1. Lab Report for Heat Parallelization

Group 7: Jinwen Pan, Mingshuai Li, Jingtian Zhao

1.1 Compiler Flags

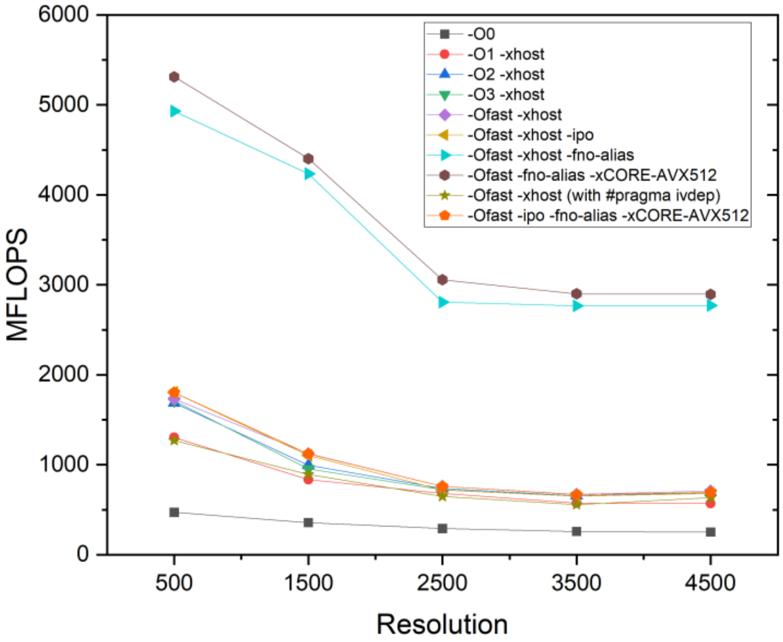


Figure 1.1: Performance With Different Compiler Options

As shown in Figure 1.1, for each combination, as the resolution increases, the FLOPS gradually decreases and eventually stabilizes. This may be because the two almost duplicate functions `relax_jacobi` and `residual_jacobi` executed in each iteration have strong data dependencies. That is, the execution of the `residual_jacobi` function must wait for the completion of the `relax_jacobi` function before it can start. This dependency introduces additional time overhead which increases with the resolution and cannot be efficiently optimized by the compiler. Additionally, the `relax_jacobi` function involves an expensive memory copy operation, and the increasing resolution amplifies the associated time overhead. Therefore, as the resolution increases, the program's runtime increases due to not only the naturally increased computational workload but also the aforementioned factors, leading to a decrease in FLOPS.

The best performance is achieved by specifying `-Ofast -fno-alias -xCORE-AVX512`. Among these three, the vast majority of the performance improvement is contributed by `-fno-alias`. By specifying `-fno-alias`, aliasing is not assumed in the program, which enables the compiler to optimize the code more progressively. Specifically, it swaps the order of the two loops in the `relax_jacobi` and `residual_jacobi` functions such that memory is accessed continuously and the cache is utilized more efficiently resulting in a better overall performance. Additionally, `-Ofast`, which includes all optimizations of `-O3`, will enable more aggressive optimizations, including loop unrolling, function inlining, and floating-point operation optimizations, while the `-xCORE-AVX512` flag enables the utilization of Intel AVX-512 vectorization instructions. These instructions allow for parallel execution of operations on multiple data elements, significantly increasing the computing speed.

With the `-O1` option, the compiler aims to cut down both code size and execution time but does not do any optimization that significantly increases compilation time. The `-O2` optimization level builds upon `-O1` by introducing further techniques that enhance execution speed, possibly at the expense of increased code size and compilation time. These additional optimizations include better variable life-cycle management and preferential storage of variables in registers over memory. At the `-O3` level, all optimizations from `-O2` are included along with other techniques that further boost execution speed. This level enables more automatic inlining, vectorization, loop optimizations, etc. These may increase the code size and potentially impact compilation time.

