Ola Toft

Performance Modeling of Adaptive Mesh Refinement for the Shallow Water Equations

Master's thesis in Computer Science
Supervisor: Jan Christian Meyer
June 2020

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

Ola Toft

Performance Modeling of Adaptive Mesh Refinement for the Shallow Water Equations

Master's thesis in Computer Science
Supervisor: Jan Christian Meyer
June 2020

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

NTNU
Norwegian University of
Science and Technology

Problem Description

This study aims to create experimentally validated performance models for adaptively refined finite difference solutions to the shallow water equations, and use them to identify performance and scalability characteristics.

Abstract

In this thesis we study the performance and scalability characteristics of adaptively refined finite difference solutions to the shallow water equations.

The shallow water equations (SWE) are a set of equations used to describe the movement of fluids, such as tides, tsunami waves and storm surges. Finite difference methods (FDM) are one way to solve the SWE numerically. Adaptive mesh refinement (AMR) is a technique for providing the necessary resolution when solving equations numerically, but only in the necessary places in space and time, in order to keep the performance as high as possible.

In this thesis we develop a proxy application which solves the SWE using the MacCormack FDM and AMR. We create a performance model and use it to make predictions about the performance and scalability of the application, and experimentally validate those predictions. We set up initial disturbances shaped like waves in the fluid. Each of our experiments uses one or two different waves.

Our results show that if we add an extra level of refinement, the total number of points will increase with a factor between 1 and $r^3 + 1$, where r is the refinement between two levels. If the load balance is unchanged and there is no communication, the runtime will increase by the same factor.

For strong scaling, the wave with the better load balance scales better with the number of nodes, both for the total runtime and for the computation time. The parallel efficiency decreases by about the same factor as the load balance. The communication time is a small share of the total runtime.

For weak scaling, as we increase the problem size and the number of nodes N by the same factor, the computation time is about the same. The communication time does not increase with N for $N > 1$. The sum of the computation and the communication time is stable for $N > 1$. This means we can increase the number of nodes and get more work done in the same time.

Samandrag

I denne avhandlinga studerer me ytings- og skalerings-karakteristikkar ved adaptivt raffinerte endelege differansar-løysingar av gruntvasslikningane.

Gruntvasslikningane er eit sett med likningar som blir nytta til å beskrive rørsler i væsker, slik som tidevatn, tsunamiar og stormflo. Endelege differansar-metodar er ein måte å løyse gruntvasslikningane numerisk. Adaptiv mesh-raffinering er ein teknikk for å sørgje for den naudsynte oppløysinga ved løysing av numeriske likningar, men kun på dei naudsynte plassane i tid og rom, for å halde ytinga så høg som mogleg.

I denne avhandlinga utviklar me ein proxy-applikasjon som løyser gruntvasslikningane ved bruk av MacCormack sin endelege differansar-metode, samt adaptiv mesh-raffinering. Me lagar ein ytingsmodell og bruker han til å gjere prediksjonar av ytinga og skalerbarheita til applikasjonen, og validerer desse prediksjonane eksperimentelt. Me set opp initielle forstyrringar forma som bølgjer i væska. Kvart av eksperimenta våre nyttar éi eller to ulike bølgjer.

Resultata våre viser at om me legg til eit ekstra nivå med raffinering, vil det totale antalet punkt auke med ein faktor mellom 1 og $r^3 + 1$, der r er raffineringa mellom to nivå. Dersom lastbalansen er uendra og det ikkje er nokon kommunikasjon, vil køyretida auke med den same faktoren.

For sterk skalering vil bølgja med best lastbalanse skalere betre med antal nodar, både for den totale køyretida og for berekningstida. Den parallelle effektiviteten minkar med omlag same faktor som lastbalansen. Kommunikasjonstida utgjer ein låg andel av den totale køyretida.

For svak skalering, når me aukar problemstorleiken og antal nodar N med same faktor, vil berekningstida vere omtrent den same. Kommunikasjonstida aukar ikkje med N for $N > 1$. Summen av berekningstida og kommunikasjonstida er stabil for $N > 1$. Det betyr at me kan auke antal nodar og få meir arbeid utført på like lang tid.

Acknowledgements

I would like to thank my supervisor Jan Christian Meyer for introducing me to the field of performance modeling, for valuable discussions about interesting ideas, for helpful feedback, and for inspiration and motivation.

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

Table of Contents

List of Tables

List of Figures

Abbreviations

AMR	=	Adaptive Mesh Refinement
CFL	=	Courant-Friedrichs-Lewy
FDM	=	Finite Difference Methods
HPC	=	High Performance Computing
MPI	=	Message Passing Interface
SWE	=	Shallow Water Equations

Chapter 1

Introduction

1.1 Motivation

The shallow water equations (SWE) are used by physicists to model the flow in the deep ocean, coastal areas and rivers. They are used to describe phenomena such as tides, tsunami waves and storm surges [1]. Finite difference methods (FDM) are one way to implement and solve the SWE numerically.

Adaptive mesh refinement (AMR) is a technique for providing the necessary resolution when solving such equations, while keeping the performance as high as possible. This is done by only applying the necessary resolution where it is required, both in space and in time.

Our motivation for creating a performance model for adaptively refined finite difference solutions to the SWE is to make it possible to make better decisions about when and how to use such an application, and to provide a starting point for understanding the performance and scalability of different real-world applications using the SWE, FDMs, AMR, or some combination of them.

1.2 Scope

In this thesis we develop a proxy application which solves the SWE using the MacCormack FDM and AMR. The application is parallelized using an MPI and OpenMP hybrid solution. We create a performance model and use it to make predictions about the performance and scalability of the application. We make predictions about the cost of an extra level of refinement, the strong scaling and how it is impacted by the load balance, and the weak scaling. We run experiments on two machines, Idun and Vilje, to validate the predictions of our performance model.

1.3 Structure

The rest of this thesis is structured as follows: In Chapter 2 we provide background information about the different parts which can be used to implement the SWE, parallelize them and model them, as well as related work.

In Chapter 3 we give an overview of how our proxy application works. In Chapter 4 and 5 we describe the different parts of our proxy application in more detail. In Chapter 4 we describe the integration parts and how we solve the SWE using an FDM. In Chapter 5 we describe the AMR parts.

In Chapter 6 we derive our performance model and make predictions about the performance and scalability of our application. In Chapter 7 we describe the experimental setup for validation of our predictions. In Chapter 8 we describe and discuss the results from our experiments and whether they validate our predictions.

In Chapter 9 we provide a conclusion of our work and point to ideas for future work.

Chapter 2

Background

In this chapter we provide background information the different parts which can be used to implement the SWE, parallelize them and model them, as well as related work.

First we describe the SWE and what they are used for. Then we describe what FDMs are. Next we talk about different types of boundary conditions and how to handle them. Then we describe AMR and its advantages. Next we classify and describe a set of different parallel programming APIs. Then we describe what a proxy application is and attempt to classify our proxy application. Next we talk about performance modeling and some important models and aspects of it. Finally, we summarize some work related to our thesis.

2.1 Shallow Water Equations (SWE)

In this section we describe the shallow water equations and what they are used for.

The shallow water equations (SWE) are a set of hyperbolic partial differential equations. They are used to describe the movement in a fluid that is relatively shallow. In this context, shallow means that the horizontal dimension is much larger than the vertical dimension. An ocean can be deep, but it's also far more wide than it is deep, making it relatively shallow.

Scientists use the SWE to model oceans, coastal regions and rivers to predict phenomena such as tides, storm surge levels and ocean currents.

2.1.1 Mathematics

The shallow water equations are given as

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

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

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

Where

- ρ: Fluid density
- η: Total fluid column height
- u, v: The fluid's horizontal flow velocity
- g: Gravity

Equation 2.1 describes the relation between the change in the fluid column height over time and the change in the horizontal velocity over space. Equation 2.2 describes the change in the horizontal velocity in the X direction over time in relation to the change over space. Equation 2.3 describes the equivalent for the Y direction.

2.2 Finite Difference Methods (FDM)

Finite difference methods (FDM) are a way to discretize differential equations into systems of difference equations that be solved by a computer.

FDM models the SWE as a regular grid of points, where each point represents the height and the velocity of the fluid at a given Cartesian coordinate.

Figure 2.1 illustrates the values of the height and the velocity of the fluid at different points in space with columns of different height. Each column can be thought of as infinitely thin and only valid at its specific point. The points should be thought of as lying at the intersection of coordinates, not between them. For the fluid column height (η) the height of the columns map straightforwardly to the height of the fluid at that point. For the velocity in the Y direction a positive column height maps to a positive velocity in the Y direction and the fluid moving forward in the Y direction, while a negative column height maps to a negative velocity and the fluid moving backward in the Y direction. The mapping for the X direction is equivalent.

The value of a point at the next time step depends on the current values of its neighbor points. This means that a given point will only have to access the memory describing its neighbor points. If the order of the method is increased, we will need to access the neighbor second removed, third removed and so on, but the application memory access pattern will still be quite local.

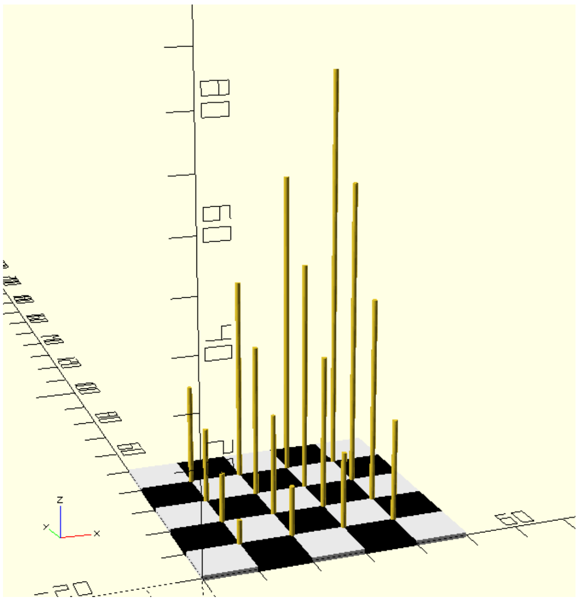


Figure 2.1: The height of the columns represent the height and the velocity of the fluid at different points in space.

Only the edge of the Cartesian coordinates on a shared memory node is sent to its neighbor nodes. This means the amount of network communication relative to local memory accesses is low, and that the amount of network communication scales linearly with the number of nodes. Nodes and shared memory are explained in Section 2.5.

Sod provides a survey of several different FDMs [2].

2.3 Boundary Conditions

Boundary conditions are constraints which are used to provide appropriate values at the boundary of a domain solved by differential equations [3].

2.3.1 Types

Dirichlet and Neumann are two major types of boundary conditions. The Dirichlet boundary condition specifies the value of a function. The Neumann boundary condition specifies the value of the derivative of the function.

In addition there are different types of combinations of Dirichlet and Neumann boundary conditions. The Robin boundary condition consists of a linear combination of Dirichlet and Neumann boundary conditions. The mixed boundary condition applies either the Dirichlet or the Neumann boundary condition to different parts of the domain. The Cauchy boundary condition applies two constraints, one Dirichlet and one Neumann boundary condition, on the whole domain. This is different from the Robin boundary condition,

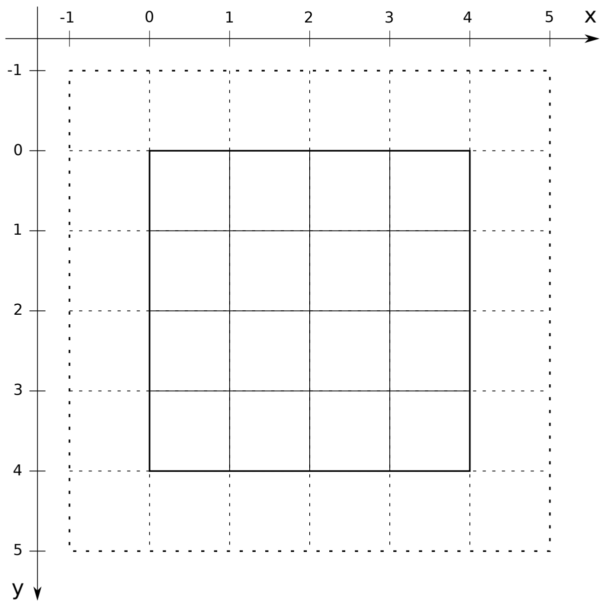


Figure 2.2: Domain with ghost points. The points in the domain are shown as the intersection of the solid lines. The boundary points are the outermost points in the domain. The ghost points are shown at the intersection of the dotted lines.

which applies only one constraint, but as a linear combination of the two major types of boundary conditions.

2.3.2 Ghost Points

The points at the edge of the domain are included in the calculations of an FDM. Ghost points are extra points outside the domain and the boundary points. This is illustrated in Figure 2.2. Ghost points can be used to implement the boundary condition in a finite difference method. This is done by setting the values of the ghost points in a specific way, depending on the boundary condition. Then the boundary points can read the ghost points as if they were a normal point in the domain, and use the same scheme as all the other points in the domain.

2.3.3 Boundaries, Borders and Terminology

Boundary conditions are used at the boundary of the global domain. When solving a problem numerically, we may split the global domain into multiple subdomains. Two such types of subdomains are the grids mentioned in Section 2.4 and the ranks mentioned in Section 2.5.3.1. Each instance of both of these types of subdomains have their own internal borders with their own border points, and depending on the implementation, may also have their own ghost points. These internal borders do not deal with reflection or any other boundary condition of that type. Their goal is simply to propagate the solution as smoothly as possible, as if there were no partitioning into subdomains. In some cases that may involve imperfect interpolation, but the goal of simply propagating the solution remains the same.

As such, we will use the phrase ”boundary condition” exclusively for the handling of the boundary of the global domain, and use the phrases ”grid borders” and ”rank borders” for internal borders.

2.4 Adaptive Mesh Refinement

Adaptive mesh refinement (AMR) [4] [5] is a technique to refine only the parts of the domain that require higher resolution, and adaptively update which areas are refined over time. Thus, the required accuracy can be achieved at a potentially far lower cost than from refining every part of the domain at all times.

In this section we first describe regular, static mesh refinement, and its advantage relative to full refinement. We then show how AMR builds upon and improves mesh refinement.

2.4.1 Mesh Refinement

Mesh refinement is a technique to refine only the parts of the domain that require higher resolution.

Figure 2.3 shows the layout and resolution of the grid in an application which doesn’t use mesh refinement. All parts of the domain are given a high level of refinement if this is required by some part of the domain. This is computationally expensive.

Figure 2.4 shows the layout and resolution of grids in an application which uses mesh refinement. Only the parts of the domain that require a certain resolution are refined to the necessary level. The other parts are given a coarser refinement. This is far less computationally expensive, but still provides the required accuracy.

Mesh refinement is well suited for applications where the parts that require a higher resolution don’t change over time, such as for an airfoil.

2.4.2 Adaptive Mesh Refinement

Regular, static mesh refinement is less suited if the parts that require a higher resolution change over time. An example of this is a tsunami moving towards the coast. The idea of AMR is to also adaptively update which areas are refined over time, moving the refined areas along with the parts that require a higher resolution.

Consider a case where some of the parts that required a higher resolution in Figure 2.4 move to the right. For regular, static mesh refinement, we have to increase the size of the refined grids for all timesteps in order to cover the refinement requirement for all parts for all timesteps, as shown in Figure 2.5. Since the area which is refined in each timestep is larger than when the parts that require a higher resolution do not move, the computational cost increases.

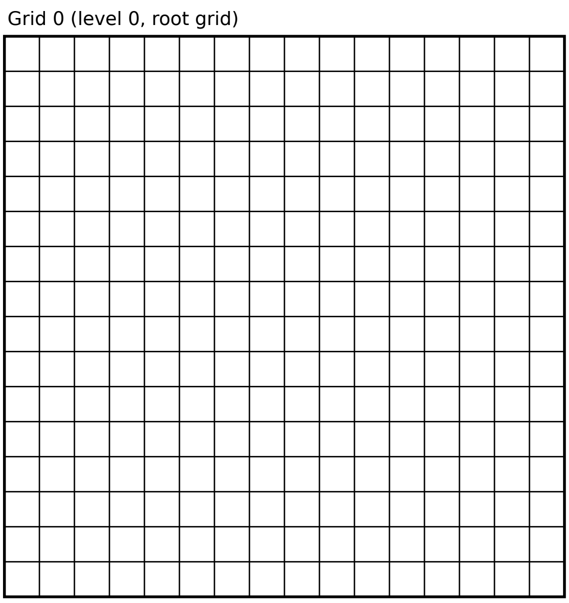


Figure 2.3: Full refinement: The layout and resolution of the grid in an application which does not use mesh refinement.

