Master's thesis

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

Jenny Veronika Ip Manne

# Performance Modeling of a Finite Volume Method for the Shallow Water Equations

Master's thesis in Computer Science
Supervisor: Assoc. Prof. Jan Christian Meyer
September 2022

NTNU
Norwegian University of
Science and Technology

Jenny Veronika Ip Manne

# Performance Modeling of a Finite Volume Method for the Shallow Water Equations

Master's thesis in Computer Science
Supervisor: Assoc. Prof. Jan Christian Meyer
September 2022

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

*Acknowledgements*
I would like to dedicate this thesis to my supervisor Jan Christian Meyer for his support and guidance throughout the project, to Prof. Anne C. Elster for letting me be a part of the HPC lab, and to my friends and family for their support and feedback to my project.

# Problem Description

This thesis aims to develop a proxy application which solves the Shallow Water Equations using a Finite Volume Method with triangular control volumes, in order to study its parallel performance and scalability. It further aims to develop and experimentally validate a quantitative performance model of these properties.

# Abstract

The thesis presents a proxy application for computing the shallow water equations through the finite volume method. A triangulation discretizes the water surface, where the coastline of Mehamn harbor is used for all tests. The application uses MPI for making parallel ranks and OpenMP for creating parallel threads for each rank, and is run on the Idun cluster provided by NTNU.

A performance model for the application predicts the parameters of the triangular structure, which is based on the number of vertices, and uses the structural estimations to predict the runtime and scaling characteristics. It separates computation and communication time, with focus on the scaling characteristics of the computation time. By applying Mehamn harbor for the triangulation, a more direct prediction is made, as the geometrical exceptions are eliminated. The roofline and Hockney models are also made through a set of benchmarking tests in order to discover the hardware properties of maximum FLOP performance and communication time of message passing. A comparison of the performance model and the benchmarking models with the application execution is used to verify the performance characteristics of the application.

The results show an accurate prediction of the parameters of the triangular structure, which confirms the first step of the performance model. Through the roofline model, the benchmarking tests confirms that the application is memory bound, where the maximum bandwidth is further used for the performance model. The Hockney model confirmed that the communication time stays constant for an increasing number of nodes and grows slowly for an increasing problem size, while the communication time between nodes were overestimated.

The results also show consistent scalability of computation time between the performance model and the application execution, for both strong and weak scaling, and for scaling of both ranks and threads, given that the problem size does not overflow the cache memory. This is supported by an even load balance between the processing units. It is discovered that the cache memory of the hardware affected the scalability, where a problem size too large to fit into cache gave a poorer performance. In addition, a single rank per node gave a poorer performance, since a node has multiple CPUs with separate cache memories, causing the threads to pollute the cache. An optimal performance is reached by choosing a problem size that uses the entire cache space and by assigning at least one rank for each CPU. By applying the constant communication time, this gives an overall stable and predictable weak scaling, suitable for scaling of nodes with multiprocessors.

# Table of Contents

# List of Tables

# List of Figures

# List of Source Code

# Abbreviations

| | | |
|---|---|---|
| HPC | = | High Performance Computing |
| MPI | = | Message Passing Interface |
| PDE | = | Partial Differential Equation |
| FVM | = | Finite Volume Method |
| FDM | = | Finite Difference Method |
| SWE | = | Shallow Water Equations |
| $V$ | = | Vertices |
| $T$ | = | Triangles |
| $t$ | = | Time |

Chapter 1

# Introduction

The shallow water equations are a set of partial differential equations that describe the flow of fluids, often used to compute and simulate waves. The finite volume method can be applied to numerically solve the SWE, by dividing the surface into volumes and using a flux flow. A triangulation of surface is flexible, compared to a discretization based on rectangles. By making a proxy application to solve SWE for FVM over a triangulation, its properties can be explored without developing the fully scaled program.

A performance model gives a prediction of application behaviours and characteristics. It can be used to identify weaknesses and to identify improvements to increase performance and scaling. By confirming a performance model of a proxy application, the results can be further used to predict the performance of the fully scaled program.

The goal of the thesis is to make a proxy application for solving the shallow water equations with the finite volume method over a triangulation, and to compare the results with the performance models, in order to verify the performance and scaling characteristics. All of the application test use Mehamn harbor, located in Gamvik municipality in Norway. The Norwegian Coastal Administration have arranged two breakwaters along the harbor, in order calm the wave conditions for the docking cargo and fishing vessels [1]. It is therefore interesting to make wave simulations based on this coastline.

## 1.1 Scope

The thesis presents a proxy application for computing the shallow water equations, through the finite volume method over a triangular discretization of the water surface. The application is parallelized by distributing the triangles to different *ranks* using MPI, where each rank spawns parallel *threads*, through OpenMP, to simultaneously execute computation. In addition, synthetic benchmarking tests are used to discover hardware characteristics. A set of multicore machines from the Idun cluster, provided by NTNU, are used to run both the application and the benchmarking tests [2].

The performance models predict the behaviour of the proxy application, by first predicting the parameters of the triangular structure based on the number of vertices, then by using the parameters to predict application runtime and scaling characteristics. The benchmarking tests are used to model the maximum hardware performance, through the roofline and Hockney model. By using a predefined geometry for all application tests, such as Mehamn harbor, the parameters of the triangular structure can be estimated directly, instead of being limited to an interval. This narrows down further runtime and scaling predictions. The analysis separates the computation time for executing SWE from the communication time between each rank, and makes a model of their scaling characteristics, with focus on the computation time. The performance models are finally compared with the triangular structure generated by the application, and the application runtime and scaling.

## 1.2 Chapters

Chapter 2 presents the relevant background material for parallel programming, performance modeling and solving shallow water equations using the finite volume method. Chapter 3 provides additional related work. Chapter 4 describes the preprocessing and the implementation of the proxy application, and Chapter 5 presents the performance models, which predict the parameters of the triangular structure, the computation time, the communication time, and the scaling characteristics. Chapter 6 gives details of the software and hardware, a setup of the benchmarking tests, and the configurations for application testing. Chapter 7 compares the performance models and benchmarking results with the application tests. Chapter 8 concludes the thesis and outlines interesting directions for future work.

Chapter 2

# Background

This chapter presents the relevant background material, by explaining parallel programming, the use of proxy applications and performance modeling, and a concrete discretization of the shallow water equations, by using the finite volume method over a triangulation of the water surface.

## 2.1 Parallel Programming

Serial computation models have been the standard for programming, as early program code was run on serial and single core computers. From the 1980s, computer designs started to incorporate instruction-level parallelism, allowing techniques to execute instructions simultaneously, such as pipelining and vectorization [3]. These features were largely transparent to the programmer, making program execution fit for a serial model. From 2005, multicore computers were made available for purchase, allowing programs to be run in parallel [4]. McCool explains in [5] how the serial models fail to keep up with the multi-core computers and suggests a parallel model should be used as the new standard. He also emphasizes the scalability of the new models, discussing how performance is keeping up as the number of parallel components increases.

A programming language translates computations between two models, from programming models into process models. A programming model is an abstract model used by the programmer in order to understand how the computation is executed. The programmer should be provided the necessary information so they can manipulate the program to run efficiently. A process model describes how the physical machine performs computation, which is hardware specific and can be used for instruction set architecture.

### 2.1.1 Programming Model

The programming model should accurately reflect the process model, while being *expressive*, *simple* and *safe*. With accurate models, the cost understood by the programmer reflects the cost of the hardware. An *expressive* model allows the programmer to concisely express understandable and efficient solutions for a target domain. The computational complexity should not be significantly different for solutions generated from different programming models. A *simple* programming model hides the unnecessary details from the programmer while exposing important cost operations, in order to make development efficient. A *safe* model protects the programmer from making common and costly mistakes.

A parallel programming model should give a similar work complexity as the serial model, to allow the programmer to write efficient code, and should include synchronization protection so that issues such as race conditions and deadlocks are avoided.

### 2.1.2 Process Model

A process model gives an overview of the underlying hardware of a parallel system, and has different variations that affects how it is viewed to the programmer.

#### Flynn's Taxonomy

Table 2.1 lists the classifications of computer architectures for parallel processes, referred to Flynn's taxonomy [6]. The most simple architecture is the single instruction stream, single data stream (SISD), which exploits no parallelism. The single instruction stream, multiple data stream (SIMD) allows an instruction to execute multiple data. SIMD is classified into the three categories of array processor, pipelined processor and associative processor. An array processor has separate memories and register files for each parallel unit. For a pipelined processor, each parallel unit reads and writes data from the same central source. An associative processor lets each parallel unit make its own decision of execution based on its local data. The multiple instruction stream, single data (MISD) is the most uncommon architecture and can be used for fault tolerance, by checking that multiple instructions operate on the same data stream give the same results. The multiple instruction stream, multiple data stream (MIMD) executes different instructions for different data, which is used by multicore processors and distributed systems, by utilizing a shared or distributed memory space. The single program, multiple data stream (SPMD) is a model based on an implementation of a parallel computer architecture, which uses the same instruction stream, but allows independent execution choices. This can be applied to both SIMD and MIMD architectures.

| | |
|---|---|
| Multiple Instruction, Single Data (MISD) | Multiple Instruction, Multiple Data (MIMD) |
| Single Instruction, Single Data (SISD) | Single Instruction, Multiple Data (SIMD) |

**Table 2.1:** The four computer architecture classifications of Flynn's Taxonomy.

#### Memory for Parallel Models

A parallel system can utilize a distributed or shared memory space. For a distributed memory, each component has its own local memory and shares data with the other components through message passing. This is often used by a distributed parallel system. The messages can be synchronous, where the component waits for a message to be received, asynchronous, where a component does not wait after sending a message, or a hybrid of the two. Ramachandran divides a parallel program into a communication and computation part, where communication consists of message passing and computation consists of the isolated instructions run on the components [7]. A synchronizer can be setup by the programmer to control the conditions before the communications.

A multicore processor can use a shared memory space, which is accessible by all processing units. From a programmer's perspective, all units can access the memory simultaneously without further configurations. McCool states that a shared memory has a low level message passing setup, which is hidden from the programmer [5]. A processor can have a cache memory in addition to the shared memory. When a change is made in the local cache memory, cache coherency protocols are needed, to keep the other processors updated, by sending low level messages between each unit.

### Task Parallel Processing Models

Task parallel processing models decompose the program into tasks to be run in parallel on the different processors. Each processor has a local memory, which gives a task access to an independent data stream, making task parallel model represent a MIMD processing model. A task communicates with other tasks through message passing or shared memory, which adds delay to the overall program, due to message processing or cache coherency protocols. A shared memory can be hard to scale, as the complexity grows with product of memory size and processing units [8]. For a scaling of processors, the number of tasks will run out, and some processors will be left idle and be wasted. An uneven size of tasks will also cause load imbalance, as the processors finish the tasks at different times. Data dependency and strict policies for accessing shared memory can also force the tasks to execute in serial, weakening the utility of having parallel hardware.

### Data Parallel Processing Models

Data parallel processing models decompose the program into segments of data, which are computed simultaneously on different processors, using the same instruction. This represents both SIMD processing model and SPMD implementation, depending on whether each processor can make an independent decision based on the data segment. Data parallel processing models are made flexible and simpler for scaling, by segmenting the data for an arbitrary number of processors, and using the same function to compute each segment, while allowing other parallelism such as pipelining to be hidden from the programmer. The data can also be aligned to improve locality, in order to reduce memory access time.

SIMD and SPMD have different scaling properties. SIMD uses the same program for each data segment, which makes implementation simple, but does not handle exceptions well. If a processor needs extra instructions due to a special case for its data segment, these instructions must be applied to the other processors as well, even if the result is unused and tossed away. This gives an even load balancing, though, each processor has the same worst execution time. SPMD allows conditional operations for each program, so special cases can be added only to the processors that need it. This may cause problems for load balancing, as the processors can have different program sizes. It can leave room for an idle processor to run another program, where it could be useful to have more data segments than number of cores.

**Vector Processing Models**

Vector processing models are based on SIMD models, but allows data to be streamed in and out of variable length storage locations. This gives a low arithmetic intensity, which is the ratio between the amount of computations and the consumed memory bandwidth. As processor speeds has been increasing, memory bandwidth is left as the limiting factor. Stream processing models are based on vector processing models, with computationally intensive kernel. The kernel contains a number of instructions for input and output streams, allowing a large number of operations to be executed for every memory access, which increases the arithmetic intensity.

Data streaming allows collective operations such as *reduction* and *scan*. A *reduction* is used to perform an arithmetic operation over a vector of multiple data, resulting with one output. *Scan* is similar to *reduction*, by allowing operations over a vector, but outputs a new corresponding vector as a result. *Gather* and *scatter* can be used as alternative ways of streaming data. *Gathers* allow kernels to read from random memory, but is less efficient than using a streaming output. *Scatters* allow kernels to write to random memory, but need deterministic rules in order to avoid memory collisions.

## 2.1.3 Parallel Hardware

Programming models and process models should generalize applications to be configured for different hardware architectures. Multicore processors have multiple cores running on the same chip, and can be used for MIMD models, with each core running a separate set of instructions. Multithreaded processors (SMT) use multiple threads of control, without replicating cores. This requires fewer resources, but has a lower scalability than having multicore. Multithreading and multicore hardware can also be combined, but can leave threads competing for resources. SPMD implementations treat the various threads as a single program and include coordination of thread scheduling, so a hardware scheduler can be applied to reduce scheduling overhead and allow finer grained threads.

Stream processors are similar to multicore processors, with on-chip managed memory and data transfer. Operations and functions can be applied to a stream of data, which reduces overhead of data transfer. Data access and communication often causes the main bottleneck of the system, and can cause varied latency based on the physical layout of the network. Using the same network for memory access and processor communication, scales poorly with the number of cores, as it increases the memory bottleneck [5]. A mesh, hypercube or hierarchical interconnect is commonly used. Locking mechanisms can be used to run atomic memory operations, but forces data to be transferred in large blocks, which weakens scaling.

### 2.1.4 Implementations

There are commonly used programming systems that use parallel models, which are suited for distributed systems, multicore processors, and for Graphics processing units.

#### MPI

Message passing interface (MPI) is a standardized message passing design for parallel architectures [9]. It is a SPMD implementation, by having independent processes of instructions, and includes collective operations such as *reduction*, *gather*, and *scatter*, used for communications between processes.

#### OpenMP

OpenMP is based on a shared memory and uses among other things a loop level parallelism, by spawning individual threads for each iteration [10]. When applied to a multicore hardware, it is a MIMD processing model, by creating multiple threads of control.

OpenMP can be run on multiple CPUs, where each CPU has a multicore hardware. The thread affinity is the protocol to decide how to distribute the threads on such systems. The two main configurations are *compact* and *scatter*, where *compact* allocates all the cores of one CPU at a time, and *scatter* distributes the threads evenly across all CPUs.

#### GPU

GPUs were originally designed for fixed programs, fitting for SIMD processing models. CUDA can use a SPMD implementation when running on a GPU, by allowing program specifications on a thread block level. On the individual thread block, all threads execute the same instructions, resembling the SIMD processing model.

## 2.2 Performance Modeling

A performance analysis of a program can describe how the program acts and performs. Traditionally, a simulation would be used to run the program on an existing system and mimic how it would act on a fully scaled system [11]. For long-running programs, this approach is costly, as the simulation would need to run on a system of a comparable size or for an unacceptable amount of time. A performance model offer analytical formulas of how a program behaves, and can be used to predict its performance before it is fully developed. Depending on what parameters and characteristics are chosen, it can show the effects of using different specifications such as hardware specifications, code designs and problem sizes. This can be combined with benchmarking tests, which are used to analyse the hardware capabilities, in order to predict the performance of the program running on the specific hardware.

### 2.2.1 Proxy Application

Scientific investigations are in increasing need of supercomputing performance, and are transitioning to Exascale systems [12]. Preparing simulations and running them on such systems can be costly, due to the size and complexity of the programs. A proxy application can be used to test certain aspects of a larger program and to analyze its performance before the program is fully scaled.

A proxy application is a miniapp, that resembles some functional aspects of a larger program. It is a tool used to experiment and analyse certain parts of the program, without needing to spend resources and time developing the full program. Since the miniapp is significantly smaller than the original program, it is flexible for changes and faster to run for testing. Due to the simplification of the proxy application, its results may be different or inaccurate from what is expected from the original program. For a performance analysis, this does not necessarily have to be a problem, as the behaviour of the proxy application is of more interest than its produced data results.

A performance model made for a proxy application can be tested and confirmed, and give further indication of how the program would behave and perform for an Exascale system.

### 2.2.2 Roofline Model

Williams describes how the *Roofline* model can be used to predict the highest attainable floating-point performance, by relating processor performance to off-chip memory traffic [13]. Operational intensity is measured in operations per byte, which can be used to predict the DRAM bandwidth needed to access main memory. For a particular computer, peak floating-point performance can be found either through hardware specifications or microbenchmarks. Fig. 2.1 shows the Roofline model for a *2.2GHz AMD Opteron X2* model, with attainable floating-point performance over operational intensity. For lower values of operational intensity, the floating-point performance is memory bound, but the upper bound increases until it reaches a certain value of operational intensity. At that point the attainable floating-point performance is compute-bound and does not get any higher.

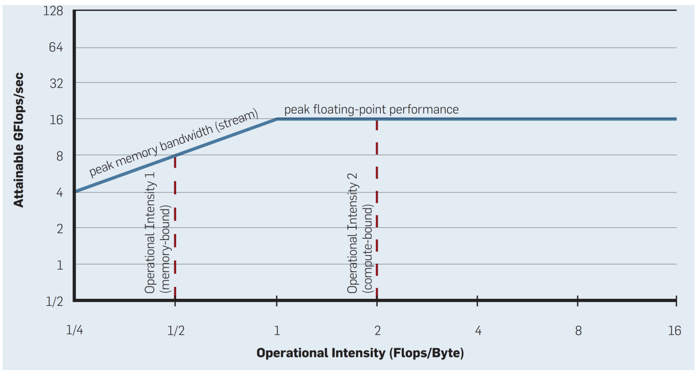


**Figure 2.1:** Roofline model for *AMD Opteron* from [13]

### 2.2.3 Scaling

Scaling is used to indicate the ability of a program to utilize computational power as the resources increases, such as the hardware, parallelization and problem size. Speedup can be defined as the ratio between the runtime $t_1$ of a serial program and the runtime $t_N$ of the program on a set of $N$ parallel processors, given as

$$\textit{speedup}_N = \frac{t_1}{t_N} \tag{2.1}$$

An ideal speedup would have a linear scaling, with a value of $N$, but overhead occurs due to the program not being entirely parallelizable, which accumulates as the number of processing units increases.

Efficiency can be defined as the ratio between the actual speedup and the ideal speedup when using $N$ processors, given as

$$\textit{efficiency}_N = \frac{\textit{speedup}_N}{N} \tag{2.2}$$

An ideal efficiency would give a value of 1, given an ideal linear speedup of the value $N$.

#### Amdahl's Law

Amdahl's law describes scaling for a number of parallel processing units, while keeping the same problem size [14]. This is known as strong scaling. The program is divided into a serial part $s$, which is the proportion of the program that must run in serial, and the other proportion $p$, which can be parallelized. This gives a speedup of

$$\textit{speedup}_N = \frac{1}{s + p/N} \tag{2.3}$$

and an efficiency of

$$\textit{efficiency}_N = \frac{1}{Ns + p} \tag{2.4}$$

With an increased number of processors, it takes less time to run the parallel code. The speedup is limited by the serial part of the code, and as $N$ increases, it can only reach a maximum of

$$\lim_{N \to \infty} \textit{speedup}_N = \frac{1}{s} \tag{2.5}$$

while the efficiency decreases towards 0.

#### Gustafsons's Law

Gustafson reevaluates Amdahl's law by scaling the problem size as the number of processing units increases [15]. This is known as weak scaling. Given the serial proportion $s$ and parallelizable proportion $p$, speedup can be formulated as

$$\textit{speedup}_N = s + pN \tag{2.6}$$

giving an efficiency of

$$\textit{efficiency}_N = \frac{\textit{speedup}_N}{N} + p \tag{2.7}$$

The speedup is no longer limited by an upper bound, and becomes more linear as the number of processors increases. The efficiency tends towards

$$\lim_{N \to \infty} \textit{efficiency}_N = p \tag{2.8}$$

as $N$ increases towards infinity.

### 2.2.4 Bulk Synchronous Model

The Bulk Synchronous Model (BSP) is a designed to work as a bridge between the hardware and parallel programs [16]. The model has three main attributes. The first attribute is a number of components of processing or memory functions. The second is a router that can deliver messages between the components. The last attribute is a setup of facilities for synchronizing the components at regular intervals.

The model represents an execution of a barrier synchronized parallel program, where the program consists of a sum of *Supersteps*. For each *Superstep*, the components process their local data. A barrier makes all components wait for the others to finish processing, before the router can send messages between the components. After receiving the messages, the next *Superstep* is executed. The total runtime of the program, is the summation of the runtime for each *Superstep* $t_s$

$$\sum_{s=1}^{S} t_s = \sum_{s=0}^{S} (w_s + gh_s + l) \tag{2.9}$$

where $S$ is the total number of *Supersteps* and $l$ is the synchronization latency. The processing time of the data is $w_s$, which is determined by the slowest component, as they are processed in parallel. The communication time $gh_s$ is determined by the inverse bandwidth $g$ and the longest message size $h_s$.

### 2.2.5 Fundamental Equation of Modeling

Barker *et al.* introduce a methodology of how to create a performance model of a parallel program, which has the function to guide the programmer to make the most efficient choices [11]. The fundamental equation of modeling

$$t_s = t_{comp} + t_{comm} - t_{overlap} \tag{2.10}$$

describes the runtime of a parallel system, where $t_{comp}$ is the total computation time for all components, $t_{comp}$ is the communication time between each component and $t_{overlap}$ is the overlap of the former two. For a repeated iterative program, similar to the parallel program described for the BSP model, the same $t_s$ can be used for each iteration $s$.

### 2.2.6 Hockney Model

Hockney describes the communication between each component of a distributed system, based on benchmarking tests of Intel iPSC/860, Paragon XP/S and Meiko CS-2 [17]. The communication time is modeled as

$$t_{comm}(n) = \alpha + \beta^{-1}n \tag{2.11}$$

for a message size of $n$ bytes, where $\alpha$ is the latency and $\beta$ is the bandwidth. Fig. 2.2 show the transfer rate over an increasing message size for each machine. The transfer rate is linear, following the inverse of the communication formula Eq. (2.11), up until a message size above 1000 bytes, when the overhead of increasing the message size becomes significant.

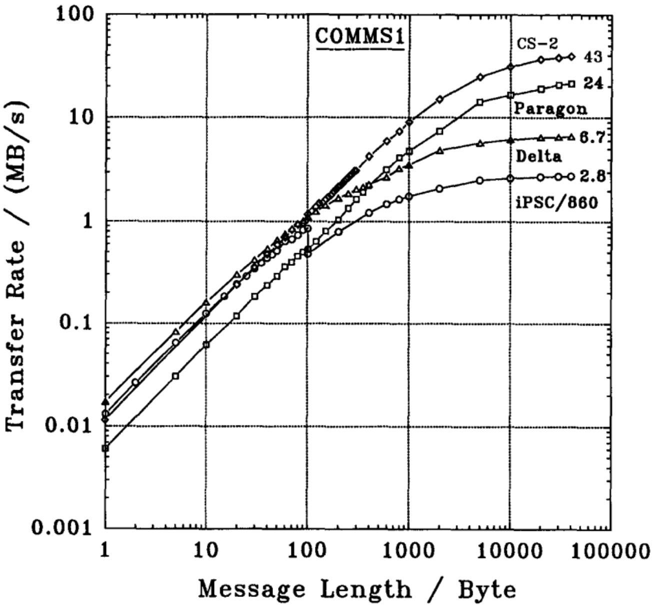


**Figure 2.2:** Hockney model for transfer rate with increasing message length from [17].

### 2.2.7 LogP Model

*LogP* is a parallel machine model, presented by Culler, that can be used to set a basis for developing fast, parallel algorithms and offer guidelines to machine designers [18]. The model is based on four parameters, which set the terms of algorithm performance.

The first parameter is communication delay ($L$), which sets an upper bound for communicating a message, containing one or a few words, from a source to a target module. The second is communication overhead ($o$), defined as the time a processor is engaged in the transmission or reception of a message, making this processor unavailable for any other task for the whole duration. The next parameter is communication bandwidth or gap ($g$), the minimum time interval between consecutive message transmissions or consecutive message reception for a processor, which also corresponds to the available communication bandwidth for a processor. The final parameter is the number of processors or memory modules ($P$), where unit time is assumed for local operations and called a cycle. A finite capacity of $L/g$ is assumed, so no more messages can be sent at a time. A similar model, *LogGP*, extends this model by adding the parameter $G$, which is the gap $g$ for longer messages per byte [19].

A parallel algorithm can be evaluated by analysing each communicating component and summing the delay, based on the four parameters, where $L$, $o$ and $g$ are measured as multiples of processor cycle, by using the maximum time and space by any processor.

## 2.3 Solving PDEs

A partial differential equation (PDE) is an equation containing partial derivatives of variables. PDEs can be classified into hyperbolic, parabolic and elliptic equations [20].

Scientific fields such as physics and engineering use PDEs to understand and analyse environments such as sounds, heat, water motion, quantum mechanics, *etc*. PDEs are not given to be generally solvable through analytical methods. They can be solved numerically, by using discretization of the domain and making an approximation for each piece. Some common methods are the finite difference method (FDM), the finite volume method (FVM) and the finite element method (FEM).

### 2.3.1 Hyperbolic PDEs

Hyperbolic partial differential equations have initial value problems, so that a point in a surface can be uniquely solved based on the neighborhood of the given point. A disturbance at a point in the surface does not immediately cause a change for the whole surface, but creates a finite propagation speed relative to the given point. This can be used to represent wave movements over time. A general wave equation for one dimension is formulated as

$$\frac{\partial^2 u}{\partial t^2} - c^2 \frac{\partial^2 u}{\partial x^2} = 0 \tag{2.12}$$

which is a basis for other wave equations such as the shallow water equations. The derivative of $u$ over time $t$ is dependent on its speed over the surface $x$ given a constant $c$, and for a given initial state for $t = 0$, a solution can be found for all $t$.

#### Conservation Laws

A conservation law states that a measurable property stays constant within the domain. It is defined as the the variation of conserved flow quantity within a given volume due to the net effect of some internal sources and the amount of quantity crossing the boundary surface. Its formula is

$$\frac{\partial}{\partial t} \int_{\Omega} U d\Omega + \oint_{S} \boldsymbol{F} \cdot d\boldsymbol{S} = \int_{\Omega} Q d\Omega \tag{2.13}$$

where $\Omega$ is the volume, $S$ the surface, $U$ the quantity, $F$ the flux vector, and $Q$ is the source. The quantity $U$ can only change due to the flux passing through the surface.

**Boundary Conditions**

Boundary conditions determine the quantity $u$ along the border of the domain. A constant value, such as setting $u = 0$ along the border, is called *the homogeneous Dirichlet*, while setting $u$ to a function is called *the inhomogeneous Dirichlet*. *Neumann boundary condition* is denoted as $u_n = \partial u/\partial n$, specifying the directional derivative of $u$ along a normal vector. For a wave equation, the Neumann boundary condition reflects the wave along the border.

### 2.3.2 Finite Difference Method

A finite difference method is a numerical method that is based on a grid partition of the domain, where every cross point is dependent on its adjacent vertices. Fig. 2.3 shows the vertex $(i, j)$ and its neighbors for a grid partition in a two dimensional $xy$ plane. There are four neighbor vertices, two on the $x$-axis and two on the $y$-axis. Every horizontal neighboring pair has the same distance $\Delta x$ and every vertical neighboring pair has the same distance $\Delta y$.

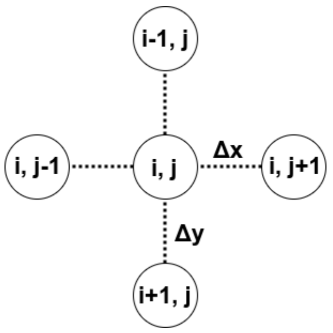


**Figure 2.3:** A FDM grid showing a vertex of coordinate $(i, j)$ with its four neighbors $(i-1, j)$, $(i+1, j)$, $(i, j-1)$ and $(i, j+1)$. The distance between any pair of two vertices on the $x$-axis is $\Delta x$ and $\Delta y$ on the $y$-axis. Every vertex $(i, j)$ is dependent on all its four neighbors.

