SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Automatic Code Generation for MPI Communication of the CROCO Ocean Model

Anna Mittermair

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Automatic Code Generation for MPI Communication of the CROCO Ocean Model

Automatische Codegenerierung für MPI-Kommunikation des CROCO-Ozeanmodells

Author:	Anna Mittermair
Supervisor:	Prof. Dr. rer. nat. Martin Schulz
Advisor:	Prof. Dr. Martin Schreiber
Submission Date:	30.09.2023

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 30.09.2023 Anna Mittermair

Abstract

Efficiently managing communication in High-Performance Computing (HPC) with distributed memory systems is a significant challenge.

This thesis describes an approach to automatically generate code that implements MPI communication for the CROCO ocean model. It creates a prototype that analyzes data dependencies to determine when communication is necessary and implements the required halo exchanges using asynchronous MPI operations. Evaluation shows that the prototype successfully identifies and creates all necessary communication operations in a test case. Additionally, it successfully reduces the number of necessary halo exchanges by approximately 50%. Asynchronous communication did, however, not help to improve performance as hoped.

All in all, the presented approach is able to automatically generate MPI communication for CROCO, which can be used to facilitate code development or potentially optimize communication.

Contents

1 Introduction

High-performance computing (HPC) refers to the use of specific computing techniques and technologies to solve problems requiring exceptionally high processing speeds and large data sets. HPC uses powerful computing systems, often consisting of clusters of processors, to perform computations and simulations at high speeds. Because of that, it plays a key role in solving complex problems that require extensive computational resources. Problems like that are found in many domains, including but not limited to: search engines, oil reservoir modeling, cosmology, medicine development, fraud detection and epidemiology [1]. Current progress in artificial intelligence is also based on HPC techniques as it is driven by very high amounts of computational power.

HPC also plays a crucial role in weather forecasting and climate modeling. Weather forecasting models rely heavily on HPC to accurately simulate and forecast atmospheric conditions within strict timelines. Similarly, climate models operate like weather forecasts but aim to make predictions over extended timeframes, making execution times even more important. Only with short computation times it is possible to forecast weather quickly and at the same time accurately and with high resolution or to simulate climate conditions over long periods of time [2].

To deal with very high computational demands like this, it is commonly necessary to use hardware consisting of multiple cores, processors, or nodes. To use many cores to work on the same problem, parallel computing is used, dividing tasks into smaller, parallelizable components that can be processed simultaneously on multiple processors or cores. This has become even more critical as traditional sequential computing, which has historically relied on increasing clock frequencies to enhance performance, reaches its limits as clock speeds can not easily be increased anymore due to escalating power consumption and heat dissipation. The result of this is an increased focus on parallelism and multi-core processors, enabling advances in computing performance without significant increases in power and heat [3].

Parallel computing has emerged as a cornerstone of HPC because it provides a scalable approach to solving computationally intensive problems. In parallel computing, the workload is distributed across multiple processors, each of which handles a portion of the task. This allows HPC systems to perform computations at much higher speeds, as multiple processors operate concurrently to solve complex problems in a fraction of the time it would take a single processor. It contrasts with traditional sequential computing, where tasks are processed one at a time, and performance improvements are limited by a single processor's clock speed and efficiency. Parallel computing is especially relevant in the context of modern HPC clusters, which often consist of clusters of processors or nodes interconnected by high-speed communication networks.

A variety of system architectures are used in parallel computing in order to make simultaneous processing possible. Multi-core and multi-processor computers have multiple processing elements in a single machine. Meanwhile, distributed systems like clusters and grids consist of several computers, each with multiple processors and cores, connected through a communication network.

One important characteristic of distributed systems is the concept of distributed memory. With distributed memory, each node has its own private memory, and these memory spaces are not directly shared between nodes. Instead, explicit data exchanges or communication between nodes are necessary whenever a task needs to access data that is not in its local memory. This distributed memory model contrasts with shared memory architectures, where several processors or cores access a single shared memory space.

Distributed memory architectures work best with scenarios that involve limited data sharing and a lot of computation with local data. Communication between nodes is essential for coordination and synchronization in distributed memory systems, especially in large-scale distributed computing environments, such as HPC clusters and grids. Efficient communication among nodes is made possible through message-passing techniques, such as the Message Passing Interface (MPI), which provides a standardized set of functions and libraries for transmitting data and coordinating processes across distributed memory systems.

1.1 Challenges for HPC

As computational demands continue to grow, ensuring that HPC systems can scale effectively to handle increasingly complex problems becomes an even bigger challenge. A persistent problem in HPC is the issue of memory bottlenecks. While computational speeds have advanced significantly over time, memory access times have not decreased as quickly. This difference in performance between processors and memory can lead to inefficiencies, as processors often spend a significant portion of their time waiting for data to be fetched from memory.

Similarly, communication between processors or nodes within an HPC system can introduce waiting times, leading to communication latency. Processors need to exchange data to facilitate coordination and synchronization. The time spent on communication, however, can easily become a limiting factor in overall system performance, especially as the number of nodes working together grows. The increasing heterogeneity of HPC architectures, including combinations of CPUs, GPUs, and specialized accelerators, presents additional challenges. Writing and optimizing code to fully use the potential of diverse hardware components while maintaining portability is difficult and ensuring that code can effectively use parallelism, communicate, manage memory efficiently, and adapt to different architectures is a significant challenge. This is especially true for domain experts that extensive knowledge in their respective fields but lack expertise in HPC. At the same time, HPC experts may not fully understand specific domains. As a result, creating HPC code that performs optimally for both specific domain problems and diverse hardware becomes even more challenging. Bridging this gap between domain and HPC experts is an another challenge in HPC.

The primary concern adressed in this thesis is the the problem of the complexity of developing efficient code that implements communication in distributed systems.

1.2 Approaches to Communication in HPC

In HPC, often many nodes or processors must work together, and thus, communication between processors or nodes is common. Reducing the time spent waiting for data transfers is therefore crucial. This is especially true for distributed systems with longer latencies.

Writing code that efficiently communicates in distributed systems is is often done with the Message Passing Interface (MPI) as a library. Often, non-blocking communication is used in order to enable overlapping computation and communication to hide communication latency. Although MPI is well-suited for this as it offers fine-grained control over communication, it can be very complex to use. An alternative to MPI offer Partitioned Global Address Space (PGAS) models [4].

PGAS is a programming model combining features of shared memory and distributed memory models. In PGAS, the global address space is logically partitioned, with each partition linked to a specific processing element or thread. PGAS models include Coarray Fortran, Unified Parallel C [5], and similar frameworks.

1.2.1 Coarray Fortran

Coarray Fortran (CAF) [6] is a parallel programming extension of Fortran designed to simplify the development of parallel and distributed applications. CAF offers an array-based parallelism model, where data structures known as coarrays can be accessed concurrently by different program units called images.

While CAF allows for parallel execution and communication between these images, it does not offer native asynchronous communication capabilities. In CAF, communication typically occurs through explicit synchronization points, such as "sync all" or "sync images," where data exchange between images is synchronized. To achieve asynchronous communication in a CAF program, programmers need to implement their own communication patterns using features provided by Fortran or by integrating CAF with other libraries or tools supporting asynchronous communication (such as MPI).

In response to these limitations, CAF 2.0 was proposed in 2009 [7]. It aimed to introduce a model for asynchronous operations like asynchronous copies and provide the potential for latency hiding [8]. However, its adoption and implementation have been very limited.

1.2.2 Compiler-Based Approaches

Another set of approaches to automatically generate or optimize code to facilitate developing efficient communication implementations involves using compilers and compiler analysis techniques to optimize communication/computation overlap in MPI code. The general idea of this approach is to give the compiler knowledge about what MPI calls do so it can use that information to optimize and create non-blocking communication [9]. This approach is also explored in [10] and [11], which investigate various communication strategies and automation methods.

One notable example, [12] uses a compiler analysis technique known as the polyhedral technique to automatically detect opportunities for communication/computation overlap. This technique analyzes exact dependencies over multiple loop iterations, increasing the overlap window between generating and using the data. It's important to note, however, that these techniques are not universally applicable and require specific conditions to be effective. While [13] implemented a more automatic and realistic analysis approach for scientific computing applications, it still relies on developer intervention to operate effectively within complex scientific codes. Like several of the mentioned approaches, it even needs manual implementation of automatically created optimization suggestions.

Developer guidance is required only to improve the accuracy of the analysis for large scientific applications, because not all source code of these applications is available and many low-level implementation details are impossible to fully automatically decipher [17]. We currently manually apply the necessary program transformations, because code need to be carefully moved across procedural boundaries

1.2.3 Dataflow Programming Models

Dataflow programming models are also commonly used to automatically optimize communication to enable latency hiding in HPC. These models involve either programming code directly in a dataflow model or translating existing code into such a model to analyze and optimize it by overlapping communication and computation.

For instance, Bamboo [14] [15] in combination with Tarragon [16] automatically translates source code that uses MPI into a version that overlaps computation and communication to hide communication latency. This scheduling is done by a runtime system based on a task-dependency graph abstraction. This approach does, however, require programmer annotations to identify regions suitable for overlapping and efficient communication reordering.

The same is true for the combination of Toucan [17] and Mate [18], a similar code translation system with a model and runtime system for latency hiding. It is able to use overdecomposition (using multiple MPI processes per processor core) without too much additional overhead and uses this to hide latency by scheduling another process to do computation on the same core while waiting for communication. It also requires adding more annotations to indicate dependencies and independent regions.

1.3 Goal of This Thesis

This thesis focuses on developing, implementing, and evaluating an automated approach for generating code that implements the MPI communication used by the *Coastal and Regional Ocean COmmunity* (CROCO) ocean model [19].

Ocean models are part of weather and climate modeling systems and have a high need for computational performance and efficiency, leading to them commonly being executed on big computing clusters. This causes a need for communication between different nodes.

CROCO is an HPC application that is written in Fortran and can be executed in a parallelized version that uses synchronous MPI messages to exchange data between threads. These communication operations are implemented with manually written code. The code is then assembled by the pre-processor into different code versions for different use cases dependent on numerical scheme, modeled situation, hardware, and other arguments. This makes writing and optimizing code in this context relatively difficult. Additionally to the general complexity of writing efficient MPI code for such a model, any communication code has to work in many different scenarios.

This thesis develops, presents, and evaluates a prototype that can automatically analyze CROCO code that is parallelized using MPI and implement necessary communication as asynchronous MPI communication. This is done in two stages. The first stage analyzes the code and its data dependencies to determine when communication is necessary and identify a time window for asynchronous communication. The second stage automatically generates code implementing this communication scheme with non-blocking MPI operations.

This should happen as automatically as possible, without needing additional annotations and with minimal additional user input.

While this thesis works on using CROCO's BASIN test case as a proof of concept, the approach needs to be general to work with different CROCO code for different use cases.

1.4 Outline

This thesis is structured in the following way: Chapter 2 gives an overview of CROCO, how it works, and how its current mode of communication looks. Chapter 3 presents the code generation system PSyclone, which was used as one of the main tools for this project, and Chapter 4 presents the general setup. Chapter 5 explains how necessary halo exchanges are determined and Chapter 6 how the code is generated automatically. Chapter 7 describes the evaluation process with 8 giving an overview of the results.

2 CROCO

The *Coastal and Regional Ocean COmmunity* (CROCO) ocean model is an ocean modeling system. As such it uses mathematical models to describe and simulate physical processes in the ocean while placing a special focus on resolving very fine scales such as in coastal areas and their interactions with larger-scale oceanic processes [19]. As outlined in its documentation [19], CROCO describes ocean behavior within a complex, coupled system of various interconnected elements such as the atmosphere, surface waves, marine sediments, biogeochemistry and ecosystems.

The following section provides an overview of CROCO's functionality and its relevance within the context of the challenges discussed in Chapter 1.

2.1 Numerics

2.1.1 Primitive Equations

CROCO is able to simulate various properties and factors present in an ocean environment. These variables include momentum, temperature, velocity, salinity and their corresponding temporal tendencies.

To describe their behavior, computational fluid dynamics rely on the Navier-Stokes equations, a highly complex set of equations, to model fluid behavior. For the sake of simplification, a more manageable set of nonlinear partial differential equations, referred to as the Primitive Equations, is used to provide an accurate approximation of ocean behavior at large scale. These equations are simplifications of the Navier-Stokes equations with the addition of a nonlinear equation of state. This equation describes how the two so-called active tracers, temperature and salinity, affect water density, impacting buoyancy and circulation in the ocean.

In this context, tracers are scalar variables tracking the movement and dispersion of substances or properties within the water. There are two categories of tracers: active and passive. Passive tracers observe the transport and mixing of specific properties such as sediment concentrations without influencing fluid flow. Active tracers, like temperature and salinity, actively affect fluid dynamics within the ocean.

The primitive equations are based on assumptions that are valid on large scales. These assumptions cover aspects of fluid dynamics, such as the incompressibility hypothesis, and take into account the shape of the Earth and oceans, such as the spherical earth Approximation. By default, CROCO employs the primitive equations for simulations, but it offers flexibility through configurable options. Disabling the `SOLVE3D` option simplifies the model to only use a simplified subset of the primitive equations called the shallow water equations. Moreover, CROCO can relax some of the assumptions underlying the PE when necessary. For example, activating the non-Boussinesq mode enables using the complete set of Navier-Stokes equations for high-resolution modeling.

This thesis will primarily focus on the default configuration, solving the primitive equations.

2.1.2 Numerical Model

This section gives a fundamental overview of CROCO's underlying numerical model. CROCO includes many additional features that will not be discussed here.

CROCO is a split-explicit, free-surface model. A free-surface model represents the ocean's dynamic surface as opposed to a fixed, rigid lid [20]. This can capture real-world changes caused by things like waves, tides and atmospheric interactions. The "split-explicit" aspect refers to CROCO's time-splitting algorithm, which is derived from the older Regional Ocen Model (ROMS) [21] and described in [20]. Their reasoning for using a time-splitting approach will be described in the following section.

Time Splitting

CROCO splits time-stepping into two different modes, a faster barotropic and a slower barotropic mode working on different time scales, needed to accurately model two different kinds of water flows.

The terms "barotropic" and "baroclinic" are used to categorize the distribution of water density and pressure in the ocean [22]. Barotropic refers to conditions with uniform water density across depths. Consequently, barotropic flows are flows that are mainly caused by pressure differentials. The horizontal velocity of a barotropic flow is only a function of depth-independent variables, such as latitude and longitude. Baroclinic flows in contrast are mostly driven by variations in water density with depth. Barotropic flows represent fast, small-scale motions of the ocean that are primarily driven by pressure gradients, such as tidess. On the other hand, the baroclinic mode represents slow, large-scale motions of the ocean that are driven by density variations, such as the thermohaline circulation, a global oceanic circulation pattern driven by temperature and salinity variations [22].

These differences mean that both flows act on different time scales, with the barotropic mode having a much shorter time scale than the baroclinic mode and therefore needing smaller time steps. If the same time step size were used for both modes, the time step would be limited by the barotropic mode, which would result in a very small time step size for the baroclinic mode. This would make the simulation computationally expensive and inefficient.

Therefore, the model uses a split-explicit time-stepping algorithm that treats the barotropic and baroclinic modes separately and uses different time step sizes for each mode. It uses multiple short time steps in the barotropic mode for surface elevation and barotropic momentum and employs a larger time step for active tracers (temperature, salinity) and baroclinic momentum [20].

Time Stepping

Time-discretization in CROCO uses a third-order predictor-corrector scheme (LFAM3) from [20]. The barotropic mode is advanced using a forward-backward time-stepping algorithm (Adams Moulton, AM3), while the baroclinic mode is advanced using a leapfrog (LF) time-stepping algorithm.

Barotropic results are calculated for many fast barotropic time steps during one slow baroclinic time step. These values are then averaged over the steps (using a filter for a weighted averaging function) and the results feed back into the 3D momenta. The averaging is needed to prevent temporal aliasing. What will be most important later is that every step in the loop consists of one slow step and a loop of *n_fast* fast steps.

Time steps in CROCO are subdivided into four parts: *step2d* (the fast barotropic step) and three parts of the slow baroclinic step, *pre_step3d*, *step3d_uv* and *step3d_t* like shown in the following commented code extract from [19]:

```
call prestep3d_thread() ! Predictor step for 3D momentum and tracers
call step2d_thread() ! Barotropic mode [occurs in an additional loop]
call step3d_uv_thread() ! Corrector step for momentum
call step3d_t_thread() ! Corrector step for tracers"
```

In every time step, *pre_step3d* is called first as the predictor step for momentum and tracers. Next, the fast, barotropic step which is implemented in *step2d* is repeated in a loop *n_fast* times to do its job. After that *step3d_t* and *step3d_uv* implement the corrector steps for 3D momentum and tracers.

2.2 Grid and Coordinates

CROCO utilizes a grid-based method to discretize space for numerical computations. This grid is based on a curvilinear horizontal coordinate system rather than a conventional Cartesian coordinate system. The two dimensional part of the CROCO grid can be viewed as a grid with coordinates awith the primary *xi* and *eta* axes representing latitude and longitude, respectively. The third, vertical, spatial dimension is organized in columns extending vertically over the grid cells, allowing for modelling of various depths and heights in the ocean.

2.2.1 Staggered Grid

CROCO uses a staggered grid which means different properties are evaluated at different locations within a cell. Scalar variables such as density, sea surface height (or free surface height), temperature and other tracers are defined at the center of a cell, while velocities and momentums are defined at the boundaries (or faces) between two cells, describing

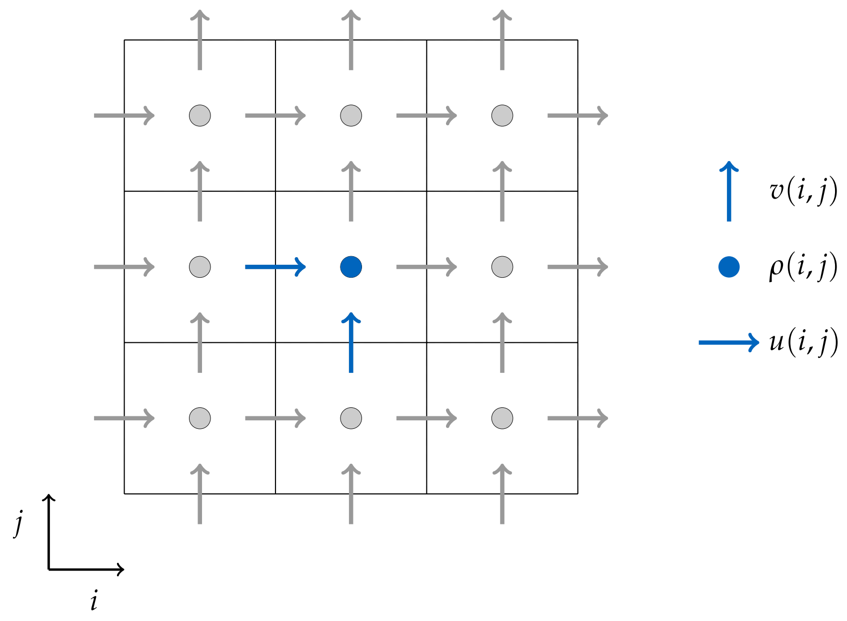


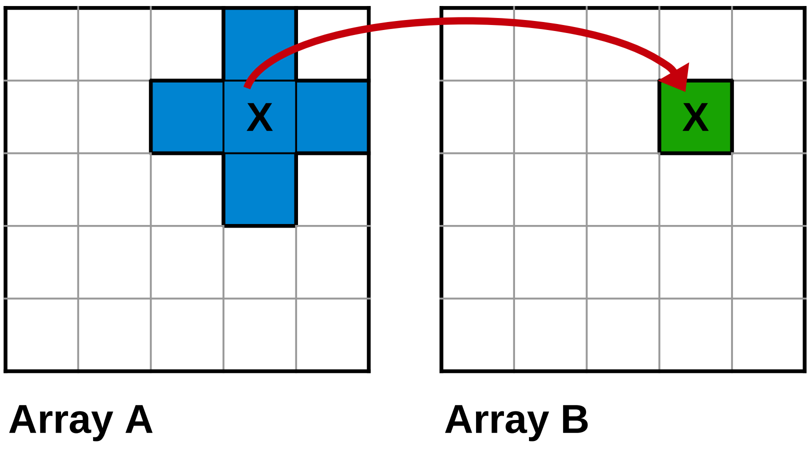
Figure 2.1: A staggered grid. Scalar variables are defined in a cell's center, velocities and momentum in the middle of its eastern or northern border, dependent on the direction. The ρ, u and v points for the grid cell (i, j) are marked in blue.

the flow from one cell to the other. Figure 2.1 depicts the staggered grid with the described locations. The central location will be referred to as the ρ-point, while the location of the zonal (east-west-direction) flow will be referred to as the u-point, and the location of the meridional (north-south-direction) flow will be referred to as the v-point. Additionally, there exist a ψ-point in the corners of the cells and a w-point that is vertically staggered, which will not be relevant here.

