Efficient Programming of Multicore Processors and Supercomputers

Group 1: A. I. Arslanpay, A. B. Temurer, and T. Zheng

June 29, 2024

1 Compiler Flags

In this section we will focus on the following compiler flags: *-O0, -O1, -O2, -O3, -Ofast, -xhost, -ipo, -fno-alias, -xCORE-AVX512*. We will first discuss about the optimization flags without any optional flags and then using flag -O3 to discuss about optional flags.

1.1 Basic Compiler Optimization Flags

Figure 1 illustrates the performance trends in MFLOPS (Million Floating-Point Operations per Second) for different optimization flags across various problem resolutions. As a whole, there is huge performance improvement between optimization flags -O0, -01 and -O2. The performance difference between flags -O3, -Ofast and -xhost is not obvious, but a abnormal increase of performance from resolutions 3500 to 4500 is observed. Possible reasons for this phenomena could be:

- Cache Effects: Larger resolutions mean more data needs to be processed. If the problem size increases but still fits within the cache hierarchy of the CPU, it can lead to better cache utilization. As the resolution grows, data locality may improve, resulting in fewer cache misses and more efficient memory accesses.
- Instruction-Level Parallelism (ILP): Higher optimization levels (*-O2, -O3, -Ofast, -xhost*) typically enable more advanced techniques to exploit ILP. With more data to process, the compiler might be able to better schedule instructions and utilize CPU pipelines more efficiently.
- Reduced Overhead of Initialization and Finalization: For smaller resolutions, the relative overhead of initialization and finalization (e.g., setting up data structures, managing memory) can be higher compared to the actual computation time. As the problem size grows, this overhead becomes a smaller fraction of the total execution time, allowing the actual computation to dominate, which can lead to higher MFLOPS.
- Loop Optimizations: Larger problem sizes can provide more opportunities for the compiler to apply loop unrolling, vectorization, and other loop transformations. These optimizations can significantly boost performance, especially at higher resolutions.

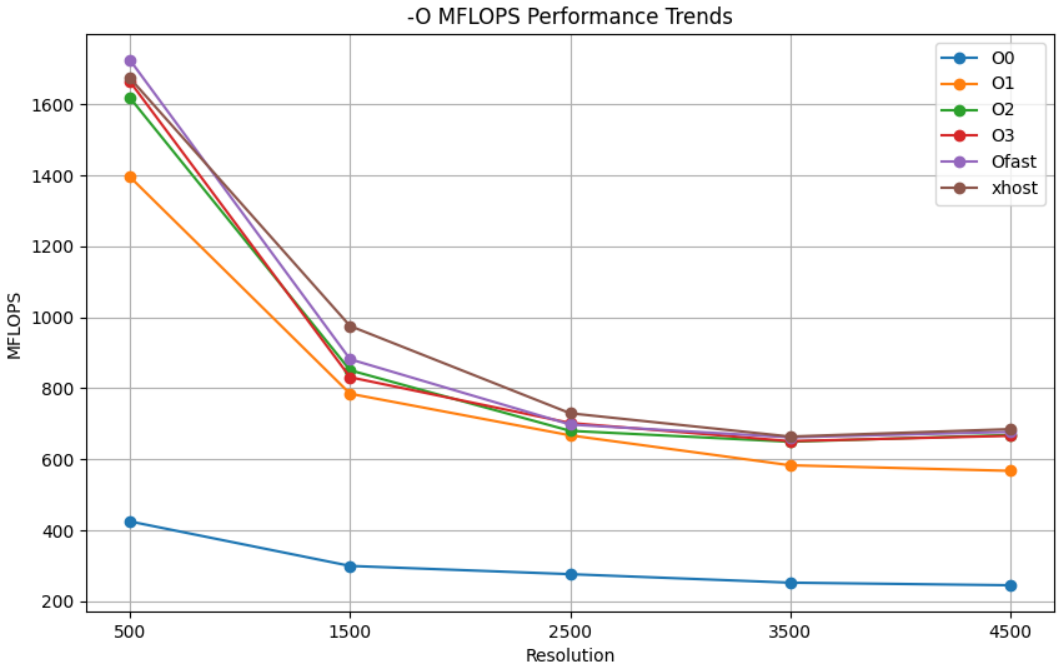

Figure 1: MFLOPS Performance Trends Across Various Optimization Flags for Different Resolutions.