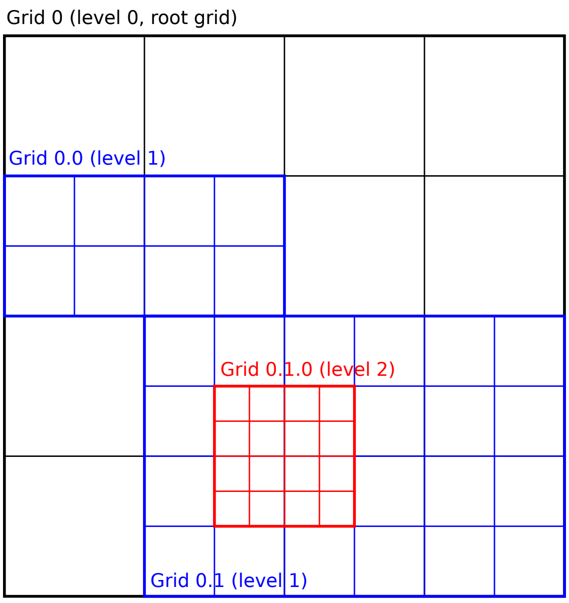


Figure 2.4: Mesh refinement: The layout and resolution of grids in an application which uses mesh refinement.

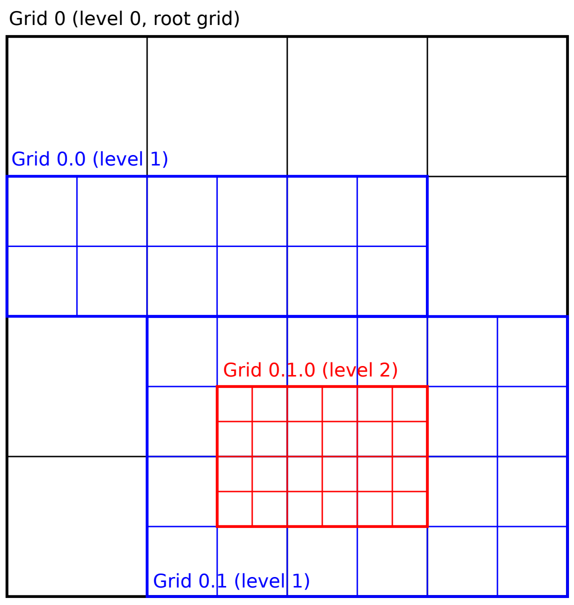


Figure 2.5: Regular, static mesh refinement when the parts that require a higher resolution change over time.

This is assuming we know which areas will need refinement. If this cannot be known, regular, static mesh refinement will have have to refine everything at maximum resolution all the time, identically to an application with no mesh refinement.

By contrast, an application using AMR will adaptively update which areas are refined over time, as shown in Figure 2.6. This reduces the area which will have to be refined, and keeps the computational cost low.

2.5 Parallel Programming APIs

In this section we look at some APIs for making our program parallel. For each API we give a short description and talk about which systems it is suited for.

2.5.1 Shared Memory Systems

Shared memory is memory which can be accessed by multiple compute units. An example of a shared memory system is a *node*, which is a collection of compute units, for instance CPUs, where each compute unit can read and write the same local memory. In this section we describe two APIs which can be used to run programs in parallel on shared memory systems.

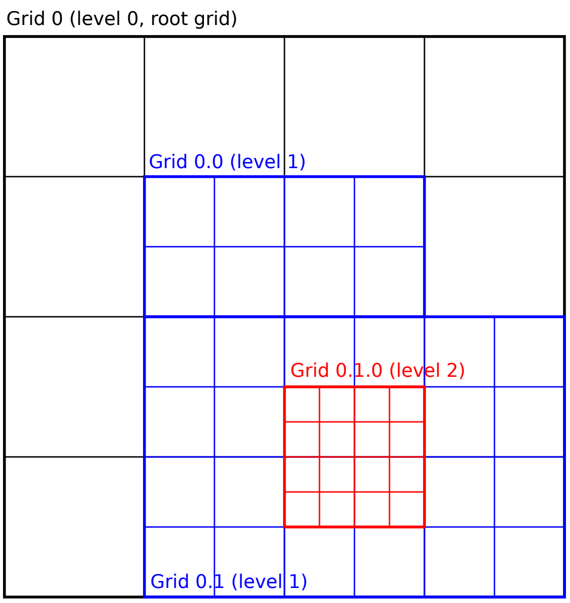


Figure 2.6: Adaptive mesh refinement after the parts that require a higher resolution have moved to the right.

2.5.1.1 POSIX Threads

POSIX threads, or pthreads, is an implementation of threads on POSIX systems [6]. It specifies a library which can be used to start threads, make the threads work on a problem in parallel, and stop the threads when they are done. The developer has to manually start, stop and assign different pieces of work to the threads.

2.5.1.2 OpenMP

OpenMP is an API for multi-platform shared-memory parallel programming in C/C++ and Fortran [7]. It provides special functions and preprocessor directives which make it simple to provide parallelism within a multi-core node [6]. By specifying a directive before a `for` loop, OpenMP will take care of creating a team of threads and dividing the work in the `for` loop between the threads.

2.5.2 GPU platforms

Graphics Processing Units, or GPUs, have hundreds of parallel *single instruction, multiple data*-like processors. In this section we describe two APIs for using GPUs for general-purpose computing.

2.5.2.1 CUDA

CUDA is a co-evolved hardware-software architecture that enables HPC developers to harness the computational power of the GPU in the environment of the C programming language [8]. Developers write a specially declared C function, called a kernel, to do a computation for one index in some data collection. Then that function is invoked with as many threads as the data collection has elements, and the GPU performs the computation in parallel with one thread per index. CUDA can only be used with NVIDIA GPUs.

2.5.2.2 OpenCL

OpenCL is an open standard for general purpose parallel programming across CPUs, GPUs and other processors [9]. Similarly to CUDA, developers write kernels which may be executed in parallel on a GPU. OpenCL is platform-independent and can be used with GPUs from multiple vendors.

2.5.3 Distributed Memory Systems

Distributed memory is memory which cannot be directly accessed by all compute units. An example of a distributed memory system is a *node cluster*, which is a collection of nodes with an interconnect network for communication between nodes. In this section we describe two APIs which can be used to distribute work between compute units with distributed memory.

2.5.3.1 MPI

Message Passing Interface (MPI) is a standardized API typically used for distributed-memory parallel programming in C/C++ and Fortran [10]. It defines a library of functions which can be used to provide parallelism between multiple nodes [6]. For instance, MPI can be used to explicitly send different parts of an array to different nodes to work on. Each node typically runs one unique instance of the program, identified by a rank. After each node completes its work, MPI can be used to combine the results again.

2.5.3.2 Chapel

Chapel is a global-view parallel language [11]. This means that the developer can treat the program and its data as if the program ran on one compute unit with shared memory, while the language transparently and implicitly distributes the work between different processors on different distributed-memory nodes.

2.6 Proxy Applications

A proxy application is a small, simplified application which shares important features and characteristics of a complete application. The advantage of proxy applications are their smaller size and complexity. Because of this they are often used as models for performance-critical computations [12].

Dosanjh *et al.* provide a classification of different types of proxy applications [13]. Our proxy application is similar to what is called a *compact app* in that it provides a simplified, but complete physics simulation. However, our application differs from a compact app in that our application can be useful in early design studies, and in its lower number of lines of code. In those regards it is more similar to a *miniapp*. Our application differs from a miniapp in that it attempts to capture many rather than one or a few performance-impacting aspects.

2.7 Performance Modeling

Performance modeling is used to predict and understand the performance and scalability of running an application on a system. A good performance model enables us to make good choices regarding our application and our system, and may help in detecting bottlenecks.

In this section we explore a set of different performance models, their parameters, and their suitability to different situations.

2.7.1 Amdahl's Law

Amdahl describes that as we increase the resources of a system while keeping the total problem size constant, the speedup will be limited by the inherently serial part of the program [14]. Suppose we have a program with serial runtime T_{serial}. The program can be divided into the part that can be parallelized, and the part which cannot be parallelized and is inherently serial. The runtime of the parallel program can be modeled as

$$T_{parallel} = \frac{p \times T_{serial}}{N} + (1 - p) \times T_{serial} \tag{2.4}$$

Where

- p: Share of program which can be parallelized
- N: Number of resources which can be used to parallelize. Examples: Number of processors, or number of nodes

The speedup of the parallel program relative to the serial program is then given as

$$S = \frac{T_{serial}}{T_{parallel}} = \frac{T_{serial}}{\frac{p \times T_{serial}}{N} + (1 - p) \times T_{serial}} \tag{2.5}$$

As we increase the resources of the system, the speedup will go towards

$$\lim_{N \to \infty} S = \frac{T_{serial}}{(1 - p) \times T_{serial}} \tag{2.6}$$

which simplifies to

$$\lim_{N \to \infty} S = \frac{1}{1 - p} \tag{2.7}$$

Suppose, for example, that $p = 0.9$. Then regardless of how much we increase the resources of our system, the speedup of our program will be at most

$$\lim_{N \to \infty} S = \frac{1}{1 - 0.9} = 10 \tag{2.8}$$

The process of increasing the resources of a system while keeping the total problem size constant is known as *strong scaling*.

The *parallel efficiency* of this speedup is defined as

$$E = \frac{S}{N} \tag{2.9}$$

A linear speedup will give an efficiency of 1.

2.7.2 Gustafson's Law

Gustafson describes that as we increase the resources of a system, we often also want to increase the total problem size [15]. Instead of doing the same work in less time, we often want to do more work in the same time. Gustafson finds that it is often the parallel part of a program which scales with the problem size, while the inherently serial part is constant. Then the share of the inherently serial part $1 - p$ is reduced. Thus, Gustafson finds that Amdahl's law is too pessimistic.

The process of increasing the resources of a system and the total problem size by the same factor is known as *weak scaling*. For weak scaling the problem size per resource stays the same.

The *scaled speedup* is defined as the factor or more work done in the same time as we increase the resources of a system. This can be expressed as

$$S_s = N(1 - p) + p \tag{2.10}$$

The *parallel efficiency* of this scaled speedup is given as

$$E = \frac{S_s}{N} = \frac{N(1 - p) + p}{N} \tag{2.11}$$

which simplifies to

$$E = p + \frac{1 - p}{N} \tag{2.12}$$

As we increase the resources of the system and the problem size by the same amount, the parallel efficiency will tend towards

$$\lim_{N \to \infty} E = p \tag{2.13}$$

Theoretically, we can increase N as much as we want and get $N \times p$ times as much work done in the same time.

2.7.3 Bulk-Synchronous Parallelism (BSP)

Bulk-Synchronous Parallelism (BSP) aims to be a bridging model between hardware and software for parallel computation [16]. The model consists of three parts:

1. A number of components which perform processing and/or memory functions.
2. A router which delivers messages between pairs of components.
3. A way to synchronize all or some of the components after a regular interval of L time units.

BSP combines all these parts into *supersteps*. In each superstep, all components perform some amount of processing and/or memory functions, all routers deliver some amount of messages, and the superstep ends with a synchronization of the components, before the next superstep begins.

The cost of one superstep for N components is modeled as

$$\max_{n}^{N}(w_n) + \max_{n}^{N}(h_n \times g) + l \tag{2.14}$$

Where

- w_n: Cost of local computation for component n
- h_n: Number of messages sent or received by component n
- g: Cost of delivering one message of size 1
- l: Cost of synchronizing all components

BSP models the computation with one parameter for the number of operations to be done. The model assumes that accessing the local memory needed and performing one operation can be done in one time step. The communication is modeled as the throughput times the number of messages to be sent, plus the latency or startup cost. Synchronization is assumed to be done every L units of time. Programs are written for v virtual parallel processors, and run on p physical processors.

The different parts of the superstep are often found in HPC applications, where a number of processors perform some computation and a number of messages is sent between nodes, before all the processors are synchronized. Once the synchronization is completed, we can be sure that all nodes have the necessary information, and we are ready to continue to the next step of our program.

2.7.4 Roofline

Roofline aims to provide insight into multicore architectures, while at the same time have low complexity [17]. It has two parameters, off-chip memory traffic measured in bytes per second, and peak floating-point performance, measured in floating-point operations per second. The number of floating-point operations per byte of DRAM traffic is called operational intensity. A low operational intensity means that the program likely is memory bound. A high operational intensity means that the program likely is processor-bound.

The attainable FLOPS per second can then be calculated as

$$[FLOP/s]_A = min([FLOP/s]_P, BW \times OI) \tag{2.15}$$

Where

- $[FLOP/s]_P$: Peak FLOP/s
- BW: Peak memory bandwidth, measured in bytes per second (B/s)
- OI: Operation intensity

The attainable FLOPS/s given the operational intensity can be plotted into a graph, showing that the performance gradually increases as the operational intensity increases and the program becomes less and less memory bound, until it hits the ”roofline” and becomes compute bound.

2.8 Related Work

Berger compares the computation time between AMR, a coarse and a fine solution for four example problems, each with a single set of parameters [4]. The comparison successfully shows that there are cases where AMR is faster than a fine solution and still reasonably accurate. Whether the author specifies the cost of the overhead, specifies how much is refined or even gives a general description of how much is refined, varies from problem to problem. The performance of one of the examples suffer from floating point underflow, and another from execution overhead. The article provides a formula for how often re-gridding should be performed in order to minimize the cost of the algorithm, given as a balance between the cost of integration and regridding. A cost estimate for integration and for regridding would be needed to use the formula.

Jameson compares lower-order AMR schemes to higher-order non-AMR schemes, and finds that the former are computationally more expensive than the latter when the goal is to reduce the error of the solution to a certain amount [18]. The author also states that the work of a higher-order AMR scheme can be less than the work of a higher-order non-AMR scheme. It is enough to look at the integration part to show that lower-order AMR schemes have a higher computational cost than higher-order non-AMR schemes. Thus, the article does not describe the other parts of AMR, neither their amount of work nor their computational cost. We provide details of how often the other parts are performed, and make estimates of their computational cost relative to each other. The article focuses at serial performance, while we describe at the effect of parallelization and scalability.

Erduran *et al.* evalute the performance of finite volume solutions to the shallow water equations [19]. They briefly consider finite difference methods in the introduction, but decide to focus on finite volume solutions because they can be implemented on unstructured grids and are easier to implement shock capturing capabilities for. Kubatko *et al.* study the performance of two finite element methods [1]. They attempt to address a weakness of finite element methods compared to finite difference methods, namely the high computational cost. We study the performance of an application using a finite difference method.

Sætra *et al.* provide an implementation of the SWE using a finite-volume method and AMR on the GPU [20]. The article makes measurements of accuracy and performance for some examples. We implement the SWE using a finite-difference method and AMR on the CPU, create a performance model and experimentally verify the predictions of our performance model.

Chapter 3

Proxy Application

We have made a proxy application which solves the SWE using an FDM and AMR. The application models a body filled with a fluid. It starts out from an initial disturbance in the fluid, and then models the development of this disturbance, how it spreads to different coordinates over time, is reflected by the walls, and how different waves interact with each other. This affects which parts are refined and to what degree, which in turn affects the solution as well as the performance.

In this chapter we describe how our proxy application works. First we describe the grid, a data structure which is central to our application. Then we give a short description of the main algorithm and its different parts, and show how all the parts fit together. Further details about each part are described in separate chapters. Finally we describe which parallel programming APIs we choose to parallelize our application, and why.

3.1 Data Structure

The main data structure of AMR is the grid. A grid is responsible for finding the solution in a specified area. It may have a number of child grids with a higher level of refinement in space and time if required. These child grids in turn are responsible for finding the solution in a subarea of their parent grid, and may recursively hold even more refined child grids of their own. The coarsest grid is referred to as the root grid. Figure 2.4 shows an example of a nested grid hierarchy with three different levels of refinement. Figure 3.1 shows how the grids in Figure 2.4 relate to each other.

