Heat Parallelization Report

Efficient Parallel Programming of Multicore Systems

Group 2: Rok Grgič Meško, Beatrice Picco, Tiia Tikkala

Technical University of Munich

1 Compiler Flags

In this assignment, different compiler optimization flags for Intel ICC compiler were compared based on how they affect the MFLOPS performance of the Jacobi relaxation method used to solve the stationary 2D heat equation.

Different levels of optimization were examined using the flags `-O1`, `-O2`, `-O3` and `-Ofast`[1]. Additionally, specific flags that enable the compiler to perform more aggressive optimizations were considered:

- `-ipo`: Enables interprocedural optimization between files.
- `-fno-alias`: Assert there is no aliasing of memory references, that is, the same memory location is not accessed via different arrays or pointers.
- `ivdep`: Instructs the compiler to ignore assumed vector dependencies.

as well as platform specific optimization flags:

- `-xhost`: Tells the compiler to generate instructions for the highest instruction set available on the compilation host processor.
- `-xCORE-AVX512`: Enables the generation of AVX-512 instructions. Leverages wide vector operations.
- `-qopt-zmm-usage mean`: Defines a level of ZMM registers usage.

The Intel compiler also provides detailed optimization reports when using the option `-opt-report`, specifically with `-qopt-report-annotate` and `-qopt-report-phase=vec,loop`. These options help in understanding how well the code is optimized; particularly regarding vectorization and loop transformations.

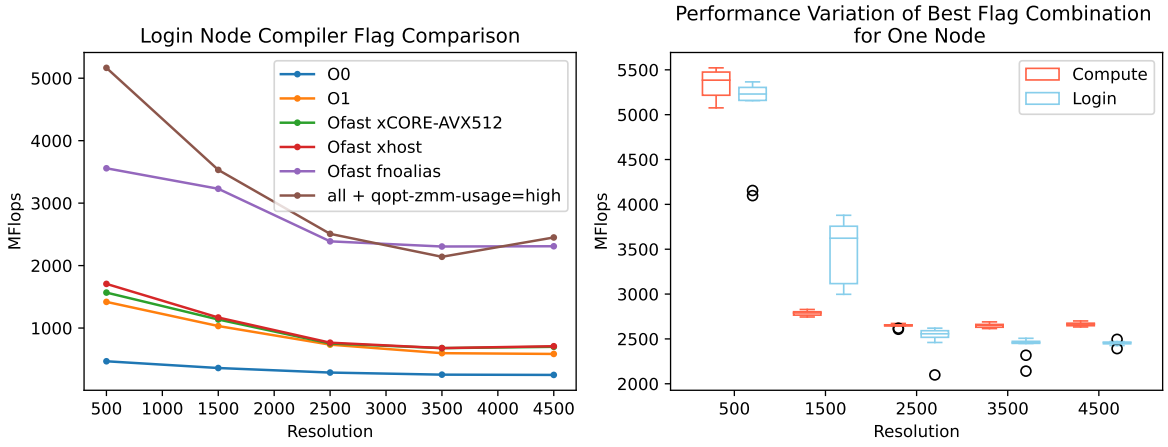

Figure 1.1: Comparison of MFLOPS performance for different compiler flag combinations that had the biggest impact on performance and variation in performance between different runs on login and compute nodes of SUPERMUC-NG (including outliers). Performance was measured for different resolutions (matrix sizes).

[1] `https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-10/overview.html`

As can be seen in Figure 1.1, the flag that had the biggest impact on performance was `-fno-alias`. This flag enabled the compiler to perform a loop interchange in the iteration over the matrix in the Jacobi relaxation method. This improved cache locality by making the fast index row-wise (C language is row-major).

In addition, the general optimization flag `-O1` had a significant impact compared to the baseline `-O0` flag: while the first does not, the second includes some speed optimizations.

Lastly, the variation in performance on the login node is much higher than on the compute node, as shown in Figure 1.1. This is expected, as the login node handles more tasks, leading to increased performance variability.