1.2 Other Compiler Flags

Figure 2 illustrates the performance trends in MFLOPS for various optional flag combinations across different problem resolutions. The combination of *-O3 -fno-alias -xCORE-AVX512* delivered the highest performance across all resolutions, with significant gains in MFLOPS. This indicates that leveraging AVX-512 instructions and assuming no aliasing can lead to substantial improvements. Potential reasons for it could be:

- Utilizes advanced compiler optimizations to transform the code for maximum execution speed.
- Leverages the AVX-512 instruction set for wide vector operations, enabling parallel processing of multiple data points.
- Assumes no pointer aliasing, allowing more aggressive optimizations of memory accesses and reducing overhead.
- Exploits the full potential of modern processors' capabilities, leading to significant performance improvements, especially for floating-point intensive tasks.

All combinations that included -xCORE-AVX512 showed better performance compared to their counterparts without it, highlighting the benefits of using advanced SIMD instructions on supported hardware. The use of -fno-alias consistently improved performance by

allowing more aggressive optimizations, especially when combined with other flags like -xCORE-AVX512.

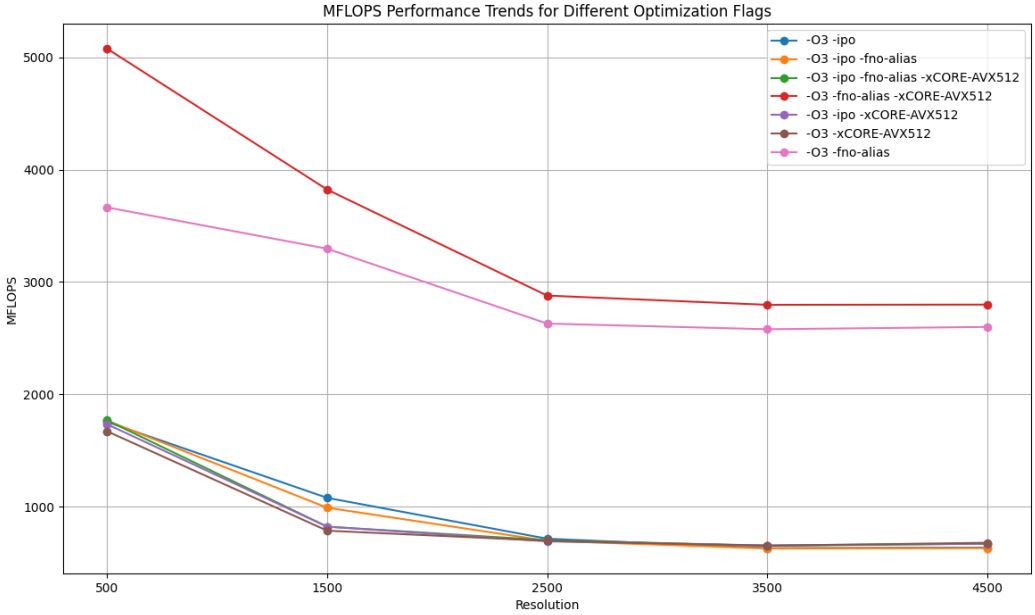

Figure 2: MFLOPS Performance Trends Across Various Optional Flags for Different Resolutions Using Optimization Flags -O3.

2 Profiling

Profiling tools, like PAPI and Intel VTune, are important for assessing the performance of an application. For this purpose, aforementioned tools were used to compare different compiler flags.

Using L2 and L3 cache accesses and misses, miss rates for these cache levels were calculated. This process was done for "-O2" and "-O3 -fno-alias -xhost" compiler flags. The results, seen in Figure 3, show a significant increase in Mflops for the compilation with "-O3 -fno-alias -xhost". For the miss rates, we see that L3 cache miss rates are quite similar between both compilations, but the difference can be seen for L2 cache miss rates. but this difference seems quite unintuitive. The reason behind this interesting result is that for these comparisons, comparing only the miss rates is deceiving. If we check the total cache accesses for the L2 cache, we can see that total access is much lower for the "-O3 -fno-alias -xhost" compilation(around 10% of "-O2" compilation for every resolution). Since there are ambiguities in what PAPI counters count a miss or a cache access, some of the miss rates are over 100%, which is not expected.

