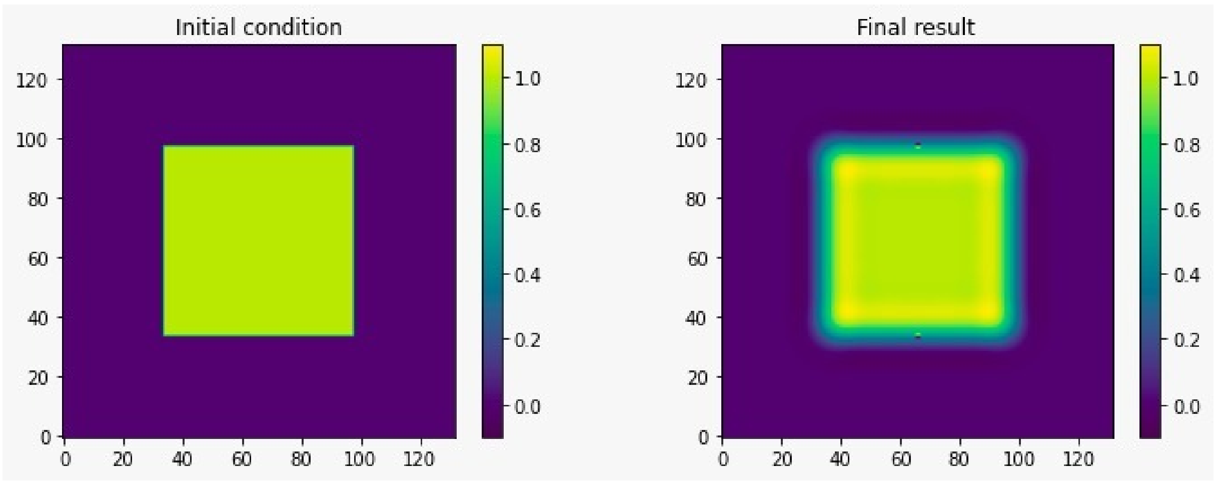
High Performance Computing for Weather and Climate, FS 2022

Work Project:

Communication strategies for halo updates

Submitted by:

Victoria Bauer, Sarina Danioth & Julia Dworzak



Zurich / August 14, 2022

Summary

The purpose of this study, conducted as part of the lecture "High Performance Computing for Weather and Climate" at ETH Zurich in the spring semester of 2022, is the implementation and performance analysis of different communication strategies using the standardised Message Passing Interface (MPI). The communication strategies were implemented into the halo update of a stencil 2D program simulating higher order diffusion. Three different experiments were conducted measuring the time needed for communication within the halo update of the different versions. Regarding the scalability of the problem, we found that a certain trade-off between decreasing the computational workload per worker and increasing the overall communication load in the program exists. While a parallel computing method certainly outperforms the serial computation of the stencil, we could not identify the optimal MPI communication strategy as all perform similarly well. Thus, choosing a communication strategy depends on individual needs concerning the problem and personal preferences.

Contents

1 Introduction

In this work project, which is part of the lecture High Performance Computing for Weather and Climate at ETH Zurich (spring semester 2022), various different communication strategies for performing halo updates with the standardised Message Passing Interface (MPI) in a Stencil 2D program are implemented. These are further investigated through a performance analysis. In the following theoretical introduction, the MPI itself, as well as some communication basics and the herein used Stencil 2D program are introduced. Subsequently, in section 2, the implemented communication strategies are explained followed by an introduction of the conducted performance analysis experiments. Thereafter, the results are presented and discussed (section 3), followed by a conclusion (section 4).

1.1 MPI

The MPI is a programming interface which has a message-passing library interface specification (Lusk et al., 2009; Nielsen, 2016). The interface allows passing messages between multiple distributed memory spaces we here call ranks (Lusk et al., 2009). There are several advantages of using a MPI, with the portability and the usability being especially prominent. MPI is crucial for building parallel programming models as it enables exchanging data by sending and receiving messages between parallel computing units (Lusk et al., 2009; Nielsen, 2016). As the MPI is only a specification, there are numerous implementations. All MPI operations are functions, subroutines or methods according to the corresponding language bindings. For example in Fortran, the MPI operations are part of the MPI standard (Lusk et al., 2009).