2 Profiling

In this assignment, the performance of the heat application with different compiler flags was profiled using hardware counters, and based on the results of the profiling, the performance of the sequential program was optimized.

Two methods of data collection were used: LIKWID and VTune. The profilers were used to compare the performance of two sets of compiler flags, `-O2` and `-O3 -fno-alias -xhost`.

Three facets of performance were compared: the MFLOPS of the application, as well as the L2 and L3 cache miss rates. The results of the LIKWID measurements are shown in figure 2.1.

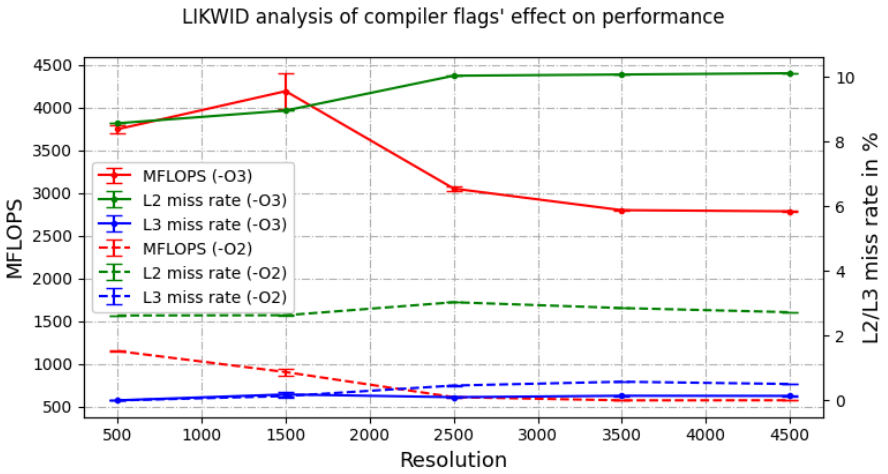

Figure 2.1: MFLOPS, L2 and L3 miss rates of the compiler flag combinations -O2 and -O3 -fno-alias -xhost.

The measurements show that while the higher compiler optimization flags achieve much better MFLOPS performance, they come with higher cache miss rates for the L2-level cache, and similar miss rates for the L3 cache. However, tables 2.2 and 2.1 show that the absolute numbers of cache misses are much lower for the higher compiler flags. This difference becomes especially dramatic when looking at the higher resolutions.

Table 2.1: L2 and L3 misses for -O2.

Resolution	L2 misses	L3 misses
500	6 623 795	1244
1500	59 040 960	4 746 497
2500	188 248 100	33 766 060
3500	347 108 400	83 620 390
4500	548 986 500	120 118 400

Table 2.2: L2 and L3 misses for -O3 -fno-alias -xhost.

Resolution	L2 misses	L3 misses
500	6 282 041	1718
1500	56 318 080	1 764 884
2500	156 413 500	1 933 281
3500	306 491 800	6 492 846
4500	506 600 600	9 072 937

Further investigating the performance with VTune gave more insight into how to improve the code's performance. The application was found to be back-end bound, with 67% of of pipeline slots spent

on memory operations. Moreover, hotspot analysis showed that 42% of CPU time was spent on a copy operation. This is the operation at the end of the Jacobi relaxation where values from the newest iteration are copied into the old array.

Based on these findings, the sequential program was optimized. The main speedup came from changing the memcpy operation to a pointer swap. In addition, the matrix access pattern was changed to align with C-style row-wise access, and the calculation of the residual was improved by combining its calculation loop with the loop of the relaxation step. This way, the stencil was only loaded to cache once instead of twice. The results of these changes are shown in figures 2.2 and 2.3.

