Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Andreas Hammer

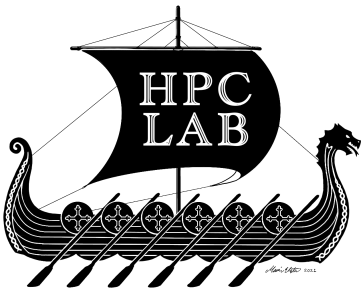
Analyzing Halo Computations on Multicore CPUs

Master's thesis in Computer Science
Supervisor: Professor Anne C. Elster
June 2021

NTNU
Norwegian University of
Science and Technology

Andreas Hammer

Analyzing Halo Computations on Multicore CPUs



Master's thesis in Computer Science
Supervisor: Professor Anne C. Elster
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Problem description

This master thesis project builds on the student pre-project (a.k.a. "fall" porject) “Study of Two-dimensional Deep Halo Exchange Using MPI blocking and non-blocking communication” which studied computing a parallelized Finite Difference 5-point stencil using deeper halos to reduce communication overhead. This concept was initially presented by Robin Holtet and Anne C. Elster, at SC as well as in Mr. Holtet´s 2003 Master thesis "Communication-reducing Stencil-based Algorithm and Methods".

In this thesis, we will investigate extending the student´s work to 3D. Streaming techniques and/or higher degree stencils are also possible directions. The benefits of deeper haloes will may also be investigated.

If time permits, the work may also applying the work to a real application in collaboration with other partners at HPC-Lab. Which exact path to be taken depends on a literature search of what was done previously and the intermediate results achieved, but the goal is to produce work good enough for a conference workshop paper.

Assignment given: 15. January 2021.
Advisor: Professor Anne C. Elster, IDI

Abstract

The continued increase in multi-core and multi-node system performance makes parallelization of algorithms necessary in order to get the best performance gains by utilizing the available CPU resources. These increasingly complex systems allow for larger datasets to be handled in parallel in order to improve performance. Stencil-based algorithms with halo exchange are typically a compute and communication heavy task benefitting from parallelization over multiple nodes. This is achieved since compute can be done between bulk communications.

This thesis focuses on implementing a 3D stencil computation benchmark using halo exchange for inter-node communication. The goal of the thesis is to benchmark and test different aspects of halo exchange-powered stencil computations. We test various optimizations to improve the computational intensity of the algorithm, observe how strong and weak scaling of the algorithm increases throughput, how halo exchange is hidden by computation, and how running on a shared cluster affects the exchange.

Our optimizations gave a total speed-up of 11.15, with stencil unrolling showing an additional 1.40x speed-up potential. Scaling showed potential with a speed-up of 9.46x from 1 to 12 nodes on the same problem size and 0.61 times performance gain when scaling both problem and node size equally. Furthermore, only 7-91% of the halo exchange was hidden with shared resources. However, node exclusivity resulted in 902% of the exchange hidden and 529% hidden for a large problem domain.

Our results show that with exclusive node access, halo exchange can be completely hidden by computation when doing a triple loop that results in a 27-point calculations. Scaling the node count will improve performance to some degree. However, the speed-up is not proportional to the nodes added as the compute loop is not a perfect parallel region.

In order to exploit the extra performance from multi-core clusters for stencil computations, exclusive node access is needed to maximize performance. Increasing the amount of resources will increase throughput, but strong and weak scaling shows that the throughput increase is not proportional to the added resources.

A discussion of how to optimize the 27-point calculations and several ideas for future work are also included.

Sammendrag

Den fortsettende ytelses økningen for flerkjerne- og flernode systemer, krever at algoritmer parallelliseres for å utnytte ytelsen til slike systemer. Beregninger på store datasett gjøres raskere i slike systemer, da en kan raskere beregne flere verdier i parallell. Parallellisert stensil beregninger er en både regne- og kommunikasjonstung metode. Parallelliseringen introduserer oppspaltning av datasettet, som igjen krever kommunikasjon mellom nodene for at beregningene skal gi rett svar. Fordelen er at mellom hver iterasjon i metoden kreves det synkronisering, men under selve iterasjonen er det ingen kommunikasjon som hemmer beregningene, noe som er fordelaktig for parallelliseringen. Dette prosjektet implementerer en 3D-stensil-beregnings algoritme som bruke halo exchange som kommunikasjons mønster mellom nodene. Denne applikasjonen blir så brukt til å teste ulike aspekter ved halo exchangen og stensil beregningene. Vi ser på hvordan ulike optimaliseringer påvirker kjøretiden, hvor godt halo exchange blir gjemt av kjøretiden, hvordan algoritmen skalerer, samt hvordan kjøring på et ressursdelte maskiner påvirker kommunikasjonen.

Vi observerte en ytelses forbedring på 11.15x ved endring av datastruktur og kalkulering av minnehopp i forkant. Videre så vi at stensilutrulling ga en ytterligere forbedring på 1.40x. Skaleringen viste en "strong scaling" på 9.46x forbedring fra 1 til 12 noder, samt en "weak scaling" på 0.61x ved proporsjonal økning av antall noder og datasettets størrelse. I de fleste kjøringene ble kommunikasjonen kun 7% av kommunikasjonen gjemt under kjøring på disse delte ressursene, uten eksklusiv tilgang til disse. Da vi introduserte eksklusiv tilgang til nodene så vi 902% av kommunikasjonen gjemt, og 529% gjemt for det store datasettet fordi her må betydelig mer kommuniseres mellom nodene.

Dette viser i all hovedsak at eksklusiv tilgang på ressursene er den største bidragsyteren for å skjule kommunikasjonen. Hvis en annen bruker blir tildelt de resterende ressursene på samme maskin kan dette føre til kniving om kommunikasjonsressursene og dermed øke ventetiden for hvert kommunikasjonssteg. Videre så vi at oppskalering av antall noder og datasett størrelse økte mengden data prosessert per sekund, men ikke proporsjonalt med ressursøkningen som ble tildelt. For å utnytte den økte ytelsen godt ser vi at eksklusiv tilgang på maskinene og gode optimaliseringer vil redusere kjøretiden for 3D stensil beregninger. Oppska-

lering av ressurstilgangen vil også positivt påvirke prosesseringstiden, dog ikke proporsjonelt med økningen i ressurser som våre skaleringseksperimenter viste.

Acknowledgement

First and foremost, I would like to thank my supervisor, Professor Anne C. Elster, for invaluable guidance and help throughout this thesis, giving me clear advice when the road was rough, and guiding me steadily towards the goal of this project.

I would like to express my gratitude to my fellow HPC-lab members, Ivar Andreas Helgestad Sandvik, Lars Bjertnes and Maren Wessel-Berg, for their various input and as discussion partners to help me complete this project.

I also would like to thank the HPC group at NTNU and the HPC-lab for providing me with a workspace and hardware to create my application and a great work environment with its facilities and great view. Furthermore, thanks to my department (IDI) for providing priority access to the Idun compute cluster, without this my benchmarking would not be possible.

Lastly, a special thanks to my family and friends for their supportive phone calls motivating me throughout the project. I would also like to thank Oda Steingland Skaug for her significant input in finalizing the thesis.

Contents

Figures

Tables

Code Listings

Chapter 1

Introduction

Over the last decades, there has been a shift from single-core to multi-core CPUs due to hitting the limits of processor frequency scaling. In order to continue to have performance gains, a shift towards parallel computation is needed. Compute heavy problems, such as finite difference calculations on a large problem domain, are time-consuming, and in some cases too large for a single system/-core to handle efficiently. This prompts the use of parallelization that split the problem domain among different cores and in order to reduce the compute time and potentially keep the problem domain within the bounds of each node's memory.

When the computational domain is decomposed and scattered throughout a compute cluster, and each node individually computes each iteration for their given subdomain. However, this process usually requires knowledge of the surrounding values often specified in a stencil representing how all surrounding values should affect the calculation. The edge values shared between each subdomain require a method of how to make the values beyond the subdomain border available as efficiently as possible. The most common technique is halo exchange.

This thesis analyzes 3D halo change performance through a 3D benchmarking application which we created. The finite difference method was chosen as the compute method with a stencil requiring data beyond the subdomain borders, prompting the need for halo exchange.

1.1 Contributions

The main contributions of this thesis are that we describe in detail the process of developing a benchmark application with 3D domains, and how it needs to handle the halo exchange to ensure a correct result of the calculations. We also describe some of the optimizations we did, how the performance benefits such optimizations. Further, we look at how the benchmarks scale, and how the distribution of the problem domain affects performance. The thesis also touches upon some

considerations to consider when running halo exchange compute on a shared resource cluster.

1.2 Outline

The rest of the thesis is outlined as follows:

- Chapter 2: A description of the most relevant background and previous work related to this thesis.
- Chapter 3: Detailed description of the benchmark implementation, both the compute and halo exchange implementation, and various optimizations implemented throughout the development process.
- Chapter 4: Present the results from the various tests performed with the benchmark. These include improvement from optimizations, how the problem scales on multiple compute nodes, and how domain distribution affects performance.
- Chapter 5: Summary and conclusion of the thesis project, including the findings presented and laying out what future work could be done.
- Appendix A: Code for the benchmark application and middle steps in the optimizations process.
- Appendix B: List of all median timing results extracted from the measurements during the benchmark runs.
- Appendix C: Contains the thisis' poster.
- Appendix D: Show the fall project done in preparation to this thesis.

Chapter 2

Background

This chapter provides the background and most relevant references related to the main topic of this thesis. The following sections give a background on parallel programming models, including shared memory and message passing, a summary of MPI and its core functionalities, a section on OpenMP and Pthreads, as well as sections describing the basics behind the LaPlacian operator used in the calculations, the core concept related to halo exchange as well as Amdahl's and Gustafson's law.

The OpenMP, Pthreads, and performance modeling are new for this thesis. The section describing the Laplacian operator is based on the fall project (Appendix D) but rewritten and extended to include a 3D operator. The rest is copied from the fall project and modified to read better, correct some minor flaws and include considerations for 3D problem domains.

2.1 Parallel programming models

There are several parallel programming models such as data parallelism, remote memory access, shared-memory models, and message-passing models. A parallel programming model is an abstract description of a parallel system's operations, such as communication between processes, how shared-memory is handled, and how spawning tasks or processes occurs. It abstracts away from specific hardware systems, making it theoretically possible to implement any parallel programming model on any hardware system, tho with different performance results [1].

Data parallelism describes a Single Instruction Multiple Data (SIMD) system where vectors of data are processed in parallel by applying the same computation/instruction on each element (e.g., Graphics Processing Units (GPUs)).

Remote memory access (RMA) is a model describing a shared memory message-passing hybrid model. Here memory located outside the processes memory space is made accessible through one-sided communication (e.g., using message-passing) without the overhead of point-to-point communication.

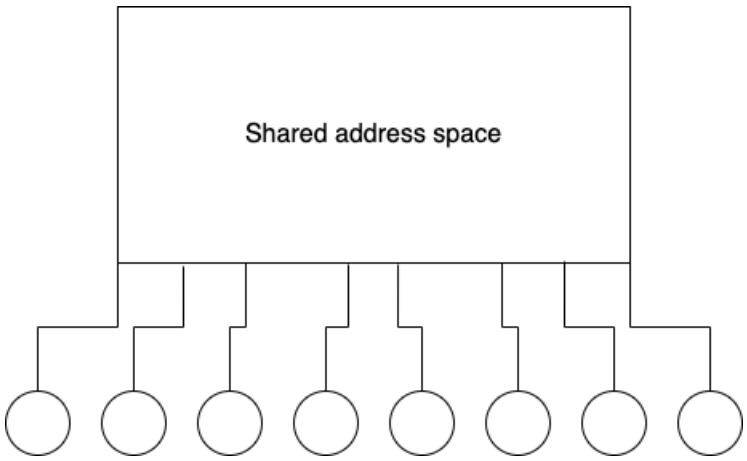


Figure 2.1: Multiple processes operating on the same address space. (Based on Figure 1.1 [2])

2.1.1 Shared memory

Control parallelism is a form of parallelism explicitly specified by the programmer as opposed to implicit parallelism due to independent data. One simple form of control parallelism is the shared memory programming model. This model uses threads of execution that run asynchronously in parallel operating on the same piece of memory. As the name entails, the entire parallel environment operates within the same shared address space, as shown in Figure 2.1. Reading and writing to a shared memory simultaneously from multiple threads can cause conflicts and race-conditions. This issue is usually handled by using a mutex or semaphore to lock all threads accessing the resource to prevent conflicts [1], [2].

Shared-memory parallel programming may use libraries such as OpenMP or Pthreads to facilitate thread life cycles. These functions in different ways. Pthreads use the POSIX threads system calls to manage the lifecycle of threads. OpenMP uses preprocessor directives within the compiler to inject structured blocks to control the OpenMP runtime library's parallelization [3].

2.1.2 Message passing

Message passing is a programming model consisting of independent processes, all with their own local memory. The communication between the processes is done using messages entailing both processes need to perform some operations to achieve this goal. Through such messages, the data stored in other processes can be shared with all system processes. Figure 2.2 shows such a system with separate address spaces for each process, all connected through a network. This network can be as simple as two computers connected via a switch to multiple computational clusters with their high speed interconnects all connected through

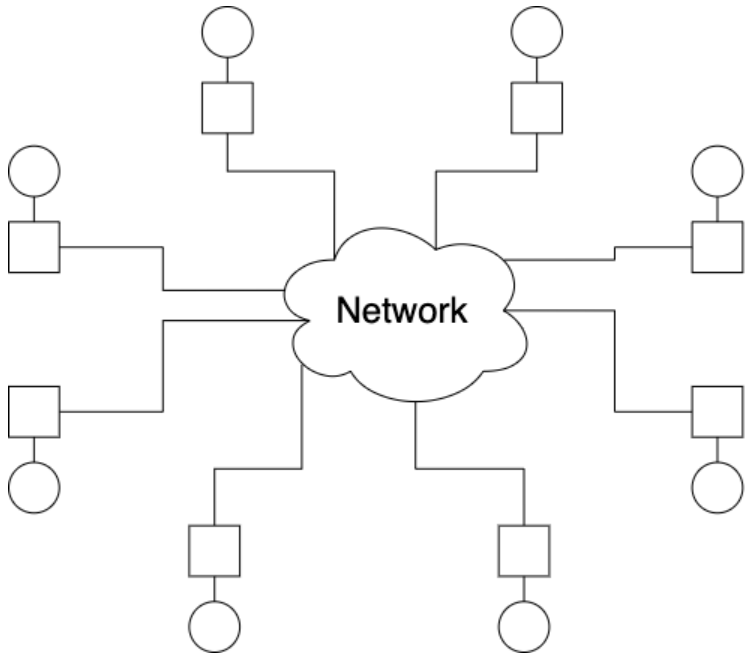


Figure 2.2: Message passing model with seperate address spaces for each process, interconnected through an arbitrary network. (Based on Figure 1.2 [2])

the internet with messages passed between the different clusters for large scale parallelism [2].

2.2 MPI

Message-Passing Interface (MPI) [2], [4] is a library specification defined to primarily address the message-passing parallel programming model, where data is moved between processes through various operations on each process. The specification defines the names, calling sequences, and results of cooperative operations, meaning one process cannot force a message upon another without the recipient actively listening. Programs utilizing an MPI implementation follow the model SPMD (single-program multiple-data) with the same code executed in all the processes. However, the data operated upon is different for each process. There are multiple MPI implementations, where each parallel computer vendor offers an implementation for their machines along with other free publicly available implementations like Open MPI and MPICH.

2.2.1 MPI concepts

Parallelization of algorithms that require communication between the different processes during the program's execution contains *syncronization points*. These are specific points in the program where some or all processes synchronize their

progression, meaning waiting for the other processes to reach the same point before continuing. An example of this is a program taking an array of numbers and scattering them to different processes where each subset of the array is summed. The resulting sum is gathered at one process to be added together, producing the total sum of the original array. The points where the process scatters the array values and gathers the resulting sums are synchronization points for this program. All processes are required to reach this MPI call before any of them can continue execution. Another example is when two processes communicate directly with each other using a send and receive call synchronizing their progress at each call.

Barriers are one such global synchronization point where all processes in a given communicator wait for every other process to reach the same point. This functionality is also seen in other collective operations.

In MPI, execution context and group of processes are both represented by the same concept, a *communicator*. This structure is often used as a parameter in point-to-point operations as the destination/source rank specified is in the communicator's context. In most MPI implementations, a *MPI_COMM_WORLD* is supplied as the communicator for all spawned processes for the program. The size and specific process' rank within a communicator is found via the MPI_Comm_size and MPI_Comm_rank calls. The resulting values from these calls in the MPI_COMM_WORLD communicator are often referred to as world size and world rank.

2.2.2 MPI fuctions

The MPI functions we use in our 3D halo benchmark program includes the following MPI functions:

- **MPI_Barrier** halts the execution until all processes in the given communicator have reached this point in the program.
- **MPI_Scatterv** distributes a non-uniform array of data to all processes in the communicator. This enables the scattering of different amounts of data to each process from a source process.
- **MPI_Gatherv** collects all data from all processes in the communicator, enabling different amounts of data from each process into one array at the source process.
- **MPI_Bcast** sends data to all other processes in the given communicator, finishing when all processes has received the data.
- **MPI_Isend** asynchronously sends data to another process.
- **MPI_Irecv** asynchronously receives data from another process.
- **MPI_Waitall** waits for all provided requests from asynchronous operations to be finished.

2.3 OpenMP

OpenMP (Open Multi-processing) API consists of various subroutines and compiler directives to parallelize different program regions. These are specified in the OpenMP API document, which defines the specification of OpenMP for C, C++, and Fortran programs. Compilers supporting OpenMP read these directives and use them as instructions on parallelizing a region of code over multiple threads. OpenMP extends the languages with various constructs to parallelize the workload and uses the fork-join parallel model of execution to parallelize over multiple threads. These include SPMD (Single Program Multiple Data), tasking, device, work sharing, and synchronization constructs with managing different access levels of memory within the parallel region such as private variables from outside the region and handling the sharing of variables between the thread [3], [5].

Pragmas are often used to specify instructions to the compiler on how to parallelize a code block over multiple threads. This can be done by using the *pragma omp parallel* over a region to parallelize it, or one can add the *for* keyword to the end to allow parallelization of for-loops. These pragmas can be used to specify how e.g. the for-loops iterations are divided among the threads or how the scheduling of the different loop chunks should function. This can be done by using the *schedule* parameter supplying a schedule mode and a custom chunk size if needed to better balance the workload. One could also use the *collapse* parameter to parallelize nested-loops also for better load balance in the case of unequal computational load for different loop iterations.

2.4 POSIX threads (Pthreads)

POSIX [6] is a standard specifying a variety of facilities for Unix-like operating systems, including a multithreaded API often referred to as Pthreads (POSIX threads). Pthreads is a library only available on POSIX systems for programming multithreaded applications. It includes control of the entire thread lifecycle, as well as features to ensure data dependent tasks execute correctly (e.g., mutexes and semaphores) [7].

2.5 Laplacian operator

The Laplacian of a problem domain highlights the regions with rapid change in intensity, making it good for detecting abrupt changes in a domain. In 2D, this can be used on image data to detect edges in the image, like the outline of a person or object. One drawback with the Laplacian operator is its sensitivity to noise. Due to this, one often chooses to smooth the data, prior to using the operator, to eliminate most noise and only leave the abrupt value changes. The operator in 3D is derived twice from the Laplacian Equation 2.1 both with respect to x, y and z

[8].

$$\hat{\Delta}^2 f(n_1, n_2, n_3) = f(n_1, n_2, n_3) * h(n_1, n_2, n_3) \tag{2.1}$$

The double derived with regards to x, y and z are then combined and calculated for the surrounding values to achieve an operator as shown in Equation 2.2.

$$\begin{aligned} &f_{xx}(n_1, n_2, n_3) + f_{yy}(n_1, n_2, n_3) + f_{zz}(n_1, n_2, n_3) \\ &= f(n_1+1, n_2, n_3) + f(n_1-1, n_2, n_3) + f(n_1, n_2+1, n_3) \\ &+ f(n_1, n_2-1, n_3) + f(n_1, n_2, n_3-1) + f(n_1, n_2, n_3+1) - 6f(n_1, n_2, n_3) = \\ &firstplane = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ &secondplane = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -6 & 1 \\ 0 & 1 & 0 \end{bmatrix} \\ &thirdplane = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{aligned} \tag{2.2}$$

2.6 Halo exchange

Halo exchanges (a.k.a, border exchanges) are used when parallelizing the algorithms where the domain is distributed across several processes/processors. Typical applications include iterative algorithms where the next iteration's calculation depends on multiple values from the previous iteration. Parallelization of stencil-based calculations require each subdomain to know what resides in the surrounding subdomains making halo exchange a viable solution to this issue.

In halo exchange, the halo is defined as the surrounding rows, columns and planes outside the current subdomain, storing data from other subdomains making this data available during the computations. The exchange of these values often happens between each iteration, or series of iterations, where all processes synchronize to share their outermost layer(s) with their neighboring subdomains.

Kjølstad and Snir [9] names this pattern of communication "Ghost cell pattern" where data is exchanged between neighboring subdomains and stored in ghost cells. The idea of storing neighbor values in ghost cells surrounding the subdomain allows improved performance instead of communicating when needed the halo data. An implementation asking for the halo values as they are needed in the computation results in huge accumulated latency from the message passing required, as well as immense complexity in the developed code to send and receive the data in between computation.

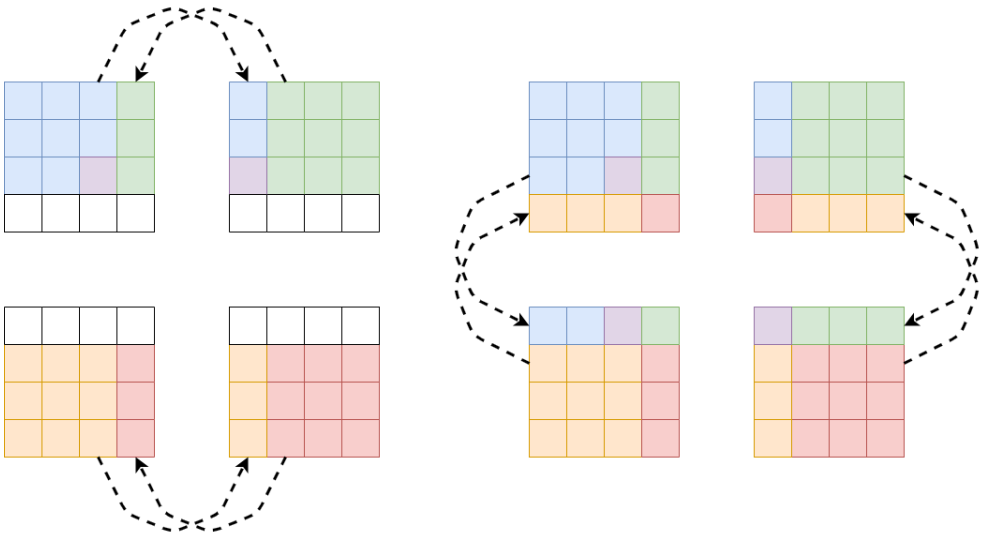


Figure 2.3: Copy diagonal values through ghost cell halo exchange. (Based on Figure 9 [9])

Another problem arises with ghost cells and the proposed communication pattern when the domain is divided into multiple dimensions (e.g., two-dimension, three-dimension). The resulting chunk to chunk communication when the computation requires diagonal data could increase the complexity. However, when synchronizing the ghost cells, they, and others, have proposed to extend the synchronized data to include the ghost cells at the ends of the rows and columns. The inclusion of the extra ghost cells will allow the passing of data between diagonal neighbors if synchronizing the processes between each exchange. The values to be copied diagonally have already been placed in the ghost cells included in another exchange direction after the synchronization is complete. As shown in Figure 2.3, this allows diagonal data transfer without any diagonal-specific communication needed.

2.6.1 Deep halo

In order to lower the cost of communications, several iterations can be computed before exchanges are needed, Holter and Elster, Kjølstad and Snir, as others "deepen" the halo size beyond the first row or column. I.e., for a halo of size 4, 4 iterations could be done before exchanges are needed. One would then have to exchange 4-deep halos, which would require larger messages, but only 1/4th the number of messages compared to exchanging between each iteration.

Extending the halo size are also beneficial in systems where the latency per message is high or the total delay of synchronization is high. As mentioned, the ex-

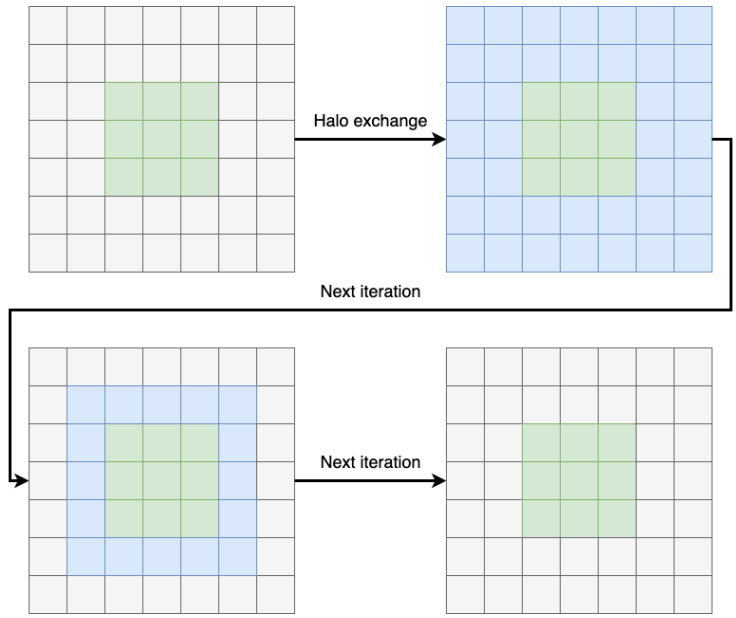


Figure 2.4: Use of deep halos to exchange computation for communication. Green cells represent chunk data, blue cells represent still valid halo data and grey cells represent invalid halo data.

tended halos provide extra layers of ghost cells the algorithm can use to compute the next iteration of ghost cell values locally and perform multiple iterations before needing to synchronize the ghost values. As shown in Figure 2.4 a 3x3 grid with two layers of halo data allocated, and after a halo exchange, the halo cells are populated and the next iteration computed. After this compute is finished, the second halo layer is still valid due to the use of a 3x3 stencil only requiring one halo layer. The second layer can be correctly calculated using the outermost halo layer, making it possible to calculate another iteration before another halo exchange is required.