1.2 Communication Basics

The key feature of MPI is sending and receiving messages between the ranks running in parallel. As the different ranks have different memory addresses and thus, cannot access each others variables, data exchange is crucial. The most basic communication mechanism is the point-to-point communication with the operations *send* and *receive*. In the senders memory a *send buffer* is defined containing the data of the message. The *send* operation is specified with information about the location, the size and the type of the send buffer. Moreover, it also contains parameters specifying the message destination. Furthermore, there is a *receive* operation where the particular message to be received is specified. This operation contains parameters about the location, size and type of receive buffer and a parameter for returning the information of the message just received. In order to distinguish messages and receive them selectively, fields with information about the message are defined. Those fields, called message envelopes, can be the message source (determined by the identity of the message sender), the destination, an integer-valued message tag or a communicator which defines the communication context. Tagging the messages can help to avoid errors when there are numerous data exchanges taking place at the same time.

An example for an error is the deadlock situation, in which the rank is waiting for an MPI operation to complete which is not happening due to an error in the program logic. Thus, the synchronisation between complete and wait is not symmetric. The introduced send and receive point-to-point communication methods are blocking methods, meaning they do not return until the communication is executed. To avoid such deadlock situations, non-blocking communication methods are helpful, in which the procedure may return before the operation is completed and before it is allowed to reuse resources (i.e. buffers). The non-blocking operation calls are indicated with a prefix *I*, which stands for immediate. In a non-blocking communication method the send operation is split into a *send start* and a *send complete* operation. This allows the initiation of the send operation without completing it and thus, the return

before the message was copied out of the send buffer. The completion will only take place after the *send* complete operation is called. The completion is needed to verify that the data was copied out of the send buffer. Similarly, the receive operation is divided into a separate *receive start* and a *receive complete* operation call. The starting operation is needed to initiate the receive operation before the message is stored into the receive buffer. Then, the completion operation is needed for the verification that the data was indeed received into the receive buffer.
When using non-blocking operations the *MPI_Waitall* function is crucial to defer the synchronisation, as the *MPI_Waitall* forces the process to wait until all requests have been completed.

1.3 Stencil 2D

In this project we analyse the performance of different MPI communication strategies in the halo update subroutine of the *stencil2d-orig.F90* program by Fuhrer (2020b). Stencil programs allow efficient implementations of numerical algorithms on a fixed grid. In general, in a stencil computation a value at a certain grid point is determined by the surrounding neighborhood grid point values. The computation of the value at a grid point is always performed by using the same pattern for each grid point. These fixed patterns, which are independent of the grid point itself, are called stencils (Sloot et al., 2002). An example for which a stencil-based program is useful, is the diffusion equation consisting of finite differences, finite volumes or finite element discretisations on a grid.
The herein used stencil program simulates the fourth order monotonic diffusion using periodic boundaries at the edge of the field, following Xue (2000). The computational domain is divided onto ranks such that the partition happens only in the horizontal.

Halo update
Halo cells or ghost cells, are used to locally replicate the domain residing in remote memories. They rely on buffers to mirror remote data and on communication methods to update their state. Thus, data transfers are needed to provide access to the remote data through an halo exchange. Within the halo exchange each processor finds the grid points adjacent to its domain and replicates the data to its own memory. Figure 1 shows how the halo update subroutine can work. In a first step, a top-bottom exchange without the edges is executed between the ranks 3 and 1. In a second step, a left-right exchange between rank 1 and rank 0 is performed, in which also the cells on the edges are transferred. Thus, at the end all the ghost-cells are updated.

In this work, we make use of the module *m_partitioner* by Sudwoj et al. (2020) which takes care of many functionalities facilitating the MPI communication for stencil programs. This includes dividing the horizontal computation domain into junks and assigning them to the ranks and ready to use functions to access left, right, top and bottom neighbors of a rank.