3.1.1 Variations and Choices for Grids

There are several variations for how to implement grids in AMR. In this section we briefly explain these variations, what we choose and why.

Each child grid could either be *patched* into, i.e. exist instead of, or *overlay*, i.e. exist in addition to, some region of its parent. Patching reduces the number of points that need processing. However, it requires storage overhead and processing typically proportional

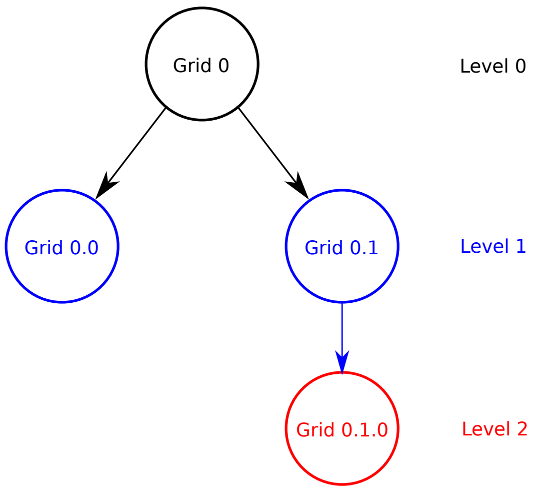


Figure 3.1: Relation between the grids shown in Figure 2.4.

to the number of points. Overlaying makes it possible to integrate each grid independently of any possible child grids. Furthermore, no additional work is needed for a parent grid if a child grid is removed during regridding. *Overlaying* also gives each grid a uniform structure. This make them well suited for vectorization and parallelization of operations, as well as caching and prefetching of memory, on modern processors. For these reasons, we choose to let each child grid *overlay* its parent.

For simplicity, we choose to only allow rectangular grids, and child grids may not be rotated with respect to parent grids. We also choose to let each child grid be contained within only one parent grid, instead of overlapping multiple parent grids.

We choose to have one root grid for each node. That is, the global domain is divided evenly among the nodes. This ensures good parallelism between nodes.

We choose to use a block-based strategy instead of cell-based strategy. A cell-based strategy refines each single point if and only if the refinement criteria is met. A block-based strategy refines blocks of cells that have been grouped together and are used to create grids. These blocks may contain some points which do not require refinement, which is something that will not happen in a cell-based strategy. However, a block-based strategy can achieve better cache locality and memory access patterns, and gives a lower cost from handling grid borders than that of a cell-based strategy. It is also possible to achieve the effect of a cell-based strategy by setting the block size to 1. The effect of such a small block size is discussed in Section 5.1.

3.1.2 Grid Refinement

We choose the same refinement in time as in space, that is

$$r_t = r_y = r_x \tag{3.1}$$

Where

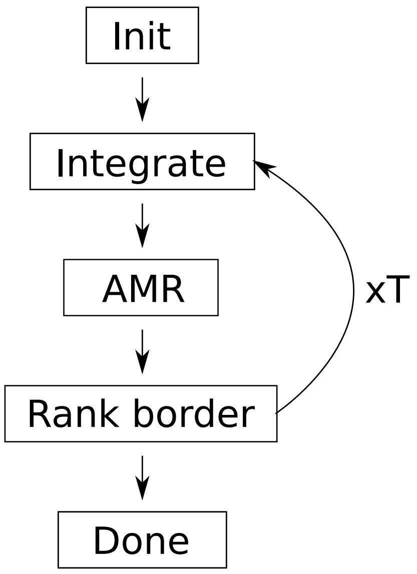


Figure 3.2: Overview of the main algorithm and its parts at the coarsest level. ”xt” denotes ”run T times”.

- r_a: Refinement in direction a

The reason for this is to keep the Courant number constant for all levels of refinement. That way, by choosing a Courant number which satisfies the Courant-Friedrichs-Lewy (CFL) condition [21] at the root level, we can be confident that the CFL condition will hold for all levels of refinement.

3.2 Algorithm

Figure 3.2 shows the main algorithm and its parts at the coarsest level. After a short initialization the algorithm enters the main loop. The loop starts by integrating the solution of the SWE in time. Then it performs AMR-specific actions to refine the solution at the necessary places. Finally, we handle the rank border by exchanging necessary values with neighbor ranks. The loop is run I times, where I is the number of timesteps we specify. After completing the loop, the application is done.

3.2.1 Initialization

Our application is run with a certain problem size and a certain number of nodes. Two of the parameters of the problem are the length X and the width Y, which together define the area and the number of points for a given timestep at the coarsest level of refinement. We assign one MPI rank to each node, and split the area equally among all the ranks.

We set up one root grid for each rank. Then we initialize the fluid column height and the velocity at each point in those grids. We create some disturbance in the fluid, for

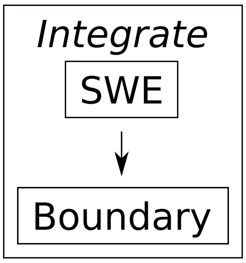


Figure 3.3: Overview of the integration step and how its substeps fit together.

instance a wave from the edge of the domain. Our goal is to let our application evolve this disturbance and its changes to the fluid over time.

3.2.2 Integration

The integration step is the main step of our application. This is where we evolve the solution in time. It consists of two subsequent substeps, as shown in Figure 3.3: solving the equations, in our case the SWE, and handling the boundary conditions.

3.2.2.1 SWE

In the SWE step we evolve the solution of our equations in time. In our case this means solving the SWE using an FDM for a given timestep. The details of how we solve the SWE using an FMD are described in Section 4.1.

3.2.2.2 Boundary Conditions

We choose to handle the boundary conditions by reflecting the fluid height and velocity. The details of how we achieve this are given in Section 4.2.

3.2.3 AMR

We use the AMR step to refine only the parts of the domain which require higher resolution, and adaptively update which areas are refined over time. Figure 3.4 shows the substeps of the AMR step and how they fit together with each other and with the integration step. AMR, its datastructures and its substeps are described in more detail in Chapter 5.

3.2.3.1 Regridding

Regridding is the process of changing the layout of child grids. This is performed in order to adapt the layout to the new features of the solution. Regridding is described in more detail in Section 5.1.

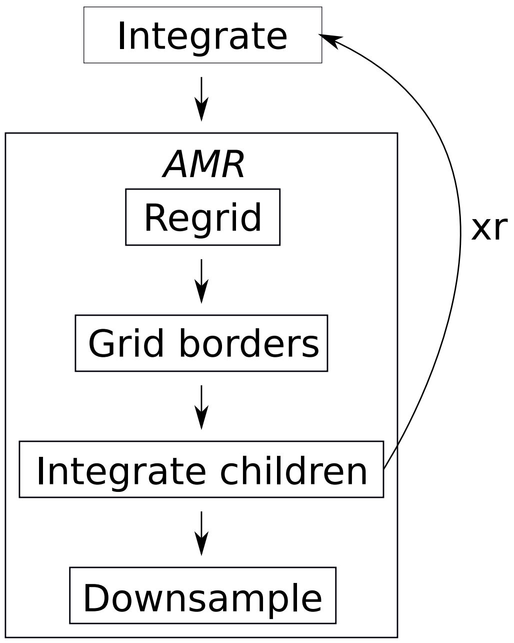


Figure 3.4: Overview of the AMR step and how its substeps fit together. ”xr” denotes ”run r times”.

3.2.3.2 Grid Borders

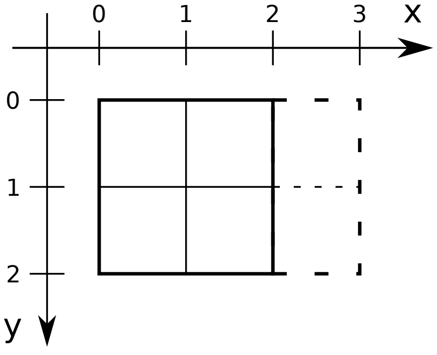
In this step we handle the borders of the child grids. We derive the ghost values of a child grid from interpolation of the values of points in its parent grid. The grid border step is described in more detail in Section 5.2.

3.2.3.3 Integrate Children

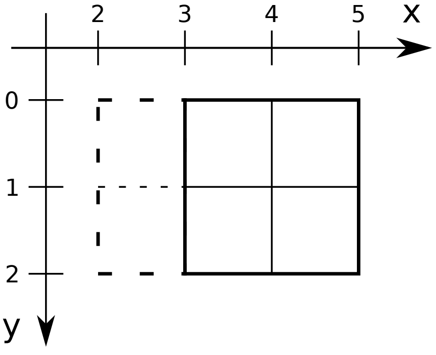
After the regridding and the handling of the grid border, a grid may have one or more child grids. In this step we recursively integrate each of these child grids in the same way as we did in Section 3.2.2. The child grids are more refined in both space and time than their parent, and therefore produce a more accurate solution.

When a child grid has finished the integration step, it continues to the AMR step and the regridding and grid borders substeps, as shown in Figure 3.4. After those substeps, the child grid may have one or more child grids itself. In that case those child grids are integrated themselves in the same way as their parent. This recursion continues to deepen until we reach grids which have no children themselves. At that point those grids continue to the next step. Each child grid runs r timesteps. After completing those timesteps the recursion is gradually nested back up, and the root grid eventually moves on to the next step of the application.

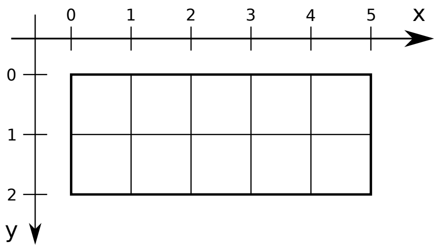
The integrate children step is described in more detail in Section 5.3.



(a) Left rank (solid line) with ghost points (dotted line)



(b) Right rank (solid line) with ghost points (dotted line)



(c) What the domain looks like to the application after the exchange of rank boundary values in Figure 3.5a and 3.5b.

Figure 3.5: Rank boundary: Figure 3.5a and 3.5b: Each rank sends its boundary values to the ghost cells of its neighbor rank. Figure 3.5c: This exchange of boundary values makes it possible to treat the domain as if it were continuous.

3.2.3.4 Downsampling

Downsampling is the process of replacing the solution in the parent grid with values from the child grid. The point of this is to achieve a more accurate solution in the parent grid. Downsampling is described in more detail in Section 5.4.

3.2.4 Rank Borders

The domain is divided into ranks, with each rank running on one distributed-memory node. Each rank sends its boundary values to the ghost points of its neighbor rank, as illustrated in Figure 3.5a and 3.5b. This exchange makes it possible to treat the domain as if it were continuous, as illustrated in Figure 3.5c.

For simplicity, we choose to only let the root grids handle the rank borders, and then let the child grids get their border values from their parents, as described in Section 5.2.

3.2.5 Full Overview

Figure 3.6 combines all the parts we have looked at into a full overview of all the parts of the application and how they fit together.

3.3 Parallel Programming APIs

In this section we describe which parallel programming APIs we choose to parallelize our application and why.

The SWE, FDMs and AMR work well on both CPUs and GPUs. We choose to implement our application on the CPU, but note that repeating our experiments on the GPU could of interest.

We choose to use MPI for parallelization and communication between distributed-memory nodes. MPI is a widely available industry standard for parallelization on distributed memory, and MPI implementations are highly optimized and tuned for specific machines.

We choose to use OpenMP for parallelization of operations on collections between processors in shared memory. pthreads gives the user a lot of control, but is also error-prone. OpenMP is implemented using pthreads, but hides the implementation details. This makes OpenMP easy to use and less error-prone.

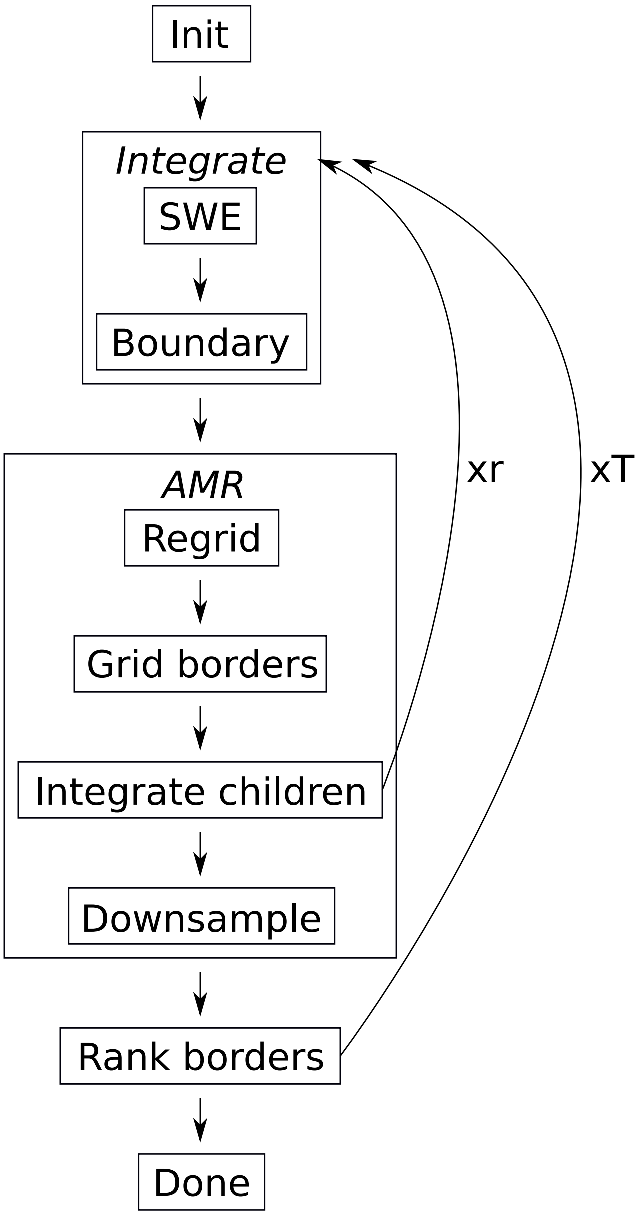


Figure 3.6: Full overview of all the parts of the application and how they fit together. ”xr” and ”xT” denote ”run r or T times”.

Chapter 4

Integration

The integration is the main step of our application. This is where we evolve the solution in time. In this chapter we look at the details of how the integration is implemented in our application. First we look at how we solve the SWE using an FDM. Then we look at how we handle the boundary conditions.

4.1 SWE

In this section we describe how we use a finite difference method (FDM) to implement the shallow water equations (SWE). First we rewrite the SWE into a form more suited for conversion to our difference equations. Then we describe the idea of one specific finite difference method and present its general equations. Finally we explain how this method can be used to implement the SWE.

4.1.1 SWE as relation between time and space difference

The shallow water equations from Section 2.1 can be rewritten to express the relation between the time difference and the space difference:

$$\frac{\partial(\rho\eta)}{\partial t} = -\frac{\partial(\rho\eta u)}{\partial x} - \frac{\partial(\rho\eta v)}{\partial y} \tag{4.1}$$

$$\frac{\partial(\rho\eta u)}{\partial t} = -\frac{\partial}{\partial x}\left(\rho\eta u^2 + \frac{1}{2}\rho g\eta^2\right) - \frac{\partial(\rho\eta uv)}{\partial y} \tag{4.2}$$

$$\frac{\partial(\rho\eta v)}{\partial t} = -\frac{\partial(\rho\eta uv)}{\partial x} - \frac{\partial}{\partial y}\left(\rho\eta v^2 + \frac{1}{2}\rho g\eta^2\right) \tag{4.3}$$

4.1.2 FDM

We choose the MacCormack method as our finite difference method. MacCormack is a predictor-corrector method which is well suited for non-linear hyperbolic partial differential equations [22].