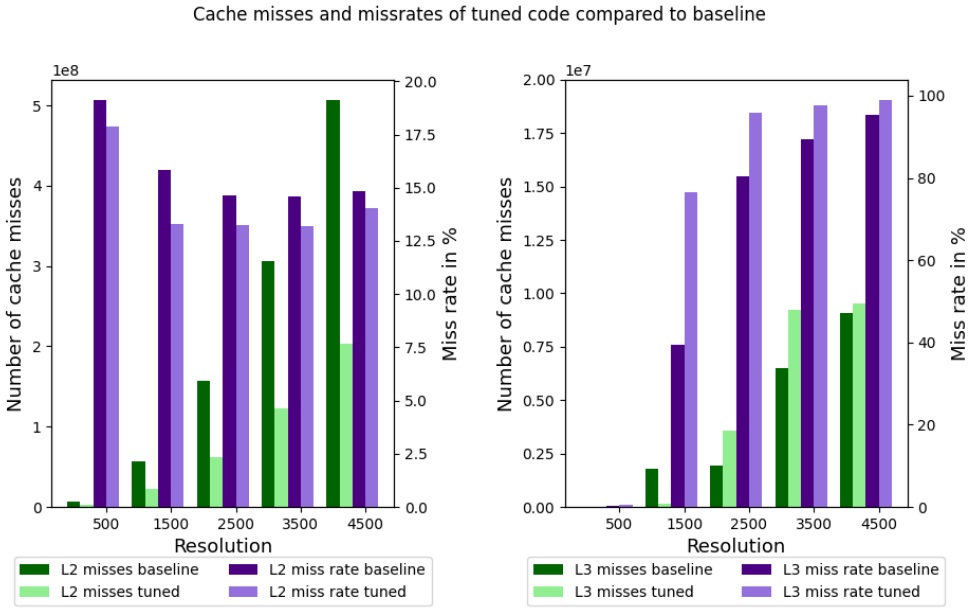

Figure 2.2: L2 and L3 cache misses and miss rates of the optimized code compared to the baseline.

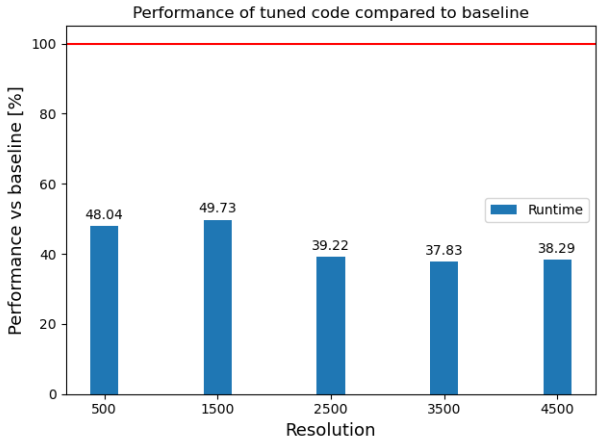

Figure 2.3: Comparison of the runtime of the optimized code versus the baseline.

From these results, it can be seen that the runtime of the code is significantly lower than before, down to only 40% of the baseline. The L2 miss rates have also gone down, as well as the absolute numbers of L2 misses. However, it can be seen that the number of L3 misses has stayed at a similar order of magnitude, with worse performance for some resolutions. The L3 miss rates are between 95-98% for the highest resolutions measured, implying that the changes made to the code still don't use the L3 cache effectively.

3 OpenMP

In this assignment the heat code was parallelized using OpenMP pragmas. Then, process distribution in the architecture was optimized using the environment variable `KMP_AFFINITY`.

Parallelization of the code

The parallelization was done on the loops accessing the main array, specifically the for loops inside the `relax_jacobi` function and the `initialize` function. The directives used for the loop in the `initialize` function were `omp for`, while for the loop inside `relax_jacobi` also `reduction(+:sum)` was added. The reduction allows to have a private variable `sum` for each process and aggregate each one at the end.

The parallelization of the initialisation loop is particularly important because it takes into account NUMA's first touch policy: the data page is allocated in the memory closest to the thread accessing this page for the first time. This allows a better memory distribution and therefore speedup compared to a sequential code, as can be seen in Figure 3.1. The miss rate is the ratio between cache misses and the sum of cache hits and misses, both obtained by LIKWID measurements.