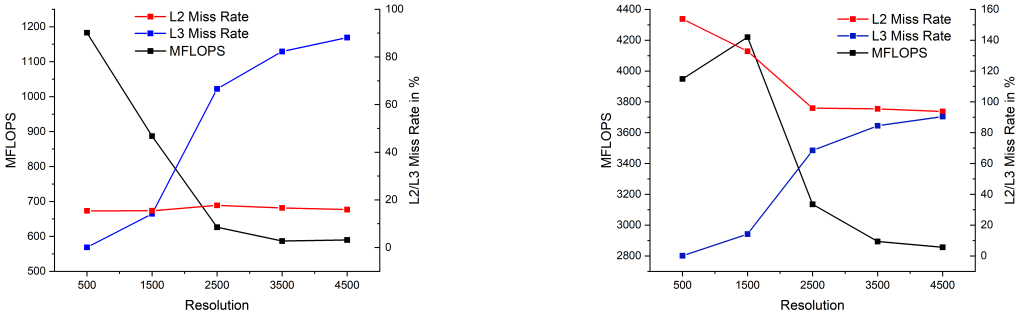
For programs involving parallel computation and matrix operations, `-xhost` can lead to a significant performance improvement. This option optimizes the code specifically for the host running it. The compiler queries the host CPU and automatically sets the target instruction set options to exploit the host hardware's capabilities to the fullest. It is because of this that when both `-xCORE-AVX512` and `-xhost` options are used simultaneously, the option that appears later in the compilation command will be overridden by the former.

Last but not least, `-ipo` and `#pragma ivdep` are also investigated. The former can leverage global information and optimize across source file boundaries to reduce function call overhead, optimize memory access patterns, and improve program performance. However, when using both `-ipo` and `-fno-alias` together, it will eliminate the significant performance gains brought by `-fno-alias`. On the other hand, `#pragma ivdep` is a compiler directive used to instruct the compiler to ignore inter-loop dependencies during vectorization optimization, resulting in the generation of more efficient vectorized code.

Notably, utilizing a higher optimization level or combining more compiler options does not always result in significant performance improvements. This also depends on factors such as code characteristics, hardware architecture, workload of the application, and more. For certain code patterns and specific applications, more aggressive optimizations may lead to more instruction cache misses, register pressure, or other unfavorable effects, thereby resulting in performance degradation.

1.2 Profiling

When using the `-O2` flag, the L2 miss rate remains stable between 15% and 17%, indicating a moderate level of cache efficiency. The L3 miss rate starts at a low value of 0.89% for a resolution of 500 but gradually increases to a significant 88% as the resolution increases. This



(a) Performance Metrics Comparison (-O2)

(b) Performance Metrics Comparison (best)

Figure 1.2: Performance Metrics Comparisons by PAPI

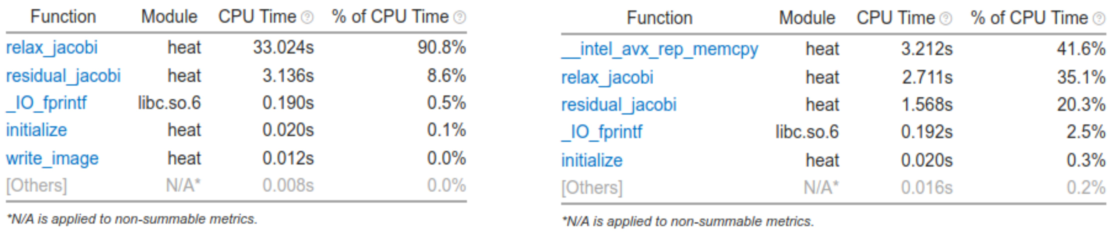
suggests that the L3 cache is being overwhelmed by the larger amount of data, resulting in a higher rate of cache miss. The MFLOPS shows a decrease from 1183 at a resolution of 500 to 589 at a resolution of 4500, which is similar to the experimental results from the previous section, but we are able to understand the reasons from a closer-to-hardware perspective, specifically by considering the cache.

