Master's thesis

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

Richard Bachmann

Performance Modeling of Finite Difference Shallow Water Equation Solvers with Variable Domain Geometry

Master's thesis in Computer Science
Supervisor: Jan Christian Meyer
June 2021

NTNU
Kunnskap for en bedre verden

Richard Bachmann

Performance Modeling of Finite Difference Shallow Water Equation Solvers with Variable Domain Geometry

Master's thesis in Computer Science
Supervisor: Jan Christian Meyer
June 2021

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

NTNU
Norwegian University of
Science and Technology

Performance Modeling of Finite Difference Shallow Water Equation Solvers with Variable Domain Geometry

Richard Bachmann

June 22, 2021

Problem Description

This study aims to develop quantitative models of performance and scalability effects when solving the Shallow Water Equations for domains with complex geometry features, and to validate these models experimentally.

Abstract

In this thesis we examine techniques for reducing the computational load imbalance which may arise when creating decompsitions of a two-dimensional finite difference domain with regions of uneven computational cost. In this case a simple Cartesian decomposition may create partitions which are not balanced, because it does not exploit the domain's features. Depending on the choice of implementation, such an imbalance can result in significant periods of idle time in the application, as a consequence of one process having to wait for the results of another.

We describe two decomposition schemes to contrast with the simple Cartesian decompositon. One of these is rooted in orthogonal recursive bisection, while the other is based on diagonal rectangulation. We then show that the increase in communication cost introduced by the more complex decomposition schemes is outweighed by the efficiency gained though reduced wait times. Finally, we implement a proxy-application which solves the shallow water equations numerically. This application introduces regions of differing computational cost, which allows us to test our ideas and demonstrate their efficacy.

Sammendrag

I denne oppgaven undersøker vi metoder som kan brukes til å redusere ubalansert beregningslast, som kan oppstå når man dekomponerer et todimensjonalt domene av endelige differanser med ujevn beregningskostnad. I dette tilfellet kan en Kartesisk dekomponering opprette ubalanserte partisjoner, fordi den ikke utnytter strukturen til domenet. Slike ubalanser kan resultere i signifikante ventetider for applikasjonen, som skyldes at en prosess må vente på å motta resultatdata fra en annen.

Vi beskriver to dekomponeringsteknikker som står i kontrast til den Kartesiske metoden. En av disse er basert på ortogonal rekursiv halvering, mens den andre er basert på diagonal rektangulering. Vi viser så at den økte kommunikasjonskostnaden som disse fører til oppveies gjennom økt effektivitet som følge av reduserte ventetider. Til slutt implementerer vi en proxy-applikasjon som løser grunntvannslikningene numerisk. Denne applikasjonen introduserer regioner med varierende beregningskostnad som lar oss teste våre ideer og demonstrere deres nytte.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter we state the motivation and scope of our work. This is followed by a description of how this paper is structured.

1.1 Motivation

A great number of domain decomposition techniques for fluid simulations exist. We identified that few studies examine these in conjunction with the cost they incur in terms of communication. By measuring the time spent on communications we develop a quantitative model, which may be used to judge the scalability of the decomposition scheme. Such models are useful tools for judging the time and computational resources needed to complete a task.

1.2 Scope

Our overarching goal is to to investigate behaviors stemming from the introduction of a heterogeneous domain into our finite difference simulation. Prime among these is the aspect of load balance and additional communication overhead.

We choose to focus our efforts on the computational aspects of our fluid simulation, at the expense of some physical realism. This allows us to spend more time on studying how the software interacts with its computational platform. For this reason, fluids simulated in this work do not accurately reflect any fluid from the real world in particular. We do, however, mirror the behaviors of proper fluid simulation programs in our own.

Our investigations are heavily tied to parallel execution and communications. There exists a great variety of architectures and design patterns for their implementation, which are too numerous to investigated all at once. We choose to focus on a hybrid MPI+OpenMP solution, where communications take place between separate nodes and computations are parallelized locally using threads. With this setup we examine inter-node communications, which we assume to be more significant in terms of cost than intra-node communications.

1.3 Related work

Parna et al [1] have created a WENO-based finite difference simulation with overlapping subdomains. The program runs on multiple GPUs in parallel, and it handles wetting and drying of complex three-dimensional terrain.

Colicchio et al [2] utilize domain decompositions of 3D space to combine two different solvers. The technique allows the authors to model water-on-deck phenomena which occur when ships are hit by tall waves.

Alfonso Sánchez-Beato outlines in his blog a technique for efficiently partitioning a screen into multiple video windows, while preserving the videos' respective aspect ratios to the best possible degree [3]. This can be viewed as a form of domain decomposition, and serves as a major source of inspiration for our work. In the Chapter 3 we extend some of his ideas to our shallow water domain.

Other spatial domain compositions exist, such as Yin-Yang grids and the Schwarz alternating method. The former can be used to decompose the surface of a three-dimensional sphere into a set of two-dimensional rectangles, which is useful for weather forecasts [4]. The latter allows for the decomposition of a domain with a complex shape into a set of overlapping domains with simple geometric shapes [5].

1.4 Structure

We draw from knowledge of three interconnected fields, each of which we introduce in their own chapter. In Chapter 2 we give an introduction to computational fluid dynamics and numerical methods. We use Chapter 3 to explain the techniques for our domain decomposition, and Chapter 4 presents methods for achieving high performance through parallelism in HPC applications. These three topics are followed by Chapter 5, where we combine them in our developed application. The chapter presents implementation details and concludes with a performance model. Chapter 6 describes the measurements we perform and introduces our computing platforms. The results of the former are then displayed and discussed in Chapter 7. Finally, we summarise our findings in the conclusion found in Chapter 8.

Chapter 2

Computational Fluid Dynamics

In this chapter we present some fundamentals of fluid simulation, which are needed to describe our application. We first give a short introduction to fluid dynamics, and then we follow this up with a description of our numerical method of choice.

Computational Fluid Dynamics (CFD) is a versatile field at the intersection of research and engineering, concerned with the simulation of fluid motion by solving non-linear partial differential equations. CFD techniques are applicable both for fluids and gases, under a wide range of conditions. It follows that there are many practical use cases, such as: Modeling the interactions of ship hulls with ocean waves [2], meteorological forecasts [6], flood prediction [1], landslide modeling [7], and modeling blood flow through elastic 3D arteries [8]. CFD simulations are typically executed on *High Performance Computing (HPC)* platforms, due to their great need for computing resources.

2.1 The Simulated Fluid

Fluids are capable of a wide variety of dynamic behaviors, which depend on the properties of the fluid in question as well as the features of its surroundings. While fluid dynamics may refer to the behaviors of both liquids and gases, we here focus on liquids alone. Thus, our fluids are treated as incompressible, such that their density does not change over time.

The vast majority of fluids experience a frictional force when interacting with their surroundings, and they have a viscosity such that they have a degree of internal friction which originates from its layered structure or individual particles. An *ideal fluid* is a much simpler construct which is *inviscid*. That is, it experiences neither internal nor external friction. It therefore has no internal shear stress, nor surface tension. Such a fluid is characterized exclusively by its isotropic pressure, density, and the shape of its surroundings.

We say that fluids in motion experience *shallow water flow*, as seen in Figure 2.1, if the width and length of the examined areas far exceed their depth. In such a flow the differences in horizontal velocities experienced by a column of fluid, as well as vertical flows, are negligible. In practice, this means that we may consider the whole column as a mass with a uniform horizontal velocity, without internal friction and turbulences. Indeed,

our model defines our fluid as a set of heights on a set of coordinates, instead of regarding it as a collection of particles.

In this work we examine open channel shallow water flows, which have a solid border below, but not on top. Our simulated fluid is an ideal, homogeneous fluid with a density approximately equal to that of water.

2.2 The Shallow Water Equations

The shallow water equations are a set of hyperbolic *Partial Differential Equations (PDEs)* used to model shallow water flows. We will use these PDEs to model two-dimensional flows, with explicit velocities in two horizontal directions. The equations also exist for one-dimensional simulations, in which case they are also known as the Saint-Venant equations. The equation set is derived from the three-dimensional Navier-Stokes equations integrated over the domain depth. The latter in turn follow from the principle of conservation of mass and momentum.

The equations may be written both in *conservative* and *non-conservative* fashions.

Since our fluid is an ideal fluid we may ignore the effects of friction and viscosity. Additionally, we amortize external forces such as the Coriolis forces and wind effects. The conservative shallow water equations may then be written as:

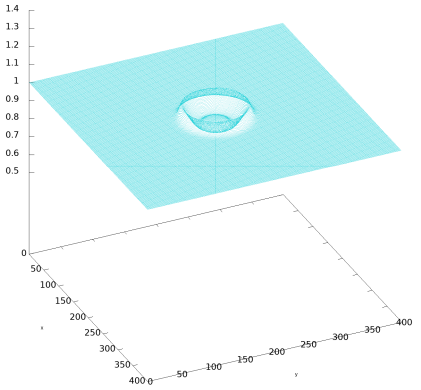


Figure 2.1: A rendering of shallow water flow, where the height of the wave is much smaller than the domain's width and length.

$$\frac{\partial(\eta\rho)}{\partial t} + \frac{\partial(\eta\rho u)}{\partial x} + \frac{\partial(\eta\rho v)}{\partial y} = 0 \tag{2.1}$$

$$\frac{\partial(\eta\rho u)}{\partial t} + \frac{\partial}{\partial x}\left(\eta\rho u^2 + \frac{1}{2}g\rho\eta^2\right) + \frac{\partial(\eta\rho uv)}{\partial y} = 0 \tag{2.2}$$

$$\frac{\partial(\eta\rho u)}{\partial t} + \frac{\partial(\eta\rho uv)}{\partial x} + \frac{\partial}{\partial y}\left(\eta\rho v^2 + \frac{1}{2}g\rho\eta^2\right) = 0 \tag{2.3}$$

Where:

- η is the total height of the fluid column at the given point.
- ρ is the density of the simulated fluid.
- g is the gravity acceleration.
- u and v are the fluid's depth-averaged velocities in x and y dimensions respectively.

We treat the density ρ as a constant in this work, and may therefore simplify Equation 2.1, 2.2, and 2.3:

$$\frac{\partial m}{\partial t} + \frac{\partial (mu)}{\partial x} + \frac{\partial (mv)}{\partial y} = 0 \tag{2.4}$$

$$\frac{\partial (mu)}{\partial t} + \frac{\partial}{\partial x}\left(mu^2 + \frac{1}{2}gm\eta\right) + \frac{\partial (muv)}{\partial y} = 0 \tag{2.5}$$

$$\frac{\partial (mu)}{\partial t} + \frac{\partial (muv)}{\partial x} + \frac{\partial}{\partial y}\left(mv^2 + \frac{1}{2}gm\eta\right) = 0 \tag{2.6}$$

Where:

- m denotes the mass represented by a fluid column.

2.3 Numerical Methods

PDEs, such as the shallow water equations, generally have no analytical solution. Numerical analysis allows for the discovery of acceptable solutions to such complex mathematical problems by the use of approximation. Solutions are computed for discrete, successive time steps.

Numerical methods may be either explicit, or implicit in their representation of the problem at hand. Explicit methods compute the next system state from the system's current state. Implicit methods state the problem as a function of the current state and the next state, which can then be solved.

2.4 Finite Differences

Numerical methods may take many shapes, depending on the chosen abstraction and procedure. The *Finite Difference Method (FDM)* is one such approach. Here the simulated domain is overlaid with a mesh, either structured or unstructured, which consists of a set of discrete points. We start with an initial condition at time $t = t_0$, where we define the fluid level and velocity at each point. This state can represent a set of conditions we wish to investigate, such as a wave on open water. In order to obtain a subsequent state we increment the time by one time step $t_{k+1} = t_k + \Delta t$ and perform an integration on an expression which involves a derivative and a selection of current states. The value obtained from the integration gives the new value at a specific point. We repeat this procedure for each point in the domain. A function to integrate may be given by forward-, backward- and centered differences, respectively shown by Equations 2.7, 2.8, and 2.9, or some combination thereof.

$$f'(x) \approx \frac{f(x_{i+1}) - f(x_i)}{\Delta x} \tag{2.7}$$

$$f'(x) \approx \frac{f(x_i) - f(x_{i-1})}{\Delta x} \tag{2.8}$$

$$f'(x) \approx \frac{f(x_{i+1}) - f(x_{i-1})}{2\Delta x} \tag{2.9}$$

FDMs are generally characterized by their order of accuracy, which indicates the lowest order derivative in the truncation error of the utilized series expansion.

We elect to use a well-studied method as a basis for our research: The MacCormack method. The equations in this work use x and y to indicate relative position in two-dimensional space, and k to indicate the iteration for which the value is valid.

2.4.1 The MacCormack Method

The MacCormack method is a well-studied explicit numerical technique for solving hyperbolic PDEs. The method is based on Taylor series expansion in time and is second order accurate in both space and time, given that a symmetric sequence of operations is used [9]. It uses a two-step predictor-corrector algorithm to calculate the next state of the domain. Backward space differences are used in the predictor step, and the corrector utilizes forward space differences. In doing so, the method eliminates most of the directional bias [10]. Furthermore, the MacCormack method belongs to the time-splitting methods, also known as "fractional time steps" methods [11], where the operations associated with each space dimension are applied individually and in succession.

The MacCormack method has gained great popularity, due to its relative ease of implementation while simultaneously delivering reliable and fast results. It is related to the Lax-Wendroff method, which it simplifies and extends with the mentioned predictor-corrector procedure. Numerous variations of the MacCormack method have been developed, such as: MacCormack-TVD schemes [7] [12] and an implicit-explicit MacCormack and Crank-Nicolson hybrid [13]

Basic Two Dimensional MacCormack steps

The predictor step is defined as:

$$m_{x,y}^{\overline{k+1}} = m_{x,y}^{k} - \Delta t \left(\frac{(mu)_{x+1,y}^{k} - (mu)_{x,y}^{k}}{\Delta x} - \frac{(mv)_{x,y+1}^{k} - (mv)_{x,y}^{k}}{\Delta y} \right) \tag{2.10}$$

The corrector step is given as:

$$m_{x,y}^{k+1} = \frac{m_{x,y}^{k} + m_{x,y}^{\overline{k+1}}}{2} - \frac{\Delta t}{2} \left(\frac{(mu)_{x,y}^{\overline{k+1}} - (mu)_{x+1,y}^{\overline{k+1}}}{\Delta x} - \frac{(mv)_{x,y}^{\overline{k+1}} - (mv)_{x,y-1}^{\overline{k+1}}}{\Delta y} \right) \tag{2.11}$$

Where:

- $u^{\overline{k+1}}$ and $v^{\overline{k+1}}$ are intermediate velocities in x and y dimensions calculated using $m^{\overline{k+1}}$

The velocities in the x dimension have their own predictor-corrector stages, which are given by:

$$u_{x,y}^{\overline{k+1}} = u_{x,y}^{k} - \Delta t \left(\frac{(u_{x+1,y}^{k})^2 - (u_{x,y}^{k})^2}{\Delta x} + \frac{(muv)_{x,y+1}^{k} - (muv)_{x,y}^{k}}{\Delta y} \right) \tag{2.12}$$

$$u_{x,y}^{k+1} = \frac{u_{x,y}^{k} + \overline{u_{x,y}^{k+1}}}{2} - \frac{\Delta t}{2}\left(\frac{(u_{x,y}^{k})^2 - (u_{x-1,y}^{k})^2}{\Delta x} + \frac{(muv)_{x,y}^{k} - (muv)_{x-1,y}^{k}}{\Delta y}\right) \tag{2.13}$$

The velocities in the y dimension are similarly calculated using:

$$\overline{v_{x,y}^{k+1}} = v_{x,y}^{k} - \Delta t\left(\frac{(v_{x,y+1}^{k})^2 - (v_{x,y}^{k})^2}{\Delta y} + \frac{(muv)_{x+1,y}^{k} - (muv)_{x,y}^{k}}{\Delta x}\right) \tag{2.14}$$

$$v_{x,y}^{k+1} = \frac{v_{x,y}^{k} + \overline{v_{x,y}^{k+1}}}{2} - \frac{\Delta t}{2}\left(\frac{(v_{x,y}^{k})^2 - (v_{x,y-1}^{k})^2}{\Delta y} + \frac{(muv)_{x,y-1}^{k} - (muv)_{x-1,y}}{\Delta x}\right) \tag{2.15}$$

The relative position of the points used to calculate the value of $m_{x,y}^{k+1}$ forms our *stencil*.