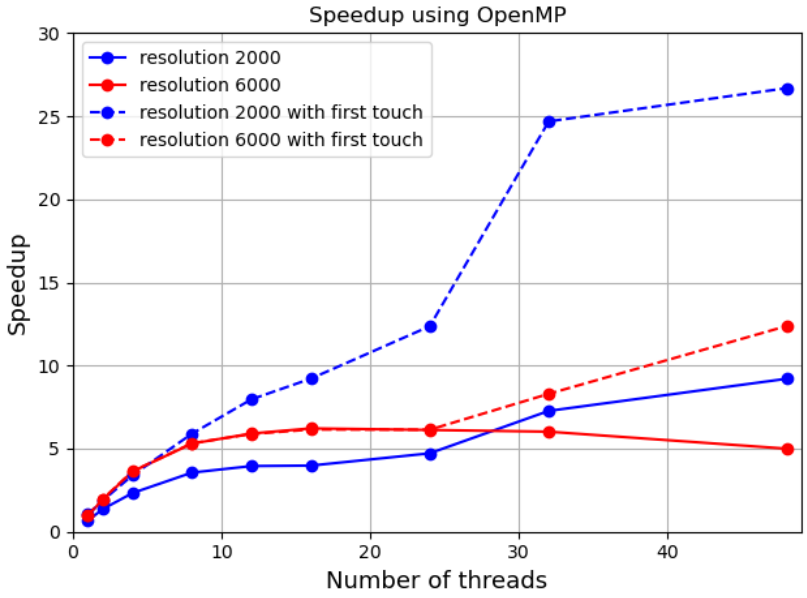

Figure 3.1: Comparison of the runtime speed up (with respect to the sequential code) for heat code with and without parallelization of the initialization.

Figure 3.1 also displays a drop in the speedup trend at 24 threads. This suggests there is a shift between the use of only one socket at 24 threads and both sockets after 24 threads. The full use of the socket can cause higher competition for the communication bandwidth within the socket.

Thread Affinity Interface

The Intel runtime library can bind OpenMP threads to physical processing units and specifically restrict the execution of the threads to a certain set of physical processing units, which can lower the runtime.

This can be done through the the environment variable `KMP_AFFINITY`, and its attributes. With the attribute `type` it is possible to set the affinity type: `scatter` distributes the threads as evenly as possible across the entire system, `compact` is its opposite and it places the current threads next to the previous thread. Other options in between are also available.

The two affinities `compact` and `scatter` were compared in Figure 3.2, which shows a clear advantage of the scatter affinity especially for the lower resolution. This could be due to the fact that there is less bus contention between the threads, and would also explain the reduction in speedup when the thread number increases.

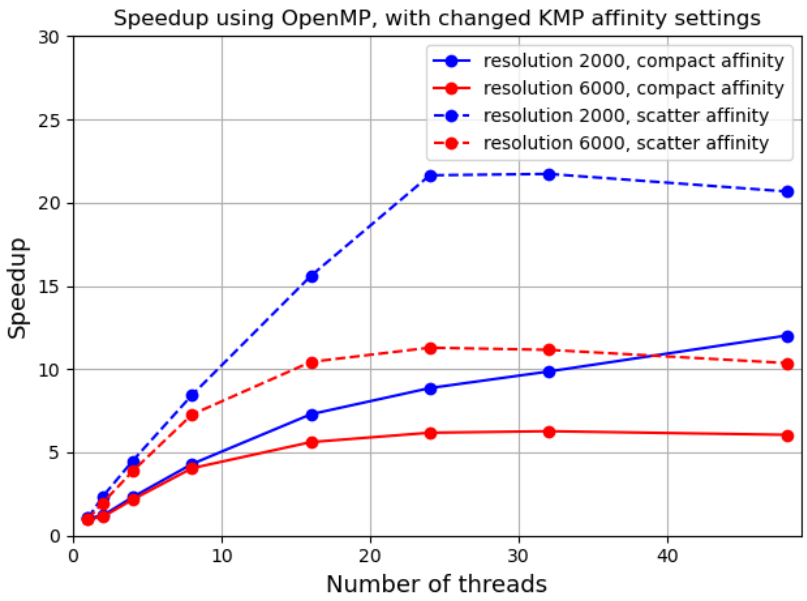

Figure 3.2: Comparison of the runtime speed up (with respect to the sequential code) for heat code with compact and scatter affinity type.