When using the best flag combination `-O3 -fno-alias -xhost`, the L2 miss rate is significantly different from before, as it exceeds 90% for different resolutions, which suggests that the cache utilization is not optimized. And the L3 miss rate remains almost consistent with the behavior observed in the previous version. While the MFLOPS initially experiences a slight increase, it subsequently maintains a similar decreasing trend as the previous version. However, there is a significant overall improvement in MFLOPS compared to the `-O2` version, which can be attributed to the `-fno-alias` option discussed in the previous section.

We have observed an unusual phenomenon in the version of the best compiler optimization, where the L2 miss rate exceeded 100% for resolutions 500 and 1500, which contradicts the logical expectation that the L2 miss rate should always be less than or equal to 100%. According to the documentation we have consulted, one possible reason is that when the CPU requests data from the L2 cache and experiences a cache miss, it may repeat the request to the L2 cache multiple times before directly sending the request to the L3 cache. This increases the overall L2 cache miss count, but it is not recorded correctly by hardware counters associated with L2 total access.

What is more, we must emphasize the importance of minimizing L3 cache misses, as they result in the need to retrieve data from the main memory, significantly increasing the time required for data access and thereby reducing the overall program execution efficiency. This assertion is further supported by the relationship between L3 miss rate and MFLOPS depicted in the two accompanying graphs.

Additionally, we also utilize VTune to analyze the other aspects of performance for both versions with 200 iterations and a resolution of 3200. According to the performance snapshots, for both versions, almost all double-precision floating-point operations have been vectorized. However, compared to the `-O2` version, the optimized version has achieved a higher level of vectorization, from 128bit to 256bit. This allows four doubles to be operated on simultaneously. Although the slightly improved IPC still does not exceed 1, the program's runtime dramatically drops from the original 36.7 seconds to 7.8 seconds. Furthermore, based on the hotspots analysis, for both versions, we discover that the majority of the runtime comes from the two Jacobi functions. Even though a significant reduction in runtime is attributed to the optimized memory copy functions provided by the Intel compiler, unnecessary memory copy still accounts for nearly half of the runtime, which indicates the direction for sequential optimization in the next section.



Function	Module	CPU Time	% of CPU Time
relax_jacobi	heat	33.024s	90.8%
residual_jacobi	heat	3.136s	8.6%
_IO_fprintf	libc.so.6	0.190s	0.5%
initialize	heat	0.020s	0.1%
write_image	heat	0.012s	0.0%
[Others]	N/A*	0.008s	0.0%

*N/A is applied to non-summable metrics.

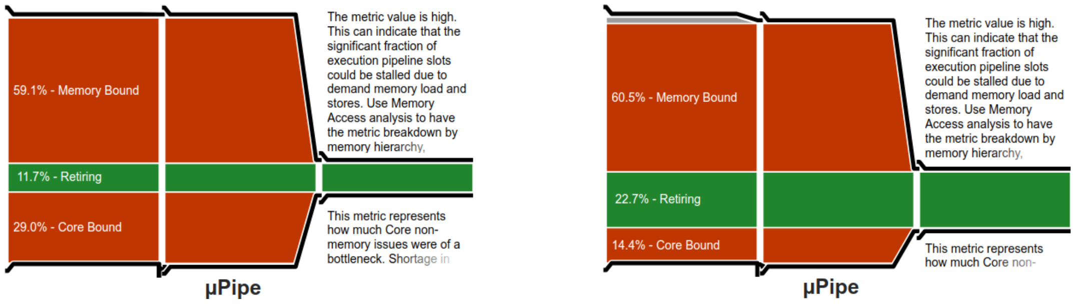
(a) Hotspots Analysis (-O2)

Function	Module	CPU Time	% of CPU Time
__intel_avx_rep_memcpy	heat	3.212s	41.6%
relax_jacobi	heat	2.711s	35.1%
residual_jacobi	heat	1.568s	20.3%
_IO_fprintf	libc.so.6	0.192s	2.5%
initialize	heat	0.020s	0.3%
[Others]	N/A*	0.016s	0.2%

*N/A is applied to non-summable metrics.

