Norwegian University of
Science and Technology

Performance modeling of CFD application scalability using co-design methods

Janusa Ragunathan
Jørgen Valstad

Master of Science in Computer Science
Submission date: July 2018
Supervisor: Jan Christian Meyer, IDI

Norwegian University of Science and Technology
Department of Computer Science

Performance Modeling of CFD Application Scalability Using Co-Design Methods

Jørgen Valstad Janusa Ragunathan

July 2018

Problem Description

This study will develop performance models of one or more proxy applications within the domain of computational fluid dynamics, based on experimental evaluations. Its objective is to evaluate the predictive power of the resulting models with respect to parallel scalability, and relate their accuracy to the choice of applied modeling techniques.

Assignment given:	February 1, 2018
Supervisor:	Jan Christian Meyer

Abstract

In this thesis, we investigate the performance and scalability of two CFD proxy applications, based on the Lattice Boltzmann Method (LBM) and Smoothed Particle Hydrodynamics (SPH). Two variants of LBM, workshare and task, are tested on two problems, Moffatt vortices and Cylinder flow. Three variants of the neighbor finding routine (the bottleneck of the SPH application) are studied on one problem, Dambreak. All variants are analyzed on three platforms, Vilje, EPT and EPIC.

We develop performance models, for all variants, that recommend how to achieve good scalability from the applications, enabling them to run on thousands of cores. This is partly due to the fact that the cost of communication is very low in both applications, and because most of the computational steps in both applications can be executed independently.

We validate our results by running the proxy applications on a machine outside of our testing platforms: ARCHER. There, our models successfully predict that running 1 rank per socket would be the better alternative for LBM, and that running 1 rank per node would be the better alternative for the best performing variant of SPH.

Acknowledgements

We would like to thank our supervisor Jan Christian Meyer for introducing us to the world of performance modeling, for guiding us through unknown terrain with long discussions, for saving us when we got locked out of our lab late at night, and for always motivating us when our courage was low.

The computations were partly performed on resources provided by UNINETT Sigma2 - the National Infrastructure for High Performance Computing and Data Storage in Norway. We have also been granted access to ARCHER, the UK national supercomputer, to do this work via the INTERTWinE FET-HPC project. We are very grateful to both parties for giving us this opportunity.

Finally, we would like to thank Tufan Arslan, for providing us with a great starting point in his SPH implementation.

Table of Contents

1 Introduction **1**

2 Motivation and Scope **3**
2.1 Application Models . . . 3
2.2 Programming Models . . . 3
2.3 Computer Architectures . . . 4
2.4 Performance and Communication Models . . . 4
2.5 Scope . . . 5

3 Background and Related Work **7**
3.1 Application Models . . . 7
3.1.1 Lattice-Boltzmann method . . . 7
3.1.1.1 Equilibrium Derivation . . . 9
3.1.1.2 Boundaries . . . 10
3.1.2 Smoothed Particle Hydrodynamics . . . 10
3.1.2.1 Boundary Conditions . . . 12
3.1.2.2 Time Integration . . . 12
3.1.2.3 Density Correction Algorithm . . . 13
3.2 Programming Models . . . 13
3.2.1 MPI . . . 13
3.2.2 OpenMP . . . 15
3.2.2.1 Mutual Exclusion . . . 16
3.2.3 OpenMP MPI Hybrids . . . 17
3.3 Computer Architectures . . . 17
3.4 Performance Models . . . 18
3.4.1 Communication Models . . . 18
3.4.1.1 Hockney . . . 18
3.4.1.2 BSP . . . 20
3.4.1.3 LogP . . . 20
3.4.2 Computation Models . . . 21
3.4.2.1 Roofline . . . 21

3.4.2.2 Dgemm . . . 23
3.4.2.3 STREAM . . . 23
3.4.3 Scalability . . . 23
3.4.3.1 Speedup and Parallel Efficiency . . . 23
3.4.3.2 Amdahl's Law . . . 24

4 Methodology **25**
4.1 LBM . . . 26
4.1.1 Moffatt Vortices and The Cylinder Problem . . . 26
4.1.2 Implementation . . . 27
4.1.2.1 Collision . . . 27
4.1.2.2 Border Exchange . . . 28
4.1.2.3 Propagate . . . 28
4.2 SPH . . . 30
4.2.1 Dam Break . . . 30
4.2.2 Implementation . . . 30
4.2.2.1 Finding Neighbors . . . 32
4.3 Computer Architectures . . . 40
4.3.1 Vilje . . . 40
4.3.2 Idun . . . 42

5 Results and Discussion **45**
5.1 Application Models . . . 45
5.1.1 LBM . . . 45
5.1.1.1 Intra-Node Parallelism . . . 45
5.1.1.2 Inter Node Communication . . . 51
5.1.2 SPH . . . 53
5.1.2.1 Intra-Node Parallelism . . . 53
5.1.2.2 Inter Node Parallelism . . . 80
5.2 Architectural Model . . . 86
5.3 Performance Models . . . 88
5.3.1 LBM . . . 88
5.3.2 SPH . . . 92
5.4 Parameter Tabels . . . 95

6 Validation **99**
6.1 Test procedure and results . . . 99
6.1.1 LBM . . . 99
6.1.2 SPH . . . 100
6.2 Evaluation . . . 100

7 Conclusion and Future Work **103**
7.1 Future work . . . 103

Appendices **109**

A Selected Source Code **111**

B LBM Speedup and Efficiency graphs **113**

C Latency and Inverse Bandwidth Heatmaps **123**

List of Figures

2.1 Time evolution of the Moffatt problem on LBM. 6
2.2 Time evolution of the Cylinder problem on LBM. 6
2.3 Time evolution of the Dambreak problem on SPH. 6

3.1 D2Q9 8
3.2 D2Q7 8
3.3 D2Q6 8
3.4 The interactions between particles that are closer, are stronger. 11
3.5 MPI Communicators. "my_communicator" is a communicator created with `MPI_Cart_create` and has a two-dimensional topology. 14
3.6 `#pragma omp parallel` spawns multiple threads to execute a stuctured block. Threads are joined after the structured block (also known as a synchronization point) is completed by all threads 15
3.7 Illustration of a shared-memory system 18
3.8 Illustration of a distributed memory system 19
3.9 Illustration of a hybrid memory system 19
3.10 A superstep in the BSP model. 21
3.11 Consecutive message transmissions with the LogP model. 22
3.12 Roofline Visual Model 22

4.1 General execution pattern for LBM and SPH 25
4.2 The lifecycle of the LBM application 26
4.3 LBM Border Exchange. The data elements are colored to illustrate which rank they originate from. The outermost regions represents the halos. Notice that the corners are successfully transferred diagonally. 29
4.4 The lifecycle of the SPH application 31
4.5 The scaling of the dam size. 31
4.6 The halo of a subdomain contains a copy of the nearby particles of the neighbouring subdomains. 32
4.7 The bucket datastructure. 34
4.8 Number of particles per bucket (shown as a square) on different timesteps. 37

4.9 The topology of Vilje . . . 41
4.10 The topology of IDUN EPT . . . 43
4.11 The topology of IDUN EPIC . . . 43

5.1 Saturation of sockets . . . 47
5.2 Single iteration time of workshare and task . . . 48
5.3 The workload of LBM's time integration phase split between Collision and Propagation . . . 49
5.4 EPIC. The time per iteration on multiple numbers of threads. . . . 49
5.5 EPT. The time per iteration on multiple numbers of threads. . . . 49
5.6 VILJE. The time per iteration on multiple numbers of threads. . . . 50
5.7 Moffatt VILJE Speedup and Efficiency on SCALE 20 with 2 ranks . . . 50
5.8 The communication pattern of the LBM application represented as an adjacency matrix, where red squares are neighbors. A square processor topology is assumed. . . . 51
5.9 Distribution in execution time of methods in time_step, for three different SPH implementations. The numbers are collected by running on 36 threads on EPIC. . . . 54
5.10 Distribution in execution time of different parts of the `find_neighbors` method (brute-force), SCALE=1. . . . 55
5.11 The relationship between number of pairs and the execution time of the neighbor loop in `find_neighbors` brute-force, 18 threads on EPIC, 200 000 timesteps. . . . 56
5.12 The relationship between number of iterations of the body of the neighbor loop and the execution time in `find_neighbors` brute-force, 18 threads on EPIC, 200 000 timesteps. . . . 57
5.13 The relationship between number of pairs and execution time of the neighbor loop in `find_neighbors` brute-force. Executed on EPIC, 20 000 timestep, SCALE=1. . . . 58
5.14 The relationship between number of iterations and execution time in `find_neighbors` brute-force. Executed on EPIC, 20 000 timesteps, SCALE=1. 59
5.15 Speedup and Efficiency of `find_neighbors` brute-force on EPIC, SCALE=1. 59
5.16 The relationship between the thread count and the execution time in `find_neighbors` brute-force on Vilje, SCALE=1 and 20 000 timesteps. . . 60
5.17 Comparison of execution time for various thread counts on Vilje and Epic for `find_neighbors` brute-force, SCALE=1 and 20 000 timesteps. . . . 60
5.18 Speedup and Efficiency of `find_neighbors` brute-force on Vilje, at SCALE=1. . . . 60
5.19 The relationship between number of threads and execution time in `find_neighbors` brute-force. Executed on EPT at SCALE=1, 20 000 timesteps. . . . 61
5.20 Comparison of execution time on various thread counts on EPT and Epic for `find_neighbors` brute-force, SCALE=1 and 20 2000 timesteps. . . 61
5.21 Speedup and Efficiency of `find_neighbors` brute-force on EPT, on SCALE=1. . . . 61

5.22 The execution time of find_neighbors brute-force as the iterations count increases at SCALE=1.2 and SCALE=2, EPIC. 62
5.23 Comparison of the execution time of brute-force find_neighbors at various SCALEs on EPIC, 20 2000 timesteps. 63
5.24 Speedup and Efficiency of brute-force find_neighbors on EPIC, at SCALE=1.5. 63
5.25 Speedup and Efficiency of brute-force find_neighbors on EPIC, at SCALE=2. 64
5.26 Distribution in execution time of different parts of the find_neighbors method (cell-linked particle list) 65
5.27 The relationship between number of pairs and the execution time of the create_pair method in the cell-linked particle implementation, 200 000 timesteps, EPIC. 66
5.28 The relationship between number of particles and the execution time of the create_pair method in the cell-linked particle implementation, 200 000 timesteps, EPIC. 66
5.29 The relationship between number of pairs and the execution time of the create_pair method in find_neighbors cell-linked particles. Executed on EPIC, 20 000 timestep, SCALE=1. 67
5.30 Speedup and Efficiency of find_neighbors cell-linked particle lists on EPIC, cell-linked particle list, SCALE=1. 68
5.31 The relationship between the thread count and the execution time in create_pairs on Vilje, SCALE=1 and 20 000 timesteps. 68
5.32 Comparison of execution time for various thread counts on Vilje and Epic for create_pairs, SCALE=1 and 20 000 timesteps. 68
5.33 Speedup and Efficiency of find_neighbors cell-linked particle lists on Vilje, at SCALE=1. 68
5.34 The relationship between the thread count and the execution time in create_pairs on EPT, SCALE=1 and 20 000 timesteps. 69
5.35 Comparison of execution time for various thread counts on EPT and Epic for create_pairs, SCALE=1 and 20 000 timesteps. 69
5.36 Speedup and Efficiency of find_neighbors cell-linked particle lists on EPT, at SCALE=1. 69
5.37 The execution time of find_neighbors cell-linked particle lists as the iterations count increases at SCALE=1.2 and SCALE=2, EPIC. 70
5.38 Speedup and Efficiency of find_neighbors cell-linked particle lists, at SCALE=1.5. 70
5.39 Speedup and Efficiency of find_neighbors cell-linked particle lists on EPIC, on SCALE=2. 70
5.40 Comparison of the execution time of create_pairs in find_neighbors cell-linked particle lists at various SCALEs on EPIC, 20 2000 timesteps. . 71
5.41 Distribution in execution time of different parts of the find_neighbors method (cell-linked pair and particle lists) 72

5.42 The relationship between number of particles and the execution time of the create_pair method in the cell-linked pair and particle list implementation, 200 000 timesteps, EPIC. 74
5.43 The relationship between number of created pairs and the execution time of the create_pair method in the cell-linked pair and particle list implementation, 200 000 timesteps, EPIC. 74
5.44 The relationship between number of actual pairs and the execution time of the create_pair method in the cell-linked pair and particle lists implementation, 200 000 timesteps, EPIC. 74
5.45 The relationship between number of pairs and the execution time of the create_pair method in find_neighbors cell-linked particle and pair lists. Executed on EPIC, 20 000 timestep, SCALE=1. 75
5.46 Speedup and Efficiency on EPIC, cell-linked particle and pair lists, SCALE=1. 76
5.47 The relationship between the thread count and the execution time in create_pairs in find_neighbors cell-linked particle and pair lists on Vilje, SCALE=1 and 20 000 timesteps. 76
5.48 Comparison of execution time for various thread counts on Vilje and Epic for create_pairs in find_neighbors cell-linked particle and pair lists, SCALE=1 and 20 000 timesteps. 76
5.49 Speedup and Efficiency of find_neighbors cell-linked pair and particle lists on Vilje, at SCALE=1. 77
5.50 The relationship between the thread count and the execution time in create_pairs on EPT, SCALE=1 and 20 000 timesteps. 77
5.51 Comparison of execution time for various thread counts on EPT and Epic for for create_pairs in find_neighbors cell-linked particle and pair lists, SCALE=1 and 20 000 timesteps. 77
5.52 Speedup and Efficiency for create_pairs in find_neighbors cell-linked particle and pair lists on EPT, on SCALE=1. 77
5.53 The execution time of find_neighbors cell-linked particle and pair lists as the iterations count increases at SCALE=1.2 and SCALE=2, EPIC. 78
5.54 Speedup and Efficiency of find_neighbors cell-linked particle and pair lists on EPIC, at SCALE=1.5. 78
5.55 Speedup and Efficiency of find_neighbors cell-linked particle and pair lists on EPIC, on SCALE=2. 79
5.56 Comparison of the execution time of create_pairs in find_neighbors cell-linked particle and pair lists at various SCALEs on EPIC, 20 2000 timesteps. 79
5.57 The communication pattern of the SPH program. A red square indicates exchanged between the corresponding ranks. 80
5.58 Maximum of all ranks on each timestep, averages of 1000 timesteps. From 0 to 200 000 timesteps, SCALE=1. 83
5.59 Maximum of all ranks on each timestep, averages of 1000 timesteps. From 0 to 200 000 timesteps, SCALE=1, SCALE=2 and SCALE=4. 83