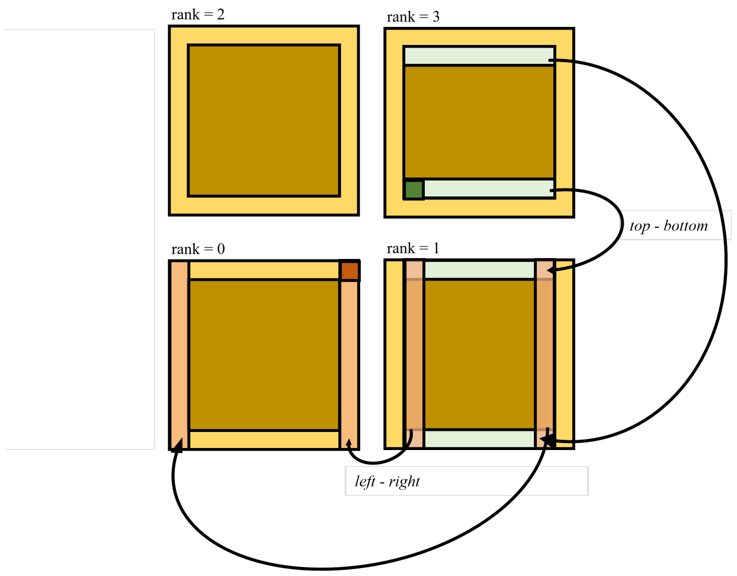


Figure 1: Schematic of the halo update subroutine. Figure adapted from lecture 5 (MPI).

2 Methods

2.1 Implementation of new strategies

To compare the performance of different communication strategies, we implement five versions of the halo update each using a different MPI communication in Fortran. We base these on the original, non-parallelised version of the 2D stencil in *stencil2d-orig.F90* and a parallelised version *stencil2d-mpi.F90* which uses an *IrecvIsend* strategy by Fuhrer (2020a). The newly implemented versions are listed below:

- **stencil2d-IrecvSend.F90** for IrecvSend
- **stencil2d-IsendRecv.F90** for IsendRecv
- **stencil2d-IrecvIsend.F90** for IrecvIsend
- **stencil2d-sendrecv.F90** for sendrecv
- **stencil2d-evenodd.F90** for Send Recv on even odd

In the following subsections each of the communication strategies and their implementations are explained in more detail.

2.1.1 Irecvsend

stencil2d-IrecvSend.F90
This communication strategy uses the non-blocking function *MPI_Irecv* to pre-post the receive statement, before sending the buffers with the blocking function *MPI_Send*. The strategy begins with exchanging the top-bottom halo points without the corners of the halo zone. After all blocking receives have been finished, the left-right halo points are exchanged including the corner points received from the top-bottom communication.

2.1.2 Isendrecv

stencil2d-IsendRecv.F90
In contrast to the strategy above, here the send function is non-blocking (*MPI_Isend*) while the receive

function is blocking (*MPI_Recv*). In this case the sends are pre-posted, followed by the blocking receives, ensuring the communication finishes before unpacking the respective buffers for top-bottom and left-right.

2.1.3 IrecvIsend

stencil2d-IsendIrecv.F90
Here non-blocking send and receive functions are used. The communication starts with the top-bottom receive function *MPI_IRecv* followed by the send function *MPI_ISend*. To ensure that all top-bottom halo points have been exchanged before continuing with the left-right communication, we call the function *MPI_Waitall*. The call to this function returns, when all operations associated with active requests, as in the non-blocking send and receive functions, are completed. Similarly, this function is called after the left-right communication calls are posed and before unpacking the received left-right halo points into the output field. This version uses the same communication strategy as *stencil2d-mpi.F90*, however, it is implemented in a slightly different way, as the mpi version preposts all non-blocking receives for left-right as well as top-bottom at the beginning of the halo update, while the new version puts the receives with the corresponding sends, such that the second receive is separated from the first by one *MPI_Waitall*.

2.1.4 sendrecv

stencil2d-sendrecv.F90
This strategy uses the function *MPI_Sendrecv*, which combines a blocking send and receive operation. First, the top-bottom communication in y-direction is performed in two half-steps. The first half-step consists of the exchange of arrays in positive y-direction meaning a send operation of the top halo array of the operating rank to its top neighbor and a receive operation of the operating rank with the bottom neighbor as source filling the bottom halo points. The second half-step points in negative y-direction and thus, the bottom halo array is sent to the bottom neighbor and the top halo data is received from the top neighbor. This is illustrated in Figure 2. In a second step, the same half-step procedure is performed to exchange the left-right communication (x-direction).

