D USYS

IAC Institute for Atmospheric and Climate Science

High Performance Computing for Weather and Climate

Project Report - FS 2024

Communication Strategies

Andri Heeb, 20-920-294, anheeb@student.ethz.ch

Angelika Koch, 20-924-494, ankoch@student.ethz.ch

Tim Zimmermann, 20-933-859, tzimmermann@student.ethz.ch

Advisor:

Simon Adamov

August 30, 2024

Contents

1 Introduction & Theoretical Background

In this project, we implement different communication strategies to improve the computation time of the halo update in a 4th-order non-monotonic diffusion model.
We are focusing on the blocking and non-blocking strategies sendrecv, Isend - Recv, Send - Irecv, and Isend - Irecv. The experiments are conducted in a Jupyter notebook using Python as the main programming language with mpi4py as a Python interface to MPI.
With the above-mentioned strategies, we will focus on the following questions:

- Which communication strategy is the most time-efficient?
- At what amount of ranks does the communication effort overcome the computational effort?
- How does the amount of halo points influence the computation time?
- How does the computation time change with different domain sizes?
- For what conditions does overlapping communication improve the computation time?
- How does the performance change when running the program on multiple nodes?

The changes in communication strategy, amount of ranks, halo points, domain size, and number of nodes are tested separately, always starting with the computation on one rank using the respective communication strategy.

1.1 Message Passing Interface (MPI)

High-performance computing (HPC) aggregates computational power and organizes it in a way to get the most performance out of it.
MPI is a message-passing standard which is implemented as a library that provides the necessary semantics. There are many implementations of MPI, available for almost any architecture. In MPI, the number of tasks sent to the cores is distributed evenly in groups, which are called the workers. Groups of workers are called communicators. Each MPI command requires a communicator, and the command will only affect the processes within that communicator. The default communicator, known as COMM_WORLD, includes all available processes. The MPI Application Program Interface (API) has to realize the exchange of messages between the workers, as they cannot access each other's variables. In the event that multiple nodes are employed, the transfer of data is guaranteed through the utilization of a network cable using a message-passing protocol.

Working on just one node of the Piz Daint supercomputer, we can use a total of 24 cores, comprising 12 physical and 12 virtual cores. In order to optimize efficiency, all cores must be engaged in the computation of a portion of the code, with minimal inactive waiting time. However, sometimes some computations cannot be run in parallel because they depend on each other, such as writing data to disk or plotting the results of a computation. To prevent misconduct, a master rank is defined, which is responsible for executing the sequential regions of the code. In general, rank 0 is the master rank because rank 0 always exists, even for a single worker.

According to Fuhrer (2024a), a modern HPC system has a latency of 0.5-1.0 μs and bandwidths are 0.1-10 GB/s for MPI communication. In our project, the latency increases due to the creation of overhead in Python.

1.2 Diffusion equation

To control small-scale noise, atmospheric and ocean models sometimes need numerical filtering, i.e. by higher-order monotonic filters. In this project, we use a higher-order diffusion from Xue (2000):

$$\frac{\partial \phi}{\partial t} = S + (-1)^{n/2+1} \alpha_n \nabla^n \phi \tag{1}$$

The following simplifications are implemented: ignore other processes (S = 0), 4^{th}-order (n = 4), and without limiter. This results in the 4^{th}-order non-monotonic diffusion equation:

$$\frac{\partial \phi}{\partial t} = -\alpha_4 \Delta_h (\Delta_h \phi) \tag{2}$$

To bring the equation in code form, we use 2^{nd}-order centered differences for the spatial discretization and the 1^{st}-order forward step (Euler step) for the discretization in time.
The combination of these discretizations results in a cross-shaped stencil.

1.3 Stencil computation

A stencil computation is a numerical data processing technique, more precisely an algorithmic motif or a fixed pattern. It is used to update elements of an array at a certain grid point, i.e. by computing the value from a compact neighbourhood of grid points in its vicinity on the computational grid using the same pattern for every grid point (Fuhrer, 2024b).
Stencils are commonly used in computational fluid dynamics, image processing, and solving partial differential equations (PDEs). Thus, stencils are especially favoured for GPUs. The utilization of stencils within codes comprising numerous repetitive elements has the potential to considerably enhance the code's execution time.

