Master's thesis

2019

Andreas Kolstø Kaldager

Master's thesis

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

Andreas Kolstø Kaldager

Comparative performance modeling of a WCSPH proxy application on GPU and CPU architectures

June 2019

NTNU
Norwegian University of
Science and Technology

Comparative performance modeling of a WCSPH proxy application on GPU and CPU architectures

Andreas Kolstø Kaldager

Algorithms and HPC - Computer Science
Submission date: June 2019
Supervisor: Jan Christian Meyer, IDI

Norwegian University of Science and Technology
Department of Computer Science

Problem description

This study aims to create experimentally validated performance models for WCSPH on GPU & CPU, and use them to identify scalability characteristics that predict how selected platforms compare at large scale.

Assignment given: January 15, 2019
Supervisor: Jan Christian Meyer

Abstract

In this thesis, we investigate the performance of a WCSPH proxy application applied to the dam-break problem when offloading the computational part to the GPU. We implement one naive and one sophisticated approach for the bottleneck of the problem, which is finding neighboring particles. We create performance models for communication, computation, and the parts therein. Using these models and empirical data, we investigate the scalability characteristics of the application, the Speedup and Efficiency of the offloading, the impact of the bottleneck on the application, devise a utility range for offloading the problem, analyse the errors of the estimates used, and create a formula for the maximum problem size given a GPU's memory. The models are experimentally validated with the empirical data. We conclude that offloading lowers the impact of the bottleneck and achieves a Speedup of 10, but lowers Efficiency. We also conclude that the communication amount will outgrow the computation amount given a sufficiently large problem size, the estimates used are below 20% *in error, and that the application scales well with no performance drawbacks from horizontal scaling.*

Acknowledgements

I would like to thank my supervisor Jan Christian Meyer for introducing me to the world of HPC and performance modeling, for always being ready to discuss and explain the complications of these fields, for both constructive critique and positive feedback and for guiding me through the complicated rules and norms of the scientific model and the academic system. Without him, this thesis would not have been possible.

I would also like to thank Nico Reissmann for helping me get the experimental data when the clusters had very long queue in a critical time and for information and tips regarding said clusters.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Smoothed Particle Hydrodynamics (SPH) is important for its uses in both astrophysical simulations and fluid simulation. These simulations can be used to model gravity in galaxies, simulate fluid in computer generated imagery, or develop maritime and oil equipment. However, fluid simulation is often very complicated and highly performance demanding. Poor implementations can increase the simulation time manifold. Proxy-applications are applications containing critical parts of a larger system. This makes them easy to profile and can be used to extract the vital part of the performance.

In this thesis we look at the performance and scalability of the Weakly Compressible SPH (WCSPH) method when the physics simulation component of the proxy application is offloaded to the GPU via the General Purpose GPU (GPGPU) platform CUDA. We investigate six aspects of offloading:

- the relationship between communication and computational complexity of the method
- the impact of the bottleneck of SPH, neighbor finding
- Speedup and Efficiency of the offloaded versions
- minimum and maximum requirement for offloading
- how we can model the performance of the application
- scalability of the application

We implement one brute-force version and a more sophisticated version for finding neighboring particles (neighbors) and propose a general performance model as well as models for both approaches. Both the brute-force and the more sophisticated version are empirically tested on two clusters, EPIC2 and V100, belonging to NTNU's supercomputer IDUN. We use the empirical data to test our performance models and develop tools that can assist in the choice of hardware and whether offloading is beneficial given a problem size or number of computing nodes.

We look at the utility range $[min < P < max]$ for two GPU architectures, where P is the number of particles and max is the number of particles that the memory of a given GPU can store. When offloading a problem to the GPU, there is additional time being spent on transferring the data to the GPU before it can be processed. Therefore min is the number of particles that is needed before the cost of transfer is diminished by the throughput of the GPU.

We predict that using the optimal configuration of number of ranks, with regards to performance, for a given problem size will cause the communicational amount to surpass the computational amount when the problem size is sufficiently large. This optimal configuration is predicted to be the same as the maximum number of ranks allowed by the problem.

We find that the high throughput of the GPU architecture is able to reduce the performance bottleneck of finding neighbors. This scales well with increasing problem size for the sophisticated method within the utility range.

We find it is always worth offloading the problem using the sophisticated neighbor-finding approach, that the naive approach outperforms the sophisticated methods at a lower number of particles, and propose a formula for calculating the maximum number of particles for the best performing method given a memory size.

We find two linear estimates that are both used by the performance models. e_p estimates the number of pairs per particle $e_p = 23.6$ that some of the physics related performance models depend on. The communication model uses b_p as an estimate for the number of particles exchanged in communication $b_p = 100 \times SCALE$. We investigate their error during iterations and different scales of the problem. We find e_p is well within the error range of $[0, 10]\%$ while b_p is within the range of $[0, 20]\%$ given sufficient iterations.

We find that using the sophisticated method on CPU as a baseline, the sophisticated method on GPU achieves a Speedup of 10. This Speedup is comparing the CPU to the GPU on the same cluster. However, the Efficiency is much lower for the GPU version.

We discover that the proxy application scales well by having no performance drawbacks from adding more computing nodes, however adding more nodes will render some nodes idle during the simulation of the dam-break problem.

Chapter 2 presents the motivation and scope for this thesis. **Chapter 3** introduces the background of the WCSPH method, background of the implementation of the proxy application, and the specific fluid simulation problem being looked at in this thesis. **Chapter 4** describes the background of N-body problems, approaches to N-body problems, the tools used for implementing the application, and work related to this field of study. **Chapter 5** discusses the methodology of this thesis. Firstly, we introduce the implementation of the proxy application. Secondly, we propose a static analysis to model the neighbor finding methods. Thirdly, we describe the performance parameter space that this thesis investigates. Lastly, we introduce the design of the experiments. **Chapter 6** shows the empirical data from the two clusters, discusses the data with regard to the performance model, validates the performance models, and gives recommendations for choice of offloading, hardware and method to use. **Chapter 7** summarizes our discoveries and discusses future work.

Chapter 2

Motivation & Scope

Performance modeling can be used to make an informed decision when buying hardware for running a problem, like a physics simulation. This can both save institutions large amounts of money, as well as make sure that the right hardware is used for the right problems. The importance of performance modeling is only growing due to computer architecture growing more complex and more purpose specific.

Cell-linked lists are a way to reduce the amount of comparisons when a problem requires comparison between many elements. This is why cell-linked lists are important to many types of algorithms, especially those of N-body simulations. These kinds of simulations can run for days or weeks which is why performance optimization is important.

Because CPUs generally have low throughput relative to GPUs, the speedup from the naive method to the cell-linked list method is large, while the naive approach often is preferable on GPGPU due to high throughput. Therefore modeling the performance is important to understand the benefit of more complex solutions on both CPUs and GPUs. The focus of this thesis is therefore to investigate scalability and performance benefits of the different methods with the cell-linked list method as main focus.

2.1 WCSPH Proxy Application

Proxy applications are programs that contain performance critical parts of larger systems in order to make it easier to analyze the critical part. This makes it easier to improve upon the bottleneck of the system and in turn improve performance and energy savings. This thesis concerns one such proxy application, WCSPH. The Dam-break problem, in which fluid contained in a dam like structure is suddenly released and is free to flow within the greater bounds, is explored.

To be able to utilize the resources on shared and distributed memory hybrid systems, the application is written with MPI and OpenMP. Ragunathan and Valstad (2018) thoroughly tested the applications on two different HPC systems. They found that the making of neighbor-lists, further explained in Chapter 3, is the performance bottleneck of the application.

2.2 Programming Models

Programming models enable programmers access to hardware according to the model. McCool (2008) identifies the most important aspects of a programming model as exposing the hardware so that the programmer is able to make the right decisions regarding optimization, as well as hide the less important aspects to make the process easier.

In this thesis, there are multiple programming models used which all target different aspect of hardware like multi-threading, inter-process-communication and target other hardware arrangements like SIMD, which we look at in Section 4.2.

OpenMP and **MPI** are well known CPU tools for making use of multi-threading and multi-processor systems. OpenMP is used to divide workload onto different threads within the same processor, while MPI is used for inter-process communication within the same system, both between cores, processors and compute nodes. This is the foundation of the SPH proxy application, and therefore plays a role in how the GPGPU implementations were made.

CUDA is a programming model for utilizing Nvidia GPUs as GPGPU and has been a large entity in the GPGPU development. The programmer is able to control many aspects of code execution using the programming model, both conventional multi-threading operations like synchronizing threads, but also the number of processors and the number of registers to use.

2.3 Hardware

The hardware used for the experimental data are heterogeneous, distributed and shared memory clusters with inter-node communication and GPUs designed especially for clusters. This makes it simple to scale the application and run on multiple GPUs as well as run problems of larger size.

The main two GPU architectures inside the scope of this thesis is Nvidia's Pascal architecture and Nvidia's Volta architecture.

2.4 Scope

In this thesis we investigate a CUDA implementation which offloads the physics simulation part of the proxy application using both a naive approach and a cell-linked list approach to finding neighbors. The two methods are compared with a performance model specifically created for offloading on GPGPU. The simulation is iterated until near equilibrium state of the fluid is reached.

We study the scalability differences between the CPU versions and the GPU versions. We then explore the performance parameter space of the CPU and GPU implementations, and validate the performance models so that we can inform the choice of hardware and algorithm in accordance to scale when running the SPH proxy application.

We look at the scalability characteristics of communication and computation and use them in addition to the performance models to predict the optimal number of ranks and how the relationship between communication and computation will scale.

We examine the naive approach, comparing all particles to each other, keeping in mind that the naive approach often performs well enough on GPUs compared to CPUs. Finally we look at the cell list implementation which divides the problem space into cells so that each particle only needs to compare to particles in neighboring cells.

Chapter 3

WCSPH Overview

In this chapter we go through the theory and background behind the WCSPH method, the dam-break problem which it is applied to, and what the basis for the WCSPH dam-break proxy application is. We also go through the different solutions used for challenges like boundary conditions, time integration and density correction.

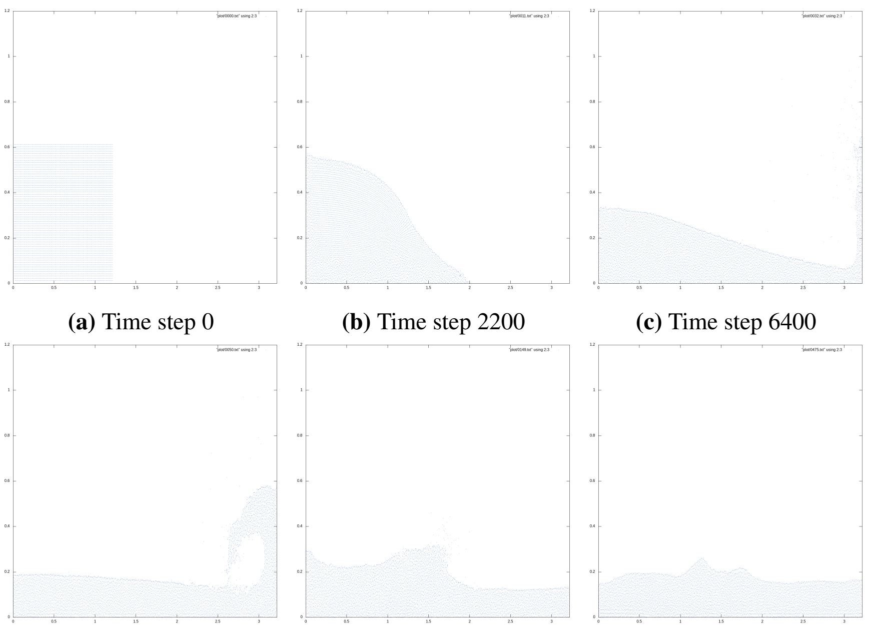
3.1 Background

Gingold and Monaghan (1977) and Lucy (1977) proposed SPH (Smoothed Particle Hydrodynamics) as a method focused on simulating compressible flow problems in astrophysics. This was later built on by Monaghan (1994) to create a version enabling simulation of incompressible free surface flow called Weakly Compressible SPH (WCSPH). The physics of the proxy application used in this thesis is based on Ozbulut et al. (2014), which applies the original formulations of WCSPH to the dam-break problem.

Figure 3.1 shows a simulation of the dam-break problem using WCSPH. The simulation will have more violent motions in the start of the simulation which will slowly settle down until the fluid has reached a state of equilibrium and is completely still. Simulation can take anywhere from seconds to weeks depending on the size of the problem, the number of particles and the number of time steps.

SPH simplifies fluid to a collection of discrete particles, containing properties such as location, velocity, density and pressure. Figure 3.2 shows how the particles interact with all of its neighboring particles within a distance which is set by the **smoothing length**. The weighing function determines the properties of each particle as a sum of contributions from its neighbors.

$$W(R,h) = \alpha_d \begin{cases} (3-R)^5 - 6(2-R)^5 + 15(1-R)^5, & 0 \leq R < 1 \\ (3-R)^5 - 6(2-R)^5, & 1 \leq R < 2 \\ (3-R)^5, & 2 \leq R < 3 \\ 0, & R \geq 3 \end{cases} \tag{3.1}$$



(a) Time step 0 **(b)** Time step 2200 **(c)** Time step 6400

(d) Time step 10000 **(e)** Time step 29800 **(f)** Time step 95000

Figure 3.1: Simulation of dam-break using SPH.

$$R = \frac{r_{i,j}}{h} \tag{3.2}$$

Equation 3.1 is the weighting function proposed by Ozbulut et al. (2014). R is defined by Equation 3.2, where the distance vector between two particles i and j is divided by the smoothing length h. The weighting function divides distances into four classes of weights, decreasing the weight by number of smoothing lengths particle j is separated from particle i. α_d is a coefficient that depends on the number of dimensions used in the problem. This thesis uses 2 dimensions, which sets $\alpha_d = 7/(478\pi h^2)$.

Equation 3.3 shows the methods that WCSPH uses to couple density with pressure through a coefficient that is the speed of sound through a medium. This explicit artificial state equation was proposed by Monaghan and Kos (1999) and later used by Ozbulut et al. (2014). p is the pressure and ρ is the density of particles, γ is the specific heat-ratio of water and is set to 7, c_0 is the reference speed of sound and ρ_0 is the reference density for the particles, which for fresh water is $1000[kg/m^3]$.

$$p = \frac{\rho_0 c_0^2}{\gamma}\left[\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right] \tag{3.3}$$

From Equation 3.3 we can see that changes to the pressure of particles are largely influenced by the density of the particles. For larger values of c_0, the time step will be smaller, which causes unnecessary increase in computational time according to Ozbulut

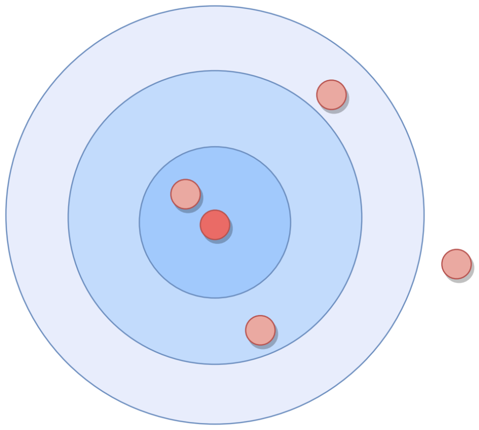


Figure 3.2: Closer particles have more impact on the particle. Reproduced from Ragunathan and Valstad (2018) with permission.

et al. (2014). For smaller values of c_0, the incompressibility condition might not be met. To meet these conditions in the dam-break problem, c_0 is set to $50[m/s]$.

Equation 3.4 and 3.5 shows how Ozbulut et al. (2014) discretize Euler's equation of motion and mass conservation respectively. However, this is without the artificial viscosity term of the motion equation.

$$\frac{d\boldsymbol{u_i}}{dt} = -\sum_{j=1}^{N}\left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}\right)\nabla_i W_{i,j} \tag{3.4}$$

$$\frac{d\rho_i}{dt} = \rho_i \sum_{j=1}^{N} \frac{m_i}{\rho_j}(\boldsymbol{u_i} - \boldsymbol{u_j}) \cdot \nabla_i W_{i,j} \tag{3.5}$$

In Equations 3.4 and 3.5, m_j refers to the mass of particle j, $\boldsymbol{u_i}$ and $\boldsymbol{u_j}$ are the velocities of particles i and j, ∇_i is the gradient operator and indicates the spatial derivative of the position of particle i.

3.1.1 Boundary Conditions

This model uses the Neumann boundary condition, which states that the applied value of the derivative of the solution is within the boundary of the domain. This is imposed by utilizing a mirroring technique called **ghost particles** or **virtual particles**, proposed by Ozbulut et al. (2014). This is in order to achieve physically correct results which requires the wall boundary condition of fluid simulation to be met. Ghost particles are particles that mirror fluid particles that are a vertical distance of $1.55h$ or less away from the boundary. These ghost particles, appearing on the other side of the boundary, are given field values (*i.e.* velocity and pressure) depending on the type of boundary condition implemented.

3.1.2 Time Integration

The time integration used by Ozbulut et al. (2014) creates a predictor-corrector pattern and is performed in two steps, a prediction step and a correction step. In the prediction step position, density and velocity of the particles are updated following the Equations 3.6, 3.7 and 3.8. Here $\boldsymbol{r_i}$ is the position and $\boldsymbol{a_i}$ is the acceleration of particle i. k_i represents the pressure in Equation 3.5.

$$\frac{d\boldsymbol{r_i}}{dt} = \boldsymbol{u_i} \tag{3.6}$$

$$\frac{d\boldsymbol{\rho_i}}{dt} = \boldsymbol{k_i} \tag{3.7}$$

$$\frac{d\boldsymbol{u_i}}{dt} = \boldsymbol{a_i} \tag{3.8}$$

$$\boldsymbol{r_i}^{n+\frac{1}{2}} = \boldsymbol{r_i}^{n} + \frac{1}{2}\boldsymbol{u_i}^{n}\Delta t \tag{3.9}$$

$$\boldsymbol{\rho_i}^{n+\frac{1}{2}} = \boldsymbol{\rho_i}^{n} + \frac{1}{2}\boldsymbol{k_i}^{n}\Delta t \tag{3.10}$$

The position is updated by Equation 3.9 and density is updated by 3.10. Using the intermediate values from the density calculation, the pressure of the particles are updated using Equation 3.3.

In the correction step we use the intermediate values from Equation 3.9 and 3.10 to solve Equation 3.11 and 3.12 to update position and density, respectively.

$$\boldsymbol{r_i}^{n+1} = \boldsymbol{r_i}^{n+\frac{1}{2}} + \frac{1}{2}\boldsymbol{u_i}^{n+1}\Delta t \tag{3.11}$$

$$\boldsymbol{\rho_i}^{n+1} = \boldsymbol{\rho_i}^{n+\frac{1}{2}} + \frac{1}{2}\boldsymbol{k_i}^{n+1}\Delta t \tag{3.12}$$

3.1.3 Density Correction

Ozbulut et al. (2014) uses the density correction algorithm to confine the pressure oscillations that occur because of numerical noise in the system, to an acceptable range.

$$\hat{\rho}_i = \rho_i - \sigma\frac{\sum_{j=1}^{N}(\rho_i - \rho_j)W_{i,j}}{\sum_{j=1}^{N} W_{i,j}} \tag{3.13}$$

Equation 3.13 is used to correct the density in WCSPH for particle i. N is the number of neighboring particles, $\hat{\rho}_i$ is the corrected density and ρ is a constant defined as $\rho = 1$ by Ozbulut et al. (2014).

3.2 Dam-break

The initial configuration of the dam-break problem is shown in Figure 3.3. The fluid, represented by particles, is initialized at $t = 0$ inside the dark region of the figure. The particles are free to flow and will therefore simulate a dam suddenly breaking.

Figure 3.3 shows the parameter `SCALE`, which adjusts the scale of the dam. Since `SCALE` scales the dam both horizontally and vertically, a linear increase will cause a quadratic increase in area. This increase will have the same effect on the number of particles. Equation 3.14 shows the relationship between the number of particles and the `SCALE`. Delta is the resolution of the problem, which in our case is set to $Delta = 0.01$. Using our values of $Delta$, L and T we can rewrite Equation 3.14 with regards to `SCALE` to get Equation 3.15.

$$N_{particles} = (1 + \frac{L \times SCALE}{Delta}) \times (1 + \frac{T \times SCALE}{Delta}) \tag{3.14}$$

$$P(SCALE) = 7200 \times SCALE^2 + 180 \times SCALE + 1 \tag{3.15}$$

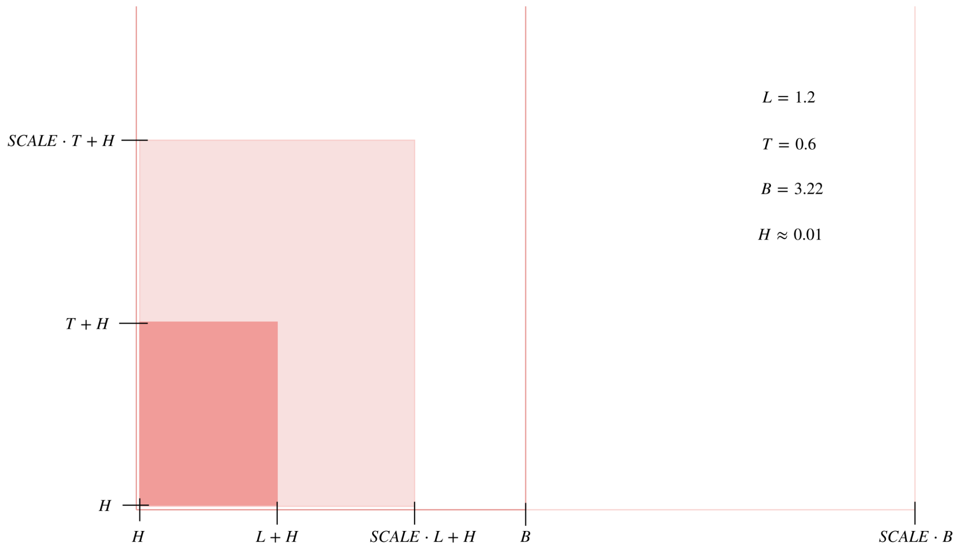


Figure 3.3: Scaling of the dam size. Reproduced from Ragunathan and Valstad (2018) with permission.

Chapter 4

Background & Related work

This chapter describes the background of the type of problem that SPH is, sophisticated solutions to them as well as solutions being used in the industry. Further, we describe useful terminology related to this thesis, background on programming models used and background of performance modeling used to build the models of this thesis.

4.1 N-Body Problems and Cell Lists

N-body simulation is a simulation of particles under influence of pairwise physical forces used in physics and astronomy. SPH falls under the N-body class of problems which is one of ”The Seven Dwarfs” of HPC according to Asanovic et al. (2006). N-body simulations can be optimized in many different ways depending on the characteristics of the specific problem, including meshes, trees and grids.

Barnes and Hut (1986) propose a method in which we split the space into cells in a hierarchical manner. The space is first split into four cells. If cells contain more than one particle, that cell is split into four again. This creates a quadtree-structure of cells where leaf nodes are particles.

Verlet lists was proposed by Verlet (1967) as an efficient way of organizing a list of particles that are in a certain distance of each other. The efficiency stems from the fact that these lists do not need constant updating. These lists can be used in combination with Monte Carlo simulation and cell-linked lists.

Cell-linked lists divide the space of the system uniformly into cells. It reduces the number of comparisons required and therefore reduces computational time. However, implementation of cell lists on GPU is challenging. The straight forward approach relies on atomic functions which slow the execution down. Green (2010) proposes a method which sorts the particles so that memory reads will be coalesced. This involves using radix-sort to sort the list of particles according to which cell it belongs to in the cell list. This method is also used by Harris (2016) in Nvidia’s physics engine, PhysX. Reissman et al. (2014) proposes that the Z-order curve is a sorting pattern that adheres to the properties of locality, which optimizes memory access given the comparison of surrounding cells.

4.1.1 Naive

The naive approach, also called the brute-force approach, is the most straight forward solution to a problem. In terms of N-body, this means to compare every particle to all other particles in the system to check if they interact. Increasing the number of particles increases the number of comparisons quadratically. That is, the problem has a complexity of $O(N^2)$. The number of comparisons can be reduced to $\frac{N(N-1)}{2}$ because an ordered pair with particle P_i and P_j is the same as an ordered pair with P_j and P_i. However, the computational complexity remains $O(N^2)$.

4.1.2 Cell-Linked List

Cell-linked list or cell list is a type of structure that uniformly divides a space into cells where elements belong to the cell they are located in. This structure reduces the number of comparisons done on a set of elements in space when the distance between the particles is large enough that the effect of interacting forces can be neglected. This can reduce the computational time of N-body problems.

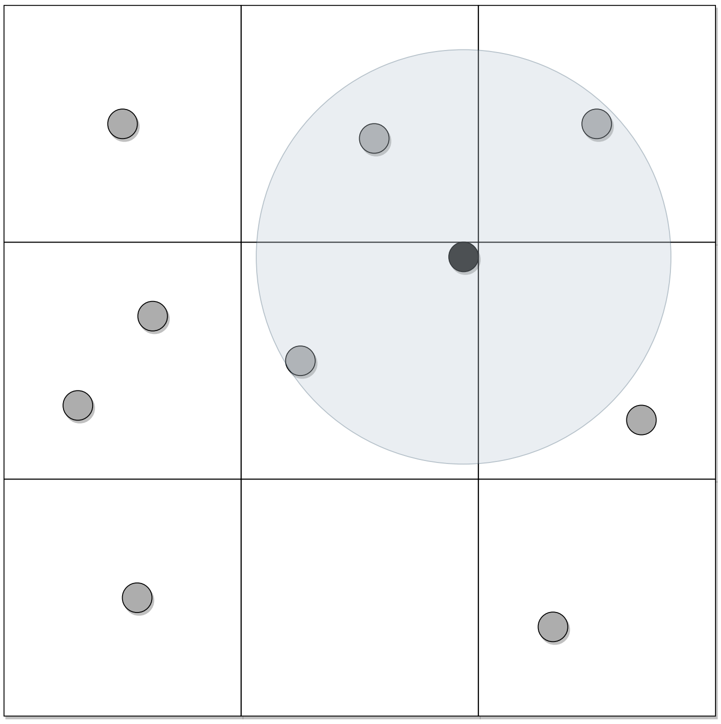


Figure 4.1: Illustration of cell list with smoothing length.

A cell will often be approximately the size of the interaction radius of the particle forces. This is because no matter the location of the particle in the cell, we only have to check the neighboring cells for interactions. While the naive approach compares all particles to all other particles, the cell list approach compares particles to all other particles in its cell and the 8 neighboring cells around it, shown in Figure 4.1. This greatly reduces the number of comparisons while using some extra operations on constructing the cell list.

4.2 Instruction Taxonomy & System Design

In this section, we introduce different computer architecture classifications with regards to handling of instructions and data. We also discuss design of systems in a heterogeneous or homogeneous way and scaling of systems.

Flynn (1966) proposes a classification system for different approaches to handling the execution of instructions and data on a computer architecture. The computing unit of the system can either process a single instruction or multiple instructions at the same time. These instructions can be used to alter single data or multiple data. The simplest kind of system is Single Instruction Single Data (SISD). One of the earliest and arguably one of the most popular computer architectures was the *von Neumann architecture*. This architecture consists of a main memory and a computing unit with a connection in between. Instructions and data are fetched from the main memory and executed one by one instruction at one data element at a time. This places the von Neumann architecture into the SISD category.