5.60 Maximum of all ranks on each timestep, averages of 1000 timesteps. From 0 to 200 000 timesteps, SCALE=1. 84
5.61 Maximum of all ranks on each timestep, averages of 1000 timesteps. From 0 to 200 000 timesteps, SCALE=1, SCALE=2 and SCALE=4. 84
5.62 Vilje, Latency, 18 Nodes, 16 ranks per node. min:9.612437e-07 max:1.662683e-05 87
5.63 Vilje, Beta Inverse, 18 Nodes, 16 ranks per node. min:1.625543e-10 max:5.536014e-09 89
5.64 EPT, Latency, 10 Nodes, 20 ranks per node. min:5.296946e-07 max:7.599199e-06 89
5.65 EPT, Beta Inverse, 10 Nodes, 20 ranks per node. min:2.59738e-10 max:4.134643e-09 90
5.66 EPIC, Latency, 6 Nodes, 36 ranks per node, min:5.356431e-07 max:8.119893e-06 90
5.67 EPIC, Beta Inverse, 6 Nodes, 36 ranks per node, min:2.17188e-10 max:8.808944e-09 91

6.1 LBM Moffatt speedup with SCALE=40 on Archer. 100
6.2 SPH speedup with SCALE=1 on Archer. 101

B.1 Vilje Moffatt peedup and efficiency 114
B.2 EPT Moffatt speedup and efficiency 115
B.3 EPIC Moffatt speedup and efficiency 116
B.4 Vilje Cylinder speedup and efficiency 117
B.5 EPT Cylinder speedup and efficiency 118
B.6 EPIC Cylinder speedup and efficiency 119
B.7 Vilje Moffatt Task speedup and efficiency 120
B.8 EPT Moffatt Task speedup and efficiency 121
B.9 EPIC Moffatt Task speedup and efficiency 122

C.1 Vilje, Latency, 8 Nodes, 16 ranks per node. min:9.750016e-07 max:1.789225e-05 124
C.2 VILJE, Beta Inverse, 8 Nodes, 16 ranks per node. min:1.662708e-10 max:3.360967e-09 125
C.3 Vilje, Beta Inverse, 1 Node, 16 ranks per node. min:1.669952e-10 max:6.072257e-10 126
C.4 Vilje, Latency, 1 Node, 16 ranks per node. min:7.157778e-07 max:1.521244e-06 126
C.5 EPT, Latency, 1 Nodes, 20 ranks per node. min:5.405903e-07 max:1.430154e-06 127
C.6 EPT, Beta Inverse, 1 Nodes, 20 ranks per node. min:2.602089e-10 max:3.849905e-10 127
C.7 EPIC, Latency, 1 Nodes, 36 ranks per node. min:5.350947e-07 max:2.622306e-06 128
C.8 EPIC, Beta Inverse, 1 Nodes, 36 ranks per node. min:2.435713e-10 max:4.464687e-10 128

C.9 Archer, Latency, 16 Nodes, 24 ranks per node. min:1.286399e-06 max:8.888543e-06 129
C.10 Archer, Beta Inverse, 16 Nodes, 24 ranks per node. min:1.371288e-10 max:2.839602e-08 130

List of Tables

3.1 The four kernels of STREAM 23

4.1 Vilje Specification 40
4.2 IDUN Specification 42

5.1 Latency parameters for our test platforms 88
5.2 Inverse bandwidth parameters for our test platforms 88
5.4 SPH Parameter table for chapter 5. 97
5.3 LBM Parameter table for chapter 5. 98

Chapter 1

Introduction

Computational Fluid Dynamics is an important area of research in the industry, particularly in the maritime sector and the oil sector where simulation can reduce time and cost of developing robust equipment for offshore conditions. However, such applications are often very complicated and difficult to reason about. An alternative approach is to create *proxy applications*, which extract parts of the application that exhibit the essential performance characteristics of the real application. By *co-designing* such proxies together with CFD domain experts and applying them to known problems, we can develop proxies that are simple to reason about and analyze, and we can be confident that the data produced by the proxies adhere to the physical restrictions of the domain.

In this thesis we develop and analyze two CFD proxy applications, *LBM* and *SPH*. Two programming models are investigated for LBM, worksharing and tasking, to understand how load balancing affects performance. For SPH, we examine the impact of three neighbor finding algorithms. All applications and variants are tested empirically on three clusters, *Vilje*, *EPIC* and *EPT*. Vilje is a national supercomputer, while EPIC and EPT are local clusters at NTNU. With these machines, we create *performance models* based on software/hardware interactions, which purpose is to predict or approximate the run time characteristics and scalability of the proxy applications. A fourth supercomputer, *Archer* (Edinburgh, UK), is used to validate the performance models.

In Chapter 2, we describe the motivation and scope for this project. In Chapter 3, we present the fluid dynamics which our applications are based on, the programming models used, the system architecture of the clusters and relevant performance models. In Chapter 4, we describe our proxy applications and our testing platforms. In Chapter 5, we develop performance models and present and discuss our findings. In Chapter 6, we validate our performance models on Archer. In Chapter 7, we summarize our findings and our work.

Chapter 2

Motivation and Scope

In this chapter, we present our motivation for the different aspects of this project, as well as the scope of the thesis.

2.1 Application Models

Proxy applications are program samples that extract critical parts of larger systems. Isolating performance critical regions aids in reasoning and analysis of an application's behaviour, as HPC programs often adhere to the phrase "Only as strong as the weakest link".

In Cicotti *et al.* (2014), a proxy application for molecular dynamics is developed to explore the trade-offs of multithreading.

Banerjee *et al.* (2016) describes a proxy application CMT-bone based on a large multiphase flow simulator CMT-nek. The proxy's purpose is to mimic the computational behaviour of CMT-nek in order to analyze certain performance metrics.

Dickson *et al.* (2016) presents a proxy application for replicating HPC I/O workloads to address performance issues.

Karlin *et al.* (2012) creates multiple variations of a proxy application "LULESH" to investigate the benefits of different programming models.

Lawson *et al.* (2015) investigates how dynamic voltage and frequency scaling affects energy efficiency when code segments are offloaded to a co-processor.

This project features two such proxy applications named after the physical models they are based on: *LBM* and *SPH*. The applications are chosen to highlight how load balance in a dynamic system (SPH) can affect scalability differently than in a static system (LBM).

2.2 Programming Models

Programming models are techniques for developing on different types of hardware. Threads on multi core machines share memory, making programming models for shared mem-

ory parallelism necessary. Servers and supercomputers use multiple processors, each of which can have their own private memory depending on the architecture.

OMP and MPI are *de facto* standard models for shared memory programming and distributed memory programming, respectively. In this work we have used them both to create hybrid applications capable of running on supercomputers with multiple cores and processors. By using proven frameworks, this work becomes relevant when discussing highly scalable CFD applications.

2.3 Computer Architectures

Cluster systems with hybrid memory models are the most popular type of architectures, according to `top500.org` (as of June 2018). By utilizing distributed and shared memory systems, we can create applications that scale incredibly well.

In this thesis, what we refer to as "Computer Architecture" might better be understood as "System Architecture", and is used to describe the node configuration of the platforms we use, *e.g* Vilje, EPIC, EPT, and ARCHER, all of which are clusters.

2.4 Performance and Communication Models

Supercomputers are expensive and complex machines, but the investment is wasted if the target application is unable to exploit all of the systems resources. A program could be memory bound and spending most of its time waiting for IO, making the investment fall short.

Barker *et al.* (2009) present a methodology called *Performance Modeling*. This method is used to predict the performance of a system, even if the system is not physically available. In our project we have developed a set of prototypes in the interest of modelling their performance on large scale systems. Therefore, our use of performance modeling in this thesis falls under the "Implementation" life cycle stage.

2.5 Scope

The LBM proxy application originates from a collaboration between PRACE (Partnership for Advanced Computing in Europe) and the INTERTWinE project. The INTERTWinE project focuses on programming model interoperability, and the collaboration project focuses on tasking. In order to do research on different aspects of tasking, a realistic program implementing task constructs is needed. The design of the LBM program makes it vulnerable to load imbalance, as some computational units may be assigned to regions of the domain where no interactions take place. Hence, it may be a good candidate for task based programming.

The study of the SPH proxy application is motivated by the ongoing research of fuel tank design at marine technology, NTNU. The research involves understanding how horizontal sloshing affects the quality of fuel. This program executes for several hours even on small program sizes making it time consuming to do research.

Both applications are serial programs and therefore not scalable, and scalable programs in addition to corresponding models will be beneficial for the previously mentioned projects. The models will aid in understanding how to effectively scale the applications for larger systems.

With this in mind, we chose to rewrite the programs with MPI and OpenMP to run on shared and distributed memory hybrid systems, and develop models describing the most effective way to execute the programs. Our preliminary studies involved: developing task versions of LBM and SPH, comparing linked list and array versions of the SPH neighbor algorithm, studying different approaches to minimize the executing time of the SPH neighbor algorithm, comparing mutual exclusion mechanisms in OpenMP and a CUDA version of the LBM application. At the end, we chose to continue to develop the following versions.

For the LBM program, we chose to focus on implementing a task model in LBM, and analyze its effect on a load imbalanced problem. We explore two problem domains: the *Moffatt vortices problem* where a complex geometry results in an load imbalance, and the *Cylinder problem* where the domain is load balanced. The task implementation of the Cylinder problem is developed as a control point. The problems are visualized in Figure 2.1 and in Figure 2.2.

For SPH, we chose to investigate the neighbor finding algorithm; the bottleneck of the program. We study how the algorithm impacts the scalability of the application. One problem is explored: the *Dambreak problem*, where a fluid contained in a dam is simulated when the dam breaks. A visualization of the process is shown in Figure 2.3.

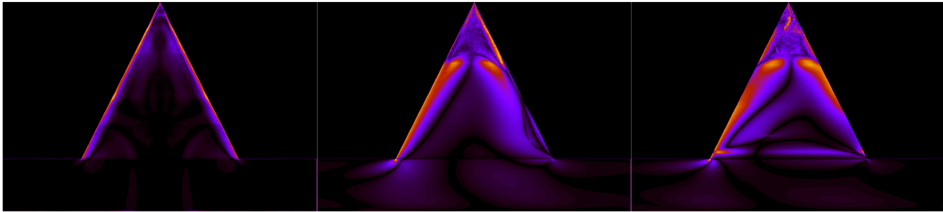


Figure 2.1: Time evolution of the Moffatt problem on LBM.

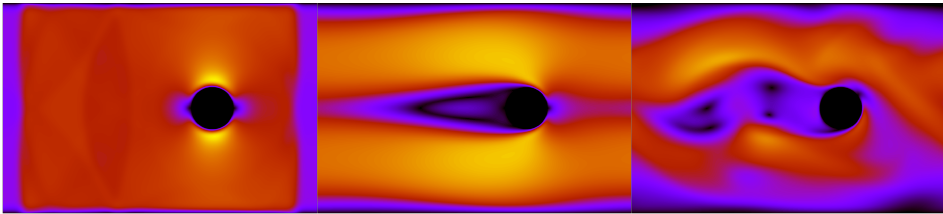
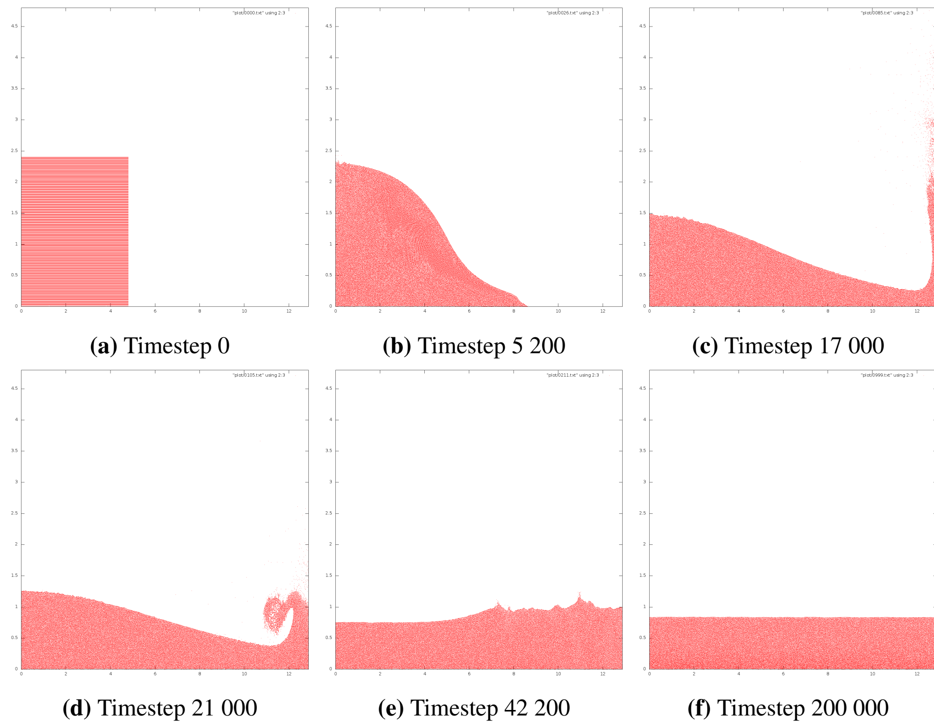


Figure 2.2: Time evolution of the Cylinder problem on LBM.



(a) Timestep 0
(b) Timestep 5 200
(c) Timestep 17 000
(d) Timestep 21 000
(e) Timestep 42 200
(f) Timestep 200 000

Figure 2.3: Time evolution of the Dambreak problem on SPH.

Chapter 3

Background and Related Work

In this chapter we introduce the theory which our work is based on.

In Section 3.1, we present the physics behind the two CFD applications, LBM and SPH. In Section 3.2, we introduce the two tools used for parallelism, MPI and OpenMP. In Section 3.3, the concepts of shared and distributed memory is described. In Section 3.4, a set of useful performance models are displayed.

3.1 Application Models

In this section we present the physics behind the two CFD applications, LBM and SPH.