At the edges of the simulation space, there are boundaries, either physical ones or for example periodic boundary conditions.

2.2.2 Stencils and Kernels

The computations that CROCO performs to model the ocean can then be seen as stencil computations on this grid. This means that each cell's value is calculated as a combination of the values of other cells in its neighborhood. The stencil that is applied contains information about which cells in the neighborhood are being evaluated and how they are contributing. An example of this can be seen in Figure 2.2. To compute each cell, the stencil is moved across the grid and applied to each cell. In code, this is implemented with iterative stencil loops which are loops iterating over the space. The code inside the loops describing and implementing the stencil computation is called a kernel (see Figure 2.2 for an example kernel).



```
B(i,j) = A(i,j) + A(i+1,j) + A(i-1,j) + A(i,j+1) + A(i,j-1)
```

Figure 2.2: Applying the blue five-point-stencil to array **A** as shown means the value of the cell marked with X in array **B** is computed using the values of the cells under its five-point-stencil on **A** (shown in blue). This calculation is done for each cell in **B** by shifting the stencil over the array.
The formula below the grids shows a code example for a very simple kernel that uses a five-point-stencil.

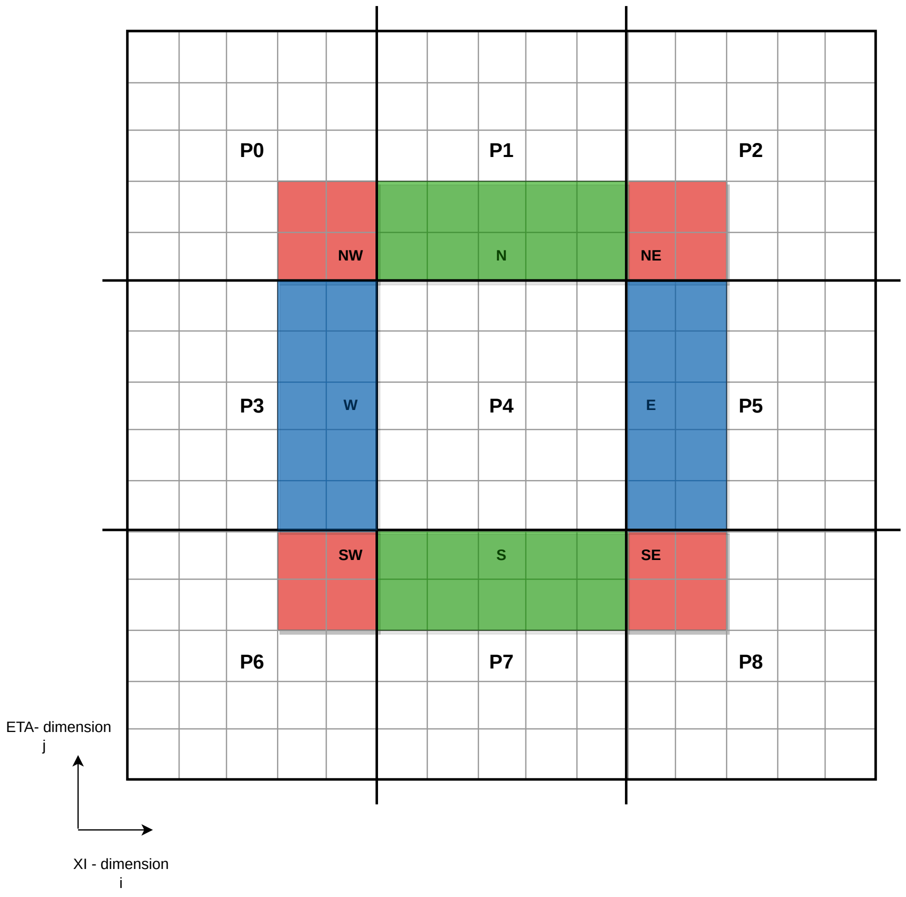


Figure 2.3: In this example, the whole grid is divided into 9 domains. Each domain has an additional copy of the cells belonging to its neighbor that are next to its boundary, called halo cells (here only shown for subdomain P4). A subdomain has two halo layers consisting of eight regions, each belonging to a neighbor in a different direction. Thus, instead of only storing its own $5x5$ values, a process stores the $9x9$ values of the whole area consisting of its interior region plus the halo.

2.3 Parallelization

The described calculation process is very computationally intensive. Therefore, a high degree of parallelization, which distributes the computational load over several processors, can be very advantageous. This is especially true for the modeling of large areas and for the resolution of small scales. CROCO provides parallelization options using either shared memory with OpenMP or distributed memory with MPI. Both strategies work on the basis of a domain decomposition in which the spatial domain is divided into several sub-domains along both the *xi* axis and the *eta* axis. In addition, the working arrays are divided into smaller subarrays, each containing only the area that corresponds to a subdomain. This allows parts of the code to be executed in parallel, with different processors working on different subdomains concurrently. For very large simulations, it is often necessary to use more than one compute node. Therefore, the Message Passing Interface (MPI) instead of OpenMP is required as CROCO's parallelizing strategy.

2.3.1 Parallelization with MPI

With distributed memory, a processor has easy access only to the working arrays of its own subdomains, which are stored in its local memory. The stencil calculation for a cell is dependent on the data of neighboring cells. For most cells, these neighbors are part of the same subdomains and their data is therefore locally available. For cells located at the boundaries of a subdomain, neighboring cells may belong to different subdomains that have been assigned to different MPI processes on different processors. Consequently, the necessary data is not in local memory, as it belongs to a different process. To access it, these two processes must communicate using MPI messages. Depending on the numerical scheme, CROCO's halos can have a width of two or three layers of cells.

To minimize the number of communication operations for accessing neighboring cell values near subdomain borders, each computational process maintains a local copy of cells directly bordering its subdomain but outside of it. These cells are known as halo or ghost cells. Their sole purpose is to provide input values for stencil operations that are centered on cells within the interior of a subdomain. An illustration of such a halo is provided in Figure 2.3. As computation progresses, these cells will contain newly computed values within one subdomain, while the halo cells in adjacent subdomains will still retain their old values. To ensure consistency in the simulation, it becomes necessary to regularly update the values of these halo cells. This is done using halo exchanges, where each process sends the values of some of its cells to other processes, which store this data in their halo cells.

When utilizing MPI for parallelization, halo exchanges account for the majority of communication between processes. Consequently, an efficient implementation of halo exchanges is critical for achieving good runtimes.

2.4 Halo Exchanges in CROCO

This section describes how halo exchanges are implemented in CROCO with the use of MPI.

2.4.1 Halo Setup

For halo exchanges in CROCO, a process with its corresponding subdomain needs to exchange data with the eight cells in its 8-neighborhood (Moore neighborhood) that each own some of the cells in its halo, four on the sides (North, South, East and West) and four on the corners (North East, North West, South East and South West). Figure 2.4 shows the exchange for one process and three selected neighbors.

2.4.2 Synchronous Halo Exchanges

CROCO implements the halo exchanges in a synchronous fashion with two-sided, blocking MPI operations. Each process exchanges the data using separate send and receive operations for each communication direction. This is implemented in the *exchange* and *MessPass* subroutines in the following waylike shown in Figure 2.5:

Communication is structured in a way so that it always happens between a pair of neighbors exchanging data in both directions before pairing up with a new neighbor. More details in this additional level of synchronization can be found in Figure 2.6.

Before transmitting data, each of the eight directional segments of the halo region is copied into a temporary one-dimensional send buffer, so it can be accessed contiguously (see Figure 2.7). Next, the process proceeds to initiate receiving data from its neighbor into a temporary receive buffer (equivalent to the send buffer) with a non-blocking *MPI_IRECV* call. This asynchronous operation allows the process to continue its tasks without waiting for the data transfer to finish. Simultaneously, while the receive operation is still operating, the process sends its data using a blocking *MPI_SEND* call. Using a blocking send means that this operation will hold until the data transmission is complete. After completing the send, it uses an *MPI_WAIT* to wait until the receive operation terminates which means that it has received data for all its halos. Then it copies the data from the receive buffer back to the corresponding halo cells. Example code for a halo exchange in one direction is provided in Figure 2.8.

Subdomains next to the border of the whole area do (dependent on the boundary conditions) not perform a halo exchange in directions where no neighbor exists. The information if a neighbor in a direction exists, if it is used for halo exchanges and what its rank (used as source and destination in the MPI calls) is, is stored in a row of variables explained in Table 2.1.

2.5 Compilation and Build Process

This section provides an overview of CROCO's compilation and build process.

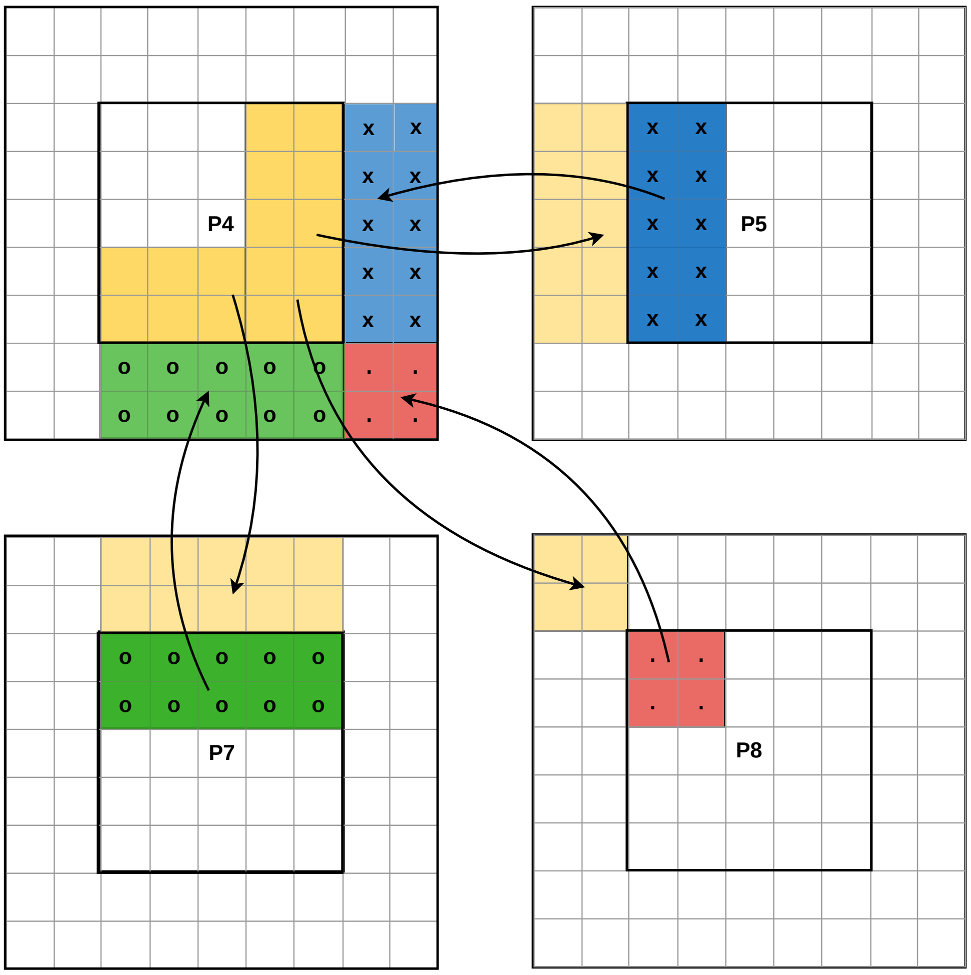


Figure 2.4: P4 and its halo exchanges with its neighbors in the South, East and South East direction. It receives updated values for its halo cells (marked with x, o and $\cdot$ and sends current data for its own cells to the corresponding neighbors.)

```
SUBROUTINE xyz
  ...
  perform computations, change X
  ...
  CALL set_boundary_conditions(X)
  CALL exchange_tile(X)
      ...

END SUBROUTINE

SUBROUTINE exchange_tile(X)
  set_periodic_boundary_boundary_conditions(X)
  CALL MessPass_tile(X)
END SUBROUTINE

SUBROUTINE MessPass_tile(X)
  DO for all neighbors:
    copy halo values to buffer
    CALL MPI_IRECV
    CALL MPI_SEND
  ENDDO

  DO for all neighbors
    wait for irecv to terminate
    copy recv buffer to halo
  ENDDO

END SUBROUTINE
```

Figure 2.5: Pseudocode for synchronous communication scheme in original CROCO code.

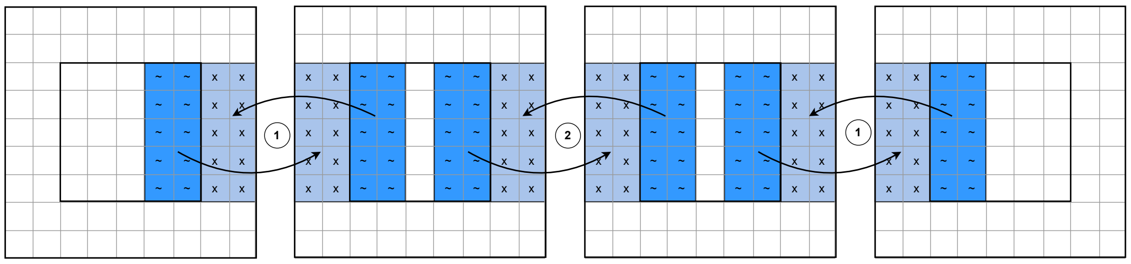


Figure 2.6: Additional synchronization level in CROCO's halo exchanges (only shown for the East / West direction). Every process pairs up with one neighbor and finishes communcation with that neighbor before communicating with the neighbor on the other side. In this example, first both exchanges marked with 1 happen (concurrently), the exchange marked with 2 only happens afterwards.

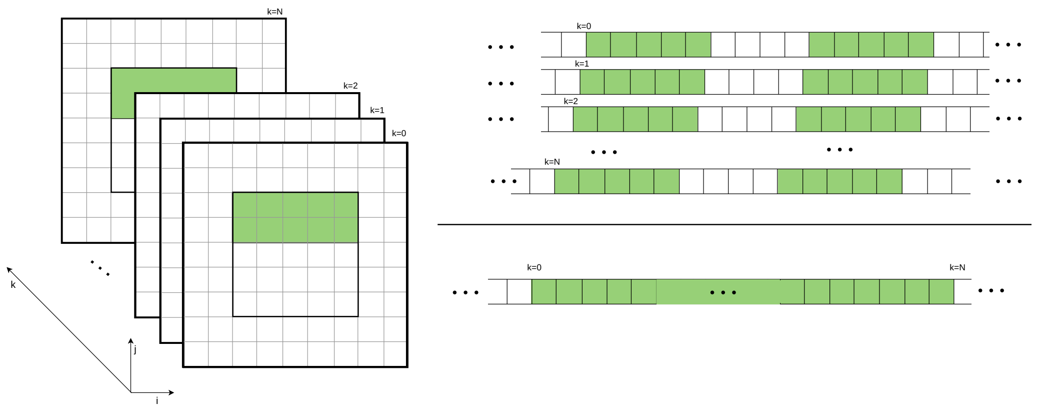


Figure 2.7: Serialization of data for a halo exchange to the south. The data in the logical array locations in green is arranged in memory in non-contiguous memory cells as shown on the top right. After copying the data to the buffer it is stored in continguous memory cells as shown on the bottom right.

```fortran
...
! copy halo cell data to send buffer
  DO j=jmin,jmax
    DO ipts=1,Npts
      jbuf_sndW(j-jmin+(ipts-1)*jshft)=A(ipts,j)
    ENDDO
  ENDDO

  ! start non-blocking receive from western neighbor
  CALL MPI_IRECV (jbuf_revW, jsize, MPI_DOUBLE_PRECISION,
                      p_W, 2, MPI_COMM_WORLD, req(1), ierr)
 ! do blocking send to western neighbor
  CALL MPI_SEND (jbuf_sndW, jsize, MPI_DOUBLE_PRECISION,
                      p_W, 1, MPI_COMM_WORLD, ierr)

  ...
  ! communicate with other neighbors
  ...

 ! wait for completion of receive
  CALL MPI_WAIT (req(1),status(1,1),ierr)

  ! copy received data into corresponding halo cells
  DO j=jmin,jmax
    DO ipts=1,Npts
     A(ipts-Npts,j)=jbuf_revW(j-jmin+(ipts-1)*jshft)
    ENDDO
  ENDDO
  ...
```

Figure 2.8: Code implementing the original halo exchange of a process with its western neighbor. Extract from subroutine MessPass2D_tile.

Table 2.1: Variables that are used in CROCO to control halo exchanges. This table provides examples for two directions.

Variable Name	Content	Notes
p_n	MPI rank of northern neighbor.	
north_inter	Variable has northern neighbor or the simulation uses periodic boundary conditions.	
north_inter2	Variable has northern neighbor for halo exchange (meaning the corresponding neighbor tile is not a land tile associated with *MPI_noland*).	If *MPI_noland* == *false* *north_inter* and *north_inter2* are identical.
p_sw	MPI rank of north eastern neighbor.	
corner_sw	Variable has south western neighbor for halo exchange and this variable is not a land tile associated with *MPI_noland*.	Equivalent to e.g. *north_inter2* for the south east corner. Corners do not have variables equivalent to e.g. *north_inter*.

CROCO primarily uses Fortran, with .F files for Fortran 77 code and .F90 files for Fortran 90 code, supplemented by .h header files. CROCO utilizes a C preprocessor with preprocessing keys to help simplify code generation and compilation. This approach provides considerable flexibility. Users can select different features, scenarios and numerical schemes to automatically generate code tailored to specific use cases.

They can configure parameters and settings in `param.h` and `cppdefs.h` using C preprocessor directives. The CROCO code is then preprocessed by the preprocessor, which selectively includes, replaces and inserts code segments based on the specified flags and directives. This creates a customized code version for the selected keys. CROCO's input options are distributed over several code files. These inputs include grid settings, options for parallelization, file paths, input and output specifications, time-stepping parameters, parameterizations, boundary conditions, numerical schemes, model configurations and additional variables [19]. These inputs are used to the design of the code to meet the specific simulation criteria.

The jobcomp script manages the compilation process, providing configurations for paths, compilers, libraries and other important settings.

2.6 Test cases

CROCO offers a range of test cases. These test cases use various options, numerical methods, scenarios, and inputs, enabling users to test CROCO on a diverse set of tasks. One example is the BASIN test case, which is used as the test case in this thesis. The CROCO manual [19] characterizes the BASIN test case as follows:

> This is a rectangular, flat-bottomed basin with double-gyre wind forcing. It produces a western boundary current flowing into a central Gulf Stream which goes unstable and generates eddies if resolution is increased.

The parameters that are relevant here are the following: It is a rectangular water basin with solid physical boundaries on all four sides, not using periodic boundary conditions. For time-stepping it uses CROCO's default model for solving the Primitive Equations that was described in Section 2.1.2. Most additional model options are disabled apart from the ones described above. MPI and OpenMP are both turned off by default but MPI will be enabled later on.

2.7 Automatic Code Generation for Communication

This section describes why automatically being able to generate code implementing efficient communication would be very beneficial in the case of CROCO.

2.7.1 Parallelization and Performance

CROCO uses the parallelization method described earlier, employing spatial decomposition to partition the computational space into subdomains that can be assigned to multiple processor cores to be processed concurrently. This approach uses a high degree of parallelism, which is essential for weather and climate simulations. These simulations frequently run on many nodes concurrently to enable quick weather predictions and efficient long-term climate modeling without excessive computational time.