(b) Hotspots Analysis (best)

Figure 1.3: Hotspots Analysis



The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory Access analysis to have the metric breakdown by memory hierarchy,

This metric represents how much Core non-memory issues were of a bottleneck. Shortage in

(a) Microarchitecture Usage (-O2)

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory Access analysis to have the metric breakdown by memory hierarchy,

This metric represents how much Core non-

(b) Microarchitecture Usage (best)

Figure 1.4: Microarchitecture Usage

As shown in Fig 1.4, although the efficiency of microarchitecture utilization has improved, the program is still primarily memory-bound, where 60.5% of pipeline slots could be stalled due to demand memory loads and stores. Notably, for the optimized version, a gray area has appeared at the top of the image, which represents a slots fraction of 2.6% where the processor's front-end undersupplies its back-end.

1.3 Sequential Optimization

We have modified and optimized the code for sequential execution as follows:

- Access pattern into the matrix:

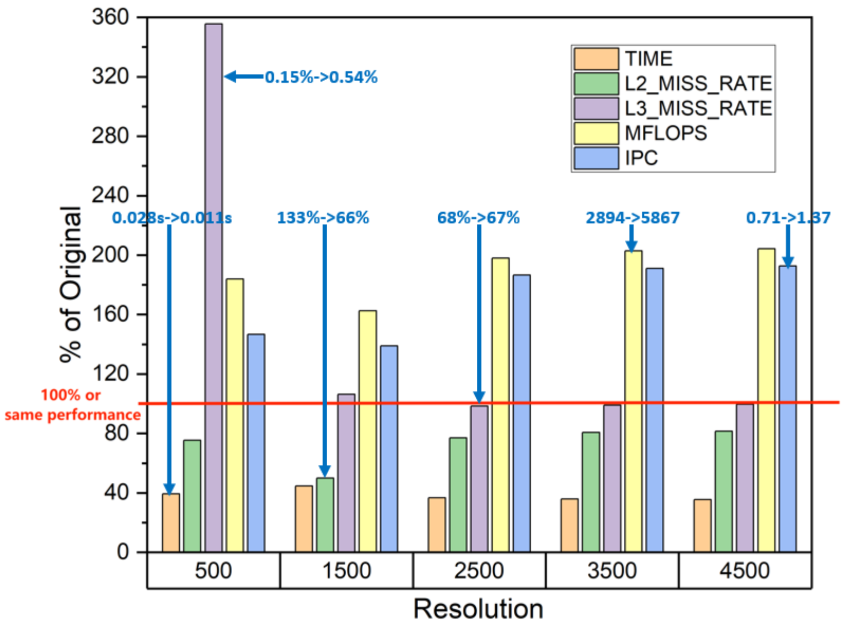


Figure 1.5: Manually Optimized Version Vs. Original Version

We swap the order of loops in the `relax_jacobi` and `residual_jacobi` functions. Since the matrices are stored in row-major order, iterating over the `i` index in the outer loop and the `j` index in the inner loop will promote better data locality and cache utilization, resulting in better performance.

- Calculation of residuum:

  In the original implementation, the functions `relax_jacobi` and `residual_jacobi` are called separately in each iteration, resulting in significant duplicate computations. To address this, we merge these two functions into a single function that returns the residual after each iteration. By doing so, we reduce the number of FLOP per Jacobi iteration at each point from 11 to 7.

- Avoid copy operation:

  Additionally, we eliminate the unnecessary copying of data between the `u` and `utmp` arrays in the `relax_jacobi` function. Even if the compiler optimizes it, memory copy still incurs significant time overhead. By just swapping the two pointers to `u` and `utmp` after each iteration, we avoid unnecessary memory copy and improve the overall performance.