3.1.1 Lattice-Boltzmann method

The *Lattice-Boltzmann method* (*LBM*) was evolved from the *lattice gas model* (due to Hardy *et al.* (1973)) after Frisch *et al.* (1986) introduced a Lattice Gas Automata with Navier-Stokes dynamics. In the lattice gas model the fluid is composed of particles which can reside at a lattice site. The main difference between the methods is that the Lattice Boltzmann model replaces the particles with particle densities. McNamara and Zanetti (1988) shows that this change mitigates the considerable fluctuations which occur in the lattice gas model.

LBM is used to simulate a system of incompressible fluid by tracking particle density distributions in a discrete, regular lattice. Particles move to neighboring lattice sites in every discrete time step according to their velocity and direction. This process is governed by kinetic rules that conserve mass, momentum and energy.

The choice of lattice structure dictates how particles are allowed to move through the domain. The naming scheme DmQn describes the lattice structure with m being the number of dimensions and n being the number of directions. An example for a two dimensional grid is *D2Q9*, where a particle can move to any of its eight neighbors or remain at rest, giving it nine possible densities. D2Q9 is shown in Figure 3.1, and is an example of a square lattice. *D2Q7* (Figure 3.2) is an example of a hexagonal lattice, where each particle

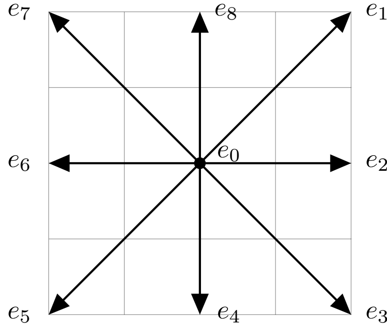


Figure 3.1: D2Q9

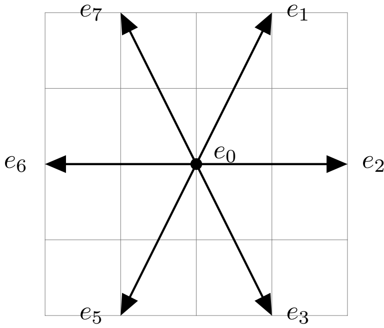


Figure 3.2: D2Q7

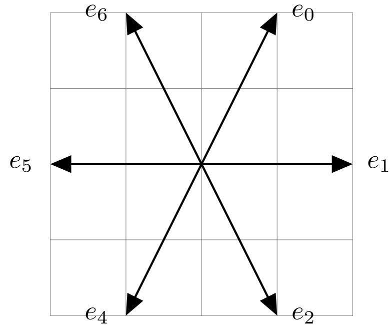


Figure 3.3: D2Q6

can move in 6 directions or remain at rest. If the rest particle is removed, this becomes a *D2Q6* lattice, as in Figure 3.3.

At each time step, the *streaming phase* and the *collision phase* occur. During the streaming phase, particle densities shift through the domain according to their velocities. During the collision phase the distribution function $f_i(x, \xi, t)$ is computed for every lattice point, where i is the lattice point index, x is the lattice site, ξ is the velocity and t is the time step. Density and momentum are macroscopic fluid variables (Chen and Doolen (1998)) and are defined in Equation 3.1 and 3.2, respectively.

$$\rho = m \int f_i(x, \xi, t) d\xi \tag{3.1}$$

$$\rho v = \int \xi f_i(x, \xi, t) d\xi \tag{3.2}$$

The discrete forms are similar (particle mass assumed to be unit mass $m = 1$):

$$\rho = \sum_{i=0}^{N} f_i \tag{3.3}$$

$$\rho v = \sum_{i=0}^{N} f_i e_i \tag{3.4}$$

$$v = \frac{1}{\rho} \sum_{i=0}^{N} f_i e_i \tag{3.5}$$

Here, N is the number of lattice velocities and e is the particle velocity. Equation 3.5 derives directly from Equation 3.4. For D2Q6, e_i is defined as:

$$e_i = \begin{cases} (0,0) & i = 0 \\ (\cos\theta_i \cdot c, \sin\theta_i \cdot c), \theta_i = (i-1)\frac{\pi}{3}, & i = 1, 2, ..., 5 \end{cases} \tag{3.6}$$

The lattice Boltzmann equation is defined, for any lattice scheme, as the time evolution of f_i:

$$f_i(\mathbf{x} + \mathbf{e_i}, t + \Delta t) - f_i(\mathbf{x}, t) = \Omega_i(f(\mathbf{x}, t)) \tag{3.7}$$

The left-hand-side of Equation 3.7 covers the streaming phase. Conversely, the right-hand-side covers the collision phase. The collision phase is complicated and difficult to compute exactly. Because the LBM method aims to simulate macroscopic dynamics, a relaxation towards some chosen equilibrium distribution is ideal. Therefore the collision operator Ω_i is approximated by a single-time-relaxation process:

$$\Omega_i(f(\mathbf{x}, t)) = \frac{f_i^{eq}(\mathbf{x}, t) - f_i(\mathbf{x}, t)}{\tau} \tag{3.8}$$

This relaxation is due to Bhatnagar *et al.* (1954). Here, the equilibrium distribution is denoted as f^{eq}. $1/\tau$ is the approach rate to equilibrium. Note that Equation 3.8 assumes no external force. If present, external force is applied to the collision operator. At each time step, energy and momentum is conserved within the system, and consequently,

$$\sum_i \Omega_i = 0$$

and

$$\sum_i \mathbf{e}_i \Omega_i = 0$$

must be true.

3.1.1.1 Equilibrium Derivation

Chen *et al.* (1994) define f_i^{eq} for D2Q7 as Equation 3.9 and f_0^{eq} as 3.10.

$$f_i^{eq} = \frac{\rho(1-\alpha)}{6} + \frac{\rho}{3}\mathbf{e}_i \cdot \mathbf{v} + \frac{2\rho}{3}(\mathbf{e}_i \cdot \mathbf{v})^2 - \frac{\rho}{6}\mathbf{v}^2 \tag{3.9}$$

$$f_0^{eq} = \alpha\rho - \rho\mathbf{v}^2 \tag{3.10}$$

Equation 3.9 is transformed to D2Q6 by removing the rest particle. This is achieved by defining the free parameter α as 0, as shown in Equation 3.11. The transformation of f_i^{eq} is shown in Equations 3.12-3.15.

$$\alpha := 0 \Rightarrow f_0^{eq} = -\rho \mathbf{v}^2 \tag{3.11}$$

$$\begin{aligned}
\sum_{i=0}^{6} f_i^{eq} &= \sum_{i=1}^{6} \left\{ \frac{\rho}{6} + \frac{\rho}{3}\mathbf{e}_i\mathbf{v} + \frac{2\rho}{3}(\mathbf{e}_i\mathbf{v})^2 - \frac{\rho}{6}\mathbf{v}^2 \right\} - \rho\mathbf{v}^2 && (3.12)\\
&= \sum_{i=1}^{6} \left\{ \frac{\rho}{6} + \frac{\rho}{3}\mathbf{e}_i\mathbf{v} + \frac{2\rho}{3}(\mathbf{e}_i\mathbf{v})^2 - \frac{\rho}{6}\mathbf{v}^2 \right\} - \sum_{i=1}^{6} \frac{\rho}{6}\mathbf{v}^2 && (3.13)\\
&= \sum_{i=1}^{6} \left\{ \frac{\rho}{6} + \frac{\rho}{3}\mathbf{e}_i\mathbf{v} + \frac{2\rho}{3}(\mathbf{e}_i\mathbf{v})^2 - \frac{\rho}{6}\mathbf{v}^2 - \frac{\rho}{6}\mathbf{v}^2 \right\} && (3.14)\\
&= \frac{\rho}{6} \sum_{i=1}^{6} \left\{ 1 + 2\underbrace{\mathbf{e}_i\mathbf{v}}_{(*)} + \underbrace{4(\mathbf{e}_i\mathbf{v})^2 - 2\mathbf{v}^2}_{(**)} \right\} && (3.15)
\end{aligned}$$

It is useful to express parts of Equation 3.15 in component form as this makes it easier to translate to C. See Equations 3.16-3.20.

$$(*) = \mathbf{ev} = e_x v_x + e_y v_y \tag{3.16}$$

$$\begin{aligned}
(**) &= 4(e_x v_x + e_y v_y)^2 - 2(v_x^2 + v_y^2) && (3.17)\\
&= 4(e_x^2 v_x^2 + 2e_x e_y v_x v_y + e_y^2 v_y^2) - 2(v_x^2 + v_y^2) && (3.18)\\
&= 4e_x^2 v_x^2 - 2v_x^2 + 8e_x e_y v_x v_y + 4e_y^2 v_y^2 - 2v_y^2 && (3.19)\\
&= 4\left((e_x^2 - \frac{1}{2})v_x^2 + 2e_x e_y v_x v_y + (e_y^2 - \frac{1}{2})v_y^2 \right) && (3.20)
\end{aligned}$$

3.1.1.2 Boundaries

Boundary conditions are rules for particle interactions with walls in the domain. A common way to handle this type of interactions is through a bounce back condition. In our approach, particles moving into solid cells are reflected back in the opposite direction.

3.1.2 Smoothed Particle Hydrodynamics

The Smoothed Particle Hydrodyamics (SPH) method was first introduced by Gingold and Monaghan (1977) and B. Lucy (1977), who focused on simulating compressible flow problems in astrophysics. Monaghan (1994) later included a Weakly Compressible SPH approach (WCSPH), enabling simulation of incompressible free surface flows. This method considered a fluid incompressible if the deviation in the fluid density was less than 1%. There are a number of different SPH variations proposed so far. This thesis is based the work done by Ozbulut *et al.* (2014), which applies the traditional, basic formulations of the WCSPH method to the dam-break problem.

The SPH method represents the fluid as a pool of interpolation points called *particles*, each having properties such as density, velocity and pressure. The particles are defined as

small parts of the medium, with finite volume. Each particle interacts with all neighbouring particles within a distance determined by the *smoothing length* (h). This can be seen in Figure 3.4. Particle properties are derived as a sum of contributions from each neighbouring particle, weighted by the analytical kernel/weighting function W(R,h). Ozbulut *et al.* (2014) uses the kernel function stated in Equation 3.21.

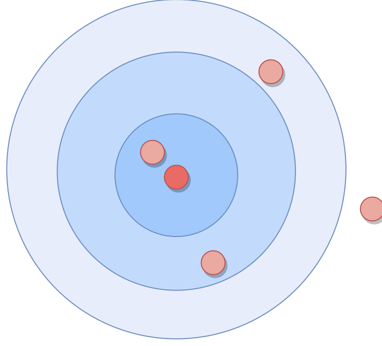


Figure 3.4: The interactions between particles that are closer, are stronger.

$$W(R,h) = \alpha_d \begin{cases} (3-R)^5 - 6(2-R)^5 + 15(1-R)^5, & 0 \leq R < 1 \\ (3-R)^5 - 6(2-R)^5, & 1 \leq R < 2 \\ (3-R)^5, & 2 \leq R < 3 \\ 0, & R \geq 3 \end{cases} \tag{3.21}$$

$$R = r_{ij}/h \tag{3.22}$$

In Equation 3.22, r_{ij} denotes the magnitude of the distance vector, $\mathbf{r_{ij}} = \mathbf{r_i} - \mathbf{r_j}$ between a pair of particles. Hence, the equation expresses R in terms of the distance vector r_{ij} between particle i and j, and the smoothing length h. In Equation 3.21, the coefficient α_d is determined by the dimensionality of the problem, $\alpha_d = 7/(478\pi h^2)$ for two dimensional problems.

The WCSPH method links the density to the pressure through Equation 3.23, which is an artificial equation of state to represent the speed of sound, Ozbulut *et al.* (2014); Monaghan and Kos (1999).

$$p = \frac{\rho_0 {c_0}^2}{\gamma} \left[\left(\frac{\rho}{\rho_0} \right)^\gamma - 1 \right], \tag{3.23}$$

In Equation 3.23, p and ρ are the pressure and density of particles, respectively, c_0 is the reference speed of sound, γ is the specific heat-ratio and ρ_0 is the reference density. γ=7 for water, and ρ_0=100$[kg/m^3]$ for fresh water. By Equation 3.23, a small change in density of the fluid particles will result in a large change in the pressure. To fulfill the incompressiblity condition, the value of c_0 must be large enough to keep the density

fluctuations small (1%). Moreover, c_0 has a direct effect on the allowed time-step though the Courant-Friedrichs-Lewy (CFL) condition, the larger the value of c_0, the smaller the value of the allowed time step. In order to minimize the overall computation cost, the value of c_0 should be as small as possible while still maintaining the incompressibility condition. In our dam-break simulation, c_0=50 [m/s] as suggested by Ozbulut *et al.* (2014).

If we disregard the artificial viscosity term, the SPH method in Ozbulut *et al.* (2014) discretizes the Euler's equation of motion and the mass conservation as in Equations 3.24 and 3.25.

$$\frac{d\mathbf{u_i}}{dt} = -\sum_{j=1}^{N}\left(\frac{p_i}{{\rho_i}^2} + \frac{p_j}{{\rho_j}^2}\right)\nabla_i W_{ij} \tag{3.24}$$

$$\frac{d\rho_i}{dt} = \rho_i \sum_{j=1}^{N} \frac{m_j}{\rho_j}(\mathbf{u_i} - \mathbf{u_j}) \cdot \nabla_i W_{ij} \tag{3.25}$$

In Equations 3.24 and 3.25, $\mathbf{u_i}$ and $\mathbf{u_j}$ are the velocities of particle **i** and **j**, respectively, and ∇_i is the gradient operator where **i** indicates the evaluation point of the spatial derivative at particle position i. m_j in Equation 3.25 is the mass of particle **j**.

3.1.2.1 Boundary Conditions

In order to achieve physically meaningful results, the solid wall boundary condition needs to be taken into account. In this thesis, we follow the model used in Ozbulut *et al.* (2014), which utilizes a mirroring technique with *ghost particles*. Here, ghost particles are created by mirroring the fluid particles lying within a vertical distance of 1.55h from the solid boundary, and thereby producing particles lying outside the fluid domain. The particles are mirrored about the solid wall. A boundary condition is implemented to determine the field variables (e.g. velocity and pressure) of the ghost particles. Ozbulut *et al.* (2014) uses the Neumann boundary condition. In the Neumann boundary condition, the field variables of the ghost particles are the same as for the corresponding fluid particles. Hence, $\Lambda_g = \Lambda_f$, where Λ_g and Λ_f are the field variables of the ghost and fluid particles, respectively, when considering the same fluid variable in both cases.