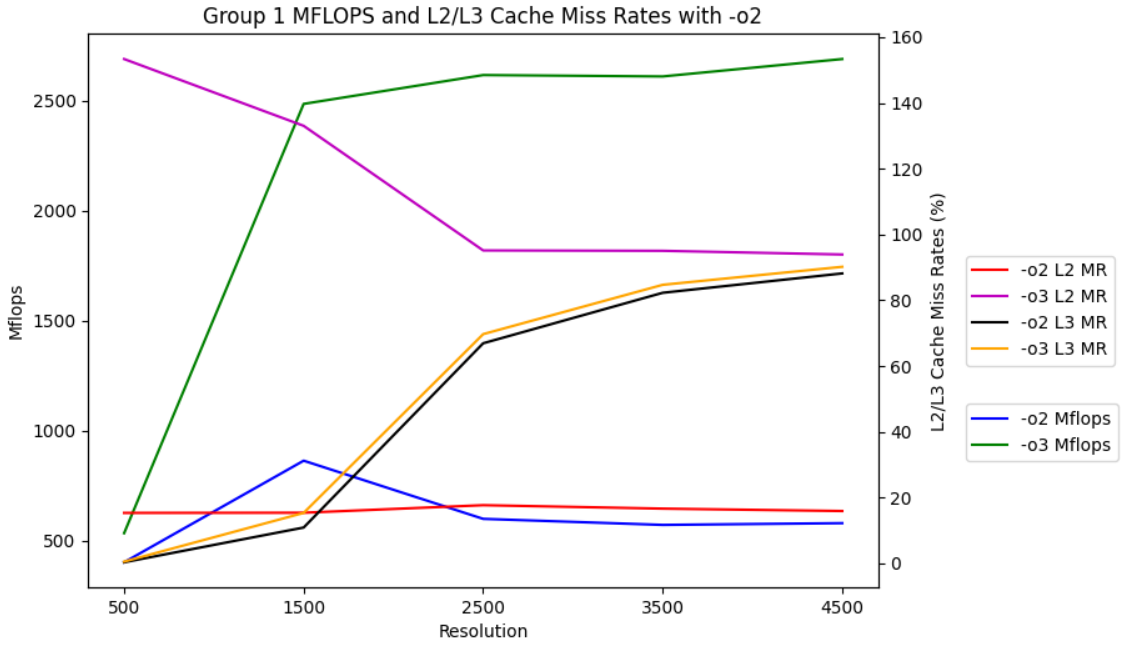

Figure 3: Speedup comparison of with and without first touch policy used in the data initialization.

3 OpenMP

Comparison of the speedups were done on the heat simulation parallelized using OpenMP to compare the effectiveness of utilization of first touch policy, and differences between different thread to core mapping techniques for different resolutions and numbers of threads.

3.1 Effect of First Touch Policy over Speedup

The idea behind the first touch policy is that data pages are allocated as close to where the thread which is accessing to that memory is running. By parallelising the data initialization, all threads get to be the first one to access the memory locations that they will be working on later. Thus, their share of the data is allocated closer to them. As it is seen in the Figure 4, utilising the first touch policy in the data initialization process in the beginning provides a significant increase in speedup for certain cases.

The effect of first touch policy is much more visible for lower resolutions and higher numbers of threads. The reason why lower resolutions get affected more is that the data initialization part causes a higher ratio of overhead to actual computations compared to higher resolutions, and parallelising this overhead affects the overall performance more for lower resolutions. As for the explanation of the relation between first touch policy and number of threads, there can be seen a jump in speedup between 24 threads and 32 threads, especially for the cases with first touch policy being utilized. This jump is caused by the

number of cores being 24 on the two CPUs of single node of SuperMUC-NG. When the number of threads go from 24 to 32, it is certain that both CPUs are being utilized by the application and some of the threads of this application are mapped to the second CPU. Thus, it becomes more important to allocate the memory pages on the memory socket that is closer to the thread accessing that memory so that there is less memory accesses done across different sockets.