2.4.2 Stability

The notion of stability in the simulated domain is central to CFD. Instabilities occur when the error between the numerical solution and exact solution is unbounded. These can manifest as *singularities*, where there is a discontinuity in some derivative. Singularities tend to spread from a single source until the whole domain is destabilized by moving along the stencil of our numerical method each iteration. The practical consequence of a singularity is a breakdown of our simulation. Singularities may also occur in the absence of numerical instabilities, such as locations where the fluid surface has excessively sharp curvatures.

Roache describes two general kinds of instabilities [11]: *Dynamic instability*, stemming from effects such as oscillatory overshoot with increasing amplitude, and *static instability*. The first of these my be avoided by selecting a sufficiently small time step Δt. To help us select such a Δt we can make use of the *Courant-Friedrich-Lewy (CFL)* condition. The CFL condition serves an indicator for whether or not a simulation can be considered as stable. It indicates broadly how far information travels in one iteration of a simulation. In the two-dimensional case the CFL condition may be written as:

$$C = \frac{|u|\Delta t}{\Delta x} + \frac{|v|\Delta t}{\Delta y} \leq C_{max} \tag{2.16}$$

Where:

- C is the Courant number.
- C_{max} is our condition for retaining stability.

We want information to travel no further than one point in a single Δt, since larger movements may cause divergences that can evolve into singularities with the MacCormack scheme. This means that our C_{max} can be no greater than one. In practice it must often times be even smaller [13].

2.5 Boundary Conditions

In simulating a finite domain one inevitably has to decide how the system is to interact with the imagined surroundings outside the simulated domain. Without physical laws that give exact specifications, we have to settle for approximations that yield sufficiently realistic behaviors. Boundaries may be passable, allowing waves to pass freely out of the domain, or reflective, such that they act as a solid wall. Other behaviors, such as inflows or outflows of fluid or oscillations are possible as well. The choice of boundary condition determine the behavior and the method of achieving them. Boundary conditions may in other works also be referred to as *interface conditions*.

Neumann boundary conditions define the derivative value of the function along the boundary explicitly. We may set this derivative to equal 0 in combination with the MacCormack scheme, which has the effect of stopping wave propagations.

$$\frac{\partial m(t)}{\partial t} = \beta = 0 \tag{2.17}$$

Chapter 3

Domain Decomposition

In this chapter we present two techniques for partitioning the simulated domain into a set of smaller sub-domains. The aim is to partition a domain with regions of varying computational load in a way that equalizes the workload of the sub-domains. The partitioning process is performed with a separate program before the application described in Chapter 5 is executed. The produced layout is exported to a file which is saved and later imported by the application.

3.1 Rectangulations

We limit our partition style to *rectangulations* which are rectangular partitionings that split a rectangular domain into a set of smaller, non-overlapping rectangles. These smaller rectangles may be referred to as *tiles*. Both the sides of the original rectangle and any tile are parallel with the coordinate axes. The placement and size of each rectangle may vary based on the implementation of the algorithm and the sequence of execution. We refer to the number of partitionings of a rectangulation as the *rectangulation degree*.

Rectangulations are concerned with the number of ways to perform such a partition for a given domain and degree. Ordinarily the term rectangulation refers to tiling a domain with a sparse distribution of points, such that each point is on the intersection of two rectangles. Point-free rectangulations are a variant of these which, instead of operating on a set of points, split the domain into a discrete lattice coordinate system. For example by splitting the axes such that $x, y \in \mathbb{N}$. We utilize only the latter, as it corresponds to the discretization of our numerical domain. By extending this notion to include the topological structure of the domains, that is the spatial relation of any two rectangles, we get what is known as a *floorplan* [14]. A spatial relation may for example be a tile being "south of" and "west of" its neighbor. The partition of a floor in a house into separate rooms is a popular metaphor for this concept. Useful applications of floorplans can be found in integrated circuit design and *Very Large Scale Integration* (VLSI) [15]. A floorplan is referred to as a *mosaic floorplan* if we also require that every region of the domain must be covered by a tile.

We implement two mosaic floorplan algorithms for our study: The first method, which we call the *scanning partitioner*, is an algorithm which is simple to implement and run. The second method is based on the process of *diagonal rectangulation*.

3.1.1 Scanning Decomposition

We implement a simple scanning partitioning scheme as a baseline for comparison. This scheme is a variant of the *Orthogonal Recursive Bisection* (ORB) algorithm described by Geoffrey C. Fox [16]. Our scheme scans across a domain, aggregating the total computational load represented by the encountered points. This load is estimated using the model described in Section 5.3. Once the scanned workload exceeds half of the domain's total load, the rectangle is split in two smaller ones. We alternate between scanning along the width and length of the domain. Each iteration the algorithm picks the most computationally intensive rectangle to be split. This continues until the number of rectangles equals the number of participating processes. The process for creating a partition with four tiles can be seen in Figure 3.1.

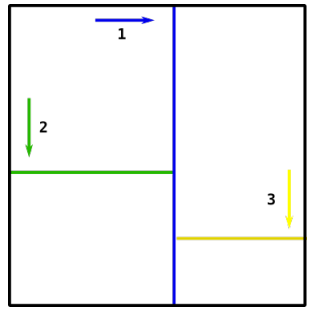


Figure 3.1: The scanning algorithm alternates between traversing the domain horizontally and vertically. Once half the load has been encountered, the current rectangle is split.

3.1.2 Diagonal Rectangulation

The *generic rectangulations* are a subset of rectangulations which have the additional property of having no points where four rectangles intersect, referred to as a *cross* [17]. This means that tiles may only meet by having parallel adjacent outlines, or through three tiles forming a 'T' shape junction. *Diagonal rectangulations* are a subclass of the generic rectangulations, which are created such that every tile contains one point along a diagonal of the original rectangle.

A diagonal rectangulation is conducted by drawing a line segment between two diagonally opposing corners of the bounding rectangle R, and assigning each discrete point the diagonal intersects with an index $1..n$. For the purpose of creating only one set of equivalent solutions, we specify that the line segment must originate in the north-western corner, and end in the south-western one. We now select points, one by one, as the start of our rectangles. The order of selection makes up our sequence *seq* for the rectangulation.

Each rectangle, indexed by its element in the sequence, is drawn by using the following algorithm:

1. Select the next indexed point p along the diagonal of R from seq.
2. Draw the largest possible rectangle such that:
    - The rectangle incorporates p.
    - The rectangle originates from the first free point furthest toward the south-western edge of R that may incorporate p.
    - The sides of the rectangle cross no existing rectangle, including R.
    - The rectangle includes no other points in the sequence along the diagonal.
    - It goes no further west than the rectangle to its south.
    - It goes no further north than the rectangle west of it.

We show a simple example of the diagonal rectangulation process in Figure 3.2.

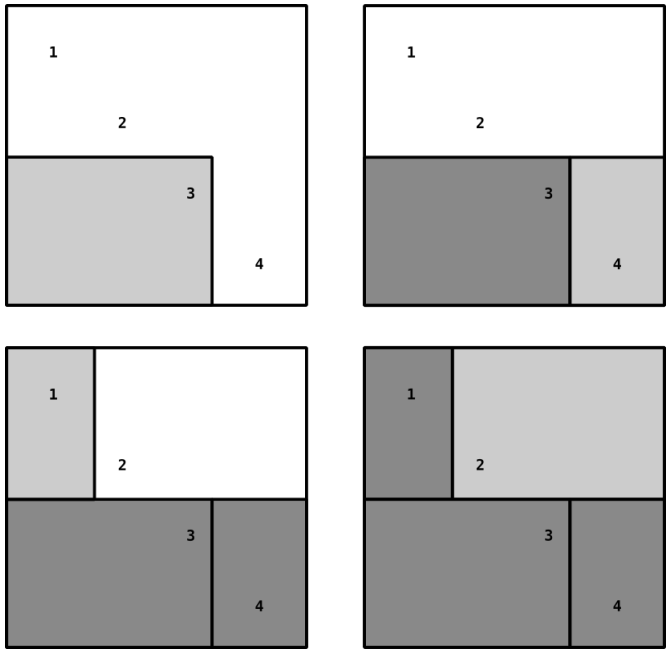


Figure 3.2: A valid diagonal rectangulation using the sequence 3,4,1,2.

Deduplication

In rectangulating R we may come across different sequences for R which yield equivalent tilings. We say that a tiling of R using sequence seq is equivalent to a tiling of R' with sequence sec' if there exists a bijection of the tiles of R to the tiles of R' which preserves

the mosaic floorplan relations of 'north of' and 'west of'. These equivalent rectangulations may be identified and avoided by use of Baxter permutations [18].

The Baxter number $B(n)$ gives the number of Baxter permutations contained in a sequence of length n. Each Baxter permutation represents a unique diagonal rectangulation. We may calculate such a Baxter number by using Equation 3.1

$$B(n) = \sum_{k=1}^{n} \frac{\binom{n+1}{k-1}\binom{n+1}{k}\binom{n+1}{k+1}}{\binom{n+1}{1}\binom{n+1}{2}} \tag{3.1}$$

Where:

- n is the number of elements in the sequence, in our case the degree of the rectangulation.
- k is the number of elements in a chosen subset of said sequence.

The growth of this number is not intuitive form this equation alone. We include a table of reference Baxter numbers in the appendix 8.1. Shen and Chu [19] prove that there exists a tight bound for the growth of the Baxter number:

$$B(n) = \Theta\left(\frac{n!2^{3n}}{n^4}\right) \tag{3.2}$$

We may identify Baxter permutations using the procedure outlined by Plott [20]. Let σ_a be an element of the sequence indexed by a, and the indexes i, j, k, and l have the relation $1 \leq i < j < k < l \leq n$. A sequence is then a Baxter permutation if it satisfies the following conditions:

1. If $(\sigma_i + 1 = \sigma_l) \wedge (\sigma_j > \sigma_l)$, then $\sigma_k > \sigma_l$
2. If $(\sigma_l + 1 = \sigma_i) \wedge (\sigma_k > \sigma_i)$, then $\sigma_j > \sigma_i$

We may state the first more plainly by saying that we for any seq pick two numbers σ_i and σ_l in seq such that $\sigma_i + 1 = \sigma_l$. We then examine the values in seq which are located between them. From these we select our all pairs of σ_j and σ_k such that $j < k$, and see whether or not the condition $\sigma_k > \sigma_l$ applies. The second condition is equivalent to the first, but reversed. For instance, the sequence $seq = 61832547$ is **not** a Baxter permutation, since we may pick $\sigma_i = 6$ and $\sigma_l = 7$. This allows us to select $\sigma_j = 8$ and $\sigma_k = 3$ from the values between the two, which is in conflict with the first condition.

We are only interested in sequences which are Baxter sequences of a given diagonal. Other sequences are considered to be duplicates and are therefore discarded.

Workload Optimization

For each diagonal rectangulation we attempt to scale the dimensions of the rectangles such that the difference in projected workload is minimized.

The workload minimization is performed iteratively using an *Sequential Least Squares Programming (SLSQP)*. This is a variation of the *Sequential Quadratic Programming*

(SQP) algorithm where the quadratic programming subproblem is replaced by linear least squares subproblem. A detailed description of the procedure is given by Kraft [21].

The minimization algorithm requires inputs in the form of our domain layout, a cost estimation function, and a start state for the floorplan. The estimated workload represented by a single point during one iteration is given through a ponderated function f of the given point's type τ. We return to this function in Section 5.3.

During the minimization process we recalculate the estimated load balance of the rectangulation for each iteration. The rectangulation is optimized with regards to the difference in load of each rectangle when compared to the domain average, as shown in Equation 3.3.

$$E = \sum_{i=0}^{n} \left(L(r_i) - \frac{\sum L(r)}{n} \right)^2 \tag{3.3}$$

Where:

- $L(r)$ is the computational load represented all points within a rectangle r.
- n is the degree of the rectangulation, which corresponds to the number of points along the diagonal.

The SLSQP algorithm also requires a Jacobian of the minimized function, which is derived from the derivative of Equation 3.3.

$$E' = \sum_{i=0}^{n} 2 * \left(L(r_i) - \frac{\sum L(r)}{n} \right) \tag{3.4}$$

Listing 3.1 summarizes the described diagonal rectangulation, deduplication, and minimization procedures in pseudocode.

```
domain_decomposition(domain, degree):
    best_rect = None

    for permutation in 1..degree:
        if permutation is Baxter:
            rect = diagonal_rectangulation(domain, permutation)
            opt_rect = SLSQP_min(rect, domain, f())
            if opt_rect < best_rect:
                best_rect = opt+rect
    return best_rect
```

Listing 3.1: Diagonal Rectangulation pseudocode.

Chapter 4

Parallel Programming and Models

HPC relies on the utilization of large-scale parallelism in order to overcome the limited performance achievable by a single CPU core. The largest modern HPC platforms rely on on highly parallel architectures, composed of many independent units cooperating through high-speed interconnects. Each of these units is can be regarded as a fully functioning computer, composed of one or multiple CPUs, local memory, and either local or remote storage. In this chapter we examine how we can divide the total work among multiple computing units operating in parallel, thereby significantly speeding up the completion of computationally intensive tasks.

In this context a *node* refers to a physical computing unit connected to a local high-bandwidth network. Each node contains at least one CPU, which in turn is composed of multiple CPU cores executing in parallel. A core is capable of executing one process thread at a time, but may switch between threads.

4.1 Approaches to Parallelism

Parallelization can take many forms, depending on the problem at hand, the platform and the context. We here consider three different approaches:

- **Sequential Pipelined:** A problem can be split into a series of sub-problems sequential in time, where each depends on the outcome of the previous one. This approach is conceptually reminiscent of a factory production line, and it allows each of the stations handling an operation to be highly specialized. We see this style of parallelism in hardware design, for instance in the pipeline of a processor.
- **Batch Programming:** Instead of reducing the problem into a series of sequential steps, it may be split into a pool of parallel tasks running from start to finish. These sub-tasks can be either atomic units of the problem space, such as a single point, or aggregates thereof. Batch programming is useful when the problem may be logically split into a number of sub-problems that far exceeds the available computing resources.

- **Bulk Synchronous Parallel:** Instead of splitting the problem into logically atomic tasks, the problem is split into a number of sub-problems that is equal to the number of computing units. In the *Bulk Synchronous Parallel (BSP)* model [22], each computing unit performs an iteration of local computations, followed by a communication and synchronization step. The communication step is used to exchange data needed for the next iteration with the other units.

For our purposes we select a BSP-inspired approach. This decision is rooted in the fact that our platforms do not have a pipelined architecture specifically designed for the shallow water equations. Furthermore, the domain may grow so large that the data exchanges required to transmit a batch-style subproblem from one node to another could become a significant strain on the system. With a BSP-style approach, each node can be assigned its own sub-domain. Data transmissions are kept at a minimum, since only the outermost rim of the sub-domains need to be exchanged. We return to the details of communication implementations in Section 5.6.3.

