Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelorarbeit

Performance modeling of one-sided and two-sided MPI-3 communication

vorgelegt von

Niclas Schroeter

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	6926553
Erstgutachter:	Jun.-Prof. Dr. Michael Kuhn
Zweitgutachter:	Jannek Squar
Betreuer:	Jun.-Prof. Dr. Michael Kuhn Jannek Squar

Hamburg, 2021-05-09

Abstract

The MPI 3.0 standard introduced numerous changes to its remote memory access interface. This interface offers support for one-sided communication. In this thesis, the performance of MPI-3 RMA is compared to the performance of the more traditional point-to-point communication with MPI. This comparison is conducted on a stencil-based application for the calculation of partial differential equations using either the Gauss-Seidel method or Jacobi's method, involving three versions of said application that communicate through different means. These versions consist of a point-to-point version and two RMA versions, one that uses shared memory and one that does not. Results indicate that neither approach to communication is clearly favored. Therefore, the better performing communication form has to be determined on a case-by-case approach for any given application.

Contents

1 Introduction

In today's environment of large-scale high performance computers, many scientific developers try to fully utilize a compute node's potential by introducing parallelization on the level of shared memory to their applications. This is most commonly done with OpenMP [2], as its entry barrier is comparatively low due to the compiler-based approach with pragmas that also allow for incremental parallelization. However, limiting these applications to a single node limits scalability, as a single node might not provide enough memory for large-scale problems for example. Introducing parallelism on distributed memory is considerably more difficult than just adding the pragmas for OpenMP though, which might be out of scope for many scientific developers.

To tackle this problem, a tool using Clang and LLVM is currently in development, called CATO [17]. The goal of this tool is to automatically transform the source code of OpenMP kernels into code that leverages distributed memory via MPI, one of the most prominent interface specifications that allows for parallel processing on both shared and distributed memory. This transformation requires the automatic insertion of all necessary MPI calls, ranging from the initialization of all processes, over the synchronization to the communication of relevant data. To communicate said data, MPI defines many different communication functions which will be discussed in greater detail in Section 3.1. For the purposes of this thesis, the most important distinction is the one between one-sided communication and two-sided communication, also known as point-to-point (P2P) communication. P2P communication requires both the sending and receiving processes to specify the necessary parameters for the communication, while one-sided communication only requires the sending process to specify these parameters. In order for the final tool to provide the best performance possible in regards to communication, the goal of this thesis is to evaluate whether or not the MPI-3's one-sided communication should be employed instead of the more traditional point-to-point communication wherever necessary.

To determine this, a stencil-based PDE solver will be implemented in multiple different ways with the means provided by MPI-3 RMA. These initial versions will undergo certain strong and weak scaling experiments. The information gained from these experiments will be used to create the most optimal RMA version of this PDE solver, which will then be compared to a version that was implemented with point-to-point communication. This comparison is once again conducted through multiple weak and strong scaling experiments on not only different sets of hardware, but also with different MPI implementations.

2 Related work

As OpenMP code is comparatively easy to write, many scientists have attempted to automatically transform OpenMP code. In order to utilize GPUs without having to recreate the whole application, automatic compiler-based transformations from OpenMP to CUDA have been investigated in the past. Their goal was to retain the programmability of OpenMP programs and convert these programs into CUDA based programs to utilize the performance benefits that GPUs have to offer [15] [21], resulting in considerable speedups for certain tested kernels. Other efforts included the automatic transformation from ANSI C with OpenMP directives to SystemC [5].

Transformation into MPI code has been worked on as well, ranging from the development of compiler techniques [1] utilizing partial replication, leading to performance that is comparable to hand-written MPI code, to fully automated compiler-runtime systems [14]. The particular system mentioned in [14] employs runtime data flow analysis as well as a novel communication generation scheme alongside certain compiler techniques, averaging 75% of the performance of hand-written MPI applications during their tests. However, both papers only utilized point-to-point messages and collective communication operations, ignoring the possible advantages of RMA entirely.

Regarding a performance comparison between traditional point-to-point communication and one-sided communication with MPI, multiple papers have been published as well, covering comparisons on both micro and application benchmarks [13] [4], where many came to the conclusion that MPI-3 RMA performs at least comparable to P2P, if not better. For [4], the communication time decreased by 5-10%, but they note that RMA is not superior in every aspect when compared to P2P, as integrating RMA into existing code requires multiple optimizations to perform well. The library support of certain MPI implementations also appears to be lackluster, which emphasizes the importance of the MPI implementation in use. Considering that [4] is based on a Fortran 2003 application and [13] was conducted on a Blue Gene/Q machine while only exploring one specific MPI implementation, further work is needed to achieve potentially universal results in this particular comparison of P2P and RMA. This thesis aims to add to the aforementioned results by utilizing both different MPI implementations and hardware, comparing performance via a C-based application benchmark.

MPI-3 also introduced a shared memory model, which would enable the user to combine both inter- and intra-node communication without having to use two parallel programming systems at once. According to [9], shared memory performs considerably better than its MPI-P2P + OpenMP counterpart, as their study on a 5-point stencil benchmark revealed that using shared memory reduces data movement time by 40% on average. In [22], MPI-3 shared memory is used to improve the runtime system of DASH, a C++ template library, by utilizing shared memory for blocking intra-node data

transfer. Their results lead to the conclusion that shared memory did indeed improve the runtime system's performance of intra-node communication while also performing comparable to other PGAS approaches like UPC or OpenSHMEM on said intra-node operations. Once again, while these results are relevant, it is not enough to universally assess the performance of shared memory, which is why this thesis aims to add to these results. It should also be mentioned that many years have passed since the release of these papers ([9] was released in 2013), during which many MPI implementations have been further improved in multiple aspects, including their implementation of shared memory.

3 Background

3.1 MPI

The Message-Passing interface (MPI) is a library interface specification, first defined in 1994, which mainly employs the message-passing parallel computing model, meaning that processes cooperate by moving data in between said processes. This is primarily executed with two functions, `MPI_Send` and `MPI_Receive`. The first function is used by a process to send data from its own address space to another process, while the latter function is used to receive such data. The MPI interface provides many extensions beyond sending and receiving a message, for example collective operations that involve multiple processes at once, parallel I/O and remote memory access operations. These remote memory access operations will be the focus of the following sections.

Remote memory access (RMA) was first introduced to the MPI standard in 1997 with the release of MPI-2.0. With RMA it is possible to specify all parameters needed for the communication on one process instead of having to specify them on both the sending and receiving process, as it is the case with point-to-point (P2P) communication in MPI. This eliminates the necessity of matching send and receive calls, which in turn reduces the need to poll for potential communication requests on the receiving process. Another key difference between P2P and RMA communication is the separation of communication and synchronization. Transferring data via message-passing incurred both communication and synchronization with the matching send and receive calls, while RMA was designed to separate them. Due to this separation, it is expected that RMA operations incur less synchronization in general, which in turn would improve the overall performance of communication. Thus, there are communication calls, such as `MPI_Put` and `MPI_Get`, and synchronization calls. Both categories will be explained thoroughly at a later stage.

The functions of the RMA interface were designed specifically to allow implementors to abuse fast communication mechanisms of the underlying hardware, such as shared memory or DMA engines [7, Chapter 11.1]. However, due to various reasons MPI-2 RMA was not widely used and played only a secondary role compared to the traditional message-passing paradigm. Among others, these reasons included an unjustifiable increase in complexity for many otherwise simple programs and that the performance gain was less significant than initially expected. In 2012, the release of the MPI-3 standard included a substantial extension to the RMA interface, as "architectural trends, such as RDMA networks and the increasing number of (potentially noncoherent) cores on each node, necessitated a reconsideration of the programming model" [10].

MPI-3 RMA is also separated into communication calls and synchronization calls, similar to the previous iteration in MPI-2. Both the communication and synchronization functions operate on MPI windows.

3.2 MPI windows

MPI windows are used to specify a contiguous memory region that is accessible to other processes in a given communicator, but not to any other process outside of this communicator. A communicator contains a group of processes and a context of communication. These windows can be initialized with different functions, namely `MPI_Win_create`, `MPI_Win_allocate`, `MPI_Win_create_dynamic` and `MPI_Win_allocate_shared`. The functions are all similar, as they return an opaque object with certain attributes. The window can then be accessed via this object, but there are differences in how these functions are used exactly. Every function is a collective call that allows each process in a given communicator to specify a nonnegative amount of bytes to expose in a window alongside a displacement unit to be used for RMA communication operations. The type of memory that can be exposed in a window depends on the function used for the initialization.

If a user initializes a window via `MPI_Win_create`, the user would expose already existing memory (previously allocated with `malloc` for example), whereas a call to `MPI_Win_allocate` would allocate new memory. This means that depending on which initialization function is called, the user has to either manage memory by themselves or passes that responsibility on to MPI.

The third function, `MPI_Win_create_dynamic`, does not allocate memory to a window nor does it create a window on previously existing memory. Instead, `MPI_Win_create_dynamic` binds only the group of processes in the given communicator. Each process can then attach and detach memory to that window locally via `MPI_Win_attach` and `MPI_Win_detach` respectively.

The last function, `MPI_Win_allocate_shared`, is used to create shared memory between the processes of a communicator. This allows processes on the same node to share their memory directly, which in turn enables local load/store actions on this memory for each process in this window. This should lead to a lower communication overhead than using the otherwise necessary RMA operations to access data in a window.

Once a window is no longer needed for RMA operations it should be destroyed with `MPI_Win_free`. If memory was allocated by creating the window, for example with `MPI_Win_allocate`, that memory will be deallocated upon destroying the window.

3.3 Communication operations

From now on, the calling process will be referred to as the origin, while the targeted process will be referred to as the target.

The two most basic communication operations are `MPI_Put` and `MPI_Get`. `MPI_Put` transfers data from the origin to the target, whereas `MPI_Get` transfers data from the target to the origin. Neither operation provides guaranteed ordering, facilitating an efficient implementation of these functions.

The function `MPI_Accumulate` also transfers data from the origin to the target, but instead of just overwriting the specified target buffer, the data is combined into that buffer according to the operation defined in the arguments. If the operation used is `MPI_REPLACE`, `MPI_Accumulate` acts as an atomic put.

Another similar function is `MPI_Get_accumulate`. This function first fetches the data from the specified target buffer, storing it in a different buffer at the origin, and then performs an accumulate at the target buffer. Using `MPI_Get_accumulate` with the operation `MPI_NO_OP` acts as an atomic get, whereas a call with the operation `MPI_REPLACE` can be used to atomically swap the contents of the target and the origin buffer.

However, these atomic reads and writes only guarantee their atomicity at an element-wise level, meaning that two simultaneous calls to `MPI_Accumulate` with the operation `MPI_REPLACE`, both targeting the same set of two elements, could lead to the set containing one element from the first call and one element from the second call. Furthermore, the ordering of accumulates is only specified for accumulates to the same or overlapping memory locations at another process on an element-wise level. Per default, this ordering is strict, meaning that overlapping updates from the same origin are committed in program order (the order of executions in a single threaded program). This ordering can be relaxed while creating the window to increase performance. Outside of this, no guaranteed ordering applies to accumulates.

There is also a request-based version of every communication function previously mentioned (for example `MPI_Rget`) that allows the user to wait for the operation to complete locally via `MPI_Test` or `MPI_Wait`. The only difference between the request-based version of a function and the normal version is an `MPI_Request` object, which is also used for non-blocking P2P communication. Local completion for operations that fetch data indicates that the data is now available in the origin buffer. For operations that write data, local completion indicates that the origin buffer used for this operation can now be updated safely. This `MPI_Request` object enables the user to overlap communication and further computation. For example, after initiating the communication with `MPI_Rget`, the user can compute additional information unrelated to the data that is supposed to be fetched. After completing the additional computation, the user can wait for the local completion of `MPI_Rget` by passing the request object to `MPI_Wait`.

Two more communication functions are defined for one-sided communication in the

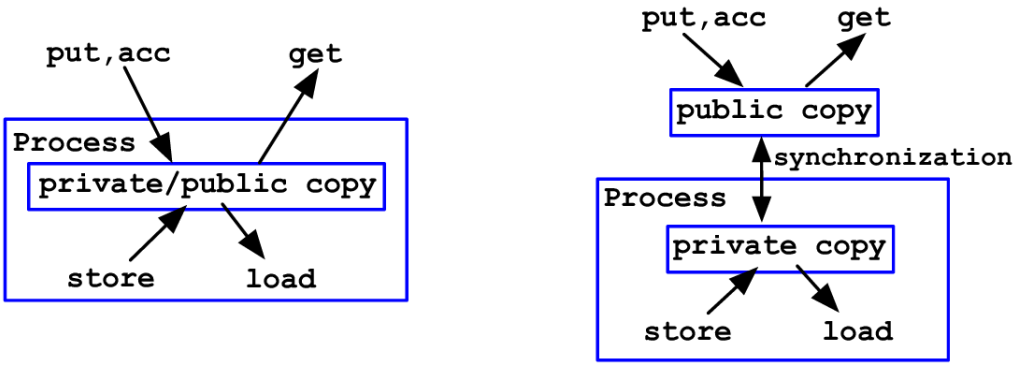
MPI-3 standard. One of them is `MPI_Fetch_and_op`, which is a specialized version of `MPI_Get_accumulate`. `MPI_Fetch_and_op` only accumulates a single element to the target buffer after fetching the previous content, allowing for a faster implementation of a certain subset of the functionality of `MPI_Get_accumulate`.

The last function is `MPI_Compare_and_swap`. This function is used to atomically compare a single value in the target buffer with a value from the origin. If the values are equal, the target value is replaced with a different value that was specified in the arguments, while the original value is stored in a buffer at the origin process, thus atomically swapping the two values.

3.4 Memory model

MPI-3 defines two different memory models. Each window is separated into a public copy (accessed by RMA operations) and a private copy (accessed by local load and store operations). This is also reflected in the hardware of many systems, as there is a public memory region that is accessible by other processes and process-local private buffers (caches or explicit communication buffers for example). These buffers are either coherent or non-coherent. A coherent buffer reflects all updates to the public memory in the private copies of that region consistently, whereas a non-coherent buffer needs to explicitly synchronize the private copies to reflect changes to the exposed memory region. These explicit synchronizations are performed with RMA functions, meaning that the coherency of the non-coherent buffers is dependent on software, while the coherent buffers rely on their hardware to maintain coherence.

The two different memory models of MPI were defined to properly accommodate for these differences. The first model is the separate model. It models systems with non-coherent buffers, where the public copy of a window is separate from the private copy. In order to ensure the correctness of a program while using the separate memory model, additional synchronization must be performed, either with the usual synchronization functions (see Section 3.5) or with `MPI_Win_sync`, which synchronizes the public and private copy of the window passed as the argument, as depicted in Figure 3.1 (b).



(a) Unified memory model (b) Separate memory model

Figure 3.1: The unified memory model (a) and the separate memory model (b) [10].

The second memory model is the unified model. In this model, the local and the public copy of a window are identical (due to hardware-managed coherency). Updates to the public copy of a window via RMA operations are eventually observable by local load operations without the need for additional synchronization calls. This allows the user to potentially omit certain synchronization calls to improve performance, also depicted in Figure 3.1 (a).

However, to maintain the portability of an MPI program, the user should program for the separate model, as a correct program in the separate model is always correct in the unified model as well, whereas the opposite is not always true, since the same program in the separate model might require additional synchronization. Therefore, in order to maintain portability while also not giving up on potential performance increases of the unified model, the user is advised to query the model used for each window and handle it accordingly.

3.5 Synchronization

The RMA communication operations are organized in epochs. An origin process can only communicate with the target if the origin is within an access epoch for the window that is supposed to be the target of the communication. Likewise, a window can only be accessed if it is within an exposure epoch. A process can simultaneously be in an access epoch and an exposure epoch for the same window. Access and exposure epochs for different windows may overlap. Two distinct access epochs for one window at the same process have to be disjoint. This is also true for exposure epochs. Upon closing an epoch with a synchronization operation, all outstanding communication operations are completed both locally and remotely. MPI provides multiple synchronization modes, namely active and passive target synchronization, their main difference being the involvement of the target process in synchronization. Furthermore, active target synchronization can be split into fence synchronization and general active target synchronization.

3.5.1 Fence synchronization

In fence synchronization mode, the `MPI_Win_fence` call is used to open both an exposure epoch and an access epoch for all processes associated with the window that is passed as the argument to the fence call. The same function is also used to close these epochs again, completing all outstanding communication operations. `MPI_Win_fence` is a collective call on all processes in the group of the targeted window and also implies a barrier synchronization of the aforementioned processes, as a process only completes the call to `MPI_Win_fence` once all other processes in the group have entered their respective fence call. The barrier synchronization can be circumvented for specific fence calls, namely a call to `MPI_Win_fence` that starts an epoch. In that case, an assertion argument can be passed to the fence call, disabling the barrier-like functionality for that specific `MPI_Win_fence`. Fence synchronization is useful in many situations, for example if the algorithm is separated into distinct communication and computation phases, as depicted in Figure 3.2.

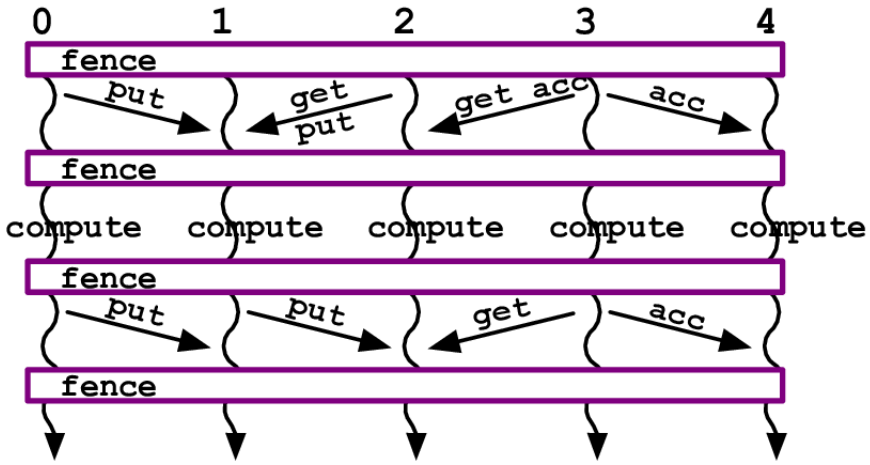


Figure 3.2: Example of active target synchronization. All processes open an epoch with the fence call, then various communication calls can occur until the epoch is closed again with another call to fence [10].

3.5.2 General active target synchronization

Compared to the fence synchronization mode where the fence calls have to include every process associated with a window, the general active target synchronization allows for a more fine-grained approach. Every process is able to define to which other process it opens an access epoch or an exposure epoch. This circumvents the entailed barrier synchronization in `MPI_Win_fence`, thus reducing the overall synchronization needed.

There are four functions for general active target synchronization: `MPI_Win_post`, `MPI_Win_start`, `MPI_Win_complete` and `MPI_Win_wait`. To start an access epoch, the origin calls `MPI_Win_start`, to which a group argument is passed that contains all the processes whose windows the origin wants to access. These processes have to match the call with the non-blocking `MPI_Win_post`, starting an exposure epoch to the processes specified in their group argument. To close the access epoch of the origin, `MPI_Win_complete` is called. This call only returns once all communication operations have completed at the origin. Similarly, to close the exposure epoch, the target process calls `MPI_Win_wait`. This call will block until all matching calls to `MPI_Win_complete` have occurred, guaranteeing completion at the origin of all RMA operations on the local window. Once the call to `MPI_Win_wait` returns, all of the RMA operations issued will be completed at the target window as well.

There is also a non-blocking variant called `MPI_Win_test`. If the accesses have not been completed this function returns `flag = false`. If the accesses have completed, the function behaves exactly like `MPI_Win_wait`. This matches the behaviour of the functions `MPI_Wait` and `MPI_Test`, which are used to test or wait for the completion of request-based communication operations like `MPI_Irecv` or `MPI_Rget`.

General active target synchronization works best if the communication pattern is mostly fixed and each process only communicates with a small number of other processes, for example with stencil-based algorithms.

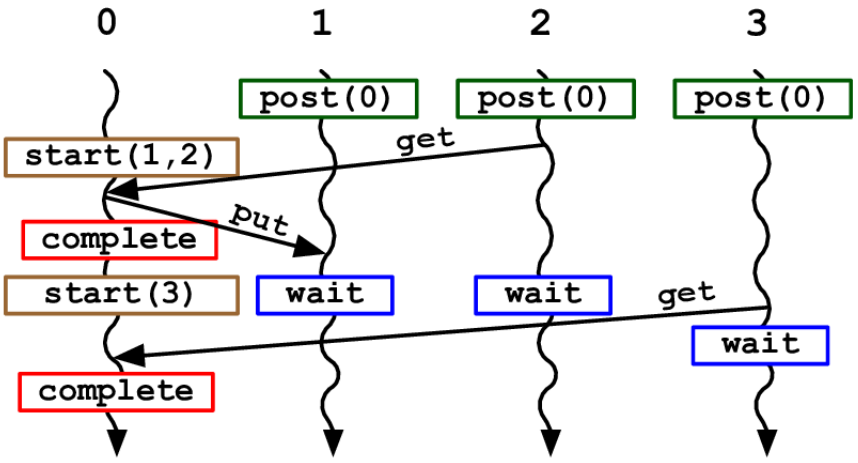


Figure 3.3: Example of general active target synchronization. Processes 1, 2 and 3 start an exposure epoch in which they can be accessed by process 0. Meanwhile, process 0 first starts an access epoch with process 1 and 2 as the targets, communicates with them before ending the epoch to start a new access epoch on process 3. After all communication is complete, processes 1, 2 and 3 end their exposure epoch by calling the wait function [10].

3.5.3 Passive target synchronization

In passive target synchronization, any window can be accessed at any time, making the concept of exposure epochs obsolete, as a window is in such an epoch constantly. In this synchronization mode the origin is the only process that is explicitly involved in communication and synchronization. Access epochs are started with `MPI_Win_lock` and end with a call to `MPI_Win_unlock`. When locking a window, the origin process can choose between a shared lock or an exclusive lock, similar to those used in reading and writing. Locking a window exclusively ensures that other lock-protected accesses to the same window do not occur concurrently. However, unlike regular read-write locks, a call to `MPI_Win_lock` does not need to wait for the lock to be acquired before returning, unless it is used to lock a local window to protect local load and store actions. Due to this, the user cannot rely on these locks for mutual exclusion and has to ensure said exclusion through other means, if necessary.

Instead of locking and unlocking only a single process per call, the `MPI_Win_lock_all` and `MPI_Win_unlock_all` functions can be used to acquire a shared lock to all other processes associated with that window or unlock them respectively. As with the other synchronization modes, a call to close the access epoch blocks until all outstanding communication operations have completed. In passive mode, this would also entail releasing any potentially exclusive locks. To circumvent that problem MPI provides multiple functions that allow the user to flush any outstanding RMA operations without releasing said locks. There are four functions, the first pair being `MPI_Win_flush` and `MPI_Win_flush_all`. Calling these will complete any outstanding RMA operations both locally and at the target. The second pair of functions is `MPI_Win_flush_local` and `MPI_Win_flush_local_all`, which complete all outstanding communication operations at the origin but not necessarily at the target. This may be useful if the user wishes to reuse a buffer that was used in a previous accumulate or put call for example.

This synchronization mode is useful for many applications that access the data of other processes at irregular times.

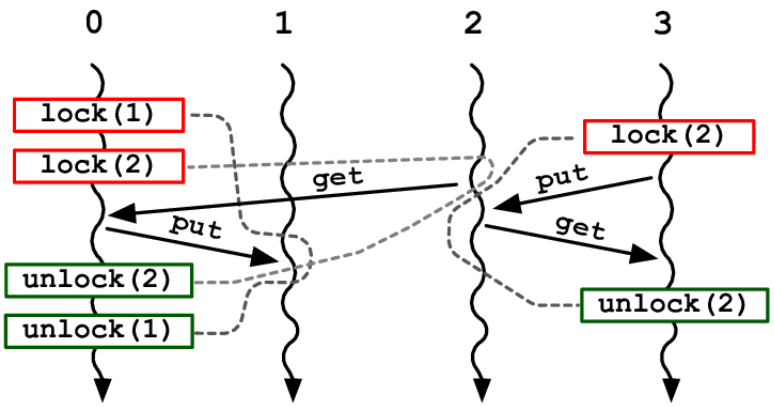


Figure 3.4: Example of passive target synchronization. Note that an origin process can be in multiple access epochs at once, but only if the target processes are distinct [10].

It is possible to use multiple synchronization modes at once, for example using fence synchronization for all regular communication and using passive target communication for any irregular updates. Though it should be noted that a window cannot be in an exposure epoch and be locked at the same time. This is considered erroneous behaviour.

4 Approach

In order to investigate whether or not MPI-3 RMA performs better than traditional point-to-point communication, multiple versions of a solver for partial differential equations (called Partdiff) were implemented. One version uses exclusively message passing and collective communication operations where necessary, while the other versions utilize exclusively RMA operations and also some collective communication operations as well. The RMA versions are split into three categories, one for each synchronization mode. A Partdiff version of the *Fence* category exclusively uses `MPI_Win_fence` for synchronization, while a version of the *Passive* category only utilizes passive target synchronization. The reasoning for using only one synchronization mode per version of Partdiff is to determine whether one synchronization mode performs better than the others.

4.1 Partdiff

Partdiff calculates the solution of the Poisson equation in $[0, 1] \times [0, 1]$, discretized in an equidistant square grid with a size of $(N + 1) \times (N + 1)$, where N is the number of intervals in every direction. In order to achieve that, either the Gauss-Seidel method or Jacobi's method is employed, depending on the user's input at the start. Both algorithms iterate over the entire matrix by using the average of the four nearest neighbours of a point and the point itself (see Figure 4.1), also known as a five-point stencil of a certain point, and then storing the result. This is done until convergence. If Jacobi's method is employed, the result is stored in a second matrix, not affecting the other results of the current iteration. If the Gauss-Seidel method is used, the result will overwrite the old value in the matrix, thus affecting the results of its neighbours in the current iteration. For more information consult [16], as any further explanation is out of scope for this thesis.

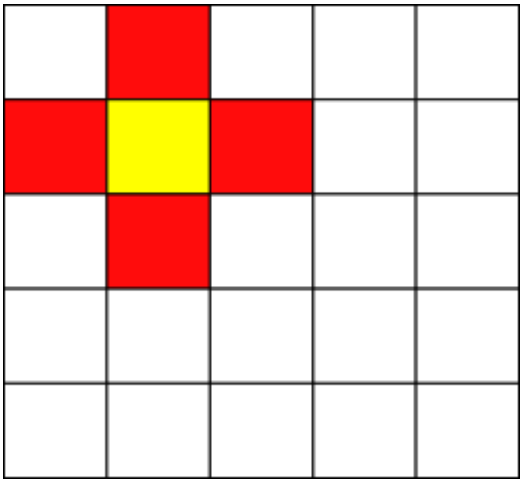


Figure 4.1: Example of a five-point stencil. In order to calculate the result of the yellow matrix entry, both the yellow entry itself and the four neighbouring red entries are taken into consideration.

Regarding Partdiff, the user can specify multiple options at startup. As previously mentioned, the user can specify the method used to solve the partial differential equations. The user can also set the size of the equidistant square grid via the interlines argument. The size of the grid depends on the interlines I in such a way that $N = I \cdot 8 + 9$. Furthermore, the user can decide between two disturbance functions to be used in the Poisson equation. The first function is $f(x, y) = 0; 0 < x < 1, 0 < y < 1$, where x and y are the coordinates of the value in the matrix divided by N. The second disturbance function is $f(x, y) = 2\pi^2 \sin(\pi \cdot x) \sin(\pi \cdot y); 0 < x < 1, 0 < y < 1$, with x and y being defined as before. Lastly, the user can decide upon the convergence criterion. The user can either provide a number of iterations that Partdiff will execute or a precision that has to be surpassed.

4.2 Implementation

The original sequential version of Partdiff was provided to students of the High-Performance Computing course at Universität Hamburg. This version serves as a base for the implementations with MPI. As previously mentioned, Partdiff offers both the Gauss-Seidel method and Jacobi's method to solve the partial differential equations.

4.2.1 Jacobi's method

As explained in Section 4.1, Jacobi's method stores the results of the average of a point and its neighbours in a second matrix. Therefore, two matrices are used for the implementation. In every odd iteration, Partdiff iterates over the first matrix and stores the results in the second matrix. Conversely, the results of every even iteration are stored in the first matrix.

For the parallel implementation with MPI, the matrices are split into lines. Every process stores a certain number of contiguous lines of the complete matrices, depending on the size of said matrices and the number of processes involved (see Figure 4.2). Each process then calculates the results of their lines and stores them in the appropriate matrix. In order to be able to calculate the results for the top and bottom rows of the process-local matrices, a process needs to access the lines above the top row and below the bottom row, globally speaking (see Figure 4.3). These halo lines are received from the neighbouring processes that contain these lines in their memory. For example, if the second process in a communicator wants to start the computation of any iteration, this process first has to receive the last line of the first process, as the values that are stored within this line are needed for the computation of the results of the first line in the second process. The same conditions apply to the last line of the second process. To calculate the results of that line, the second process needs access to the first line of the third process. These halo lines are communicated between processes via MPI. Swapping these halo lines has to be done at the start of each iteration.

Therefore, the parallel version of Jacobi's method can be split into two distinct phases: a communication phase in which halo lines are exchanged between processes, and a computation phase in which the halo lines and the rest of the local matrix of every process are used to calculate the results of that iteration.

Due to this split, terminating at the correct time is trivial. If the termination criterion is the number of iterations, every process stops after the necessary iterations are completed. If a certain precision needs to be surpassed, a collective communication operation is performed at the end of each iteration, in which the so-called residuum of every process is reduced to the maximum value of all processes. This maximum is then communicated to every process involved and once the necessary precision is reached, each process exits the calculation.

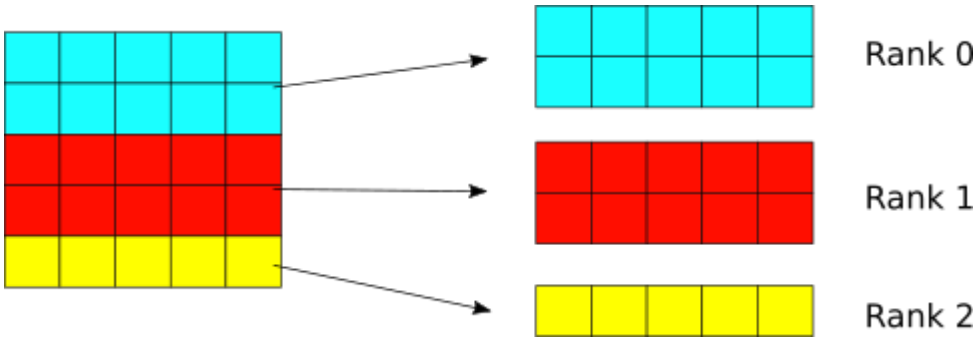


Figure 4.2: Example for the distribution of matrix lines between multiple processes. In this particular example, five lines are distributed between three ranks as evenly as possible.