3.1.2.2 Time Integration

Ozbulut *et al.* (2014) represents the time integration as a predictor-corrector pattern where the position, density and velocity of a particle is updated as in Equations 3.26, 3.27 and 3.28, respectively.

$$\frac{d\mathbf{r_i}}{dt} = \mathbf{u_i} \tag{3.26}$$

$$\frac{d\rho_i}{dt} = k_i \tag{3.27}$$

$$\frac{d\mathbf{u_i}}{dt} = \mathbf{a_i} \tag{3.28}$$

In Equation 3.26, $\mathbf{r_i}$ is the position of particle **i**, in Equation 3.27, k_i represents Equation 3.25 and in Equation 3.28, $\mathbf{a_i}$ is the acceleration of particle **i**.

The time integration is performed in two steps, a prediction step and a correction step. During the prediction step, the positions and densities of the particles are updated by Equations 3.29 and 3.30, respectively.

$$\mathbf{r_i}^{n+1/2} = \mathbf{r_i}^{n} + 0.5\mathbf{u_i}^{n}\Delta t \tag{3.29}$$

$$\rho_i^{n+1/2} = \rho_i^{n} + 0.5k_i^{n}\Delta t \tag{3.30}$$

Utilizing these intermediate density values, the pressures of the particles are updated by Equation 3.23. Similarly, the velocities of the particles are calculated with the acceleration values computed in the prediction step.

In the correction step, the positions and densities of the particles are updated using the corresponding intermediate values by Equations 3.31 and 3.32.

$$\mathbf{r_i}^{n+1} = \mathbf{r_i}^{n+1/2} + 0.5\mathbf{u_i}^{n+1}\Delta t \tag{3.31}$$

$$\rho_i^{n+1} = \rho_i^{n+1/2} + 0.5k_i^{n+1}\Delta t \tag{3.32}$$

3.1.2.3 Density Correction Algorithm

In order to keep the pressure fields from oscillating at a high rate because of noise, Ozbulut *et al.* (2014) includes a density correction algorithm. This is presented in Equation 3.33.

$$\hat{\rho_i} = \rho_i - \sigma\frac{\sum_{j=1}^{N}(\rho_i - \rho_j)W_{ij}}{\sum_{j=1}^{N}W_{ij}} \tag{3.33}$$

In Equation 3.33, N denotes the number of particles that are neighbors of particle **i**, ρ denotes a constant and $\hat{\rho}$ denotes the corrected density. Following the work of Ozbulut *et al.* (2014), ρ=1.

3.2 Programming Models

In this section we describe two programming models: MPI and OpenMP.

3.2.1 MPI

Message-Passing Interface (referred to as *MPI*) is an API for developing parallel software targeting distributed memory architectures. A process is an instance of a process control block and is spawned by the operating system when an application is started. Each block is associated with a private address space to avoid being disrupted by other processes. *Message-passing* is used to enable communication between processes, as data can be transferred between private memory by sending and receiving messages.

When a program is launched by submitting

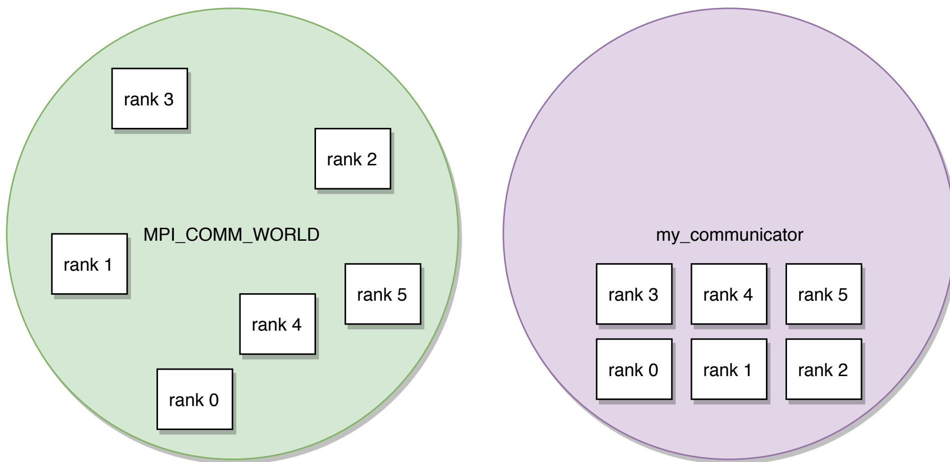


Figure 3.5: MPI Communicators. "my_communicator" is a communicator created with `MPI_Cart_create` and has a two-dimensional topology.

```
$ ./my_program
```

in the command line, a single process is spawned to manage the program. Using MPI, we can spawn multiple processes like this

```
$ mpirun ./my_program
```

The number of processes spawned is called *size* and each process is given a unique id, called *rank*, between 0 and size-1.

All communication in MPI takes place inside a *communicator*. All processes are part of the default communicator `MPI_COMM_WORLD`, which is initiated when the program calls `MPI_Init`. All transfer-functions must include a communicator to specify which "universe" the communication occurs in. It is possible to create application specific communicators with topology information attached. `MPI_Cart_create` can be used to initiate a two-dimensional array of processes. MPI will provide all ranks with coordinates which can be retrieved with the function `MPI_Cart_coords`. This is useful for applications where communication primarily takes place between neighbors. Two example communicators are shown in Figure 3.5.

The transfer operations of MPI fall into two categories, *point-to-point* operations and *collective* operations. Point-to-point operations involve two ranks, examples are `MPI_Send` and `MPI_Recv`. Collective operations involve all ranks in a communicator, examples are `MPI_Reduce` and `MPI_Broadcast`.

One useful point-to-point function is `MPI_Sendrecv`. Standard send and receive operations can be blocking (depending on the implementation) and must be scheduled to avoid the scenario where both ranks are sending and none are receiving. Sendrecv will take care of the scheduling so that deadlocks are avoided.

Latency is the constant delay that occurs each time a message is sent and is caused by the overhead getting a message ready before it is submitted. Therefore it is cheaper to send

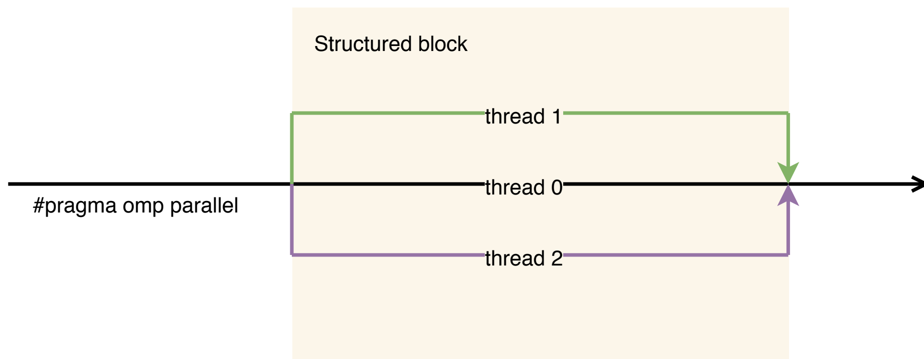


Figure 3.6: `#pragma omp parallel` spawns multiple threads to execute a stuctured block. Threads are joined after the structured block (also known as a synchronization point) is completed by all threads

all the data in one large message than to send many smaller messages. MPI provides tools for combining multiple elements into singular MPI datatypes, known as *derived datatypes*. In Listing 3.1 we see an example of a derived datatype. `MPI_Type_vector` is used to define a datatype representing columns in a two dimensional array.

```
int data[10][20];
MPI_Datatype column_type;
MPI_Type_vector(10, 1, 20, MPI_INT, &column_type);
MPI_Type_commit(&column_type);
```

Listing 3.1: MPI Derived Types. The data array has a column size of 10. The column type can be used in all types of communication.

3.2.2 OpenMP

OpenMP is an API for developing parallel software with a shared memory architecture, for example within the nodes of a cluster. A higher throughput can be achieved by utilizing all available cores in a multi-core architecture and OMP provides us with useful abstractions for running threads on such cores. In this section we describe the OpenMP for loop, which is often referred to as a *worksharing* construct, and the OpenMP task, which is often referred to as a *tasking* construct.

OMP enables parallelism through compiler directives known as pragmas, which work on structured blocks of code. For example, the function `foo()` can be executed by multiple threads in the following way:

```
#pragma omp parallel
foo();
```

The event is pictured in Figure 3.6. In this example the number of threads is determined by the environment variable `OMP_NUM_THREADS`. If the variable is unset, the number of threads is defined by the OpenMP implementation.

Parallelizing a loop is done by adding a similar directive:

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {...}
```

The work of the loop is evenly divided among the threads, so each thread will perform N/T iterations, where T is the number of threads. This method is referred to as worksharing.

Usage of `parallel for` is limited to for loops which adhere to canonical form[1]. OpenMP must be able to determine the number of iterations from the `for` statement, and therefore no `break` can occur in the body of the loop. This makes work-sharing constructs unsuitable for applications with irregular memory access patterns, such as traversing and processing nodes of a linked list.

The task construct covers the need for parallel execution of regions with unknown size. The following directive declares `foo()` as a task ready to be executed by a thread.

```
#pragma omp task
foo();
```

A common pattern is to use tasks in conjunction with the `single` directive, as shown in Listing 3.2. One thread creates tasks while the others execute tasks as they are made available. The task generating thread will join the others in completing tasks after all tasks have been generated. The threads that are not generating tasks will be waiting at the implicit barrier on line 10 before they start executing tasks.

```
1  #pragma omp parallel
2  #pragma omp single nowait
3  {
4      node* p = linked_list_head;
5      while (p) {
6          #pragma omp task
7          foo(p);
8          p=p->next;
9      }
10 } // Implicit barrier at the end of parallel region
```

Listing 3.2: Linked list iteration with OpenMP tasks.

Tasks can also be used to execute loops in parallel by prepending the *taskloop* construct to the loop, which is the task counterpart of `omp parallel for`. A single thread will create tasks of the loop iterations, and the others will execute them.

```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop
for (int i = 0; i < N; i++) {...}
```

3.2.2.1 Mutual Exclusion

Race conditions occur when multiple threads are working on the same memory address and at least one thread is writing. Access to such regions of memory must be serialized to avoid non-deterministic bugs. OpenMP provides three mechanisms for ensuring mutual exclusion: *critical*, *atomic* and *locks*.

[1] OpenMP Architecture Review Board (2015), p. 53

`#pragma omp critical` ensures that a structured block of code can only be entered by at most one thread simultaneously.

`#pragma omp atomic` makes use of hardware instructions for updating memory in a single instruction. This can only be used on load/store expressions, but is potentially faster than critical sections.

Locks are invoked with OpenMP library functions and are more general that the previously mentioned alternatives. They allow for fine grained control over memory access and are useful when locking parts of an array. In Listing 3.3, the array `secret` can be accessed simultaneously because each index has its own lock, so no thread will update the same memory address.