In order to assess the gains and costs of our parallelism we introduce the notion of *speedup* and *efficiency*. We define the speedup $S(c)$ to be how much faster a progam performs when utilizing multiple computing units.

$$S(c) = \frac{T_1}{T_c} \tag{4.1}$$

Where:

- T is the observed execution time.
- c is the number of discrete computing units.

The speedup tells us the attainable performance gain when compared to using only a single computing unit. The efficiency E is the time spent performing productive operations. It can be defined as a function of the speedup:

$$E(c) = \frac{S(c)}{c} \tag{4.2}$$

Efficiency decreases from the ideal value of 1, due to the overhead associated with using multiple processing units and interconnecting them.

4.2 Performance Modelling

The complexity of modern computing platforms is a great gain in terms of performance, but makes it difficult to reason about how an application will perform. Even individual computing units can be regarded as complex systems with many features intended to speed up the computation. Processors utilize pipelining to progress on multiple instructions simultaneously [23]. Memory is cached using multiple intermediate layers, in order to balance access speed with the expense of cache capacity [24]. Prefetching further cuts down on the cost of memory accesses, be retrieving information just before it is requested [25]. Communication can be delegated to lower level hardware systems [26]. These boons

come at the expense of our ability to reliably reason about aspects of a program's execution, such as execution time. At some point there are too many involved mechanisms to easily understand them and their mutual interactions.

Performance models provide a layer of abstraction. They may sacrifice some accuracy and resolution in return for a manageable reduced parameter space.

Developed models may give developers insight which help in tuning and scaling the application. Models may also help users estimate the required computing resources required to perform a given task. Additionally, models may be used by a platform's scheduler to efficiently allocate resources such as compute nodes and cores.

Jingwei Sun *et al* describe three general types of performance models [27]:

1. **Analytical models** These describe an application's performance using arithmetic formulas. This category of model requires in-depth understanding of the application in question, but may offer quick results for relatively simple programs.

2. **Replay-based models** By recording the execution of the application that is to be modelled, one may construct a synthetic replica which models parts of it and reproduces the same behaviors and strains on the system. This approach is limited by the fact that it may require a preventatively large amount of storage for the recordings, and that it can reproduce specific execution paths.

3. **Statistical models** Machine learning is used to take the replay-based model one step further. After training the model provides a compact way to predict an application's performance. The model can then be given new input parameters, to see how the real application will perform. These parameters, however, represent a limitation of the range of this approach. Some aspects, such as non-scalar input and dynamic tuning, may be very difficult to capture.

Additionally, Williams *et al* describe an higher-level approach called *bound and bottleneck analysis* [28]. This approach is rooted in the idea that the performance of most applications can be traced back to small set of bottlenecks. The authors present the *Roofline Model*, which maps the attainable computational performance into a memory-bound region and a compute-bound region. The modelled program is limited by the memory bandwidth until it reaches a platform-specific compute-operations to memory access operations threshold. After this it is the speed of the processor which determines performance.

We choose an analytical model for our purposes, since the developed proxy-application is small enough to be comfortably analyzed by the developer. Our model starts with the "fundamental equation of modeling" described by Barker *et al* [29]. This simple model splits the execution time of an application into three contributing categories:

$$T_{total} = T_{comp} + T_{comm} - T_{overlap} \tag{4.3}$$

That is, the total execution time is given by the time required to compute the local solution, the time spent on exchanging results with other computing units, and the degree to which the former two may be overlapped. Here the cost of initialization is included in T_{comp}. We return to our modelled values of T in Section 5.6, after introducing the features of our application on which the model depends.

4.3 Domain Decomposition

In the realm of mathematics the process of decomposing may refer to splitting a boundary value problem, such as ours, into multiple smaller ones. This subdivision can make an otherwise insurmountably large computational problem possible by leveraging HPC and parallelism. The problem is first split into multiple sub-problems, which may be computed individually by a processing unit, and then later recombined for a complete solution.

Many decomposition strategies exist, depending on the problem at hand. Decompositions may be *overlapping* or *non-overlapping*, depending on whether parts of one partitions are made up of section from another.

I.M Navon and Y. Cai outline four reasons for decomposing a domain [30], which may be summarised as follows:

1. The domain decomposition technique may be used to exploit the geometry of irregular domains.

2. Decompositions can be used both to apply different resolutions and different methods to specific sections of a domain. In doing so, one can increase the results level of detail in regions of particular interest or combine the benefits of multiple methods in the same solution.

3. An appropriate method may be selected for each partition, based on special conditions within that sub-problem, such as singularities large gradients.

4. Specific properties can be applied to the phenomena simulated in certain partitions. For fluid simulations, such as ours, this may include viscosity or compressibility.

The first of these is our main interest in this work, as further outlined in Chapter 5. Our rectangulation techniques described in Chapter 3 are suitable for creating our BSP-style decompositions, where the majority of the processed data remains local to a processing unit.

4.4 Communication Models

Our decomposition strategy has a clear expected effect on the computation time, which is an evening out of T_{comp} among all computation units. Its effect on the communication time, however, is not as clear. We must determine whether or not the increased complexity of the sub-domain layouts has a tangible effect on how communication is done. Communications are difficult to predict, as their behavior are a function of multiple tiers of software and hardware, as well as the behaviors of the application itself. To assist with this, we employ a communication model, which serves as an abstraction.

4.4.1 The LogGP Model

Numerous models for parallel computations exist, one of them being the *LogGP* model [31]. With it, we may make predictions about the time required to transmit data between two communicating processes in one direction. This model expands upon the LogP model

of Culler *et al* [32] by taking into account that the size of transmitted messages may vary and thereby influence transmission performance. The LogGP model requires five parameters in order to be able to make predictions:

- **L:** The latency of the a transmission represents the time a unit of information spends in the network between the sending and receiving ranks. L is given as an upper bound, *i.e.* the worst-case scenario.

- **o:** The act of setting up a data transfer carries an overhead, which is captured by o. o represents the time the CPU is busy with communication only. This includes time needed to initialize communications, designate a sending or receiving buffer, and interacting with the network interface. o may be split into a sending overhead o_s and a receiving overhead o_r, which need not be identical. We use the average of the two as our o, due to our symmetric communication pattern, described in Section 5.5, where each send implies a corresponding receive operation.

- **g:** In many situations multiple messages are sent in bursts. g is the gap in time between the end of a transmission and the start of the next. g represents the startup bottleneck of the network, which is created by actions such as the opening of a communication channel. The inverse of g corresponds to the modelled per-processor network bandwidth for short messages.

- **G:** Similarly to the gap between messages g, there also exists a gap in time between the transmission of individual bytes which are part of the same message. This gap is represented by G. The inverse of G corresponds to the modelled per-processor long message bandwidth.

- **P:** The number of processes partaking in the communication. For one-to-one communications such as MPI_Send() and MPI_Recv() this parameter will intuitively be equal to 2. Other operations, such as MPI_Bcast(), may have a much larger number of processes.

With a LogGP model, a single one-to-one communication may in terms of time cost be expressed as:

$$o + (s - 1)G + L + o \tag{4.4}$$

Where:

- s is the number of bytes in a single message.

LogGP parameters depend on the computing platform which facilitates the communication. Primarily, these are determined by processor, memory, and interconnects used by the system. Hoefler *et al*. propose a fast technique for the measurement of the LogGP parameters on HPC platforms [33]. Similarly to other approaches, the authors implemented a ping-pong benchmark where messages are passed back and forth between two nodes. We recount Hoefler's method here, as it is used for our benchmarking measurements in Section 6.2.4.

The *Round Trip Time (RTT)*, which is the time it takes for a message to travel from one node to the other and back, is measured on only one of the two nodes. This way there is no need for micro-second accurate time synchronization between the nodes. The RTT is parameterized (PRTT) by noting that it depends on the message's size s, the number of messages i, and the artificial delay between sent messages d. $PRTT(i, d, s)$ is then measured for

Using Equation 4.4 we can see that the time needed for a single round trip $PRTT(1, 0, s)$ must be:

$$PRTT(1, 0, s) = 2 \cdot (L + 2o + (s - 1)G) \tag{4.5}$$

We call the sum of transmission gaps of a single on-way transmission G_{all}:

$$G_{all} = g + (s - 1)G \tag{4.6}$$

This leads to i transmitted messages without delay incurring a PRTT of:

$$PRTT(i, 0, s) = 2 \cdot (L + 2o + (s - 1)G) + (i - 1)G_{all} \tag{4.7}$$

Where:

- we require $o < G_{all}$

Equation 4.5, when combined with Equation 4.7 yields:

$$PRTT(i, 0, s) = PRTT(1, 0, s) + (i - 1)G_{all} \tag{4.8}$$

We may extend this to a general expression for for $PRTT(i, d, s)$, and then rewrite it further to gain an expression for o:

$$PRTT(i, d, s) = PRTT(1, 0, s) + (i - 1) \cdot max(o + d, G_{all}) \tag{4.9}$$

$$\frac{PRTT(i, d, s) - PRTT(1, 0, s)}{i - 1} = max(o + d, G_{all}) \tag{4.10}$$

$$o = \frac{PRTT(i, d, s) - PRTT(1, 0, s)}{i - 1} - d \tag{4.11}$$

where:

- a d is selected such that $d > G_{all}$

In order to determine g and G we may rewrite Equation 4.8 and expand G_{all} such that:

$$G(s - 1) + g = \frac{PRTT(i, 0, s) - PRTT(1, 0, s)}{i - 1} \tag{4.12}$$

This is a linear function, which may be fitted using the least squares method.

Finally, L is approximated with:

$$L \approx \frac{PRTT(1, 0, 1)}{2} \tag{4.13}$$

This is done because L and o overlap, as some of the operations associated with the overhead are performed in parallel with data transiting through the network. As such, L may not be measured directly.

A part of the ping-pong benchmark is shown in Figure 4.1. The described technique has, in addition to the speed of the instrumentation, the added benefit of not relying on flooding the network, which other approaches may do. Taking the measurements should therefore not have significant negative impact on similar measurements and operations performed by other processes on the platform. Furthermore, o may be computed from a single measurement and without relying on other measured LogGP parameters. The graph for G_{all} is useful for revealing protocol changes in the underlying network, which manifest as discontinuities.

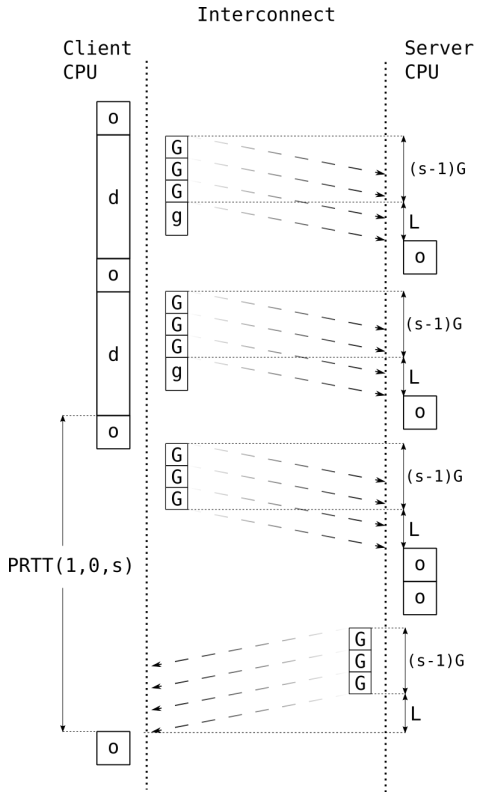


Figure 4.1: A single ping-pong iteration for $i = 3$ in the synthetic LogGP benchmark, provided by Netgauge [34].

4.5 Parallel Programming APIs

Similarly to the domain, the program itself must be adapted to functioning in a parallel manner. In order to accomplish this the program must provide a unit for parallel execution and potentially also a way for these to communicate between one another. The units may be threads, processes, or entire nodes.

OpenMP and MPI are two popular software solutions for incorporating the management of these units in software applications. The former specializes in the use of threads in shared-memory systems, the latter in separate processes and processes running on multiple nodes. Hybrid programs which incorporate both are also a popular choice. We provide additional details about OpenMP and MPI in the next sections, as we implement such a hybrid in our application.

4.5.1 OpenMP

> OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs.[35]

Open Multi-Processing (OpenMP) [36] provides an API which simplifies the use of *threads* in order to achieve parallelism. On most platforms this means the use of POSIX threads, also referred to as *pthread*. OpenMP can be used for parallel operations within the same process, utilizing the available cores to execute independent subproblems in separate threads. OpenMP threads exchange data using shared-memory operation, which is very efficient, but restricts the communication to occurring on the same address space. This space is usually only available locally within a single node.

4.5.2 MPI

The *Message Programming Interface (MPI)* is a standard for message passing between multiple concurrent processes [37] . It may be used in cases where local techniques such as shared memory cannot be used, for instance when multiple separate nodes with distributed memory participate in the same computation. Each process represents a running instance of the same program, which is referred to as a *rank*. MPI can be used to split the computational load among multiple nodes on multi-core and multi-node systems.

MPI provides the opportunity to easily use a number of communication patterns between ranks. The simplest of them are the *Point-to-Point (P2P)* functions, which facilitate the exchange of information between exactly two ranks. MPI_Send() and MPI_Recv() are the most basic basic functions for P2P communication. When one rank calls its communication function, it expects its peer to call the corresponding opposite. When both are engaged in the data exchange the target data is transmitted from one to the other. MPI_Isend() and MPI_Irecv() are variants of the basic send/receive functions which are explicitly non-blocking. Once executed, these functions return once the data transfer has been engaged and before it has been completed, which allows the rank to perform other actions in the meantime. These methods are useful for two reasons: They allow a certain degree of overlap in communication and computation, and they are useful for the purpose of avoiding deadlocks, which may occur if a circle of blocking ranks forms.

MPI also provides a set of collective functions, which provide simple and efficient ways of performing one-to-many, many-to-one, and many-to-many communications. The collective functions include synchronizing operations such as barriers.

MPI does not refer to a single program. Multiple implementations of the MPI standard exist, such as Open MPI, MPICH, and numerous commercial ones. We focus on Open MPI and settle on this implementation for our application.

4.5.3 Open MPI

Open MPI is a widely adopted open source implementation with a modular architecture. It can be split into three main functional areas, which form layers in the MPI transmission stack [26, 38]:

1. **MCA:** The *Modular Component Architecture (MCA)* provides management services for the other network transmission layers. It detects available components at build time, and accepts run-time parameters from higher level abstractions such as the *mpirun* and *srun* command line tool command line tools.

2. **Component Frameworks:** Frameworks provide specific services to the Open MPI installation. They logically group together components and manage modules.

3. **Components:** Components are self-contained software units with well-defined interfaces.

Table 4.1 gives an overview of a selection of relevant Open MPI component frameworks and their roles.

Framework	Full Name	Purpose
PML	Point-to-Point Messaging Layer	Implements the logic for MPI communications between two peers
BTL	Byte Transfer Layer	Responsible for data delivery between two network interfaces
BML	BTL Management Layer	Discovery, management and sharing of BTLs with upper layers
COLL	Collective Communication	Backend for one-to-many, many-to-one and many-to-many communications
MPool	Memory Pool	Memory allocation/deallocation services, shared among layers
TOPO	Process Topology	Cartesian and graph functionality, allows topology and locality-based optimizations

Table 4.1: A non-exhaustive list of Open MPI frameworks and their purposes.