As illustrated in Figure 1.5, for different resolutions, the runtime is reduced to almost 40% of the original version, while MFLOPS increases by up to 104%. The 60% reduction in runtime does not lead to an approximately 2.5× increase in MFLOPS because the optimized version also experiences a decrease in the total number of FLOP. In addition, IPC (Instructions Per

Cycle) also increases by up to 92% going from less than 1 to greater than 1, which suggests that the processor is achieving some level of instruction-level parallelism, executing multiple instructions concurrently, or experiencing optimizations that lead to improved efficiency and performance.

In addition, due to the improvement in memory access patterns, the L2 cache miss rate has significantly decreased compared to before, while the L3 cache miss rate has remained almost unchanged (except at a resolution of 500). This may be because the decrease in L2 cache miss rate results in a reduction in the total number of L3 cache accesses, thereby somewhat increasing the L3 cache miss rate. At a resolution of 500, the L3 miss rate exhibits a dramatic relative change, but in reality, it only increases from 0.15% to 0.54%, resulting in an absolute change of 0.39%.

1.4 OpenMP

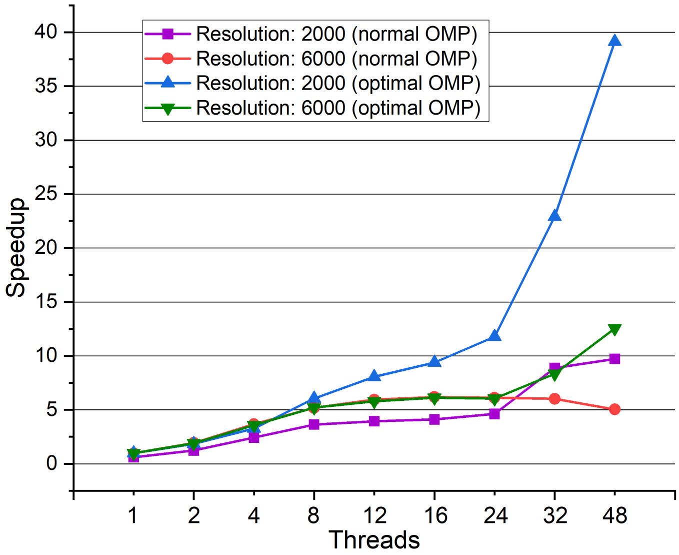


Figure 1.6: Optimal OpenMP Scalability

For the version of normal OpenMP, we just add a line of code `#pragma omp parallel for private(j, unew, diff) reduction(+:sum) schedule(static)` before the outer loop of the Jacobi iteration. The reason we do not also place this line of code before the inner loop, or exclusively before the inner loop, is to avoid significant time overhead caused by multiple fork-join operations. Apart from that, the reason we choose static scheduling instead of dynamic, guided, or auto scheduling is that the workload difference between different iterations is not significant, and we want to explicitly know the mapping relationship between iterations and OpenMP threads before running the application (For example, the iterations from 0 to n-1 will be assigned to the 0th OpenMP thread, and the iterations from n to 2n-1 will be assigned to the 1st OpenMP thread, and so on.). This allows us to further establish affinity between OpenMP threads and physical threads, optimizing overall performance. Additionally, when we do not specify a chunk size for the schedule, the iterations will be evenly distributed among all the utilized OpenMP threads (with any remaining iterations evenly distributed among the earlier OpenMP threads when the total number of iterations is not divisible by the number of threads). This ensures that the memory required for iterations executed within the same OpenMP thread is as contiguous as possible, effectively utilizing cache locality. As a result, the performance of the normal OpenMP is similar to the performance of auto-parallelization offered by the Intel compiler.

