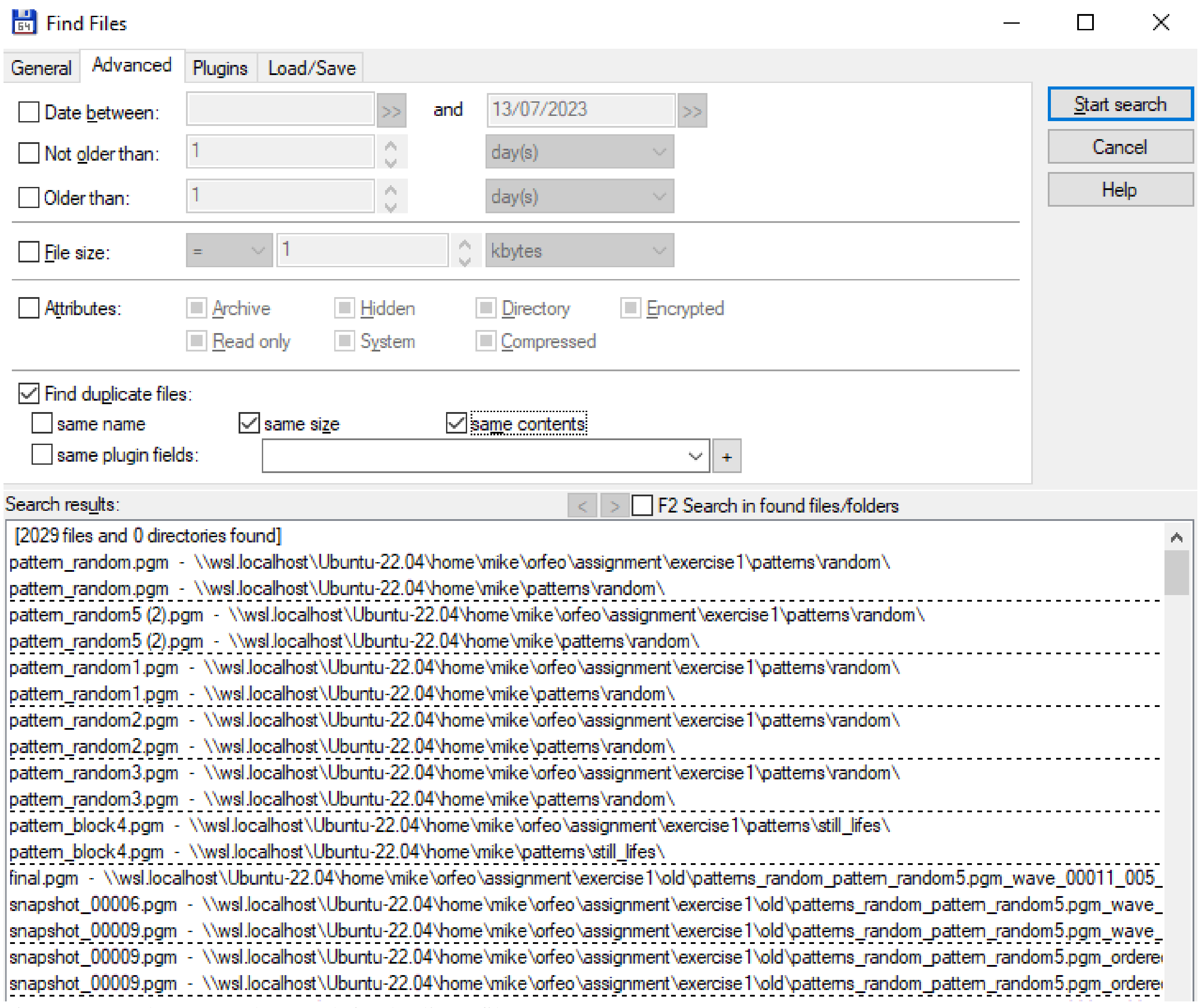
Appendix 9: game of life patterns

Some of the most commonly known patterns have been tested in the ordered evolution of the game of life, and the gif or mp4 animations have been generated for visual inspection as described in appendix 4:

```
patterns/oscillators/pattern_blinker32.pgm
patterns/oscillators/pattern_blinker32.pgm_static_2023-05-
18_19_41_38.mp4
patterns/oscillators/pattern_blinker5.pgm
patterns/oscillators/pattern_blinker5GIMP.pgm
patterns/oscillators/pattern_pulsar32.pgm
patterns/oscillators/pattern_pulsar32.pgm_static_2023-05-
18_19_45_54.mp4
patterns/pattern_blinker.pgm
patterns/pattern_pulsar.pgm
patterns/pattern_ship.pgm
patterns/ship.gif
patterns/spaceships/pattern_ship32.pgm
patterns/spaceships/pattern_ship32.pgm_static_2023-05-
18_19_51_20.mp4
patterns/still_lifes/pattern_block4.pgm