A hyperbolic time dependent PDE can be solved for every time step $t = n + 1$ using the values from last time step $t = n$. From [21], Taylor's formula can be used to solve for every quantity $u_i$ from its adjacent nodes for every time step $t = n+1$, giving the formula

$$u_{i,j}^{n+1} = u_{i,j}^{n} + \Delta t \left( \frac{u_{i-1,j} + u_{i+1,j}}{2\Delta y} + \frac{u_{i,j-1} + u_{i,j+1}}{2\Delta x} \right) \tag{2.14}$$

Since FDM bases the quantity on point values, the space between the vertices are discarded, which weakens the accuracy. With fixed distances between the vertices of $\Delta x$ and $\Delta y$, FDM is not flexible to adjust the accuracy for certain areas by allowing the vertices to be denser.

### 2.3.3 Finite Volume Method

The finite volume method (FVM) is a discretization of the PDE, by using volumes that can be made of any geometrical shape. Each element is represented as an average value. The basis of the FVM is taking use of the conservation law from Eq. (2.13), meaning the flux entering a volume element is equal to the flux leaving the adjacent volume element.

The conservation law can be used to find the quantity $U$ numerically, by approximating the flux flow of each volume element. For a hyperbolic time dependent PDE with a set initial state for $t = 0$, each value $u_i^{n+1}$ for element $i$ and time step $t = n + 1$ can be estimated directly, as it is only dependent on the state for the previous time step $t = n$. The flux $F_{i,j}$, from element $j$ to $i$, can be found by summing the quantity flow to and from each neighboring element. This gives the general equation for each element

$$u_i^{n+1} = u_i^n + \Delta t \cdot \sum_{j \in Adj(i)} F_{i,j} \tag{2.15}$$

where $\Delta t$ is the time difference between each time step $n$.

#### Triangulation

Eq. (2.15) can be applied to a triangulation, a subdivision of a two-dimensional plane into triangles. Each triangle represents a volume $\Omega$ that holds a quantity $U$. From the neighbouring triangles, the flux flow $F$ can get through the border surface $S$. Fig. 2.4 shows the flux flow of a triangle coming from its three neighbors. $\Delta d$ is the distance between the centers of two neighboring triangles, and affects the amount of flux passing through the border.

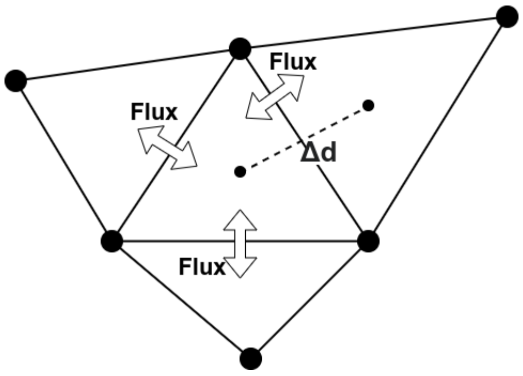


**Figure 2.4:** The flux flow of a triangle from its neighbors. $\Delta d$ is the distance between the centers of a specific pair of neighboring triangles.

The advantages this type of planar division, is the given flexibility. The vertices can be distributed arbitrarily so different values of accuracy can be given across the plane, since areas with a high density of vertices will have smaller triangles. A triangular mesh is also more flexible in filling irregular shaped areas, compared to using a grid.

A Delaunay triangulation is a triangulation of $P$ vertices, so that no vertex is inside the circumcircle of any triangle [22]. The minimum of the angles of the triangles is maximized, so sliver triangles are avoided. Fig. 2.5 shows such a triangulation on five vertices.

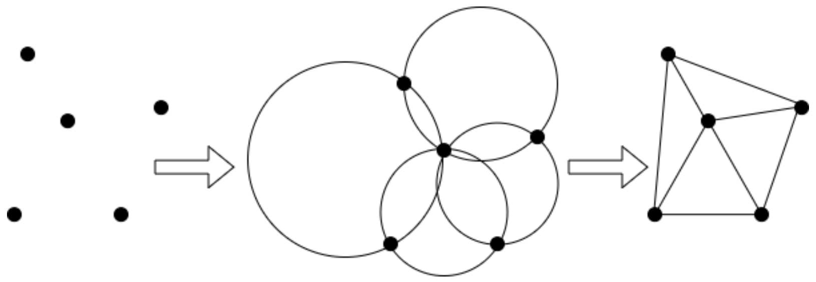


**Figure 2.5:** Delaunay triangulation of five vertices, with a maximization of the minimal angles of the triangles.

### The Lax-Friedrichs Method

Lax-Friedrichs method is an averaging method, that uses the average of the quantity $u$ for the neighboring elements when applying the flux flow. Eq. (2.15) can be rewritten as

$$u_i^{n+1} = \frac{\sum_{j \in Adj(i)} u_j^n}{|Adj(i)|} + \Delta t \cdot \sum_{j \in Adj(i)} F_{i,j} \tag{2.16}$$

This works as a numerical diffusion that damps the instabilities that can arise from the original equation. Over time, this causes the overall flux flow to shrink, so the quantity $u$ reaches equilibrium.

For a variant of this method, $u$ can be included when averaging out its neighbors. A weighted variable $w \in [0, 1]$ can be used to make

$$u_i^{n+1} = u_i^n \cdot w + \frac{\sum_{j \in Adj(i)} u_j^n}{|Adj(i)|} \cdot (1 - w) + \Delta t \cdot \sum_{j \in Adj(i)} F_{i,j} \tag{2.17}$$

## 2.3.4 Shallow Water Equations

The shallow water equations (SWE) are a set of hyperbolic PDEs describing the flow of a liquid on a surface. The overall mass and the linear momentum are the basis of the conservation laws [20]. For a two dimensional surface in the $xy$ plane, this gives the conservative formula for SWE as

$$\frac{\partial(\rho\eta)}{\partial t} + \frac{\partial(\rho\eta u)}{\partial x} + \frac{\partial(\rho\eta v)}{\partial y} = 0 \tag{2.18}$$

$$\frac{\partial(\rho\eta u)}{\partial t} + \frac{\partial}{\partial x}\left(\rho\eta u^2 + \frac{1}{2}\rho g \eta^2\right) + \frac{\partial(\rho\eta u v)}{\partial y} = 0 \tag{2.19}$$

$$\frac{\partial(\rho\eta v)}{\partial t} + \frac{\partial(\rho\eta uv)}{\partial x} + \frac{\partial}{\partial y}\left(\rho\eta v^2 + \frac{1}{2}\rho g\eta^2\right) = 0 \tag{2.20}$$

where $\rho$ is the fluid density, $\eta$ the total fluid height, $g$ the gravitation acceleration, and vector $(v, u)$ the fluid's velocity. Eq. (2.18) is derived from the conservation of mass $\rho\eta$, and the equations Eq. (2.19) and Eq. (2.20) are derived from the conservation of linear momentum $\rho\eta u$ and $\rho\eta v$.

### Discretization

The SWE formulas Eq. (2.18), Eq. (2.19) and Eq. (2.20) can be discretized by replacing the partial derivatives with difference estimations, giving formulas

$$\frac{\Delta(\rho\eta)}{\Delta t} + \frac{\Delta(\rho\eta u)}{\Delta x} + \frac{\Delta(\rho\eta v)}{\Delta y} = 0 \tag{2.21}$$

$$\frac{\Delta(\rho\eta u)}{\Delta t} + \frac{\Delta}{\Delta x}\left(\rho\eta u^2 + \frac{1}{2}\rho g\eta^2\right) + \frac{\Delta(\rho\eta uv)}{\Delta y} = 0 \tag{2.22}$$

$$\frac{\Delta(\rho\eta v)}{\Delta t} + \frac{\Delta(\rho\eta uv)}{\Delta x} + \frac{\Delta}{\Delta y}\left(\rho\eta v^2 + \frac{1}{2}\rho g\eta^2\right) = 0 \tag{2.23}$$

to describe the mass and momentum differences over discrete time steps and positions on the $xy$ plane.

The conservation law Eq. (2.13) can be applied to find the mass and momentum for each time step, using the flux flow of mass and momentum, giving the formulas

$$\rho\eta_i^{n+1} = \rho\eta_i^n - \Delta t \cdot \sum_{j\in Adj(i)} F_{i,j}(\rho\eta) \tag{2.24}$$

$$\rho\eta u_i^{n+1} = \rho\eta u_i^n - \Delta t \cdot \sum_{j\in Adj(i)} F_{i,j}(\rho\eta u) \tag{2.25}$$

$$\rho\eta v_i^{n+1} = \rho\eta v_i^n - \Delta t \cdot \sum_{j\in Adj(i)} F_{i,j}(\rho\eta v) \tag{2.26}$$

for each position or element $i$, depending on what numerical method is applied. Methods such as the FVM decide the estimations of the flux flow $F(\rho\eta)$, $F(\rho\eta u)$ and $F(\rho\eta v)$.

### The Lax-Friedrichs Method

Lax-Friedrichs method of Eq. (2.17) can be applied to Eq. (2.24), Eq. (2.25) and Eq. (2.26) by using the weighted average value from position $i$ and its neighbors $j$, giving

$$\rho\eta_i^{n+1} = \rho\eta_i^n \cdot w + \frac{\sum_{j\in Adj(i)} \rho\eta_j^n}{|Adj(i)|} \cdot (1-w) - \Delta t \cdot \sum_{j\in Adj(i)} F_{i,j}(\rho\eta) \tag{2.27}$$

$$\rho\eta u_i^{n+1} = \rho\eta u_i^n \cdot w + \frac{\sum_{j\in Adj(i)} \rho\eta u_j^n}{|Adj(i)|} \cdot (1-w) - \Delta t \cdot \sum_{j\in Adj(i)} F_{i,j}(\rho\eta u) \tag{2.28}$$

$$\rho\eta v_i^{n+1} = \rho\eta v_i^n \cdot w + \frac{\sum_{j\in Adj(i)} \rho\eta v_j^n}{|Adj(i)|} \cdot (1-w) - \Delta t \cdot \sum_{j\in Adj(i)} F_{i,j}(\rho\eta v) \tag{2.29}$$

with the weighted variable $w \in [0,1]$.

**Flux Estimation for FVM with Triangulation**

For a grid, every neighbor of a vertex is aligned directly on the right and left on the $x$-axis, and directly over and under on the $y$-axis. This forces the flux flow to be aligned in the same manner, so the velocity $u$ only moves to its left and right neighbor over $\Delta x$, and the velocity $v$ only moves to its neighbors up and down over $\Delta y$. For a triangulation, the volume elements can be aligned arbitrarily, shown in Fig. 2.4, so the flux can be directed diagonally along both the horizontal and vertical axis. Each element gets both velocity $u$ and $v$ from all neighbors over the distance $\Delta d = \sqrt{\Delta x^2 + \Delta y^2}$. For a triangulation, each element is a triangle that has exactly three neighbors. The mass flux flow from Eq. (2.24) for element $i$ can be formulated as

$$\sum_{j\in Adj(i)} F_{i,j}(\rho\eta) = \left(\frac{\Delta(\rho\eta u)}{\Delta d} + \frac{\Delta(\rho\eta v)}{\Delta d}\right) \tag{2.30}$$

and the momentum flux flow from Eq. (2.25) and Eq. (2.26) as

$$\sum_{j\in Adj(i)} F_{i,j}(\rho\eta u) = \frac{\Delta(\rho\eta uv)}{\Delta d} + \frac{\Delta}{\Delta d}\left(\rho\eta u^2 + \frac{1}{2}\rho g\eta^2\right) \tag{2.31}$$

and

$$\sum_{j\in Adj(i)} F_{i,j}(\rho\eta u) = \frac{\Delta}{\Delta d}\left(\rho\eta v^2 + \frac{1}{2}\rho g\eta^2\right) + \frac{\Delta(\rho\eta uv)}{\Delta d} \tag{2.32}$$

The mass flux flow for element $i$ can be estimated by taking the average difference of $\rho\eta u_i$ and $\rho\eta u_j$ for each neighbor $j$ over $\Delta d$ giving

$$\frac{\Delta(\rho\eta u)}{\Delta d} = \frac{1}{3} \cdot \sum_{j\in Adj(i)} \frac{\rho\eta u_j - \rho\eta u_i}{\Delta d} \tag{2.33}$$

and the average difference of $\rho\eta v_i$ and $\rho\eta v_j$ over $\Delta d$ giving

$$\frac{\Delta(\rho\eta v)}{\Delta d} = \frac{1}{3} \cdot \sum_{j\in Adj(i)} \frac{\rho\eta v_j - \rho\eta v_i}{\Delta d} \tag{2.34}$$

Similarly, the momentum flux can be estimated by using the average differences of $\frac{\Delta(\rho\eta uv_i)}{\Delta d}$, $\frac{\Delta}{\Delta d}\left(\rho\eta u_i^2 + \frac{1}{2}\rho g\eta_i^2\right)$ and $\frac{\Delta}{\Delta d}\left(\rho\eta v_i^2 + \frac{1}{2}\rho g\eta_i^2\right)$ over $\Delta d$, for each neighbor $j$ of element $i$, giving

$$\frac{\Delta(\rho\eta uv)}{\Delta d} = \frac{1}{3} \cdot \sum_{j\in Adj(i)} \frac{\rho\eta uv_j - \rho\eta uv_i}{\Delta d} \tag{2.35}$$

$$\frac{\Delta}{\Delta d}\left(\rho\eta u^2+\frac{1}{2}\rho g\eta^2\right)=\frac{1}{3}\cdot\sum_{j\in Adj(i)}\left(\left(\rho\eta u_j^2+\frac{1}{2}\rho g\eta_j^2\right)-\left(\rho\eta u_i^2+\frac{1}{2}\rho g\eta_i^2\right)\right) \tag{2.36}$$

and

$$\frac{\Delta}{\Delta d}\left(\rho\eta v^2+\frac{1}{2}\rho g\eta^2\right)=\frac{1}{3}\cdot\sum_{j\in Adj(i)}\left(\left(\rho\eta v_j^2+\frac{1}{2}\rho g\eta_j^2\right)-\left(\rho\eta v_j^2+\frac{1}{2}\rho g\eta_i^2\right)\right) \tag{2.37}$$

### Boundary Conditions for Triangulation

The boundary conditions decide the quantity at the border of the domain. For a triangulation, *ghost triangles* can be added along the borders, shown in Fig. 2.6. Every *ghost triangle* is mirrored over the border, so the flux vector into the domain is perpendicular. The mass and momentum can be set on these triangles, depending on which boundary condition is chosen, to decide the flux flow into the main domain. If for example the mass and momentum are always set to zero, the mass and momentum will leak from the domain.

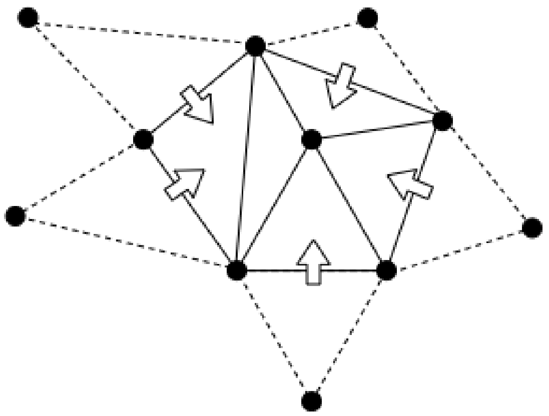


**Figure 2.6:** A triangulation with *ghost triangles* mirrored along the border. The mass and momentum decide the flux flowing into the domain.

For the Neumann boundary condition, the fluid is reflected back along the border. In order to stop the mass from leaking, the mass $\rho\eta_{neumann}$ of the *ghost triangles* can be set to the same value as its neighbor, given as

$$\rho\eta_{neumann}=\rho\eta_{Adj(neumann)} \tag{2.38}$$

To make the fluid reflect back into the domain, the momentum can be flipped by setting it to the negative value of its neighbor, giving

$$\rho\eta u_{neumann}=-\rho\eta u_{Adj(neumann)} \tag{2.39}$$

and

$$\rho\eta v_{neumann} = -\rho\eta v_{Adj(neumann)} \tag{2.40}$$

Chapter 3

# Related Work

This chapter looks at related works and experiments, concerning the Finite Volume Method and performance testing using the libraries OpenMP, BLAS and MPI, and at their relevance to this thesis.

## 3.1 Experiments for Finite Volume Method

In [23], an implementation of 2D shallow water equations using *FVM* is presented. With *Message Passing Interface*, a distributed memory architecture can split the domain into sub-domains and run the calculations in parallel. *FVM* is implemented by applying the conservation law to the shallow water equations. The water surface is split into grids of rectangular cells, with the equation formulated for each cell, where the average value is stored at its center. To find an estimate of the flux, a strategy based on the approximate Riemann solver of Roe is used. The entire domain is split into sub-domains, which is handed out among the processors so each gets an evenly amount of cells. The split is done vertically, so each sub-domain has two neighbors with a shared border. For each iteration, the processor communicates with its two neighbors, in order to update the overlapping cells along the border. Each processor simultaneously calculates their own cells, before starting over with the border exchange. The master node handles the partitioning and decides when to stop the iterations. Implementation and measurements showed a close to linear speedup as the number of processors increased, with some deviations based on technical specifications. Most of the time is spent on computation, while the overhead from communication stayed relatively small, even as the communication time increased with the number of processors. The workload is also evenly distributed, since the processes have the same amount of cells using the same function, giving little delay due to processors waiting for one another. Still, some delays occurred due to cache memory issues related to the master node using additional memory space, and during initial state when the master node has to distribute cells to all processors. Despite the increased performance, using a master node would not scale well, so a parallel I/O would be worth testing out.

A performance study comparing *FDM* and *FVM* is issued in [24], by using MAC algorithm and vorticity-stream function over lid-driven flow and natural flow, commonly used for computational fluid mechanics and numerical heat transfer. For MAC algorithm momentum and energy conservation are solved explicitly, while through vorticity-stream function, convection term is solved implicitly by deferred correction. Since vorticity-stream function is dependent on the boundary conditions, three types of conditions, *Thom*, *Woods* and *Jenson*, are issued. Uniform mesh is issued for both *FDM* and *FVM*, in addition to using second-order central differences and second-order upwind scheme, in order to make them comparable. Experiments show that both *FDM* and *FVM* have a high accuracy for high mesh density, but accuracy sinks faster for *FDM* as the mesh density decreases. *FVM* has a higher accuracy for all boundary conditions for vorticity-stream, and for both second-order central differences and upwind scheme, with upwind scheme indicating to give a better accuracy. With the same rounding error for *FDM* and *FVM*, *FVM* is shown to be more stable, as the mesh density sinks, and especially for Thom and Woods boundary conditions, where *FDM* diverges from physical meaning. *FVM* also shows to have a higher robustness, as it uses fewer iterations than *FDM* as the relaxation factor increases. This also indicates that *FVM* is more effective as it uses fewer iterations, which is confirmed through measurements, where *FVM* has a slightly faster calculation time given that *FVM* and *FDM* have the same mesh density.

*FVM* has a low flop-to-byte ratio and irregular memory access, making *FVM* memory bound and difficult to utilize on high-density computers [25]. Especially during the phase when the flux flow is calculated, a large amount of global memory access causes latency load problems, in addition to the registers being filled up. An algorithmic modification for GPU platforms is presented to handle the issue, by utilizing memory accessing. By observing the algorithm itself, global memory access can be reduced by finding overlapping memory access, and store them on a data depot. With *SMEM* memory, different threads can efficiently save compute and memory time by saving and reusing one other's data, saving around 50 percent of global reads. A further method is to pair up operations, so the accessed memory overlap. The first operation of an element can for example be paired up with the second operation of the neighbor element, reducing the global reads to 75 percent. For each group of threads with a shared memory, the *halo* around the effective area must be saved as well, which could result with an imbalance of workload of the threads, as some have to handle the halo processing. Separating boundary processing through *halo threads*, can release other threads from extra delay and restore balance. By testing out the algorithmic modifications on different types of GPUs and compared with hardware based optimizations, such as minimizing access latency of global memory, tuning register usage for each thread and splitting the kernel and streaming concurrency, it is found that different approaches work for different hardware. Still, implementing *halo threads* is beneficial for all. When comparing the hardware based optimized program with the fully optimized program, including all the modifications described above, the gained speedup is higher for more complex programs. In addition, having larger space of on-chip memory is of great importance. Though, an Euler solver gave 62 to 102 percent performance benefit, while the Shallow Water equation solver only gave 9 to 36 percent further performance speedup, these methods nevertheless give an overall performance boost to all tested hardware of *Kepler*, *Pascal*, and *Volta* GPU platforms.

## 3.2 Performance Testing for Libraries and Models

Rosales presents a performance analysis on the Xeon Phi coprocessor, a multicore chip with 61 cores [26]. A multiphased Lattice Boltzmann computation was applied for testing, which was found to be memory bound, meaning most of the performance went to bandwidth traffic. The Xeon Phi coprocessor also needed several threads per core in order to fully utilize its bandwidth. One of the aspects studied, was the scaling abilities for different distributions of threads to the cores, using OpenMP library. Fig. 3.1 shows the thread distribution to a single multiprocessor with four cores, of the three thread affinities *Compact*, *Scatter* and *Balanced*. *Compact* fills up one core at a time, *Scatter* distributes the threads in a round-robin matter, and *Balanced* distributes the threads evenly while keeping consecutive threads together. *Scatter* and *Balanced* gave the same performance when there was only a single thread for each core, where *Scatter* stopped improving when multiple threads were applied for each core. *Compact* had a poor utilization of the bandwidth when entire cores were left idle, and reached the same maximum performance as *Balanced* once all the cores were filled up. *Balanced* gave overall the best performance.

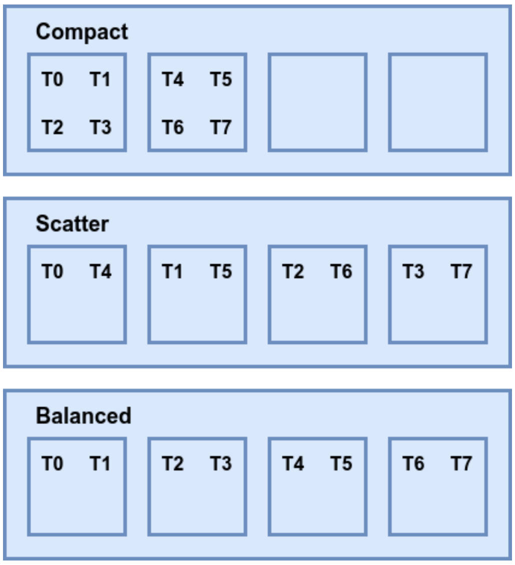


**Figure 3.1:** The thread affinities *Compact*, *Scatter* and *Balanced* for eight threads and a multicore processor with four cores.

The Intel Math Kernel Library (MKL) is a library for optimized math computations, which uses Basic Linear Algebra Subprograms (BLAS) specification for low level execution of linear algebra. Zuckerman presents a method to improve matrix multiplication for a parallel multicore processor, which is based on the sequential implementation of a unicore *DGEMM* kernel, before having it compared with MKL Parallel, a parallelization provided by Intel [27]. The different factors of matrix sizes, cache size and matrix blocking, and hardware specifications were explored, where the highest performance was accomplished with cache utilization. This was achieved by having block sizes fit into cache and avoiding cache pollution, which could be found through autotuning and benchmarking. The best results from the given method gave twice the performance than the MKL Parallel.

The *LogP* and *LogGP* performance model can be useful to predict efficiency of a parallel program running on a distributed system. Kielmann issues the challenge of measuring the parameters of *LogP* and *LogGP*, especially for long messages and measuring the gap parameter, and presents a way of handling this by using message passing platforms [28]. The parameters of *parameterized LogP* are presented, which uses $o$ og $g$ as functions $o(m)$ and $g(m)$ of the message size $m$. Additionally, $o(m)$ is split into $o_s(m)$ and $o_r(m)$ for overhead for the sender and receiver. All the original parameters from *LogP* and *LogGP* can be derived from these. The gap parameter has been measured before by saturating the communication link, to make it possible to observe the link capacity. This can disturb ongoing communication and consumes time for long messages, especially if the network has high latency or low bandwidth. The presented solution is to only saturate the communication link when measuring the gap $g(0)$, for messages of zero bytes. Two normal signals can be set up for two nodes, in order to find the other parameters. By measuring the round-trip time between the nodes, the latency $L$ and $g(m)$ can be calculated, in addition to having $o_s(m)$ and $o_r(m)$ to be directly measured. A sufficient number of measurements should be executed for different values of $m$, until the error rate is eligible. Some issues are also addressed if the nodes are asymmetrical or if the network flow is irregular. By using *MPI* with the functions `MPI_Send` and `MPI_Recv`, both the old and new method of measuring the gap has been implemented. The new method gave the same results, but with small deviations where cache misses sometimes gave larger values of $g(m)$. The time spent was significantly decreased with the new method, but it was still concluded to be too slow for real-time applications.

## 3.3 Relevance

The proxy application of this thesis implements the shallow water equations by using the finite volume method over a triangulation of the water surface. It is therefore interesting to study similar implementations, such as [23], which implements the SWE with FVM over a discretization of rectangular cells, and [24], which compares the finite volume method with the finite difference method. The application is executed on multicore processors by using MPI and OpenMP, which makes it interesting to study an SWE implemention with FVM for a GPU environment from [25].

The hardware from the Idun cluster has two multicore CPUs per node, and only allows a single thread for each core. The thread affinity explained from [26] is applied to the application, by deciding the order to assign the threads to the two CPUs. A simple matrix multiplication through the BLAS specification is used to find the maximum floating point performance for the benchmarking tests, where improvements of this method are explained in [27]. The application also makes an analysis of the communication time of message passing by using the Hockney model, where [28] uses the *LogP* and *LogGP* models.

Chapter 4

# Proxy Application

This chapter describes the proxy application, which consists of a preprocessing of the data and an implementation to execute the Shallow Water Equations.

The proxy application uses a triangulation to represent the water surface, and applies the Lax-Friedrichs method for a discretization of SWE from Section 2.3. The triangles are split into groups, to be distributed for different ranks. Each rank represents a node from a distributed system, a core from a multicore processor or a mixture of both, where each rank makes the SWE computations for their local triangles simultaneously. The ranks use message passing to exchange updated data for each neighboring triangle, and spawn parallel threads.

All the tests are based on a map of Mehamn harbor in the municipality of Gamvik, while the proxy application can be configured for other maps.

## 4.1 Preprocessing

The preprocessing includes generating a triangulation of a given map for a given number of vertices, and of distributing the triangles to the different ranks.

### 4.1.1 Triangulation

The triangulation is made based on the number of vertices $V$ from input. The script starts with generating the vertices by reading a color coded map, and distributes the vertices $v_i$ based on the coverage $a_i$ and on a predefined density $d_i$ of each color section $i$. Fig. 4.1 is a map of Mehamn harbor divided into four color sections. The green and red area covers the inner coast, which has a higher predefined density of vertices than the outer sea, colored with blue and yellow. After the vertices are generated, a Delaunay triangulation is made.

