# OpenMP and MPI Parallelism: A comparative performance analysis

Saskia Drossaart van Dusseldorp, Dario Peier, and Colin Tully

7th August 2020

**High Performance Computing for Weather and Climate, ETH Zürich, SS 2020**
Supervisors: Tobias Wicky, Oliver Fuhrer

In this report we describe a benchmarking exercise that we performed on three different versions of a stencil program used for calculating high-order monotonic numerical diffusion. The first version applies distributed memory parallelism only, using the message passing interface MPI, and the latter two integrate shared memory parallelism, using OpenMP, with do-loop optimization. We performed several sets of tests on all versions, with different configurations of the numbers of nodes, MPI ranks and OpenMP threads. To asses and compare the performance of the codes at given configuration, we use run-time as a proxy. We find that combining MPI and OpenMP has no favorable effect on performance in our example. A first step of loop optimization improves code performance, which we attribute to the omission of a temporary memory field. Further optimization has a negative impact on the run-time. We find the code to run most efficiently using the first loop optimization step and running on four nodes with 96 MPI ranks. Finally we discuss shortcomings of our experiments and suggest improvements regarding further investigation.

# 1 Introduction

Solving the complex dynamics of weather and climate in order to provide a forecast or projection requires a great amount of computing resources, beyond what a conventional personal computer can feasibly handle. Supercomputing on large computer clusters spread around the globe are employed to run such large simulations. The advantage of using a cluster is that programs can utilize parallelism in order to optimize their usage of computing resources, with the goal of using as little as possible. However, scientists who wish to write parallel programs for such computer clusters nowadays have to face a highly hierarchical system design, such as the hierarchy on the PizDaint cluster run by the Swiss National Computing Center (CSCS), which is used in this project.
Parallelism (or parallel computing) refers to the simultaneous execution of several computations. Out of the numerous parallel computing concepts, we focus on two that utilize specific cluster architecture. The first is the Message Passing Interface (MPI), which distributes memory for the computation across more than one node as nodes do not share memory. Each node, in a modern computing cluster like Piz Daint, will consist of some number of CPUs that perform the bulk of the computation over several cores. Within a single node memory is shared, and parallelism of the computation can also be applied across several threads using Open Multi-Processing (OpenMP).
Both parallelism methods provide an opportunity to enhance computational performance. For example, it is possible to use only MPI, treating every CPU core as a separate entity with its own address space by assigning it a rank. Similarly, one may also employ a mixed model, where more than one MPI rank is run with multiple threads on a single node. However, programming a hybrid model that uses OpenMP for parallelization within a single node and MPI for parallelization over several nodes may provide a greater opportunity to enhance computational performance. For all of these three approaches it is important to note that by increasing the number of processors involved in the parallel code, the performance does not increase infinitely according to Amdahl's law [Amdahl, 1967]. It states that increasing parallelization leads to an increasing overhead caused by synchronizing communication between the processors, and to an increasing load imbalance between the workers (i.e threads or ranks). Increasing the performance through more parallelization alone ,therefore, has its limits. One can can attempt to counteract this by using OpenMP scheduling that aims to evenly distribute the load (Section 3.2).
The synchronization overhead can also be mitigated through optimization of the code, namely loop optimization and improvement of cache performance. The idea is to write the code in a way that reading from and writing to memory are optimized. Introduction or omission of temporary memory fields, which is done by frequency reduction or loop jamming respectively, can have a significant impact on the performance of a code that is machine-independent.
In this project, we apply OpenMP and MPI parallelism in order to solve the high-order monotonic diffusion scheme by Xue [2000], and investigate the differences in performance using various configurations of MPI ranks and OpenMP threads on a different number of nodes. As stated by Rabenseifner et al. [2009] it is not a trivial task to determine a model configuration of MPI and OpenMP for the best performance. However, we hypothesize that a hybrid version, combining the parallelism of both MPI and OpenMP, will yield the best performance results. We test our hypothesis in this report by testing the same sets of configuration on different versions of the stencil program. Section 2 gives an overview of the code and the different test that were performed. In Section 3, we present our results, which then are discussed in Section 4.