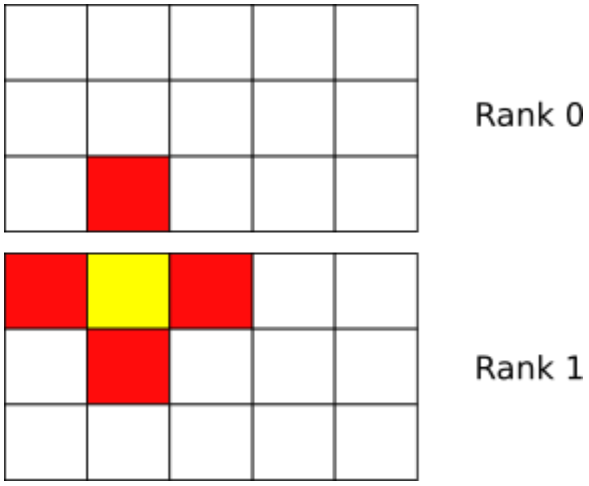


Figure 4.3: Example of a process using the values of a neighbouring rank's halo lines for local calculations. In order for rank 1 to calculate the results of the entry marked in yellow, it needs to know the value of a matrix entry that is stored by rank 0. Due to this, communicating the halo lines is necessary to successfully complete computation.

4.2.2 Gauss-Seidel method

Compared to Jacobi's method, the Gauss-Seidel method is more difficult to parallelize efficiently, as there is no global split into two distinct phases. As mentioned, all values are stored in a single matrix, which includes both the old values of the previous iteration and the new values that have already been calculated in the current iteration. Thus, exchanging the halo lines at the start of an iteration is not possible, as the last row of the preceding rank of any process has to be updated first before it can be sent to said process.

In order to have every process compute something at every point in time, a different approach to the parallelization is required. The lines of the matrix are partitioned between processes in the same way as before, the actual computation of the results is done differently though. Consider the following example: the first process (rank 0) starts the computation. Rank 0 processes its local matrix until it reaches its last line. In order to continue, the first row of the next process (rank 1) is required. Once rank 0 receives this line, it can finish the computation for this iteration. Once finished, the updated last line of rank 0 is sent to rank 1. Now, rank 1 can start its computation of the local matrix until it reaches its last line as well, repeating the swap of the respective halo lines with the next process (rank 2) and so forth. Meanwhile, rank 0 can start the computation for the next iteration and only has to stop once it reaches its last line once again, repeating the process that was illustrated previously.

If the Gauss-Seidel method is parallelized as such, every process computes something at every point in time while waiting times are minimized, not considering the short

startup period. While this approach utilizes the resources given to Partdiff in an efficient manner, terminating correctly is more challenging than it is with Jacobi's method.

If the termination criterion is a certain number of iterations, terminating correctly is still trivial. However, terminating once a certain precision is reached is more difficult, as every process is in a different iteration compared to the other processes at almost every point in time. In order to minimize waiting times, using a collective operation to reduce the residuum at the end of every iteration is only feasible if the operation is non-blocking. This operation will not complete until the last process of the communicator executes this operation. If the result of that operation meets the termination criterion, every process should theoretically terminate immediately. However, as the other processes have continued their computation while the last process matched the collective communication operation, the other processes are in different iterations altogether. At this point the process with rank 0 will be the process that has completed the most iterations. As soon as this process is notified that the termination criterion has been surpassed, the first process has to inform every other process of the number of iterations that they have to complete to catch up to the first process. Otherwise, the local matrices will contain results of different iterations upon termination, which leads to incorrect results overall.

For this reason, the parallelized Gauss-Seidel method terminates some iterations later than the sequential version with the same input at the start. The negative impact of increasing the number of iterations is outweighed by the performance gained from parallelization though.

4.2.3 Details regarding implementations and order

The first parallel Partdiff version to be built is the version using point-to-point communication. All communication is done via non-blocking sends and receives, which contain the necessary halo lines, alongside some collective communication operations wherever necessary. All following mentions of the P2P version of Partdiff refer to this version.

Afterwards, the first RMA versions are built. To determine whether `MPI_Put` and `MPI_Get` perform comparably, multiple benchmarks will be executed first, and then two Partdiff versions using fence synchronization are created. The only difference between the versions is that one uses exclusively `MPI_Put` for any communication necessary, while the other utilizes only `MPI_Get`. Depending on the communication operation that is utilized, different parts of the local matrices have to be exposed in MPI windows. The version communicating via `MPI_Get` exposes the local matrices' halo lines themselves with `MPI_Win_create` while the version using `MPI_Put` allocates new memory via `MPI_Win_allocate` in which every process can buffer the necessary halo lines.

Based on the results, the better version of the two will be used for further testing and the versions using general active target and passive target synchronization, which will be built afterwards, will utilize the better-performing communication operation wherever feasible.

It should also be noted that hints are passed wherever possible. These hints are used to provide additional information regarding the function they are called with, allowing the MPI implementation to potentially deliver increased performance in some way. However,

an implementation is free to ignore these hints. But in order to create an efficient program regardless of the actual MPI implementation used, the decision was made to pass as many hints as possible. If the implementation that is used does not support them, the hints will not have any negative impact on the performance whatsoever. Therefore, providing these hints has no drawbacks.

Examples for functions that offer hints are the window initialization functions, such as `MPI_Win_create`. Hints for this function include `no_locks`, which can be passed if passive target synchronization is not used on the created window at any time, or `accumulate_ordering`, which is used to control the ordering of accumulates on this window, if necessary.

Many synchronization functions offer hints as well. For example, the hint `MPI_MODE_NOPRECEDE` can be passed to `MPI_Win_fence`, which indicates that this particular synchronization call does not end an epoch. This information could be used to disable the barrier-like behaviour for this particular call of `MPI_Win_fence`.

5 Implementation and optimization

Unless specified otherwise, the following experiments are conducted on multiple AMD Opteron 6344 nodes with 48 cores each on the cluster provided by the research group Scientific Computing (WR) of Universität Hamburg. All nodes are connected via Infiniband. The MPI implementation used is MPICH version 3.3.2, unless explicitly stated otherwise. Every version was compiled at optimization level 3 (`-O3`). Every measurement was taken three times, the results displayed are the average of these three measurements, alongside their respective standard deviations. In Section 4.1 it is mentioned that the user can choose one of two disturbance functions. The disturbance function used for all following tests is the first one ($f(x, y) = 0; 0 < x < 1, 0 < y < 1$), as it will take less time to locally compute every single result, which emphasizes the communication times.

This chapter explores different implementation approaches with MPI RMA and provides a subsequent evaluation of these approaches. No comparisons to P2P communication are conducted in this chapter. Instead, these comparisons can be found in Chapter 6.

The content of this chapter is separated into three sections:

- Section 5.1 compares `MPI_Get` and `MPI_Put` to determine if one offers performance advantages over the other.
- In Section 5.2, Partdiff is implemented thrice, one version per synchronization mechanism. These versions are then compared across multiple experiments.
- Section 5.3 proposes possible optimizations for the different Partdiff versions and explores their impact.

5.1 Put vs Get

To determine if `MPI_Get` and `MPI_Put` perform differently, multiple tests are conducted. The first tests involve the utilization of the OSU Micro Benchmarks [20], version 5.7. The first set of benchmarks involve a latency test. For these, the default options were used, meaning that the windows were allocated via `MPI_Win_allocate` and the synchronization was done with `MPI_Win_flush`. During the benchmark, OSU tests different message sizes. For each message size, several iterations are done in which one process uses either `MPI_Get` or `MPI_Put`, depending on the specific benchmark, to communicate with a second process. The time measured in these benchmarks includes both the actual communication and the following synchronization.

These benchmarks were executed for both basic RMA communication operations with two different setups, one that involved intra-node communication and one that involved inter-node communication. The benchmarks revealed that inter-node communication incurs higher latencies than intra-node communication, which is to be expected. They also revealed that neither communication operation is superior. The latencies for all 23 different sizes were within 10 - 20μs of each other across both the inter- and intra-node benchmarks, save for a single exception.

The next set of benchmarks involved a bandwidth test. Once again, the default options were used, meaning that the same window allocation and synchronization operations were used for these benchmarks as well. To calculate the bandwidth, multiple back-to-back calls to `MPI_Get` or `MPI_Put` are made, followed by `MPI_Win_flush` in this particular case. The elapsed time until completion and the bytes read or written are then used to determine the bandwidth.

Once again the intra-node benchmark revealed no difference between the two communication operations. The highest bandwidths were achieved while communicating messages of 4096 to 8192 bytes (around 11000MB/s). Both smaller and bigger messages resulted in smaller bandwidths.

Inter-node communication produced differing results though. `MPI_Get` hit its maximum bandwidth at similar message sizes as previously. The maximum bandwidth for inter-node communication was around 2800MB/s. This bandwidth did not decrease as the messages got larger, as it was the case for the intra-node benchmark. `MPI_Put` hit a maximum at a larger message size. For messages of 16384 Bytes and above, the bandwidth plateaued at roughly 2000MB/s. This difference of roughly 800MB/s is observable for all of the nine tested message sizes of 16384 Bytes or larger, indicating that `MPI_Get` achieves better results for large messages when communicating between multiple nodes. In all other cases, both communication operations perform comparably.

To determine whether or not this difference is observable outside of a benchmark environment, the Partdiff versions mentioned previously will be investigated now.

Figure 5.1 pictures the runtime of Partdiff while using Jacobi's method on a matrix with 4105 lines, terminating after 5000 iterations. This demonstrates that the use of `MPI_Get` or `MPI_Put` does not make a noteworthy difference in terms of runtime on the given hardware and with the current MPI implementation, as the differences are within the natural variance. It should be noted that the entire matrix contains 16,851,025 entries, which amounts to a size of 134.8 MB. However the messages sent are only 32.84 kB in size. This is most likely the reason for the increase in runtime between using 72 and 96 processes, as the problem size is too small for higher numbers of processes. To test whether or not bigger messages result in different runtimes and to investigate how well both `MPI_Get` and `MPI_Put` scale in general, the same test has been conducted with different parameters while also increasing the size of the matrix proportionally to the number of processes used.

As opposed to the previous experiment that utilizes strong scaling (the total problem size is fixed), the next experiment features weak scaling, for which the problem size scales

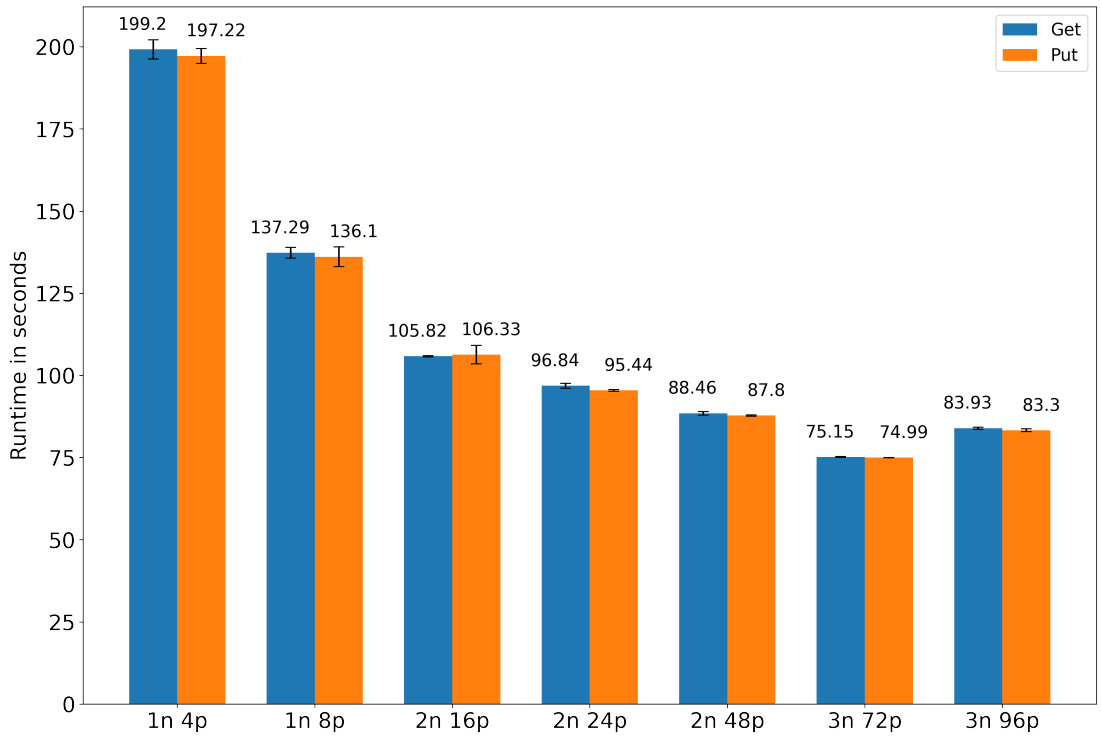


Figure 5.1: Runtimes of Partdiff using the Jacobi method with 4105 lines and 5000 iterations (using n nodes and p processes per node)

with the number of processes, as each process is assigned a fixed problem size. For this particular experiment, every local matrix of a process contains roughly 1 GB of data or 125,000,000 entries respectively. It should be noted that increasing the overall problem size also increases the message size. The message sizes vary from 179 kB while using 4 processes to 876 kB with 96 processes. The exact problem sizes for each number of processes are displayed in Table 5.1. Conducting this experiment with the initial versions of RMA-driven Partdiff resulted in the runtimes shown in Figure 5.2.

As with the strong scaling experiments, `MPI_Put` and `MPI_Get` achieve similar results. The only irregularities are the results while using 72 and 96 processes respectively. For these, using `MPI_Put` achieves faster runtimes. The differences in the other runtimes are within natural variance.

no. of processes	interlines	entries	size of one matrix in GB
1	1396	124,925,329	1.00
4	2794	500,014,321	4.00
8	3951	999,634,689	8.00
16	5588	1,999,252,369	15.99
24	6844	2,998,767,121	23.99
48	9679	5,997,108,481	47.98
72	11855	8,996,332,801	71.97
96	13689	11,994,849,441	95.96

Table 5.1: Problem sizes used in the weak scaling experiments

Both the results from the initial Partdiff versions and the benchmark results seem to indicate that neither `MPI_Put` nor `MPI_Get` grants a performance advantage when compared to each other. The observed difference in inter-node communication bandwidth does not affect Partdiff either. This might be due to the fact that inter-node communication rarely takes place, as only the first and last process of a node have to send messages between different nodes at most. So while it appears that there is no performance difference between the two operations, it should be noted that they cannot be used interchangeably without certain repercussions.

Consider the following example: one process, called rank 0 now, wants to communicate with another process called rank 1. The synchronization mode used is the passive mode. Rank 0 wants to access a certain variable that rank 1 contains. This scenario presents the user with two options. Either rank 0 uses `MPI_Get` to access the memory on its own, granted that the memory was exposed in a window previously, or rank 1 uses `MPI_Put` to write the variable's content into previously exposed memory of rank 0.

When going for the first option, rank 0 knows exactly when the communication is complete, as it has to call `MPI_Win_unlock` to end its access epoch. Once the call to this function returns, the communication is complete and rank 0 can use the value read from rank 1's memory.

When going for the second option, a problem regarding the synchronization arises. Rank 1 starts an access epoch, uses `MPI_Put` to write the value into the window of rank 0 and then ends the access epoch. Meanwhile however, rank 0 has no way of knowing when this operation is complete, as it is not involved in any of the synchronization. Assume that this example is extended into a loop. Rank 0 can never determine whether or not rank 1 executed a call to `MPI_Put`, as rank 1 does not necessarily write a different value into the window during each iteration. Therefore rank 0 cannot determine if any communication has taken place even if the value is monitored constantly. If rank 0 had to act on the value sent in each iteration in any way, it could not do so without further synchronization. For example, rank 1 could send a message to rank 0 after the call to `MPI_Win_unlock` has returned. This would however add an extra message to the entire

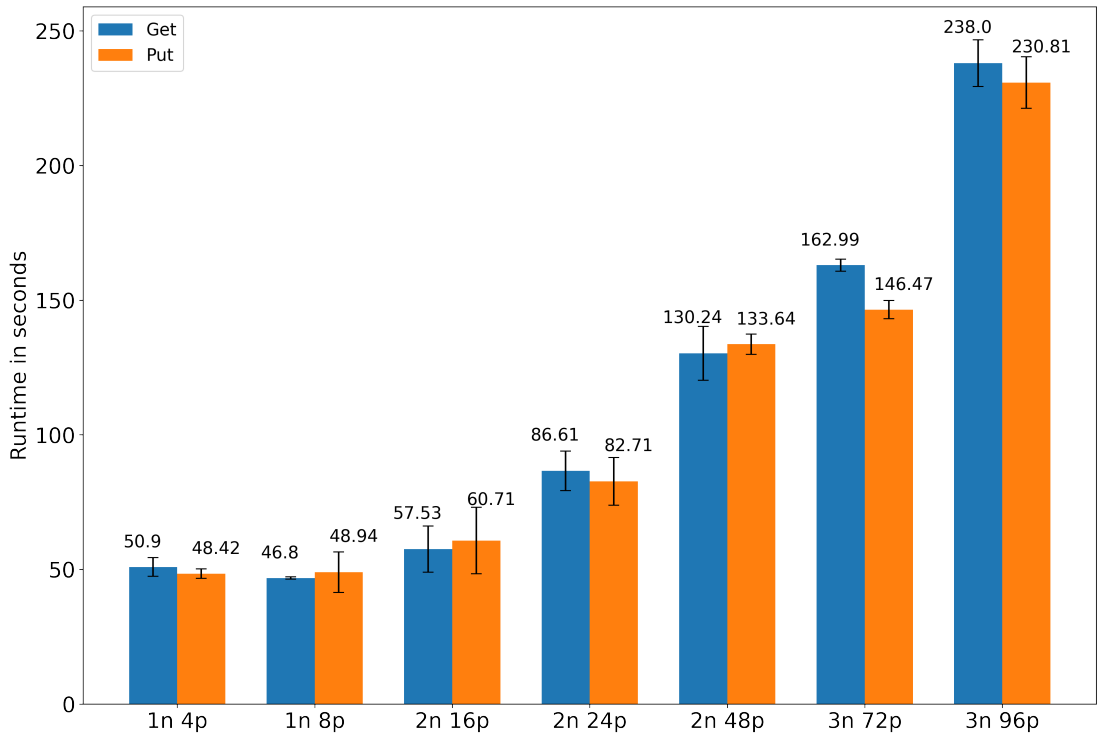


Figure 5.2: Runtimes of Partdiff using the Jacobi method with weak scaling and 100 iterations (using n nodes and p processes per node)

communication when compared to the usage of `MPI_Get`.

This problem also affects the Gauss-Seidel implementation, given that passive synchronization mode is used, which is why future versions of Partdiff will utilize `MPI_Get` over `MPI_Put` when passive mode is used, as it requires one message less per communication step.

To conclude, while neither communication operation outperforms the other one, using `MPI_Put` is not viable in some scenarios involving passive target synchronization, as each communication in such a scenario would need to be accompanied by additional synchronization, which is why `MPI_Get` is used instead in these events. For other scenarios than the one described, including the usage of the other synchronization modes for example, the communication operations can be used interchangeably, given that the memory which is either read or written to is managed accordingly.

5.2 Comparing the different synchronization modes

The next step is the implementation of Partdiff with each synchronization mode, resulting in the first three RMA-driven Partdiff versions. All windows are allocated using `MPI_Win_create`. The communication and synchronization of Jacobi's method was comparatively easy to implement, as it only required to swap the sends from the P2P version to `MPI_Put`

or `MPI_Get` while removing all receives and also adding the appropriate synchronization functions.

The implementation of the Gauss-Seidel method has proven to be more challenging, with the general active target synchronization being the exception. As previously explained in Section 4.2.2, the computing processes are not in the same iteration at all times. Due to this, using active target synchronization properly becomes difficult. Under normal circumstances, `MPI_Win_Fence` is used to start or end an epoch respectively. The function call to end this epoch entails a global barrier synchronization. Considering that every process is in a different iteration, using `MPI_Win_Fence` before and after every communication operation is highly inefficient, as it forces the first process to wait for the completion of the last process before it can continue its calculations. In order to prevent this from happening, `MPI_Win_Fence` is called once before the first iteration starts to begin an epoch. The call to close the epoch is made after the last iteration is finished. Using the fences as described leads to every communication operation happening in the same epoch.

However, due to the special usage of the fences, synchronizing the processes has to be done differently. In order to test whether or not a communication call has completed, the request-based version of the call is used, in this case `MPI_Rget`. As described in Section 3.3, these calls have a request object attached to them, which allows the user to test or wait for the completion of said operation. While this solved the problem of knowing when the communication is completed, another problem regarding the proper synchronization of the processes remained.

Consider the following example: the first process (rank 0) reaches the end of the first iteration and needs to access the first line of rank 1. Using `MPI_Rget`, rank 0 can fetch said halo line, finish computing the first iteration and begin the second one. Suppose rank 0 reaches the point at which it needs to access rank 1's halo line again before rank 1 finishes the computation of its first line. In this scenario, rank 0 would access the unfinished halo line and continue its computation with erroneous values.

In order to inhibit such behaviour, the neighbouring processes have to communicate information about their current state. In this version of Partdiff, each process creates one new window per halo line, in which it exposes a single integer, used as a flag. This flag contains either the value 0 if the halo line is not ready to be accessed yet or a 1 once it is ready. These flags are initialized with 0 and once a process finishes calculating the results for its own halo line, the value is then set to 1. Meanwhile, the neighbouring process accesses the flag corresponding to the specific halo line once said process reaches the point in computation at which it needs access to the line. If the flag is set to 0, the neighbouring process enters a loop in which it continuously reads that flag until it contains the value 1. Only then will the process read the halo line via `MPI_Rget` and wait for the completion. After `MPI_Rget` is completed, the accessing process overwrites the flag with the value 0 via `MPI_Put`, signaling the completion of the access and ensuring that said process cannot access the line again until the accessed process updates the flag.

While this approach achieves the correct results, it should be noted that the usage of request based RMA operations is only valid in passive target epoch, according to [7, Chapter 11.3.5]. Therefore, this version of Partdiff is not valid and it is not guaranteed

to work with other MPI implementations. However, using the request-based operations was the only option to efficiently implement the Gauss-Seidel method using active target synchronization. Due to this and the fact that the calculations are correct regardless, the Partdiff version that is described above will still be considered in some of the following experiments involving the Gauss-Seidel method, though using active target synchronization for similar algorithms is not advised, as the other synchronization modes are better suited for such.

While passive synchronization is better suited for the implementation of the Gauss-Seidel method compared to active target synchronization, it still suffers from some similar problems. Communication and synchronization are simpler, as the origin process is the only one to issue any calls for that. This means that once a process needs to access a halo line, it can lock the appropriate window, issue a call to `MPI_Get` and end the access epoch by unlocking the window. As mentioned, the communication will be complete once the call to `MPI_Win_unlock` has returned. However, using passive target synchronization alone does not rule out the potential race condition that was described above. To solve that problem, the same strategy involving the flags is used once again, the only difference being that the use of request-based operations is not necessary, as locking the window and then using `MPI_Win_flush` where appropriate offers the same functionality.

General active target synchronization turned out to be the most suitable synchronization mode for an algorithm such as the Gauss-Seidel method due to its fine-grained synchronization approach. As explained in Section 3.5.2, the origin process starts its access epoch by calling `MPI_Win_start`. After that, any necessary communication operation can be issued. To close the epoch and therefore guarantee that the communication is complete, the `MPI_Win_complete` function is called. Since the target process has to explicitly open an exposure epoch via `MPI_Win_post`, the previously described race condition is not possible anymore, as the target process only starts its exposure epoch for a halo line once the computation for the line is complete. That access epoch ends shortly before the start of the next access epoch for the next iteration. This guarantees that the origin process can only access any halo line once per iteration and therefore eliminates any race conditions.

These three versions of Partdiff were then used in experiments similar to the previous ones. Each version was subjected to eight distinct experiments, four of them for each method. These experiments are divided by the termination criterion, either a set number of iterations or a precision that has to be surpassed. Then they are divided again into one using weak scaling and the other using strong scaling. For weak scaling, the same problem sizes as for the previous experiment apply. In order to be able to properly compare the results of the experiments with a differing termination criterion, the precision was chosen in such a way that the necessary iterations to reach that precision are equal to the iterations defined as the termination criterion for the other experiment. During the weak scaling experiments, these precisions vary between experiments with differing numbers of processes, as the underlying matrix changes, which leads to different precisions being

necessary to require 100 iterations.

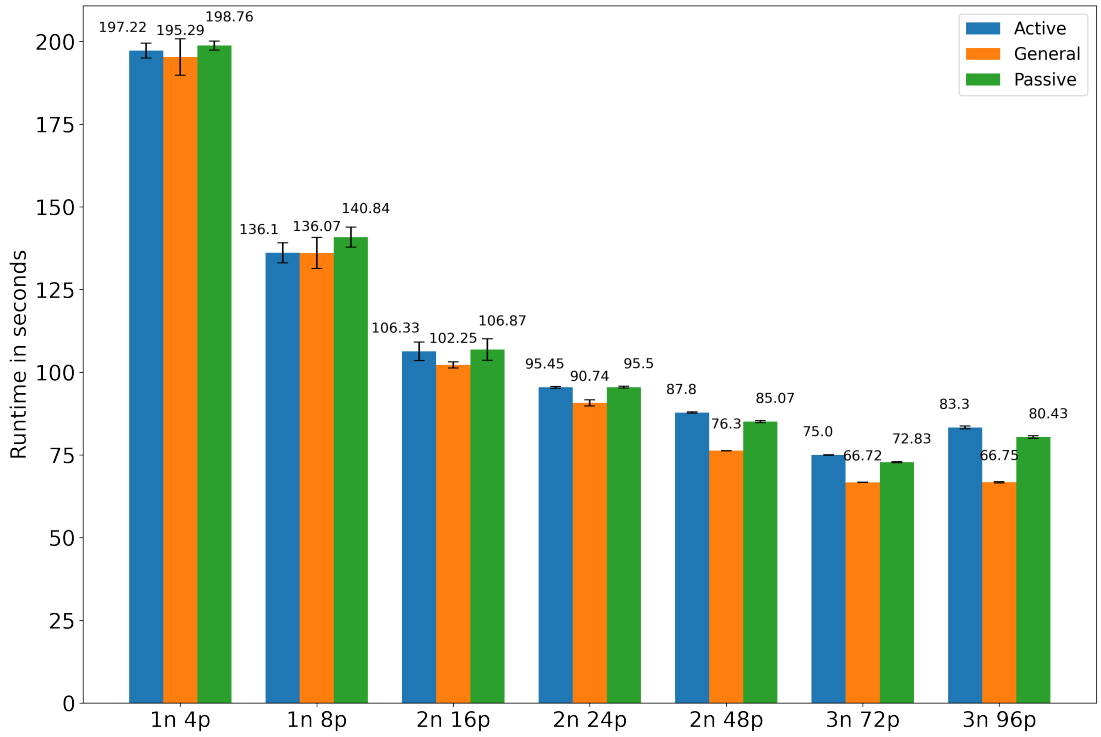


Figure 5.3: Runtimes of the first Partdiff versions using Jacobi's method with 4105 lines and 5000 iterations (using n nodes and p processes per node)

The results in Figure 5.3 show that the version using general active target synchronization is the fastest out of the three for Jacobi's method. Increasing the number of nodes also led to an increase in the difference between general mode and the rest. Active and passive target synchronization achieve very similar results, although the version using passive target synchronization appears to scale better, as the difference in runtime between these two increases with the number of processes used. However, that difference is comparatively small with roughly 3 seconds.

These results were expected, because the use of active target synchronization entails implicit barriers, which will slow down the runtime for large numbers of processes, as the waiting time at these barriers increases since every process has to reach it before continuation. The version using passive target synchronization also necessitates the use of explicit barriers, because the fact that only the origin process takes part in synchronization and communication leads to a race condition in which multiple processes can be in different iterations and access halo lines that were not meant to be accessed at that point in time. General active target synchronization does not suffer from these problems, as the blocking synchronization calls are not global and therefore do not impact the runtime as much due to decreased waiting times. Since both the origin and the target process are involved in the synchronization, the use of any explicit barriers is not necessary either, leading to the fastest runtime of the three versions.

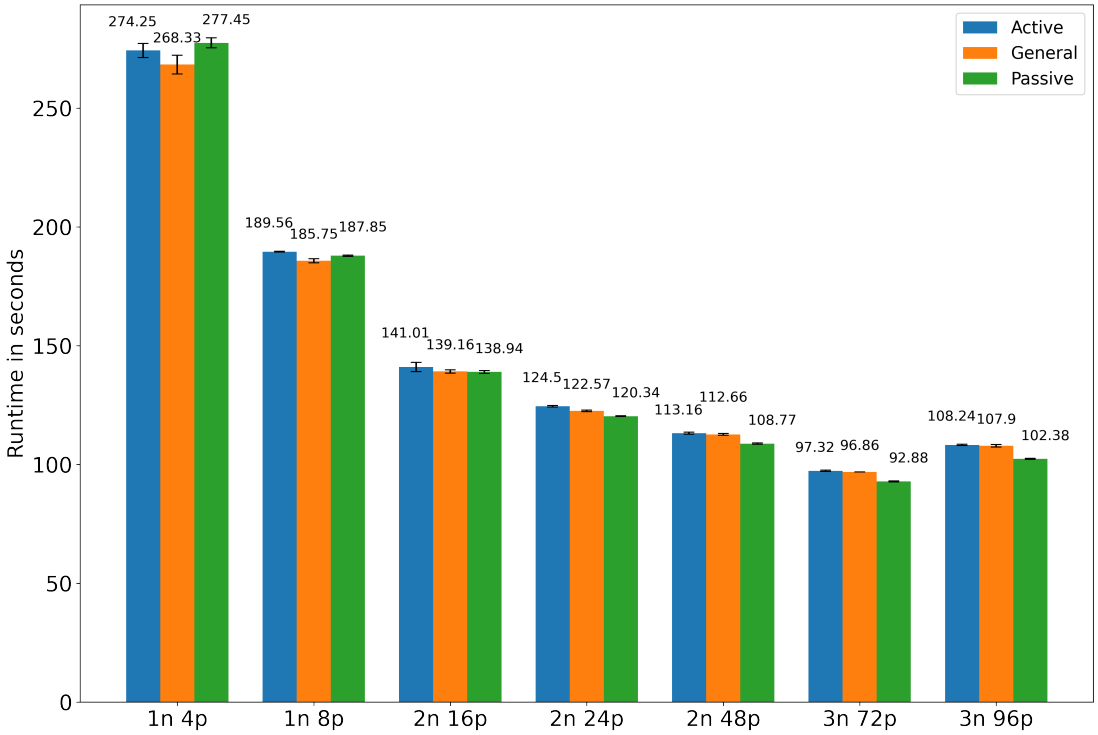


Figure 5.4: Runtimes of the first Partdiff versions using Jacobi's method with 4105 lines and a target precision of 7.0637e-5 (using n nodes and p processes per node)

However, this trend does not continue in the next experiment. The results depicted in Figure 5.4 show that using precision as the termination criterion increases the overall runtime. This was to be expected since a collective operation (`MPI_Allreduce` for this case in particular) is needed to determine the residuum after each iteration. That collective operation is the only difference between using precision or a number of iterations as the termination criterion for all three versions. Regardless of that, general active target synchronization is not the fastest in this experiment. Only while using less than 16 processes it remains the fastest, afterwards the runtimes of general active target synchronization are on par with active target synchronization, while passive target synchronization scales the best and achieves the lowest runtimes of the three versions at higher numbers of processes. It should be noted that the aforementioned barriers for the version using passive synchronization are not necessary if precision is the termination criterion. Instead, a call to `MPI_Allreduce` is applied. While collective communication operations are not required to synchronize [7, Chapter 5.1], the results of all previous and all following experiments were always correct. Whether this happens because all tested MPI implementations synchronize all processes as a side effect of invoking `MPI_Allreduce` or if it happens because the Partdiff implementation is robust enough cannot be determined at this time, because testing this would require many experiments with many different MPI implementations, which is out of scope.