Open MPI utilizes the *openib* BTL for communications across the Infiniband interconnects used by the larger platforms described in Section 6.2. This component determines how the data exchanges take place, for instance the choice of protocol. It manages connections dynamically, estabilishing new ones as the ranks send messages to a peer for the first time. Two protocols are supported by the Open MPI PML:

- **Eager Protocol**, which is a low-latency option used to transmit short messages.
- **Rendezvous Protocol**, which is a high-bandwidth option used for larger messages.

Eager protocol transfers may utilize *Remote Direct Memory Access* (RDMA) for its data transfers. In these on rank is able to directly write to a memory region owned by the remote host, without that host's CPU being involved. This style of transfer also avoids context switches and caching. The Infiniband implementation supports two-sided send and receive operations, as well as one-sided put-get operations for these transfers [26]. Use of RDMA requires preallocating and registering the utilized memory regions. This poses a limit to its scalability and means that the use of RDMA ordinarily is restricted, for example by only adding new connections up to a specified limit.

The Rendezvous protocol may also be implemented with RDMA capabilities [39], but is slower because it involves a negotiation process to determine the receiver's buffer availability. This incurs additional RTT times.

We show Openib settings relevant to the choice of transfer protocol in Table 6.4.

Chapter 5

The Proxy Application

We develop a proxy application for solving the shallow water equations numerically on a heterogeneous domain. It is based on the MacCormack method described in Section 2.4.1 and supports both Cartesian and non-Cartesian decompositions. In this chapter we describe this application and develop an analytical communication model based on its traits and the model described in Section 4.4.1. The developed proxy application is the subject of our studies in Chapter 6.

5.1 Goal

Proxy applications are simplified HPC applications which are simpler to analyse. They address the fact that production-grade HPC applications can be highly complex, which makes their performance difficult to reason about. By mirroring the full-size application's performance bottlenecks inside of a proxy application we can study them and find optimizations at a lower cost. Proxy applications and similar concepts are sometimes also referred to as *MiniApps*.

Every proxy application represents a trade-off between suitability, stability, accuracy, and scalability. Our proxy application mirrors the *Bulk Synchronous Parallel* behavior seen in many HPC applications, where ranks perform a relatively intensive local computation step, followed by a data exchange and synchronization stage. With this we can examine how the cost of processing compares to the cost of data exchanges, and how the data exchange between ranks takes place.

5.2 The Grid

We assign one rectangular sub-domain of the global domain to each rank. This sub-domain corresponds to a *grid*. A grid contains a set of arrays which in turn hold the data that makes up the bulk of the area's state. Each array entry represents a value belonging to a point within or along the rim of the domain. The stored state includes the points' fluid heights

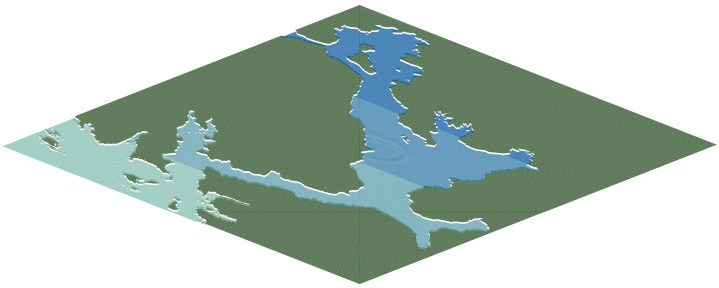
and velocities, as well as partial results of the integration steps. For the purpose of the simulation, each point represents a coordinate on a dense Cartesian grid, where each point is separated from the next by a constant Δx and Δy. The grid corresponds to the pixels in our input map, such that one pixel is one grid point. Figure 5.1 shows how the conversion from map to grid data takes place.

We define boundaries to be the perimeter regions of the full domain, and borders to be the regions between adjacent grids.

We limit ourselves to flat domains without slopes or varying depths, and scenarios where the bulk of the total fluid mass is at rest.

(a) Map export from ©Kartverket [40], Geonorge N250Raster 132 and 133, EURF89 zone 33, 2d, TIFF.

(b) Converted bitmap file. Areas colored black are treated as land, white areas are fluid regions.



(c) A rendering of a simulation performed on the derived domain. The center contains a small perturbation. The different shades of the fluid indicate the domains of the participating ranks.

Figure 5.1: The Trondheimsfjord, converted into a bitmap used as input for ours simulations.

5.3 Load Balance

Since computation and communication occur in sequence, the next computation step may not begin before a rank has finished communicating with all of its neighbors. It follows that an uneven load balance may cause one rank to hold back others and cause CPU cycles to be wasted.

Imbalances may be caused by a number of effects, depending on the implementation and features of the application in question. One may for instance choose to refine the resolution of specific areas through *Adaptive Mesh Refinement* [41–43]. We introduce a simple difference in computational load of individual points through the introduction of non-fluid points, hereafter referred to as *land points*. Land points are points where no fluid flows. These behave as infinitely tall solid walls in the context of our fluid simulation.

During the simulation, the computational cost of each individual point is determined by the number of *Floating Point Operation (FLOP)* and the number of *Memory Access (MA)* needed to compute the new height value of that point in a given iteration. The FLOPs are associated with performing mathematical computations, while the MAs are a result of fetching the grid values required by the stencil from memory.

Utilizing the Roofline model, we reason that the number of FLOPs compared to the number of MAs is so low that the application is *memory bound*, such that the performance is limited by the node's ability to fetch data from memory. For this reason, we adopt the number of MAs required to compute a given point as a measure of it's workload. We find this number through code inspection and show the result in Equation 5.1.

$$f(\tau) = \begin{cases} 68 & \tau = fluid \\ 11 & \tau = land \end{cases} \tag{5.1}$$

Memory accesses associated with land points stem from the act of checking the given point's state. Through the introduction of Chunking, as described in Section 5.4.1, the presented count becomes an upper bound instead of an exact value.

5.3.1 Semi-static Map Parsing

The geography of a grid determines which points are defined either as 'fluid' or 'land'. It is defined by an $n \times m$ bitmap file, which we supply to the application. We see three different approaches to this:

1. Statically, where the map is provided at compile-time.
2. Semi-statically, such that the map is read at the initialization stage of the program's execution.
3. Dynamically, which allows for the map to be re-read and applied arbitrarily at run-time.

We utilize a semi-static implementation, which allows us to switch between maps without having to re-compile the application.

Each rank reads the segment of the bitmap corresponding to its sub-domain during the initialization process of a grid. First, an integer grid that corresponds to the values of the

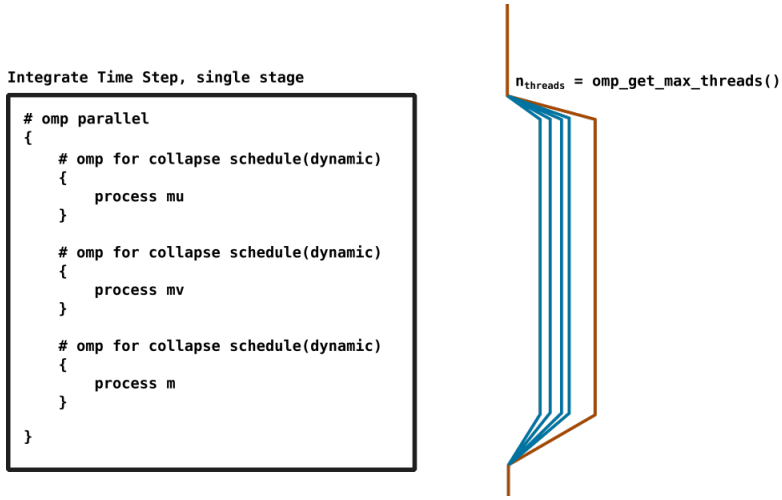


Figure 5.2: A single stage of our time step integration procedure, which may be either the predictor or the corrector stage. We use OpenMP to turn the local processing task into a pool of parallel subproblems. These are then assigned to available threads.

bitmap is created. This grid is then analyzed to create a secondary grid, which reflects the state of each point's neighbors. Our implementation stores the state of directly adjacent points, as well as points on the target's diagonals. A border exchange of the first array is required in order to accurately capture the neighbor state of points along the rank's borders.

5.4 Parallelism

The bulk of our application's processing time is spent on solving the SWE and updating the local state, and not on communications. This limits our sampling rate and significantly increases the cost of gathering them. We can artificially skew the total time consumption in favor of communications by altering the domain, for example by creating very long, thin sub-domains. Doing that, however, could jeopardize the realism of the application. We might, for instance, no longer use the same caching and prefetching patterns as the applications we seek to model.

Instead, we reduce the computation step time by efficiently utilizing the resources available on our platforms. Each platform, as described in greater detail in Section 6.2, has a given number of cores available on each node. We can involve these in the processing steps by parallelizing the main data loops using OpenMP. Each spawned OpenMP thread is assigned part of the loop to process by itself. Furthermore, we can split the operations performed per-point into separate loops, thereby increasing the number of tasks which can be run in parallel. Figure 5.2 shows how we do this.

5.4.1 Chunking

The processing cost of land points in the domain may be further reduced by disabling larger accumulations of them. This reduces the number of memory accessed needed to check the points' state from one per point to one per accumulation. We split the sub-domain of each rank into $n \times n$ *chunks*, which have a state which is either active or inactive. Chunks which contain no fluid points are set to be inactive, which means that their points are skipped during processing.

We implement the chunks ourselves, instead of relying on the OpenMP's loop chunking, which allows us to control their state. The *collapse* directive shown in 5.2 combines the iteration across our x-y chunk grid into a single loop. Chunks are assigned to threads using OpenMP dynamic scheduling. This results in an unprocessed chunk being assigned with a "first come first served" heuristic, which is appropriate since the workload represented by a chunk may vary from one to the next. The optimal chunk size depends on the distribution of land points in the domain. We settled on a chunk size of 50 by 50 pixels for our experiments, as it was found to perform well in the general case.

Figure 5.3 demonstrates what a 5×5 chunking would look like on an 8×8 grid.

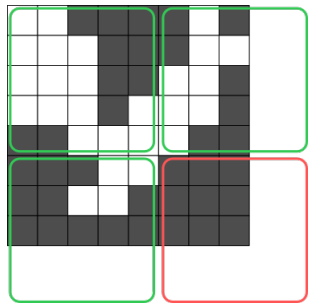


Figure 5.3: The grid of fluid points (white) and land points (black) is processed chunk by chunk. The bottom right chunk is disabled, as it contains no fluid points.

Chunking may be enabled or disabled at compile-time.

5.5 Ghosts Points and Interfaces

It is of interest to enforce a strict set of conditions where fluid points are adjacent to the domain's outermost edge, despite our ability to draw arbitrary domain layouts. We do this through the introduction of *ghost points*, similarly to the ghost cells described by Randall J. LeVeque [44]. Ghost points are a set of artificial points which line the rim of each grid. Each point stores the same values for depth, velocity, *etc.* as a regular point, which in turn may be accessed by the stencil of neighboring points during the integration step. They are, however, not updated by the numerical method each iteration, as one would update a regular point. For this reason we consider our domains to be non-overlapping.

Ghost points have two uses: In areas where the ghost points lie at the boundary of the global domain we apply our selected boundary condition to them. This way the desired

effect is transferred to adjacent fluid, without affecting the rendered domain. Ghost points along the border of two grids serve as buffers for their *interfaces*.

A grid has an interface for each of its neighboring grids, as shown by Figure 5.4. The interface contains a reference to the neighboring grid's MPI rank and a set of offsets which specify sources and destinations for the data exchange. Grids transfer fluid data from the line of regular points that is parallel to the referenced neighboring grid. Received data is stored in the adjacent, parallel line of ghost points. This way the ghost points emulate the domain of the neighboring grid, producing the effect of a contiguous domain. The data transfer is done in bulk, one column or row of a given value at a time.

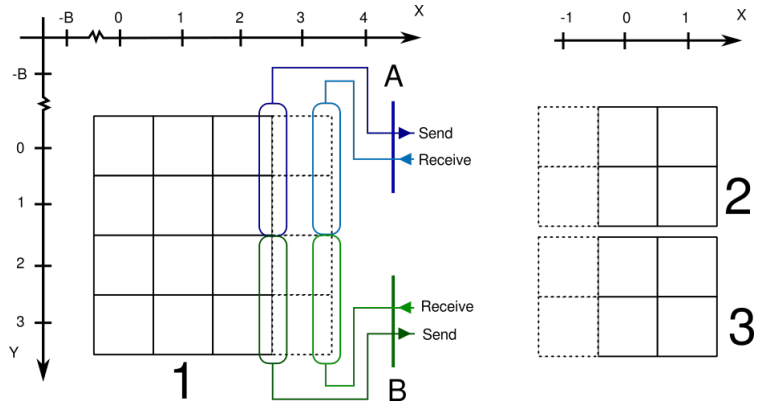


Figure 5.4: Grid 1 has two neighboring grids along the y axis, 2 and 3. The interfaces A and B specify where to transmit data to and from during the communication steps. Dotted lines show the location of a border where ghost points are located. The variable B on the axes refers to the border width, which is equal to 1 for our implementation.

5.5.1 Collision Handling

We define the boundaries of the domain and all individual land points to be *hard obstacles*, that is, immovable obstacles of infinite height. Collisions between waves and obstacles can therefore occur at any place where fluid points are adjacent to either outer boundaries or land points.

We implement the reflection of waves using the Neumann boundary condition as given by Equation 2.17. For any point adjacent to a hard obstacle, we set the simulated velocity of that point to 0 in both north/south and east/west directions. Diagonally adjacent points are counted as adjacent as well. We reflect waves without dampening in our proxy application, although a dampening factor can easily be applied when desired. Figure 5.5 shows how our fluid interacts with a complex domain.

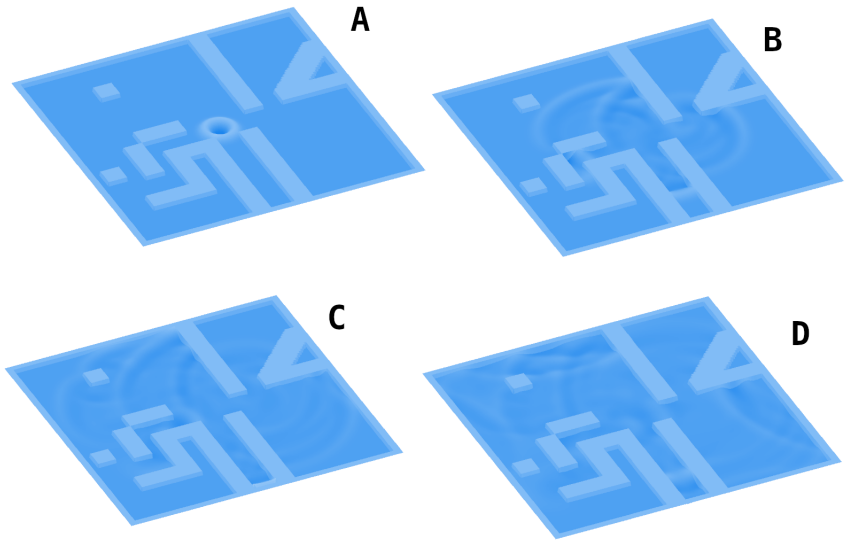


Figure 5.5: An initial condition with a disturbance in the center. Fluid is reflected off solid walls.

5.6 A Model for the Application

By building upon the modified fundamental equation, see Equation 4.3, we can construct a performance model for our application.