Although being some kind of derivative of the von Neumann architecture, most modern architectures are Multiple Instructions Multiple Data (MIMD) systems. This is because the computing unit has become more complex, with cores that can compute different instructions in parallel and cache memory that saves the cost of accessing main memory. These cores are independent of each other and can be scheduled different processes to run.

Graphical Processors Units (GPUs) were designed with throughput and not versatility in mind. To be able to achieve this throughput these systems can execute a single instruction on massive amounts of data in parallel. This generally places GPUs under the Single Instruction Multiple Data (SIMD) category. These architectures consist of many processor cores that all execute the same instruction set on data that has been distributed over the cores. Although modern GPUs are based on the SIMD architectures, most modern GPUs have a Single Instruction Multiple Thread (SIMT) architecture. This entails grouping threads and running them on the same Streaming Multiprocessor in parallel. This can be seen as *tiled SIMD* according to McCool et al. (2012). The grouping of threads is elaborated in Section 4.3.3.

In these systems, memory can be arranged in a shared manner, a distributed manner or a hybrid of the two. In a shared memory system the computing units have a connection to the same memory which they can fetch and store data on. In a distributed memory system the computing units all have their own local memory which they can fetch or store data on. Hybrid systems are often a unit of multiple computing units sharing memory through a connection while that memory is connected to other units of multiple processors sharing memory. This type of hybrid systems is similar to the modern MIMD multi-core processor system.

Heterogeneous systems describe a system that takes advantage of dissimilar components in order to gain performance or energy saving benefits. In contrast, Homogeneous systems describes a system that takes advantage of similar components for the same cause.

An example of a homogeneous system is a simple GPU, which is able to process a lot of data at the same time by having many of the same components running in parallel. However, modern GPUs have specialized components like Tensor cores or a ray-casting accelerator which makes them heterogeneous. In this thesis heterogeneous clusters are used, meaning that each of the individual computing nodes have components that are dissimilar to each other in order to gain benefits. In this case, the dissimilar components in the node is a CPU and two GPUs.

When increased performance is required, it is possible to scale a system. While vertical scaling indicates upgrading the system with more powerful components, horizontal scaling implies adding more machines to the pool of resources.

4.3 Programming Models

In this section, we describe the programming models that are used in this thesis.

Scalable programming models are important to the High Performance Computing (HPC) environment. Because the increase of performance in a single processor has started to reach its physical limitations, modern hardware takes advantage of multiple processors. The HPC environment takes advantage of multiple computing nodes with multiple processors in each of the nodes. This requires a programming model that can both utilize multiple processors and also scale with added computing nodes. McCool (2008) lists many scalable programming models for different platforms, design philosophies and processing models. Among these are **OpenMP**, **MPI** and **CUDA**. These models are used in this thesis.

4.3.1 OpenMP

Open Multi-Processing (OpenMP or OMP) is an API for developing parallel software with a shared memory architecture. OpenMP increases throughput by utilization of all available cores in a multi-core architecture, while abstracting most of the underlying execution of the parallel execution. This is done through compiler directives known as pragmas. A pragma with the keyword `OMP` indicates that the next block of code should be executed according to the directives or clauses following the keyword. One method is task constructing. This enables a block of code to be carried out as a task on another thread, independent of the other threads. The environment variable `OMP_NUM_THREADS` determines the number of threads used. If the variable is not set, the number of threads depends on the OpenMP implementation.

```c
#pragma omp for
for (int i = 0; i < N; i++) {
    // Do something
}
```

Listing 4.1: Example of worksharing in OpenMP.

The specification used in the implementation in this thesis is `parallel for`, which means that the workload of the following for loop is to be divided among a number of threads as shown in Listing 4.1. However, the inner part of the loop requires independent iterations, which is elaborate in Section 4.3.4. This method is referred to as worksharing.

This will spawn a number of threads that all execute the code block with different input, for example the loop index value. The threads will run their course before being synchronized with the main thread, as shown in Figure 4.2.

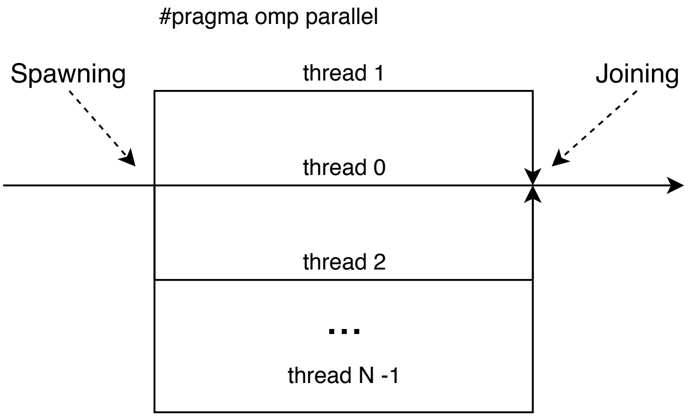


Figure 4.2: The execution of an OpenMP parallel block.

When using the default worksharing model, OpenMP makes the ”masterthread” execute the block of code with the rest of the threads. Using the `single` with the pragma, the ”masterthread” will spawn tasks from inside the code block when there are available threads, similar to a threadpool. This can also be done with the directive `taskloop` before a for loop, where the ”masterthread” will launch tasks of the loop body from the available threadpool.

4.3.2 MPI

MPI (Message-Passing Interface) is an API that enables inter-process communication. This tool is especially important for distributed memory architectures, where the processes can be located on different nodes in the system. When spawning a process, it is given a private address space by the Operating System. This is meant to provide security against other processes using the same memory space. MPI spawns a number of copies of the same process and gives each a unique rank. The processes can use this rank to communicate with other processes through Message-passing with methods like send, receive and broadcast. This allows the user to run multiple ranks per node over multiple nodes, sharing data and synchronizing.

Launching a process with MPI is done by calling the MPI program with the intended program as an argument, for example `mpirun ./program`. This creates an MPI process that governs the process and communicates based on MPI library calls from the process. This communication takes place inside a *communicator*. Although MPI creates a default communicator that all MPI-processes have access to, it is possible to create new communicators with configurable topology. This offers flexible communication and more control over communication patterns. When a process wants to pass a message to another process, it has to specify in which communicator they are communicating. When using a communication pattern that is not peer to peer like *send* and *receive*, the message is sent to all of the ranks in the given communicator. This is called *collective* operations and can be used with communication patterns like *broadcast* and *reduce*.

In the same manner as processes and resources, communication can create deadlocks when they are blocking. An example of this is if both senders have a blocking send call before trying to receive from each other. Both senders will wait for the other to receive their message before continuing execution. Therefore, scheduling of messages is important to make sure that deadlocks are avoided. To solve this, MPI offers the `MPI_Sendrecv` library function. This automatically schedules the send and receive so that deadlock does not occur.

4.3.3 CUDA

CUDA (Compute Unified Device Architecture) is an API and a parallel computing platform created by Nvidia for developing general purpose programs on Nvidia GPUs. It gives the programmer direct access to the GPU's virtual instruction set, and exposes the hardware layout so the programmer can choose how the program will run on the hardware.

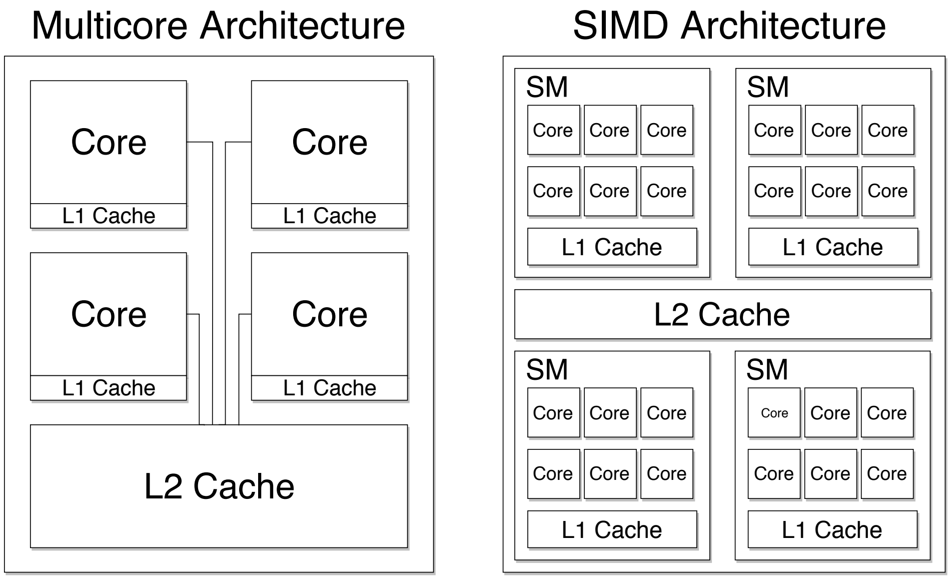


Figure 4.3: Simplified architecture of a CPU (left) and GPU (right).

An important part of how CUDA is utilized has to do with the SIMD layout of the hardware. Calls that are invoked from the CPU can copy data and run CUDA kernels on the GPU. Kernels and data can either be defined as `global` or `device`. `global` can be accessed from both CPU and GPU, while `device` is only accessible from the GPU. Figure 4.3 shows that the layout of GPU architectures are different from CPU architectures. Because of this, it is necessary to inform the hardware how to execute kernels. Listing 4.2 shows an example of launching a CUDA kernel.

```
dim3 grid(gx, gy, gz);
dim3 block(bx, by, bz);
some_kernel<<< grid, block>>>(parameters...);
```

Listing 4.2: Example of launching a CUDA kernel.

The block is a three dimensional array that describes how many threads per block the kernel requires. The grid is a three dimensional array that describes how many blocks the kernel requires. These arrays can also be two or one dimensional. The block is an abstraction for a streaming multiprocessor which contains cores that can execute the same instructions in parallel on different data, shown in Figure 4.4.

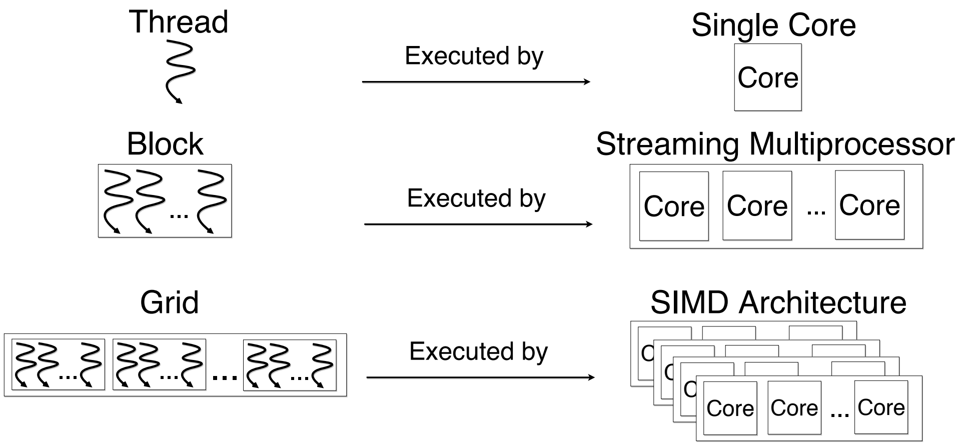


Figure 4.4: How a kernel is executed from a software perspective.

Because all code executed by the individual execution cores of a streaming multiprocessor needs to execute the exact same instructions we need to be certain that all threads in that core executes the same instructions. When running a thread block, all the threads get grouped into groups of 32 threads called a warp. When threads in a warp take diverging branches, threads will want to execute different instructions. This is handled by executing all the threads in the warp that take the same path before executing the rest of the threads that take the other path. This ensures that only the same instructions are executed in parallel. If there are many paths executed by the threads, there will be fewer threads executed in parallel which will limit the parallelism of the code. Therefore, branching is avoided as much as possible in GPGPU code. Listing 4.3 shows an example of a kernel and how it accesses identifiers.

```
__global__ void some_kernel(parameters...) {
    int block = blockIdx.x;
    int thread = threadIdx.x;
    int block_size = blockDim.x;
    int unique_1d_number = (block * block_size) + thread;
    // Do something
}
```

Listing 4.3: Example of a CUDA kernel and it's identifiers.

4.3.4 Mutual Exclusion

When multiple threads in a multi-core system request the same resource, a race condition can occur. This can cause data to become incorrect due to the non-deterministic nature of scheduling processes as shown in Table 4.1.

Thread 1	Thread 2	Data	Thread 1	Thread 2	Data
		1			1
read		1	read		1
	read	1	increment		1
increment		1	write		2
	increment	1		read	2
write		2		increment	2
	write	2		write	3

Table 4.1: An example of incorrect result (left) due to race condition and correct result (right).

One solution to race conditions is mutual exclusion. This normally takes form as *locks* and *atomic operations* which can have both hardware and software implementations. Both CUDA and OpenMP offer atomic functions to prevent race conditions. OpenMP also offers *locks* and *critical section*. While critical section ensures that only one thread is allowed to execute the section at a time, atomic operations ensures determinism in read/write operations. This makes atomic operations more restrictive, but potentially faster than the critical section. Locks prevent the use of the same lock at the same time for all threads. A thread can take hold of an initialized lock and until the thread stops holding the lock, other threads that are requesting that same lock must wait. This makes locks the most flexible of the methods of mutual exclusion mentioned. However, all of the methods slow down performance relative to code that does not have to take mutual exclusion into account.

4.3.5 Hybrid systems

Combining the different abilities of the programming models mentioned, we can create hybrid systems. MPI allows the workload to be shared between processes and nodes while OpenMP and CUDA allows for compute parallelism on local nodes. Rabenseifner (2003) looks at different strategies for implementation of OpenMP and MPI in the HPC environment and proposes a classification of hybrid systems with pure MPI in one end and pure

OpenMP in the other end, and combinations of the two in between. By the hybrid classification, this proxy application implementation falls under the *Hybrid masteronly* class where MPI calls are only done outside of parallel regions. Because the CUDA implementation uses MPI for inter-process communication in the same manner as the OpenMP implementation, both fall under the Hybrid masteronly classification. Although this class is one of the easier to implement in terms of concurrency and synchronization, it has three drawbacks. Firstly, while the MPI communication takes place, only one thread is active while the rest are idle. This results in low utilization of the hardware whenever communication is done. Secondly, only having one thread sending packages through MPI will not take full advantage of the bandwidth of the interconnect. Thirdly, it introduces additional overhead in terms of OpenMP or CUDA synchronization and through flushing the cache of the message passing routines. Having all threads communicate will leave most cores on the CPU idle while sending and receiving and therefore a balance between communicating threads and non-communicating threads should be implemented. However, classes that have overlapping communication are immensely demanding to implement according to Rabenseifner (2003).

4.4 Performance Models

In this section, we describe some useful models for performance and scalability.

Barker et al. (2009) introduces a methodology for accurately designing and modeling performance of large scale applications. This enables modeling of the different life cycles of an application, such as designing the application, implementing the application, better understanding what hardware should be bought and used for the application, installing the application, improving the application and maintaining the application. With this in mind, this thesis falls under the implementation stage of the life cycle.

Valiant (1990) presents the Bulk Synchronous Parallel (BSP) model. This model is used for both theoretically and experimentally analyzing efficiency of iterative solutions. This include sparse linear systems, molecular dynamics and partial differential equations on a discrete grid. Molecular dynamics is similar to SPH, which also is a BSP problem.

Models can utilize both theoretical and empirical values. Theoretical values can be found in datasheets from the manufacturer or derived from static analysis. Empirical values can be found using benchmarks. Two useful benchmarks are SGEMM and STREAM. Single-precision General Matrix Multiplication (SGEMM) is a standard part of Basic Linear Algebra Subprogram (BLAS) packages, proposed by Dongarra et al. (1990), and is useful to measure peak Floating Point Operations per second (FLOP/s) of a system. This is because the algorithms used are compute bound and high on operational intensity. STREAM is a benchmark proposed by McCalpin et al. (1995) to measure bandwidth of a system. This is done by using four operations that vary in the number of bytes as well as the operational intensity. The bandwidth for an operation can be calculated by dividing the number of bytes loaded over the runtime of the operation.

Amdahl's law comes from an observation made by Dr. Gene Amdahl in Amdahl (1967) and states that unless a given program can be fully parallelized, its Speedup will be limited no matter the number of processors utilized. This can be used to determine theoretical speedup in a multi-processor system and calculate parallel execution time. This signifies that even for perfectly parallelizable parts the speedup will always be limited by the serial part of the program. However, this does not take into account the problem size of the program being run. This is what Gustafson (1988) did when he reevaluated Amdahl's law and found that the execution time of the serial fraction of the program tends to decrease as the problem size increases.

4.4.1 Fundamental Equation

A fundamental equation of modeling was introduced by Barker et al. (2009), shown in Equation 4.1. This equation breaks down, from a top-down approach, the runtime of large-scale applications into three parts. The computations done on the data, the communication between the nodes and the overlapping region between communication and node-local computation.

$$T_{total} = T_{computation} + T_{communication} - T_{overlap} \tag{4.1}$$

4.4.2 Hockney

The Hockney model was proposed by Hockney (1994) as a means to approximate the time it takes to send a message. The model breaks down communication as latency and the time it takes to send the message itself from sender to receiver. The total time used on communication is the sum of all communication between the two same end points, which can be seen in Equation 4.2.

$$T_{comm} = \sum_{i=1}^{N} \alpha + M_i \beta^{-1} \tag{4.2}$$

The transfer time is dependant on the bandwidth in between the end points and is denoted with β. Another factor of the transfer time is the size of what is being transferred and is denoted by M_i. The latency is the overhead time of establishing a communication channel between the end points as well as the time used to create data packets and extract data from the data packets. Latency is denoted by α.

In a heterogeneous system, because latency and bandwidth may vary between processors and connections, α and β can be extended to a neighbor matrix between processors α_{ij} and β_{ij}. This is known as the Heterogeneous Hockney model, defined in Lastovetsky et al. (2010).

4.4.3 CUDA Models

Performance modeling of GPGPUs is still not as well established as for CPUs. CUDA was released in 2007, enabling acceleration of general purpose programs on GPUs. Soon other application programming interfaces (API) were introduced like OpenACC and OpenCL. This has made offloading onto accelerators like GPUs an easier task and introduced the need for performance modeling for GPU architectures, because of the large differences in the architecture of GPUs and CPUs.

Kothapalli et al. (2009) presents a prediction performance model for the CUDA GPGPU platform. The model is similar to asymptotic analysis and can be used to give an estimate for the performance of a given CUDA kernel with a given GPU architecture. We use this model to predict the performance of the CUDA kernels.

Baghsorkhi et al. (2010) presents an adaptive performance modeling for GPU architectures. They take impactful aspects of running GPGPU programs and proposes a detailed model to accurately predict performance. This can be used as a tool to profile and improve applications as well.

4.4.4 Roofline

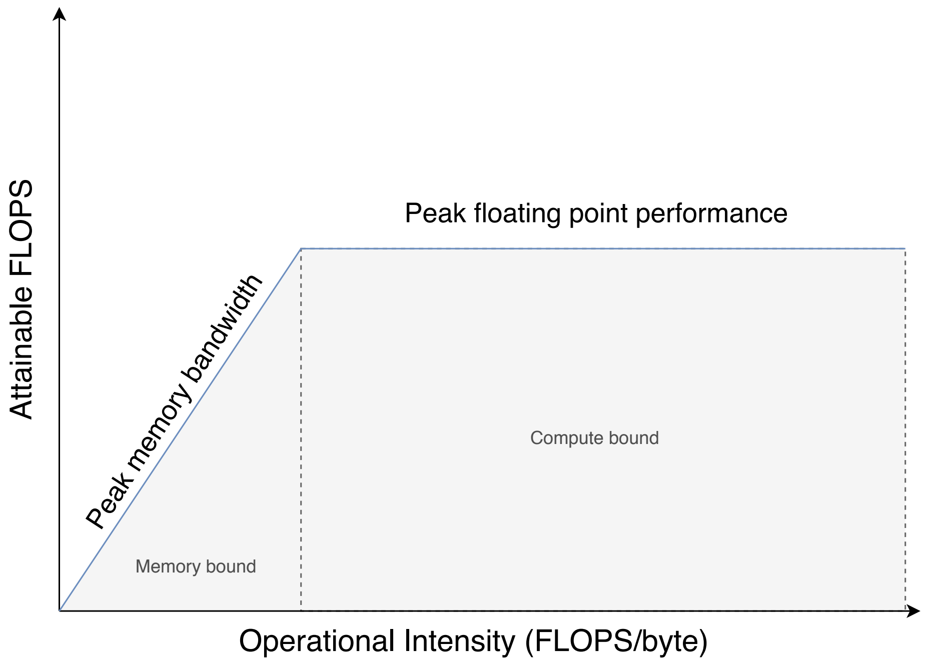

Attainable FLOPS
Peak floating point performance
Peak memory bandwidth
Compute bound
Memory bound
Operational Intensity (FLOPS/byte)

Figure 4.5: The Roofline model

When running programs on hardware, the performance of that program is limited by the hardware. For a given program and hardware, we say that the program is compute-bound if the required data is loaded and ready to be computed. If the computation is dealt with faster than the memory can provide the data needed for the computation, we say that the program is memory-bound. In this case, the CPU will stall waiting for the data to be loaded. Williams et al. (2009) introduces the Roofline model, which can be used to model performance of programs, showing if the program is memory or compute-bound. The model has a ceiling of the peak floating-point performance of the given hardware and a slope leading up to it signifying peak memory bandwidth, shown in Figure 4.5. Programs appearing closer to the ”roof” of the model are seen as compute-bound, while programs appearing further away from it are considered memory-bound. Roofline can be a tool used for changing programs to run better on hardware or to figure out what aspects are important in choice of hardware when running a specific problem.

4.4.5 Speedup and Efficiency

Speedup and Efficiency are two important terms in parallel programming. Speedup is defined by Pacheco (2011) as the execution time of the given program in serial divided by the execution time of the given program in parallel, as shown in Equation 4.3.

$$S = \frac{T_{serial}}{T_{parallel}} \tag{4.3}$$

Speedup can give a sense of how much the program can benefit from parallelism, but does not take into account how effective it is in the use of the resources of the system. Given p cores that is being used to execute the program in parallel, if that program is able to utilize all of the cores to peak performance, Speedup value should be equal to p. This is called a *linear speedup*, which signifies that the work is equally divided among the cores, and there is no overhead added by running on multiple cores. In practice, linear speedup is close to impossible to achieve. Even if the workload is equally divided, the overhead of running on multiple cores is hard to avoid. The more cores added to the execution, the more overhead is created. The full utilization of multi-core systems is impossible as stated by Esmaeilzadeh et al. (2011). Because the maximum value of speedup is p, we can derive the efficiency equation by dividing Equation 4.3 by p, as shown in Equation 4.4. This equation will give us a number between 0 and 1, in which we can gauge the efficiency of the parallelization.

$$E = \frac{S}{p} = \frac{T_{serial}}{p \times T_{parallel}} \tag{4.4}$$

Chapter 5

Methodology

This chapter describes the implementation of the SPH proxy application as well as the implementation details of finding neighbors, the bottleneck of the proxy application. A static analysis is performed on the proxy application as a whole, and performance models are devised. The performance parameter space is introduced for the two scalable axes of this thesis: architectures and the problem size. Finally, the method concerning the design of the experiments is described.

5.1 Implementation

In this section we will introduce the implementation of the proxy application as well as the different methods for finding neighbors. All the code listings in this section are simplified. A selected part of the source code can be found in the appendix. The proxy application is based on a version developed by Ragunathan and Valstad (2018). This version is a OpenMP + MPI hybrid, using what Rabenseifner (2003) describes as the Masteronly strategy.

The area of the dam is equally divided between the ranks into areas called subdomains. Because of the nature of worksharing through MPI and in order to have a correctly flowing fluid, the overlapping area in between the subdomains has mirror particles from the neighboring subdomain. This area is shown in Figure 5.1. The mirror particles are sent in between the subdomains so their physical properties work on particles in both subdomains, ensuring correct flow of particles through subdomains despite the divide. This transition is called a ”Border Exchange”

Ghost particles are generated on the other side of the boundaries of the outer box. These particles mirror the actual particles and therefore provide a wall like effect when a particle hits a wall, because of the Neumann boundary condition.

Figure 5.2 shows the flow chart of SPH applications. We can see that the fluid simulation part consists of five steps: `Find Neighbors`, `Kernel`, `cont_density`, `correction`, `int_force` and `ext_force`. `Find Neighbors` iterates over the particles, finding which particles affect each other and makes a list of particle pairs that are used for the other steps.

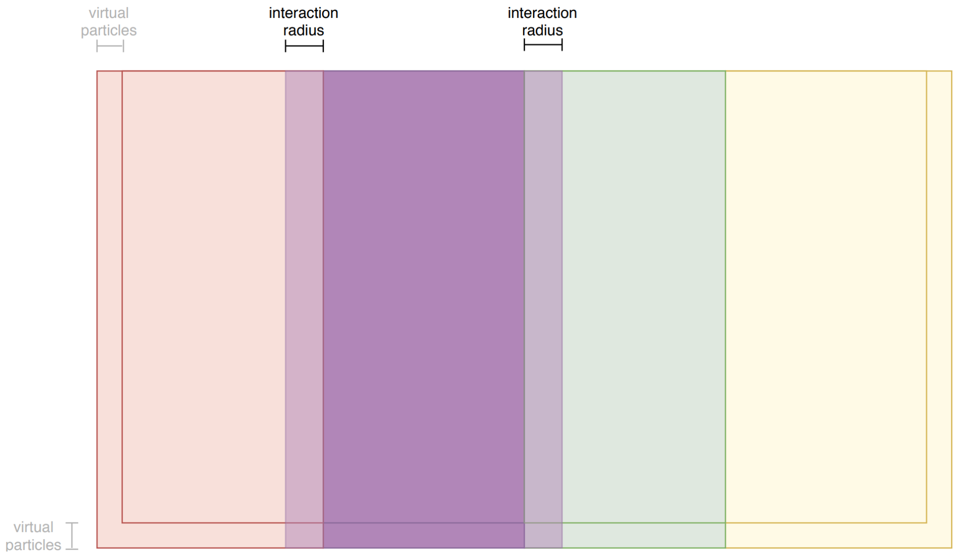


Figure 5.1: Overlap of subdomains in which copies of particles from the neighboring subdomains exist. Reproduced from Ragunathan and Valstad (2018) with permission.

Ragunathan and Valstad (2018) found that the `Find Neighbors` usually accounts for somewhere between 60% and 95% of the execution time, depending on the `SCALE`, making it the performance bottleneck of the application.