2.7.2 Communication

The need for high performance in cases like these makes it necessary to use distributed systems like clusters for some applications. This makes communication between nodes necessary, causing - dependent on the specific case - additional communication latency which decreases performance. The frequent need for halo exchanges necessitates efficient communication strategies. CROCO's halo exchanges are implemented as synchronous MPI communication as explained in Section 2.3.1. The locations of these halo exchanges are determined manually and are part of CROCO's code, and only slightly modified by the preprocessor.

This leaves room to optimize both the implementation and the placement of CROCO's halo exchanges in order to decrease (or hide) communication latency.

2.7.3 Automatic Code Generation

As seen, CROCO has the ability to automatically generate customized code for specific use cases. This makes manual optimization challenging due to the large number of possible code variations. Automated detection of communication needs, combined with code generation, simplifies and facilitates this process, allowing for optimization tailored to each case.

Both automatically optimizing existing communication schemes for higher efficiency and automatically generating code that newly implements communication are beneficial here.

3 PSyclone

This chapter gives a short overview of PSyclone, a code generation system used in this project for parsing and analyzing CROCO code and modifying it to implement communication.

PSyclone is a system for code generation with the aim to support domain-specific languages (DSLs) for finite element, volume and difference codes [23]. With its additional ability to support existing codes, this makes it a good choice for the parsing, analyzing and modifying CROCO code.

The following sections present an overview of PSyclone's basic structure and some of its functionalities as described in its documentation [23].

3.1 PSyKAl Separation of Concerns

The core concept within PSyclone is the "PSyKAl" separation of concerns, which divides the code into three distinct layers: the Parallelization System (PSy) layer, Kernel layer, and Algorithm layer.

The Algorithm layer is where the PSyclone users define their algorithms through calls to kernel routines and built-in operations, working with entire fields as their logical operation units. The Kernel layer, on the other hand, determines the implementation of these kernels, focusing on local fields (e.g., a column or a set of cells). Kernels in this layer operate on raw Fortran arrays, allowing for compiler optimization. Neither the Algorithm nor the Kernel layer allow any parallelization directives, as this is the domain of the PSy layer in between them. The PSy layer connects Algorithm and Kernel layers, managing arguments, invoking kernels, executing operations, and handling distributed memory operations such as halo exchanges.

Furthermore, PSyclone offers an automatic code generation feature for the PSy layer. This simplifies the optimization process and reduces the likelihood of errors, making it more efficient and accessible.

3.2 Nemo API

PSyclone also supports working with existing code not following the presented separation of concerns. This is done with an API for an ocean model called Nemo **nemo**. To make Nemo code fit with PSyclone's three layers, it is viewed as PSy layer code with inlined kernels and without an algorithm layer. The existing PSy layer code is converted into a higher-level representation by applying rules based on the Nemo coding conventions. PSyclone can then process and modify that representation and generate new code. Although the Nemo API was developed for this specific ocean model, it can also be used with CROCO code with minor modifications. (see Chapter 4).

3.3 PSyIR representation

To be processed with PSyclone, existing code like Nemo is parsed and transformed into an abstracted representation called PSyclone Internal Representation (PSyIR). Apart from being created by parsing existing code like with the Nemo API, this representation can also be built from scratch. PSyIR representations exist for both kernel and PSy layers. As Nemo code is seen as PSy layer code with inlined kernels, much of it uses kernel-layer PSyIR representation.

PSyclone uses different classes to represent code elements, referring to them as PSyIR nodes. Examples include Loops, References, ArrayReferences, Literals, Calls, or Operations. These nodes can be organized in a tree known as the PSyIR tree, which captures the entire structure of the code. Additional methods are available to modify and traverse this tree and to create new PSyIR nodes.

PSyclone provides a Python script called *psyclone* that can be used to parse existing CROCO code into an analogous PSyIR representation. PSyclone also includes transformation scripts that can be applied to the PSyIR of a code file and change it. Afterwards script can be used to generate new code based on the altered PSyIR. It also allows to write new transformation scripts

3.4 PSyclone Use in This Project

In this project PSyclone's Nemo API is used to parse CROCO code into an analogous PSyIR representation using the provided *psyclone* script. This representation can then be analyzed to find out when communication needs to happen. Then a transformation can be used to add new communication operations and removes old ones and generate the modified code.

4 Setup

As stated in Chapter 1 the approach represented in this thesis has two parts: First, code analysis and the identification of necessary communication options, and then code generation. This section describes how these two stages are embedded in the context of CROCO, its build pipeline, and PSyclone and Poseidon as important tools.

4.1 General Approach

The general approach is the following: First, CROCO code that uses MPI is built and compiled by adding the `# define MPI` directive as an additional parameter. The result of that build contains code for managing subdomains, subprocesses, and setup and initialization for using MPI. It is, therefore, used as the basis for generating a code version with optimized communication and also using two-sided MPI communication. For this, the original synchronous halo exchanges need to be removed while newly generated asynchronous halo exchanges are inserted into the code. Asynchronous communication is used with the goal of overlapping computation and communication, thus hiding communication latency.

PSyclone's `psyclone` script can, however, only parse and analyze individual files. For a global analysis of variable dependencies that is needed here, this is not enough. This prototype, therefore, uses a tool called Poseidon for global analysis.

During this analysis, kernels and stencils are extracted from the code to analyze data dependencies and identify when halo exchanges of which variables are necessary. Then, this data is used to determine optimal locations for communication operations. Finally, PSyclone is used to modify the original code by inserting code in those locations implementing an asynchronous communication scheme.

4.2 Poseidon

Poseidon is a tool that integrates PSyclone function to be easily usable to do global analysis on CROCO code. It provides a build pipeline that can be used to build CROCO code and modify it based on both specific arguments and code analysis using PSyclone. Additionally, it contains functionalities for kernel extraction, basic dependency, and stencil analysis, benchmarking, and plotting-

4.2.1 Poseidon Pre-parser Pipeline

Poseidon's pre-parser and its associated build pipeline provide tools for an automated build of CROCO code, executing PSyclone transformations on that code, global analysis, testing, and benchmarking. Additionally, it offers patching capabilities to modify configuration and code files before the build process.

Configuration parameters are specified in a `config.json` file, containing options like file paths and the PSyclone transformations to be applied to these files, the specific test case, or the chosen parallelization method.

This pipeline involves several steps: Before the actual build process, the CROCO code is modified. These modifications include both changes to the input files (such as choosing a test case or adjusting the number of steps) and minor changes to CROCO's build process and code. One such change is creating a file specifying which of the code files should be processed by transformation scripts (as taken from the configuration file) and adding code to the `jobcomp` file that adds this processing step to CROCO's build process.

These transformations are automatically executed between pre-processing and compilation. The files that need global analysis (as specified in the configuration file) are then parsed with the *poseidon-pre-parser*, conducting global analysis using and saving intermediate results. This process involves parsing the code, extracting kernels, conducting basic dependency analysis, building an ordered list of kernels, and generating temporary results. Afterward, a second build of CROCO is executed, using the PSyclone transformations once more and utilizing the saved results to generate new code. After that, the simulation can be executed to generate results.

This entire pipeline is executed through a Python script named *bench*, which also supports automated testing, benchmarking, and plotting.

4.2.2 Patching

Additionally, Poseidon has some more patching abilities that make it possible to change arguments in configuration and Makefiles to adapt the build process and to make additional smaller changes to CROCO's code before the build process.

There are two ways in which the patching is done. The first one uses git's patching abilities [24], replacing specified lines in specified files with provided alternatives. A list of these patches can be given in a configuration file and will automatically be applied before the build process. Some patches are provided that concern Poseidon's build pipeline and minor fixes to make CROCO code compatible with PSyclone. This way of patching is also used to change the input parameters of the test case for using different grid dimensions.

The other way uses Python code to make changes. Poseidon provides a set of functions that are able to replace given strings or lines of code in designated files. This works similarly to the first way but allows for more flexibility regarding the inputs. This is used for making changes like enabling MPI with `# define MPI` or adjusting the number of time steps.

The patching functions are also used for the fixes in Section 4.3.1 (using the git patches) and Section 4.3.2 (using Python code).

Patches

4.3 Patching Process

To allow PSyclone to parse the CROCO code, some additional adaptations were made using the patching process described in Section 4.2.2. Section 4.3.1 describes a patch that was made using git.

4.3.1 Step3d Patch

Because of some limitations in PSyclone, certain array shapes used in CROCO cannot be parsed correctly. For example

```
REAL :: A(- 1 : Lm + 2 + padd_X, - 1 : Mm + 2 + padd_E), gamma
```

is parsed to

```
a: DataSymbol<UnknownFortranType('REAL␣::␣A(-␣1␣:␣Lm␣+␣2␣+␣padd_X,␣-␣1␣:␣
    Mm␣+␣2␣+␣padd_E),␣␣gamma'), Local>
```

adding multiple variable names into one DataSymbol and its UnknownFortranType. This causes problems as it hides variables like gamma from PSyclone and fparser. A following declaration of the therefore unknown variable as a parameter

```
parameter (gamma = 1)
```

will lead to an error. Patching this for all parameters with this problem in all files is necessary. This is done with a patch that gives all scalar parameters their own variable definition, changing, for example

```
real A(GLOBAL_2D_Array), gamma
parameter (gamma = 1)
```

to

```
real A(GLOBAL_2D_Array)
real gamma
parameter (gamma = 1)
```

This allows PSyclone to correctly parse those files.

4.3.2 MPI Version Patch

Because of a bug in the version of fparser that is part of the PSyclone version used for this project, parsing code files with include statements was not possible. This made it necessary to switch from using `include 'mpif.h'` in the original CROCO code to using the more modern `use mpi` as it's recommended in the official MPI standard [25]. To do this, a patching function in the pre-processing stage (after the pre-processor) searches all files for subroutines containing the line `include 'mpif.h'`. This line is then deleted, and instead, `use mpi` is inserted in a line directly before the `implicit none` statement.

Changing this makes it also necessary to modify some MPI calls in the existing CROCO code that work with `include 'mpif.h'` but cause errors with `use mpi`. This is fixed by additional patches adding a missing variable `ierror` in the `MPI_abort` calls where there is none and modifying the status variable in several `MPI_wait` calls to correct its data type.

5 Implementation I - Analyzing Stencils and Scheduling Exchanges

This section describes the approach to first part of the problem, determining when halo exchanges of which variables are necessary.

5.1 Basic Approach

First, PSyclone is used to parse all necessary code files and construct the PSyIR tree. This tree is then traversed to extract a list of kernels (in execution order) with their corresponding loops and stencils. Afterwards, these kernels and stencils are used to determine data dependencies and determine when a halo exchange is necessary and find optimal time windows for asynchronous halo exchanges. Later, code will be generated and inserted at these determined locations to implement the halo exchanges.

5.2 Kernels

It can be assumed that in the CROCO code, all relevant computations (for this task) are done using kernels implementing stencil computations. These kernels are nested inside a set of kernel loops that iterate over the space. In these loops, CROCO uses the loop variables *i* and *j* for iteration over the *xi*- and *eta*-dimensions of the two-dimensional space of a subdomain. Both, loops that iterate over different dimensions (e.g. time) and loops that iterate over parts of these dimensions only use other indices (for example *ipts* for the iterator operating only on the halo part of the *i*-dimension).

The kernels and kernel loops can be nested with other loops, for example iterating over the vertical space, time or tracers. A kernel and all loops around it can be seen as one kernel block.

5.2.1 Kernel Extraction

The process of parsing, traversing and extracting the kernels was adopted from the way the *poseidon-pre-parser* handles it with very minor changes. The kernels are extracted from code by traversing the routines making up the PSyIR tree of the CROCO code. All nests of loops in a routine are tested whether they contain loops using each of the control variables *i* and *j* and therefore represent a kernel block. If yes, the corresponding kernel is extracted. Every extracted kernel is represented by a kernel object that is saved in a list of kernels in execution order. To store information on its kernel loops, each kernel has the PSyclone node representing the outermost loop of its kernel block as an attribute. Even though some of the kernels are executed multiple times in the call tree, there is only one kernel object instance created per kernel. This object can be inserted into the ordered kernel list multiple times if necessary.

Each kernel stores a list of variables and assignments, mapping the left hand side (lhs) result of each assignment to its right hand side (rhs) arguments.

The following example is a simplified code extract from *step2d_fb_tile*:

```
DO j=jstrv-1,jend
  DO i=istru-1,iend
          zeta_new(i,j)=zeta(i,j,kstp) + dtfast*pm(i,j)*pn(i,j)*(duon(i,j)-
              duon(i+1,j)+dvom(i,j)-dvom(i,j+1))
  ENDDO
ENDDO
```

This code of a kernel block contains a kernel inside of the two kernel loops. The corresponding kernel object has the following attributes:

- The outer loop, in this case the j-loop (as a PSyIR Loop) as the root of the whole kernel block. This loop does also contain the PSyIR Nodes of all loops nested within it.
- The lhs result ($zeta\_new(i,j)$) as a PSyIr Reference together with all its rhs arguments as PSyIR Nodes (References, Literals or Operations):
  - zeta$(i,j,kstp)$
  - $dtfast$
  - $pm(i,j)$
  - $pn(i,j)$
  - $duon(i,j)$
  - $duon(i+1,j)$
  - $dvom(i,j)$
  - $dvom(i,j+1)$

Kernel Preprocessing

Information on stencils and data dependencies will later be stored and traced through the code with the help of the array names as keys. Because of this, the indices and variables are further pre-processed in the following way: For all variables that are used for array indexing it is tested whether they are the control variable of one of the loops in the kernel block. If they are not, this information is stored by appending the variable name to the array name. In the given example, the i and j variables are the control variables of the two kernel loops. $kstp$, however does not correspond to any loop. Therefore $zeta(i, j, kstp)$ is stored with the name *zeta__kstp*.

This is done to differentiate between accessing slices of a dimension and the full dimension with the same index. In the following (made up) example there are two kernel blocks with identical kernels both having the same array reference on *zeta* as the lhs result.

```
! This accesses only one slice of the array
kstp = 1
DO i=...
  DO j=...
    ! stored with the name zeta__kstp
    zeta(i,j,kstp) = ...
  ENDDO
ENDDO

! This loops over the third dimension
DO kstp=...
  DO i=...
    DO j=...
      ! stored with the name zeta
      zeta(i,j,kstp) = ...
    ENDDO
  ENDDO
ENDDO
```

But since they are used within different loops they access different parts of the array and are therefore stored using different names.

Exception: Tracer Loops

CROCO uses two different types of kernels to operate on the tracer space (see Figure 5.1). While some of them are located within a regular set of nested loops, other kernel blocks are located in one subroutine that is called from within an additional loop iterating over the tracers, in another subroutine. In the second case, the outer loop in another subroutine is seen as part of the kernel block to make the representation of both types of kernels consistent. For this, every kernel gets a *base_node* attribute that is the PSyclone PSyIR representation of the outermost loop iterating over a variable that is used as an array index inside the kernel. For the kernels operating on the tracer space this means, the base node is always the outermost for-loop of a kernel, no matter how it is called.

```
  DO itrc=1,nt ! base node of the kernel
   DO k=1,n
     DO j=jstr,jend
       DO i=max(istr-1,1),min(iend+2,lm+1)
         fx(i,j)=(t(i,j,k,nadv,itrc)-t(i-1,j,k,nadv,itrc))
       ENDDO
     ENDDO
     if (istr.eq.1) then
       DO j=jstr,jend
         fx(0,j)=fx(1,j)
       ENDDO
     endif
     if (iend.eq.lm) then
        DO j=jstr,jend
         fx(lm+2,j)=fx(lm+1,j)
       ENDDO
     endif
     ...
```

```
  DO itrc=1,nt ! base node of kernel
    call t3dmix_tile(istr,iend,jstr,jend, itrc, ...)
  ENDDO


subroutine t3dmix_tile (istr,iend,jstr,jend, itrc, ...)
  ...
  DO k=1,n ! rest of the kernel
   ...

   DO j=jstr,jend
     DO i=istr,iend+1
         fx(i,j)=0.5D0*diff3u(i,j)*pmon_u(i,j)*(hz(i,j,k)+hz(i-1,j,k))*(t(i,
             j,k,nrhs,itrc)-t(i-1,j,k,nrhs,itrc))
     ENDDO
   ENDDO

   ...
```

Figure 5.1: Two different types of kernels containing *itrc*. On the top there is a simple kernel, on the bottom, the first kernel loop (the base node) is located in a different subroutine than the rest of the kernel.

5.2.2 Kernel Connectors

Kernel connectors (or connector kernels) implement passing arrays as arguments for subroutine calls. Many of the subroutine calls in CROCO have arrays as dummy arguments. Most of them pass array variables to other subroutines using the scratch files *A2d* and *A3d* (for two- and three-dimensional arrays respectively), but other arguments are also used.

In this way, arrays and the data dependencies associated with them get passed to other routines, modified there and returned. As stated before, tracing these dependencies is done using the array names as keys. Using a variable in a function call usually leads to this variable having a different name in the caller and callee subroutines. Without any measures to link both names for the argument together this would mean information on the dependencies getting lost at the start and end of subroutine calls.

To link the names of the dummy arguments of the subroutine that is called to the corresponding names in the caller, a call is represented by a connector kernel. Connector kernels are equivalent to simply assigning all values of one array to a second one. This is not only used for representing passing arguments in routine calls but will be used for modifying time indices later. An example for a subroutine call and its corresponding connector kernels is shown in Figure 5.2.

Subroutine calls are implemented by inserting a start connector before the kernels of a subroutine and an end connector afterwards to link changes in the input arrays during the subroutine execution back to the original arrays.

Connector kernels are also used to represent changing time indices. Figure 5.3 shows an example for this. The timestep reassignment represented by the code on the left (a simplified version of the one used in CROCO in *step2d*) has an effect equivalent to that of the code on the right side (for all affected arrays). Therefore the changes to variables used as time indices, which are happening outside of a kernel, are represented by connector kernels containing assignments like the ones shown on the right side of Figure 5.3.

Additionally, some minor changes need to be made to some of the kernels.

5.2.3 Changes to Loop Bounds

The original CROCO code does not iterate over the whole array with all its halo layers in each kernel loop. In some kernel loops, the values of some of the halo cells are not used (e.g. directly before halo exchanges). This causes incorrect results when determining the necessary halo exchanges as the presented approach assumes that all kernels are executed over the entirety of an array. This is solved by extending some kernel loop bounds to cover a larger area. This change is made by modifying the loops' PSyIR representation to later generate code with the new loop bounds (see Figure 5.4).

```
subroutine step2d(tile)
  ...

  call step2d_fb_tile(istr, iend, jstr, jend, a2d(1,1,trd), a2d(1,2,trd),
      a2d(1,3,trd), a2d(1,4,trd), a2d(1,5,trd), a2d(1,6,trd), a2d(1,7,trd),
       a2d(1,8,trd), a2d(1,9,trd), a2d(1,10,trd), a2d(1,11,trd), a2d(1,12,
      trd), a2d(1,13,trd))
  return
end subroutine step2d

subroutine step2d_fb_tile(istr, iend, jstr, jend, zeta_new, dnew, rubar,
    rvbar, drhs, ufx, ufe, vfx, vfe, urhs, vrhs, duon, dvom)
  ...
```

```
! start connector
DO i= ! whole xi-domain
  DO j = ! whole eta-domain
   zeta_new(i, j) = a2d(1,1,trd, i, j)
   dnew(i, j) = a2d(1,2,trd, i, j)
    ...
  ENDDO
ENDDO

! end connector
DO i= ! whole xi-domain
  DO j = ! whole eta-domain
   a2d(1,1,trd, i, j) = zeta_new(i, j)
   a2d(1,2,trd, i, j) = dnew(i, j)
    ...
  ENDDO
ENDDO
```

Figure 5.2: Top: Subroutine *step2d_FB_tile* is called by *step2d* with multiple dummy arguments that have different names in both subroutines (e.g. *a2d(1,1,trd)* vs. *zeta_new*).
Bottom: The code for two connectors representing the changes in array names associated with the call. The start connector is inserted before the first kernel in *step2d_FB_tile*, the end connector after the last one to return the results in the arrays.

```
kold = kbak
kbak = kstp
kstp = knew
knew = kstp + 1
```

```
DO i= ! whole xi-domain
  DO j = ! whole eta-domain
   zeta(i, j, kold) = zeta(i, j, kbak) ! kold=kbak
   zeta(i, j, kbak) = zeta(i, j, kstp) ! kbak=kstp
   zeta(i, j, kstp) = zeta(i, j, knew) ! kstp=knew
 ! knew=kstp+1 not necessary as knew is first
     overwritten
  ENDDO
ENDDO
```

Figure 5.3: A time step reassignment like the on the left (note: this is manually written code giving a simplified version of a time step reassignment occuring in CROCO) can be implemented using a connector kernel representing the assignments on the right side.

```
DO j=jstrr,jendr
   do i=istrr,iendr
    z_w(i,j,0)=-h(i,j)
   ENDDO
   DO k=1,n,+1
      cff_w =hc*(sc_w(k)-cs_w(k))
      ...
      DO i=istrr,iendr
         zetatmp=zt_avg1(i,j)
         ...
      ENDDO
   ENDDO
ENDDO
```

```
DO j = jstrv - 2, jend + 1, 1
   DO i = istru - 2, iend + 1, 1
      z_w(i,j,0) = -h(i,j)
   ENDDO
   DO k = 1, n, +1
      cff_w = hc * (sc_w(k) - cs_w(k))
      ...
      DO i = istru - 2, iend + 1, 1
        zetatmp = zt_avg1(i,j)
        ...
      ENDDO
   ENDDO
ENDDO
```

Figure 5.4: Original code (left), with extended loop bounds (right) for the loops in subroutine *set_huv_tile* that need modified loop bounds.