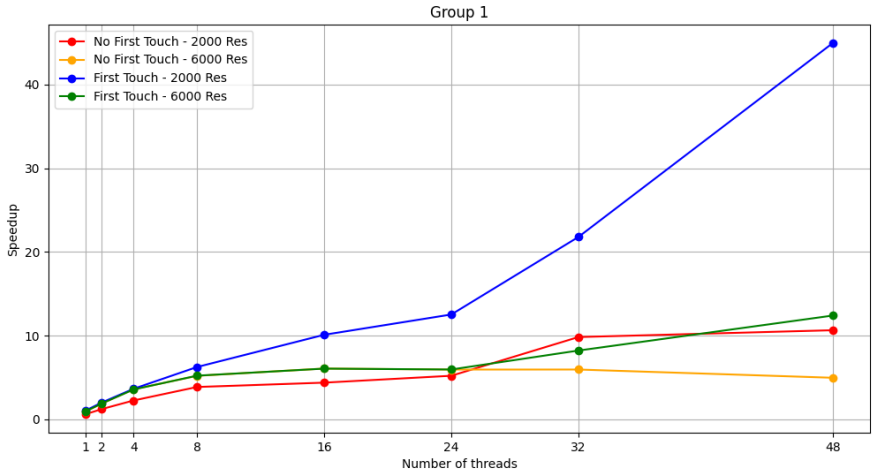

Figure 4: Speedup comparison of with and without first touch policy used in the data initialization.

3.2 Speedup for Different Thread to Core Mapping

The two mapping types tested are called scatter and compact. To change the default mapping, an environment variable with the name KMP_AFFINITY was changed accordingly. For scattering, the system maps threads to cores as evenly as possible, meaning that threads with close IDs are mapped distantly. For compact mapping, the opposite of scattering, each thread is mapped to cores as closely to the threads with close IDs as possible.

Speedup differences between these two mappings, as seen in the Figure 5, are caused by the better usage of memory bandwidth. For the case of scattered threads, these threads will utilize the memory bandwidth in better ways since they are mapped farther from each other, and most probably on different CPU sockets on the node. But after some increase in number of threads, the speedup either gets stable or decreases. The reason for this might be that, the number of threads in a single socket gets too much and causes congestion on the memory bus. This affects lower resolutions less, since the arrays are smaller and caches are used much more efficiently. And for the compact mapping, threads are mapped to the cores as close as possible. This causes an early congestion on the memory bus and a worse overall performance in our application, but the effect of number of threads to this performance drop is less significant for this mapping.

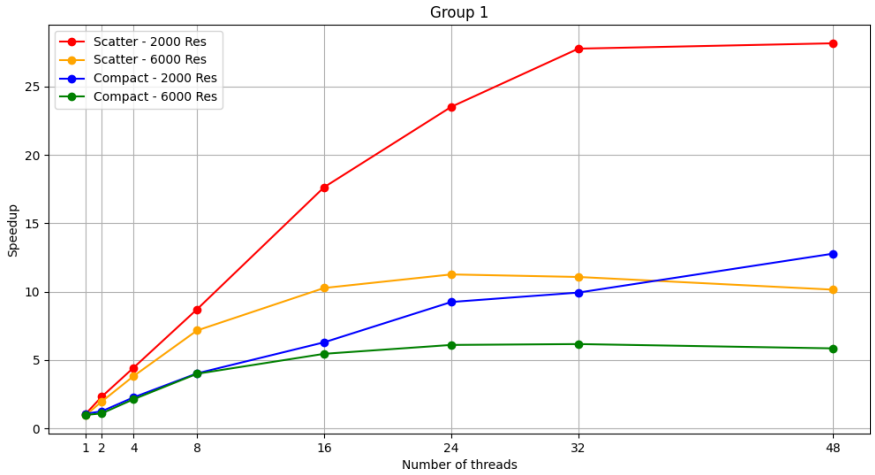

Figure 5: Speedup comparison of thread to core mapping using environment variables for resolutions 2000 and 6000 and different numbers of threads.

4 Heat MPI Parallelization

The results presented analyze the speedup of a Heat MPI application at resolutions of 2000 and 6000, comparing blocking and non-blocking communication methods. The analysis is performed using up to four nodes, ensuring that new nodes are added only when all physical cores on the current nodes are fully utilized to maintain efficiency.

4.1 Speedup Comparison for Different Resolutions

Figure 6 compares speedups for resolutions 2000 and 6000 across different configurations for both blocking and non-blocking communication.