It is most likely due to this difference that the passive target version achieves the best results at higher numbers of processes, as the use of the collective communication operation did not make any of the synchronization calls of the other two versions obsolete.

It should be noted that the increase in runtime for 96 processes compared to 72 processes is once again most likely due to the problem size being to small for such a high number of processes, much like in Figure 5.1.

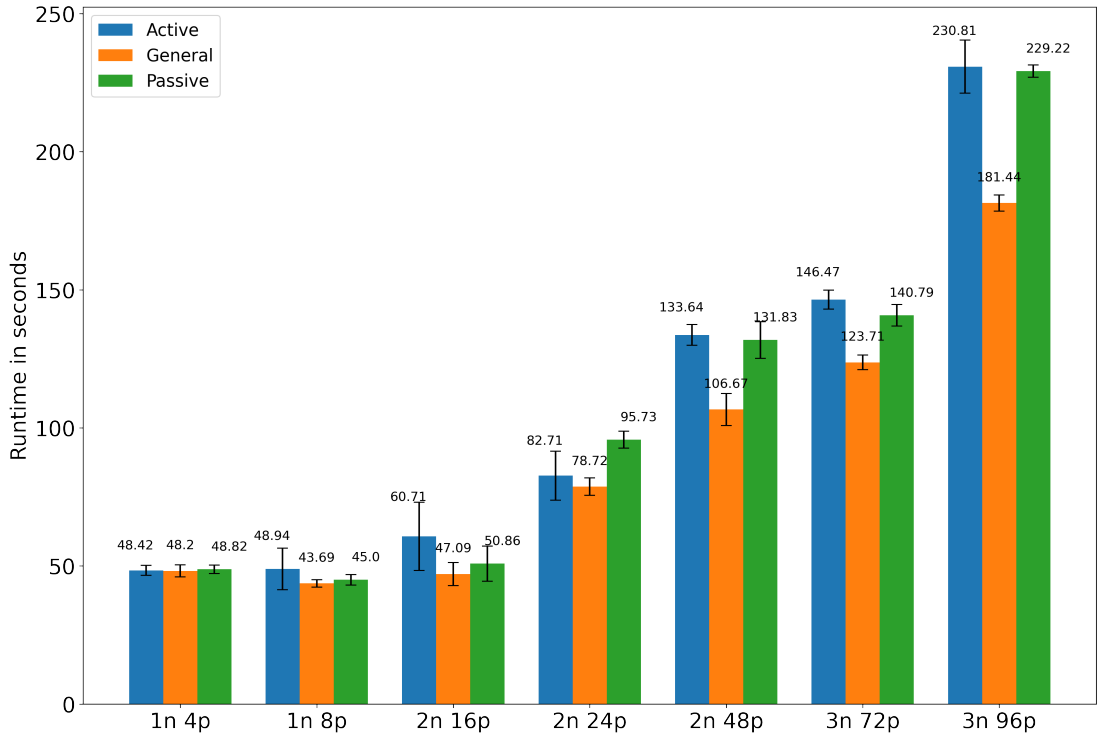


Figure 5.5: Runtimes of the first Partdiff versions using Jacobi's method with weak scaling and 100 iterations (using n nodes and p processes per node)

The results of the strong scaling experiments that terminated after a fixed number of iterations are similar to the results of the same experiment with weak scaling (Figure 5.5). General active target synchronization performs the best, with the absolute and also the relative difference increasing alongside the number of processes, indicating that general active target synchronization works best for both small and large problem sizes while using Jacobi's method and using iterations as the termination criterion. Once again the other two modes are almost equal, with a few exceptions while using 16 or 24 processes.

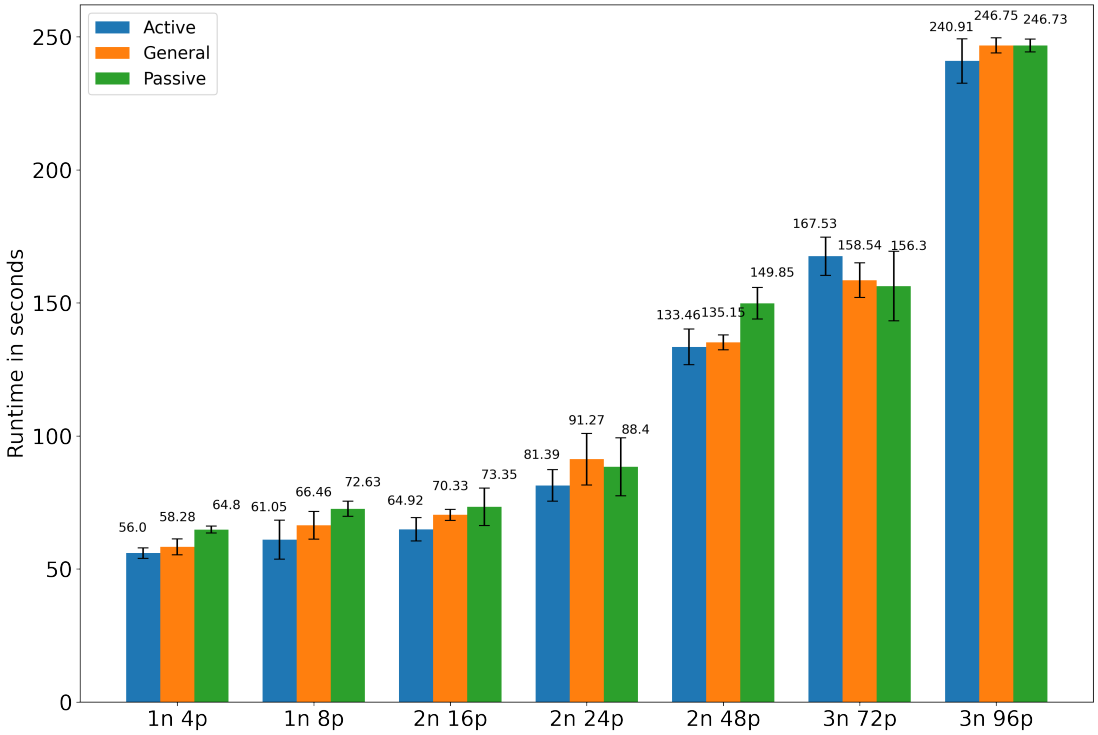


Figure 5.6: Runtimes of the first Partdiff versions using Jacobi's method with weak scaling and a varying precision that requires 100 iterations to reach (using n nodes and p processes per node)

The results of the weak scaling experiment that uses precision as the termination criterion differ from the previous ones though, as displayed in Figure 5.6. The runtimes that were achieved indicate that active target synchronization performs the best on large problem sizes with only one exception. The strong scaling counterpart of this experiment placed passive target synchronization as the fastest one. The other two modes achieved similar results while only being marginally slower than the passive mode. Considering that the corresponding standard deviations of the respective results are quite high compared to the absolute differences of the mean values, repeating the experiment with a much higher number of individual measurements would be advised to test whether or not these results repeat themselves or if the results turn out to be closer to each other.

Next, the results of the experiments with the Gauss-Seidel method will be inspected.

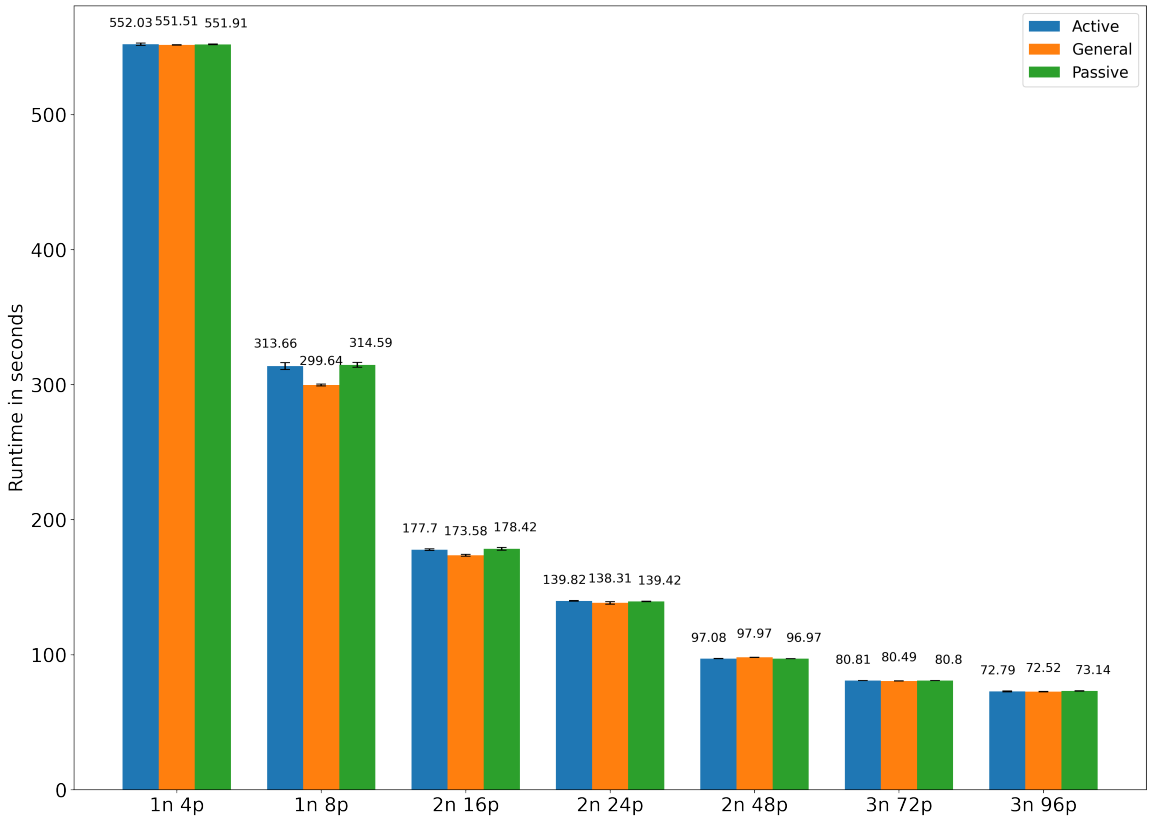


Figure 5.7: Runtimes of the first Partdiff versions using the Gauss-Seidel method with 4105 lines and 5000 iterations

Both experiments (Figure 5.7 and Figure 5.8) produce similar results, as the three synchronization modes achieve runtimes that are almost equal to each other, with only very few and small exceptions (like the runtime for 8 processes in Figure 5.7). Once again, the runtimes for using precision as the termination criterion surpass their respective counterparts in the other experiment. It should also be noted that the number of iterations needed to reach the specified precision is 5000 for the sequential version of Partdiff, but due to the specific implementation for the parallel Gauss-Seidel method, the parallel Partdiff versions execute a few extra iterations after reaching the targeted precision, as explained in Section 4.2.2. The number of extra iterations also increases with the number of processes used, as the maximum possible difference in iterations between the first and the last ranked process increases. As an example, two specific measurements of the Partdiff version using general active target synchronization are considered. The first measurement is from the experiment using 4 processes. After terminating, 5005 iterations were completed. Meanwhile, using 96 processes led to Partdiff iterating over the entire matrix 5162 times.

To see whether or not one mode is better suited than the others in regards to the performance, the weak scaling experiments depicted in Figure 5.9 and Figure 5.10 have to be evaluated.

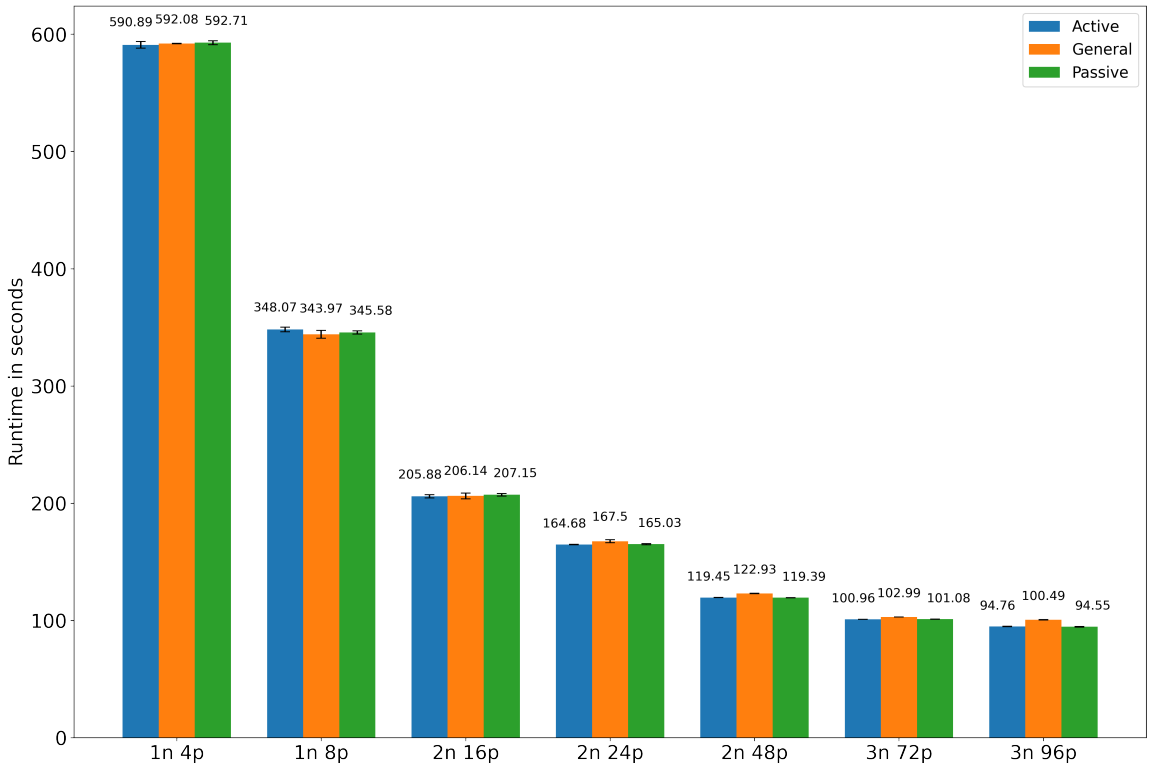


Figure 5.8: Runtimes of the first Partdiff versions using the Gauss-Seidel method with 4105 lines and a target precision of 7.0753e-5

Once again, the results from the two experiments are similar. Although this time the version using passive synchronization seems to perform the worst by 1-2% for every configuration. Even though this difference occurs consistently, it is small enough that it could still be attributed to natural variance. Active and general active target synchronization achieve results that are for the most part almost equal.

Looking at the results from all experiments involving Gauss-Seidel, a preliminary conclusion regarding the best performing synchronization mode can be drawn. All three modes produce very similar results. As previously noted, the specific implementation of the Gauss-Seidel method using active target synchronization is not valid due to the use of request-based one-sided communication operations. Due to this and because active target synchronization only performed equal to the others, it will not be considered anymore for further experiments involving the Gauss-Seidel method. From this point onward, only the passive and the general active target mode will be investigated further for the final implementation of the Gauss-Seidel method.

Regarding the results from the experiments with Jacobi's method, preliminary conclusions can be drawn as well, albeit not as clear cut. For both the strong and the

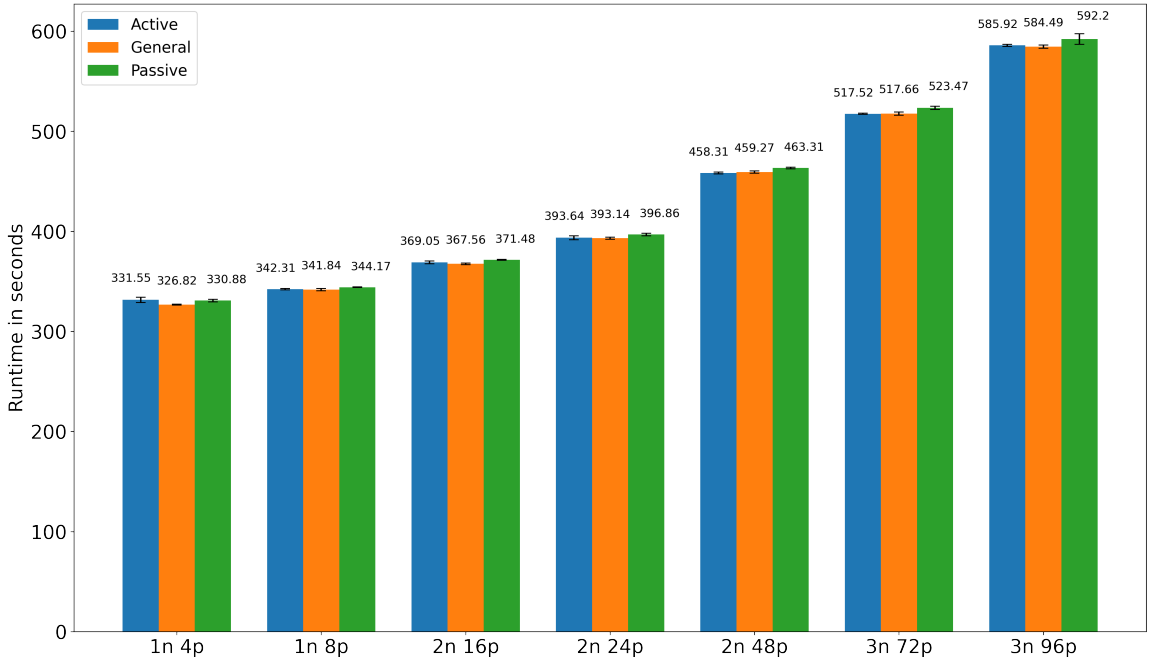


Figure 5.9: Runtimes of the first Partdiff versions using the Gauss-Seidel method with weak scaling and 100 iterations

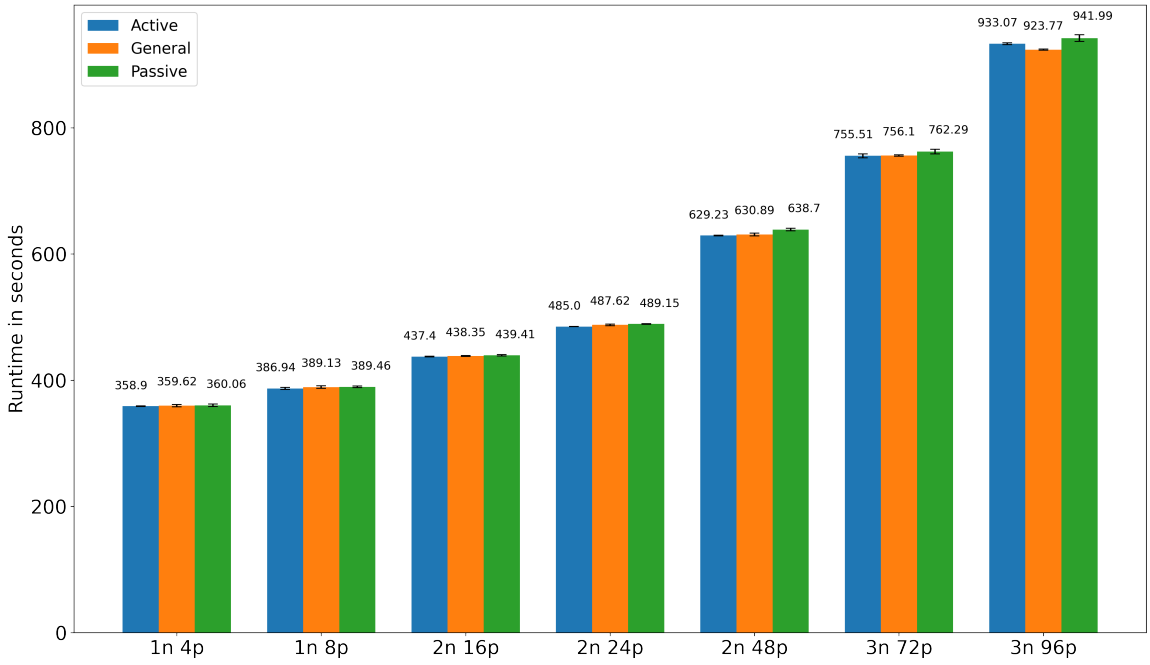


Figure 5.10: Runtimes of the first Partdiff versions using the Gauss-Seidel method with weak and a varying precision that requires 100 iterations to reach

weak scaling experiments that involved terminating after a set number of iterations,

general active target synchronization performed the best, especially with higher numbers of processes. However, the results of the experiments terminating after reaching a targeted precision produced different results. During the strong scaling experiment with a comparatively small problem size, the passive mode achieved the best results, while the results of the weak scaling counterpart saw active target synchronization as the best performing mode, although these results should be considered inconclusive due to previously mentioned reasons. Going forward, all three synchronization modes will still be subjected to different experiments.

5.3 Improving the performance

These first RMA-based versions will now be improved upon through various means, as their implementations are rather naive and pose much room for possible improvements, which is why the aforementioned conclusions are only preliminary.

This section is separated into multiple subsections, each testing a different approach to potentially increase the performance of Partdiff:

- Section 5.3.1 examines a better approach for the calculation of the residuum.
- In Section 5.3.2, the passive version's synchronization outside of the locks is improved.
- Section 5.3.3 investigates the passive version as well. This subsection investigates different possibilities regarding the exact use of `MPI_Win_lock` and `MPI_Win_unlock`.
- The advantages and disadvantages of combining multiple windows into a single one are discussed in Section 5.3.4.
- Section 5.3.5 features an experimental approach to active target synchronization, aimed at reducing the global nature of `MPI_Win_fence` by introducing multiple small communicators instead of one large communicator.
- Section 5.3.6 explores the options of shared memory in MPI RMA.

5.3.1 Residuum calculation

After comparing these initial RMA-based versions to their point-to-point counterpart, it was found that the Gauss-Seidel method with precision as the termination criterion was performing the worst. As an example, the runtime for the general active target version with 16 processes was 206.14 seconds, while the P2P version only needed 166.58 seconds (more thorough comparisons are conducted in Chapter 6). The reason for this difference is the calculation of the residuum. After calculating the residuum for its local matrix,

each process would send that residuum to the last process, which would then locally determine the maximum residuum from these values for a given iteration and inform the first process if the termination criterion has been met. Due to this, the last process served as a bottleneck for the runtime.

To stop this from happening, the load of the communication has to be evenly distributed. In order to achieve this, each process sends its local residuum not to the last process, but to its successor. The successor then compares the residuum of the predecessor to its own local residuum once the iteration over the matrix is completed and sends the higher value to the next rank. Once the last rank receives the residuum from its predecessor, it also compares the received residuum to its own local residuum and then informs the first process, if necessary.

Both the P2P version and the RMA version initially calculated the residuum in the way that is described first, however the performance of the P2P version was not negatively impacted by this, as comparing the runtime of both approaches with both weak and strong scaling shows no meaningful difference (all runtimes differ by 1% at most). The difference for the runtime of the RMA versions was much higher, as the results in Figure 5.11 from a strong scaling experiment show.

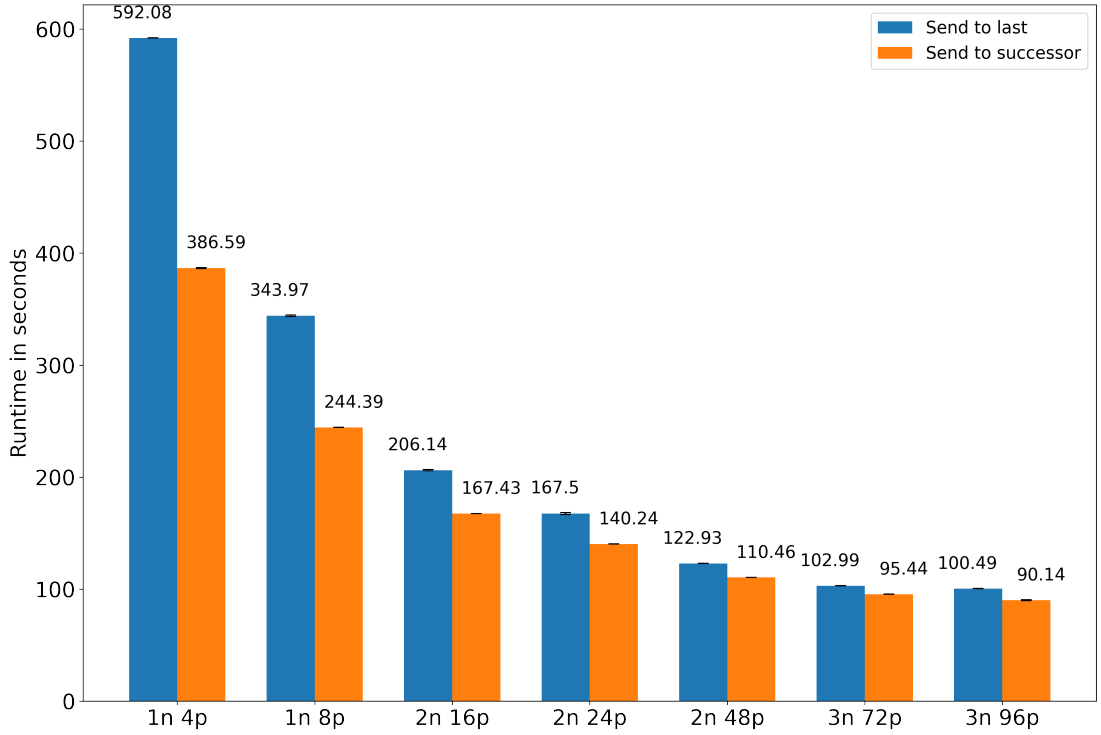


Figure 5.11: Runtimes of the different approaches to residuum calculation using 4105 lines and a target precision of 7.0753e-5

The results of a similar weak scaling experiment reinforce these findings, as the runtimes observed in this experiment differ by at least 120 seconds for any tested number of processes, indicating that the new approach to residuum calculation is superior on both

small and large problem sizes.

Another approach to residuum calculation was the usage of `MPI_Accumulate`. For this, a new window was created that was empty on most processes, excluding the last one. The last process exposed two times the number of overall processes worth of memory in double values. The other processes accumulated their local residuum in one of these entries, depending on their current iteration. Once the last process reaches the end of its iteration, it only has to read the corresponding entry of the window and compare it to its local residuum. Afterwards the first process can be notified if necessary. While this approach yielded better results than the original one, it is still vastly inferior to the previously evaluated approach during which the residuum is always sent to the successor.

Therefore the optimal one-sided version will calculate the residuum during the Gauss-Seidel method by having each rank send the minimum of the residuum of the predecessor and its local residuum to their successor.

5.3.2 Substituting flags

The next improvement, among others, was inspired by [4]. During their implementation of a halo swap with passive target synchronization, they also encountered the problem that passive mode lacks any synchronization at the target process. Their solution to this problem was to send an empty message via point-to-point communication once the one-sided communication operation was finished.

To determine whether this approach is superior to the method used for Partdiff, a version using point-to-point messages instead of flags was created. This explicit synchronization was necessary for the Gauss-Seidel method, as the absence of any synchronization led to possible race conditions in which a process could potentially read the same halo line twice for different iterations. In regards to performance, no version was superior to the other in both a strong and a weak scaling experiment. However, using P2P messages is still considered as the better alternative, as it increases readability substantially. A comparison of both approaches is displayed in Listing 5.1.

```
1 //old version trying to read the last line of the
    ↪ predecessor:
2 //check if the last line is ready to be used by this
    ↪ process, wait if it's not
3 int ready;
4 MPI_Win_lock(MPI_LOCK_EXCLUSIVE, pre, MPI_MODE_NOCHECK,
    ↪ winLastLineReady);
5 MPI_Get(&ready, 1, MPI_INT, pre, 0, 1, MPI_INT,
    ↪ winLastLineReady);
6 MPI_Win_flush_local(pre, winLastLineReady);
7 while(ready == 0){
8     MPI_Get(&ready, 1, MPI_INT, pre, 0, 1, MPI_INT,
```

```
        ↪ winLastLineReady);
 9      MPI_Win_flush_local(pre, winLastLineReady);
10  }
11
12  //once ready, get the line and inform the other process
       ↪ that the last line can now be overwritten again
13  MPI_Win_lock(MPI_LOCK_EXCLUSIVE, pre, MPI_MODE_NOCHECK,
       ↪ winLastLine);
14  MPI_Get(bufferForLineFromPrecedingRank, N+1, MPI_DOUBLE,
       ↪ pre, 0, N+1, MPI_DOUBLE, winLastLine);
15  ready = 0;
16  MPI_Put(&ready, 1, MPI_INT, pre, 0, 1, MPI_INT,
       ↪ winLastLineReady);
17  MPI_Win_unlock(pre, winLastLine);
18  MPI_Win_unlock(pre, winLastLineReady);
19
20  //new version with P2P:
21  //Once this message is received, the desired halo line is
       ↪ ready to be read
22  MPI_Recv(NULL, 0, MPI_INT, pre, results->stat_iteration,
       ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE);
23  MPI_Win_lock(MPI_LOCK_EXCLUSIVE, pre, MPI_MODE_NOCHECK,
       ↪ winLastLine);
24  MPI_Get(bufferForLineFromPrecedingRank, N+1, MPI_DOUBLE,
       ↪ pre, 0, N+1, MPI_DOUBLE, winLastLine);
25  MPI_Win_unlock(pre, winLastLine);
```

Listing 5.1: Using P2P messages for necessary synchronization at the target process instead of flags

For the approach using flags, a process first queried the state of the flag for the respective halo line (lines 3-6). If the flag is set to `0`, the querying process then enters a loop (lines 7-10) that can only be exited once the flag is set to `1` by the neighbouring process. Once this section is completed, the querying process can then access the halo line (lines 13, 14, 17), but it also has to reset the flag to `0` (lines 15, 16, 18) in order to guarantee that this particular halo line will not be accessed before the neighbouring process is finished with said line.

The approach using P2P messages is considerably shorter. Instead of having to repeatedly access the flags, the origin process instead waits for a message from the neighbour via `MPI_Recv` (line 22). Since this function blocks until the corresponding message is sent by the neighbouring target process, the line cannot be accessed before the target process is finished with said line. After receiving the message, the halo line can be accessed just as before (lines 23-25).

Another approach to eliminate the use of flags was to lock the windows locally if a

process has not finished the calculations of its own halo lines. This would block any remote accesses until the local lock is released, at which point the remote access would take place. This approach was not successful though, as this did not eliminate the possibility of the aforementioned race condition, it only stopped the access of incomplete halo lines.

Therefore, any previous implementations that relied on the flags now use point-to-point messages to greatly increase readability without sacrificing performance.

5.3.3 Different approaches to locking windows

Another key aspect of the design of the passive mode version described in [4] was the proper usage of the MPI locks. During their experiments they came to the conclusion that acquiring all locks at the start with `MPI_Win_lock_all` and only releasing these locks at the very end in the finalization procedure yields better results compared to locking and unlocking a window repeatedly. Usually calling `MPI_Win_unlock` and therefore ending an epoch is necessary to guarantee completion of the communication operations issued, but this need can be circumvented by using `MPI_Win_flush` to flush any outstanding RMA operations.