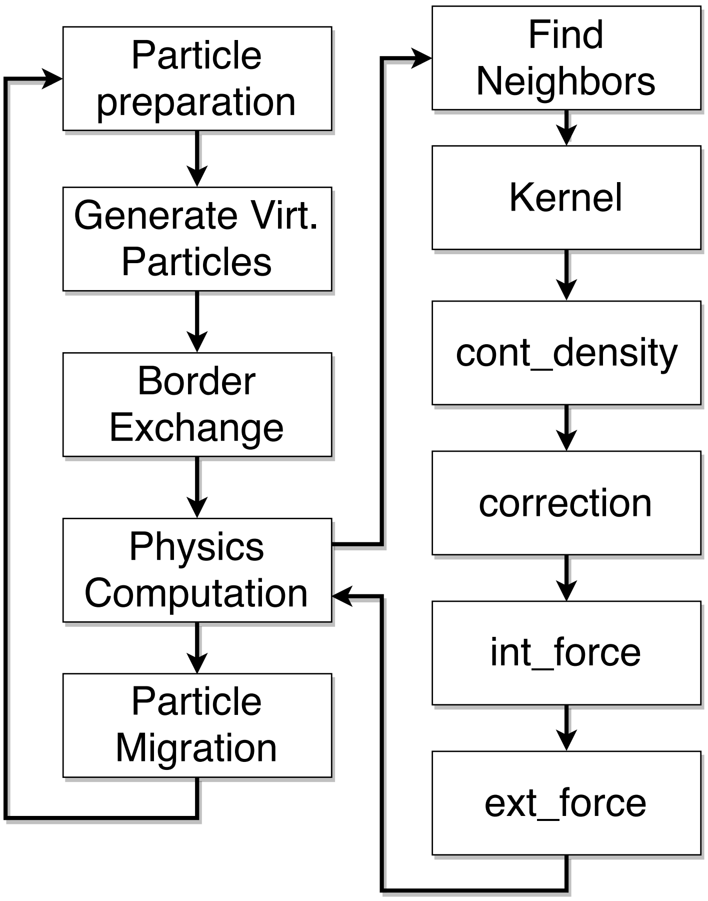


Figure 5.2: Life cycle of the SPH proxy application.

5.1.1 Data structures

The **Particle** data structure consists of one local subdomain ID, one global ID and the ID of the cell it belongs to if the cell-list method is used. The structure also has the physical properties of the particle, such as position, velocity, mass, density, pressure and differential values. This data structure has a size of 104 bytes in our implementation.

The **Pair** data structure contains the IDs of the two particles in the pair and the physical properties of the pairing, such as Euclidean distance between the particles and influence on velocity that the pairing has. This data structure has a size of 40 bytes in our implementation.

The **Cell** data structure is different for the CPU and the GPU implementation. The cell structure of the CPU implementation contains a pointer to a particle and a pointer to the next cell, creating a linked list. The cell structure of the GPU implementation is a matrix where the rows correspond the cells and the elements in the matrix is the particle ID of a particle belonging to that cell.

5.1.2 Finding Neighbors

This section will discuss the implementation of the two different approaches to finding neighbors, namely:

- the naive approach
- the cell-linked list approach

Because the proxy application is targeted towards distributed memory architecture, the nature of the GPGPU part is as well. On each node, the MPI ranks will distribute themselves over the available GPUs. The implementations are exactly identical, except the physics time step. The GPGPU implementation offloads the various parts of the time step to the GPU while the CPU version sets the number of threads equal to the number of threads available on the node.

While the CPU version reallocates the pair list when the allocated amount is surpassed, dynamic memory allocation on GPU is more challenging, and therefore estimations and pre-allocation are used instead of reallocation. If the particles could flow around freely without physical interaction, the theoretical upper bound of pairs for one particle is $P-1$, P being the number of particles. However, according to Ozbulut M., Arslan T. (2018), because the physical forces do not allow the particles to occupy the same space as well as push other particles away, the upper bound of particles-pairs is estimated to 300. This brings down the memory usage of allocating pairs from $\frac{P(P-1)}{2}$ to $300 \times P$.

A normal approach of using GPGPU to offload problems is to offload the bottleneck of the problem, in this case finding neighbors. However, this would require the memory transfer of P from DRAM to the GPU and $P \times 300 + P$ back from the GPU to DRAM when it has found the pairs. Because the pairs are only needed for the physics computation, it is not necessary to transfer the pairs back if all of the computation is done on the GPU. This leaves the total memory transfer of $2 \times P$ instead of $302 \times P$.

Naive

The naive (or brute-force) implementation only allocates memory space for the particle, the interaction counter and pair-list. This makes it the most memory efficient implementation. After transferring the particle data, it creates the pair list as shown in Listing 5.1. Both the GPU and the CPU implementations have a global counter for the current index of the pair-list which also serves as a length counter. The counter is incremented by calling the CUDA function `atomicAdd` or the OpenMP pragma `#pragma omp atomic capture`, because more than one thread can read the value of the counter at once. This function achieves thread and memory atomicity, and returns the old value to the caller. The CUDA kernel is called distributing the particles over threads and blocks, considering that a thread block can only have 1024 threads running at the same time. Similarly, the CPU version workload shares the particles over the available threads on the system. This implementation does not require any more setup than the allocation of memory, and therefore uses no time on setup for computation.

```
__global__ void naive(...) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int j = blockDim.y * blockIdx.y + threadIdx.y;
    float difx = particles[i].x[0] - particles[j].x[0];
    float dify = particles[i].x[1] - particles[j].x[1];
    // If distance is less than radius
    if (sqrt(difx*difx + dify * dify) < RADIUS) {
        // Add pair of particle i and particle j to pair list
    }
}
void naive() {
    #pragma omp for
    for ( int_t i=0; i<n_total-1; i++ ) {
        for ( int_t j=i+1; j<n_total; j++ ) {
            float difx = particles[i].x[0] - particles[j].x[0];
            float dify = particles[i].x[1] - particles[j].x[1];
            // If distance is less than radius
            if (sqrt(difx*difx + dify * dify) < RADIUS) {
                // Add pair of particle i and particle j to the pair list
            }
        }
    }
}
```

Listing 5.1: The naive approach to finding neighbors

Cell List

The cell list method has to both allocate more space and do more setup before computation. Each cell has to have a list of particles that resides within the cell. The approach to the data structure of the list differs from the CPU and GPU implementation. The CPU uses a linked list to keep track of the particles in one cell. It will gradually add particles with positions corresponding to the cell on the end of the linked list using an OpenMP lock. This is memory efficient, because it uses no more space than the actual number of particles that occupy the cell. However, distributing and traversing linked lists on GPU-type architectures is slow. Therefore estimates and pre-allocation is used. Given that a particle will have at most 300 pairs and a particle can only have pairs in neighboring cells, we can calculate the upper bound of particles in a cell. Equation 5.1 shows that a cell, based on the interaction area and area of the cell, have an upper bound of 96 particles. Because of this, the cell list method uses an additional 96 $\times$ n_cells of memory. The cell list is a 2D list, row major, that is divided into cells of `N_CELLS_X` horizontally and `N_CELLS_Y` vertically, defined by Equations 5.2 and 5.3.

$$\begin{aligned} cell_{area} &= Radius^2 \\ particle\_range_{area} &= \pi Radius^2 \\ particle_{coverage} &= \frac{particle\_range_{area}}{cell_{area}} = \pi \\ \implies particle\_per\_cell &= \left\lceil \frac{300}{\pi} \right\rceil = 96 \end{aligned} \tag{5.1}$$

$$N\_CELLS\_X = \left\lceil \frac{subdomain_{end} - subdomain_{begin} + 2R}{R} \right\rceil \tag{5.2}$$

$$N\_CELLS\_Y = \left\lceil \frac{1.5T + 1.55H}{R} \right\rceil \tag{5.3}$$

Equation 5.2 shows that the number of horizontal cells are dependent on the size of the subdomain belonging to the current rank. The additional 2 interaction radii account for the overlapping area in between ranks. The subdomain and the extra radii are divided into the interaction radius of the particles for the benefits discussed in Section 4.1.2, page 14.

Because the fluid does not have an upper bound on the vertical axis, we simplify by making the total area of the cells up to 50% higher than the initial height of the dam and 55% lower, to accommodate for the ghost particles created outside of the boundary below as shown in Equation 5.3. The particles potentially located above the uppermost cell is put into the uppermost cell.

Since both equations are divided by the interaction radius, the cells become a square with the interaction radius as width and height. We can then calculate the cell coordinates of the particles by Equations 5.4 and 5.3. This is done each time step on both implementations. The cell ID is calculated by Equation 5.6 and stored in each particle data structure.

$$cell\_x = min(\frac{p_x - subdomain_{begin} + R}{R}, N\_CELLS\_X - 1) \tag{5.4}$$

$$cell\_y = min(\frac{p_x - 1.55H}{R}, N\_CELLS\_Y - 1) \tag{5.5}$$

$$cell\_id = cell\_y + N\_CELLS\_Y \times cell\_x \tag{5.6}$$

To save additional setup and allocation time in the GPU implementation, the cell's memory allocation is not freed before the application is finished. Since we only read from the part of the cell's particle list that have been written to, we only need to reset the counter of the cell. This can be done with a call to `cudaMemset`. The CPU implementation needs to free the data structure that holds the pointers to the particles that occupied the cell in the previous iteration. This is done by simple recursion and the `free` function of C/C++.

```
__global__ void fill_cells(...) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int cell_id = particles[i].cell_id;
    // Add particle ID to cell[cell_id]'s list
}
void fill_cells(...) {
    #pragma omp for
    for (int i = 0; i < n_total; ++i) {
        // Lock possible critical section
        omp_set_lock(&(lock[cell_id]));
        // Create a new link in the linked list
        // Add particle to new link
        // Unlock
        omp_unset_lock(&(lock[cell_id]));
    }
}
```

Listing 5.2: Fills the cells with particles.

Listing 5.2 shows filling of the cells for both GPU and CPU implementations. The CUDA kernel is distributed over the particles, each thread reading the cell ID and adding the index of the particle into the particle list of the cell. The list is allocated as a 2D memory aligned matrix with the `cudaMallocPitched` function. This method uses the `atomicAdd` function to increment the cells counter, because more than one thread can try to access it at once. Another approach to filling the cells could use shared memory among the thread block and distribute the cells over thread blocks. Accessing shared memory is faster than accessing global memory and will therefore have a locality benefit. Writing to the array can be done by either having a shared counter and using atomic adding, or having one index per thread and compacting the array after all particles are found. The shared array of particles can be copied at the end of the kernel by one thread to the cell in question. Instead of atomic functions, the CPU version uses OpenMP locks to make sure that only one thread at a time accesses the same cell and particle.

Cell List Pair Creation

The method for creating pairs is very similar for both the CPU and GPU versions. The functions compare one particle with the cell the particle is in and the particles in the 8 neighboring cells, shown in Listing 5.3. The GPU version distributes each particle over the thread blocks so that each thread has its own particle, while the CPU version shares the number of particles over the available number of threads. This is done by calling the function shown in Listing 5.4. Both versions use atomic operations to increment the counter of pairs before writing pairs to the pair list.

```
__global__ void find_pairs(...) {
    int p_id = blockDim.x * blockIdx.x + threadIdx.x;
    particle_t* particle = &particles[p_id];
    int cell_id = particle->cell_id;
    // Compare particle with all the particles in this cell
    compare(cell_id, particle, ...);
    // and all the cells around it
    compare(cell_x + 1, cell_y +1, particle);
    ...
    compare(cell_x -1, cell_y -1, particle);
}
void find_pairs(...) {
    #pragma omp for
    for (int_t i = 0; i < n_total-1; ++i) {
        particle_t* particle = &list[i];
        int cell_x = particle->cell_x;
        int cell_y = particle->cell_y;
        // Compare particle with all the particles in this cell
        create_pairs(cell_id, particle, ...);
        // and all the cells around it
        create_pairs(cell_x+1, cell_y+1, particle, ...);
        ...
        create_pairs(cell_x-1, cell_y-1, particle, ...);
    }
}
```

Listing 5.3: The cell list approach to finding neighbors.

```
__device__ void compare(...) {
    for (int i = start; i <= end; i++) {
        particle_t *other_particle = &particles[i];
        float difx = particle->x[0] - other_particle->x[0];
        float dify = particle->x[1] - other_particle->x[1];
        // If distance is less than radius
        if (sqrt(difx*difx + dify * dify) < RADIUS) {
            // Add pair of particle i and j to pair list
        }
    }
}
void compare(...) {
    cell_t* current = cells[(bx, by)];
    while (current != NULL && current->particle != NULL) {
        float difx = particle->x[0] - current->particle->x[0];
        float dify = particle->x[1] - current->particle->x[1];
        if (sqrt(difx*difx + dify * dify) <= RADIUS) {
            // Add pair of particle i and j to the pair list
        }
        current = current->next;
    }
}
```

Listing 5.4: Comparing a particle with all the particles of a given cell.

The GPU version method has an advantage in fetching the index of the particles because the matrix used has aligned memory. This means that the data is padded, after every row, so that each row is only in its own memory blocks. Accessing each row will be faster, because some of it can be accessed in parallel by the memory control unit. However, the CPU version has an advantage of using less space for the cell data structure, because the links in the linked list are created on demand. Given that the transfer rate of the internal bandwidth of GPUs are generally higher than the bandwidth connecting the DRAM to the CPU, this gives the GPU version an upper hand in fetching memory. This is beneficial in this proxy application, because it is memory-bound and not compute-bound, according to the Roofline model shown in Section 5.2.1.

Both implementations use a list of data structures for both particles and pairs. Using a list of data structures is slower than a data structure of lists in many cases, because we often iterate over lists in local regions of the list which lends itself to the locality principle.

5.2 Performance Models

5.2.1 Roofline Analysis

Using the Roofline model, proposed by Williams et al. (2009), we can discern if the proxy application is memory-bound or compute-bound. The model defines operational intensity as the number of floating point operations per byte of memory traffic and can be seen in Equation 5.7. The term W represents the number of floating point operations while Q represents the number of bytes of memory traffic needed to complete these operations.

$$I = \frac{W}{Q} \tag{5.7}$$

We can model the Roof of the model by using Equation 5.8. The term β signifies the maximum bandwidth of the system, ν refers to the maximum floating point operations per second of a given system, and I is the operational intensity.

$$y = min(\nu, \beta \times I) \tag{5.8}$$

If the bottleneck of the application is memory-bound, the whole application will be memory bound as well. Therefore we will find the Roofline value of the bottleneck, finding neighbors. Using the hardware specifications from Table 5.3 for the Intel Xeon E5-6132 v5 and the code for the naive approach, we can discern that the inner loop requires 100 bytes of memory access and 68 floating point operations (using Table 5.2 for the number of cycles per operation). When increasing `SCALE` we increase the number of particles and therefore the number of comparisons needed. The maximum bandwidth of the CPU is $\beta = 119GB/s$ We can calculate the FLOPS value of the CPU to be product of the number of floating point accelerators it has, the number of cycles per floating point operation and the base frequency of a single core. This gives us

$$\nu = 2 \times 32[\frac{flop}{cycles}] \times 2.6[GHz] = 166[GFLOPS] \tag{5.9}$$

However, because the number of comparisons increase both W and Q by the same factor, it can therefore be factored out. This also implies that the Roofline analysis is independent of `SCALE`. Using the stated values of β and ν with Equation 5.7 and 5.8 we can visualize the Roofline model as seen in Figure 5.3. Here we see that the application is in the memory-bound region of the model and therefore is memory-bound.

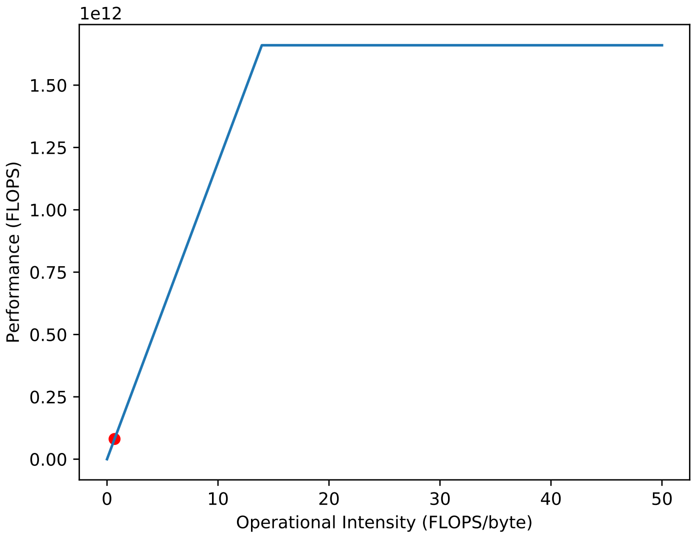


Figure 5.3: The Roofline analysis of the WCSPH proxy application bottleneck for the V100 cluster.

5.2.2 Static Analysis

The significant components of this proxy application is the time it takes to compute the physics and the time it takes to communicate with other nodes in order to send particles over subdomains. Because computation and communication have no overlap, we can generalize the execution time of the proxy application using the fundamental equation of modeling from Section 4.4 as shown in Equation 5.10.

$$T_{total} = T_{communicate} + T_{compute} \tag{5.10}$$

Compute

According to Ragunathan and Valstad (2018), finding neighboring particles are the bottleneck of the SPH proxy application. As such, we can divide the compute step into two steps: finding neighbors and using the pairs found to propagate the physics simulation as shown in Equation 5.11.

$$T_{compute} = T_{find\_neighbors} + T_{physics} + T_{transfer} \tag{5.11}$$

When offloading to the GPU, all particles in the subdomain must be transferred twice. Once to the GPU before finding neighbors and once from the GPU after the particles have been propagated another step in the physics simulation. This means that the cost of the transfer for one iteration can be written as shown in Equation 5.12. The constant at the end of the expression is the cost of transferring to the GPU and involves bandwidth and size of the particle data object. When using the performance model for the CPU version, the transfer-term, $T_{transfer}$, is set to zero.

$$T_{transfer} = 2P \times C_1 \tag{5.12}$$

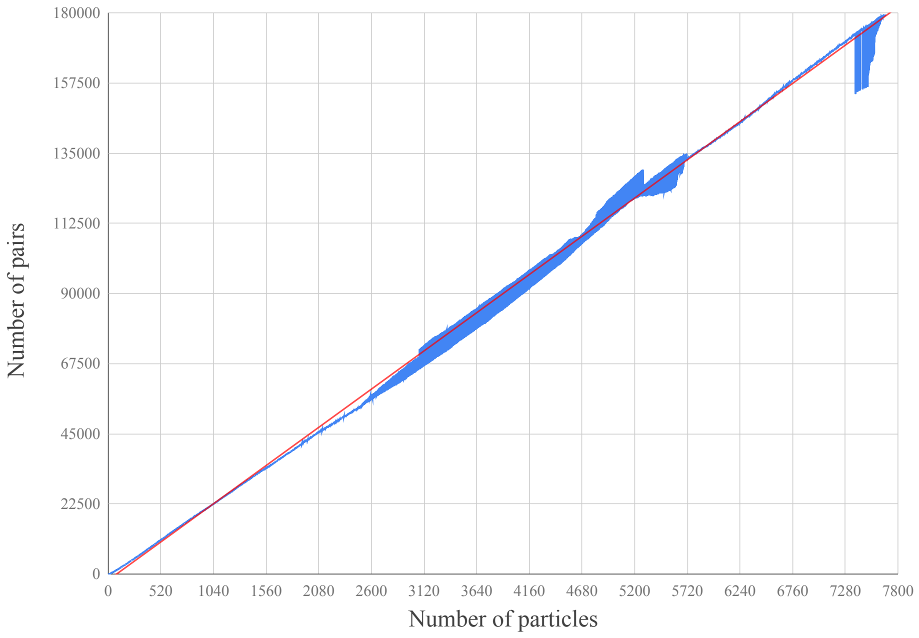


Figure 5.4: The relationship between number of particles and number of pairs found with regression line of $f(x) = 23.6x \pm 1966$ and $R^2 = 0.993$.

In order to discuss the performance of the $T_{physics}$ term as well as memory usage of the proxy application, we have to be able to estimate how many pairs each particle will have. This is because the physics kernels iterates over the number of pairs. Memory-wise, the pairs are the most significant part of the memory because there will be more pairs than particles. We have an upper bound of pairs per particle, but the upper bound is likely never reached and will therefore not give a realistic image of the memory usage. Figure 5.4 shows a clear trend between number of particles and number of pairs so that we can use the estimation of $\frac{n\_pairs}{n\_particles} = 23.6$. We call this estimate e_p, defined by Equation 5.13 which we can use to estimate the number of pairs with Equation 5.14.

$$e_p = 23.6 \tag{5.13}$$

$$N_{pairs}(P) = e_p \times P \tag{5.14}$$

Finding neighbors mainly consists of creating the pair list. However, if we offload finding neighbors to the GPU, we have to account for the time it takes to transfer the particles to the GPU, and similarly, if we use more sophisticated algorithms for finding neighbors, we have to account for the time it takes to setup the algorithms. This can be seen in Equation 5.15. T_{setup} would be zero if a naive approach is used.

$$T_{fn} = T_{setup}(P, alg) + T_{create\_pairs}(P, alg) \tag{5.15}$$

Equation 5.16 depends on the number of comparisons needed when comparing all particles to all others. As mentioned in Section 4.1, this can be reduced as shown in Equation 5.17. If we construct a $P \times P$ matrix for comparing all the particles to each other, we only need to check the upper or lower triangular form of the matrix. All of the models will therefore use Equation 5.17. The C_2 term from both equations denotes the throughput of the architecture that runs the comparisons.

$$T_{compare}(P, naive) = P^2 C_2 \tag{5.16}$$

$$T_{compare}(P, naive) = \frac{P(P-1)}{2} C_2 \tag{5.17}$$

Equation 5.18 shows the static analysis of the naive approach. The setup term is not present in this analysis because the naive approach does not require any work upfront like the cell-linked list needs.

$$T_{naive} = T_{compare}(P, naive) \tag{5.18}$$

Equation 5.19 shows the static analysis of the cell-linked list method. The equation shows how the cell list method reduces execution time by the number of cells used. The number of cells, however, cannot be smaller than the interaction radius, because we will not be guaranteed the correct pair list which will result in an invalid fluid simulation. The setup term is elaborated in Equation 5.20. The n_cells term represents the number of cells and is equivalent of $N\_CELLS\_X \times N\_CELLS\_Y$.

$$T_{cell} = \frac{T_{compare}(P, cell)}{n\_cells} C_3 + T_{setup}(P, cell) \tag{5.19}$$

$$T_{setup}(P, cell) = T_{setup}(P, naive) + T_{fill\_cells}(P) \tag{5.20}$$

$$T_{fill\_cells}(P) = PC_4 \tag{5.21}$$

The physics computation takes the pairs that were found and uses them to calculate forces on the particles in order to step the simulation further. The components of the physics computation can be seen in Equation 5.22. The components linearly iterates over either the number of particles or the number of pairs. Table 5.1 shows what parts are bound by particles and what parts are bound by pairs. In terms of performance modeling, all of the terms would have a constant that is multiplied with the number the term is bound by. The constant represents the number of operations and hardware specifications.

$$T_{physics} = T_{kernel} + T_{cont\_density} + T_{correction} + T_{int\_force} + T_{ext\_force} \tag{5.22}$$

To model the different CUDA kernels in this thesis, we have used the performance model proposed by Kothapalli et al. (2009) as shown in Equation 5.23. In this equation,

Part	**Bound by**
Kernel	N_{pairs}
Cont_density	N_{pairs}
Correction	N_{pairs}
Internal force	N_{pairs}
External force	$N_{particles}$

Table 5.1: The components of the physics computation, seen in Equation 5.22, and what they are bound by in terms of complexity.

N_b is the number of thread blocks per Streaming Multiprocessor (SM), N_w is the number of warps per block, N_t is the number of threads per warp, C_t is the maximum number of cycles it takes for any thread, N_c is the number of CUDA cores per SM, D is the depth of the core instruction pipeline and GPU_{Hz} is the clock rate of the GPU. A kernel thread may be held back by either high computational intensity or high memory access intensity. The scheduler will try to hide latencies due to overuse of one or the other. By the best effect of the scheduler, the number of cycles of a thread can be expressed as $max(N_{comp}, N_{memory})$, where N is the number of cycles required by a thread of the different types. Kothapalli et al. (2009) calls this the max model and further suggests that if the effort of the scheduler does not match the max model, the sum model can be used instead. The sum model is expressed by $N_{comp} + N_{memory}$ and is the model used in this thesis.

$$T_{kernel} = N_b \times N_w \times N_t \times C_t \times \frac{1}{N_c \times D} \times \frac{1}{GPU_{Hz}} \tag{5.23}$$

Because N_b is the number of thread blocks per streaming multiprocessor, we can calculate $N_b = \lceil \frac{N_{blocks}}{N_{SM}} \rceil$. From CUDA standards, the number of warps per block, N_t, is 32. From this we can calculate the number of warps per block to be $N_w = \lceil \frac{N_{threads}}{N_t} \rceil$. The number of threads per thread block used in this implementation is 512. This is to balance out the number of memory accesses within the same thread block. The number of thread blocks needed can then be calculated as $N_{blocks} = \lceil \frac{N}{512} \rceil$, where N is either number of particles or number of pairs, depending on the kernel. The value for the depth of the pipeline D used in this thesis is $D = 4$.

The number of cycles for both memory and computation needs to be derived from the actual CUDA code. However, since different operations take different amount of cycles, the operations need to be multiplied by the number of cycles needed to complete the operation. The amount of cycles per operations used in this thesis can be found in Table 5.2. Because the performance model does not take atomic operations into account, we simplify them to be a global memory read/write.

Global Memory R/W	500
Shared Memory R/W	4
Floating point add/mul	4
Integer add/mul	4
Floating point special	32

Table 5.2: The number of cycles per operations for the GPUs used in the performance models in this thesis.