5.6.1 Initialization Cost

The initialization cost depends primarily on the reading of the input files and the initialization of the MPI topology. The former is an IO-heavy operation which scales with the domain size. It is, however, a one-time operation, such that it may be amortized after a number of iterations. The topology is initialized using the MPI_Dist_graph_create_adjacent() operation, which creates a distributed graph. This operation scales well, due to the fact that each rank only needs to be aware of its own neighbors, not the global topology. We refer to Hoefler *et al* for a more detailed comparison of distributed graphs and globally defined graphs [45]. Furthermore, at the investigated scales our decompositions create no neighborhood graphs which should challenge this. We therefore disregard this initialization cost as well.

5.6.2 Computation Cost and Overlap

With our cost function $f(\tau)$ and our knowledge of the communication system 4.5.2 we expect the bulk of the execution time to be spent on the computation of the local domain, which is captured by T_{comp}. The local computation time of a rank is expected to scale proportionally to the size of its subdomain. The global computation time of the program as a whole depends on the size of the global domain, as well as the load balance between

the ranks.

Due to our BSP-like architecture $T_{overlap}$ may be disregarded. There may be a small degree of indirect overlap, stemming from the fact that our ranks only synchronize communications with their direct neighbors in the topology, as opposed to the whole communicator. The graph topology's dependencies should however keep this in check. A slow rank will hold back its direct neighbors, which in time will hold back their neighbors and so on. We include no explicit overlap between communications and computation.

5.6.3 Communication Model

This leaves us the construction of a model for how the communication time T_{comm} develops, which must be a function of both domain size and decomposition. Ranks communicate to their direct neighbors through their corresponding interface. The transmission contents are unique for each rank pair. Communication therefore relies on MPI's single send and receive operations, rather than collective methods such as broadcasts. The only MPI collective we make use of after initialization is the Barrier function, which synchronizes the ranks for the purpose of instrumentation.

From the LogGP model presented in Section 4.4.1 we may devise a performance model for these communications. Remember that with LogGP a single send will have a cost given by Equation 4.4:

$$o + (s-1)G + L + o \tag{4.4}$$

Each rank may have an arbitrary number of neighbors P_{neigh} with which it must exchange border values. Each neighbor represents a one-to-one communication. We also observe that regions along the outer boundary of the domain need not exchange boundary values with any other rank.

The interface-based proxy application uses MPI Isend and Irecv calls for their communication. By using nonblocking routines we avoid communication deadlocks that could arise as a consequence of cycles in the domain's neighbor graph. Every communication cycle participating ranks initiate their transmissions by issuing all Isend calls in bulk, followed by the corresponding Irecv calls and a wait for these to complete.

We may from the LogGP model derive an expression that allows us to estimate the time each rank spends on communication per iteration:

$$T_{comm} = \left((P-1) \cdot g + \sum_{i=0}^{P} (2 \cdot o + L + (s_i - 1) \cdot G) \right) \cdot n_{comm} \tag{5.2}$$

where:

- s_i is the number of bytes of information along the length of the given interface.
- n_{comm} is the number of interface exchanges per iteration. For our application n_{comm} is 14, as shown in Figure 5.7.

5.7 Application Structure

The developed proxy application executes as shown in Figure 5.6.

The initialization stage accepts our input parameters and data files. The rectangle layout file specifies the number and domain dimensions of the participating ranks. Each rank then reads the interfaces file, which tells it what ranks to communicate with and what data to exchange. Based on this information we create an *MPI Distributed Graph*, where each rank specifies its neighbors for the creation of the topology. This makes MPI aware of which ranks will interact and allows for optimizations in the newly spawned communicator. After this each rank reads the geography of its local domain from the map file and initializes the local state. This state includes an initial condition for the fluid levels, which allows us to describe waves and other disturbances.

During the execution phase we run the MacCormack method in order to solve the shallow water equations numerically. The main execution loop calculates the desired number of time steps sequentially. We may periodically take a snapshot of the domain's current state at the beginning of this loop. The next three stages update the partial results based on the current height and velocity values, perform the predictor-corrector integration step, and finally update the height and velocity values.

The execution is completed by outputting the results of our instrumentations.

Communications between ranks are as shown in Figure 5.7. Each exchange involves the data values required by the next computation stage and overlaps the sending and reception stages. We exchange vertical interfaces first, followed by the horizontal interfaces in a separate stage.

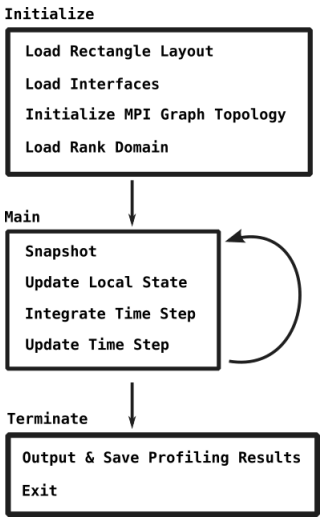


Figure 5.6: Execution stages in a run of our proxy application.

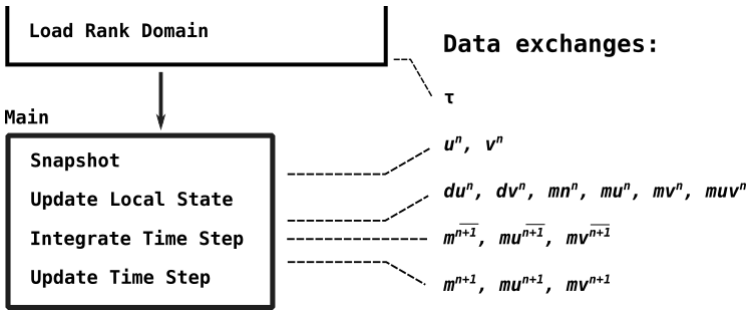


Figure 5.7: Data exchanges taking place between neighboring ranks at various stages of program execution. Values on the right side are intermediate results which are exchanged along the borders of two adjacent domains at various stages of execution.

$$T_{total} = T_{overlap} + T_{comp} + T_{comm}$$

$$T_{overlap} \approx 0$$

$$T_{comp} >> T_{comm}$$

$$T_{comm} = ((P-1)\cdot g + \Sigma(2\cdot o + L + (s-1)G)\cdot n_{comm}$$

Figure 5.8: A summary of our developed performance model, based on the features of our application, our assumptions, and the LogGP model.

Chapter 6

Methodology

We determine the potential gains and costs of our decomposition technique experimentally. The potential gain lies primarily in dividing the total workload represented by a heterogeneous domain more evenly than a traditional Cartesian layout can. Potential costs include the time spent on the decomposition itself, as well as the additional communication overhead that is introduced. We perform a series of experiments to verify the functionality of our implementation, the efficiency of our load balancing and the accuracy of our communication model. These are conducted on multiple HPC platforms: Idun, Fram, and Betzy.

6.1 Experiments

Our experiments utilize a provided open source toolchain based on GCC, OpenMPI and, OpenMP. Temporal instrumentations rely on the function omp_get_wtime(), which is provided by OpenMP. Experiments on Idun and Fram execute 2500 iterations. On Betzy this number is doubled to 5000, in order to compensate for the reduced execution time which follows from the greater availability of cores.

Property	Value
η_{steady}	1.0
$max(\eta_{wave})$	0.01
dt	0.004
dx	0.1
dy	0.1

Table 6.1: Important parameters which are shared by the experiments.

The following experiments were performed on our proxy-application:

6.1.1 Decomposition

Decomposition Scalability

We decompose a domain using our scanning and diagonal decomposition algorithms. The time required to create these decompositions is measured, and we determine when it becomes impractical to perform the full minimization. Since our decomposition algorithms are not parallelized they are unsuitable for execution on our HPC platforms. We therefore perform our decomposition tests on an ordinary workstation with an *Intel(R) Core(TM) i7-4610M 3.00GHz* CPU.

In addition to this we make observations on the decomposition layouts we receive in practice. The layout determines the number of neighbors each rank will communicate with. The theoretical increase in communicating neighbors permitted by the scanning and diagonal rectangulations could lead to communication-based load imbalances. An example of this could be one rank being placed in the center such that it neighbors all other ranks. The produced decompositions should give an indication of whether or not these concerns are warranted.

Reduction of Load Imbalance

We measure the time each rank spends waiting for its peers before performing a data exchange. In doing so we gain an estimate of how well the computational load of the ranks is balanced, since ranks with less workload will finish faster and spend more cycles waiting. A comparison of the load balance when using the Cartesian, scanning, and diagonal decompositions allows us to see whether or not the techniques produce a reduction of required compute time. Each decomposition technique is tested on the same heterogeneous domain, with a varying number of ranks. We expect the per-rank imbalance to decrease as the assigned sub-domain size decreases. For this experiment we disable the chunking feature, since our estimate of point cost $f(\tau)$ does not account for it. Figure 6.1 shows the map used for thes experiment. Its fluid to land ratio is approximately 2:3.

Figure 6.1: Load imbalance tested on the 1800×1000 point Trondheimsfjord map.

6.1.2 Communication

Verification of Stable Communication Cost

Our communication model predicts that the time spent on communications should be constant over time. We ascertain that this is the case. To do so, we set up a simple simulation where the domain is divided among two ranks which communicate through a singe interface. We measure the total communication time of each rank during each iteration, and compare it to the others.

Strong Scaling Comparison of Rectangulations

We compare the communication cost of regular Cartesian decomposition and irregular decomposition. For this experiment we keep the problem size constant and scale up the available computing resources. This translates to maintaining the size of the domain and adding more nodes, each of which runs a single process. A regular and irregular decomposition is computed for each rank count. We then examine how the total time spent on communications between ranks changes as the number of exchanges increases with the rank count, and the size of exchanged messages decreases. By inserting Equations 6.1 and 6.2, or Equations 6.3 and 6.4, into Equation 5.2 together with our interface sizes we obtain a prediction for the communication cost. In our case the communication times should not exceed 1 second, a small fraction of the total run time, for any of the tests. The size of the selected 1800×1800 point domain limits the maximum possible single message size to 14400 bytes. Figure 6.2 shows the used map for this experiment.

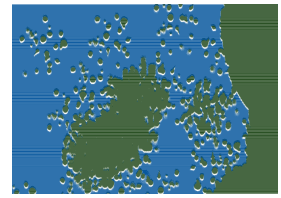


Figure 6.2: Strong scaling tested on a custom 1800×1800 point archipelago map.

6.2 Computation Platform

6.2.1 Idun

Idun is a rapid testing and prototyping HPC cluster hosted at the Norwegian University of Science and Technology [46]. It is a collaborative effort between various departments and faculties, which contribute by providing the hardware that makes up the machine.

The hardware and software specifications of Idun related to our computations are shown in Table 6.2.

Property	Value
Compiler version	mpicc - gcc 8.3.0
Compiler Flags	-std=c99 -g O3 -fopenmp -lm -lnetpbm -lgomp
MPI version	OpenMPI 3.1.4
OpenMP version	4.5
Processor	Intel Xeon E5-2630
Processors per node	2
Cores per node	20
Memory per node	120 GB
Interconnect	3 Mellanox passive FDR switches
Launcher	srun

Table 6.2: Hardware and software properties of the experimental setup on Idun

6.2.2 Fram

Fram is a large-scale parallel distributed-memory HPC platform hosted at UiT Arctic University of Norway. It is part of Sigma2 Metacenter collaboration and was brought online in October 2017.

Property	Value
Compiler version	mpicc - gcc 8.3.0
Compiler Flags	-std=c99 -g O3 -fopenmp -lm -lnetpbm -lgomp
MPI version	Open MPI 3.1.4
OpenMP version	4.5
Processor	Intel E5-2683v4, 2.1 GHz
Processors per node	2
Cores per node	32
Memory per node	64GB
Interconnect	Infiniband, island topology
Launcher	srun

Table 6.3: Hardware and software properties of the experimental setup on Fram

Setting	**Value**	**Description**
btl_openib_eager_rdma_threshold	16	Use RDMA for short messages after this number of messages are received from a given peer
btl_openib_message_coalescing	false	If nonzero, use message coalescing
btl_openib_max_eager_rdma	16	Maximum number of peers allowed to use RDMA for short messages
btl_openib_eager_rdma_num	17	Number of RDMA buffers to allocate for small messages
btl_openib_use_async_event	true	If nonzero, use thread that will handle InfiniBand asynchronous events
btl_openib_eager_limit	12288	Maximum size (in bytes, including header) of "short" messages

Table 6.4: Relevant openib configuration options on Fram and Betzy.

Mellanox Island Topology

The nodes of Fram are placed in groups referred to as islands, shown in Figure 6.3. Nodes on the same island are interconnected using Infiniband with a two-level fat tree topology. There is a limited connection between islands, but jobs are under ordinary circumstances scheduled such that their ranks land on the same island. Fat tree topologies, as described by Al-Fares *et al* [47], have historically been a popular choice. On Fram one can expect the maximum theoretical bandwidth between two nodes in the same island to be 100Gbits/s. The interconnect network is however implemented with some oversubscription, which

may cause the effective bandwidth to decrease in situations with many communication-heavy jobs.

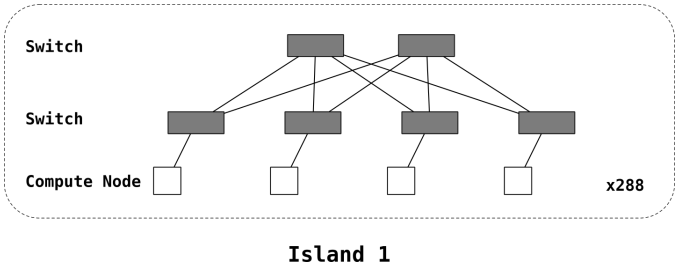


Figure 6.3: Nodes on Fram are organized in an island topology. Nodes in the same island are connected using a 2-level fat tree topology.

6.2.3 Betzy

Betzy is the most recent machine of the Sigma2 Metacenter collaboration. It is also a large-scale parallel distributed-memory system, and it has been operational since 2020. Betzy is located at NTNU Trondheim.

Property	Value
Compiler version	mpicc - gcc 8.3.0
Compiler Flags	-std=c99 -g O3 -fopenmp -lm -lnetpbm -lgomp
MPI version	OpenMPI 3.1.4
OpenMP version	4.5
Processor	AMD Epyc 7742, 2.25GHz
Processors per node	2
Cores per node	128
Memory per node	256GB
Interconnect	InfiniBand HDR 100, Dragonfly+ topology
Launcher	mpirun

Table 6.5: Hardware and software properties of the experimental setup on Betzy

Dragonfly+ Topology

Betzy's compute nodes, of which there are 1344 in total, are interconnected using an Infiniband Dragonfly+ topology. The Dragonfly+ topology is a modern, highly scalable configuration which Shpiner *et al* [48] describe in greater detail. Figure 6.4 and 6.5 illustrate how the routers and nodes are connected. Connections follow the HDR-100 standard, which allows for a theoretical bandwidth of 100 Gbits/s between two nodes.

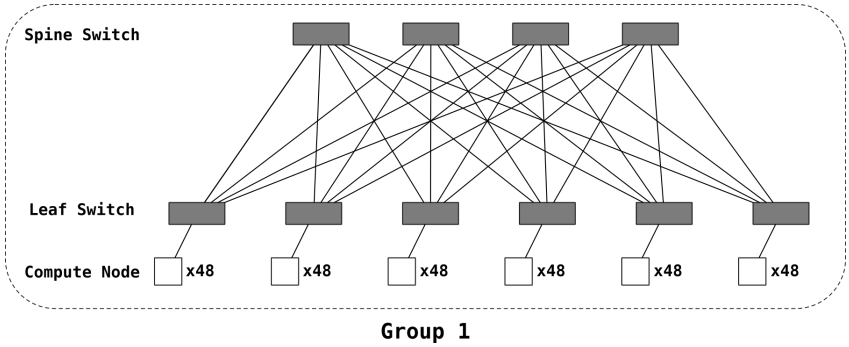


Figure 6.4: Structure of a group in Betzy's Dragonfly+ topology.