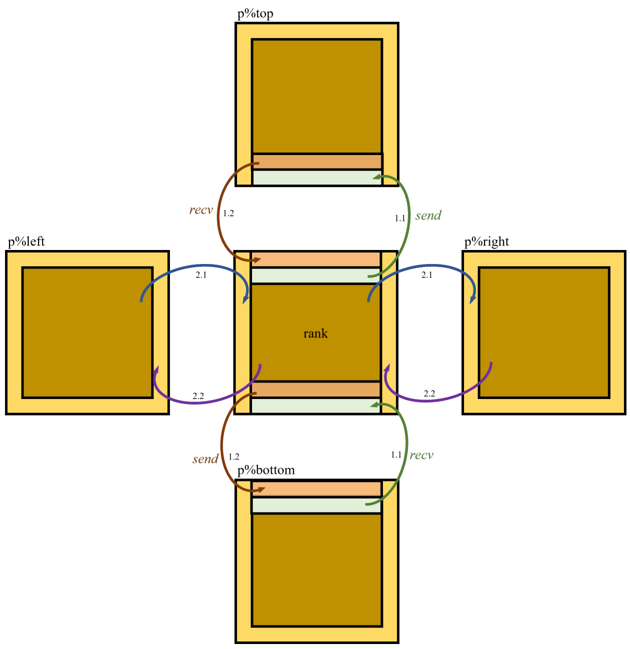
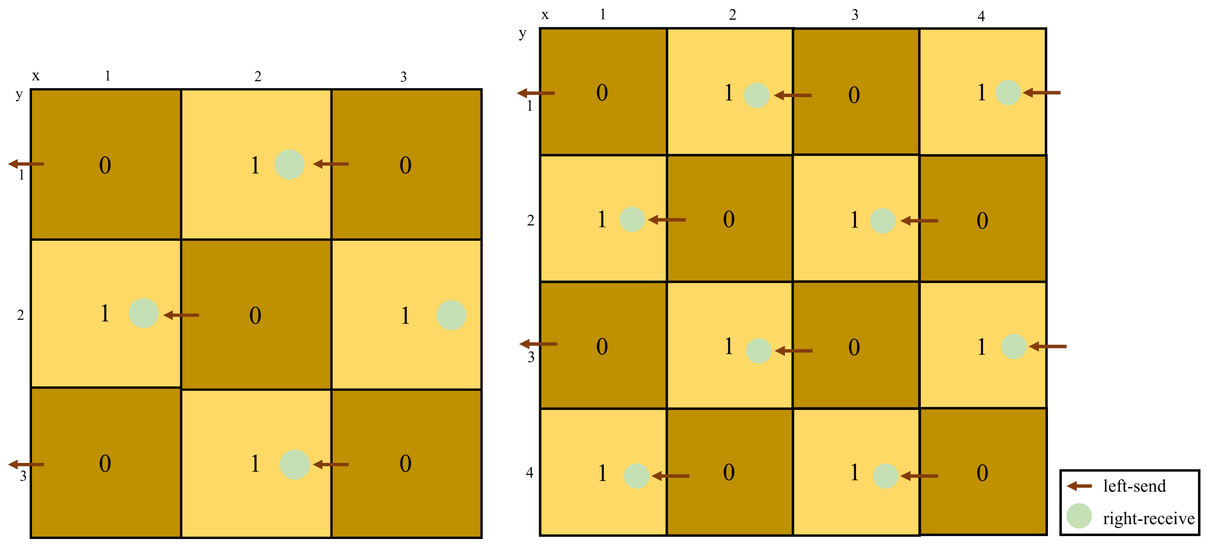


Figure 2: Schematic of *sendrecv* communication strategy.



(a) rank partition on a 3×3 grid (b) rank partition on a 4×4 grid

Figure 3: Schematic of the Send Recv on even odd communication strategy.

2.1.5 Send Recv on even odd

stencil2d-evenodd.F90
This strategy does not make use of any non-blocking methods. Instead we divide all ranks into two disjunct groups and perform the communication in two steps: first, all ranks from the first group send their send-buffers making use of the blocking send *MPI_Send* while all ranks from the second group are waiting to receive, using the blocking receive *MPI_Recv*. Secondly, the ranks which previously were receiving are now sending and vice versa. To divide the ranks into groups we use the grid that is assigned to the ranks by the partitioner. Each rank is assigned an x and y coordinate on the grid as shown in Figure 3a and 3b. The ranks are then divided into two groups as follows:

- Group 0: (x-position mod 2 + y-position mod 2) mod 2 = 0
- Group 1: (x-position mod 2 + y-position mod 2) mod 2 = 0

In the first step, group 0 is sending while group 1 is receiving, followed by the second step where group 1 is sending and group 0 is receiving. As seen in Figure 3a and 3b, this is not always straight forward as the partition of the ranks on the grid does not guarantee for an alternating pattern of the groups, especially across the periodic boundaries. In this work we implement this communication strategy only for rank numbers of $2^{2n} > 4$.