Communicate

Communication done by MPI can be divided into two parts, migrating particles between subdomains and exchanging particles in the border of two subdomains so that the subdomains mirrors each other at the border. This can be seen in Equation 5.24. The $migrate$ term refers to the migration of particles from one subdomain to the next when it crosses the border. The $bexchange$ term refers to the border exchange of particles in the overlapping areas around the borders between the subdomains. This model is the exact same for all of the implementations.

$$T_{communicate} = T_{migrate} + T_{bexchange} \tag{5.24}$$

Equations 5.25 to 5.33 were proposed by Ragunathan and Valstad (2018) as a model for communication using the Hockney model. There are two phases to the communication taking place, communicating the number of particles that are about to be sent and sending the particles. This can be seen in Equation 5.25 and 5.26. The term s is defined as 1 if `MPI_Sendrecv` uses the same amount of execution time as `MPI_Ssend` and 2 if not. The setup terms are the time needed to prepare what particles should be migrated or exchanged across the border. Using Equation 5.27, we can model the time being used by the setup for border exchange shown in Equation 5.28 and Equation 5.29 for the setup of migration. Although both equations only need one pass over the particles, thereby giving them the same complexity, different operations are needed for each of them. Border exchange only needs to send the particles in the border area while migration has to remove the particles from its own list of particles before sending them. Therefore, the constant term is different for the setup equations.

$$T_{bexchange} = s \times (T_{bexchange\_count} + T_{bexchange\_transfer}) + T_{bexchange\_setup} \tag{5.25}$$

$$T_{migrate} = s \times (T_{migrate\_count} + T_{migrate\_transfer}) + T_{migrate\_setup} \tag{5.26}$$

$$T_{cpu\_execute} = \frac{Workload}{N_{ranks} \times N_{cores}} \times \frac{N_{operations}}{Frequency} \tag{5.27}$$

$$T_{bexchange\_setup} = \frac{P}{N_{ranks} \times N_{cores}} \times \frac{N_{operations}}{Frequency} \tag{5.28}$$

$$T_{migrate\_setup} = \frac{P}{N_{ranks} \times N_{cores}} \times \frac{N_{operations}}{Frequency} \tag{5.29}$$

Equations 5.30 to 5.33 show a Heterogeneous Hockney model approach to modeling the communication. In these equations α is the latency, β is the bandwidth, N is the number of ranks and $4B$ is the size of an `MPI_LONG` data type.

$$\begin{aligned} T_{bexchange\_count} \approx & \max_{rank \in N} \alpha(rank, neighbor(rank)_{east}) + \\ & 4B \times \max_{rank \in N} \beta(rank, neighbor(rank)_{east}) + \\ & \max_{rank \in N} \alpha(rank, neighbor(rank)_{west}) + \\ & 4B \times \max_{rank \in N} \beta(rank, neighbor(rank)_{west}) \end{aligned} \tag{5.30}$$

$$\begin{aligned} T_{bexchange\_transfer} \approx & \max_{rank \in N} \alpha(rank, neighbor(rank)_{east}) + \\ & \max_{rank \in N} N_{bytes,east} \times \max_{rank \in N} \beta(rank, neighbor(rank)_{east}) + \\ & \max_{rank \in N} \alpha(rank, neighbor(rank)_{west}) + \\ & \max_{rank \in N} N_{bytes,west} \times \max_{rank \in N} \beta(rank, neighbor(rank)_{west}) \end{aligned} \tag{5.31}$$

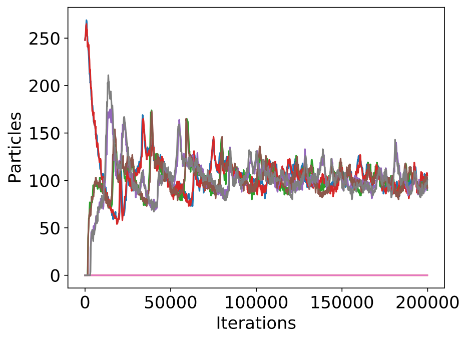
$$\begin{aligned} T_{migrate\_count} \approx & \max_{rank \in N} \alpha(rank, neighbor(rank)_{east}) + \\ & 4B \times \max_{rank \in N} \beta(rank, neighbor(rank)_{east}) + \\ & \max_{rank \in N} \alpha(rank, neighbor(rank)_{west}) + \\ & 4B \times \max_{rank \in N} \beta(rank, neighbor(rank)_{west}) \end{aligned} \tag{5.32}$$

$$\begin{aligned} T_{migrate\_transfer} \approx & \max_{rank \in N} \alpha(rank, neighbor(rank)_{east}) + \\ & \max_{rank \in N} N_{bytes,east} \times \max_{rank \in N} \beta(rank, neighbor(rank)_{east}) + \\ & \max_{rank \in N} \alpha(rank, neighbor(rank)_{west}) + \\ & \max_{rank \in N} N_{bytes,west} \times \max_{rank \in N} \beta(rank, neighbor(rank)_{west}) \end{aligned} \tag{5.33}$$

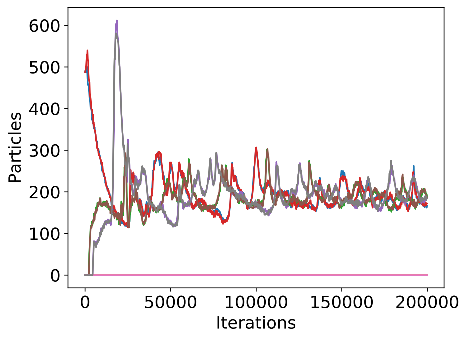
From Ragunathan and Valstad (2018), we know that the latency of the application is approximately constant and depends on the system. This is because the communication pattern of only communicating with neighboring ranks does not increase with added ranks.

The latency used in this thesis is $\alpha = 2\times10^{-6}ms$. As shown in Table 5.3, the V100 cluster has a Infiniband FDR switch. This has a bandwidth of $20GB/s$ which is the value used for β. However, to simulate the effect of communication slowing down when having more ranks than 2, the bandwidth is divided by two as shown in Equation 5.34.

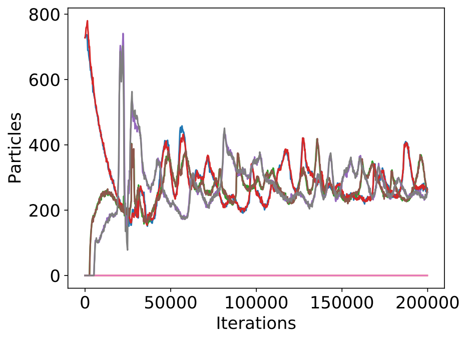
$$\beta = \begin{cases} 20GB/s & if N_{ranks} <= 2 \\ \frac{20}{2}GB/s & if N_{ranks} > 2 \end{cases} \tag{5.34}$$



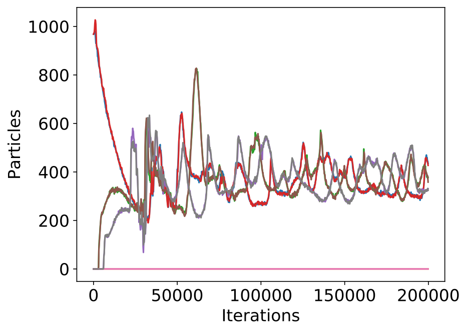
(a) SCALE=1



(b) SCALE=2



(c) SCALE=3



(d) SCALE=4

Figure 5.5: The number of exchanged particles for each rank in both directions. Each curve represents one direction and one rank, using 4 ranks and SCALE 1 to 4. Notice that the range of the y axis change.

Figure 5.5 shows the number of particles exchanged in both directions for each rank. Here we can see a clear trend of the number of particles converging to $100\times$SCALE except for the first rank sending westward and the last rank sending eastward. The different curves are not important, however the trend is. Therefore, we estimate the number of particles exchanged in border exchange as shown in Equation 5.35.

$$b_p = 100 \times SCALE \tag{5.35}$$

5.3 Parameter Space

5.3.1 Architectures

	V100	EPIC2
Nodes	5	19
Processor per node	2	2
Cores per processor	14	12
Processor type	E5-6132 v5	E5-2650 v4
Processor Clock Frequency	2.60GHz	2.20GHz
GPU count per node	2	2
GPU type	Nvidia Tesla V100	Nvidia Tesla P100
GPU cuda cores	5 120	3 584
GPU SM count	80	56
GPU Clock Frequency	1.53 GHz	1 126 MHz
GPU L2 Cache	6 MB	4 MB
GPU Bandwidth	900 GB/S	720 GB/S
GPU Memory	16 GB	16 GB
GPU Theoretical Performance	14 TFLOPS/s	9.3 TFLOPS/s
Primary Memory per node	764GB	64GB
Interconnect	Infiniband - FDR	Infiniband - EDR

Table 5.3: The specification of the clusters used from IDUN. The topology of the clusters is shown by Figure 5.6 and 5.7.

This thesis made use of the IDUN supercomputer at NTNU. IDUN has multiple queues for different types of nodes. The two queues used by this thesis is EPIC2 and V100. Table 5.3 shows the specifications of these queues. Both EPIC2 and V100 are heterogeneous clusters where each node contains two CPUs and two GPUs, EPIC2 having Xeon E5-2650 v4 server chip and two Nvidia Tesla P100 and V100 having Xeon E5-6132 v5 server chip and two Nvidia Tesla V100. The server chips both contain two CPUs, with each CPU having 12 or 14 cores. The CPUs are connected to their memory through Non-uniform memory access (NUMA). The GPUs are connected to the server chip with PCI-Express 3.0. The topology for the V100 queue is shown in Figure 5.6 and the topology for the EPIC2 queue can be found in Figure 5.7.

In order to achieve higher throughput, both CPUs have a SIMD unit of the type Advanced Vector Extensions (AVX). This allows the CPU to execute floating point operations in a similar fashion to a GPU, that is, many of them in parallel. E5-2650 has two AVX2, one per processor, and has a 256 bit wide instruction set, while E5-6132 has two AVX-512, one per processor, which has a 512 bit wide instruction set. The AVX2 has the performance of 16 single precision floating point operations per cycle while the AVX-512 has 32.

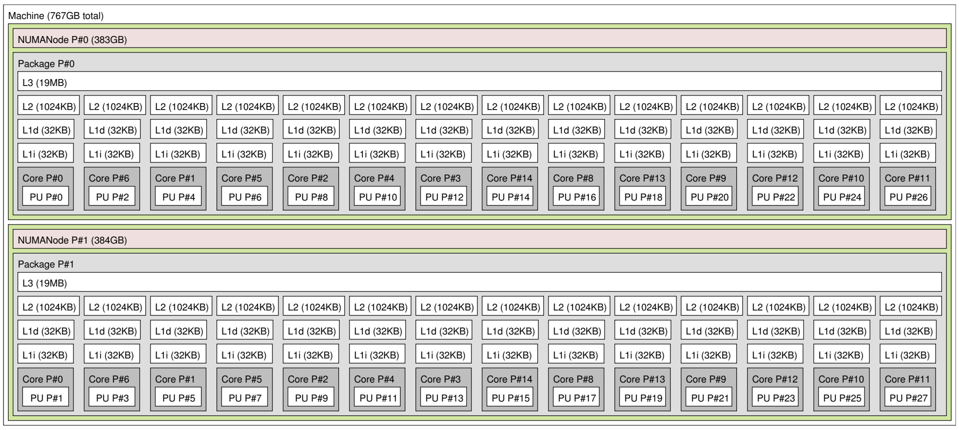


Figure 5.6: Topology of IDUN's V100. Consists of one 383GB and one 384GB NUMA node and two packages of 14 cores with 19MB shared L3 cache, 1024KB L2 cache, 32KB L1 data cache, and 32KB L1 instruction cache.

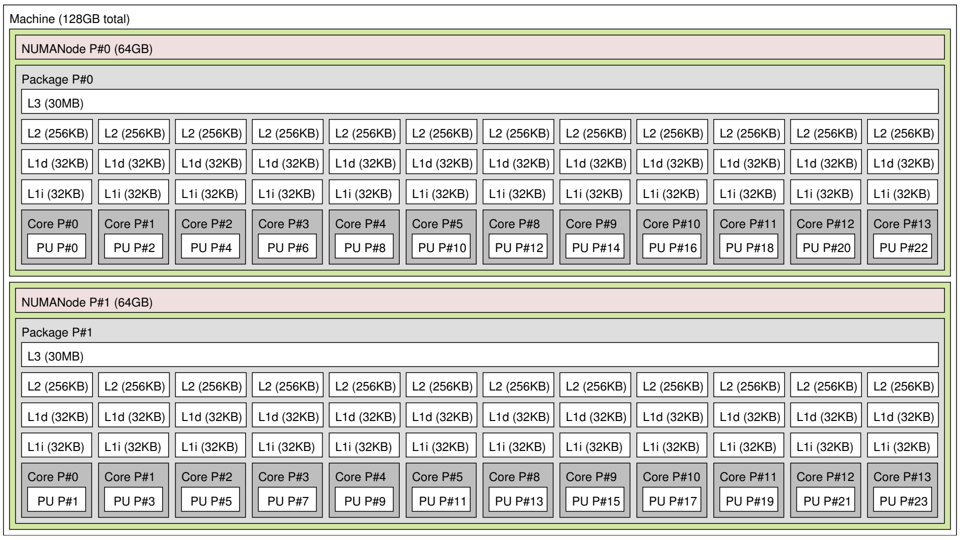


Figure 5.7: Topology of IDUN's EPIC2. Consists of two 64GB NUMA nodes and two packages of 12 cores with 30MB shared L3 cache, 256KB L2 cache, 32KB L1 data cache, and 32KB L1 instruction cache.

Although the theoretical performance of the GPUs is high, it is difficult achieving close to theoretical performance. Problems that fit the execution model of SIMD will reach performance levels closer to the theoretical limit. The WCSPH dam-break problem does not lend itself well to the SIMD architecture and will therefore realistically only be able to make use of half of the performance the GPUs have available. However, because of the large differences in performance and throughput between GPUs and CPUs, even half will give the application a speedup with low efficiency.

5.3.2 Scale

The scale of the dam-break problem can be set by the compile-time variable `SCALE`. As shown in Figure 3.3, `SCALE` increases both the size of the dam as well as the size of the problem, including number of particles. A linear increase in `SCALE` gives a quadratic increase in particles. For the offloading to GPU to be useful, the computational time saved by the speedup has to outweigh the cost of transferring data back and forth. This gives us a lower bound of the range of usefulness: $min < P$, where P is the number of particles. The offloading also has a maximum size, either when the size of the data passes the available memory on the GPU or when the CPU outperforms the GPU at said size of data. This gives us a upper bound of the range of usefulness: $P < max$, and the total range $[min < P < max]$. This range describes when it is beneficial to offload the problem to a GPU as well as when the problem becomes either too small for the transfer cost of offloading or too big to handle. Because of the nature of the proxy application, the application is able to utilize multiple GPUs. Adding another node and/or GPU, shifts the number of particles towards the min side of the range. It is possible to continue adding GPUs, this however, faces the same problem as Patterson (2004) proposes where latency becomes the limiting factor.

5.4 Experimental Design

The different methods were executed on different hardware at different values of `SCALE` over 100 000 iterations in order to reach a near equilibrium state of the fluid. This was done on very few particles as well as many particles in order to get close to the hardware memory limit as possible. They were timed using the timing function `MPI_Wtime` to get wall time. This is often a wrapper for `clock_gettime` or `gettimeofday` which has a precision of $10\mu s$, which means that all timing from the experiment has the same precision. Parts relating to the significance of the life cycle, as shown in Figure 5.2, are timed in every iteration. The timing is summed up for each part, giving the total amount of time spent on a certain part of the proxy application. Each rank times its own execution. Because the problem is divided into static parts, the workload is not equally divided among ranks, especially at higher values of `SCALE` where the simulation needs more iterations to arrive at the same state as lower values of `SCALE`. Therefore, some of the data is taken from the average over the ranks, while some data uses the ranks timing directly. The purpose of the experiments was to test the different terms of the proposed performance model in this chapter, and then validate them.

Because migration happens less frequently and with fewer particles than the border exchange, we can approximate communication to be $T_{communicate} \approx T_{border\_exchange}$. Furthermore, because of a non-parallelized loop of the particles in the inherited code for particle-migration, the execution time of migration would be higher than expected.

Chapter 6

Experimental Results & Discussion

In this chapter we go through five categories of experiments, validation of the performance models and then give recommendations based on the results. The experimental results are presented in Section 6.1 and then discussed in Section 6.2. The purpose of the categories is to examine the performance parameter space as well as terms of the performance models and scalability of the application as a whole. If nothing else is stated, the data presented is taken from a 100 000 iteration execution of the application. When figures have shown similar results for both clusters, the figure for the V100 cluster has been chosen while the others can be found in the appendix.

The first category, **Computational vs Communicational Complexity**, measures the growth of computational intensity and communicational intensity and compares the difference between the two with varied parameters.

The second category, **Finding Neighbors & Time Step**, examines the relationship between finding neighbors and the rest of the time step of the simulation.

The third category, **Speedup & Efficiency**, measures the Speedup and Efficiency of the different approaches on different architectures to different baselines.

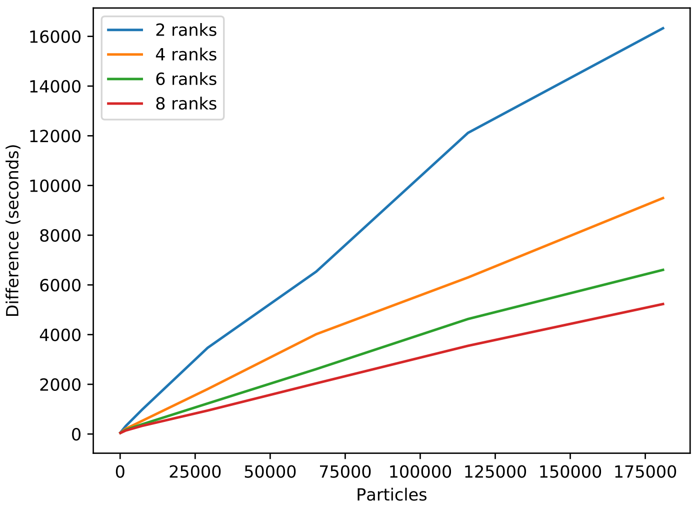
The fourth category, **Problem Scaling**, studies the implications of increasing problem size and horizontal growth.

The fifth category, **Utility range**, explores the limits of the different approaches as well as the benefit of offloading the problem to an accelerator.

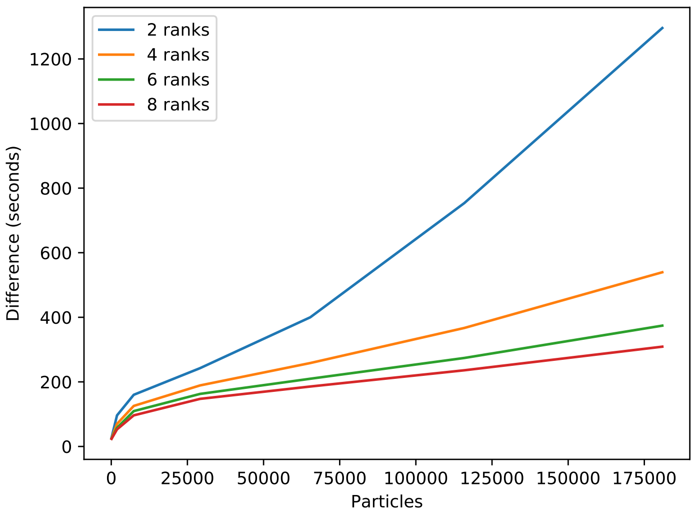
6.1 Results

6.1.1 Computational vs Communicational Complexity

This section studies the complexity of the computation and the communication between nodes. This is done by taking the difference of time spent on computation and the time spent communicating for a given scale and number of ranks.



(a) CPU version



(b) GPU version

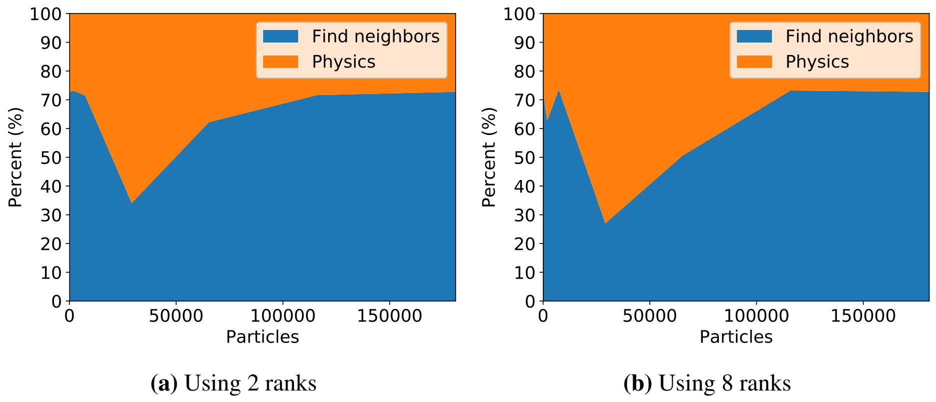
Figure 6.1: The difference between time spent computing and time spent communicating using the cell-linked list method on the V100 queue. Notice that the range of the y axis change.

The difference between the execution time for computation and communication can be expressed by Equation 6.1 and can be seen in Figure 6.1. Here we can see that 2 ranks has a high difference while 8 ranks has a low difference. This is because 2 ranks uses less time on communication and more on computation, while 8 ranks uses more time on communication and less time on computation. Therefore, lower values of difference are generally better. Both the GPU and the CPU version show a trend of computational execution time being higher than communicationial execution time, although with increasing amount of ranks the difference declines. The CPU version can be seen to grow steadily from a low value of SCALE and slowly even out, especially when more ranks are added. The GPU version, however, can be seen to have a rapid growth at lower values of SCALE that evens out before a slow steady growth can be seen.

$$T_{cvsc\_diff} = T_{compute} - T_{communicate} \tag{6.1}$$

6.1.2 Finding Neighbors & Time Step

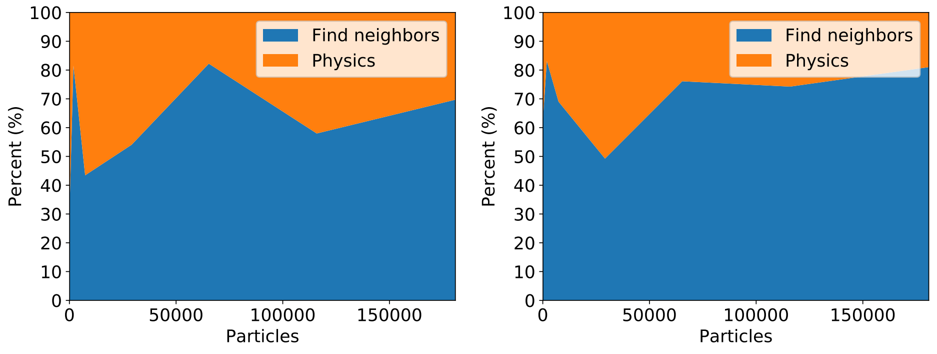
This section studies the relationship of finding neighbors and the rest of the physics time step. This is done with the computational time of the different approaches, architectures and problem scales.



(a) Using 2 ranks **(b)** Using 8 ranks

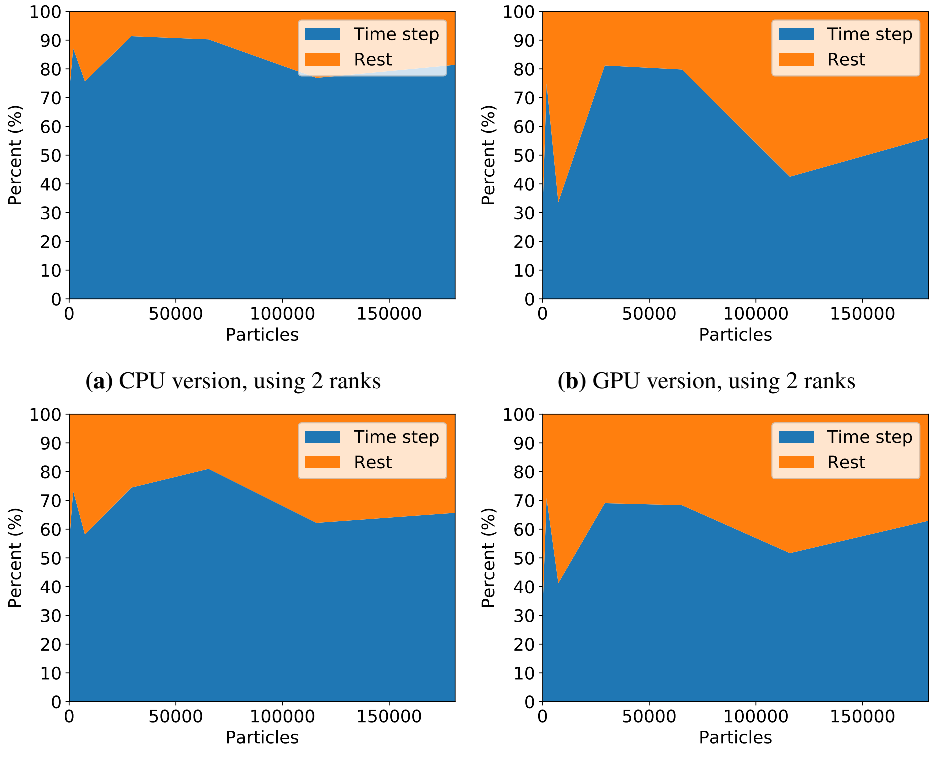
Figure 6.2: The percentage of time the components of time step uses when running the CPU version on V100.

As shown by Ragunathan and Valstad (2018), Figure 6.2 shows that finding neighbors with the cell-linked list method on CPU will take up about 80% of the time step each iteration for most values of SCALE. Around 30 000 particles the percentage of neighbor-finding drops to 50% steadily increasing from there until 115 000 particles. Figure 6.3 shows that the cell-linked list method on GPU will take up about 40% at the lowest. While the cell-linked list CPU version had a fall in percentage around $[30000 \leq P \leq 115000]$, the GPU version has an increase around these values. Increasing the amount of ranks will increase the percentage finding neighbors takes of the time step.



(a) Using 2 ranks (b) Using 8 ranks

Figure 6.3: The percentage of time the components of time step uses when running the GPU version on V100.



(a) CPU version, using 2 ranks **(b)** GPU version, using 2 ranks

(c) CPU version, using 8 ranks **(d)** GPU version, using 8 ranks

Figure 6.4: The percentage of time the execution of time step takes out of the whole execution with cell-linked list on different architectures and ranks. Data from V100

Because both finding neighbors and the physics computation are offloaded, comparing them only serves to show how the relationship between the two parts have become after being offloaded and not their non-offloaded counterparts. Figure 6.4 shows the comparison of the time used on time step and the time used on the rest of the execution. This lets us compare the time step of the CPU version to the time step of the GPU version since the rest of the execution is exactly the same. As the number of ranks increases, the percentage of time step for the CPU version is lowered. However, the GPU version has a lower percentage no matter the number of ranks used in this experiment, showing that the bottleneck of the problem has been reduced. Using 2 ranks, the CPU version is at lowest around 80% and highest is close to 90%, while the GPU version is at lowest around 35% and at most around 80%. The CPU version has an average around 82% while the GPU version has a average around 57%. Adding more ranks shows a clear decline in percentage of the CPU version going from 82% to 74% to 70% to 67% for the ranks 2, 4, 6 and 8 respectively. However, adding more ranks for the GPU versions only show small fluctuations of $56\% \pm 1$.

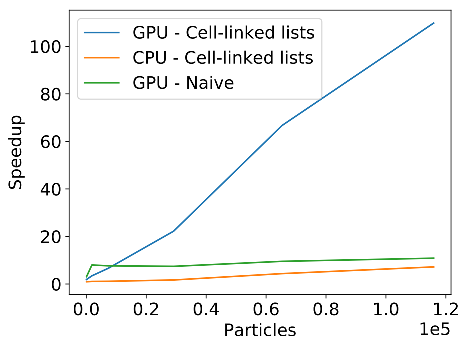
6.1.3 Speedup & Efficiency

This section studies the Speedup and Efficiency of the different versions on different scales and ranks. This is done by taking total execution time and using a naive execution of the same scale and rank for baseline as comparison.

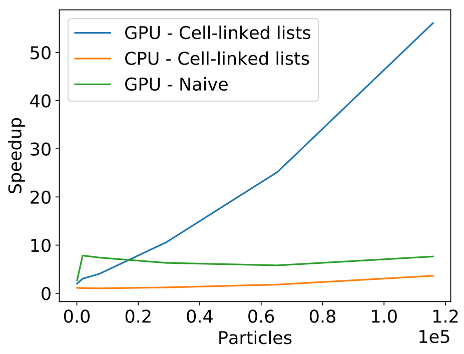
The Speedup of the different versions can be seen in Figure 6.5. The speedup of the cell-linked list GPU version is significantly faster than the corresponding CPU version. Both GPU versions can be seen either plateauing or starting to decrease its growth. While the cell-linked list CPU version has a slower growth, it has not started evening out like the other methods. The naive GPU version has a higher speedup than the cell-linked list CPU version for lower values of `SCALE`. It also has a higher speedup than the cell-linked list GPU version for very low values of `SCALE`, because of the overhead of setting up the cell method. However, the naive GPU version starts plateauing earlier than the cell-linked list CPU version and given sufficiently large value of `SCALE`, the CPU version will reach a higher Speedup as can be seen when using 6 and 8 ranks. From Figure 6.5, we can see that the general Speedup achieved declines when more ranks are added. The range of problem sizes are kept the same, while the workload is divided over more nodes. This is more beneficial to the naive method because the naive method grows faster than the cell method and the cell method has a setup phase in addition.