The total number of vertices $V$ is a summation of

$$V = \sum_{i=1}^{n} V_i \tag{4.1}$$

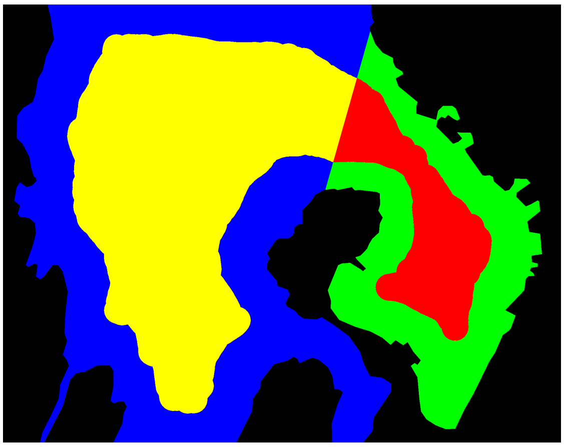


**Figure 4.1:** Map of Mehamn harbor in Gamvik municipality, Norway. The sea is divided in four colors, allowing varied densities of triangles for each section. The *height* and *width* denotes the vertical and horizontal dimensions of the map.

where $V_i$ is the number of vertices in each color section $i$ for a total of $n$ sections. The value $a_i$ is the water coverage for each color section $i$, adding up to

$$\sum_{i=1}^{n} a_i = 1 \tag{4.2}$$

which does not include the land area. The density $d_i$ of each color $i$ is defined as

$$\frac{V_i}{a_i} = d_i \cdot k \tag{4.3}$$

where $k$ is a constant for all $i$. This means $d_i \cdot k$ is a factor to determine how many vertices each sections has per unit of area, which gives a total vertices of

$$\sum_{i=1}^{n} V_i = \sum_{i=1}^{n} d_i \cdot k \cdot a_i \tag{4.4}$$

The factor $V_i/k$ can be calculated from the given $a_i$ and $d_i$ through

$$\frac{V_i}{k} = a_i \cdot d_i \tag{4.5}$$

which gives a total number of vertices

$$\sum_{i=1}^{n} \frac{V_i}{k} = \frac{V}{k} \tag{4.6}$$

giving

$$k = \frac{V}{\sum_{i=1}^{n} \frac{V_i}{k}} = \frac{V}{\sum_{i=1}^{n} a_i \cdot d_i} \tag{4.7}$$

Since $V$ and $V_i/k$ are known, $k$ can be found and used to find $V_i$ through

$$V_i = a_i \cdot d_i \cdot k = \frac{a_i \cdot d_i \cdot V}{\sum_{i=1}^{n} a_i \cdot d_i} \tag{4.8}$$

The next step is to find the coordinates of the vertices, so that each color section $i$ has $V_i$ evenly distributed vertices. The value $c_i$ is defined to describe the coverage of section $i$ of the map, including both the water and land area. For each color section, an overlapping collection of $V/c_i$ vertices are made that covers the entire area of the map. Fig. 4.2 shows the four color sections from Fig. 4.1, where each collection has $V/c_i$ vertices covering the entire area. The vertices outside its section are then removed, so each collection is left with $V_i$ vertices, before they merged together. Fig. 4.3 shows the remaining vertices after removing those outside of its section, merged together on the same map.

In order to find the coordinates for each section on Fig. 4.2, an estimate of the number of vertices on both axes are made, so the vertices can be evenly distributed in a grid pattern. The average number of vertices both horizontally and vertically is

$$avg(x_i, y_i) = \sqrt{\frac{V_i}{c_i}} \tag{4.9}$$

where $x_i$ is the estimated number of horizontal vertices for section $i$, and $y_i$ is the estimated number of vertical vertices for section $i$. The number of vertices on each axes is set to

$$x_i = avg(x_i, y_i) \cdot \frac{width}{avg(width, height)} \tag{4.10}$$

and

$$y_i = avg(x_i, y_i) \cdot \frac{height}{avg(width, height)} \tag{4.11}$$

where $height$ and $width$ are the vertical and horizontal dimensions of Fig. 4.1.

After generating the coordinates for all vertices $V$, a Delaunay triangulation is applied, where the edges that crosses land are manually removed. Fig. 4.4 shows the triangulation of Fig. 4.3.

### 4.1.2 Data Distribution

After generating the triangulation, the triangles are distributed to each processing unit, referred to as ranks. A preprocessing script takes the triangulation data and restructures it into new files before distribution.

First the triangles are sorted by their vertical coordinates $y$ of the triangles' center point. By dividing them evenly among the ranks, most neighboring triangles stay in the same group, which is shown in Fig. 4.5. Each rank keeps track of the neighboring triangles belonging to another group, so these are added to the file as well, referred to as *Halo triangles*. That means the ranks need to communicate to update the information of all *Halo triangles*. All triangles at the border are grouped together and placed at the end of the file right before the *Halo triangles*. These border and halo groups are sorted by an identification number *id*, so that a group of *Border triangles* for one rank is identical to the *Halo triangles* of the neighboring rank.

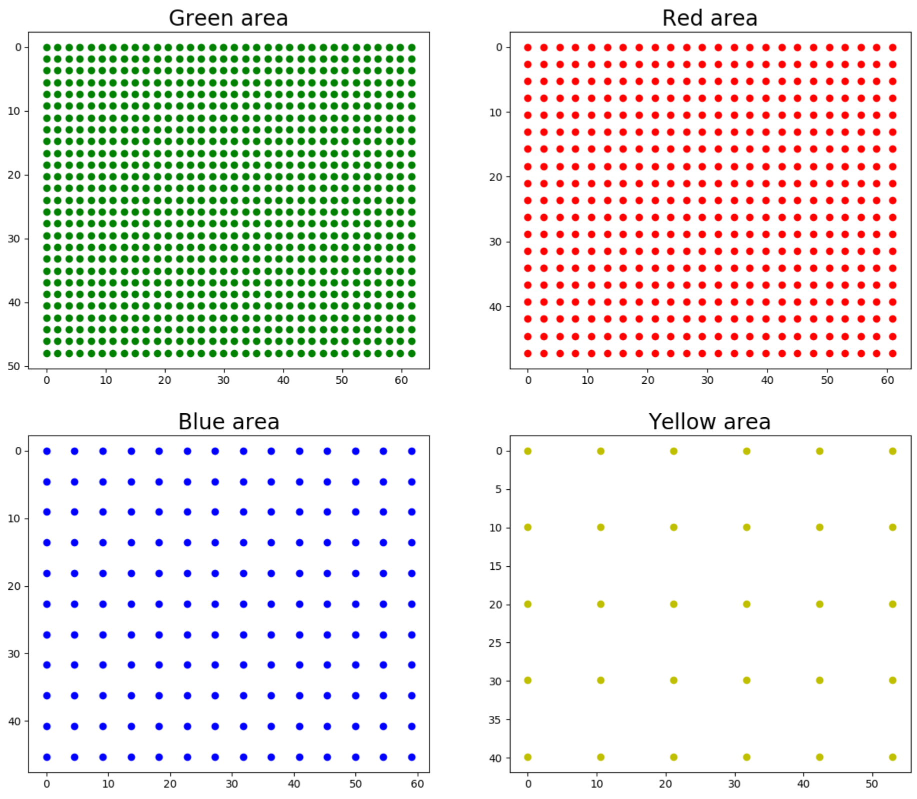


**Figure 4.2:** Vertices for each color section from Fig. 4.1, before the vertices outside its section is removed. Each section has a total of $V_i/c_i$ vertices, where $V_i$ is the final number of vertices and $c_i$ is the coverage for each section $i$. The total number of vertices is set as $V = 200$ and the density is set for $d_{green} = 50$, $d_{red} = 25$, $d_{blue} = 9$, and $d_{yellow} = 2$.

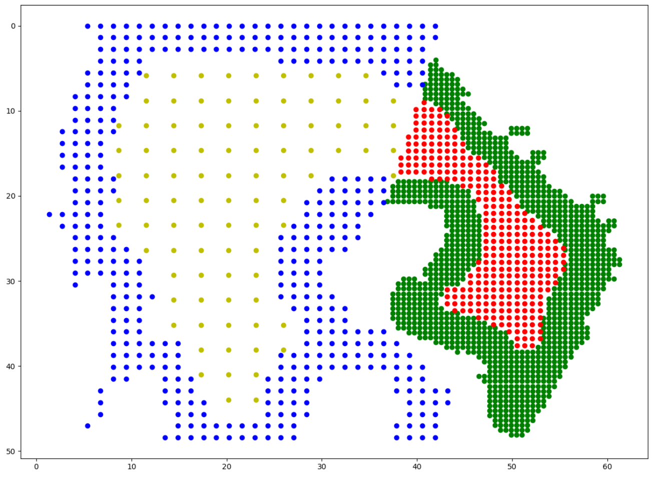


**Figure 4.3:** The coordinates of vertices after removing vertices outside its section and merging them together on the same map. The total number of vertices is set to be roughly $V = 2000$, giving 1922 vertices, and the density is set for $d_{green} = 50$, $d_{red} = 25$, $d_{blue} = 9$, and $d_{yellow} = 2$.

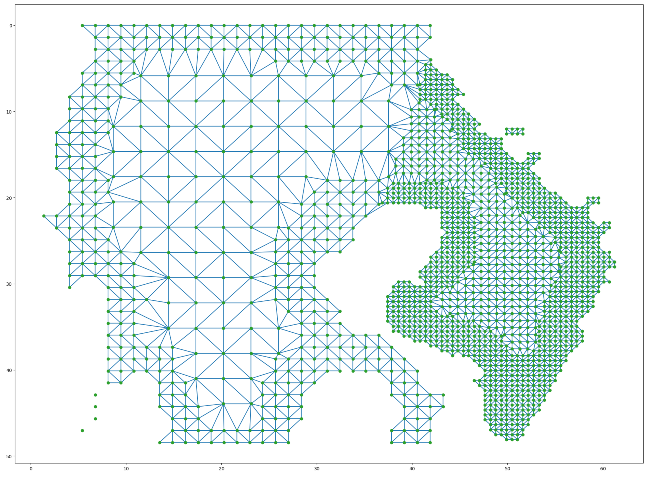


**Figure 4.4:** A Delaunay triangulation of the vertices from Fig. 4.3. The total number of vertices is set to be approximately $V = 2000$, giving 1922 vertices, and the density is set for $d_{green} = 50$, $d_{red} = 25$, $d_{blue} = 9$, and $d_{yellow} = 2$.

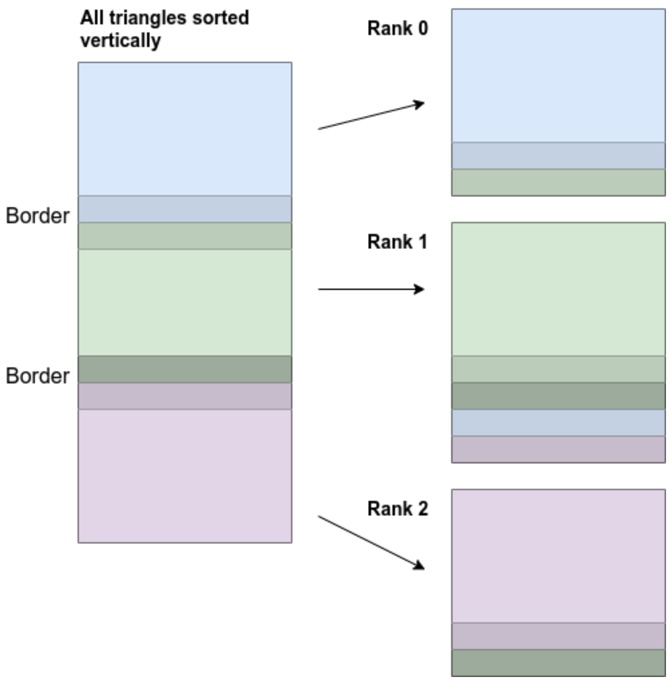
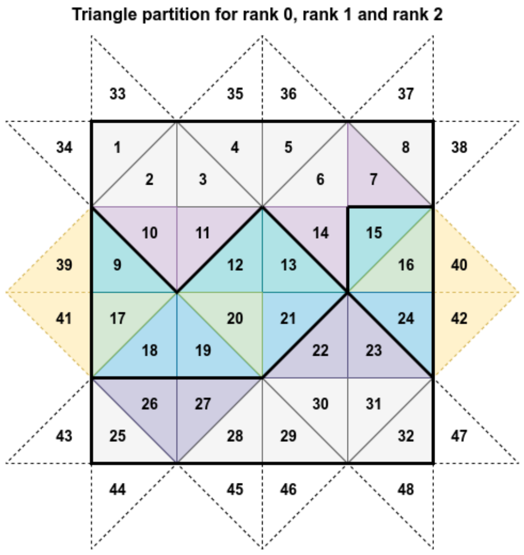
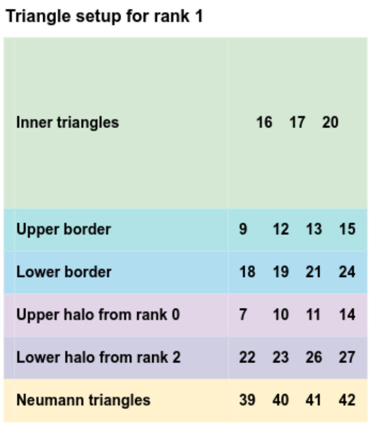


**Figure 4.5:** Illustration of how triangles are split among three ranks. The left shape represents a list of all triangles sorted by their vertical coordinates. The list is split evenly among the ranks, and the triangles bordering another group are marked and shown with a darker color. The right side shows each grouping of the triangles. The triangles along the border are placed at the bottom along with the *Border triangles* from the neighboring group. Every group with the same color tone contains the same triangles in the same order.

Fig. 4.6 shows an example of how triangles are organized for a rank. As shown in Fig. 4.5, the inner triangles are listed first, followed by the *Border* and *Halo triangles*. Mirror triangles are created along the entire border, called *Neumann triangles*. These triangles are created by mirroring triangles across the border for each triangle missing neighbors. *Neumann triangles* belonging to this rank are listed in the end.



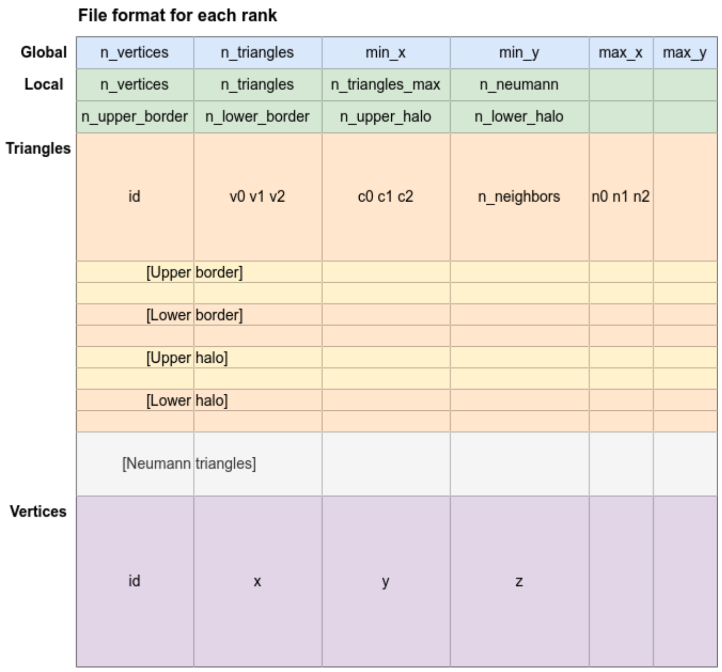

Triangle setup for rank 1

| | | | | |
|---|---|---|---|---|
| Inner triangles | 16 | 17 | 20 | |
| Upper border | 9 | 12 | 13 | 15 |
| Lower border | 18 | 19 | 21 | 24 |
| Upper halo from rank 0 | 7 | 10 | 11 | 14 |
| Lower halo from rank 2 | 22 | 23 | 26 | 27 |
| Neumann triangles | 39 | 40 | 41 | 42 |

**Figure 4.6:** An example of how triangles are organized for a rank. The left figure has 32 triangles split evenly among three ranks. Additionally, mirror triangles, with dotted edges, are added along the edge, and referred to as *Neumann triangles*. Triangles belonging to rank 1 are colored green and blue, with blue *Border triangles* and green inner triangles. *Halo triangles* for rank 1 are colored purple and *Neumann triangles* are colored yellow. The right figure shows the order of triangles for rank 1, starting with the inner triangles, the *Border triangles*, the *Halo triangles*, and finally the *Neumann triangles*.

With the triangles sorted out, each file can be created, shown in detail in Fig. 4.7. First all triangle neighbors, regardless if they belong to another group or the *Neumann triangles*, are found. Each triangle has at most three neighbors, in which the identification *id* are used. Then, a simple iteration is used to find the vertices relevant for each rank, by filtering out all vertices connected to any local triangle, *Halo triangle* or *Neumann triangle*. Information such as the total number of vertices and triangles are added to the file, in addition to the local number of vertices, triangles and *Neumann triangles*, and the number of border and *Halo triangles*. Then each triangle is listed, with its *id*, its three vertices, the center coordinates, number of neighbor triangles, and the *ids* of each neighbor. Finally the vertices are listed with *ids* and coordinates.

Table 4.1 sums up the different categorizations of the triangles. In total there are $T$ triangles, not including the *Neumann triangles*, where each rank gets $T_{local} = T/ranks$. Each rank has two sets of *Halo triangles* $T_{halo}$, one for the upper border and one for the lower border. $T_{neumann}$ is the total number of *Neumann triangles*, which gives approxi-



**File format for each rank**

| | | | | | | |
|---|---|---|---|---|---|---|
| **Global** | n_vertices | n_triangles | min_x | min_y | max_x | max_y |
| **Local** | n_vertices | n_triangles | n_triangles_max | n_neumann | | |
| | n_upper_border | n_lower_border | n_upper_halo | n_lower_halo | | |
| **Triangles** | id | v0 v1 v2 | c0 c1 c2 | n_neighbors | n0 n1 n2 | |
| | [Upper | border] | | | | |
| | | | | | | |
| | [Lower | border] | | | | |
| | | | | | | |
| | [Upper | halo] | | | | |
| | | | | | | |
| | [Lower | halo] | | | | |
| | | | | | | |
| | [Neumann | triangles] | | | | |
| **Vertices** | id | x | y | z | | |

**Figure 4.7:** A detailed description of a file belonging to a single rank. First line has global values for all triangles, including total number of vertices and triangles, and minimum and maximum coordinates for both axes. Second and third lines have values relevant to the corresponding rank, including the local number of vertices, triangles and *Neumann triangles*, the largest number of triangles among the ranks, and the number of each border and *Halo triangle*. All local triangles are then listed, with *Border triangles* and *Neumann triangles* at the end in the same way as in Fig. 4.6. Each triangles has an *id*, the *id* of its three vertices, the three coordinates of its center, number of neighbors, and the *id* of the neighboring triangles. Finally all vertices connected to the local triangles are listed, along with their *id* and *x*, *y* and *z* coordinates.

mately $T_{neumann}/ranks$ for each rank. $T_{tot}$ is the total number of triangles including the *Neumann triangles*, and each rank has a total of $T_{local} + 2 \cdot T_{halo} + T_{neumann}/ranks$ triangles.

| Term | Meaning |
|---|---|
| $T$ | Total number of triangles |
| $T_{local}$ | Number of triangles per rank |
| $T_{halo}$ | Length of a border between two ranks called *Halo triangles* |
| $T_{neumann}$ | Total number of *Neumann triangles* |
| $T_{tot}$ | Sum of $T$ and $T_{neumann}$ |

**Table 4.1:** List of the triangular terms.

## 4.2 Implementation

The implementation includes an overview of the data structure used in the program, how the program is executed, and how the program is distributed to the hardware.

### 4.2.1 Data Structure

The proxy application needs to keep track of both the triangular properties and the SWE attributes for each triangle. These categories are separated into two types of lists, with an array for the triangular properties, and a set of arrays for each attribute from the SWE.

#### Description

The program reads a file in the format of Fig. 4.7 for each rank. The *structs* for a triangle and a vertex are shown in Fig. 4.8. A triangle takes the attributes *id*, the three vertices, the center coordinates, the number of neighbors, and the neighboring triangles from the file, and configures local ids for all triangles. Each vertex uses the *id* and the coordinates from the same input file.

Fig. 4.9 illustrates how local *ids* are configured and used for each rank. The triangles are counted from *0* in the same order as in the file input. Hence, triangles can be listed in the same order, using the local *id* as an index, making accesses to neighboring triangles faster. Local *ids* are never shared with other ranks. Additionally, *Neumann triangles* are read from the file. These are mirrored triangles along the entire border, shown in Fig. 4.6, and have always one neighbor along the border. All local triangles have three neighbors, whether it be another local triangle, a *Halo triangle* or *Neumann triangle*.

#### Memory Space

Table 4.2 lists the datatypes used in the application. Both *Int* and *Double* take up 8 bytes. The triangular structure from Table 4.8 is represented through the type *Triangle*, where each triangle take up 128 bytes. The *Vertex* takes up $32B$, but is only used for preprocessing, meaning it is not stored in local memory for the execution of the program. Table 4.3

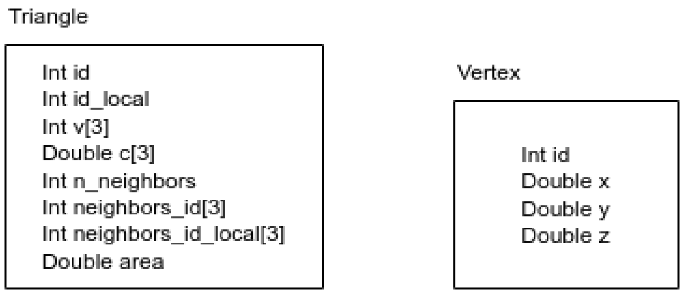


**Figure 4.8:** The structure of a triangle and a vertex. Each triangle has a global and local *id*, three vertices *ids*, the coordinates of its center, number of neighboring triangles, and a list of the neighbors' *ids* and local *ids*. Each vertex has an *id* and three coordinates for each axis.

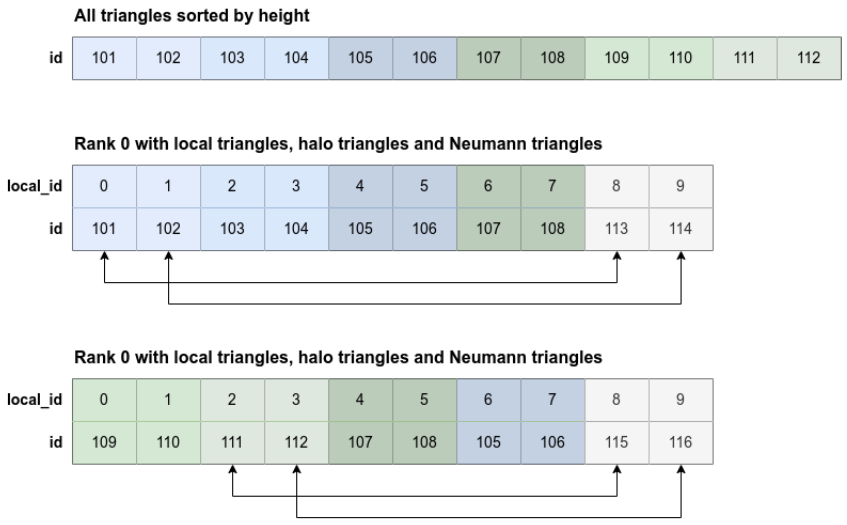

All triangles sorted by height
id
101
102
103
104
105
106
107
108
109
110
111
112
Rank 0 with local triangles, halo triangles and Neumann triangles
local_id
0
1
2
3
4
5
6
7
8
9
id
101
102
103
104
105
106
107
108
113
114
Rank 0 with local triangles, halo triangles and Neumann triangles
local_id
0
1
2
3
4
5
6
7
8
9
id
109
110
111
112
107
108
105
106
115
116

**Figure 4.9:** An example of how local *ids* are used when splitting triangles into groups. The triangles are first sorted by their vertical coordinates before they are split in two groups, colored with blue and green. The triangles that are at the border of the other group are marked with a darker color, and triangles at the edge that are missing a neighbor are marked with a lighter color. Rank 0 has the blue triangles in addition to the neighboring triangles, and rank 1 has the green triangles in addition to the neighboring triangles. Rank 0 and 1 also need to create mirror triangles, referred to as *Neumann triangles* and shown in Fig. 4.6, that border the triangles missing a neighbor. For rank 0, triangle *113* is a mirror of triangle *101*, always containing the same data as its mirror. All triangles are given a local id starting from *0* which is only visible for its rank.

lists the 11 attributes from the Shallow Water Equations, that are needed for each triangle in order to compute the mass and momentum flux.

The application has two types of arrays of significant memory space, where the first is a single array of the triangular properties and the second consists of 11 arrays for each attribute from the SWE. Table 4.4 describes the properties of the two, which has a total memory space of

$$T_{tot} \cdot 128B + 11 \cdot T_{tot} \cdot 8B = (T + T_{neumann}) \cdot 216B \tag{4.12}$$

for all ranks. The ranks on a multicore processor can use a shared memory, even as they only access their local triangles. If the ranks are distributed on different processors with their own memory, each rank takes up

$$\left(T_{local} + 2 \cdot T_{halo} + \frac{T_{neumann}}{ranks}\right) \cdot 216B \tag{4.13}$$

of memory space.

| Datatype | Memory space [B] |
|---|---|
| *Int* | 8 |
| *Double* | 8 |
| *Triangle* | 128 |
| *Vertex* | 32 |

**Table 4.2:** The datatypes and their memory space.

| SWE attribute | Description |
|---|---|
| $\rho\eta^n$ | Mass |
| $\rho\eta^{n+1}$ | Mass next |
| $\rho\eta u^n$ | Horizontal momentum |
| $\rho\eta u^{n+1}$ | Horizontal momentum next |
| $\rho\eta v^n$ | Vertical momentum |
| $\rho\eta v^{n+1}$ | Vertical momentum next |
| $\rho\eta uv$ | Momentum |
| $u$ | Horizontal speed |
| $v$ | Vertical speed |
| $\rho\eta u^2 + \frac{1}{2}\rho g\eta^2$ | Horizontal acceleration |
| $\rho\eta v^2 + \frac{1}{2}\rho g\eta^2$ | Horizontal acceleration |

**Table 4.3:** The 11 SWE attributes used in the proxy application. Each uses the datatype *Double* of $8B$.

| Array | Datatype | Elements | Memory space [B] |
|---|---|---|---|
| Triangles | *Triangle* | $T_{tot}$ | $T_{tot} \cdot 128$ |
| SWE attribute | *Double* | $11 \cdot T_{tot}$ | $11 \cdot T_{tot} \cdot 8$ |

**Table 4.4:** The arrays and their total memory space for all ranks, used for the proxy application, where $T_{tot} = T + T_{neumann}$.

### 4.2.2 Execution

Listing 4.1 shows an overview of the program code, which is executed for each rank. The main function is the outer frame of the program, and starts with the preprocessing from Section 4.1. The triangulation is created and distributed to the different ranks. The main part is an iteration for all *supersteps*, which is a computation of the Shallow Water Equations for a single time step, described in Section 2.3.4. After each computation, a halo exchange is executed, where the ranks communicate with their neighboring ranks through message passing, to give an update of the triangles along their border. The program finishes by freeing all memory and summarizing the runtime.

The computation function has three parts $p_1$, $p_2$ and $p_3$. The first part executes the boundary conditions for all *Neumann triangles*, which is defined to reflect the water back into the domain, and updates attributes from Table 4.3 for all triangles belonging to the rank. The second part computes both the horizontal and vertical momentum flux, and the third part computes the mass flux for each triangle $T_{local}$.

Table 4.5 lists the number of iterations, memory operations per iteration and FLOP per iteration, for each part $p_1$, $p_2$ and $p_3$. Both $p_2$ and $p_3$ access the arrays of Triangles and SWE attributes from Table 4.4, for all local triangles $T_{local}$. For $p_1$, the Triangles array is only accessed for the *Neumann triangles* $T_{neumann}$, which add up to a memory space of

$$T_{neumann} \cdot 128B + 11 \cdot T_{tot} \cdot 8B = T \cdot 88B + T_{neumann} \cdot 216B \tag{4.14}$$

for all ranks, where each rank uses

$$T_{local} \cdot 88B + \frac{T_{neumann}}{ranks} \cdot 216B \tag{4.15}$$

of memory space.

| Part | Iterations | Memory operations | FLOP |
|---|---|---|---|
| $p_1$ | $T_{neumann}/ranks$ | 8 | 0 |
| $p_1$ | $T_{neumann}/ranks + 2T_{halo} + T_{local}$ | 22 | 18 |
| $p_2$ | $T_{local}$ | 47 | 70 |
| $p_3$ | $T_{local}$ | 36 | 49 |

**Table 4.5:** The number of memory and FLOP operations and iterations for each part $p_1$, $p_2$ and $p_3$ in the proxy application.

```
void computation()
{
    /* PART 1 */
    //find boundary condition
    for each t in T_neumann {
        access_triangle(t);
        update_wave_attribute(t);
    }
    for each t in T_local, T_neumann, T_halo {
        update_wave_attribute(t);
    }

    /* PART 2 */
    //find momentum flux
    for each t in T_local {
        access_triangle(t);
        update_wave_attribute(t);
    }

    /* PART 3 */
    //find mass flux
    for each t in T_local {
        access_triangle(t);
        update_wave_attribute(t);
    }
}

void main()
{
    preprocessing();
    for each superstep {
        computation();
        halo_exchange();
    }
    finish();
}
```

**Listing 4.1:** Pseudocode of the proxy application. Each *superstep* has computation and halo exchange, and each computation step has three parts $p_1$, $p_2$ and $p_3$.