5.3 Stencil and Dependency Extraction and Analysis

This section explains how the extracted kernels and their data are used to determine when halo exchanges for which arrays are necessary.

5.3.1 When and Why Halo Exchanges Are Necessary

First it is important to understand when and why halo exchanges need to happen. Halo exchanges become necessary when the values in the halo cells become inaccurate and outdated.

As previously mentioned, CROCO's kernel computations take place within kernel loops operating on 2D or 3D space. These computations use a combination of neighboring cells' values to compute the result for a given cell. The pattern describing if and how each cell in the neighborhood of a cell contributes to the combination can be viewed as a stencil. Applying a stencil to an entire array is done by moving the stencil over the grid, iterating over the kernel loops.

This means that the value of the center cell of the stencil is dependent on all the other cells that are part of the stencil. Consequently, it becomes impossible to calculate the result of a stencil computation when the stencil around a cell extends beyond the processor's combined interior and halo area as there is no value to access. The result of the stencil computation for this cell is therefore invalid. An example for such a case can be found in Figure 5.5. This means that every application of a kernel causes cells next to the edge of a subdomain area to have invalid values as they would be dependent on values outside of the combined halo-interior area. Subsequent applications of other stencils using these cells' invalid values lead to the area of invalid values within the subdomain growing even more. Figure 5.5 gives an example describing this over multiple steps.

As previously stated, a processor must ensure that all interior cell values are correct. This means it can never happen that the invalid area grows from the halo cells into the interior meaning that cells in the interior have incorrect values. Before this could happen the values of the cells in the invalid area need to be updated. Fortunately, the halo cells belong to the interior of the subdomain of another process calculating their correct values. Hence, it is possible to get these values from the responsible process in a halo exchange. After the halo exchange, all array values are once again correct. In summary, halo exchanges are needed before the invalid area of a subdomain grows from its outside through its halo into its interior.

Validity Vectors

The area of an array A that contains invalid values using a validity vector v. This vector has a length of four and indicates, in each cardinal direction (West, East, South, North), how many rows or columns are invalid:

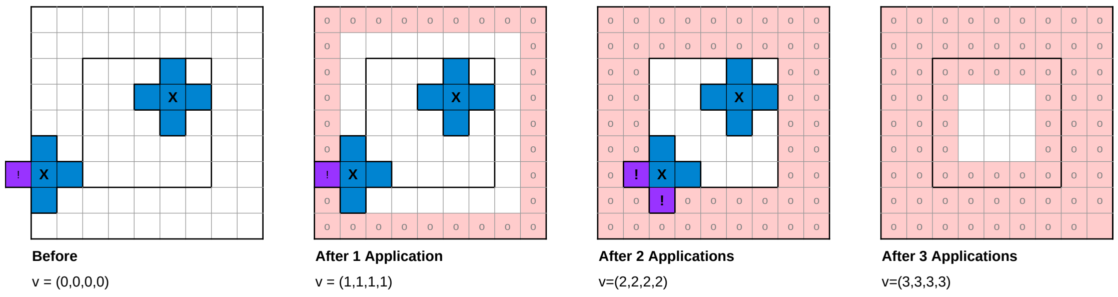


Figure 5.5: Applying the same stencil multiple times to an array leads to the area of cells inside the array that have a valid value shrinking. With every application, here, another layer of cells is dependent on cells outside of this subdomain or its halo.

$$v_A = (v_A^W, v_A^E, v_A^S, v_A^N) \tag{5.1}$$

A halo exchange becomes necessary when any of the four components of the validity vector grows larger than the number of halo layers. Alternatively this can be seen as the effect of accumulating stencils. The computation repeatedly applies stencils to a grid. This means that after some steps a variable is not only dependent on the values under its stencil but as the values under the stencil are themselves dependent on the values under their previous stencil. This means that a cell's "dependency area" is the area of the stencil that was accumulated through multiple applications. This accumulated stencil can be calculated by applying the second stencil to every cell of the first one. This step can be repeated to accumulate even more stencils.

Figure 5.6 shows the accumulated stencils for the repeated application of a 5-point stencil. The accumulated stencil can be used to visually determine which cells are invalid at a point by determining for what cells the accumulated stencil extends over the edge of the subdomains combined area (interior plus halo). The stencil extent, given by the dimensions of the minimal bounding box of this accumulated stencil can be interpreted as the validity vector's components (see Figure 5.6 for examples). If any of these components is larger than the halo size, it implies that applying the stencil to some interior cells of a subdomain is impossible without needing values from cells outside the domain.

Stencil Offsets

This section explains how the stencils are accumulated. The following code (a modified extract from CROCO's *step2d*) serves as an example:

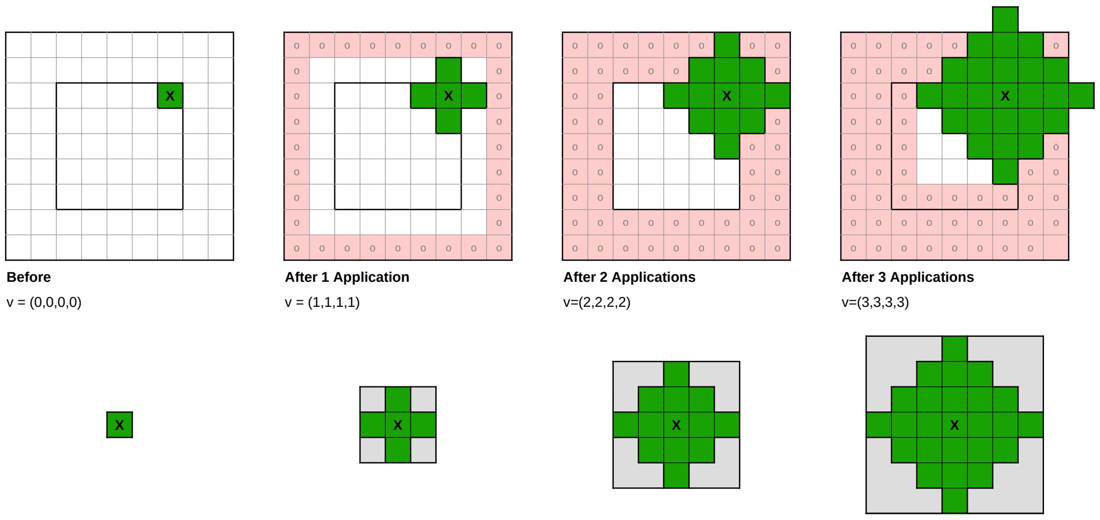


Figure 5.6: Applying the same stencil to an array multiple times causes the area of cells inside the array that have a valid value to shrink. With every application of the 5-point stencil, here, another layer of cells is dependent on cells outside of both this subdomain's interior and its halo.

```
DO j=jstrv-1,jend
  DO i=istru-1,iend
          zeta_new(i,j)=zeta(i,j,kstp) + dtfast*pm(i,j)*pn(i,j)*(duon(i,j)-
              duon(i+1,j)+dvom(i,j)-dvom(i,j+1))
  ENDDO
ENDDO
```

In this code block, the stencil for calculating *zeta_new* can be viewed as a combination of five different stencils, one for each of the kernel's input arrays. To accumulate the stencils for each rhs argument separately, assume they have the following validity vectors before executing the kernel:

$$v_{\text{pm}} = (0,0,0,0) \tag{5.2}$$

$$v_{\text{pn}} = (0,0,0,0) \tag{5.3}$$

$$v_{\text{zeta\_kstp}} = (1,1,1,1) \tag{5.4}$$

$$v_{\text{dvom}} = (1,1,2,1) \tag{5.5}$$

$$v_{\text{duon}} = (2,1,2,2) \tag{5.6}$$

(Note that *zeta(i,j,kstp)* is denoted as *zeta_kstp* as it is an access to the *kstp* time slice of *zeta* and *kstp* is not the iterator of one of the kernel loops (such as *i* and *j*).)

Now, to accumulate the separate stencils for every rhs arguments, consider the indices of the rhs accesses as descriptions of offsets to the accumulated input stencils. For example, when both the rhs and the lhs variable use the indices (i, j) this means the accumulated stencil of the rhs variable is applied without any offset. The lhs variable using (i, j) and the rhs variable $(i, j + 1)$ implies that the stencil is applied with an offset of 1 cell to the right. This offset is given by the dependency distance between the indices of the lhs result and a rhs argument. The dependency distance for a given index and variable (for example i) can be calculated by subtracting the index of the access to the lhs variable from the index of the access to the rhs argument. In the given example, the dependency distances (or offsets) are calculated as follows (always in relationship to the lhs result *zeta_new(i,j)*):

$$o_{\text{pm(i,j)}} = (i - i, j - j) = (0, 0) \tag{5.7}$$

$$o_{\text{pn(i,j)}} = (i - i, j - j) = (0, 0) \tag{5.8}$$

$$o_{\text{zeta\_kstp(i,j)}} = (i - i, j - j) = (0, 0) \tag{5.9}$$

$$o_{\text{dvom(i,j)}} = (i - i, j - j) = (0, 0) \tag{5.10}$$

$$o_{\text{dvom(i,j+1)}} = (i - i, (j + 1) - j) = (0, 1) \tag{5.11}$$

$$o_{\text{duon(i,j)}} = (i - i, j - j) = (0, 0) \tag{5.12}$$

$$o_{\text{duon(i+1,j)}} = ((i + 1) - i, j - j) = (1, 0) \tag{5.13}$$

Applying these offsets to the given validities is accomplished by adding or subtracting them to the corresponding values in the validity vector. So, in the example from before, for *duon(i+1,j)* an offset of $(1, 0)$ is added to the accumulated stencil of *duon* to get the component stencil for *duon(i+1,j)*. Applying an offset $o = (o^i, o^j)$ to a validity vector $v = (v^W, v^E, v^S, v^N)$ is achieved as follows:

$$v = (\max(v^W - o^i, 0), \max(v^E + o^i, 0), \max(v^S - o^j, 0), \max(v^N + o^j, 0)) = \tag{5.14}$$

$$= \max_{\text{element wise}} (v + (-o^i, o^i, -o^j, o^j), 0) \tag{5.15}$$

Visually, applying an offset to a stencil means shifting the stencils center o_i cells to the left and o_j cells down.

All these component stencils can then be combined to form a collective right-hand side (rhs) stencil. This is done by overlaying the stencils and determining the maximum extent for each dimension. This collective stencil, along with its validity, becomes the new stencil and validity of the variable on the left-hand side of the assignment. Table 5.1 shows an the validity vectors and accumulated stencils before the execution of the given example kernel, the offsets for all components, how the offset stencils (component stencils) look like and how these component stencils are added to a collective rhs stencil. The collective stencil for the presented example can be seen in Table 5.1, its validity or extent vector is $(2, 2, 2, 2)$.

Kernel: $zeta\_new(i,j) = zeta(i,j,kstp) + dtfast \cdot pm(i,j) \cdot pn(i,j) \cdot (duon(i,j) - duon(i+1,j) + dvom(i,j) - dvom(i,j+1))$

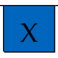
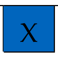
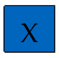
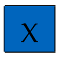
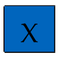
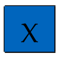
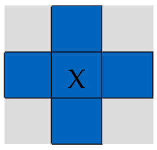
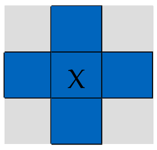
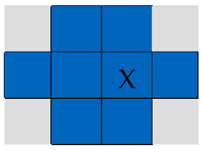
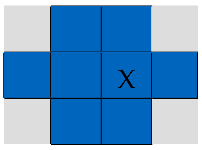
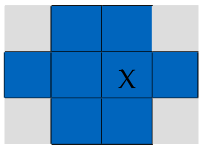
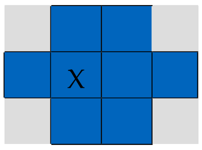
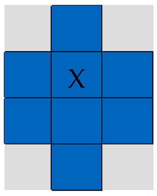
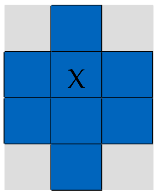
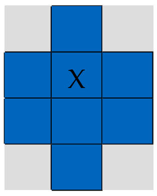
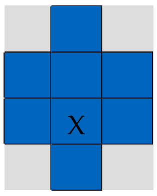
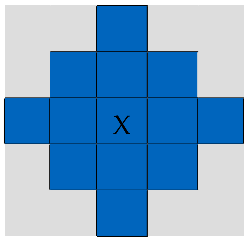
Argument	Before		Dependency Distance		After	
Access	Stencil	Validity	Index	Offset	Stencil	Validity
dtfast		$(0,0,0,0)$	-	$(0,0)$		$(0,0,0,0)$
pm(i,j)		$(0,0,0,0)$	i,j	$(0,0)$		$(0,0,0,0)$
pn(i,j)		$(0,0,0,0)$	i,j	$(0,0)$		$(0,0,0,0)$
zeta(i,j,kstp)		$(1,1,1,1)$	i,j	$(0,0)$		$(1,1,1,1)$
duon(i,j)		$(2,1,1,1)$	i,j	$(0,0)$		$(2,1,1,1)$
duon(i+1,j)		$(2,1,1,1)$	$i+1,j$	$(1,0)$ $\leftarrow$		$(1,2,1,1)$
dvom(i,j)		$(1,1,2,1)$	i,j	$(0,0)$		$(1,1,2,1)$
dvom(i,j+1)		$(1,1,2,1)$	$i,j+1$	$0,1$ $\downarrow$		$(1,1,1,2)$
Result						
zeta_new(i,j)						$(2,2,2,2)$

Table 5.1: Stencil calculation for the example kernel
$zeta\_new(i,j) = zeta(i,j,kstp) + dtfast * pm(i,j) * pn(i,j) * (duon(i,j) - duon(i+1,j) + dvom(i,j) - dvom(i,j+1))$
and the input validities given before.

For every rhs access in the kernel, a row shows the input validities and stencils (given). It then shows the offset determined from the access indices, both as an offset (or dependency distance) vector and if applicable as an arrow showing the corresponding shift direction of the stencil's center. The resulting shifted stencil (or component stencil) and its validity vector are shown on the right. "Adding" all stencils by overlaying them taking the stencils' centers into account gives the resulting accumulated stencil for *zeta_new*.

5.3.2 Stencil Calculations and Accumulation in Practice

For the extracted kernels, the stencil offsets and variable dependencies are extracted and analyzed using PSyclone.

The calculation is done as presented in the last section. For every assignment inside a kernel, each variable on the left hand side gets assigned a validity vector. In the beginning, each accumulated stencil or validity vector is initialized to 0.

Dependency Distances

Calculating the stencil offsets is done using PSyclones `_get_dependency_distance` function. This function returns the signed dependency distance between two input indices (or index terms) with respect to a given index variable. Calculating the corresponding dependency distance between an rhs argument of an assignment and it lhs result with respect to the kernel loop variables *i* and *j* results in a dependency distance vector of length 2. This calculation is done for all pairs of lhs results and corresponding rhs arguments.

The dependency vector is calculated for all types of variables - even scalars and arrays not operating on either of the *i* or *j* dimensions. This is necessary because a scalar variable in a spatial loop is automatically associated with the corresponding loop indices at every iteration. The dependency distance between a scalar and an array can be determined using the loop variables (*i,j*) as assumed indices for the scalar. Arrays that are missing an *i* or *j* component are treated the same way.

The dependency distances for a kernel are only calculated once, then the kernel stores the results in a dependency distance list for each of its assignments. This list consists of the dependency distance vectors between the lhs result and each rhs argument.

5.4 Scheduling Halo Exchanges

The extracted dependencies and stencils are later used to analyze the access stencils and find out when halo exchanges are needed.

Algorithm 1: Calculate Validities

```
Input  : assignment $a_x$ of kernel $k$, validities $V$
Output: updated validities $V$, list of rhs_component_stencils

rhs_component_stencils = {}
forall rhs arguments $r_y \in$ assignment $a_x$ do
begin
    if $r_y \notin V$ then
    begin
        $V[r_y] = (0,0,0,0)$
    end
    // offset = pre-calculated dependency distance between l_x and r_y
    $o = k$.dependency_distances$(l_x, r_y)$
    // Apply offset: Shift stencil center $o_i$ cells left, $o_j$ cells down
    composed_stencil = $V[r_y] + (-o^i, o^i, -o^j, o^j)$
    rhs_component_stencils$[r_y] = \max_{\forall d \in \{NSEW\}}$(composed_stencil$^d$, 0)
end
// To "add" all rhs stencils calculate the maximum extent of them
// for each direction d
$V(l_x) = \max_{\forall d \in \{NSEW\}}$(rhs_component_stencils$[r_y]^d$, $\forall\, r_y \in$ assignment $a_x$)
return $V$, rhs_component_stencils
```

5.4.1 Calculating Stencil Compositions

The dependency distances correspond to the offsets from the beginning of this section. This means that applying an offset to the validity vector of the corresponding rhs argument of an assignment gives the component stencil for this variable. Then the addition of all the rhs component stencils gives the new accumulated stencil and the validity vector for a variable.

The first step for this is to go through all kernels in execution order and calculate the validity vectors (or bounding box of the accumulated stencils) for all variables. This is done with Algorithm 1 using the method described in Section 5.3.1.

For each assignment, this algorithm calculates a stencil vector for the result, consisting of four values. The four values, one for every cardinal direction represent the minimal bounding box around the composition of all the component stencils for the rhs arguments of the assignment. Each value indicates how far the combined stencil representing all dependencies of an array cell reaches in one of the cardinal directions. With every

```
DO j = jmin, jmax, 1 ! Modified loop bounds
  DO i = imin + 1, imax, 1 ! Modified loop bounds
    drhs(i,j) = cff1 * zeta(i,j,kstp) + cff2 * zeta(i,j,kbak) + cff3 * zeta
        (i,j,kold) + h(i,j)
    duon(i,j) = 0.5d0 * (drhs(i,j) + drhs(i - 1,j)) * on_u(i,j) * urhs(i,j)
    dvom(i,j) = 0.5d0 * (drhs(i,j) + drhs(i,j - 1)) * om_v(i,j) * vrhs(i,j)
  ENDDO
ENDDO
```

Figure 5.7: Assume that the validity for all time slices of *zeta* at the beginning of the kernel is $(2,2,2,2)$. Calculating the accumulated stencils gives $v_{\text{drhs}} = (2,2,2,2)$ and $v_{\text{duon}} = (3,2,2,2)$ with the component with the stencil causing the 3 being *drhs*. Nevertheless exchanging *drhs* or *duon* before the kernel does solve not the problem as their stencils only grow this large during the kernel. It is zeta that is causing the problem and needs to be exchanged. (The code sample is a modified extract from the *step2d* subroutine of CROCO's BASIN test case.)

assignment, the dependencies and stencils are further propagated to new variables.