The Efficiency of the different versions can be seen in Figure 6.6. Here we see that although the Speedup of the GPU versions are high, the Efficiency is very low. The cell-linked list version has slightly more Efficiency than the naive version. The CPU version however, has better Efficiency and in turn, is able to utilize the hardware to a higher degree. Although, the more ranks added, the less efficient the versions become. The cell-linked list CPU version at $P =$115 921 has the Efficiency of $E \approx$ 0.26 using 2 ranks, $E \approx$ 0.07 using 4 ranks, $E \approx$ 0.04 using 6 ranks and $E \approx$ 0.02 using 8 ranks. The same trend happens with adding ranks to the GPU versions as well, although harder to see on the graph.

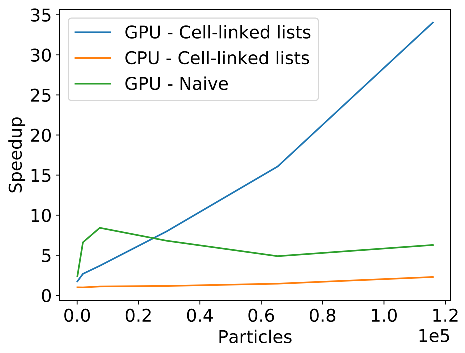
Figure 6.7 shows the Speedup of the cell-linked list GPU version using the corresponding CPU version as a baseline using the same amount of ranks. No matter the number of ranks, both the V100 and P100 GPU versions achieve around $S \approx 10$ at their peak.



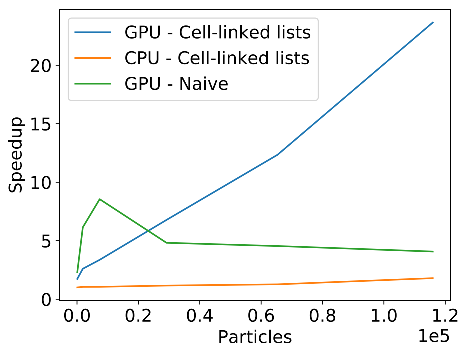
(a) Using 2 ranks



(b) Using 4 ranks



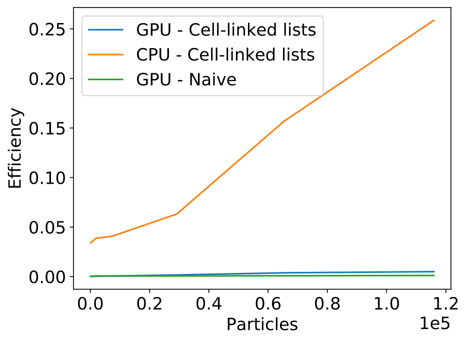
(c) Using 6 ranks



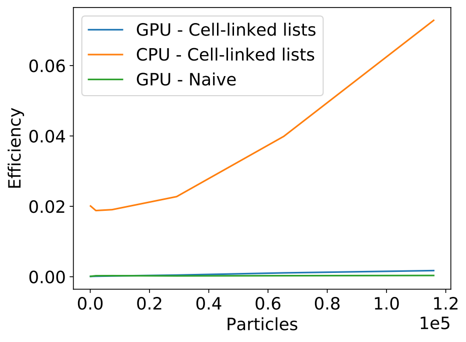
(d) Using 8 ranks

Figure 6.5: Speedup from a parallel naive CPU baseline of the SPH dam-break problem on the V100 queue. Notice that the range of the y axis change.

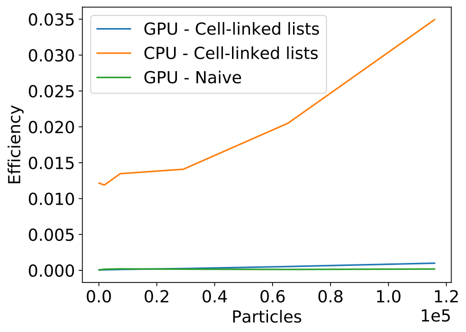
However, with increasing `SCALE` the Speedup declines. The executions with fewer ranks decline faster than the executions with more ranks, but can also be seen to slow the decline of Speedup after the initial decline. This indicates that the speedup will slowly converge, given enough ranks.



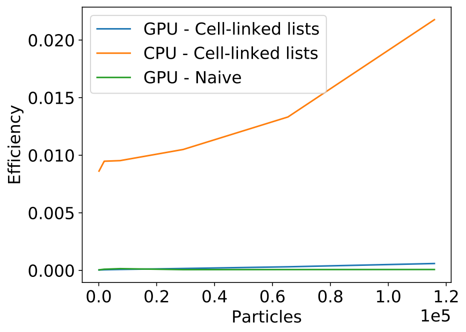
(a) Using 2 ranks



(b) Using 4 ranks

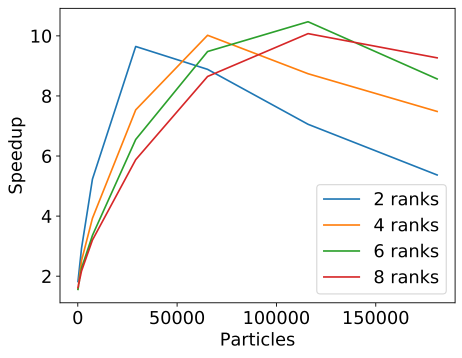


(c) Using 6 ranks

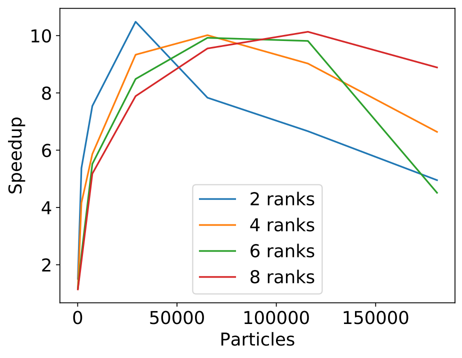


(d) Using 8 ranks

Figure 6.6: Efficiency from a parallel naive CPU baseline of the SPH dam-break problem on the V100 queue. Notice that the range of the y axis change.



(a) Data from V100.



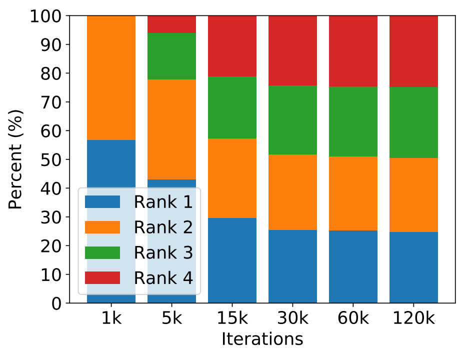
(b) Data from EPIC2.

Figure 6.7: The Speedup of the cell-linked list GPU version using the cell-linked list CPU version as a baseline. Note that this Speedup is comparing the CPU on the cluster to the GPU on the same cluster.

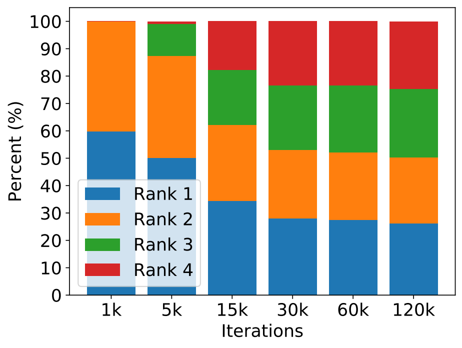
6.1.4 Problem Scaling

In this section we study how the workload is distributed among the ranks according to number of ranks and `SCALE` as well as what implications it has for workload when ranks are added and `SCALE` is increased.

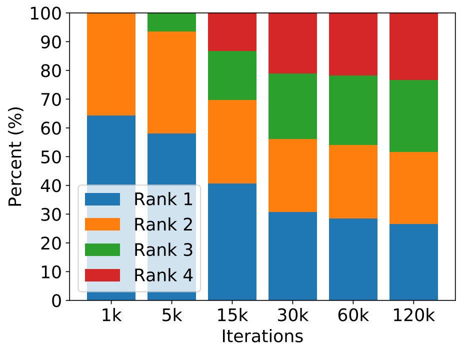
Because of how the workload divides the domain, the fluid of the dam-break will traverse the ranks from west to east. This means that the east most ranks will be idle while the west most ranks are not idle. The larger the scale, the longer the fluid takes to traverse the subdomains which increases execution time. Increasing scale also increases the number of iterations needed in order to achieve the goal state of near equilibrium. If the number of iterations are low, there might be ranks that never do any computation. Given enough iterations, the workload will approach an even split as the fluid reaches an equilibrium of motion. Figure 6.8 shows how the execution time accumulated by ranks will get closer to complete balance as the number of iterations increase. However, because this application uses the Masteronly approach to communication, the ranks with less actual work have to wait for the other ranks to finish their computation.



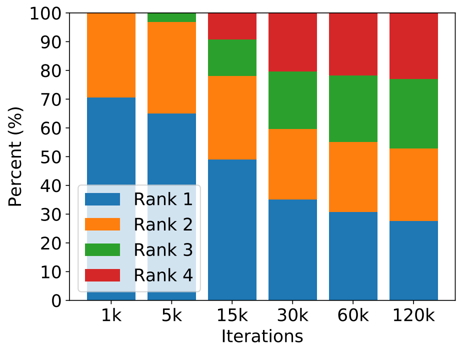
(a) `SCALE`=1.0



(b) `SCALE`=2.0



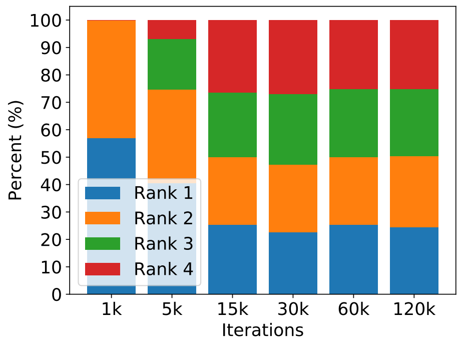
(c) `SCALE`=3.0



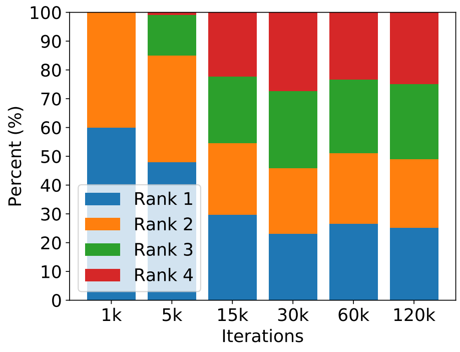
(d) `SCALE`=4.0

Figure 6.8: Workload balance at 1k, 5k, 15k, 30k, 60k, 120k iterations over 4 ranks with `SCALE` 1 to 4. Data from V100, CUDA with cell-linked list.

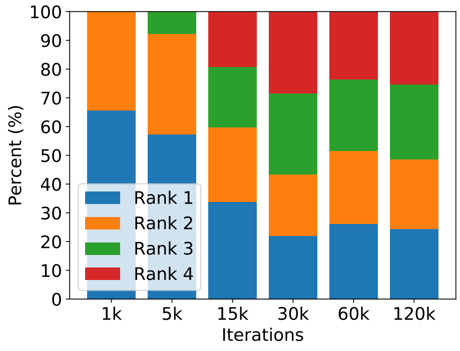
While Figure 6.8 shows the balance over iterations, Figure 6.9 shows the percentage of distribution of particles among ranks at different iterations. It is possible to see the wave motion in the distribution of particles. The larger the scale, the more iterations the wave needs to reach equilibrium, the less balanced the workload becomes. The higher the scale, the slower the iterations become, making the time to reach equilibrium even longer and the time the west-most ranks spend idle even longer.



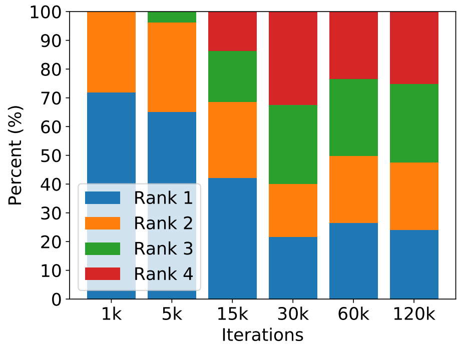
(a) `SCALE`=1.0



(b) `SCALE`=2.0



(c) `SCALE`=3.0



(d) `SCALE`=4.0

Figure 6.9: Distribution of particles per rank at 1k, 5k, 15k, 30k, 60k, 120k iterations over 4 ranks with `SCALE` 1 to 4. Data from V100, CUDA with cell-linked list.

6.1.5 Utility range

In this section we study the $[min < P < max]$ range of the proxy application and how it varies for different aspects of the parameter space.

The minimum side of the range depends on the time it takes to transfer the necessary data to the GPU and back, which in turn depends on the scale of the problem. The benefit from offloading to GPU can be expressed by Equation 6.2 where B will be positive if it is beneficial using a GPU and negative if it is not beneficial. Figure 6.10 shows a comparison of the different approaches on both CPU and GPU. The benefit B from Equation 6.2 is set to be the cell-linked list versions on both CPU and GPU. Here we see that the overhead of transferring the particles to the GPU is diminished by the throughput available on the GPU. This can be seen as the benefit is never negative and has the same trend as the cell-linked list version of the CPU.

$$B = T_{CPU\_compute} - T_{GPU\_compute} \tag{6.2}$$

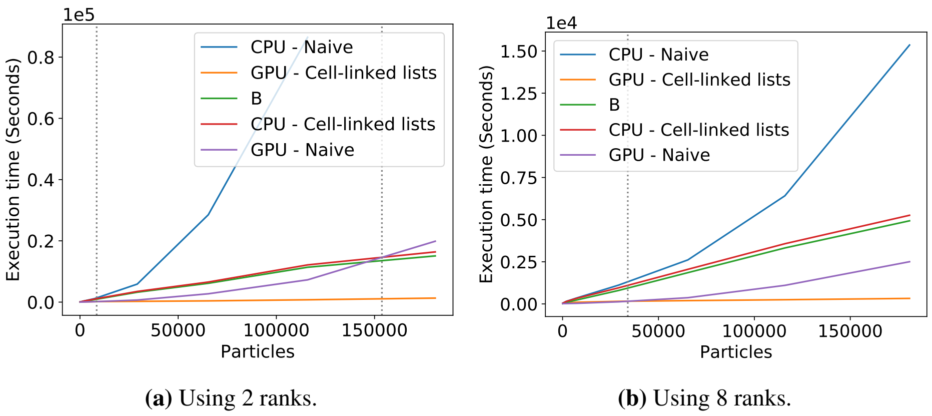
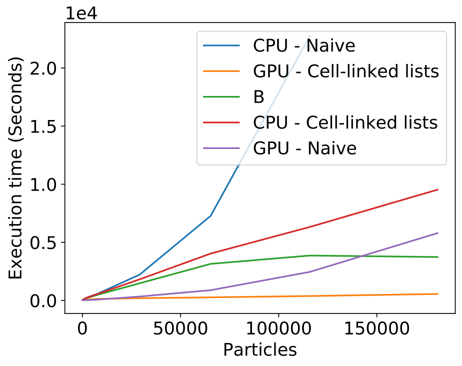


Figure 6.10: Comparisons of algorithm runtimes of CPU and GPU. B is Equation 6.2 with the cell-linked list versions of both systems. The vertical lines are the intersection of naive GPU & cell-list GPU and naive GPU & cell-list CPU respectively. Data from V100. Notice that the range of the y axis change.

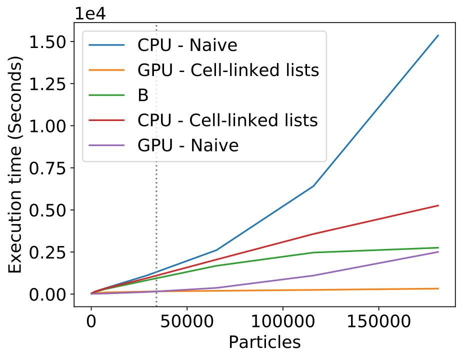
Figure 6.10 shows that the naive GPU version is faster than the cell-linked list GPU version. Using 2 ranks, the naive version overtakes the runtime of the cell-linked list version just before P ≈8 500. As the number of ranks are increased, the point where the versions cross is offset. Using 8 ranks, the naive version becomes slower at P ≈34 000. Based on this result, we can see that the naive GPU version would be beneficial to use over the cell-linked list CPU version which is shown in Figure 6.11.

Figure 6.11 shows that using 4 ranks or less, the naive GPU version still has a benefit over the cell-linked list CPU version. However, while both versions have the same asymptotical complexity, that is $O(P^2)$, the CPU version starts at a higher execution time because of the lower throughput of the CPU, but the GPU version has a higher constant to its growth and will therefore outgrow the CPU version. This can be seen as the benefit starts trending towards a negative value, without reaching it. This means that using the naive GPU version is beneficial for problem size $P < 175000$. Using 8 ranks however, diminishes the benefit of the naive GPU version. The benefit starts declining at a lower problem size and stops being beneficial around P ≈154 000.

The maximum side of the range depends on the hardware specific limitations of the architecture and the memory consumption of the algorithms. For both Tesla V100 and P100, the maximum memory size is $16GB$. This part assumes that each rank of the program runs on one GPU each. If one GPU is used by more than one rank, the maximum number is divided by the number of ranks sharing that GPU. This is shown in Equation



(a) Using 4 ranks.



(b) Using 8 ranks.

Figure 6.11: Comparisons of algorithm runtimes of CPU and GPU. B is Equation 6.2 with the naive GPU version and the cell-linked list CPU version. Data from V100. Notice that the range of the y axis change.

6.3, where the GPU memory capacity is divided by the number of ranks using it. We also assume 64-bit variables, except for floating points, which is 32-bit. Figure 6.12 shows the maximum amount of particles the different method can have in relation to the number of ranks as well as how ranks influence the memory used.

$$Memory\_Size = \frac{GPU_{mem}}{N_{ranks}} \tag{6.3}$$

The naive methods uses the least amount of memory, not using any extra memory for structuring comparisons. Equation 6.4 shows the memory usage of the naive method in bytes. $S_{particle}$ is the data structure of the particles and S_{pairs} is the data structure of the pairs. The proxy application used in this thesis had $sizeof(S_{particle}) = 104$ and $sizeof(S_{pair}) = 40$. Using this, the maximum number of particles for the naive method is at most $P \approx$1 321 877.

$$Naive_{mem} = P \times sizeof(S_{particle}) + 300P \times sizeof(S_{pair}) \tag{6.4}$$

The cell method depends on the cell structure to create pairs. This is in addition to the memory used by the naive method. Equation 6.5 shows the memory usage of the cell method. The structure consists of a set number of cells for the size of the dam as well as an upper bound for the maximum number of particles a cell can contain, shown as $N_{cells} \times N_{p\_per\_cell}$. It is necessary to track how many particles are in each cell which is done by a simple list that is the same length as the number of cells, hence the additional N_{cells}. The number of cells depends on the size of the subdomain and the size of the dam. The dam is equally divided between the subdomains. All of these rely on the initial values, shown in Figure 3.3 which all scale with SCALE except H. Using these values, we can calculate the number of cells given a SCALE and the number of ranks. For each additional rank, the extra memory usage of the cell method decreases because of the decreasing size of the subdomain. The more ranks, the closer it will come to the naive method.

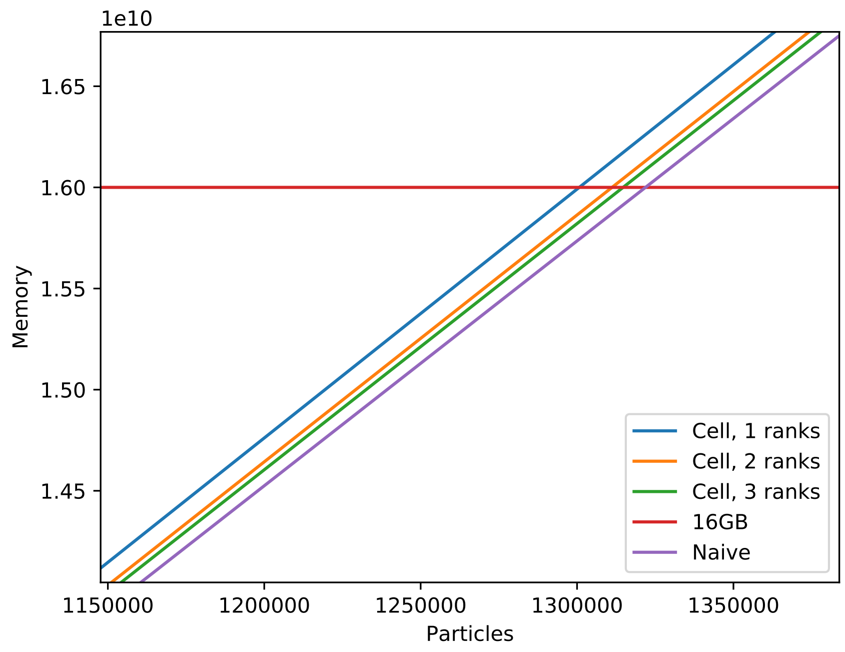


Figure 6.12: Scaling of memory usage by cell method and naive method on GPU.

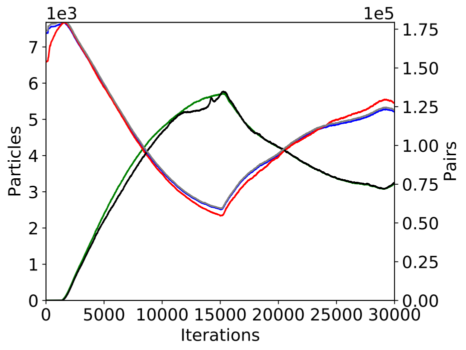
However, the subdomains cannot be smaller than the interaction radius because otherwise the simulation will not work. Using linear regression, with $R^2 \approx 1$ to 8 decimal places, we can make the estimate of Equation 6.6, which yields the maximum number of particles for the cell method is $P \approx$1 300 700 for one rank. Figure 6.12 shows how adding extra ranks increases the number of particles closer to the same maximum amount as the naive, however the increase declines for each added rank.

$$Cell_{mem} = Naive_{mem} + N_{cells} \times N_{p\_per\_cell} \times sizeof(int) + N_{cells} \times sizeof(int) \quad (6.5)$$

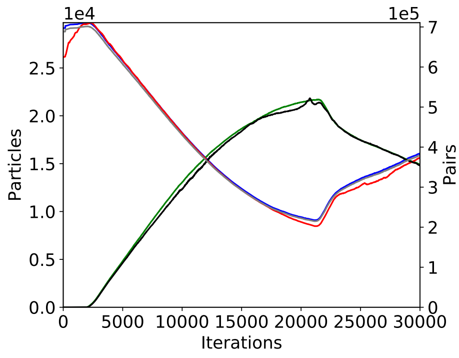
$$max_{cell} = \lfloor \frac{Memory\_Size - 315541.69}{12300.82} \rfloor \quad (6.6)$$

Using the upper bound of 300 pairs per particle establishes the upper bound of memory usage of the naive and cell-linked list methods. However, this does not give a realistic view of memory usage or take into account that the upper bound could be lowered. Changing the upper bound to the estimate $e_p = 23.6$ in Equation 6.4 gives us the estimate of memory usage by the naive method. Using this, the maximum number of particles for the naive method is $P \approx$15 267 175, which is 13 966 475 more than the upper bound of memory usage. Similarly, using the estimate for the cell-linked method we can formulate Equation 6.7. From this equation, we find that the estimated maximum number of particles is $P \approx$12 855 505, which is 11 554 805 more than the upper bound.

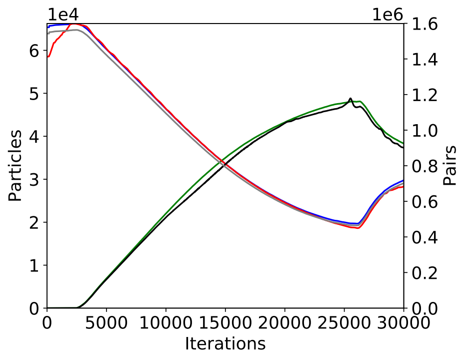
$$est_{cell} = \lfloor \frac{Memory\_Size - 1323129.29}{1244.50} \rfloor \tag{6.7}$$



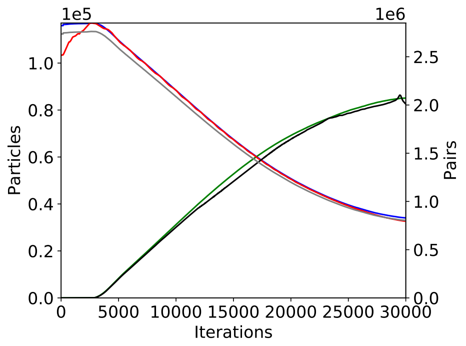
(a) SCALE=1



(b) SCALE=2



(c) SCALE=3

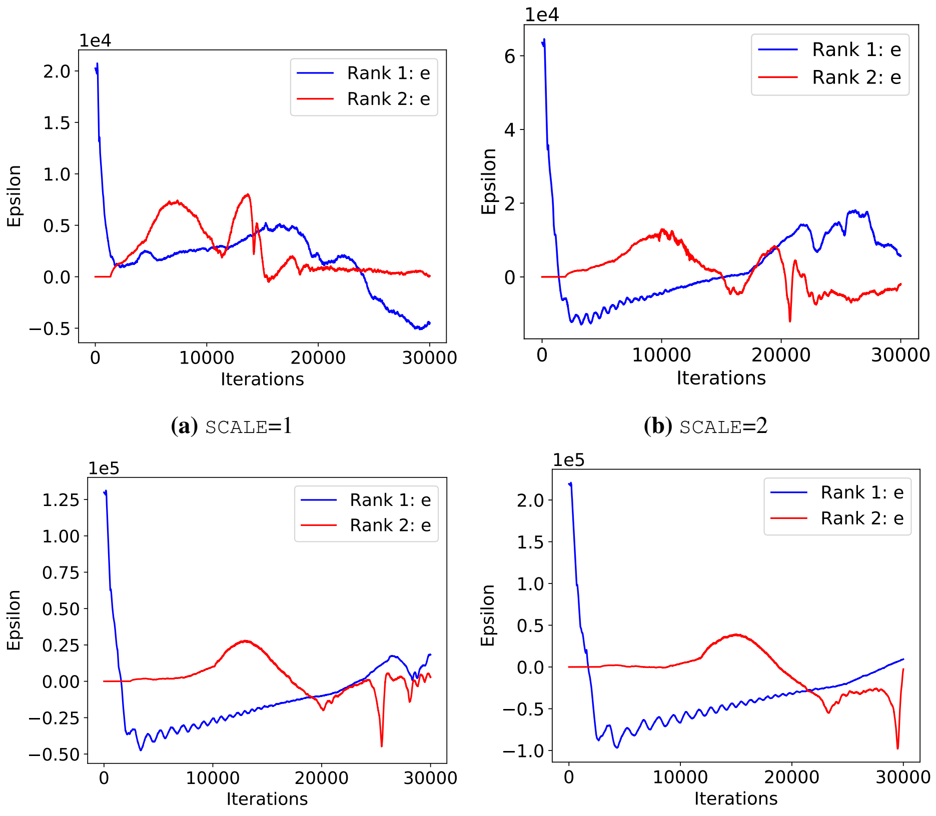


(d) SCALE=4

Figure 6.13: The relationship between two rank's particles and pairs during a given iteration for different scales. The red and blue lines are the first rank's pairs and particles. The green and black lines are the second rank's pairs and particles. The grey line is the estimate. Notice that the range of the y axis change.