Through the attribute `modifier` it is possible to set the granularity, which describes the lowest level that a thread can float in. This is useful for runtime, because binding threads to a specific thread context is not necessarily beneficial. In this case the best performing granularity was `fine`, which binds each thread to a single thread context.

4

Heat MPI parallelization

In this assignment, the heat application was parallelized using MPI. Only the data used by each process was initialized locally. This way, the Jacobi relaxation and coarsening for the output were parallelized. To split the domain, MPI Cartesian topology was used.

Both blocking and non-blocking communication were trialed. Different ways of splitting up the domain were tried and compared, with divisions done in the x- and y-dimensions, and as a combination.

Figure 4.1 shows the speedups for different processor configurations compared to the sequential version.

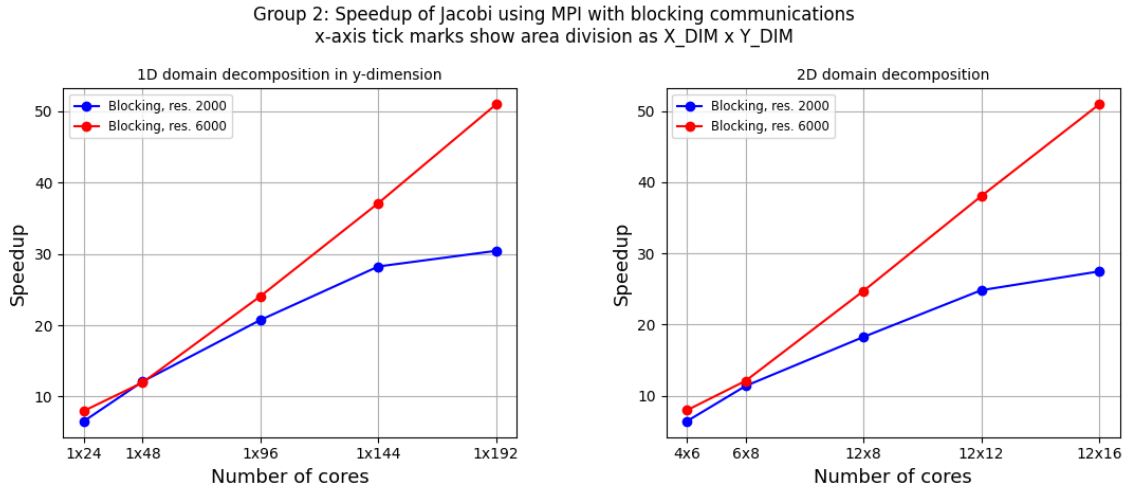

Figure 4.1: Comparison of speedups for domain splitting in only the y-dimension, and in both dimensions.

As can be seen from the results, for the maximum number of cores available, 192 cores spread over 4 nodes, the speedups achieved by the program are just above 50 for the higher resolution, and around 30 for the lower resolution. These results are worse than for the OpenMP parallelization, where 48 cores achieved a speedup of 27, compared to 13 for the MPI implementation. This is likely due to the increased communication overhead of MPI processes, and the fact that OpenMP threads could more effectively utilize the shared memory within one node.

It can also be seen that the there is a performance difference between the two resolutions, with the smaller resolution fairing worse. For the smaller resolution, communication between processes needs to happen more frequently as the relaxation computation is faster, which adds overhead. Additionally, the time spent for code startup is a larger percentage of the total execution time for the smaller resolution. These factors may explain the lower speedups of the 2000 resolution.