```
int secret[10] = {0};
omp_lock_t lock[10];
for (int i = 0; i < 10; i++) { omp_init_lock(&(lock[i])); }
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    if (isPrime(i)) {
        digit = i%10
        omp_set_lock(&(lock[digit]));
            secret[digit]++;
        omp_unset_lock(&(lock[digit]));
    }
}
```

Listing 3.3: OpenMP Locks makes it possible to update elements of the array without placing the entire memory region in a critical section.

3.2.3 OpenMP MPI Hybrids

Using MPI and OpenMP in conjuction enables us to exploit clusters where distributed nodes have private memory and the cores within the nodes share memory. In Rabenseifner (2003), the author describes different strategies for hybrid programming, with pure MPI and pure OpenMP at each their ends for the spectrum. In between we find *Hybrid masteronly*, where all calls to MPI are made outside parallel regions. The article highlight two drawbacks with Hybrid masteronly: The first is being that most threads are idle during the communication phase, except for the master thread. The second is that one thread will struggle to saturate the interconnect when sending messages. However, the master only approach is easier to program compared to overlapping communication with many threads, which, according to the author, “needs extreme programming effort”.

3.3 Computer Architectures

In this section, we describe some useful classifications of computer architectures.

The *von Neumann architecture* was one of the earliest computer architectures ever proposed. It consists of a memory unit, CPU and an interconnection connecting the two. The main memory contains both data and instructions which must be fetched by the CPU.

In terms of Flynn’s taxonomy (introduced in Flynn (1966)), a von Neumann machine fits into the category *SISD*, *single instruction single data* because only a single instruction can be executed at a time, and only a single data element can be fetched or stored at a time.

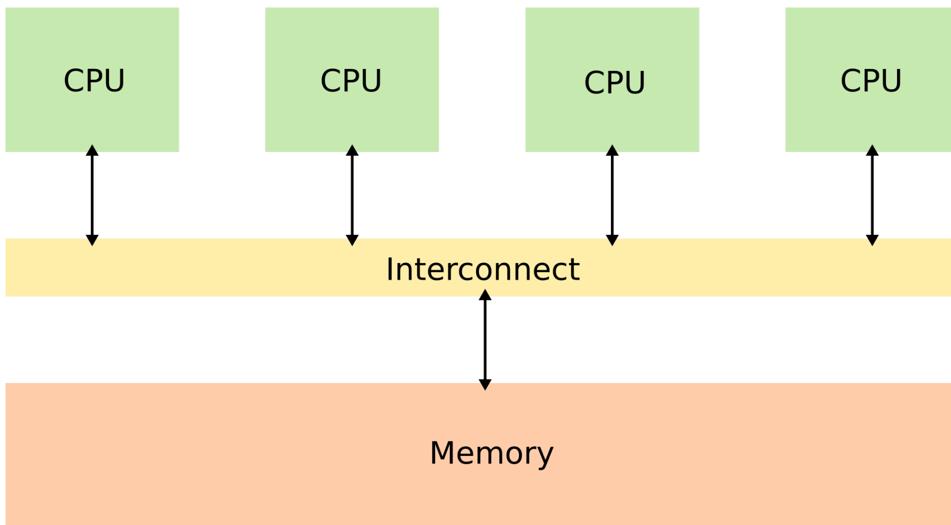


Figure 3.7: Illustration of a shared-memory system

Most modern architectures are *MIMD* systems, *multiple instruction multiple data*. Off-the-shelf desktop systems have multi-core processors which execute hundreds of processes seemingly simultaneously through multitasking, all potentially working on different data streams. The processors in a MIMD system are self contained and can execute instructions independently of each other.

MIMD systems use either *shared memory*, *distributed memory* or a hybrid of both. In shared memory systems the compute units can access all memory, while compute units in distributed memory systems only has access to private memory. In hybrid systems there are individual *nodes* of shared memory, connected by a network forming a distributed systems. Illustrations of shared, distributed and hybrid memory systems are shown in Figure 3.7, Figure 3.8 and Figure 3.9, respectively.

SPMD, *single program multiple data*, is a subcategory of MIMD which is relevant for MPI programs. As mentioned in Section 3.2.1, MPI spawns multiple processes from one program binary. Parallelism is achieved by branching on the `rank` and `size` attributes, and further by using threads to share work or tasks.

3.4 Performance Models

In this section we describe some useful communication and performance models.

3.4.1 Communication Models

3.4.1.1 Hockney

A kernel running on a distributed memory architecture will require some form of communication. The Hockney model, described in Lastovetsky *et al.* (2010), is used to approx-

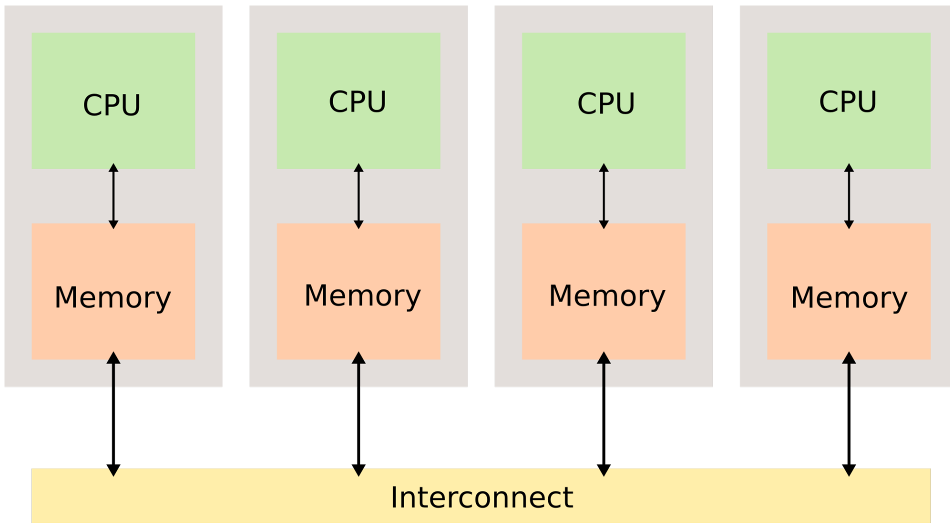


Figure 3.8: Illustration of a distributed memory system

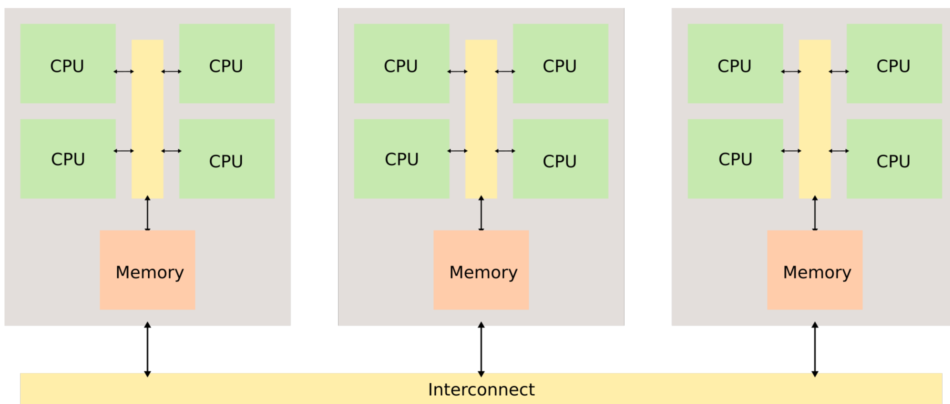


Figure 3.9: Illustration of a hybrid memory system

imate the time needed for message transmissions. The communication time for a single message can be decomposed into its start-up time or latency α and its transfer time or bandwidth β, see Equation 3.34. The latency is the time required for packing and unpacking the data (Wilkinson (2005)), as well as the overhead in establishing a communication channel. This value will stay close to constant during the lifetime of a kernel, and can be approximated by sending messages with no data. The bandwidth is the time required for sending a single data element (usually a byte) multiplied by the number of data elements *M*. The total communication time is the sum of the individual communication times, as seen in Equation 3.35.

$$t_{\text{comm},i} = \alpha + M_{\text{i}}\beta \tag{3.34}$$

$$t_{\text{comm}} = \sum_{i=1}^{N} t_{\text{comm},i} \tag{3.35}$$

The Heterogeneous Hockney model defined in Lastovetsky *et al.* (2010) distinguishes processors and links that are distinct, by introducing different parameters α_{ij} and β_{ij} for different processor pairs (i,j). Hence, the parameters α and β can be organized as $p \times p$ matrices, where p is the number of processors.

3.4.1.2 BSP

The *Bulk-Synchronous Parallel model* (BSP) is a bridging model between software and hardware for parallel computation proposed by Valiant (1990). It aims to be a universal standard, and consists of three attributes: a set of processor or memory components, a router delivering point-to-point messages, and a synchronization functionality for periodic synchronization of one or a subset of the components. A computation consists of a sequence of *supersteps*. During a superstep, each component performs local computation and communicates with the other components through sending and receiving messages as shown in Figure 3.10. Each component can only send or receive a maximum of h messages within a superstep. This communication pattern is called a *h-relation*. The barrier synchronization concludes the superstep. In Valiant (1990), a global check is performed after L time units to determine if a superstep is completed. In the case of completion, the machine proceeds to the next superstep. Otherwise, the machine allocates L additional time units to the unfinished superstep.

3.4.1.3 LogP

The *LogP model* is a communication model developed by Culler (1993) in which the processors communicate though point-to-point messages. The model characterizes a parallel machine through four main parameters; the processor/memory module count (P), an upper bound on the communication latency during a transmission of a small word (or a few small words) (L), the communication overhead during transmission or reception of a message (o), and the gap defined as the time duration between successive message transmissions or successive message receptions (g). Moreover, the model assumes that the network has a finite capacity of $\lceil \frac{L}{g} \rceil$.

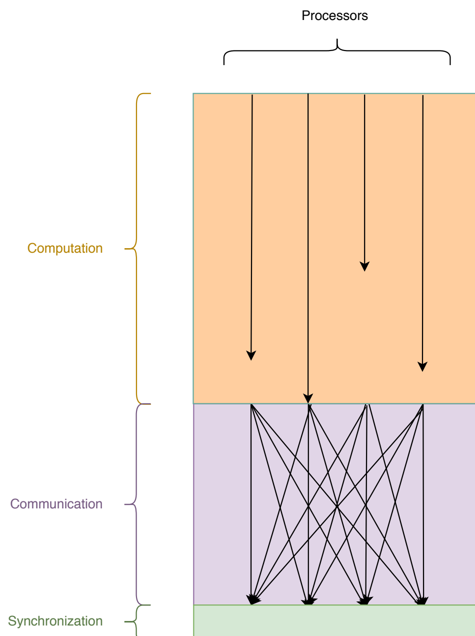


Figure 3.10: A superstep in the BSP model.

The time duration of a point-to-point message transmission can be estimated as L + 2o. Furthermore, Lastovetsky *et al.* (2010) presents the transmission of a large message as consecutive transmissions of small messages as described in Equation 3.36. The term M in Equation 3.36 is the number of small messages. Note that Equation 3.36 assumes that $g >= o$. A visualization of Equation 3.36 is shown in Figure 3.11.

$$l + 2o + (M - 1)g \tag{3.36}$$

There are several extensions to the LogP model. Alexandrov *et al.* (1997) introduces the LogGP model where messages can be of an arbitrary size through the inclusion of a gap per byte parameter. Kielmann *et al.* (2000) describes a parameterized LogP model (PlogP) where some of the parameters are piecewise linear functions of the message size and the meaning of the parameters differ slightly from those in LogP.

3.4.2 Computation Models

3.4.2.1 Roofline

Kernel performance is constrained by the hardware that executes it. Depending on the memory access pattern, a kernel is either compute-bound or memory-bound. Kernels that rarely load/store data and spend most of their time operating on data are compute-bound. Inversely, kernels that spend most of their time waiting for memory will stall the CPU and are therefore memory-bound. The Roofline model, introduced in Williams *et al.* (2009),

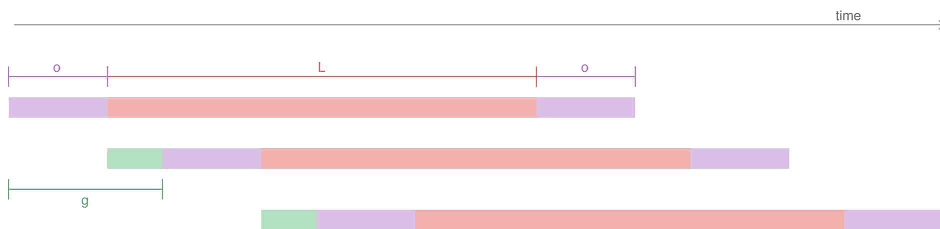


Figure 3.11: Consecutive message transmissions with the LogP model.

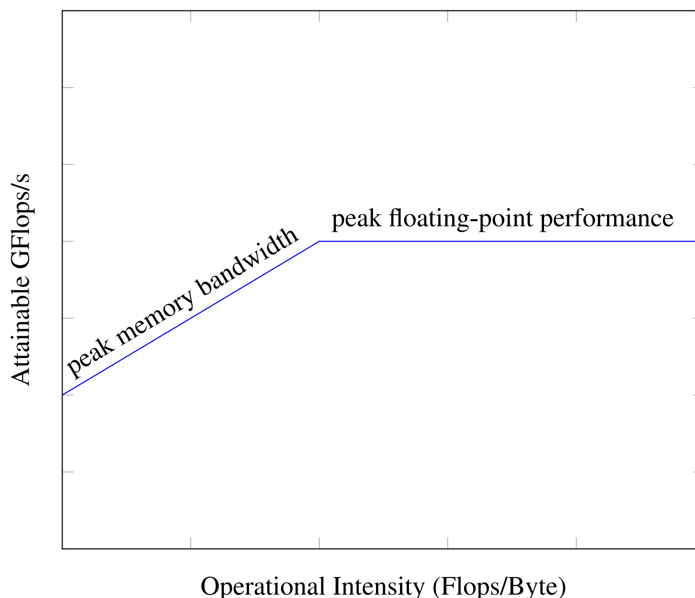


Figure 3.12: Roofline Visual Model

models the performance of kernels, highlighting whether the kernel is memory or compute bound. The Roofline model consists of a flat ceiling indicating peak floating-point performance, and a skewed ceiling indicating peak memory bandwidth. An example can be seen in Figure 3.12. The graph is based on Equation 3.37. The term operational intensity is defined as operations per byte of DRAM traffic. That is, the number of flops that we are able to execute per byte that is loaded from memory.

$$\begin{matrix}\text{Attainable}\\\text{GFlops/sec}\end{matrix} = \min\left(\begin{matrix}\text{Peak Floating-Point}\\\text{Performance}\end{matrix}, \begin{matrix}\text{Peak Memory}\\\text{Bandwidth}\end{matrix} \times \begin{matrix}\text{Operational}\\\text{Intensity}\end{matrix}\right) \tag{3.37}$$

COPY:	$a(i) = b(i)$
SCALE:	$a(i) = q \cdot b(i)$
SUM:	$a(i) = b(i) + c(i)$
TRIAD:	$a(i) = b(i) + q \cdot c(i)$

Table 3.1: The four kernels of STREAM

3.4.2.2 Dgemm

dgemm is part of the level 3 Basic Linear Algebra Subprograms (BLAS), as described by Dongarra *et al.* (1990). It performs a “double general matrix matrix multiply”, to solve Equation 3.38. In Equation 3.38, $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are matrices, while α and β are scalars.

$$\mathbf{C} = \alpha\mathbf{A}\mathbf{B} + \beta\mathbf{C} \tag{3.38}$$

dgemm is useful for establishing peak floating point performance on a system as it is commonly used in scientific computing.

3.4.2.3 STREAM

STREAM is a benchmark proposed by McCalpin (1995) that is used to measure memory bandwidth. It does so by computing four *vector operations*, as shown in Table 3.1.

The data rate for an operation can be retrieved by counting the number of bytes loaded per operation and dividing it by the execution time of the operation.

3.4.3 Scalability

In this section, we present some concepts that are useful when discussing parallel scalability.

3.4.3.1 Speedup and Parallel Efficiency

The terms speedup and efficiency are useful when discussing the scalability of a program. Pacheco (2011) defines the *speedup S* as the execution time of a serial program divided by the execution time of the program in parallel as presented in Equation 3.39.

$$S = \frac{T_{serial}}{T_{parallel}} \tag{3.39}$$

If a parallel program is run on p cores, the optimal value of speedup is p, meaning that a program being executed on p cores will be p times faster than a serial program. This is called *linear speedup*, and happens when all work is divided equally among the cores, without any added work being introduced as a result of the cores running together. In practice, however, linear speedup is unlikely. The overhead of running multiple cores in parallel is significant. Furthermore, this overhead often increases as the number of cores being exploited increases, resulting in a decrease in speedup per core. This notion of speedup per core is called *efficiency*. Given Equation 3.39, the efficiency *E* of a program can be presented as in Equation 3.40.

$$E = \frac{S}{p} = \frac{T_{serial}}{p \cdot T_{parallel}} \tag{3.40}$$

3.4.3.2 Amdahl's Law

In Amdahl (1967), Gene Amdahl presented an observation that was later known as Amdahl's law. Amdahl's law is often used to determine the theoretical speedup when using several processors. It states that, unless the entire program can be perfectly parallelized, the speedup will be limited regardless of the number of processors being used.

The overall parallel execution time, $T_{parallel}$, can be formulated as shown in Equation 3.41.

$$T_{parallel} = f \cdot \frac{T_{serial}}{s} + (1 - f) \cdot T_{serial} \tag{3.41}$$

In Equation 3.41, f is the fraction of the serial execution time that can be parallelized, s is this part's speedup, and T_{serial} is the execution time of the inherently serial part of the program. If the parallel part can be perfectly parallelized, s can be replaced by p, the number of cores being used. This is shown in Equation 3.42.

$$T_{parallel} = f \cdot \frac{T_{serial}}{p} + (1 - f) \cdot T_{serial} \tag{3.42}$$

Given, Equations 3.41 and 3.39, the theoretical speedup can be presented as in Equation 3.43.

$$S = \frac{1}{\frac{f}{s} + (1 - f)} \tag{3.43}$$

Equation 3.43 shows that the speedup will always be limited by the non-parallelizable part. Hence, even if the parallel part can be perfectly parallelized as in Equation 3.42, the speedup will always be $S <= \frac{1}{1-f}$.

In Gustafson (1988) (Gustafson's law), the author reevaluates Amdahl's law to take problem size into the consideration. The serial fraction of a program tends to decrease as the problem size increases.

Chapter 4

Methodology

This chapter describes our implementation of two proxy applications. Both applications use a similar structure to the one depicted in Figure 4.1. The initialization step consists of allocating memory and distributing the domain across ranks. The latter is necessary because the programs are parallelized using MPI, and each rank will only perform computation on its own subdomain. The time integration step is the main loop of the program, where time is discretized into time steps. Each time step consists of computation and communication, with IO occuring periodically. The computation is parallelized with OpenMP. During the finalize step, memory is deallocated.

Both applications are OpenMP + MPI hybrids, using the masteronly strategy described in Section 3.2.3. Because each time step consists of computation and communication, the applications follow the BSP model, as described in Section 3.4.1.2

The rest of the chapter is structured as follows: In Section 4.1 we show our LBM application which simulates fluid flowing through a channel with a wedge. In Section 4.2 we present our SPH application which simulates a fluid released from a rectangular starting position. In Section 4.3, we discuss the platforms Vilje, EPIC and EPT.

Note that the iterations of the outer loop of SPH are referred to as time steps, and that the iterations of the outer loop of LBM are referred to as iters/iterations/time steps.

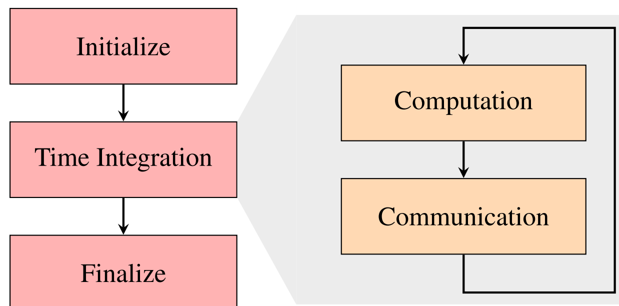


Figure 4.1: General execution pattern for LBM and SPH

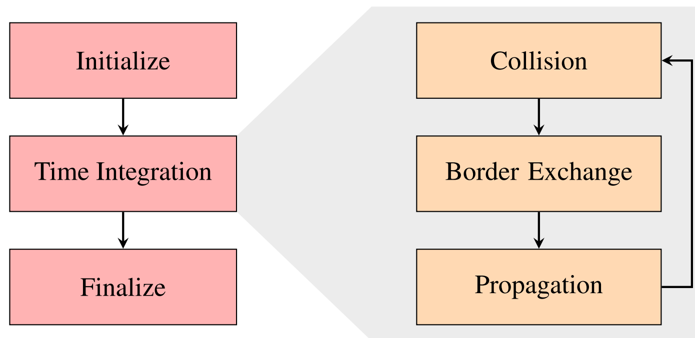


Figure 4.2: The lifecycle of the LBM application

4.1 LBM

This section describes the test problems, as well as our implementation of the time integration routine in the LBM application. Its structure can be seen in Figure 4.2. The program is based on the D2Q6 model: The problem domain is a two dimensional grid where each element is a lattice point. A lattice point is a structure containing six densities and two velocities. The grid is distributed over a two dimensional set of processing units (ranks) using MPI.

The geometry of the domain is specified with a boolean ghost map, where a value of '1' indicates a solid point, and '0' indicates a liquid lattice point. No computation takes place on a solid point, and the ghost map makes it simple to evaluate a lattice site.

The size of the domain is decided by a parameter *SCALE*. The width and height of the domain is decided by Equations 4.1 and 4.2, respectively.

$$\text{WIDTH} = 300 \cdot \text{SCALE} \tag{4.1}$$

$$\text{HEIGHT} = 300 \cdot \text{SCALE} \tag{4.2}$$

Both the collision and the propagation step are parallelized at the thread level with OMP. As the bulk of the work is done in loops, they are particularly easy to modify with the `omp parallel for` pragma or the `omp taskloop` pragma.

4.1.1 Moffatt Vortices and The Cylinder Problem

LBM is developed and tested with two geometries; the Moffatt problem and the Cylinder problem.

When water flows in a straight channel with a wedge where the angle of the wedge is less than 146°, Moffatt vortices will begin to form as described by Moffatt (1964). This is depicted in Figure 2.1.

The Cylinder problem describes flow through a straight channel with a cylinder placed in the middle. This is depicted in Figure 2.2.

4.1.2 Implementation

4.1.2.1 Collision

The collision method iterates over every lattice point in the subdomain and performs local computations on particle velocities and densities for the next time step. A lattice only interacts with it's neighbors, making the program similar to a classic *stencil* application (see dwarf number five in Asanović *et al.* (2006) for more information about stencil applications).

Lattice point velocity v Velocity is determined by applying Equation 3.5 at every lattice point, as shown in Listing 4.1. The density ρ and the sum of $f_i e_i$ is computed at lines 4-9, and the velocity v is computed at lines 11-12.