Figure 6.13 shows how the number of particles and pairs follow the same kind of pattern. The particle count includes all particles, ghost and mirror particles as well, because they can also be part of a pair. At $t = 0$ the particles are all spaced out, not interacting. This causes the sudden rise in pairs that can be seen following rank 1. The fluid flows towards the right wall, causing the fluid to be spread thinner and fewer pairs to be created. The peak of pairs at the end of SCALE=1 marks the wave hitting the right wall and shifting back towards the left. Hitting a wall will create extra particles because a considerable amount of fluid is gathered at one point as well as ghost particles being created for all the particles that are at the boundary of the floor and wall.

Following the rank 2 lines shown in Figure 6.13, we can observe that as the fluid is flowing towards the right wall, it has less pairs because there are fewer particles there. When the wave crashes into the wall, particles are gathering at that point before some of them are launched into the air. This can be seen in the decrease in pairs until they land. The landing causes another launch of particles before the violent flow settles down. All of these patterns can be seen in all the scales.



(a) SCALE=1 **(b)** SCALE=2

(c) SCALE=3 **(d)** SCALE=4

Figure 6.14: The error of $e_p = 23.6P$, as shown in Equation 6.8. Notice that the range of the y axis change.

Violent flow in the fluid causes fluctuations in the number of pairs. Equation 6.8 shows how we can model the error of $e_p = 23.6P$, for a given iteration i and rank r, and is shown in Figure 6.14. The Figure shows how the error tend towards zero as the number of iterations increases. With more iterations, the more stable the fluid becomes, until it is motionless. However, with increasing scale, it takes more iterations to advance the simulation to the same point, as seen in Figure 6.13. Because of this, increasing scale while iterations remain the same makes the estimate less precise. However, the violent motion of the fluid has more influence on the error of the estimates. At the point where the fluid is motionless given a scale and iteration, the estimate will be the most precise.

$$\epsilon_{i,r} = 23.6P - N_{i,r} \tag{6.8}$$

6.2 Discussion

6.2.1 Computational vs Communicational Complexity

The computational complexity of finding neighbors is known to be $O(P^2)$. The rest of the physics computation where the pairs discovered are used all have linear complexity and therefore the complexity of the whole time step is $O(P^2)$ as well. However, from Figure 6.1 we see that the difference is decreasing with increased number of ranks. This is because for the ranks and scales shown, the computational amount is larger than the communicational. Adding more ranks does increase computational throughput, however it also lowers the bandwidth of the communication because it is being shared by all the ranks. This horizontal scaling can be viewed as a large computer with decreasing bandwidth of the bus. From this we know that at some point, adding more ranks will not be useful because the computational amount will be smaller than the communication amount.

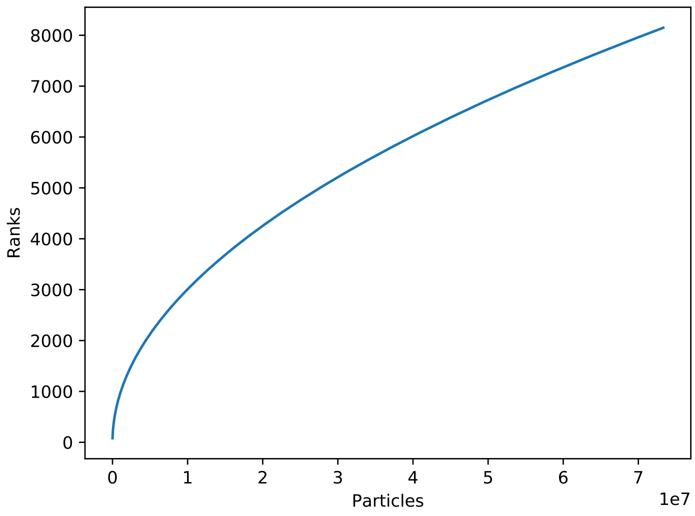


Figure 6.15: The maximum amount of ranks for values of `SCALE` from 1 to 100.

Because the subdomain size has to be larger than the interaction radius of the particles in order for the simulation to work, we can work out the maximum number of ranks possible for a given scale, shown in Equation 6.9. This equation gives us the first whole number of ranks where the subdomain is bigger than the interaction radius of the particles. Figure 6.15 shows the relationship of possible ranks and number of particles. Here we see that there is a sharp increase at the early stages while slowly declining in growth as the number of particles increases. Using the maximum rank numbers and the performance models, we can predict the difference between computational and communicational amount as seen in Figure 6.16. Here we see that although the computational amount is higher than the maximum communication amount at lower scales, the communication amount grows faster than the computational. This is because by adding ranks we increase the communicational amount while decreasing computational. By this prediction, using the cell list GPU version, the difference will be zero around `SCALE`$\approx$ 18.8 with 1510 ranks and keep declining with increasing values of `SCALE`.

$$\begin{aligned} Radius &= 3H \\ subdomain &= \frac{B}{N_{ranks}} \\ subdomain &> Radius \\ \implies N_{max\_ranks} &= \lfloor \frac{B}{Radius} + 1 \rfloor \end{aligned} \tag{6.9}$$

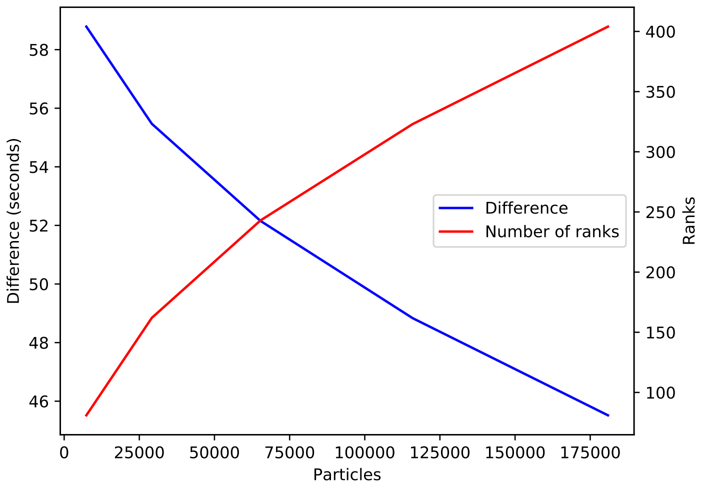
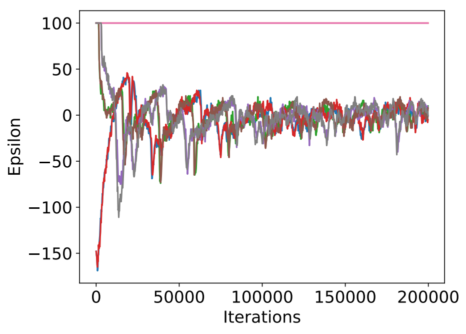


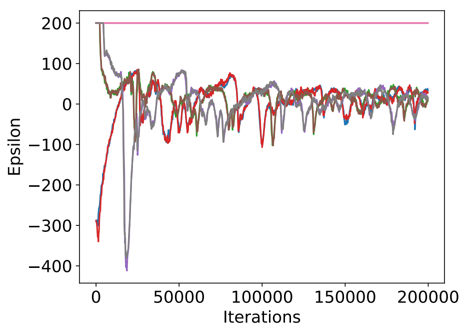
Figure 6.16: Prediction of difference between execution time of communication and computation for cell-linked list GPU version with maximum amount of ranks for each value of `SCALE` from 1.0 to 5.0 for the V100 cluster.

As shown by Figure 6.16, using the maximum number of ranks will make the communicational amount surpass the computational. However, this only shows the relationship between the two main components of the proxy application. To find the best amount of ranks in the range of $[1, N_{max\_rank}]$, we need to find the minimal execution time. Using Equation 6.10 we can find the optimal number of ranks to use in terms of execution time of the whole proxy application. Using this, we find that ”best” is one rank under the maximum amount of ranks, which means that although the communicational amount becomes greater than computational the overall performance of the application still increases. The reason that the best is $N_{max\_ranks} - 1$ is that the performance model has an incremental and not continuous growth because of how it models the distribution of thread blocks over streaming multiprocessors. The trend of the prediction continues for values of SCALE above 5 as well according to the model.

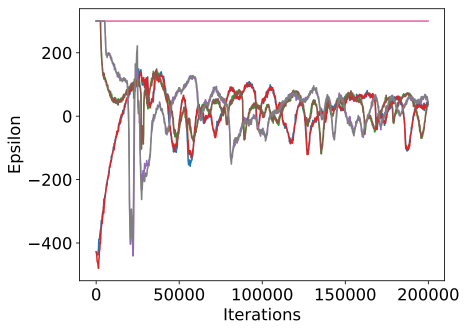
$$\begin{aligned} T_{total}(ranks) &= T_{communicate}(ranks) + T_{compute}(ranks) \\ T_{total\_best} &= min(T_{total}(1), T_{total}(2), ..., T_{total}(N_{max\_rank})) \end{aligned} \tag{6.10}$$



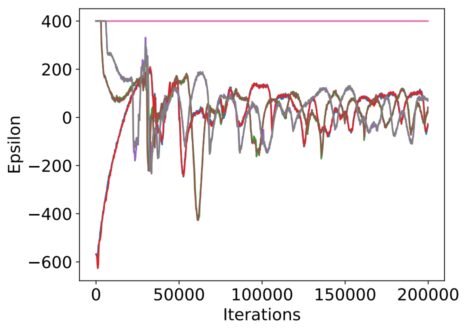
(a) SCALE=1



(b) SCALE=2



(c) SCALE=3



(d) SCALE=4

Figure 6.17: The error of $b_p = 100 \times SCALE$, as shown in Equation 6.11. Each curve represents one direction and one rank, using 4 ranks and SCALE 1 to 4. Notice that the range of the y axis change.

The prediction of communication relies on the performance model of communication which in turn relies on our estimate that the number of particles exchanged in the border exchange is constant and can be expressed as $b_p = 100\times$SCALE. The size of the overlap region between the subdomains is highly dependant on the SCALE value. However, due to the particles being in the west-most ranks at $t = 0$, there are no particles being exchanged between the rank that has no particles. There are also no particles being exchanged westward by the west-most rank and eastward by the east-most rank. Similar to the error of the estimate of pairs per particles, we can model the error as shown in Equation 6.11. Here the subscript i refers to the iterations of the problem, d refers to the direction of the exchange (west or east), r refers to the specific rank, and N refers to the actual amounts of particles exchanged.

$$b\epsilon_{i,r,d} = 100 \times SCALE - N_{i,r,d} \tag{6.11}$$

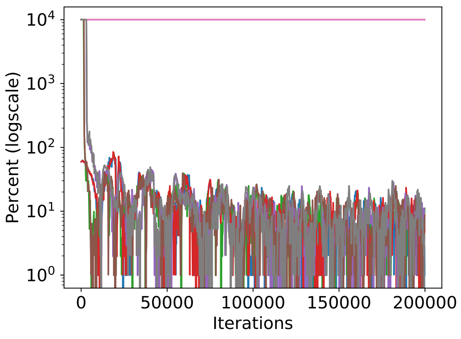
Similar to the estimate of e_p, we see that the error is generally at its largest in the first thousand iterations, as shown by Figure 6.17. However, increasing the value of SCALE increases the fluctuations in the number of particles being exchanged. This is due to the progression of the dam-break problem requiring a greater number of iterations to reach the same state at higher values of SCALE. For all values of SCALE, $b\epsilon$ trends towards zero as the fluid reaches a state of equilibrium. In Figure 6.17 we also observe that the west-most rank sending westward and east-most rank sending eastward will always have an error of $100\times$SCALE due to the fact that particles are not being exchanged. Because some ranks and directions have a negative value of $b\epsilon$ while other ranks have a positive value for the same number of iterations, the sum of errors from $b\epsilon$ evens out. Disregarding the ranks and directions that cannot exchange particles and using 200 000 iterations, the sum of $b\epsilon$ is -17 384 for SCALE=1, 50 266 for SCALE=2, 146 350 for SCALE=3 and 225 513 for SCALE=4. This indicates that the estimate will model too few particles if the number of iterations are high and the value of SCALE is low.

Using Equation 6.11 we can find the percentage of error of the estimate b_p by dividing $b\epsilon$ by the actual number of particles being exchanged for a given rank, iteration and direction. This is shown by Equation 6.12 and visualized in Figure 6.18. Here we see that the error of the outer ranks are around 10 000% and only growing with increased scale. We also observe that the eastward exchange of rank 1 and 2 as well as the westward exchange of rank 2 and 3 has a high percentage of error as well. This is because there are no particles in these ranks at this state of the problem. With increasing iterations, the error of the estimates decline quickly among these ranks with these directions. The number of iterations needed however, increases with the value of SCALE.

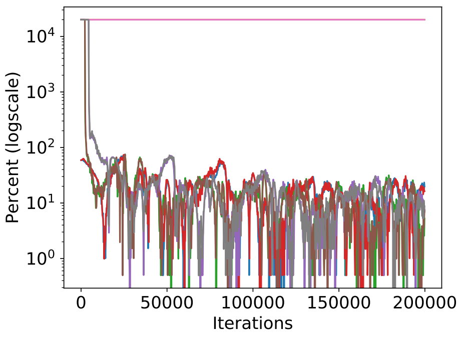
$$\frac{b\epsilon_{i,d,r}}{N_{i,d,r}} = \frac{\mid 100 \times SCALE - N_{i,d,r} \mid}{N_{i,d,r}} \tag{6.12}$$

Using the mathematical clamp function on Equation 6.12 to ensure that the value will be 100% or below gives us a clearer picture of the development of the error percentage with increasing iterations, shown in Figure 6.19. Here we see that at SCALE=1, the percentage of error trends towards a value approximately equal to 10% with spikes at 80%, 50% and 40%. Increasing values of SCALE have similar spikes with an offset in iterations with the same rank and directions. Therefore, to achieve the same error rate at a higher value of SCALE, more iterations are needed. At 200 000 iterations with SCALE=1, all of the ranks and directions are around or below 10%. Because the number of particles exchanged in border exchange is relatively low, the impact of an error rate of 80% is a difference of 80 particles at SCALE=1, 160 particles at SCALE=2, 240 particles at SCALE=2, *et cetera*. Given that the complexity of communication is linear this difference of particles has little impact on the prediction of communication.

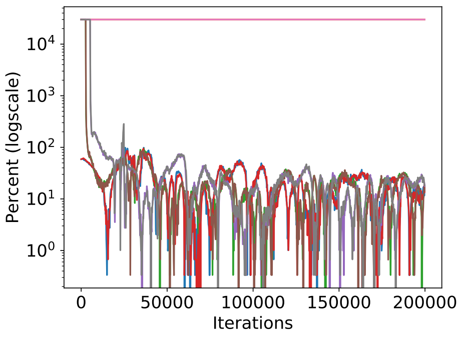
In order to devise a better estimate of the number of particles exchanged in the border exchange, it would be necessary to take into account the flow of the fluid at a given iteration



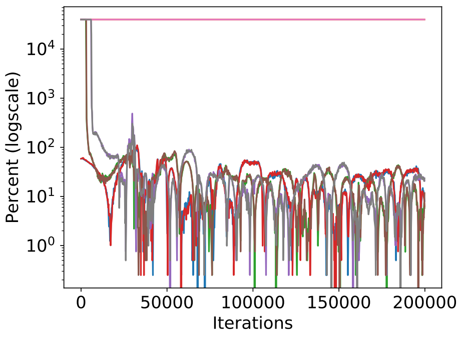
(a) `SCALE`=1



(b) `SCALE`=2



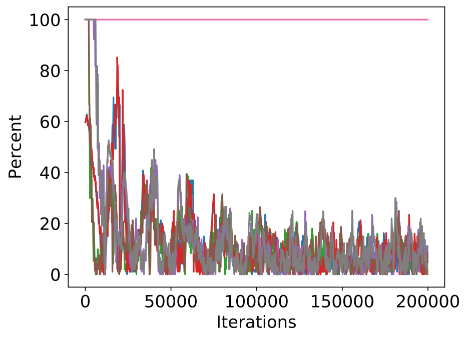
(c) `SCALE`=3



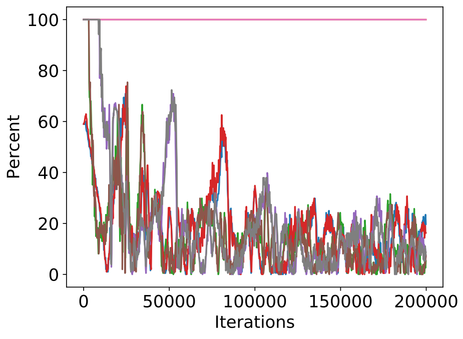
(d) `SCALE`=4

Figure 6.18: The percentage of the difference between the estimated number of particles being border exchanged and the actual number of particles using a logarithmic y-axis scale. Each curve represents one direction and one rank, using 4 ranks and `SCALE` 1 to 4. Notice that the range of the y axis change.

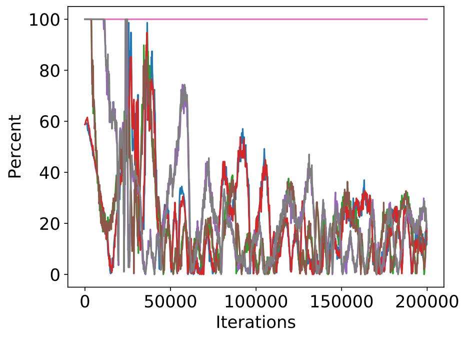
and the size of the region that overlaps the subdomain borders. However, in order to have the most precise picture of the flow of the fluid, we would need to run the simulation and extract empirical data. This defeats the purpose of using the performance models as a means to predict the execution on a given system. A better approach would be to approximate the flow of the fluid for a given number of ranks, iterations and `SCALE`.



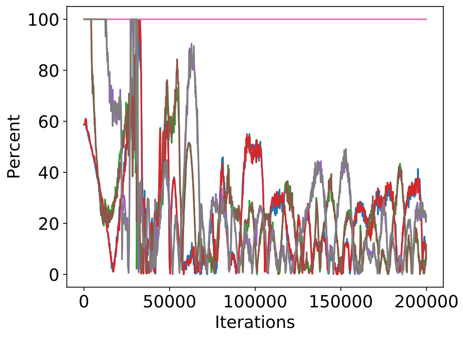
(a) SCALE=1



(b) SCALE=2



(c) SCALE=3



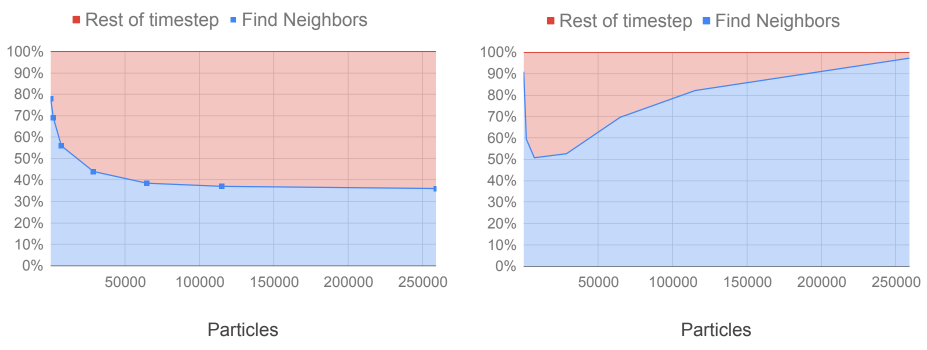
(d) SCALE=4

Figure 6.19: The percentage of the difference between the estimated number of particles being border exchanged and the actual number of particles, clamped at 100%. Each curve represents one direction and one rank, using 4 ranks and SCALE 1 to 4.

6.2.2 Finding neighbors & Time Step

The complexity of finding neighbors and the rest of the time step depends on the number of particles and the number of pairs. Finding neighbors depends only on the number of particles while the processing of the particles and the pairs in the rest of the time step depends on both number of particles and number of pairs. The number of pairs depends on the number of particles, but also configuration of the fluid. In different stages of the dam-break problem, the particles may be bunched up while crashing against the wall or having space around it while being in the air. This will affect the number of pairs and therefore the processing of pairs. However, the particle-pair relationship is close to linear which makes the complexity of the neighbor finding asymptotically higher than the rest of the physics step. This is why we expect to see an increase in the percentage while increasing SCALE.

Offloading only the neighbor-finding will reduce the bottleneck percent of the application more than offloading the whole time step. However, offloading the whole time step will further increase overall performance and might keep the bottleneck at the same level of importance. From our results we observe that the bottleneck is only slightly reduced, but the overall performance greatly increased. Figure 6.20 shows the percentage of the components of computation when only finding neighbors are offloaded to the GPU. From this, we can see how we can expect the growth and impact of finding neighbors will become when `SCALE` is increased. The cell-linked list approach can be seen to have leveled off the growth for our interval of `SCALE`. However, the similar growth to the naive method will occur when `SCALE` is sufficiently high. Given that this interval is already nearing the limits of our GPU hardware, the bottleneck of finding neighbors have been greatly reduced by offloading.



(a) Cell-linked list method.

(b) Naive method.

Figure 6.20: The percentage find neighbors make up of the time step with different methods when only finding neighbors is offloaded on Tesla V100.

Because this application is memory-bound in terms of the Roofline model, these results reflect how well the architecture and specific machines handle memory-bound applications. Although the GPUs have multiple times the theoretical peak performance of FLOPs of the CPUs, what makes the GPU more suited for this problem is the speed of the memory. While the Tesla V100 has a bandwidth of 900 GB/s, the Xeon E5-6132 v5 only has the bandwidth of about 120 GB/s in comparison. Tesla P100 has a similar high bandwidth of 720 GB/s. The theoretical peak bandwidth of 900 GB/s is hard to achieve in practice. For instance, the application would have to be able to use coalesced memory access. While this implementation does not give the opportunity of coalesced memory access, using only 20% of the speed as an example would give it a bandwidth of 180 GB/s and therefore still be faster than the bandwidth of the CPU. In addition, one of the largest portions of the memory used by the application is the particle pairs. Because the pairs are created, used and freed on the GPU, this means that the pairs does not have to be transferred over the slower bus of the node it is connected to.

6.2.3 Speedup & Efficiency

From both Figure 6.5 and 6.7, the cell-linked list GPU version can be seen to have a high Speedup over the naive baseline and the cell-linked list CPU version. However, the growth of the Speedup can be seen to start waning, while the corresponding CPU version does not. The fewer the ranks, the faster it starts to wane. This suggests that there is a point where the CPU version will catch up to the GPU version, given a sufficiently high value of `SCALE`. However, there are four things to consider:

- If the Speedup will ever converge at $S = 1$ or if it will converge at another higher value of Speedup.
- The maximum amount of particles that a GPU can hold and if that point is within that range without adding more ranks.
- If the increase of number of ranks offsets the waning of the Speedup too much for the CPU to catch up.
- When scaling horizontally the limiting factor always becomes communication. Here the GPU has the benefit of needing less ranks to achieve better performance so that the inevitability of communication as the limiting factor might be offset by the throughput.

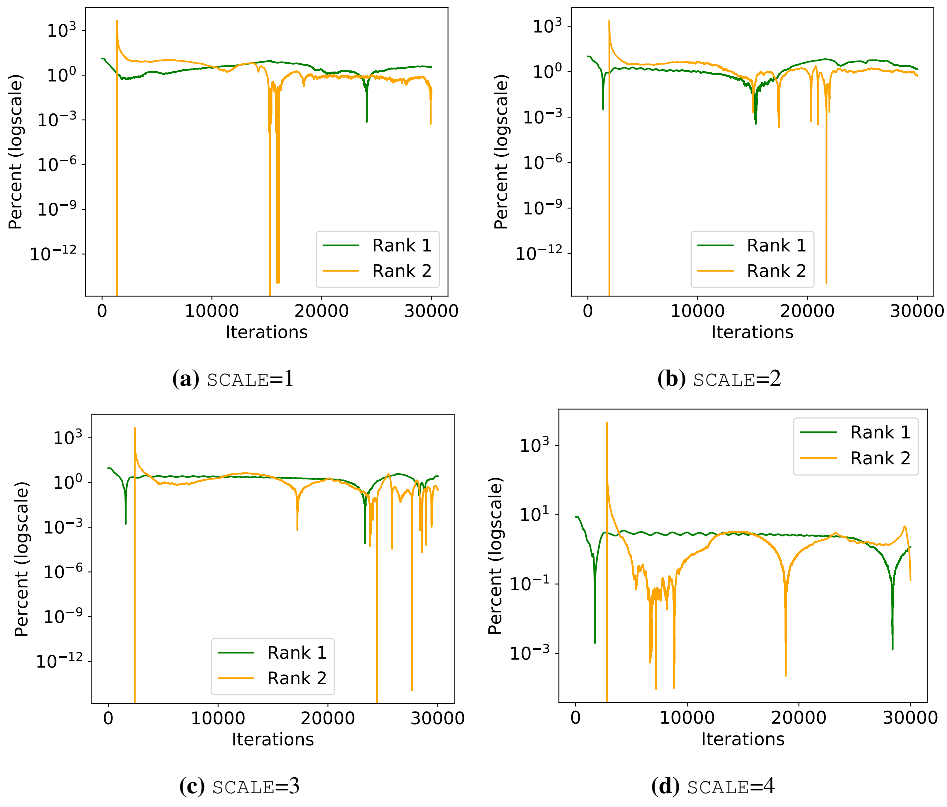
Figure 6.6 shows that the Efficiency of the GPU versions are very close to zero. This shows that we are not using the full potential of the GPU. The throughput of GPUs mainly come from two components, the fast bandwidth and highly parallel computation. As mentioned in Section 6.2.2, the memory bandwidth of GPUs benefit this problem because it is memory-bound according to the Roofline model, even if it does not fully utilize the bandwidth. The core of finding neighbors is discerning whether or not two particles are a pair. This requires a conditional statement which causes branching. As explained in Section 4.3.3, because SIMD architecture requires a single instruction, branching causes the threads to be serialized. This slows down the execution because it cannot run all threads in parallel at once. The launch parameters of a CUDA kernel will also affect how many cores that is used. If the problem size is too small to be divided over the maximum amount of thread blocks, all of the cores cannot be utilized.

6.2.4 Problem Scaling

Because of the way the workload is split, when increasing `SCALE`, adding more ranks will increase the overall performance of the application. However, doing so will increase the number of idle ranks. Increasing the problem scale also increases the number of iterations needed for the fluid to reach the next rank. According to the results at `SCALE`=4 with 4 ranks it takes up to about 15k iterations before all ranks are utilized. With this in mind, there is a balance to be struck between number of idle ranks and the performance of the early stages of the execution in order to have enough ranks to accelerate the early stages so that the east-most ranks are not idle for long. However, this is mainly a problem when the configuration of the fluid is in the dam-break setup and the rank-workload sharing is done in a static manner, such as in this thesis. With the dam-break problem specifically in mind, the distribution of ranks can be optimized better. An example of this would be to have more ranks at each end of the ”tank” containing the fluid. Although this would not be needed if some kind of dynamic workload sharing were to be used. Depending on implementation, this might make the initial configuration of the fluid irrelevant. This would however be much harder to implement and increase the communication between the ranks.