### 4.2.3 Hardware Distribution

A rank can be assigned to an entire processor of multiple cores or a subset of cores, which utilizes a shared memory. The processors can be a part of a multiprocessor system, distributed on separate nodes, or use a mixture of both. The iterations from Table 4.5 compute SWE attributes for independent triangles, meaning the computations can be parallelized. Each loop can be divided into threads, so the iterations are run simultaneously on all the cores assigned to the rank. Fig. 4.10 shows a distribution of four ranks to two processors, with eight cores per processor. Since each rank has four cores with a shared memory, they can spawn four parallel threads. Each rank must execute halo exchange through message passing, even if they share the same memory space.

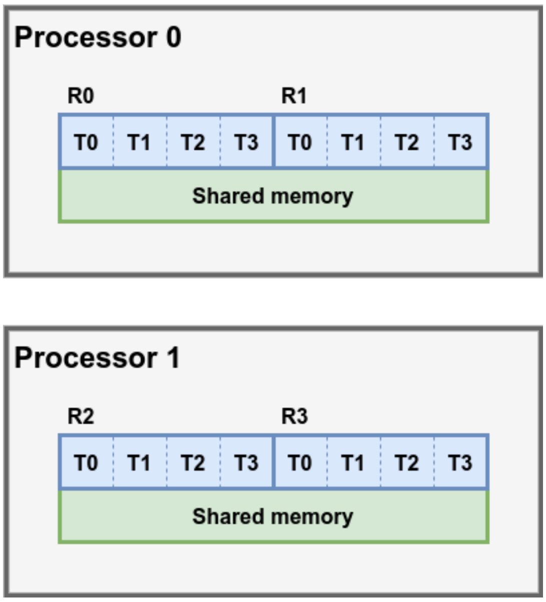


**Figure 4.10:** The distribution of four ranks to two processors, where each processor has eight cores. Each rank has four cores with a shared memory, allowing four threads to be created. The ranks, including those sharing a memory space, communicate through message passing to execute the halo exchange.

Chapter 5

# Performance Analysis

This chapter presents performance models of the application, by predicting the parameters of the generated triangulation, and creating a model of the computation time and communication time, based on the parameter predictions, where a deeper analysis of the scaling characteristics are made for the computation time. Both the predictions of the triangular parameters and runtime predictions are made for a triangulation of Mehamn harbor.

## 5.1 Predictions of Triangulation

A triangular mesh is used to represent the surface for the fluid flow. The mesh is split horizontally and shared evenly among the ranks, so they can process the mesh in parallel. The overall performance is dependent on the total number of triangles, the number of *Halo triangles* and the number of *Neumann triangles*, where *Halo triangles* represent the horizontal border size and *Neumann triangles* represents the outline around the domain. A rank has up to two neighboring ranks, adding up to two sets of *Halo triangles*. With Delaunay triangulation, the triangular boundaries can be estimated, given the number of vertices.

Table 5.1 lists the terms used for the structural predictions, so each type of triangles can be expressed with the number of vertices.

| Term | Meaning |
|---|---|
| $V$ | Total number of vertices |
| $T$ | Total number of triangles |
| $T_{local}$ | Number of triangles per rank |
| $T_{halo}$ | Length of a border between two ranks called *Halo triangles* |
| $T_{neumann}$ | Total number of *Neumann triangles* |

**Table 5.1:** List of terms used for predicting the number of triangles.

### 5.1.1 Total Triangles

To find a lower and upper bound for the number of triangles $T$, a continuous mesh of triangles is assumed, described in Fig. 5.1. This means there are no floating vertices without edges, shown in the left part of the figure, or any partitions where the two parts of the mesh are only connected by a single vertex, shown in the right part of the figure. Exceptions can appear after the Delaunay triangulation is created, when the edges crossing land are removed, shown in Fig. 5.2. These can appear along the coastline where small irregularities cannot be covered by triangles. Since most of the triangles, including the ones on the edge of the domain, follow the criteria, these exceptions can be ignored when estimating the lower and upper bound of the number of triangles.

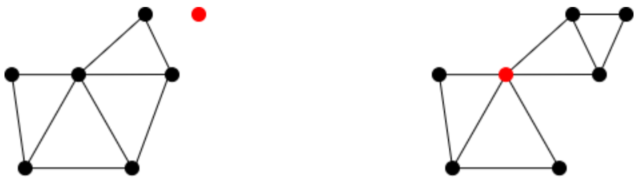


**Figure 5.1:** The triangular mesh is assumed to have no disconnected vertices, as in the left figure, and no parts only connected by a single vertex, such as the figure on the right.

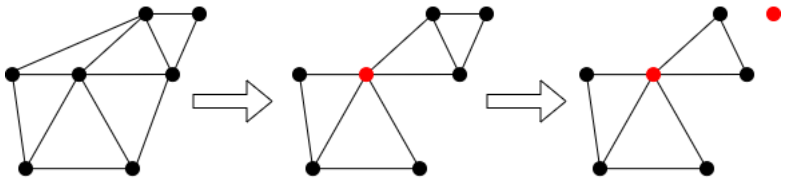


**Figure 5.2:** After the triangular mesh is created, the triangles crossing land are removed, and can leave exceptions to the assumptions, with sections connected through a single vertex or disconnected vertices.

Fig. 5.3 shows an initial state of a triangle and three vertices, with a new vertex being added for each step. Given the criteria from Fig. 5.1, at least one triangle must be created for every new vertex. This gives a lower bound for the number of triangles of

$$T \geq V - 2 \tag{5.1}$$

where $V \geq 3$ is assumed.

The Euler formula describes the relation between the number of vertices $v$, the number of edges $e$, and the number of regions $r$. A region is a closed cycle, which includes the unbound area outside the whole graph. From [29], the Euler formula states that

$$v - e + r = 2 \tag{5.2}$$

This can be proved by induction. A graph of a single vertex has one outer unbound region and no edges, giving

$$1 - 0 + 1 = 2 \tag{5.3}$$

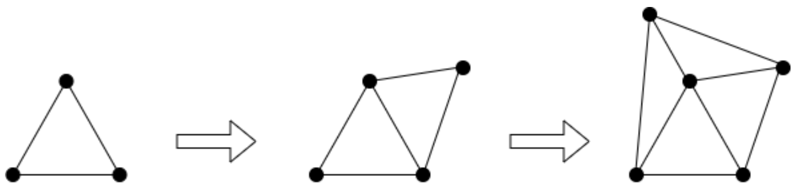


**Figure 5.3:** A triangular mesh, with one vertex being added at a time. For every new vertex, a new triangle must be created.

Eq. (5.2) can be assumed to be correct for all graphs up to $e = n$ edges. For a graph of $e = n + 1$ edges, one edge from a cycle can be removed. This will reduce the number of regions by one, as two regions are merged. This gives

$$v - (e - 1) + (r - 1) = v - e + r = 2 \tag{5.4}$$

If the graph has no cycles, a vertex connected to only one edge can be removed as well as the connecting edge, which gives

$$(v - 1) + (e - 1) + r = v - e + r = 2 \tag{5.5}$$

Thus, the equation holds for all number of edges.

Each edge in a triangular mesh has up to two connecting triangles. If all possible triangles are counted for each edge $e$, while allowing overlapping triangles, the sum is $2 \cdot e$ possible triangles. During the count, all the existing triangles $T$ are counted three times, since each triangle has exactly three edges. This gives

$$3 \cdot T \leq 2 \cdot e \Rightarrow \frac{3T}{2} \leq e \tag{5.6}$$

In the triangular mesh, there are fewer triangles than regions, since bent edges are not allowed and other shapes than triangles are not counted. From Eq. (5.2), $v$, $e$ and $r$ can be expressed with the number of vertices $V$ and the number of triangles $T$, giving

$$V - \frac{3T}{2} + T \geq 2 \tag{5.7}$$

resulting in an upper bound of

$$T \leq 2V - 4 \tag{5.8}$$

With the total number of triangles within

$$T \in [V - 2, 2V - 4] \tag{5.9}$$

the size of $T$ depends on the size of the coastline. A long and narrow river would create a mesh closer to the lower bound, while a large gulf or an ocean would create a mesh closer to the upper bound.

Since the triangles are shared evenly among the ranks, each rank has $T/ranks$ triangles. Given Eq. (5.9), the local triangles are defined as

$$T_{local} \in \left[\frac{V - 2}{ranks}, \frac{2V - 4}{ranks}\right] \tag{5.10}$$

### 5.1.2 Halo and Neumann Triangles

For a serial program, there is no horizontal partition of the triangular mesh, resulting in zero *Halo triangles*. For two or more ranks, the number of triangles are distributed evenly. A rank's halo cannot be exceed $T_{local}$, since the halo can only overlap with the neighboring rank's local triangles. Two ranks give the maximum $T_{local} = T/2$, making the bounds of *Halo triangles*

$$T_{halo} \in \left[0, \frac{T}{2}\right] = [0, V-2] \tag{5.11}$$

There must at least be three *Neumann triangles*, since the whole the domain is surrounded by these. A continuous mesh of triangles is assumed from Fig. 5.1, meaning each triangle must have at least one neighbor and a maximum of two Neumann edges, making the number of *Neumann triangles* within

$$T_{neumann} \in [3, 2 \cdot T] = [3, 4V-8] \tag{5.12}$$

where each rank has an average of

$$\frac{T_{neumann}}{ranks} \in \left[\frac{3}{ranks}, \frac{4V-8}{ranks}\right] \tag{5.13}$$

## 5.2 Performance Model

BSP is a model for cost prediction of a parallel program, that runs iterations over the same *superstep* [16]. Each *superstep* has computations over the data, communications between the different parallel processing units, and a synchronization process. The estimated total runtime is a summation of all *supersteps* $S$, given as

$$\sum_{s=1}^{S} t_s = \sum_{s=1}^{S} (w_s + gh_s + l) \tag{5.14}$$

where $t_s$ is the runtime, $w_s$ is the maximum computation time for one processing unit, $g$ is the communication bandwidth, $h_s$ is the maximum number of messages between a pair of two processing units, and $l$ the is synchronization latency.

From [11], the runtime $t_s$ for each *superstep* can be written as

$$t_s = t_{comp} + t_{comm} - t_{overlap} \tag{5.15}$$

by setting $t_{comp} = w_s$ and $t_{comm} = gh_s + l$. The value $t_{overlap}$ is the runtime of the overlap between the computations and communications. For the proxy application, computations and communications are separated with a barrier, making $t_{overlap} = 0$.

Fig. 5.4 shows the details of $t_{comp}$ and $t_{comm}$. Each rank has an equal share of $T_{local}$ triangles. They are assigned to different physical processing units, and run the computations in parallel for the computation time $t_{comp}$. The ranks can be assigned to multiple processors, and spawn parallel *threads*, so the computations of $T_{local}$ can be done in parallel as well. This sums up to a total of $ranks \cdot threads$ parallel processing units, also called *processes*. A barrier makes the ranks wait until all the computations are finished, in order to make sure the data is ready before exchanging the *Halo triangles* $T_{halo}$ with the upper and lower neighboring rank. The time to execute halo exchange is referred to as the communication time $t_{comm}$.

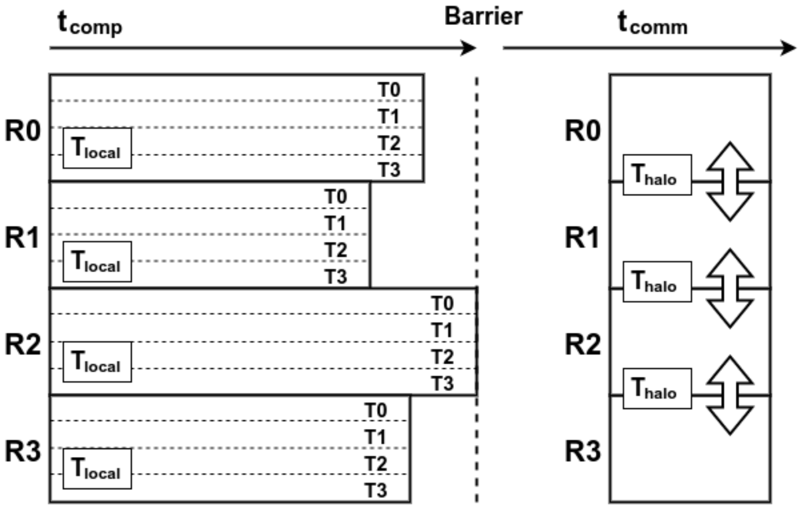


**Figure 5.4:** Details of the computation time $t_{comp}$ and communication time $t_{comm}$ for four $ranks$ and $threads$. During $t_{comp}$, each of the ranks $R0$, $R1$, $R2$ and $R3$ computes their local triangles $T_{local}$. Each rank has four parallel threads $T0$, $T1$, $T2$ and $T3$, which sums up to 16 parallel processing units. A barrier forces the ranks to wait before they can exchange the halo $T_{halo}$ for the communication part.

## 5.2.1 Computation Time

The value $t_{comp}$ represents the runtime for calculating all the triangles $T$ for each iteration. It has three parts, $p_1$, $p_2$ and $p_3$. The first part is a preprocessing of the triangular mesh, the second part calculates the momentum flux, and the third part calculates the mass flux.

An ideal parallel program has the runtime of

$$t_{comp} = \frac{t_T \cdot T_{local}}{threads} = \frac{t_T \cdot T}{ranks \cdot threads} \tag{5.16}$$

where $t_T$ is the computation time of single triangle. In this case, there is a linear speedup and no overhead as the number of parallel processes increases. For the proxy application, each rank has two sets of *Halo triangles*, $2 \cdot T_{halo}$, and an average of $T_{neumann}/ranks$ *Neumann triangles*. The runtime $t_{comp}$ is dependent on these variables as well, which makes Eq. (5.16) too simplified.

To make a detailed estimation of $t_{comp}$, the number of byte operations and floating point operation can be counted and compared. The arithmetic intensity is the ratio between floating point and memory operations, and can be used to find the maximum attainable performance through the roofline model [13].

### Memory Function

The memory operations can be counted for each part $p_1$, $p_2$ and $p_3$. Table 5.2 lists the number of iterations for each part, and the number of $8B$ memory operations for each iteration.

| Part | Iterations | Memory operations |
|---|---|---|
| $p_1$ | $T_{neumann}/ranks$ | 8 |
| $p_1$ | $T_{neumann}/ranks + 2T_{halo} + T_{local}$ | 22 |
| $p_2$ | $T_{local}$ | 47 |
| $p_3$ | $T_{local}$ | 36 |

**Table 5.2:** The number of memory operations and iterations for each part $p_1, p_2$ and $p_3$ in the proxy application.

Each part has the total of

$$p_1 = \left(\left(\frac{T_{neumann}}{ranks}\right) \cdot 8 + \left(\frac{T_{neumann}}{ranks} + 2 \cdot T_{halo} + T_{local}\right) \cdot 22\right) \cdot 8B \tag{5.17}$$

$$p_2 = (T_{local} \cdot 47) \cdot 8B \tag{5.18}$$

and

$$p_3 = (T_{local} \cdot 36) \cdot 8B \tag{5.19}$$

By setting $T_{local} = \frac{T}{ranks}$, this sums up to

$$p_1 + p_2 + p_3 = \left(240 \cdot \frac{T_{neumann}}{ranks} + 352 \cdot T_{halo} + 840 \cdot \frac{T}{ranks}\right) B \tag{5.20}$$

### FLOP Function

Table 5.3 lists the number of iterations for each part $p_1$, $p_2$ and $p_3$, and the number of floating point operations for each iteration.

| Part | Iterations | FLOP |
|---|---|---|
| $p_1$ | $T_{neumann}/ranks + 2T_{halo} + T_{local}$ | 18 |
| $p_2$ | $T_{local}$ | 70 |
| $p_3$ | $T_{local}$ | 49 |

**Table 5.3:** The number of floating point operations and iterations for each part $p_1, p_2$ and $p_3$ in the proxy application.

The total number of floating point operations for each part is

$$p_1 = \left(\frac{T_{neumann}}{ranks} + 2 \cdot T_{halo} + T_{local}\right) \cdot 18FLOP \tag{5.21}$$

$$p_2 = T_{local} \cdot 70FLOP \tag{5.22}$$

and

$$p_3 = T_{local} \cdot 49FLOP \tag{5.23}$$

Having $T_{local} = \frac{T}{ranks}$, this sums up to

$$p_1 + p_2 + p_3 = \left(18 \cdot \frac{T_{neumann}}{ranks} + 36 \cdot T_{halo} + 137 \cdot \frac{T}{ranks}\right) FLOP \tag{5.24}$$

**Arithmetic Intensity**

The arithmetic intensity of the proxy application is the ratio of the total byte operations from Eq. (5.20) and the total floating point operations from Eq. (5.24), given as

$$AI = \frac{18 \cdot \dfrac{T_{neumann}}{ranks} + 36 \cdot T_{halo} + 137 \cdot \dfrac{T}{ranks}}{240 \cdot \dfrac{T_{neumann}}{ranks} + 352 \cdot T_{halo} + 840 \cdot \dfrac{T}{ranks}} FLOP/B \tag{5.25}$$

Eq. (5.25) converges as each of all the four variables, $T$, $T_{halo}$, $T_{neumann}$, and $ranks$, approaches infinity, resulting in

$$\lim_{T \to \infty} AI = \frac{137}{840} \approx 0.163FLOP/B \tag{5.26}$$

$$\lim_{T_{halo} \to \infty} AI = \frac{36}{352} \approx 0.102FLOP/B \tag{5.27}$$

$$\lim_{T_{neumann} \to \infty} AI = \frac{18}{240} \approx 0.075FLOP/B \tag{5.28}$$

and

$$\lim_{ranks \to \infty} AI = \frac{36}{352} \approx 0.102FLOP/B \tag{5.29}$$

The arithmetic intensity has a maximum of $0.163FLOP/B$, for an infinite number of triangles $T$.

From Section 5.1.2, both $T_{halo}$ and $T_{neumann}$ are limited by $T$, and since they are both $\mathcal{O}(T)$, they cannot approach infinity faster than $T$. In order to find the minimum arithmetic intensity, the values $T_{halo}$ and $T_{neumann}$ must be maximized, so the impact of $T$ is lessened. This gives

$$AI_{min} = \frac{18 \cdot \frac{2 \cdot T}{ranks} + 36 \cdot \frac{T}{2} + 137 \cdot \frac{T}{ranks}}{240 \cdot \frac{2 \cdot T}{ranks} + 352 \cdot \frac{T}{2} + 840 \cdot \frac{T}{ranks}} FLOP/B = \frac{18 + \frac{173}{ranks}}{176 + \frac{1320}{ranks}} FLOP/B \tag{5.30}$$

for the minimum value $AI_{min}$. The minimum arithmetic intensity is found when $ranks$ approaches infinity, which is $0.102FLOP/B$, the same for when $T_{halo}$ approaches infinity, as $T_{halo}$ is the only value not divided by $ranks$ in Eq. (5.25). The value $ranks$ is limited by the number of physical processors, which makes the absolute minimum unlikely. For a single rank, the lowest arithmetic intensity is

$$AI_{min}(ranks = 1) = \frac{18 + \frac{173}{1}}{176 + \frac{1320}{1}} FLOP/B \approx 0.128FLOP/B \tag{5.31}$$

With an arithmetic intensity of

$$AI \in [0.102, 0.163] \tag{5.32}$$

the proxy application is memory bound, meaning the floating point operations will not be the main performance bottleneck. From the roofline model, the highest attainable performance is determined by the memory bandwidth. The computation time $t_{comp}$ can be estimated using only the number of memory operations from Eq. (5.20), formulated as

$$\begin{gathered} t_{comp} = \frac{(p_1 + p_2 + p_3) \cdot 8B}{bandwidth \cdot threads} = \\ \left(240 \cdot \frac{T_{neumann}}{ranks} + 352 \cdot T_{halo} + 840 \cdot \frac{T}{ranks}\right) \frac{B}{bandwidth \cdot threads} \end{gathered} \tag{5.33}$$

### 5.2.2 Scaling of Computation Time

Scalability is the ability to handle work as the resources increase. The predicted computation time, $t_{comp}$ from Eq. (5.33), can be analyzed for its ability to handle the triangles $T$, $T_{halo}$ and $T_{neumann}$ for the number of processes, given by $ranks$ and $threads$.

#### Load Balancing

Load balancing is the distribution of workload for the processes. In Fig. 5.4, the computation time $t_{comp}$ is limited by the slowest rank, since the barrier forces the other ranks to wait. Each rank must also wait for its own threads to finish. An unbalanced workload causes a longer wait. This adds up as overhead for speedup, and gets more significant for larger amount of data to compute.

In the proxy application, the triangles $T$ are distributed evenly among the ranks. The halo and *Neumann triangles* $T_{halo}$ and $T_{neumann}$ can vary for each rank depending on the triangulation, and can cause imbalance and overhead for scaling of ranks. For a triangulation where $T$ is significantly larger than $T_{halo}$ and $T_{neumann}$, the imbalance caused by $T_{halo}$ and $T_{neumann}$ gives little overhead, meaning they must be close to their absolute maximum to have an effect for scaling.

Each rank distributes workload from loop iterations to the threads, which gives each thread an equal workload regardless of the number of $T_{halo}$ and $T_{neumann}$. This means the data distribution, given from the triangulation, does not add overhead for thread scaling.

For the computation time prediction Eq. (5.33), an equal number of triangles $T_{local}$, *Halo triangles* $T_{halo}$ and *Neumann triangles* $T_{neunmann}$ is assumed, as well as an equal workload for all threads. The predicted overhead for scaling is thereby not affected by load imbalance.

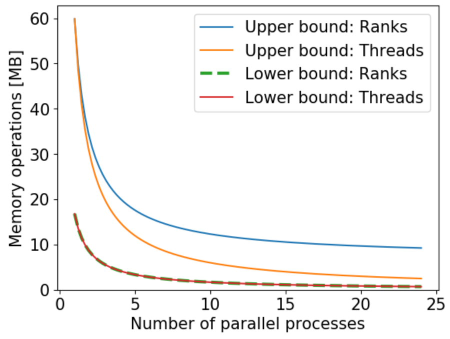
#### Ranks and Threads

Fig. 5.5 shows strong and weak scaling of Eq. (5.33). The number of processes describes the number of parallel processing units, which can be both ranks and threads. The scaling for ranks and threads are separated. For a scaling of ranks, each rank spawns a single thread, and for scaling of threads, a single rank spawns all the threads. Eq. (5.33) uses the number of memory operations divided by the bandwidth to predict runtime, as the proxy application is memory bound. Since the bandwidth is unknown, the number of memory operations is used to describe the predicted performance, being only a constant away from the predicted runtime. The upper and lower bounds of predicted performance is found by setting $T$, $T_{halo}$ and $T_{neumann}$ to their maximum and minimum, given a fixed number of vertices $V$ for strong scaling and a fixed number of vertices $V$ per processing unit for weak scaling.

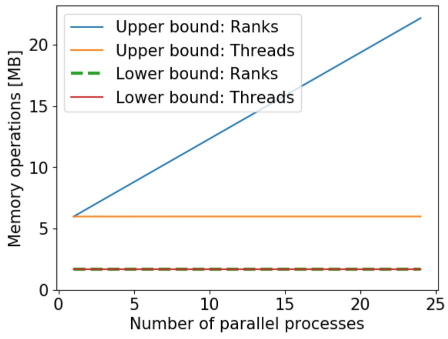
Fig. 5.5a shows strong scaling of Eq. (5.33) with a total number of vertices of $V = 20000$. In Eq. (5.33), only $T$ and $T_{neumann}$ is shared for $ranks$, meaning their impact shrinks for rank scaling, while $T_{halo}$ stays unaffected. For thread scaling, all the triangular values are divided by $threads$. For the upper bound, the difference between ranks and threads is shown, as rank scaling converges toward the constant provided by $T_{halo}$ while thread scaling converges toward zero. The lower bound has no *Halo triangles*, which gives the same scaling for $ranks$ and $threads$.

Fig. 5.5b shows weak scaling of Eq. (5.33) with $V = 2000$ for each process. For the upper bound, the number of memory operations increases linearly for rank scaling, since $T_{halo}$ can grow unaffected by $ranks$, while $T/ranks$ and $T_{neumann}/ranks$ stay as a constant. For thread scaling, all triangles are divided by $threads$, which gives a constant number of memory operations. For the lower bound, there are zero *Halo triangles*, which makes the affect of ranks and threads the same.

Ideally, strong and weak scaling follow the pattern of Fig. 5.5 for an infinite number of triangles and processes. Because of hardware limitations, such as limited memory storage, the performance is expected to fall at a certain problem size.



**(a)** Predicted performance for strong scaling, with a total of 20000 vertices.



**(b)** Predicted performance for weak scaling, with 2000 vertices per process.

**Figure 5.5:** A performance estimation for strong and weak scaling, based on Eq. (5.33). Since the bandwidth is unknown, the number of memory operations is displayed, as this is a factor of the predicted runtime. The number of parallel processors is the product of $ranks$ and $threads$. For a given number of vertices, the upper and lower performance is found, by setting $T_{local}$, $T_{halo}$ and $T_{neumann}$ to maximum and minimum. Both scaling for $ranks$ and $threads$ are used, by fixing the other to 1.

### 5.2.3 Communication Time

The value $t_{comm}$ describes the time spent on halo exchange, where all the ranks exchange *Halo triangles* with their upper and lower neighboring rank. The exchange is assumed to be handled simultaneously, giving a constant communication time for any number of ranks, so $t_{comm}$ can be given as the total exchange time for a single rank with both its neighbors.

The Hockney model formulates the communication time between two processors as a linear function

$$t(n) = \alpha + \beta^{-1} n \tag{5.34}$$

for the message size $n$, with bandwidth $\beta$ and latency $\alpha$ [17]. The function is linear until the message size reaches a maximum, where additional overhead starts to show.

The function of the Hockney model can be applied to the communication estimation from Eq. (5.14) and Eq. (5.15), resulting in

$$t_{comm} = gh_s + l = \beta^{-1} n + \alpha \tag{5.35}$$

where the communication in a *superstep* from the BSP model $gh_s$ is set as $\beta^{-1}n$, and the synchronization latency $l$ from BSP is set as the communication latency $\alpha$.

The proxy application exchanges data for mass flux $\rho\eta$, and momentum flux $\rho\eta u$ and $\rho\eta v$ with both upper and lower neighbor, which adds up to $2 \cdot 3$ messages. Each message is an array of 8 byte doubles, with the size of the number of *Halo triangles* $T_{halo}$. This gives a communication estimate of

$$t_{comm} = 2 \cdot 3 \cdot \left(\beta^{-1}(8B \cdot T_{halo}) + \alpha\right) \tag{5.36}$$

This model assumes that $8B \cdot T_{halo}$ is within the message size $n$ where the function stays linear.

## 5.3 Case Study: Mehamn Harbor

The proxy application uses a triangulation of Mehamn harbor, shown in Fig. 5.6. Mehamn harbor is located in Gamvik municipality, and have two breakwaters arranged by the Norwegian Coastal Administation, where it is of interest to make wave simulations [1]. The map has four sections with different numbers of vertices, which has a higher density of vertices in the sections along the coastline. Each section has vertices placed in a grid pattern, so the distances within a section have the same distance vertically and horizontally. A Delaunay triangulation is made for these vertices, and the triangulation is split into even horizontal segments, so each rank can calculate its own segment in parallel. An approximation of the structure can be made using the properties of the map, and further approximate the performance of the proxy application.