```
1  /* Compute velocity unless lattice site is a ghost */
2  if ( ! ghost[GY(y)][GX(x)] )
3  {
4      for ( int i=0; i<6; i++ )
5      {
6          rho += lattice[LIB(x, y)].density[i][NOW];
7          lattice[LIB(x, y)].velocity[0] += c[i][0] * lattice[LIB(x, y)].density[i][↩
               NOW];
8          lattice[LIB(x, y)].velocity[1] += c[i][1] * lattice[LIB(x, y)].density[i][↩
               NOW];
9      }
10     /* rho*u = sum_i( Ni*ci ), so divide by rho to find u: */
11     lattice[LIB(x, y)].velocity[0] /= rho;
12     lattice[LIB(x, y)].velocity[1] /= rho;
13 }
```

Listing 4.1: The velocity is computed at non-ghost points. Note that there are six particle densities, hence the inner loop.

Equilibrium distribution f_i^{eq} The equilibrium function introduced in Equation 3.15 has many components and is therefore split into three parts, see Listing 4.2 The variable `qi_uaub` contains the result of Equation 3.20 (divided by four), while `uc` contains the result of Equation 3.16. f_i^{eq} is stored in the variable `N_eq`.

```
1  for ( int i=0; i<6; i++ )
2  {
3      float qi_uaub, N_eq, delta_N;
4      qi_uaub =
5          ( c[i][1] * c[i][1] - 0.25 ) * lattice[LIB(x, y)].velocity[1] * lattice[LIB↩
               (x, y)].velocity[1] +
6          ( c[i][1] * c[i][0]        ) * lattice[LIB(x, y)].velocity[1] * lattice[LIB(↩
              x, y)].velocity[0] +
7          ( c[i][0] * c[i][1]        ) * lattice[LIB(x, y)].velocity[0] * lattice[LIB(↩
              x, y)].velocity[1] +
8          ( c[i][0] * c[i][0] - 0.25 ) * lattice[LIB(x, y)].velocity[0] * lattice[LIB↩
               (x, y)].velocity[0];
9      uc = lattice[LIB(x, y)].velocity[0] * c[i][0] + lattice[LIB(x, y)].velocity[1] ↩
           * c[i][1];
10
11     // Equilibrium
12     N_eq = ( rho / 6.0 ) * ( 1.0 + 2.0 * uc + 4.0 * qi_uaub );
```

Listing 4.2: The Equilibrium Distribution

Collision Operator $\Omega_i(f(\mathbf{x},t))$ With the Equilibrium function computed, we can determine the result of the collision step as shown in Listing 4.3. `delta_N` holds the result of Equation 3.8. `LAMBDA` is defined to be -1.

External force is applied at lines 4-5 to emulate a pump pushing fluid into the domain from west. At lines 8-11, the particle density of the lattice points are set to be the collision operator plus the external force.

```
1  delta_N = LAMBDA * ( lattice[LIB(x, y)].density[i][NOW] - N_eq );
2
3  // Apply external force at one end of the domain
4  if ( FORCE_COND )
5      delta_N += (1.0/3.0) * (c[i][0]*force[0] + c[i][1]*force[1]);
6
7  // Reflections at ghosts
8  if( ! ghost[GY(y)][GX(x)] )
9      lattice[LIB(x, y)].density[i][NEXT] = lattice[LIB(x, y)].density[i][NOW] + ↩
           delta_N;
10 else
11     lattice[LIB(x, y)].density[(i+3)%6][NEXT] = lattice[LIB(x, y)].density[i][NOW];
```

Listing 4.3: The last piece of the collision step. The densities at ghost sites are mirrored back into the domain, emulating a bounce-back condition.

4.1.2.2 Border Exchange

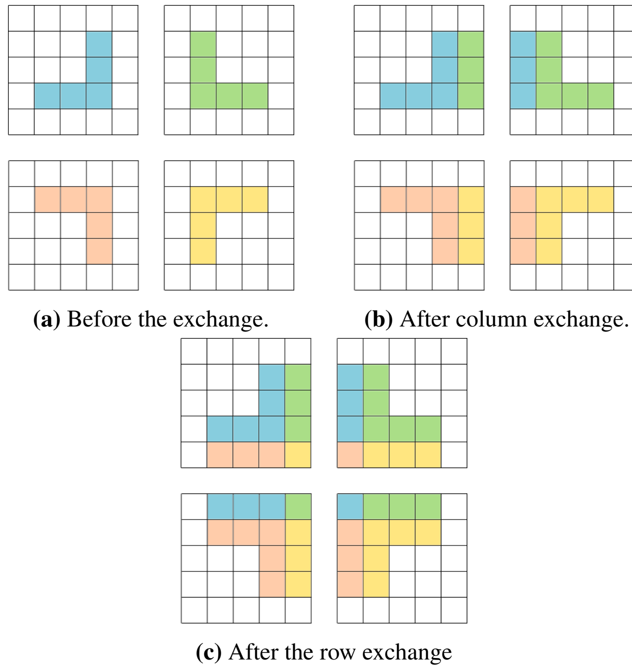
The edges of a subdomain must be transferred to the neighboring ranks before the propagation step can begin. All ranks must create 'halos', which purpose is to store the neighboring lattice sites. The halos are updated every iteration. To minimize the amount of required messages, row and column types are created in MPI. These are contiguous memory types which enable us to send a halo as a single message, keeping the overhead of communication low. The process is shown in Listing 4.4.

The corner element of the subdomain must also be communicated for the simulation to be correct. This is done by sending the vertical borders first. The horizontal borders are sent only after the vertical exchange have completed. The horizontal border must include the halo element on both sides to ensure that corner elements are transferred diagonally. Note that this process incurs a cost in the form of two implicit synchronizations, while sending corners individually would only result in one synchronization after all the messages have been dispatched. The process is depicted in Figure 4.3.

4.1.2.3 Propagate

The propagate method iterates over all lattice sites and transfers the new density of a lattice point out to its six neighbors. This is the opposite of traditional stencil applications, where the value of a point is decided by a function of its neighbors. This is why the halos must be exchanged before propagation can occur. The primary usage of the halos is to influence the local lattice sites, not vice versa. The method is shown in Listing 4.5.

```
1  void propagate ( void ) {
2      for( int y=0; y<local_grid_height_halo; y++ ) {
3          for( int x=0; x<local_grid_width_halo; x++ ) {
4              for( int i=0; i<6; i++ ) {
5                  int n_x = local_neighbor_x(y, x, i);
6                  int n_y = local_neighbor_y(y, i);
7
```



(a) Before the exchange. **(b)** After column exchange.

(c) After the row exchange

Figure 4.3: LBM Border Exchange. The data elements are colored to illustrate which rank they originate from. The outermost regions represents the halos. Notice that the corners are successfully transferred diagonally.

```c
void border_exchange() {
  MPI_Sendrecv (
    &lattice[LI(local_grid_width_halo-2, 1)], 1, border_col, east, 0,
    &lattice[LI(0, 1)], 1, border_col, west, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE
  );

  MPI_Sendrecv (
    &lattice[LI(1, 1)], 1, border_col, west, 1,
    &lattice[LI(local_grid_width_halo-1, 1)], 1, border_col, east, 1,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE
  );

  MPI_Sendrecv (
    &lattice[LI(0, 1)], 1, border_row, north, 0,
    &lattice[LI(0, local_grid_height_halo-1)], 1, border_row, south, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE
  );
  MPI_Sendrecv (
    &lattice[LI(0, local_grid_height_halo-2)], 1, border_row, south, 0,
    &lattice[LI(0, 0)], 1, border_row, north, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE
  );
}
```

Listing 4.4: Border Exchange in LBM