6.2.5 Utility range

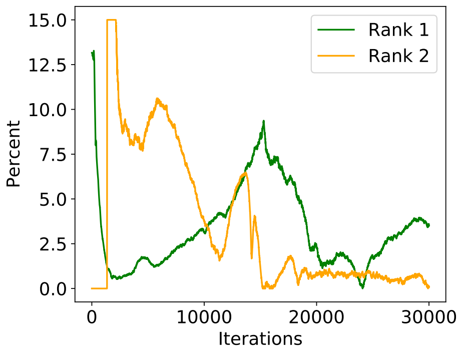
The results show that the min value for the utility range of the cell-linked list version is zero, because offloading is faster no matter the values of SCALE or rank for the hardware used in this thesis. The same is true for the naive version. However, the naive version has a max value not connected to memory usage, because the scaling of this method is worse than the cell-linked list versions. For the V100 nodes, using Intel Xeon E5-6132 v5 and Tesla V100, we have come to understand that the max value for the naive method is max ≈160 000 while using 8 ranks. The less ranks used, the higher the value of max becomes and similarly the more ranks being used, the less the value of max becomes.



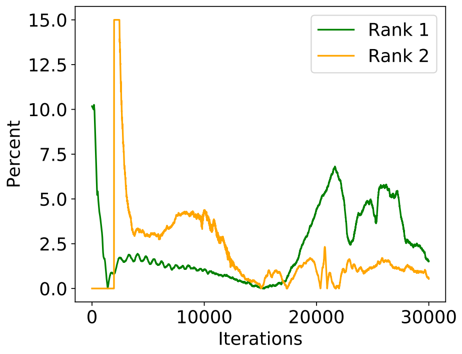
(a) SCALE=1 **(b)** SCALE=2 **(c)** SCALE=3 **(d)** SCALE=4

Figure 6.21: The percentage of the difference between the estimated number of pairs and the actual number of pairs using a logarithmic y-axis scale. Notice that the range of the y axis change.

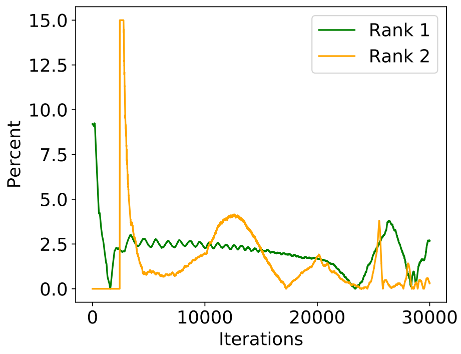
The estimate of the number of pairs for a given amount of particles is essential to the implementation of this problem as well as the max part of the utility range. If the amount of iterations and the SCALE does not allow the simulation to run for long enough, the estimate might come with a large error margin. Figure 6.14 shows that ending the simulation after 5000 steps, the error margin has been as much as $7.5k$ for SCALE=1, $10k$ for SCALE=2, $50k$ for SCALE=3 and $100k$ for SCALE=4. From this pattern of increase, we can see that the maximum absolute error increases by a magnitude for each doubling of the SCALE. This implies that it is even more important on higher values of SCALE to run sufficient amount of iterations.



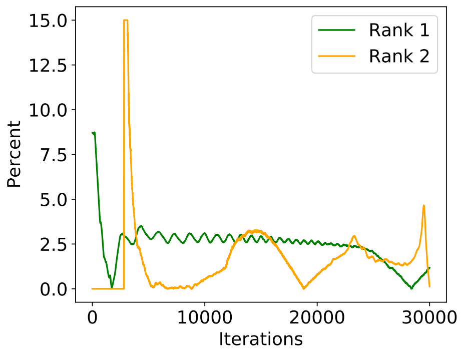
(a) SCALE=1



(b) SCALE=2



(c) SCALE=3



(d) SCALE=4

Figure 6.22: The percentage of the difference between the estimated number of pairs and the actual number of pairs, clamped at 15%.

A more precise estimate will have to take into account the motion of the fluid, the SCALE and the current iteration. Figure 6.21 shows how many percent difference of the estimate and the number of pairs make up of the total number of pairs, formulated by Equation 6.13. The figure shows that the percentage of error is approximately 0% at all times, except for a large spike of error somewhere between the 0th and the 5000th iteration, peaking at somewhere between $2500 - 5000\%$. The spike only occurs in rank 2 and is caused by the large error in the estimate when the number of particles are few. This happens when the particles start migrating from rank 1 to rank 2. The first couple of particles have high speed because of the dam breaking and are spread out. This means that there are particles, but no pairs. When the first particles form a pair, the error is high for around 50 iterations for

SCALE=1. An example would be 10 particles have migrated and the last two form a pair: $P = 10, N_{pair} = 1 \implies \epsilon = 23.6 * 10 - 1 = 235$ which would give the error percentage of 23500%. The duration of the spike declines as the SCALE rises because more particles are migrating the border at the same time and form pairs more quickly. This can be seen in Figure 6.22. The more particles migrating, the more correct the estimate becomes.

$$\frac{\epsilon_{i,r}}{N_{i,r}} = \frac{\mid 23.6P - N_{i,r} \mid}{N_{i,r}} \tag{6.13}$$

Figure 6.22 shows the same percentage of error as Figure 6.21, only clamped at 15%. We observe the effect of violent flow in the fluid causing greater error. At $t = 0$, the error in rank 1 is at its highest. This is because the particles are arranged in a grid and not interacting with each other. After the fluid is allowed to move, the error decreases.

As the SCALE increases, the error decreases and becomes more stable as shown in Figure 6.22. The violent flow does not cause as big increases in error as at lower values of SCALE. At SCALE=1, the highest error lies around 10%, SCALE=2 lies around 7%, SCALE=3 lies around 4% and SCALE=4 lies just below 4%. With this trend, when SCALE grows, the percentage of error approaches $2 - 3\%$. This implies that using the estimate at higher scales will give more accurate estimates than at lower scales.

6.3 Validation

This section will validate the different parts of the performance model, split into **Computation** and **Communication**. Tables 6.1 and 6.2 summarize the models for computation and communication respectively and refers to the figures corresponding to the models.

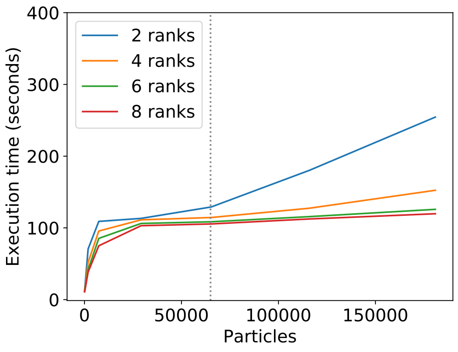
6.3.1 Computation

Part	Model	Fig
Computation	$T_{find\_neighbors} + T_{physics} + T_{transfer}$	
Find Neighbors	$T_{setup}(P, alg) + T_{create\_pairs}(P, alg)$	6.23
Physics	$T_{kernel} + T_{cont\_density} + T_{corr} + T_{int\_force} + T_{ext\_force}$	6.24
Transfer	$2P \times C_1$	6.25

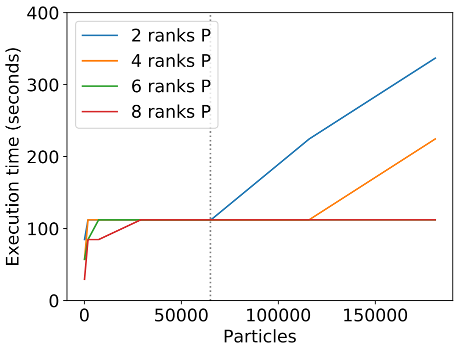
Table 6.1: The performance models of computation which the experiments measure.

Finding Neighbors

Figure 6.23 shows the execution time of finding neighbors of the experimental data from the V100 cluster and the predicted execution time from the performance model. Here we see that the prediction follows the same pattern of growth as the experimental data, with a sharp increase that then evens out before steadily increasing, depending on rank. While it seems many of the predicted executions have evened out completely, this is because when calculating the N_b term of the CUDA performance model, we round the answers up to nearest whole number because the streaming multiprocessors cannot execute non-whole thread blocks when we request more thread blocks than streaming multiprocessors. In reality, this one overflowing thread block would be scheduled to the streaming multiprocessor that finishes first. The prediction of an execution with 2 ranks have an increase at the same number of particles, around P =65 000, shown in Figure 6.23 as the vertical line.



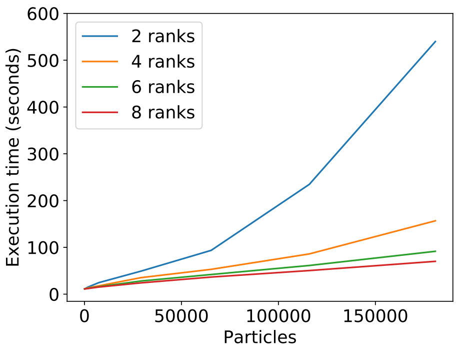
(a) Data from V100.



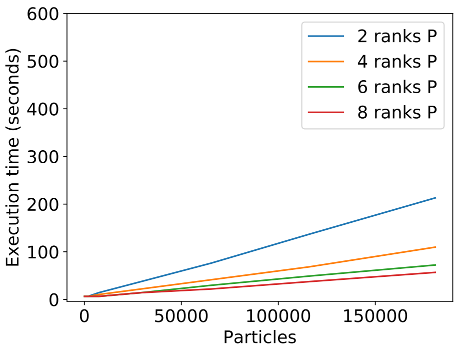
(b) Prediction for V100.

Figure 6.23: Execution time of finding neighbors of the cell-linked list GPU version. The vertical line shows an increase in execution time in both data and prediction for 2 ranks.

Physics



(a) Data from V100.



(b) Prediction for V100.

Figure 6.24: Execution time of physics computation of the GPU version.

Figure 6.24 shows the execution time of the experimental data from the V100 cluster and the predicted execution time from the performance model. Here we can see that the prediction follows the data very closely except for when there are 2 and 4 ranks. The prediction for the ranks are also in the right order of each other. However, the growth of the prediction is similar to the growth of the data up until a certain number of particles. Because the underlying components of the data and prediction are linear in complexity, we attribute this kind of quadratic growth to oversaturation of the GPU. This also explains why the execution with 4 ranks deviates from the linear growth at a higher number of particles, because workload sharing between the ranks offsets the oversaturation. This oversaturation is not accounted for in the performance models, which is why this effect does not appear in the predicted execution time. This also shows the importance of adding enough ranks, so that the GPU is not oversaturated when executing the application. The prediction relies on the pair estimate of $e_p = 23.6$ which is also a source of error.

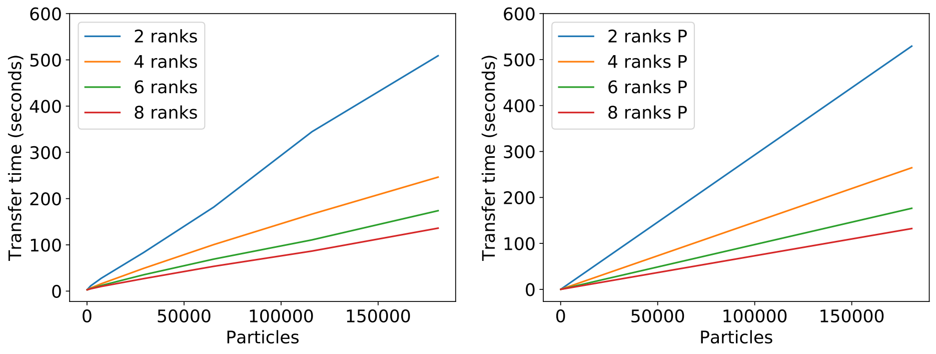
Transfer

$$T_{transfer} = 2P \times C_1 \tag{6.14}$$

Figure 6.25 shows the time spent transferring particles from main memory to GPU and back of the experimental data from the V100 cluster and the predicted time from the performance model. The bottleneck of the data transfer is the PCI-Express that connects the GPUs to the board. Both GPUs have PCI-E 3.0 and support a maximum of 16 lanes. This means that it has a theoretical throughput of $15.8GB/s$. Using this we can calculate our constants to be:

$$C_1 = \frac{sizeof(S_{particle})[B]}{transfer\_rate[B/s]} = \frac{104}{15.8 \times 10^9} s \tag{6.15}$$

However, there are two major factors to the difference in transfer model prediction and



(a) Data from V100. **(b)** Prediction for V100.

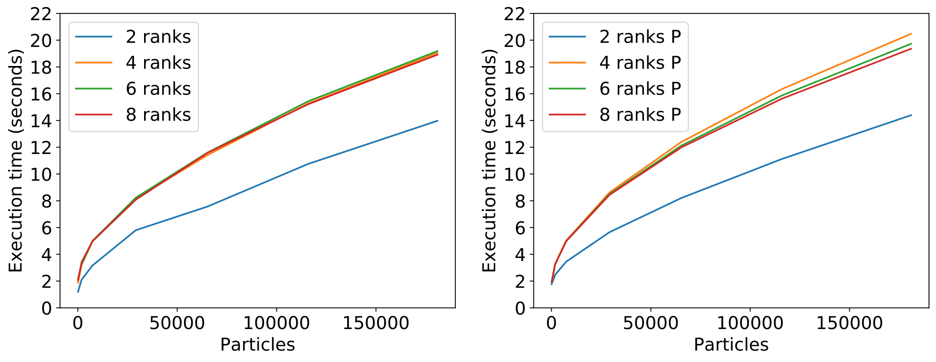
Figure 6.25: Time spent transferring particles from main memory to GPU and back.

experimental data. The first major factor is that there is an error of the distribution of particles because the data is gathered over many iterations and the model assumes the particles are evenly distributed among the ranks. This however, will diminish as the number of iterations increase. The other major factor is the transfer rate of PCI-E 3.0 with 16 lanes is a theoretical throughput and is difficult to achieve in an experimental setting. This is why the prediction does not use the theoretical transfer rate, but a measured transfer rate of $3.5GB/s$. This value was measured by transferring large amounts of data to the GPU and dividing the amount by the execution time of the transfer. The reason why the measured value is much lower than the theoretical throughput is due to the fact that the two ranks executing on the same node operate on the same bus and therefore have to share the bandwidth of that bus.

6.3.2 Communication

Part	Model
Communication	$T_{migrate} + T_{bexchange}$
Border Exchange	$s \times (T_{bexchange\_count} + T_{bexchange\_transfer}) + T_{bexchange\_setup}$
Migrate	$s \times (T_{migrate\_count} + T_{migrate\_transfer}) + T_{migrate\_setup}$

Table 6.2: The performance models of communication which the experiments measure. Prediction of communication can be seen in Figure 6.26.



(a) Data from V100. **(b)** Prediction for V100.

Figure 6.26: Execution time of communication.

Figure 6.26 shows the recorded data from the experiments and the predicted execution time for the communication of the proxy application. Here we can see in both data and prediction that running on 2 ranks will reduce the execution time. This is because there are only two nodes that are communicating with each other. Therefore, the nodes only have one other rank to communicate with, approximately halving the communicational amount. Because a node only communicates with its neighbors, this has no further effect when adding ranks above 2 ranks. Adding more ranks does however lower the setup term because each rank will have fewer particles to prepare for sending as well as smaller number of particles to send. This is why the prediction predicts a gap between the ranks above 2 with the most ranks being the fastest and the fewest ranks being the slowest. This effect can also be seen from the data, although the gap is smaller. The prediction tells us that with high values of `SCALE`, the gap will grow bigger for the data as well.

6.4 Recommendations

Based on these results we can make the following recommendations. In general, the cell-linked list method should be used regardless of `SCALE` or number of ranks, except when the particles are few enough that the naive method outperforms the cell-linked list method on GPU. However, using the naive CPU approach is not recommended. Generally, offloading the problem to the GPU will perform better than running it on the CPU and the characteristics of scaling seems to show this trend as long as the combination of `SCALE` and ranks are within the max memory usage of the GPU's available memory. To see if a specific GPU's memory is within that maximum, Equation 6.6 should be used to calculate maximum number of particles for that GPU. Should the desired amount of particles be higher than the maximum, the problem can be divided over more GPUs, which might put the number of particles per subdomain below the maximum. The more ranks being used, the more likely the Equation 6.7 is to be correct and more particles can be processed by the GPUs.

Because this problem scales well horizontally, adding more ranks will improve the performance and is important when increasing the problem scale. However, because HPC nodes are usually a shared resource, there is a balance to be struck between values of `SCALE` and number of ranks such that the east-most ranks does not stay idle too long and that there are enough ranks so that the fluid is allowed to propagate quickly to the other ranks from the start of the dam-break problem. Adding more ranks will however, decrease Efficiency for both CPU and GPU versions.

There are two factors to consider regarding the choice of hardware, speed versus expense. While the GPU version is able to achieve a Speedup of 10 over the CPU version at most, the cost of GPUs are generally much higher than the cost of CPUs. Therefore, we can reduce the choice to a matter of *fast, but expensive* or *economical, but slow*.

Chapter 7

Conclusion & Future Work

7.1 Conclusion

In this thesis, we have implemented a GPGPU version of the proxy application built on WCSPH for the dam-break problem. Using these implementations we have created performance models to predict the performance as well as a utility range for use of the CUDA GPGPU version.

We find that the utility range, $[min < P < max]$, has a min term of zero. Therefore offloading finding neighbors and the physics computation to the GPU is always beneficial for the hardware used in this thesis. If the number of simulated particles is small enough, using a brute-force approach to finding neighbors can be faster than the more advanced methods.

We predicted that using the maximum number of ranks according to the problem size with the GPU version will cause the communicational amount to outgrow the computational amount. Furthermore, we predicted that the optimal number of ranks for overall performance of the application is the same as the maximum number of ranks.

We find that offloading the neighbor-finding reduces its impact as the bottleneck, as well as reduces the computational amount of the application as a whole. This scales well for problem sizes within the utility range.

We proposed a formula for the maximum problem size for a given size of memory of a GPU for the cell-linked list version.

We proposed the linear estimate for the number of pairs per particles, $e_p = 23.6$, and the linear estimate for the number of particles exchanged in communication, $b_p = 100 \times SCALE$, that are used in the performance models. We examined the error of both estimates and where they stem from. We concluded that e_p is within the error range of $[0, 10]\%$ and b_p is within the error range of $[0, 20]\%$ given sufficient iterations.

We found that the Speedup of the GPU version, using the CPU version as a baseline, is at most 10. However, we found that the Efficiency of the GPU version is considerably worse than the CPU version.

We discovered that the proxy application scales well and that there are no performance drawbacks from horizontal scaling.

7.2 Future Work

There is a lot of interesting further work to be done relating to this thesis.

Offloading has proven to increase the performance as well as the scalability. However, while offloading, all cores of the CPU are idle while one thread feeds the GPU with instructions. With smart workload balancing, the idle threads of the CPU can be given an appropriate amount of work while the GPU takes the bigger part of the workload. This would utilize more of the hardware being used, raise efficiency and increase performance.

The communication is serial when the ranks communicate with its neighbors. This could be paralellized in order to utilize more of the bandwidth and decrease the idle time of the CPU.

There are many algorithms that can be used to decrease the amounts of comparisons needed in a N-body problem. An approach that was outside the scope of this thesis is the Verlet list. This approach has potential to be more efficient since it can account for a extra padding of smoothing length so as not to need updating each iteration. Another approach is the quadtree method that splits all areas into four until all particles has their own area.

However, the particles rarely move enough between each iteration to reconstruct both pair-wise lists and cell-linked lists. This should be investigated as to how scale affects how many updates are needed every nth-iteration, and what the lower bound of updates per iteration is before the simulation does not adhere to the physical realm. This has the potential to be used in combination with the Verlet list to decrease the number of updates needed and thus increasing the performance of the proxy application.

Because of the scaling characteristics of the problem and application, the static workload sharing leaves ranks idle in the start of the dam-break simulation, while the first rank has a lot of work. This evens out throughout the simulation, depending on how many iterations are done. The workload could be shared in a smarter way between the ranks so that no ranks are idle throughout the simulation.

Bibliography

Amdahl, G. M., 1967. Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, Spring Joint Computer Conference. AFIPS '67 (Spring). ACM, New York, NY, USA, pp. 483–485. URL http://doi.acm.org/10.1145/1465482.1465560

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., Yelick, K. A., 2006. The landscape of parallel computing research: A view from berkeley. Tech. rep., TECHNICAL REPORT, UC BERKELEY.

Baghsorkhi, S. S., Delahaye, M., Patel, S. J., Gropp, W. D., Hwu, W.-m. W., 2010. An adaptive performance modeling tool for gpu architectures. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '10. ACM, New York, NY, USA, pp. 105–114. URL http://doi.acm.org/10.1145/1693453.1693470

Barker, K. J., Davis, K., Hoisie, A., Kerbyson, D. J., Lang, M., Pakin, S., Sancho, J. C., Nov 2009. Using performance modeling to design large-scale systems. Computer 42 (11), 42–49.

Barnes, J., Hut, P., Dec 1986. A hierarchical o(n log n) force-calculation algorithm. Nature 324, 446. URL https://doi.org/10.1038/324446a0

Dongarra, J. J., Cruz, J. D., Hammarling, S., Duff, I. S., Mar. 1990. Algorithm 679: A set of level 3 basic linear algebra subprograms: Model implementation and test programs. ACM Trans. Math. Softw. 16 (1), 18–28. URL http://doi.acm.org/10.1145/77626.77627

Esmaeilzadeh, H., Blem, E., Amant, R. S., Sankaralingam, K., Burger, D., 2011. Dark silicon and the end of multicore scaling. In: 2011 38th Annual international symposium on computer architecture (ISCA). IEEE, pp. 365–376.

Flynn, M. J., Dec 1966. Very high-speed computing systems. Proceedings of the IEEE 54 (12), 1901–1909.

Gingold, R. A., Monaghan, J. J., 1977. Smoothed particle hydrodynamics: theory and application to non-spherical stars. Monthly notices of the royal astronomical society 181 (3), 375–389.

Green, S., 2010. Particle simulation using cuda. NVIDIA whitepaper 6, 121–128.

Gustafson, J. L., 1988. Reevaluating amdahl's law. Communications of the ACM 31 (5), 532–533.

Harris, M., 2016. Cuda fluid simulation in nvidia physx. URL: `http://sa08.idav.ucdavis.edu/CUDA_physx_fluids.Harris.pdf`.

Hockney, R. W., 1994. The communication challenge for mpp: Intel paragon and meiko cs-2. Parallel Computing 20 (3), 389 – 398.
URL `http://www.sciencedirect.com/science/article/pii/S0167819106800219`

Kothapalli, K., Mukherjee, R., Rehman, M. S., Patidar, S., Narayanan, P., Srinathan, K., 2009. A performance prediction model for the cuda gpgpu platform. In: High Performance Computing (HiPC), 2009 International Conference on. IEEE, pp. 463–472.

Lastovetsky, A., Rychkov, V., O'Flynn, M., 2010. Accurate heterogeneous communication models and a software tool for their efficient estimation. The International Journal of High Performance Computing Applications 24 (1), 34–48.
URL `https://doi.org/10.1177/1094342009359012`

Lucy, L. B., 1977. A numerical approach to the testing of the fission hypothesis. The astronomical journal 82, 1013–1024.

McCalpin, J. D., et al., 1995. Memory bandwidth and machine balance in current high performance computers. IEEE computer society technical committee on computer architecture (TCCA) newsletter 2 (19–25).

McCool, M., Reinders, J., Robison, A., 2012. Structured Parallel Programming: Patterns for Efficient Computation, 1st Edition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

McCool, M. D., May 2008. Scalable programming models for massively multicore processors. Proceedings of the IEEE 96 (5), 816–831.

Monaghan, J., 1994. Simulating free surface flows with sph. Journal of Computational Physics 110 (2), 399 – 406.
URL `http://www.sciencedirect.com/science/article/pii/S0021999184710345`

Monaghan, J., Kos, A., 1999. Solitary waves on a cretan beach. Journal of waterway, port, coastal, and ocean engineering 125 (3), 145–155.

Ozbulut, M., Yildiz, M., Goren, O., 2014. A numerical investigation into the correction algorithms for sph method in modeling violent free surface flows. International Journal of Mechanical Sciences 79, 56–65.

Ozbulut M., Arslan T., 2018. personal communication.

Pacheco, P., 2011. An introduction to parallel programming. Elsevier.

Patterson, D. A., Oct. 2004. Latency lags bandwith. Commun. ACM 47 (10), 71–75.
URL http://doi.acm.org/10.1145/1022594.1022596

Rabenseifner, R., 2003. Hybrid parallel programming on hpc platforms. In: proceedings of the Fifth European Workshop on OpenMP, EWOMP. Vol. 3. pp. 185–194.

Ragunathan, J., Valstad, J., 2018. Performance modeling of cfd application scalability using co-design methods. Master's thesis, Norwegian University of Science and Technology (NTNU).

Reissman, N., Meyer, J. C., Jahre, M., May 2014. A study of energy and locality effects using space-filling curves. In: 2014 IEEE International Parallel Distributed Processing Symposium Workshops. pp. 815–822.

Valiant, L. G., Aug. 1990. A bridging model for parallel computation. Commun. ACM 33 (8), 103–111.
URL http://doi.acm.org/10.1145/79173.79181

Verlet, L., Jul 1967. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. Phys. Rev. 159, 98–103.
URL https://link.aps.org/doi/10.1103/PhysRev.159.98

Williams, S., Waterman, A., Patterson, D., 2009. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Tech. rep., Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States).

Appendix A

Selected Source Code

```
__global__ void naive(const int_t n_total, particle_t* particles, pair_t *all_pairs,
  unsigned int* pair_count, int* interactions) {
    const int_t i = blockDim.x * blockIdx.x + threadIdx.x;
    const int_t j = blockDim.y * blockIdx.y + threadIdx.y;

    if (i == j || i >= n_total - 1 || j <= i || j >= n_total)
        return;

    const real_t difx = particles[i].x[0] - particles[j].x[0];
    const real_t dify = particles[i].x[1] - particles[j].x[1];
    if (difx*difx + dify * dify < POW2(scale_k*H)) {
        // Is pair
        unsigned int kk = atomicAdd(pair_count, 1);
        interactions[i] += 1;
        interactions[j] += 1;
        all_pairs[kk].i = i;
        all_pairs[kk].j = j;
        all_pairs[kk].r = sqrt(difx*difx + dify * dify);
        all_pairs[kk].q = sqrt(difx*difx + dify * dify) / H;
        all_pairs[kk].w = 0.0f;
        all_pairs[kk].dwdx[0] = all_pairs[kk].dwdx[1] = 0.0f;
    }
}
```

Listing A.1: The CUDA kernel of the naive approach to finding neighbors.

```
#define THREADS_PER_BLOCK 512
#define SHIFTCID(dx, dy) (cell_id + dy + dx * dev_n_cells_y)
#define GET_CELL_ELEMENT(cell_id) ((unsigned int *)((char*)cells + cell_id * pitch))
__device__ inline void compare(int_t cell_id, int_t n_cells, unsigned int** cells,
     particle_t* particles, particle_t particle, int* interactions, pair_t * pairs,
     unsigned int* pair_count, size_t pitch, unsigned int* cell_particle_count) {
    if (cell_id < 0 || cell_id >= n_cells)
        return;
    const unsigned int total_p = min(cell_particle_count[cell_id], INITIAL_MAX_cell
     -1);
    unsigned int* cell = GET_CELL_ELEMENT(cell_id);
    for (unsigned int i = 0; i < total_p; i++) {
        particle_t other_particle = particles[cell[i]];
        if (other_particle.local_idx < particle.local_idx) {
            const real_t difx = particle.x[0] - other_particle.x[0];
            const real_t dify = particle.x[1] - other_particle.x[1];
            if (difx*difx + dify * dify < POW2(scale_k*H)) {
                interactions[particle.local_idx]++;
                interactions[other_particle.local_idx]++;

                pair_t loc_pair;
                loc_pair.i = particle.local_idx;
                loc_pair.j = other_particle.local_idx;
                loc_pair.r = sqrt(difx*difx + dify * dify);
                loc_pair.q = sqrt(difx*difx + dify * dify) / H;
                loc_pair.w = 0.0f;
                loc_pair.dwdx[0] = loc_pair.dwdx[1] = 0.0f;
                unsigned int pair_idx = atomicAdd(pair_count, 1);
                pairs[pair_idx] = loc_pair;
            }
        }
    }
}

__global__ void
__launch_bounds__(THREADS_PER_BLOCK)
 find_pairs(unsigned int** cells, int_t n_cells, particle_t* particles, int_t
     n_particles, int* interactions, pair_t * pairs, unsigned int* pair_count, size_t
      pitch, unsigned int* cell_particle_count) {
    const int_t p_id = blockDim.x * blockIdx.x + threadIdx.x;
    if (p_id >= n_particles)
        return;

    particle_t particle = particles[p_id];
    const int cell_id = particle.cell_id;
    // Center / local
    compare(cell_id, n_cells, cells, particles, particle, interactions, pairs,
     pair_count, pitch, cell_particle_count);
    // North West
    compare(SHIFTCID(-1, 1), n_cells, cells, particles, particle, interactions, pairs
     , pair_count, pitch, cell_particle_count);
    // North
    compare(SHIFTCID(0, 1), n_cells, cells, particles, particle, interactions, pairs,
      pair_count, pitch, cell_particle_count);
    // North East
    compare(SHIFTCID(1, 1), n_cells, cells, particles, particle, interactions, pairs,
      pair_count, pitch, cell_particle_count);
    // East
    compare(SHIFTCID(1, 0), n_cells, cells, particles, particle, interactions, pairs,
      pair_count, pitch, cell_particle_count);
    // South East
    compare(SHIFTCID(1, -1), n_cells, cells, particles, particle, interactions, pairs
     , pair_count, pitch, cell_particle_count);
    // South
    compare(SHIFTCID(0, -1), n_cells, cells, particles, particle, interactions, pairs
     , pair_count, pitch, cell_particle_count);
    // South West
    compare(SHIFTCID(-1, -1), n_cells, cells, particles, particle, interactions,
     pairs, pair_count, pitch, cell_particle_count);
    // West
    compare(SHIFTCID(-1, 0), n_cells, cells, particles, particle, interactions, pairs
     , pair_count, pitch, cell_particle_count);
}
```

Listing A.2: The CUDA kernel of the cell-linked list approach to finding neighbors.