2.7 Performance scaling

When talking about performance scaling, often strong and weak scaling are mentioned. Strong scaling is when the problem size is fixed, and only the processing power is scaled and is governed by Amdahl's law. On the other hand, weak scaling is governed by Gustafson's law and mentions the scaling of both the problem size and the processing power. Figure 2.5 shows the relationship between these two laws in a $P = 80$ processors scenario, with the serial fraction of the program on the x-axis and the speed-up on the y-axis. The curves are modeled using Equation 2.3 and Equation 2.4 as Amdahl's law and Gustafson's law respectively.

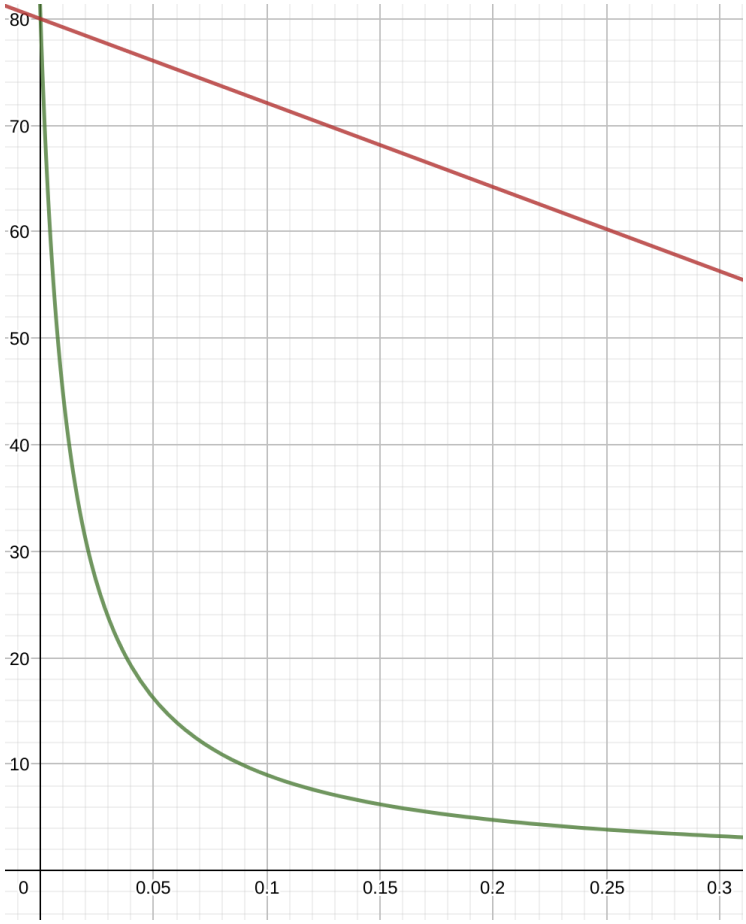


Figure 2.5: Relationship between Amdahl's law and Gustafson's law. Amdahl's law (green). Gustafson's law (red). (Inspired by Figure 1 [10]).

2.7.1 Ahmdal's law

Put forth by Gene Amdahl [11] in a session of the 1967 American Federation of Information Processing Societies meeting. He argued that the speed-up of an algorithm in parallel is dependent on its sequential parts, with an estimation that an algorithm is always about 10% sequential, and the memory accesses would impose another 25% of serial overhead. He stated that with these limitations, probably about 65% of an algorithm were parallel regions benefitting from parallelization directly. The other parts would not be affected by the increased compute resources. Equation 2.3 shows the formula used in Amdahl's chart of speed up based on computing resources and the sequential fraction of the algorithm based on fixed problem size. f is the serial fraction, and P is the number of processors for the parallel region [12].

$$\frac{1}{f + \frac{1-f}{P}} \tag{2.3}$$

2.7.2 Gustafson's law

John Gustafson [13] in the 1980s, after earlier research showing performance exceeding Amdahl's predictions, argued that the performance was a function of both problem size and the number of processors. Since a larger problem size would, in time, decrease the size of the serial fraction resulting in Gustafson's law shown in Equation 2.4. Same as for Amdahl's law (Equation 2.3) f if the serial fraction of the algorithm, and P is the number of processors used [14].

$$f + P(1-f) = P - f(P-1) \tag{2.4}$$

Chapter 3

Creating a 3D Benchmark with Halo Exhanges

This chapter outlines the different aspects of implementing a 3D stencil-based benchmarking application. It can be used to benchmark computer clusters to find the current bottleneck for 3D stencil computation and how the algorithm scales for the given cluster. The benchmark is structured as shown in Listing 3.1.

```
int main() {
  process_parameters();
  generate_global_domain();
  decompose_to_subdomains();
  compute_and_exhange();
  recompose_from_subdomains();
  print_timings();
}
```

Code listing 3.1: Benchmark program structure.

This chapter thus contains a description of the input parameters 3.1, the global domain creation 3.2, the subdomain lifecycle 3.3, how the stencil computation takes place 3.4, a description of the halo exchange 3.5, how the benchmarks timings are positioned, structured, and handled 3.6, various optimizations implemented throughout the project 3.7, how we handled the difficulty of visualizing the 3D subdomains 3.8.1 was handled, and pitfalls to avoid if implementing a similar application 3.8.

3.1 Input parameters

For a benchmarking application to be versatile and to provide ease of testing different problems and cluster specifications, we added many parameters to affect the benchmarking process. As seen in Table 3.1 the benchmark provides a level of versatility for testing on different cluster sizes, stencils, and problem sizes. Width, height, and depth represent the size of the global domain, with the stencil param-

Table 3.1: Parameters used to specify how and what the benchmark computes.

Parameter	Description
Width	Problem domain width
Height	Problem domain height
Depth	Problem domain depth
Iterations	Number of iterations to compute on the domain
Stencil	Which stencil to use in the computations
X-axis domain distribution	Number of subdomains in x-direction
Y-axis domain distribution	Number of subdomains in y-direction
Z-axis domain distribution	Number of subdomains in z-direction
Minimal output	Necessary/Comprehensive output (0/1)
Halo depth	Number of deep halo layers

eter allowing to select from the stencils available with the benchmark. As for now, this list only contains a 3x3x3 stencil but could easily be extended if one wanted to look into computation with different stencils. The domain distribution parameters specify how the global domain should be separated into subdomains. E.g., a 4x2x1 domain would mean a domain split along 4 times along the width, 2 times along the height, and keep the global depth for each subdomain. In conjunction with different domain sizes, the domain distribution parameter allows for testing if a non-cubic domain would perform better if divided into cubed subdomains or some other domain division performs better.

3.2 Global domain creation

The global domain containing all values to be computed upon is for this benchmark generated with all values set to 1 as shown in Listing 3.2. We create a three-dimensional array storing each value on an (z, y, x) coordinate within the array. The selected data structure is the z-plane pointers in the outermost array. Stored at each element in the plane are the y-column pointers, which in turn have elements that point to the x-row array storing the actual values.

```
float ***domain = malloc(domain_depth * sizeof(float **));
for (int z = 0; z < domain_depth; z++) {
  domain[z] = malloc(domain_height * sizeof(float *));
  for (int y = 0; y < domain_height; y++) {
    domain[z][y] = malloc(domain_width * sizeof(float));
    for (int x = 0; x < domain_width; x++) {
      domain[z][y][x] = 1;
    }
  }
}
```

Code listing 3.2: The global domain stored in a 3D array, created to the specifications from the input parameters.

3.3 Subdomain lifecycle

After creating the global domain, its lifecycle consists of 2 distinct steps separated by the computation on each process's subdomain. Firstly the domain is decomposed into smaller subdomains scattered one to each available node, and after the computations are finished, all subdomains are gathered into a resulting global domain. Provided the domain distribution, the global domain size is divided into subdomains where the MPI rank determines each subdomains position in the global subdomain space. Suppose the global domain size is not divisible by the number of subdomains in the respective direction. The last domain for the given direction is responsible for the remaining domain values, which may be more than the other subdomain's sizes in that direction. However, this is how the benchmark handles domain sizes not divisible by the distribution in every direction. Both the global domain and subdomains map the x-direction to the width of the domain, the y-direction to the height, and the z-direction to the depth of the domain. After calculating the size of each subdomain and displacements within the send buffer, the domain data is converted to a one-dimensional send buffer and scattered throughout the subdomains using MPI_Scatterv. Each process receives and stores the subdomain data in an array created to the calculated size of its respective subdomain.

After the computations are finished, each process sends its subdomain data in an MPI_Gatherv call to the root process. Here the individual subdomains are placed back into their correct location in the global domain array based on the size of each subdomain and the location in the global subdomain space. The reconstruction of the global domain is a reverse of the previous splitting into the one-dimensional send buffer in the lifecycle's decomposition stage.

3.4

The benchmark performs stencil computations in each iteration to calculate the next iteration's input or the result of the entire computation. The computation is done by iterating over the subdomain applying the stencil to calculate the next value using the neighboring values. These values are multiplied with the value in the stencil according to their relative position from the value currently computed. A basic approach is shown in Listing 3.3 where the input *subdomain* is used with the *stencil* to compute the next iteration's values stored in the *temp_buffer*. The benchmark supports any n^3 stencil, where n is an odd number. The resulting value is saved in a temporary buffer since overwriting the values directly in the subdomain would influence other calculations and give the wrong result. After all the computations are done, and one moves on to the next iteration or continues towards completion of the benchmark, the subdomain and temporary buffers are swapped. They are swapped to ensure the subdomain buffer reference always contains the correct values into either the next iteration or the recomposition of

the global domain.

```
int halo_size = floor(stencil_size / 2);
for (int z = 0; z < subdomain_depth; z++) {
  for (int y = 0; y < subdomain_height; y++) {
    for (int x = 0; x < subdomain_width; x++) {
      float result = 0;
      for (int i = 0; i < stencil_size * stencil_size * stencil_size; i++) {
        int dx = (i % stencil_size) - halo_size;
        int dy = ((i / stencil_size) % stencil_size) - halo_size;
        int dz = ((i / (stencil_size * stencil_size)) % stencil_size) - halo_size;
        result += subdomain[z + dz][y + dy][x + dx] * stencil[i];
      }
      temp_buffer[z][y][x] = result;
    }
  }
}
```

Code listing 3.3: Naive implementation of the subdomain value compute.

3.4.1 Compute split

The computation is split into two steps in order to hide some of the halo exchange latency as shown in Listing 3.4. While exchanging the border values, the inner parts of each subdomain are computed as these are not affected by the halo data and therefore can be computed without the halo exchange completed. After the halo exchange is completed and the inner parts of the subdomain are computed, the calculations on the edge values are performed. This is done in 6 sub-steps calculating the different faces of the three-dimensional subdomain separately. These computations use the halo values fetched from the surrounding processes through the halo exchange to calculate the values of the next iteration for the subdomain's border values. The depth into the subdomain these steps compute is dependent on both the stencil size and halo depth (see Chapter 3.5 for more information). The implemented stencil is a 3x3x3 stencil resulting in 27 values calculated.

```
for (int i = 0; i < iterations; i++) {
  start_halo_exchange_thread();
  inner_compute();
  join_halo_exchange_thread();
  outer_compute();
  swap_buffers(subdomain, temp_buffer);
}
```

Code listing 3.4: Basic layout of the iteration loop with offloaded halo exchange. (see Chapter 3.5)

3.4.2 Using OpenMP for stencil computations

Parallelization of stencil computations can be done using multiple MPI processes either locally or between nodes in a computing cluster. If running multiple MPI

processes locally, the need for inter-process halo exchange and storing of halo data within the same memory results in unnecessary amounts of used storage and complexity following the halo exchange. Although one could use shared memory within and between different nodes, this benchmark has chosen not to use an inter-node shared memory approach. However, instead of using multiple MPI processes within the same node, we use OpenMP to parallelize the compute using all available cores and reaping the benefit of fewer subdomains resulting in fewer total halo exchanges. OpenMP provides a set of pre-processor directives to parallelize a program, and in this benchmark, it was used to parallelize the three-dimensional for loop in the compute functions. The parallelization was achieved by using the *#pragma omp parallel for collapse(3)* clause on the line above the outermost loop. The clause mentioned above parallelizes the region covered by the for loops and tells OpenMP to parallelize 3 layers of for-loops to distribute the computations over all cores equally.

3.5 Implementing 3D Halo exchange

Parallelization of stencil computations requires each subdomain to know what lies beyond its border values as the stencil requires the surrounding data to calculate correctly. These bordering values reside in other subdomains making the algorithm require data beyond its borders. The benchmark uses a n^3 where n is an odd number also requiring an exchange of diagonal values meaning that one subdomain relies on up to six neighboring subdomains and the 20 subdomains neighboring diagonally. To streamline the exchange of this diagonal data to avoid unnecessary exchanges, we chose to restrict the ordering of the halo exchange directions by only performing one direction of exchanges at a time. Only after a node has sent and received all data from its two neighbors in the respective direction will it exchange in the next exchange direction, resulting in three such steps (north-south, east-west, and front-back).

First, we performed a north-south halo exchange sending data between potential subdomains above and below. Such subdomains and their respective rank were given by calculating each rank's position in the global domain distribution (x, y, z). This position was then used to check if a subdomain existed above (y - 1) or below (y + 1) and then calculating the neighbor's rank based on the current process's rank and the domain distribution. The neighbor's rank, along with the data residing in the layers facing the neighbor, was used in MPI_Isend, and the received halo data from MPI_Irecv were stored in the halo layers facing the same neighbor. After registering the receive buffer and call with MPI_Irecv and the data was sent, the process waits to complete both the send and receive requests before inserting the data into the subdomain data structure and continuing onto the next halo exchange direction. After the north-south direction is completed, the program continues onto the east-west direction followed by the front-back direction to complete the halo exchange of all six faces of the subdomain cube. When performing

these exchanges in order, including a synchronization step before continuing onto the next exchange direction, the diagonal data is also exchanged correctly. As Figure 2.3 shows, for a two-dimensional plane, correct ordering and synchronization ensure that the corner values are also transferred indirectly without the need for specific exchanges of the corner values.

3.5.1 Our Pthread approach

To optimize the computation, we chose to hide the latency of the halo exchange with computation. The latency hiding was enabled by splitting the different compute functions, with one operating on the non-halo dependent data and the other computing the values dependent on the halo data. Instead of computing the inner data and performing the halo exchange in series, we introduced a separate thread to handle the halo exchange. The main thread computes the inner subdomain data in parallel with the exchange, allowing the processor's cores to exchange the haloes and use all downtime from both the wait associated with the exchanges and the other cores not used in the exchange to maximize CPU utilization and minimize compute time. The introduction of Pthreads allowed us to launch a communication thread before calling the inner compute function and joining it with the main thread after the inner compute and halo exchange had finished.

3.6 Benchmarking approach

The benchmark performs stencil computations with halo exchange to allow scaling over multiple CPUs and nodes. Multiple aspects of the computations need to be timed to identify how well a cluster performs. The performance can be measured in raw compute power, inter-node communication, and how the implementation scales. Further, locating the bottleneck regarding a specific type of problem on the given system is another reason different extensive timing of the application is essential. These results enlighten if the communication or raw compute power is the bottleneck or if another domain distribution can increase performance by testing multiple parameters to minimize the time needed.

The different regions timed are the entire problem solution, the compute time, the individual halo exchanges, and the difference in the time the inner compute on a node uses and the exchange with its neighbors. Here the entire problem is defined as from before decomposition occurs to the final result is recomposed in the main process. Listing 3.5 shows the location of both compute and problem timings, and when it comes to the halo exchange timings, these are performed by each thread taking the time of all three functions for north-south, east-west, and front-back exchange as shown in Listing 3.6.

```
int main() {
  process_parameters();
  generate_global_domain();
start_problem_time();
  decompose_to_subdomains();
start_compute_time();
  compute_and_exhange();
end_compute_time();
  recompose_from_subdomains();
end_problem_time();
  print_timings();
}
```

Code listing 3.5: Timing structure of the main part of the benchmark.

```
void halo_exchange() {
start_timing();
  north_south_exchange();
  east_west_exchange();
  front_back_exchange();
end_timing_and_store_timing();
  join_main_thread();
}
```

Code listing 3.6: Timing structure of the individual halo exchanges.

Each iteration stores the time reference for completion of the inner compute within the compute loop, and the halo exchange thread stores the time reference for completing the exchange. These two times are used to calculate the difference in compute and exchange time ($diff = exchange\_time - inner\_compute\_time$) to allow a per-node view of where the bottleneck for each iteration is, as shown in Listing 3.7. A positive difference show that the inner compute finished before the exchange, and with a negative difference, the halo exchange finished before the inner compute.

```
for (int i = 0; i < iterations; i++) {
  start_halo_exchange_thread();
  inner_compute();
end_time_inner_comput();
  join_halo_exchange_thread();
calculate_inner_compute_vs_communication_end();
  outer_compute();
  swap_buffers(subdomain, temp_buffer);
}
```

Code listing 3.7: Timing structure of the exchange and compute difference.

Since printing is writing to I/O and would significantly slow down the program and provide incorrect results, each timing of the halo exchange and difference in communication and computation is saved to a pre-allocated array on every process. These arrays are printed systematically by allowing each process to print both their exchange times and difference calculations after all timing are done. This results in an easy-to-read and, most importantly, error-free printing of all results as all processes printing at once can result in data being overwritten in the

output stream.

3.7 Optimizations

Throughout creating the final benchmark version, different optimizations were identified to increase the stencil computation's performance. Firstly a switch to a one-dimensional underlying data structure with stride calculations was implemented to avoid multiple memory lookups to access one value. This change added an additional offset calculation to the pre-existing calculations of relative positions when performing the stencil computation. The additional stride calculation added to the total amount of operations needed to complete the calculations. The next optimization was identified as the offset calculations. These were not directly dependant on the position within the array, so they were removed from the compute loop to be pre-calculated. These optimizations are further explained in Section 3.7.1 and 3.7.2. Lastly, we identified that the implemented stencil required fewer calculations, as all stencil values, including the zero-values, were calculated. Section 3.7.3 shows a simple implementation of stencil unrolling to further optimize the computations by only calculating the needed values.

3.7.1 1D underlying data structure

The most intuitive solution when working with a three-dimensional data structure would be using a three-dimensional array as described in Chapter 3.2. Storing the data in a three-dimensional array is done by mapping each row, column, and plane to a three-dimensional coordinate within the array. However, dealing with this data structure requires multiple lookups per point accessed. The stencil computations (Listing 3.3) show this approach has some potential performance challenges, with each value accessed requiring 3 memory lookups. Firstly, one would have a memory lookup on $z + dz$ to retrieve the wanted plane. Then use this value to locate the row $y + dy$ and lastly, with this value $x + dx$ find the actual value for the coordinate in question. This requirement of multiple lookups and calculations per value accessed could be sub-optimal.

To combat this multiple lookup time that comes with using a three-dimensional array, one could change the underlying data structure to a one-dimensional array. The position of each value is then given by using Equation 3.1 to calculate the stride s relative to the start of the array. This change would reduce both memory lookups and stride calculations within the lookup to 1 of each. The main change would be to calculate the domain position before the stencil iteration and calculate an offset to the domain position based on the position in the stencil as shown in Listing 3.8.

$$\begin{aligned} s &= (z * subdomain\_width * subdomain\_height) \\ &\quad +(y * subdomain\_width) + x \\ &= (z * subdomain\_height + y) * subdomain\_width + x \end{aligned} \tag{3.1}$$

```
float result = 0;
int domain_pos = (z * subdomain_height + y) * subdomain_width + x;
for (int i = 0; i < stencil_size * stencil_size * stencil_size; i++) {
  int dx = (i % stencil_size) - halo_size;
  int dy = ((i / stencil_size) % stencil_size) - halo_size;
  int dz = ((i / (stencil_size * stencil_size)) % stencil_size) - halo_size;
  int offset = (dz * subdomain_height + dy) * subdomain_width + dx;
  result += subdomain[domain_pos + offset] * stencil[i];
}
temp_buffer[domain_pos] = result;
```

Code listing 3.8: Subdomain compute loop for the 1D underlying data structure.

3.7.2 Memory access offset calculations

The stencil computation is the part of the code responsible for each domain point's next value calculations. As shown in Listings 3.3 and 3.8 it iterates over the inner parts of the loop $subdomain\_depth * subdomain\_height * subdomain\_width * stencil\_size^3$ times causing the dx, dy and dz calculations to be performed $stencil\_size^3$ times per value. As seen in these listings, the offset values calculated depend on the stencil size and the subdomain dimensions in the one-dimensional case. These values are never changing while undergoing the compute and can be pre-computed to remove the unnecessary calculations from the loop to increase performance. The loop from Listing 3.9 was moved to outside the compute loop after the domain decomposition, storing all offset values for the process-specific subdomain. These were then used in the compute function as seen in Listing 3.10 where the relative position was given by the current position added in the offset for the given stencil value. Here the OpenMP clause has to be changed to use a *collapse(2)* due to the calculation of *domain_pos_out* prior to the iteration along the width. This move was also done to reduce the operations within the last loop, as an increment to the position is sufficient.

```
int offset[stencil_size * stencil_size * stencil_size];
for (int i = 0; i < stencil_size * stencil_size * stencil_size; i++) {
  int dx = (i % stencil_size) - halo_size;
  int dy = ((i / stencil_size) % stencil_size) - halo_size;
  int dz = ((i / (stencil_size * stencil_size)) % stencil_size) - halo_size;
  offset[i] = (dz * subdomain_height + dy) * subdomain_width + dx;
}
```

Code listing 3.9: The pre-calculation of offset values within the subdomain based on the subdomain dimensions and the stencil selected.

```
for (int z = 0; z < subdomain_depth; z++) {
  for (int y = 0; y < subdomain_height; y++) {
    int domain_pos_out = (z * subdomain_height + y) * subdomain_width;
    for (int x = 0; x < subdomain_width; x++) {
      float result = 0;
      for (int i = 0; i < stencil_size * stencil_size * stencil_size; i++) {
        result += subdomain[domain_pos_out + offset[i]] * stencil[i];
      }
      temp_buffer[domain_pos_out++] = result;
    }
  }
}
```

Code listing 3.10: The optimized subdomain compute with both 1D data structure and pre-calculated offsets.

3.7.3 Stencil unrolling

One flaw with the current compute loop is the iteration over the stencil values. If the stencil contains all non-zero values, the approach is sound, but if one opts to use a 7-point stencil or any other non n^3-point stencil, additional calculations resulting in zero values are performed. This is due to the *stencil[i] = 0* in the multiplication for some of the values resulting in wasted operations. One approach is to explicitly state all stencil values in the compute loop and only include those non-zero values to combat the zero value multiplications. However, this is not an approach with ease of stencil change, but the given stencil might perform better. Listing 3.11 shows the compute loop with stencil unrolling to further optimize the compute performance of the benchmark. Here we used the same offset calculations for the stencil to access the values but multiplied the values with the corresponding stencil value. E.g., for the used 3x3x3 stencil, instead of 27 iterations where 20 of these would result in a zero value, we slimmed this down to only the 7 values actually used in the stencil.

```
for (int z = 0; z < subdomain_depth; z++) {
  for (int y = 0; y < subdomain_height; y++) {
    int domain_pos_out = (z * subdomain_height + y) * subdomain_width;
    for (int x = 0; x < subdomain_width; x++) {
      temp_buffer[domain_pos_out++] =
        subdomain[domain_pos_out + offset[4]] +
        subdomain[domain_pos_out + offset[10]] +
        subdomain[domain_pos_out + offset[12]] +
        (-6 * in[domain_pos_out + offset[13]]) +
        subdomain[domain_pos_out + offset[14]] +
        subdomain[domain_pos_out + offset[16]] +
        subdomain[domain_pos_out + offset[22]];
    }
  }
}
```

Code listing 3.11: The optimized subdomain compute with additional stencil unrolling.

3.8 Pitfalls

During the development of the benchmark, we had some struggles with mental visualization of the 3D subdomains in relation to each other. We also had some minor issues resulting in huge result deviations. We have briefly discussed these pitfalls to ease future implementations of this benchmark or other applications based on similar principles and structures.

3.8.1 Visualization problem

Working with a three-dimensional domain, trying to visualize and see which parts of the subdomain represent the halo data and faces affected by the halo data was difficult. To solve this, as drawing the matrices on paper was no option since a piece of paper is two-dimensional, we opted to use Minecraft to create the subdomains in three dimensions. This gave the benefit of traversing between the points in the matrix to better visualize and figure out what needed to be computed separately and what was supposed to be communicated in the halo exchange.

3.8.2 Halo cell data initialization

One bug resulting in random numbers affecting the calculations was using the malloc function to allocate the subdomain memory. This only allocates the memory with no regard to the allocated memory space's content. If not manually cleaning the halo cell data, the halo exchange did not alter those data regions not facing another subdomain. This allowed the random data to persist and ruin the calculations. This can easily be combatted by using the calloc function to set the entire allocated memory region to all zero before populating it or manually cleaning the regions if other boundary conditions are to be used.