```
8                 if (n_x <= -1 || n_y <= -1 || n_x >= local_grid_width_halo || n_y ↩
                      >= local_grid_height_halo) {
9                     continue;
10                }
11
12                lattice[LI(n_x, n_y)].density[i][NOW] = lattice[LI(x, y)].density[i↩
                      ][NEXT];
13            }
14        }
15    }
16 }
```

Listing 4.5: The propagation step of LBM

4.2 SPH

This section describes the CFD problem, Dambreak, that our program computes, and our implementation of the time integration routine in the SPH application.

4.2.1 Dam Break

The initial geometry of the simulated fluid is a box representing a dam, as shown in the dark red region of Figure 4.5. The dam will immediately begin to collapse on itself as there is no solid wall on the east side of the dam, creating an event called *dam break*.

The dam can be scaled by adjusting the parameter `SCALE`. Because the dam is in two dimensions, a linear increase of the `SCALE` parameter will result in a quadratic increase of the area of the dam.

4.2.2 Implementation

Each time step in Time Integration can be broken into 5 parts: particle preparation, generating virtual particles, neighbor exchanging, node local physics, and particle migration. An overview can be found in Figure 4.4.

All particles are stored in a hash map to avoid array fragmentation when particles move to other ranks. At the particle preparation step, particles are copied from the hash map into a contiguous array, which makes it easier to iterate through the elements.

Virtual particles are generated for every particle close to a wall. These virtual particles mimick the *actual particles* and provide a bounce back effect when particles hit the wall.

In 'Border Exchange' the halo of each rank is sent to its neighbors. A halo is a set of particles (including virtual particles) which interaction radius (cf. Section 3.1.2) overlaps with a subdomain border. As such, all ranks must send and receive both to their left and their right neighbor. The exception is the leftmost and rightmost ranks, which has a maximum of one neighbor. The concept is illustrated in Figure 4.6. For simplicity, only the halos of the purple region is shown.

The border exchange routine executes a total of four message exchanges per rank. First, the ranks need to communicate the number of particles they intend to send to their neighbors. This enables the ranks to allocate space for their receive buffers. Rank n will send particle counts to n-1 and n+1 and likewise receive particle counts from n-1 and n+1. This accounts for the first two exchanges.

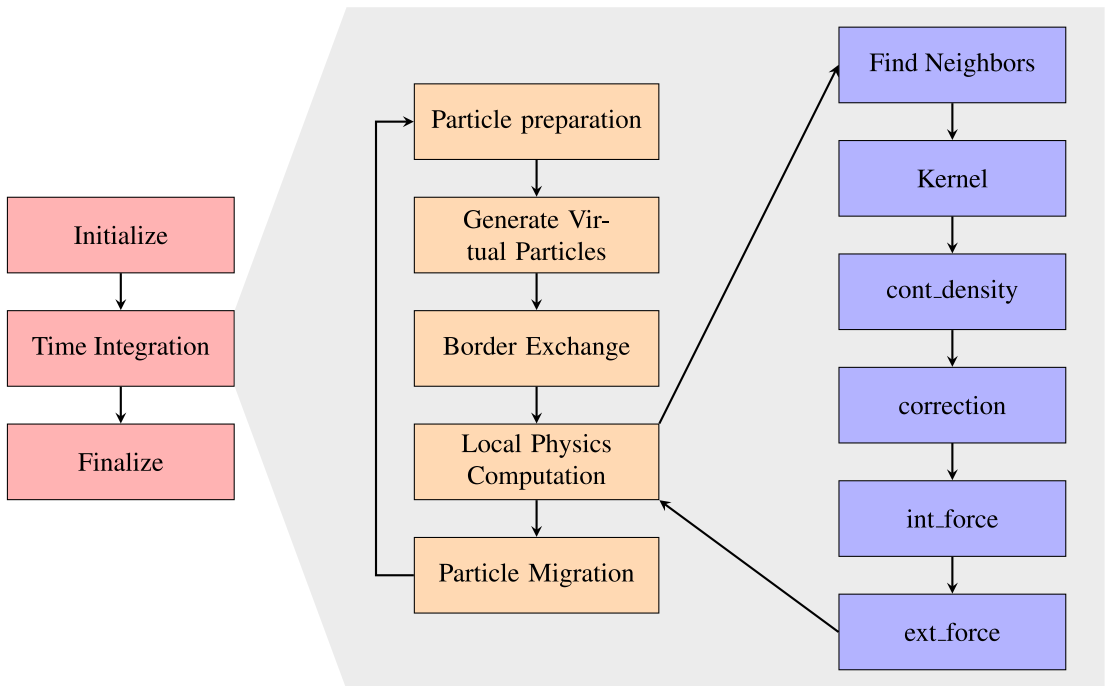


Figure 4.4: The lifecycle of the SPH application

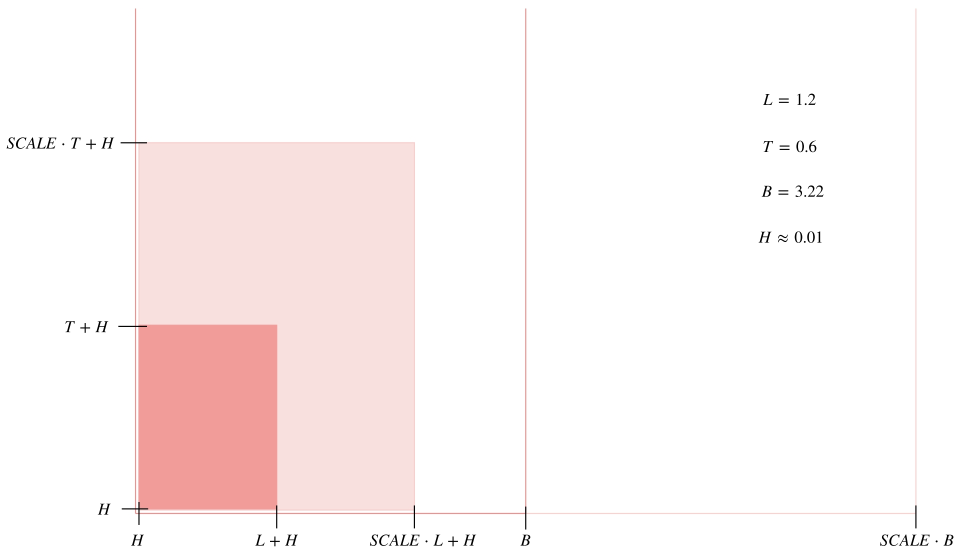


Figure 4.5: The scaling of the dam size.

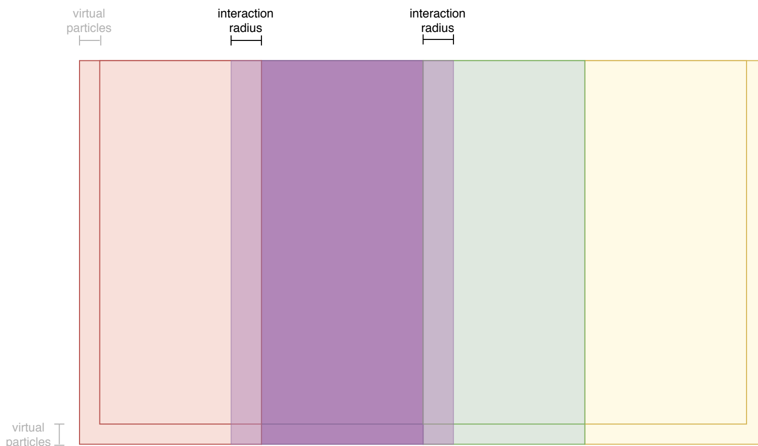


Figure 4.6: The halo of a subdomain contains a copy of the nearby particles of the neighbouring subdomains.

Then, the ranks prepare their send buffers so that the emigrating particles reside in contiguous memory. This memory is then sent to the neighboring ranks with two messages. The halo particles received from other ranks are referred to as *mirror particles*.

After borders have been exchanged, the particle dynamics are computed. First, we need to determine which particles interact. This is the main objective for the `find_neighbors` routine. This routine is the subject of much of our later analysis and is therefore described in detail in Section 4.2.2.1. The output of `find_neighbors` is a list of particle pairs. The routines `kernel`, `cont_density`, `correction`, `int_force` and `ext_force` all operate on the pairs, resulting in updated particles positions.

After the physical properties of the particles have been updated, they are written back to the hash map of the subdomain. Particles that have moved outside of the subdomain are now migrated to neighboring ranks.

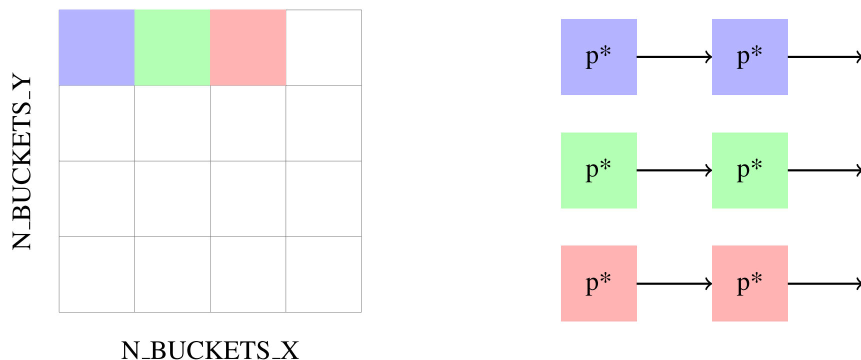
4.2.2.1 Finding Neighbors

This Section describes three methods for finding neighbors: *brute force*, *particle cell-linked list* and *pair/particle cell-linked list*. Within these, thread parallelism is implemented with worksharing.

Brute Force The brute force method, as seen in Listing 4.6, checks the euclidean distance between every possible pair of particles. A new pair is created if the distance is within the interaction radius. All pairs are assigned a unique index `kk`, and this index must be updated atomically to avoid race conditions in the thread pool. Furthermore, each particle tracks the number of interactions it is part of at any given iteration. This number must also be updated atomically, as multiple threads can operate on the same particle simultaneously. The cost of updating the interaction number is delayed by creating a thread local list `interactions` where threads can read and write without blocking each other.

```
#pragma omp parallel
{
    int interactions[n_total];
    for ( int_t i=0; i<n_total; i++ )
        interactions[i] = 0;

    #pragma omp barrier
    #pragma omp for
    for ( int_t i=0; i<n_total-1; i++ ) {
        for ( int_t j=i+1; j<n_total; j++ ) {
            real_t dist_sq =
                (X(i)-X(j))*(X(i)-X(j)) + (Y(i)-Y(j))*(Y(i)-Y(j));

            if ( dist_sq < (scale_k*H)*(scale_k*H) ) {
                int_t kk;

                #pragma omp atomic capture
                kk = n_pairs++;

                interactions[i] += 1;
                interactions[j] += 1;
                pairs[kk].i = i;
                pairs[kk].j = j;
                pairs[kk].r = sqrt(dist_sq);
                pairs[kk].q = pairs[kk].r / H;
                pairs[kk].w = 0.0;
                pairs[kk].dwdx[0] = pairs[kk].dwdx[1] = 0.0;
            }
        }
    }

    #pragma omp barrier
    for ( int_t i=0; i<n_total; i++ ) {
        #pragma omp atomic
        INTER(i) += interactions[i];
    }
}
```

Listing 4.6: Finding pairs through brute force



(a) Division of the subdomain into smaller sections known as cells or buckets. Each cell is the head of a linked list of particles.

(b) Linked lists of particles.

Figure 4.7: The bucket datastructure.

The interactions arrays are combined after all pairs have been discovered.

Cell-linked list for particles The idea of the cell-linked list is to only check for possible pairs in the close proximity of a particle. This requires some setup. First, the subdomain is divided into a grid of *buckets*[1] as seen in Figure 4.7. The width and height of a bucket is defined to equal the interaction radius. Therefore, the interaction space of a particle within a bucket does not extend beyond the neighboring buckets. Before populating the buckets we iterate through all particles and compute which bucket they belong to, as seen in Listing 4.7. The variables `N_BUCKETS_X` and `N_BUCKETS_Y` represent the number of buckets in the x/y direction. They are computed with Equations 4.3–4.5.

The subdomain is closed in the x-direction, meaning there is a finite width to create buckets. We get the number of buckets in the horizontal direction by adding 2R (2 times the interaction radius) to the subdomain width, and dividing the result by the bucket width. We add 2R to create bucket space for halo particles, or virtual particles at the west or east global border, assuming that halo width (R = 3.0 times H) is larger than the width of the virtual particles boundary (1.55 times H).

The height of the subdomain is not closed. Instead of creating an infinite number of buckets in the y-direction, we settle on a height which is 50% taller than the height of the dam prior to the dam break, represented as $1.5T$. Particles moving higher than $1.5T$ are placed in the end row buckets. We add $1.55H$ to accommodate for virtual particles at the

[1] What we call buckets if often referred to as *cells* in the literature. See Mattson and Rice (1999)

```
/* Compute bucket_x and bucket_y for all particles */
#pragma omp for
for (int i = 0; i < n_total; ++i) {
    particle_t *particle = &list[i];

    int actual_x = MIN((int) (((particle->x[0] - subdomain[0])+RADIUS) / ↩
         BUCKET_RADIUS) , N_BUCKETS_X-1);
    int actual_y = MIN((int) ((particle->x[1]+1.55*H) / BUCKET_RADIUS),N_BUCKETS_Y↩
         -1);

    particle->local_idx = i;
    particle->bucket_x = actual_x;
    particle->bucket_y = actual_y;
}
```

Listing 4.7: All particles are assigned to a bucket coordinate. Note that the particles are not inserted into a bucket yet.

bottom of the tank. $1.55H$ is the virtual particle boundary.

$$R = \text{Interaction Radius} = \text{Bucket Width} = \text{Bucket Height} \tag{4.3}$$

$$\texttt{N_BUCKETS_X} = \left\lceil \frac{x_{\text{end}} - x_{\text{begin}} + 2R}{R} \right\rceil \tag{4.4}$$

$$\texttt{N_BUCKETS_Y} = \left\lceil \frac{1.5T + 1.55H}{R} \right\rceil \tag{4.5}$$

The co-ordinates of the bucket which a particle belongs to are represented as `bucket_x` and `bucket_y`. They are computed with Equation 4.6–4.7. p_x and p_y represent a particles global x and global y position, respectively. The minimum is taken to ensure that no particle is assigned to an out of range bucket. R and $1.55H$ is added to make sure that the position does not come out negative for halo particles or virtual particles, respectively. We subtract x_{begin} from p_x to attain the local x-position of particle p.

$$\texttt{bucket_x} = \min\left(\frac{p_x - x_{\text{begin}} + R}{R}, \texttt{N_BUCKETS_X} - 1\right) \tag{4.6}$$

$$\texttt{bucket_y} = \min\left(\frac{p_y + 1.55H}{R}, \texttt{N_BUCKETS_Y} - 1\right) \tag{4.7}$$

The `fill_buckets` method is shown in Listing 4.8. Each thread is given it's own set of buckets to populate. As each bucket is the head of a linked list, insertion must happen atomically. However, since there is at max one thread operating at a particular bucket, no conflicts occur.

An alternative approach would be to distribute particles among threads. This would bring the complexity down from $\mathcal{O}(\texttt{N_BUCKETS_X} \cdot \texttt{N_BUCKETS_Y} \cdot \texttt{n_total})$ to $\mathcal{O}(\texttt{n_total})$, but would incur a serial region for updating buckets, as multiple threads would attempt to update the same bucket. The penalty of the serial region is mitigated by using OpenMP locks, which ultimately results in this approach being faster.

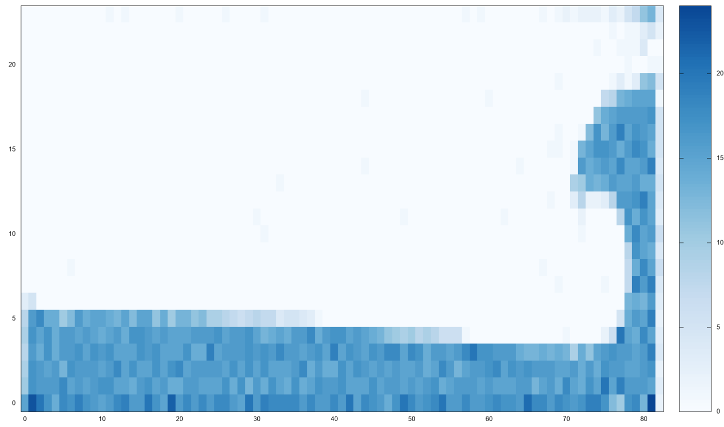
In Figure 4.8, we see how the particles are distributed to buckets illustrated by a heat map. One realization to draw from this is that iteration over the bucket domain should be