There is a consistent increase in speedup as the number of cores increases in both blocking communication and non-blocking communication. However, when the resolution was 2000, an increase in speedup was observed as the number of cores increased up to 80, and then, as the number of cores increased, it was observed that unlike the resolution 6000, speedup got stuck between a certain band (between 33-39).

While the resolution was 6000, a speedup similar to the 2000 resolution was observed for blocking and non-blocking communication in the 16x7 and 8x14 (112 cores total) configuration. While the number of cores used was less than 112, higher speedup values were achieved at 2000 resolution, while as the number of cores increased, higher speedup values were achieved at 6000 resolution. This indicates that the application scales better with larger problem sizes. This situation is because at lower core counts, the computational load per core is relatively high, and the communication overhead is proportionally lower. This results in better utilization of each core, hence higher speedup for lower resolution. As the core count increases, the computational load per core decreases, but the communication overhead doesn't scale down at the same rate. The amount of computation per core for higher resolutions remains significant even with more cores, making them less affected by

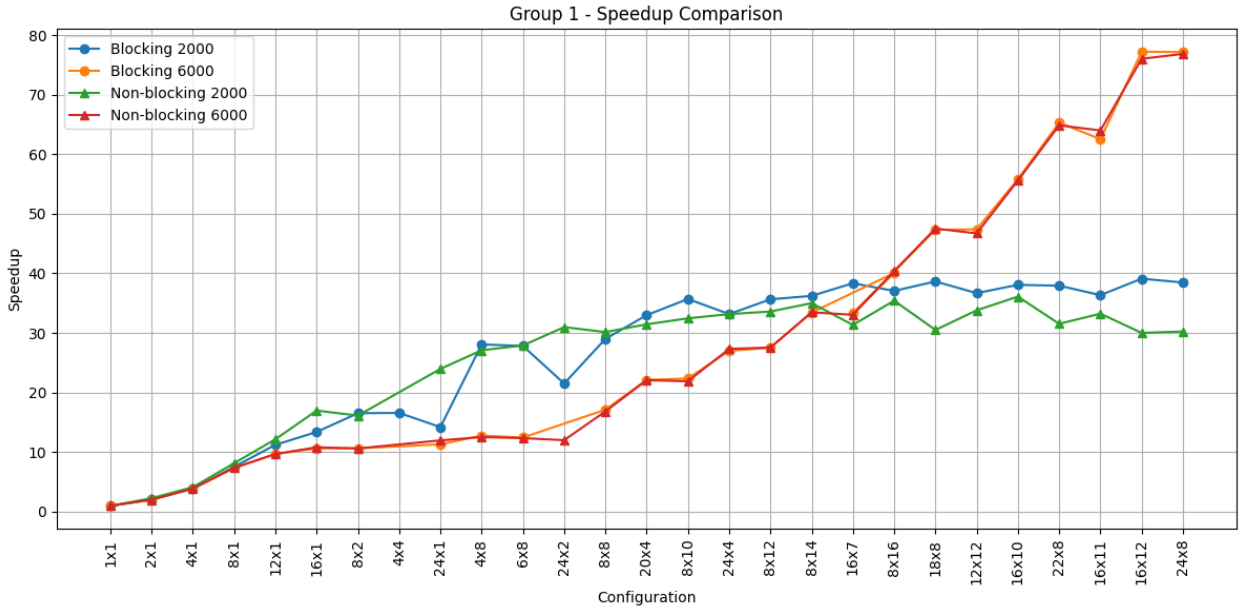

Figure 6: Speedup comparison of blocking and non-blocking MPI implementations at resolutions 2000 and 6000 across various configurations.

communication overhead. For instance, the maximum speedup at 2000 resolution is 39.11 (16x12), while at 6000 resolution, it reaches 77.22 (16x12).

In the comparison of blocking communication and non-blocking communication, it was observed that when the resolution was 2000, non-blocking communication showed a slightly better performance in configurations created with a low number of cores (up to 64). This suggests that non-blocking communication can be more efficient for lower resolutions due to lower overhead and more effective parallelism. However, as the number of cores increases, the advantage shifts towards blocking communication; blocking communication performs better. Due to the increased efficiency in handling communication at larger scales.