3.8.3 2D to 3D halo exchange changes

When moving from developing a 2D halo exchange to 3D, one more obvious pitfall is how to think about what should be exchanged between each subdomain. Our initial thought was strongly influenced by how one does this in two dimensions with borders surrounding the 2D grid. Using this thought, treating each subdomain's face as a 2D plane with a surrounding border, we were led to believe it very complicated. We thought that both the border compute and regions of the subdomain affected by the exchange were located on each edge of each face. This is a lot more complicatedly placed and complex than it actually should be. As shown in Figure 3.1 only the edges of each face were thought to be both affected by the halo exchange, as well as exchanged, resulting in a very complex implementation to deal with each face's edge. This interpretation would have resulted in wrong calculations as all face values are affected by the exchange, not just the face edges. The figure shows the colored cells as the ones considered halo-affected

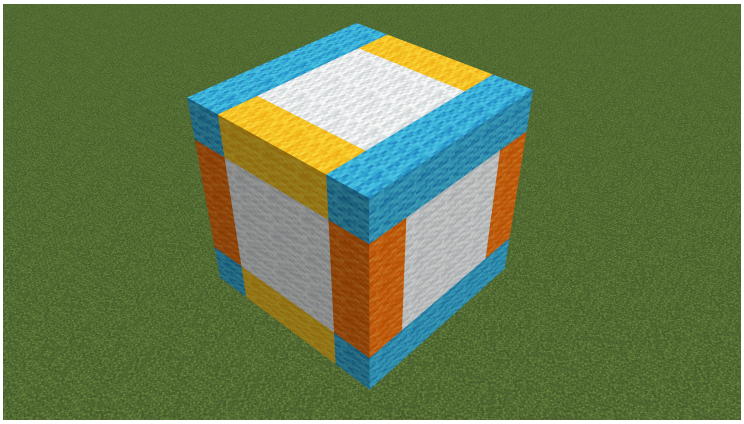


Figure 3.1: Wrongfull visualization of 3D edge compute and halo exchange. Colored regions thought as outer compute and exchange regions.

regions and the white cells as the unaffected subdomain data. This is incorrect as the entire face is affected by the halo data, not just the edges of each face.

3.8.4 Subdomain position calculations

If an approach requires knowledge of the subdomain's position in the global subdomain distribution given an MPI rank, one should be careful when calculating this. If assigning these positions using modulo and division calculation rather than a simple counter, one needs to be careful not to make a small mistake affecting the entire program. Here we had an issue with the calculations of the y-coordinate as it was divided by the wrong distribution dimension. This resulted in a segmentation fault when performing the border exchanges as the application rightfully assumed it had an exchange in the y-direction but did not invoke it as it found no subdomain above or below. It presented an array of MPI_Requests that contained no requests, but the pre-calculated request count said there were some requests issued. Due to the calculation of the request count only checking the one dimension the exchange would take place in, the actual check for the subdomain to exchange information also checked the other dimensions of the coordinate, resulting in no neighbor found.

3.8.5 Halo exchange data borders

Implementing the halo exchange logic regarding which parts of each subdomain to exchange, one must remember to transfer the entire plane, including the halo data, to the bordering subdomains. This does not include the haloes where the received data from said domain will be placed, but the halo data on the edges of planes to send. E.g., sending the data on the top of the domain, one should have included the entire x-z-plane while limiting the y-axis to skip the halo layer above and start on the y-coordinate that the actual subdomain data resides. This will

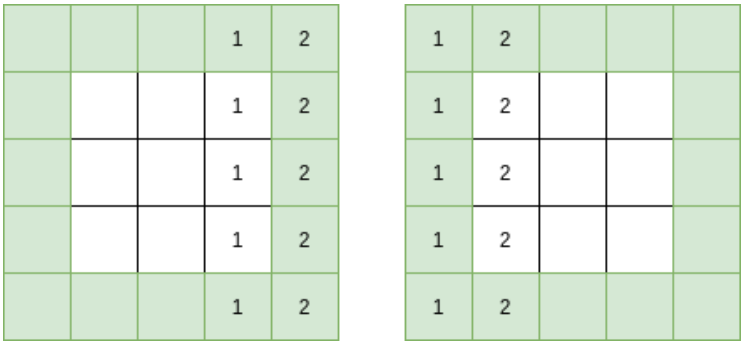


Figure 3.2: Representation of what to exchange to ensure corret diagonal border exchange. Subdomain data (white) and surrounding halo data (green).

include the halo data in x and z directions, but not the halo data outside the plane in the y-direction as this is where the subdomain residing above in 3D space's halo data will be places when received. In Figure 3.2 one can see a 2D representation of the issue and what halo data needs to be sent. The figure presents the cells labeled with *1* and *2* show where they are located in the original subdomain and where they are placed after the exchange. The vital point is the inclusion of the green halo cells above and below the subdomain data to correctly transfer the diagonal data in a 2x2 subdomain grid as shown in Figure 2.3. Extending this to three dimensions, including the halo data as shown, will ensure the diagonal data transfer also in a 2x2x2 subdomain grid.

Chapter 4

Results & Discussion

In this chapter, we present some experimental results we got from using some clustered local workstations and Idun, the central compute cluster at NTNU, where most of our tests were done.

4.1 Test Setup - Idun

The Idun cluster is a collaboration between different faculties and departments on NTNU to share their combined compute power. Each shareholder gets a proportional share of the compute time based on their contribution to the cluster. The cluster consists of 1936 cores and 92 GPUs with some different CPU and memory configurations throughout the nodes. Each node holds 2 processors who share the node's memory equally between them. Topologically each CPU and its share of memory is one NUMA node, resulting in each compute node consisting of 2 NUMA nodes. The general non-GPU parts of the cluster use 3 Mellanox passive FDR switches as interconnects between the nodes with a 56 Gb/s bandwidth. The used nodes in this thesis with their respective CPU specifications are shown in Table 4.1 [15].

Table 4.1: Idun nodes [16] used in this thesis.

Node(s)	CPU (all Intel Xeon)	C/T	Cache (MB)	Clock (GHz)	Memory (GB)
idun-02-[01-26]	2x E5-2630 v4	10/20	25	2.20	128
idun-02-27	2x Gold 6132	14/28	19.25	2.60	192
idun-03-[01-12]	2x Gold 6132	14/28	19.25	2.60	192
idun-03-[13-20]	2x Gold 6242	16/32	22	2.80	192
idun-03-[23-26]	2x Gold 6252	24/48	35.75	2.10	192

Table 4.2: Other computers used in this thesis.

Name(s)	CPU (all Intel)	C/T	Cache (MB)	Clock (GHz)	Memory (GB)
clab[02,04, 09,12,15]	Core i7-7700K	4/8	8	4.20	32
clab[06,21,23]	Core i7-8700	6/12	12	3.20	32
selbu	2x Xeon Gold 6230	20/40	19.25	2.10	383

4.1.1 UCX disabled

During the initial runs on Idun with moderate data size, e.g., 128∗128∗128, would fail and report an error from the initial *MPI_Scatterv* call in the decomposition step. This was prompted by a bug [17] in the OpenMPI version available on Idun where some data would be corrupted when using the Unified Communication X (UCX) point-to-point message layer (PML). To circumvent this issue, we disabled the UCX PML by supplying *mpirun* with the parameter *–mca pml '^ucx*' telling OpenMPI not to use a UCX PML. This could have negatively affected the optimal performance of the benchmark, but since we disabled it for all runs and cannot check the difference with it enabled and disabled can not say if this is the case. However, as mentioned, with all runs performed with UCX disabled, comparing the performance of the runs is reasonable.

4.2 Test setup - other computers

Some tests were conducted outside the Idun cluster to see how the benchmark performed on consumer-grade CPUs, and slower interconnects. We also tested on a many-core single node with 2 CPUs to see how such a scenario would affect the benchmark. In Table 4.2 we list all the computers used for these tests, with the *clab* computers using the consumer CPUs, and the dual-socket configuration sitting in *selbu*. Between these computers sits a 100Mbit ethernet switch functioning as the interconnect as these are workstations set up in a SLURM cluster and not a compute cluster like Idun.

4.3 Benchmark run conditions

When performing the runs on Idun, we could not maintain the same node selection and configuration between each run. Due to this, our results are not directly comparable, as some runs use a different configuration than others. Table 4.4 and 4.5 shows all runs performed both on Idun and the other computers. The list shows a short description of the parameters used in the run, which nodes were used and the name the runs are referred to as later in this chapter. All runs were performed by calculating 100 iterations with a compute load of a 27-point sten-

Table 4.3: Software and arguments used to compile and run the benchmark.

Compiler	GCC v10.2.0 (Idun) GCC v7.5.0 (other)
Compiler arguemnts	-O3
OpenMPI	v4.0.5 (Idun) OpenMPI v2.1.1
LMod modules	foss/2020b & CMake/3.18.4-GCCcore-10.2.0

cil, except for the *c-1d-unroll* run calculating using a 7-point stencil. We ran each benchmark 100 times on all runs when gathering the results<, with an exception for the *1d-sub, 3d-sub, 1d-big* and *1d-big-e* runs where 20 runs was performed. All runs performed on the Idun cluster used 10 cores with an equal number of OpenMP threads per node, the *selbu* computer used 20 OpenMP threads per CPU, and runs on the *clab* named computers used 4 OpenMP threads per node. All Idun cluster runs were done without exclusive access to the node, meaning the unused resources in the node could be allocated to other users. However, *1d-big-e* and all *c* and *s* prefixed runs had exclusive access to all of it's nodes. To compile and run the application, we used the software and arguments listen in Table 4.3 with Idun to load different software bundles using the LMod environment module system.

4.4 Analysis of results

All results gathered were saved in individual files for each of the 20-100 runs. These contained the total compute time for the iterations performed, timing the stencil computations and halo exchanges, and the total problem time, including global domain gather and scatter. Lastly, the files also contained the individual halo exchange times and the difference in inner compute and halo exchange time. When analyzing these results, we used the median time to avoid a single large deviating time to drastically affect the result, as it could have done using the average. With the halo exchange times and differences, the sheer number of data points was massive. We chose to reduce them to find the most representative median value for each run. The data was grouped first by run number, then each iteration's value was grouped by MPI rank. The resulting three-dimensional data structure of timings contained in the first dimension data per run. The second dimension was the MPI rank grouped, and the third dimension, each timing value. Firstly, we found the median value for each MPI rank group of values as the most representative value for the given rank. Then we combined these values across all runs and found the median value for all runs to represent the exchange and difference results for the given run.

4.5 Data structure and optimizations

During the development of this benchmark application, we implemented multiple optimizations to increase the compute performance of the application (see Section

Table 4.4: Parameters for all runs on Idun. Node name abriviations used are i2=idun-02- and i3=idun-03-.

Name	Data structure	Domain distribution	Domain size	Node(s)
1d	1D	8x1x1	1024x128x128	i2[12,27] i3[06,18,20, 24-26]
1d-big	1D	2x2x1	1024x1024x1024	i3[06,12-13,26]
1d-big-e	1D	2x2x1	1024x1024x1024	i2[03-04,27] i3[04]
1d-1x1x8	1D	1x1x8	1024x128x128	i2[08,12,16-17] i3[14-15,17-18]
1d-2x2x2	1D	2x2x2	1024x128x128	i2[08,12,16-17] i3[14-15,17-18]
1d-1s	1D	1x1x1	1024x128x128	i3[20]
1d-4s	1D	4x1x1	1024x128x128	i2[16-17,27] i3[03]
1d-12s	1D	12x1x1	1024x128x128	i2[01,03,10-11, 16-17,21,23] i3[01,03,13-14]
1d-1w	1D	1x1x1	128x128x128	i3[20]
1d-4w	1D	4x1x1	512x128x128	i2[16-17,27] i3[03]
1d-12w	1D	12x1x1	1536x128x128	i2[01,03,10-11, 16-17,21-23] i3[01,13-14]
1d-sub	1D	8x1x1	1024x128x128	i2[08,12, 16-17,27] i3[15,18,20]
3d	3D	8x1x1	1024x128x128	i2[08,12, 16-17,27] i3[15,18,20]
3d-1x1x8	3D	8x1x1	1024x128x128	i2[08,12,16-17] i3[14-15,17-18]
3d-2x2x2	3D	2x2x2	1024x128x128	i2[08,12,16-17] i3[14-15,17-18]
3d-sub	3D	8x1x1	1024x128x128	i2[08,12, 16-17,27] i3[15,17-18]

Table 4.5: Shows all runs performed on *selbu* and the *clab* machines. All runs performed on the *selbu* computer uses a *s*-prefix, the other *clab* computers are noted with a *c*-prefix.

Name	Data structure	Domain distribution	Domain size	Node(s)
c-1d	1D	2x1x1	1024x128x128	clab[02,04,06, 09,12,15, 21,23]
c-1d-big	1D	2x1x1	1024x1024x1024	clab[02,04,06, 09,12,15, 21,23]
c-1d-unroll	1D	2x1x1	1024x128x128	clab[02,04,06, 09,12,15, 21,23]
s-1d-big	1D	2x1x1	1024x1024x1024	selbu

Table 4.6: Timing results from optimized and sub-optimal runs.

Run	Compute time (seconds)	Problem time (seconds)	Compute fraction
1d	7.291	7.619	95.70%
1d-sub	81.305	82.860	98.12%
3d	10.598	11.017	96.19%
3d-sub	75.917	76.878	98.75%

3.7). The results of these optimizations can be viewed in Figure 4.1 showing the difference between the 1D and 3D variants, and the *1d-sub* and *3d-sub* showing the results for each core data structure variant with the stride calculations within the compute loop. Each column represents the total runtime to solve the given problem, with the blue part representing the compute time, including the halo exchanges. The red tops represent the preparation and finalization phases of the run, where the global domain is scattered to the nodes and gathered to recreate the resulting global domain. The specific values and results from the run can be seen in Table 4.6 with each speed-up relative to the data structures sub-optimal version (*1d-sub* and *3d-sub*).

Firstly, the major difference in performance between the sub-optimal versions and the optimized versions is noticeable. The use of pre-calculated strides sped up the 1D variant 11.15x and 7.16x on the 3D variant as shown in Table 4.7. Further, the optimized 1D data structure variant outperforms the 3D data structure with a 1.45x speed-up. To find a clue why a 1D data structure, in this case, performs better, we can compare the implementations of the two variants. Looking at Listing 3.10 and A.1 the most significant difference is the access pattern for the *in/-*

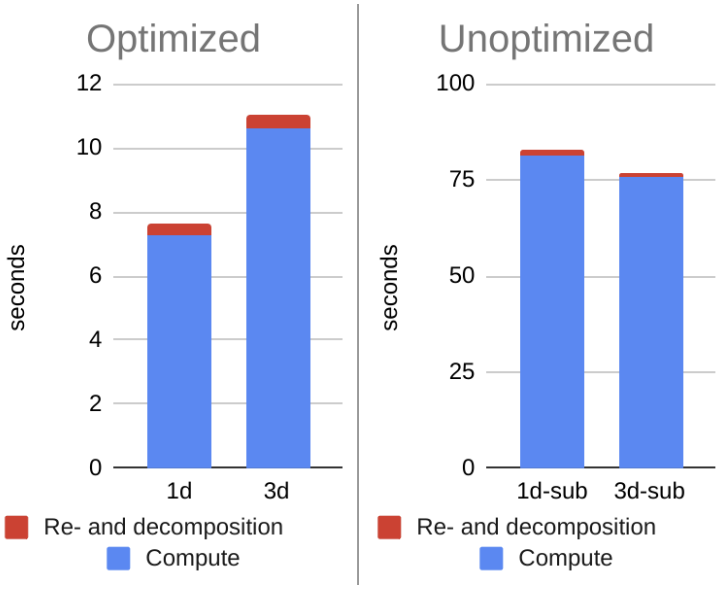


Figure 4.1: Optimizations and data structure timings. Red is preparation and finalization of domain gather and scatter. Blue is compute time with halo exchange.

Table 4.7: Speed-up from sub-optimal run with same data structure, and the sub-optimal run performance difference.

Compared runs	Speed-up
1d-sub → 1d	11.15x
3d-sub → 3d	7.16x
1d-sub → 3d-sub	1.07x

subdomain and *out/temp_buffer* arrays. Both use a pre-calculated stride array, so accessing the different stride values is equal. However, the 3D array requires 3 memory accesses for each value while the other requires only 1. This is where the significant difference in the performance is lost for the 3D array.

Taking the optimized 1D variants performance gain from its 3D counterpart, the situation for the sub-optimal variants at a glance looks abnormal. We see the 1D version performing worse than the 3D version with a 1.07x speed-up for the 3D version. However, comparing the compute loops from Listing 3.3 and 3.8 reveal a key reason for this performance difference. We see an additional calculation needed for the 1D calculations as the initial naive approach calculated the delta *x, y* and *z* values. When changing the data structure, we opted for as few as possible changes outside the data structure to see how this change would affect the performance. This resulted in the calculation of the 1D offset based on the *dx, dy* and *dz* values as shown in Equation 3.1. The additional calculation in the compute loop was performed for each stencil value, contributing to the observed performance difference when only changing the data structure.

This large speed-up between the optimized and sub-optimal versions shows the effectiveness of pre-calculating the strides. This performance gain was no surprise since most of the calculations within the compute loops before the optimizations were the stride calculations. Comparing the compute loops in Listing 3.8 and 3.10, we see all the calculations removed from the compute loop' contributing to the gain in performance. With these calculations performed once before the computations take place, we can replace them with a single memory access to an offset array. This drastically reduces the time spent calculating each point. Since the stride calculations were within the subdomain iterations and the stencil iterations over each subdomain point, the strides were calculated 27 times for the 3x3x3 stencil. A 128x128x128 subdomain would calculate the strides over 56 million times instead of the mere 27 times required for any domain size with pre-calculation.

4.6 Scaling, strong and weak

Stencil computations on a larger problem domain that might also require many iterations performed is a time-consuming feat. One can choose to use more computing power to reduce the time taken to solve the problem. If it is a memory issue where the problem is too large for a single node to handle efficiently, one could also benefit from the additional nodes to make the subdomains fit within memory and cache constraints. Due to this, we tested our benchmark's both strong and weak scaling capabilities on the shared Idun resources with both 4, 8, and 12 nodes. We used the *1d-1s* and *1d-1w* 1 node runs as baseline readings to compare the other runs against. The theoretical performance speed-up was calculated from these one node runs and is represented by the red line in both Figure 4.2 and 4.3. The blue line is the relative performance gain from the one node run. These fig-

Table 4.8: Speed-up with respect to 1-node variant.

Run	Compute speed-up	Theoretical Comparison
1d-1s	1x	1
1d-4s	3.07x	0.77
1d-8s	5.29x	0.67
1d-12s	9.47x	0.79
1d-1s	1x	
1d-4s	0.74x	0.74
1d-8s	0.64x	0.64
1d-12s	0.61x	0.61

ures show the strong and weak scaling of the benchmark with increased compute resources. The weak scaling saw an adjustment to the global domain size proportional to the compute resources as seen in Table 4.4. All results were obtained from only timing the compute loop containing the halo exchange and stencil computations. This was done to see the speed-up of the actual parallel region of the benchmark, with the constraints of the halo exchange. Applying Amdahl's law for the strong scaling and Gustafson's law for the weak scaling, we have estimated that the compute loop solely benefits from the parallelization. Therefore, we set the serial fraction of the compute to 0 and get an assumed speed-up as shown in Equation 4.1 and 4.2.

The Equation 4.1 and 4.2 shows a theoretical performance as shown in Figure 4.2 and 4.3. The latter shows a flat theoretical speed-up which at a glance is different from what Equation 4.2 shows. This is because the equation calculates the throughput of data, and the speed-ups measured in the figures are speed-up in compute time. The second line of both Equation 4.1 and 4.2 shows how the speed-up is factored in problem size N. Comparing the different strong scaling approaches, the N is equal, and the theoretical performance scales solely with P. However, the weak scaling scales both with P and N equally, resulting in an equal theoretical compute time overall, but with higher throughput.

$$\begin{aligned} \frac{1}{f+\frac{1-f}{P}} &= \frac{1}{\frac{1}{P}} = P \\ \frac{\frac{1}{f+\frac{1-f}{P}}}{N} &= \frac{P}{N} \end{aligned} \tag{4.1}$$

$$\begin{aligned} P-f(P-1) &= P \\ \frac{P-f(P-1)}{N} &= \frac{P}{N} \end{aligned} \tag{4.2}$$

Scaling the node count for the same problem size gave no surprising results. As

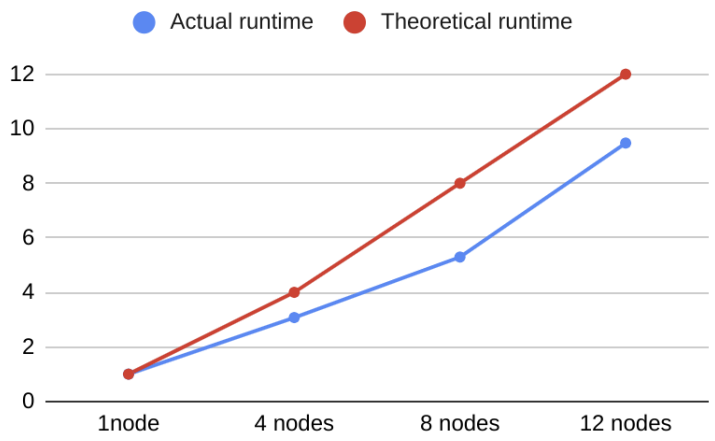


Figure 4.2: Strong scaling of benchmark implementation. Naive theoretical speed-up based on the 1 node run (red) versus relative speed-up on the different node configurations (blue).

we only timed the near parallel only region of the computation, we saw a speed-up when adding more nodes as calculated in Equation 4.1. These speed-ups are listed in the upper part of Table 4.8, with a speed-up varying from 3-9x. Looking at Figure 4.2 we see a near-perfect match of the curves, however at a lower performance rate. When comparing the theoretical speed-up to the observed speed-up, we see they miss the target by about 30%. This difference is mostly due to the halo exchange slowing down the compute as the different nodes used in these runs had different clock speeds and would finish each iteration at different times. Furthermore, since all runs were conducted without exclusive access to the nodes, this negatively affected performance due to other users using the same node and interconnect. This is addressed in more detail in Section 4.7.

Weak scaling was done with the same parameters as for the strong scaling, and the results are shown in Figure 4.3. The figure shows a declining performance relative to the problem size and node count, with performance dropping to 0.66-0.78x the theoretical performance. We see the same tendency in performance degradation, as with the strong scaling, with an approximate 25-30% degradation. Like the strong scaling case, this is affected by the halo exchange as nodes wait for each other to finish, and the bandwidth is shared with other users.

These results show that our compute loop, which was thought to be a perfect parallel region, is more dependent on the halo exchange than initially realized. Since a shared resource cluster with multiple applications running on the same nodes, the latency and bandwidth between the nodes are affected. This results in performance degradation for weak and strong scaling compared to an optimal run. It is not conclusive evidence that our benchmark scales poorly regarding strong and weak scaling. However, it requires more testing on exclusively accessed nodes to ensure no interference from other applications.

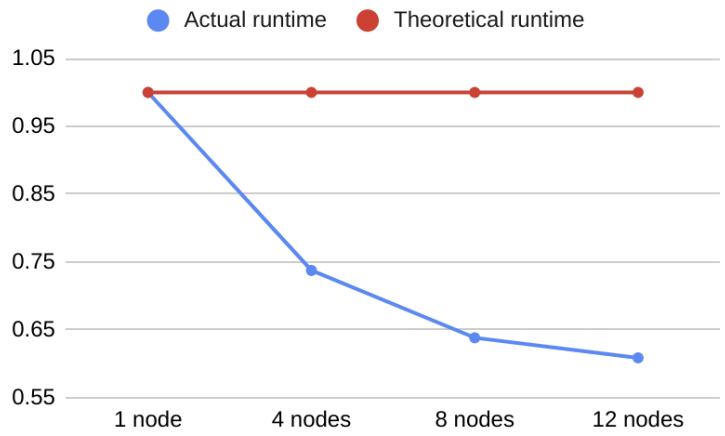


Figure 4.3: Weak scaling of benchmark implementation based 1 node run. Naive theoretical speed-up (red), actual speed-up (green).

4.7 Halo exchange

Halo exchange is a vital part of the stencil computations as it enables the separation of subdomains that depend upon each other with minimal communication overhead. The exchange is performed in one or more steps, each synchronizing with the current nodes exchanging. If splitting a global domain only in one direction, like with the *1d* run as many others, the nodes only depend on their neighbor along that one axis resulting in only one synchronization point. For another distribution like the *1d-2x2x2* run, each node requires exchange in all three directions between iterations. Figure 4.4, 4.6 and 4.7 show the time used on halo exchange in blue, and the red column is the difference between inner compute and exchange finishing. A negative difference value would suggest the exchange finished before the compute, and a positive the opposite. Since the difference between these columns is so small, Figure 4.5 shows the actual difference between the two columns to see the difference better. Here a positive value means the halo exchange time is longer than the compute time and vice versa.