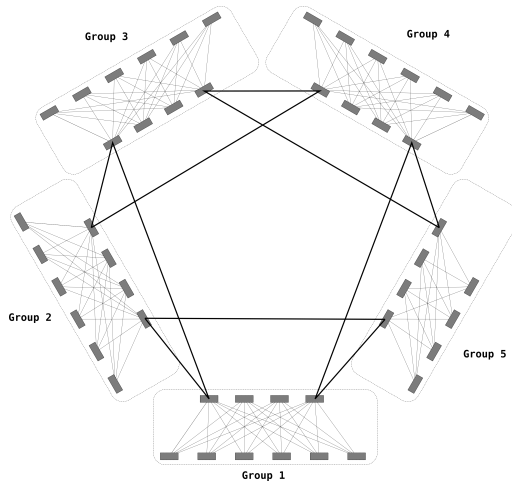


Figure 6.5: Dragonfly+ largest size topology on Betzy. Each group is directly connected to all other groups through at least one spine switch. Note that the fifth group has two fewer leaf switches than the others.

6.2.4 LogGP Communication Performance

We measured the LogGP parameters of our platforms as described in Section 4.4.1, in order to complete our communication model. The results are displayed in Figures 6.6, 6.7, and 6.8, in addition to Table 6.6. These measurements were taken using the open-source network measurement framework Netgauge [34] and repeated three times, such that different node placements were probed. We fit curves to our LogGP parameter measurements in order to derive corresponding functions of message size.

The measurements on Fram yield Equations 6.1 and 6.2:

$$G_{all}(s) = 0.00010(s-1) + 0.259 \tag{6.1}$$

$$o(s) = 0.00010s \tag{6.2}$$

On Betzy we have the corresponding Equations 6.3 and 6.4:

$$G_{all}(s) = 0.00009 * (s - 1) + 0.103 \tag{6.3}$$

$$o(s) = -2.56744 + 2.73683^{\sqrt[s]{106281.417}} \tag{6.4}$$

We note that these approximations are valid only on the measured message domain. Even though an exponential function gave a superior fit compared to a linear one for Betzy we do not exprect the overhead growth to follow this curve indefinitely.

Platform	$\overline{L}$ (μs)	σ
Idun	1.631257	0.031566
Fram	0.939723	0.015453
Betzy	1.218127	0.009428

Table 6.6: Measurement of the latency parameter L. $\overline{L}$ is the averaged latency from three measurements which we input into our model. σ is the standard deviation.

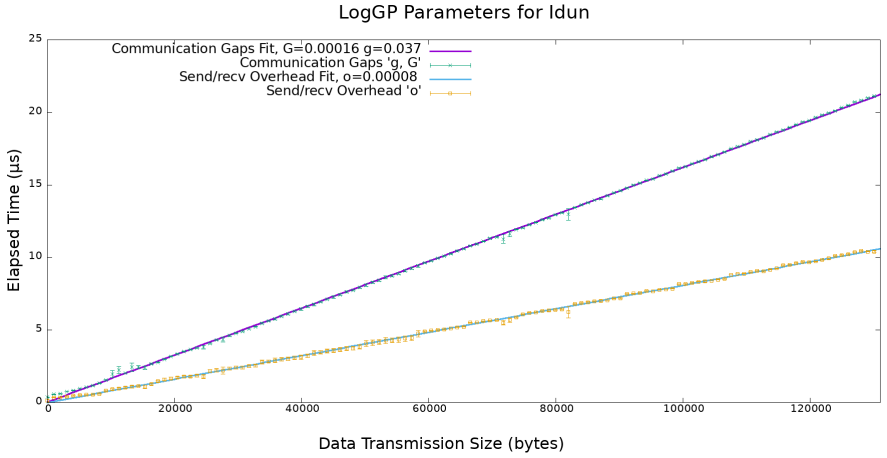


Figure 6.6: Measurement of o, g and G parameters on Idun. Both the communication gap and overhead may be modelled using linear functions.

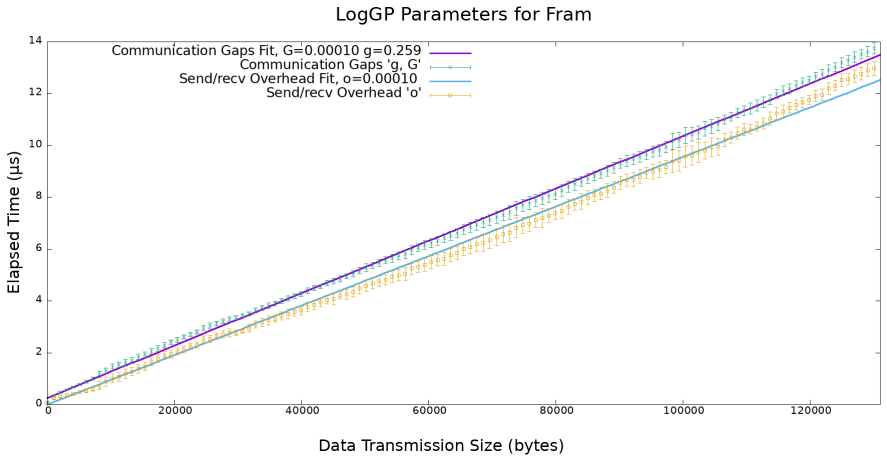
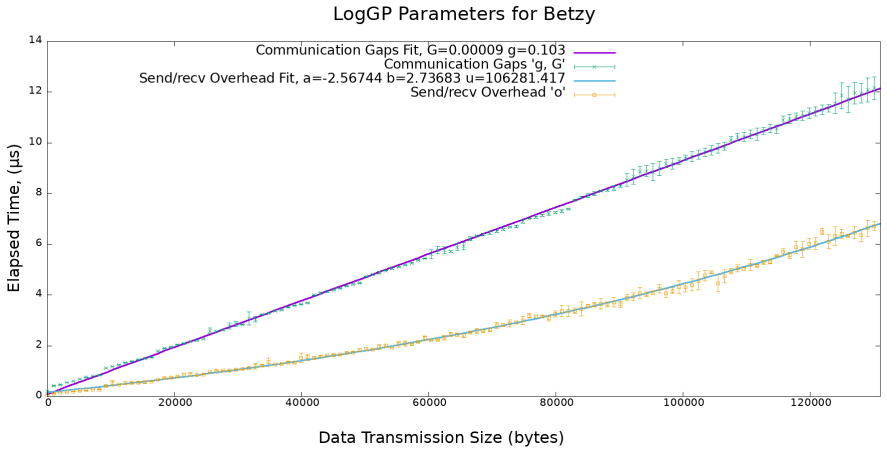


Figure 6.7: Measurement of o, g and G parameters on Fram. Both the communication gap and overhead may be modelled using linear functions.



Measurement of o, g and G parameters on Betzy. We observe that the communication gap follows a linear curve, while the overhead associated with sending and receiving is approximately exponential for the examined transmission sizes.

Figure 6.8:]

6.3 Parameter Space

The following is an overview of the parameters we consider particularly influential in regards to the outcome of our simulations and measurements.

6.3.1 Domain Dimensions and Geography

The total size of the domain determines the number of individual points which need to be calculated. The ratio between the number of fluid and land points influences the cost of these calculations. Furthermore, their distribution, specifically whether or not they are clustered together, determines the efficacy of our chunking technique.

6.3.2 Domain Decomposition

The decomposition of the domain determines the load balance between ranks,see Section 5.3, as well as the communication graph of the participating ranks. Load imbalances can negatively impact the total execution time. Communications do so too, but we expect that the effect will be much smaller in comparison.

6.3.3 Rank Distribution and Resource Availability

The physical placement of nodes and how they are interconnected can affect their communication time. We rely on the scheduler to give us nodes with reasonable proximity to one another. Additionally we specify an MPI topology, such that adjacent ranks are assigned to adjacent nodes whenever possible.

6.3.4 Initial condition

The shape and dimensions of the wave created during initialization determines the fluid levels of the following states. It does not affect the computational workload. However, it can cause destabilizations and affect where and when they occur. We know from Equation 2.16 that the velocity is key in the calculation of our Courant number. The step equations of our MacCormack method 2.4.1 show that the velocity in a direction is a function of both the existing velocity and fluid level at a given point. This means that excessive wave heights may put us over the CFL threshold. Besides the height of the wave at initialization we must also consider how it will interact with the domain. The geography may lead to wave energy being focused in regions such as corners, which can put the fluid level above the height of the initial wave.

6.3.5 Step Size

The parameters Δx, Δy and Δt indicate the distance in space between points and the distance in time between time steps. They affect the speed at which perturbations in the fluid travel across the domain. We select these such that we gain meaningful fluid movements, yet avoid instabilities. The values we settled on can be found in table 6.1.

6.3.6 Number of Iterations

The number of iterations I determines the total length of the simulation. As the simulation progresses, the chance of singularities increases, which means that the domain could destabilize. Once this happens our load balance estimates no longer apply. To avoid such situations we settled on running around 2500 to 5000 iterations for the experiments, depending on the platform and domain size of the experiment.

Chapter 7

Results and Discussion

In this chapter we display the outcomes of our experiments and discuss their implications. We examine the performance and behaviors of the decompositons themselves, and see what effect they have on the load balance of our application. Then we investigate the effects on our communication times, and compare the results to our model.

7.1 Rectangulations

Our three decomposition techniques yield different behaviors, both in the decomposition layout itself and in the executing proxy application. We describe these here and compare their effects on the proxy application's performance.

7.1.1 Decompositions and Scalability

The cost of producing the three decompositions varies strongly. For the domain sizes we investigated the Cartesian decomposition completes almost immediately, while the scanning decomposition completes within a minute. Our diagonal rectangulation algorithm is by far the most expensive one to run. Table 7.1 gives an idea of how the algorithm scales for a selection of very small problem sizes.

Degree \ Dimensions	250×450	500×900	1000×1800
3	1	3	13
4	3	12	59
6	51	246	-

Table 7.1: The time required to run the diagonal rectangulation algorithm to completion, in minutes.

Observations on Cartesian Decomposition

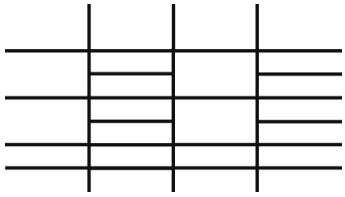


Figure 7.1: A 24 degree Cartesian partition.

The Cartesian decomposition produces a regular grid layout, an example of which is shown in Figure 7.1. Consequently a sub-domain may have at most four neighbors, each of which translates into one communication interface. In cases where we do not have a degree equal to a power of two the algorithm defaults to creating unevenly spaced regions.

Observations on the Scanning Rectangulations

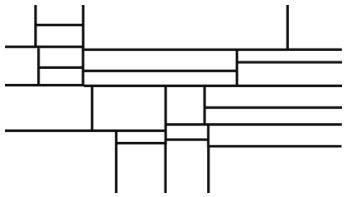


Figure 7.2: A 24 degree scanning partition.

The scanning decomposition produces more irregular grids, though we can still see artifacts that resemble the regular grid layout of the cartesian grid. With this decomposition a sub-domain may have more than four neighbors, see Figure 7.2. In practice this number rarely exceeds six neighbors.

Observations on the Diagonal Rectangulations

The diagonal rectangulation algorithm is able to produce a much more varied set of decompositions than the Cartesian one. Figure 7.3 gives an example of such a decomposition. Stopping the process early reveals that the exploration of the solution space tends to start with a scaled set of exclusively horizontal or vertical splits, and then moves on to ever more complex patterns. It does however scale poorly in terms of computational time, as shown in Table 7.1. Increases in domain size noticeably increase the computation time, while increase in decomposition degree may increase the computation time.

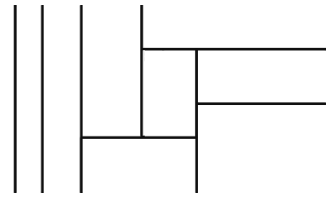


Figure 7.3: A 9 degree diagonal rectangulation.

We observe that the non-Cartesian rectangulations settle on decompositions where the center ranks have around 6 neighbors, up to 8 at most. This means that there is no dramatic increase in communications per-node, when compared to the 4 rank maximum of the Cartesian decomposition. Furthermore, this means that we don not exceed the maximum number of RDMA peers specified by the openib configuration shown in Table 6.4.

The steep cost curve of the diagonal rectangulation algorithm makes its application a question which has to be considered more carefully than the presented alternatives. It can be said that the obtainable gains from pre-computing a rectangulation are theoretically infinite, since the rectangulations may be re-used for any number of experiments. However, if we are only interested in a few executions the cost of decomposing this way may outweigh the gains.

There exist two practical work-arounds for the long rectangulation times: The algorithm can be run on a downscaled version of the target domain, which produces faster results at the cost of resolution. The speedup is gained by reducing the convergence time for each evaluated Baxter permuation. Alternatively, the algorithm may be terminated early before it has explored the full solution space. The diagonal rectangulation algorithm discovers many valid solution before reaching its conclusion, which may be used at the cost of them not necessarily being the most optimal solutions. In our implementation these early solutions usually involve minor variations of column-wise partitions. For large rectangulation degrees only the second approach is valid, due to the Baxter number's rapid growth.

7.1.2 Reduction of Load Imbalance

Figures 7.4a, 7.4b, and 7.4c show how much time each rank spends waiting for other ranks to finish their local computation before a communication step. In this scenario the rank with the greatest computational load determines the wait time of all the other ones. In our figures each horizontal bar in a column shows the wait time of the corresponding rank, as a percentage of the time spent in the main execution loop. Each column then shows the unutilized computation time of the whole MPI communicator. For instance, the column for a 12 rank decomposition in Figure 7.4a shows that four ranks spent approximately 70% of their execution time waiting, and that the idle time of the whole rank group amounts to about half of the total execution time.

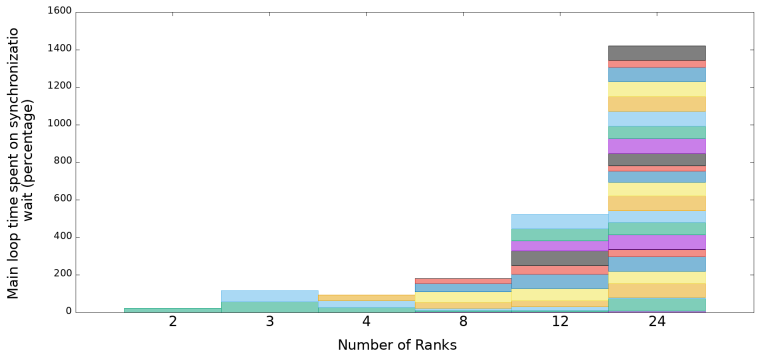
The regular Cartesian rectangulations predictably have the worst performance, since they do not exploit the map's variations in computational load. The scanning decomposition gives only a moderate reduction in wait times compared to the Cartesian algorithm in the worst case (P=3), but far exceeds it when the number of ranks is a power of two. It is only in this case that no ranks with approximately twice the computational cost of another can be found. The diagonal rectangulations have the best performance in the general case, but are just barely beaten by the scanning rectangulations when the number of ranks is a power of two. Here, the worst case wait time of a single rank is 23% of its execution time, and the wait times of the communicator as a whole don't exceed 16% of the total execution time.