2.2 Validation

To check whether the implementation of the chosen communication strategies in the halo update of the stencil 2D program was done correctly, a validation has to be done. Therefore, the output file of the original version of the program is compared to the output file of each newly implemented version. To do a consistent comparison the same parameters (dimensions, number of iterations) have to be selected for each implementation. The fields validate if the two arrays are element-wise equal within a tolerance. All fields validate with a tolerance of 10^{-3}.

2.3 Performance analysis

Experiments

To compare the different communication strategies, we performed three experiments of which the first two refer to the scalability of the strategies and the third experiment compares the run time when increasing the message size of each rank in the vertical.

In the strong scalability experiment the problem size remains constant, while the number of ranks increases in steps of 12 (as in our project one node consists of 12 ranks). Thus, the message size decreases with increasing number of ranks. The problem size is given by the horizontal domain $nx \times ny = 128 \times 128$ and the vertical domain of $nz = 64$. The stencil diffusion program runs for 64 iterations.

To look at the weak scalability of the different strategies we keep the problem size per rank constant while increasing both the horizontal domain and the number of ranks. The vertical domain was kept constant at $nz = 64$. Table 1 shows how the ranks and the domain are increased, keeping the message size constant at 512 grid points per rank. Each simulation was again run for 64 iterations.

Table 1: Chosen *number of ranks*, nx and ny values to measure the weak scalability.

ranks	2	4	8	16	32	64	128
$nx \times ny$	4×4	8×4	8×8	16×8	16×16	32×16	32×32

To analyse the dependence of the MPI communication on the message size, the number of the vertical grid cells (nz) is varied in the third experiment. Therefore, the *original* version (without parallel communication) is compared with the *mpi* version. While nz set to 2^n points, nx and ny are kept constant at 128. The number of ranks is set to 12 and 64 iterations are done.

Time measurements

We evaluate the performance in two ways: on the one hand, we measure the time the stencil needs to perform the whole diffusion computation, the part of the program that is parallelised, referred to as the work time. On the other hand, we investigate the time needed only for the halo update containing the MPI communication. We refer to this as the halo update run time. As seen in Figure 4 the work time is significantly larger than the halo update run time for small numbers of ranks but both converge towards each other for larger amounts of ranks. This means that as the number of ranks is increased the communication takes up a larger proportion of the total workload.

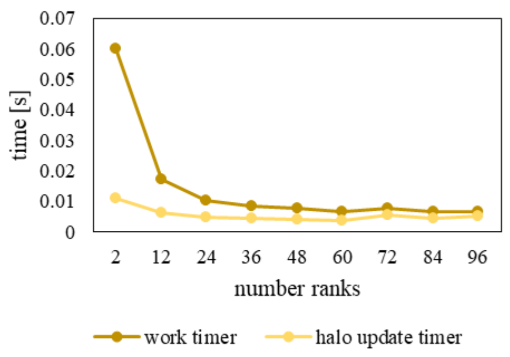


Figure 4: Difference between measuring the whole work time and the time needed for the halo update of the mpi version. y-axis: time [s]. x-axis: number of ranks. The other parameters are set as: $nx = ny = 128$, $nz = 64$ and $number\ of\ iterations = 64$)

3 Results

In this section, we analyse the different communication strategies. Figure 5 shows the speedup between the original non-parallel stencil program and the different parallel strategies. Here, we measure the whole speedup gained through the parallelisation including the calculation of diffusion. All MPI versions perform significantly better than the original non-parallel version. However, the speedup stagnates after approximately 60 ranks. All versions behave comparably well with the exception of the *Irecvsend* version, which has a significantly longer run time especially within the increase of the first 10 ranks. The overall fastest run with 0.006 026 s is version *sendrecv* with 80 ranks.