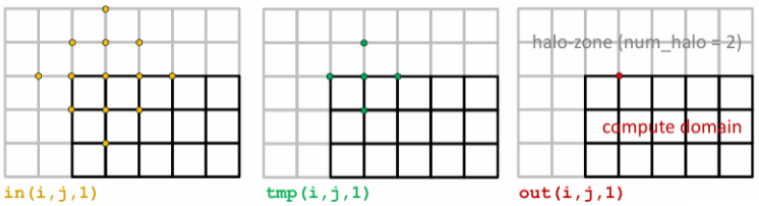

Fig. 1. Stencil used in the computation of the halo update. The grey lines depict the halo zone (here two halo points) while the black grid shows the computed domain. The coloured points show the shape of the stencil. Figure from Fuhrer (2024b).

1.4 Halo update

Border cells present a challenge for the utilization of stencils. If the objective is to apply the stencil to the outermost cells, the stencil would require neighbouring cells which do not exist. One potential solution is to extend the array with additional elements, which are referred to as halo points or ghost cells. It is essential to consider the content of these cells in great detail. One method is displayed in Fig. 2, where the array on the left side of the original data set acts as the halo line of the right side and vice versa and the bottom line acts as the halo line of the top and vice versa. The corners have to

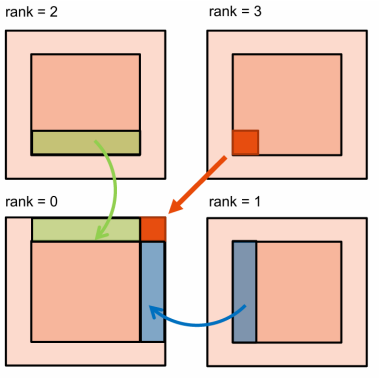

Fig. 2. Halo update including appropriate treatment of the corners.

be treated separately to avoid miscalculations. Here, we use the no-sync strategy, where the bottom-left corner is used as the top-right halo point, the bottom-right corner is used as the top-left halo point, and so on.

1.5 Validation

It is essential to validate the results of the communication strategy to guarantee its correct functioning. This is achieved by plotting the output field after each model run. When executed correctly, the output field should look like the example shown in Fig. 3.

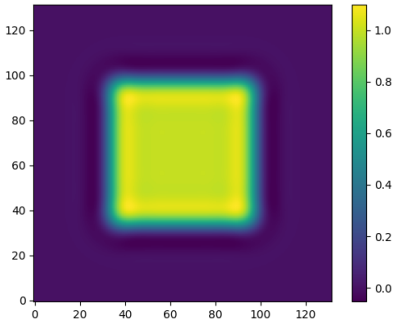

Fig. 3. Output field after the halo update and the applied diffusion, plotted for visual validation.

2 Methods

2.1 Blocking vs non-blocking communication strategies

To improve the computation time for the halo update, we implement several communication strategies in the stencil computation, including both blocking and non-blocking strategies. In a general sense, blocking communication stops or blocks the code from progressing until the communication is completed. If this communication request does not occur, a deadlock will occur and the script will not proceed.

Sendrecv Since Send and Recv are often used together, there is a built-in option Sendrecv() which executes both at the same time and thus avoids the risk of a deadlock.

Isend/Irecv The utilisation of the Isend and Irecv functions is a viable solution to prevent a deadlock. The deployment of a non-blocking send and a non-blocking receive averts the risk of a deadlock by a false timed send or receive order.

Isend/Recv This communication strategy has a non-blocking send and a blocking receive. This leads to the possibility of posting multiple sends before the first receive. However, when a receive is posted, the corresponding send must already be posted as the receive is still blocking.

Send/Irecv This communication strategy has a blocking send and a non-blocking receive. This is essentially the opposite of the Isend/Recv described above. Unfortunately, this strategy only works on a specific number of ranks (1, 2, 3, 4, and 8). The other ranks result in a segmentation error which has to do with memory access and buffer overflow. Accordingly, this approach will not be pursued further.

evenodd By splitting the work into two separate groups, we are able to prevent deadlocks even with blocking communication strategies. This is achieved by having all odd ranks only send and even ranks only receive in the first round. This is then repeated and the even ranks send and the odd ranks receive. If this process is implemented correctly, it could result in a highly time-efficient strategy. Unfortunately, our version always ends in a deadlock, due to non-matching prior receive. Accordingly, this approach will not be pursued further.