This approach to locking the windows was tested on Partdiff in the implementation of the Gauss-Seidel method. A strong scaling experiment revealed that neither version was superior to the other. The runtimes for both did not differ outside of a few seconds which can be attributed to natural variance. The same can be said for a subsequent weak scaling experiment.

There are multiple possible reasons for the absence of a performance difference. For example, the MPI implementation used in the aforementioned paper is Cray MPICH v7.5.5. They also executed it on different hardware, starting their experiments with no less than 128 processes. The underlying algorithm they used to test RMA involves many more neighbours per process than any process in Partdiff as well, resulting in potentially skipping many more locking and unlocking procedures compared to Partdiff. Also, Partdiff is written in C, whereas their modeling framework in use is written in Fortran 2003, which could also lead to potential differences.

In conclusion, the different approach to locking the windows did not lead to a performance advantage for Partdiff, most likely due to the reasons given above, but this different approach is certainly worth investigating for any RMA-based program, considering the potentially tremendous difference in runtime, as described in the aforementioned paper.

5.3.4 Combining multiple windows

The next attempt to increase performance was to reduce the number of windows. Consider the following example: while using Jacobi's method with fences, each process has to issue two calls to `MPI_Win_fence`, one for each window, as the halo lines are stored in two different windows. By decreasing the number of windows, the calls to `MPI_Win_fence`

would also decrease by 50%, resulting in fewer global synchronization calls. Decreasing the number of barriers should lead to a substantial increase in performance, however the following experiment shows that this is not the case.

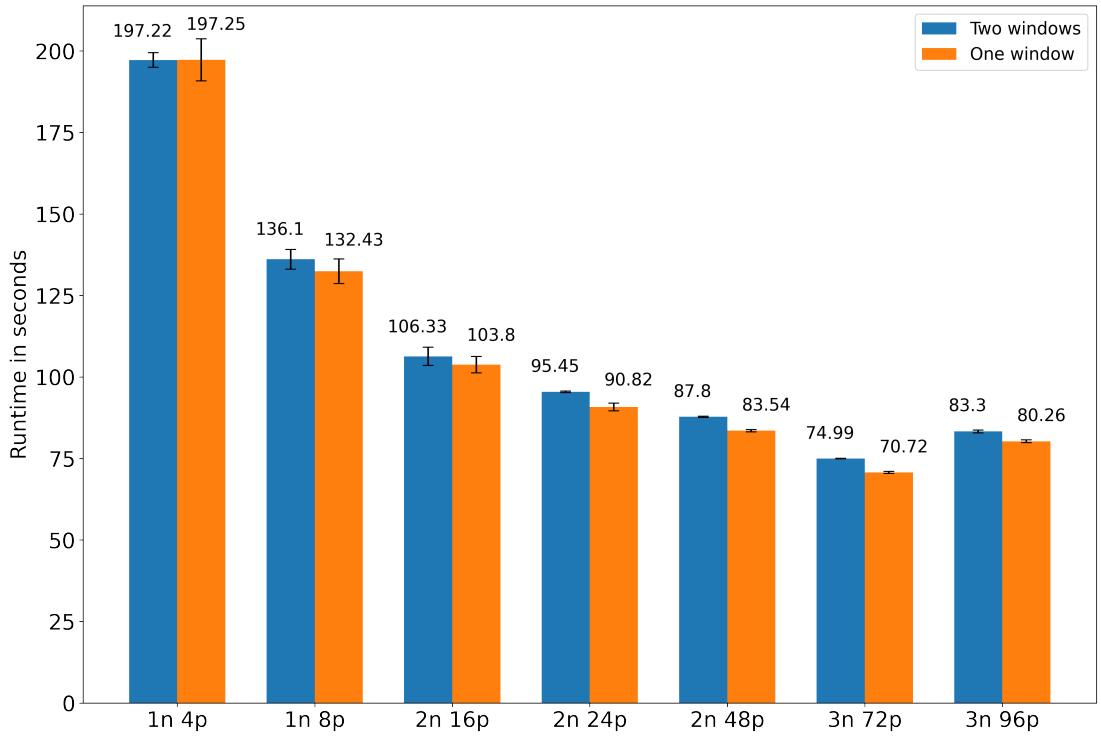


Figure 5.12: Runtimes of the different approaches to the window setup using 4105 lines and 5000 iterations

While using just one window instead of two indeed seems to increase the performance, especially with rising numbers of processes, the impact is comparatively small. But considering that the performance increase is not the only advantage, it is still an improvement nonetheless, as using only one window minimizes the amount of necessary calls to `MPI_Win_create` as well. This is most likely not going to heavily impact the runtime of Partdiff, but on algorithms that require more than 2 lines of a matrix to be accessible it is likely to be best practice, as it reduces code size while also presumably minimizing setup time, alongside the reduction of necessary synchronization calls. Whether or not using only a single window increases readability has to be decided on a case by case basis though. For some scenarios and users, substituting the long, but descriptive names for a high number of windows with a single window, enabling and enhancing the array-like access of RMA communication operations, can increase readability by deflating the code. But this might not be the case for every scenario and user.

5.3.5 Splitting the communicator

In an attempt to decrease the amount of global synchronization operations while using `MPI_Win_fence`, a new implementation for Jacobi's method was created that does not communicate over one global communicator, but over many small ones. One communicator was created for every process. Each communicator contained up to three processes, one distinct process and its neighbours. For example, if Partdiff is run with 16 processes, 16 communicators are created before the first iteration begins. The first one contains rank 0 and rank 1, the second one contains rank 0, rank 1 and rank 2, the third one contains rank 1, rank 2 and rank 3 and so on. In using this, the calls to `MPI_Win_fence` do not act as global barriers anymore, but as smaller ones that involve only the neighbouring processes. This approach does however increase the overall number of `MPI_Win_fence` calls that a process has to issue, but since they are not global anymore, this might still increase the overall performance. Splitting the communicator in the way that is described above therefore turns the global barriers into something akin to the synchronization mechanisms of general active target synchronization for this algorithm.

The usual strong and weak scaling experiments that use up to 96 processes on 3 nodes did not show any conclusive differences between the approach with the many small communicators and the approach with the global communicator. An experiment with 200 processes using Jacobi's method and terminating after 5000 iterations on a square matrix with 24009 lines however resulted in a runtime of 1310.16 seconds (standard deviation: 5.59) for the version with the split communicator, while the same version without the split needed 1398.64 seconds (standard deviation: 4.70) to terminate. Considering this result and that the runtimes on smaller numbers of processes were comparable, splitting the communicator does improve the performance. It should be noted that this improvement comes at the cost of readability, as creating multiple windows for different overlapping communicators involves the creation of dummy windows, as depicted in Listing 5.2.

```
1  for(int x = 0; x < worldSize; x++)
2  {
3      if(allSmallComms[x] != MPI_COMM_NULL)
4      {
5          if(rank == x)
6          {
7              MPI_Win_create(NULL, 0, 1, info, ownComm,
                   ↪ &winHaloLines);
8          }
9          else if(rank == x-1)
10         {
11             MPI_Win_create(haloLines, sizeof(double) * 2 *
                   ↪ (N+1), sizeof(double) * (N+1), info,
                   ↪ successorComm, &winPost);
12         }
13         else
14         {
```

```
15                MPI_Win_create(haloLines, sizeof(double) * 2 *
                     ↪ (N+1), sizeof(double) * (N+1), info,
                     ↪ predecessorComm, &winPre);
16            }
17        }
18    }
```

Listing 5.2: Creating windows for multiple overlapping communicators

The array `allSmallComms` holds `worldSize` entries of communicators. If the process that is currently executing this loop is not part of the communicator in one of the entries, that entry holds the value `MPI_COMM_NULL`. If that is the case for this specific process, nothing has to be done. If this process is part of the communicator, a window has to be created. For this, three cases have to be differentiated. Before the different cases are explained though, it should be mentioned that `MPI_Win_create` is a collective call over the entire communicator specified in the second to last argument.

The first case is that the communicator is the process's *own* communicator. For example, the process rank 1 would consider the entry at `allSmallComms[1]` its own communicator, as this communicator only contains rank 1 and its neighbours. In that case, rank 1 would create a window without exposing any of its memory, the only important thing is the last argument, a window variable. This variable will be used to access the windows of its neighbours in this specific communicator. Meanwhile, the neighbouring processes are also executing this loop. If rank 1 is currently in the iteration where $x = 1$, then rank 0 and rank 2 will also be in that iteration, as they are both part of the communicator in `allSmallComms[1]`.

For rank 0, it is the communicator of the successor, hence passing the argument `successorComm` when creating the window, while rank 2 passes `predecessorComm`. Both of these processes do expose memory during their window creation. Due to this, rank 1 can access that memory by accessing its window called `winHaloLines`, either reading the halo lines from its predecessor or its successor. Meanwhile, both rank 0 and rank 2 will never access `winPost` or `winPre` respectively. These window variable were only defined because `MPI_Win_create` requires the last argument to be non-null.

Note that the number of necessary dummy windows rises with each window created for these communicators. Every new window would spawn two new dummy windows in this particular example.

In conclusion, while this approach does increase the performance if a high number of processes is used, the example given above shows that splitting the communicator into many small ones that have to overlap is most likely not feasible for most programs outside of Partdiff, as the window creation makes the code increasingly unreadable the more windows are needed or the more processes are part of these split communicators. For example, consider a data distribution that leads to processes having four neighbours. In that case, each communicator would contain five processes, which would lead to five different cases in the loop while each process has to define 5 dummy window variables per window.

5.3.6 Shared memory windows

The last possible improvement to the RMA versions of Partdiff is the introduction of shared memory windows. These windows allow for the creation of a shared memory segment that enables a process to operate on memory of other processes via usual load and store accesses. To create these windows, `MPI_Win_allocate_shared` is called which then allocates the memory specified in the arguments. Before calling this function though, it has to be ensured that the processes of the communicator that is passed as an argument to the window allocation function are able to create a shared memory segment. To guarantee this, the `MPI_Comm_split_type` function is called beforehand, splitting the initially global communicator if necessary.

After creating the windows, a process can query the process-local address of the memory segments of any other process in its communicator by using the `MPI_Win_shared_query` function, which returns a base pointer for that memory region, the size of the region and a displacement unit. This allows for the use of C pointer arithmetic to access any entry inside this shared memory region. However, these windows can solely be accessed by the processes in the shared memory capable communicator. If any communication has to happen between processes that do not have the ability to create shared memory between them, other global windows or P2P communication are necessary. An example for such a scenario would be the usage of two different nodes. The communication between processes on different nodes would necessitate means of communication other than shared memory.

Any following Partdiff version in this section will communicate via RMA to exchange their halo lines. Once again, multiple versions were created with the different synchronization modes to investigate whether or not one is superior to the others in terms of performance. They were then subjected to the same strong and weak scaling experiments that were used throughout this thesis. Considering that the communication outside of these shared windows happens only between the last process of one shared communicator and the first of another one, the assumption that the usage of different synchronization modes only impacts the runtime minimally, if at all, seems reasonable. While the experiments featuring the Gauss-Seidel method indeed show only minimal differences, the experiments involving Jacobi's method produced results that show considerable differences between the three modes.

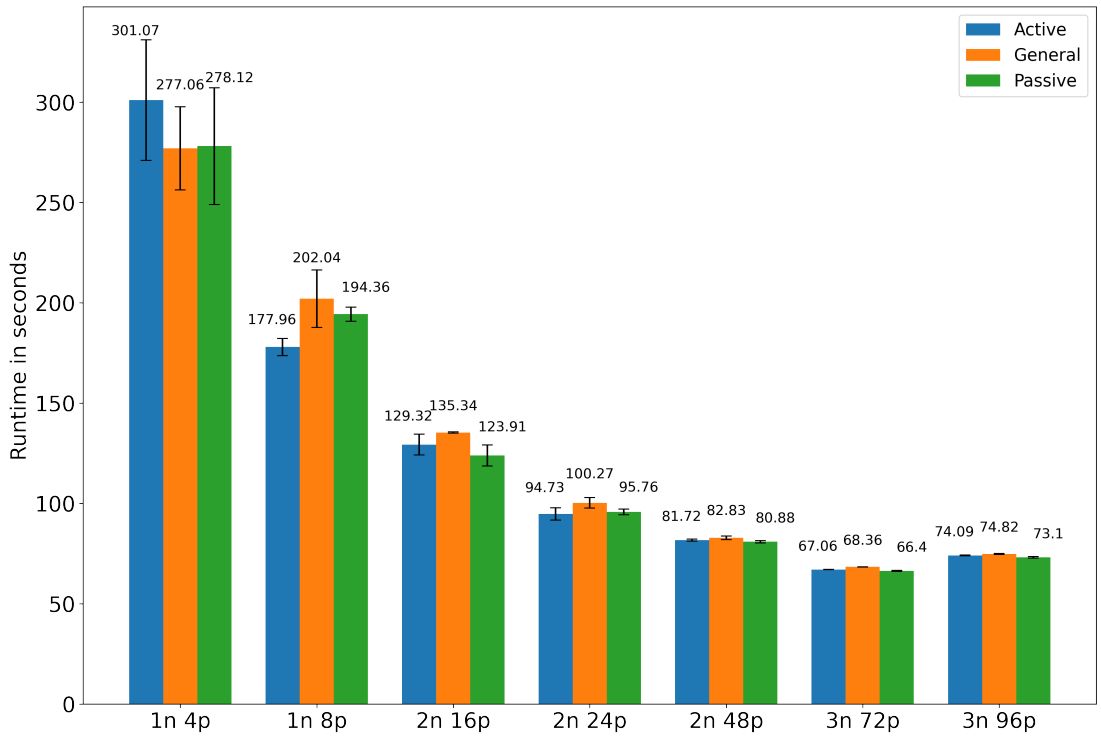


Figure 5.13: Runtimes of the shared memory Partdiff versions using Jacobi's method with 4105 lines and 5000 iterations

The results in Figure 5.13 do not show a clear best mode. The results for the first two configurations seem erratic, especially due to the comparatively high standard deviation. The results for higher numbers of processes are more even, with small standard deviations as well and no single mode consistently surpassing the others. One can assume that increasing the number of measurements for the first two configurations would even these runtimes out as well, considering the other results.

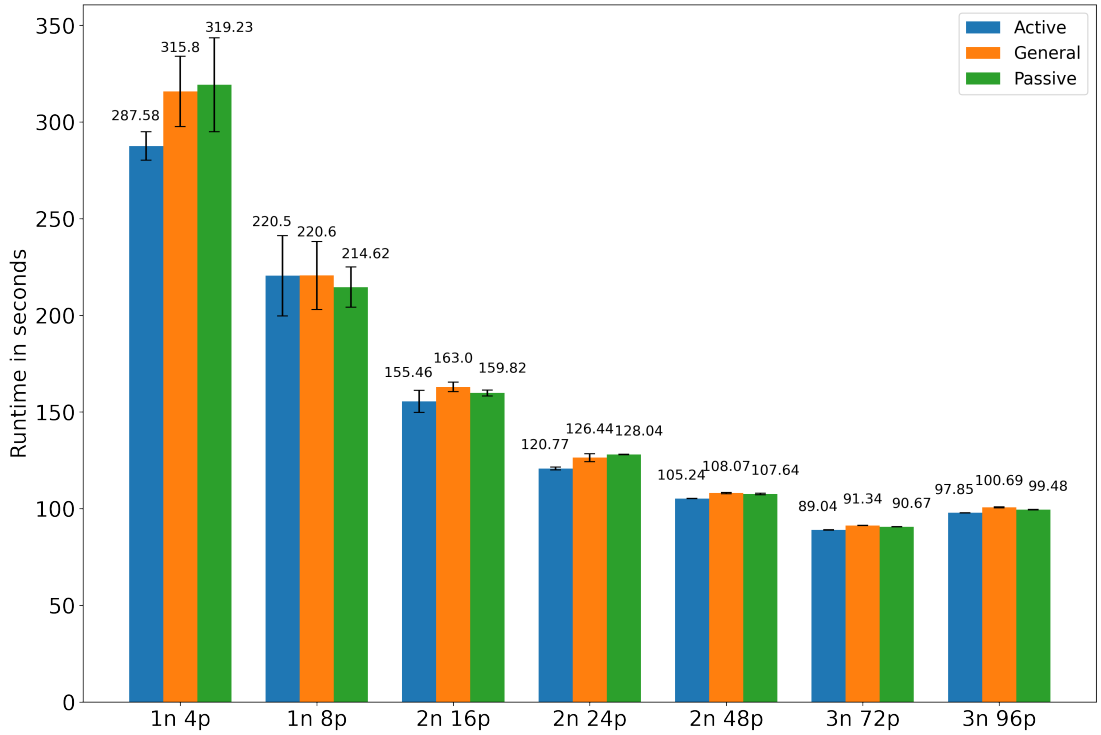


Figure 5.14: Runtimes of the shared memory Partdiff versions using Jacobi's method with 4105 lines and a target precision of 7.0637e-5

The runtimes displayed Figure 5.14 lead to similar assumptions, although the version using active target synchronization seems to be slightly faster considering the results while using 16 and 24 processes. The runtime for 4 processes also has a comparatively small standard deviation while being considerably faster at the same time, so the runtime is probably going to stay around that level if more measurements were taken, while it is unlikely for the other two modes to reach the same level after more experiments.

Before any conclusions on the performance of Jacobi's method are drawn, the weak scaling experiments will once again be considered first.

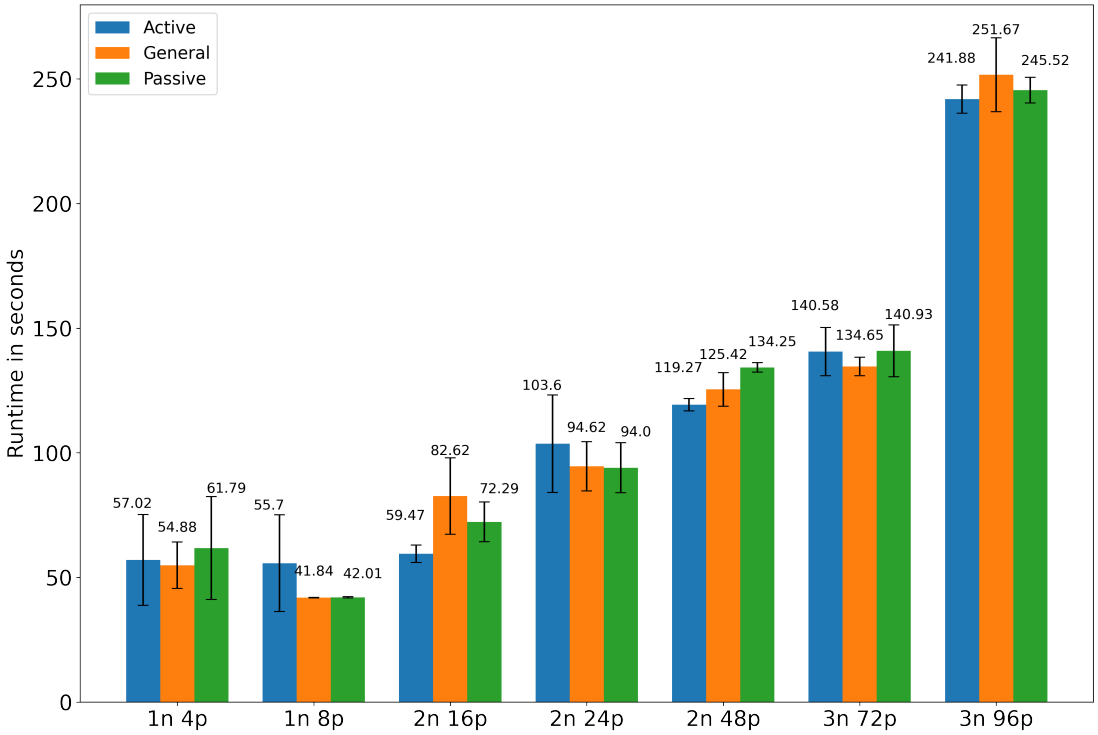


Figure 5.15: Runtimes of the shared memory Partdiff versions using Jacobi's method with weak scaling and 100 iterations

The results from the weak scaling experiment do not allow for any reliable conclusions either. The best mode in Figure 5.15 varies too much, while the standard deviation of most measurements is also very high in relation to the runtime overall. Perhaps longer lasting experiments with more individual measurements could lead to more conclusive results.

The runtimes displayed in Figure 5.16 still have comparatively large standard deviations attached to them, however not as drastically and not as frequently either. If one was to only consider the runtimes, passive mode would seem to perform the best, however given that the differences are mostly between 1 and 2 seconds and that the standard deviations are still higher than these differences, such an assumption could prove to be misleading. The measurements were only taken thrice, which means that any further experiments are likely to considerably change the displayed mean runtimes and standard deviations, considering that the standard deviations are as large as they are after these three measurements.

Therefore, forming a conclusion on the best mode for the implementation of Jacobi's method while using shared memory is not possible with the available test data. In order to potentially be able to produce more conclusive results, the experiments would have to be performed over a larger sample size.

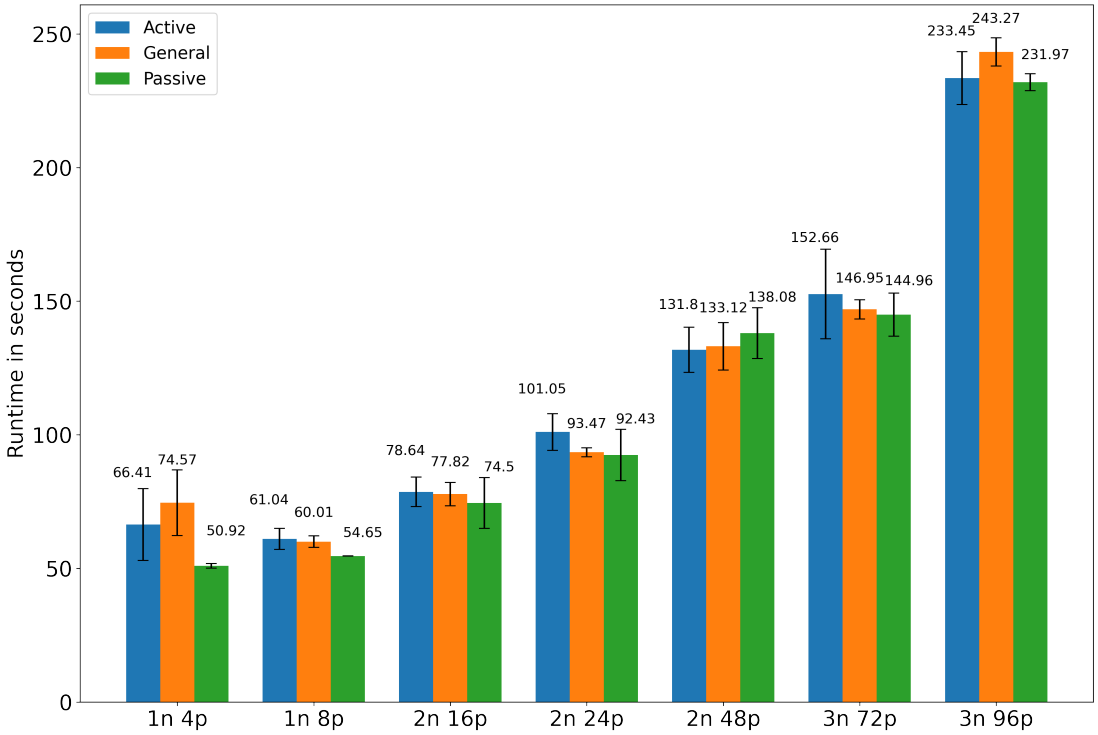


Figure 5.16: Runtimes of the shared memory Partdiff versions using Jacobi's method with weak scaling and a varying precision that requires 100 iterations to reach

Once again, the Gauss-Seidel method was used in the same experiments. This time, the runtimes matched the previous expectation that the different modes should not influence the performance heavily, as inter-communicator communication is rarely necessary.

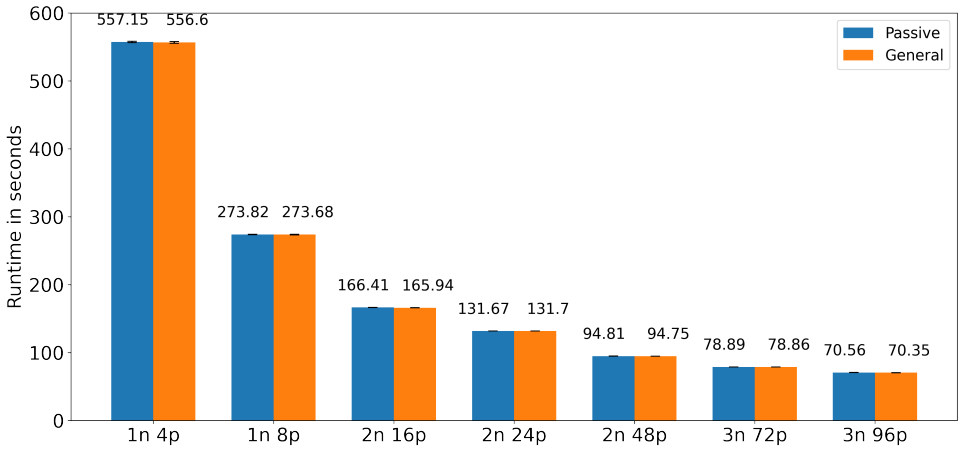


Figure 5.17: Runtimes of the shared memory Partdiff versions using the Gauss-Seidel method with 4105 lines and 5000 iterations

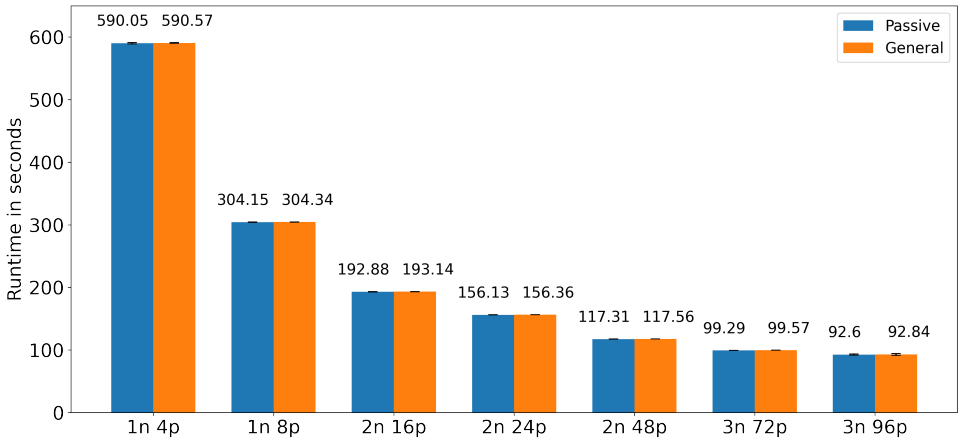


Figure 5.18: Runtimes of the shared memory Partdiff versions using the Gauss-Seidel method with 4105 lines and a target precision of 7.0753e-5

As mentioned, the resulting runtimes are almost exactly equal (see Figure 5.17 and Figure 5.18), showing that both modes work equally well on a matrix of the given size, no matter which termination criterion is used. During the weak scaling experiments a slight difference occurred though, as seen in Figure 5.19 and Figure 5.20.

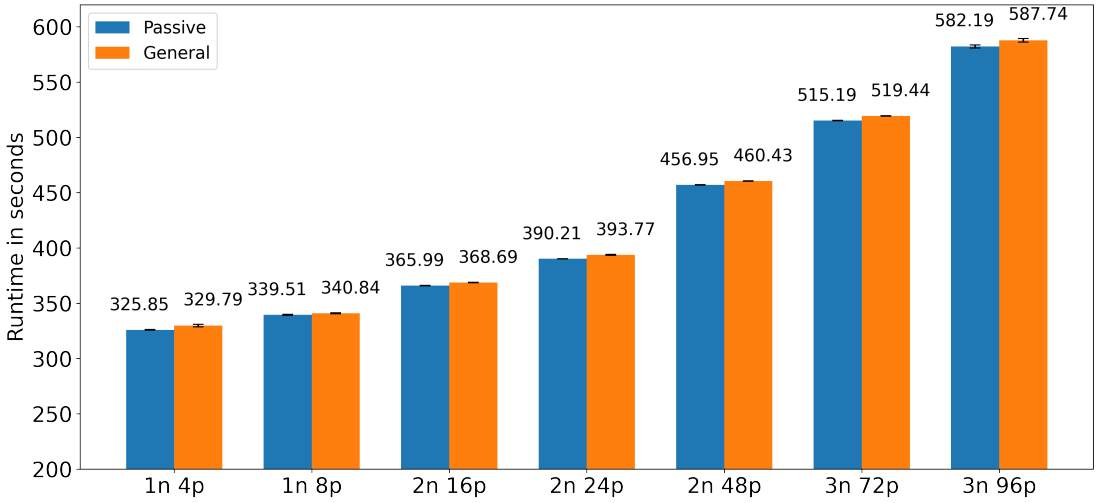


Figure 5.19: Runtimes of the shared memory Partdiff versions using the Gauss-Seidel method with weak scaling and 100 iterations, figure scale starting at 200 seconds

During both of the weak scaling experiments, the passive mode appears to perform slightly better. Regarding the results displayed in Figure 5.19, passive mode's mean was lower by at least 3 seconds except for the measurement involving only 8 processes. This difference is small enough that it could be considered natural variance, however the fact that it occurred on every measurement except for one alongside the small standard deviations of all measurements involved leads to believe that the passive mode

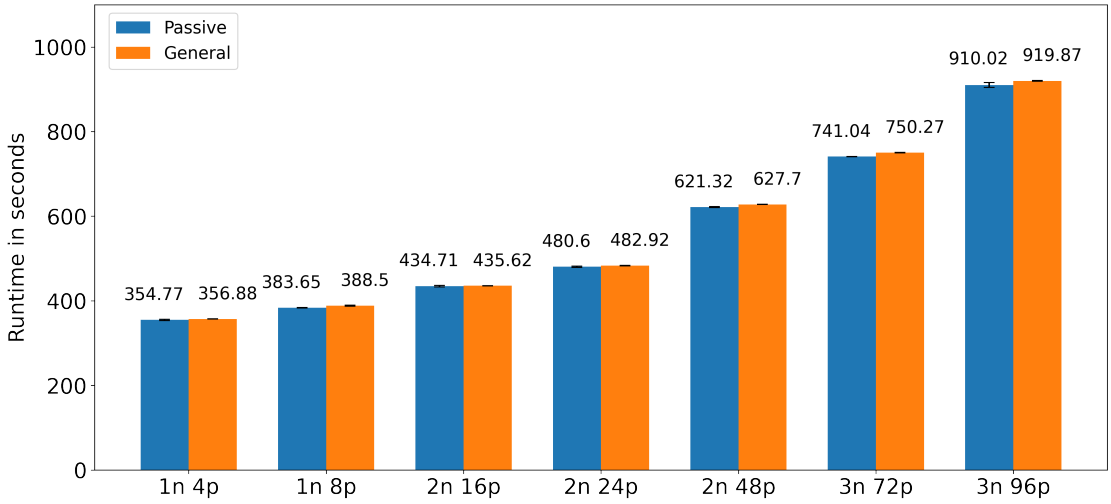


Figure 5.20: Runtimes of the shared memory Partdiff versions using the Gauss-Seidel method with weak scaling and a varying precision that requires 100 iterations to reach

indeed performs slightly better than the general active target mode. Similar observations can be made with the results of the experiment displayed in Figure 5.20 that used precision as the termination criterion. The measurements to the right that involved a higher number of processes show a growing difference, albeit a relatively small difference.