# 2 Methods

The code was run on the Piz Daint XC50 cluster maintained by CSCS. The machine itself consist of several nodes, each consisting of two Intel Haswell CPUS with 12 cores, resulting in 24 cores per node. The number of nodes and cores used varied depending on the test run (see below).

## 2.1 Code

The code is written in Fortran 90 and compiled with the Cray Fortran compiler. The field[x, y, z] has the dimensions nx = ny = 128, nz = 64 and diffusion is calculated over 1024 iterations. We tested three different versions of the code:

- MPI (stencil2d-mpi.F90):
  This is the reference version of the stencil program for our tests. The diffusion calculation is executed in the subroutine "apply_diffusion", which calls the laplacian 2D-advection subroutine twice to calculate the new field for the next iteration step. The standardized message passing interface MPI(3.1) is used to organize distributed memory parallelism (www.mpi-forum.org).

- Hybrid (stencil2d-mpiomp.F90)
  The hybrid version combines the distributed memory parallelism with shared memory parallelism using MPI and the application programming interface OpenMP (https://www.openmp.org), respectively. The stencil was modified for do-loop optimization such that the laplacian 2D-advection subroutine was integrated into the "apply_diffusion" subroutine, and the outer loop (k-loop) was parallelized and distributed on the different threads. This version was adapted further to allow us to test differnt OpenMP tuning parameters (i.e. loop scheduling), namely 'static', 'guided', and 'dynamic'.

- On-the-fly (OTF) (stencil2d-mpiomp-otf.F90):
  For the on-the-fly version, the hybrid version of the stencil program was further optimized by performing both steps of the 2D-advection in a single loop, and thereby obviating the allocation of temporal storage.

We validated our output fields for each version of the stencil program by comparing them to those from the reference (MPI-only) version, using a tolerance interval (up to 0.01% relative, but not more than 0.0001 absolute deviation from the original field) in order to account for some numerical distortion of the calculations,

## 2.2 Tests

In order to investigate and compare ideal settings for the different versions of our stencil program, and for comparability in our results, we performed four different sets of tests:

1. single node, upper limit:
   On a single node, we tested the code with 2, 4, 6, 8, 12, 16, 20, and 24 MPI ranks. For OpenMP-enabled versions, the number of threads is kept constant at one. In initial tests between the MPI-only and Hybrid versions of the stencil program, we attempted to run the code over 28 and 32 MPI ranks, but were restricted to 24, which is in line with the number of cores available per single node on the Piz Daint system, and signifies the upper-limit in terms of computing resources we can utilize on a single node.

2. single node, minimum:
   On a single node, we tested the code with 2, 3, 4, 6, 8, 12, and 24 MPI ranks. Meanwhile, in the OpenMP-enabled versions, we decreased the number of threads correspondingly, using 12, 8, 6, 4, 3, 2, and 1.

3. multi node, minimum:
   On four nodes we were able to test a wider range of MPI ranks starting at 4, and continuing to 8, 12, 16, 24, 32, 48, and 96. We also tested a decreasing number of threads for the OpenMP-enabled versions of the stencil program, using 24, 12, 8, 6, 4, 3, 2, and 1 threads.

4. strong scaling:
   Here, we increase the number of nodes from 1 to 12, inclusively, while simultaneously increasing the number of MPI ranks from 8 to 96 in intervals of eight. The number of ranks per node is constantly held at eight, and the number of OpenMP threads at three.

Each test was performed to compare the results of MPI-only version of the stencil program to the Hybrid and OTF versions. Various OpenMP tuning parameters (Section 2.1) using the Hybrid version were tested following the method for the single node minimum.