When the resolution was 6000, and the number of cores used in the configurations was less than 144, blocking communication and non-blocking communication showed similar performance. However, it has been observed that blocking communication performs better by a small margin, especially when the number of nodes is greater than 3, with higher core counts and larger configurations. This indicates that the overheads in non-blocking communication may outweigh the benefits at higher scales. For example, with blocking communication, the maximum speedup at 6000 resolution was 77.22 (16x12), while with non-blocking communication, the maximum speedup was measured as 76.86 (24x8). A similar situation was observed when the resolution was 2000. With blocking communication, the maximum speedup was observed as 39.11 (16x12) at 2000 resolution, while in non-blocking communication, the maximum speedup remained at 36.1 (16x10).

Figure 7 shows the speedup achieved with different configurations using blocking and non-blocking communication at 6000 resolution and in configurations where blocking communication gives better results, corresponding to most configurations. The x-axis represents various configurations, while the y-axis shows acceleration.

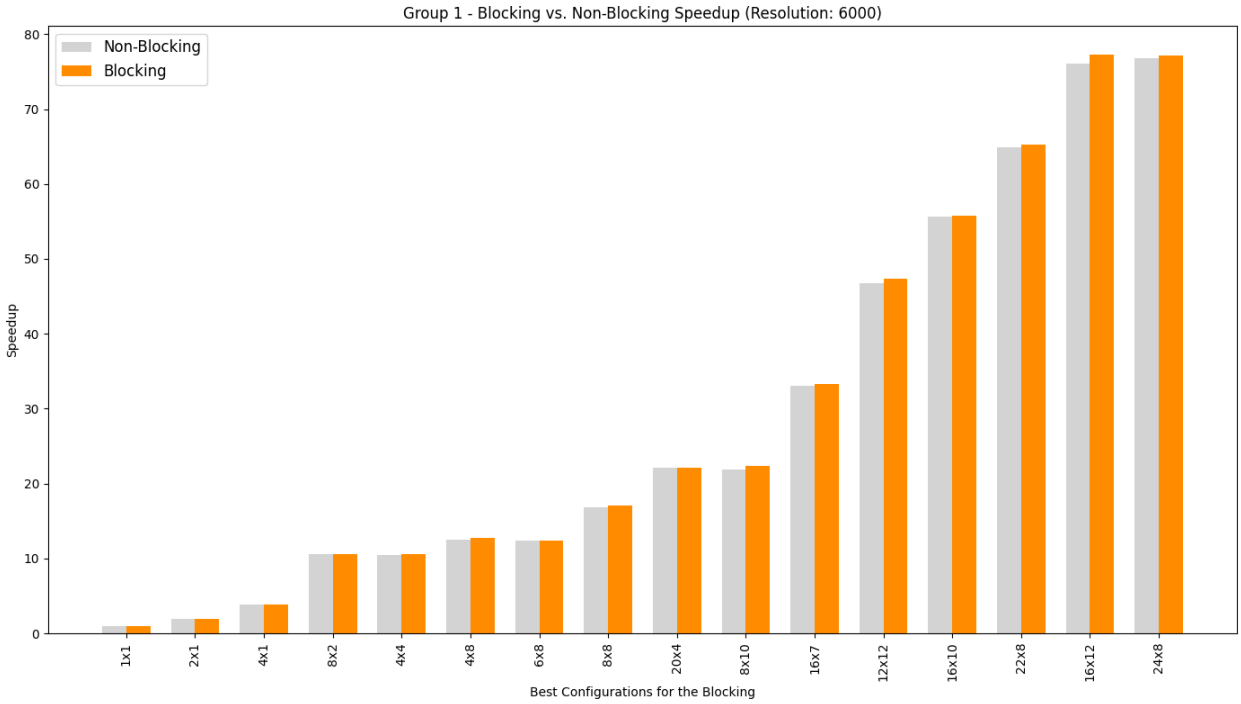

Figure 7: Speedup comparison of blocking and non-blocking configurations at resolution 6000 across various configurations.

As seen in Figure 7, the speedup value also increases as the number of cores increases in both blocking and non-blocking communication. Also, blocking communication consistently outperforms non-blocking in terms of speedup for most configurations. This situation becomes relatively more evident, especially when the number of nodes used is more than 3. The highest speedup value obtained was observed in blocking communication as 77.22 in the 16x12 configuration. As mentioned above, it is clearer in Figure 7 that the application scales better with larger problem sizes.