5.4.2 Identifying Necessary Halo Exchanges

Section 5.3.1 showed that halo exchanges are necessary whenever an access stencil grows larger than the number of halo layers or, differently phrased, whenever the dimensions of the bounding box of the accumulated stencil of an assignment are larger than the number of halo layers. Then a halo exchange is needed for some variable to reset its validity to 0. The next step is determining which variable needs to be exchanged as it is not enough to just do a halo exchange for any variable with an access stencil that has grown too large. It is not possible to just execute halo exchanges in the middle of a kernel block as communication operations can only take place before and afterwards. Doing a halo exchange for a variable before a kernel block might not achieve the goal of keeping its accumulated stencil small enough if for example the corresponding variable is overwritten at the start of the kernel. The code sample in Figure 5.7 shows an example for this.

For every access stencil in a kernel that grows to large it is therefore necessary to backtrace it to the beginning of the kernel to determine which of its components caused it to grow this large. The arrays associated with these components need to be exchanged before the kernel.

Determining when and which halo exchanges are necessary is done with Algorithm 2. It determines a set of kernel-variable tuples that list all necessary halo exchanges for variables and the kernels before which the halo exchanges need to be done.

Algorithm 2: Determine Halo Exchanges

```
Input    : List of kernels in execution order ordered_kernels
Output   : List of tuples $(k, v)$ of all exchanges of a variable $v$ necessary before a
           kernel $k$

validities ← {}
results ← {}

forall kernel $k \in$ ordered_kernels do
begin
    for assignment $A_i \in k$ with LHS variable $l_i$ do
    begin
        $v_{l_i}$, rhs_components ← calculate_validities($v_{before}$, $A_i$)
        validities[$l_i$] = $v_{l_i}$

        // If a stencil is too large a halo exchange needs to happen
        if $v_{l_i}^d >$ num_layers for any direction $d$ then
        begin
            transitive_causes$^d$ ← {$l_i$}

            // Trace back assignments through the ones before them
            for assignment $A_j \in [A_i ... A_0] of this$ kernel$k$, with LHS variable $l_j$ do
            begin
                if $l_j \in$ transitive_causes$^d$ then
                begin
                    transitive_causes$^d$ ← transitive_causes$^d$ − {$l_j$}

                    // Determine variables with stencils with
                    // causal influence on the validity $v_{l_j}^d$
                    for $r_k \in$ get_rhs_arguments($A_j$) do
                    begin

                        // look at component stencil of $r_k$
                        if rhs_components[$r_k$]$^d >$ num_layers then
                        begin
                            causal_influences ← causal_influences + {$r_k$}
                        end
                    end
                    transitive_causes ← transitive_causes + causal_influences
                end

                // transitive_causes caused the too large stencil earlier
                for $x \in$ transitive_causes do
                begin
                    results ← results + {($x$, $k$)}
                    $v_x = (0, 0, 0, 0)$                    // Reset validity
                end
            end
            Jump to beginning of kernel to calculate validity vectors with new values
        end
        else
        begin
            Go to next assignment
        end
    end
end
```

5.4.3 Determining the Time Window for Communication

While the synchronous halo exchanges are executed and can be scheduled at one specific point in time, the communication for the asynchronous halo exchanges is executed during a window of time. For every variable - kernel pair that was determined in the step before, this window starts after the last write access on this variable before the kernel end ends at the start of the kernel. The code that initializes the communication needs to be inserted at the beginning of the time window.The code that terminates the communication for the halo exchanges needs to be inserted at the end of the time window and therefore just before this kernel whose read access made the halo exchange necessary.

5.5 Time Stepping

As discussed in Chapter 2, CROCO's main step loop is divided into four substeps. The parts within the slow mode calculation are executed once per step, while the *step2d* part is executed *nfast* times. Separate counters are used for these two loops. In the fast mode, information from time steps $t-1$, $t-2$, and $t-3$ is used to calculate new values at time t, resulting in arrays with a time dimension of size four. The slow mode references only two other time steps, and the corresponding arrays have a time dimension of three.

5.5.1 Time Indexing Throughout Substeps

The time indices for the current and previous time steps are incremented by one at each step to move to the next time step. They reset to one when they grow larger than the length of the time dimension, which is either 3 or 4. This shifting of time indices ensures that the array slice representing the present time in the previous time step becomes the array slice for the past time in the current step.

However, the time indexing process is not entirely consistent. While time indices are increased once at the beginning of each time step, different substeps may use different time indices for the same variables. Additionally, some substeps modify these time indices during their execution. Aliasing further complicates the process, causing differently named indices to refer to the same time slice in some time steps but not in others. For referencing the three different time steps used in the slow mode for example, the code uses five index variables. Figure illustrates the time index modifications and aliases throughout the step loop of the BASIN test case.

When such a change to a time index happens, it is important to also update the validities accordingly to ensure all values are transferred to the corresponding new time slice of an array. This is done using connectors in the way explained in Section 5.2.2. So whenever one or multiple time indices are modified, this is representded by a connector kernel.

The code modifying the time indices and calling the four subcomponent's subroutines of *step* is relatively complex, being nested among loops and conditional statements, and

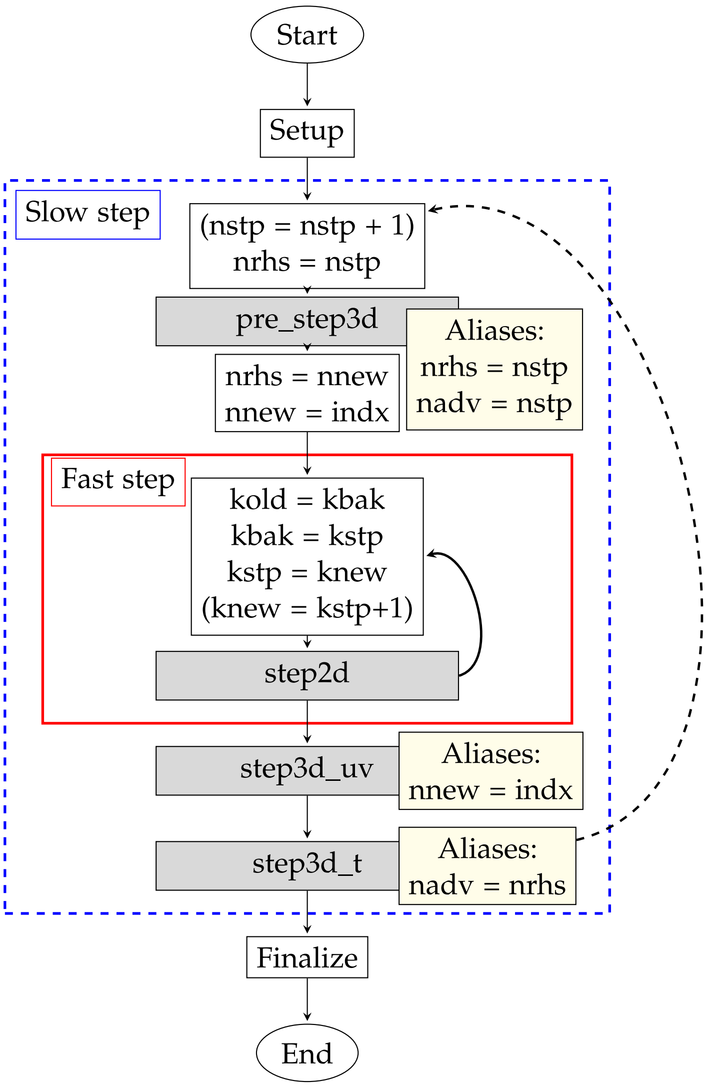


Figure 5.8: The time stepping loop and the time index modifications that were used here. The dashed arrow symbolizes the slow step loop which is not considered in the time stepping model. Assignments in parentheses are redundant and only shown for better understanding.

```
if substep == 0:
    aliases = {"nrhs": "nstp", "nadv": "nstp"}
    reassignments_start = [("nrhs", "nstp")]
    reassignments_end = [("nrhs", "nnew"), ("nnew", "indx")]
elif substep == 1:
    aliases = {}
    reassignments_start = [("kold", "kbak"), ("kbak", "kstp"), ("kstp", "
        knew")]
elif substep == 2:
    aliases = {"indx":"nnew"}
elif substep == 3:
    aliases = {"nadv": "nrhs"}
```

Figure 5.9: Python code containing information about the time indices. *aliases* contains pairs of indices that are aliases, both reassignment lists contain tuples (lhs, rhs) representing the timestep reassignments $\text{lhs} = \text{rhs}$. *reassignments_start* is applied before a substep's subroutine, *reassignment_end* afterwards. The four substeps have indices 0 to 3 in execution order.

spread out over multiple subroutines. This makes it very hard to automatically process in a way that accurately and usefully captures the dependencies and relationships of the counters, indices and subroutine calls over multiple loop iterations. This needs to be partially done by the user, who inputs a manually created list of assignments for the time indices and their locations.

Fortunately, most of these changes occur at the beginning or end of a substep. Therefore, this is solved by the user providing a list of assignments that need to be implemented by connector kernels at the start and end of each substep. Figure 5.9 shows such a list, representing the modifications of time indices throughout the step loop of the BASIN test case.

Splitting up the Time Step

Because of the differences in time indices, the *step* subroutine is not analyzed in one single piece, and instead, each of its four subroutines is analyzed individually. This ensures that within each of the parts, time indexing is consistent and does not change.

Additionally, communication can not be spread over multiple different parts. This means that every time window for a halo exchange has to be located in one part entirely and can not start and end in different ones.

Time Indexing Patch

At certain times, the *step2d* subroutine indexes arrays' time slices in the following way:

```
rufrc(i,j)=cff1*cff + cff2*rufrc_bak(i,j,3-nstp)
```

Automatically linking the $3 - nstp$ to the equivalent time index (in this case *nnew*) would again be very difficult. These array accesses are therefore replaced with equivalent ones using simple time indices. In this case, the result would be:

```
rufrc(i,j)=cff1*cff + cff2*rufrc_bak(i,j,nnew)
```

This is implemented using a patch and manually determining the appropriate indices.

5.5.2 Modeling the Step Loops

As the steps are executed in a loop, this needs to be modelled, too.

Outer Slow Step Loop

For this, only one slow 3D time step at a time will be considered. While it would be possible to consider data dependencies over multiple time steps in a row, for example by unrolling the main step loop, doing so becomes very complex rather quickly. Each unrolled iteration works on different input stencils and can therefore require different communication operations.

Again, the timestep reassignment is too complex to automatically be usefully captured and analyzed. Therefore an automatic dependency analysis over multiple time steps was not possible.

The presence of the inner substep loop also complicates things as it would necessitate unrolling both loops simultaneously, making everything even more difficult. In this case all validities and operations would be dependent on the iteration index of each of those loops. Unrolling the inner loop by x iterations and the outer loop by y means there are potentially x different cases. As creating code implementing the communication for all these different cases would be very complex, this is out of the scope of this thesis.

Fast Step Loop

As a proof of concept, the inner fast mode loop is unrolled into *n_layers* (the number of halo layers) iterations. This does - in theory - make it possible to update a variable only every second iteration. Additionally, communication windows are allowed to be spread over different iterations of step2d loop, here. This means that, for example, a communication window can start in the first iteration and end in the second one, increasing the potential length of this window.

Periodic Resets

To ensure that every iteration of a loop works on identical input stencils and dependencies, it is necessary to do halo exchanges before the and of a substep. This means that, in addition to halo exchanges triggered when accumulated stencils exceed the halo size, some halo exchanges are scheduled at the end of a substep. This ensures that a substep or the next time step starts with "freshly exchanged" halos, allowing all stencils to be initialized with 0.

To minimize the frequency of these halo exchanges, they are only executed for variables that are not overwritten before being read the next time.

5.6 Limitations

The presented approach works under a row of assumptions and has a certain set of limitations.

It assumes that all kernel loops use i and j as control variables which should however always be true in CROCO's code. Additionally it ignores everything happening outside of kernels. Information on things happening outside of kernels (apart from subroutine calls) currently needs to be manually collected given as a program input. This is especially true for everything regarding time indices and the main time stepping loop.

It is assumed that assignments in the same kernel have no additional dependencies among them that need to be considered. The approach to calculating stencil extends assumes that the center of a stencil is always part of the stencil and its minimal bounding box. While this should generally be the case, if it is not the approach overestimates the stencil extend, possibly causing unnecessary halo exchanges. It does, however, never underestimate it which would lead to using fewer halo exchanges than needed and incorrect simulation results.

The approach does also not guarantee a minimal number of halo exchanges even if the challenges with time stepping are ignored. It uses a greedy approach and schedules halo exchanges once they become necessary (because an accumulated stencil grows too large). In some cases however, using an additional halo exchange for one variable can lead to a reduction in halo exchanges of other variables that are dependent on the first one.

The presented approach also ignores any `if` and `goto` constructs and assumes every piece of code is executed every time. This could easily cause problems as deviations in control flow can change which kernels are executed when and therefore influence stencils and communication schedule. However it did not cause any problems in this case.

6 Implementation II - Generating Asynchronous MPI Code

The scheduling step determines when halo exchanges are necessary and identifies the beginning and end of the time window for an asynchronous halo exchange. This uses these results to generate code that implements this communication pattern.

6.1 Communication Scheme

The asynchronous halo exchanges are implemented using the non-blocking *MPI_IRECV* and *MPI_ISEND* calls instead of their blocking counterparts that are used in the original CROCO code. This allows a process to initiate the sending process and continue computation while the communication is happening. This helps to to hide the communication latency compared to having to wait for the entire send and receive operation to finish and the blocking call to terminate before continuing computation. Both non-blocking calls need corresponding *MPI_WAIT* call for completion. The *MPI_WAIT* calls for *MPI_ISEND* ensures the message has been sent, allowing safe modification of the data in the send buffer. Conversely, the *MPI_WAIT* call for *MPI_IRECV* confirms the message's receipt and provides access to the received data in the receive buffer.

```
...
write access on var
irecv
copy to send buffer
isend
...
wait for isend
wait for irecv
copy from recv buffer
read access on var
...
```

Figure 6.1: Communication scheme for asynchronous halo exchanges.

6.1.1 Communication Implementation

Figure 6.2 shows the start of the asynchronous halo exchange. After initializing an non-blocking receive, the data is copied from the array into a one-dimensional buffer array that is then passed as an argument to the non-blocking send operation. An example of how this is implemented is shown in Figure 6.3.

```
...
! start communication here
if (has neighbor in this direction)
     then
    MPI_IRECV from this neighbor,
        to receive buffer
end if

if (has neighbor in this direction)
     then
    copy data to send buffer
    MPI_ISEND from send buffer, to
        this neighbor
end if
```

```
...
! end communication here
if (has neighbor in this direction)
     then
    MPI_WAIT for MPI_IRECV to
        terminate
    copy data from receive buffer
        to array halo
    MPI_WAIT for MPI_ISEND to
        terminate
```

Figure 6.2: Left: Pseudocode describing the copy, send and receive operations used at the beginning of the communication window.
Right: Pseudocode describing the wait and copy operations at the end of the communication window.

After initiating the non-blocking send process can return to computation without having to wait for the data to be transmitted first.

The transmission will ideally happen in the background during computation until the code arrives at the end of the communication window. There one *MPI_WAIT* finalizes the send operation, waiting until the send has been completed and another one finalizes the receive operation, waiting until all data has been received. The received data is then copied ffrom the buffer to the corresponding array's halo cells. Figure 6.4 shows an implementation example for this.

6.2 Communication Implementation Details

This section gives further details on how this communication is implemented and how the corresponding code is generated automatically.

```
! Receive
IF (north_inter2) then
  CALL MPI_IRECV(vbar_knew_buf_rev_n, size_x, MPI_DOUBLE_PRECISION, p_n, 1,
       MPI_COMM_WORLD, vbar_knew_req(4), ierr)
end if
! other directions
...

! Copy to buffer and send
if (north_inter2) then
    DO jpts = 1, npts, 1
      DO i = imin_mpi, imax_mpi, 1
        vbar_knew_buf_snd_n(1 + (i - imin_mpi) + (1 + (imax_mpi - imin_mpi)
            ) * (jpts - 1)) = vbari,jpts + mmmpi - npts,knew)
      ENDDO
    ENDDO
    call MPI_ISEND(vbar_knew_buf_snd_n, size_x, MPI_DOUBLE_PRECISION, p_n,
        3, MPI_COMM_WORLD, bar_knew_sreq(4), ierr)
END IF
! other directions
...
```

Figure 6.3: Code example implementing the beginning of a halo exchange with a tile's southern neighbor.

```
...
! wait for receive to terminate
IF (north_inter2) then
  CALL MPI_WAIT(vbar_knew_req(4), MPI_STATUS_IGNORE, ierr)
  DO jpts = 1, npts, 1
    DO i = imin_mpi, imax_mpi, 1
      vbar(i,jpts + mmmpi,knew) = vbar_knew_buf_rev_n(1 + (i - imin_mpi) +
          (1 + (imax_mpi - imin_mpi)) * (jpts - 1))
    ENDDO
  ENDDO
ELSE
  ! if there is no neighbor in that direction fill the halo with the
      values on the inside next to it
  IF (north_inter) then
    DO jpts = 1, npts, 1
      DO i = imin_mpi, imax_mpi, 1
        vbar(i,jpts + mmmpi,knew) = vbar(i,jpts + mmmpi - npts,knew)
      ENDDO
    ENDDO
  END IF
END IF

! wait for send to terminate
IF (north_inter2) then
    CALL MPI_WAIT(vbar_knew_sreq(4), MPI_STATUS_IGNORE, ierr)
END IF
```

Figure 6.4: Code implementing the termination of the halo exchange with a tile's northern neighbor.

6.2.1 Additional Send and Receive Buffers

The data comprising one of the eight directional parts of a subdomains halo is stored in multiple noncontiguous array cells. For sending and receiving it is first linearized and copied into a one-dimensional contiguous buffer, similarly to CROCO's original code (see Section 2.3.1).

In contrast to the original code, which can only linearize and exchange arrays or array slices with two or three dimensions (corresponding to the three spatial dimensions), the implemented approach can automatically generate code to linearize and communicate arrays with more and different dimensions.

In CROCO code this can be used for arrays that have an additional dimension for the different tracers (for example *t* which has five dimensions: i, j, k (space), time and tracers). In the original code, the halo exchanges for *t* (or more specifically for a four dimensional time-slice of it) are handeled by doing one single and separate halo exchange for every 3D slice of the 4D array that corresponds to a tracer. In the BASIN example which uses two tracers, for example, exchanges for *t* are implemented by using one 3D halo exchange for each tracer's slice of the array. However, in the implemented approach, these exchanges are combined into a single '4D' exchange. This reduces the amount of messages that are necessary, thereby potentially reducing communication overhead.

In the case of communication windows spanning multiple time steps in the fast time loop, it is important to consider that the array slice that is exchanged at the start of the communication window has a time different index than the same slice at the end of the window. It is necessary to adapt the second copy operation accordingly to copy the data back to the time slice using the new index.

Array Dimension Analysis for MPI Calls

To allocate temporary buffers of the correct size and correctly copy the halo data to and from them, both the sizes of the dimensions of an array and information on how the data inside is typically accessed is needed. The array sizes can be extracted from the corresponding the PSyIR ArrayReference. The information in Table 6.1 is then used to determine if an array dimension is accessed as a whole using a loop (for all three spatial dimensions and tracers), or just one slice at a time (for time and weight dimensions). For each dimension that is accessed fully, a correctly sized loop is generated automatically to copy data into and from the buffer. Additionally this data is used to automatically create code for handling periodic boundary conditions if necessary.

6.2.2 Neighbors

In the case of physical boundary conditions, subdomains at the edges do not have a neighbor in certain direction to exchange halo values with. This is handeled in the same way the original CROCO code handles these cases, by testing the relevant flags (e.g. *north_inter2*) and doing a halo exchange only if a communication partner exists. The

Table 6.1: Array access details.