Combining the observations made in both weak scaling experiments involving Gauss-Seidel, the assumption can be made that the passive target mode is indeed slightly better suited than the general active target mode. However it should be noted that this difference is very marginal. Experiments with bigger sample sizes could produce different results, but given that the runtimes are as close together as they are alongside their comparatively small standard deviations, it is unlikely that one mode outperforms the other one by a considerable amount.

Another option that has been tried in this context is the use of P2P messages to handle inter-communicator message passing. For the experiments involving Jacobi's method, the results for this P2P version with shared memory are as erratic as the previous ones while displaying similar runtimes. For the experiments involving the Gauss-Seidel method, the results of the P2P version were similar to the presented ones as well while not surpassing them. Considering these results, the usage of point-to-point messages for the necessary communication between multiple shared communicator is a viable alternative to RMA calls, as they seem to perform on a similar level while also not requiring any window setup or explicit synchronization calls.

Before moving on to the next section, an interesting bug was discovered in the OpenMPI implementation of shared memory windows. The OpenMPI version in question is 3.1.6.

This was discovered during the weak scaling experiments. Initially, the weak scaling experiments were constructed in such a way that every process would store 2GB worth of matrix data. Problems arose for Jacobi's method though, as the execution of Partdiff with 48 or more processes was not possible with this experiment setup on the given hardware. The error messages given from MPICH were lacking critical information though and therefore a different MPI implementation was used to narrow down the cause of the problems for higher process numbers with Jacobi's method. The error messages from OpenMPI were much more useful and due to these, the error was discovered.

The amount of memory that was supposed to be exposed in shared memory exceeded the hardware's limitation, which is why the weak scaling experiments were adjusted accordingly, decreasing the size of the matrices to the ones previously explained. This did solve the problem for Partdiff when using MPICH, however it appears that OpenMPI was still not able to execute Partdiff with the adjusted matrix sizes. The error message displayed in Listing 5.3 was produced while using OpenMPI.

```
1  It appears as if there is not enough space for
     ↪ /dev/shm/osc_sm.abu4.cdc50001.0.4 (the shared-memory
     ↪ backing
2  file). It is likely that your MPI job will now either abort
     ↪ or experience
3  performance degradation.
4
5    Local host:  abu4
6    Space Requested: 133105378760 B
7    Space Available: 67367100416 B
8  --------------------------------------------------
9  [abu4:54385] *** An error occurred in
     ↪ MPI_Win_allocate_shared
10 [abu4:54385] *** reported by process [3452239873,0]
11 [abu4:54385] *** on communicator MPI COMMUNICATOR 3
     ↪ SPLIT_TYPE FROM 0
12 [abu4:54385] *** MPI_ERR_INTERN: internal error
13 [abu4:54385] *** MPI_ERRORS_ARE_FATAL (processes in this
     ↪ communicator will now abort,
14 [abu4:54385] ***    and potentially your MPI job)
```

Listing 5.3: Error message while creating shared memory windows with OpenMPI

Prior to the creation of this error message, Partdiff was supposed to executed with 11400 interlines on two nodes. As the method in question is Jacobi's method, two matrices of that size would need to be allocated, which would amount to a total of about 133.1GB worth of matrix data. This data would however be split across two nodes, meaning that each node would store about 66.55GB of data in their shared memory. The available space for shared memory is slightly higher than 67GB per node, as displayed in the error message above as well. Considering these numbers, executing Partdiff like this

would not exhaust the available memory and, as previously mentioned, the execution with MPICH was successful. For OpenMPI though, it appears that their shared memory window creation tries to allocate the entirety of the shared memory on a single node, instead of spreading the memory across the nodes. The reason for this remains unclear, however it heavily impedes the use of shared memory when utilizing OpenMPI and should be addressed, if it has not happened already.

6 Results

6.1 The improved RMA versions

Using the results gathered from the experiments on the different versions of Partdiff, an optimized version of Partdiff will be created. However, given that the results from the Partdiff versions using shared memory and the results from those that do not utilize that feature were considerably close, two versions will be built, one using the optimal modes and approaches for shared memory and one using the normal windows. These will then be compared to the P2P version of Partdiff during multiple different experiments to determine whether or not the usage of RMA provides a performance improvement. Any memory used for communication within the RMA versions will be allocated with MPI functions to enable any possible performance advantages that an MPI implementation may provide.

First, the non-shared memory version will be considered. Looking at the results from the initial Partdiff versions and the following attempts to improve performance, general active target mode still seems to be suited the most for Jacobi's method, especially considering the results while using the iterations as the termination criterion. These runtimes have not been surpassed by any other version and since the experiments using precision as the termination criterion did not produce a clear winner on any version of Partdiff, using general active target for that is feasible as well.

The number of windows will be reduced to one, according to the results of Section 5.3.4. While these experiments were conducted on an active target version, the usage in a general active version should entail no drawbacks, as it decreases the number of necessary synchronization calls.

The Gauss-Seidel method will also utilize the general active target mode, as both the general active target and the passive mode produced similar runtimes, but since the general active target mode seemed to perform marginally better, passive mode will not be utilized, although this difference could be attributed to natural variance, as it is quite small. In any case, using general active target mode shows no drawbacks when compared to passive mode, which is why it is a viable choice regardless.

Acting in line with the results from Section 5.3.1, the residuum will be calculated by passing the minimum of the local residuum and the predecessor's residuum on to the successor, given that precision serves as the termination criterion. Passing these residuum values on will also utilize the general active target mode, as it also provides the necessary synchronization operations.

It is worth mentioning that the other synchronization modes (barring active target

synchronization for Gauss-Seidel) do not perform significantly worse, which is why the usage of these other modes is feasible if the user is more comfortable with their utilization or has any other reason to use a different synchronization mode.

Now the shared memory version will be constructed. In the previous section, the data retrieved from the experiments did not allow for a clear decision on what mode performs the best for Jacobi's method. Therefore, the choice for which mode to choose is almost arbitrary.

The active target synchronization mode will be used for the shared memory version of Jacobi's method, as it lends itself more naturally to an algorithm such as Jacobi's method than the other two modes. Once again, all memory that has to be exposed for inter-communicator message passing is part of a single window to reduce the amount of necessary synchronization calls.

The Gauss-Seidel method will be implemented with passive target synchronization, as it appears to perform marginally better than general active target mode for this specific context. The residuum calculation will utilize the same approach as the non-shared version, but the values will be sent via P2P messages, since the passive mode does not provide any synchronization operations for the target process. Due to this, manual synchronization via P2P messages would be necessary if the values were to be sent with RMA operations. So instead of using RMA operations or reading it from shared memory and then sending a P2P message to confirm that the communication is completed, the value will be sent directly with the P2P message, skipping the RMA operation entirely.

Once again, it should be noted that the usage of the other synchronization modes is also a viable option since they did not perform much worse than the ones that are now in use, especially considering the results from the experiments involving Jacobi's method.

The final versions have also undergone certain refactorings that were learned of during the creation of said versions. These refactorings can also be a reason for differences between results from previous experiments and upcoming results, alongside the various adjustments mentioned above.

6.2 WR cluster

In the following, the point-to-point version will be compared to the two improved RMA versions, one that is using shared memory and one that is not. For this, all versions will undergo a set of experiments, starting with the already familiar setup for strong and weak scaling experiments. The results produced in these experiments will be used to determine whether or not using RMA provides a performance advantage over the more traditional point-to-point communication for the tested algorithms. Before this comparison, the OSU Benchmarks will be used again to determine whether one- or two-sided communication performs better in a micro benchmark environment.

6.2.1 Comparing P2P and RMA with OSU benchmarks

As the results from `MPI_Get` were marginally better than those of `MPI_Put` after the previous execution of these benchmarks in Section 5.1, both the latency and the bandwidth from `MPI_Get` and point-to-point communication will be compared. As before, the benchmarks were executed twice to cover both inter- and intra-node communication.

The first set of results regarded will be from the latency benchmark. The latency benchmark for point-to-point communication is executed in a ping-pong-like manner. One process sends a message of a certain size to the other one and waits for a reply with the same size. The average one-way latency is then calculated over many iterations per message size. For an explanation of the `MPI_Get` latency benchmark, refer to Section 5.1.

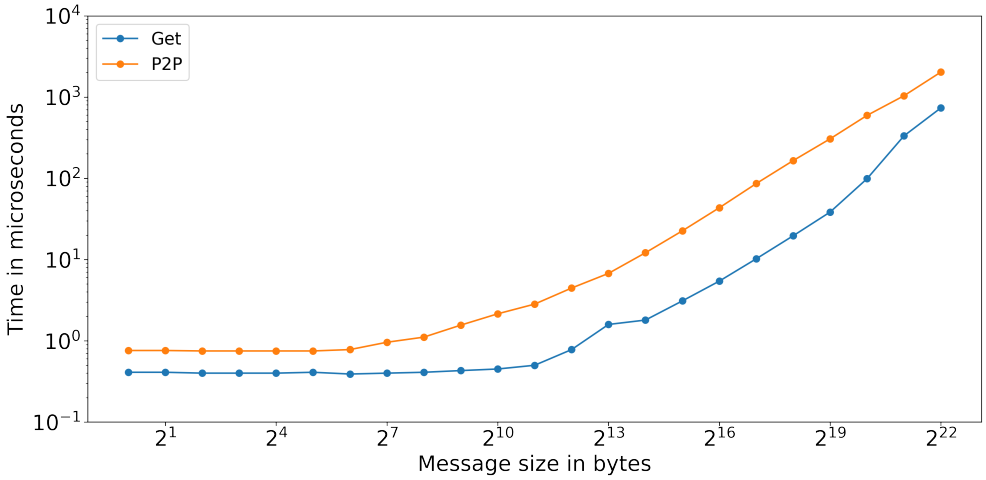


Figure 6.1: Comparing P2P and RMA intra-node latency with the OSU Micro Benchmarks (logarithmic scales)

As previously mentioned, both inter- and intra-node communication are differentiated in this experiment. In Figure 6.1, the intra-node latency is displayed on a logarithmic scale. In this scenario, the latency of `MPI_Get` is lower at all times, regardless of message size.

The results from the inter-node benchmark differ though (see Figure 6.2), as both forms of communication are on par for many different message sizes. For message sizes between 1 and 128 bytes, point-to-point communication is slightly faster, although this difference is a lot smaller than it appears due to the logarithmic scale. The absolute difference amounts to less than a microsecond. Messages ranging from 256 bytes to roughly 10 kB are sent with roughly the same latency. Message sizes ranging between 10 kB and close to 1 MB barely favor P2P again. For message sizes beyond 1 MB both communication forms are roughly on par again.

This leads to the conclusion that intra-node communication heavily favors RMA,

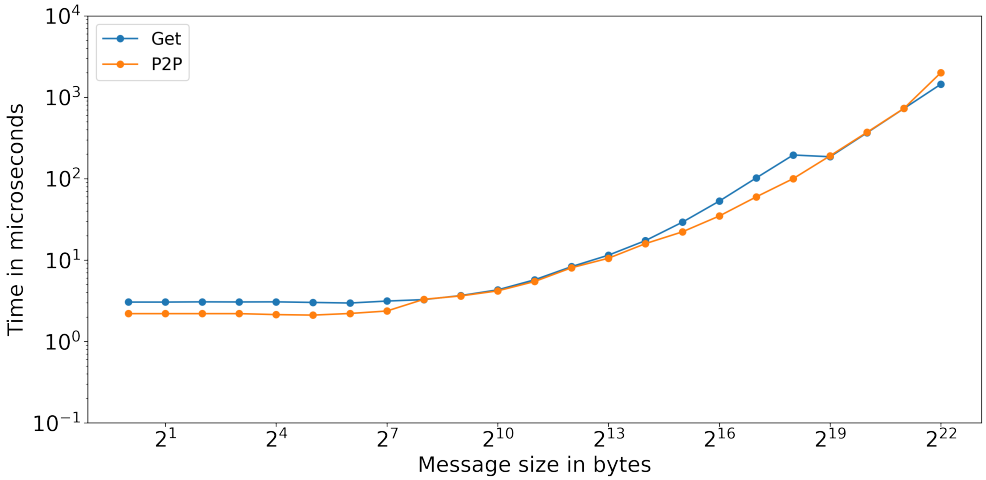


Figure 6.2: Comparing P2P and RMA inter-node latency with the OSU Micro Benchmarks (logarithmic scales)

whereas inter-node communication barely favors P2P, although RMA produced competitive results for many message sizes. Note that intra-node communication incurs smaller latencies, which is to be expected, as the communication overhead should be smaller than the overhead for inter-node communication. The lower latency of `MPI_Get` in Figure 6.1 is also according to previous expectations, as the one-sided communication requires less synchronization and only requires the origin to actively participate in this communication. However, the reason as to why P2P is able to surpass RMA during inter-node communication remains unknown. Possible options include the MPI implementation, as the RMA operations could potentially be less optimized for inter-node communication than point-to-point messages.

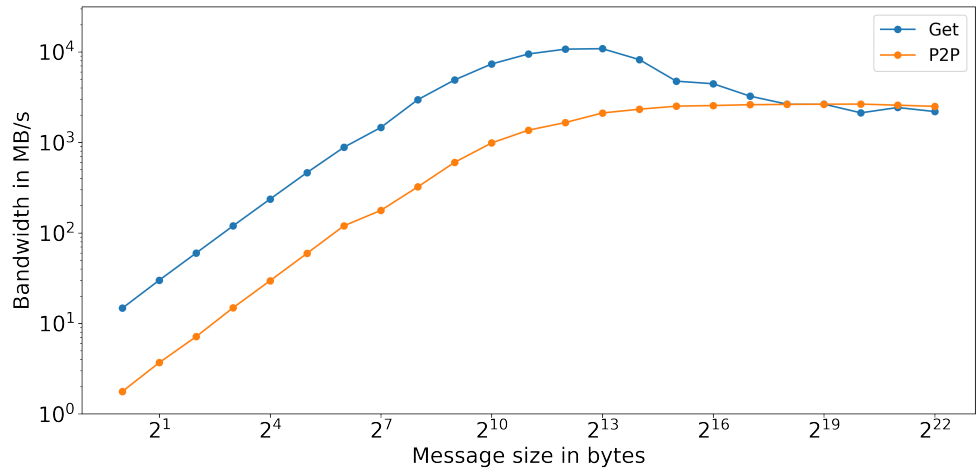


Figure 6.3: Comparing P2P and RMA intra-node bandwidth with the OSU Micro Benchmarks (logarithmic scales)

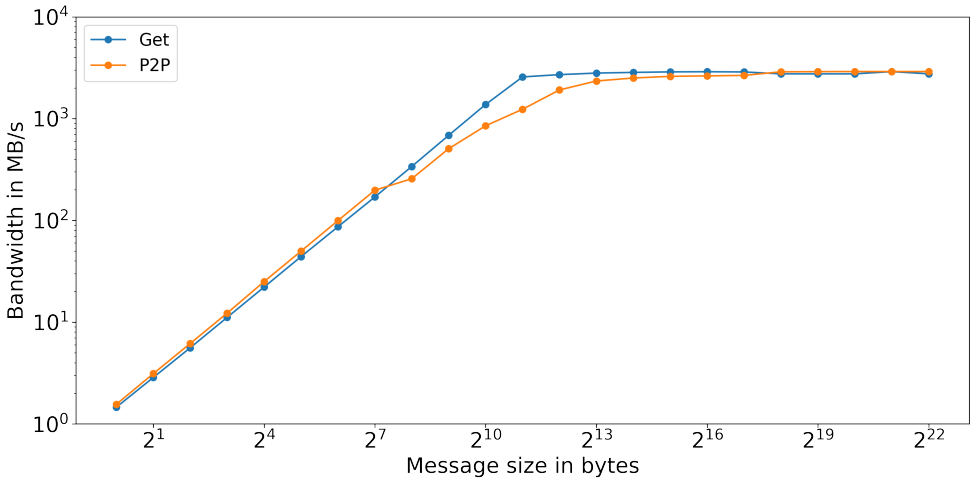


Figure 6.4: Comparing P2P and RMA inter-node bandwidth with the OSU Micro Benchmarks (logarithmic scales)

Regarding the bandwidth benchmark, the point-to-point version is similar to the one for `MPI_Get` as well. The sending process sends a fixed number of messages to the receiver and waits for a reply, which will only be sent after all previously mentioned messages have arrived. This process is repeated multiple times. The elapsed time and the number of bytes sent are then used to calculate the bandwidth.

As before, the intra-node bandwidth will be investigated first, displayed in Figure 6.3. The bandwidth for point-to-point communication plateaus after reaching a message size of roughly 32 kB, at around 2500 MB/s. The bandwidth for RMA not only starts off higher, but its increase is also stronger, reaching a peak at message sizes of 4 kB, with a bandwidth of more than 10000 MB/s. For larger message sizes, the bandwidth declines until it reaches roughly the same level as P2P communication at the largest tested message sizes.

For inter-node communication (see Figure 6.4), both communication forms achieve similar bandwidths throughout most message sizes, with multiple exceptions around 1 kB messages. For these, `MPI_Get` achieves higher bandwidths, but these differences disappear at the highest tested message sizes just as before.

Once again RMA, or rather `MPI_Get`, performs better than P2P communication in terms of bandwidth. For inter-node communication, both are mostly on par with a few exceptions, but the results from intra-node communication clearly indicate that the usage of `MPI_Get` allows for larger bandwidths. Combining the results from the latency and bandwidth benchmarks, using `MPI_Get` is superior to traditional point-to-point communication in a micro benchmark environment. This statement is only valid for this particular hardware and MPI implementation though, as the benchmarks were not executed for a different setup.

6.2.2 Strong and weak scaling experiments for Jacobi's method

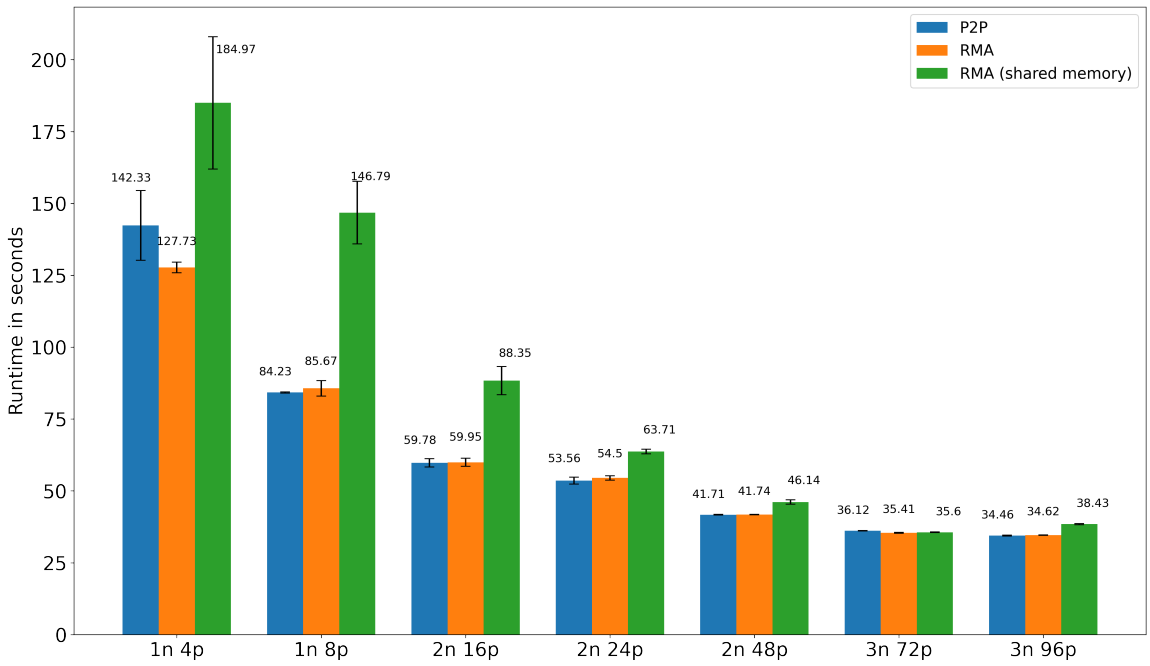


Figure 6.5: Runtimes of the different Partdiff versions using Jacobi's method with 4105 lines and 5000 iterations

The first two experiments involved Jacobi's method (see Figure 6.5 and Figure 6.6). In the first one, the non-shared memory RMA and the P2P version are on par for most configurations, barring the first one. Surprisingly, the shared memory version is by far the worst, especially for the configurations that involve only a small number of processes. With rising process numbers, the difference between the shared memory version and the others diminishes, until they are almost equal at 96 processes.

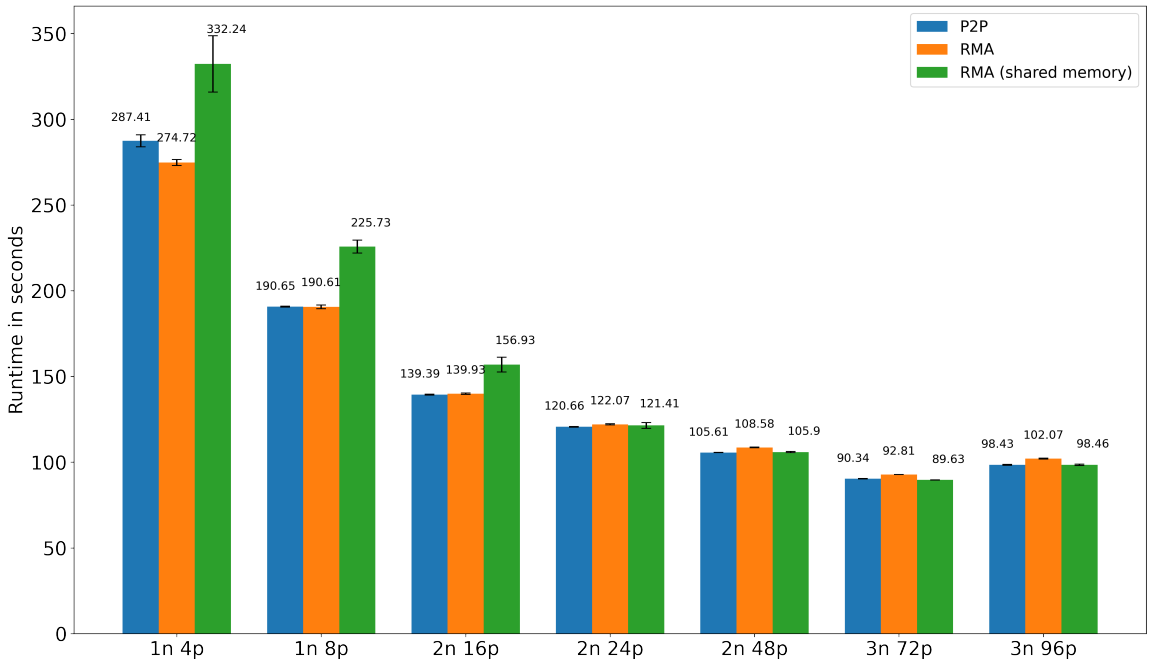


Figure 6.6: Runtimes of the different Partdiff versions using Jacobi's method with 4105 lines and a target precision of 7.0637e-5

The experiment involving the same method but terminating after a certain precision is reached shows similar results. The P2P and the non-shared RMA version are almost equal for every configuration, disregarding the first one. The shared memory version is once again inferior with small process numbers, but catches up significantly at higher process numbers, where it performs comparably to the other two versions, similarly to the previous experiment.

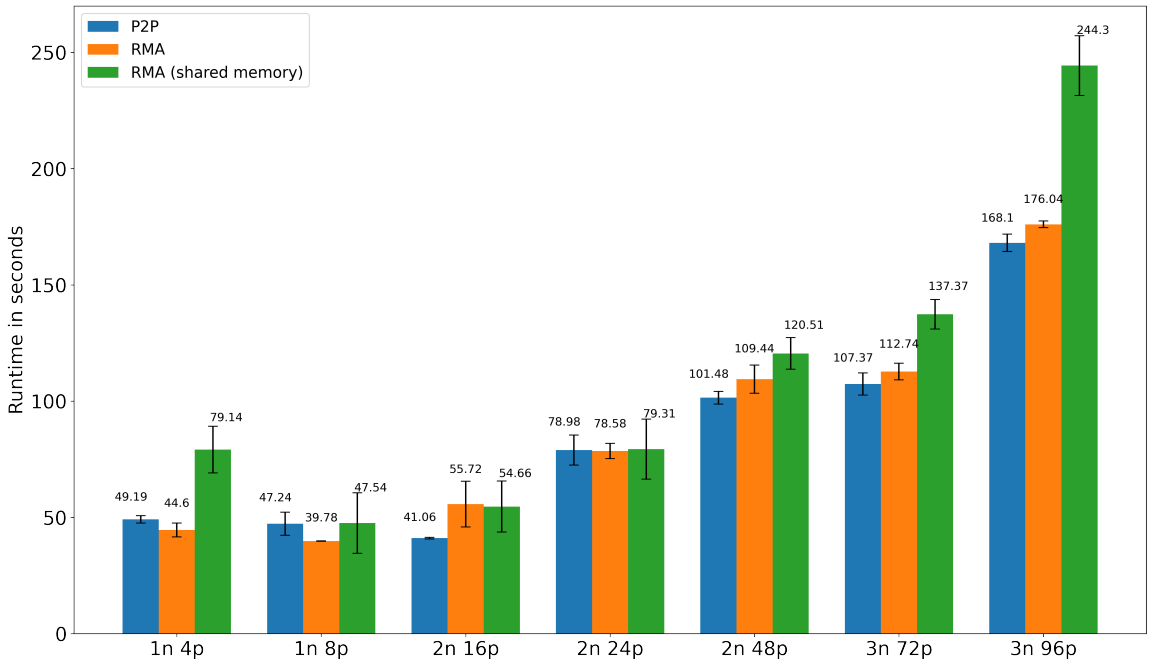


Figure 6.7: Runtimes of the different Partdiff versions using Jacobi's method with weak scaling and 100 iterations

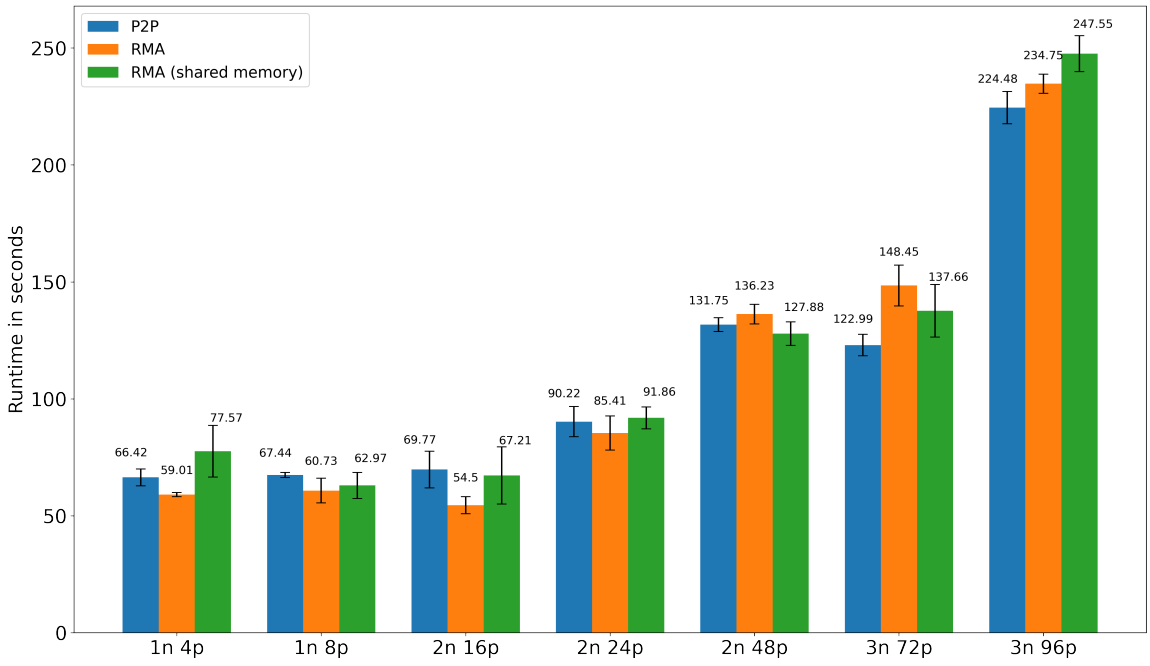


Figure 6.8: Runtimes of the different Partdiff versions using Jacobi's method with weak scaling and a varying precision that requires 100 iterations to reach

For the weak scaling experiments, the observations made are different compared to

the results of their respective strong scaling counterparts. The runtimes for the P2P and the non-shared RMA version are not on par anymore for most configurations. Instead, the RMA version is superior while Partdiff is only executed on a single node, while the P2P version is faster on almost all configurations involving 2 or more nodes. The shared memory version is once again inferior, especially on the later measurements to the far right of Figure 6.7.

For the weak scaling experiment where precision acts as the termination criterion, different observations are made yet again. As before, the non-shared RMA version is the fastest of all three while executing on a single node, however this time this advantage can also be observed on the first two configurations involving 2 nodes. From there on though, the P2P version overtakes the non-shared RMA version again, as it appears to scale better with rising numbers of processes and a proportionally rising problem size. The shared RMA version seems to perform better on this experiment, as the produced runtimes are either at a similar level compared to the other versions or even slightly better, as it is the case for the configuration using 48 processes. However this statement can only be made if the first and the last measurements in Figure 6.8 are disregarded, in which the shared version performs the worst again.

From these experiments, the following preliminary conclusion about the best approach for the implementation of Jacobi's method can be drawn. For small numbers of processes, using a non-shared RMA version in Jacobi's method seems to perform the best, whereas bigger cluster configurations and also bigger problem sizes seem to benefit point-to-point communication. The shared memory version rarely produced competitive results, surprisingly, which is why it is deemed to be the worst approach in this preliminary conclusion.

6.2.3 Strong and weak scaling experiments for the Gauss-Seidel method

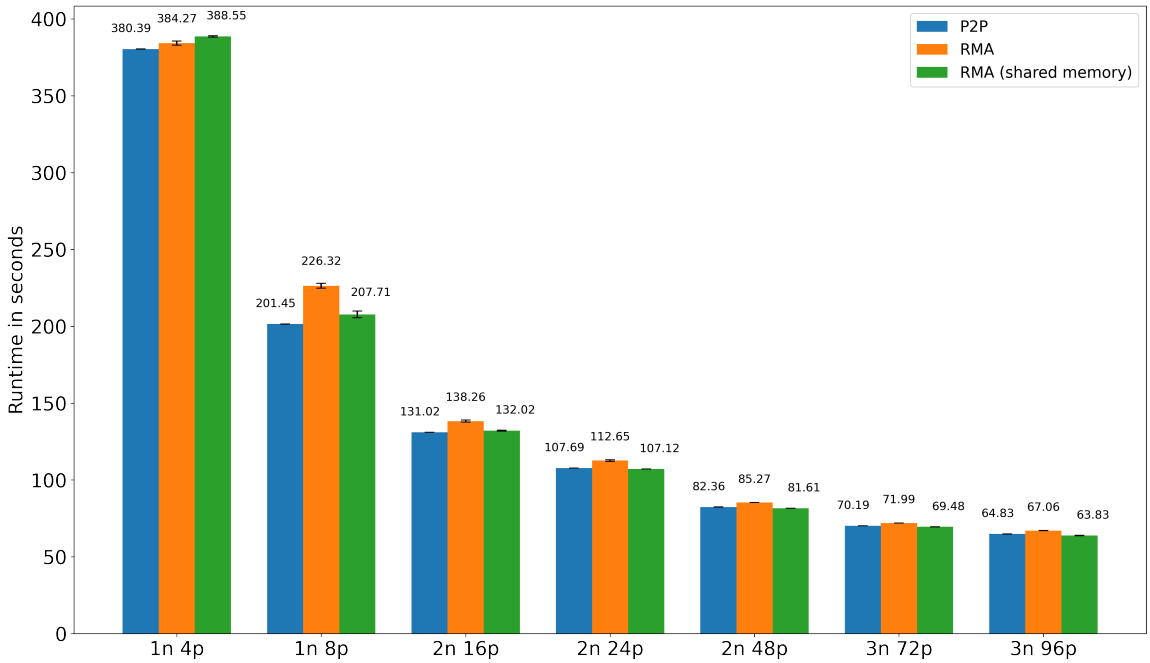


Figure 6.9: Runtimes of the different Partdiff versions using the Gauss-Seidel method with 4105 lines and 5000 iterations

For the experiments involving the Gauss-Seidel method, different observations can be made yet again. In the strong scaling experiment involving termination with iterations as the criterion, the P2P version performs the best on the measurements taken on a single node. For all measurements involving more than one node, the shared memory version is at least on par with the P2P version, indicating that the shared version might scale the best with higher process numbers out of the three test candidates. The non-shared RMA version appears to be the worst at all tested configurations, disregarding the very first, although the difference becomes less significant with rising process numbers.