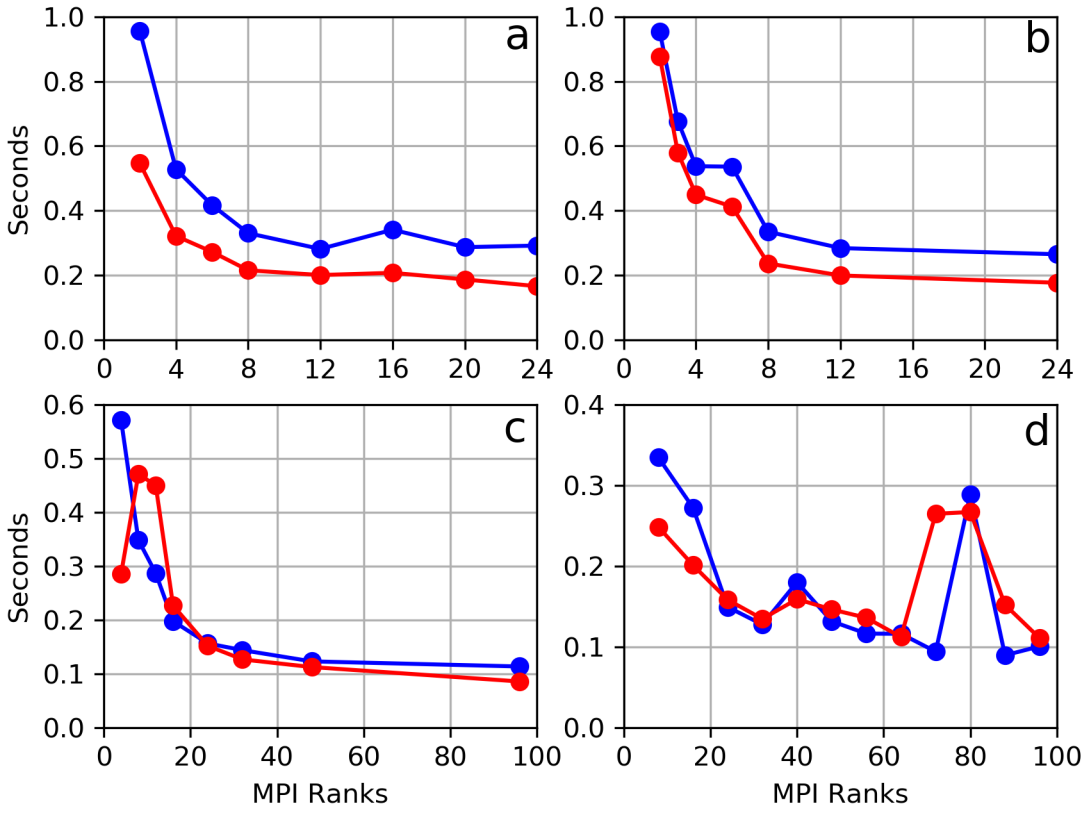


Figure 1: Results from the four benchmarking tests between the MPI-only (blue) and Hybrid (red) versions of the stencil program. The upper-limit test with a single node is shown in (a), the single node minimum is shown in (b), the multi-node minimum is shown in (c), and the strong-scaling test is shown in (d). The number of MPI ranks tested in each is the x-axis, while the y-axis is the job run-time in seconds.

### 2.3 Performance assessment

We use computation time as a simple proxy to compare the performance of the different test settings for our code, with less computing time being associated with a better performance of the system. While this metric is not directly related to computing resource efficiency, our approach seeks to highlight time-saving opportunities for higher-level users. To measure the time accurately, we make use of the MPI_Wtime function, which serves as a high resolution clock to measure elapsed time between two arbitrary points in the code. In our case, we implement the function at the start and the end of the stencil calculation.

## 3 Results

In the following subsections we present the results of the tests described in Section 2.2, using the MPI-only, Hybrid, and OTF versions of the stencil program. The sections include the benchmarking studies using the base version of both stencil program versions, OpenMP tuning of the hybrid version, and an OTF versus pre-computation comparative analysis in the hybrid version.