patterns/still_lifes/pattern_block4GIMP.pgm
patterns/still_lifes/pattern_loaf6.pgm
patterns/still_lifes/pattern_loaf6GIMP.pgm
...
```

Appendix 10: binary comparison

Software tools have been used to search for duplicate files and analyse binary differences to evaluate program correctness and behaviour. Common command line tools are grep, diff, cmp, od, awk, but graphical tools like Total Commander[19] with plugins can be of great help to investigate, diagnose problems and detect patterns in data.

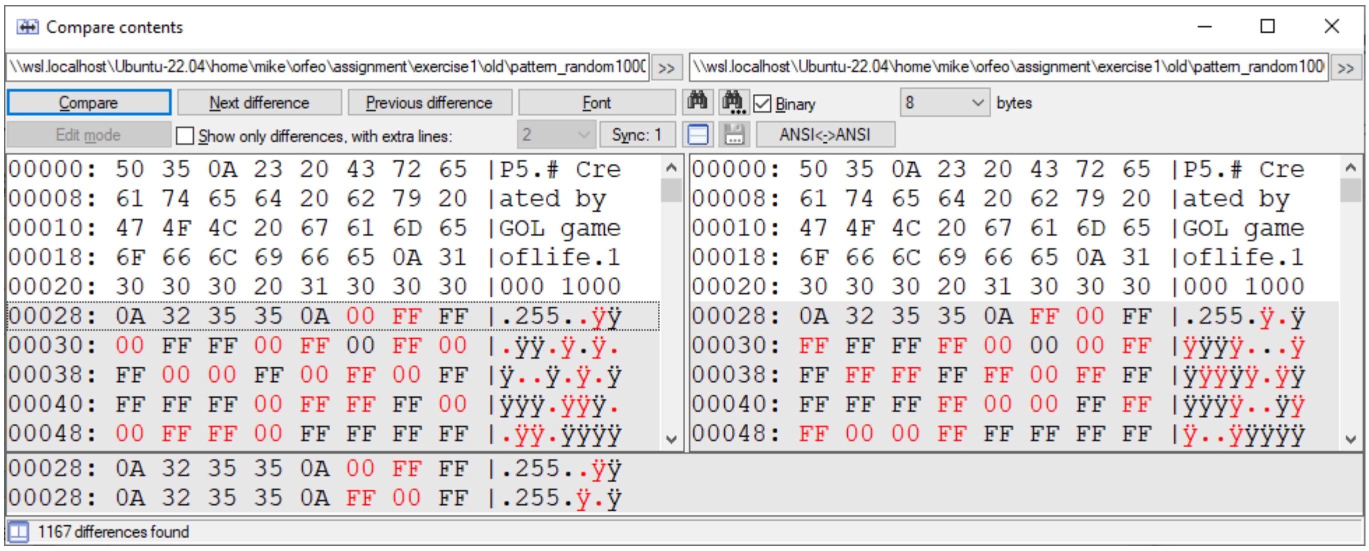


☐ Date between: >> and 13/07/2023 >>
☐ Not older than: 1 day(s)
☐ Older than: 1 day(s)
☐ File size: = 1 kbytes
☐ Attributes: ☐ Archive ☐ Hidden ☐ Directory ☐ Encrypted ☐ Read only ☐ System ☐ Compressed
☑ Find duplicate files:
☐ same name ☑ same size ☑ same contents
☐ same plugin fields:

Search results: ☐ F2 Search in found files/folders

```
[2029 files and 0 directories found]
pattern_random.pgm - \\wsl.localhost\Ubuntu-22.04\home\mike\orfeo\assignment\exercise1\patterns\random\
pattern_random.pgm - \\wsl.localhost\Ubuntu-22.04\home\mike\patterns\random\
-------------------------------------------------------------------------------------------------
pattern_random5 (2).pgm - \\wsl.localhost\Ubuntu-22.04\home\mike\orfeo\assignment\exercise1\patterns\random\
pattern_random5 (2).pgm - \\wsl.localhost\Ubuntu-22.04\home\mike\patterns\random\
-------------------------------------------------------------------------------------------------
pattern_random1.pgm - \\wsl.localhost\Ubuntu-22.04\home\mike\orfeo\assignment\exercise1\patterns\random\
pattern_random1.pgm - \\wsl.localhost\Ubuntu-22.04\home\mike\patterns\random\
-------------------------------------------------------------------------------------------------
pattern_random2.pgm - \\wsl.localhost\Ubuntu-22.04\home\mike\orfeo\assignment\exercise1\patterns\random\
pattern_random2.pgm - \\wsl.localhost\Ubuntu-22.04\home\mike\patterns\random\
-------------------------------------------------------------------------------------------------
pattern_random3.pgm - \\wsl.localhost\Ubuntu-22.04\home\mike\orfeo\assignment\exercise1\patterns\random\
pattern_random3.pgm - \\wsl.localhost\Ubuntu-22.04\home\mike\patterns\random\
-------------------------------------------------------------------------------------------------
pattern_block4.pgm - \\wsl.localhost\Ubuntu-22.04\home\mike\orfeo\assignment\exercise1\patterns\still_lifes\
pattern_block4.pgm - \\wsl.localhost\Ubuntu-22.04\home\mike\patterns\still_lifes\
-------------------------------------------------------------------------------------------------
final.pgm - \\wsl.localhost\Ubuntu-22.04\home\mike\orfeo\assignment\exercise1\old\patterns_random_pattern_random5.pgm_wave_00011_005_
snapshot_00006.pgm - \\wsl.localhost\Ubuntu-22.04\home\mike\orfeo\assignment\exercise1\old\patterns_random_pattern_random5.pgm_wave_
snapshot_00009.pgm - \\wsl.localhost\Ubuntu-22.04\home\mike\orfeo\assignment\exercise1\old\patterns_random_pattern_random5.pgm_wave_
snapshot_00009.pgm - \\wsl.localhost\Ubuntu-22.04\home\mike\orfeo\assignment\exercise1\old\patterns_random_pattern_random5.pgm_ordere
snapshot_00009.pgm - \\wsl.localhost\Ubuntu-22.04\home\mike\orfeo\assignment\exercise1\old\patterns_random_pattern_random5.pgm_ordere
```

Figure 44: Total Commander search for duplicate files



\\wsl.localhost\Ubuntu-22.04\home\mike\orfeo\assignment\exercise1\old\pattern_random100(>>
\\wsl.localhost\Ubuntu-22.04\home\mike\orfeo\assignment\exercise1\old\pattern_random100 >>

☑ Binary 8 bytes
☐ Show only differences, with extra lines: 2 Sync: 1

```
00000: 50 35 0A 23 20 43 72 65 |P5.# Cre     00000: 50 35 0A 23 20 43 72 65 |P5.# Cre
00008: 61 74 65 64 20 62 79 20 |ated by      00008: 61 74 65 64 20 62 79 20 |ated by
00010: 47 4F 4C 20 67 61 6D 65 |GOL game     00010: 47 4F 4C 20 67 61 6D 65 |GOL game
00018: 6F 66 6C 69 66 65 0A 31 |oflife.1     00018: 6F 66 6C 69 66 65 0A 31 |oflife.1
00020: 30 30 30 20 31 30 30 30 |000 1000     00020: 30 30 30 20 31 30 30 30 |000 1000
00028: 0A 32 35 35 0A 00 FF FF |.255..ÿÿ     00028: 0A 32 35 35 0A FF 00 FF |.255.ÿ.ÿ
00030: 00 FF FF 00 FF 00 FF 00 |.ÿÿ.ÿ.ÿ.     00030: FF FF FF FF 00 00 00 FF |ÿÿÿÿ...ÿ
00038: FF 00 00 FF 00 FF 00 FF |ÿ..ÿ.ÿ.ÿ     00038: FF FF FF FF FF 00 FF FF |ÿÿÿÿÿ.ÿÿ
00040: FF FF FF 00 FF FF FF 00 |ÿÿÿ.ÿÿÿ.     00040: FF FF FF FF 00 00 FF FF |ÿÿÿÿ..ÿÿ
00048: 00 FF FF 00 FF FF FF FF |.ÿÿ.ÿÿÿÿ     00048: FF 00 00 FF FF FF FF FF |ÿ..ÿÿÿÿÿ

00028: 0A 32 35 35 0A 00 FF FF |.255..ÿÿ
00028: 0A 32 35 35 0A FF 00 FF |.255.ÿ.ÿ
```