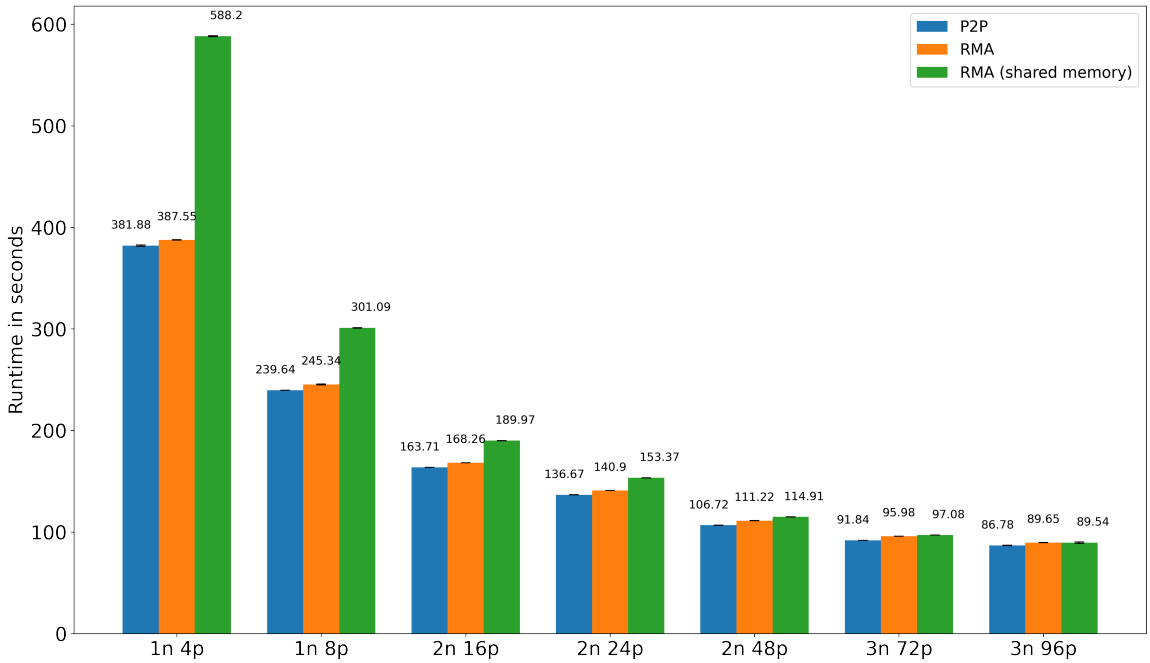


Figure 6.10: Runtimes of the different Partdiff versions using the Gauss-Seidel method with 4105 lines and a target precision of 7.0753e-5

The runtimes involving termination after a certain precision is reached do not allow for a similar observation, as displayed in Figure 6.10. The P2P version performs the best for all configurations, whereas the shared memory version performs significantly worse compared to the previous experiment. It is only with higher process numbers that the shared version catches up to the others. The non-shared RMA version produced runtimes that are always marginally worse than their respective counterparts of the P2P version. Once again, it appears that the shared memory version scales better with higher process numbers than the others, although this scaling did not enable the shared version to surpass the other two candidates on just 96 processes.

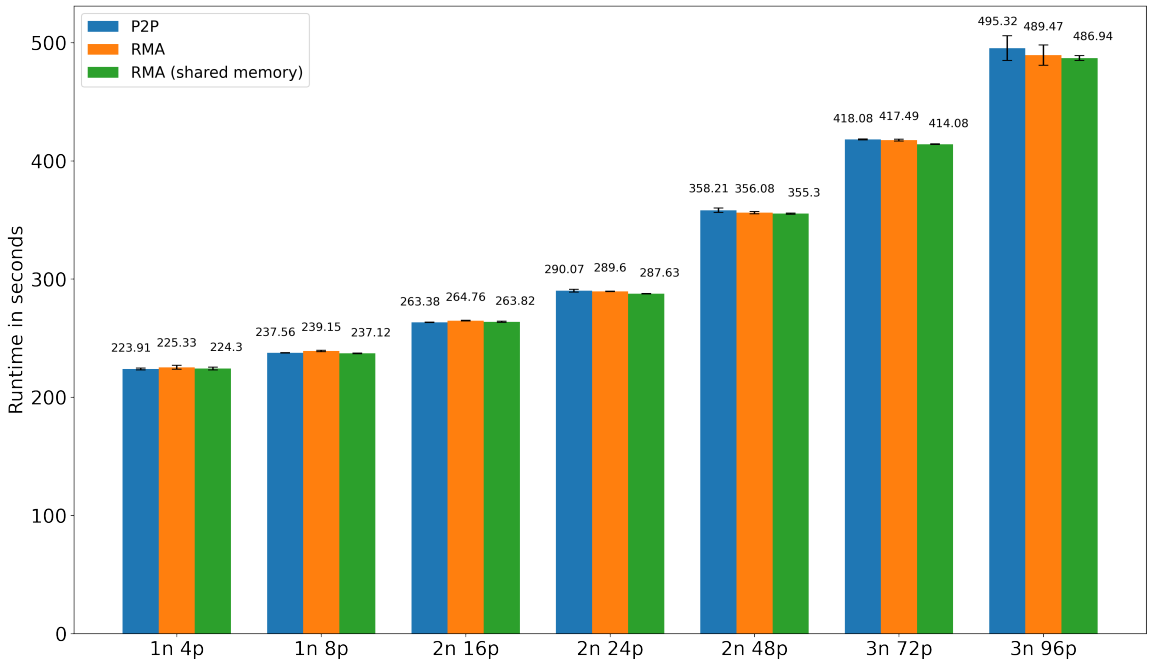


Figure 6.11: Runtimes of the different Partdiff versions using the Gauss-Seidel method with weak scaling and 100 iterations

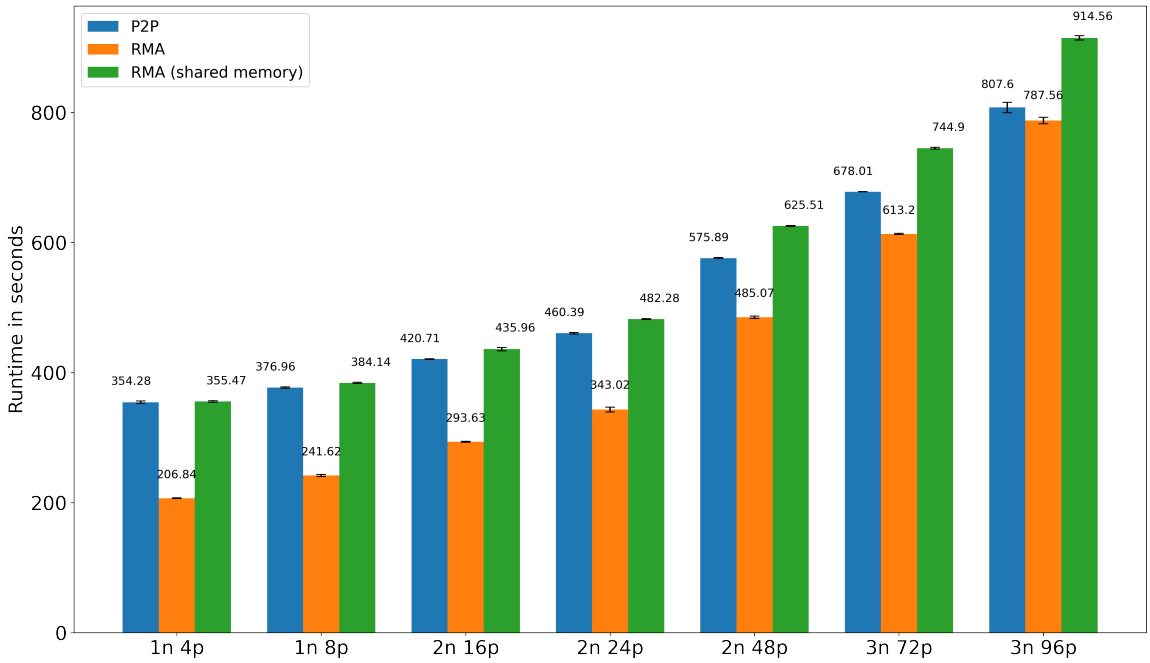


Figure 6.12: Runtimes of the different Partdiff versions using the Gauss-Seidel method with weak scaling and a varying precision that requires 100 iterations to reach

A look at the first weak scaling experiment (results displayed in Figure 6.11) corroborates the assumption that the shared memory version scales better than its competitors. The first runtimes differ within an interval that is presumed to be natural variance, but the difference seems to increase for the later configurations, for which the shared memory version performs slightly better. However this difference is still marginal and could dissipate over a larger sample size of measurements. If that would be the case, the three candidates would most likely still perform comparably, given the previous results.

The next and therefore last weak scaling experiment shows vastly different results, as the three versions produce wildly differing runtimes while using precision as the termination criterion when compared to the previous results. The reason for this behaviour is currently unknown. If precision acts as the termination condition, all three versions end any given iteration by sending their residuum to the successor, as previously explained in Section 5.3.1. This is the only significant difference in the implementation of the two termination conditions, and these differences are consistent across all three examined Partdiff versions, which is why this result is not only unexpected, but also currently inexplicable.

In this experiment, the non-shared RMA version performs the best on all occasions, although that difference decreases with larger process numbers. The P2P and the shared memory candidates perform comparably for small numbers of processes, but as the process numbers increase, so does the difference in runtime, as the P2P version appears to perform better when subjected to the scaling problem and cluster size.

As before, a preliminary conclusion regarding the Gauss-Seidel method can be drawn. If it was not for the results of the last experiment, this conclusion would unequivocally have been that the P2P version performs the best for small process numbers and problem sizes, whereas the best results for large process numbers and problem sizes are produced by the shared version. That difference is very small though, but even if it is caused by natural variance, both versions perform comparable at the very least. However, the results from the last weak scaling experiment differ greatly, as it appears that the non-shared RMA version performs the best when all candidates are subjected to large problem sizes while using the Gauss-Seidel method and using precision as the termination criterion.

6.2.4 Communications overhead and large process numbers

The purpose of the next experiment is to determine whether or not inter-node communication and intra-node communication incur different message overheads. To investigate this, the different Partdiff versions are once again executed with a square matrix of 4105 lines and 5000 necessary iterations. However, the cluster setup is different this time. For every version, four different configurations were tested. All of them involved 36 processes, but each configuration involves a different number of nodes. The first one involves one node and 36 processes, the second one involves 2 nodes with 18 processes each and so on. The results are displayed in Figure 6.13 and Figure 6.14. Note that the y-axes for the

following figures do not start at 0 seconds.

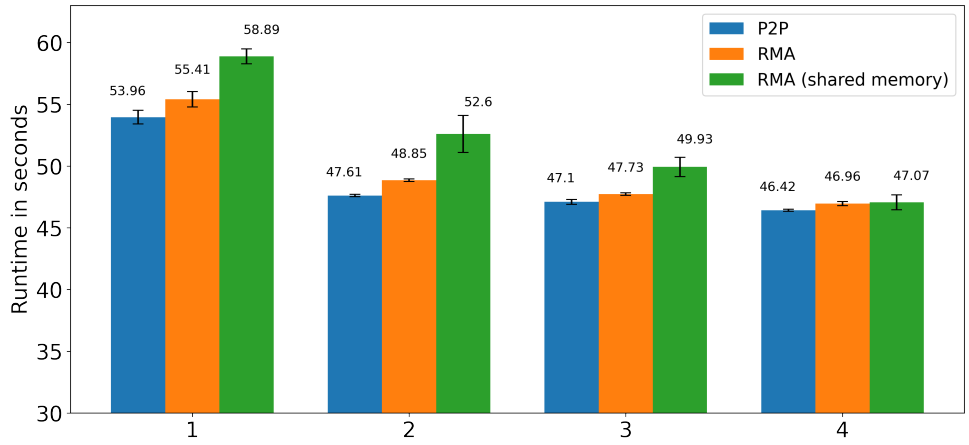


Figure 6.13: Runtimes of the different Partdiff versions using Jacobi's method with 4105 lines and 5000 iterations while using 36 processes in total (using n nodes)

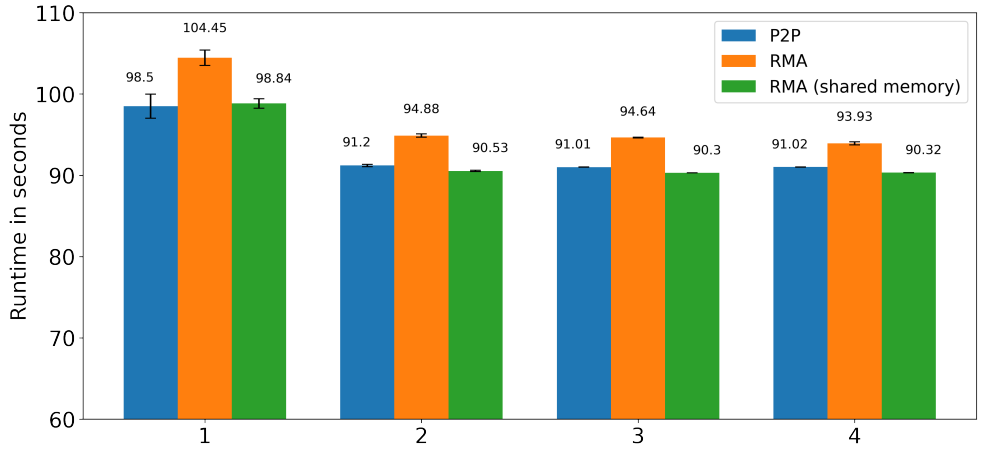


Figure 6.14: Runtimes of the different Partdiff versions using the Gauss-Seidel method with 4105 lines and 5000 iterations while using 36 processes in total (using n nodes)

Starting with the experiment involving Jacobi's method (Figure 6.13), it appears that intra-node communication takes more time compared to inter-node communication. Apparently, this is especially the case for the shared memory version. The runtimes for this version decrease further the more nodes are involved, whereas the other two versions only saw a decrease when going from one node to two, leading to the assumption that intra-node communication incurs the larger overhead. This result is unexpected, especially for the shared memory version, as shared memory should excel at computing on just one node, as all of the necessary data is accessible as shared memory, which should allow for faster accesses compared to having to send the data via messages.

The previous results do not appear to be an outlier, as the runtimes displayed in Figure 6.14 lead to a similar conclusion. Once again, all versions experience a decrease in runtime when shifting to running the computation on two nodes instead of one. This time however the runtime for the shared memory version does not decrease further with an increase in the number of nodes. Note that the differences in runtimes between the three versions all match with previous observations from the strong scaling experiments.

These results could lead to the conclusion that intra-node communication is somehow less efficient than inter-node communication. This would however contradict the earlier findings during the application of the OSU benchmarks in Section 6.2.1. While it is possible that the change of circumstances, namely the usage of an application benchmark instead of a micro benchmark, could cause different results, it seems much more likely that this effect is caused by something else. The suspected reason for this behaviour is the underlying hardware and AMD's proprietary core boosting technology. Chapter 7 further expands on this topic.

The next experiment involves testing with high numbers of processes. To determine which version scales the best with many processes, all versions are subjected to an experiment in which Partdiff is executed with 200 processes across five nodes, iterating 5000 times over a square matrix with 24009 lines. The results are displayed in the following graph.

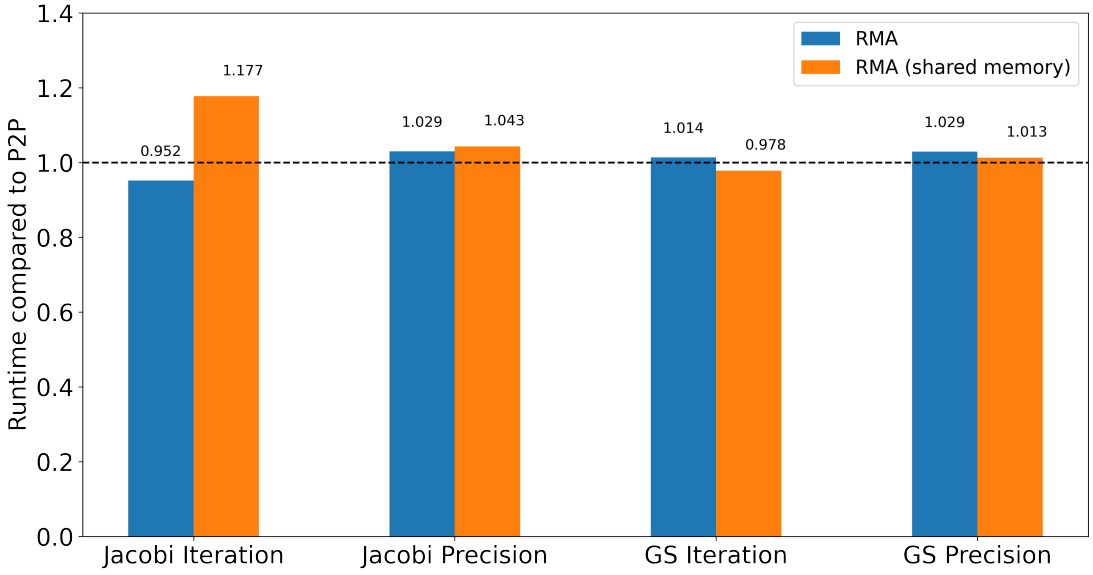


Figure 6.15: Comparison of the runtimes of the different Partdiff versions while iterating 5000 times over a 24009 line square matrix using 200 processes across 5 nodes. These values were obtained by dividing the respective runtime of the RMA version by the runtime of the P2P version.

The values displayed in Figure 6.15 were obtained by dividing the respective runtime of the RMA version by the runtime of the P2P version. Therefore, the P2P version

serves as a baseline and is represented by the dotted horizontal line. Thus, a value lower than 1, such as the value of the non-shared RMA version on the far left, means that this particular runtime is lower than the corresponding runtime of the P2P version by the factor displayed above the bar. Conversely, a value larger than 1 means that the RMA version needed more time than the corresponding P2P version, meaning it was slower than the P2P version.

The result on the far left involving Jacobi's method shows that the non-shared RMA version seems to perform best when Partdiff is executed with such a high number of processes. This result is only partially in line with the previous observations from the strong and weak scaling experiment. During the strong scaling experiment, the P2P version and the non-shared RMA version showed similar results for high process numbers, while the weak scaling experiment revealed that the P2P version scales better with the problem size than the RMA versions. This leads to the assumption that the non-shared RMA version scales better with rising process numbers on comparably small problem sizes, but as soon as the problem size reaches a certain threshold, the P2P version performs better. The performance of the shared memory version is consistent with the previous observations in both experiments though, as it still performs the worst out of all three.

Switching the termination condition changes the results though, as the P2P version achieves the best runtimes under these conditions. The non-shared RMA version still performs the worst, but the difference diminishes. These results are somewhat in line with previous observations, as the P2P version performed better on high process numbers during both the corresponding strong and weak scaling experiment. The shared memory version's results are also consistent, as it performed the worst during these experiments as well, while it was also able to decrease the gap to its competitors when compared to the previous experiment's results.

The experiment with the Gauss-Seidel method when terminating after a certain number of iterations also produced results that are consistent with previous observations. During the previous experiments involving this method, the shared RMA version performed just as well as the other versions, if not marginally better. The results from this particular experiment seem to confirm the previous assumption that the shared memory version scales the best with high process numbers while using the Gauss-Seidel method and this termination condition.

Lastly, the experiment involving precision as the termination criterion will be regarded. During the previous strong scaling experiment with this method, the P2P version was slightly superior on high process numbers, as it is in this experiment as well. The respective weak scaling experiment revealed that the non-shared RMA version performed the best on all configurations. Combining these previous findings with the result from the experiment displayed in Figure 6.15 leads to the assumption that the P2P version scales best on small problem sizes, but as soon as the problem size exceeds a certain limit the non-shared RMA version is superior. The results from the shared memory version are also consistent with previous observations, as it performed slightly worse than the P2P version on all previous occasions regarding the Gauss-Seidel method and precision as the condition.

6.3 Mistral

In order to increase the available data to ultimately come to a conclusion, similar tests with the three Partdiff versions were conducted on Mistral, the high-performance computing system of the DKRZ, which occupies the 109th rank of the Top500 list as of November 2020. The nodes used for these tests consist of two Intel Xeon E5-2695 v4 CPUs, amounting to 36 physical or 72 logical cores per node with 64 GB main memory. The MPI implementation that was used to compile and execute the Partdiff versions is OpenMPI 2.0.2p2_hpcx-gcc64, a high-performance implementation of OpenMPI. Additionally, MellanoX Messaging [19] is utilized to further accelerate message passing. As before, both Jacobi's method and the Gauss-Seidel method were subjected to similar strong and weak scaling experiments. Note that for these experiments, the shared memory version was slightly changed. In the previous versions, the entire local matrix was exposed in shared memory. The new version only exposes the halo lines in shared memory to accommodate for the lower amount of shared memory on the Mistral nodes in use. The new shared memory version was also tested on the same hardware that was used for the previous experiments, however the results were similar to the previous version, which is why neither version is strictly better than the other one. Therefore, presenting these results will be omitted.

Note that the following experiments utilize the same process configurations as the previous experiments on the WR cluster. The reason for this choice is to maintain comparability between the experiments on the two different hardware setups.

6.3.1 Jacobi's method on Mistral

The results from the previous experiments involving Jacobi's method indicated that the non-shared RMA version performs better on small cluster configurations and problem sizes, but is overtaken by the P2P version for setups involving larger cluster configurations, especially with increasing problem size. The shared memory version performed the worst on all occasions.

The results from strong scaling experiments on Mistral differ slightly from these previous observations. In both Figure 6.16 and Figure 6.17, the point-to-point version of Partdiff and the non-shared RMA version differ by fractions of a second on every tested cluster configuration. The shared memory version however performs the worst again, especially when using the number of iterations as the termination condition, although the severity of the difference has decreased when compared to the previous experiments. When using precision as the termination criterion, this difference becomes even smaller, as the shared memory version only performs worse when using four processes and a single node. For all other configurations, the runtimes of all three versions are essentially equal.

The results from the weak scaling experiments are vastly different when compared to the results from the previous experiments. As seen in Figure 6.18, all three versions perform comparably when using a single node. However, increasing both the number of

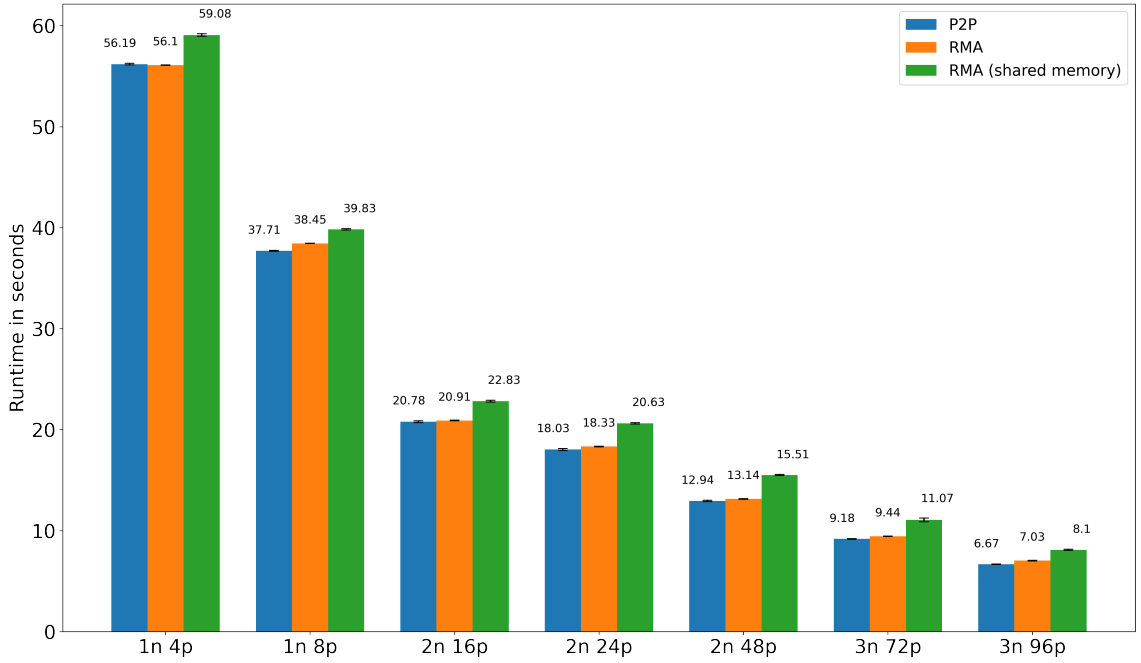


Figure 6.16: Runtimes of the different Partdiff versions using Jacobi's method with 4105 lines and 5000 iterations on Mistral

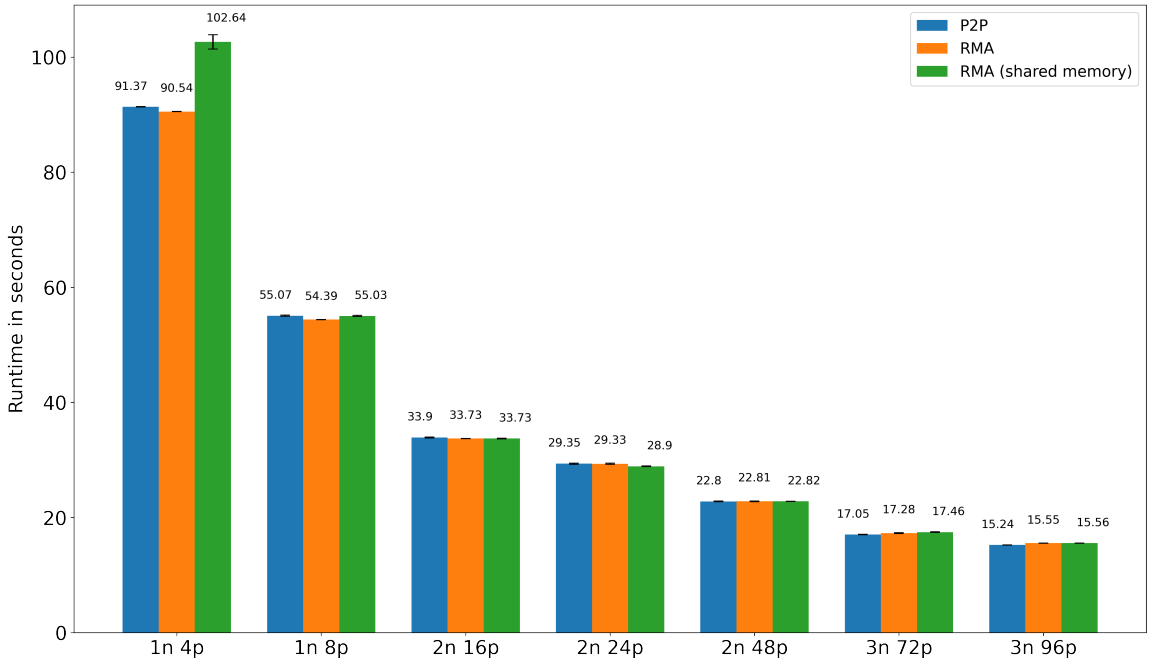


Figure 6.17: Runtimes of the different Partdiff versions using Jacobi's method with 4105 lines and a target precision of 7.0637e-5 on Mistral

processes and nodes seem to negatively affect the P2P version, as it performs the worst

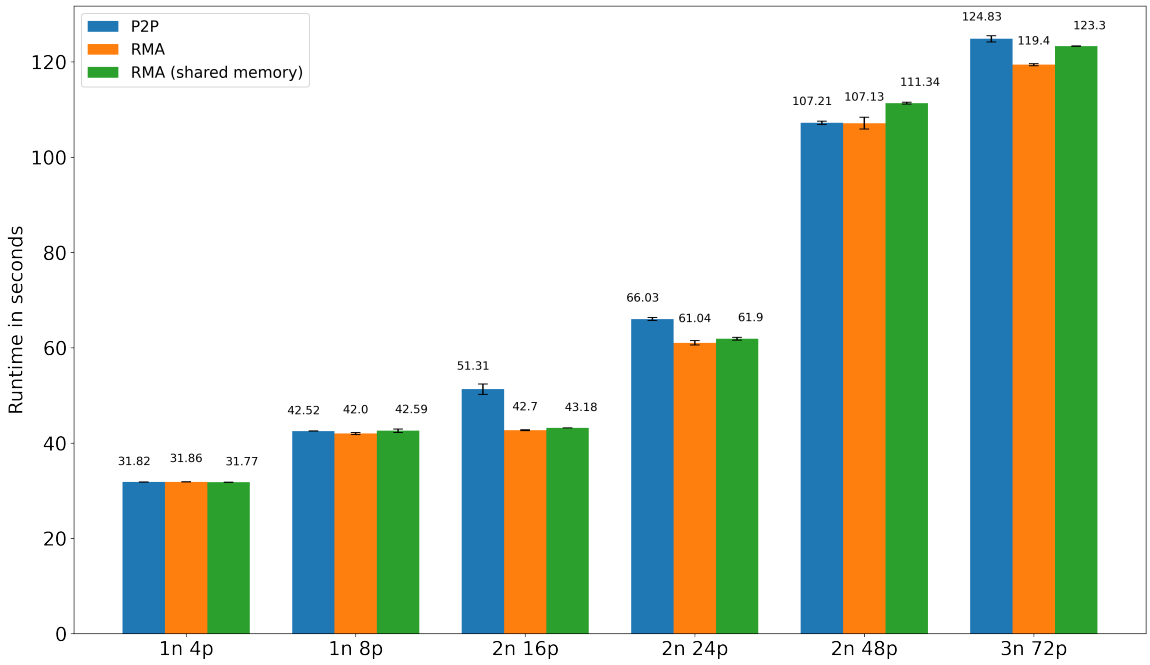


Figure 6.18: Runtimes of the different Partdiff versions using Jacobi's method with weak scaling and 5000 iterations on Mistral

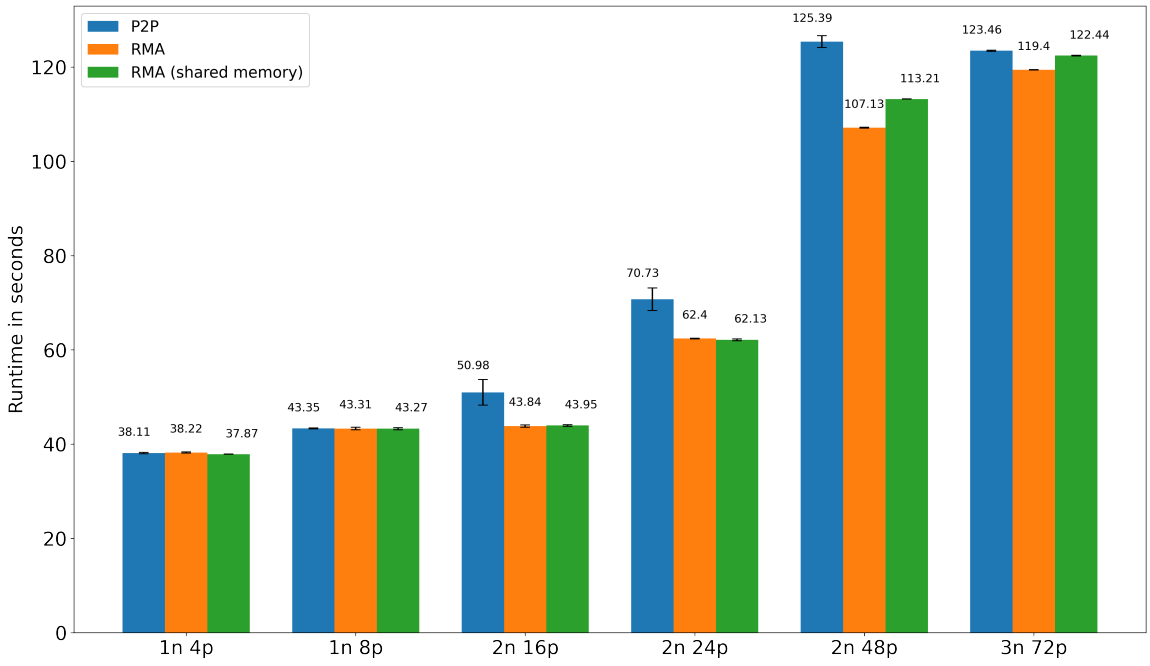


Figure 6.19: Runtimes of the different Partdiff versions using Jacobi's method with weak scaling and a varying precision that requires 100 iterations to reach on Mistral

on three of the four configurations that use more than one node. The shared memory version also appears to perform worse when subjected to higher process numbers and problem sizes, but only once subjected to 48 or more processes and their respective problem size. The non-shared RMA version however seems to perform the best when subjected to increasing problem sizes, as the respective runtimes are at least equal to the other two versions, or even smaller than the others with regards to the last configuration.