### 3.1 Flat MPI versus Hybrid

The first series of tests comprised a benchmarking exercise, focusing on finding and comparing the optimal settings for best performance of the MPI-only version and the Hybrid version. The results are shown in Figure 1a. It is immediately clear that the Hybrid version of the program is more efficient than the MPI-only version, as the computation time is less for all ranks. Both versions decrease their computation time asymptotically, indicating a theoretical upper-limit to the optimal number of MPI ranks needed for our programs and configuration. The largest relative increase in efficiency between the two programs occurs at two MPI ranks, whereas the most efficient version of the Hybrid program runs on 24 ranks.
Next, we compared the computational efficiency on a single node with an increasing number of MPI ranks (both MPI-only and the Hybrid) and a decreasing number of OpenMP threads (Hybrid version only, Figure 1b). Increasing the number of ranks increases the efficiency of both programs. However, the shared memory parallelism from OpenMP appears to have little to no efficiency effect as the pattern of the time decrease of the two programs with increasing number of ranks is nearly identical. It is difficult to isolate the cause of this behavior, but it may be linked to the do-loop optimization in the Hybrid version as it is more efficient for all MPI ranks.
We conducted the next test over four nodes. Similar to the single-node test, we increased the number of MPI ranks from 4 to 96, while decreasing the number of openMP threads from 24 to 1. In Figure 1c, the MPI-only version, as before, decreases computation time asymptotically. Unlike previously, the Hybrid version only decreases computation time after peaking at eight MPI ranks with 12 openMP threads. It utilizes more resources until 24 MPI ranks with four openMP threads. The relative efficiency remains smaller between the two versions than in previous tests when increasing the number of MPI ranks passed 24.
The final benchmarking study we name "strong-scaling" to compare performance between both versions of the stencil program with a wider range in the number of MPI ranks and nodes than in previous tests. We conducted several tests on 1 to 12 nodes, with increasing numbers of MPI ranks from 8 to 96. OpenMP threads were kept constant at three. Results are presented in Figure 1d. The asymptotic decrease with increasing ranks is not as evident like in previous studies. In fact, both programs show a dramatic increase in computation time around 72-80 MPI ranks. Additionally, the Hybrid version is not more efficient for higher numbers of ranks, and the difference between the two version remains small. This indicates the three OpenMP threads is not a suitable in terms of job efficiency.
Common throughout all benchmarking tests was unequal output between the MPI-only and Hybrid versions of the stencil program for specific configurations of MPI ranks, OpenMP threads, and nodes. We discuss this in more detail in the Discussion section below.

### 3.2 OpenMP tuning

In order to find the ideal OpenMP loop scheduling for our Hybrid stencil version, the configuration as in test 2 (Figure 1b) was run three times, using 'static', 'dynamic' and 'guided' scheduling with their respective default chunk-sizes.

This series of tests follows the single-node minimum series (Section 2.2), with eight different parameter settings to exclude the distributed memory effects on performance from our testing. To reiterate, the MPI ranks increase from 1 to 24 whereas the OpenMP threads decrease from 24 to 1. Results are presented in Figure 2. It should be noted, that the first parameter setting using one MPI rank could not compile, due to a floating point exception caused by the partitioner program (Section 4.1), which is why those values are not shown. Figure 2a shows the absolute performance of the Hybrid version of the stencil program with the different OpenMP scheduling. We observe a similar expected steady increase in performance with increasing MPI ranks using scheduling, as in the reference benchmarking test above (Figure 1b), however it is difficult to determine which OpenMP scheduling method performs best. Figure 2b shows the relative performance of the dynamic and guided settings as compared to the static scheduling, and therefore depicts the performance difference in a more precise manner. Dynamic scheduling stands out as having the largest positive difference in performance, meaning a longer run time compared to static scheduling, with four MPI ranks and six OpenMP threads. It does not stand out clearly in any other configuration, and does not appear to follow a positive or negative trend in terms of performance. Guided scheduling shows the opposite behavior than dynamic scheduling at four MPI ranks, with the largest negative difference. At six ranks the guided scheduling relative performance is positive, which continues to 24 ranks. The relative performances