2.2 Send/Recv vs send/recv

The majority of MPI methods comprise two versions, one written with a capital letter and one written with a lowercase letter in front. The primary distinction between the two versions is that the one with a lowercase letter can be utilized for any Python object, but it creates an overhead. This is not the optimal approach for HPC, which is why the use of Send/Recv is typically preferred. The uppercase version requires the size of the object in bytes, which is more efficient due to reduced overhead but carries a higher risk of errors.

2.3 Overlapping computation

Performance can significantly be increased by overlapping communication and computation. This approach allows us to start a new calculation without waiting for the previous communication to complete. The new calculation does not depend

on the prior communication, enabling us to utilise resources more efficiently. The calculation will continue to run while the communication is still in progress. In order to achieve this, it is essential to utilise non-blocking communication. MPI also provides the statement req.wait(), which can be used at a point in the script where all communication must be completed before continuing.
To implement the overlapping strategy, the computation of the Laplacian field is overlapped with communication. To achieve this, we modify our program to execute the Laplacian calculation between the MPI.Isomething() and req.wait() commands.

2.4 comm.Barrier() vs comm.wait()

Both of these statements are used to synchronize communication when necessary. The comm.wait() is used to make all ranks wait on a specific non-blocking communication. It does not synchronize all processes, it synchronizes only the request associated with the communication. The Comm.Barrier() function is used to ensure that all communication processes are synchronised. This guarantees that all processes conclude communication before proceeding.

2.5 Multiple halo points

An increase in the number of halo points improves the precision of the output, but this is a more computationally demanding process, resulting in a longer halo update time.
The default value for num_halo is set to 2. To test for computation time changes for a larger amount of halo points, we increase num_halo to 8. The results can be seen in Fig. 4.

2.6 Domain size

To elaborate on whether the computation time gain is larger than the overhead, we increase the domain size (nx, ny). Given that an increased domain size is expected to lead to longer computation times, one of the fastest and most robust stencil computation versions (Isend_Irecv_2hp) is used. Please note that the number of grid points must fall within the range of 0 to 1024×1024. This is provided by the template from the course (Fuhrer, 2024a). In this project, we implement nx = ny $\in$ [64, 128, 256, 512, 1024] in the non-blocking communication strategy with two halo points for a number of ranks of 1-24.

2.7 Multiple nodes

To assess the potential for performance improvement across multiple nodes, tests are conducted with an equal number of ranks on one, two, and four nodes. Given that the maximum number of ranks is contingent on the number of nodes, we have also set up model runs for the maximum number of ranks feasible for each test.

3 Results & Discussion

3.1 Computation strategies with differing number of ranks and halo points

To answer the question of which communication strategy is the most time-efficient, the different communication strategies mentioned in section 2.1 are tested with a varying number of ranks, starting at one and ending with the maximal possible number of ranks (1 node = 24 ranks). Additionally, the influence of the number of halo points on the computation time is implemented.