The fact that the accumulated wait times of the diagonal rectangulation algorithm are consistently low shows that the analytical load estimate in Equation 5.1 is reasonably accurate, though not quite perfect.

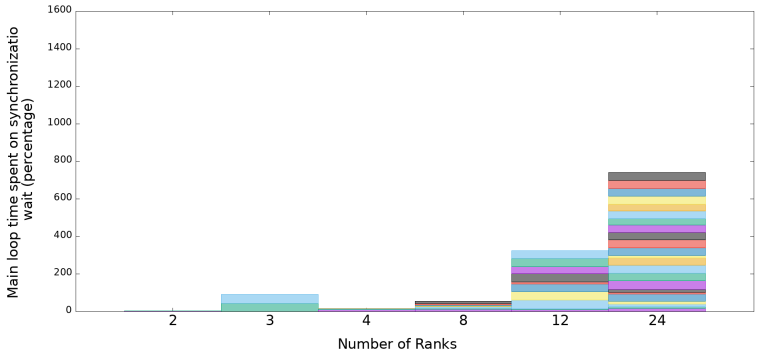
Figure 7.5 displays the attained speedup and efficiency during the load balancing tests. It reveals that the rectangulating decomposition is not thrown off by decomposition degrees which are not powers of two.

7.2 Communication

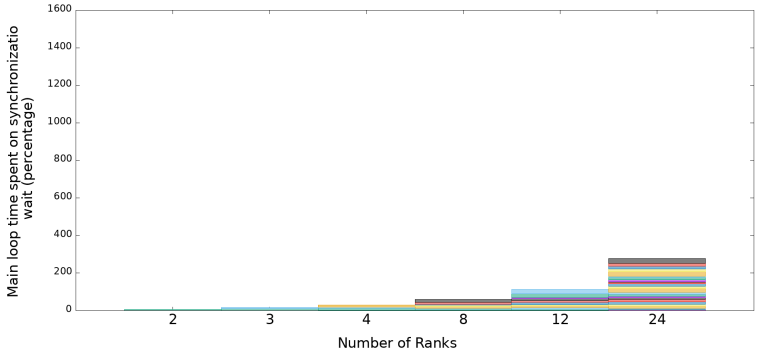
We investigate the effects of the decompositon on the cost of communications between ranks on separate nodes. First we verify constant communication cost, then we compare the communication times of a Cartesian layout with non-Cartesian ones.



(a) Cartesian decomposition wait time

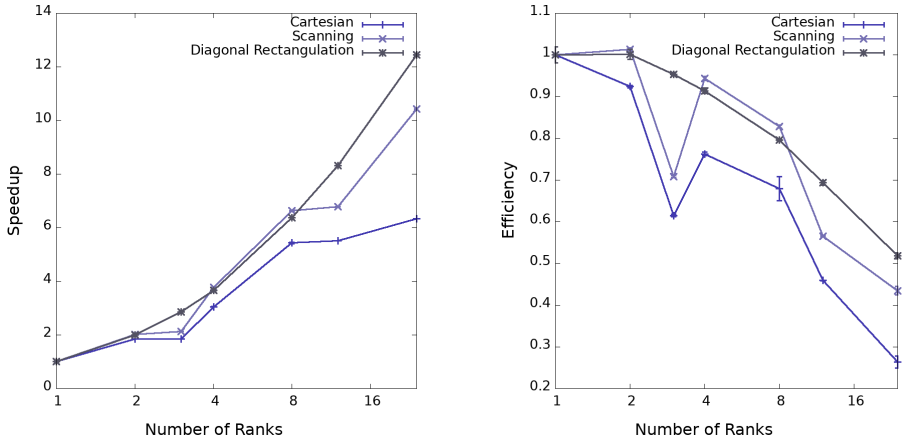


(b) Scanning decompositon wait time



(c) Diagonal rectangulation wait time

Figure 7.4: Time spent idle while waiting for other ranks to finish their iteration, when using different decompositions on Fram. The y-axis shows time in terms of percentage, compared to an executions total time spent in the main processing loop.



(a) **(b)**

Figure 7.5: Measured speedup and efficiency for our decompositions on Fram.

7.3 LogGP Benchmarks

The LogGP benchmarks of our platforms, displayed in Section 6.2, indicate that both Idun, Betzy and Fram behave reasonably similarly in terms of point-to-point communications when using matching toolchains. The benchmarks show that the transmission cost per byte is strongly linear with regards to the message size on the examined size range. The same can be said about the transmission overhead, except on Betzy, where the overhead is better fitted to an exponential curve on the examined domain.

Interestingly, we do not see any clear discontinuities indicating protocol switches, as described by Hoefler *et al* [33]. We would expect at least one in the lower region on x-axis, at the point specified by the btl_openib_eager_limit listed in Table 6.4. It is at this point that the benchmark should switch from using eager to rendezvous protocol. We attribute this to the fact that the eager and rendezvous protocols implemented in the PML may have similar communications costs at the point of transition.

7.3.1 Verification of Stable Communication Cost

We sampled the communication cost of a single iteration over the course of a complete run, in order to eliminate the possibility that it could increase or decrease over time. This could conceivably happen due to our application structure or system mechanisms such as caching.

Figure 7.6 shows the results of these measurements on our computation platforms. It is Fram that has the longest communication times, in spite of its superior LogGP benchmark and more sophisticated interconnect when compared to Idun.

We observe that although the communication time varies somewhat from iteration to iteration, there is no significant upward or downward trend.

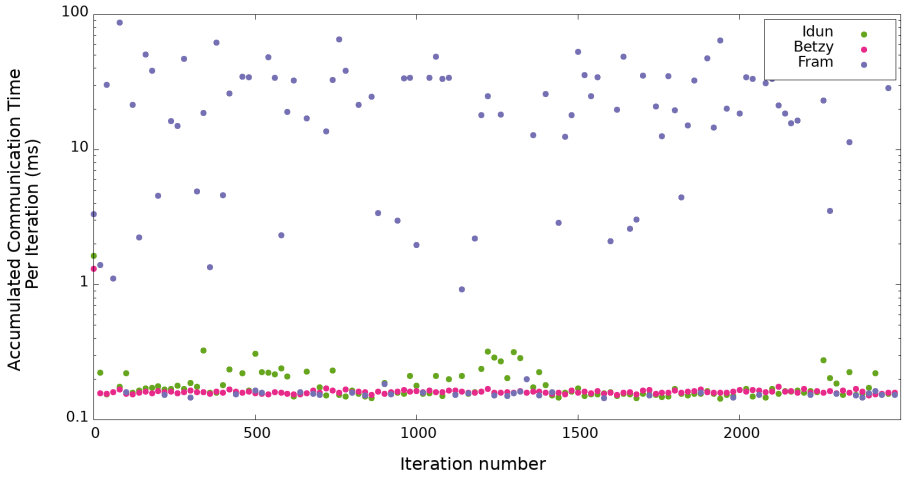


Figure 7.6: Time spent on communication per iteration for out platforms, Betzy, Fram and Idun. We observe no steady trend of increase or decrease.

7.3.2 Strong Scaling Comparison of Rectangulation

Figures 7.7 and 7.8 show how the average per-rank communication time develops as the domain is split into smaller and smaller sub-domains. This increases the total number of data exchanges, but reduces the number of exchanged bytes per message. Figure 7.9 shows how much time is spent on communications across the whole communicator in each test. The per-rank communication times for Betzy never exceed 1% of the total execution time. For Fram communications account for at most 3% of the execution time. We see a consistent difference between the Cartesian and irregular domains, which is proportional to the average message size.

Our communication model, described in Section 5.6.3, correctly predicts that the communication time does not vary over the course of a full execution. In figure 7.7 we can see that the prediction converges towards the measured results around 32 participating ranks.

However, it fails at capturing the communication times of lower-degree decompositions before this points. On Fram the model is proven wholly insufficient. The outcome is hinted at by Figure 7.6, but cannot be explained by means of the benchmark and LogGP parameters, which do not differ that significantly between Betzy and Fram. This indicates that there is at least one important factor which the LogGP model does not take into account, which we attribute to three primary causes:

1. The model does not consider network contention. Contention may stem from the experiment itself, as many ranks synchronized by the instrumentation barrier send messages at the exact same point in time. Other experiments running concurrently on the same platform can have the same, and likely an even greater, effect. While our experiments exclusively reserve all of the resources available on a single node for the duration of their execution, the network interconnect remains shared.

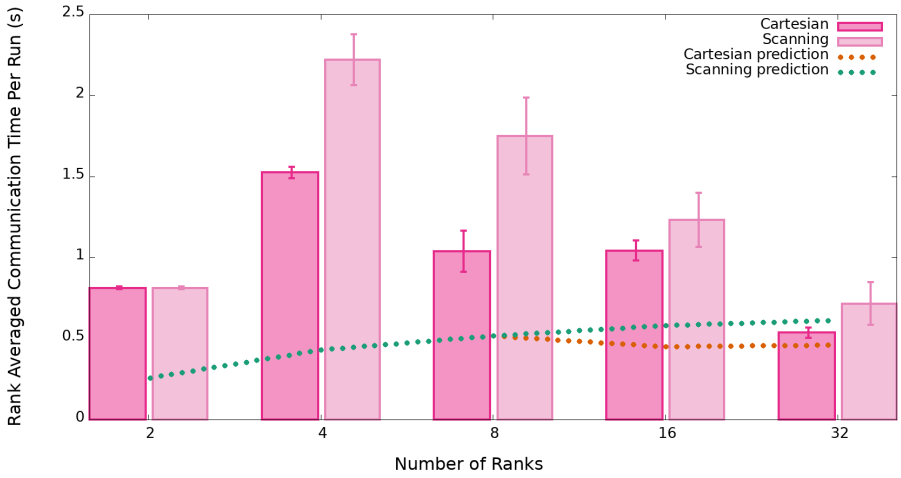


Figure 7.7: Average accumulated communication times observed on Betzy with strong scaling.

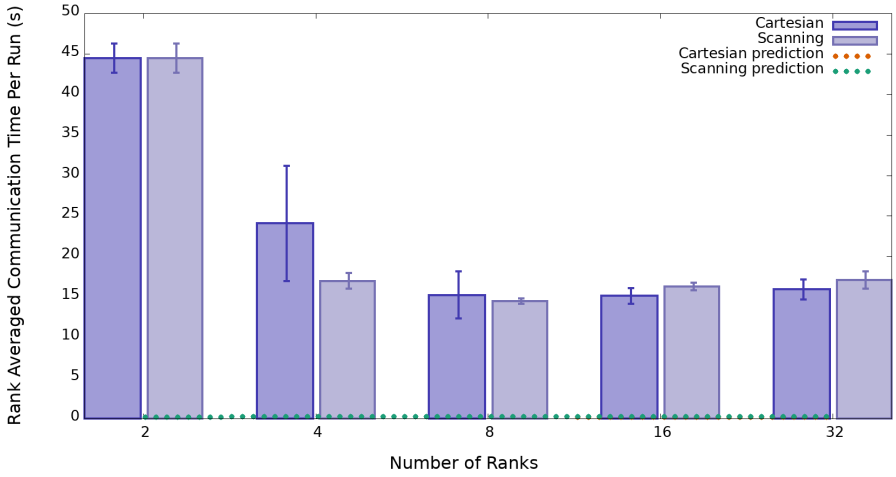


Figure 7.8: Average accumulated communication times observed on Fram with strong scaling

2. The model does not consider synchronization overheads. Other variants of the LogP model, such as the LogGPS [49] and LogGOPSim [50] models, consider the synchronization costs of MPI data exchanges. These synchronizations cost should, however, only occur when rendezvous-style messages are exchanged.

3. There may be shortcomings in terms of our use of the model. The LogGP model and its Netgauge benchmark was designed around ordinary Send() and Receive() calls, not their non-blocking counterparts.

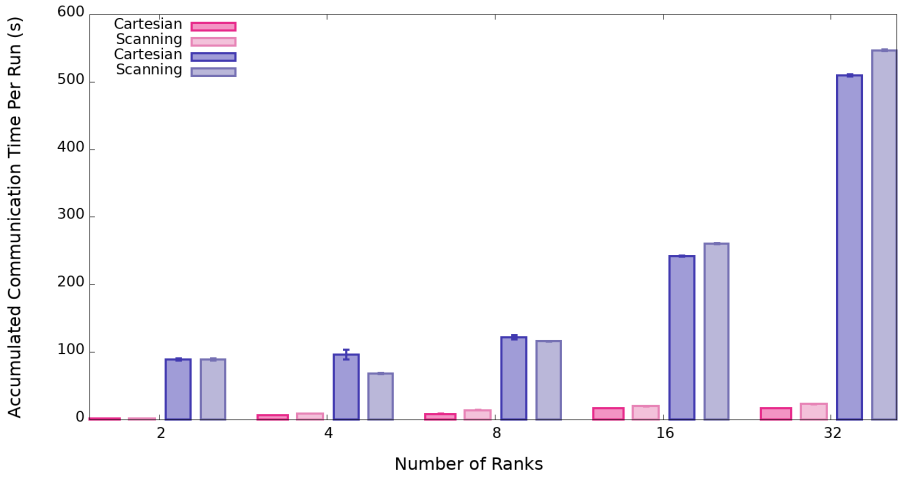


Figure 7.9: Total accumulated communication times observed on both Betzy and Fram with strong scaling

Chapter 8

Conclusion

In this thesis we demonstrate that diagonal rectangulation is a valuable domain decomposition technique for two-dimensional solution spaces of partial differential equations.

We show this though the implementation of a proxy application, which serves as a simplified model of large scale HPC applications. Our application solves the shallow water equations numerically on a domain that contains mixed fluid and land areas, which introduces differences in computational cost. Using the application we are able to test and compare cartesian decompositions, decompositions obtained through orthogonal recursive bisection, and decompositions by diagonal rectangulation. We show that the diagonal rectangulations have the greatest benefit, but their computation times scale poorly with the problem size. The ORB decompositions prove to be a valid compromise with lower efficiency.

Furthermore, we show that the additional communication cost introduced by the decomposition is insignificant at the examined scales when compared to the gains of improved load balance. We predict this through a LogGP model, the parameters of which we find for our platforms, and confirm it with our instrumentation.

8.1 Future Work

We recognize that our diagonal rectangulation algorithm produces superior results in terms of load balance, but is quite expensive to run at scale. In contrast to our proxy-application, it is implemented in a strictly sequential manner and scales poorly for larger domains and greater numbers of sub-domains. We propose two improvements which may address this:

- The generation of a rectangulation and its tuning by a minimization algorithm can be done independently of the next. This stage can therefore be considered as *embarassingly parallel* and can be easily parallelized.
- It seems likely that the SLSQP minimization algorithm we selected may be tuned to converge faster, for example by adjusting the tolerance and deltas. The algorithm could also be exchanged for a more efficient one altogether.

Furthermore, the employed analytical measure of per-point computational cost works well for our proxy-application, but may be difficult to create for larger applications. An overview of estimated local cost for popular processing and optimization techniques would be useful in combination with the decomposition algorithms.

Acknowledgements

I would like to thank Jan Christian Meyer, my supervisor, for his patient and experienced guidance and for introducing me to the world of scientific computing at large.

The computations were performed on resources provided by the NTNU IDUN/EPIC computing cluster [46], as well as resources provided by UNINETT Sigma2 - the National Infrastructure for High Performance Computing and Data Storage in Norway.

Appendix

8.1.1 Supplementary Tables

n	$B(n)$
1	1
2	2
3	6
4	22
5	92
6	422
7	2074
8	10754
9	58202
10	326240
20	29949238543316
30	7101857696077190042814
40	2554987813422078288794169298972
50	1163558691573487855005674103586862832160

Table 8.1: A Selection of Baxter Numbers $B(n)$.