The predictor step uses the forward Euler method [23] in time and a forward difference in space.

$$u_i^{\overline{n+1}} = u_i^n - \Delta t \left(\frac{f(u_{i+1}^n) - f(u_i^n)}{\Delta x} \right) \tag{4.4}$$

Where

- u: Value at a given coordinate at a given time
- Δa: Discrete step size in direction a

The corrector step is similar to the predictor step, but takes the average of the current time step and the predictor step, uses backward differences, and divides the spatial difference in half.

$$u_i^{n+1} = \frac{u_i^n + u_i^{\overline{n+1}}}{2} - \frac{\Delta t}{2} \left(\frac{f(u_i^{\overline{n+1}}) - f(u_{i-1}^{\overline{n+1}})}{\Delta x} \right) \tag{4.5}$$

4.1.3 Finite Difference Equations for the SWE

4.1.3.1 Fluid height

Predictor step:

To better explain the conversion from the shallow water differential equations to the finite difference equations, we note that Equation 4.4 can also be written as

$$\frac{u_i^{\overline{n+1}} - u_i^n}{\Delta t} = -\frac{f(u_{i+1}^n) - f(u_i^n)}{\Delta x} \tag{4.6}$$

By applying Equation 4.6 to Equation 4.1 we get a difference equation describing the fluid column height at the predictor step:

$$\frac{\rho\eta^{\overline{t+1}} - \rho\eta^t}{\Delta t} = -\frac{\rho\eta u_{x+1} - \rho\eta u_x}{\Delta x} + \frac{\rho\eta v_{y+1} - \rho\eta v_y}{\Delta y} \tag{4.7}$$

We rewrite the equation to express the predictor time step as the difference between the current time step and forward difference in space:

$$\rho\eta^{\overline{t+1}} = \rho\eta^t - \Delta t \left(\frac{\rho\eta u_{x+1} - \rho\eta u_x}{\Delta x} + \frac{\rho\eta v_{y+1} - \rho\eta v_y}{\Delta y} \right) \tag{4.8}$$

Corrector step:

By applying Equation 4.5 to Equation 4.1 in the equivalent way to what we did for the predictor step, we get a difference equation describing the fluid column height at the corrector step:

$$\rho\eta^{t+1} = \frac{\rho\eta^{t} + \rho\eta^{\overline{t+1}}}{2} - \frac{\Delta t}{2}\left(\frac{\rho\eta u_{x}^{\overline{t+1}} - \rho\eta u_{x-1}^{\overline{t+1}}}{\Delta x} + \frac{\rho\eta v_{y}^{\overline{t+1}} - \rho\eta v_{y-1}^{\overline{t+1}}}{\Delta y}\right) \tag{4.9}$$

4.1.3.2 Velocity in x-direction

By applying Equation 4.4 and 4.5 to Equation 4.2 in the equivalent way to Section 4.1.3.1, we get a set of difference equations describing the predictor and the corrector step for the velocity in the x-direction:

Predictor step:

$$\rho\eta u^{\overline{t+1}} = \rho\eta u^{t} - \Delta t\left(\frac{du_{y,x+1} - du_{y,x}}{\Delta x} + \frac{\rho\eta uv_{y+1,x} - \rho\eta uv_{y,x}}{\Delta y}\right) \tag{4.10}$$

Where $du_{y,x}$ is shorthand notation for

$$du_{y,x} = \rho\eta u_{y,x}^{2} + \frac{1}{2}\rho g\eta_{y,x}^{2} \tag{4.11}$$

Corrector step:

$$\rho\eta u^{t+1} = \frac{\rho\eta u^{t} + \rho\eta u^{\overline{t+1}}}{2} - \frac{\Delta t}{2}\left(\frac{du_{y,x} - du_{y,x-1}}{\Delta x} + \frac{\rho\eta uv_{y,x} - \rho\eta uv_{y-1,x}}{\Delta y}\right) \tag{4.12}$$

4.1.3.3 Velocity in y-direction

By applying Equation 4.4 and 4.5 to Equation 4.3 in the equivalent way to Section 4.1.3.1 we get a set of difference equations describing the predictor and the corrector step for the velocity in the y-direction:

Predictor step:

$$\rho\eta v^{\overline{t+1}} = \rho\eta v^{t} - \Delta t\left(\frac{dv_{y+1,x} - dv_{y,x}}{\Delta y} + \frac{\rho\eta uv_{y,x+1} - \rho\eta uv_{y,x}}{\Delta x}\right) \tag{4.13}$$

Where $dv_{y,x}$ is shorthand notation for

$$dv_{y,x} = \rho\eta v_{y,x}^{2} + \frac{1}{2}\rho g\eta_{y,x}^{2} \tag{4.14}$$

Corrector step:

$$\rho\eta v^{t+1} = \frac{\rho\eta v^{t} + \rho\eta v^{\overline{t+1}}}{2} - \frac{\Delta t}{2}\left(\frac{dv_{y,x} - dv_{y-1,x}}{\Delta y} + \frac{\rho\eta uv_{y,x} - \rho\eta uv_{y,x-1}}{\Delta x}\right) \tag{4.15}$$

4.2 Boundary Conditions

We choose to handle the boundary conditions by reflecting the fluid height and velocity. This can be implemented using the Neumann boundary condition [24]. We achieve this effect by specifying that the derivative of the function should be 0.

$$\frac{\partial A}{\partial n} = 0 \tag{4.16}$$

Where

- A: The value we want to find, in our case the fluid height and velocity
- n: The normal to the boundary over which the change takes place, in our case the spatial dimensions X and Y in which the components of the changes in the fluid are moving

In a finite difference method, this can be implemented using ghost points. By setting the values of the ghost points to be equal to the values of the points inside the domain which are next to the boundary points, we ensure that the derivatives of the values of the boundary points are 0. Thus, a reflection effect is created.

Chapter 5

Adaptive Mesh Refinement

In this chapter we look at Adaptive Mesh Refinement (AMR) and its different parts in more detail.

5.1 Regridding

Regridding is the process of changing the layout of child grids. This is performed regularly after a given number of timesteps, in order to adapt the layout to the new features of the solution. It may involve adding, removing, changing or keeping existing child grids.

Our regridding algorithm can be summarized as follows:

1. Flag the points which require refinement.
2. Cluster the flagged points into rectangles. For this purpose we use the algorithm of Berger and Rigoutsos [25]. A brief overview of the algorithm is given below.
3. Create new child grids defined by the areas covered by the rectangles. Let these new child grids replace the old child grids.

In short, the algorithm of Berger and Rigoutsos checks if a large enough share of the points in a given rectangle require refinement. If this share is above the requirement, the rectangle is accepted and a new child grid is created based on it. Otherwise, the rectangle is split in such a way that neighboring points end up in the same rectangle to the largest extent possible, and the algorithm is performed recursively on each new rectangle. A rectangle is also accepted if the algorithm is unable to find a way split it, but it contains at least one point which requires refinement.

The major goal of our algorithm, and of AMR in general, is to refine as few unnecessary points as possible. Thus, our rectangles should contain as few points which do not require refinement as possible. This can be achieved by increasing the share of flagged points required to accept a rectangle, which leads to a larger number of and smaller rectangles, covering a smaller area in total. However, a large amount of small rectangles also

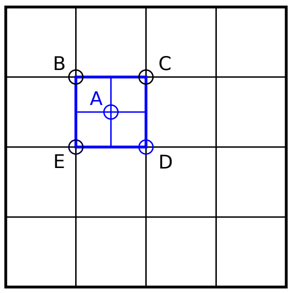


Figure 5.1: The value of point A in the blue child grid is derived from spacial interpolation of points B, C, D and E in its black parent grid.

leads to a longer cumulative boundary, which increases the overhead from handling grid borders. Thus, in order to get the best performance we should try to strike a balance between refining as few unnecessary points as possible, and having as few rectangles as possible.

The values of a new child grid are derived from spatial interpolation of points in its parent. This is done by taking a weighted linear average of the values of the closest points in the parent, as illustrated in Figure 5.1.

5.2 Grid Borders

Figure 5.2 illustrates how child grids handle their borders. The ghost values of a child grid are derived from space-time interpolation of points in its parent. This is done by taking a weighted linear average of the values of the closest points in space and time in the parent. The ghost values are then used to find the new values of the border points through normal time integration.

For simplicity of implementation, we choose to let all child grids get their grid border values from their parents, instead of letting those child grids who have a neighbor child grid get their grid border values from that neighbor.

5.3 Integrate Children

Figure 5.3 shows how the integrate children step recursively evolves the solution in time for different levels of refinement. $t_{l,i}$ is the time at the beginning of timestep number i in the reference system of a grid at level l. Each timestep evolves the solution from time $t_{l,i}$ to $t_{l,i+1}$. The algorithm first makes a timestep in the root grid, at level 0, from time $t_{0,0}$ to $t_{0,1}$. The root grid now has the knowledge to interpolate the boundaries of its child grids. Each child grid has a refinement in time which is r times that of its parent. Thus, each child has to perform r timesteps in its own reference system, each of size $\frac{1}{r}$ times the size of a timestep in the reference system of its parent, in order to get to the same point in time as its parent. But even after just one timestep, a child grid at level 1 has the knowledge to interpolate the boundaries of its child grids at level 2, and the algorithm can recursively

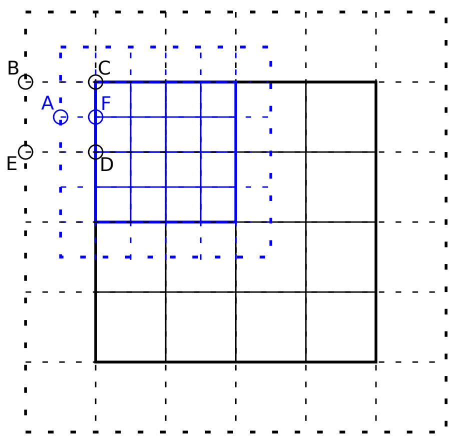


Figure 5.2: The ghost value A in the blue child grid is derived from space-time interpolation of points B, C, D and E in its black parent. The ghost value A can then be used to find the new value of boundary point F in the child grid.

evolve the solution r timesteps at level 2, then take another timestep at level 1, and finally r new timesteps at level 2. At this point, all grids will recursively have evolved up to the end of the first timestep at the root level, and the root grid can proceed with the next timestep.

5.4 Downsampling

Downsampling is the process of replacing the solution in the parent grid with values from the child grid. This is done to achieve a more accurate solution in the parent grid. Furthermore, if a child grid disappears during regridding, the parent grid will already contain the more accurate solution based on the child grid, and so no additional work is required for the parent grid during regridding.

Figure 5.4 illustrates how downsampling is performed. The values of the points in the parent grid which overlap with the child grid are replaced with values from overlapping points in the child grid. For instance, in Figure 5.4 the values of point D and E in the parent grid are replaced by the values of point D and E in the child grid. The point in the child grid between D and E, however, is not used to update the parent grid directly. Such points are only used internally in the child grid to find better values of D and E than the parent grid could do on its own.

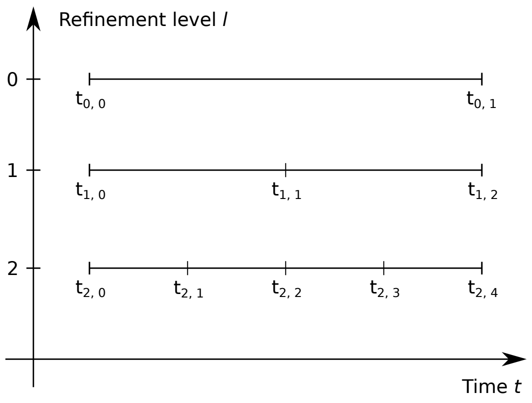


Figure 5.3: Recursive evolution of the solution in time for different levels of refinement. $t_{l,i}$ is the time at the beginning of timestep number i in the reference system of a grid at level l. In this example the refinement in time between each level is 2.

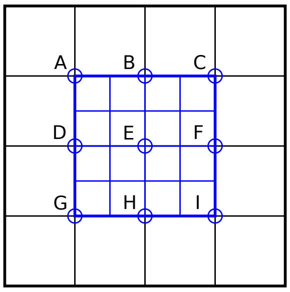


Figure 5.4: Downsampling: The values of points A-I in the black parent grid are replaced with values from points A-I in the blue child grid.

Chapter 6

Performance Modeling

In this chapter we derive our performance model. First we model the runtime of each part of our application on a single node. Then we combine the different parts into one model and describe how the model is affected by scaling the number of nodes. Finally we make predictions about the performance and scalability of our application.

6.1 Parts

In this section we model the performance of each part of our application on a single node in a given timestep. Table 6.1 provides an overview of some important properties of each part. The relevant hardware parameters for the different parts are

- $FLOP/s$: Floating point-operations per second
- M_{BW}: Memory bandwidth
- M_A: Memory allocation performance
- $comm_{local}$: Local communication, that is, point-to-point communication between nodes

6.1.1 Integration

6.1.1.1 SWE

The runtime of the SWE step is modeled as

$$T_{SWE} = P_{SWE} \times W_{SWE} \tag{6.1}$$

Where

- P_{SWE}: Number of points in the SWE step

- W_{SWE}: How fast a point is processed in the SWE step

The number of points in the SWE step is given as

$$P_{SWE} = P = \sum_{g}^{G} X_g \times Y_g \tag{6.2}$$

Where

- P: Total number of points
- G: All grids at all levels at all timesteps
- X_g, Y_g: Length in direction in grid g

Relevant Hardware Parameters
The SWE step in our application solves the equations given in Section 4.1.3 by applying floating-point operations to data saved in shared memory. Thus, we argue the relevant hardware parameters for this step are $FLOP/s$ and M_{BW}. Both of these are determined by characteristics of a node class.

6.1.1.2 Boundary Conditions

The runtime of handling the boundary conditions is modeled as

$$T_B = P_B \times W_B \tag{6.3}$$

Where

- P_B: Number of points in the handling of the boundary condition
- W_B: How fast a point is processed in the handling of the boundary condition

The number of points in the handling of the boundary conditions is given as

$$P_B = 2(X + Y) \tag{6.4}$$

Where

- X, Y: Length of domain in direction

Relevant Hardware Parameters
We handle the boundary conditions by applying floating-point operations to data saved in shared memory. Thus, we argue the relevant hardware parameters for this step are $FLOP/s$ and M_{BW}. Both of these are determined by characteristics of a node class.

6.1.2 AMR

6.1.2.1 Regridding

The runtime of regridding is modeled as

$$T_R = T_{RA} + T_{AR} \tag{6.5}$$

Where

- T_{RA}: Runtime of the regridding algorithm itself, where we figure out how the child grids should be regridded
- T_{AR}: Runtime of applying the regridding specified by the regridding algorithm to the child grids

Regridding Algorithm

The regridding algorithm takes a grid with a number of points, looks at which points require refinement, and figures out how to organize those points into child grids.

We model the runtime of the regridding algorithm as

$$T_{RA} = P_{RA} \times W_{RA} \tag{6.6}$$

Where

- P_{RA}: Number of points which go into the regridding algorithm
- W_{RA}: How fast a point is processed in the regridding algorithm

The number of points which go into the regridding algorithm is given as as

$$P_{RA} = \frac{p_{parent}}{RI} = \frac{\sum_{g}^{G_{parent}} X_g \times Y_g}{RI} \tag{6.7}$$

Where

- p_{parent}: Total number of points in parents
- G_{parent}: All parent grids at all levels at all timesteps
- RI: Regridding interval, number of timesteps between each regridding

Dividing by the regridding interval gives an amortized cost of the regridding algorithm for a given timestep.

Regridding Algorithm: Relevant Hardware Parameters

The regridding algorithm applies floating-point operations to data saved in shared memory. Thus, we argue the relevant hardware parameters for this step are $FLOP/s$ and M_{BW}. Both of these are determined by characteristics of a node class.

Apply Regridding

The applying of the regridding consists of taking the result from the regridding algorithm and applying them to the child grids of the input grid. It may involve adding, removing, changing or keeping existing child grids.

The runtime of the applying of the regridding is modeled as

$$T_{AR} = P_{AR} \times W_{AR} \tag{6.8}$$

Where

- P_{AR}: Number of points in the applying of the regridding
- W_{AR}: How fast a point is processed in the applying of the regridding

The number of points in the applying of the regridding is modeled as

$$P_{AR} = \frac{p_{child}}{RI} = \frac{\sum_{g}^{G_{child}} X_g \times Y_g}{RI} \tag{6.9}$$

Where

- p_{child}: Total number of points in children
- G_{child}: All child grids at all levels at all timesteps

Apply Regridding: Relevant Hardware Parameters
We apply the regridding by allocating and reallocating memory, as well as setting that memory by applying floating-point operations on data saved in shared memory. Based on this we argue that the relevant hardware parameters for this step are $FLOP/s$, M_{BW} and M_A. All of these are determined by characteristics of a node class.