Furthermore, based on the normal OpenMP implementation, our optimal test results are achieved when utilizing the "first touch" policy in NUMA systems, without specifying affinity. To do this, only one line of code `#pragma omp parallel for schedule(static)` is added before the outer loop of the initialization for `u` and `uhelp`. This is the first access to `u` and `uhelp`, and we are using the same OpenMP configuration and memory access pattern as in the Jacobi iteration. This ensures that the data pages allocated for `u` and `uhelp` are placed in the memory closest to the thread accessing them for the first time, following the First Touch Placement Policy. This means that `u` and `uhelp` are evenly distributed among the cores rather than being concentrated on a single core, avoiding potential time overhead caused by cross-socket access.

Even if we have not explicitly specified the affinity setting, the default environment variable `KMP_AFFINITY=granularity=thread,compact,1,0` will take effect. This particular affinity setting ensures the desired affinity mode for our application. The finest granularity level "thread" causes each OpenMP thread to be bound to a single physical thread avoiding the floating of OpenMP threads between two threads of the same core or different physical threads within the same node. Furthermore, `compact,1,0` ensures that OpenMP threads are compactly bound in order at the core level without any offset, which not only avoids the potential performance degradation caused by hyper-threading but also maximizes the contiguity of `u` and `uhelp` within a single socket. In an ideal situation, the upper half of the grid is contiguous and closer to one socket, while the equally contiguous lower half is closer to the other socket.

As shown in Figure 1.6, when the resolution is 2000, the optimal version of OpenMP heat shows a significant improvement in performance, especially in the context of 32 and 48 threads. The speedup for the version utilizing "first touch" is almost four times that of the original OpenMP version at most, achieving up to about 40×. This is attributed to our full utilization of the "first touch" and the compact affinity to minimize the time overhead of cross-socket access when using more than 24 threads. However, this performance improvement is not significant when using fewer than 24 threads within the same socket.

At a resolution of 6000 with the number of threads less or equal to 24, the utilization of the "first touch" policy doesn't appear to offer significant performance improvements. However, for 32 and 48 threads, the optimization based on "first touch" does bring a moderate increase in speedup, which is because the memory requirement for a resolution of 6000 is substantial, making the performance improvement from "first touch" less evident.