The previous statements regarding the weak scaling experiment with iterations as the termination condition also apply to the usage of precision as the condition, as displayed in Figure 6.19. All versions perform equally well on a single node, but an increase in process numbers and problem size negatively affects both the P2P version and the shared memory version.

In conclusion, when running on Mistral, the Partdiff implementation of Jacobi's method that performs the best is the non-shared RMA version. This difference is only noticeable on larger problem sizes though, as the runtimes for both the P2P version and the RMA version were equal during the strong scaling experiments, but diverged on larger problem sizes during the weak scaling experiments. The shared memory version performed at best comparably to the non-shared RMA version, however it could not keep up on all tested configurations, which leaves the non-shared RMA version as the superior one.

Note that the results from the weak scaling experiments do not feature a configuration with 96 processes across 3 nodes. This is due to a lack of memory on the Mistral nodes used for these experiments. As outlined in Table 5.1, the local matrix consists of 1 GB of data per process for this particular weak scaling experiment. This number is doubled when using Jacobi's method, as two matrices are necessary to conduct the calculations. This leads to 64 GB of matrix data per node, which is barely too much for the memory available.

6.3.2 Gauss-Seidel method on Mistral

The results from the previous experiments involving the Gauss-Seidel method indicated that the P2P version performs the best on small problem sizes, whereas shared memory achieves the best results on large problem sizes out of the tested versions. There was an outlier though, as the non-shared RMA version performed the best when using precision as the termination condition on large problem sizes.

Some of these previous results are also observable on Mistral. As it was the case with the former strong scaling experiment, the best runtimes are achieved by using point-to-point communication (see Figure 6.20). Shared memory performs equally on a single node, however increasing the number of nodes involved does not improve the performance of the shared memory version as much as it improves the P2P version's. Note that the non-shared memory RMA version achieves the worst results, especially on low process and node numbers.

Similar results are produced when precision serves as the termination condition, as

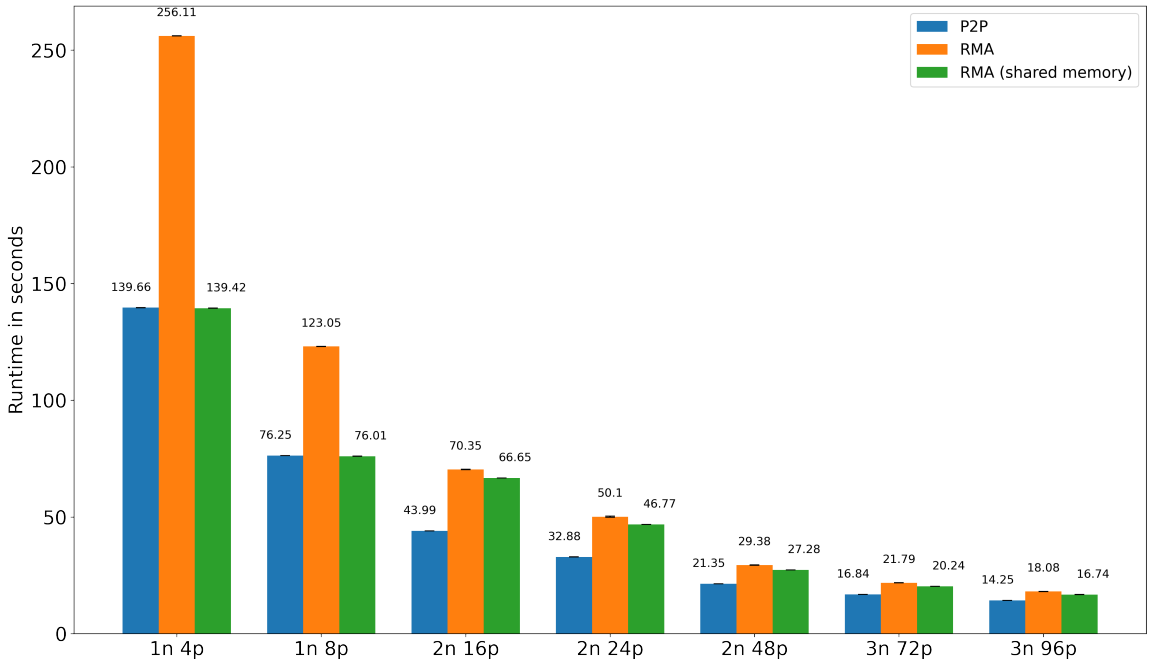


Figure 6.20: Runtimes of the different Partdiff versions using the Gauss-Seidel method with 4105 lines and 5000 iterations on Mistral

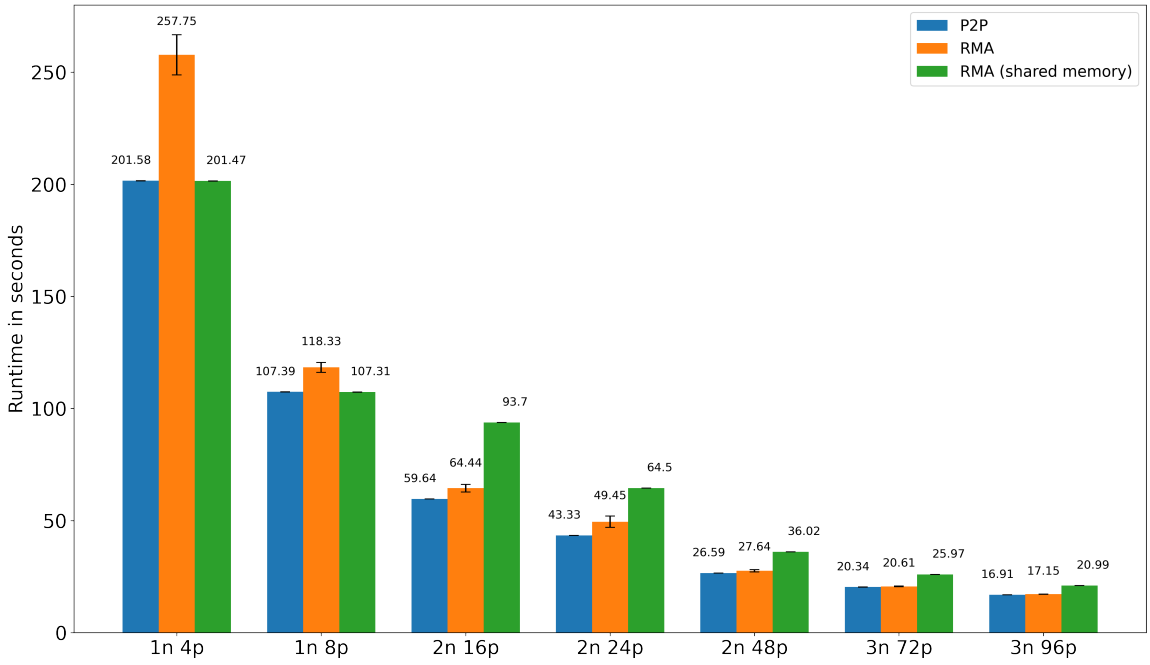


Figure 6.21: Runtimes of the different Partdiff versions using the Gauss-Seidel method with 4105 lines and a target precision of 7.0637e-5 on Mistral

displayed in Figure 6.21. Using point-to-point achieves the lowest runtimes on every

tested configuration or is at least on par with the other tested versions. Once again shared memory achieves similar runtimes on a single node, but increasing the number of nodes also increases the gap between the P2P and the shared memory version. The non-shared RMA version still performs the worst when using a single node, but is able to catch up significantly while using higher node and process numbers, as it produces similar results when compared to the P2P version on 48 processes and up.

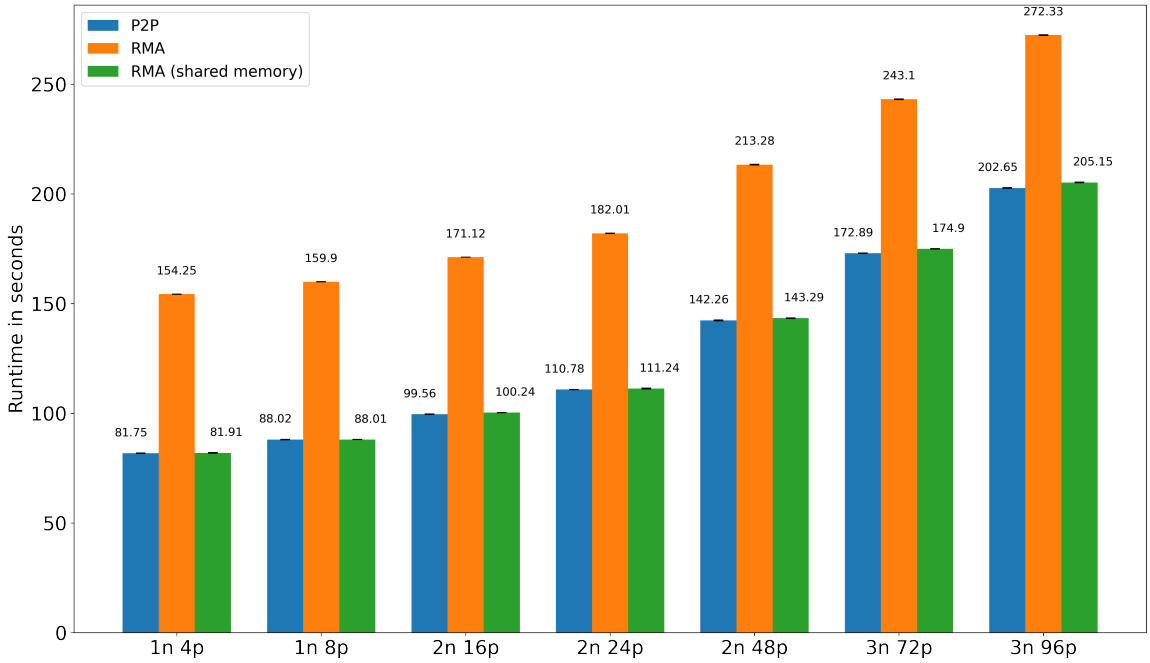


Figure 6.22: Runtimes of the different Partdiff versions using the Gauss-Seidel method with weak scaling and 5000 iterations on Mistral

Applying weak scaling when using the number of iterations as the termination criterion allows for the same observation that was made during the former weak scaling experiment with the Gauss-Seidel method. Referring to Figure 6.22, both point-to-point and shared memory achieve similar runtimes, closing the gap that was observed in Figure 6.20. This indicates that shared memory scales better with an increasing problem size and could possibly overtake the P2P version on even larger problem sizes than the ones used for these particular experiments. As before, using regular RMA without shared memory achieves the worst results by far.

When using the other termination condition, a similar effect cannot be observed, as the results displayed in Figure 6.23 show that the P2P version performs the best on all occasions. The difference to its competitors is comparatively small on low process numbers, but as both the problem size and the number of processes increases, so does the difference in runtime.

In conclusion, when using the number of iterations as the terminating factor, the results

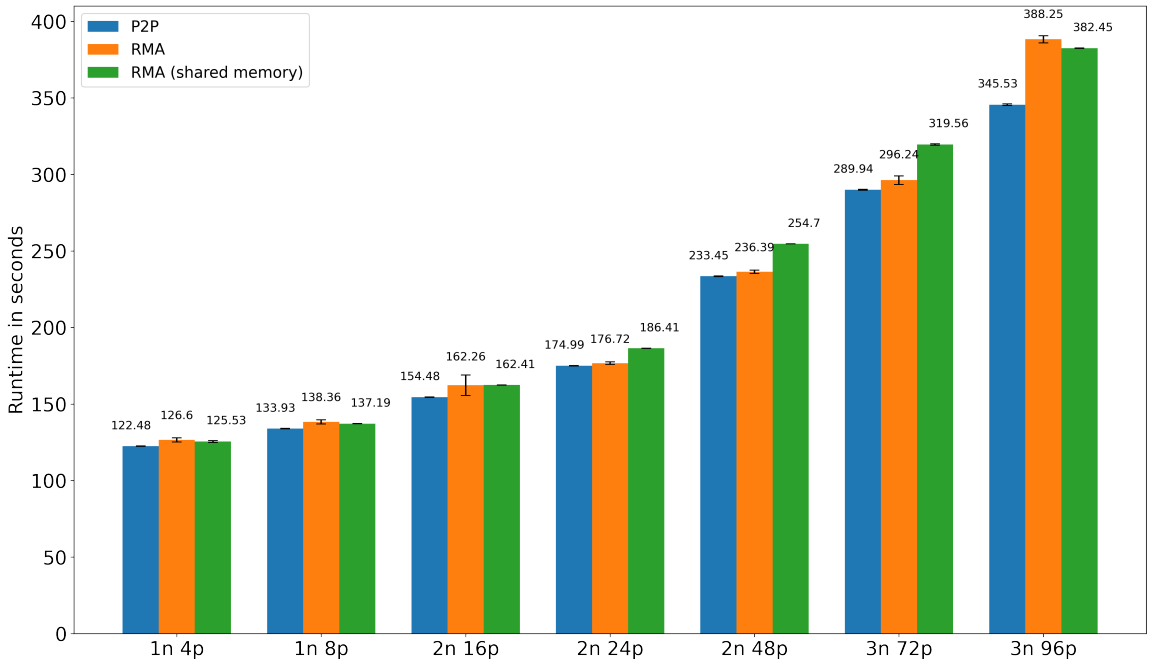


Figure 6.23: Runtimes of the different Partdiff versions using the Gauss-Seidel method with weak scaling and a varying precision that requires 100 iterations to reach on Mistral

from the Mistral experiments are similar to the results from the former experiments, as the P2P version performs better on small problem sizes but is matched by the shared memory version on larger problem sizes, leading to the assumption that shared memory will overtake point-to-point on even larger problems. However when using precision as the termination condition, using point-to-point appears to be superior on both small and large problem sizes, regardless of the number of processes or nodes used, as both RMA version perform equally at best, but also considerably worse on multiple occasions.

The last experiment was conducted to determine whether or not inter- and intra-node communication perform comparably. Similar experiments were conducted on the previously used hardware, resulting in the conclusion that intra-node communication is less efficient. The experiment on Mistral utilizes the same setup, all Partdiff versions were executed with 36 processes in total and varying numbers of nodes, ranging from one to four nodes. The method used is the Gauss-Seidel method with the number of iterations serving as the termination condition. The runtimes displayed in Figure 6.24 lead to the conclusion that both inter- and intra-node communication perform at a similar level for both the P2P version and the non-shared memory RMA version. However when using shared memory, the performance decreases significantly once more than one node is involved, which is the expected result, as shared memory should work best when every process can access the necessary halo lines via local read operations from shared memory instead of having to send the necessary data via other MPI calls.

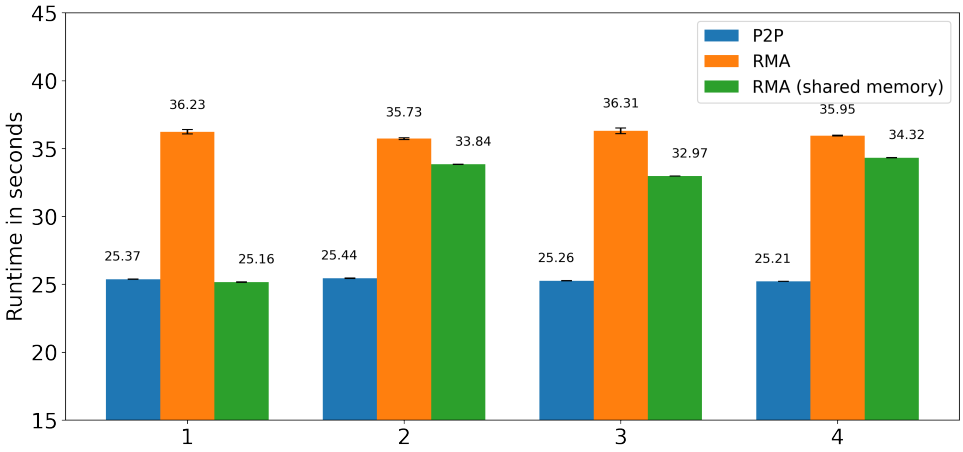


Figure 6.24: Runtimes of the different Partdiff versions using the Gauss-Seidel method with 4105 lines and 5000 iterations while using 36 processes in total on Mistral (using n nodes)

The results from the Mistral experiments are as previously expected and show that the efficiency of inter- and intra-node communication are dependent on the specific hardware in use, which explains the differing results from the experiments on the different hardware setups.

6.3.3 The ideal scenario for shared memory

Combining these observations regarding the performance of shared memory, one could deduce that running a job on a single node at maximum capacity is the ideal environment for shared memory. To determine whether or not shared memory performs better than point-to-point communication in this scenario, another experiment has been conducted on Mistral. Both the shared memory version and the P2P version of Partdiff were executed on a single node with 72 processes using Hyper Threading, executing the Gauss-Seidel method on a matrix with 8385 lines, which equates to roughly 0.5 GB of matrix data per process.

As the results in Table 6.1 show, shared memory performs better than the P2P version in this particular scenario. But once again, as soon as more nodes are involved, splitting the 72 processes across two nodes and therefore making Hyper threading obsolete as well, point-to-point communication proves to be superior in regards to runtime. So while the deduction that shared memory performs the best when using a single node at full capacity seems correct and even better than point-to-point communication, shared memory still does not seem feasible, as its performance decreases significantly compared to P2P when using more than one node. It should be noted that a similar experiment has been conducted without the use of Hyper Threading, using only 36 processes per

node, but the results remained the same.

no. of nodes	shared	P2P
1	29.14 (0.18)	31.58 (0.53)
2	29.55 (0.05)	21.66 (0.03)

Table 6.1: Runtime in seconds of shared memory and P2P version when using 72 processes in total. Standard deviations are displayed in parentheses.

7 Tracing

To further investigate the differences in runtime between the Partdiff versions, all three of the final ones were instrumented via Score-P [12] and the resulting traces were then observed with Vampir [11]. The goal is to discern whether or not the lacking performance of the shared memory version is caused by improper usage of synchronization and communication that could lead to processes being idle for extended periods of time. The traces displayed in the following are all generated by using Jacobi's method with 512 interlines and using 5000 iterations as the termination condition. To ensure that the traces are not disrupted by excessive process migration, certain flags have been used to ensure that the processes are bound to their respective core (`-bind-to core` for `mpiexec` and `--cpu_bind=cores` for `srun` respectively). Traces have been generated on both the WR cluster and Mistral.

7.1 Traces on Mistral

First, the traces generated on Mistral will be investigated. It should be noted that both the compiler and the MPI implementation used to generate the traces differ from the ones used for the previous experiments, as there was no matching Score-P version available on Mistral at the time. The previous experiments used gcc and OpenMPI 2.0.2p2_hpcx-gcc64. To generate the traces, Partdiff was built with intel 18.0.4 and openmpi/2.0.2p2_hpcx-intel14 and instrumented with scorep 4.1-openmpi-intel16.

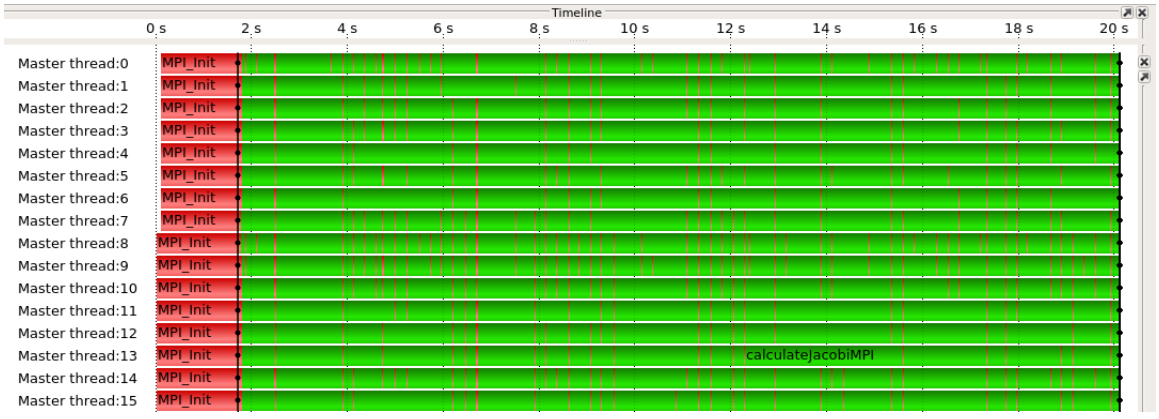


Figure 7.1: Overview of the trace data of the shared memory version on Mistral. Partdiff was executed on two nodes with 8 processes each.

The runtimes of the instrumented versions are all similar at roughly 20 seconds

(20.134 seconds for shared memory, 20.451 seconds for non-shared memory RMA). Figure 7.1 displays the overview of the traces for the shared memory version. Apart from the initialization done by `MPI_Init`, the processes are rarely waiting, as indicated by the few red lines in the graphic.

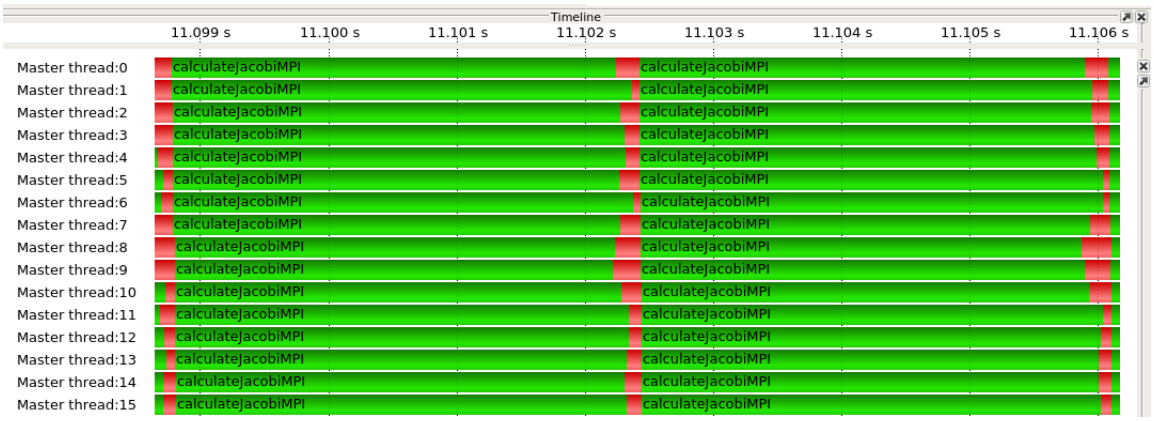


Figure 7.2: Inspecting a singular epoch of the trace data of the shared memory version on Mistral. Partdiff was executed on two nodes with 8 processes each.

Zooming closer (Figure 7.2) enables the separation into different epochs. The shared memory version used active target synchronization and therefore barrier-like fences for its synchronization. Zooming even closer would reveal that the processes with the numbers 7 and 8 communicate via `MPI_Put` just before the next epoch begins. Every other process is able to access the necessary halo lines via shared memory. Note that a singular epoch lasts roughly 0.003 seconds.

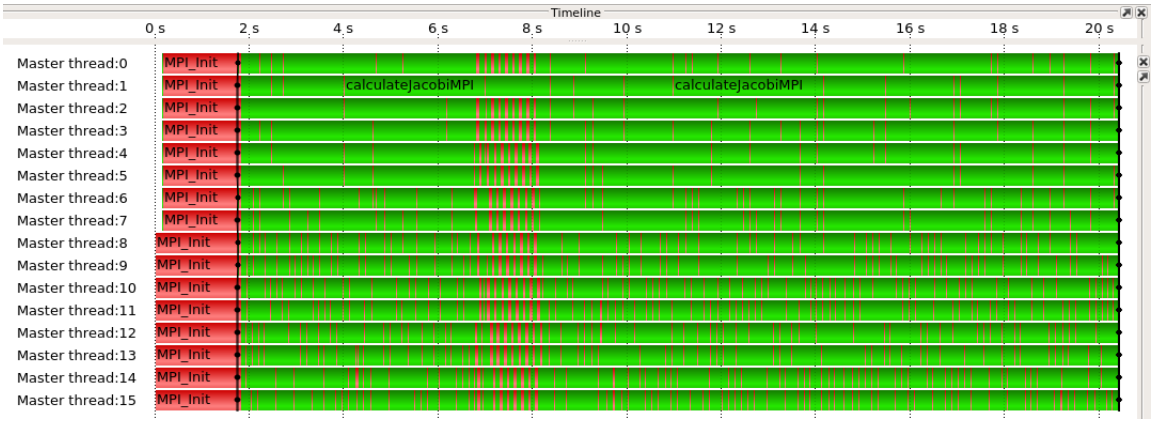


Figure 7.3: Overview of the trace data of the RMA version on Mistral. Partdiff was executed on two nodes with 8 processes each.

The global overview of the non-shared memory RMA version appears to be quite similar to the shared memory version's. The only difference being that there are a few more red lines present, indicating that processes have to wait more often. This observation

is also supported by the function summary of Vampir. According to this summary, all processes spent an accumulated 11 seconds blocked by either `MPI_Win_fence` or `MPI_Put`, while the processes of the non-shared version are blocked by `MPI_Put` and the respective synchronization functions for roughly 23 seconds. To reiterate, the non-shared memory version used general active target synchronization for the calculations with Jacobi's method. This means that the non-shared memory version's processes are blocked about 12 seconds longer than the processes of the shared memory version are. Dividing this number by the number of processes involved shows that every process is blocked for about 0.75 seconds longer when compared to those of the shared memory version. However, the non-shared memory version's processes spend less time overall to calculate the results. The shared memory version spent 282.454 seconds in the `calculateJacobiMPI` function, while the non-shared version only needed 274.271 seconds. Due to this difference during the calculation, the overall runtime between the two versions only differs by 0.3 seconds.

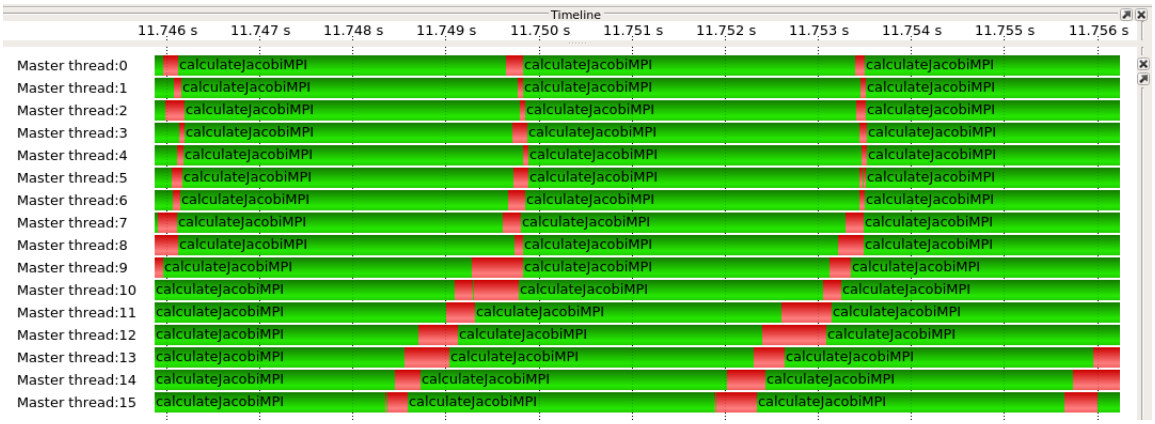


Figure 7.4: Inspecting a singular epoch of the trace data of the RMA version on Mistral. Partdiff was executed on two nodes with 8 processes each.

Zooming closer at a similar point in time as previously reveals that even though there is no global synchronization, one can still discern singular epochs in the traces, although the processes do not start a particular epoch all at the same time. Note that an epoch lasts roughly 0.003 seconds once again.

To summarize, the traces generated on Mistral are mostly in line with what is to be expected. All processes need roughly the same amount of time to complete an epoch, regardless of the synchronization mechanism and also regardless of the use of shared memory. Apparently the communication without shared memory is fast enough to produce competitive results. Only a small share of the overall runtime is spent waiting for other processes or the completion of communication operations. Note that these observations slightly differ from the ones previously made in Section 6.3.1, where the shared memory version was slower by 2 seconds on the same node setup. As previously mentioned though, the MPI implementation used was a different one. The traces of the P2P version were omitted in this section, as they are very similar to the traces of the non-shared RMA version and would not provide any additional insight.

7.2 Traces on WR cluster

The traces created on the WR cluster show different characteristics. The instrumented runs of Partdiff were executed under the same conditions as before, barring the different MPI implementation (MPICH), compiler (gcc) and general hardware. In this section, only the traces from the shared memory version will be displayed, as the others produced such similar traces that analyzing each one of them independently would be redundant. An overview of the first traces is displayed in Figure 7.5.