4.2 Efficiency Considerations

4.2.1 Node Utilization:

- Adding a new node only when all physical cores on the already used nodes are busy is a crucial strategy to maintain efficiency. This approach minimizes the idle time of cores and maximizes resource utilization.

- The graphs indicate that this strategy is effective, as we see a clear improvement in speedup up to the point where adding more nodes becomes necessary.

4.2.2 Communication Overhead:

- Non-blocking communication, while theoretically reducing idle times by allowing computation and communication overlap, introduces complexity and potential inefficiencies

due to synchronization and management overheads.

- Blocking communication, despite being simpler, benefits from reduced management overhead and can be more predictable in performance, especially as the number of cores increases.

5 Compare Hybrid Parallelization to Pure MPI Parallelization

The performance comparison between pure MPI and hybrid (MPI+OpenMP) parallelization reveals different results in terms of efficiency and scalability across different configurations and resolutions.

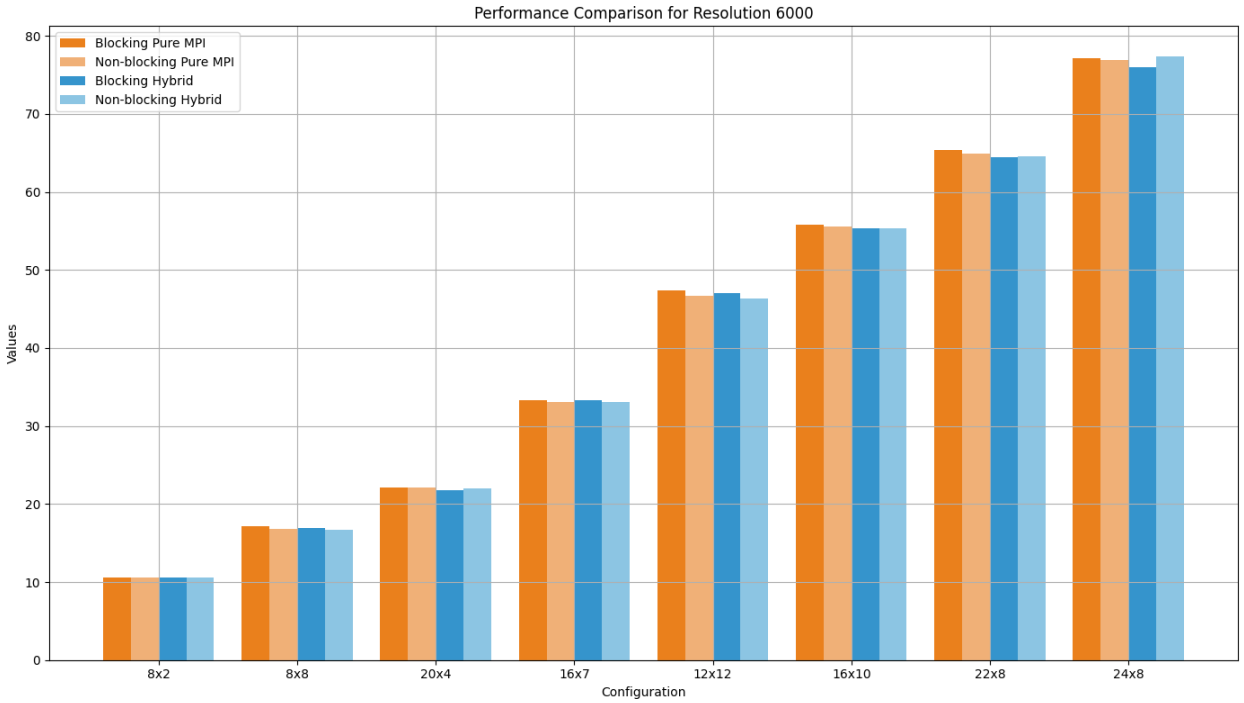

Figure 8: Comparison of blocking and non-blocking MPI and hybrid performance at resolution 6000 across different configurations.

As mentioned in the previous section, for 6000 resolution, blocking communication consistently outperforms non-blocking communication in a pure MPI setup. As seen in Figure 8, in the hybrid configuration, it has been observed that non-blocking communication gives better results with a very small difference. Only in the 24x8 configuration is a relatively notable difference in the hybrid solution; While the speedup was 77.37 in non-blocking, it was observed as 75.97 in blocking communication.