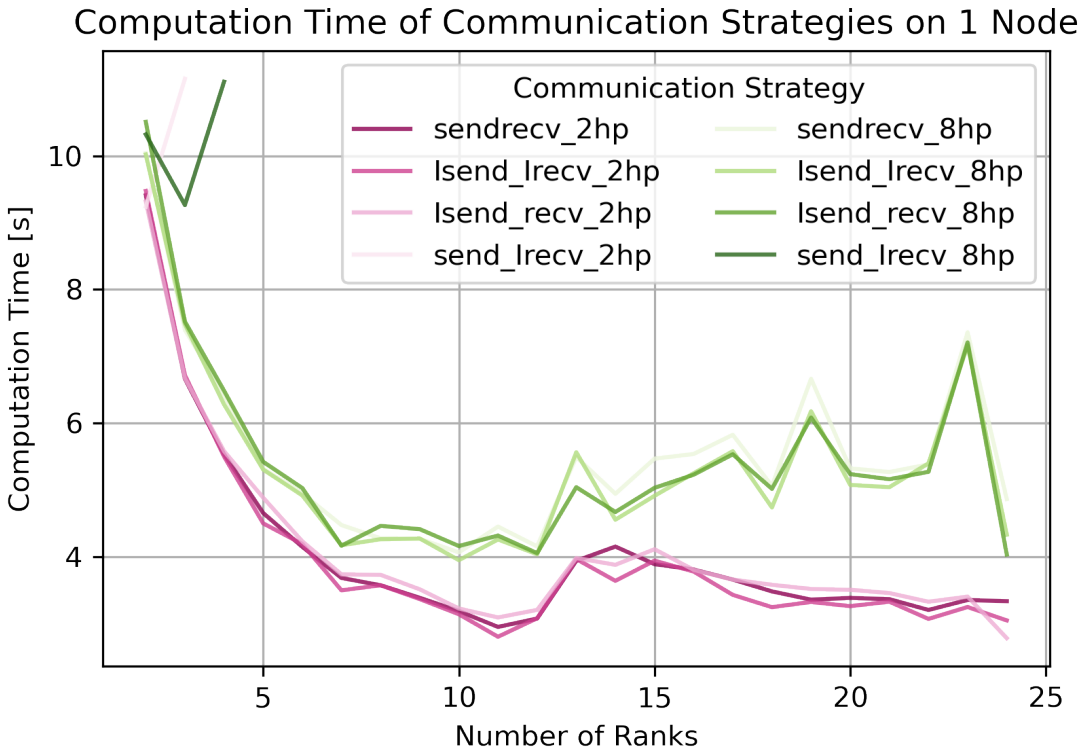

Fig. 4. Computation time of different communication strategies with a different amount of ranks used. The green shades show the computation time if eight halo lines are implemented, while the pink shades represent the computation time with two halo lines. In all the strategies, the domain size nx = ny = 128, nz = 64, and 1024 iterations was calculated.

Fig. 4 shows a decreasing computation time for all strategies except for the Send/Irecv. As mentioned in section 2.1, this strategy does not work appropriately for higher numbers of ranks. Therefore, only the other three strategies will be pursued.

The communication strategies have very similar computation times when implementing the same amount of halo points. This means that the question of the time efficiency of the communication strategy cannot fully be answered, as the computation time depends more on the number of ranks than on the implemented communication strategy. The overall fastest computation time is achieved with Isend/Recv (two-halo-points version) for 24 ranks (2.7813 s), closely followed by Isend/Irecv for 12 ranks (2.802 s).

The two-halo-points (2hp) version consistently exhibits a lower computation time than the eight-halo-points (8hp) version. However, there is a notable decline in efficiency shortly after 12 ranks for both versions. The 8hp version has a higher

computational cost than the 2hp version because the calculation of more halo points means more computational effort, which is reflected in the computation time.

3.2 Computation time on multiple nodes

Since we are working on a supercomputer, the communication strategies are tested on two and four nodes to simulate running the program on a whole cluster. In addition to the three strategies shown in Fig. 4, we also implement the overlapping communication strategy introduced in section 2.3 on multiple nodes.