6.1.2.2 Grid Borders

The runtime of handling the grid borders is modeled as

$$T_{GB} = P_{GB} \times W_{GB} \tag{6.10}$$

Where

- P_{GB}: Number of points in the handling of the grid borders
- W_{GB}: How fast a point is processed in the handling of the grid borders

The number of points in the handling of the grid borders is given as

$$P_{GB} = \sum_{g}^{G} 2(X_g + Y_g) \tag{6.11}$$

Relevant Hardware Parameters
We handle the grid borders by applying floating-point operations to data saved in memory. Thus, we argue the relevant hardware parameters for this step are $FLOP/s$ and M_{BW}. Both of these are determined by characteristics of a node class.

6.1.2.3 Integrate Children

The integration of child grids is part of the AMR algorithm. However, in our performance model we choose to model the integration of all grids, both parent grids and root grids, under one common integration part. The performance model for the integration part is described in Section 6.1.1.

6.1.2.4 Downsampling

The runtime of downsampling is modeled as

$$T_D = P_D \times W_D \tag{6.12}$$

Where

- P_D: Number of points in the downsampling
- W_D: How fast a point is processed in the downsampling

The number of points in the downsampling is given as

$$P_D = \frac{p_{child}}{r^3} = \frac{\sum_g^{G_{child}} X_g \times Y_g}{r^3} \tag{6.13}$$

Relevant Hardware Parameters

We handle the downsampling by applying floating-point operations to data saved in memory. Thus, we argue the relevant hardware parameters for this step are $FLOP/s$ and M_{BW}. Both of these are determined by characteristics of a node class.

6.1.3 Rank Borders

The runtime of handling the rank borders is modeled as

$$T_{RB} = P_{RB} \times W_{RB} \tag{6.14}$$

Where

- T_{RB}: Runtime of handling the rank borders
- P_{RB}: Number of points at the rank borders
- W_{RB}: How fast a point is processed in the handling of the rank borders

6.1.3.1 Relevant Hardware Parameters

We handle the rank borders by exchanging values at the rank borders with neighbor ranks. Thus, we argue the relevant hardware parameter for this step is $comm_{local}$.

6.1.4 Overview

Table 6.1 provides an overview of the different parts, equations for their number of points, and their relevant hardware parameters. We note that 5 out of 9 parts have both $FLOP/s$ and M_{BW} as relevant hardware parameters, and 4 parts have those as their only relevant hardware parameters. Thus, if we wanted to model how fast a point is processed in each part, Roofline [17] could be a good starting point, as it depends on precisely those parameters.

Part	**Number of points**	**Hardware parameters**
SWE	$P_{SWE} = P = \sum_g^G X_g \times Y_g$	FLOP/s, M_{BW}
Boundary conditions (B)	$P_B = 2(X+Y)$	FLOP/s, M_{BW}
Regridding algorithm (RA)	$P_{RA} = \frac{p_{parent}}{RI} = \frac{\sum_g^{G_{parent}} X_g \times Y_g}{RI}$	FLOP/s, M_{BW}
Applying of regridding (AR)	$P_{AR} = \frac{p_{child}}{RI} = \frac{\sum_g^{G_{child}} X_g \times Y_g}{RI}$	FLOP/S, M_{BW}, M_A
Grid borders (GB)	$P_{GB} = \sum_g^G 2(X_g + Y_g)$	FLOP/S, M_{BW}
Downsampling (D)	$P_D = \frac{p_{child}}{r^3} = \frac{\sum_g^{G_{child}} X_g \times Y_g}{r^3}$	M_{BW}
Rank borders (RB)		$comm_{local}$

Table 6.1: Overview of the different parts, equations for their number of points, and their relevant hardware parameters

6.2 Comparison of Parts

In this section we compare the different parts in Section 6.1 with each other, and identify which of them are likely to have a large impact on the total runtime.

6.2.1 Sum of Length and Width of Boundaries and Borders vs Product

Both P_{GB} and P_{SWE} depend on the length and width of each grid. However, P_{GB} depends on the sum of the length and width, while P_{SWE} depends on the product. We assume that no sides have a very short length, for instance 1. Then we expect that $P_{SWE} \gg P_{GB}$, and therefore that $T_{SWE} \gg T_{GB}$.

P_B depends on the sum of the length and width of just the domain. Thus, we expect that $P_{GB} > P_B$, and therefore that $P_{SWE} \gg P_B$ and $T_{SWE} \gg T_B$.

6.2.2 Number of Child Points vs Total Number of Points

P_{AR} and P_D both depend on the number of child points p_{child}, while P_{SWE} depends on the total number of points P. The relation between P and p_{child} is

$$P = p_{root} + p_{child} \tag{6.15}$$

Where

- p_{root}: Total number of points in the root grids

which means that $p_{child} < P$. p_{child} is also limited by the share of points which are refined. The number of points at a level l is given as

$$p_l = req_{l-1} \times p_{l-1} \times r^3 \tag{6.16}$$

Where req_{l-1} is the share of points at level $l-1$ which are refined. p_{child} is then given as

$$p_{child} = \sum_{l}^{L} p_l \tag{6.17}$$

If $req_{l-1} > \frac{1}{r^3}$ for all levels then p_{child} will grow exponentially from level to level, and $p_{child} \approx P$. However, the number of levels L is limited by factors such as floating-point precision, the granularity of the physical system, and user choices about requirements. Thus, there are multiple factors limiting p_{child}, and therefore P_{AR} and P_D.

In addition, applying of regridding is only run every RI timestep, while downsampling is only run for every r^3 point. By comparison, the SWE step is run for every point for all timesteps. In total, this makes us claim that both P_{AR} and P_D are small compared to P_{SWE}, which in turn makes us claim that T_{AR} and T_D are small compared to T_{SWE}.

6.2.3 Regridding Interval vs Every Timestep

The applying of regridding is only run every RI timestep. This is also true for the regridding algorithm. In addition, P_{RA} depends on P_P. The relation between P and P_{parent} is

$$P = p_{parent} + p_{leaf} \tag{6.18}$$

Where p_{leaf} is the number of leaf child points, child points which do not themselves have children. This means that $P \geq p_{parent}$. AMR applications are useful in cases where at least some of the points are refined. In that case, p_{parent} will be smaller than P.

In total, these factors make us claim that T_{RA} is small compared to T_{SWE}.

6.2.4 Communication vs Computation

T_{RB} depends on the performance of local communication between neighbor nodes. All the other parts depend on resources that are characteristics of node type, namely $FLOP/s$, M_{BW} and M_A. This makes the scalability of T_{RB} fundamentally different to the other parts.

As we increase the number of nodes, the number of points of each of the other parts may be divided among the nodes, and each node may get less work to do for each of these parts.

The rank border part, on the other hand, will have no work for 1 node. As we increase the number of nodes, a given rank will get more neighbors to communicate with during border exchange, until it has 4 neighbors, one on each side. From that point, even as we continue to scale up the number of nodes, and thus the total amount of communication, the amount of communication per node will begin to decrease due to the shorter rank border. P_{RB} depends on the sum of the sides of each rank, while P_{SWE} depends on the product of the sides of the rank in addition to the product of the sides of each child grid within a rank. Thus, we expect that $P \gg P_{RB}$.

We recognize that T_{RB} is part of the total runtime, but due to the fundamentally different scalability properties of T_{RB}, as well as its dependence on a different class of hardware parameters, namely parameters between nodes, a detailed model of T_{RB} is beyond the scope of this thesis.

6.2.5 Summary

We claim the runtime of the other parts are small compared to the runtime of the SWE part. All parts depend on hardware parameters which are characterized by the node type, except the handling of the rank borders, which depends on the performance of local communication between nodes. The way the handling of the rank borders scales with the number of nodes is more complicated than for the other parts.

6.3 Load Balance

The load balance measures how well work is distributed. The load balance is perfect if all compute units, for instance processor cores or nodes, have work which take the same time. The load balance is lower than perfect if one or a few compute units have work which takes a long time, while the other compute units have to wait while doing nothing.

Our application consists of a number of timesteps. A given node can not start with the next timestep until it has exchanged border values with its neighbor nodes. If one of its neighbors has a lot of work to do, the node will have to wait as long as if it had the same amount of work to do as its neighbor. For this reason we define the load balance LB as

$$LB = \frac{P}{P_{LB}} \tag{6.19}$$

Where

- P: Total number of points for all timesteps for all nodes
- P_{LB}: Total number of points for all timesteps for all nodes if all nodes had the same number of points in a given timestep as the node with the largest number of points in that timestep

P_{LB} can be expressed and measured as

$$P_{LB} = \sum_{i}^{I} \max_{n}^{N}(P_n) \times N \tag{6.20}$$

Where

- n: Node number
- N: Number of nodes
- P_n: Number of points for node number n

This is similar to BSP [16], which models the cost of the local computation for a given superstep as

$$\max_n^N(w_n) \tag{6.21}$$

P_{LB} can not be smaller than P. Thus, the load balance will be a number between 0 and 1, with 1 indicating a perfect load balance, and a number close to 0 indicating a very bad load balance, strongly influencing the runtime.

6.4 Performance Model

In Section 6.2 we argued that the runtime of the other parts likely is small compared to the runtime of the SWE part. We also argued that the handling of the rank borders was the only part which depended on communication instead of resources internal to a node, and that the handling of the rank border did not scale with the number of nodes in the same way that the other parts did. In Section 6.3 we argued that the load balance impacts the scalability of our application.

In this section we use these arguments to build a model for the runtime of our application. We model the runtime of our application as

$$T_{tot} = T_{comp} + T_{comm} \tag{6.22}$$

Where

- T_{tot}: Total runtime
- T_{comp}: Runtime of the computation
- T_{comm}: Runtime of the communication between ranks

The runtime of the computation is modeled as

$$T_{comp} = \frac{T_{SWE}}{LB \times N} \tag{6.23}$$

This can be expanded as

$$T_{comp} = \frac{P \times W_{SWE}}{LB \times N} \tag{6.24}$$

The runtime of the computation is simply modeled as

$$T_{comm} = T_{RB} \tag{6.25}$$

This means our model can be written as

$$T_{tot} = \frac{P \times W_{SWE}}{LB \times N} + T_{RB} \tag{6.26}$$

6.5 Predictions

In this section we make predictions based on our performance model.

6.5.1 Cost of an Extra Level of Refinement

In this section we show that the runtime of our application will be between 1 and $r^3 + 1$ times larger if we add an extra level of refinement rather than keeping the current number of levels of refinement.

First we show that the most refined level has between 1 and r^3 times as many points as the second most refined level. Then we show that the total number of points will be between 1 and $r^3 + 1$ times larger if we add an extra level of refinement. Finally we show that this increase in the number of points will lead to the same increase in the runtime of our application.

6.5.1.1 Cost of the Most Refined Level in Number of Points

Let

$$A_{l,g,ref} = Y_{l,g,ref} \times X_{l,g,ref} \tag{6.27}$$

Where

- $A_{l,g,ref}$: Area of grid g at level l in reference system ref
- $Y_{l,g,ref}$, $X_{l,g,ref}$: Area of grid g in direction Y or X at level l in reference system ref

Note that this definition of area is focused on the number of points in a grid at a given timestep, and not necessarily on the space a grid takes up in the global reference system, since our focus is on the number of points, which depends on the number of points in each grid, as well as the number of timesteps.

Then the area of a parent grid in its own reference system is larger than or equal to the sum of the areas of its children in the parent's reference system:

$$\sum_{g}^{G_{parent}} A_{L,g,parent} \leq A_{L-1,parent,parent} \tag{6.28}$$

This is because the children of a grid may refine all or some of the grid's areas, but not more, as they are bound by the borders of their parent.

Assuming that the refinement r is the same in both spatial directions, the area of a child grid is r^2 times larger in the child's own reference system than in the reference system of its parent. From this it follows that the sum of the areas of the children in their own reference system is up to r^2 times larger the area of their parent in its own reference system:

$$\frac{\sum_{g}^{G_{parent}} A_{L,g,self}}{A_{L-1,parent,self}} \leq r^2 \tag{6.29}$$

The number of points for all timesteps is defined by the product of the number of steps in the temporal and both of the spatial dimensions. Assuming that the refinement is the same in the temporal dimension as in both spatial dimensions, the number of timesteps of a child is up to r times that of its parent. Combining this with Equation 6.29 we find that the sum of the number of points for each child of a grid is up to r^3 times larger than the number of points of the parent:

$$\frac{\sum_g^{G_{parent}} p_{L,g}}{p_{L-1,parent}} \leq r^3 \tag{6.30}$$

Where

- $p_{l,g}$: Number of points in a grid g at level l

Since this holds for every grid at every level, the relation can be rewritten for the total number of points for all grids at a given level:

$$\frac{p_L}{p_{L-1}} \leq r^3 \tag{6.31}$$

Where

- p_l: Number of points at level l

Thus we have shown that the most refined level has up to r^3 more points than the second most refined level.

6.5.1.2 Total Cost of an Extra Level of Refinement in Number of Points

The total number of points for an application with L levels is

$$P_L = \sum_l^L p_l \tag{6.32}$$

Equation 6.31 can be rewritten as

$$p_L \leq r^3 p_{L-1} \tag{6.33}$$

This recursive relation can be expanded to

$$p_L \leq r^3(r^3 p_{L-2}) \tag{6.34}$$

and simplified as

$$p_L \leq r^{3\times2} p_{L-2}) \tag{6.35}$$

Fully expanded this becomes

$$p_L \leq r^{3L} p_{root} \tag{6.36}$$

Combining this with Equation 6.32 gives

$$P_L \leq p_{root} \sum_{l}^{L} r^{3L} \tag{6.37}$$

This is a geometric series which can be written as

$$P_L \leq p_{root} \frac{1 - r^{3L}}{1 - r^3} \tag{6.38}$$

Adding an extra level of refinement yields

$$\frac{P_L}{P_{L-1}} \tag{6.39}$$

times more points than by keeping the maximum refinement level at $L - 1$.
Inserting Equation 6.32 into Equation 6.39 gives

$$\frac{\sum_l^L p_l}{\sum_l^{L-1} p_l} \tag{6.40}$$

which can be written as

$$\frac{\sum_l^{L-1} p_l + p_L}{\sum_l^{L-1} p_l} \tag{6.41}$$

If the area and number of timesteps refined at level L is low, p_L will go towards 0, and adding an extra level of refinement will yield

$$\frac{\sum_l^{L-1} p_l}{\sum_l^{L-1} p_l} = 1 \tag{6.42}$$

times more points than by keeping the maximum refinement level at $L - 1$, making this the lower bound on the cost of adding an extra level of refinement. Conversely, if the area and number of timesteps refined at level L is high, p_L will tend towards r^3. The upper limit on the cost of adding an extra level of refinement can then be found as follows:
Inserting Equation 6.38 into Equation 6.39 gives

$$\frac{p_{root} \frac{1-r^{3L}}{1-r^3}}{p_{root} \frac{1-r^{3(L-1)}}{1-r^3}} \tag{6.43}$$

This can be simplified as

$$\frac{1 - r^{3L}}{1 - r^{3(L-1)}} \tag{6.44}$$

For the smallest case with only two levels, this becomes

$$\frac{1 - r^{3\times 2}}{1 - r^3} \tag{6.45}$$

which simplifies to

$$1 + r^3 \tag{6.46}$$

In the more general case we have that

$$\lim_{L\to\infty} \frac{1 - r^{3L}}{1 - r^{3(L-1)}} = \lim_{r\to\infty} \frac{1 - r^{3L}}{1 - r^{3(L-1)}} = \frac{r^{3L}}{r^{3(L-1)}} \tag{6.47}$$

which simplifies to

$$\lim_{L\to\infty} \frac{1 - r^{3L}}{1 - r^{3(L-1)}} = \lim_{r\to\infty} \frac{1 - r^{3L}}{1 - r^{3(L-1)}} = r^3 \tag{6.48}$$

To summarize the upper and lower bound:

$$1 \leq P_{cost} \leq r^3 + 1 \tag{6.49}$$

for $L \to 2 \wedge r \to 0$, and

$$1 \leq P_{cost} \leq r^3 \tag{6.50}$$

for $L \to \infty \vee r \to \infty$.
Where

- P_{cost}: Cost of an extra level of refinement measured as the number of points after adding an extra level of refinement relative to keeping the number of levels unchanged.