In contrast, as seen in Figure 9, the blocking hybrid method performs better in most configurations than its non-blocking counterpart, especially evident in 20x4, 16x7, 12x12, 16x10, 22x8, and 24x8. This indicates that the overhead of non-blocking communications

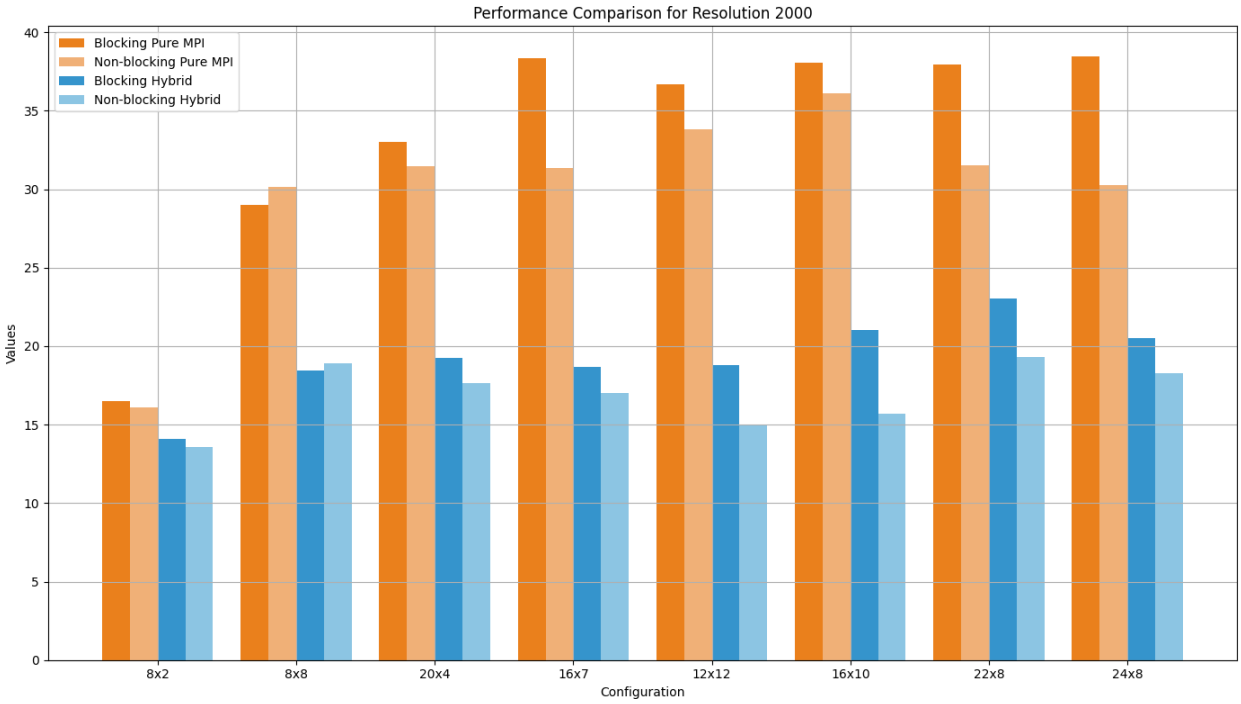

Figure 9: Comparison of blocking and non-blocking MPI and hybrid performance at resolution 2000 across different configurations.

may not be effectively offset by the performance gains from overlapped computation and communication at lower resolutions.

5.1 Practical Implications and Recommendations

- **Scalability Considerations:** Pure MPI tends to offer better scalability across varied configurations and resolutions, possibly due to its straightforward communication model compared to the hybrid approach that requires managing both inter-node and intra-node parallelisms.
- **Complexity vs. Performance Trade-offs:** While the hybrid model allows for more granular control over computational resources, its benefits must be weighed against the increased complexity. The decision to employ a hybrid model should consider specific application needs, problem size, and system capabilities.

Hybrid parallelization presents a versatile yet complex approach to computational tasks, combining distributed and shared memory processing. However, its advantages over a pure MPI model are not universally applicable and heavily depend on particular problem characteristics and resource constraints.