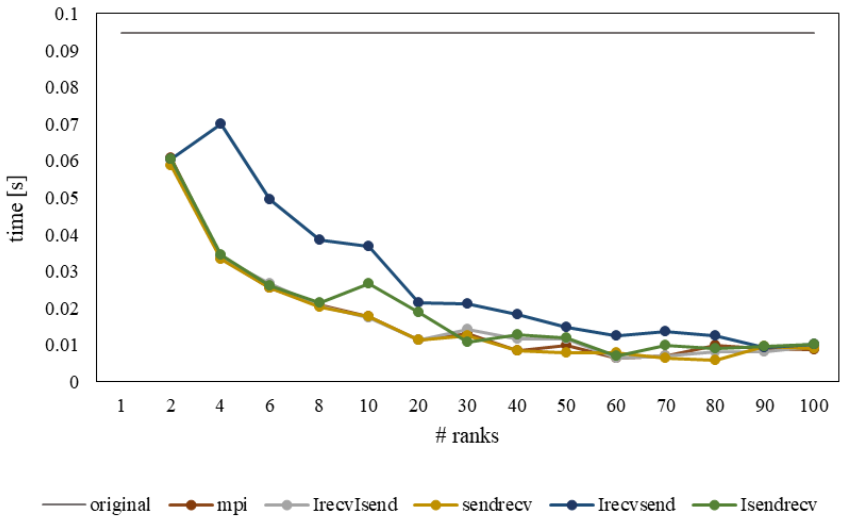


Figure 5: Performance overview. y-axis: time [s]. x-axis: number of ranks. The other parameters are set as: $nx = ny = 128$, $nz = 64$ and $number\ of\ iterations = 64$

To compare the communication strategies directly we use the halo update run time measurement described in subsection 2.3 for the following experiments.

3.1 Strong scalability

The strong scalability analysis shows that the time spent on the halo update decreases for most strategies until approximately 60 ranks (Figure 6). Thereafter, only strategy *IrecvIsend* has a constant decrease in run time, while other strategies do not show a speedup anymore. At 12 ranks *Irecvsend* shows a distinct maximum. The run time of strategy *sendrecv* increases and decreases alternately between 60 and 108 ranks. At a number of ranks higher than 84, there seems to be a slowdown in run time which could be due to inefficient communication as the problem size remains constant and many messages with very small message size are exchanged. We excluded the data for the even odd strategy since this experiment was only done for $16, 32, 64$ and 128 ranks. However, the results do not differ greatly from the other communication strategies, ranging between $5.33 \times 10^{-3}s$ and $8.90 \times 10^{-3}s$.

In an ideal computational world we would expect a linear decrease in run time with increasing number of ranks compared to a non-parallelised version. However, this assumption does not hold in this case primarily due to the time measurement method used as we only compare the time spent on the halo update. The halo update for the original non-parallel stencil program is comparably fast for all strategies. Thus, even though there is no significant speedup in the new strategies for the halo update, the parallel version still outperforms the non-parallel version as the speedup of the computation of the diffusion outweighs the communication load in the halo update.

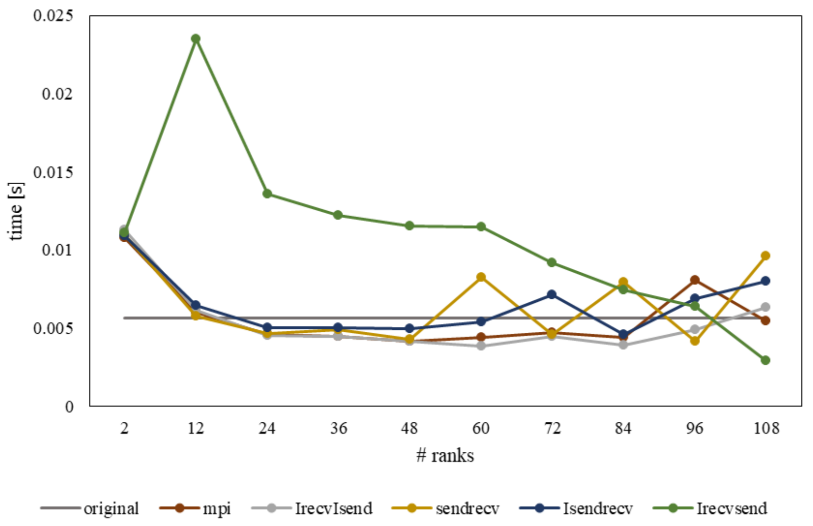


Figure 6: Strong scalability analysis of the different communication strategies. y-axis: time [s]. x-axis: number of ranks. The other parameters are set as: $nx = ny = 128$, $nz = 64$ and *number of iterations* $= 64$.