### 5.3.1 Predictions of Triangulation

A triangulation of a grid will create twice as many triangles as vertices, where two triangles are made for each rectangle. Since there are different sections of grids for the Mehamn

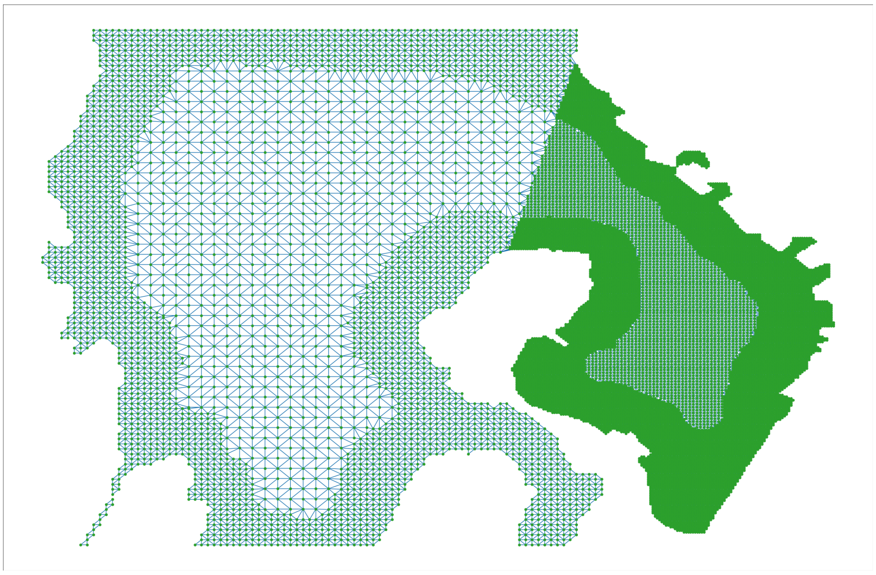


**Figure 5.6:** A triangulation of Mehamn harbor using 20000 vertices.

map, irregular triangles are created at the borders of the sections. Most of the triangles are placed completely within a section, so the total number of triangles $T$ can be approximated as

$$T \approx 2V \tag{5.37}$$

The mesh is split evenly among the ranks, which makes each segment

$$T_{local} \approx \frac{2V}{ranks} \tag{5.38}$$

The triangulation is based on a 2088 × 1638 image of the Mehamn harbor. In order to make each section of the grid squared, the number of vertices vertically and horizontally are adjusted based on the image height and width, estimated as

$$V_{vertical} = \sqrt{V} \cdot \frac{height}{\frac{width+height}{2}} = \sqrt{V} \cdot \frac{1638}{\frac{2088+1638}{2}} \approx \sqrt{V} \cdot 0.88 \tag{5.39}$$

and

$$V_{horizontal} = \sqrt{V} \cdot \frac{width}{\frac{width+height}{2}} = \sqrt{V} \cdot \frac{2088}{\frac{2088+1638}{2}} \approx \sqrt{V} \cdot 1.12 \tag{5.40}$$

The halo of a rank is expected to be a straight line, since the triangular mesh is split horizontally. This map in particular has roughly two parts, where the right part has the two most dense sections, and the left part has the two most sparse sections. A vertical line will have a varied number of triangles, since it depends whether it is placed on the right or left side of the map, but any horizontal line can be approximated to be of the same size, making the estimation of halo

$$T_{halo} \approx V_{horizontal} \approx \sqrt{V} \cdot 1.12 \tag{5.41}$$

Given the number of vertices, *Neumann triangles* can be assumed to increase in the same rate as the *Halo triangles*. An approximation of *Neumann triangles* can be written as

$$T_{neumann} \approx k \cdot T_{halo} \tag{5.42}$$

where $k$ is a constant of the ratio. The constant $k$ can be found by using $T_{neumann}$ and $T_{halo}$ for a triangulation of a specific number of vertices $V$. Table 5.4 lists the total number of *Neumann triangles* and the number of *Halo triangles* for each rank, given a triangulation of 20000 vertices and four ranks.

| Neumann | Rank 0 | Rank 1 | Rank 2 | Rank 3 |
|---|---|---|---|---|
| 1156 | 0 | 138 | 138 | 145 |
| | 137 | 136 | 146 | 0 |

**Table 5.4:** The total number of *Neumann triangles* and the number of *Halo triangles* for each rank, for 20000 vertices and four ranks.

A lower number of vertices will ignore the small details surrounding the coast, which results in a shorter estimated coastline. This effect becomes less significant as the number of vertices increases, and more of the details are covered. Assuming that a triangulation of 20000 vertices, from Fig. 5.6, has yet to cover the full size of the coast, the smallest halo from Table 5.4 can be picked, so $k$ can be slightly overestimated, resulting in

$$k = \frac{T_{neumann}(V = 20000)}{min(T_{halo}(V = 20000))} = \frac{1156}{136} = 8.5 \tag{5.43}$$

For all number of vertices, the number of *Neumann triangles* can be formulated as

$$T_{neumann} \approx 8.5 \cdot T_{halo} \approx 8.5\sqrt{V} \cdot 1.12 \tag{5.44}$$

with an average of

$$\frac{T_{neumann}}{ranks} \approx \frac{8.5\sqrt{V} \cdot 1.12}{ranks} \tag{5.45}$$

for each rank.

## 5.3.2 Performance Model

An estimation of a *superstep* from Eq. (5.15) can be applied to Mehamn harbor by using the new triangular estimates of $T$, $T_{halo}$ and $T_{neumann}$ for $t_{comp}$ and $t_{comm}$.

### Computation Time

The computation time $t_{comp}$ is memory bound, where each of the parts $p_1$, $p_2$ and $p_3$ can be taken from the memory operations of Eq. (5.17), Eq. (5.18) and Eq. (5.19). By applying the new values of $T$, $T_{halo}$ and $T_{neumann}$, each part can be written as

$$\begin{gathered}p_1 = \frac{T_{neumann}}{ranks} \cdot 8 + \left(\frac{T_{neumann}}{ranks} + 2 \cdot T_{halo} + T_{local}\right) \cdot 22 = \\ \left(\frac{8.5\sqrt{V} \cdot 1.12}{ranks}\right) \cdot 8 + \left(\frac{8.5\sqrt{V} \cdot 1.12}{ranks} + 2 \cdot \sqrt{V} \cdot 1.12 + \frac{2V}{ranks}\right) \cdot 22\end{gathered} \tag{5.46}$$

$$p_2 = T_{local} \cdot 47 = \frac{2V}{ranks} \cdot 47 \tag{5.47}$$

and

$$p_3 = T_{local} \cdot 36 = \frac{2V}{ranks} \cdot 36 \tag{5.48}$$

which gives a total of

$$\begin{gathered}t_{comp} = \frac{(p_1 + p_2 + p_3) \cdot 8B}{bandwidth \cdot threads} = \\ \left(\frac{210V}{ranks} + \frac{285.6\sqrt{V}}{ranks} + 49.28\sqrt{V}\right) \frac{8B}{bandwidth \cdot threads}\end{gathered} \tag{5.49}$$

### Load Balancing

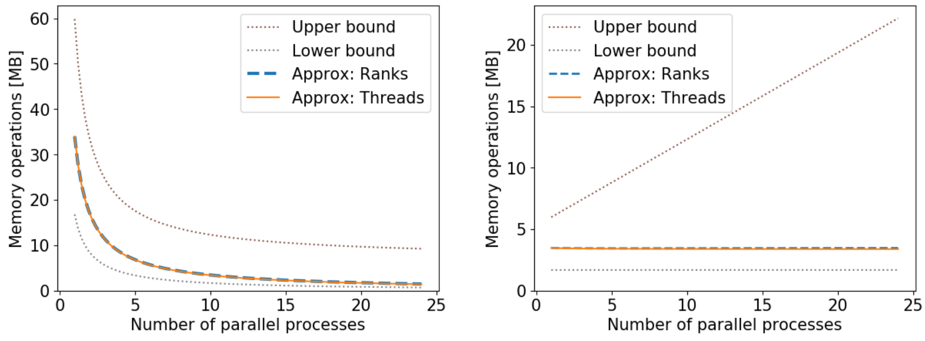
Each of the ranks has an equal distribution of local triangles $T_{local}$, but can have varied *Halo triangles* $T_{halo}$ and *Neumann triangles* $T_{neumann}$. The values $T_{halo}$ and $T_{neumann}$ are significantly smaller than $T$, as $T_{halo}$ and $T_{neumann}$ are factors of $\sqrt{T}$, meaning an imbalanced distribution of $T_{halo}$ and $T_{neumann}$ will cause little overhead of the speedup. The time prediction Eq. (5.49) can thereby assume an equal share of all triangular values for all ranks.

### Ranks and Threads

Fig. 5.7 shows strong and weak scaling for the time prediction Eq. (5.49). The number of parallel processes is applied for both $ranks$ and $threads$, where the Eq. (5.49) is scaled for both, by fixing the other value to 1. The number of memory operations is used as a scalar of predicted runtime, since the bandwidth is unknown. The upper and lower bounds from Fig. 5.5 are shown as dotted lines, to be used for comparison.

Fig. 5.7a shows strong scaling of Eq. (5.49) for a total of 20000 vertices. The estimate lies within the upper and lower bound, and has a similar function shape. The gap between Eq. (5.49) and the upper bound is caused by the large maximum of $T_{halo}$ and $T_{neumann}$, since the two plots uses the same $T_{local} = 2V$. The halo $T_{halo}$ of Mehamn harbor is significantly smaller than $T$, which causes the scaling for $ranks$ and $threads$ to be overlapping.

Fig. 5.7b shows weak scaling of Eq. (5.49) for 2000 vertices per process, which lies between the upper and lower bound. Since $T_{halo}$ is of a squared size from $T$, it cannot grow faster than $T$, avoiding it from causing a linear time growth as the upper bound. For the same reason, scaling for $ranks$ and $threads$ are overlapping.



**(a)** Predicted performance for strong scaling, with a total vertices of $V = 20000$.

**(b)** Predicted performance for weak scaling, with vertices of $V = 2000$ per process.

**Figure 5.7:** A performance estimate of triangulation of Mehamn harbor, for strong and weak scaling, from Eq. (5.49). Since the bandwidth is unknown, the number of memory operations is displayed, which is a scalar of the predicted runtime. The number of processes is the product of $ranks$ and $threads$, but the scaling for $ranks$ and $threads$ are separated, so each one is scaled while the other value is fixed to 1. The upper and lower limits from Fig. 5.5 are shown as dotted lines for comparison.

## Communication Time

The communication time from Eq. (5.36) can be applied to Mehamn harbor by using the estimate for $T_{halo}$, given as

$$t_{comm} = 2 \cdot 3 \cdot \left(\beta^{-1}(8B \cdot T_{halo}) + \alpha\right) = 6 \cdot \left(\beta^{-1}(8B \cdot \sqrt{V} \cdot 1.12) + \alpha\right) \qquad (5.50)$$

for bandwidth $\beta$, message size of $8B \cdot T_{halo}$, and communication latency $\alpha$. The communication time estimation is constant for any number of communicating endpoints. Since communication time is dependent on $T_{halo}$ and computation time is dependent on $T$, a constant communication time is assumed, as $T_{halo}$ is predicted to be significantly smaller than $T$ for Mehamn harbor.

Chapter 6

# Experimental Setup

This chapter explains the experimental setup of the software and hardware used, and the configurations of the benchmarking and application tests.

## 6.1 Software and Hardware

For the implementation of the proxy application, a set of software APIs and hardware supercomputers are selected as tools, which are used for parallel computing, preprocessing and performance analysis, and for running the application.

### 6.1.1 API

Table 6.1 lists the selected software and their version. MPI and OpenMP are combined for parallel programming, so MPI creates parallel ranks, and OpenMP allows each rank to spawn parallel threads [9, 10]. MPI also provides the function *MPI_Wtime()* to measure time for runtime testing.

Python 3 is used for preprocessing of data, predicting performance and for displaying results, by implementing libraries from Table 6.1 [30]. The Image library allows reading and modifying of an image of Mehamn harbor, before a triangulation through the Delaunay library is made [31, 32]. The Numpy library provides mathematical tools for various calculations such as making performance analysis, and the Pyplot library allows plots, so performance predictions and results can be displayed [33, 34].

### 6.1.2 The Idun Cluster

The Idun cluster is a research project provided by the NTNU's faculties and IT division, used for providing rapid testing and prototyping for HPC software [2]. Table 6.2 lists the properties used from the Idun cluster. The selected nodes are Dell PowerEdge, featuring Intel Xeon processors, with two CPUs of 12 cores each and 30 MB cache [35, 36]. They have Intel ICC compilers including MPI implementation for Linux [37, 38].

| API | Version |
|---|---|
| MPI | 3.1 |
| OpenMP | 4.5 |
| Python | 3.8.6 |
| Image from PIL | 8.0.1 |
| Delaunay from SciPy | 1.5.4 |
| Numpy | 1.19.4 |
| Pyplot from Matplotlib | 3.3.3 |

**Table 6.1:** List of API and its version used for the proxy application.

| Attribute | Detail |
|---|---|
| Node | Dell PowerEdge R730 |
| Processor | Intel Xeon E5-2650 v4 |
| Compiler | ICC 19.1.3.304 |
| Compile flags | -O2 -std=c99 -qopenmp -g -lm -liomp5 |
| MPI implementation | Intel® MPI Library for Linux* OS |
| CPU | 2 |
| Cores per CPU | 12 |
| Cache per CPU [MB] | 30 |

**Table 6.2:** List of specifications of a selected group of nodes from the Idun cluster.

## 6.2 Benchmarking

A set of benchmarking tests are used to analyse the hardware limitations of the chosen nodes from Table 6.2, before running the proxy application.

### 6.2.1 Roofline Model

A roofline model is a prediction of peak performance, shown in Fig. 6.1, which is decided by the maximum floating point performance and the maximum bandwidth, given the arithmetic intensity of the application. By running benchmarking tests, the peak bandwidth and $FLOP/s$ are estimated, in order to find the peak performance of the proxy application given the hardware from the Idun cluster.

#### Peak Floating Point Performance

To find the maximum floating point performance, a benchmark of simple multiplications of two matrices is used, as two matrices with size $n \times n$ gives $\mathcal{O}(n^3) FLOP$ and $\mathcal{O}(n^2)$ memory operations. For large $n$, the floating point operations become so significantly larger than memory operations that memory operations can be ignored when measuring runtime.

The test has repeated function calls to *DGEMM*, which is a level 3 function routine from the BLAS library, that calculates multiplication of two matrices $A$ and $B$ of size

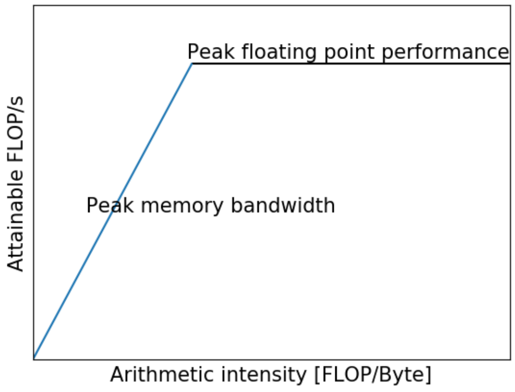

Peak floating point performance
Peak memory bandwidth
Attainable FLOP/s
Arithmetic intensity [FLOP/Byte]

**Figure 6.1:** Illustration of the roofline model, that shows peak performance given the arithmetic intensity of a program. This is decided by the maximum bandwidth and floating point performance, where bandwidth is the gradient of the slope line and maximum $FLOP/s$ is the horizontal line.

$M \times K$ and $K \times N$, resulting with matrix $C$ of size $M \times N$ [39]. The two constants $\alpha$ and $\beta$ are multiplied with the matrices, making

$$C \leftarrow \alpha AB + \beta C \tag{6.1}$$

This adds up to

$$MNK + MNK + 2MNFLOP = 2MN \cdot (K+1)FLOP \tag{6.2}$$

The benchmark sets the same matrix size $n$, giving $M = N = K = n$ and updating Eq. (6.2) to $2n^2(n+1)FLOP$. The runtime is measured for each call to the *DGEMM* function, and since memory operations are insignificant in runtime for large $n$, the performance is estimated by dividing the number of FLOP with the runtime. The highest performance of 500 function calls is selected as the peak performance. The matrix size $n = 7000$ and $n = 10000$ is used for testing, and then increased until the peak performance stops improving.

### Peak Memory Bandwidth

An estimate of the maximum bandwidth can be found by running benchmark of memory operations. OpenMP has a thread affinity policy that decides how the threads are assigned to the different cores of the hardware. The peak bandwidth estimate can be confirmed by comparing the results using affinity of *scatter* and *compact*, and see if they behave as expected when testing for an increasing number of threads.

The node selected from the Idun cluster has two CPUs of 12 cores each. Fig. 6.2 shows the order threads are allocated the cores. The affinity *scatter* distributes the threads evenly across the CPUs, while *compact* fills up one CPU at a time. While *compact* is only utilizing one CPU, it is expected to reach towards half the bandwidth of *scatter*, since it has half the resources. When all 24 cores are utilized, *scatter* and *compact* are expected to reach the same maximum bandwidth.

The benchmarking test measures the time to access the memory of a $100MB$ array of doubles, and uses the arithmetic mean of the runtime for 50 tests. This is repeated for both affinities, *scatter* and *compact*, and for all number of threads, from 1 to 24. The peak bandwidth is selected from the highest value of the tests.

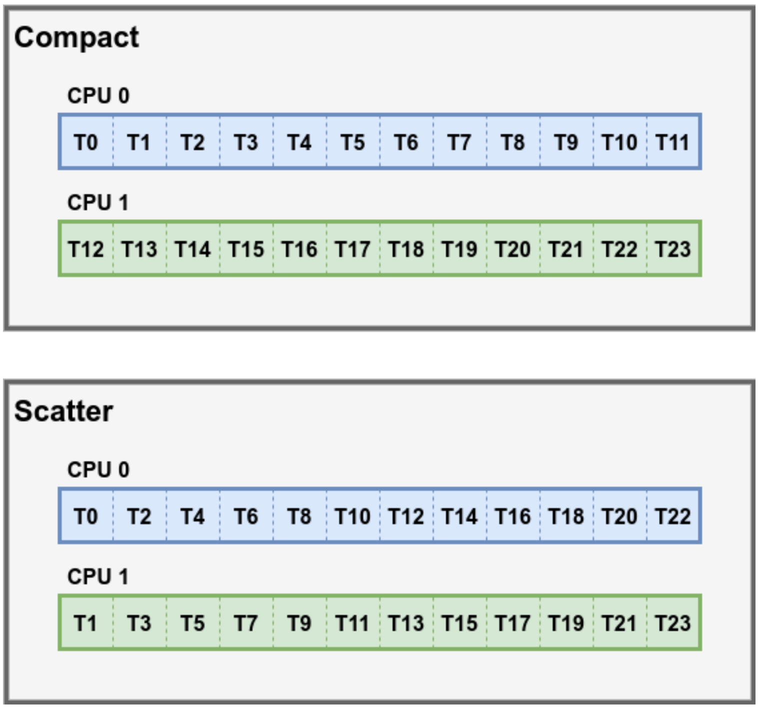


**Figure 6.2:** The order threads are distributed across the cores of the Idun cluster, with OpenMP thread affinity for *compact* and *scatter*. The *compact* affinity fills up one CPU at a time, and *scatter* distributes the threads evenly.

### 6.2.2 Communication

The benchmarking of communication is a test of MPI communication between ranks. Fig. 6.3 shows three different ways of MPI communication, where the communicating ranks can be on the same CPU, on different CPUs, but within the same node, and on two different nodes. The test aims to find the communication time for all three variants, by finding the latency $\alpha$ and MPI bandwidth $\beta$ for the Hockney model

$$t(n) = \alpha + \beta^{-1} n \tag{6.3}$$

for a message size $n$.

The latency $\alpha$ is found through a ping pong test, where two ranks send a message of $1B$ to the other repeatedly, so the time spent on latency is significantly longer than the time sending and receiving the message. The latency is estimated by dividing the runtime with the number of repetitions, which is set to $1000000$ for the benchmark.

The MPI bandwidth $\beta$ is estimated as the gradient of the communication time for an increasing message size. For each of the three communication variations from Fig. 6.3, the benchmarking test samples the communication time of a $100$ different arrays of sizes from zero to $16KB$. The message size of the application is not expected to exceed $12KB$, which is further discussed in Section 6.3.2. Each test is repeated 50 times, and the average time is used.

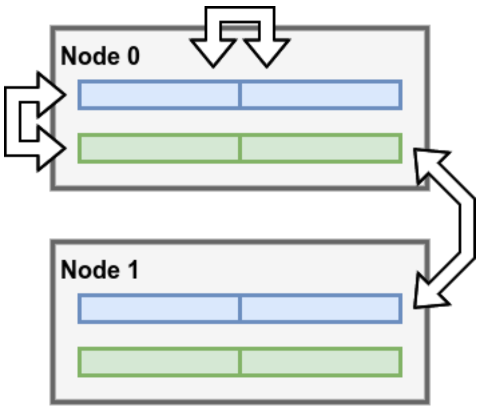


**Figure 6.3:** The three types of communication between ranks, which can be between two ranks on the same CPU, between two ranks on different CPU, but on the same node, and between two ranks on different nodes.

## 6.3 Application Prework

The proxy application uses Mehamn harbor for all its tests. Before running the proxy application, the configurations of the shallow water equations must be set. In addition, a structural analysis can be made, given the chosen sizes of the vertices $V$.

### 6.3.1 SWE Configuration

The application consists of a repeated processing of a triangulation. From the BSP model, the runtime is given as a sum of *supersteps*,

$$\sum_{s=1}^{S} t_s = \sum_{s=1}^{S} (w_s + gh_s + l) = \sum_{s=1}^{S} (t_{comp} + t_{comm} - t_{overlap}) \tag{6.4}$$

where $t_{comp}$ represents the time to compute the SWE. From Section 2.3.4, the discretization of SWE is

$$\rho\eta_i^{n+1} = \rho\eta_i^n \cdot w + \frac{\sum_{j\in Adj(i)} \rho\eta_j^n}{|Adj(i)|} \cdot (1-w) - \Delta t \cdot \sum_{j\in Adj(i)} F_{i,j}(\rho\eta) \tag{6.5}$$

$$\rho\eta u_i^{n+1} = \rho\eta u_i^n \cdot w + \frac{\sum_{j\in Adj(i)} \rho\eta u_j^n}{|Adj(i)|} \cdot (1-w) - \Delta t \cdot \sum_{j\in Adj(i)} F_{i,j}(\rho\eta u) \tag{6.6}$$

and

$$\rho\eta v_i^{n+1} = \rho\eta v_i^n \cdot w + \frac{\sum_{j\in Adj(i)} \rho\eta v_j^n}{|Adj(i)|} \cdot (1-w) - \Delta t \cdot \sum_{j\in Adj(i)} F_{i,j}(\rho\eta v) \tag{6.7}$$

for each triangle $i$, where $Adj(i)$ are all the neighbors of $i$. A time step $n$ from the formula is presented as a *superstep* from the BSP model.

The SWE configurations includes setting the attributes needed for the SWE formula, and setting the initial state for the first *superstep*.

### Attributes

Table 6.3 lists the number of *supersteps*, the number of time samples taken from the *supersteps*, and constants from Eq. (6.5), Eq. (6.6) and Eq. (6.7). There are fewer time samples than *supersteps* to give time for the hardware to warm up.

| **Attribute** | **Value** |
|---|---|
| *Supersteps* | 5000 |
| Time samples | 40 |
| $\Delta t$ | 0.05 |
| $g$ | 9.81 |
| $\rho$ | 997.0 |
| $w$ | 0.4 |

**Table 6.3:** List of constants for the discretization of the Shallow Water Equations from Eq. (6.5), Eq. (6.6) and Eq. (6.7). In addition, *supersteps*, the number of iterations of the formula, and time samples are listed.

### Initial State

The first *superstep* starts with a drop at the center of the triangulation, giving a lower mass than its surroundings, which will force wave propagation out of the center for the next time steps. The initial momentum is set as zero, making

$$\rho\eta u_i^0 = 0 \tag{6.8}$$

and

$$\rho\eta v_i^0 = 0 \tag{6.9}$$

The drop has a circular shape with a radius of

$$r = \frac{width + height}{2} \cdot \frac{1}{10} \tag{6.10}$$

where $width$ and $height$ are the maximum axes sizes of the triangulation. For each triangle $i$, the horizontal and vertical distance to the center is defined as

$$c(x_i) = x_i - \frac{width}{2} \tag{6.11}$$

and

$$c(y_i) = y_i - \frac{height}{2} \tag{6.12}$$

given the x-coordinate $x_i$ and y-coordinate $y_i$. An exponential function is used to make an initial drop shape, making the mass for each triangle $i$

$$\rho\eta_i^0 = \begin{cases} \frac{1}{1000}\left(1 - 1.5\exp\left(-\frac{4c(x_i)^2}{width} - \frac{4c(y_i)^2}{height}\right)\right), & \text{if } c(x_i)^2 + c(y_i)^2 < r^2 \\ \frac{\rho}{1000}, & \text{otherwise} \end{cases} \tag{6.13}$$

Fig. 6.4 shows the initial mass for the triangulation for 20000 vertices, based on Eq. (6.13).

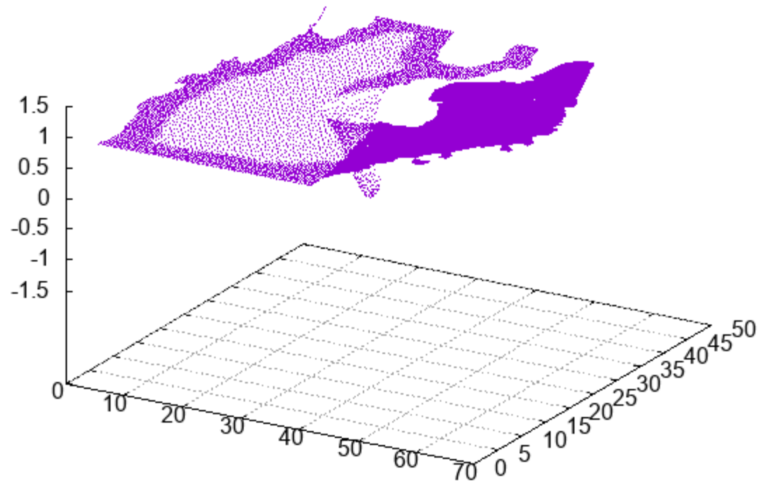


**Figure 6.4:** The initial state of the triangulation of Mehamn harbor of 20000 vertices. The height of each dot represents the mass $\rho\eta_i$ for triangle $i$ from Eq. (6.13). The momentum for both $\rho\eta u_i$ and $\rho\eta v_i$ for triangle $i$ are zero, from Eq. (6.8) and Eq. (6.9).

### 6.3.2 Triangulation

After a triangulation is made, given a number of vertices $V$, the triangular sizes $T$, $T_{halo}$ and $T_{neumann}$ are compared with their estimations. This further validates the performance models of application runtime and scaling, as they are dependent on the parameters of the triangular structure.