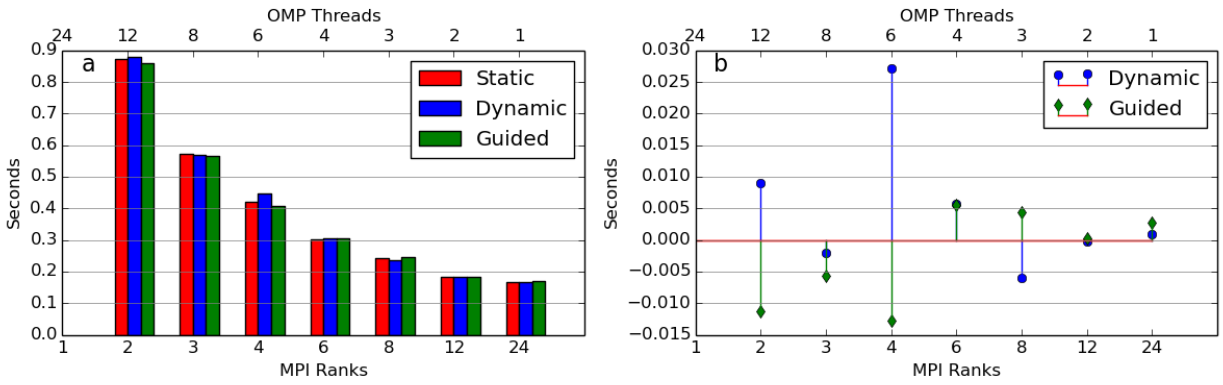


Figure 2: (a) Performances of the three different scheduling types applied to the second benchmarking test on a single node (Figure 1 (b) with increasing/decreasing MPI ranks/openMP threads. b) Shows the difference from the static scheduling performance for the static (red line), the dynamic (blue) and the guided (green) schedules for each run (bins in (a)).

of both types of OpenMP scheduling appear to converge towards static with a higher number of MPI ranks. With the exception of the run with four ranks, the addition of OpenMP scheduling appears to be negligible in terms of performance.

### 3.3 On-the-fly versus pre-computation

The final series of tests follow the first, except that we compared the results between the base Hybrid version and an adapted Hybrid version with "on-the-fly" (OTF) computation instead. Figures 3a-d show the result of the four tests. Here, pre-computation refers to the base Hybrid version of the program.
Results for the single-node upper-limit test for MPI ranks (Figure 3a) and the single-node minimum (increasing ranks from 1 to 24, and decreasing OpenMP threads from 24 to 1, Figure 3b) show similar asymptotic decreases in computation time with increasing number of ranks, similar to earlier tests. The Hybrid version outperforms the OTF version for all configurations. As the patterns of the two program versions are nearly identical, this implies that the slower run-time of the OTF version is due to the extra resources needed for the diffusion do-loop computation, and not necessarily the OpenMP parallelism.
Figure 3c presents the results of the multi-node minimum test. The performance of the two programs is similar to the base tests in Section 3.1, however the OTF version includes to run-time peaks at 8 and 24 MPI ranks. The Hybrid version is more efficient at all ranks than the OTF version except at 96 ranks where the two versions appear to converge. We encountered the same issue with unequal output using 24 ranks for this test as before; we discuss this issue in more detail in Section 4.
Finally, we repeated the strong-scaling test (Figure 3d). Instead of an obvious decrease in computation time, both programs appear to follow a bowl-type trend, with a decrease for mid-value ranks that increases again at a higher number. Both versions peak at 88 MPI ranks. Here, the OTF outperforms the Hybrid version, which continues to 96 ranks. This is the opposite behavior to the upper-limit and single-node minimum tests. Instead, with the number of OpenMP threads kept constant (three), the communication between an increasing number of ranks in the Hybrid version outweighs the pre-computation, adding to the total computation time.