3.2 Weak scalability

The results for the weak scaling experiment are shown in Figure 7. Overall, we see that the time to complete the halo update increases with an increasing number of ranks. In some cases, the time increases by a factor of up to 5. The strategies do not differ significantly. While *Irecvsend* is the slowest strategy on a small number of ranks, it is the fastest one at 128 ranks. The *evenodd* strategy was tested on $16, 32, 64$ and 128 ranks and has again performed averagely, outperforming the other strategies at 16 ranks but lying in between the *mpi* version and the *sendrecv* on 128 ranks. Perfect weak scaling would appear as constant time needed for constant workload per rank. However, the observed increase in time is presumably due to increased communication latency owed to limited bandwidth in the communication network.

3.3 Variation of message size

In Figure 8 the result of the message size variation experiment is shown. In general, the result illustrates that with increasing *nz*, both versions, the *original* and the *mpi* version, need more time for the halo update. For small *nz* values, the increase in time is very similar between the two versions. For *nz* approximately larger than 32, the run time increases stronger in the *mpi* version compared to the *original* version. This might be explained by the additional work within the *mpi* version, needed for the communication process including the packing and unpacking steps. In contrast, the non-parallel version is not limited this. Furthermore, as the field is small and the computation is not memory bound, the memory bandwidth is an increasingly limiting factor for increasing message sizes. This could also explain the smaller run time of the original version.

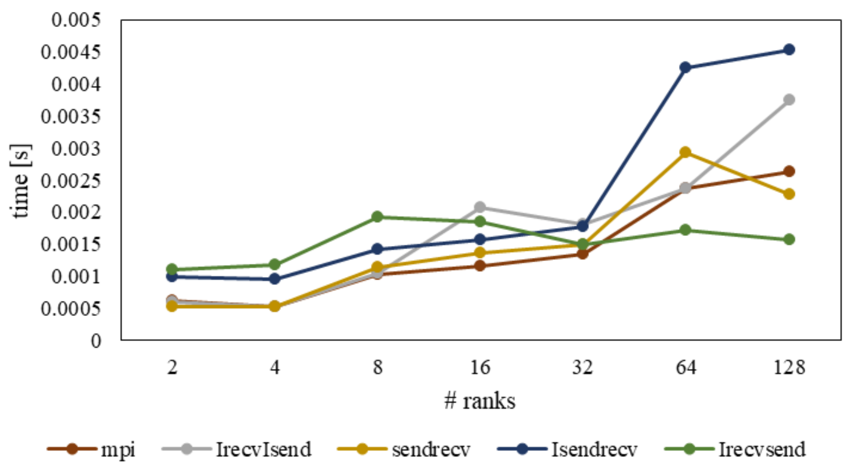


Figure 7: Weak scalability analysis of the different versions. y-axis: time [s]. x-axis: number of ranks. The other parameters are set as: $nx = ny = 128$, $nz = 64$ and $number\ of\ iterations = 64$

.