Table 6.4 lists a subset of the vertices $V$ used in the application test, between size $50000$ and the maximum $160000$, with the lower and upper bound of triangles $T$, *Halo triangles* $T_{halo}$ and *Neumann triangles* $T_{neumann}$. $T_{halo}$ can vary depending on the number of ranks, where the average of the maximum $T_{halo}$ for each configuration is used. The triangular bounds to be compared with are taken from Section 5.1.1 and Section 5.1.2.

| $V$ | $T$ | $T_{halo}$ | $T_{neumann}$ |
|---|---|---|---|
| | $[V-2, 2V-4]$ | $[0, V-2]$ | $[3, 4V-8]$ |
| $50000$ | $[49998, 99996]$ | $[0, 49998]$ | $[3, 199992]$ |
| $100000$ | $[99998, 199996]$ | $[0, 99998]$ | $[3, 399992]$ |
| $200000$ | $[199998, 399996]$ | $[0, 199998]$ | $[3, 799992]$ |
| $400000$ | $[399998, 799996]$ | $[0, 399998]$ | $[3, 1599992]$ |
| $800000$ | $[799998, 1599996]$ | $[0, 799998]$ | $[3, 3199992]$ |
| $1600000$ | $[1599998, 3199996]$ | $[0, 1599998]$ | $[3, 6399992]$ |

**Table 6.4:** List of initial vertices and the upper and lower triangular bounds from Section 5.1.1 and Section 5.1.2.

The gap between the lower and upper bound is too large to make an accurate validation of the performance models. Since only Mehamn harbor is used, an estimation of the triangular sizes is applied, which is found in Section 5.3.1. Table 6.5 lists the same vertices $V$ from Table 6.4 and the estimates of the triangular sizes $T$, $T_{halo}$ and $T_{neumann}$. The ranks communicate by exchanging *Halo triangles* $T_{halo}$. The maximum expected size for $T_{halo}$ of 1417 *Halo triangles* gives an overestimate of a maximum message size of $1500 \cdot 8 = 12KB$.

| $V$ | $T$ | $T_{halo}$ | $T_{neumann}$ |
|---|---|---|---|
| | $2V$ | $\sqrt{V} \cdot 1.12$ | $8.5\sqrt{V} \cdot 1.12$ |
| $50000$ | $100000$ | $250.44$ | $2128.74$ |
| $100000$ | $200000$ | $354.18$ | $3010.49$ |
| $200000$ | $400000$ | $500.88$ | $4257.47$ |
| $400000$ | $800000$ | $708.35$ | $6020.98$ |
| $800000$ | $1600000$ | $1001.76$ | $8514.95$ |
| $1600000$ | $3200000$ | $1416.70$ | $12041.95$ |

**Table 6.5:** List of initial vertices and the triangular estimations from Section 5.3.1.

## 6.4 Application Test

The proxy application is run for the triangulation of Mehamn harbor. The performance model is compared with the sampled runtime of the application, for different setups of scaling and distribution of hardware cores.

### 6.4.1 Runtime Test

A runtime test is made for all application runs, where the time $t_{comp}$ and $t_{comm}$ for different scaling and hardware setups are compared with the performance models.

From Section 5.3.2, the computation time $t_{comp}$ for processing Mehamn harbor is estimated as

$$t_{comp} = \left( \frac{210V}{ranks} + \frac{285.6\sqrt{V}}{ranks} + 49.28\sqrt{V} \right) \frac{8B}{bandwidth \cdot threads} \tag{6.14}$$

given the number of $ranks$ and $threads$, and the number of vertices $V$. The maximum bandwidth of the roofline benchmarking test, from Section 6.2.1, is used as an overestimated bandwidth for the prediction of $t_{comp}$.

The communication time $t_{comm}$ is estimated as

$$t_{comm} = 6 \cdot \left( \beta^{-1}(8B \cdot \sqrt{V} \cdot 1.12) + \alpha \right) \tag{6.15}$$

The proxy application measures both $t_{comp}$ and $t_{comm}$ through the MPI function *MPI_Wtime()*, which is sampled for each rank. The ranks send each time sample to the master rank, and use the average value. After all iterations of the *supersteps* are finished, the average of the time samples is found through the general formula

$$\bar{t} = \frac{\sum_{i=1}^{40} t_i}{40} \tag{6.16}$$

for the runtime $t$ and for 40 samples from Table 6.3. The standard deviation,

$$SD = \sqrt{\frac{\Sigma_{i=1}^{40}(t_i - \bar{t})^2}{39}} \tag{6.17}$$

is used to check that the measured runtime is stable.

#### Computation Partitions

The computation time $t_{comp}$ is divided into three parts $p_1$, $p_2$ and $p_3$, which are from Section 5.3.2 estimated as

$$p_1 = \left( \left( \frac{8.5\sqrt{V} \cdot 1.12}{ranks} \right) \cdot 8 + \left( \frac{8.5\sqrt{V} \cdot 1.12}{ranks} + 2 \cdot \sqrt{V} \cdot 1.12 + \frac{2V}{ranks} \right) \cdot 22 \right) \frac{8B}{bandwidth \cdot threads} \tag{6.18}$$

$$p_2 = \frac{2V}{ranks} 47 \cdot \frac{8B}{bandwidth \cdot threads} \tag{6.19}$$

and

$$p_3 = \frac{2V}{ranks} 36 \cdot \frac{8B}{bandwidth \cdot threads} \tag{6.20}$$

for the number of $ranks$ and $threads$ and the number of vertices $V$. Each part is measured to be compared with the estimation formula.

## 6.4.2 Comparison with Benchmark

The results from the application can be compared with the hardware benchmarking tests, by using the measured computation and communication time $t_{comp}$ and $t_{comm}$.

#### Roofline Model

The results from the application is compared with the roofline model from the benchmarking tests in Section 6.2.1, by comparing application performance of $FLOP/s$ over arithmetic intensity.

From Section 5.2.1, the number of floating point and memory operations are

$$\left(18 \cdot \frac{T_{neumann}}{ranks} + 36 \cdot T_{halo} + 137 \cdot \frac{T}{ranks}\right) FLOP \tag{6.21}$$

and

$$\left(240 \cdot \frac{T_{neumann}}{ranks} + 352 \cdot T_{halo} + 840 \cdot \frac{T}{ranks}\right) B \tag{6.22}$$

given the triangular sizes $T$, $T_{halo}$ and $T_{neumann}$. The arithmetic intensity of each application test is found by dividing the number of floating point operations with the number of memory operations, and the performance by dividing the number of floating point operations with the runtime $t_{comp}$.

The performance from the application test is plotted on the roofline model made from benchmarking, in order to compare peak performance from the roofline with the application performance, given the same arithmetic intensity. The configurations used for the comparison are taken from Section 6.4.4, where a single full node of 24 cores are utilized.

#### Communication

The benchmarking test for MPI communication, from Section 6.2.2, is compared with the communication time from the application test, by using the estimations of $\alpha$ and $\beta$ in Eq. (6.15) and comparing it with $t_{comm}$.

There are three types of $\alpha$ and $\beta$ from the benchmarking tests, which is communication between two cores on the same CPU, two cores on different CPUs and two cores on different nodes. The measured communication time for the application does not differentiate between the first two, and has only communication between two cores on the same node and two cores on different nodes. The measurements of communication time $t_{comm}$ are taken from Section 6.4.6, which has configurations with multiple nodes.

### 6.4.3 Load Balancing

Each rank samples the computation time $t_{comp}$, which is compared to the other ranks, in order to show the time distribution and determine the load balance of the ranks. The ranks are run in serial, by only spawning a single thread each. OpenMP uses a scheduler to balance out the threads spawned by each rank, so an even load balance for threads can be assumed. The ranks used are

$$ranks \in [1, 2, 4, 8, 16, 24] \tag{6.23}$$

### 6.4.4 Ranks and Threads

The number ranks and threads can be configured arbitrarily. In order to compare the effects of having different configurations of ranks and threads, a single full node with full utilization of all 24 cores are used, such as in Fig. 6.5, which has a full utilized node of four ranks with six threads per rank.

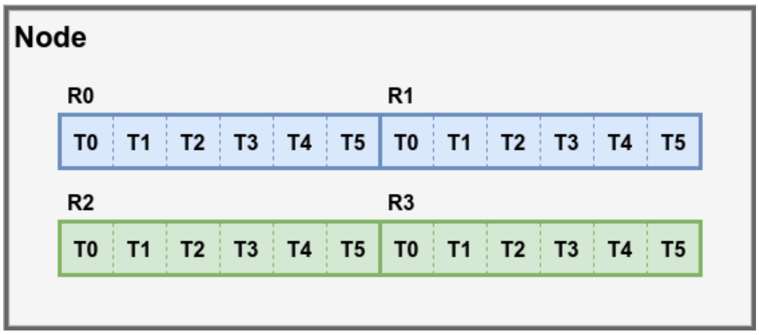

Node
R0
R1
T0 T1 T2 T3 T4 T5 T0 T1 T2 T3 T4 T5
R2
R3
T0 T1 T2 T3 T4 T5 T0 T1 T2 T3 T4 T5

**Figure 6.5:** The distribution of four ranks with six threads each rank, for a node from the Idun cluster. Each node has two CPUs of 12 cores. A rank can be assigned to multiple cores and will spawn threads to cover them. All the CPUs do not necessarily have to be covered, as the number of cores for each rank is decided through the configuration.

The computation time $t_{comp}$ is used to compare the variations of ranks and threads, and is also compared with the estimation Eq. (6.14). Table 6.6 lists the setup for the application test, which tests all variations of ranks and threads for a fully utilized node, and for the same vertices $V$.

### 6.4.5 Scaling of Computation Time

For the scaling tests of the application, the computation time $t_{comp}$ is compared with the computation time estimation Eq. (6.14) for strong and weak scaling. The speedup and efficiency are also measured. Both cases have a scaling of cores, which is the product of ranks and threads. Ranks and threads are separated for different tests, where one is fixed to 1 while the other is scaled. The number of vertices $V$ represents the problem size, since the number of triangles $T$ is bounded by $V$.

| Cores | Ranks | Threads | Vertices | | |
|---|---|---|---|---|---|
| 24 | 24 | 1 | 50000 | 100000 | 200000 |
| 24 | 12 | 2 | 50000 | 100000 | 200000 |
| 24 | 8 | 3 | 50000 | 100000 | 200000 |
| 24 | 6 | 4 | 50000 | 100000 | 200000 |
| 24 | 4 | 6 | 50000 | 100000 | 200000 |
| 24 | 3 | 8 | 50000 | 100000 | 200000 |
| 24 | 2 | 12 | 50000 | 100000 | 200000 |
| 24 | 1 | 24 | 50000 | 100000 | 200000 |

**Table 6.6:** List of test configurations for a full utilization of a node from Idun. All 24 cores are covered for all combinations of ranks and threads, and are tested for all initial vertices.

The node selected from the Idun cluster has two CPUs of 12 cores each, where the scaling tests are run from 1 to all 24 cores. The cores allocated are similar to the OpenMP *compact* affinity from Fig. 6.2, where the cores close to one other and on the same CPU are allocated first.

### Strong Scaling

Amdahl's law of strong scaling uses a constant problem size, meaning $V$ must be constant for an increased number of $N$ processing units, so each core gets a smaller number of triangles to process. The speedup is

$$speedup_N = \frac{t_1(V)}{t_N(V)} \tag{6.24}$$

where $t_N(V)$ is the computation time $t_{comp}$ given $V$ as the input size and $t_1(V)$ is the computation time for a serial program. The efficiency is

$$efficiency_N = \frac{speedup_N}{N} \tag{6.25}$$

Table 6.7 lists the configurations for ranks and threads, so both are scaled for separate tests, and the number of vertices $V$ for each configuration.

### Weak Scaling

Gustafson's law of weak scaling uses a scaled problem size as the number of $N$ processing units increases. $V$ must be scaled so that each core has the same number of triangles to process. The efficiency is

$$efficiency_N = \frac{t_1(V)}{t_N(V \cdot N)} \tag{6.26}$$

where $t_N(V)$ is the computation time $t_{comp}$ given $V$ as the input size and $t_1(V)$ is the computation time for a serial program. The speedup is

$$speedup_N = efficiency_N \cdot N \tag{6.27}$$

| Cores | Ranks | Threads | Ranks | Threads | Vertices | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 50000 | 100000 | 200000 |
| 2 | 2 | 1 | 1 | 2 | 50000 | 100000 | 200000 |
| 4 | 4 | 1 | 1 | 4 | 50000 | 100000 | 200000 |
| 8 | 8 | 1 | 1 | 8 | 50000 | 100000 | 200000 |
| 16 | 16 | 1 | 1 | 16 | 50000 | 100000 | 200000 |
| 24 | 24 | 1 | 1 | 24 | 50000 | 100000 | 200000 |

**Table 6.7:** List of configurations for strong scaling for both ranks and threads separately. While one value is scaled, the other is fixed to one, so the number of cores is scaled the same way. All vertices are tested for both scalings.

Table 6.8 lists the configurations of ranks and threads, where each is scaled for a separate test. The number of vertices are scaled so each core has a constant number of triangles.

| Cores | Ranks | Threads | Ranks | Threads | Vertices | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 3125 | 6250 | 12500 |
| 2 | 2 | 1 | 1 | 2 | 6250 | 12500 | 25000 |
| 4 | 4 | 1 | 1 | 4 | 12500 | 25000 | 50000 |
| 8 | 8 | 1 | 1 | 8 | 25000 | 50000 | 100000 |
| 16 | 16 | 1 | 1 | 16 | 50000 | 100000 | 200000 |
| 24 | 24 | 1 | 1 | 24 | 75000 | 150000 | 300000 |

**Table 6.8:** List of configurations for weak scaling for both ranks and threads separately. While one value is scaled, the other is fixed to one, so the number of cores is scaled the same way. The vertices are scaled so each core has a constant number of vertices.

### 6.4.6 Node Scaling

The application is also scaled for up to eight nodes, where each node has a full utilization of all 24 cores, shown in Fig. 6.6. Fig 6.6 has a setup of two nodes, with six ranks for each node and four threads for each rank. The computation time $t_{comp}$, the speedup and the efficiency are compared for strong and weak scaling.

The tests for node scaling have two configurations of ranks and threads, with six ranks per node and four threads per rank, and with eight ranks per node and three threads per rank.

#### Strong Scaling

Table 6.9 lists the configurations for strong scaling of nodes, where both variations of ranks and threads are tested for all number of vertices. The speedup is

$$\mathit{speedup}_N = \frac{t_1(V)}{t_N(V)} \tag{6.28}$$

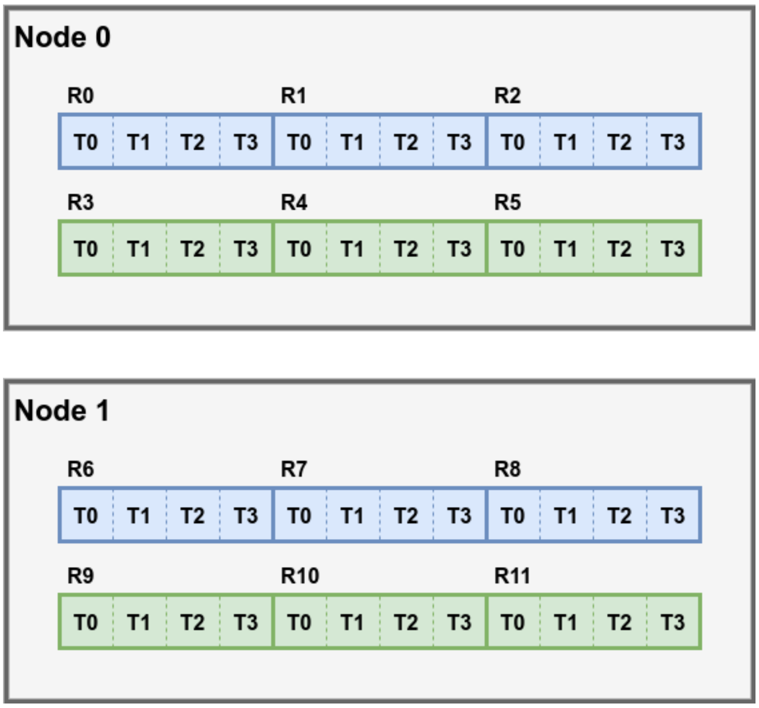

Node 0
R0
R1
R2
T0 T1 T2 T3 T0 T1 T2 T3 T0 T1 T2 T3
R3
R4
R5
T0 T1 T2 T3 T0 T1 T2 T3 T0 T1 T2 T3
Node 1
R6
R7
R8
T0 T1 T2 T3 T0 T1 T2 T3 T0 T1 T2 T3
R9
R10
R11
T0 T1 T2 T3 T0 T1 T2 T3 T0 T1 T2 T3

**Figure 6.6:** The distribution of six ranks per node and four threads per rank, for two nodes from the Idun cluster. Each node has two CPUs of 12 cores, adding up to 48 cores. Rank 5 and 6 must communication across the nodes, while the other ranks only communicate within the same node.

where $N$ is the number of nodes. The value $t_N(V)$ is the computation time $t_{comp}$ given $V$ as the input size and $t_1(V)$ is the computation time for a single node. This gives an efficiency of

$$\textit{efficiency}_N = \frac{\textit{speedup}_N}{N} \tag{6.29}$$

| Nodes | Cores | Ranks | Threads | Ranks | Threads | Vertices | |
|---|---|---|---|---|---|---|---|
| 1 | 24 | 6 | 4 | 8 | 3 | 100000 | 200000 |
| 2 | 24 | 6 | 4 | 8 | 3 | 100000 | 200000 |
| 4 | 24 | 6 | 4 | 8 | 3 | 100000 | 200000 |
| 8 | 24 | 6 | 4 | 8 | 3 | 100000 | 200000 |

**Table 6.9:** List of configurations for strong node scaling, where each node is fully utilized. The cores are filled up with six ranks and four threads, and eight ranks and three threads, adding up to 24 cores. All initial vertices are tested for both variants of ranks and threads.

### Weak Scaling

Table 6.10 lists the configurations for weak scaling of nodes, with both variations of ranks and threads. The number of vertices is scaled so the number of triangles is constant for each node. This gives an efficiency of

$$\textit{efficiency}_N = \frac{t_1(V)}{t_N(V \cdot N)} \tag{6.30}$$

where $N$ is the number of nodes. The value $t_N(V)$ is the computation time $t_{comp}$ given $V$ as the input size and $t_1(V)$ is the computation time for a single node. The speedup is

$$\textit{speedup}_N = \textit{efficiency}_N \cdot N \tag{6.31}$$

| Nodes | Cores | Ranks | Threads | Ranks | Threads | Vertices | |
|---|---|---|---|---|---|---|---|
| 1 | 24 | 6 | 4 | 8 | 3 | 100000 | 200000 |
| 2 | 24 | 6 | 4 | 8 | 3 | 200000 | 400000 |
| 4 | 24 | 6 | 4 | 8 | 3 | 400000 | 800000 |
| 8 | 24 | 6 | 4 | 8 | 3 | 800000 | 1600000 |

**Table 6.10:** List of configurations for weak node scaling, where each node is fully utilized. The cores are filled up with six ranks and four threads, and eight ranks and three threads, adding up to 24 cores. The initial vertices are scaled for both variants of ranks and threads, so each node has a constant number of vertices.

Chapter 7

# Results and Discussion

This chapter takes the results from running the proxy application for a triangulation of Mehamn harbor, with the configurations of Section 6.3.1, and makes a comparison with the performance predictions.

First, the prediction of the parameters of the triangular structure are compared with the generated triangulation for all application tests, which are used to validate the performance models, as they are based on the triangular parameters. Then, the roofline and Hockney model, based on the results from the benchmarking tests, are compared with the application runtime. The benchmarking results are used to validate assumptions made for the performance models, including a memory bound computation time, and a constant communication time.

The analysis separates computation time and communication time, which are compared with the performance models. Since a constant communication time is assumed in Section 5.3.2, the performance tests for scaling are focused on computation time. This includes results for load balancing, comparing the distribution of threads and ranks, and comparing the computation time with its prediction, with both strong and weak scaling of ranks and threads. A scaling of nodes from the Idun cluster is also provided, where each node has a full utilization of all cores.

## 7.1 Triangulation

The triangulation of Mehamn harbor, from the configurations of Section 6.3.2, is compared with the predictions from 5.1 and 5.3.1, for the triangles $T$, *Halo triangles* $T_{halo}$ and *Neumann triangles* $T_{neumann}$, given the number of vertices $V$.

### 7.1.1 Total Triangles

Fig. 7.1 shows the number of generated triangles $T$ for vertices $V$ between the size 50000 and 1600000. The estimation of $T$ and the upper and lower bounds are

$$T \approx 2V \in [V-2, 2V-4] \tag{7.1}$$

which are shown in the figure.

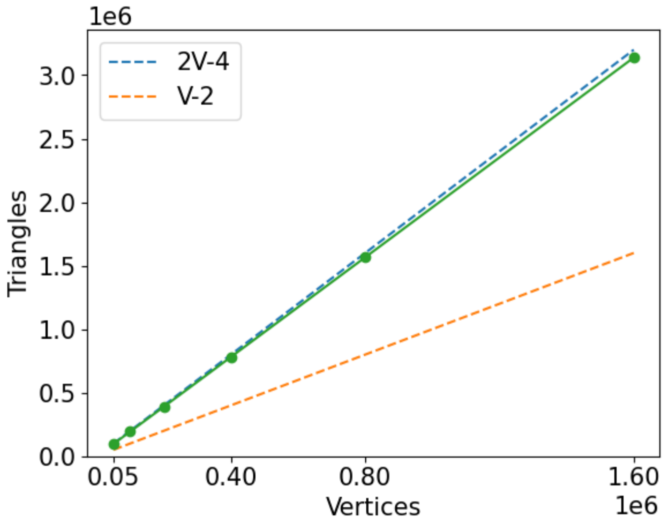


**Figure 7.1:** The number of generated triangles $T$ from the triangulation of Mehamn harbor, given the number of vertices $V$. The dotted lines shows the upper and lower limits of $T$, where $T$ is estimated to follow the upper limit.

The number of triangles $T$ increases almost in the same rate as the predicted $T$, but can be seen to be slightly smaller for large $T$. This can be attributed to the geometry of Mehamn harbor. The vertices are spread out in grid patterns, making room for two triangles in each grid. A rectangular surface would give $2V$ triangles for large $V$, but since Mehamn harbor has an irregular shaped border surrounding the surface, the grid pattern is cut off, which can decrease the total number of triangles.

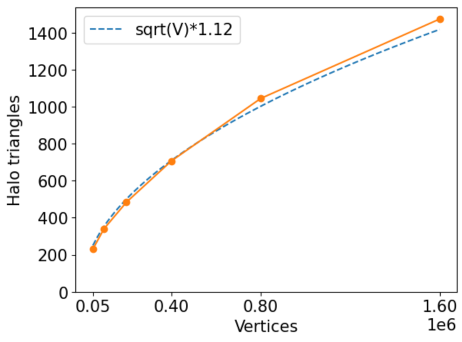
### 7.1.2 Halo and Neumann Triangles

Fig. 7.2a and Fig. 7.2b show the number of generated *Halo triangles* and *Neumann triangles*, for the number of vertices $V$. The estimation and the upper and lower bounds of $T_{halo}$ and $T_{neumann}$ are

$$T_{halo} \approx \sqrt{V} \cdot 1.12 \in [0, V-2] \tag{7.2}$$

and

$$T_{neumann} \approx 8.5\sqrt{V} \cdot 1.12 \in [3, 4V-8] \tag{7.3}$$



**(a)** $T_{halo}$ compared to estimation $\sqrt{V}1.12$. **(b)** $T_{neumann}$ compared to estimation $8.5\sqrt{V}1.12$.

**Figure 7.2:** The number of generated *Halo triangles* $T_{halo}$ and *Neumann triangles* $T_{neumann}$ of Mehamn harbor, given the number of vertices $V$. The dotted lines are the predictions of $T_{halo}$ and $T_{neumann}$.

where only the estimations are shown in the figure.

$T_{halo}$ follows the prediction, but is starting to outgrow the prediction at 1600000 vertices. This is attributed to the geometry of Mehamn harbor, which is used to find the prediction formula. The prediction is based on a uniform distribution of vertices, where irregularities are expected from the map of Mehamn, due to it having four sections with different densitites of vertices.

$T_{neumann}$ follows the prediction up to 800000 vertices, and is outgrowing the prediction for larger $V$. This is also attributed to the geometry of Mehamn harbor, used to find the prediction formula. $T_{neumann}$ is predicted to grow in the same rate as $8.5 \cdot T_{halo}$, where the constant 8.5 is taken from a triangulation of 20000 vertices. A triangulation of 20000 vertices will take shortcuts and ignore small irregular shapes along the border, because of the low resolution. As $V$ increases, details such as small slopes are included, making $T_{neumann}$ grow slightly faster than $T_{halo}$, which is represented as a straight horizontal line of the triangulation.

The curvature of the measured $T_{halo}$ and $T_{neumann}$ indicate that they will stay below their upper bounds of $V - 2$ and $4V - 8$, as the number of vertices increases indefinitely, and continue to grow significantly slower than the number of triangles $T$. This is attributed to the shape of Mehamn harbor, which has a large and dense two dimensional area, while $T_{halo}$ and $T_{neumann}$ are based on one dimensional borders.

### 7.1.3 Validation of Performance Formula

For the vertices up to size 1600000, which includes all vertices sizes of the application tests, $T$, $T_{halo}$ and $T_{neumann}$ are confirmed to follow their prediction sizes, meaning they can all be predicted based only on the number of vertices $V$. This confirms the prediction formulas of the application tests from Section 6.4.1, which already use the predicted values of $T$, $T_{halo}$ and $T_{neumann}$.

For other triangulations, it may not be possible to accurately predict the triangular sizes, other than knowing the upper and lower bounds. This can make performance predictions inaccurate, if they end up having a wide range between the upper and lower bound of the performance formulas. In this case, it can be useful to know if $T$ is significantly bigger than $T_{halo}$ and $T_{neumann}$, which is the case for triangulations of large areas, to determine whether the effects from $T_{halo}$ and $T_{neumann}$ can be ignored for large $V$. $T$ can also be estimated as $2V$, if the triangulation has a large and dense area.

## 7.2 Comparison with Benchmarking

The benchmarking tests from Section 6.2, for making a roofline model of peak performance and for making a model for MPI communication, are compared with the application performance and application communication from Section 6.4.2.

### 7.2.1 Roofline Model

The roofline model gives an estimation of the maximum possible performance, given the arithmetic intensity of a program, and can be used to confirm if a program is compute or memory bound.

#### Benchmarking

The estimation of peak floating point performance is based on matrix multiplication tests, and gave a maximum performance of $357.785 GFLOP/s$, for matrices of $n \times n$ where $n = 7000$. Since the application is memory bound, it is outside of this scope to test if rectangular matrices give a higher performance.

Fig. 7.3 shows the measured bandwidth from the benchmarking tests, where the peak memory bandwidth is $42.677 GB/s$. The figure displays the bandwidth for both OpenMP thread affinity *compact* and *scatter* for an increasing number of threads. An analysis of their bandwidths pattern are used to assure that the peak memory bandwidth was found. *Compact* fills up the cores of one CPU at a time, and for the first 12 threads, it reaches the maximum bandwidth of a single CPU with the value $20 GB/s$, which is half of the maximum bandwidth. For the threads added after, the bandwidth increases linearly, as the second CPU gets increasingly utilized. *Scatter* distributes the threads evenly for the two CPUs, and reaches the peak memory bandwidth faster than *compact*. An even number of threads with *scatter* would utilize the CPUs more than an odd number, which is shown as an alternating pattern on the figure.

Fig. 7.4a shows the roofline models based on the estimated peak floating point performance and on the estimated peak memory bandwidth. Fig. 7.4b shows the same model with the upper and lower limits of the arithmetic intensity of the application. With an upper limit of $0.1631$, the application is memory bound, meaning the application is expected to spend a significantly longer time for memory operations than for computations.

#### Runtime Test

Fig. 7.5 shows the estimated roofline model from Fig. 7.4b with the performance results from the application, where all 24 cores are utilized for a single node. The application per-

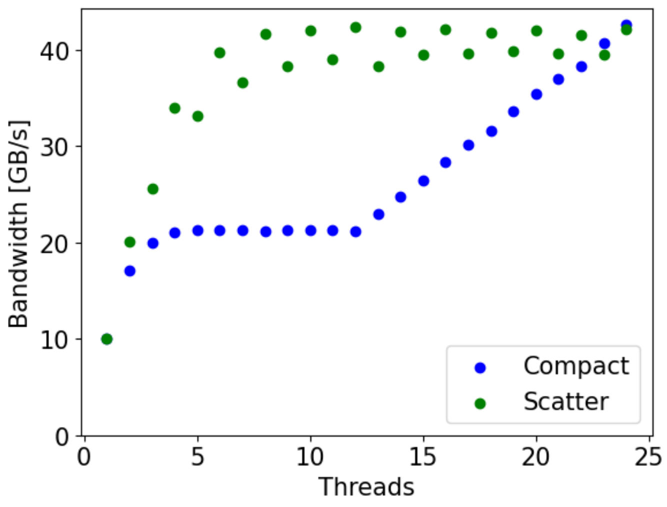
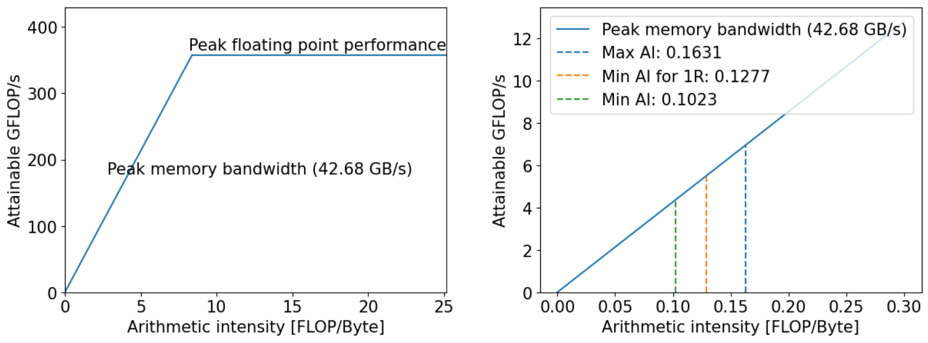


**Figure 7.3:** Measured bandwidth from benchmarking test, for increasing number of threads, with OpenMP thread affinity *compact* and *scatter*.