The figure also shows that splitting the domain in only one dimension made no significant difference to splitting it in two dimensions. This seems incorrect, as splitting the domain in both dimensions should provide a performance boost. This is because there are fewer border cells to be communicated between processes when the domain is split in two dimensions, requiring less communication overhead. The

reason for these counter-intuitive results was not found, as the domain is split correctly by the program. One area that could be the cause is the communication of non-contiguous columns, as splitting the domain only in the x-dimension, causing only columns to be communicated, has a worse speedup when compared to splitting in only the y-dimension, where only contiguous rows are communicated. However, for now this is still unconfirmed as we could not find the fault in the code.

Blocking and non-blocking communications were compared, as implementing non-blocking communication can sometimes result in a performance boost due to communication latency being hidden by computationally heavy parts of the program. The inner part of the local Jacobi relaxation was computed while the communications were started. Then, MPI_Waitall() was called to wait for the communications to finish. Figure 4.2 shows the results of the comparison.

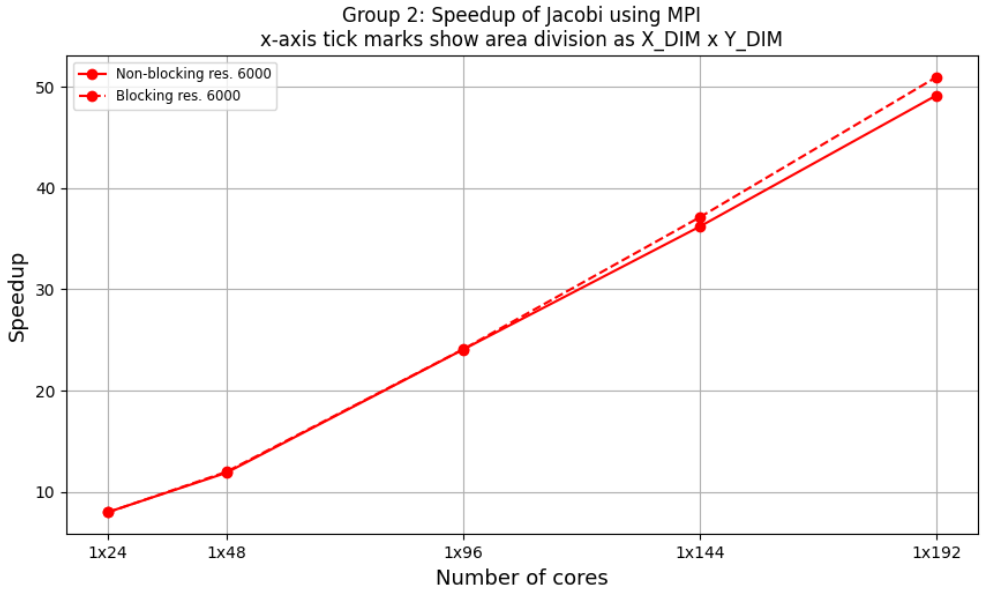

Figure 4.2: Comparison of non-blocking and blocking communication speedups for resolution 6000.

As can be seen from the figure, non-blocking communication did not yield a boost in performance, but slightly decreased it. As it is also more cumbersome for the programmer to implement, it is not worth the effort for this use case. The decrease in performance is likely due to the computation of the Jacobi relaxation in the inner domain not being heavy enough to hide the communication latency. Non-blocking communication also introduces additional overhead due to the use of MPI_Requests. One way to improve the performance of non-blocking communication could be to only exchange data every N iterations, but this would make the parallel code incorrect when compared to the sequential code.

Overall, this assignment showed that implementing MPI with blocking communication yields scalable results for parallelizing the Jacobi method.

5 Hybrid parallelization

In this assignment, MPI and OpenMP were combined to investigate the advantages of hybrid parallelization of the heat application. In the previous two chapters, it was found that OpenMP yields higher speedups than MPI when parallelizing within one node, through the efficient use of shared memory, and MPI could be used for communication between nodes. Combining these two programming models could thus yield more scalable results than pure MPI, as then there would be no need for message passing within shared memory regions.