Array dimension	Loop variable	Dimension size
xi-dimension	*i*	*size_x*
eta-dimension	*j*	*size_e*
third (vertical) spatial dimension	*k*	1 - *N* / 0 - *N*
time	none (accessed one slice at a time)	1 - 3 / 1 - 4
weight	none (accessed one slice at a time)	1 - 5
tracers	*itrc*	1 - *NT*

rank of the subprocess to communicate with is also copied from the original CROCO code (e.g. *p_N* for the northern neighbor).

6.2.3 Variables, Symbol Tables and Bounds Computations

Some variables that are needed for making the MPI calls are not declared in the subroutines where MPI calls need to be inserted. PSyclone needs to know about variables, symbols and their properties for code generation. Sometimes, it becomes necessary to introduce a variable from one subroutine in one file into another. To access important variables for iteration ranges, array indices, and other purposes, a patch is used to generate a file, that is used as a lookup table. This file contains a list of subroutines and variables, along with flags for the C preprocessor and header inclusions. After preprocessing, this file's PSyIR representation can be used to easily access the PSyIR symbols for the included variables. Whenever there's a need to add variables or compute iteration ranges this information is duplicated and inserted into other files.

Common Variables

Because of register optimization applied by the Fortran compiler, non-blocking MPI calls don't always work as intendent [25]. The compiler works under the assumption that subroutine calls do not influence the data in variables that are neither common variables nor arguments of the subroutine. In this case it will assume that calling a wait operation for a receive like

```
CALL MPI_WAIT(vbar_knew_req(4), MPI_STATUS_IGNORE, ierr)
```

from the code example before does not influence the array *vbar_knew_buf_rev_n* if it is a local array. Consequently it will use this assumption during compilation and possibly modify register usage and operation order accordingly. In this case, however this assumption is simply not true and the optimization can lead to incorrect results.

To address this problem, one of the solutions recommended by the MPI standard [25] involves making the buffers global variables. This causes the compiler to drop the assumption that the values in *vbar_knew_buf_rev_n* can not change during the wait.

However, PSyclone does not provide an option to implement this directly. Therefore, a workaround is necessary, which involves adding the global declaration to a subroutine's symbol table pretending the entire declaration is the name of one variable. Additionally, request buffers must also be made global since they are accessed in different subroutines.

6.2.4 Kernels Occuring Multiple Times

Multiple subroutines and kernels they consist of are called multiple times during program execution. They can also be called during different execution phases (e.g. setup and main time stepping phase) or with different arguments. These different phases will have different needs for communication, a halo exchange that is necessary at one point might not be necessary at another or at one point it might be necessary to terminate (wait) a halo exchange that was initiated before, but it actually wasn't initiated during every call of a subroutine. Therefore, the generated halo exchange instructions are surrounded by if instructions referencing multiple of CROCO's counter variables to make sure they are only executed during the correct program phase. An example for this can be seen in Figure 6.5. The asynchronous halo exchanges are also only generated for the main time stepping loop. For the setup phase CROCO's original halo exchanges are still used. Section 5.5 created the need for checking if a statement is executed in the first, last or in general which iteration of a loop. The indicators and counter variables in Table 6.2 are used for that.

Table 6.2: A list of indexes and indicators useful for identifying the current code phase or the iteration index (counter) of the current loop.

Code Phase	Code Phase Indicator	Substep Counter	Global Iteration Index
setup	$iic == 0$	-	-
pre_step3D	$iif == 0$	-	*iic* (*ntstart* to *ntstart* + *ntimes*), *nbstep3d* (0 to *ntimes*)
step2D	$iif > 0$ .and. $(iif - nfast) == 0$	*iif* (1 to *nfast*)	*iic*, *nbstep3d*
step3d_uv	$iif - nfast == 1$	-	*iic*, *nbstep3d*
step3d_t	$iif - nfast == 2$	-	*iic*, *nbstep3d*

6.2.5 Periodic Boundary Conditions

In the original CROCO code the exchange subroutines call the *MessPass* subroutines that implement the MPI calls for the halo exchanges. These subroutines also implement setting periodic boundary conditions if applicable. Whenever the original code calls an *exchange* subroutine, the asynchronous code variant replaces this by a call to a corresponding subroutine *ex_periodic_bc* as shown in Figure 6.5. These are copies of the *exchange* subroutines that only implement the boundary conditions but do not call

MessPass that contains the synchronous halo exchange. Alternatively these subroutines can be generated fully automatic

```
! only execute exchange if this is called during setup phase
IF (iic == 0) THEN
   CALL exchange_v2d_tile(istr, iend, jstr, jend, vbar(-1,-1,knew))
! otherwise only set boundary conditions
ELSE
   CALL ex_pbc_v2d_tile(istr, iend, jstr, jend, vbar(-1,-1,knew))
END IF
```

Figure 6.5: Replacing exchange calls by a subroutine *ex_pbc_tile* that implements periodic boundary conditions. The if condition ensures that the original exchange will be used if the code is executed during setup before time stepping.

7 Evaluation

This chapter covers testing and evaluation, with the aim of evaluating the generation process, the code itself and its performance for both generated and original source code.

7.1 Evaluation Setup and Experiments

Testing was carried out using the CROCO BASIN test case, utilizing the previously described input settings.

7.1.1 Test Case Setup

To ensure accurate numerical results, a suitable timestep size was chosen, small enough to work with all problem sizes without causing numerical instability. The number of fast timesteps during one slow timestep was set to 66, which is close to the original 65 and numerically stable but fullfills the additional requirement of being a multiple of the number of iterations of the fast step loop that are unrolled during dependency analysis as specified in Section 5.5. For performance analysis across various problem sizes, different versions of the BASIN test case were created with grid dimensions ranging from 24 to 240 cells in both the *xi* and *eta* dimension. These changes were implemented using the corresponding patches mentioned in Section 4.2.2. For benchmarking, printing (intermediate) results and diagnostics were both switched off.

7.1.2 Code Versions for Comparison

The input code for stencil analysis and code generation was derived from a patched version of the original CROCO MPI code, as outlined in Section 4.2.2. This patched variant of the original code was used as the baseline for all comparisons ensuring a fair evaluation.

In the previous chapters, two key aspects to generate code for CROCO's halo exchanges were discussed: the identification of when halo exchanges are necessary (as detailed in Chapter 5) and the implementation of these exchanges using asynchronous MPI operations (as detailed in Chapter 6). The evaluation will to analyze and compare the results from using these two approaches and the baseline code. The baseline was therefore compared to two other code variants:

- Reduced Halo Exchanges (*reduced-mpi*) This variant applies the approach described in Chapter 5 to determine when halo exchanges are necessary. Then CROCO's original synchronous exchange operations are inserted at the identified locations. This test the automatic scheduling of halo exchanges without changing the communication mode to asynchronous communication.

- Autogenerated Asynchronous Halo Exchanges (*rposeidon-mpi-async*) Similar to the first variant, this uses the approach outlined in Chapter 5 for determining necessary halo exchanges. However, it uses the method from Chapter 6 to automatically generate code for these asynchronous halo exchanges. This tests the combination of automatically scheduling halo exchanges and their asynchronous implementation.

As explained in Section 5.5.2, additional halo exchanges are required at the end of each substep in the main step loop to ensure consistent initial stencils for all iterations. These exchanges are restricted to variables that are also exchanged in the original code, for both automatically generated code variants. This uses information extracted from the original code to avoiding resetting temporary working arrays. Most importantly, it improves the comparability of performance across all three versions, as they are more likely to involve the same exchanges, with variations primarily occurring in implementation and location. This means it is possible to compare the impact of both changes in location and number and changes from synchronous to asynchronous and from the original to the automatically generated code.

It's worth noting that the original CROCO code appears to follow a similar principle. For each use case, a unique version of the code is generated by the pre-processor. To ensure proper functioning and an appropriate number of halo exchanges for all of them, it is reasonable to assume that the original code follows a rule where substeps must be implemented in such a way that certain variables require a halo exchange at the end of a subroutine if the halo is modified.

7.1.3 Technical Setup

Benchmarking was conducted using the Grid'5000 testbed [26][1] , on a Dell PowerEdge C6420 with Intel Xeon Gold 6130 processors. These processors have 16 cores per CPU and a clock speed of 2.10 GHz. The network infrastructure used was Omni-Path configured at a rate of 100 Gbps, enabled by Intel Omni-Path HFI Silicon 100 Series hardware.

Compilation was done using the gfortran compiler.Open MPI 4.1.0 was used as MPI implementation.

This means that the code was executed binding each process to one processor core and using the omnipath interconnects.

[1]Experiments presented in this section were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

7.1.4 Benchmarking Experiments

To assess code performance, scaling experiments were conducted, spanning a range of problem sizes and varying the number of processors or cores. Each experiment was conducted on 2, 4, 8, 16, and 32 cores, corresponding to the number of MPI processes and associated subdomains. The simulation was executed for a number of steps that was constant but scaled by problem size to avoid excessive run times.

Runtime was measured using Python's timeit module to capture the execution time for running the code using the shell command `mpirun ./croco <options>`. Each measurement was repeated 10 times for accuracy.

7.1.5 Correctness of Results

All simulation results were validated by comparison with the output produced by sequential computation. A result was considered correct if it was either identical or within a relative tolerance of 10^{-9} of the reference output.

8 Results

This section compares the results of code generation and benchmarking for the original code and the two versions of the automatically generated code.

8.1 Code Generation

The prototype using the presented approaches analyzed the content of 21 files containing code that is a part of the main time-stepping loop. During code generation, it created a modified variant of 8 of them, removing the original halo exchanges and creating new, asynchronous ones.

8.1.1 Number and Location of Exchanges

This section compares the halo exchanges for the BASIN test case in the original CROCO code, and the code version using CROCO's original halo exchanges in automatically determined locations (*reduced-mpi*).

The original code used 45 halo exchanges, with 3 occurring within the step2d subroutine implementing the fast step and the remaining 42 occurring in other subroutines. These 45 halo exchanges were replaced by only 19 automatically generated exchanges. This substantial reduction indicates that many original exchanges were not strictly necessary and thus not recreated automatically. Comparing the halo exchanges in both code variants (see Table 8.1 and Table 8.2) shows how none of the exchanges in the *set_vbc_tile*, *set_depth_tile* or *omega_tile* subroutines also occur in the auto-generated code version. These exchanges are redundant in the original code. They probably exist because they are needed in different use cases using additional model features or numerical schemes but are not useful in this case.

Table 8.1: List of automatically created synchronous halo exchanges for each of the four sections in execution order. Stencil extents (validities) at exchange are shown next to the variable names.

	Reduced Exchanges			
Section	List of exchanges	2D exchanges	3D exchanges	'4D' exchanges
pre_step3D	set_HUV_tile		huon (1,0,0,0)	t_nnew (2,0,2,0)
	exchange_u3d_tile(istr, iend, jstr, jend, huon(-1,-1,1))		hvom (0,0,1,0)	
	exchange_v3d_tile(istr, iend, jstr, jend, hvom(-1,-1,1))		u_nnew (2,0,1,1)	
			v_nnew (1,1,2,0)	
	pre_step3d_tile		u_indx (1,0,1,0)	
	exchange_u3d_tile(istr, iend, jstr, jend, u(-1,-1,1,nnew))		v_indx (1,0,1,0)	
	exchange_v3d_tile(istr, iend, jstr, jend, v(-1,-1,1,nnew))			
	exchange_r3d_tile(istr, iend, jstr, jend, t(-1,-1,1,nnew,itrc))			
	uv3dmix_tile			
	exchange_u3d_tile(istr, iend, jstr, jend, u(-1,-1,1,indx))			
	exchange_v3d_tile(istr, iend, jstr, jend, v(-1,-1,1,indx))			
step2D	step2d_FB_tile	zeta_knew (0,1,0,1)		
	exchange_r2d_tile(istr, iend, jstr, jend, zeta(-1,-1,knew))	ubar_knew (1,1,1,1)		
	exchange_u2d_tile(istr, iend, jstr, jend, ubar(-1,-1,knew))	vbar_knew (1,1,1,1)		
	exchange_v2d_tile(istr, iend, jstr, jend, vbar(-1,-1,knew))			
step3D_uv	set_HUV2_tile	ubar_knew (2,2,2,2)	u_nrhs (1,1,1,1)	
	exchange_u3d_tile(istr, iend, jstr, jend, u(-1,-1,1,nrhs))	vbar_knew (2,2,2,2	v_nrhs (1,1,1,1)	
	exchange_v3d_tile(istr, iend, jstr, jend, v(-1,-1,1,nrhs))		u_nnew (2,2,2,2)	
			v_nnew (2,2,2,2)	
	step3d_uv2_tile		huon (2,2,2,2)	
	exchange_u3d_tile(istr, iend, jstr, jend, u(-1,-1,1,nnew))		hvom (2,2,2,2)	
	exchange_v3d_tile(istr, iend, jstr, jend, v(-1,-1,1,nnew))			
	exchange_u2d_tile(istr, iend, jstr, jend, ubar(-1,-1,knew))			
	exchange_v2d_tile(istr, iend, jstr, jend, vbar(-1,-1,knew))			
	exchange_u3d_tile(istr, iend, jstr, jend, huon(-1,-1,1))			
	exchange_v3d_tile(istr, iend, jstr, jend, hvom(-1,-1,1))			
step3D_t	t3dmix			t__nnew (1,0,1,0)
	exchange_r3d_tile(istr, iend, jstr, jend, t(-1,-1,1,nnew,itrc))			
Sum (slow / fast)		2 / 3	12 / 0	2 / 0

Some variables, like the tracer field *t* and the transports *huon* and *hvom*, are exchanged in both versions but need fewer exchanges in the automatically generated code. Most of the halo exchanges that happen in both code variants are located in the same subroutines.

The auto-generated code also performs a halo exchange of both *u_indx* and *v_indx* (the *indx* time slice of *u* and *v*) at the end of *pre_step3d* that the original code does not need. Checking the results of a code variant with those two exchanges manually removed proves that these exchanges are in fact unnecessary, as removing them does not change the simulation results. They are only created because of the time stepping model described in Section 5.5 to reset the accumulated stencils at the end of the *pre_step3d* section to be able to guaranuee consistent input stencils for all substeps. These variables are reused as *u_nnew* and *v_nnew* in the *step3d_uv* substep before the next time step and exchanged again before its end. The first exchange is therefore not strictly necessary and only an artifact of the division of each step into the four substeps to handle the time stepping. This means that the way the analysis of the main time-stepping loop and the time indices are implemented causes the generation of additional and non-necessary halo exchanges for this test case.

Table 8.2: List of original synchronous halo exchanges for each of the four sections in execution order.

	Original code			
Section	List of exchanges	2D exchanges	3D exchanges	'4D' exchanges
pre_step3D	set_HUV_tile	bustr,	huon,	t_nnew
	exchange_u3d_tile(istr, iend, jstr, jend, huon(-1,-1,1))	bvstr,	hvom	
	exchange_v3d_tile(istr, iend, jstr, jend, hvom(-1,-1,1))	sustr,		
		svstr,	we	
	set_vbc_tile	stflx_itemp,	u_nnew	
	exchange_u2d_tile(istr, iend, jstr, jend, bustr)	srflx,	v_nnew	
	exchange_v2d_tile(istr, iend, jstr, jend, bvstr)	btflx_itemp		
	exchange_u2d_tile(istr, iend, jstr, jend, sustr(-1,-1))			
	exchange_v2d_tile(istr, iend, jstr, jend, svstr(-1,-1))			
	exchange_r2d_tile(istr, iend, jstr, jend, stflx(-1,-1,itemp))			
	exchange_r2d_tile(istr, iend, jstr, jend, srflx(-1,-1))			
	exchange_r2d_tile(istr, iend, jstr, jend, btflx(-1,-1,itemp))			
	omega_tile			
	exchange_w3d_tile(istr, iend, jstr, jend, we(-1,-1,0))			
	pre_step3D_tile			
	exchange_r3d_tile(istr, iend, jstr, jend, t(-1,-1,1,nnew,itrc))			
	exchange_u3d_tile(istr, iend, jstr, jend, u(-1,-1,1,nnew))			
	exchange_v3d_tile(istr, iend, jstr, jend, v(-1,-1,1,nnew))			
step2D	step2d_FB_tile	zeta_knew,		
	exchange_r2d_tile(istr, iend, jstr, jend, zeta(-1,-1,knew))	ubar_knew,		
	exchange_u2d_tile(istr, iend, jstr, jend, ubar(-1,-1,knew))	vbar_knew		
	exchange_v2d_tile(istr, iend, jstr, jend, vbar(-1,-1,knew))			
step3D_uv	set_depth_tile	zt_avg1	z_w, z_r,	
	exchange_r2d_tile(istr, iend, jstr, jend, zt_avg1)		hz,	
	exchange_w3d_tile(istr, iend, jstr, jend, z_w(-1,-1,0))		hz_bak	
	exchange_r3d_tile(istr, iend, jstr, jend, z_r(-1,-1,1))	ubar_knew,	huon,	
	exchange_r3d_tile(istr, iend, jstr, jend, hz(-1,-1,1))	vbar_knew	hvom,	
	exchange_r3d_tile(istr, iend, jstr, jend, hz_bak(-1,-1,1))		u_nrhs,	
			v_nrhs	
	set_huv2_tile		we	
	exchange_u3d_tile(istr, iend, jstr, jend, huon(-1,-1,1))		u_nnew,	
	exchange_v3d_tile(istr, iend, jstr, jend, hvom(-1,-1,1))		v_nnew,	
	exchange_u3d_tile(istr, iend, jstr, jend, u(-1,-1,1,nrhs))		huon, hvom	
	exchange_v3d_tile(istr, iend, jstr, jend, v(-1,-1,1,nrhs))			
	omega_tile			
	exchange_w3d_tile(istr, iend, jstr, jend, we(-1,-1,0))			
	step3d_uv2_tile			
	exchange_u3d_tile(istr, iend, jstr, jend, u(-1,-1,1,nnew))			
	exchange_v3d_tile(istr, iend, jstr, jend, v(-1,-1,1,nnew))			
	exchange_u3d_tile(istr, iend, jstr, jend, huon(-1,-1,1))			
	exchange_v3d_tile(istr, iend, jstr, jend, hvom(-1,-1,1))			
	exchange_u2d_tile(istr, iend, jstr, jend, ubar(-1,-1,knew))			
	exchange_v2d_tile(istr, iend, jstr, jend, vbar(-1,-1,knew))			
step3D_t	omega_tile		we	t_nnew
	exchange_w3d_tile(istr, iend, jstr, jend, we(-1,-1,0))			
				t_nnew
	step3d_t_tile			
	exchange_r3d_tile(istr, iend, jstr, jend, t(-1,-1,1,nnew,itrc))			
	t3dmix_tile			
	exchange_r3d_tile(istr, iend, jstr, jend, t(-1,-1,1,nnew,itrc))			
Sum (slow / fast)		10 / 3	19 / 0	3 / 0

Still, the automatically generated code versions use less than half the number of halo exchanges in the slow mode parts of a step while needing the same number in the fast step (*step2d*).

8.1.2 Asynchronous Exchanges

In addition to the number of halo exchanges, their locations are also relevant. For asynchronous halo exchanges, the positions of the operations that initialize or finalize a halo exchange and their distances are of particular interest.

As explained before, these exchanges were implemented by determining a time window during which they need to be executed and then initiating the exchange at the start of this window and finalizing it at the end. Figures 8.1, 8.2, 8.3 and 8.4 illustrate kernels and communication operations in all four main parts of step. They show all kernels (but no connectors) with the execution order indicated in red and the communication operations as additional nodes in yellow and blue.

Section 5.5.2 described how parts of the code that are part of the fast step loop are treated differently during analysis and code generation. The fast mode will, therefore, be discussed separately in Section 8.1.2.

Slow Mode

The three step-sections that are part of the slow mode (*pre_step3d*, *step3d_uv* and *step3d_t*) are all treated as singular sections. Looking at the communication patterns, it is evident that in all three parts, most communication occurs clustered toward the end of each section. Many halo exchanges are not scheduled because their dependency stencils are growing too large but rather because all arrays with stencil extents greater than zero must be exchanged at the end of each of the four sections. This causes their wait operations to all be located at the end of a section, with exchange being initiated after the execution of the last kernel with a write access on this variable. As a result, much of the communication occurs in very short time windows, hindering the possibility of truly asynchronous communication.