1.5 MPI Parallelization

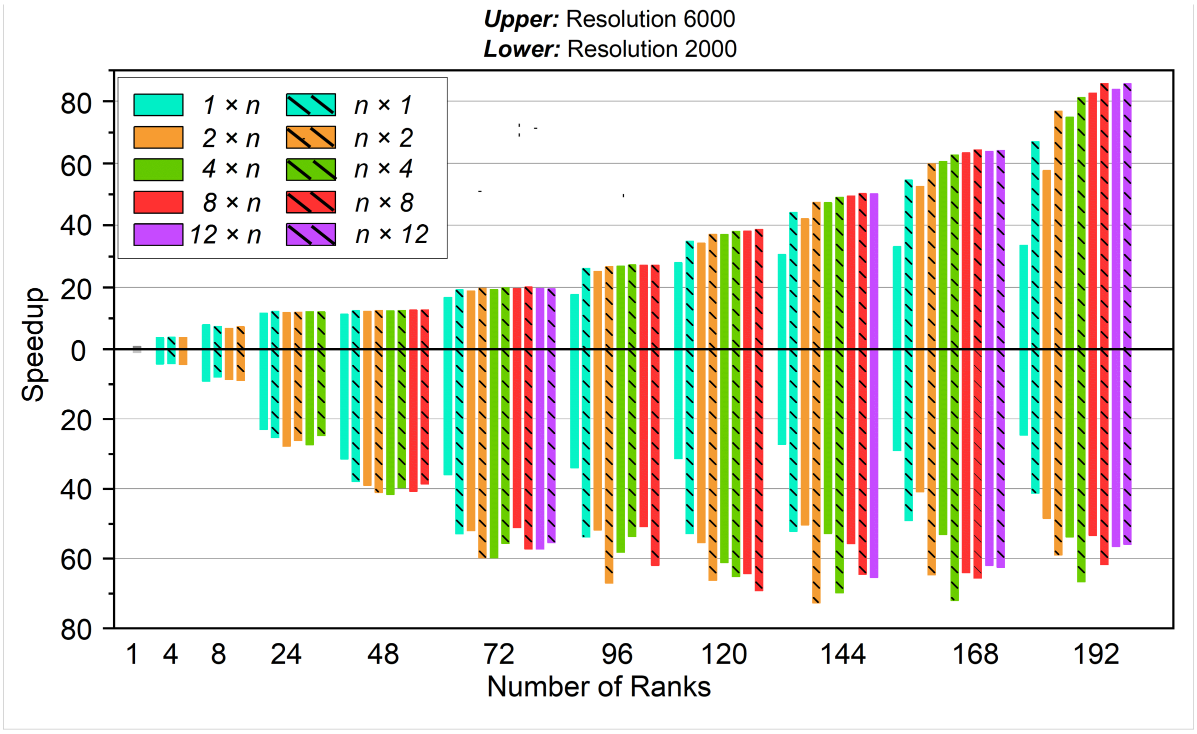


Figure 1.7: MPI Heat Scalability (Blocking Communication)

We evenly distribute the interior of the grid, excluding the boundaries, among the two-dimensional MPI processes. In addition, each process is allocated an extra surrounding ghost area. During initialization, the boundary values of the global grid are stored in the ghost area of the outer processes. For blocking communication, before each Jacobi iteration, we use four successive `if` statements to determine the position of the current process and make it send and receive information to and from the corresponding direction where we use `MPI_Sendrecv` to avoid potential deadlock. After the communication is completed, the original Jacobi iteration is executed on each process. For non-blocking communication, we overlap communication and computation. After the communication is completed, the values of the outermost layer, excluding the ghost area, are updated on each process.

For `coarsen`, we iterate over the grid on each process, considering only the contribution of the points within the current process's region. After the coarsening process, the `uvis` values are reduced from each process. The sequential version of the `coarsen` function has a bug where some points are redundantly traversed. To ensure consistent results, we have to use the same traversal pattern in the coarsening process. As a result, the final output matrix `uvis` has a numerical error of 1 at the last digit at a few points due to the floating-point operation ordering, but the overall logic and the resulting image remain consistent with the sequential version.

Based on the experimental results of Figure 1.7, when the number of nodes ranges from 1 to 2, the speedup for resolution 2000 and resolution 6000 both improves. Notably, the speedup for resolution 2000 is significantly higher than that for resolution 6000. When fully utilizing 2 nodes, the maximum speedup for resolution 2000 can reach approximately 67× (achieved with a topology of 2 × 48), while the maximum speedup for resolution 6000 only reaches around 27× (achieved with a topology of 4 × 24).

When the number of nodes increases to 3 and 4, the maximum speedup for resolution 2000 almost stabilizes. However, the performance differences between different process configurations have become more pronounced. Surprisingly, in some cases, the maximum speedup with 192 processes (approximately 65×) is lower than that with 168 processes (approximately 70×). On the other hand, the maximum speedup for resolution 6000 continues to steadily increase. With 192 processes, the maximum speedup approaches 85× (achieved with a topology of 12 × 16). This upward trend shows no signs of slowing down, indicating that using more nodes could potentially achieve higher speedups for resolution 6000.