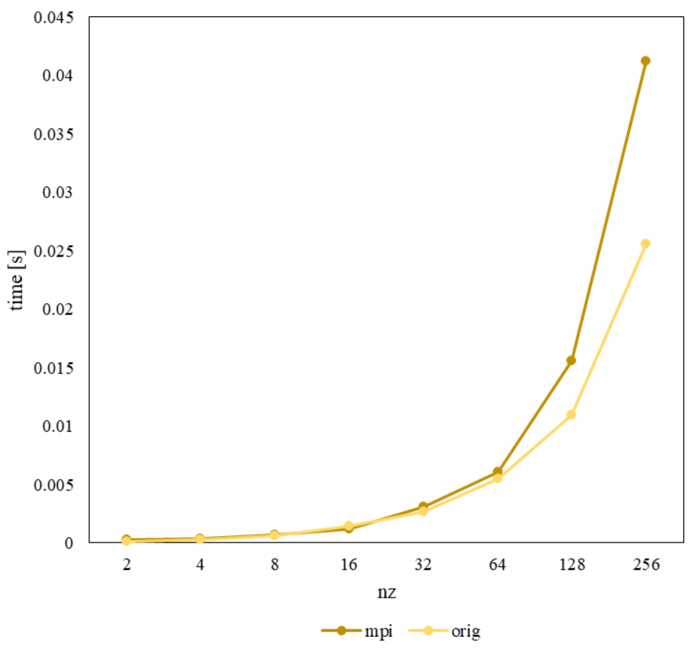


Figure 8: Time used for the halo update as a function of the message size nz. The other parameters are set as: $nx = ny = 128$, $number\ of\ ranks = 12$ and $number\ of\ iterations = 64$.

4 Discussion and Conclusion

Overall, the implementation of MPI communication strategies in the halo update of the stencil 2D is useful to reduce the work time of the program. This statement is supported by the results found by conducting three different experiments. We found that all different versions of the program using MPI run faster than the original version without communication (see Figure 5). The fastest run time was achieved by the *sendrecv* version with the parameters set to $nx = ny = 128$, $nz = 64$, $\textit{number of iterations} = 64$ and $\textit{number of ranks} = 80$. However, the speed is still comparable to the other versions and not very much faster. The *evenodd* version of the stencil performs comparably to the other versions, however, it is not applicable in all cases since it can only be used on 2^{2n} number of ranks.

Furthermore, subsection 3.1 showed that with a decreasing problem size the computation is sped up until there is an overhead introduced in the halo update by an increased amount of time spent on the communication of ghost points, outweighing the speed up achieved by parallelising more work. Similarly, in subsection 3.2 we see that with a constant work load but increasing number of ranks the communication slows down the halo update due to latency resulting from limited bandwidth availability. With an increasing message size as in subsection 3.3 the duration of the halo update increases faster for the *mpi* version than for the non-parallelised original owing to an increased workload of packing and unpacking the communication data. All in all, when using MPI communication for parallel stencil computation, one needs to keep in mind a certain trade-off between decreasing the computational workload per rank and increasing the overall communication load.

As we conducted each experiment only once without numerous repetitions, our results have to be considered with some uncertainty. However, as the variance calculated from 10 repetitions of the *mpi* version with 12 ranks is very small ($\sigma = 1.696\,17 \times 10^{-9}\,\mathrm{s}$), the uncertainty is negligible.
Further limitations of our results could stem from the limited capacities available for our experiments. The experiments were conducted with comparatively small domain sizes. When comparing the communication strategies for very large domain sizes, the results might differ as the problem would then become memory-bound. However, as all strategies involve either blocking communication or a call of the function *MPI_Waitall* we do not expect significantly different results.

Further investigation might be needed when considering different problems than two-dimensional diffusion. Moreover, the MPI library offers more send and receive functions such as *MPI_Sendrecv_replace* or *MPI_Rsend*, which could achieve even more efficient communication run times.
Considering our results, the best communication strategy depends on individual needs concerning the problem and personal programming preferences. Apart from this, all MPI communication strategies perform similarly well and thus, each presents a suitable method to implement parallel communication in a stencil program.

References

Fuhrer, Oliver (May 2020a). *stencil2d-mpi.F90*.

Fuhrer, Oliver (May 2020b). *stencil2d-orig.F90*.

Lusk, E et al. (2009). “MPI: A message-passing interface standard Version 3.0”. In: *International Journal of Supercomputer Applications* 8(3/4). ISSN: 15206882.

Nielsen, Frank (2016). “Introduction to MPI: The Message Passing Interface”. In: pp. 21–62. DOI: `10.1007/978-3-319-21903-5{\_}2`.

Sloot, Peter M. A. et al., eds. (2002). *Computational Science — ICCS 2002*. Vol. 2331. Springer Berlin Heidelberg: Berlin, Heidelberg. ISBN: 978-3-540-43594-5. DOI: `10.1007/3-540-47789-6`.

Sudwoj, Michal and Oliver Fuhrer (June 2020). *m_partitioner.F90*.

Xue, Ming (Aug. 2000). “High-Order Monotonic Numerical Diffusion and Smoothing”. In: *Monthly Weather Review* 128(8), pp. 2853–2864. ISSN: 0027-0644. DOI: `10.1175/1520-0493(2000)128<2853:HOMNDA>2.0.CO;2`.