Figure 4.4 show a tendency that all halo exchanges use the same amount of time as the compute threads need to wait on the exchange. Looking at Figure 4.5 we see the same trend with minimal difference in the timings except in the *1d-4s* run. Here the exchange and computation overlapped to some degree showing some communication hiding. Even though the exchange runs in a separate thread to the compute, the incomplete communication hiding is due to multiple factors. Firstly, different CPUs for different nodes prompt some nodes to wait longer than others within a run, causing the attempt at hiding the communication to fail the more they need to wait. Secondly, non-exclusive nodes could affect the performance with less cache available as the benchmark, and the other user both use the shared L3 cache. This other user might also use the interconnect in their exper-

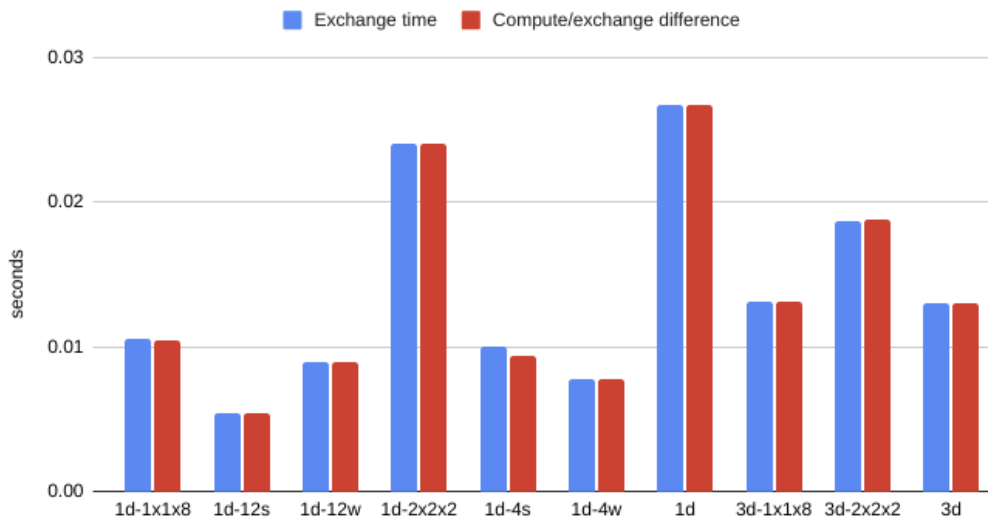


Figure 4.4: Time used on exchange versus time inner compute thread waited for exchange thread to finish.

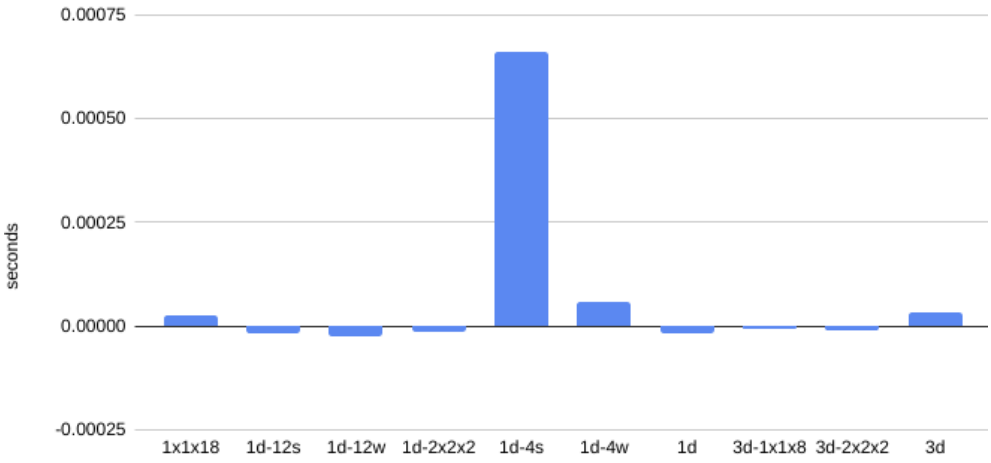


Figure 4.5: Difference in time used on exchange and time waited for exchange to finish. Positive value indicates exchange used more time then compute thread waited for it to complete.

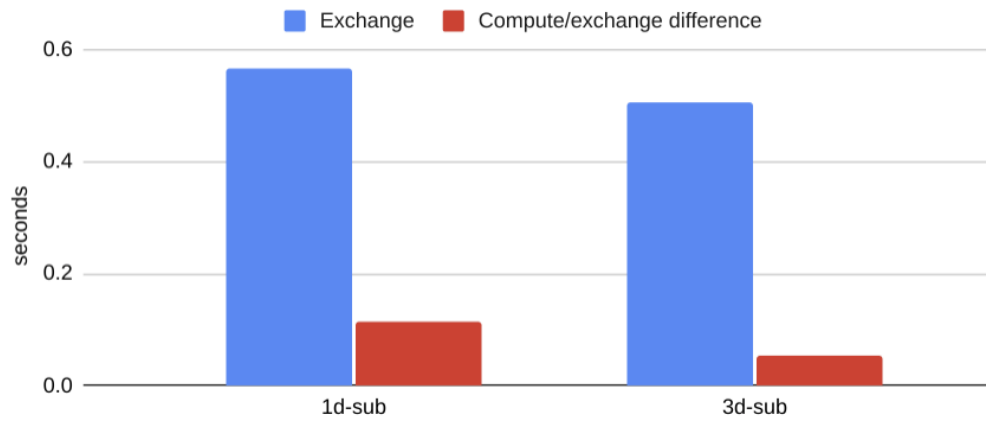


Figure 4.6: Compute time versus exchange-compute difference for the sub-optimal implementation.

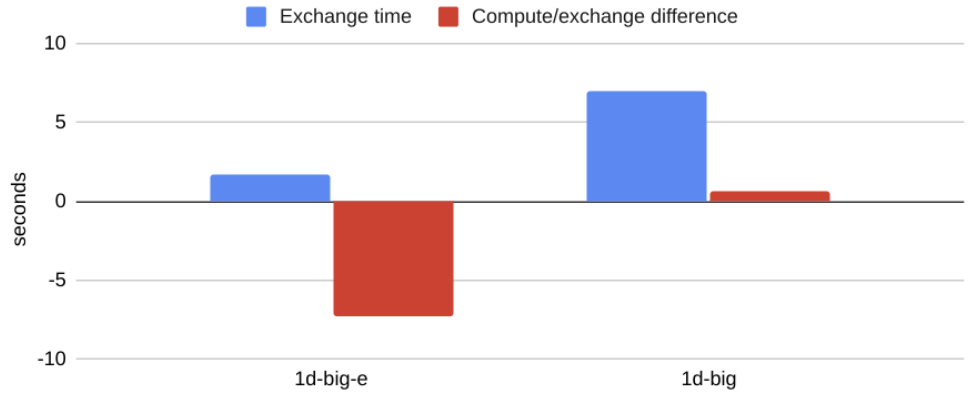


Figure 4.7: Compute time versus exchange-compute difference using a 1024x1024x1024 problem domain.

iment, making the latency between nodes increase as both users fight over access to the same resource. A third effect that for some runs could have affected the performance negatively is that the inner compute on smaller domains executes fast. Hence the halo exchange thread is not scheduled until after the compute is finished. This is possibly what happened for those runs showing a negative difference in Figure 4.5 as in these cases the exchange started after the compute finished.

How can we ensure that our implementation trying to hide the exchange with computation works? We tested both a slower compute with a less optimal implementation and an increased problem size without scaling the processing power. The sub-optimal implementation and increased problem size were used to ensure each node would use a lot more time in computation. The sub-optimal variants showed that a large amount of the exchange is performed in parallel with the

Table 4.9: Fraction of exchange hidden by compute.

Run	Hidden fraction
1d-4s	7%
1d-sub	80%
3d-sub	89%
1d-big	91%
1d-big-e	529%

Table 4.10: Halo exchange performed outside of Idun with exclusive node access.

Run	Compute time	Exchange/compute difference	Hidden fraction
c-1d	0.007	-0.057	902%
c-1d-big	0.857	-4.242	595%
c-1d-unroll	0.007	0.008	-17%

computation. A considerable slow down of the compute performance resulted in an 80-89% hide of the exchange as seen in Table 4.9. This was a massive improvement from the best of the base cases, with the *1d-4s* run successfully hiding 7% of the exchange. However, the entire exchange was not hidden, which led us to drastically increase the problem size with *1d-big* to ensure that the compute slowed more than the 11x from using the sub-optimal version. This contributed to slightly increasing the hidden fraction to 91%, but in increasing the problem size by 64x, we also increase the amount of data to be transmitted drastically.

One phenomenon mentioned but not tested is exclusive access on nodes in compute and exchange. This allows us to use all node resources without any other user occupying the other cores or using the interconnect on the node simultaneously. We did this in the *1d-big-e* run using 4 exclusive nodes to run the benchmark. We still kept the large domain from *1d-big*, so only the exclusivity and nodes were changed between the runs. As one can both see in Figure 4.4 and Table 4.9 the difference just by using exclusive nodes was massive. Instead of hiding 91% of the exchange, we managed to hide the entire exchange and could have fitted another 4 exchanges. This shows just how much exclusive node access affects the overall performance of the halo exchange on a shared cluster. We also tested on some other computers with the *c-1d*, *c-1d-big* and *c-1d-unroll* all with exclusive node access. As Table 4.10 shows, exclusive node access completely hides exchange in these cases as well, with 902% and 595% hidden even on a 100Mbit ethernet connection. However, the *c-1d-unroll* run did not hide any exchange, probably due to the speed-up of the compute itself by only calculating the 7-points in the stencil. Removing 20 of the 27 points of the other runs resulted in the exchange being scheduled after the compute was finished.

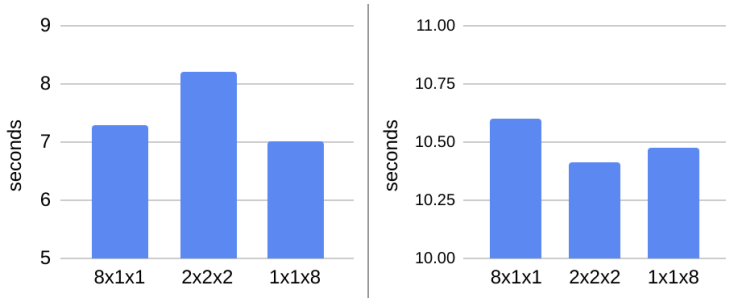


Figure 4.8: Runtime on equal problem domaim with different domain distribution. Left is 1D. Right is 3D.

Table 4.11: Domain distribution speed-up.

Run	Speed-up
1d-8x1x1 → 1d-2x2x2	0.89x
1d-8x1x1 → 1d-1x1x8	1.04x
3d-8x1x1 → 3d-2x2x2	1.02x
3d-8x1x1 → 3d-1x1x8	1.01x

4.8 Domain distribution differences

When one parallelizes an algorithm working on a global domain, the parallelization sees the domain split into multiple subdomains for which one process or node is responsible. The way one chooses to split the global domain into subdomains varies from the number of nodes available and the shape of the global domain. Here we tested 3 different domain distributions for both the 3D and 1D underlying data structure. As Figure 4.8 shows, the major difference previously observed where the optimized 1D version outperforms the 3D version still holds. The left 3 columns are from the 1D version and the right 3 columns are the 3D version. Table 4.11 shows the speed-up from the 8x1x1 domain distribution. This was chosen to split the global domain into equal 128x128x128 subdomains, with its counterparts splitting it into a 512x64x64 subdomain and 1024x128x16 subdomain.

The results from Figure 4.8 show that for the 1D version, a 1x1x8 domain distribution with subdomain sizes of 1024x128x16 performed best. On the other hand, for the 3D version, subdomain sizes of 512x64x64 in a 2x2x2 domain distribution performed best. These results are heavily affected by the difference in nodes and the non-exclusivity when tested on Idun, leading them to show a wrong picture of the reality in ideal conditions. The distrust of the results beyond the node difference and the difference in communication load from other users is the math behind the data transmitted. In Equation 4.3, 4.4 and 4.5 we sum the total amount of values sent per halo exchange for the different distribution. H_{wxhxd} is the halo data for the given distribution, the global domain size given by w, h and d, and

s is the padding of halo data surrounding the subdomain. The calculations were done by using the same global domain as used in the runs 1024x128x128. The 8x1x1 variant transmits less data per exchange with about 17k values, with the 2x2x2 sends over 4 times the data and 1x1x8 even more. Furthermore, the 2x2x2 distribution will send the data in three different exchanges in all three directions with a synchronization point between each exchange direction. This should furthermore add to the total exchange time when compared to ideal conditions. Our tests show a different picture, leading us to conclude that our results for this comparison are too affected by unexpected circumstances outside of our control and analysis capability. Because of this, new tests have to be performed to find a more conclusive result on what domain distribution performs best.

$$\begin{aligned} H_{8x1x1} &= (h+2s)*(d+2s) \\ &= 130*130 = 16900 \end{aligned} \tag{4.3}$$

$$\begin{aligned} H_{2x2x2} &= (\frac{h}{2}+2s)*(\frac{d}{2}+2s)+(\frac{w}{2}+2s)*(\frac{h}{2}+2s)+(\frac{w}{2}+2s)*(\frac{d}{2}+2s) \\ &= 66*66+514*66+514*66 = 72204 \end{aligned} \tag{4.4}$$

$$\begin{aligned} H_{1x1x8} &= (w+2s)*(h+2s) \\ &= 1026*130 = 133380 \end{aligned} \tag{4.5}$$

4.9 Stencil unrolling

As mentioned in Section 3.7.3 the compute loop calculates many values that are unnecessary as the stencil values are mostly 0s in the used stencil for this benchmark. However, with the implementation naively iterating through all n^3 stencil values, the potential speed-up is huge. We used the aforementioned 7-point Laplacian stencil, expressed in 27 values, where only seven are non-zero. Because of this, we chose to test a simple case with a hardcoded 7-point stencil to remove all unnecessary calculations. Figure 4.9 shows the compute time for a regular 27-point calculation and a 7-point calculation on 8 nodes, and Table 4.12 shows the speed-up with the unrolled stencil.

As shown in Figure 4.9 there is a considerable gain in performance going from a 27-point naively used stencil to a 7-point stencil. We observed a speed-up of 1.40x from the regular calculations, which are unsurprising as the sheer amount of calculations necessary are drastically reduced. As we only tested the one configuration and stencil, we can only guess how this would affect larger stencils, like a 5x5x5 or 9x9x9 stencil not occupying the entire stencil grid. E.g., a 9x9x9 25-point stencil would with stencil unrolling require 25 values calculated. However, if expressed in the 9x9x9 grid naively used in the calculations, it would require 729 value calculations resulting in a massive performance degredation.

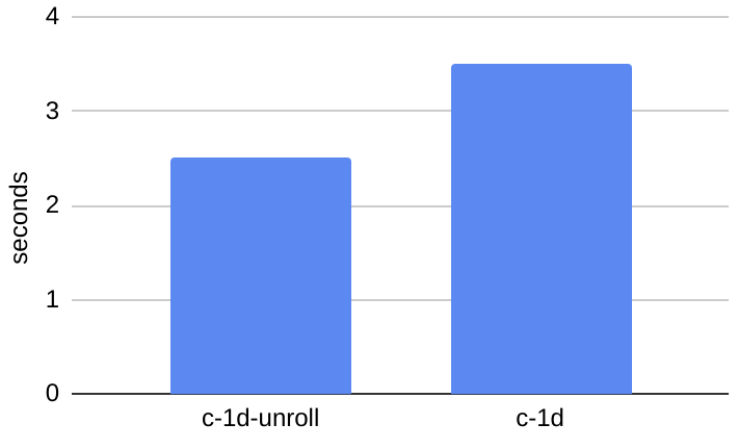


Figure 4.9: Unrolled 7-point stencil computation time versus generic stencil computation time.

Table 4.12: Speed-up with stencil unrolling.

Run	Speed-up
c-1d → c-1d-unroll	1.40x

4.10 Profiling

To find what parts of the benchmark does the heaviest lifting, we used a profiling tool called *perf* to profile our application as it ran on multiple nodes. We used the *1d-big* run configuration to profile, not interfering with its results, but as a separate run. As seen in Figure 4.10 the inner compute function by far did the most work running 96% of the entire run time. However, looking at another node's profiling output, we see a completely different balance between the compute and halo exchange functions. It can be seen in Figure 4.11 with still the inner compute as the most time-consuming function. However, the pthread_spin_lock used by MPI for busy waiting on exchange takes a considerable amount of the total run time. This is not surprising as a fast node would complete its computations before its slower counterparts and then need to wait between each iteration for the others to finish.

Outside of showing the computations to be the most time-consuming portions of the application, we see that the potential loss in waiting for slower nodes is considerable with an unbalanced run. This could be used to compute on a larger subdomain or other tasks if using a shared cluster. The latter would probably introduce overhead with context switches, but the point stands, these faster nodes waists computing resources in such scenarios. One could choose to implement CPU-based task partitioning to balance the compute load relative to each node's performance to reduce the need for waiting. This is mentioned in more detail in Section 5.1.2.

Overhead	Command	Shared Object	Symbol
96.11%	main	main	[.] inner_compute._omp_fn.0
0.70%	main	main	[.] outer_compute._omp_fn.2
0.67%	main	main	[.] outer_compute._omp_fn.3
0.43%	main	main	[.] east_west_exchange
0.32%	main	[unknown]	[k] 0xffffffffaee00b87
0.27%	main	libc-2.28.so	[.] __memmove_avx_unaligned_erms
0.22%	main	main	[.] domain_recompose
0.20%	main	main	[.] domain_decompose
0.19%	main	main	[.] outer_compute._omp_fn.4
0.19%	main	main	[.] outer_compute._omp_fn.5
0.19%	main	main	[.] min
0.09%	main	main	[.] outer_compute._omp_fn.0
0.09%	main	main	[.] outer_compute._omp_fn.1
0.02%	main	libfabric.so.1.14.0	[.] rxm_ep_do_progress
0.02%	main	libfabric.so.1.14.0	[.] vrb_cq_read
0.02%	main	libfabric.so.1.14.0	[.] ofi_cq_readfrom
0.02%	main	[vdso]	[.] __vdso_clock_gettime
0.02%	main	libpthread-2.28.so	[.] pthread_spin_lock
0.02%	main	main	[.] main

Figure 4.10: Profiling of *1d-big* run on slow node.

Overhead	Command	Shared Object	Symbol
59.17%	main	main	[.] inner_compute._omp_fn.0
15.36%	main	libpthread-2.28.so	[.] pthread_spin_lock
3.51%	main	[vdso]	[.] 0x0000000000009c7
2.18%	main	[vdso]	[.] __vdso_clock_gettime
1.51%	main	libfabric.so.1.14.0	[.] ofi_cq_readfrom
1.28%	main	libfabric.so.1.14.0	[.] vrb_cq_read
1.28%	main	libfabric.so.1.14.0	[.] rxm_ep_do_progress
1.18%	main	libfabric.so.1.14.0	[.] vrb_poll_cq
1.11%	main	mca_mtl_ofi.so	[.] ompi_mtl_ofi_progress_no_inline
1.02%	main	libopen-pal.so.40.20.5	[.] opal_progress
0.81%	main	libfabric.so.1.14.0	[.] ofi_cq_progress
0.69%	main	libpthread-2.28.so	[.] pthread_spin_unlock
0.60%	main	main	[.] outer_compute._omp_fn.3
0.59%	main	main	[.] outer_compute._omp_fn.2
0.58%	main	libfabric.so.1.14.0	[.] rxm_ep_progress
0.58%	main	libc-2.28.so	[.] __clock_gettime
0.58%	main	libfabric.so.1.14.0	[.] pthread_spin_lock@plt
0.58%	main	libfabric.so.1.14.0	[.] pthread_spin_unlock@plt
0.52%	main	libopen-pal.so.40.20.5	[.] opal_timer_linux_get_cycles_sys_timer

Figure 4.11: Profiling of *1d-big* run on fast node.

After profiling our benchmark, we see that the compute load on different nodes with an equal distribution causes some nodes to wait excessively, which could be used more wisely to reduce total compute time with unequal subdomains. However, the most important thing to note is that unsurprisingly the compute functions are the most time-consuming, and where optimizations would benefit the benchmark the most. One could use AVX or SSE instructions to calculate multiple values at once to speed up the computations further.

Chapter 5

Conclusions and Future Work

Dwindled single-core performance gain due to frequency scaling prompt the need for parallelization techniques that take advantage of multi-core and multi-node system performance. Finite difference computation using stencils are known to be compute-heavy, and for larger iterative problems, require multiple runs. In addition, larger problems that do not fit in one processor´s memory, can still be accommodated for in-core (in RAM) computations by utilizing memories across multiple nodes.

This thesis focused on implementing and analyzing a 3D stencil-based benchmarking application using halo exchanges between nodes on a shared resource cluster. Analysis of the change of data structure, pre-calculation of memory strides, stencil unrolling optimizations, and how these affect compute time were included. Lastly, we benchmarked and analyzed how running stencil computations with inter-node communication on a shared resource cluster affected the exchange performance and what considerations are necessary for such run conditions.

Implementing 3D computations with halo exchange showed to be challenging regarding which subdomain parts were to be transmitted in a given exchange, and how the computations should be efficiently and correctly split to allow exchange hiding. Firstly, the implemented exchange had many road bumps with visualization of the domains to better see what part to transmit. This was done using Minecraft to build the 3D domain, which turned out to be a nice way to visualize the issues with our initial approach. We fixed these issues to enable a correct exchange implementation. This also helped with splitting the compute loops into inner and outer compute to facilitate the hiding of the exchange.

Our implementation was not optimized during the initial development, and as the profiling of the runs showed, we used almost all the time of the run in the computational loops. This is why optimizing these loops was the highest priority, resulting the following 3 attempts at optimization: Firstly, we changed the underlying data structure to a 1D array to remove many memory lookups to find one

value. This performed worse than the 3D data structure due to the added calculation to find the 1D stride from the 3D delta values. Secondly, we looked into pre-calculation of the strides to reduce the compute time, which it did drastically. With this change, the 70-80 seconds compute time was dropped to 7-12 seconds with a performance improvement of 7-11x for our test cases. This pre-calculation also presented the advantage of a 1D data structure with a 1.07x speed-up from the 3D data structure. Lastly, we tried stencil unrolling to reduce the number of useless stencil calculations as the used 7-point stencil only required seven calculations instead of the 27 calculations done with the generic approach first used. Stencil unrolling saw a further speed-up of 1.40x, concluding that the 1D data structure with pre-calculated strides and stencil unrolling had the fastest compute time.

Benchmarking on shared computing resources had its challenges as other users' workload affect performance. When all initial runs were performed on our central Idun cluster [15], we saw none of them hiding the exchange more than 7% at best. We tried to extend the compute time, as the computation was too fast for the communication to finish. Both the sub-optimal version and a huge problem domain did not hide the exchange completely, managing at best to hide 91% of the exchange. We then tried using exclusive node access and got a massive improvement. We saw that more than 5 exchanges could be hidden for each computational triple-loop (529% hidden) when using a large problem domain (1024x1024x1024). We also did some tests on a smaller dedicated cluster. This resulted in a similar performance, hiding 902% of the exchange for our most common domain size (1024x128x128), and 595% for the large domain size. This led us to conclude that exclusive access is essential when using a shared cluster with halo exchanges as other users' work will otherwise impact communication time and overall performance.

We also scaled the node count and problem sizes, and tried different distributions of the problem domain over the nodes. First, we tested the strong scaling and compared it with a theoretical performance according to Amdahl's law. Here we saw performance increase when scaling, but the estimated theoretical speed-up of the compute loop was wrong. We observed that it had a 25-35% serial fraction due to the serial nature of the halo exchange. We did not test how it scaled with hidden haloes, something that should be looked at further. Furthermore, the distribution of the problem domain did not give us any conclusive results since the difference in node performances of our heterogenous cluster, and non-exclusive node access, gave inconclusive results. These results compared with the amount of transmitted data shows that in some cases, the one with the most data sent outperformed the one with the least data. With the node exclusivity impacting the halo exchange, we concluded that the effect of the distribution needs to be tested further with exclusive node access and equal nodes to ensure more reliable results.

5.1 Future work

Theis work can be extended in many directions. Following are some of our ideas for future work.

5.1.1 Deep halos

Using a larger halo depth to compute multiple iterations between each halo exchange is not tested in this benchmark. However, extending the benchmark to include this is interesting when dealing with a global domain distributed over many compute nodes with high inter-node latency or low bandwidth. If either the subdomains are small due to a high node count or a small node count with large subdomains and low bandwidth, the halo exchange is more time-consuming than the compute itself. In these cases, to trade some compute time with halo exchanges, one can compute multiple iterations between each exchange instead of exchanging between each iteration. In such a case, one can use deep halos. When computing these inter-exchange iterations, one skips the bad data layers to compute those subdomain regions necessary to ensure a correct calculation. This should be investigated as this could result in less computation on each iteration before resetting the amount after a halo exchange to further increase the performance gain with deep halos.

5.1.2 Hardware related task partitioning

Using a large shared compute cluster, the different nodes have different compute and interconnect loads based on other users using the same nodes. The different nodes one gets access to might also have different specifications like different CPUs or memory, which creates an unbalance between the nodes with some nodes require waiting for the others to finish the compute before exchanging. One possible solution is task partitioning based on the difference in CPU and memory between nodes. If this information is used to partition the subdomains unevenly to match the time a node is finished with the compute with all nodes in the run. This could increase the overall performance of the application as well as the utilization of each node. All runs conducted in this thesis used the smallest number of cores available as a bottleneck for all CPUs. Some CPUs matched the core count requested, but in some cases, other nodes only used 10 out of 24 available cores. Such a partitioning would require an asymmetric global domain distribution, with subdomain sizes scaled to each node's core count, available memory, and CPU clock speed.

5.1.3 AVX instructions in stencil computations

The compute loop is the most time-consuming, as seen in Figure 4.10, and if applying a deep halo technique, this would increase the time spent on the stencil computations. To increase the efficiency of these stencil calculations, one could

employ AVX or SSE instructions to reduce the required compute drastically by computing multiple values within the same clock cycle. This would require an extensive restructuring of the compute loop to load into the AVX registres and compute on these as efficiently as possible. One might also need to restructure the underlying data structure based on how values are located in relation to each other. However, an AVX/SSE approach is worth checking out to increase performance drastically.

5.1.4 Stencil unrolling

The used stencil is expressed in an n^3 array and the generic implementation of the computation will always loop through every stencil value. For stencils containing zero-values, this is very sub-optimal, with many calculations taking place before multiplying with a 0. One could restructure either the compute loop to directly use the stencils value hardcoded into the loop, which we did briefly try in Section 3.7.3. This was only tested for a single run configuration as a spur-of-the-moment idea. It should be looked further into how this affects performance under different conditions and how one can support different stencils without rewriting the compute loop.