This means that if we add an extra level of refinement, the number of points will increase by a factor

$$1 \leq P_{cost} \leq r^3 + 1 \tag{6.51}$$

for all L and r.

6.5.1.3 Total Cost of an Extra Level of Refinement in Runtime

In this section we examine the effects of an extra level of refinement on the runtime.
Equation 6.24 gives the runtime of the computation as

$$T_{comp} = \frac{P \times W_{SWE}}{LB \times N} \tag{6.52}$$

If we keep N constant, then increasing P from P_A to P_B will increase T_{comp} by

$$T_{comp,cost} = \frac{T_{comp,B}}{T_{comp,A}} = \frac{\frac{P_B \times W_{SWE}}{LB_B \times N}}{\frac{P_A \times W_{SWE}}{LB_A \times N}} \tag{6.53}$$

Where $T_{comp,cost}$ is the cost of adding an extra level of refinement, measured in the runtime of the computation.
In the general case, this can be simplified as

$$T_{comp,cost} = \frac{T_{comp,B}}{T_{comp,A}} = \frac{P_B \times LB_A}{P_A \times LB_B} \tag{6.54}$$

If $LB_A = LB_B$, this can be further simplified as

$$T_{comp,cost} = \frac{T_{comp,B}}{T_{comp,A}} = \frac{P_B}{P_A} \tag{6.55}$$

If A is the case before we add an extra level of refinement, and B is the case after, then $P_B = P_A \times P_{cost}$. In that case we have that

$$T_{comp,cost} = \frac{T_{comp,B}}{T_{comp,A}} = P_{cost} \tag{6.56}$$

Since $1 \leq P_{cost} \leq 1 + r^3$, we have that

$$1 \leq T_{comp,cost} \leq 1 + r^3 \tag{6.57}$$

Since N is unchanged, T_{comm} will also be unchanged.

6.5.1.4 Summary

We predict that if the load balance is unchanged, the runtime of our application will be between 1 and $r^3 + 1$ times larger, in addition to some constant communication overhead, if we add an extra level of refinement rather than keeping the current number of levels of refinement. Furthermore, the increase in the runtime will be equal to the increase in the number of points.

6.5.2 Strong Scaling

In this section we predict what happens as we scale up the number of nodes while keeping the problem size constant. This is known as strong scaling.

Equation 6.24 gives the runtime of the computation as

$$T_{comp} = \frac{P \times W_{SWE}}{LB \times N} \tag{6.58}$$

If LB is perfect for all N, and we keep P constant, then increasing N from N_A to N_B will give T_{comp} a speedup of

$$S_{comp} = \frac{T_{comp,A}}{T_{comp,B}} = \frac{\frac{P \times W_{SWE}}{LB \times N_A}}{\frac{P \times W_{SWE}}{LB \times N_B}} \tag{6.59}$$

Where

- S_{comp}: Speedup of the computation
- $T_{comp,A}$, $T_{comp,B}$: Runtime of computation for configurations A and B
- N_A, N_B: Number of nodes for configurations A and B

This can be simplified as

$$S_{comp} = \frac{T_{comp,A}}{T_{comp,B}} = \frac{N_B}{N_A} \tag{6.60}$$

If LB is not perfect for all N, and we keep P constant, then increasing N from N_A to N_B will give T_{comp} a speedup of

$$S_{comp} = \frac{T_{comp,A}}{T_{comp,B}} = \frac{\frac{P \times W_{SWE}}{LB_A \times N_A}}{\frac{P \times W_{SWE}}{LB_B \times N_B}} \tag{6.61}$$

Where LB_A and LB_B are the load balance for configurations A and B. This can be simplified as

$$S_{comp} = \frac{T_{comp,A}}{T_{comp,B}} = \frac{LB_B \times N_B}{LB_A \times N_A} \tag{6.62}$$

T_{comm} will be greater than 0 for $N > 1$. However, due to the communication being local, we expect T_{comm} to stabilize as N increases enough to make some nodes reach their maximum number of neighbors.

6.5.2.1 Summary

We expect the speedup to increase with the number of nodes. However, an increase in N may lead to a decrease in LB, which may dampen the speedup. T_{comm} will stabilize as N increases enough to make some nodes reach their maximum number of neighbors.

6.5.3 Weak Scaling

In this section we predict what happens to the runtime as we increase the problem size and the resources by the same factor. This is known as weak scaling.

We increase the problem size by increasing P, and increase the resources by increasing N. Equation 6.24 gives the runtime of the computation as

$$T_{comp} = \frac{P \times W_{SWE}}{LB \times N} \tag{6.63}$$

If we increase P and N by the same factor, and LB remains unchanged, the runtime of the computation internal to each node will be the same, and thus T_{comp} will be the same, and the parallel efficiency E will be stable. If LB decreases when N increases, T_{comp} will increase, and E will decrease.

Just as for the strong scaling, we expect T_{comm} to stabilize as N increases enough to make some nodes reach their maximum number of neighbors.

6.5.3.1 Summary

We expect that there will be some communication overhead for $N > 1$, but that it will stabilize as N increases. If LB remains unchanged, we expect T_{tot} and E to stabilize, so that we can increase N and get more work done in the same time.

Chapter 7

Experimental Setup

In this chapter we describe the setup for our experiments. First we describe the setup of the machines we run our experiments on. Then we describe the parameters which affect the performance and scalability of the solution. Finally we describe the setup for validation of our predictions.

7.1 Machine Setup

The experiments are performed on two machines, Idun and Vilje. On both machines we assign 1 node to each MPI rank, and 1 processor core to each OpenMP thread.

7.1.1 Idun

Idun is a research cluster at NTNU which serves as a platform for rapid testing and prototyping of HPC software [26]. Table 7.1 summarizes some important properties of the hardware and software of Idun. Two of the node types of the machine are the Dell PE630 and the Dell PEC6420. We run our experiments on one of these node types at a time. We set the thread affinity to assign OpenMP thread n+1 to processor core n by setting the `KMP_AFFINITY` environment variable to `compact`.

7.1.2 Vilje

Vilje is a SGI Altix ICE X system procured by NTNU together with met.no and UNINETT Sigma [27]. Table 7.2 summarizes some important properties of the hardware and software of Vilje. We set the thread affinity to assign OpenMP thread n+1 to processor core n by wrapping our application command with the `OMPLACE` command.

Property	Value
Compiler	ICC 19.0.5.281
Compiler flags	-std=c99 -g -O3 -qopenmp -lm -liomp5
MPI	Intel(R) MPI Library for Linux* OS, Version 2018 Update 5 Build 20190404
OpenMP version	4.5
Node type	a) Dell PE630, b) Dell PEC6420
Processor	a) Intel Xeon E5-2630 v2, b) Intel Xeon Gold 6132
# Processors per node	2
# Cores per node	a) 20, b) 28
Memory per node	a) 128 GB, b) 192 GB

Table 7.1: Properties of Idun

Property	Value
Compiler	ICC 18.0.1
Compiler flags	-std=c99 -g -O3 -qopenmp -lm -liomp5
MPI	SGI MPT 2.14
OpenMP version	4.5
Processor	Intel Xeon E5-2670 ('Sandy Bridge')
# Processors per node	2
# Cores per node	16
Memory per node	32 GB

Table 7.2: Properties of Vilje

7.2 Parameter Space

In this section we describe the parameters which affect the performance and scalability of the solution.

7.2.1 Amount of Refinement

The coarsest level of refinement is defined by the size of its space steps and its time step. Each finer level is defined by some refinement in both space and in time relative to its parent level. There can either be infinitely many finer levels, or there can be some limit, either defined by a maximum number of levels or by a maximum amount of refinement. In our experiments we set a maximum number of levels.

Adding an extra level of refinement increases the amount of work that needs to be done, which affects the performance of the application. In Section 7.3.1 we examine this effect.

7.2.2 Size of Refinement

A large refinement between each level could mean less total work at levels between the coarsest and the finest level. On the other hand, it could also mean that some areas are refined more than necessary, and it could lead to a lower accuracy in the initial and border values the child grids get from their parent. In our experiments we choose a refinement ratio of 2.

7.2.3 Initial Condition

The initial condition of the height and velocity of the fluid may be changed. This will affect the development of the solution, and thus the areas which require refinement as well as their total size across different levels of refinement. This will in turn affect how much work the integration algorithm has to do, and thus its performance. In Section 7.3.2 we set up two different initial conditions of the height and compare their effect on the performance.

7.2.4 Refinement Requirement

The requirement for refining a given point may depend on something mathematical, like an error estimate, or something physical, like the fluid height. The value of the requirement affects the number of points which need refinement, which in turn affects the amount of work and thus the performance for the integration step. In our experiments we require refinement for a point if its fluid height is above some threshold. This threshold increases with the refinement.

A subarea is refined using a grid if a large enough share of the points within that subarea require refinement. A larger required amount results in a smaller unnecessarily refined area in total, which leads to less amount of work and thus better performance for the integration step. On the other hand, a larger required amount also leads to a larger number of grids. This in turn leads to a longer border in total and thus a larger total overhead from border handling. This is explored in more detail in Section 5.1. In our experiments we refine a subarea if at least 0.9 of the points require refinement.

7.2.5 Regridding Interval

Regridding is the process of changing the layout of child grids. The more often we regrid, the more precisely we can refine only what is needed. We may increase the size of each grid with a buffer to ensure that features which require refinement don't move outside a given grid before the next regridding. Then the more often we regrid, the smaller this buffer can be, and thus a smaller area needs to refined, which means less work and thus higher performance for the integration step. On the other hand, the more often we regrid, the larger the total overhead from the regridding algorithm. In our experiments, we strike a balance between these factors by setting the regridding interval to 5.

7.2.6 Size of Domain

The length and width in space, as well as the duration in time, affects the amount of work to be done. In Section 7.3.2 we keep the total length and width of the domain, but decrease one of these dimensions per node by increasing the number of nodes. In Section 7.3.3 we increase the one of these dimensions of the domain, but keep the length and the width constant on each node by increasing the number of nodes.

7.3 Setup For Validation of Predictions

In Section 6.5 we made a number of predictions about the performance and scalability characteristics of our application. In this section we describe our methodology for experimentally validating those predictions. First we describe the setup that all our experiments have in common. Then we describe the experiment-specific details. In Chapter 8 we perform these experiments and discuss to what degree our predictions are validated.

We set up initial disturbances in the fluid height η shaped like waves. Each of our experiments use one or both of the following waves:

- **Wave from edge:** Sine wave which starts at one edge of the domain, as shown in Figure 7.1, and moves in the Y direction towards the other edge, as shown in Figure 7.2.
- **Wave from middle:** Sine wave which starts in the middle of the domain, as shown in Figure 7.3, and moves outwards in both the positive and the negative Y directions, as shown in Figure 7.4.

The figures show the refined areas as areas with a higher density of points. The waves are adaptively refined over time as they move, and areas without a wave at a given timestep are not refined.

A single point at the coarsest level requires refinement if

$$\eta - \eta_0 > 0.1 \times \eta_{wave,max} \times r_L \tag{7.1}$$

Property	Value
η_0	1
$\eta_{wave,max}$	0.1
dy_{root}	0.01
dx_{root}	0.01
dt_{root}	1.25×10^{-4}
r	2
req_{grid}	0.9
RI	5

Table 7.3: Parameters which all experiments have in common

Where

- η_0: Normal fluid height
- $\eta_{wave,max}$: Max fluid height of wave above η_0
- r_L: Total refinement relative to root at refinement level L

This means we require refinement where a wave is located at any given time. Regions with a wave will therefore have a higher number of points than regions without a wave, and ranks with these regions will have a higher number of points than ranks without such regions.

Table 7.3 summarizes some of the parameters of the setup which all experiments have in common. dy_{root}, dx_{root} and dt_{root} are the step sizes in the spatial and temporal directions for root. req_{grid} is the share of points in an area which require refinement in order to make a child grid.

7.3.1 Cost of an Extra Level of Refinement

In this experiment we add levels of refinement and look at what happens to the number of points and the total runtime. We run our experiment for 1, 2, 3 and 4 levels of refinement L on Vilje and on both node types on Idun.

If we add an extra level of refinement, Equation 6.49 says the number of times more points will be

$$1 \leq P_{cost} \leq r^3 + 1 \tag{7.2}$$

In our experiments the refinement ratio r is 2. In that case we have that

$$1 \leq P_{cost} \leq 9 \tag{7.3}$$

We run our application on 1 node. Then LB will be perfect in all cases, and thus unchanged. In that case Equation 6.57 predicts the cost in computation time $T_{comp,cost}$ of adding an extra level will be

$$1 \leq T_{comp,cost} \leq 1 + r^3 \tag{7.4}$$

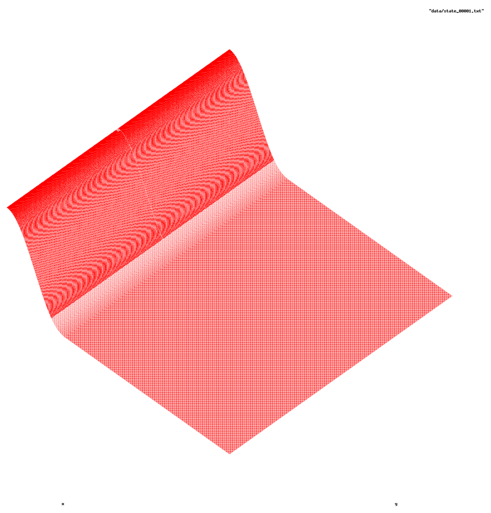


Figure 7.1: Wave from edge initially. The refined area has a higher density of points.

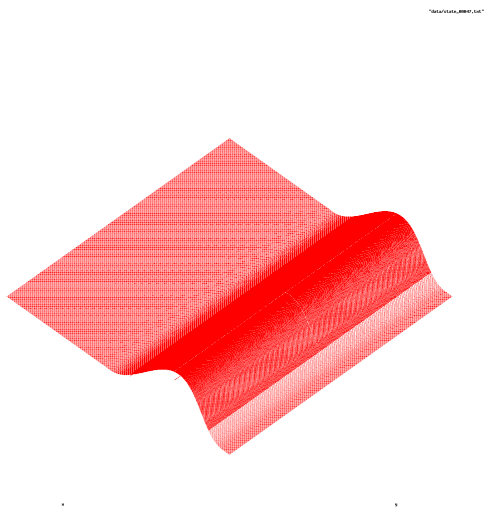


Figure 7.2: Wave from edge after moving across the domain. The refined area has a higher density of points.

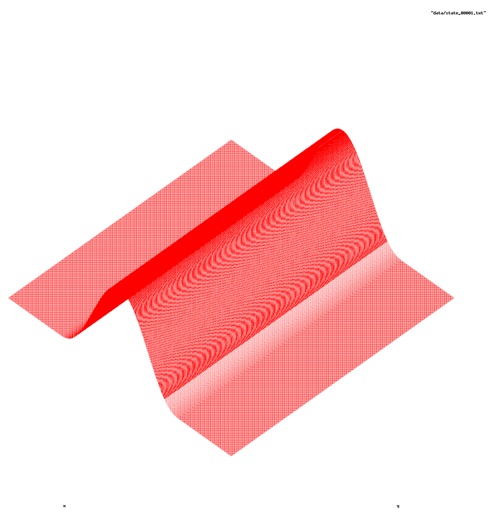


Figure 7.3: Wave from middle initially. The refined area has a higher density of points.

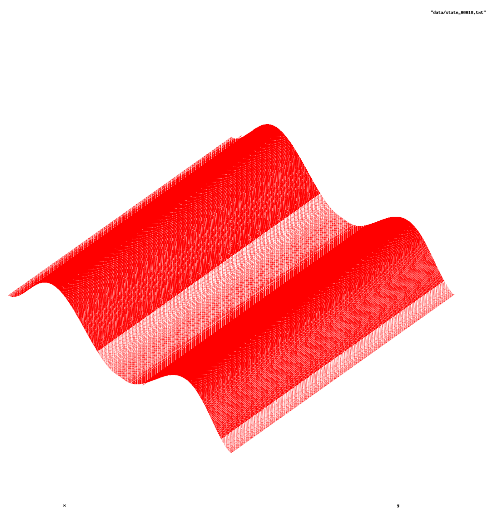


Figure 7.4: Wave from middle after being split and moving outwards. The refined areas have a higher density of points.