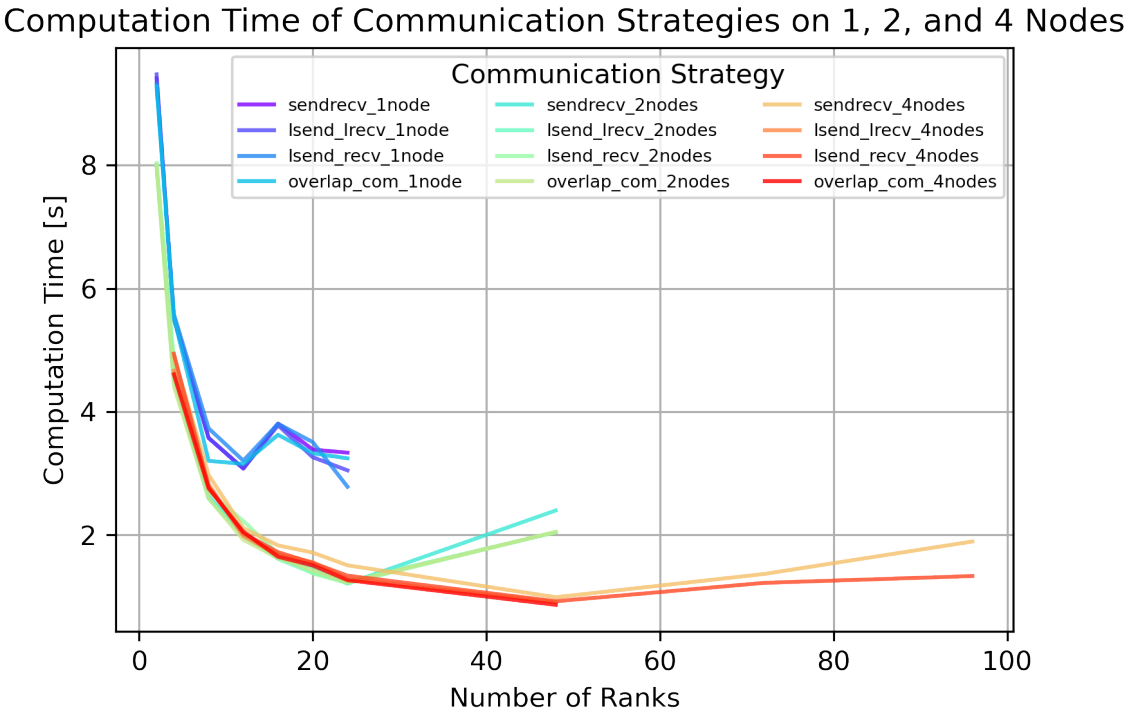

Fig. 5. Computation time of different communication strategies with different number of nodes and ranks. The blue shades show the communication strategies run on one node (as in Fig. 4). The green shades depict the strategies run on two nodes, while the orange lines represent the computation on four nodes. In all the strategies, the two-halo-points version with nx = ny = 128, nz = 64, and 1024 iterations was calculated.

In Fig. 5, the computation time for the different number of nodes can be differentiated by colour. As one node provides 24 ranks, the blue shades have a maximum of 24, the greens of 48, and the orange shades of 96 ranks. It can be seen that more nodes and more ranks generally lead to faster computation. The behaviour of the strategies on one node is discussed in section 3.1. The computation time for the strategies on two nodes strongly decreases until 24 ranks and increases again for 48 ranks. The minimal computation time on two nodes is reached with the strategy sendrecv (1.2078 s). The test with four nodes shows a similar pattern, where the computation time reaches a minimum of 0.8641 s for 48 ranks for the overlapping communication strategy and increases again for a higher number of ranks. Interestingly, for each single/multiple node version, the minimal computation time is reached at half of the available ranks respectively. This may relate to the number of physical and virtual cores.

A comparison of the results from the multiple nodes indicates that the communication effort for multiple nodes increases at a slower rate than the computational effort decreases, meaning that on average, the four-node version has the lowest

computational time out of the three. As the computation gain from two nodes to four nodes is already not that big, we assume that using even more nodes would not lead to a significant decrease in the computation time any more.

Looking at the computation time of the overlapping communication on one node, there is almost no difference from the original Isend/Irecv. For some runs, it is even slightly slower than the original version. A reason might be that the code already is optimal and letting the laplacian run in parallel does not improve the computation time any more. Increasing the number of nodes leads to a slight improvement in the overlapping communication relative to the other strategies. This decrease in communication time is vanishingly small.

3.3 Computation time for increasing domain size

Increasing the domain sizes in x- and y-direction (nx and ny) leads to a drastic increase in the computational effort. As mentioned in section 3.2, the overlapping communication strategy could not be proved more efficient than its non-blocking counterpart. Thus, we want to find out, whether the overlapping communication strategy turns out to lead to an improvement of the computation time if the computational effort increases.