One approach for a generic stencil implementation would be to iterate through the stencil and saving the non-zero values to a separate stencil array. Then use this and the different value's position to pre-calculate the strides for each value. In the compute loop, instead of iterating over every value in the stencil, one iterate over the reduced stencil array and the stride array calculated from the reduce variant to only calculate the non-zero stencil values.

Bibliography

[1] C. Kessler and J. Keller, "Models for parallel computing: Review and perspectives," *Mitteilungen-Gesellschaft für Informatik eV, Parallel-Algorithmen und Rechnerstrukturen*, vol. 24, pp. 13–29, 2007.

[2] G. William, L. Ewing, and S. Anthony, *Using MPI : Portable Parallel Programming with the Message-Passing Interface.* Ser. Scientific and Engineering Computation. The MIT Press, 2014, vol. Third edition, ISBN: 978-0-262-52739-2.

[3] O. A. R. Board, *OpenMP Application Programming Interface*, English, Nov. 2020. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf (visited on 12/17/2020).

[4] "MPI: A Message-Passing Interface Standard," en, Tech. Rep., p. 868. [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf (visited on 09/11/2020).

[5] R. Chandra, Ed., *Parallel programming in OpenMP*. San Francisco, CA: Morgan Kaufmann Publishers, 2001, ISBN: 978-1-55860-671-5.

[6] "IEEE Standard for Information Technology—Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7," *IEEE Std 1003.1, 2016 Edition (incorporates IEEE Std 1003.1-2008, IEEE Std 1003.1-2008/Cor 1-2013, and IEEE Std 1003.1-2008/Cor 2-2016)*, pp. 1–3957, Sep. 2016, Conference Name: IEEE Std 1003.1, 2016 Edition (incorporates IEEE Std 1003.1-2008, IEEE Std 1003.1-2008/Cor 1-2013, and IEEE Std 1003.1-2008/Cor 2-2016). DOI: 10.1109/IEEESTD.2016.7582338.

[7] P. S. Pacheco, *An introduction to parallel programming*. Amsterdam : Boston: Morgan Kaufmann, 2011, ISBN: 978-0-12-374260-5.

[8] P. A. Mlsna and J. J. Rodríguez, "Gradient and Laplacian Edge Detection," in *Handbook of Image and Video Processing (Second Edition)*, ser. Communications, Networking and Multimedia, A. BOVIK, Ed., Second Edition, Burlington: Academic Press, 2005, pp. 535–553, ISBN: 978-0-12-119792-6. DOI: 10.1016/B978-012119792-6/50095-4.

[9] F. B. Kjolstad and M. Snir, "Ghost Cell Pattern," en, in *Proceedings of the 2010 Workshop on Parallel Programming Patterns - ParaPLoP '10*, Carefree, Arizona: ACM Press, 2010, pp. 1–9, ISBN: 978-1-4503-0127-5. DOI: 10.1145/1953611.1953615.

[10] J. L. Gustafson, G. R. Montry, and R. E. Benner, "Development of Parallel Methods for a 1024-Processor Hypercube," en, *SIAM Journal on Scientific and Statistical Computing*, vol. 9, no. 4, pp. 609–638, Jul. 1988, ISSN: 0196-5204, 2168-3417. DOI: 10.1137/0909041.

[11] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," en, in *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*, Atlantic City, New Jersey: ACM Press, 1967, p. 483. DOI: 10.1145/1465482.1465560.

[12] R. K. Karmani, G. Agha, M. S. Squillante, J. Seiferas, M. Brezina, J. Hu, R. Tuminaro, P. Sanders, J. L. Träffe, R. A. Geijn, J. L. Träff, R. A. Geijn, M. B. Sander, J. L. Gustafson, R. O. Dror, C. Young, D. E. Shaw, C. Lin, J.-K. Lee, R.-G. Chang, C.-B. Kuan, G. Kollias, A. Y. Grama, Z. Li, R. C. Whaley, and R. W. Vuduc, "Amdahl's Law," en, in *Encyclopedia of Parallel Computing*, D. Padua, Ed., Boston, MA: Springer US, 2011, pp. 53–60, ISBN: 978-0-387-09765-7 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_77.

[13] J. L. Gustafson, "Reevaluating Amdahl's law," en, *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, May 1988, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/42411.42415.

[14] A. Kalyanaraman, K. Hammond, J. Nieplocha, M. Krishnan, B. Palmer, V. Tipparaju, R. Harrison, D. Chavarrıa-Miranda, J. Makino, D. Bader, G. Cong, B. Hendrickson, J. Shalf, D. Donofrio, C. Rowen, L. Oliker, M. Wehner, and J. L. Gustafson, "Gustafson–Barsis Law," en, in *Encyclopedia of Parallel Computing*, D. Padua, Ed., Boston, MA: Springer US, 2011, pp. 825–825, ISBN: 978-0-387-09765-7 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_2187. (visited on 05/30/2021).

[15] M. Själander, M. Jahre, G. Tufte, and N. Reissmann, "EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure," *arXiv:1912.05848 [cs]*, Dec. 2020, arXiv: 1912.05848. [Online]. Available: http://arxiv.org/abs/1912.05848 (visited on 05/06/2021).

[16] *Idun Hardware – High Performance Computing Group*, en-US. [Online]. Available: https://www.hpc.ntnu.no/idun/hardware (visited on 05/06/2021).

[17] *Data corruption with OpenMPI 4.0.4 and UCX 1.9.0*, en, Issue board, Apr. 2021. [Online]. Available: https://github.com/open-mpi/ompi/issues/8442 (visited on 05/25/2021).

Appendix A

Benchmark code and optimizations

A.1 Optimizations

```
void inner_compute(DATA_TYPE ***in, DATA_TYPE ***out, int width, int height,
  int depth, DATA_TYPE *kernel, int kernel_size, int halo_size, int halo_depth,
  point_offset_t *point_offset)
{
  int k3 = kernel_size * kernel_size * kernel_size;
  int upper_z = depth - (halo_size * (halo_depth + 1));
  int upper_y = height - (halo_size * (halo_depth + 1));
  int upper_x = width - (halo_size * (halo_depth + 1));
  int lower = halo_size * (halo_depth + 1);
#pragma omp parallel for collapse(3)
  for (int z = lower; z < upper_z; z++)
  {
    for (int y = lower; y < upper_y; y++)
    {
      for (int x = lower; x < upper_x; x++)
      {
        DATA_TYPE value = 0;
        for (int i = 0; i < k3; i++)
        {
          point_offset_t offset = point_offset[i];
          value += in[z + offset.dz][y + offset.dy][x + offset.dx] * kernel[i];
        }
        out[z][y][x] = value;
      }
    }
  }
}
```

Code listing A.1: 3D inner compute variant with pre-calculated domain stride values.

A.2 Benchmark implementation

```
#include "compute.h"
#include "constants.h"
#include "domain_lifecycle.h"
#include "exchange.h"
#if __has_include(<openmpi/mpi.h>)
#include <openmpi/mpi.h>
#else
#include <mpi.h>
#endif
#include <omp.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

DATA_TYPE laplace_7_kernel[] = {
    0, 0, 0,
    0, 1, 0,
    0, 0, 0,
    0, 1, 0,
    1, -6, 1,
    0, 1, 0,
    0, 0, 0,
    0, 1, 0,
    0, 0, 0
};

DATA_TYPE *kernels[] = {laplace_7_kernel};
int kernel_sizes[] = {3};
char *kernel_names[] = {"7-point␣laplce␣kernel"};
int kernel_count = 1;

void swap_buffers(DATA_TYPE **a, DATA_TYPE **b)
{
  DATA_TYPE *tmp = *a;
  *a = *b;
  *b = tmp;
}

void print_data(DATA_TYPE *data, int width, int height, int depth)
{
  for (int z = 0; z < depth; z++) {
    printf("---------------------------------\n");
    for (int y = 0; y < height; y++) {
      int domain_pos = POSZY(z, y);
      for (int x = 0; x < width; x++) {
        printf("%.2f\t", data[domain_pos++]);
      }
      printf("\n");
    }
  }
  printf("---------------------------------\n");
}

void error(char *message, int world_rank, void (*cleanup)(void *data), void *data)
{
```

```
  if (world_rank == 0)
  {
    printf("%s", message);
  }
  if (cleanup != NULL && data != NULL)
  {
    (*cleanup)(data);
  }
  MPI_Finalize();
  exit(0);
}

int main(int argc, char **argv)
{
  // Number of MPI nodes
  int world_size;

  // ID of the current MPI node
  int world_rank;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  if (argc < 11) {
    error("Usage␣main␣<width>␣<height>␣<depth>␣<iterations>␣<kernel-selection>
␣␣␣␣␣␣<domain-distribution-x>␣<domain-distribution-y>␣<domain-distribution-z>
␣␣␣␣␣␣<minimal-output>␣<halo-depth>\n", world_rank, NULL, NULL);
  }

  // Global domain with, height, depth
  int width = atoi(argv[1]);
  int height = atoi(argv[2]);
  int depth = atoi(argv[3]);

  // Number of compute iterations
  int iterations = atoi(argv[4]);

  // Selected kernel for compute
  int kernel = atoi(argv[5]);

  // Number of subdomains in all three dimension (X,Y,Z)
  int domain_distribution[3] = {atoi(argv[6]), atoi(argv[7]), atoi(argv[8])};

  // True/false - mute most output from processing
  int minimal_output = atoi(argv[9]);

  // Halo size is the size of each layer of halo required for computing the kernel
  // (kernel half size/nth order stencil)
  int halo_size = kernel_sizes[kernel] / 2;

  // Halo depth is number of deep halos to use in compute
  int halo_depth = atoi(argv[10]);

  if (minimal_output == 0 && world_rank == 0)
  {
    int num_threads_available;
#pragma omp parallel
    {
      num_threads_available = omp_get_num_threads();
    }
```

```
    printf("|---------------------------------\n");
    printf("| Run configuration\n");
    printf("|---------------------------------\n");
    printf("| Domain width: %d\n", width);
    printf("| Domain height: %d\n", height);
    printf("| Domain depth: %d\n", depth);
    printf("| Iteration count: %d\n", iterations);
    printf("| Halo size: %d\n", halo_size);
    printf("| Halo depth: %d\n", halo_depth);
    printf("| Kernel: %s\n", kernel_names[kernel]);
    printf("| Domain distribution: %dx%dx%d\n", domain_distribution[0],
      domain_distribution[1], domain_distribution[2]);
    printf("| Compute threads available: %d\n", num_threads_available);
    printf("|---------------------------------\n");
  }

  if (width <= 0 || height <= 0 || depth <= 0)
  {
    error("Width, height and depth need a non-zero positive value\n", world_rank,
      NULL, NULL);
  }
  if (kernel < 0 || kernel >= kernel_count)
  {
    error("Kernel selected not valid\n", world_rank, NULL, NULL);
  }
  if (iterations <= 0)
  {
    error("A non-zero positive number of iterations is required\n", world_rank,
      NULL, NULL);
  }
  if (domain_distribution[0] <= 0 || domain_distribution[1] <= 0 ||
    domain_distribution[2] <= 0)
  {
    error("A non-zero positive distribution of chunks in all 3 axis is required\n",
      world_rank, NULL, NULL);
  }
  if (domain_distribution[0] * domain_distribution[1] * domain_distribution[2] !=
    world_size)
  {
    error("Requested domain decomposition dimensions is not possible with the
      available nodes\n", world_rank, NULL, NULL);
  }

  // Global domain data
  DATA_TYPE *domain = NULL;
  if (world_rank == 0)
  {
    domain = malloc(depth * height * width * sizeof(DATA_TYPE));
    for (int i = 0; i < depth * height * width; i++)
    {
      domain[i] = 1;
    }
  }

  double *timing_data = malloc(iterations * sizeof(double));
  double *timing_diff = malloc(iterations * sizeof(double));

  // Subdomain data
  DATA_TYPE *subdomain = NULL;
```

```
// Width, height, depth of every subdomain
domain_size_t *subdomain_sizes = NULL;

double totalBegin, totalEnd;
double computeBegin, computeEnd;
double comm_end, comp_end;
if (world_rank == 0)
{
  totalBegin = MPI_Wtime();
}

domain_decompose(domain, &subdomain, width, height, depth, world_size, world_rank,
  domain_distribution, &subdomain_sizes, halo_size, halo_depth);

domain_size_t subdomain_size = subdomain_sizes[world_rank];
int *point_offset = malloc(kernel_sizes[kernel] * kernel_sizes[kernel] *
  kernel_sizes[kernel] * sizeof(int));
for (int i = 0;
  i < kernel_sizes[kernel] * kernel_sizes[kernel] * kernel_sizes[kernel]; i++)
{
  int dx = (i % kernel_sizes[kernel]) - halo_size;
  int dy = ((i / kernel_sizes[kernel]) % kernel_sizes[kernel]) - halo_size;
  int dz = ((i / (kernel_sizes[kernel] * kernel_sizes[kernel])) %
    kernel_sizes[kernel]) - halo_size;

  point_offset[i] = (dz * subdomain_size.width * subdomain_size.height) +
    (dy * subdomain_size.width) + dx;
}
subdomain_size.depth = (subdomain_size.depth + (2 * halo_size * halo_depth));
subdomain_size.height = (subdomain_size.height + (2 * halo_size * halo_depth));
subdomain_size.width = (subdomain_size.width + (2 * halo_size * halo_depth));

DATA_TYPE *tmp_buffer = calloc(subdomain_size.depth * subdomain_size.height *
  subdomain_size.width, sizeof(DATA_TYPE));

pthread_t comm_thread;
int i = 0;
border_exchange_args_t comm_args;
comm_args.domain_distribution = domain_distribution;
comm_args.rank = world_rank;
comm_args.world_size = world_size;
comm_args.subdomain_data = subdomain;
comm_args.subdomain_size = &subdomain_size;
comm_args.halo_size = halo_size;
comm_args.halo_depth = halo_depth;
comm_args.timings = timing_data;
comm_args.iteration = &i;
comm_args.comm_end = &comm_end;

if (world_rank == 0)
{
  computeBegin = MPI_Wtime();
}
// Halo iteration given from (i % halo_depth) + 1
for (i = 0; i < iterations; i++)
{
  comm_args.subdomain_data = subdomain;
  pthread_create(&comm_thread, NULL, border_exchange, &comm_args);
  inner_compute(subdomain, tmp_buffer, subdomain_size.width,
    subdomain_size.height, subdomain_size.depth, kernels[kernel],
```

```
    kernel_sizes[kernel], halo_size, halo_depth, point_offset);

  // Handle exchange result, and place into subdomain
  // Wait pthread is finished :P
  comp_end = MPI_Wtime();
  pthread_join(comm_thread, NULL);
  timing_diff[i] = comm_end - comp_end;

  outer_compute(subdomain, tmp_buffer, subdomain_size.width,
    subdomain_size.height, subdomain_size.depth, kernels[kernel],
    kernel_sizes[kernel], halo_size, halo_depth, (i % halo_depth) + 1,
    point_offset);
  swap_buffers(&subdomain, &tmp_buffer);
}
if (world_rank == 0) {
  computeEnd = MPI_Wtime();
}

domain_recompose(domain, subdomain, width, height, depth, world_size, world_rank,
  domain_distribution, subdomain_sizes, halo_size, halo_depth);

if (world_rank == 0)
{
  totalEnd = MPI_Wtime();
  double time = totalEnd - totalBegin;
  printf("Total problem time: %.20f sec\n", time);
}
if (world_rank == 0)
{
  double time = computeEnd - computeBegin;
  printf("Total compute time: %.20f sec\n", time);
}
int cur_rank = 0;
while (cur_rank < world_size)
{
  int new_rank = cur_rank;
  if (world_rank == cur_rank)
  {
    for (int i = 0; i < iterations; i++)
    {
      printf("(%d,%d): %.20f sec\n", world_rank, i, timing_data[i]);
    }
    for (int i = 0; i < iterations; i++)
    {
      printf("(%d,%d): %.20f sec diff\n", world_rank, i, timing_diff[i]);
    }
    new_rank++;
  }
  MPI_Bcast(&new_rank, 1, MPI_INT, cur_rank, MPI_COMM_WORLD);
  cur_rank = new_rank;
}

free(tmp_buffer);
free(subdomain);
if (world_rank == 0)
{
  free(domain);
}
free(subdomain_sizes);
free(timing_data);
```

```
  free(timing_diff);
  free(point_offset);
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();

  return 0;
}
```

Code listing A.2: main.c

```
#include "compute.h"
#include <omp.h>

void outer_compute(DATA_TYPE *in, DATA_TYPE *out, int width, int height, int depth,
  DATA_TYPE *kernel, int kernel_size, int halo_size, int halo_depth,
  int halo_iteration, int *point_offset)
{
  int k3 = kernel_size * kernel_size * kernel_size;
  // Front face
#pragma omp parallel for collapse(2)
  for (int z = halo_size * halo_iteration; z < halo_size * (halo_depth + 1); z++)
  {
    for (int y = halo_size * halo_iteration;
      y < height - (halo_size * halo_iteration); y++)
    {
      int domain_pos_out = POSZYX(z, y, halo_size * halo_iteration);
      for (int x = halo_size * halo_iteration;
        x < width - (halo_size * halo_iteration); x++)
      {
        DATA_TYPE value = 0;
        for (int i = 0; i < k3; i++)
        {
          value += in[domain_pos_out + point_offset[i]] * kernel[i];
        }
        out[domain_pos_out++] = value;
      }
    }
  }

  // Back face
#pragma omp parallel for collapse(2)
  for (int z = depth - (halo_size * (halo_depth + 1));
    z < depth - (halo_size * halo_iteration); z++)
  {
    for (int y = halo_size * halo_iteration;
      y < height - (halo_size * halo_iteration); y++)
    {
      int domain_pos_out = POSZYX(z, y, halo_size * halo_iteration);
      for (int x = halo_size * halo_iteration;
        x < width - (halo_size * halo_iteration); x++)
      {
        DATA_TYPE value = 0;
        for (int i = 0; i < k3; i++)
        {
          value += in[domain_pos_out + point_offset[i]] * kernel[i];
        }
        out[domain_pos_out++] = value;
      }
    }
  }

  // Left face
#pragma omp parallel for collapse(2)
  for (int z = halo_size * (halo_depth + 1);
    z < depth - (halo_size * (halo_depth + 1)); z++)
  {
    for (int y = halo_size * halo_iteration;
      y < height - (halo_size * halo_iteration); y++)
    {
```

```
      int domain_pos_out = POSZYX(z, y, halo_size * halo_iteration);
      for (int x = halo_size * halo_iteration;
        x < halo_size * (halo_depth + 1); x++)
      {
        DATA_TYPE value = 0;
        for (int i = 0; i < k3; i++)
        {
          value += in[domain_pos_out + point_offset[i]] * kernel[i];
        }
        out[domain_pos_out++] = value;
      }
    }
  }

  // Right face
#pragma omp parallel for collapse(2)
  for (int z = halo_size * (halo_depth + 1);
    z < depth - (halo_size * (halo_depth + 1)); z++)
  {
    for (int y = halo_size * halo_iteration;
      y < height - (halo_size * halo_iteration); y++)
    {
      int domain_pos_out = POSZYX(z, y, width - (halo_size * (halo_depth + 1)));
      for (int x = width - (halo_size * (halo_depth + 1));
        x < width - (halo_size * halo_iteration); x++)
      {
        DATA_TYPE value = 0;
        for (int i = 0; i < k3; i++)
        {
          value += in[domain_pos_out + point_offset[i]] * kernel[i];
        }
        out[domain_pos_out++] = value;
      }
    }
  }

  // Top face
#pragma omp parallel for collapse(2)
  for (int z = halo_size * (halo_depth + 1);
    z < depth - (halo_size * (halo_depth + 1)); z++)
  {
    for (int y = halo_size * halo_iteration;
      y < halo_size * (halo_depth + 1); y++)
    {
      int domain_pos_out = POSZYX(z, y, halo_size * (halo_depth + 1));
      for (int x = halo_size * (halo_depth + 1);
        x < width - (halo_size * halo_iteration); x++)
      {
        DATA_TYPE value = 0;
        for (int i = 0; i < k3; i++)
        {
          value += in[domain_pos_out + point_offset[i]] * kernel[i];
        }
        out[domain_pos_out++] = value;
      }
    }
  }

  // Bottom face
#pragma omp parallel for collapse(2)
```

```
  for (int z = halo_size * (halo_depth + 1);
    z < depth - (halo_size * (halo_depth + 1)); z++)
  {
    for (int y = height - (halo_size * (halo_depth + 1));
      y < height - (halo_size * halo_iteration); y++)
    {
      int domain_pos_out = POSZYX(z, y, halo_size * (halo_depth + 1));
      for (int x = halo_size * (halo_depth + 1);
        x < width - (halo_size * halo_iteration); x++)
      {
        DATA_TYPE value = 0;
        for (int i = 0; i < k3; i++)
        {
          value += in[domain_pos_out + point_offset[i]] * kernel[i];
        }
        out[domain_pos_out++] = value;
      }
    }
  }
}

void inner_compute(DATA_TYPE *in, DATA_TYPE *out, int width, int height, int depth,
  DATA_TYPE *kernel, int kernel_size, int halo_size, int halo_depth,
  int *point_offset)
{
  int k3 = kernel_size * kernel_size * kernel_size;
  int upper_z = depth - (halo_size * (halo_depth + 1));
  int upper_y = height - (halo_size * (halo_depth + 1));
  int upper_x = width - (halo_size * (halo_depth + 1));
  int lower = halo_size * (halo_depth + 1);
#pragma omp parallel for collapse(2)
  for (int z = lower; z < upper_z; z++)
  {
    for (int y = lower; y < upper_y; y++)
    {
      int domain_pos_out = POSZYX(z, y, halo_size * (halo_depth + 1));
      for (int x = lower; x < upper_x; x++)
      {
        DATA_TYPE value = 0;
        for (int i = 0; i < k3; i++)
        {
          value += in[domain_pos_out + point_offset[i]] * kernel[i];
        }
        out[domain_pos_out++] = value;
      }
    }
  }
}
```