Property	Value
Machines	Idun (PE630, PEC6420), Vilje
Wave type	Edge
Y	64
X	32000
I	12000
N	1
L	1, 2, 3, 4

Table 7.4: Experimental setup for cost of extra level

and $P_{cost} = T_{comp,cost}$.

Since we run on 1 node, T_{comm} will be 0, and the increase in T_{tot} will be equal to $T_{comp,cost}$, and thus P_{cost}.

Table 7.4 summarizes some of the parameter values unique to the setup of this experiment.

7.3.2 Strong Scaling

In Section 6.5.2 we predict that as we scale up the number of nodes N, the speedup will increase. We also predict that the load balance LB may decrease as N increases, which may dampen the speedup. Finally, T_{comm} will stabilize as N increases enough to make some nodes reach their maximum number of neighbors.

In our experiment we increase N and look at what happens to the speedup for the two different waves. We expect the speedup of the computation part to be as in Equation 6.62,

$$S_{comp} = \frac{T_{comp,A}}{T_{comp,B}} = \frac{LB_B \times N_B}{LB_A \times N_A} \tag{7.5}$$

In the case where $N_A = 1$, we have that $LB_A = 1$. Then the speedup from 1 to N_B nodes simplifies to

$$S_{comp} = LB_B \times N_B \tag{7.6}$$

Based on this we predict that the wave with the highest LB_B will have the highest speedup for a given N_B.

Table 7.5 summarizes some of the parameter values unique to the setup of this experiment.

7.3.3 Weak Scaling

In this experiment we scale up the number of nodes and the problem size by the same factor. We scale up the problem size by increasing the width X, from 512 for $N = 1$, to $512 \times 16 = 8192$ for $N = 16$. We use a wave from the edge which moves in the Y direction, and split the domain into ranks in the Y direction only. This ensures a perfect load balance, as not only the area, but also the amount of work is scaled by the same amount as the number of nodes.

Property	Value
Machines	Vilje
Wave types	Middle, Edge
Y	64
X	102400
I	12000
N	1, 2, 4, 8, 16
L	2

Table 7.5: Experimental setup for strong scaling

Property	Value
Machines	Idun (PE630), Vilje
Wave type	Edge
Y	512
X	$512 \times$ N
I	40000
N	1, 2, 4, 8, 16
L	2

Table 7.6: Experimental setup for weak scaling

In that case, Gustafson and Equation 2.13 says that as we increase N, the parallel efficiency E will tend towards

$$\lim_{N \to \infty} E = p_{comp} \tag{7.7}$$

where p_{comp} is the share of the computation part of our application which can be parallelized. In our model we assume that $p_{comp} = 1$, and predict that the parallel efficiency E of T_{comp} relative to 1 node will be 1. The assumption that $p_{comp} = 1$ is of course an approximation, but as long as $p_{comp} \approx 1$, this assumption does not make much of a difference for weak scaling.

Since the ranks are split in only one dimension, a rank will have at most 2 neighbors. Therefore, we expect the communication time to be greater than 0 for $N > 1$, but to stabilize as we increase N. To measure T_{comm} we put in a barrier where all ranks have to wait for each other to catch up, just before the handling of the rank border. Once all ranks reach the barrier, we start measuring T_{comm}. This ensures that we don't end up measuring the time some rank has to wait for another rank in addition to the communication.

We choose to also measure T_{tot} and T_{comp} for this application with the barrier before the communication. The downside is that the application without the barrier has better performance. The upside is that by using the same measurement run for the different types of values, we can remove one source of uncertainty.

Table 7.6 summarizes some of the parameter values unique to the setup of this experiment.

Property	Extra level	Strong scaling	Weak scaling
Machines	Idun (PE630, PEC6420), Vilje	Vilje	Idun (PE630), Vilje
Wave types	Edge	Middle, Edge	Edge
Y	64	64	512
X	32000	102400	$512 \times N$
I	12000	12000	40000
N	1	1, 2, 4, 8, 16	1, 2, 4, 8, 16
L	1, 2, 3, 4	2	2

Table 7.7: Summary of setups for the different experiments

7.3.4 Summary

Table 7.7 summarizes some of the values for the setup of the different experiments.

Chapter 8

Results and Discussion

In this chapter we look at the results from the measurements, compare them to the predictions from our performance model, and discuss their implications for the performance and scalability characteristics.

8.1 Cost of an Extra Level of Refinement

In this section we examine the cost of adding an extra level of refinement relative to keeping the current level of refinement, and discuss to what extent our predictions are validated. First we look at the increase in the number of points P from adding an extra level. Then we look at the effect of the increase in P on the total runtime.

8.1.1 Measurements

8.1.1.1 Number of Points

Figure 8.1 shows the factor of increase in P from adding an extra level of refinement. The measured results are well within the predicted range of 1 and $r^3 + 1 = 9$.

The increase differs between different pairs of levels. We attribute this to different requirements for refinement. Level 1 includes all the points in the root grid. Level 2 refines only the points where the wave is present and has a height above the threshold, which excludes much of the root grid. Level 3 and 4 refine most of the areas of their previous levels, since that is where the wave is, but they have gradually stricter requirements for refining a point.

8.1.1.2 Runtime

Figure 8.2 shows the factor of increase in P from adding an extra level of refinement. Our modeled increase in T_{tot} is given as the measured increase in P, as shown in Figure 8.1. Figure 8.3 shows the difference between the measured and the modeled increase T_{tot}. The

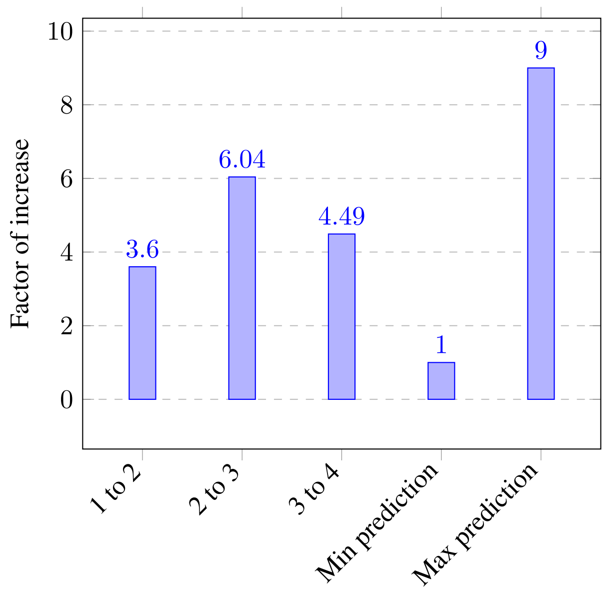


Figure 8.1: Factor of increase in the total number of points from adding an extra level of refinement.

measured increase from 1 to 2 nodes is larger than modeled for all machines. We attribute this to

- **Caching**: As we increase L, and thus the P, our memory usage increases. To minimize effects of different cache levels with different performance on our runtime, we have chosen a problem size which makes our memory usage larger than the capacity of the L3 cache even for just 1 level of refinement. However, a larger share of our memory will fit in L3 for 1 level of refinement than for 2 levels of refinement, contributing to a larger increase in runtime.

- **Other computation parts**: In our model we exclude the parts of the computation which have a low impact on T_{tot}. However, these parts still have an impact larger than nothing. Regridding and downsampling are performed for 2 levels, but not 1 level, and will thus contribute to a larger increase in runtime.

8.1.2 Summary

If we add an extra level of refinement, the factor of increase in the number of points P will be between 1 and $r^3 + 1$ relative to the current level of refinement. If the load balance is unchanged and there is no communication, the total runtime will increase by the same factor as P.

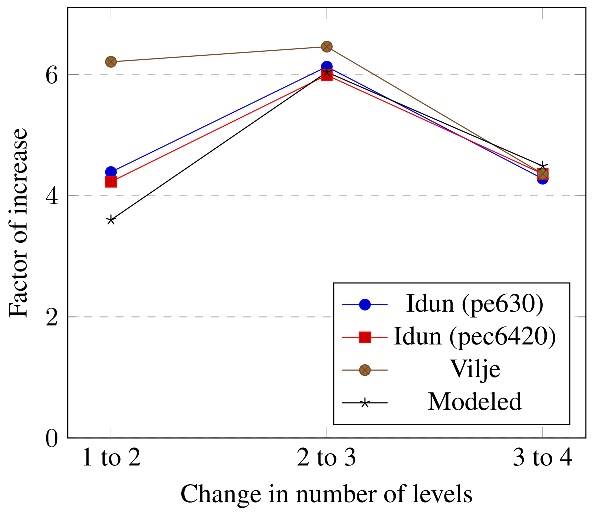


Figure 8.2: Factor of increase in the total runtime from adding an extra level of refinement.

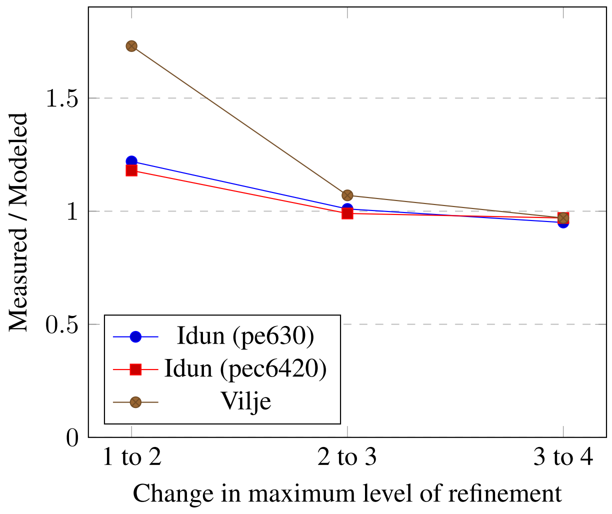


Figure 8.3: Cost of extra level: Ratio between measured and modeled increase in runtime

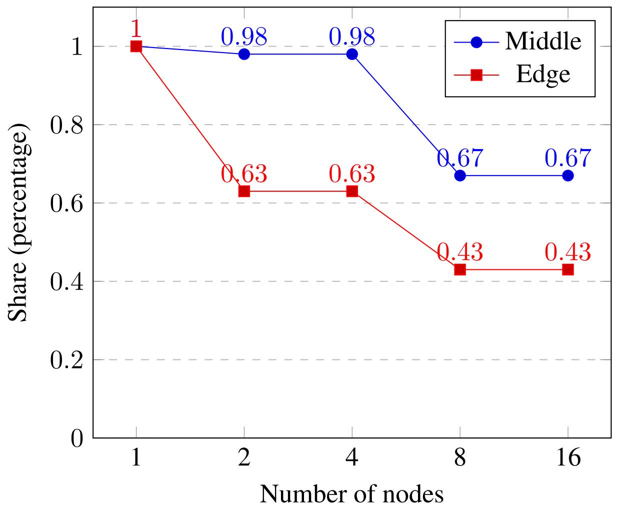


Figure 8.4: Load balance for two different initial waves as we scale up the number of nodes

8.2 Strong Scaling

In this section we look at strong scaling. We make measurements of what happens as we scale up the number of nodes, and discuss to what extent our predictions are validated.

First we measure the load balance of two different initial waves for different numbers of nodes. Then we measure the total runtime for both waves, and compare their speedup. Next, we measure the share of the communication relative to the the total runtime. Finally, we compare the measured speedup of the computation part of the two waves to each other, and to the modeled speedup.

8.2.1 Measurements

8.2.1.1 Load Balance

Figure 8.4 shows the load balance LB for the two initial waves as we scale up the number of nodes N. LB decreases with N for both waves. Thus, we predict that the scalability for both waves will be dampened by a factor of LB. The wave from the middle has a better LB than the wave from the edge for all $N > 1$. Thus, we predict that the wave from the middle will scale better with N than the wave from the edge.

8.2.1.2 Speedup of Total Runtime

Figure 8.5 shows the speedup of T_{tot} relative to 1 node as we scale up the number of nodes. The total runtime includes both the time spent on computation and the time spent on communication. Figure 8.6 shows the parallel efficiency of T_{tot} for the same case. The wave with the best load balance for all $N > 1$, namely the wave from the middle, scales better with N than the wave from the edge. The efficiency decreases with N for both

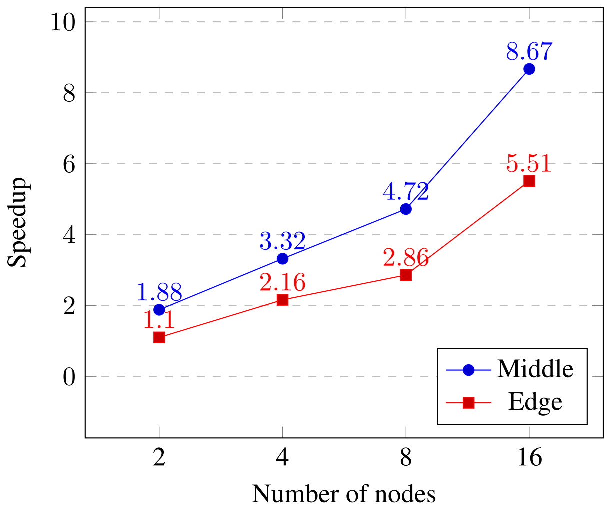


Figure 8.5: Strong scaling of total runtime: Speedup relative to 1 node as we scale up the number of nodes

waves, to almost half for the wave from the middle, and to almost a third for the wave from the edge.

8.2.1.3 Share of Communication

Figure 8.7 shows the share of the communication time T_{comm} relative to the total runtime T_{tot}, measured in percentage. The main takeaway is that T_{comm} takes up a small share of T_{tot}.

Some other notes of interest:

- T_{comm} takes about the same share of T_{tot} for both waves.
- T_{comm} increases with N. We expect that T_{comm} will increase as each rank gets more neighbor ranks. The highest number of neighbors a rank has for a given N increases from each of our N to the next. Thus, we attribute the increase in T_{comm} to the increase in the number of neighbor ranks.

8.2.1.4 Speedup of Computation

Figure 8.8 shows the speedup of the computation time T_{comp} relative to 1 node as we scale up the number of nodes N. The modeled speedups from 1 node to N nodes are found by putting the different pairs of N and LB from Figure 8.4 into Equation 7.6:

$$S_{comp} = LB \times N \tag{8.1}$$

Where S_{comp} is the speedup of the computation part for strong scaling.

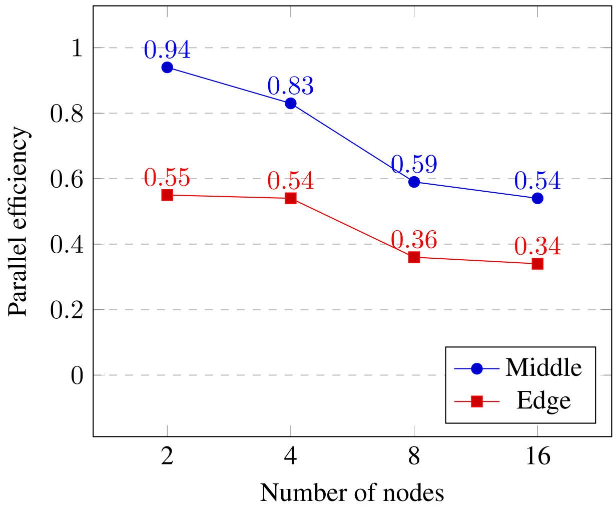


Figure 8.6: Strong scaling of total runtime: Parallel efficiency relative to 1 node as we scale up the number of nodes

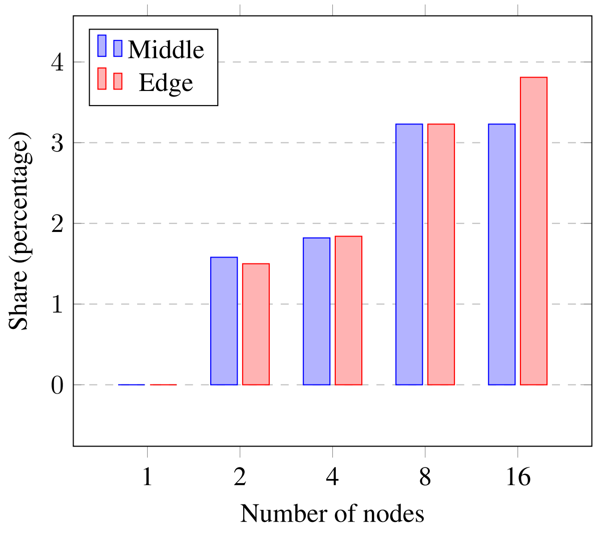


Figure 8.7: Strong scaling: Share of communication relative to the total runtime, measured in percentage

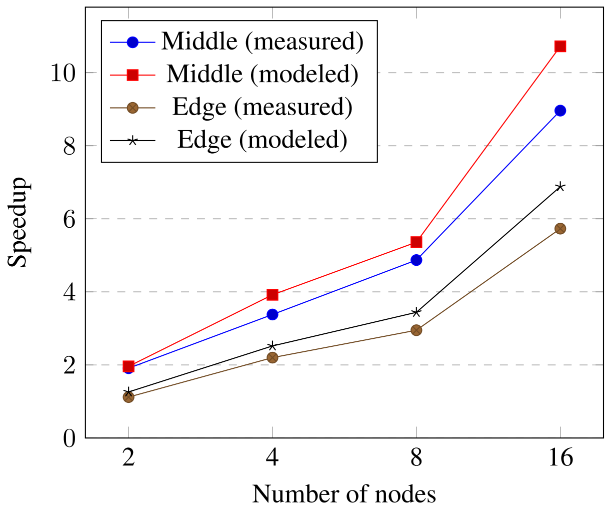


Figure 8.8: Strong scaling of computation time: Speedup relative to 1 node as we scale up the number of nodes

The wave from the middle has an almost perfect load balance for 2 and 4 nodes. As a result, the modeled speedup for those N are almost linear for that wave. The measured speedup for the same configuration is also almost linear. All other $N > 1$ for both waves have an LB smaller than 1. Figure 8.9 shows the parallel efficiency of the computation part for strong scaling. The modeled efficiencies decrease with LB. The measured efficiencies decrease by about the same factor.