## 4 Discussion and Conclusions

After analyzing our results, it becomes apparent that of the different versions of the code, the Hybrid version is best suited for the given architecture. On a single node, it consistently outperforms the MPI-only version as well the OTF version. However, our results also show that this is rather due to the diffusion do-loop optimization introduced in this version, rather than shared memory parallelism. This becomes apparent

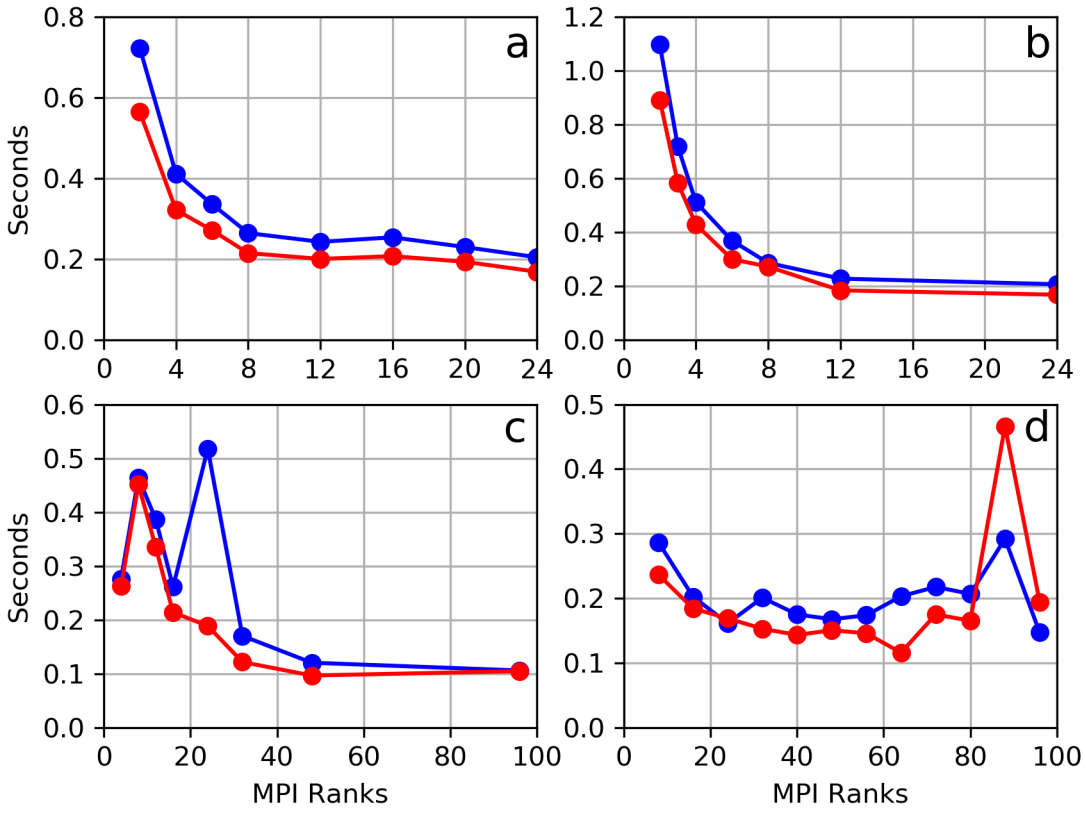


Figure 3: As in Figure 1, results from the four benchmarking tests for On-the-fly computation (blue) and the Hybrid (red) versions of the stencil program.

when comparing the shape of the two curves in Figures 1b and 3b. The introduction of OpenMP threads to the calculation has a very small impact on calculation time as compared to the additional MPI ranks, which continue to dominate the run time of the stencil. This is also true for the multi-node minimum test series (Figures 1c and 3c), with the exception of a peak at 12 threads for the Hybrid version and 4 threads for the OTF version. These sudden rises in computation time indicate that we utilized non-ideal configurations, which led to additional time due to synchronization overhead (e.g. communication between the nodes). As described in the course, our stencil example provides iteration chunks with approximately the same workload. Loop scheduling should therefore not have an impact on performance, except perhaps in a negative sense due to the communication overhead caused by dynamic or guided scheduling. Hence, changing the type of scheduling of the shared memory parallelism does not improve the benefit of introducing more threads, as the impact of the scheduling chosen is inconsistent and negligible compared to the impact of the number of MPI ranks (Figure 2). These findings contradict our hypothesized increase in performance upon the introduction of multiple threads, but it does not seem to be an uncommon phenomenon [e.g. Rabenseifner et al., 2009]. [Smith and Bull [2000] for example found that hybrid programming, while being a very effective tool to make some programs run more efficiently, does nothing to improve the performance for others, and can -on the contrary- decrease performance by adding to the synchronization overhead. On the other hand, Chorley and Walker [2010] found that the advantages of a hybrid MPI-OpenMP program are only evident when several nodes are used, while diminishing performance when running the same program with fewer nodes.
The performance increase due to loop optimization indicates that the omission of one of the temporary fields, and thus the reduction of data that must be read from and written to memory, makes our stencil work