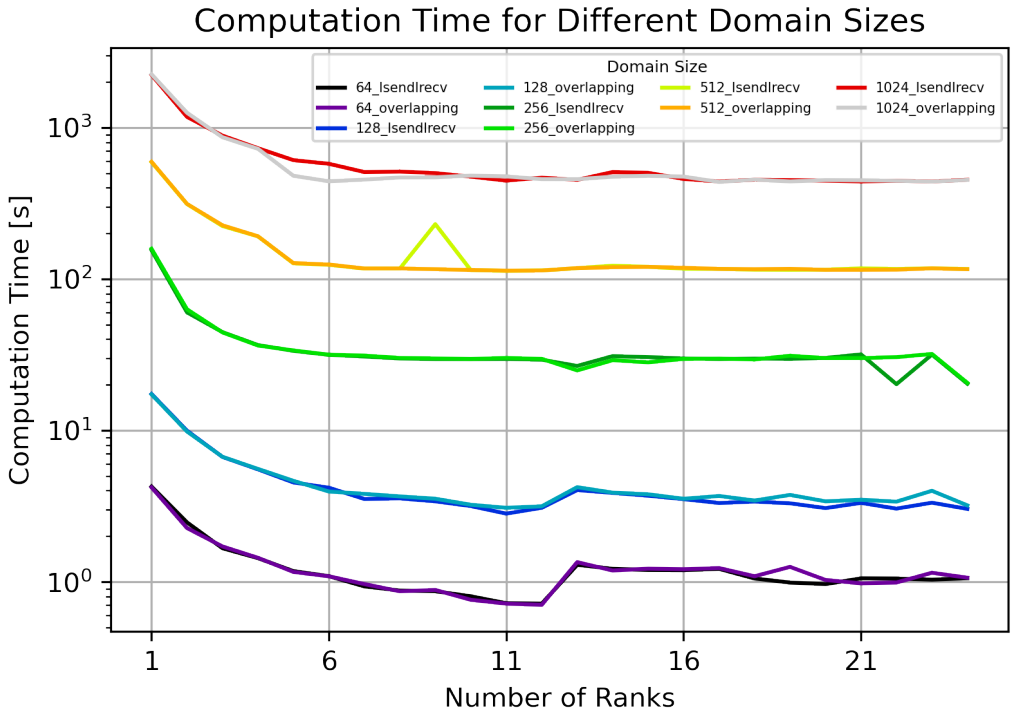

Fig. 6. Computation time of the communication strategy IsendIrecv and overlapping communication with a different number of ranks. The colour scheme represents the domain size applied for each of the two communication strategies (DomainSize_CommunicationStrategy). In all the computations, the two-halo-point versions with nz = 64 and 1024 iterations were used. The y-axis is on a log-scale.

Fig. 6 shows the computation time on a logarithmic scale for the double-non-blocking and the overlapping strategy for five different domain sizes: nx = ny $\in$ [64, 128, 256, 512, 1024]. As expected, the computation time increases non-linearly with increasing domain size, while slightly decreasing for increasing number of ranks. The increase in computation time after rank 12 does not occur for the domain size 1024.

The two analyzed strategies do not considerably differ within the same domain size. To analyze the difference in efficiency between the double-non-blocking and the overlapping strategy, the difference in computation time is illustrated in Fig. 7.

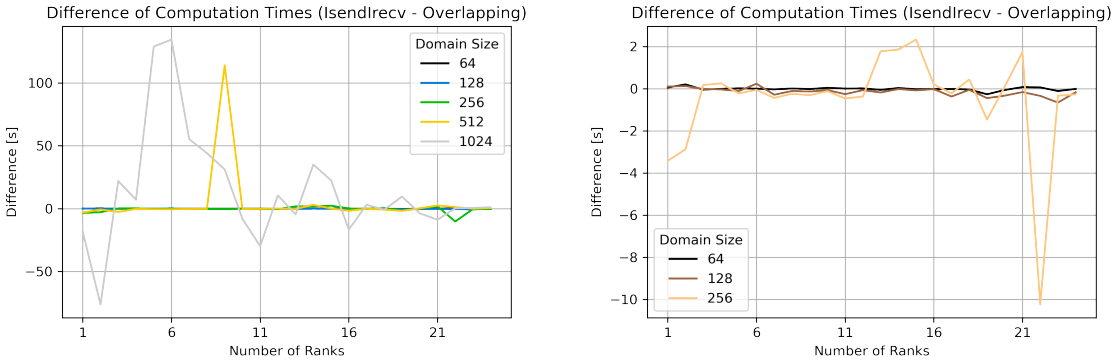

Fig. 7. Difference of computation time of the communication strategy IsendIrecv and overlapping communication for different domain sizes and ranks. In all computations, the two-halo-points version with nz = 64 and 1024 iterations were used.
Left: domain sizes 64, 128, 256, 512, and 1024. Right: domain sizes 64, 128, and 256. It is the same calculation as on the left but zooms in to make the differences within the smaller domain sizes visible.