**(a)** The estimated roofline model from the benchmarking tests.

**(b)** The roofline model with the boundaries of the arithmetic intensities for the application.

**Figure 7.4:** The roofline model made from benchmarking tests on the Idun cluster, with the maximum and minimum boundaries of the arithmetic intensity of the proxy application. The minimum arithmetic intensity, given a single rank, is also included.

formance is confirmed to be memory bound, since the performance is below the rooftop based on the peak memory bandwidth and has a low arithmetic intensity. The arithmetic intensity of the application is approximately the value of the upper bound. This is attributed to the geometry of Mehamn harbor, which has a significantly larger number of triangles $T$ than $T_{halo}$ and $T_{neumann}$. From Section 5.2.1, the upper limit of arithmetic intensity is caused by having a significantly large $T$, and the lower limit of arithmetic intensity is caused by having a significantly large $T_{halo}$ or ranks. The number of ranks is limited by the number of hardware cores, and cannot outgrow the number of triangles.

The performance formula from Section 5.2.1 is based on the assumption that the application is memory bound, which is validated by having that assumption confirmed through the analysis of the roofline model. The maximum bandwidth from the roofline model is also used as an estimate of the bandwidth in the performance formula, which is expected to be overestimated, since the application performance is lower than the rooftop performance.

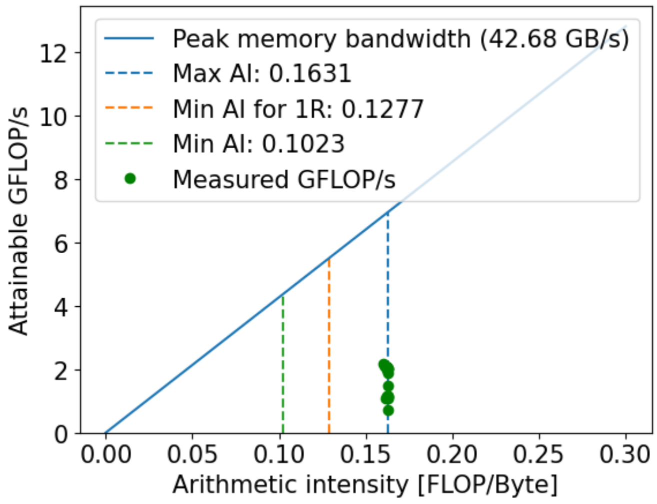


**Figure 7.5:** The roofline model with the boundaries of arithmetic intensity from Fig. 7.4b, and with the application performance for all tests utilizing all 24 cores on a single node of the Idun cluster.

## 7.2.2 Communication

The Hockney model gives an estimation of the communication time between two processing units given a message size, and is used for communication benchmarking.

### Benchmarking

The communication formula of the Hockney model is

$$t(n) = \alpha + \beta^{-1}n \tag{7.4}$$

where $t(n)$ is the communication time given the message size $n$ and the constants $\alpha$ and $\beta$. There are three types of communications explained in Section 6.2.2, for cores on the same CPU, different CPU and on different nodes, which gives three versions of $\alpha$ and $\beta$.

Fig. 7.6 shows the communication time for messages up to $15000B$, which covers the maximum size of the *Halo triangles* $T_{halo}$ sent for all application tests. The three different types are all linear, with the exception of communication within the same node for small messages, which is attributed to latency due to packaging protocols. The communication time between nodes are longer than the other two, which is attributed to software cost of making system calls to the network rather than using the memory bus, in addition to having a longer physical communication distance.

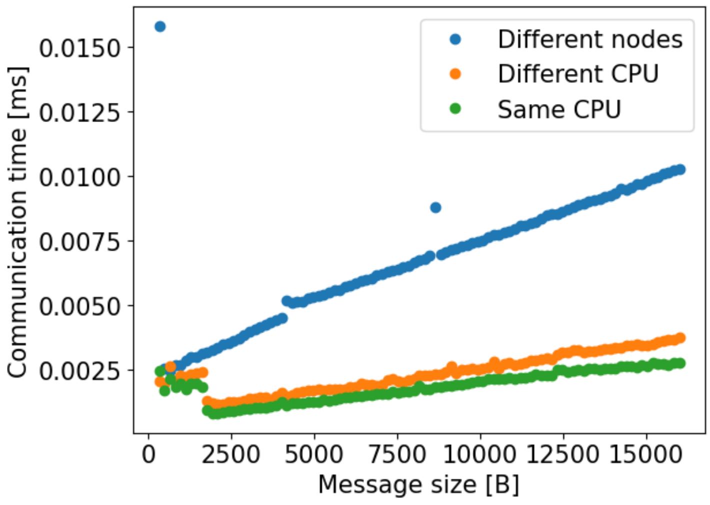


**Figure 7.6:** The MPI communication time from the benchmarking tests of the three types, which are communication between cores on the same CPU, on different CPU and on different nodes.

Table 7.1 lists the estimated $\alpha$ and $\beta$ for the three types of communication. The constants of $\alpha$ are measured from ping pong tests described in Section 6.2.2, and the constants of $\beta$ are taken from the gradients of each result in Fig. 7.6. The constants of $\alpha$ are small compared to $\beta$, which indicates a small communication latency. The $\alpha$ constants have the lowest value for communication within the same CPU, and the highest value for communication between nodes, while the $\beta$ constants have the highest value for communication within the CPU, and the lowest value for communication between nodes. The communication between cores on the same CPU and different CPU have more similar constants than communication between nodes. This is attributed to the physical network and distance, where two nodes are expected to have a greater distance and slower network than any communication link within the same node.

The application tests do not differentiate between communication of different and same CPU within the same node, as it is assumed they have a close communication time compared to communication between two nodes. The results from Table 7.1 can be used to confirm this assumption.

| Constant | Different nodes | Different CPUs | Same CPU |
|---|---|---|---|
| $\alpha$ | $2.829573\mu s$ | $1.162463\mu s$ | $0.728319\mu s$ |
| $\beta$ | $2.013228GB/s$ | $5.657298GB/s$ | $7.218045GB/s$ |

**Table 7.1:** The measured constants $\alpha$ and $\beta$ of the extended Hockney model. There are three types of communications, which are for cores on the same CPU, cores on different CPU, and cores on different nodes.

### Runtime Test

The application tests for MPI communication are taken from the configurations for node scaling from Section 6.4.6. Each node has utilization of all 24 cores, with two configurations of ranks and threads, of six ranks and four threads, and of eight ranks and three threads. There are two types of communication, which are between two cores on the same node, and two cores on different nodes.

Fig. 7.7 shows the average measured communication time for the application, over an increasing number of nodes. The total number of vertices is 200000, which makes approximately 500 *Halo triangles* to be sent for each message. The communication time between nodes is more varied compared to communication time within the same node. This is attributed to each node having different communication properties, as they are different physical machines. This makes the communication time dependent on which node from the cluster is chosen. The number of nodes does not seem to affect communication time. This can be applied to the communication prediction, where the same communication time is assumed for any number of nodes.

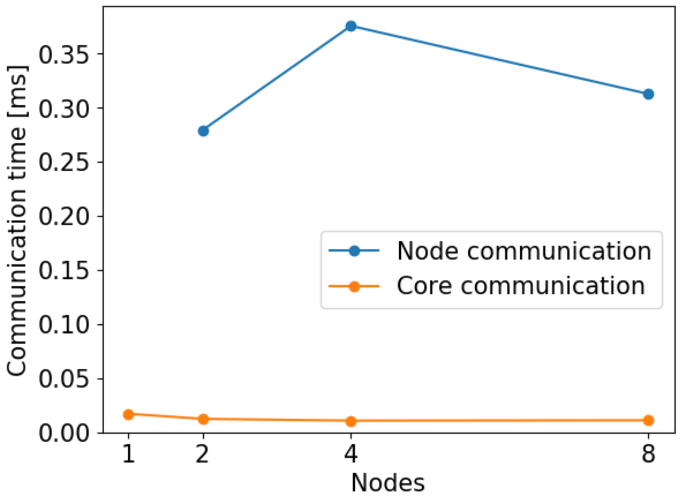


**Figure 7.7:** The average measured MPI communication time for the application tests with a total of 200000 vertices, for an increasing number of nodes. There are two types of communication, with messages sent between nodes and sent between cores within the same node.

The constants $\alpha$ and $\beta$ from Table 7.1 are used to formulate a prediction of communication time

$$t_{comm} = 2 \cdot 3 \cdot \left(\beta^{-1}(8B \cdot T_{halo}) + \alpha\right) \tag{7.5}$$

which is described in Section 5.2.3. Fig. 7.8a and Fig. 7.8b show a comparison of the predicted and measured communication time over the number of *Halo triangles* $T_{halo}$. The results for node communication are split into three groups and plotted separately, because of the assumption that communication is dependent on the specific node selected. The figure only shows the lowest group of node communication, as there is not enough data from the other two to be used in comparison to the prediction. The arithmetic mean of the time with the standard deviation for each group is listed in Table 7.2, which confirms the split of the different communication times.

| Group | Average communication time |
|---|---|
| Lowest | $0.286908 \pm 0.023409$ |
| Middle | $0.582280 \pm 0.073029$ |
| Highest | $1.025291 \pm 0.010927$ |

**Table 7.2:** The average communication time between nodes split into three different groups.

The prediction from Fig. 7.8a shows a similar communication time for cores within the same node, to the core communication from Fig. 7.8b, in addition to having an increasing communication time with the number of *Halo triangles*. The node communication from the lowest group is also increasing slightly, following a linear pattern fitting for a Hockney model, but is not corresponding to the prediction of node communication. This is attributed to latency due to the application having a heavy workload of computations between each communication, which take up significantly more local memory than *Halo triangles*, causing cache pollution.

### 7.2.3 Benchmarking and Performance Models

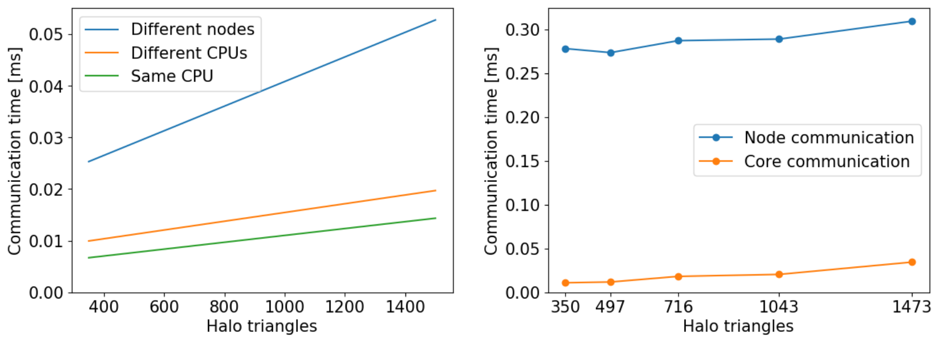
Section 5.2.1 and Section 5.2.3 give the estimation of computation time $t_{comp}$ and communication time $t_{comm}$

$$t_{comp} = \left(240 \cdot \frac{T_{neumann}}{ranks} + 352 \cdot T_{halo} + 840 \cdot \frac{T}{ranks}\right) \frac{B}{bandwidth \cdot threads} \tag{7.6}$$

and

$$t_{comm} = 2 \cdot 3 \cdot \left(\beta^{-1}(8B \cdot T_{halo}) + \alpha\right) \tag{7.7}$$

The benchmarking results show that the roofline model fits better for the prediction of $t_{comp}$, than the Hockney model fits for the prediction of $t_{comm}$. The comparison of the roofline model and the application measurements confirms that the application is memory bound, which is further assumed for the performance models. The performance models can in addition use the maximum memory bandwidth to make an overestimate of the predicted computation time, and use that to predict scaling characteristics. The Hockney model confirmed a constant communication time for a different number of processing units. The communication time of the application is expected to grow significantly slower



**(a)** Predicted communication time from Eq. (7.5). **(b)** Measured communication time.

**Figure 7.8:** Comparison of predicted and measured MPI communication, for the number of *Halo triangles* $T_{halo}$ being sent. The prediction has three types of communication, for cores on different nodes, different CPUs and same CPU, while the application uses two types, for cores on different and same nodes. The nodes with different communication time are separated into three groups, where only the fastest group is displayed.

than the computation time of the application, which is due to the communication time being dependent on the *Halo triangles*, while the computation time is dependent on all the triangles. The overall communication time can thereby be confirmed to stay constant, and the performance models can be focused on the computation time.

## 7.3 Scaling of Computation Time

The analysis of the application's scaling characteristics is mainly focused on the computation of the triangulation of Mehamn harbor. This includes both scaling of resources and problem size, which is made by adding processor cores and nodes from the Idun cluster, and by increasing the number of vertices $V$. The number of ranks and threads in the application scales the parallel resources, explained in Section 5.2. The predictions from Section 5.3.2 are compared with the measurements from Section 6.4.3, Section 6.4.4, Section 6.4.5 and Section 6.4.6.

The results from Section 7.1 are used to confirm the predictions of the triangular sizes $T$, $T_{halo}$ and $T_{neumann}$ for Mehamn harbor, which are applied to the performance predictions of the computation time $t_{comp}$, given as

$$t_{comp} = \left(\frac{210V}{ranks} + \frac{285.6\sqrt{V}}{ranks} + 49.28\sqrt{V}\right)\frac{8B}{bandwidth \cdot threads} \tag{7.8}$$

The results from the roofline model from Section 7.2 are used to confirm that $t_{comp}$ can be estimated through counting memory operations, in addition to giving a maximum bandwidth to the $t_{comp}$ formula, so Eq. (7.8) can used for further scaling predictions.

Since the scaling analysis is focused on computation time, the prediction of the communication time

$$t_{comm} = 6 \cdot \left(\beta^{-1}(8B \cdot \sqrt{V} \cdot 1.12) + \alpha\right) \tag{7.9}$$

is not used for scaling prediction. The results from Section 7.2 show that communication time is unaffected by the number of processing units, meaning it has little effects from a scaling of nodes. The communication time is also dependent on the *Halo triangles* $T_{halo} = \sqrt{V} \cdot 1.12$, which makes it grow significantly slower than $t_{comp}$, dependent on the total triangles $T = 2V$. This makes communication time constant for the problem size $V$, as computation time has a significantly longer runtime. For triangulations of other maps, such as an irregular triangulation of a one dimensional horizontal line, the communication time can be scalable, since it would give approximately the same number of $T_{halo}$ and $T$.

### 7.3.1 Load Balancing

For a given number of ranks, the overall computation time $t_{comp}$ equals the computation time of the slowest rank, leaving the other ranks to wait. This overhead grows as the problem size and the number of processing units increases, which makes load balance affect the scaling abilities.

Fig. 7.9 shows the computation time, measured in proportions for each rank. Each rank makes the computations in serial, since they only have a single thread each. For up to all 24 numbers of ranks, the workload is balanced, which adds little overhead for application scaling.

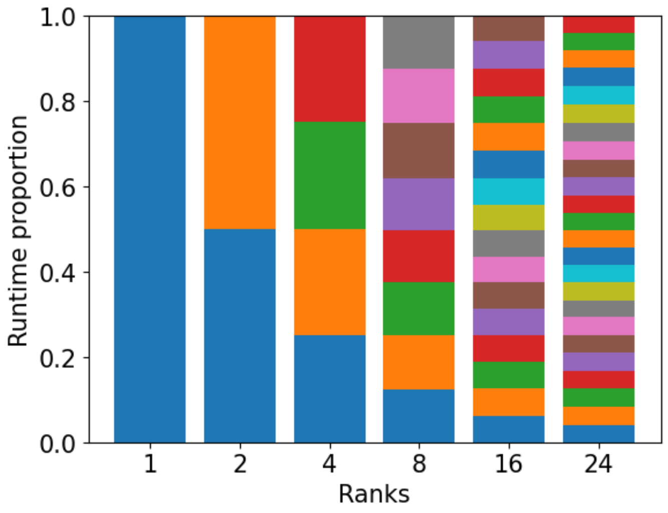


**Figure 7.9:** The computation time for each rank, measured in proportions of one other, where each rank has one thread.

### 7.3.2 Ranks and Threads

The prediction formula from Eq. (7.8) estimates the same runtime for both scaling of ranks and threads, where the extra runtime for ranks can be ignored, since it is only bigger by a factor of $\sqrt{V}$. The prediction $t_{comp}$ is $\mathcal{O}(V)$, meaning it is expected to scale linearly for the problem size $V$.

Fig. 7.10a and Fig. 7.10b show a comparison of predicted and measured computation time for different ranks and threads, where all 24 cores are utilized. The measured runtime is about three time as high as the predicted runtime for 50000 and 100000 vertices, and over six times as high for 200000 vertices. The time difference is attributed to the bandwidth used in the prediction formula, which is the highest memory bandwidth from the roofline benchmarking tests, making the prediction the highest attainable performance.

The predicted runtime shows a linear scaling for all vertices $V$, whereas the measured runtime only has linear scaling for 50000 and 100000 vertices. This is attributed to the local memory capacity. The selected nodes from the Idun cluster has a local memory of $60MB$, described in Section 6.1.2. From Section 4.2.1 and Section 4.2.2 the triangles take up a memory space of

$$T_{tot} \cdot 128B + 11 \cdot T_{tot} \cdot 8B = (2V + 8.5\sqrt{V} \cdot 1.12) \cdot 216B \tag{7.10}$$

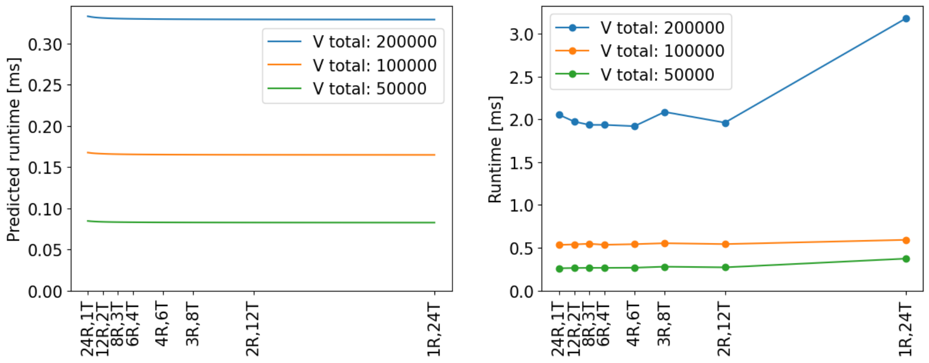
with $T_{tot} = T + T_{neumann}$, where the memory storage of each number of vertices are listed in Table 7.3. 50000 and 100000 vertices take up storage that fit in the local memory of $60B$ while 200000 vertices surpasses it. This causes the latter to access main memory during the computation, which is slower than accessing cache, and can hinder the scalability of the problem size $V$.

| Vertices | Memory storage [MB] |
|---|---|
| 50000 | 22.059807 |
| 100000 | 43.850265 |
| 200000 | 87.319614 |

**Table 7.3:** The memory storage for each number of vertices $V$, from Eq. (7.10).

The runtime prediction has the same performance for all combinations of ranks and threads that add up to 24 cores, which is the maximum performance for a single node. The runtime for 50000 and 100000 vertices have the same pattern for ranks and threads, while the runtime of 200000 vertices is significantly longer for a single rank. This is attributed to the overhead of OpenMP synchronization with multiple CPUs. The used nodes from the Idun cluster have two CPUs with $30MB$ memory each. For an even number of ranks, the triangles are completely separated on different CPUs, and are only interfering through the MPI communication for halo exchange. For a single rank, there is no halo exchange through MPI, and all cores from both CPUs have access to the same data, which can cause delay when cores from different CPUs are accessing the same data on the main memory. This would happen more frequently for 200000 vertices, which takes up a larger storage space than the cache size. A higher frequency of cache pollution can also occur as the cores from both CPUs with different cache memories needs access to the same data.

The configuration of three ranks and eight threads with 200000 vertices give a slightly longer computation time. This is attributed to overhead from OpenMP synchronization with two CPUs, as one of the ranks will have four cores from each CPU that have access to the same triangles from main memory. A setup of 24 ranks with one thread each has also given a slightly longer computation time. This can be attributed to the overhead of manually using MPI ranks, than the overhead of the internal handling of the OpenMP threads.



**(a)** Runtime prediction for ranks and threads. **(b)** Measured runtime for ranks and threads.

**Figure 7.10:** The predicted and measured runtime for the numbers of ranks and threads that adds up to all 24 cores, and for different number of vertices.

Fig. 7.11a and Fig. 7.11b compare the predicted and measured computation time for ranks and threads configurations that add up to 24 cores, for each computation part $p_1$, $p_2$ and $p_3$, and for 200000 vertices. From two ranks and upwards, predicted time for $p_1$ is longer than measured $p_1$, in proportion to $p_2$ and $p_3$. This is attributed to caching. Section 4.2.1 describes the memory storage of the application, where the triangles are stored in a $128B$ array, and the values from the Shallow Water Equations are stored in eleven different $8B$ arrays. In $p_1$, the triangular array is only accessed for *Neumann triangles*, which adds up to

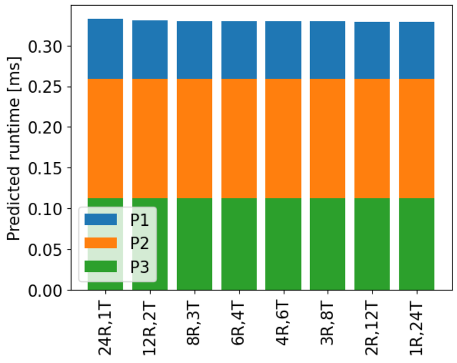
$$T_{neumann} \cdot 128B + 11 \cdot T_{tot} \cdot 8B = (8.5\sqrt{V} \cdot 1.12) \cdot 216B + (2V) \cdot 88B \qquad (7.11)$$

memory storage used. 200000 vertices use $36.12MB$ for $p_1$, which fits in cache memory of $60MB$.

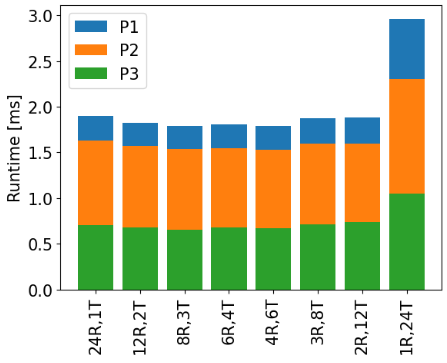
The configurations of a single rank give a similar distribution of $p_1$, $p_2$ and $p_3$ for predicted and measured runtime, making $p_1$ slower. This is attributed to cache pollution, and lack of cache optimization for $p_1$. Since the cores from both CPUs have access to all the data, an update in one of the cache memories will invalidate the data from the other.

### 7.3.3 Runtime Scaling

The analysis of scaling of ranks and threads are made separately, by measuring the computation time for an increasing number of threads and an increasing number of ranks. The results are compared with the performance model of Eq. (7.8).



**(a)** Runtime prediction for ranks and threads.

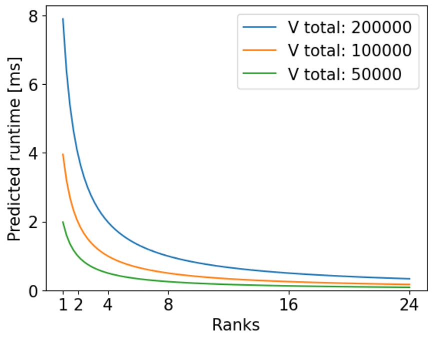


**(b)** Measured runtime for ranks and threads.

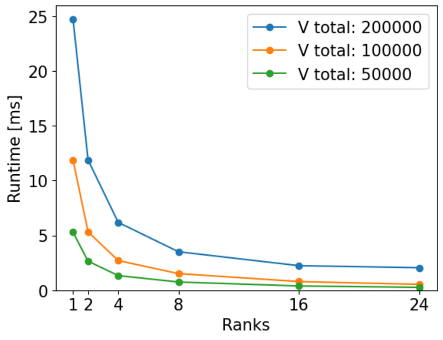
**Figure 7.11:** The predicted and measured runtime for the numbers of ranks and threads adding up to all 24 cores, given 200000 vertices. The computation time is split in the three parts $p_1$, $p_2$ and $p_3$.

### Ranks

Fig. 7.12a and Fig. 7.12b show a comparison of the predicted and measured computation time for a strong scaling of ranks, with a single thread for each rank. They both have the same pattern of a reduced runtime as more ranks are added, which increases parallel processing of the triangles. The predicted runtime is shorter by a factor of three, which is attributed to the peak memory bandwidth used in the formula, as discussed in Section 7.3.2. The measured runtime has a poorer scaling for 200000 vertices, which is attributed to caching. Section 7.3.2 also discusses how 200000 vertices take up a larger memory space than the cache size, from Table 7.3, which causes delay due to more main memory accesses.



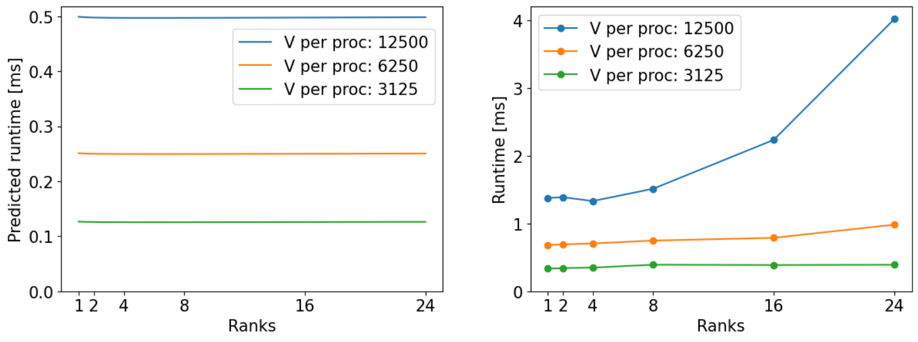
**(a)** Predicted computation time for ranks.



**(b)** Measured computation time for ranks.

**Figure 7.12:** The predicted and measured computation time for a strong scaling of ranks, while having a single thread per rank. Note that the measured time is approximately three times the predicted time.

Fig. 7.13a and Fig. 7.13b show a comparison of predicted and measured computation time for weak scaling of ranks. The prediction formula estimates a constant runtime as the number of ranks increases, since each rank has a constant problem size, and estimates a linear scale as the number of vertices per rank increases. For up to eight ranks, the measured runtime is constant for increasing ranks, and scales linearly for increasing number of vertices per rank. From 16 ranks, the runtime of 12500 vertices per rank increases. This is attributed to caching, as an overfilled cache causes delay due to main memory accesses, as discussed for strong scaling of ranks. Table 7.4 lists the memory storage for each number of processes, which is referring to the number of ranks, and for each number of vertices per process. The cache memory of $60MB$ is overfilled for 12500 vertices for 16 and 24 ranks, and for 6250 vertices for 24 ranks, but Fig. 7.13b only shows a measured overhead for 12500 vertices. Section 7.3.2 discusses caching during the first compute phase $p_1$, despite having a larger storage space than the cache size, which could be the case for 6250 vertices, as it only takes up $65.60MB$ memory for 24 ranks.