Figure 8.10 shows the ratio between the measured and the modeled speedup of the computation part relative to 1 node as we scale up the number of nodes. We attribute the difference between measurements and model to

- **Amdahl's law**: As we increase the resources of a system, the speedup will be limited by the inherently serial part of the program.
- **Other computation parts**: In our model we exclude the parts of the computation which have a low impact on the total runtime. However, these parts still have an impact larger than nothing. The downsampling and the applying of the regridding only affect the points in child grids, which means they will make the load balance slightly worse.

Both the model and the measurement shows that the wave with the best load balance for all $N > 1$, namely the wave from the middle, scales better with N than the wave from the edge.

8.2.2 Summary

The wave with the better load balance scales better with the number of nodes, both for the total runtime and for the computation time. The parallel efficiency decreases by about

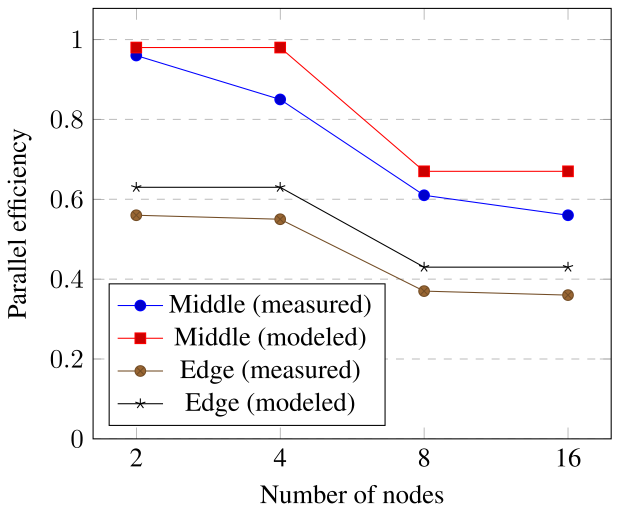


Figure 8.9: Strong scaling of computation time: Parallel efficiency relative to 1 node as we scale up the number of nodes

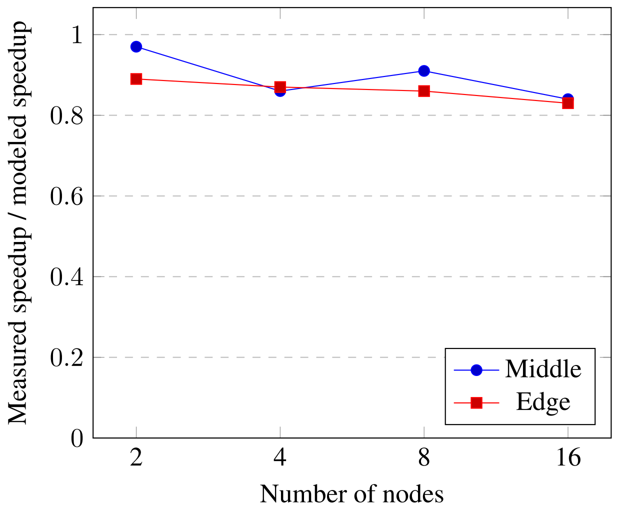


Figure 8.10: Strong scaling of computation time: Ratio between measured and modeled speedup relative to 1 node as we scale up the number of nodes

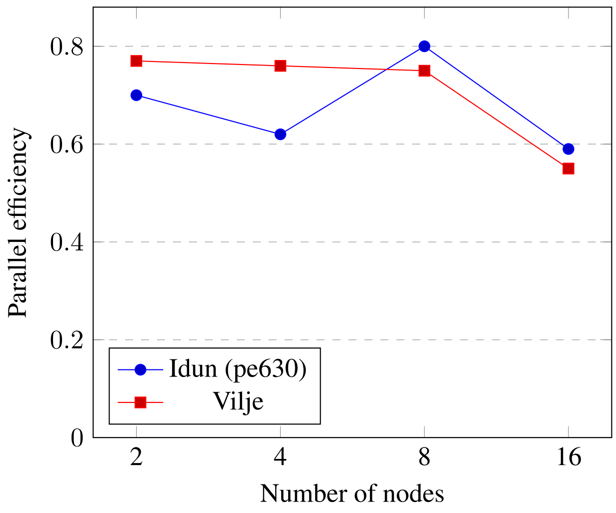


Figure 8.11: Weak scaling: Total runtime: Parallel efficiency relative to 1 node as we increase the number of nodes and the problem size by the same factor

the same factor as the load balance. The communication time is a small share of the total runtime.

8.3 Weak Scaling

In this section we examine weak scaling. We measure the change in runtime as we scale up the number of nodes and the problem size by the same factor, and discuss to what extent our predictions are validated.

8.3.1 Measurements

8.3.1.1 Total Time

Figure 8.11 shows the parallel efficiency of the total runtime relative to 1 node as we increase the number of nodes and the problem size by the same factor. The results vary between the different numbers of nodes, and dip below 0.6 for $N = 16$. We attribute these differences and the dip to different rates of computation between different nodes on both our machines. We measure the time each node has to wait for the other nodes to reach the barrier. Then we withdraw the difference between the maximum and the minimum time a node has to wait from the total time.

8.3.1.2 Computation and Communication Time

After removing this wait time, we are left with the computation and the communication time. The parallel efficiency of this time compared to 1 nodes is shown in Figure 8.12.

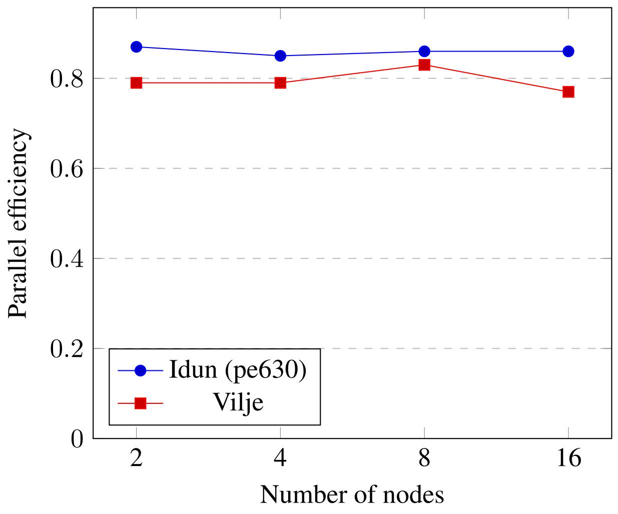


Figure 8.12: Weak scaling: Computation + communication time: Parallel efficiency relative to 1 node as we increase the number of nodes and the problem size by the same factor

The results are stable for all $N > 1$. This means we can increase the number of nodes 8 times from 2 to 16, and get 8 times as much work done in the same time.

The parallel efficiency is below 1. We attribute this to the communication time.

We attribute the share of the communication time relative to the combined computation and the communication time to caching. The problem size of our experiment makes the memory for each node fit in the L3 cache, which is faster than main memory. This makes the computation part faster by some constant. The communication part is unaffected, and thus takes up a larger share of the combined computation and communication time.

8.3.1.3 Computation time

Figure 8.13 shows the parallel efficiency of the computation time relative to 1 node as we increase the number of nodes and the problem size by the same factor. The results show that the computation part can handle about 16 times as much work in the same time if we increase the number of nodes 16 times.

8.3.2 Summary

As we increase the problem size and the number of nodes by the same factor, the computation time is stable. The communication time does not increase with N for $N > 1$. The sum of the computation and the communication time is constant for $N > 1$. This means we can increase the number of nodes and get more work done in the same time.

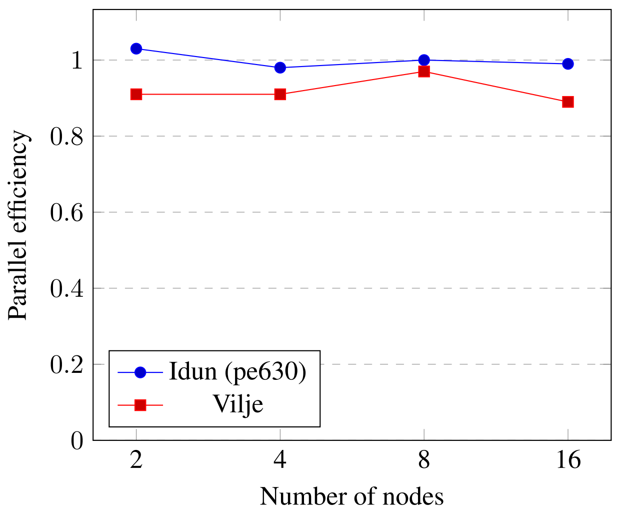


Figure 8.13: Weak scaling: Computation time: Parallel efficiency relative to 1 node as we increase the number of nodes and the problem size by the same factor

Chapter 9

Conclusion

In this thesis we have studied the performance and scalability characteristics of adaptively refined finite difference solutions to the shallow water equations.

We have developed a proxy application which solves the SWE using the MacCormack FDM and AMR. We have created a performance model and used it to make predictions about the performance and scalability of the application, and experimentally validated those predictions on two machines, Idun and Vilje. We have set up initial disturbances shaped like waves in the fluid. Each of our experiments has used one or two different waves.

Our results show that if we add an extra level of refinement, the total number of points will increase with a factor between 1 and $r^3 + 1$. If the load balance is unchanged and there is no communication, the runtime will increase by the same factor.

For strong scaling, the wave with the better load balance scales better with the number of nodes, both for the total runtime and for the computation time. The parallel efficiency decreases by about the same factor as the load balance. The communication time is a small share of the total runtime.

For weak scaling, as we increase the problem size and the number of nodes N by the same factor, the computation time is about the same. The communication time does not increase with N for $N > 1$. The sum of the computation and the communication time is stable for $N > 1$. This means we can increase the number of nodes and get more work done in the same time.

9.1 Future work

Our performance model acknowledges that there is some communication cost, but does not model it in detail. A better model for the communication cost could improve our understanding of the performance of the application, especially at very large scales.

The SWE using AMR have been successfully implemented on GPU. A performance model for the SWE using an FDM and AMR, or a comparison to our CPU application, could give us a better understanding of the impact of hardware on the performance and scalability of the application.

We show that the load balance can greatly impact the performance and scalability of our application. A possible mitigation is to implement load balancing. However, such a solution would come with an overhead. Investigating this trade-off could improve our understanding of when and how to use load balancing.

Bibliography

[1] E. J. Kubatko, S. Bunya, C. Dawson, J. J. Westerink, and C. Mirabito, “A performance comparison of continuous and discontinuous finite element shallow water models,” *Journal of Scientific Computing*, vol. 40, no. 1-3, pp. 315–339, 2009.

[2] G. A. Sod, “A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws,” *Journal of computational physics*, vol. 27, no. 1, pp. 1–31, 1978.

[3] (2020, April) What are Boundary Conditions? SimScale. [Online]. Available: https://www.simscale.com/docs/content/simwiki/numerics/what-are-boundary-conditions.html

[4] M. J. Berger, “Adaptive mesh refinement for hyperbolic partial differential equations,” STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, Tech. Rep. Report No. STAN-CS-KL-924, 1982.

[5] M. J. Berger, P. Colella *et al.*, “Local adaptive mesh refinement for shock hydrodynamics,” *Journal of computational Physics*, vol. 82, no. 1, pp. 64–84, 1989.

[6] P. Pacheco, *An Introduction to Parallel Programming*. Elsevier, 2011.

[7] (2020, March) Home - OpenMP. OpenMP Architecture Review Board. [Online]. Available: https://www.openmp.org/

[8] D. Luebke, “CUDA: Scalable parallel programming for high-performance scientific computing,” in *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*. IEEE, 2008, pp. 836–838.

[9] A. Munshi, “The OpenCL specification,” in *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 2009, pp. 1–314.

[10] (2020, March) FAQ: General information about the Open MPI Project – What is MPI? What is Open MPI? OpenMP Architecture Review Board. [Online]. Available: https://www.open-mpi.org/faq/?category=general#what-is-mpi

[11] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the Chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[12] (2019, December) ECP Proxy Applications. Exascale Proxy Application Project. [Online]. Available: https://proxyapps.exascaleproject.org/

[13] S. S. Dosanjh, R. F. Barrett, D. Doerfler, S. D. Hammond, K. S. Hemmert, M. A. Heroux, P. T. Lin, K. T. Pedretti, A. F. Rodrigues, T. Trucano *et al.*, "Exascale design space exploration and co-design," *Future Generation Computer Systems*, vol. 30, pp. 46–58, 2014.

[14] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483–485.

[15] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[16] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[17] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[18] L. Jameson, "AMR vs high order schemes," *Journal of scientific computing*, vol. 18, no. 1, pp. 1–24, 2003.

[19] K. Erduran, V. Kutija, and C. Hewett, "Performance of finite volume solutions to the shallow water equations with shock-capturing schemes," *International journal for numerical methods in fluids*, vol. 40, no. 10, pp. 1237–1273, 2002.

[20] M. L. Sætra, A. R. Brodtkorb, and K.-A. Lie, "Efficient GPU-implementation of adaptive mesh refinement for the shallow-water equations," *Journal of Scientific Computing*, vol. 63, no. 1, pp. 23–48, 2015.

[21] R. Courant, K. Friedrichs, and H. Lewy, "On the partial difference equations of mathematical physics," *IBM journal of Research and Development*, vol. 11, no. 2, pp. 215–234, 1967.

[22] R. MacCormack, "The effect of viscosity in hypervelocity impact cratering," *Journal of spacecraft and rockets*, vol. 40, no. 5, pp. 757–763, 2003.

[23] A. Hindmarsh, P. Gresho, and D. Griffiths, "The stability of explicit euler time-integration for certain finite difference approximations of the multi-dimensional advection–diffusion equation," *International journal for numerical methods in fluids*, vol. 4, no. 9, pp. 853–897, 1984.

[24] (2020, April) Generalization: reflecting boundaries. Center for Biomedical Computing. [Online]. Available: http://hplgit.github.io/INF5620/doc/pub/sphinx-wave/._main_wave003.html

[25] M. Berger and I. Rigoutsos, “An algorithm for point clustering and grid generation,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, no. 5, pp. 1278–1286, 1991.

[26] M. Själander, M. Jahre, G. Tufte, and N. Reissmann, “EPIC: An energy-efficient, high-performance gpgpu computing research infrastructure,” *arXiv preprint arXiv:1912.05848*, 2019.

[27] (2020, June) About Vilje. NTNU HPC GROUP. [Online]. Available: https://www.hpc.ntnu.no/ntnu-hpc-group/vilje/about-vilje