In fact, 11 of the 16 halo exchanges in this section occur with no kernels between their beginning and end, effectively making them synchronous exchanges. Only four halo exchanges involve a large number of kernels between their beginnings and ends.

Fast Mode and Asynchronous Exchanges in a Loop

Multiple time steps and communication between different sections (or instances of the same section) at the boundaries were only considered for the fast steps using the *step2d* subroutine.

As the BASIN test case defaults to using two halo layers, the fast step loop is considered as a series of two instances of *step2d* here (linked by connectors to implement time index

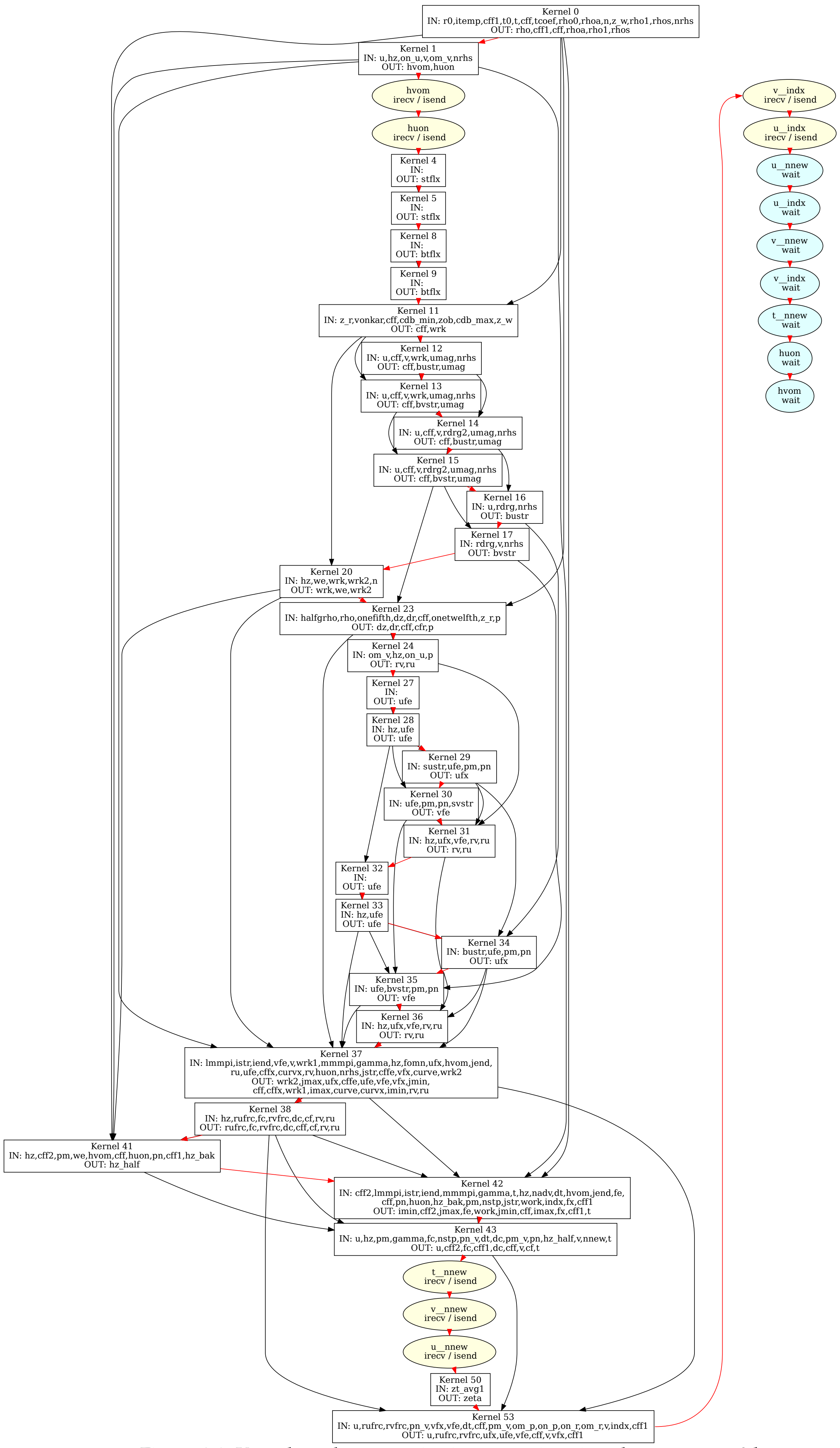


Figure 8.1: Kernels and communication operations in substep *pre_step3d*.

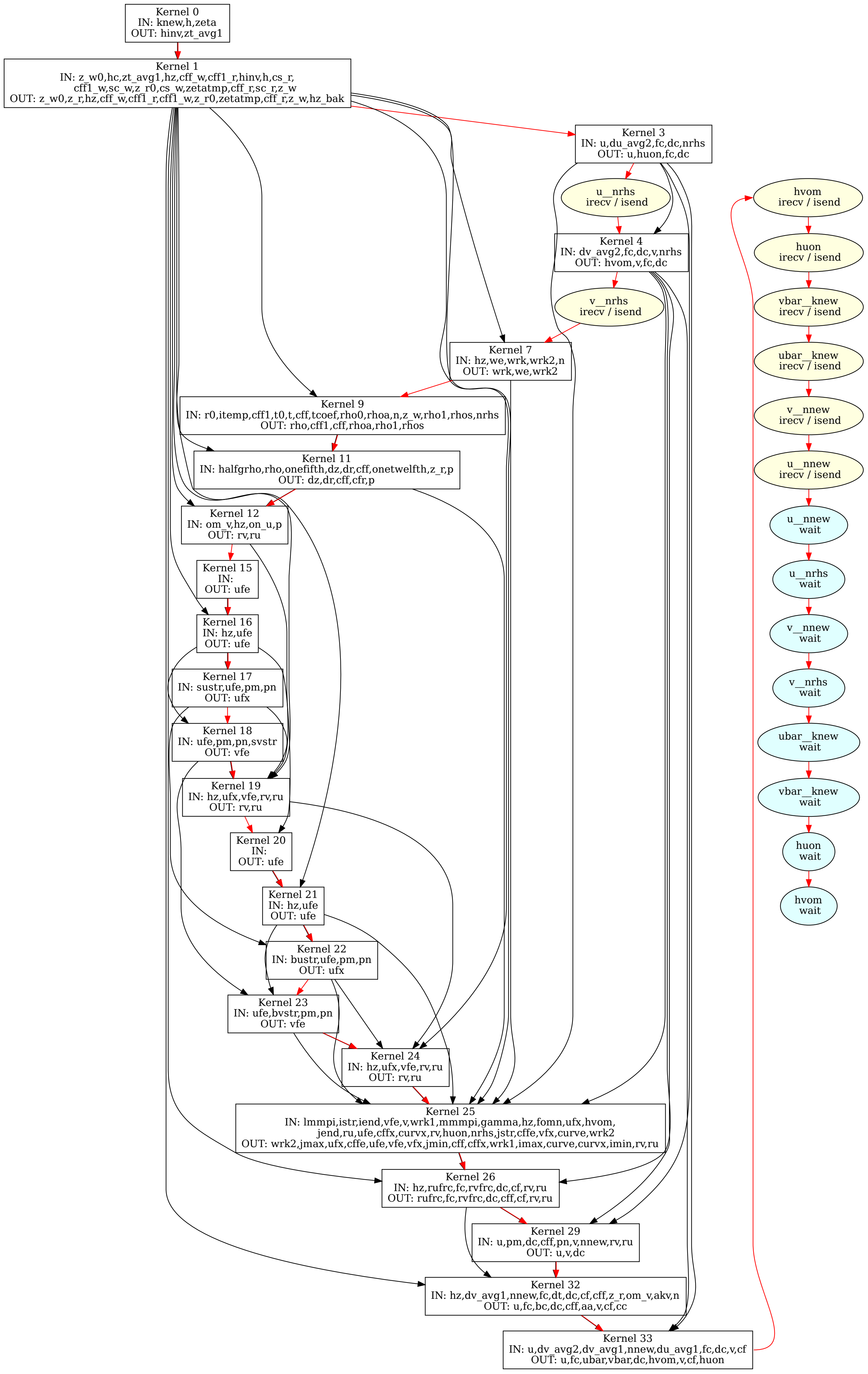

Kernel 0 IN: knew,h,zeta OUT: hinv,zt_avg1
Kernel 1 IN: z_w0,hc,zt_avg1,hz,cff_w,cff1_r,hinv,h,cs_r, cff1_w,sc_w,z_r0,cs_w,zetatmp,cff_r,sc_r,z_w OUT: z_w0,z_r,hz,cff_w,cff1_r,cff1_w,z_r0,zetatmp,cff_r,z_w,hz_bak
Kernel 3 IN: u,du_avg2,fc,dc,nrhs OUT: u,huon,fc,dc
u__nrhs irecv / isend
Kernel 4 IN: dv_avg2,fc,dc,v,nrhs OUT: hvom,v,fc,dc
v__nrhs irecv / isend
Kernel 7 IN: hz,we,wrk,wrk2,n OUT: wrk,we,wrk2
Kernel 9 IN: r0,itemp,cff1,t0,t,cff,tcoef,rho0,rhoa,n,z_w,rho1,rhos,nrhs OUT: rho,cff1,cff,rhoa,rho1,rhos
Kernel 11 IN: halfgrho,rho,onefifth,dz,dr,cff,onetwelfth,z_r,p OUT: dz,dr,cff,cfr,p
Kernel 12 IN: om_v,hz,on_u,p OUT: rv,ru
Kernel 15 IN: OUT: ufe
Kernel 16 IN: hz,ufe OUT: ufe
Kernel 17 IN: sustr,ufe,pm,pn OUT: ufx
Kernel 18 IN: ufe,pm,pn,svstr OUT: vfe
Kernel 19 IN: hz,ufx,vfe,rv,ru OUT: rv,ru
Kernel 20 IN: OUT: ufe
Kernel 21 IN: hz,ufe OUT: ufe
Kernel 22 IN: bustr,ufe,pm,pn OUT: ufx
Kernel 23 IN: ufe,bvstr,pm,pn OUT: vfe
Kernel 24 IN: hz,ufx,vfe,rv,ru OUT: rv,ru
Kernel 25 IN: lmmpi,istr,iend,vfe,v,wrk1,mmmpi,gamma,hz,fomn,ufx,hvom, jend,ru,ufe,cffx,curvx,rv,huon,nrhs,jstr,cffe,vfx,curve,wrk2 OUT: wrk2,jmax,ufx,cffe,ufe,vfe,vfx,jmin,cff,cffx,wrk1,imax,curve,curvx,imin,rv,ru
Kernel 26 IN: hz,rufrc,fc,rvfrc,dc,cf,rv,ru OUT: rufrc,fc,rvfrc,dc,cff,cf,rv,ru
Kernel 29 IN: u,pm,dc,cff,pn,v,nnew,rv,ru OUT: u,v,dc
Kernel 32 IN: hz,dv_avg1,nnew,fc,dt,dc,cf,cff,z_r,om_v,akv,n OUT: u,fc,bc,dc,cff,aa,v,cf,cc
Kernel 33 IN: u,dv_avg2,dv_avg1,nnew,du_avg1,fc,dc,v,cf OUT: u,fc,ubar,vbar,dc,hvom,v,cf,huon
hvom irecv / isend
huon irecv / isend
vbar__knew irecv / isend
ubar__knew irecv / isend
v__nnew irecv / isend
u__nnew irecv / isend
u__nnew wait
u__nrhs wait
v__nnew wait
v__nrhs wait
ubar__knew wait
vbar__knew wait
huon wait
hvom wait

Figure 8.2: ernels and communication operations in substep *step3d_uv*.

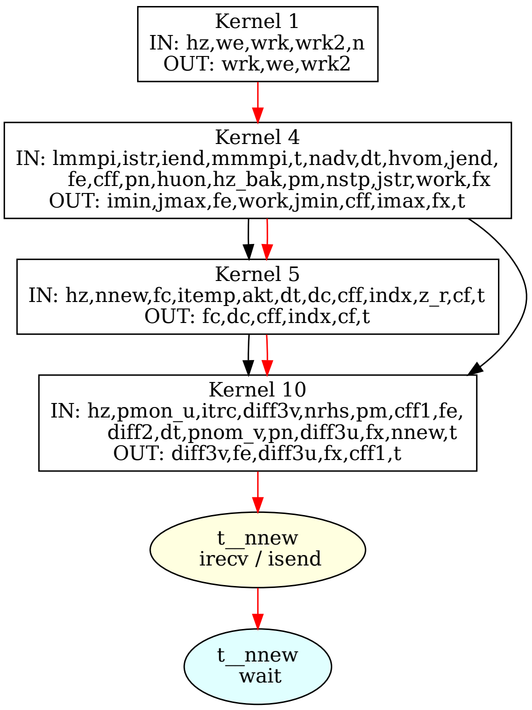

Kernel 1
IN: hz,we,wrk,wrk2,n
OUT: wrk,we,wrk2
Kernel 4
IN: lmmpi,istr,iend,mmmpi,t,nadv,dt,hvom,jend,
fe,cff,pn,huon,hz_bak,pm,nstp,jstr,work,fx
OUT: imin,jmax,fe,work,jmin,cff,imax,fx,t
Kernel 5
IN: hz,nnew,fc,itemp,akt,dt,dc,cff,indx,z_r,cf,t
OUT: fc,dc,cff,indx,cf,t
Kernel 10
IN: hz,pmon_u,itrc,diff3v,nrhs,pm,cff1,fe,
diff2,dt,pnom_v,pn,diff3u,fx,nnew,t
OUT: diff3v,fe,diff3u,fx,cff1,t
t__nnew
irecv / isend
t__nnew
wait

Figure 8.3: Kernels and communication operation in substep *step3d_t*.

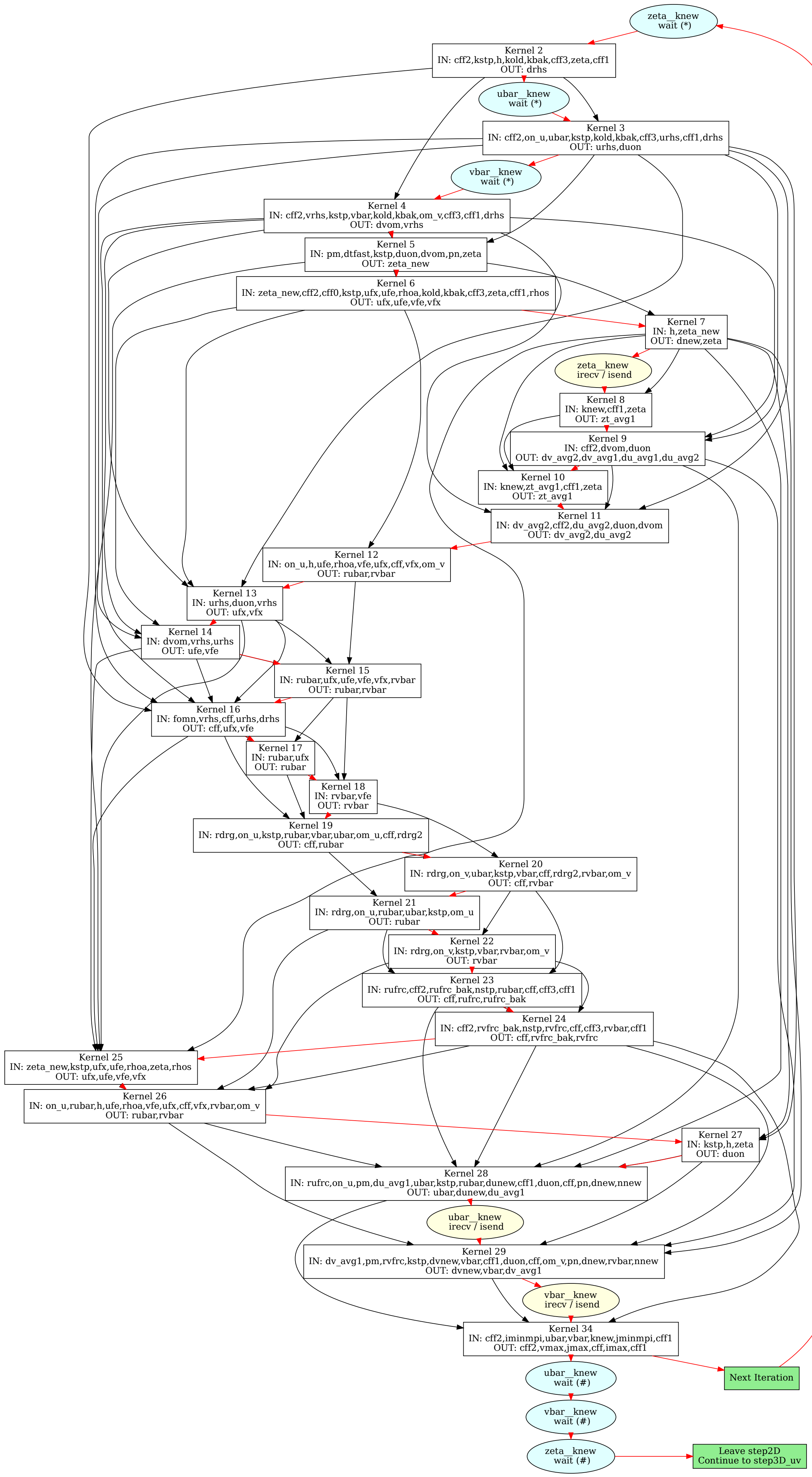

zeta__knew
wait (*)
Kernel 2
IN: cff2,kstp,h,kold,kbak,cff3,zeta,cff1
OUT: drhs
ubar__knew
wait (*)
Kernel 3
IN: cff2,on_u,ubar,kstp,kold,kbak,cff3,urhs,cff1,drhs
OUT: urhs,duon
vbar__knew
wait (*)
Kernel 4
IN: cff2,vrhs,kstp,vbar,kold,kbak,om_v,cff3,cff1,drhs
OUT: dvom,vrhs
Kernel 5
IN: pm,dtfast,kstp,duon,dvom,pn,zeta
OUT: zeta_new
Kernel 6
IN: zeta_new,cff2,cff0,kstp,ufx,ufe,rhoa,kold,kbak,cff3,zeta,cff1,rhos
OUT: ufx,ufe,vfe,vfx
Kernel 7
IN: h,zeta_new
OUT: dnew,zeta
zeta__knew
irecv / isend
Kernel 8
IN: knew,cff1,zeta
OUT: zt_avg1
Kernel 9
IN: cff2,dvom,duon
OUT: dv_avg2,dv_avg1,du_avg1,du_avg2
Kernel 10
IN: knew,zt_avg1,cff1,zeta
OUT: zt_avg1
Kernel 11
IN: dv_avg2,cff2,du_avg2,duon,dvom
OUT: dv_avg2,du_avg2
Kernel 12
IN: on_u,h,ufe,rhoa,vfe,ufx,cff,vfx,om_v
OUT: rubar,rvbar
Kernel 13
IN: urhs,duon,vrhs
OUT: ufx,vfx
Kernel 14
IN: dvom,vrhs,urhs
OUT: ufe,vfe
Kernel 15
IN: rubar,ufx,ufe,vfe,vfx,rvbar
OUT: rubar,rvbar
Kernel 16
IN: fomn,vrhs,cff,urhs,drhs
OUT: cff,ufx,vfe
Kernel 17
IN: rubar,ufx
OUT: rubar
Kernel 18
IN: rvbar,vfe
OUT: rvbar
Kernel 19
IN: rdrg,on_u,kstp,rubar,vbar,ubar,om_u,cff,rdrg2
OUT: cff,rubar
Kernel 20
IN: rdrg,on_v,ubar,kstp,vbar,cff,rdrg2,rvbar,om_v
OUT: cff,rvbar
Kernel 21
IN: rdrg,on_u,rubar,ubar,kstp,om_u
OUT: rubar
Kernel 22
IN: rdrg,on_v,kstp,vbar,rvbar,om_v
OUT: rvbar
Kernel 23
IN: rufrc,cff2,rufrc_bak,nstp,rubar,cff,cff3,cff1
OUT: cff,rufrc,rufrc_bak
Kernel 24
IN: cff2,rvfrc_bak,nstp,rvfrc,cff,cff3,rvbar,cff1
OUT: cff,rvfrc_bak,rvfrc
Kernel 25
IN: zeta_new,kstp,ufx,ufe,rhoa,zeta,rhos
OUT: ufx,ufe,vfe,vfx
Kernel 26
IN: on_u,rubar,h,ufe,rhoa,vfe,ufx,cff,vfx,rvbar,om_v
OUT: rubar,rvbar
Kernel 27
IN: kstp,h,zeta
OUT: duon
Kernel 28
IN: rufrc,on_u,pm,du_avg1,ubar,kstp,rubar,dunew,cff1,duon,cff,pn,dnew,nnew
OUT: ubar,dunew,du_avg1
ubar__knew
irecv / isend
Kernel 29
IN: dv_avg1,pm,rvfrc,kstp,dvnew,vbar,cff1,duon,cff,om_v,pn,dnew,rvbar,nnew
OUT: dvnew,vbar,dv_avg1
vbar__knew
irecv / isend
Kernel 34
IN: cff2,iminmpi,ubar,vbar,knew,jminmpi,cff1
OUT: cff2,vmax,jmax,cff,imax,cff1
Next Iteration
ubar__knew
wait (#)
vbar__knew
wait (#)
zeta__knew
wait (#)
Leave step2D
Continue to step3D_uv

Figure 8.4: Kernels and communication operations in substep *step2d_thread*.

modifications before each fast step). The resulting communication schedule is visualized in Figure 8.4.