By comparing different MPI topologies with an equal number of processes, we have observed that swapping the numbers of rows and columns of the topology leads to performance changes. This change becomes particularly evident when there is a significant difference between the number of rows and columns. For example, in the case of 192 processes, switching the topology from 1 × 192 to 192 × 1 can result in a speedup change of approximately 100% for both resolutions. This observed variation stems from the different shapes of processes in the two different MPI topologies after dividing the square global grid. The 1 × 192 topology corresponds to a "tall-skinny" matrix shape for each process, while the 192 × 1 corresponds to a "short-fat" shape. When each process communicates with its left and right neighbors, the required information is not contiguous in memory, which necessitates multiple calls to MPI communication functions within a loop or the allocation of additional temporary arrays for memory copying specifically for communication with the left and right neighbors. The "tall-skinny" process structure amplifies the time overhead of communication with the left and right neighbors. On the contrary, the "short-fat" process structure minimizes this overhead, leading to a disparity in speedup between the two topological configurations.

Apart from that, we also compare the performance of blocking communication and non-blocking communication for the heat application. The performance of blocking communication is even slightly better than non-blocking. This could be because in this application, the workload of computation and communication is not balanced, and different processes have different workloads. As a result, even though computation and communication can overlap, there is a significant amount of time spent waiting for all processes to complete their communication before continuing with the outermost computation on each process (excluding the ghost area).

1.6 Compare Hybrid Parallelization to Pure MPI Parallelization

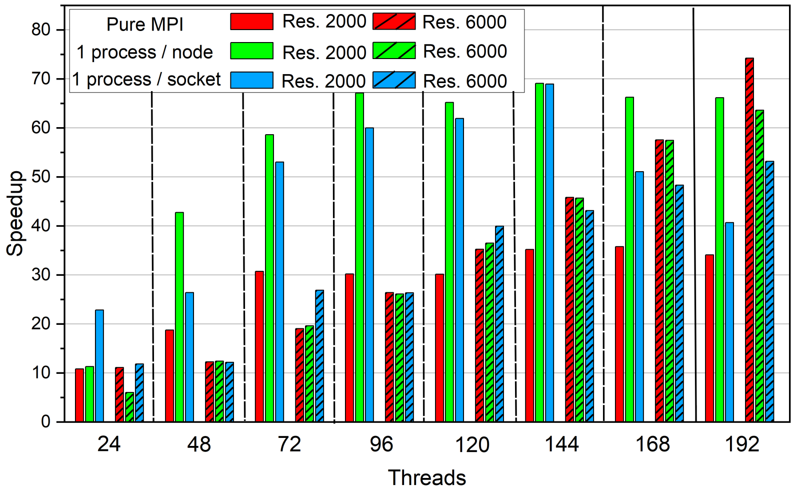


Figure 1.8: Hybrid Parallelization Vs. Pure MPI Parallelization

Because for the heat application blocking and non-blocking communication have almost the same performance and the former performs slightly better based on our experiment in the previous section, only blocking communication is applied and investigated in this section. And the MPI virtual topology here is always based on the optimal configuration in the version of the pure MPI blocking communication. For hybrid parallelization, we investigate two scenarios that utilize exact one process per node or exact one process per socket. Additionally, because we only use a new node when all 48 physical cores are used on the already allocated nodes and always utilize two sockets evenly in the same node for the case of "one process per socket", the experimental configurations shown in the Figure 1.8 are unique.

In the case of resolution 2000, compared to the pure MPI, hybrid parallelization consistently improves the speedup ranging from 85% to 100% across different thread counts. This is because the MPI inter-process communication with higher overhead is significantly reduced and the advantages of shared memory communication come into play. This can also be similarly applied to explain why, in many cases, the "1 process per node" scheme is generally better than the "1 process per socket" scheme.

However, in the case of resolution 6000, the performance improvement from hybrid parallelization is not significant, and it even performs noticeably worse than the pure MPI when using 192 threads. This could be because the increased resolution introduces a significant amount of communication overhead between nodes, which offsets some of the performance improvements from hybrid parallelization. Furthermore, the differences between two hybrid parallelization schemes are also not pronounced for the resolution of 6000. All in all, hybrid parallelization combines the advantages of two techniques and demonstrates its scalability and flexibility in certain scenarios, but it does not always outperform a single parallelization scheme and proves to be more challenging to design and tune.