Code listing A.3: compute.c

```c
#include "domain_lifecycle.h"
#include "utils.h"
#if __has_include(<openmpi/mpi.h>)
#include <openmpi/mpi.h>
#else
#include <mpi.h>
#endif
#include <stdio.h>
#include <stdlib.h>

void domain_decompose(DATA_TYPE *domain, DATA_TYPE **subdomain, int width,
  int height, int depth, int world_size, int world_rank, int *domain_distribution,
  domain_size_t **subdomain_sizes, int halo_size, int halo_depth)
{
  calculate_subdomain_sizes(width, height, depth, world_size, domain_distribution,
    subdomain_sizes);
  int depth_with_halo =
    ((*subdomain_sizes)[world_rank].depth + (2 * halo_size * halo_depth));
  int height_with_halo =
    ((*subdomain_sizes)[world_rank].height + (2 * halo_size * halo_depth));
  int width_with_halo =
    ((*subdomain_sizes)[world_rank].width + (2 * halo_size * halo_depth));
  *subdomain = calloc(depth_with_halo * height_with_halo * width_with_halo,
    sizeof(DATA_TYPE));

  int *subdomain_counts = NULL;
  int *subdomain_displacement = NULL;
  int **subdomain_positions = NULL;

  calculate_subdomain_counts(&subdomain_counts, world_size, *subdomain_sizes);
  calculate_subdomain_displacements(&subdomain_displacement, world_size,
    subdomain_counts);
  calculate_subdomain_positions(&subdomain_positions, world_size,
    domain_distribution);

  DATA_TYPE *send_buffer = NULL;
  DATA_TYPE *recv_buffer = malloc(subdomain_counts[world_rank] * sizeof(DATA_TYPE));
  if (world_rank == 0)
  {
    send_buffer = malloc(width * height * depth * sizeof(DATA_TYPE));
    int pos = 0;
    for (int i = 0; i < world_size; i++)
    {
      int *subdomain_position = subdomain_positions[i];
      domain_size_t subdomain_size = (*subdomain_sizes)[i];
      for (int z = subdomain_position[2] * subdomain_size.depth;
        z < min((subdomain_position[2] + 1) * subdomain_size.depth, depth); z++)
      {
        for (int y = subdomain_position[1] * subdomain_size.height;
          y < min((subdomain_position[1] + 1) * subdomain_size.height, height); y++)
        {
          int domain_pos = POSZYX(z, y,
            (subdomain_position[0] * subdomain_size.width));
          for (int x = subdomain_position[0] * subdomain_size.width;
            x < min((subdomain_position[0] + 1) * subdomain_size.width, width); x++)
          {
            send_buffer[pos++] = domain[domain_pos++];
          }
        }
```

```
      }
    }
  }

  MPI_Scatterv(send_buffer, subdomain_counts, subdomain_displacement, MPI_DATA_TYPE,
    recv_buffer, subdomain_counts[world_rank], MPI_DATA_TYPE, 0, MPI_COMM_WORLD);

  domain_size_t subdomain_size = (*subdomain_sizes)[world_rank];
  for (int z = 0; z < subdomain_size.depth; z++)
  {
    for (int y = 0; y < subdomain_size.height; y++)
    {
      int pos = (z * subdomain_size.width * subdomain_size.height) +
        (y * subdomain_size.width);
      int subdomain_pos = POSZYXW(z + (halo_size * halo_depth),
        y + (halo_size * halo_depth), halo_size * halo_depth,
        width_with_halo, height_with_halo);
      for (int x = 0; x < subdomain_size.width; x++)
      {
        (*subdomain)[subdomain_pos++] = recv_buffer[pos++];
      }
    }
  }

  if (world_rank == 0)
  {
    free(send_buffer);
  }

  free(subdomain_counts);
  free(subdomain_displacement);
  free_subdomain_positions(subdomain_positions, world_size);
  free(recv_buffer);
}

void domain_recompose(DATA_TYPE *domain, DATA_TYPE *subdomain, int width,
  int height, int depth, int world_size, int world_rank, int *domain_distribution,
  domain_size_t *subdomain_sizes, int halo_size, int halo_depth)
{
  int *subdomain_counts = NULL;
  int *subdomain_displacement = NULL;
  int **subdomain_positions = NULL;
  domain_size_t subdomain_size = subdomain_sizes[world_rank];
  int depth_with_halo = subdomain_size.depth + (2 * halo_size * halo_depth);
  int height_with_halo = subdomain_size.height + (2 * halo_size * halo_depth);
  int width_with_halo = subdomain_size.width + (2 * halo_size * halo_depth);

  calculate_subdomain_counts(&subdomain_counts, world_size, subdomain_sizes);
  calculate_subdomain_displacements(&subdomain_displacement, world_size,
    subdomain_counts);
  calculate_subdomain_positions(&subdomain_positions, world_size,
    domain_distribution);

  DATA_TYPE *send_buffer = malloc(subdomain_counts[world_rank] * sizeof(DATA_TYPE));
  DATA_TYPE *recv_buffer = NULL;

  if (world_rank == 0)
  {
    recv_buffer = malloc(width * height * depth * sizeof(DATA_TYPE));
  }
```

```
  for (int z = 0; z < subdomain_size.depth; z++)
  {
    for (int y = 0; y < subdomain_size.height; y++)
    {
      int pos = POSZYW(z, y, subdomain_size.width, subdomain_size.height);
      int subdomain_pos = POSZYXW(z + (halo_size * halo_depth),
        y + (halo_size * halo_depth), halo_size * halo_depth,
        width_with_halo, height_with_halo);
      for (int x = 0; x < subdomain_size.width; x++)
      {
        send_buffer[pos++] = subdomain[subdomain_pos++];
      }
    }
  }

  MPI_Gatherv(send_buffer, subdomain_counts[world_rank], MPI_DATA_TYPE, recv_buffer,
    subdomain_counts, subdomain_displacement, MPI_DATA_TYPE, 0, MPI_COMM_WORLD);

  if (world_rank == 0)
  {
    int pos = 0;
    for (int i = 0; i < world_size; i++)
    {
      int *subdomain_position = subdomain_positions[i];
      domain_size_t subdomain_size = subdomain_sizes[i];
      for (int z = subdomain_position[2] * subdomain_size.depth;
        z < min((subdomain_position[2] + 1) * subdomain_size.depth, depth); z++)
      {
        for (int y = subdomain_position[1] * subdomain_size.height;
          y < min((subdomain_position[1] + 1) * subdomain_size.height, height); y++)
        {
          int domain_pos = POSZYX(z, y,
            (subdomain_position[0] * subdomain_size.width));
          for (int x = subdomain_position[0] * subdomain_size.width;
            x < min((subdomain_position[0] + 1) * subdomain_size.width, width); x++)
          {
            domain[domain_pos++] = recv_buffer[pos++];
          }
        }
      }
    }
    free(recv_buffer);
  }
  free(subdomain_counts);
  free(subdomain_displacement);
  free_subdomain_positions(subdomain_positions, world_size);
  free(send_buffer);
}

void calculate_subdomain_sizes(int width, int height, int depth, int world_size,
  int *domain_distribution, domain_size_t **subdomain_sizes)
{
  *subdomain_sizes = malloc(world_size * sizeof(domain_size_t));
  int i = 0;
  for (int z = 0; z < domain_distribution[2]; z++)
  {
    for (int y = 0; y < domain_distribution[1]; y++)
    {
      for (int x = 0; x < domain_distribution[0]; x++)
```

```
      {
        // Determining subdomain width
        if (x + 1 == domain_distribution[0])
        {
          int rest = width;
          for (int j = 0; j < domain_distribution[0] - 1; j++)
          {
            rest -= width / domain_distribution[0];
          }
          (*subdomain_sizes)[i].width = rest;
        } else {
          (*subdomain_sizes)[i].width = width / domain_distribution[0];
        }

        // Determining subdomain height
        if (y + 1 == domain_distribution[1])
        {
          int rest = height;
          for (int j = 0; j < domain_distribution[1] - 1; j++)
          {
            rest -= height / domain_distribution[1];
          }
          (*subdomain_sizes)[i].height = rest;
        } else {
          (*subdomain_sizes)[i].height = height / domain_distribution[1];
        }

        // Determining subdomain depth
        if (z + 1 == domain_distribution[2])
        {
          int rest = depth;
          for (int j = 0; j < domain_distribution[2] - 1; j++)
          {
            rest -= depth / domain_distribution[2];
          }
          (*subdomain_sizes)[i].depth = rest;
        } else {
          (*subdomain_sizes)[i].depth = depth / domain_distribution[2];
        }

        i++;
      }
    }
  }

  while (i < world_size)
  {
    (*subdomain_sizes)[i].width = 0;
    (*subdomain_sizes)[i].height = 0;
    (*subdomain_sizes)[i].depth = 0;
    i++;
  }
}

void calculate_subdomain_counts(int **subdomain_counts, int world_size,
  domain_size_t *subdomain_sizes)
{
  *subdomain_counts = malloc(world_size * sizeof(int));

  for (int i = 0; i < world_size; i++) {
```

```
    (*subdomain_counts)[i] = subdomain_sizes[i].width * subdomain_sizes[i].height *
      subdomain_sizes[i].depth;
  }
}

void calculate_subdomain_displacements(int **subdomain_displacements,
  int world_size, int *subdomain_counts)
{
  *subdomain_displacements = malloc(world_size * sizeof(int));

  int sum = 0;
  for (int i = 0; i < world_size; i++)
  {
    (*subdomain_displacements)[i] = sum;
    sum += subdomain_counts[i];
  }
}

void calculate_subdomain_positions(int ***subdomain_positions, int world_size,
  int *domain_distribution)
{
  *(subdomain_positions) = malloc(world_size * sizeof(int *));

  for (int i = 0; i < world_size; i++)
  {
    (*subdomain_positions)[i] = malloc(3 * sizeof(int));
    (*subdomain_positions)[i][0] = i % domain_distribution[0];
    (*subdomain_positions)[i][1] =
      (i % (domain_distribution[0] * domain_distribution[1])) /
      domain_distribution[0];
    (*subdomain_positions)[i][2] =
      i / (domain_distribution[0] * domain_distribution[1]);
  }
}

void free_subdomain_positions(int **subdomain_positions, int world_size)
{
  for (int i = 0; i < world_size; i++)
  {
    free(subdomain_positions[i]);
  }
  free(subdomain_positions);
}
```

Code listing A.4: domain_lifecycle.c

```c
#include "exchange.h"
#include "utils.h"
#if __has_include(<openmpi/mpi.h>)
#include <openmpi/mpi.h>
#else
#include <mpi.h>
#endif
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

// HaloThickness
#define HT (halo_size * halo_depth)

void *border_exchange(void *arguments)
{
  border_exchange_args_t *args = arguments;
  int world_rank = args->rank;
  int world_size = args->world_size;
  int halo_size = args->halo_size;
  int halo_depth = args->halo_depth;
  int *domain_distribution = args->domain_distribution;
  DATA_TYPE *subdomain = args->subdomain_data;
  domain_size_t *domain_size = args->subdomain_size;
  int **subdomain_positions = NULL;
  calculate_subdomain_positions(&subdomain_positions, world_size,
    domain_distribution);

  double commBegin, commEnd;
  commBegin = MPI_Wtime();
  north_south_exchange(subdomain, domain_size->width, domain_size->height,
    domain_size->depth, world_rank, world_size, halo_size, halo_depth,
    subdomain_positions, domain_distribution);
  east_west_exchange(subdomain, domain_size->width, domain_size->height,
    domain_size->depth, world_rank, world_size, halo_size, halo_depth,
    subdomain_positions, domain_distribution);
  front_back_exchange(subdomain, domain_size->width, domain_size->height,
    domain_size->depth, world_rank, world_size, halo_size, halo_depth,
    subdomain_positions, domain_distribution);

  commEnd = MPI_Wtime();
  double time = commEnd - commBegin;
  args->timings[*(args->iteration)] = time;

  free_subdomain_positions(subdomain_positions, world_size);
  *args->comm_end = MPI_Wtime();
  pthread_exit(NULL);
}

void north_south_exchange(DATA_TYPE *domain, int width, int height, int depth,
  int world_rank, int world_size, int halo_size, int halo_depth,
  int **subdomain_positions, int *domain_distribution)
{
  int request_count = (subdomain_positions[world_rank][1] > 0) +
    (subdomain_positions[world_rank][1] < domain_distribution[1] - 1);
  if (request_count == 0)
  {
    return;
  }
```

```
MPI_Request *requests = malloc(2 * request_count * sizeof(MPI_Request));
MPI_Status *statuses = malloc(2 * request_count * sizeof(MPI_Status));
int neighbor_rank = -1;
int cursor = 0;
DATA_TYPE **send_buffer = malloc(request_count * sizeof(DATA_TYPE *));
DATA_TYPE **recv_buffer = malloc(request_count * sizeof(DATA_TYPE *));

// true if a neighbor is present above
if (find_rank_at(subdomain_positions[world_rank][0],
  subdomain_positions[world_rank][1] - 1, subdomain_positions[world_rank][2],
  world_size, &neighbor_rank, subdomain_positions))
{
  int buffer_size = width * HT * depth;
  send_buffer[cursor] = malloc(buffer_size * sizeof(DATA_TYPE));
  recv_buffer[cursor] = malloc(buffer_size * sizeof(DATA_TYPE));
  int i = 0;
  for (int z = 0; z < depth; z++) {
    for (int y = HT; y < 2 * HT; y++) {
      int domain_pos = POSZY(z, y);
      for (int x = 0; x < width; x++) {
        send_buffer[cursor][i++] = domain[domain_pos++];
      }
    }
  }
  MPI_Irecv(recv_buffer[cursor], buffer_size, MPI_DATA_TYPE, neighbor_rank, 0,
    MPI_COMM_WORLD, &requests[2 * cursor]);
  MPI_Isend(send_buffer[cursor], buffer_size, MPI_DATA_TYPE, neighbor_rank, 0,
    MPI_COMM_WORLD, &requests[(2 * cursor) + 1]);
  cursor++;
}

// true if a neighbor is present bellow
if (find_rank_at(subdomain_positions[world_rank][0],
  subdomain_positions[world_rank][1] + 1, subdomain_positions[world_rank][2],
  world_size, &neighbor_rank, subdomain_positions))
{
  int buffer_size = width * HT * depth;
  send_buffer[cursor] = malloc(buffer_size * sizeof(DATA_TYPE));
  recv_buffer[cursor] = malloc(buffer_size * sizeof(DATA_TYPE));
  int i = 0;
  for (int z = 0; z < depth; z++) {
    for (int y = height - (2 * HT); y < height - HT; y++) {
      int domain_pos = POSZY(z, y);
      for (int x = 0; x < width; x++) {
        send_buffer[cursor][i++] = domain[domain_pos++];
      }
    }
  }
  MPI_Irecv(recv_buffer[cursor], buffer_size, MPI_DATA_TYPE, neighbor_rank, 0,
    MPI_COMM_WORLD, &requests[2 * cursor]);
  MPI_Isend(send_buffer[cursor], buffer_size, MPI_DATA_TYPE, neighbor_rank, 0,
    MPI_COMM_WORLD, &requests[(2 * cursor) + 1]);
  cursor++;
}

MPI_Waitall(2 * request_count, requests, statuses);
cursor = 0;

// true if a neighbor is present above
if (find_rank_at(subdomain_positions[world_rank][0],
```

```
    subdomain_positions[world_rank][1] - 1, subdomain_positions[world_rank][2],
    world_size, &neighbor_rank, subdomain_positions))
  {
    int i = 0;
    for (int z = 0; z < depth; z++) {
      for (int y = 0; y < HT; y++) {
        int domain_pos = POSZY(z, y);
        for (int x = 0; x < width; x++) {
          domain[domain_pos++] = recv_buffer[cursor][i++];
        }
      }
    }
    free(send_buffer[cursor]);
    free(recv_buffer[cursor]);
    cursor++;
  }

  // true if a neighbor is present bellow
  if (find_rank_at(subdomain_positions[world_rank][0],
    subdomain_positions[world_rank][1] + 1, subdomain_positions[world_rank][2],
    world_size, &neighbor_rank, subdomain_positions))
  {
    int i = 0;
    for (int z = 0; z < depth; z++) {
      for (int y = height - HT; y < height; y++) {
        int domain_pos = POSZY(z, y);
        for (int x = 0; x < width; x++) {
          domain[domain_pos++] = recv_buffer[cursor][i++];
        }
      }
    }
    free(send_buffer[cursor]);
    free(recv_buffer[cursor]);
  }
  free(send_buffer);
  free(recv_buffer);
  free(requests);
  free(statuses);
}

void east_west_exchange(DATA_TYPE *domain, int width, int height, int depth,
  int world_rank, int world_size, int halo_size, int halo_depth,
  int **subdomain_positions, int *domain_distribution)
{
  int request_count = (subdomain_positions[world_rank][0] > 0) +
    (subdomain_positions[world_rank][0] < domain_distribution[0] - 1);
  if (request_count == 0)
  {
    return;
  }
  MPI_Request *requests = malloc(2 * request_count * sizeof(MPI_Request));
  MPI_Status *statuses = malloc(2 * request_count * sizeof(MPI_Status));
  int neighbor_rank = -1;
  int cursor = 0;
  DATA_TYPE **send_buffer = malloc(request_count * sizeof(DATA_TYPE *));
  DATA_TYPE **recv_buffer = malloc(request_count * sizeof(DATA_TYPE *));

  // true if a neighbor is present to the left
  if (find_rank_at(subdomain_positions[world_rank][0] - 1,
    subdomain_positions[world_rank][1], subdomain_positions[world_rank][2],
```

```
  world_size, &neighbor_rank, subdomain_positions))
{
  int buffer_size = HT * height * depth;
  send_buffer[cursor] = malloc(buffer_size * sizeof(DATA_TYPE));
  recv_buffer[cursor] = malloc(buffer_size * sizeof(DATA_TYPE));
  int i = 0;
  for (int z = 0; z < depth; z++) {
    for (int y = 0; y < height; y++) {
      int domain_pos = POSZYX(z, y, HT);
      for (int x = HT; x < 2 * HT; x++) {
        send_buffer[cursor][i++] = domain[domain_pos++];
      }
    }
  }
  MPI_Irecv(recv_buffer[cursor], buffer_size, MPI_DATA_TYPE, neighbor_rank, 0,
    MPI_COMM_WORLD, &requests[2 * cursor]);
  MPI_Isend(send_buffer[cursor], buffer_size, MPI_DATA_TYPE, neighbor_rank, 0,
    MPI_COMM_WORLD, &requests[(2 * cursor) + 1]);
  cursor++;
}

// true if a neighbor is present to the right
if (find_rank_at(subdomain_positions[world_rank][0] + 1,
  subdomain_positions[world_rank][1], subdomain_positions[world_rank][2],
  world_size, &neighbor_rank, subdomain_positions))
{
  int buffer_size = HT * height * depth;
  send_buffer[cursor] = malloc(buffer_size * sizeof(DATA_TYPE));
  recv_buffer[cursor] = malloc(buffer_size * sizeof(DATA_TYPE));
  int i = 0;
  for (int z = 0; z < depth; z++) {
    for (int y = 0; y < height; y++) {
      int domain_pos = POSZYX(z, y, width - (2 * HT));
      for (int x = width - (2 * HT); x < width - HT; x++) {
        send_buffer[cursor][i++] = domain[domain_pos++];
      }
    }
  }
  MPI_Irecv(recv_buffer[cursor], buffer_size, MPI_DATA_TYPE, neighbor_rank, 0,
    MPI_COMM_WORLD, &requests[2 * cursor]);
  MPI_Isend(send_buffer[cursor], buffer_size, MPI_DATA_TYPE, neighbor_rank, 0,
    MPI_COMM_WORLD, &requests[(2 * cursor) + 1]);
  cursor++;
}

MPI_Waitall(2 * request_count, requests, statuses);
cursor = 0;

// true if a neighbor is present to the left
if (find_rank_at(subdomain_positions[world_rank][0] - 1,
  subdomain_positions[world_rank][1], subdomain_positions[world_rank][2],
  world_size, &neighbor_rank, subdomain_positions))
{
  int i = 0;
  for (int z = 0; z < depth; z++) {
    for (int y = 0; y < height; y++) {
      int domain_pos = POSZY(z, y);
      for (int x = 0; x < HT; x++) {
        domain[domain_pos++] = recv_buffer[cursor][i++];
      }
```

```
      }
    }
    free(send_buffer[cursor]);
    free(recv_buffer[cursor]);
    cursor++;
  }

  // true if a neighbor is present to the right
  if (find_rank_at(subdomain_positions[world_rank][0] + 1,
    subdomain_positions[world_rank][1], subdomain_positions[world_rank][2],
    world_size, &neighbor_rank, subdomain_positions))
  {
    int i = 0;
    for (int z = 0; z < depth; z++) {
      for (int y = 0; y < height; y++) {
        int domain_pos = POSZYX(z, y, width - HT);
        for (int x = width - HT; x < width; x++) {
          domain[domain_pos++] = recv_buffer[cursor][i++];
        }
      }
    }
    free(send_buffer[cursor]);
    free(recv_buffer[cursor]);
  }
  free(send_buffer);
  free(recv_buffer);
  free(requests);
  free(statuses);
}

void front_back_exchange(DATA_TYPE *domain, int width, int height, int depth,
  int world_rank, int world_size, int halo_size, int halo_depth,
  int **subdomain_positions, int *domain_distribution)
{
  int request_count = (subdomain_positions[world_rank][2] > 0) +
    (subdomain_positions[world_rank][2] < domain_distribution[2] - 1);
  if (request_count == 0)
  {
    return;
  }
  MPI_Request *requests = malloc(2 * request_count * sizeof(MPI_Request));
  MPI_Status *statuses = malloc(2 * request_count * sizeof(MPI_Status));
  int neighbor_rank = -1;
  int cursor = 0;
  DATA_TYPE **send_buffer = malloc(request_count * sizeof(DATA_TYPE *));
  DATA_TYPE **recv_buffer = malloc(request_count * sizeof(DATA_TYPE *));

  // true if a neighbor is present in front
  if (find_rank_at(subdomain_positions[world_rank][0],
    subdomain_positions[world_rank][1], subdomain_positions[world_rank][2] - 1,
    world_size, &neighbor_rank, subdomain_positions))
  {
    int buffer_size = width * height * HT;
    send_buffer[cursor] = malloc(buffer_size * sizeof(DATA_TYPE));
    recv_buffer[cursor] = malloc(buffer_size * sizeof(DATA_TYPE));
    int i = 0;
    for (int z = HT; z < 2 * HT; z++) {
      for (int y = 0; y < height; y++) {
        int domain_pos = POSZY(z, y);
        for (int x = 0; x < width; x++) {
```

```
        send_buffer[cursor][i++] = domain[domain_pos++];
      }
    }
  }
  MPI_Irecv(recv_buffer[cursor], buffer_size, MPI_DATA_TYPE, neighbor_rank, 0,
    MPI_COMM_WORLD, &requests[2 * cursor]);
  MPI_Isend(send_buffer[cursor], buffer_size, MPI_DATA_TYPE, neighbor_rank, 0,
    MPI_COMM_WORLD, &requests[(2 * cursor) + 1]);
  cursor++;
}

// true if a neighbor is present behind
if (find_rank_at(subdomain_positions[world_rank][0],
  subdomain_positions[world_rank][1], subdomain_positions[world_rank][2] + 1,
  world_size, &neighbor_rank, subdomain_positions))
{
  int buffer_size = width * height * HT;
  send_buffer[cursor] = malloc(buffer_size * sizeof(DATA_TYPE));
  recv_buffer[cursor] = malloc(buffer_size * sizeof(DATA_TYPE));
  int i = 0;
  for (int z = depth - (2 * HT); z < depth - HT; z++) {
    for (int y = 0; y < height; y++) {
      int domain_pos = POSZY(z, y);
      for (int x = 0; x < width; x++) {
        send_buffer[cursor][i++] = domain[domain_pos++];
      }
    }
  }
  MPI_Irecv(recv_buffer[cursor], buffer_size, MPI_DATA_TYPE, neighbor_rank, 0,
    MPI_COMM_WORLD, &requests[2 * cursor]);
  MPI_Isend(send_buffer[cursor], buffer_size, MPI_DATA_TYPE, neighbor_rank, 0,
    MPI_COMM_WORLD, &requests[(2 * cursor) + 1]);
  cursor++;
}

MPI_Waitall(2 * request_count, requests, statuses);
cursor = 0;

// true if a neighbor is present in front
if (find_rank_at(subdomain_positions[world_rank][0],
  subdomain_positions[world_rank][1], subdomain_positions[world_rank][2] - 1,
  world_size, &neighbor_rank, subdomain_positions))
{
  int i = 0;
  for (int z = 0; z < HT; z++) {
    for (int y = 0; y < height; y++) {
      int domain_pos = POSZY(z, y);
      for (int x = 0; x < width; x++) {
        domain[domain_pos++] = recv_buffer[cursor][i++];
      }
    }
  }
  free(send_buffer[cursor]);
  free(recv_buffer[cursor]);
  cursor++;
}

// true if a neighbor is present behind
if (find_rank_at(subdomain_positions[world_rank][0],
  subdomain_positions[world_rank][1], subdomain_positions[world_rank][2] + 1,
```

```
    world_size, &neighbor_rank, subdomain_positions))
  {
    int i = 0;
    for (int z = depth - HT; z < depth; z++) {
      for (int y = 0; y < height; y++) {
        int domain_pos = POSZY(z, y);
        for (int x = 0; x < width; x++) {
          domain[domain_pos++] = recv_buffer[cursor][i++];
        }
      }
    }
    free(send_buffer[cursor]);
    free(recv_buffer[cursor]);
  }
  free(send_buffer);
  free(recv_buffer);
  free(requests);
  free(statuses);
}
```

Code listing A.5: exchange.c

```c
#include "utils.h"
#include <stdio.h>

int min(int a, int b) {
  return (a < b) ? a : b;
}

int ceil_div(int a, int b) {
  double _a = a, _b = b;
  double d = _a / _b;
  int r = (int)d;
  if (d > r) {
    return r + 1;
  }
  return r;
}

int find_rank_at(int x, int y, int z, int world_size, int *neighbor_rank,
  int **subdomain_positions)
{
  for (int i = 0; i < world_size; i++)
  {
    if (subdomain_positions[i][0] == x && subdomain_positions[i][1] == y &&
      subdomain_positions[i][2] == z)
    {
      *neighbor_rank = i;
      return 1;
    }
  }
  *neighbor_rank = -1;
  return 0;
}
```

Code listing A.6: utils.c

```
// utils.h
#define POSZ(Z) (width * height * (Z))
#define POSZW(Z, W, H) (W * H * (Z))
#define POSZY(Z, Y) ((width * height * (Z)) + (width * (Y)))
#define POSZYW(Z, Y, W, H) ((W * H * (Z)) + (W * (Y)))
#define POSZYX(Z, Y, X) ((width * height * (Z)) + (width * (Y)) + (X))
#define POSZYXW(Z, Y, X, W, H) ((W * H * (Z)) + (W * (Y)) + (X))

// constants.h
#define DATA_TYPE float
#define MPI_DATA_TYPE MPI_FLOAT

// domain_lifecycle.h
typedef struct domain_size_t
{
  int width;
  int height;
  int depth;
} domain_size_t;

// exchange.h
typedef struct
{
  int world_size;
  int rank;
  DATA_TYPE *subdomain_data;
  domain_size_t *subdomain_size;
  int *domain_distribution;
  int halo_size;
  int halo_depth;
  double *timings;
  int *iteration;
  double *comm_end;
} border_exchange_args_t;
```

Code listing A.7: Various defines and structs

Appendix B

Timing results

Table B.1: All median results from all runs performed.

Run	Compute time	Problem time	Exchange time	Exchange/compute difference
1d-118	8.203	8.548	0.011	0.010
1d-12s	4.074	4.681	0.005	0.005
1d-12w	7.645	8.293	0.009	0.009
1d-1s	38.597	38.729	0.000	-0.127
1d-1w	4.654	4.672	0.000	0.000
1d-222	7.021	7.383	0.024	0.024
1d-4s	12.552	12.809	0.010	0.009
1d-4w	6.309	6.497	0.008	0.008
1d-big-e	803.652	812.974	1.685	-7.226
1d-big	714.938	728.532	6.934	0.637
1d	7.291	7.619	0.027	0.027
1d-sub	81.305	82.86	0.567	0.115
3d-118	10.415	10.788	0.013	0.013
3d-222	10.472	10.85	0.019	0.019
3d	10.598	11.017	0.013	0.013
3d-sub	75.917	76.878	0.506	0.055
s-1d-big	190.251	203.575	0.248	-1.421
c-1d	3.499	13.589	0.017	0.000
c-1d-big	163.657	807.652	1.503	-0.627
c-1d-unroll	2.505	12.608	0.028	0.027

Appendix C

Poster

NTNU
Norwegian University of Science and Technology

Analyzing Halo Computations on Multicore CPUs

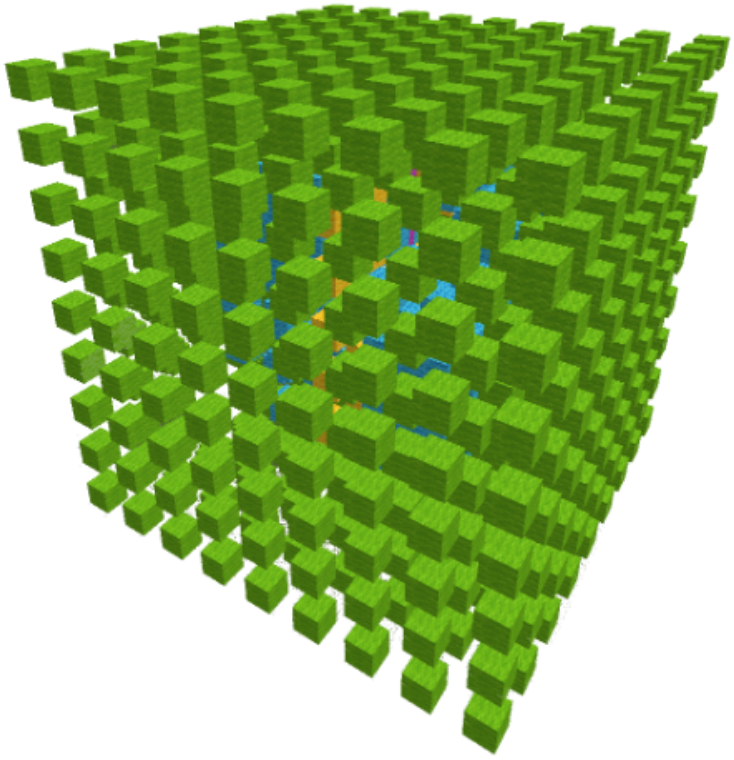
Master thesis by Andreas Hammer.
Advisor: Professor Anne C. Elster

Compute heavy problems, like finite difference calculations on a large problem domain, are time-consuming and might not fit in a single system/node's memory. This prompts parallelization as it can spread the memory load onto multiple nodes and perform the compute on each node using all available cores to minimize compute time. The imposed stencil computations can be parallelized on multiple nodes using halo exchange to communicate the neighboring values to ensure a correct result, with as little performance degradation as possible. This thesis looks into 3D stencil computation with halo exchange on a shared resource cluster.