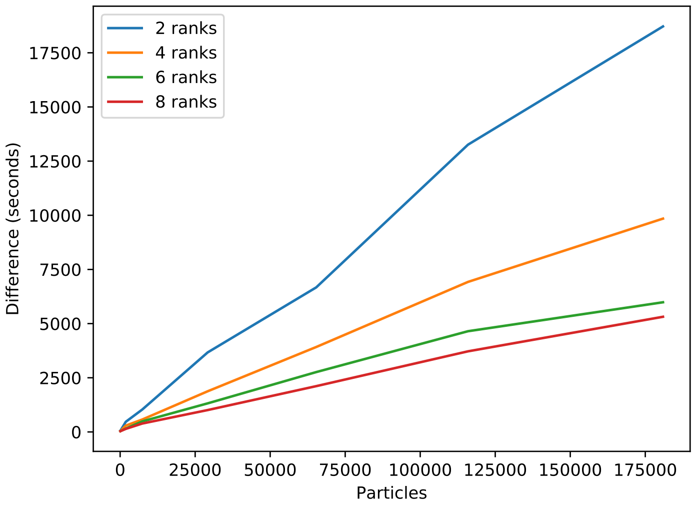
```
void create_pairs(int cx, int cy, cell_t** cells,
                  particle_t* particle, int_t* n_pairs,
                  int_t* interactions) {
    if (cx >= n_cells_x || cx < 0 || cy >= n_cells_y || cy < 0 || particle == NULL) {
        return;
    }
    cell_t* current = cells[CID(cx, cy)];
    while (current != NULL && current->particle != NULL) {
```

```c
        if (current->particle->local_idx < particle->local_idx) {
            real_t distance = sqrt(
                POW2(particle->x[0] - current->particle->x[0]) +
                POW2(particle->x[1] - current->particle->x[1])
            );
            if (distance <= RADIUS) {
                interactions[particle->local_idx]++;
                interactions[current->particle->local_idx]++;
                int pair_idx;
                #pragma omp atomic capture
                pair_idx = (*n_pairs)++;
                pairs[pair_idx].i = particle->local_idx;
                pairs[pair_idx].j = current->particle->local_idx;
                pairs[pair_idx].ip = particle;
                pairs[pair_idx].jp = current->particle;
                pairs[pair_idx].r = distance;
                pairs[pair_idx].q = distance / H;
                pairs[pair_idx].w = 0.0f;
                pairs[pair_idx].dwdx[0] = pairs[pair_idx].dwdx[1] = 0.0f;
            }
        }
        current = current->next;
    }
}
void find_neighbors( void ) {
    int_t n_total = n_field + n_virt + n_mirror;
    n_pairs = 0;
    #pragma omp parallel for
    for ( int_t k=0; k<n_total; k++ )
        INTER(k) = WSUM(k) = AVRHO(k) = 0;
    if ( n_pair_cap < (n_total*PAIR_ESTIMATE) )
        resize_pair_list ( n_total*PAIR_ESTIMATE );
    omp_lock_t lock[n_cells_x*n_cells_y];
    for (int i=0; i<n_cells_x*n_cells_y; i++)
        omp_init_lock(&(lock[i]));
    #pragma omp parallel
    {
        /* Init cells */
        #pragma omp for nowait
        for (int x = 0; x < n_cells_x; ++x) {
            for (int y = 0; y < n_cells_y; ++y) {
                if (cells[CID(x, y)] != NULL) {
                    cells[CID(x, y)]->particle = NULL;
                    cells[CID(x, y)]->next = NULL;
                }
            }
        }
        /* Compute cell_x and cell_y for all particles */
        #pragma omp for
        for (int i = 0; i < n_total; ++i) {
            particle_t *particle = &list[i];
            int_t actual_x = MIN((int_t) (((particle->x[0] - subdomain[0])+RADIUS) / cell_RADIUS) , n_cells_x -1);
            int_t actual_y = MIN((int_t) ((particle->x[1]+1.55f*H) / cell_RADIUS), n_cells_y -1);
            particle->local_idx = i;
            particle->cell_x = actual_x;
            particle->cell_y = actual_y;
        }
        /* Fill cells */
        fill_cells(cells, n_total, lock);
        int_t* interactions = calloc(n_total, sizeof(int_t));
        /* Create neighbors */
        #pragma omp for
        for (int_t i = 0; i < n_total; ++i) {
            particle_t* particle = &list[i];
            int cx = particle->cell_x;
            int cy = particle->cell_y;
            /* Center */
            create_pairs(cx, cy, cells, particle, &n_pairs, interactions);
            /* North West */
            create_pairs(cx-1, cy+1, cells, particle, &n_pairs, interactions);
            /* North */
            create_pairs(cx, cy+1, cells, particle, &n_pairs, interactions);
            /* North East */
            create_pairs(cx+1, cy+1, cells, particle, &n_pairs, interactions);
            /* East */
            create_pairs(cx+1, cy, cells, particle, &n_pairs, interactions);
            /* South East */
            create_pairs(cx+1, cy-1, cells, particle, &n_pairs, interactions);
            /* South */
            create_pairs(cx, cy-1, cells, particle, &n_pairs, interactions);
            /* South West */
            create_pairs(cx-1, cy-1, cells, particle, &n_pairs, interactions);
            /* West */
            create_pairs(cx-1, cy, cells, particle, &n_pairs, interactions);
    }
        /* Collect interactions */
        if (n_total > 0) {
            int t = omp_get_thread_num();
            int numThreads = omp_get_num_threads();
            int privateStart = n_total*t/numThreads;
            int privateStop = (privateStart + n_total - 1) % n_total;
            int_t i = privateStart;
            while(1) {
                #pragma omp atomic
                INTER(i) += interactions[i];
                if(i == privateStop)
                    break;
                i = (i + 1)%n_total;
            }
        }
        free(interactions);
        /* Free cells */
        #pragma omp for
        for (int x = 0; x < n_cells_x; ++x) {
            for (int y = 0; y < n_cells_y; ++y) {
                cells[CID(x, y)]->particle = NULL;
                if (cells[CID(x, y)]->next != NULL) {
                    free_cells(cells[CID(x, y)]->next);
                }
            }
        }
    }
    for (int i=0; i<n_cells_x*n_cells_y; i++)
        omp_destroy_lock(&(lock[i]));
}
```

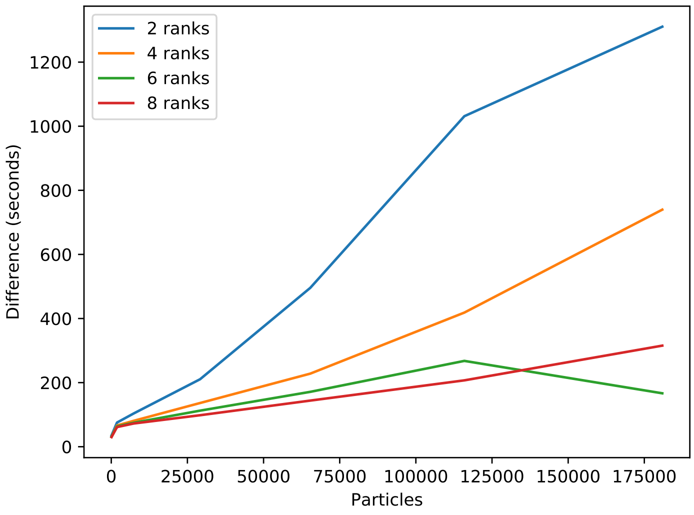
Listing A.3: The cell-linked list approach to finding neighbors in C.

Appendix B

EPIC2 Results & Validation

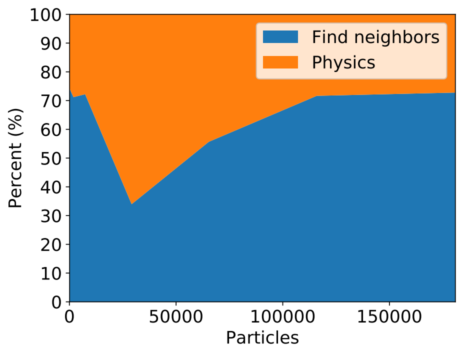


(a) CPU version

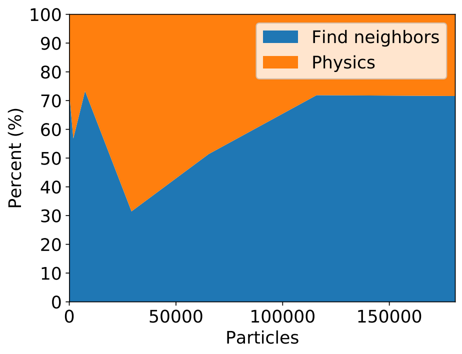


(b) GPU version

Figure B.1: The difference between time spent computing and time spent communicating using the cell-linked list method on the EPIC2 queue. Notice that the range of the y axis change.

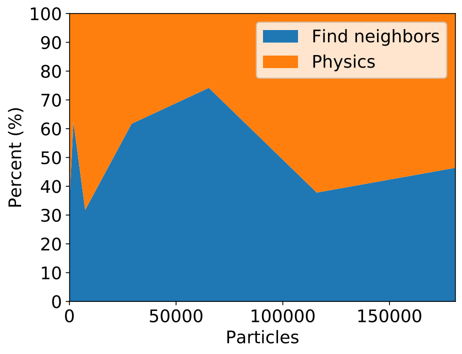


(a) Using 2 ranks

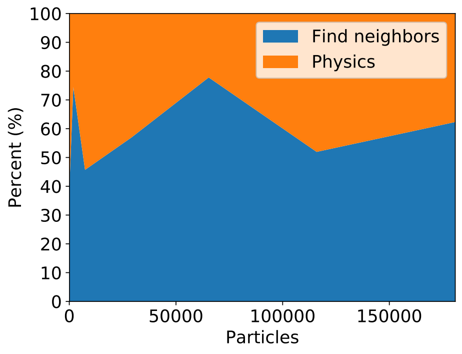


(b) Using 8 ranks

Figure B.2: The percentage of time the components of time step uses when running the CPU version on EPIC2.

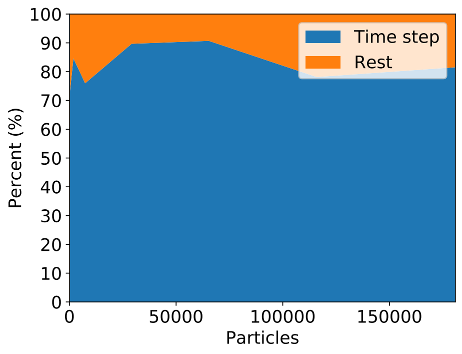


(a) Using 2 ranks

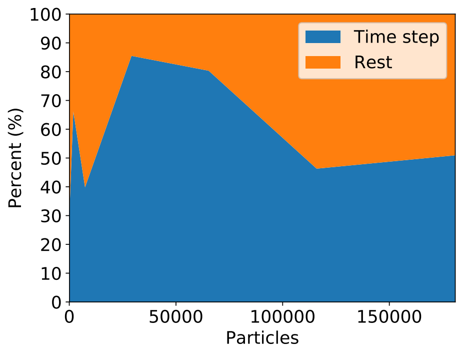


(b) Using 8 ranks

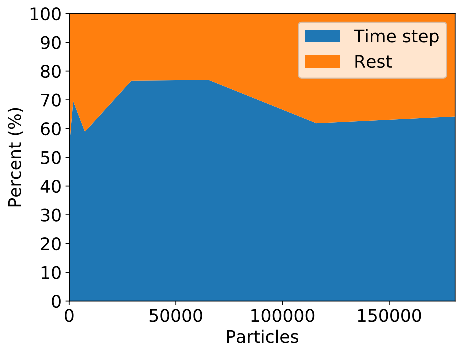
Figure B.3: The percentage of time the components of time step uses when running the GPU version on EPIC2.



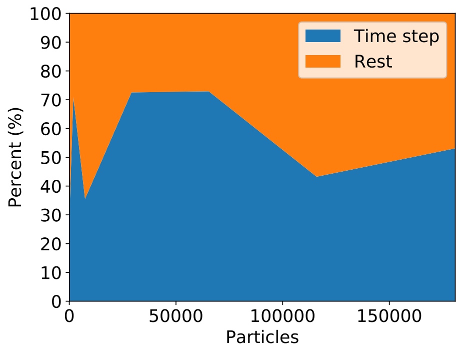
(a) CPU version, using 2 ranks



(b) GPU version, using 2 ranks

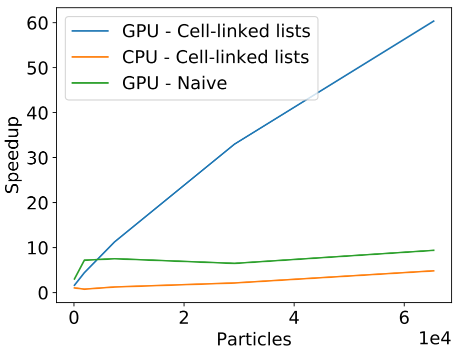


(c) CPU version, using 8 ranks

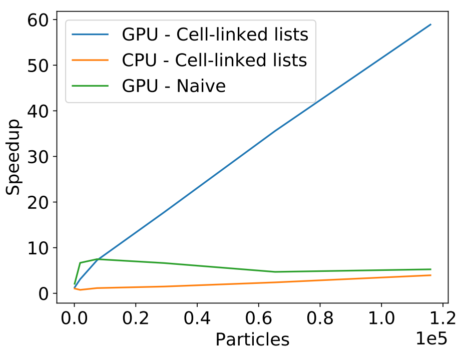


(d) GPU version, using 8 ranks

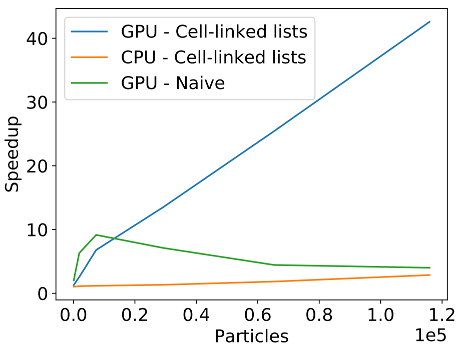
Figure B.4: The percentage of time the execution of time step takes out of the whole execution with cell-linked list on different architectures and ranks. Data from EPIC2



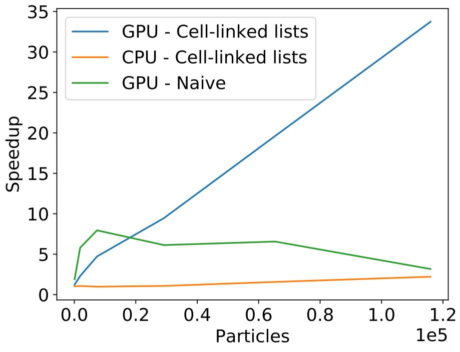
(a) Using 2 ranks



(b) Using 4 ranks

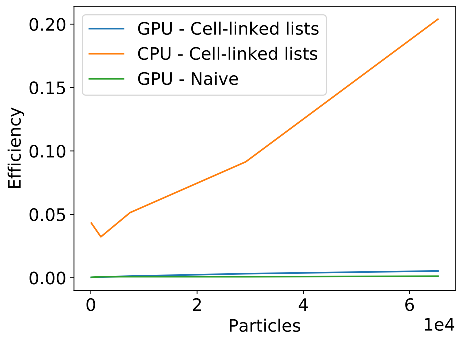


(c) Using 6 ranks

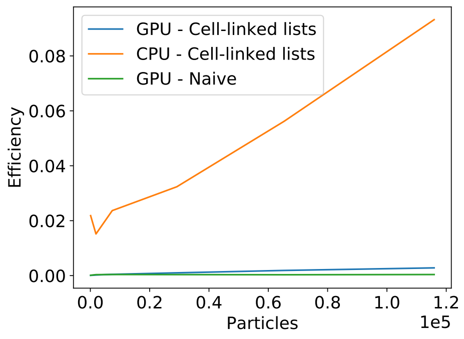


(d) Using 8 ranks

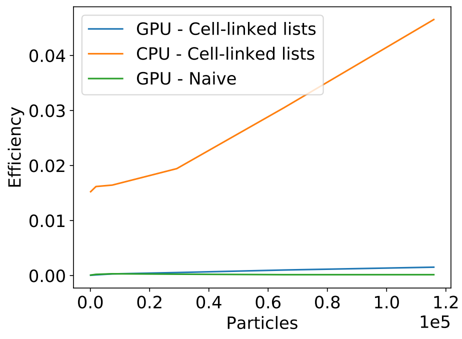
Figure B.5: Speedup from a parallel naive CPU baseline of the SPH dam-break problem on the EPIC2 queue. Notice that the range of the y axis change.



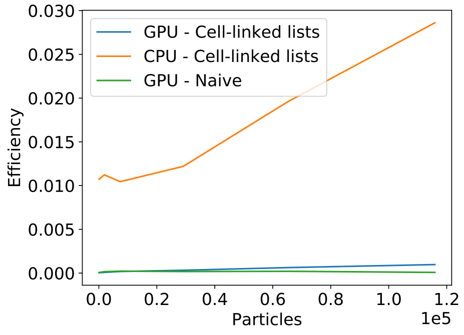
(a) Using 2 ranks



(b) Using 4 ranks

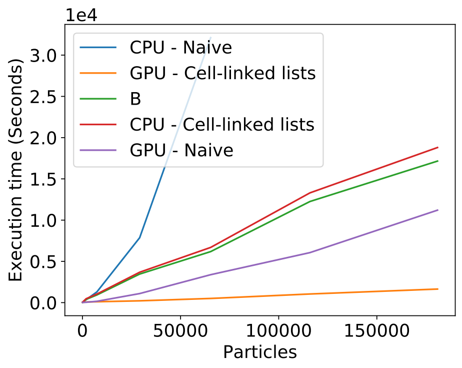


(c) Using 6 ranks

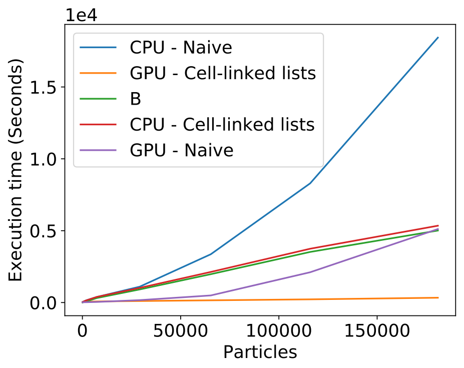


(d) Using 8 ranks

Figure B.6: Efficiency from a parallel naive CPU baseline of the SPH dam-break problem on the EPIC2 queue. Notice that the range of the y axis change.

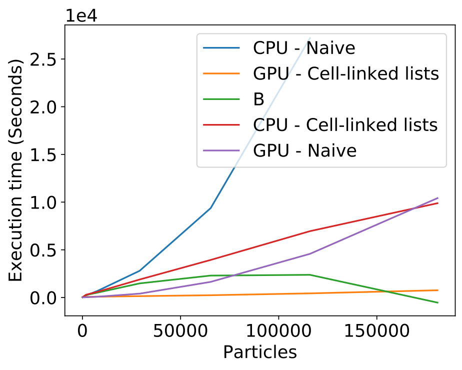


(a) Using 2 ranks.

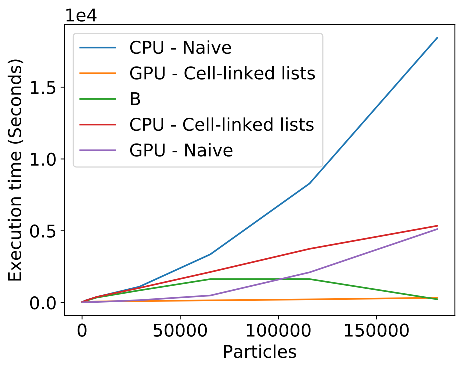


(b) Using 8 ranks.

Figure B.7: Comparisons of algorithm runtimes of CPU and GPU. B is Equation 6.2 with the cell-linked list versions of both systems. Data from EPIC2. Notice that the range of the y axis change.

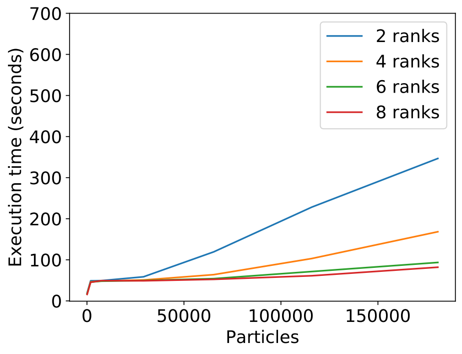


(a) Using 4 ranks.

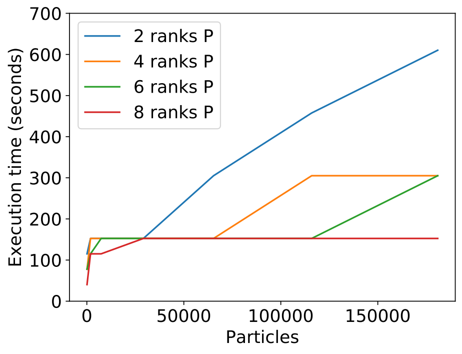


(b) Using 8 ranks.

Figure B.8: Comparisons of algorithm runtimes of CPU and GPU. B is Equation 6.2 with the naive GPU version and the cell-linked list CPU version. Data from EPIC2. Notice that the range of the y axis change.

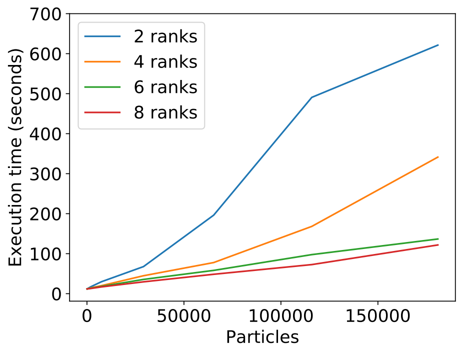


(a) Data from EPIC2.

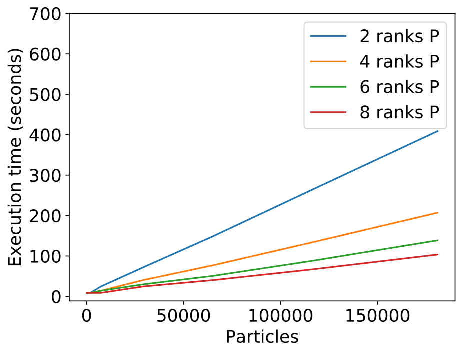


(b) Prediction for EPIC2.

Figure B.9: Execution time of finding neighbors of the cell-linked list GPU version.

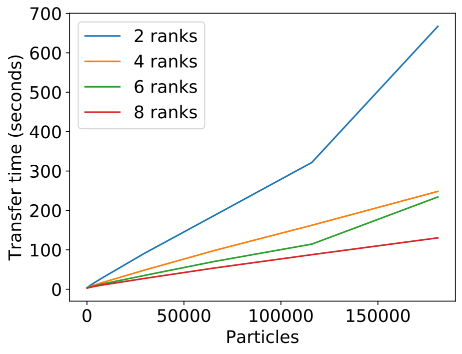


(a) Data from EPIC2.

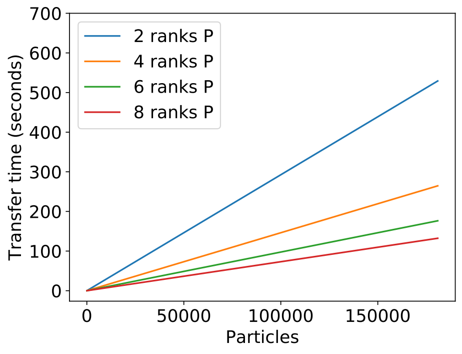


(b) Prediction for EPIC2.

Figure B.10: Execution time of physics computation of the GPU version.

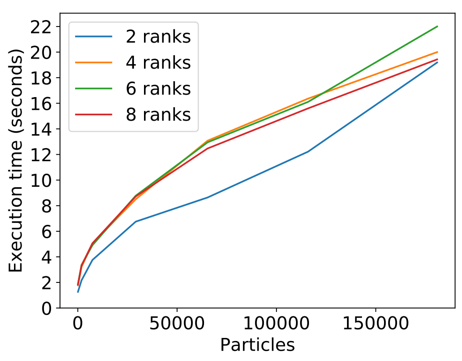


(a) Data from EPIC2.

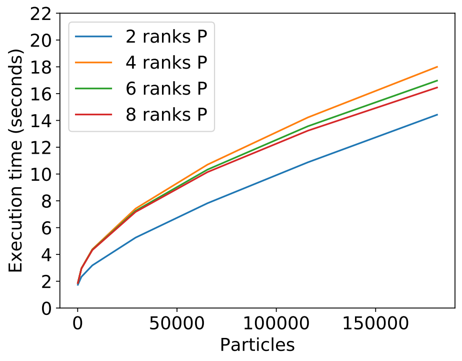


(b) Prediction for EPIC2.

Figure B.11: Time spent transferring particles from main memory to GPU and back.



(a) Data from EPIC2.



(b) Prediction for EPIC2.

Figure B.12: Execution time of communication.