In *step2d*, the communication window for all three variables begins and ends in different loop iterations. The halo exchange is initiated during one iteration and finalized in the next iteration.

Communication operations marked with *(*)* do not occur in the first iteration, while those marked with *(#)* only occur in the final iteration of the fast loop. This ensures that all halo exchanges initiated during the loop are completed before the loop ends. It prevents the code from attempting to finalize a halo exchange, which has not yet been initiated in its first iteration. They are implemented with conditional statements as explained in Section. Therefore, all *MPI_wait* operations in *step2d* are conditional on the index of the loop variable *iic*. As no halo exchanges have yet been initiated, the wait operations at the start cannot happen in the first iteration, so they cannot be finalized. The three waits in the end are there to finalize the communication after the last iteration.

8.2 Benchmarking Results

Counting the number of halo exchanges revealed that the automatically generated code utilized roughly half the number of halo exchanges compared to the original code. Consequently, one might assume that the performance in the benchmark would be better, but the results do not support this assumption. Figure 8.5 and Figure 8.6 compare the execution times of the three code versions on different problem sizes with different numbers of MPI processes. It can be seen that, in general, the differences in execution times between all three variants are minimal. For a given problem size and number of MPI processes, there are hardly any noticeable differences in the mean, minimum, and maximum execution times for the three code variants. The same is true for the minimum and maximum execution times.

Figure 8.6 compares the runtimes for different problem sizes and shows a small performance advantage of the variant using the reduced number of automatically scheduled original synchronous exchanges over both the asynchronous auto-generated exchanges and the synchronous baseline. However, even this difference is only noticable for small problem sizes and higher numbers of MPI processes. There are no notable differences for small numbers of MPI processes and the differences for any number of processes tend to disappear with larger problem sizes because they are overshadowed by other factors such as the general increase in computation needed for large problem sizes.

The fact that the code using about half as many of the synchronous MPI exchanges performs nearly identically to the original version shows that halo exchanges do not significantly impact runtimes in this specific test case and context. This shows that communication might not be a significant factor regarding runtime in this test case and circumstances.

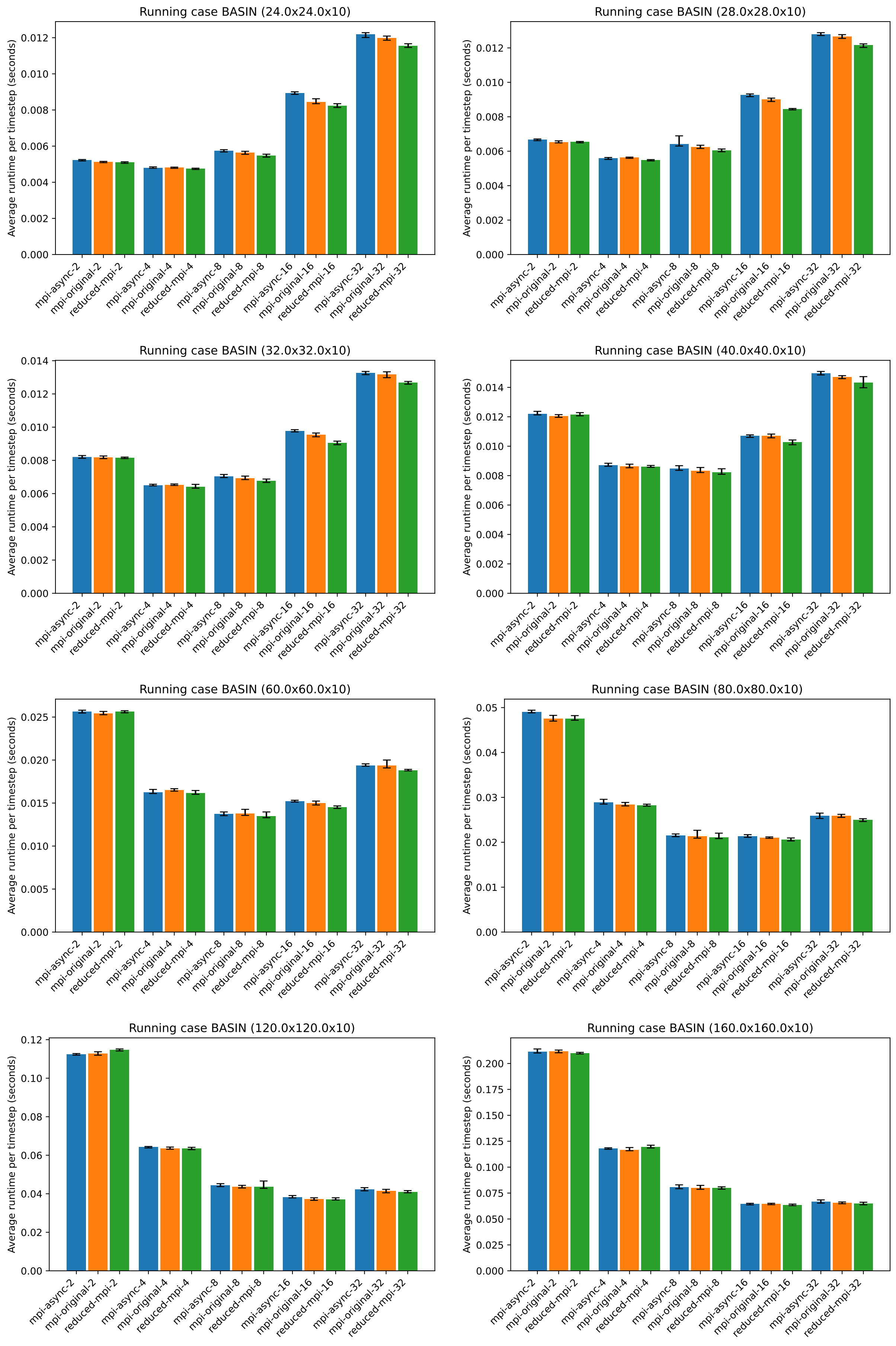
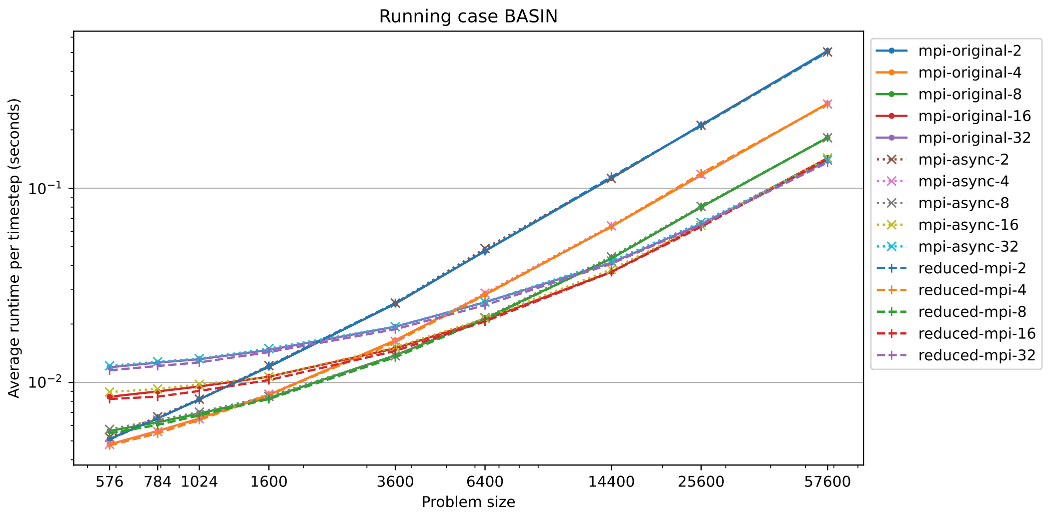


Figure 8.5: Benchmarking results for the BASIN test case for different problem sizes and numbers of MPI processes. The runtime is given as the average runtime of 8 simulations, with the minimum and maximum runtime indicated in black. All runtimes are divided by the number of steps for easier comparison.

Asynchronous Communication

Compared to the code version using auto-scheduled synchronous code exchanges, the asynchronous communication performs slightly worse when using 16 or 32 MPI processes (see Figure 8.6). A number of factors explain this. First, as the impact of communication on runtime is already small, the asynchronous variant already has a small improvement potential. Additionally as seen before, many of the 'asynchronous' exchanges are scheduled in a way that makes them not actually asynchronous, further decreasing the potential of the asynchronous communication to improve runtimes. Furthermore, the asynchronous variant introduces a number of additional computational operations (for example, extending the loop bounds of several kernel loops), possibly increasing the runtime in this way.

Another, even bigger problem could be a general lack of actual asynchronous communication in the tested scenario. To actually communicate asynchronously the communication has to progress in the background while the main thread is doing computation. Just using non-blocking MPI operations does not necessarily mean this is happening. In such a situation, communication is primarily performed during explicit communication operations rather than in the background asynchronously. Consequently, in this case most of the communication likely takes place during the wait operations, and not asynchronously in the time window before them, again leading to synchronous communication with no chance of overlapping communication and computation and latency hiding.



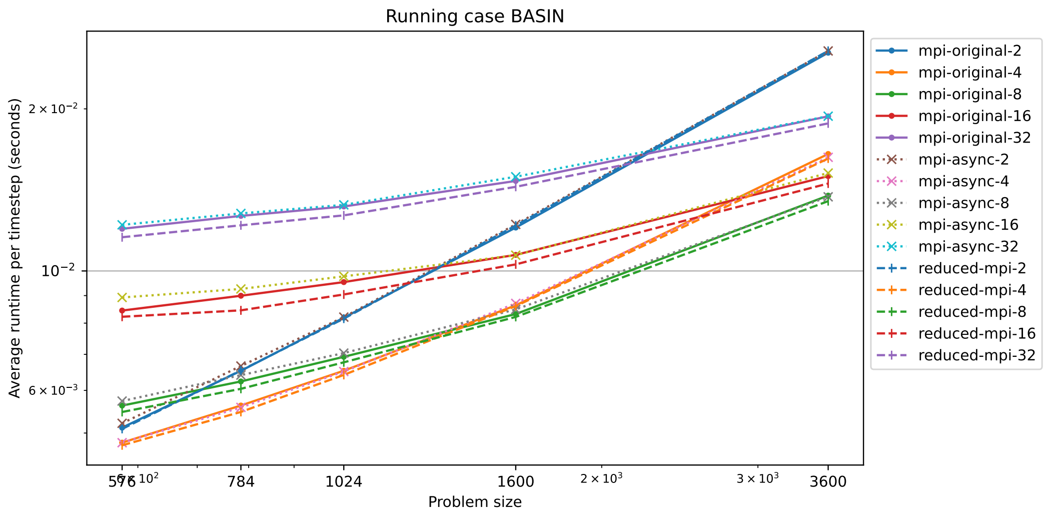


Figure 8.6: Benchmarking results for the BASIN test case for different problem sizes and numbers of MPI processes. The runtime is given as the average runtime of 8 simulations. All runtimes are divided by the number of steps for easier comparison.

9 Conclusion

The main goal of this thesis was to create an automated approach to generate code that efficiently implements communication in the form of halo exchanges for CROCO. The presented prototype effectively accomplishes this objective. Its approach uses PSyclone and Poseidon to parse and analyze CROCO code, extracting essential elements such as kernels and dependencies. It uses the algorithms described in Section 5.4 to trace stencils, their sizes, and data dependencies across multiple kernels. This information can then be used to identify when which halo exchanges are necessary and to determine a time window for asynchronous communication. Navigating the complexities of the time step loop was a significant challenge, but the approach effectively addressed some of these issues. The prototype can successfully analyze CROCO's BASIN test case and generate code implementing necessary communication operations, including both synchronous exchanges using CROCO's original implementation and asynchronous exchanges.

However, despite significantly reducing the number of halo exchanges by about 50%, benchmarking results showed only marginal performance improvements. This suggests that, in this specific context, halo exchanges did not significantly impact runtime.

The introduction of asynchronous communication, intended to enhance performance by hiding communication latency, gave worse results in many scenarios, not providing a performance advantage and occasionally even leading to slightly worse performance. This outcome may be attributed to the fact that the generated communication schedule was not truly asynchronous, with many operations clustering at the end of substeps. Furthermore, the lack of support for asynchronous for progression in MPI the used MPI implementation may not have permitted any asynchronous communication in this case, causing this outcome.

Nevertheless, the presented approach effectively automates the analysis of CROCO code and the modification of code by generating and inserting MPI code for necessary halo exchanges. This addresses some of the problems described in the thesis, making it easier for developers or domain experts to write CROCO code with parallelization using MPI without the need to handle communication manually. It also has the potential to reduce the number of exchange operations in certain cases, potentially improving performance.

All in all, the presented approach is able to automatically analyze CROCO code and modify it by generating and inserting MPI code that implements the necessary halo exchanges using MPI. This is useful and does solve some of the problems stated at the beginning of this thesis. It allows developers (or even domain experts) to write code for CROCO without having to worry about handling communication, as the code can be created automatically. It can also be used - at least in some (probably the more simple) cases (like the BASIN case) to reduce the number of exchange operations, potentially increasing performance.

The implemented asynchronous communication did not lead to higher performance in this case but could potentially allow for latency hiding in other cases, especially with some future changes.

9.1 Future Work

An immediate goal would, therefore, be to ensure that asynchronous communication actually works as intended. This involves addressing the problem of progressive asynchronous communication in mpi. This can be approached by using a different MPI implementation with supporting hardware.

The MPI implementation of the communication could be improved by using derived datatypes. This would avoid having to copy the data into a temporary buffer before sending and from the buffer to the target array after reception and, therefore, decrease overheads, potentially improving performance.

Addressing the problems with modeling the time stepping would also help to improve asynchronous communication if it enables analysis without stepping every step into multiple substeps. To achieve this, clearer software design in CROCO, particularly regarding time indexing, is necessary. This should allow for easier and better modeling of the time stepping loops which in turn should make it possible to analyze at least entire time steps or even multiple time steps at once without splitting.

Bibliography

[1] T. Sterling, M. Brodowicz, and M. Anderson, *High performance computing: modern systems and practices*. Morgan Kaufmann, 2017.

[2] J. Michalakes, “Hpc for weather forecasting,” *Parallel Algorithms in Computational Science and Engineering*, pp. 297–323, 2020.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, *et al.*, “The landscape of parallel computing research: A view from berkeley,” 2006.

[4] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, *et al.*, “Productivity and performance using partitioned global address space languages,” in *Proceedings of the 2007 international workshop on Parallel symbolic computation*, 2007, pp. 24–32.

[5] T. El-Ghazawi and L. Smith, “Upc: Unified parallel c,” in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006, 27–es.

[6] R. W. Numrich and J. Reid, “Co-array fortran for parallel programming,” in *ACM Sigplan Fortran Forum*, ACM New York, NY, USA, vol. 17, 1998, pp. 1–31.

[7] J. Mellor-Crummey, L. Adhianto, W. N. Scherer III, and G. Jin, “A new vision for coarray fortran,” in *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, 2009, pp. 1–9.

[8] W. N. Scherer III, L. Adhianto, G. Jin, J. Mellor-Crummey, and C. Yang, “Hiding latency in coarray fortran 2.0,” in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, 2010, pp. 1–9.

[9] A. Danalis, L. L. Pollock, D. M. Swany, and J. Cavazos, “Mpi-aware compiler optimizations for improving communication-computation overlap,” *Proceedings of the 23rd international conference on Supercomputing*, 2009.

[10] A. Danalis, L. L. Pollock, and D. M. Swany, “Automatic mpi application transformation with asphalt,” *2007 IEEE International Parallel and Distributed Processing Symposium*, pp. 1–8, 2007.

[11] L. Fishgold, A. Danalis, L. Pollock, and M. Swany, “An automated approach to improve communication-computation overlap in clusters,” in *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, 2006, 7 pp.-. DOI: 10.1109/IPDPS.2006.1639590.

[12] S. Pellegrini, T. Hoefler, and T. Fahringer, “Exact dependence analysis for increased communication overlap,” in *European MPI Users Group Meeting*, 2012.

[13] J. Guo, Q. Yi, J. Meng, J. Zhang, and P. Balaji, "Compiler-assisted overlapping of communication and computation in mpi applications," *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 60–69, 2016.

[14] T. Nguyen, P. Cicotti, E. J. Bylaska, D. J. Quinlan, and S. B. Baden, "Automatic translation of mpi source into a latency-tolerant, data-driven form," *J. Parallel Distributed Comput.*, vol. 106, pp. 1–13, 2017.

[15] T. Nguyen, P. Cicotti, E. Bylaska, D. Quinlan, and S. B. Baden, "Bamboo – translating mpi applications to a latency-tolerant, data-driven form," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11. DOI: `10.1109/SC.2012.23`.

[16] P. Cicotti, *Tarragon: a programming model for latency-hiding scientific computations*. University of California, San Diego, 2011.

[17] S. M. Martin, M. J. Berger, and S. B. Baden, "Toucan — a translator for communication tolerant mpi applications," *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 998–1007, 2017.

[18] S. M. Martin and S. B. Baden, "Mate, a unified model for communication-tolerant scientific applications," in *International Workshop on Languages and Compilers for Parallel Computing*, 2018.

[19] S. Jullien, M. Caillaud, R. Benshila, L. Bordois, G. Cambon, F. Dumas, S. L. Gentil, F. Lemarié, P. Marchesiello, S. Theetten, F. Dufois, M. L. Corre, G. Morvan, S. L. Gac, J. Gula, and J. Pianezze, *Croco technical and numerical documentation*, Dec. 2022. DOI: `10.5281/zenodo.7400922`.

[20] A. F. Shchepetkin and J. C. McWilliams, "The regional oceanic modeling system (roms): A split-explicit, free-surface, topography-following-coordinate oceanic model," *Ocean Modelling*, vol. 9, pp. 347–404, 2005.

[21] *WikiROMS — myroms.org*, `https://www.myroms.org/wiki/Documentation_Portal`, [Accessed 28-09-2023].

[22] A. E. Gill, *Atmosphere-ocean dynamics*. Academic press, 1982, vol. 30.

[23] A. Coughtrie, R. Ford, J. Henrichs, I. Kavcic, A. Porter, and S. Siso, *Psyclone user guide*, `https://psyclone.readthedocs.io/en/stable/index.html`, [Accessed 18-09-2023).

[24] *Git - git-am Documentation — git-scm.com*, `https://git-scm.com/docs/git-am`, [Accessed 29-09-2023].

[25] Message Passing Interface Forum, *MPI: A message-passing interface standard version 4.0*, Jun. 2021.

[26] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, ser. Communications in Computer and Information Science, I. I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan, Eds., vol. 367, Springer International Publishing, 2013, pp. 3–20, ISBN: 978-3-319-04518-4. DOI: `10.1007/978-3-319-04519-1\_1`.