In the implementation, MPI was used for the communication between nodes, and OpenMP for the calculations inside each node. There is one MPI process per node, and all 48 cores of each node are used to create 48 OpenMP threads per node. With a maximum of 4 available nodes, the maximum number of OpenMP threads used was 192, with 4 MPI processes, and the domain could be split into a maximum of 4 areas. At the start of each iteration, the main OpenMP thread waits for the boundaries to be exchanged between MPI processes, and then 48 threads calculate the parallel loop Jacobi relaxation, with a sum reduction for the residual.

Figure 5.1 shows the speedups obtained through the hybrid approach. Shown are the results for 2 nodes (96 cores), 3 nodes (144 cores) and 4 nodes (192 cores). The domain was divided in only one dimension at a time as well as for both dimensions. For example, the domain was divided in x in the splits 2x1, 3x1 and 4x1. In the 2D case, only one configuration was possible, which was the 2x2 split.

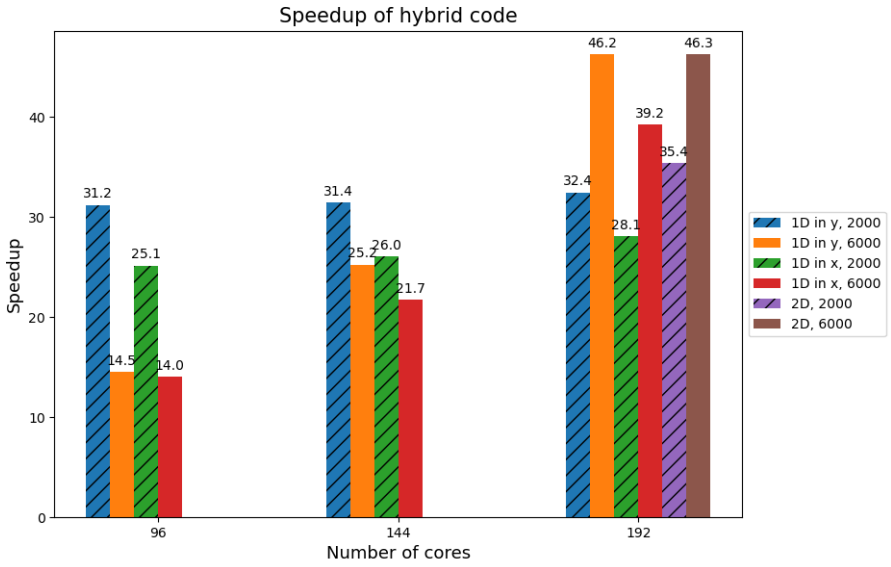

Figure 5.1: Speedups of hybrid implementation.

As can be seen from the figure, the highest speedups obtained from the hybrid implementation are around 46, which is slightly below the maximum speedups obtained by the pure MPI implementa-

tion. This might be due to the fact that the communication between nodes was not threaded, as the main OpenMP thread waits while the MPI processes exchange boundary data. Even though MPI non-blocking communication was used, most threads are thus sleeping while the communication is started.

As with the pure MPI version, there is no clear difference between splitting the domain in only one dimension compared to splitting it in both dimensions. This is likely caused by the same culprit as in the pure MPI version, which, as previously stated, remains unknown. In addition, the higher resolution results in higher speedups, which is consistent with the pure MPI version.

The results also show that the speedups for resolution 2000 do not scale with increasing number of processors. This could be because the parallelizable code for the smaller resolution is a smaller percentage of the total execution time, and per Amdahl's law is limited to some upper limit. Similar behaviour could be seen in the pure MPI version in figure 4.1, where the lower resolution's speedup graph starts to taper off reaching its limit at around a speedup of 30.

In conclusion, this assignment showed that this model of hybrid parallelization, where for each node, one MPI process handles communication and OpenMP threads handle the computation, does not present a performance boost when compared to a pure MPI implementation. This can be due to our specific implementation, which already showed some difficulty in the pure MPI version.