1167 differences found

Figure 45: Total Commander binary file comparison of pgm image files

References

[1] Cozzini Stefano and Tornatore Luca. 2022. Final assignments FHPC course 2022/2023 https://github.com/Foundations-of-HPC/Foundations_of_HPC_2022/tree/main/Assignment

[2] Johnston Nathaniel. 2023. A community for Conway's Game of Life and related cellular automata. https://conwaylife.com/

[3] Wikipedia. 2023. Conway's Game of Life. https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

[4] Tornatore Luca. 2022. exercise 1: parallel programming. https://github.com/Foundations-of-HPC/Foundations_of_HPC_2022/tree/main/Assignment#exercise-1--parallel--programming

[5] Cozzini Stefano. 2022. exercise 2 : Comparing MKL, OpenBLAS and BLIS on matrix-matrix multiplication. https://github.com/Foundations-of-HPC/Foundations_of_HPC_2022/tree/main/Assignment#exercise-2--comparing-mkl-openblas-and-blis-on-matrix-matrix-multiplication

[6] Wikipedia. 2023. FLOPS. https://en.wikipedia.org/wiki/FLOPS

[7] Cozzini Stefano. 2021. ORFEO-DOC. https://orfeo-documentation.readthedocs.io/en/latest/

[8] Advanced Micro Devices, Inc.. 2023. AMD EPYC™ 7H12 Processors | AMD https://www.amd.com/en/products/cpu/amd-epyc-7h12

[9] Intel Corporation. 2023. Intel® Xeon® Gold 6126 Processor. https://www.intel.com/content/www/us/en/products/sku/120483/intel-xeon-gold-6126-processor-19-25m-cache-2-60-ghz/specifications.html

[10] GadgetVersus.com. 2023. Dual AMD EPYC 7H12 Processor Benchmarks and Specs. https://gadgetversus.com/processor/dual-amd-epyc-7h12-specs/

[11] GadgetVersus.com. 2023. Dual AMD EPYC 7H12 GFLOPS performance. https://gadgetversus.com/processor/dual-amd-epyc-7h12-gflops-performance/

[12] Intel Corporation. 2021. Article ID: 000057415 Content Type: Product Information & Documentation Last Reviewed: 10/18/2021 Where Can I Find Information about FLOPS Per Cycle for Intel® Processors? https://www.intel.com/content/www/us/en/support/articles/000057415/processors.html

[13] Hermes Senger, Jaime F. de Souza, Edson S. Gomi, Fabio Luporini, Gerard J. Gorman. 2019. Submitted on 9 Aug 2019 (v1), last revised 19 Aug 2019 (this version, v2). Performance of Devito on HPC-Optimised ARM Processors. https://doi.org/10.48550/arXiv.1908.03653

[14] De Petris Mike. 2022. Foundations of HPC 2022 final project. https://github.com/mikedepetris/Foundations_of_HPC_2022_final_project

gameoflife. https://github.com/mikedepetris/gameoflife
exercise2. https://github.com/mikedepetris/exercise2

[15] Ask Ubuntu. 2023. How do I create an animated gif from still images (preferably with the command line)? https://askubuntu.com/questions/648244/how-do-i-create-an-animated-gif-from-still-images-preferably-with-the-command-l

[16] Stack Overflow. 2014. How to create a video from images with FFmpeg? https://stackoverflow.com/questions/24961127/how-to-create-a-video-from-images-with-ffmpeg

[17] Poskanzer. Jef. 2016. pgm - Netpbm grayscale image format. https://netpbm.sourceforge.net/doc/pgm.html

[18] Poskanzer. Jef. 1991. Man Pages Copyright Respective Owners. Site Copyright (C) 1994 - 2023 Hurricane Electric. pgm(5) File Formats Manual pgm - portable graymap file format. http://man.he.net/man5/pgm

[19] Ghisler. 2023. Total Commander. https://www.ghisler.com/