Figure 7.5: Overview of the trace data of the shared memory version on WR cluster. Partdiff was executed on two nodes with 8 processes each.

Compared to the overview of the Mistral traces, multiple things differ greatly. Not only is the overall runtime more than four times as long as it was on Mistral, but the processes spent much more time being blocked, as indicated by the red marks. The black dots on processes 7 and 8 are visualizations of `MPI_Put`, as these two processes are on the borders of their respective shared memory communicator. There also seems to be a pattern in this overview, a right-pointing arrow. Starting on processes 2 and 13, some iterations take much longer than the same iterations on other processes. These longer lasting iterations seem to wander towards the processes in the middle over the course of the 5000 iterations, until they reach processes 7 and 8 at the end. As previously mentioned, the traces for the non-shared memory RMA version and the P2P version also display such a pattern, leading to the assumption that this pattern is caused by the data of the calculation.

When executing Partdiff as it has been for this particular run, the underlying matrix is symmetrical at the start. The borders of the matrix contain values between 0 and 1, while every other entry of the matrix is set to 0. In the beginning, only the first and the last process have comparatively complicated calculations to make, while the other processes mostly add zeroes together. As the iterations go by though, these more challenging calculations wander towards the middle from the top and the bottom. If one was to visualize this movement over time, a similar arrow-shaped pattern would be the result. Therefore, the assumption that this pattern is caused by the composition of the

underlying matrix seems plausible.

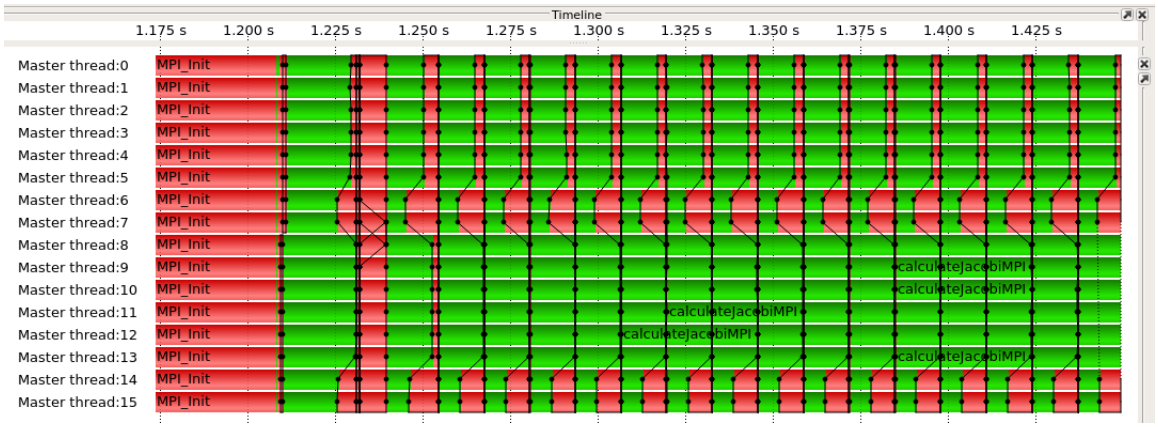


Figure 7.6: Traces of the first iterations of the shared memory version on WR cluster. Partdiff was executed on two nodes with 8 processes each.

A closer look at the traces of the first iterations (Figure 7.6) reveals a rather strange pattern. Processes 0-5 and 8-13 take more time to complete a single iteration than the processes 6, 7, 14 and 15. Partdiff was executed on two nodes with 8 processes each. This phenomenon can be observed for multiple iterations, going further beyond the depicted iterations in said figure. This pattern is most likely caused by the underlying hardware.

Each node consists of an AMD Opteron Processor 6344, as previously mentioned. This processor contains four sockets and 12 cores per socket. Furthermore, it consists of eight NUMA nodes. The combination of the observed pattern and the hardware details leads to the following assumption. Processes 0-5 are executed on the first NUMA node. The other two processes, process 6 and 7, are executed on the second NUMA node, meaning that there are four less processes than the node could potentially run. Due to this circumstance, the AMD Turbo Core Technology (ATC) [3] boosts the frequency of the cores in use, which in turn leads to the faster completion of the iterations compared to the other processes. This theory is based on the assumption that each NUMA node is considered as a separate unit for the purposes of ATC. Other traces for a different process configuration further fortify this theory. These traces will be discussed later on. For now though, the previously displayed traces will be investigated further.

After roughly half of all iterations are completed, the traces result in the patterns shown in Figure 7.7. Processes 5 and 10 need much longer to complete their iterations than the others. These spikes align with the previous theory, stating that they are caused by the underlying matrix data. However, process 10 seems to consistently take more time than process 5. This phenomenon has also been observed on the other versions. This pattern was also reproducible on different nodes with the same hardware and also on nodes with different hardware that were available on the WR cluster, consisting of an Intel Xeon CPU X5650 per node. Since these nodes consist of Intel hardware, the

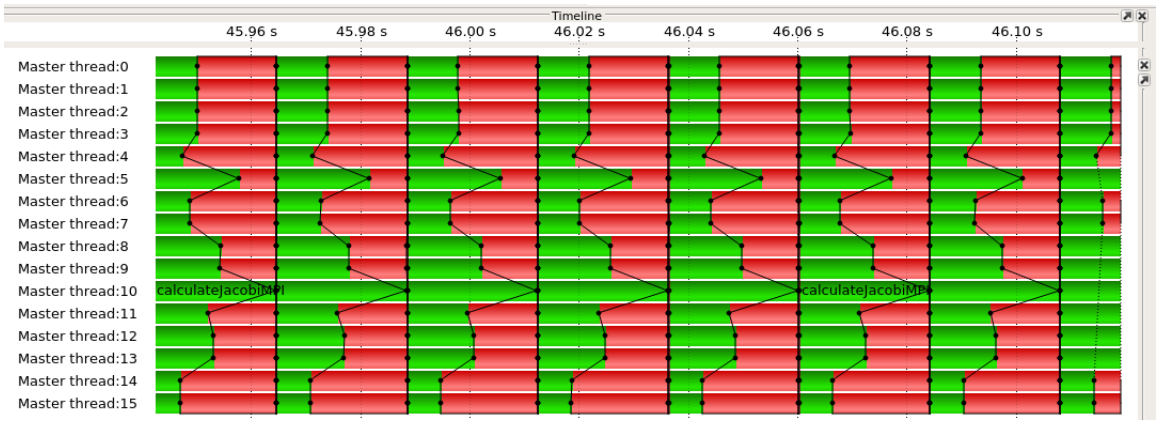


Figure 7.7: Traces of the middle iterations of the shared memory version on WR cluster. Partdiff was executed on two nodes with 8 processes each.

underlying boosting technology differs as well (Intel Turbo Boost). This consistency leads to the assumption that this phenomenon is not coincidental, however there is currently no explanation available.

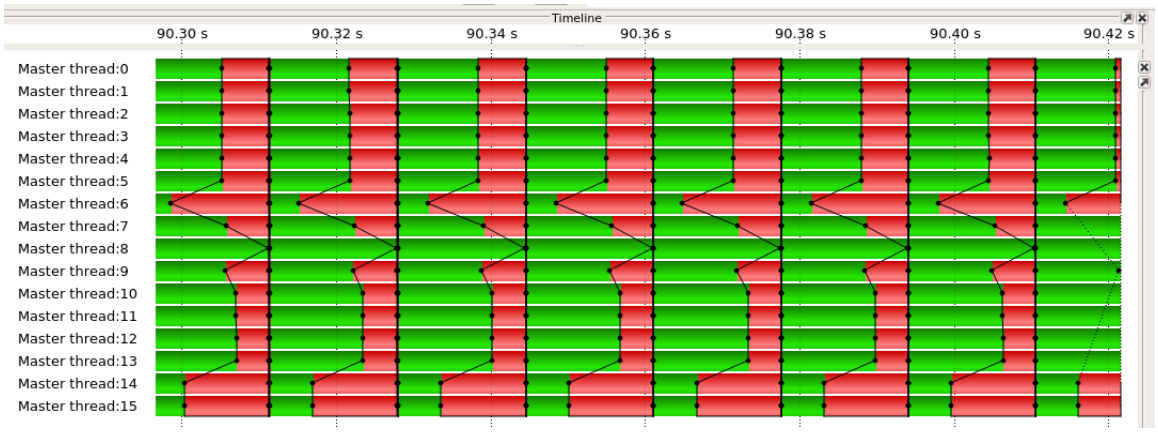


Figure 7.8: Traces of the final iterations of the shared memory version on WR cluster. Partdiff was executed on two nodes with 8 processes each.

Back to the AMD cores, the traces of the iterations close to completion once again fortify the theory regarding the arrow-like pattern. Process 8 takes considerably longer to complete its iteration. According to this theory, process 7 should also take much longer to complete its iterations, but due to the aforementioned ATC, the calculations are finished considerably faster in comparison to process 8. Comparing processes 6 and 7 confirms this, as these processes have previously both been on par in regards to their time per iteration. The only other exception to this was a few seconds prior to the traces displayed in Figure 7.8, during which process 6 took longer to complete its iterations, which is also in line with the theory regarding the arrow-like pattern.

As previously mentioned, other traces have been generated that further fortify the assumption regarding the impact of ATC. To confirm whether or not the pattern at the start (first 6 processes need more time than the last two per node) is caused by ATC, the instrumented version of Partdiff has been executed with 24 processes across two nodes. This ensures that two NUMA nodes are fully utilized on both nodes.

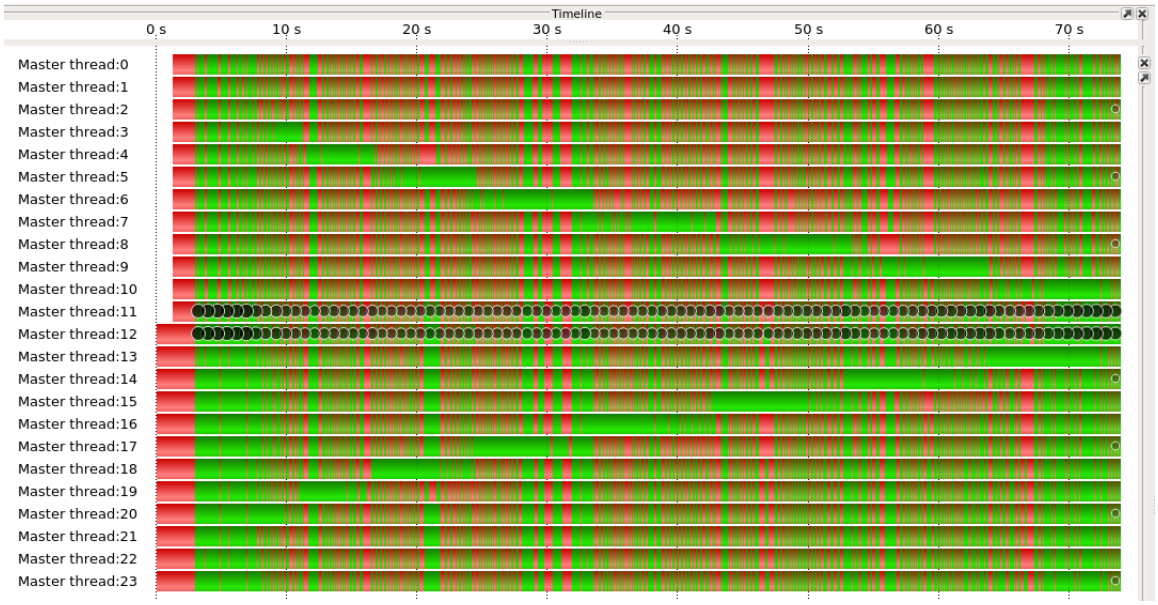


Figure 7.9: Overview of the trace data of the shared memory version on WR cluster. Partdiff was executed on two nodes with 12 processes each.

The overview once again reveals the arrow-shaped pattern. When executing Partdiff as such, the processes 11 and 12 are on the borders of their respective shared memory communicator, hence they communicate their halo lines via `MPI_Put`.

A closer look at the first iterations seems to confirm that ATC was the reason for the difference between the first 6 and the last two processes on each node. Fully utilizing both NUMA nodes instead of under-utilizing one negates the previously observed effects. Now, all processes that are executed on the same node need almost the exact same amount of time for an iteration at the start. Considering this newly attained information, manually binding all processes when not utilizing a node at its maximum capacity could enable faster runtimes due to the boost in CPU frequency provided by ATC. While this should be investigated to minimize runtimes further, it is out of scope for this particular thesis.

Note that the differences in runtime between the two nodes that were used to generate the traces depicted in Figure 7.10 are most likely caused by the particular hardware in use, as executing Partdiff on a different pair of nodes with the same hardware led to the disappearance of said difference.

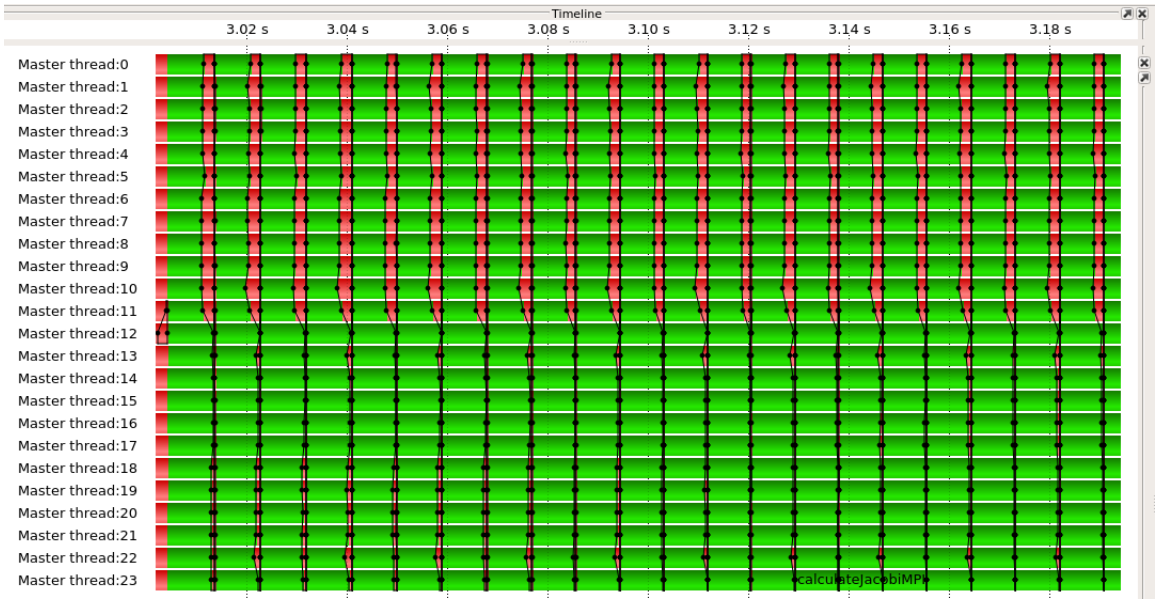


Figure 7.10: Traces of the first iterations of the shared memory version on WR cluster. Partdiff was executed on two nodes with 12 processes each.

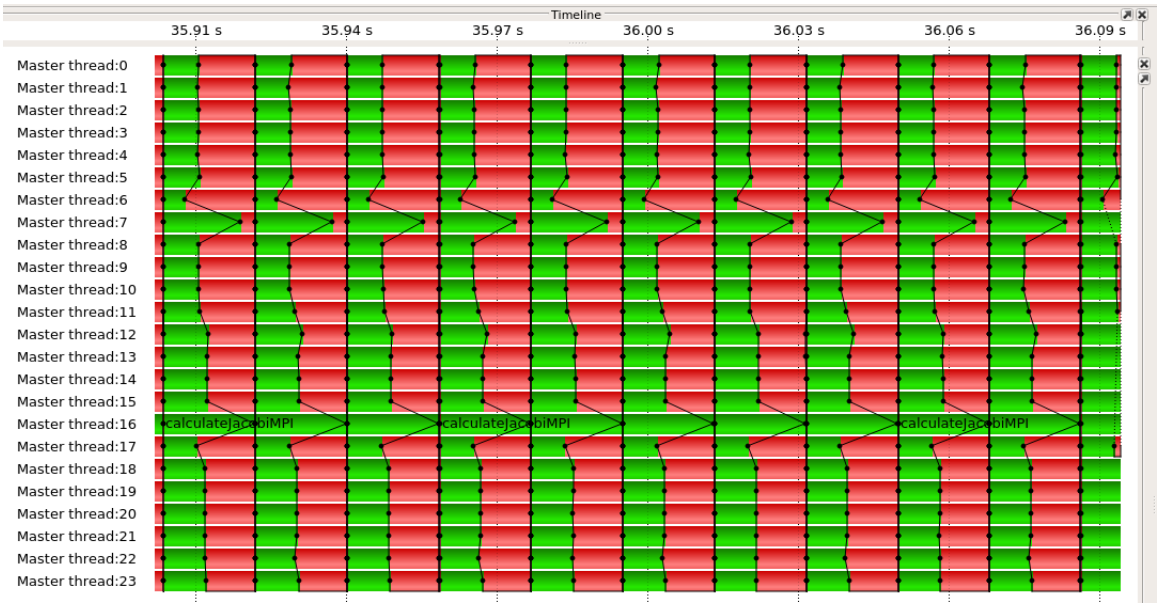


Figure 7.11: Traces of the middle iterations of the shared memory version on WR cluster. Partdiff was executed on two nodes with 12 processes each.

The iterations in the middle once again show two processes that need longer than the others to complete their calculations, as displayed in Figure 7.11. This is in accordance with the theory regarding the underlying matrix as the cause once again. Note that, as before, the first of these two processes needs slightly less time than the latter one. This effect did not disappear when using a different pair of nodes, unlike the difference in Figure 7.10.

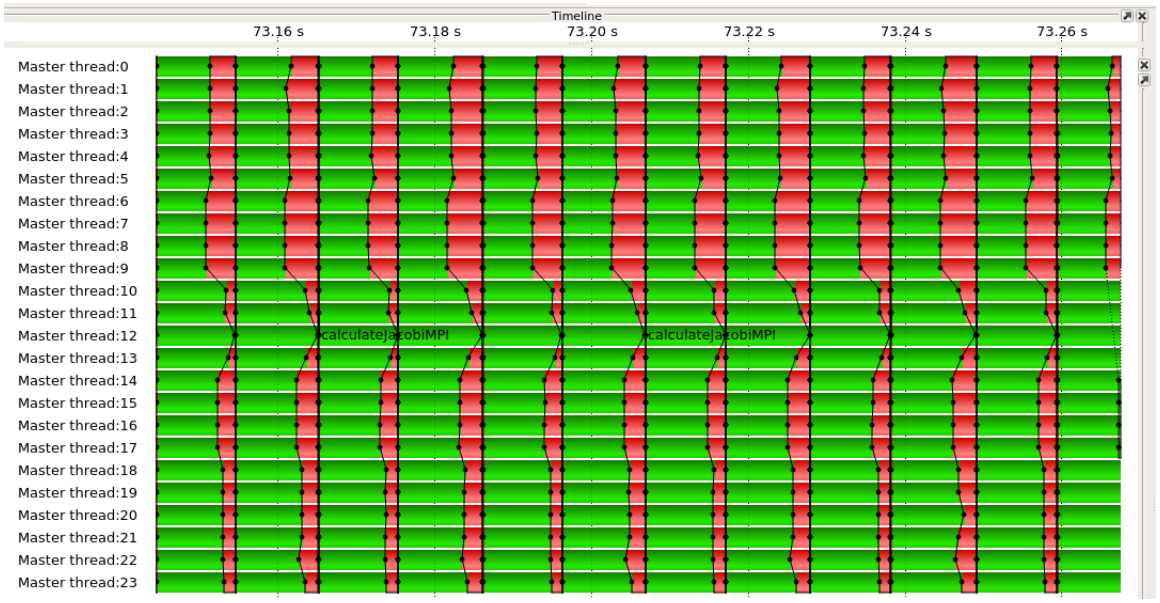


Figure 7.12: Traces of the final iterations of the shared memory version on WR cluster. Partdiff was executed on two nodes with 12 processes each.

The final iterations once again act in accordance with the previously stated theory that the nature of the underlying matrix and the particular calculations performed on it lead to more difficult calculations wandering from the outside towards the middle, resulting in the spike in time taken per iteration for processes 10-13. With this much evidence, the theory regarding the arrow-like shape of the traces will be considered to be true. However, given that this pattern did not appear on Mistral, either the specific hardware or the MPI implementation used on the WR cluster is responsible for said pattern. While this should be investigated further, for example by using different MPI implementations, it is once again out of scope for this particular thesis.

In conclusion, generating traces of Partdiff and investigating them did not lead to any new discoveries as to why the shared memory version performs so much worse than anticipated, as the trace data for all three Partdiff versions was very similar. However, the traces did once again reinforce that the hardware used has a major impact on multiple aspects of performance. Execution on Mistral led to very little time blocked and an even pattern throughout the entire run, while the execution on the WR cluster led to much more blocking and also to an arrow-like shape of all traces caused by the particular calculations and the structure of the matrix. It also revealed that undersubscribing to a node and only using a few particular cores in order to boost the frequency of the CPU could be an interesting avenue to further increase performance and should certainly be further investigated at another time.

8 Conclusion

8.1 Limitations

The results from the experiments are not universally applicable to all algorithms and their performance with MPI-3 RMA, as the experiments only involved a PDE solver that uses a 5-point stencil with one of two algorithms, namely the Gauss-Seidel method or Jacobi's method. Therefore, no statements about other algorithms can be made. Furthermore, it is possible that the tested implementations of Partdiff are not optimal, as there might be potentially missed hints or other techniques regarding MPI that were not used. The general C code of Partdiff outside of the MPI function calls is most likely not optimal either, as there are certain keywords (like `restrict`) that were not used, but they could potentially increase performance in some shape. This also applies to the comparisons of the different RMA versions, as the tried methods to increase performance might not have been optimally executed either, whereas other approaches might have been left out entirely.

There are limitations regarding the measurements as well. Each measurement has only been taken three times. Increasing this number would lead to more precise results and also allow for the proper calculation of confidence intervals, but doing so was not feasible with many runs taking more than 10 minutes to complete once. The measurements have also not been taken at the same time of day, on the exact same nodes and with the exact same usage of the entire cluster either, affecting both the results and reproducibility.

And lastly, the experiments only involved two MPI implementations. The results might be vastly different for other MPI implementations. The same is true for the hardware, as only two different sets of hardware were involved in testing. Considering that the results already varied significantly depending on the hardware and MPI implementation used, the assumption that using different hardware or a different implementation would produce different results yet again seems reasonable.

8.2 Final conclusion

In this thesis, the performance of traditional MPI point-to-point communication was compared to the MPI-3 RMA approach. For this, a solver for partial differential equations, using either the Gauss-Seidel method or Jacobi's method, was implemented in various ways. To create the best RMA version possible, multiple different approaches regarding the specific implementation have been tested. These approaches differed mainly in their employed synchronization mechanism.

Initial experiments on these versions suggested that the different mechanisms achieve different runtimes on the tested application. These versions have then been refined further and put through similar experiments once again, leading to the first final RMA-based version of Partdiff, which employed general active target synchronization for both the Gauss-Seidel method and Jacobi's method, as this synchronization mechanism performed the best during the experiments.

A second RMA version utilizing shared memory was created as well. As the communication between two communicators that are not able to utilize shared memory together has to be implemented with either P2P or the standard RMA communication calls, the same experiments have been conducted on the shared memory approach as well. Multiple versions using the different synchronization mechanisms were tested in said experiments, leading to the final version employing active target synchronization for Jacobi's method, while using passive target synchronization for the Gauss-Seidel method.

These final RMA versions were then tested in extensive weak and strong scaling experiments, comparing the resulting runtimes to those of a P2P version of Partdiff. During these experiments, no clear superior approach was found. For the most part though, the shared memory version performed worse than the other two and therefore stays vastly behind previous expectations regarding the performance of shared memory.

The P2P version and the non-shared memory RMA version performed better on most occasions. However regarding these two versions, none outperformed the other one consistently. Considering that these differences were also not consistent across different sets of hardware and MPI implementations, no universal statement regarding a superior approach can be made. Therefore, the best communication form for any given application or kernel has to be determined through a case-by-case approach with multiple experiments.

Bibliography

[1] Ayon Basumallik and Rudolf Eigenmann. Towards automatic translation of OpenMP to MPI. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 189–198, 2005.

[2] OpenMP Architecture Review Board. *OpenMP Application Programming Interface version 5.1*, 2020. `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf`.

[3] Alexander Branover, Denis Foley, and Maurice Steinman. AMD Fusion APU: Llano. *IEEE Micro*, 32(2):28–37, 2012.

[4] Nick Brown, Michael Bareford, and Michèle Weiland. Leveraging MPI RMA to optimize halo-swapping communications in MONC on Cray machines. *Concurrency and Computation: Practice and Experience*, 31(16):e5008, 2019.

[5] Piotr Dziurzanski, Wlodzimierz Bielecki, Konrad Trifunovic, and M Kleszczonek. A system for transforming an ANSI C code with OpenMP directives into a SystemC description. In *2006 IEEE Design and Diagnostics of Electronic Circuits and systems*, pages 151–152. IEEE, 2006.

[6] Message Passing Interface Forum. *MPI 2.0 documents*, 1997. `https://www.mpi-forum.org/docs/mpi-2.0/mpi2-report.pdf`.

[7] Message Passing Interface Forum. *MPI 3.1 documents*, 2015. `https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`.

[8] Torsten Hoefler. MPI remote memory access programming (MPI3-RMA) and advanced MPI programming. `http://hpac.cs.umu.se/teaching/pp-18/Hoefler.pdf`. Presented at RWTH Aachen, Jan. 2019, last access on 05.01.2021.

[9] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. MPI+ MPI: a new hybrid approach to parallel programming with MPI plus shared memory. *Computing*, 95(12):1121–1136, 2013.

[10] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. Remote memory access programming in MPI-3. *ACM Transactions on Parallel Computing (TOPC)*, 2015.

[11] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The Vampir Performance Analysis Tool-Set. In Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing*, pages 139–155, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[12] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir. In Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M Resch, editors, *Tools for High Performance Computing 2011*, pages 79–91, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[13] Sameer Kumar and Michael Blocksome. Scalable MPI-3.0 RMA on the Blue Gene/Q supercomputer. In *Proceedings of the 21st European MPI Users' Group Meeting*, pages 7–12, 2014.

[14] Okwan Kwon, Fahed Jubair, Rudolf Eigenmann, and Samuel Midkiff. A hybrid approach of OpenMP for clusters. *ACM SIGPLAN Notices*, 47(8):75–84, 2012.

[15] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, 44(4):101–110, 2009.

[16] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes in C*. Cambridge University Press Cambridge, 1988.

[17] Jannek Squar, Tim Jammer, Michael Blesel, Michael Kuhn, and Thomas Ludwig. Compiler Assisted Source Transformation of OpenMP Kernels. In *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 44–51. IEEE, 07 2020.

[18] Erich Strohmaier, Horst Simon, Martin Meuer, and Jack Dongarra. TOP500 list. `https://www.top500.org/lists/top500/list/2020/11/`, November 2020.

[19] Mellanox Technologies. *MellanoX Messaging Library User Manual Rev 3.0*, 2014. `https://www.dkrz.de/en-pdfs/en-docs/en-docu-mistral/en-Mellanox_MXM_User_Manual.pdf`.

[20] Ohio State University. OSU micro-benchmarks. `https://mvapich.cse.ohio-state.edu/static/media/mvapich/README-OMB.txt`. last access on 16.02.2021.

[21] Chenle Yu, Sara Royuela, and Eduardo Quiñones. OpenMP to CUDA graphs: a compiler-based transformation to enhance the programmability of NVIDIA devices.

In *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems*, pages 42–47, 2020.

[22] Huan Zhou, Kamran Idrees, and José Gracia. Leveraging MPI-3 shared-memory extensions for efficient PGAS runtime systems. In *European Conference on Parallel Processing*, pages 373–384. Springer, 2015.

Reproducibility

This section provides all necessary information to reproduce the results from the previously conducted experiments. In its current state Partdiff consists of three files, namely partdiff.c, partdiff.h and askparams.c. The Makefile for the compilation is displayed in Listing 1.

```
1  CC = mpicc
2
3  CFLAGS = -std=c11 -Wall -Wextra -ggdb -O3
4  LIBS = -lm
5
6  TGTS = partdiff
7  OBJS = partdiff.o askparams.o
8
9  all: partdiff
10
11 partdiff: $(OBJS) Makefile
12     $(CC) $(CFLAGS) -o $@ $(OBJS) $(LIBS)
13
14 partdiff.o: partdiff.c Makefile
15
16 askparams.o: askparams.c Makefile
17
18
19 %.o: %.c
20     $(CC) -c $(CFLAGS) $*.c
21
22
23 clean:
24     $(RM) $(OBJS)
25     $(RM) $(TGTS)
```

Listing 1: Makefile for Partdiff

The job scripts for the experiments differ depending on the platform. The first set of experiments was conducted on the cluster provided by the Scientific Computing research group at Universität Hamburg. An example of a job script is provided in Listing 2. The parameters passed to Partdiff and the number of nodes and processes used differ depending on the specific experiment. The compiler used on this cluster is gcc 9.3.0 and

the MPI implementation is MPICH 3.3.2, as previously mentioned.

```
#!/bin/bash

#SBATCH --time=02:00:00
#SBATCH --partition=abu
#SBATCH --nodes=2 --tasks-per-node=8
#SBATCH --error=Strong_GS_Iter_2_16.err
   ↪ --output=Strong_GS_Iter_2_16.out

mpiexec -n 16 ./partdiff 1 1 512 1 2 5000
mpiexec -n 16 ./partdiff 1 1 512 1 2 5000
mpiexec -n 16 ./partdiff 1 1 512 1 2 5000
```

Listing 2: Example job script for the Scientific Computing cluster

The second set of experiments was conducted on Mistral, the high performance computing system of the DKRZ. The job scripts for this hardware set up the environment for the proper use of MellanoX Messaging before any computation is started. The stack size and core file size are also set beforehand. The MPI implementation that was used on this hardware is OpenMPI 2.0.2p2_hpcx-gcc64, a high-performance implementation of OpenMPI. The compiler that was used is gcc 9.1.0.

```
#!/bin/bash

#SBATCH --time=02:00:00
#SBATCH --partition=compute2
#SBATCH --nodes=2 --tasks-per-node=8
#SBATCH --error=Strong_GS_Iter_2_16.err
   ↪ --output=Strong_GS_Iter_2_16.out
#SBATCH --account=*******

ulimit -s 102400
ulimit -c 0

module load gcc/9.1.0-gcc-7.1.0
module load openmpi/2.0.2p2_hpcx-gcc64

export OMPI_MCA_pml=cm
export OMPI_MCA_mtl=mxm
export OMPI_MCA_mtl_mxm_np=0
export MXM_RDMA_PORTS=mlx5_0:1
export MXM_LOG_LEVEL=ERROR

export OMPI_MCA_coll=^ghc

```

```
23 srun --propagate=STACK,CORE ./partdiff 1 1 512 1 2 5000
24 srun --propagate=STACK,CORE ./partdiff 1 1 512 1 2 5000
25 srun --propagate=STACK,CORE ./partdiff 1 1 512 1 2 5000
```

Listing 3: Example job script for Mistral

List of Figures

List of Listings

List of Tables