In Fig. 7, it can be seen that the IsendIrecv has a considerably higher computation time for nx = ny = 1024 and five to seven ranks. For this domain size, the difference in computation time changes between positive (overlapping more efficient than IsendIrecv) and negative (IsendIrecv more efficient than overlapping) values quite frequently with the number of ranks. Further, the computation time for the overlapping strategy is more than 100 s faster than the IsendIrecv strategy for the domain size of 512 for eight ranks. As this is the only peak for the 512 domain size, it might also be regarded as an outlier. To look at the difference in computation time for the smaller domain sizes, the large domain sizes have to be left out. The domain size nx = ny = 256 shows some variation between positive and negative values, too. For 21 ranks, the IsendIrecv is 10 s faster than the overlapping version.

Finally, the overlapping strategy achieves some improvement of computation time for large domain sizes, but only for a specific amount of ranks. In some cases, the overlapping communication also performs worse than the IsendIrecv strategy. On average, the two strategies are similarly efficient, except for the largest domain size, where the overlapping strategy represents a clear improvement.

4 Conclusion

We present four different communication strategies with overlapping computation for one strategy. Results show that if we increase the number of nodes and ranks for the computation, we are rewarded with better performance. However, one has to consider that an optimum is often reached at the maximum available number of physical cores. Overlapping computation and communication benefits most when the work is distributed across different nodes, but shows no real benefit for working on one node. Also, for larger domain sizes, overlapping computation and communication shows promising results. The non-blocking method outperforms the blocking methods in most of our experiments, indicating its reliability for higher performance. Furthermore, our findings indicate a non-linear relationship between domain size and computation time, which presents a challenge when attempting to achieve higher resolution. Finally, the even/odd strategy needs further investigation to see why it does not work at higher rank sizes.

4.1 Outlook

Further strategies such as even/odd communication could be tested to see if they perform even better than the ones examined in this project. Furthermore, the scripts could be converted from Python to FORTRAN to reduce the overhead and possibly result in a reduction in computation time. The gain in efficiency would have to be to such an extent that it outweighs the increase in incomprehensibility of the FORTRAN code version. Additionally, the extent of performance gain for overlapping computation and communication could be quantitatively estimated (i.e. by implementing computationally demanding tasks). Finally, the utilization of MPI-datatypes for packing/unpacking into/from message buffers and their influence on the computation time could be investigated.

References

Oliver Fuhrer. 02-mpi-introduction.ipynb, 2024a. https://github.com/ofuhrer/HPC4WC/blob/164fc1c00bf3dbf4b2ae1c7a18e33d0088bd75e2/day3/02-MPI-introduction.ipynb. Course HPC4WC June 2024.

Oliver Fuhrer. Slides02-intro-and-stencils.pdf, 2024b. https://github.com/ofuhrer/HPC4WC/blob/164fc1c00bf3dbf4b2ae1c7a18e33d0088bd75e2/day1/02-stencil-program.ipynb. Course HPC4WC June 2024.

Ming Xue. High-order monotonic numerical diffusion and smoothing. *Monthly Weather Review*, 128(8):2853 – 2864, 2000. doi: 10.1175/1520-0493(2000)128⟨2853:HOMNDA⟩2.0.CO;2. URL https://journals.ametsoc.org/view/journals/mwre/128/8/1520-0493_2000_128_2853_homnda_2.0.co_2.xml.

Appendix

List of abbreviations

sendrecv	blocking sendreceive	
Send/Recv	blocking send and	blocking receive
Isend/Recv	non-blocking send and	blocking receive
Send/Irecv	blocking send and	non-blocking receive
Isend/Irecv	non-blocking send and	non-blocking receive
High Performance Computing	HPC	
2hp	two halo points	
8hp	eight halo points	

AI use statement

For this report, we used AI for translating some sentences from German to English by using Grammarly, DeepL, and DeepL Write. Furthermore, ChatGPT was used whenever we required assistance with the code.