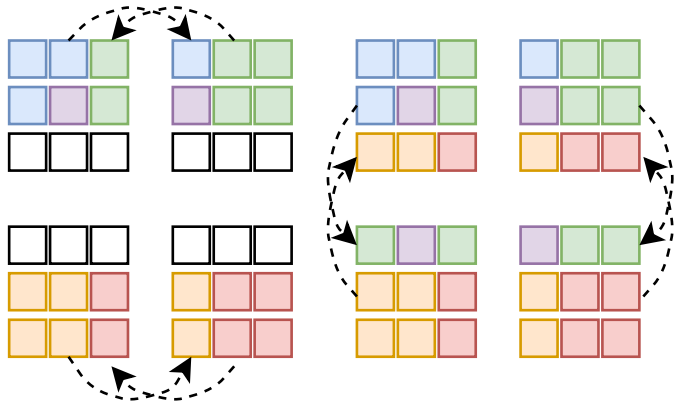
We implemented 3D stencil computation with halo exchange. Our implementation was optimized by changing the underlying data structure from a 3D array to a 1D array. We also pre-calculated the memory strides and tested stencil unrolling. The tests were performed on a shared resource cluster and some local workstations to see how the halo exchange performed and how our optimizations affected the compute time.

The optimizations show an 11.15x speed-up with 1D data structure and pre-calculated strides. Our quick test of stencil unrolling shows an additional speed-up of 1.40x.

The halo exchange on a shared cluster, where other users use the other available resources, shows 7-91% of the exchange time hidden by computation. When testing with exclusive node access, we observe 529%-902% of the exchange hidden.

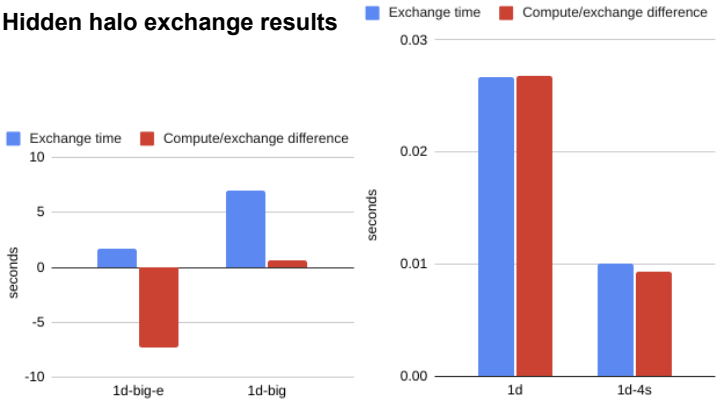


Halo exchange considerations



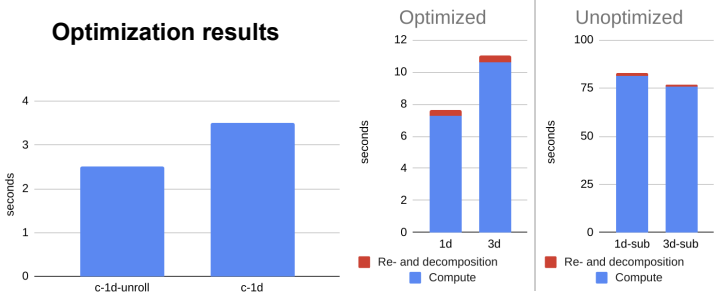
Synchronization between exchange directions and inclusion of halo data in transfers ensures diagonal data transfers without additional exchanges needed [1].

Hidden halo exchange results



- 7% hidden with shared nodes.
- 91% hidden w/big domain on shared nodes.
- 529% hidden w/big domain & exclusive nodes.

Optimization results



- Change from 3D to 1D data structure (c-1d, 1d-sub, 1d).
- Pre-calculate memory strides (c-1d, 1d, 3d).
- Stencil unrolling 27 to 7 points calculated (c-1d-unroll).
- 11.15x speed-up with pre-calculation and data structure change.
- 1.40x further speed-up with stencil unrolling.

Future work

- Test how deep halos can affect performance of total exchange latency and compute time.
- Implement hardware based task partitioning to reduce node wait time since different hardware compute faster (clock speed, memory speed, etc.).
- Use AVX instructions in stencil computations to improve compute time by calculating multiple values at once.
- Further test stencil unrolling, how it

References

[1] F. B. Kjolstad and M. Snir, “Ghost Cell Pattern,” in Proceedings of the 2010 Workshop on Parallel Programming Patterns - ParaPLoP ’10, Carefree, Arizona, 2010, pp. 1–9. doi: 10.1145/1953611.1953615.
[2] A. Hammer, "Study of two-dimensional deep halo exchange using MPI blocking and non-blocking communication,". Fall project 2020.

Appendix D

Fall project

Norwegian University of
Science and Technology

Study of two-dimensional deep halo exchange using MPI blocking and non-blocking communication

Author
Andreas Hammer

Specialization Project in Computer Science
Department of Computer Science
Norwegian University of Science and Technology,

June 9, 2021

Supervisor Professor Anne C. Elster

Abstract

Iterative stencil-based computing on large datasets is a time-consuming endeavor and with the lack of gain in single-core performance over the last years solving such problems in parallel is a way to minimize the time for a solution. This paper looks into the use of deep halos in edge detection compute to figure out what halo size is beneficial to achieve the best performance. Further, we look at the difference using blocking and non-blocking MPI calls during the halo exchange to see what performance increase this can have. Given the different halo sizes on the 10k by 10k image, we found a halo size of 2 was most beneficial. We also found that given the same communication pattern between blocking and non-blocking communication performs about the same.

Contents

List of Figures

List of Tables

Listings

1 Introduction

Iterative computing on large datasets is a time-consuming endeavor and with the lack of gain in single-core performance over the last years solving such problems in parallel is a way to minimize the time for a solution. Parallelizing iterative workloads with large datasets like stencil-based edge detection often require information about the surrounding cells to calculate the next iteration in the algorithm. Parallelization of such algorithms usually decompose the problem domain (e.g., pixel data in an image) and spread the computation throughout the compute resources available.

Solving problems using iterative algorithms in a timely manner can be a difficult feat, and in this paper we will test out some different approaches to halo exchange between different compute nodes to ensure both a correct result and a time-efficient solution. What speedups can be achieved by using deep halos to exchange communication for computations when performing a stencil-based edge detection algorithm? Is there a difference in performance when using blocking versus non-blocking message-passing for stencil-based edge detection?

First following this introduction is the chapter 2 giving a theoretical basis of the techniques used to illuminate where halo exchange performs best. Chapter 3 presents the implemented solution used for the tests and how the halo exchange and computations are performed. Chapter 4 talks about the results from the experiment and put them in the context of this paper's goal, talking about possible error sources and how the experimental results were obtained using the implemented solution. Chapter 5 summarises the results and concludes the questions posited, and looking into the next step in researching this topic.

2 Background

2.1 Computation cluster

A computation cluster is a collection of nodes or computers, interconnected for fast communication between each other. A node usually consists of its own local memory and one or more CPUs or GPUs, which are accessible for use when performing computations on the cluster. From a programmer's perspective, a cluster can be viewed as one machine with a lot of processing power capable of massive parallelism of one or more workloads. The cluster's scheduling software usually distributes the workload throughout the available nodes and the number of nodes requested. The type of hardware may differ when running on different nodes dependent on the cluster being homogenous with all equal nodes or heterogeneous with different hardware configurations throughout the cluster.

2.2 Integer factorization

Integer factorization is the process of calculating a composite number's smaller integer components. A branch of factorization is prime factorization, where the integer components only consist of prime numbers. [3] The easiest factorization algorithm to implement is using trial division. For small numbers, the performance will not differ too much, but this is a slow algorithm when computing for larger numbers. More performant algorithms are the Fermat factorization, Pillord rho, brent factorization and Pillord 1-p algorithms. The last three mentioned performs about the same, with the Fermat factorization performing about the same with numbers smaller than four digits.[4]

2.3 Closest pair probleam

The closest pair problem in computation geometry is how to figure which pair of points in space are closest together. Solving this naively can be done with a brute-force approach testing the distance between all points, with the resulting points being closest. When calculating the points distance subtracting the location of point b from point a with the absolute value being the distance between them. The pair with the shortest distance results in the closest pair. [5] This distance comparison calculation can be used not just to find which two points in space are closest together but also to find which predefined pairs are closest together.

2.4 Geometric decomposition problem

When working on one- or multi-dimensional arrays of data in parallel, the decomposition of such data into smaller subdomains or chunks is beneficial. This decomposition or splitting a region into smaller subregions based on geometry (e.g., an image into smaller subimages or an array of numbers into smaller arrays of numbers) is called geometric decomposition. The optimal granularity of the decomposition is dependent on the architecture of the runtime system. In a distributed memory system, matching the chunk sizes to fit within one package or the exchange data needed fits one such package can increase efficiency when using the interconnects before or during the computations. For shared-memory systems where the cost

of synchronization is fairly low, matching chunk sizes to the system's memory access patterns is beneficial to get the best performance.[6]

2.5 Parallel programming models

There are several parallel programming models such as data parallelism, remote memory access, shared-memory models, and message-passing models. A parallel programming model is an abstract description of a parallel system's operations, such as communication between processes, how shared-memory is handled, and how spawning of tasks or processes occurs. It abstracts away from specific hardware systems, making it theoretically possible to implement any parallel programming model on any hardware system, tho with different performance results.[7]

Data parallelism describes a Single Instruction Multiple Data (SIMD) system where vectors of data are processed in parallel by applying the same computation/instruction on each element (e.g., Graphics Processing Units (GPUs)).

Remote memory access is a model describing a shared memory message-passing hybrid model where memory located outside the processes memory space is made accessible through one-sided communication (e.g., using message-passing) without the overhead of point-to-point communication.

2.5.1 Shared memory

Control parallelism is a form of parallelism that is explicitly specified by the programmer as opposed to implicit parallelism due to independent data. One simple form of control parallelism is the shared memory programming model. This model uses threads of execution that run asynchronously in parallel operating on the same piece of memory. Like the name entails, the entire parallel environment operates within the same shared address space, as shown in figure **??**. Reading and writing to a shared memory simultaneously from multiple threads can cause conflicts and race-conditions. This issue is usually handled by using a mutex or semaphore to lock all threads accessing the resource to prevent conflicts.[1][7]

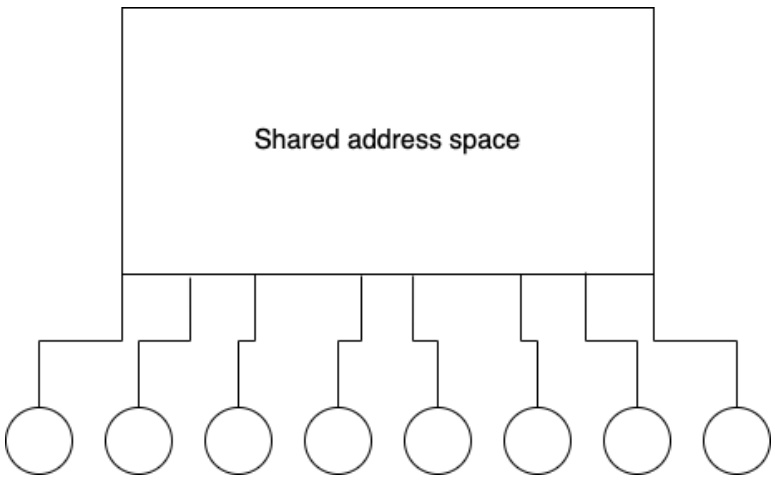


Figure 1: Multiple processes operating on the same address space. (Based on figure 1.1[1])

Programming shared-memory parallel programs, one may use libraries such as OpenMP or Pthreads to facilitate thread life cycles. These functions in different ways. Pthreads use the POSIX threads system calls to manage the lifecycle of threads. OpenMP uses preproccessor directives within the compiler to inject structured blocks to control the OpenMP runtime library's parallelization. [8]

2.5.2 Message passing

Message passing is a programming model consisting of independent processes, all with their own local memory. The communication between the processes is done using messages entailing both processes need to perform some operations to achieve this goal. Through such messages, the data stored in other processes can be shared with all system processes. Figure **??** shows such a system with separate address spaces for each process all connected through a network. This network can be as simple as two computers connected via a switch to multiple computational clusters with their high speed interconnects all connected through the internet with messages passed between the different clusters for large scale parallelism.[1]

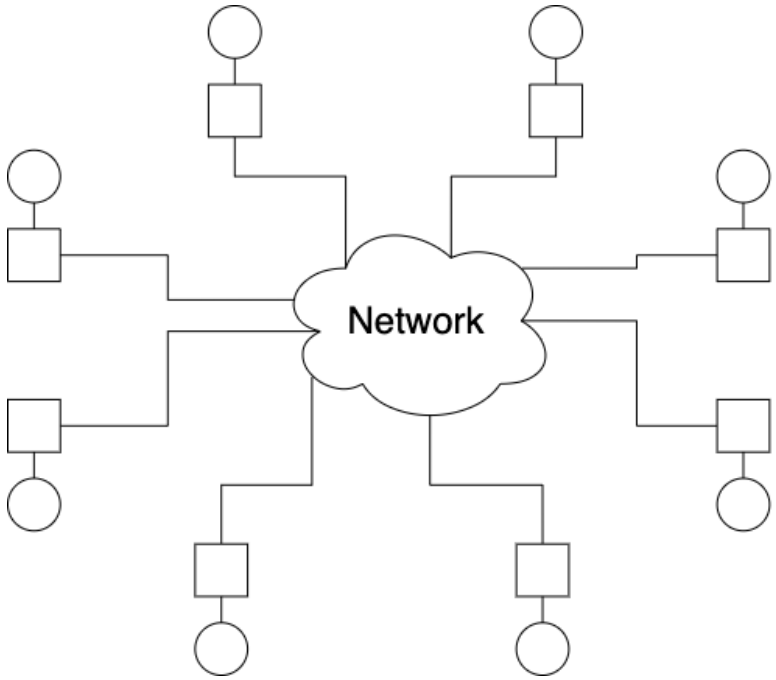


Figure 2: Message passing model with seperate address spaces for each process, interconnected through an arbitrary network. (Based on figure 1.2[1])

2.6 MPI

Message-Passing Interface (MPI)[1][9] is a library specification defined to primarily address the message-passing parallel programming model, where data is moved between processes through various operations on each process. The specification defines the names, calling sequences, and results of cooperative operations, meaning one process cannot force a message upon another without the recipient actively listening. Programs utilizing an MPI implementation follows the model SPMD (single-program multiple-data) with the same code executed in all the processes, but the data operated upon is different for each process. There are mul-

tiple implementations of the MPI specification, where each parallel computer vendor offers an implementation for their machines and other free publicly available implementations like Open MPI and MPICH.

2.6.1 MPI consepts

Parallelization of algorithms that require communication between the different processes during the program's execution contains *syncronization points*. These are specific points in the program where some or all processes synchronize their progression, meaning waiting for the other processes to reach the points before continuing. An example of this is a program taking an array of numbers and scattering them to different processes where each subset of the array is summed, and the resulting sum gathered at one process to be added together, producing the total sum of the original array. The points where the process scatters the array values and gathers the resulting sums are synchronization points for this program as all processes are required to reach this instruction before any of them can continue the program's execution. Another example is when two processes communicate directly with each other using a send and recieve call synchronizing their progress.

Barriers are one such global synchronization point where all processes in a given communicator wait for every other process to reach the same point. This functionality is also seen in other collective operations.

In MPI execution context and group of processes are both represented by the same consept, a *communicator*. This structure is often used as a parameter in point-to-point operations as the destination/source rank specified is in the communicator's context. In most MPI implementations, a *MPI_COMM_WORLD* is supplied as the communicator for all spawned processes for the program. Both the size and the specific process' rank within a communicator is found via the MPI_Comm_size and MPI_Comm_rank calls. The resulting values from these calls in the MPI_COMM_WORLD communicator are often referred to as world size and world rank.

2.6.2 MPI operations

- **MPI_Barrier** halts the execution untill all processes in the given communicator has reached this point in the program.
- **MPI_Scatterv** distributes a non-uniform array of data to all processes in the communicator. This enables the scattering of different amounts of data to each process from a source process.
- **MPI_Gatherv** collects all data from all processes in the communicator, enabling different amounts of data from each process into one array at the source process.
- **MPI_Sendrecv** sends and receives data between two processes, removing the fear of a deadlock where both are either sending or receiving at once.
- **MPI_Isend** asynchronously sends data to another process.
- **MPI_Irecv** asynchronously receives data from another process.
- **MPI_Waitall** waits for all provided requests from asynchronous operations are finished.

2.7 Edge detection

In image analysis, one of the more fundamental operations is edge detection, and it is the detection of boundaries separating different image regions based on different features, often

gray level or luminance.[10] There are different approaches to find edges in images, gradient edge detectors, zero crossing, Laplacian of Gaussian, gaussian detectors, and colored edge detectors. These vary in noise sensitivity, what area is used to decide if there is an edge with Gaussian methods using a larger area than more classic methods like gradient detectors and zero crossing detectors. When deciding what algorithm to use, the data and type of edges need to be taken into account as some edge detectors are faster but more prone to noise and inaccuracies. More accurate algorithms are more time-consuming, like colored detectors and the gaussian detector without Laplace both are time-consuming compared to the other algorithms. [11]

2.7.1 Laplace operator

The laplacian of an image highlights the regions with rapid change in intensity making it good for edge detection. One drawback with laplace edge detection is the sensitivity to noise, due to this one often choose to smooth the image to get rid of most noise. The operator it self is derived twice from the laplacian equation (2.1) both with respect to x and y.[10]

$$\hat{\Delta}^2 f(n_1, n_2) = f(n_1, n_2) * h(n_1, n_2) \tag{2.1}$$

The double derived with regards to x and y are then combined and calculated for the surrounding values to achieve an operator 2.2.

$$\begin{aligned} & f_{xx}(n_1, n_2) + f_{yy}(n_1, n_2) \\ = & f(n_1+1, n_2) + f(n_1-1, n_2) + f(n_1, n_2+1) + f(n_1, n_2-1) - 4f(n_1, n_2) \\ = & \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \end{aligned} \tag{2.2}$$

Dependent on the chosen size of the applied noise reduction filter, derivative approximator and the kernel size, other 3x3 filters possible includes the two filters in equation 2.3.

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} -1 & 2 & -1 \\ 2 & -4 & 2 \\ -1 & 2 & -2 \end{bmatrix} \tag{2.3}$$

To use these edge detection filters one apply them during convolutions over the image for multiple iterations to get the resulting edges from the image.

2.8 Halo exchange

For iterative algorithms where the next iterations calculation depends on multiple values from the previous iteration, like a stencil-based edge detection algorithm, halo exchange is often used when parallelizing the algorithm. In halo exchange, the halo is defined as the surrounding rows and columns outside the current chunk, storing data from other chunks making this data available during computation. The exchange of these values often happens between each iteration where all processes synchronize to share the outermost layer(s) of the current processing chunk.

Kjolstad and Snir names this pattern of communication "Ghost cell pattern" where data is exchanged between neighboring chunks. [2] The idea of having ghost cells surrounding the

chunk storing values from the neighbor chunks to improve performance, as an implementation asking for the boundary values as they are needed in the computation results in huge accumulated latency from the message passing required.

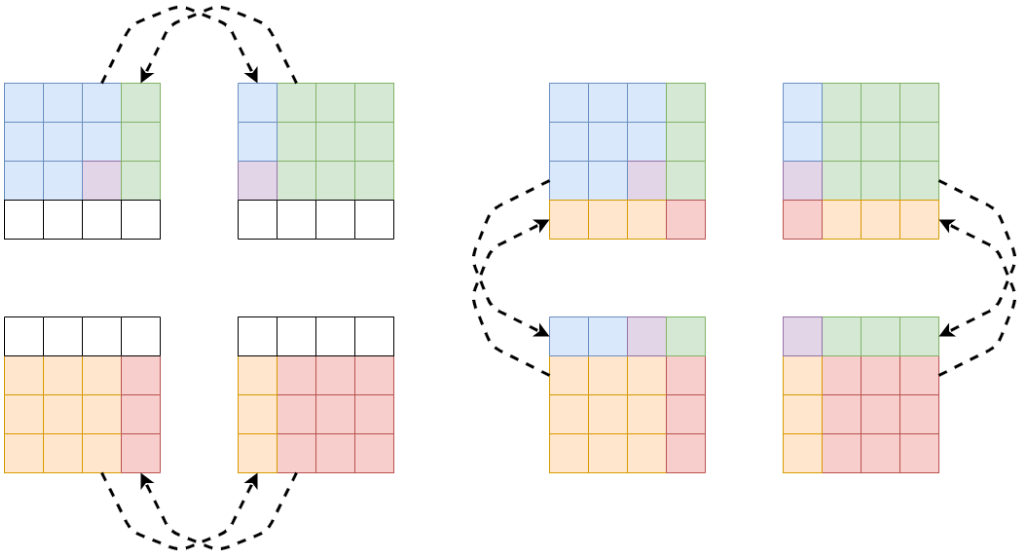


Figure 3: Copy diagonal values through ghost cell halo exchange. (Based on figure 9 [2])

Another problem that ghost cells and the proposed communication pattern are when dividing the domain into multiple dimensions (e.g., two-dimension, three-dimension). The resulting chunk to chunk communication when the computation requires diagonal data increases the complexity and number of communications. However, when synchronizing the ghost cells, they proposed extending the data being synchronized to include the ghost cells at the ends of the row or column. Inclusion of the extra ghost cells is because when synchronizing the values first horizontally then vertically, the values to be copied diagonally have already been copied in the usual synchronization, and no extra diagonal specific communication is necessary as shown in figure 3.

2.8.1 Deep halo

Kjolstad and Snir extended upon this idea of ghost cells surrounding the processing chunks and talked about extending the halo size beyond the first row or column. This approach would be required if using larger stencils or implementing algorithms using values beyond the closest neighbors.

Extending the halo size can also be beneficial in systems where the total delay of synchronization is high. The extended halos provide extra layers of ghost cells the algorithm can use to compute the next iteration of ghost cells locally and perform multiple iterations before needing to synchronize the values. As shown in figure 4 a 3x3 grid with two layers of halo data allocated and after a halo exchange, the cells are populated, and the next iteration computed. After this compute is finished, the second layer is still valid due to the use of a 3x3 stencil as it is calculated using the outer most layers data. This makes it possible to calculate another iteration before all halo data is invalid, and another halo exchange is required.

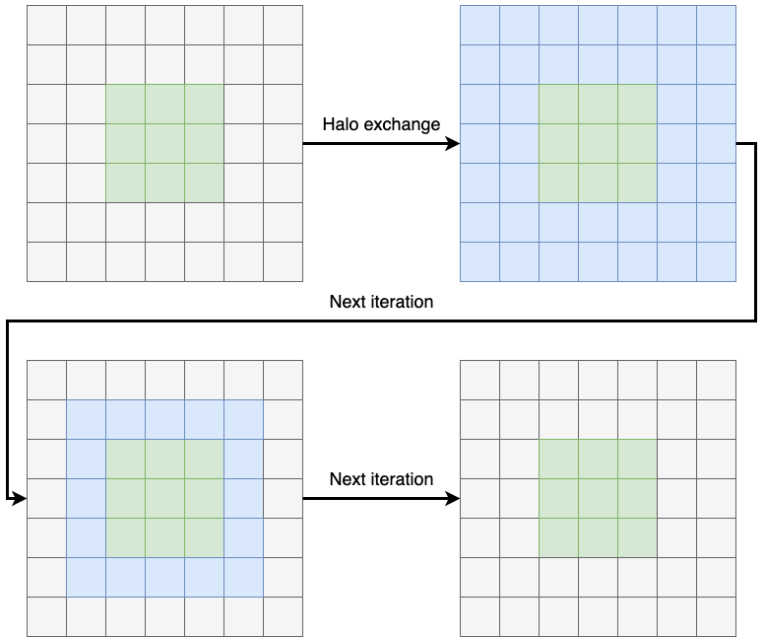


Figure 4: Use of deep halos to exchange computation for communication. Green cells represent chunk data, blue cells represent still valid halo data and grey cells represent invalid halo data.