8.2 Symbols

B	Baxter number
c	Number of computing units
C	Courant number
d	Delay between message transmissions
E	Efficiency
η	Fluid column height
g	Time gap between transmitted messages when they are sent/received one after the other
G	Time gap between the transmission of two bytes in the same message
i	Number of transmitted messages
I	Number of iterations
k	Iteration or time step number
L	Transmission latency, upper bound
m	Mass
n	Number of points in a sequence, located along the diagonal of a bounding rectangle
o	Per-rank communication overhead
p	Point along the diagonal of a rectangle
P	Number of ranks participating in a given communication
P_{neigh}	Number of domains adjacent to a given domain
ρ	Fluid density
R	Bounding rectangle of a diagonal rectangulation
s	Transmitted data size in bytes
S	Speedup
T	Time
u	Velocity in the x dimension
v	Velocity in the y dimension

Table 8.2: Symbols and definitions

8.3 Other software

Notable software in the program and production of this work:

Netgauge	2.4.6	A network performance measurement toolkit
Libnetpbm	10.73.33	A Library for reading pbm and pgm files.
gnuplot	5.4.1-1	An interactive plotting program
Gimp	2.10.24	Open source image manipulation software

Glossary

floorplan A decomposition of a 2D domain into rectangular tiles and the spatial relation between the tiles. 9

ideal fluid An inviscid fluid, characterized exclusively by its isotropic pressure, density and the shape of its surroundings. 3, 4

mosaic floorplan A floorplan where every part of the domain is covered by tiles. 9

node An independent computing unit, composed of a CPU, memory and an interconnect, which is part of an HPC platform. 14

rank A single-process instance of an MPI program. 21

rectangulation degree The number of partitions produced by rectangulating a rectangle. 9

shallow water flow Fluid flow where the width and length of the flow far exceeds its height. 3

Acronyms

BML BTL Management Layer. 22

BSP Bulk Synchronous Parallel. 15, 17, 23, 30

BTL Byte Transfer Layer. 22

CFD Computational Fluid Dynamics. 3, 7

CFL Courant-Friedrich-Lewy. 7

CPU Central Processing Unit. 14, 18

FDM Finite Difference Method. 5, 6

FLOP Floating Point Operation. 25

HPC High Performance Computing. 3, 14, 23

MA Memory Access. 25

MCA Modular Component Architecture. 21

MPI Message Programming Interface. 21, 29, 31

P2P Point-to-Point. 21

PDEs Partial Differential Equations. 4, 5, 6

PML Point-to-Point Messaging Layer. 22, 47

PRTT Parameterized Round Trip Time. 19

RDMA Remote Direct Memory Access. 22, 44

RTT Round Trip Time. 19, 22

SLSQP Sequential Least Squares Programming. 12, 13

SQP Sequential Quadratic Programming. 12, 13

TOPO Process Topology. 22

Bibliography

[1] P. Parna, K. Meyer, and R. Falconer. “GPU driven finite difference WENO scheme for real time solution of the shallow water equations”. In: *Computers and Fluids* 161 (2018).

[2] Guiseppina Colicchio et al. “Towards a fully 3D domain-decomposition strategy for water-on-deck phenomena”. In: *Journal of Hydrodynamics, Ser. B* 22.5, Supplement 1 (2010).

[3] Alfonso Sánchez-Beato. *How to share your video-conference window among attendees or, the many ways of splitting a rectangle in many*. `https://www.alfonsobeato.net/math/the-many-ways-of-splitting-a-rectangle-in-many/`. [Online; accessed 23-May-2021].

[4] Abdessamad Qaddouri et al. “Optimized Schwarz methods with an overset grid for the shallow-water equations: preliminary results”. In: *Applied Numerical Mathematics* 58.4 (2008), pp. 459–471.

[5] Alfio Maria Quarteroni and Alberto Valli. *Domain decomposition methods for partial differential equations*. Oxford University Press, 1999.

[6] Steven R. Hannah et al. “Detailed Simulations of Atmospheric Flow and Dispension in Downtown Manhattan: An Application of Five Computational Fluid Dynamics Models”. In: *Bulletin of the American Meteorological Society* 87.12 (2006).

[7] Chaojun Ouyang et al. “A MacCormack-TVD finite difference method to simulate the mass flow in mountainous terrain with variable computational domain”. In: *Computers & Geosciences* 52 (2013).

[8] Yuqi Wu and Xiao-Chuan Cai. “A fully implicit domain decomposition based ALE framework for three-dimensional fluid-structure interaction with application in blod flow computation”. In: *Journal of Computational Physics* 258 (2014).

[9] Reinaldo Garcia and Rene A. Kahawita. “Numerical solution of the St. Venant equations with the MacCormack finite-difference scheme”. In: *International Journal for Numerical Methods in Fluids* 6.5 (1986).

[10] Robert J. Fennema and M. Hanif Chaudhry. “Explicit Methods For 2-D Transient Free-Surface Flows”. In: *Journal of Hydraulic Engineering* 116.8 (1990).

[11] Patrick J. Roache. *Computational Fluid Dynamics*. Albuquerque, New Mexico: Hermosa Publishers, 1972.

[12] Hseng Tseng Ming and Chia R Chu. “Two-dimensional shallow water flows simulation using TVD-MacCormack scheme”. In: *Journal of Hydraulic Research* 38.2 (2000), pp. 123–131.

[13] Eric Ngondiep, Alqahtani T Rubayyi, and Jean C Ntonga. “A MacCormack Method for Complete Shallow Water Equations with Source Terms”. In: *arXiv preprint arXiv:1903.11104* (2019).

[14] Eyal Ackerman, Gill Barequet, and Ron Y Pinter. “On the number of rectangulations of a planar point set”. In: *Journal of Combinatorial Theory, Series A* 113.6 (2006), pp. 1072–1091.

[15] Xianlong Hong et al. “Corner block list: An effective and efficient topological representation of non-slicing floorplan”. In: *IEEE/ACM International Conference on Computer Aided Design. ICCAD-2000. IEEE/ACM Digest of Technical Papers (Cat. No. 00CH37140)*. IEEE. 2000, pp. 8–12.

[16] Geoffrey C Fox. “A graphical approach to load balancing and sparse matrix vector multiplication on the hypercube”. In: *Numerical Algorithms for Modern Parallel Computer Architectures*. Springer, 1988, pp. 37–61.

[17] Nathan Reading. “Generic rectangulations”. In: *European Journal of Combinatorics* 33.4 (2012), pp. 610–623.

[18] Glen Baxter. “On fixed points of the composite of commuting functions”. In: *Proceedings of the American Mathematical Society* 15.6 (1964), pp. 851–855.

[19] Zion Cien Shen and Chris CN Chu. “Bounds on the number of slicing, mosaic, and general floorplans”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22.10 (2003), pp. 1354–1361.

[20] Sean S. Plott. “Functions of the Binomial Coefficient”. PhD thesis. Harvey Mudd College, 2008.

[21] Dieter Kraft. *A software package for sequential quadratic programming*. Tech. rep. Institut fuer Dynamik der Flugsysteme, DFVLR Obersfaffeuhofen, Germany, 1988.

[22] Leslie G. Valiant. “A bridging Model for Parallel Computation”. In: *Commun. ACM* 33.8 (1990).

[23] Allan Hartstein and Thomas R Puzak. “The optimum pipeline depth for a microprocessor”. In: *ACM Sigarch Computer Architecture News* 30.2 (2002), pp. 7–13.

[24] Alan Jay Smith. “Line (block) size choice for CPU cache memories”. In: *IEEE transactions on computers* 100.9 (1987), pp. 1063–1075.

[25] Alan Jay Smith. “Sequential program prefetching in memory hierarchies”. In: *Computer* 11.12 (1978), pp. 7–21.

[26] G.M Shipman et al. “Infiniband scalability in Open MPI”. In: *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*. 2006.

[27] Jingwei Sun et al. “Automated Performance Modeling of HPC Applications Using Machine Learning”. In: *IEEE Transactions on Computers* 69.5 (2020).

[28] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: an insightful visual performance model for multicore architectures”. In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[29] Kevin Barker et al. “Using Performance Modeling to Design Large-Scale Systems”. In: *IEEE Computer* 42 (Nov. 2009), pp. 42–49. DOI: 10.1109/MC.2009.372.

[30] I.M. Navon and Y. Cai. “Domain decomposition and parallel processing of a finite element model of the shallow water equations”. In: *Computer Methods in Applied Mechanics and Engineering* 106.1 (1993).

[31] Albert Alexandrov et al. “LogGP: Incorporating Long Messages into the LogGP Model-One Step Closer towards a Realistic Model for Parallel Computation”. In: *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*. Association for Computing Machinery, 1995.

[32] David Culler et al. “LogP: Towards a realistic model of parallel computation”. In: *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*. 1993, pp. 1–12.

[33] Torsten Hoefler, Andre Lichei, and Wolfgang Rehm. “Low-Overhead LogGP Parameter Assessment for Modern Interconnection Networks”. In: *2007 IEEE International Parallel and Distributed Processing Symposium*. 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370593.

[34] Torsten Hoefler et al. “Netgauge: A Network Performance Measurement Framework”. In: *High Performance Computing and Communications*. Ed. by Ronald Perrott et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 659–671. ISBN: 978-3-540-75444-2.

[35] OpenMP ARB. *OpenMP web description*. https://www.openmp.org/about/openmp-faq/. [Online; accessed 02-December-2020].

[36] L. Dagum and R. Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *IEEE Computational Science and Engineering* 5.1 (1998).

[37] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. https://wwww.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf. [Online; accessed 07-June-2021].

[38] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. “Open MPI: A Flexible High Performance MPI”. In: *Parallel Processing and Applied Mathematics*. Springer Berlin Heidelberg, 2006.

[39] Sayantan Sur et al. “RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits”. In: *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2006, pp. 32–39.

[40] Kartverket. *Kartdata fra Kartverket*. https://www.kartverket.no. [Online; accessed 21-May-2021].

[41] Marsha J Berger and Phillips Colella. “Local adaptive mesh refinement for shock hydrodynamics”. In: *Journal of computational Physics* 82.1 (1989).

[42] Marsha J Berger and Joseph Oliger. “Adaptive mesh refinement for hyperbolic partial differential equations”. In: *Journal of computational Physics* 53.3 (1984).

[43] Ola Toft and Jan Christian Meyer. “Parallel Scalability of Adaptive Mesh Refinement in a Finite Difference Solution to the Shallow Water Equations”. In: *Norwegian conference for ICT-research and education* 1.1 (2020).

[44] Randall J LeVeque. *Finite-Volume Methods for Hyperbolic Problems*. Cambridge University Press, 2004. Chap. 7.

[45] Torsten Hoefler et al. “The scalable process topology interface of MPI 2.2”. In: *Concurrency and Computation: Practice and Experience* 23.4 (2011), pp. 293–310.

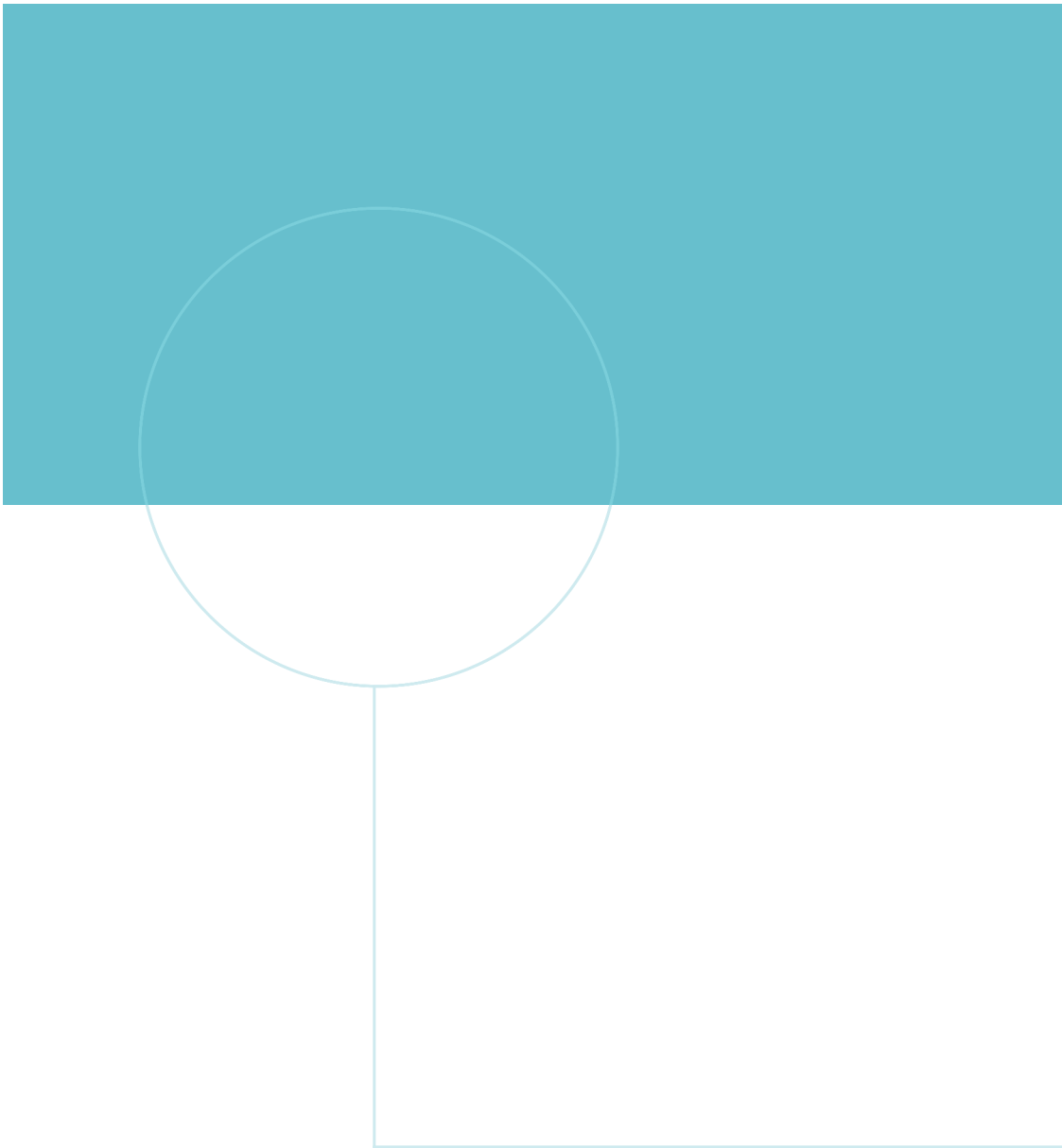
[46] Magnus Själander et al. *EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure*. 2019. arXiv: 1912.05848 [cs.DC].

[47] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. “A Scalable, Commodity Data Center Network Architecture”. In: *SIGCOMM Comput. Commun. Rev.* 38.4 (2008).

[48] Alexander Shpiner et al. “Dragonfly+: Low Cost Topology for Scaling Datacenters”. In: *2017 IEEE 3rd International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*. 2017, pp. 1–8.

[49] Fumihiko Ino, Noriyuki Fujimoto, and Kenichi Hagihara. “LogGPS: a parallel computational model for synchronization analysis”. In: *Proceedings of the eighth ACM SIGPLAN symposium on Principlesand practices of parallel programming*. 2001, pp. 133–142.

[50] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. “LogGOPSim: simulating large-scale applications in the LogGOPS model”. In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. 2010, pp. 597–604.



NTNU

Kunnskap for en bedre verden