**(a)** Predicted computation time for ranks. **(b)** Measured computation time for ranks.

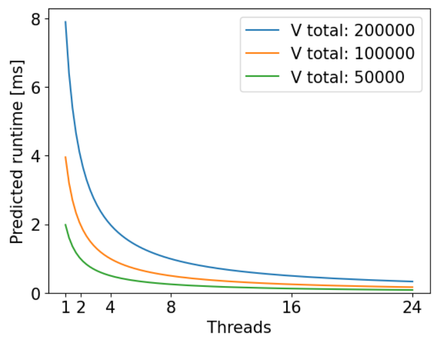
**Figure 7.13:** The predicted and measured computation time for a weak scaling of ranks, while having a single thread per rank. Note that the measured time is approximately three times or longer than the predicted time.

| Processes | 3125 | 6250 | 12500 |
|---|---|---|---|
| 1 | $1.46MB$ | $2.86MB$ | $5.63MB$ |
| 2 | $2.86MB$ | $5.63MB$ | $11.13MB$ |
| 4 | $5.63MB$ | $11.13MB$ | $22.06MB$ |
| 8 | $11.13MB$ | $22.06MB$ | $43.85MB$ |
| 16 | $22.06MB$ | $43.85MB$ | $87.32MB$ |
| 24 | $32.96MB$ | $65.60MB$ | $130.73MB$ |

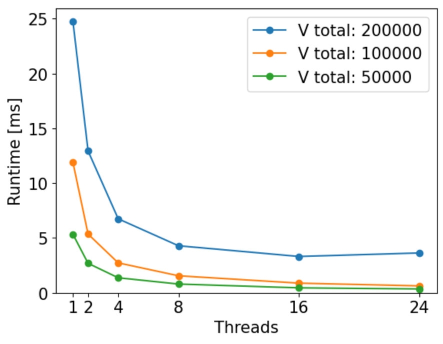
**Table 7.4:** The memory storage for weak scaling from Eq. (7.10), where each processing unit has a constant number of vertices of 3125, 6250 and 12500.

### Threads

Fig. 7.14a and Fig. 7.14b show a comparison of predicted and measured runtime for strong scaling of threads, while having a single rank, and Fig. 7.15a and Fig. 7.15b show the same comparison for weak scaling. The scaling of threads have the same pattern as both strong and weak scaling of ranks, where 200000 vertices have a poorer strong scaling and 12500 vertices per process have a poorer weak scaling. This is also attributed to caching, where $200000$ and $12500 \cdot 16 = 200000$ vertices take up more storage space than the cache size. The scaling of ranks and threads are different for large number of vertices, where the overhead is worse for thread scaling. This is attributed to a main memory issues for two CPUs when only having a single rank, which is discussed in Section 7.3.2.

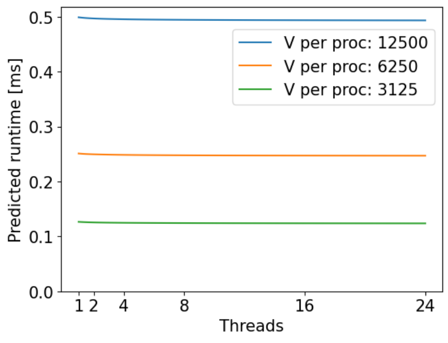


(a) Predicted computation time for threads.

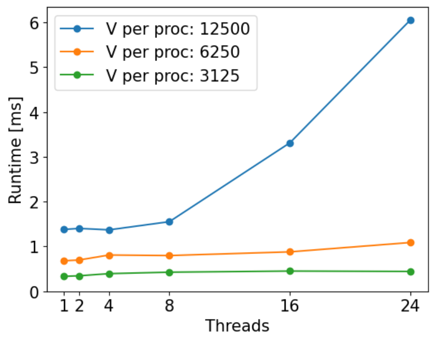


(b) Measured computation time for threads.

**Figure 7.14:** The predicted and measured computation time for a strong scaling of threads, while keeping a fixed singled rank. Note that the measured time is approximately three times the predicted time.



(a) Predicted computation time for threads.



(b) Measured computation time for threads.

**Figure 7.15:** The predicted and measured computation time for a weak scaling of threads, while keeping a fixed singled rank. Note that the measured time is approximately three times or longer than the predicted time.

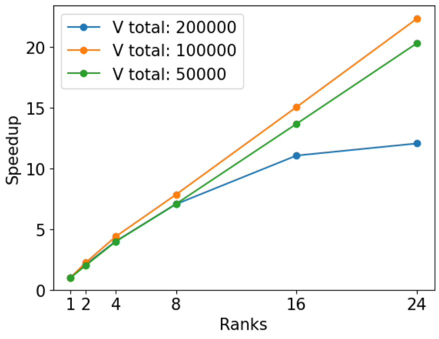
### 7.3.4 Speedup and Efficiency

A measurement of speedup and efficiency are made for both scaling of ranks and threads, with respect to strong scaling from Amdahl's law and of weak scaling from Gustafson's law.

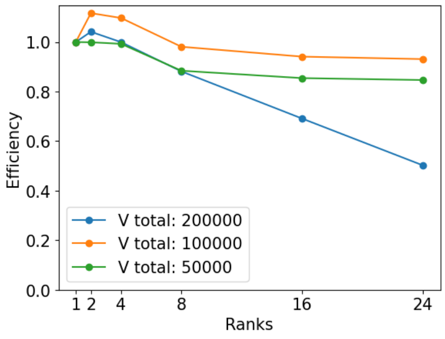
#### Ranks

Fig. 7.16a and Fig. 7.16b show the strong speedup and efficiency for ranks. An ideal speedup is linear, where there is no added overhead by increasing the number of ranks. This is the case for 50000 and 100000 vertices, whereas 200000 vertices is falling in speedup. This is attributed to having full cache memory, discussed in Section 7.3.3.

An ideal efficiency is a constant of the value 1, with each rank performing the same as a single rank. The speedup of 100000 vertices is superlinear for two and four ranks, meaning it has a higher efficiency than 1. This can be attributed to a suboptimal serial performance. Shown for the computation time in Fig. 7.12b, the runtime is more than twice the length between 50000 and 100000 vertices for a single rank, meaning it is not scaling linearly over the problem size $V$ for a serial application setup. A rank with a single thread can only utilize a single CPU of $30MB$ memory space, where a 100000 vertices take up a storage space of $43.85MB$, from Table 7.3 in Section 7.3.2. The benchmarking tests from Section 7.2.1 also showed a poorer bandwidth when using a single CPU.



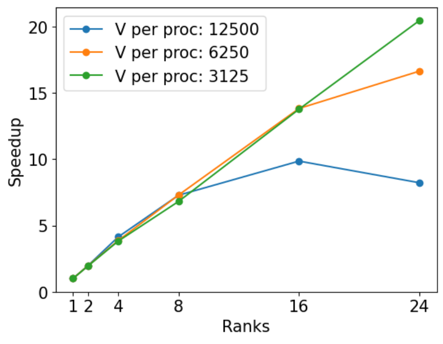
**(a)** Strong speedup for ranks.



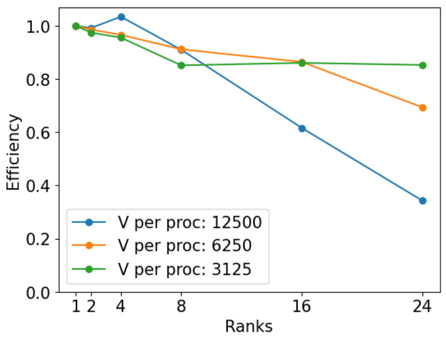
**(b)** Strong efficiency for ranks.

**Figure 7.16:** The speedup and efficiency for strong scaling of ranks, while having a single thread per rank.

Fig. 7.17a and Fig. 7.17b show the weak speedup and efficiency for ranks. The scaling is similar to strong scaling from Fig. 7.16a and Fig. 7.16b, but has a larger fall in speedup for 6250 and 12500 vertices per rank than the fall in speedup for 100000 and 200000 vertices for strong scaling. This is attributed to the scaling of the problem size, as weak scaling increases the number of vertices $V$ and takes up more memory space. 6250 vertices take up $65.60MB$ for 24 ranks and 12500 vertices take up $87.32MB$ for 16 ranks, which is larger than the cache size, and causes main memory accesses. There is also a superlinear speedup for 12500 vertices with four ranks, which can be attributed to an increased bandwidth, discussed for strong speedup for ranks.



**(a)** Weak speedup for ranks.



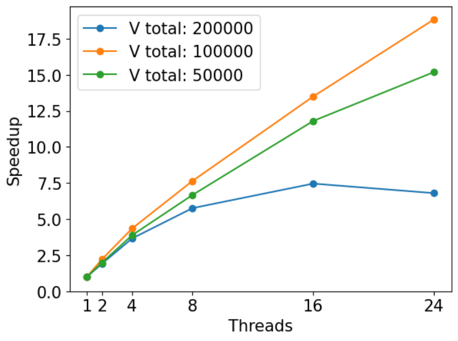
**(b)** Weak efficiency for ranks.

**Figure 7.17:** The speedup and efficiency for weak scaling of ranks, while having a single thread per rank.

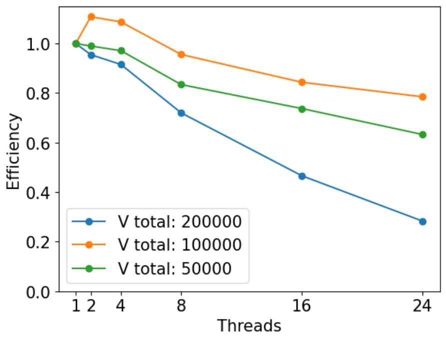
### Threads

Fig. 7.18a and Fig. 7.18b show the strong speedup and efficiency for thread scaling, and Fig. 7.19a and Fig. 7.19b show the weak speedup and efficiency for thread scaling. Both strong and weak scaling have the same pattern as for strong and weak scaling for ranks, where 200000 vertices is falling for strong speedup, and where 6250 and 12500 vertices per process is falling for weak speedup. This is attributed to a full cache memory causing frequent main memory accesses. There is also a superlinear speedup with 100000 vertices for strong scaling of both ranks and threads, which is attributed to a poorer serial performance of 100000 vertices due to both having a full cache and slower bandwidth.

The performance of thread scaling is generally lower than the performance for ranks, especially in the cases of 200000 vertices for strong scaling and 12500 for weak scaling. This is attributed to synchronization issues of using main memory for a single rank, discussed in Section 7.3.2.

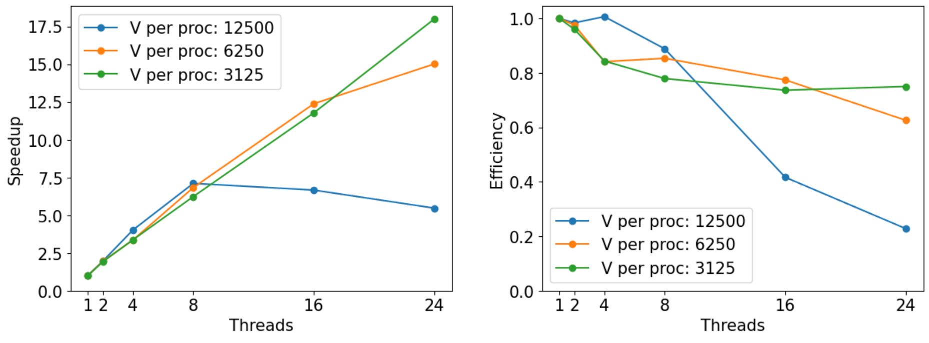


**(a)** Strong speedup for threads.



**(b)** Strong efficiency for threads.

**Figure 7.18:** The speedup and efficiency for strong scaling of threads, while keeping a fixed single rank.



**(a)** Weak speedup for threads. **(b)** Weak efficiency for threads.

**Figure 7.19:** The speedup and efficiency for weak scaling of threads, while keeping a fixed single rank.

### 7.3.5 Node Scaling

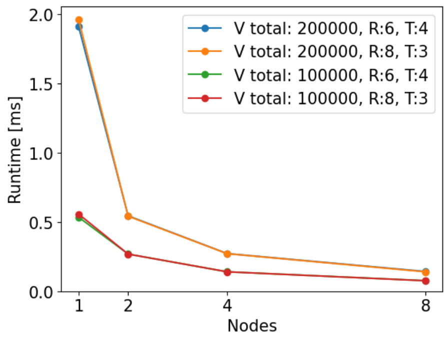
A scaling has been made for an increasing number of nodes, where each node has a full utilization of all 24 cores. The application tests have two configurations, of six ranks with four threads each and of eight ranks with three threads each. Strong and weak scaling are analysed for 100000 and 200000 total vertices, and 100000 and 200000 vertices per node.

Fig. 7.20a shows the runtime for strong scaling of nodes, which decreases as the workload is shared among the nodes. The runtime of 200000 vertices is four times the runtime of 100000 vertices for a single node, and twice as long for multiple nodes. This is attributed to delay due to a full cache memory, as 200000 vertices take up $87.32MB$ for a $60MB$ cache memory. For two nodes, only half the storage room is needed for each node, in which the cache memory can be utilized.

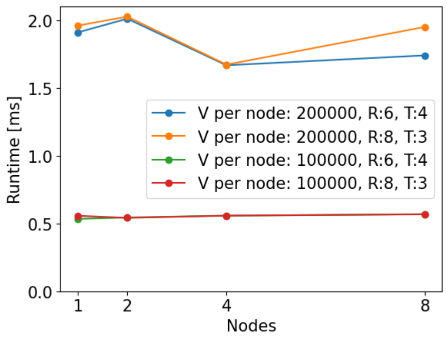
Fig. 7.20b shows the runtime for weak scaling of nodes, where each node has a constant workload, but with a lower runtime for four nodes with 200000 vertices each. This can be attributed to fewer cache misses, since the memory storage allocated is too large to be utilized by cache memory. The runtime of 200000 vertices per node is four times the runtime of 100000 vertices per node, which is also attributed to cache issues, as each node of 200000 vertices has a constant memory space that is larger than available cache space.

The two configurations of ranks and threads give similar runtimes, which is looked into in Section. 7.3.2

Fig. 7.21a and Fig. 7.21b show the strong and weak speedup, and Fig. 7.22a and Fig. 7.22b show the strong and weak efficiency. The speedup for both strong and weak scaling is close to linear for 100000 vertices. Strong scaling has a superlinear speedup for 200000 vertices, which is attributed to a suboptimal serial code, which uses more memory space than cache memory. Weak scaling also has a superlinear speedup for 200000 vertices for four nodes, which can be attributed to having fewer cache conflicts due to the used memory being too big to be utilized for caching.

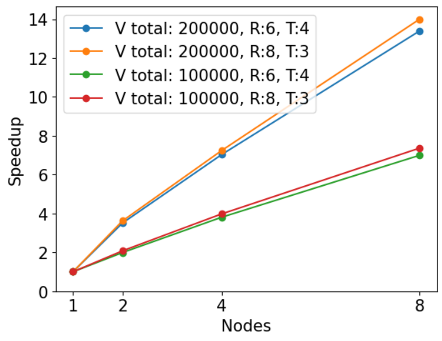


**(a)** Strong scaling of runtime over nodes.

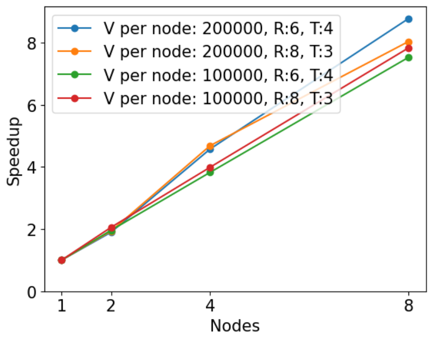


**(b)** Weak scaling of runtime over nodes.

**Figure 7.20:** Strong and weak scaling of computation time for nodes, with two configurations to utilize all 24 cores in each node. There are six ranks with four threads each, and eight ranks with three threads each.

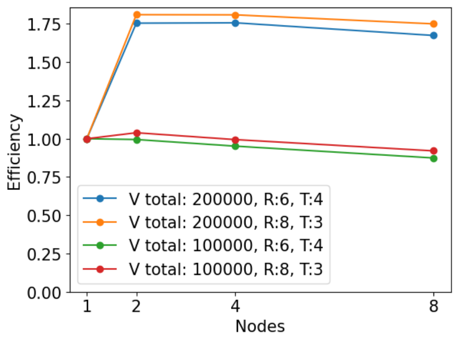


**(a)** Strong speedup of nodes.

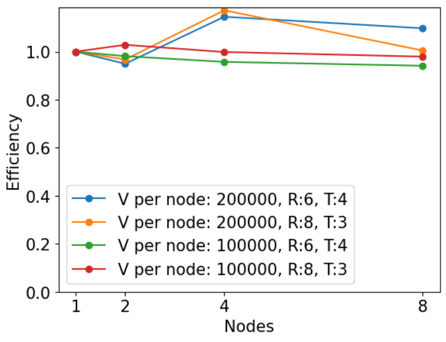


**(b)** Weak speedup of nodes.

**Figure 7.21:** Strong and weak speedup for nodes, with two configurations to utilize all 24 cores in each node. There are six ranks with four threads each, and eight ranks with three threads each.



**(a)** Strong efficiency of nodes.



**(b)** Weak efficiency of nodes.

**Figure 7.22:** Strong and weak efficiency for nodes, with two configurations to utilize all 24 cores in each node. There are six ranks with four threads each, and eight ranks with three threads each.

### 7.3.6 Summary of Scaling Characteristics

The proxy application can be scaled for the problem size, the number of vertices $V$, and for the number of processing units, determined by the number of ranks, threads and nodes. The prediction of computation time

$$t_{comp} = \left( \frac{210V}{ranks} + \frac{285.6\sqrt{V}}{ranks} + 49.28\sqrt{V} \right) \frac{8B}{bandwidth \cdot threads} \tag{7.12}$$

is the basis for all scaling analysis.

The results show that the application can be scaled for both threads and ranks, where the predicted runtime have the same scaling pattern as the measured runtime. Since the triangles are evenly distributed for the ranks, an even load balance gives little overhead to scaling. A better performance is gained by using both ranks and threads to fill out all the cores, whereas using a single rank and 24 threads gives the worst performance, which can be attributed to poor utilization of the two CPUs when using a single rank.

The performance model does not take in account caching of the hardware, which makes linearity of scaling dependent on whether the same type of memory is used. For weak scaling of ranks or threads for a single node, the memory usage increases, which can make the node switch from utilizing cache memory to using the main memory. For a strong scaling of nodes, the memory usage per node decreases, which can make the nodes switch from using main memory to utilizing cache. When scaling for nodes, a number of vertices between 100000 and 150000 for each node gives an optimal performance, where weak scaling gives a stable and predictable performance.

Chapter 8

# Conclusion

In this thesis, a proxy application was developed for solving shallow water equations using the finite volume method over a triangulation. Performance models were created for the application, including models made from benchmarking results, and validated through comparison with application execution of the Mehamn harbor.

The performance model was based on the triangulation structure, generated from the number of vertices. A general triangulation gave a wide range between the upper and lower bound for the number of triangles, which also gave a wide range for further performance estimations. By using Mehamn harbor geometrical exceptions could be ignored, and more accurate estimations were made. The new predictions of the triangular sizes were confirmed to fit the generated triangulations from all the application tests.

The benchmarking tests gave a basis to make the roofline and the Hockney models. The roofline model confirmed that the application was memory bound, where the maximum bandwidth was used to create an overestimated performance model of the application computation time. The Hockney model was used to predict communication time for message passing, which was similar to the application communication time between cores on the same node, but predicted approximately ten times lower communication time than the application communication time between nodes. The analysis also predicted the communication time to stay constant over an increasing number of ranks, and to grow slightly with the message size, given by the number of *Halo triangles*, which both were confirmed with the measured communication time from the application.

The runtime predictions separate application computation time and communication time. Since the communication time was estimated and confirmed to stay constant with the number of processors, the performance model for scaling was focused mainly on the computation time. Computation time was also expected to grow faster for the problem size, as it is dependent on the number of triangles, whereas communication time is only dependent on the *Halo triangles*.

The predicted computation time was expected and confirmed to overestimate the measured computation time from the application, and confirmed to scale similarly with both ranks and threads, for both strong and weak scaling. An exception occurs when assigning a single rank to multiple CPUs, where multiple threads cause cache pollution to the separate cache memories. The load balance was even, as the application distributes triangles evenly among the ranks. It was discovered that the cache size affected the performance, where an overfull cache caused further delay. An optimal utilization of cache, where the problem size take up a memory space of the entire cache, gives an optimal performance for computation time. Together with a constant communication time, this gives an overall optimal performance, with a predictable and stable weak scaling of multiprocessor nodes.

## 8.1 Future Work

The performance model did not take in account the effects of caching, where the results differed from the model when the cache was overfilled. A model that considers the hardware used, could improve the accuracy of the runtime predictions. The Hockney model also did not manage to predict communication time between nodes. A deeper study of the message time could be used to discover a new communication model.

The proxy application provides a flexible solution by using an arbitrary triangulation of the water surface. It could be of interest to compare this to other applications to locate the advantages and disadvantages of the different approaches, such as solutions using the finite difference method or different types of discretizations.

# Bibliography

[1] Weizhi Wang, Hans Bihs, Arun Kamath, and Øivind Asgeir Arntsen. Large scale cfd modelling of wave propagation into mehamn harbour, 2017. URL `http://hdl.handle.net/11250/2485580`.

[2] Magnus Själander, Magnus Jahre, Gunnar Tufte, and Nico Reissmann. EPIC: An energy-efficient, high-performance GPGPU computing research infrastructure, 2019.

[3] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *The Journal of Supercomputing*, 7(1-2):9–50, 1993. doi: 10.1007/bf01205181.

[4] Shameem Akhter and Jason Roberts. *Multi-core programming: Increasing performance through software multi-threading*. Intel Corporation, 2006.

[5] M.D. McCool. Scalable programming models for massively multicore processors. *Proceedings of the IEEE*, 96(5):816–831, May 2008. doi: 10.1109/jproc.2008.917731.

[6] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972. doi: 10.1109/tc.1972.5009071.

[7] U. Ramachandran, M. Solomon, and M. Vernon. Hardware support for interprocess communication. *Proceedings of the 14th annual international symposium on Computer architecture - ISCA '87*, 1987. doi: 10.1145/30350.30371.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture: A quantitative approach*. Morgan Kaufmann, 2019.

[9] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.1*, June 2015. URL `https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`. Accessed: 2022-08-09.

[10] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 4.5*, November 2015. URL `https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`. Accessed: 2022-08-09.

[11] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Michael Lang, Scott Pakin, and José Carlos Sancho. Using performance modeling to design large-scale systems. *Computer*, 42(11):42–49, 2009. doi: 10.1109/mc.2009.372.

[12] S.S. Dosanjh, R.F. Barrett, D.W. Doerfler, S.D. Hammond, K.S. Hemmert, M.A. Heroux, P.T. Lin, K.T. Pedretti, A.F. Rodrigues, T.G. Trucano, and et al. Exascale design space exploration and co-design. *Future Generation Computer Systems*, 30: 46–58, 2014. doi: 10.1016/j.future.2013.04.018.

[13] Samuel Williams, Andrew Waterman, and David Patterson. Roofline. *Communications of the ACM*, 52(4):65–76, 2009. doi: 10.1145/1498765.1498785.

[14] Gene M. Amdahl. Computer architecture and amdahl's law. *Computer*, 46(12):38–46, 2013. doi: 10.1109/MC.2013.418.

[15] John L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5): 532–533, May 1988. doi: 10.1145/42411.42415.

[16] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990. doi: 10.1145/79173.79181.

[17] Roger W. Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel Computing*, 20(3):389–398, 1994. doi: 10.1016/s0167-8191(06)800 21-9.

[18] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming - PPOPP '93*, 1993. doi: 10.1145/155332.155333.

[19] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997. doi: 10.1006/jpdc.1997.1346.

[20] Randall J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge Univ. Press, 2004.

[21] E. Kreyszig. *19.4 Methods for Elliptic Partial Differential Equations*, page 962–971. John Wiley & Sons, Inc, 8 edition, 1999.

[22] Øyvind Hjelle and Morten Dæhlen. *Triangulations and Applications*. Springer Berlin, Heidelberg, 2006.

[23] A.I. Delis and E.N. Mathioudakis. A finite volume method parallelization for the simulation of free surface shallow water flows. *Mathematics and Computers in Simulation*, 79(11):3339–3359, 2009. doi: 10.1016/j.matcom.2009.05.010.

[24] Renwei Liu, Dongjie Wang, Xinyu Zhang, Wang Li, and Bo Yu. Comparison Study on the Performances of Finite Volume Method and Finite Difference Method. *Journal of Applied Mathematics*, 2013:1–10, 2013. doi: 10.1155/2013/596218.

[25] Jingheng Xu, Guangwen Yang, Haohuan Fu, Wayne Luk, Lin Gan, Wen Shi, Wei Xue, Chao Yang, Yong Jiang, and Conghui He. Optimizing Finite Volume Method Solvers on Nvidia GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 30(12):2790–2805, 2019. doi: 10.1109/tpds.2019.2926084.

[26] C. Rosales. Porting to the intel xeon phi: Opportunities and challenges. *2013 Extreme Scaling Workshop (xsw 2013)*, 2013. doi: 10.1109/xsw.2013.5.

[27] Stéphane Zuckerman, Marc Pérache, and William Jalby. Fine tuning matrix multiplications on multicore. *High Performance Computing - HiPC 2008*, page 30–41, 2008. doi: 10.1007/978-3-540-89894-8_7.

[28] Thilo Kielmann, Henri E. Bal, and Kees Verstoep. Fast measurement of logp parameters for message passing platforms. In José Rolim, editor, *Parallel and Distributed Processing*, pages 1176–1183, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-45591-2.

[29] Kenneth H. Rosen. *10.7 Planar Graphs*, page 718–726. McGraw-Hill, 7 edition, 2012.

[30] Python Software Foundation. *Python 3.8.6 documentation*, September 2020. URL `https://docs.python.org/release/3.8.6/`. Accessed: 2022-08-09.

[31] Clark, Alex. *Pillow (PIL Fork) Documentation Release 8.3.1*, July 2021. URL `https://usermanual.wiki/m/fdbc524aadbf083a75306ee7c4c1cdda38923a84f85395cec4f73a436162a37e.pdf`. Accessed: 2022-08-09.

[32] SciPy community. *SciPy Reference Guide Release 1.5.4*, November 2020. URL `https://docs.scipy.org/doc/scipy-1.5.4/scipy-ref-1.5.4.pdf`. Accessed: 2022-08-09.

[33] Numpy community. *Numpy User Guide Release 1.19.0*, June 2020. URL `https://numpy.org/doc/1.19/numpy-user.pdf`. Accessed: 2022-08-09.

[34] John Hunter, Darren Dale, Eric Firing, Michael Droettboom, and The Matplotlib Development Team. *Matplotlib Release 3.3.3*, December 2020. URL `https://matplotlib.org/3.3.3/Matplotlib.pdf`. Accessed: 2022-08-09.

[35] Dell Inc. Poweredge r730, December 2016. URL `https://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/Dell-PowerEdge-R730-Spec-Sheet.pdf`. Accessed: 2022-08-09.

[36] Intel Corporation. Intel® Xeon® Processor E5-2600 v4 Product Family, March 2016. URL `https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-e5-brief.pdf`. Accessed: 2022-08-09.

[37] Intel Corporation. Intel® Parallel Studio XE 2020 Update 4, October 2020. URL https://www.intel.com/content/dam/develop/external/us/en/documents/ipsxe-2020-update-4-release-notes-en.pdf. Accessed: 2022-08-09.

[38] Intel Corporation. Intel® MPI Library for Linux* OS, 2019. URL https://www.intel.com/content/dam/develop/external/us/en/documents/intelmpi-2019u4-developer-guide-linux.pdf. Accessed: 2022-08-09.

[39] University of Tennessee. *Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard*, August 2001. URL https://netlib.org/blas/blast-forum/blas-report.pdf. Accessed: 2022-08-10.

Jenny Veronika Ip Manne

Performance Modeling of a FVM for the SWE

NTNU
Norwegian University of
Science and Technology