3 Implementation

In the process of parallelizing a two-dimensional edge detection algorithm with halo exchange, a prototype or proof of concept was implemented. This only divided the image data into a one-dimensional chunk array where the halo exchange direction never changed (i.e., always north-south or east-west exchanges). The reason to build a prototype illustrating the final result is to both have a working implementation to compare the result from the final implementation to for correctness. To figure out in a more simple environment how the common building blocks of the application should fit together before introducing more complex functionality to avoid shuffling bugs in the upgraded parts of the application with simpler common-ground sections of code.

The prototype one-dimensional halo exchange application paved the way for a two-dimensional implementation, but some considerations and improvements had to be made to facilitate the two-dimensional approach. When handling two-dimensional domain decomposition as opposed to the prototype's one-dimensional decomposition, the chunk distribution and size calculations and data deconstruction and reconstruction have to be carefully modified to distribute the domain throughout the processes available correctly.

```
void trial_division(
    int world_size,
    int *width,
    int *height
) {
    for (b = 1; b < world_size; b++) {
        if (world_size % b == 0) {
            *width = world_size / b;
            *height = b;
            break;
        }
    }
}
```

Listing 3.1: Trial division integer factorization.

Chunk distribution was previously, with the one-dimensional approach, structured as a $1 * n$ where n is the number of processes available. To transform this into a variable two-dimensional grid $w * h$ without predefined grid dimensions, we chose to apply a simple integer factorization algorithm to calculate the different integer pairs whose product is equal to the world size as shown in listing 3.1. This trial division will always return $1 * world\_size$, which is correct, but not an equal distribution of chunks in both dimensions. For prime numbers, this distribution of $1 * world\_size$ will be correct, but for other numbers, a width and height value as close to each other as possible is the goal.

```
void trial_division_closest_pair(
    int world_size,
    int *width,
    int *height
) {
    int min_score = 2 * world_size;
    for (b = 1; b < world_size; b++) {
        if (world_size % b != 0) continue;
        int a = world_size / b;
        if (abs(a + b) < min_score) {
            *width = world_size / b;
            *height = b;
            min_score = abs(a + b);
        }
    }
}
```

Listing 3.2: Trial division integer factorization with closest pair search.

To achieve the best distribution of two factors, the trial division shown in listing 3.1 need to be modified to find not only a solution but the most equally distributed solution. The goal is to get the difference between the width and height as low as possible for the valid factorized pairs. The solution was inspired by the closest pair of points brute-force algorithm [5] comparing the next pair from the trial division to the previously closest pair of factors selecting the closest factors in the end as shown in listing 3.2. The resulting width and height are used to describe the different world ranks position in the global domain, with each rank moving through the width and then wrapping to the first column and next line as shown in figure 5.

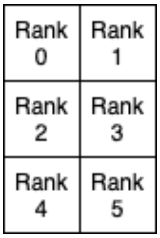


Figure 5: Rank position in the decomposed problem domain with a chunk distribution of $2 * 3$.

Size calculations for each chunk are taking the $floor(image\_width/chunk\_distribution\_width)$, and for each chunk at the end of a row, the potential extra columns of data are appended, giving the end chunk possibly a wider domain. The same approach is used for the chunk height, and for the bottom row of chunks, any extra rows are appended to their domain.

Deconstruction and reconstruction of the image data are handled in the same conceptual way, but reconstruction is performed in reverse from deconstruction. The deconstruction iterates over each pixel calculating which chunk the pixels position is contained within, then appending the pixel data to the chunks respective array segment to be sent to the correspond-

ing process. Reconstruction takes all the different chunks' resulting pixel data. Through the same iteration, it calculates the corresponding chunk and position within the chunk pixel array to the position in the global pixel domain using the size of the chunks as guidance to reconstruct the final image after the computations.

3.1 Setup

After setting up the MPI environment and fetching the rank and world size, the various parameters providable include the number of iterations to calculate, halo size, the input and output image to use, and an optional width and height parameter. Loading the data on which to compute is done in two different approaches. If the width and height parameters are both supplied and non-zero, these are used to generate an image of the specified size with random RGB data for ease of testing different sizes of image data. If the width and height are not supplied or has a zero value, the specified input image BMP file will be loaded from disk and used in the following computations.

As mentioned above, the chunk distribution is calculated using a basic integer factorization algorithm with a validation on the optimal equal distribution as listing 3.2 shows. The entailing chunk sizes calculated from the data width and height are used to prepare the data scattering to the system's various ranks. Firstly calculating the number of pixels each chunk represents and calculating the displacements in the global pixel domain for the data. These values were applied in the scattering along with the data array containing the pixel data extrapolated and flattened from the two-dimensional image data array into a one-dimensional array sequentially containing each chunk's pixel data with the displacements specifying at what point the different chunks' data starts. This data is after the scattering pieced together in a smaller image structure adhering to the chunks domain's size adding room for the specified halo size.

3.2 Computations

The edge detection is done through a series of iterations where the previous iteration is the input data to the next. Each chunk iterates over its pixel subdomain calculating the resulting values based on a selected kernel/stencil matrix, a laplacian stencil as shown in figure **??**. Every iteration computes the chunks subdomain's values and the halo data stored in each chunk. When using deep halos, it provides multiple iterations worth of data stored in each chunk after a halo exchange. This data needs to be calculated for the next iterations and extending the calculations beyond the chunks specific subdomain into the halo data.

3.2.1 Border exchange

To ensure a correct result in a neighbour dependent algorithm as the implemented edge detection algorithm, data needs to be exchanged between the different processes. This can be done in multiple ways, in this implementation we used a simple synchronous and asynchronous approach, differing how many blocking calls are used, and how the messages propagate throughout the chunks. Furthermore, the use of deep halos to exchange communication for computation is handled with the halo depth controlling before which iterations the halo exchange should take place.

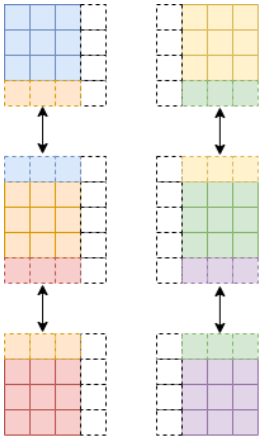


Figure 6: First step of halo exchange in the north-south direction, with halos shown as dotted blocks.

Synchronous halo exchange

The halo exchange takes place in two directions, including the halo data in the exchange. This is to ensure the diagonal values are transferred correctly without any specific logic for the diagonal values. The exchange first takes place in a north-south direction, as shown in figure 7. With the layout matching that of figure 5, halo exchange occurs between ranks 0 and 2, ranks 2 and 4, and ranks 1 and 3, and ranks 3 and 5. These are in a sense independent, but with the synchronous approach, firstly all chunks having a chunk above itself, will exchange with that chunk using the blocking MPI_Sendrecv call, making so that exchanges between ranks 2 and 4 are dependent on the completion of the exchange between ranks 0 and 2. This halo exchange dependence will then propagate throughout the entire chunk space, making the last row of chunks wait for all previous rows to complete before continuing execution. After a chunk has finished all north-south exchange, it progresses to the east-west exchange. Even though rank 0 can start its east-west exchange before the other chunks are finished with the north-south exchange, it will still stay consistent as no process will progress into the next exchange direction before all data sent and received in the previous exchange. As the entire process uses blocking message passing calls, it stays consistent as all data is done in one direction before continuing in the next direction.

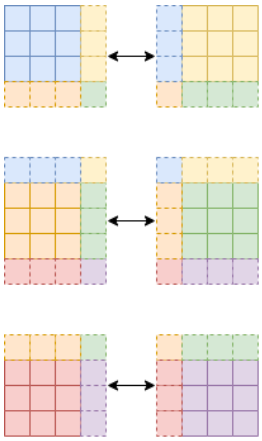


Figure 7: Final step of halo exchange in the east-west direction, with halos shown as dotted blocks.

Asynchronous halo exchange

Asynchronous halo exchange is implemented similarly as the synchronous, but using some other message passing calls reduces the number of synchronization points. This is done using the MPI_Isend and MPI_Irecv calls and waiting for all calls for the given process to finish before continuing execution. Instead of having each call block execution as the synchronous version does, all calls are either sent or set up to receive before having one blocking call to await all calls at once. After this blocking call, the received data is placed into the correct halo for the chunk. Using asynchronous send and receive calls utilizes the available bandwidth on the network more efficiently, as the message in this program both in size and amount never will cause any network congestion. This plays a role in the total theoretical delay for the halo exchange. Each direction delay only depends on the slowest communication, with the synchronous approach and the message delay propagation will accumulate the total communication delay for each direction.

3.3 Gathering of data

The gathering process uses the same previously calculated chunk sizes and displacements within the one-dimensional data stream combined through the gathering call. Each chunk strips all halo data from the data being sent back to the main process, as this data is mostly outdated and more importantly another chunk's subdomain. The main process then using the chunk distribution and sizes, puts all the individual chunk data back into place in the result image structure before this is saved to disk.

4 Results and discussion

4.1 Experimental setup, timing and error sources

The experiments were run on four machines running in a slurm cluster connected with a 100Mbit Ethernet interconnection. All machines were located within the same subnet making the packages routed through the same router, and only that one router. The four machines all consist of the same hardware as shown in table 1 and reading data from local storage, and all metric output was written to local storage on machine 0. The metrics were printed from the process after all computation was finished to not interfere with the computations and timing. The run environment was done with exclusive access to all four nodes, running Ubuntu 10.04.5 LTS with OpenMPI version 2.1.1 installed.

The experiments conducted tested asynchronous non-blocking messages' performance with different halo sizes and synchronous blocking messages versus asynchronous non-blocking messages' performance with the same halo size variations. All experiments were run 10 times for each configuration, selecting each run's median value as the representative value for the run. The timing was done at two different places in the program. One timing of the entire computation process from just before the scattering of data all the way to the data was gathered after the computations. The other timing was done on each halo exchange individually stored in memory and gathered after the computations were finished to not interfere with the computations. MPI_Wtime was in the end used to time the different parts of the experiment. Each process timed every halo exchange and stored each time delta seperatly during the computations. This array of time data was gathered after the computations and each process's total time and separate iterations times was printed.

Each node executed 2 MPI processes each so as not to strain the cache excessively, reducing performance with it able to store more instructions and data reducing potential cache misses. The data computed on each run of the program was randomly generated prior to the scattering for each run, with a supplied image size of 10k by 10k pixels and computed 5000 iterations in total. Random data was used to get a suppliable image size and test easily with different sizes. As the computations, regardless of the data performed the same operations, the fact that the data was random would not affect the performance of the algorithm.

When performing experiments, the consideration of error sources is important, and in this

CPU	Intel i7-7700k @ 4.20GHz 4C/8T Min. clock 800MHz
Memory	32GB DDR4
Interconnect	100Mbit Ethernet

Table 1: Hardware specification for the clusters machines.

experiment there are some possible sources of errors/incorrect timing that could occur. One could argue that some combinations of numbers in multiplication/addition could be calculated faster by the CPU than others, but this is not investigated and only mentioned as a small possible error source. Furthermore, the cluster used for the computations could not have clock-locked CPUs, and could vary their CPU clock between 800MHz and the rated 4.2GHz. This could affect the performance as there might occur some waiting before the other process is ready to synchronize with each iteration. This could make the CPU slow down the clock and introduce a slightly lower performance in the instant it starts after the wait, and with many iterations this could accumulate to a possible noticeable performance reduction. Lastly, the ethernet interconnect between the nodes uses the internal network in the lab, but is not exclusive to interconnect packages between the nodes, and if some other nodes/computers used the network while running the experiments, this could affect the message passing latency.

4.2 Asynchonous deep halos

The first experiment was looking into the exchange of communication for synchronisation to speed up the algorithm's solution time. Figure 8 presents the median difference in total running time with 0 being the fastest solution, which turned out to be with a halo size of 2, meaning every other iteration a halo exchange would occur. The actual measured median running time of the algorithm with a halo size of 2 was 1306369.801 seconds.

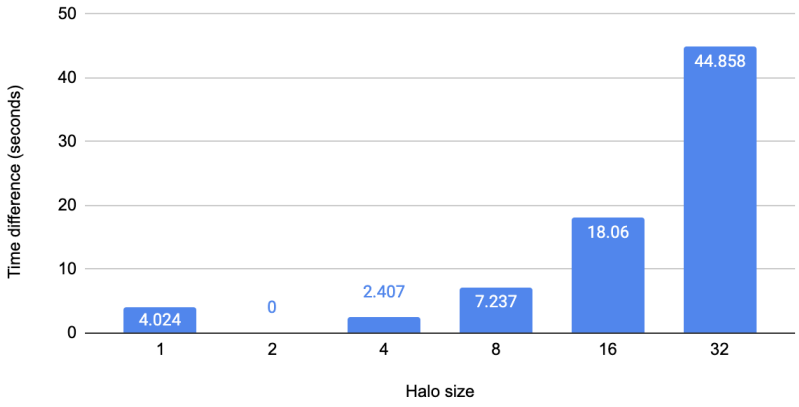


Figure 8: Median time difference for asynchronous halo exchange with deep halos.

When analyzing the results, we can see that a halo size of four is the next best with halo size one following and any higher than four looks to be inefficient. In this implementation, each point in the halo exchange consisted of 3 bytes of color data, and for an algorithm requiring a larger amount of data transferred with necessarily an unequal increase in computation, a larger halo size would be preferable as the halo exchange needed to exchange more data. As this implementation for every exchange calculates the entire halo, also the parts of the halo that are not to be used, an increased halo size above eight would probably continue the same trend as seen from halo size 8 to 32 in the chart. If one were to limit the number of halos calculated each iteration, going from 32 halos in the first iteration after an exchange all the way not to calculate any halo data on the last iteration before an exchange would decrease

Halo size	Halo pixel count
1	15008
2	32016
4	64128
8	120512
16	242048
32	488192

Table 2: For the 10k by 10k image processed, the amount of deep halo pixels needed to be processed given the different halo sizes.

the amount of computation and possibly provide a better running time for the larger halo sizes.

The basis of these results is when performing the edge detection on a 10k by 10k image and each chunk governs a 5k by 2.5k sub image. The total size of the all surrounding halo data is as shown in table 2. The comparatively small differences in the total time for halo sizes eight and smaller a smaller subdomain resulting from a smaller image or more processes could affect what halo size performs the best. This is due to the total number of halo pixels would drastically be reduced, and given the latency of message passing at some point probably catching up with the time used for the extra computation, it could result in better performance for larger halo sizes.

4.3 Asynchronous vs. synchronous message passing

MPI supplies multiple ways to perform message passing, and we have looked into both synchronous and asynchronous message passing calls to measure if there is a difference when using either of them for this edge detection algorithm. The difference being that the synchronous calls block the execution and multiple such calls are needed for the border exchange. On the other hand, asynchronous calls execute them awaiting the completion of these only after all are started. This makes the order of completion the optimal order, with them finishing any call as soon as possible without the need to wait for the order it was performed. This removes the accumulative delay with blocking calls, and only the slowest delay is the maximum delay for the entire exchange.

As mentioned, the accumulated delay for an exchange is theoretically better with the non-blocking approach. Figure 9 shows the median time difference between the different tested halo sizes and the use of blocking and non-blocking message passing. With a halo size of one, the blocking approach performed 79ms better than the non-blocking approach. As this chart shows, the difference in total running time for the different schemes and this total varies from 1302.658 seconds to 1347.517 seconds for the different halo sizes, and improvement of 3-4 seconds is minuscule at best. These variations can easily be due to randomness in using a semi-public network for the interconnect or the processors not locking their clock. This would suggest that the implementation of the MPI standard used has optimized the communication calls to such an extent, at least for relatively small messages, that the advantage of using non-blocking calls without any other optimizations is minimal to none. The results presented for the increasing halo size might have more variations between each run as the amount of computation each process has to do increases, and with the amount of cache misses poten-

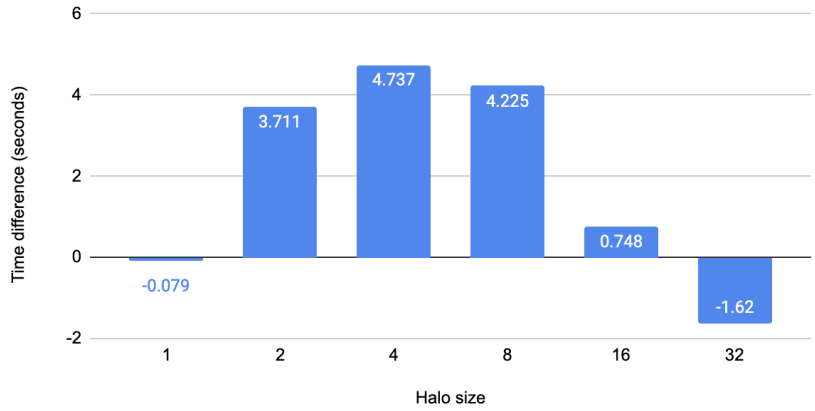


Figure 9: Median time difference for edge detection using synchronous vs. asynchronous halo exchange.

tially increased between each iteration, it might introduce more accumulative waiting for the bordering processes to finish execution.

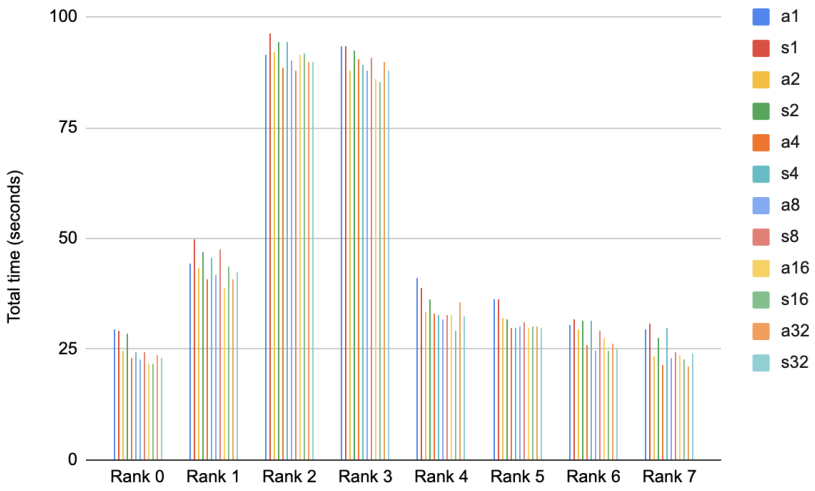


Figure 10: Median total time for halo exchange at both asynchrnonous and synchronous exchange patterns. Series prefix *a* is asynchronous, prefix *s* is synchronous and numbering is the halo size.

Since OpenMPI distributes the same ranks equally for every run on the tested execution environment, comparing each rank's total time spent in halo exchange with the different halo sizes both for blocking and non-blocking messages is possible and shows how the halo exchange time evolves with different execution parameters. Figure 10 shows for the eight ranks used the median total time spent in halo exchange. This is calculated by measuring and summing the total time spent on each halo exchange. The chart shows that the difference is small for the different halo sizes and communication scheme, but with the larger halos, one can see a consistent reduction in halo exchange, but not proportional to the data transmitted or halo size. This is expected as the total number of exchanges is greatly reduced and the total delay between the different processes is reduced. Due to the fact, more data is transmitted by each exchange the time consumed is not drastically improved. The difference between the

blocking and non-blocking for the different ranks and halo sizes is not consistent in closer examination. In som ranks like rank one, non-blocking outperforms blocking calls, but when examining other ranks, this is reversed or no notable difference supporting the finds from figure 9 with minimal to no difference between the different schemes.

4.4 Optimizations

The running time could be optimized in a variety of ways. For the total running time, the use of a hybrid programming model would be beneficial. This entails that each individual node has one message passing process handling the node-to-node communication. Each node is divided its chunk of the global problem domain, and each subdomain is processed on the different nodes. However, the difference from this paper's approach is that internally on each node, the use of shared memory parallel programming to utilize all resources without the message passing overhead and potentially more worker threads on the same CPU. This experiment used half of the available cores, as each process has its memory and to avoid issues like cache misses the four cores available were not used, but with a shared memory model one could use the entire four cores, or even the eight threads available on the CPU.

Another optimization is hiding the message passing delay by using inner computation while the exchange is taking place. This entails computing the outer values right after the exchange is finished, then exchanging the already computed borders and computing all the chunk's inner values not dependent on any halo data or any part of outgoing halo data as well. This would overlap the time used for the inner computations with the delay of performing asynchronous message passing. One could not perform this using synchronous blocking message passing since the blocking aspect would not allow the computations to take place parallel to the communication and would only be an option for the asynchronous non-blocking approach.

5 Conclusion

Halo exchange is used when parallelizing workloads where calculations require data from other problem subdomains like stencil-based algorithms. This paper looked into two different approaches to deal with halo exchange with a message passing programming model. First, we investigated the use of deep halos to exchange communication for computation on the implemented stencil-based edge detection. This resulted in the best performance achieved with a halo size of 2 on a 10k by 10k image. With the more naive implementation tested, not optimizing the halo data compute and the few processes used in testing coupled with the relatively low latency message passing deep halos will not provide a huge performance gain. Looking into the performance difference when using blocking or non-blocking message passing, we found that the total time difference was so small and with a halo size of 1 where the most communications would happen, the potential gain from asynchronous did not show. This results in the find that non-blocking message passing used with the same communication pattern as a blocking message passing results in no performance increase. Still, non-blocking message passing provides the flexibility to hide parts of the latency by combining the exchange with inner computation, making the potential performance increase present, but not tested in this work.

5.1 Future Work

To improve the results found in this paper, the use of inner execution to mask the halo exchange delay would be beneficial. Instead of computing the halos, border values and all other chunk data and only between the iterations perform the halo exchange. One could optimize this by computing the border values first, and during the computation of the inner chunk data perform asynchronous halo exchange to use the wait during the exchange to compute other data and see if this would outperform a synchronous message passing approach.

Another optimization that would be interesting to look into is the speedup potential when dealing with deep halos. Instead of computing the entire halo data each iteration, one could use the knowledge that for each iteration, another layer of halo data is rendered obsolete and skip computing that part of the halo to save on compute time. Also, extending this idea into smaller subdomains with either a smaller global problem domain or more processes to see if the halo data compute is significantly reduced to provide a larger halo size better total time results.

Often when parallelizing on multi-core nodes in a larger cluster, a hybrid programming model is usually used as this eliminates the overhead of message passing between cores on the same node by using a shared memory model within the node. Taking this approach and testing how much speedup and at what halo size this approach would make sense would be interesting to see as an all message passing based approach might introduce a large accumulated overhead and with the shared memory approach, a larger number of cores can process the halo data and utilize the data better when using deep halos.

In a paper by Holems, Laoide-Kemp and Parsons [12] they use a stream based approach instead of halos to process data with minimal message passing delay on one-dimensional circular data. The processes' subdomain revolves over the data as the iterations continue, and given the right buffer sizes and values combination when sending the buffer data to the neighboring process, the compute could continue with very little delay given that the different processes are continuing in an equal pace and not lagging behind. Taking this approach and looking into the possibility of using this in both non-circular data as an image and extending it to multi-dimensional data could improve the performance and memory overhead on each process.

Bibliography

[1] G. William, L. Ewing, and S. Anthony, *Using MPI : Portable Parallel Programming with the Message-Passing Interface.*, ser. Scientific and Engineering Computation. The MIT Press, 2014, vol. Third edition. [Online]. Available: http://search.ebscohost.com/login.aspx?direct=true&db=e000xww&AN=906701&site=ehost-live

[2] F. B. Kjolstad and M. Snir, "Ghost Cell Pattern," in *Proceedings of the 2010 Workshop on Parallel Programming Patterns - ParaPLoP '10*. Carefree, Arizona: ACM Press, 2010, pp. 1–9. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1953611.1953615

[3] "Integer factorization," Nov. 2020, page Version ID: 989285688. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Integer_factorization&oldid=989285688

[4] W. A. Borders, A. Z. Pervaiz, S. Fukami, K. Y. Camsari, H. Ohno, and S. Datta, "Integer factorization using stochastic magnetic tunnel junctions," *Nature*, vol. 573, no. 7774, pp. 390–393, Sep. 2019. [Online]. Available: http://www.nature.com/articles/s41586-019-1557-9

[5] "Closest pair of points problem," Jul. 2020, page Version ID: 966956388. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Closest_pair_of_points_problem&oldid=966956388

[6] T. Mattson and M. Murphy, "Geometric Decomposition | Our Pattern Language," Nov. 2020. [Online]. Available: https://patterns.eecs.berkeley.edu/?page_id=213

[7] C. Kessler and J. Keller, "Models for parallel computing: Review and perspectives," *Mitteilungen-Gesellschaft für Informatik eV, Parallel-Algorithmen und Rechnerstrukturen*, vol. 24, pp. 13–29, 2007.

[8] O. A. R. Board, "OpenMP Application Programming Interface," Nov. 2020. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf

[9] "MPI: A Message-Passing Interface Standard," Tech. Rep. [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[10] P. A. Mlsna and J. J. Rodríguez, "Gradient and laplacian edge detection," in *Handbook of Image and Video Processing (Second Edition)*, second edition ed., ser. Communications, Networking and Multimedia, A. BOVIK, Ed. Burlington: Academic Press, 2005, pp. 535 – 553. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780121197926500954

[11] M. Sharifi, M. Fathy, and M. T. Mahmoudi, "A classified and comparative study of edge detection algorithms," in *Proceedings. International Conference on Information Technology: Coding and Computing*, Apr. 2002, pp. 117–120.

[12] D. J. Holmes, C. Laoide-Kemp, and M. Parsons, “Streams as an alternative to halo exchange,” in *Parallel Computing: On the Road to Exascale*. IOS Press, Apr. 2016, pp. 305–316, google-Books-ID: PKotDAAAQBAJ.

Andreas Hammer

Analyzing Halo Computations on Multicore CPUs

NTNU
Norwegian University of
Science and Technology