more efficiently. However, further optimization of the loop, and the concomitant omission of and additional temporary field does not yield the same effect, and, in fact, increases the computation time rather than decreasing it.
We find the best performance (and thus shortest run time) to be on four nodes with the maximum amount of ranks when running the Hybrid version (Figure 1c). The same number of ranks run on more nodes, however, does not yield the same results. These configurations never reach run time values as low, and we find a sudden increase in run time around 10 nodes (Figures 1d and 3d). The decrease at 12 nodes proves that this is not the definitive tipping point at which the overhead outweighs the performance increase. Instead, we conclude that our observed behavior follows Chorley and Walker [2010], where the advantage of using a hybrid program is only evident with a higher number of nodes. Further investigation of the program behavior on more nodes and different OpenMP settings could provide more insight and would be crucial for a conclusive result.

## 4.1 Limitations

Our performance estimation in this report is limited to run-time only. In the future, one should consider performance in terms of intensity of the individual jobs by comparing the allocated resources via a FLOPS or bytes counter, like in the Roofline model example from this course.This would allow for a more in-depth analysis which would help guide the way to further improvement of the code.
For several configurations, we encountered unequal output between the various versions of the stencil program, and between the MPI-only version using different configurations. We attribute this to invalid variable truncation (i.e. a round-off error) due to the division of the job between ranks, which propagate during the iteration. A heatmap of the differences between the unequal outputs (not shown) showed a random and evenly distributed pattern, rather than a systematic error. An increase of the initial tolerance range in the compare_fields script (from $10^{-8}$ to $10^{-4}$) resolved that issue. Deeper analysis of the origin of these errors would not only be interesting, but also necessary to either reduce them or find a reasonable range of tolerance.
We were limited to running our code with more than one MPI rank due to a floating point exception error (i.e. a divide by zero arithmetic error) originating in the partitioner program, which is responsible for dividing the stencil program according the specified number of ranks. Due to time limitations, we were not able to resolve this issue. However, as we were focused on comparing the performance between OpenMP and MPI, running the code with a single rank would not yield a valid version of the MPI-only version of the stencil program for our analysis.
Finally, we also found that for different runs of the same code with the same settings, the run-times displayed different values. As this offset stayed well below the differences between the different settings in the majority of our simulations, we are confident that this has no impact on our findings. However, for a more scientifically robust analysis, one should record the results of multiple runs per configuration in each test, and then find the standard deviation of the dataset to provide a more in-depth analysis of uncertainty.

# References


Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

Ming Xue. High-Order Monotonic Numerical Diffusion and Smoothing. *Monthly Weather Review*, 128(8): 2853–2864, 08 2000. ISSN 0027-0644. doi: 10.1175/1520-0493(2000)128⟨2853:HOMNDA⟩2.0.CO;2. URL `https://doi.org/10.1175/1520-0493(2000)128<2853:HOMNDA>2.0.CO;2`.

R. Rabenseifner, G. Hager, and G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 427–436, 2009.

Lorna Smith and Mark Bull. Development of mixed mode mpi/openmp applications. *Scientific Programming*, 9, 08 2000. doi: 10.1155/2001/450503.

Martin J Chorley and David W Walker. Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters. *Journal of Computational Science*, 1(3):168–174, 2010. ISSN 1877-7503. doi: https://doi.org/10.1016/j.jocs.2010.05.001. URL `http://www.sciencedirect.com/science/article/pii/S1877750310000396`.