```c
#define BID(X,Y) (Y + N_BUCKETS_Y * X)

/* Populate Buckets */
#pragma omp for
for (int x = 0; x < N_BUCKETS_X; ++x) {
    for (int y = 0; y < N_BUCKETS_Y; ++y) {
        bucket_t* bucket = buckets[BID(x, y)];

        for (int i = 0; i < n_total; ++i) {
            particle_t* particle = &list[i];

            if (particle->bucket_x != x || particle->bucket_y != y)  {
                continue;
            }

            if(bucket->particle == NULL) {
                bucket->particle = particle;
            } else {
                bucket_t* new_bucket = (bucket_t*)malloc(sizeof(bucket_t));
                new_bucket->particle = particle;
                new_bucket->next = bucket;

                buckets[BID(x, y)] = new_bucket;
            }
        }
    }
}
```

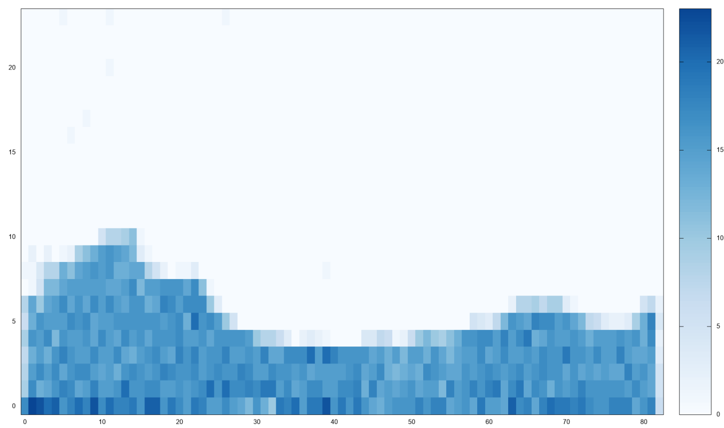
Listing 4.8: Buckets are populated without introducing any critical sections.

```c
#define BID(X,Y) (Y + N_BUCKETS_Y * X)

/* Populate Buckets */
#pragma omp for
for (int i = 0; i < n_total; ++i) {
    /* Give particle a local index */
    list[i].local_idx = i;

    /* Compute bucket coordinates */
    int bucket_x = list[i].bucket_x;
    int bucket_y = list[i].bucket_y;

    /*Lock possible critical section*/
    omp_set_lock(&(lock[BID(bucket_x,bucket_y)]));

    /* Pointer to relevant bucket */
    bucket_t* bucket = buckets[BID(bucket_x, bucket_y)];

    if (bucket->particle == NULL) {
        bucket->particle = &list[i];
        bucket->next = NULL;
    } else {
        bucket_t* new_bucket = (bucket_t*)malloc(sizeof(bucket_t));
        new_bucket->particle = &list[i];
        new_bucket->next = bucket;

        buckets[BID(bucket_x, bucket_y)] = new_bucket;
    }
    /*Unlock*/
    omp_unset_lock(&(lock[BID(bucket_x,bucket_y)]));
}
```

Listing 4.9: Race conditions are handled by introducing a lock .



(a) The large wave is apparent, timestep 9 300, SCALE 1.



(b) The wave has almost settled, timestep 25 000, SCALE 1.

Figure 4.8: Number of particles per bucket (shown as a square) on different timesteps.

```c
/* Create neighbors */
 #pragma omp for
for (int_t i = 0; i < n_total-1; ++i) {
    particle_t* particle = &list[i];
    int bx = particle->bucket_x;
    int by = particle->bucket_y;

    /* Center */
    create_pairs(bx, by, buckets, particle, &n_pairs, interactions);
    /* North West */
    create_pairs(bx-1, by+1, buckets, particle, &n_pairs, interactions);
    /* North */
    create_pairs(bx, by+1, buckets, particle, &n_pairs, interactions);
    /* North East */
    create_pairs(bx+1, by+1, buckets, particle, &n_pairs, interactions);
    /* East */
    create_pairs(bx+1, by, buckets, particle, &n_pairs, interactions);
    /* South East */
    create_pairs(bx+1, by-1, buckets, particle, &n_pairs, interactions);
    /* South */
    create_pairs(bx, by-1, buckets, particle, &n_pairs, interactions);
    /* South West */
    create_pairs(bx-1, by-1, buckets, particle, &n_pairs, interactions);
    /* West */
    create_pairs(bx-1, by, buckets, particle, &n_pairs, interactions);
}
```

Listing 4.10: The create pairs routine is called for all nine buckets of interest.

done vertically. This way, all threads will have similar work loads. If the bucket domain were split horizontally, the thread with the top rows of buckets would have no particles to compute!

With the buckets fully populated, we are ready to create pairs. All particles are distributed among threads, and therefore the concerns regarding race conditions mentioned in Section 4.2.2.1 still apply for the pair array and the interaction attribute of the particles. However, there are less potential pairs to evaluate because of the bucket design, and therefore less contention for memory.

The pair creation process is detailed in Listing 4.10 and Listing 4.11. Any given particle is compared to other particles in the bucket they share, and to all particles in the eight neighboring buckets. The global index (`idx`) of the particles are compared to ensure that a pair is not added twice.

As before, the thread local `interactions` array is collected after all pairs are discovered.

Cell-linked list for particles and pairs This method extends the previous design by distributing pairs to buckets. This eliminates the need for critical sections in the 'Local Physics Computation' stage.

A two dimensional array similar to the one depicted in 4.7 is created to hold pairs. Each cell in this new array is the head of a linked list of pairs.

A base particle is checked against child particles in the same bucket and all particles in the eight neighboring buckets. The resulting pairs are inserted into the linked list which share the same bucket co-ordinates as the base particle.

As opposed to the previous pair creation routine, the threads are responsible for a set

```c
void create_pairs(int bx, int by, bucket_t** buckets,
                  particle_t* particle, int_t* n_pairs,
                  int_t* interactions) {
    if (bx >= N_BUCKETS_X || bx < 0 || by >= N_BUCKETS_Y || by < 0 || particle == NULL) {
        return;
    }

    bucket_t* current = buckets[BID(bx, by)];
    while (current != NULL && current->particle != NULL) {
        if (current->particle->idx < particle->idx) {
            double distance = sqrt(
                                   pow(particle->x[0] - current->particle->x[0], 2) +
                                   pow(particle->x[1] - current->particle->x[1], 2)
                                   );
            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 = 0.0;
                pairs[pair_idx].dwdx[0] = pairs[pair_idx].dwdx[1] = 0.0;

            }
        }
        current = current->next;
    }
}
```

Listing 4.11: Pair creation with buckets.

of buckets instead of a set of particles. This means that pair creation can be done without locks. However, pairs across buckets will be created twice. This produces some trouble for the methods `kernel`, `cont_density`, `correction` and `int_force` as they all work through the pair arrays. To accommodate, they are modified so that work done on the "non-base" part of a pair is only written back if the "non-base" particle have the same bucket co-ordinates as the "base particle". This modification takes care of any duplicate forces, and makes the `local physics` routine completely lockless. Note that a "base particle" is a particle we find when iterating through buckets, and a "non base particle" is a particle we find when we search for neighbors of the "base particle".

The interaction attribute of the base particle is incremented for every pair it participates in. The other particle is only updated if it shares the same bucket as the base particle, as this pair is not evaluated again later.

The complete pair creation routine is shown in Listing A.1.

4.3 Computer Architectures

In this section we describe the computer architectures used to analyze our applications. In Subsections 4.3.1 and 4.3.2, the supercomputers Vilje and Idun are introduced, respectively.

4.3.1 Vilje

Vilje is a cluster machine procured by NTNU, met.no and UNINETT Sigma. It is a SGI Altix ICE X with a total of 22464 cores. The details are listed in Table 4.1, and its topology is depicted in Figure 4.9.

Nodes	1404
Processors per node	2
Cores per processor	8
Processor type	Intel Xeon E5-2670
Processor Clock Frequency	2.6 GHz
L1 Instruction Cache (private)	32 KB
L1 Data cache (private)	32 KB
L2 Cache (private)	256 KB
L3 Cache (shared)	20 MB
Primary Memory per node	32 GB
Interconnect	Infiniband
Theoretical peak performance	467 Teraflop/s
LINPACK rating	396.70 Teraflop/s

Table 4.1: Vilje Specification

Figure 4.9: The topology of Vilje

	EPT	EPIC
Nodes	27	8
Processors per node	2	2
Cores per processor	10	18
Processor type	E5-2630 v4	E5-2695 v4
Processor Clock Frequency	2.2 GHz	2.1 GHz
L1 Instruction Cache (private)	32 KB	32 KB
L1 Data cache (private)	32 KB	32 KB
L2 Cache (private)	256 KB	256 KB
L3 Cache (shared)	25 MB	45 MB
Primary Memory per node	64 GB	64 GB
Interconnect	Infiniband	Infiniband
Theoretical peak performance	N/A	N/A
LINPACK rating	N/A	N/A

Table 4.2: IDUN Specification

4.3.2 Idun

IDUN is a cluster machine at NTNU with multiple queues. The two queues used in this project are *EPT* and *EPIC*, detailed in Table 4.2. EPT is a homogeneous cluster, while EPIC is a heterogeneous cluster where each node contains two Tesla P100 GPUs in addition to Xeon CPUs. The GPUs are not used in this project. The topologies of EPT and EPIC are shown in Figure 4.10 and in Figure 4.11, respectively.

Figure 4.10: The topology of IDUN EPT

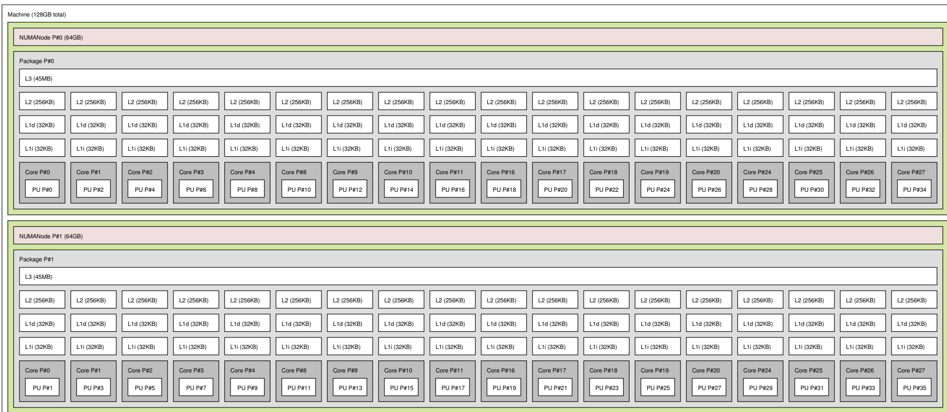


Figure 4.11: The topology of IDUN EPIC

Chapter 5

Results and Discussion

In this chapter, we develop our performance models and present and discuss our results. In Section 5.1, we discuss our application models on intra and inter node level. In Section 5.2, we present the networking parameters of our testing platforms, in terms of the heterogeneous Hockney model. In Section 5.3, we summarize the work in Sections 5.1 and 5.2 into a set of performance models and recommendations. In Section 5.4, we list some useful parameters and their meaning.

5.1 Application Models

In this section we analyze and discuss the characteristics of our proxy applications on intra and inter node level.

5.1.1 LBM

The total computation time of LBM is composed of the computation time and the communication time. In Section 5.1.1.1, we discuss the computation part and in Section 5.1.1.2, we discuss the communication part.

5.1.1.1 Intra-Node Parallelism

Collision and propagation are the sole contributors to the computational load of the LBM application. On Vilje, the collide function is responsible for 62.6 % of the computational load while running with 8 threads. Similarly, the collide function is 53.7 % of the computational load on EPIC while running with 18 threads. Both measured with 2 ranks on one node and SCALE = 20. These results are depicted in Figure 5.3.

Both Collide and Propagate is profiled with high resolution to capture the behaviour of the compute steps in the application. The wall time spent per iteration is close to constant, which means that early iterations are equally costly as late iterations. This can be attributed to the static nature of the application; lattice sites are immovable, and each thread must

compute the same number of lattice sites for all iterations. This effect is shown in Figures 5.4, 5.5 and 5.6, for the Moffatt problem on EPIC, EPT and Vilje, respectively. The results for the Cylinder problem are similar.

Because the time per iteration is constant, the total run time can be easily computed by Equation 5.1, where the constants c_c and c_p can be measured by running the application for a small number of iterations. N_{it} is the number of iterations.

$$\text{T}_{\text{compute}} = (\text{T}_{\text{collide}} + \text{T}_{\text{propagate}})N_{\text{it}} \approx (c_c + c_p)N_{\text{it}} \tag{5.1}$$

Note that the constants in Equation 5.1 are specific for a given thread count, architecture and number of lattice points. Therefore, the result from the execution must be retrieved with the same thread count and on the same architecture as the one we wish to model. We claim that the execution time on different number of lattice points, can be obtained by multiplying the average execution time per lattice point by the number of lattice points. This approximation is captured in Equation 5.2.

$$\text{T}_{\text{compute, height_b, width_b}} = \frac{\text{width_b} \cdot \text{height_b}}{\text{width_a} \cdot \text{height_a}} \cdot \text{T}_{\text{compute, height_a, width_a}} \tag{5.2}$$

In Equation 5.2, the terms width_a and height_a are the width and height of the lattice that the execution result is drawn from (baseline), while width_b and height_b are the width and height of the lattice that we wish to predict the computation time of.

Speedup and efficiency is measured for all architectures with three configurations: Moffatt work-share, Cylinder work-share and Moffatt with taskloops. The baseline for the measurements is execution time on a single thread.

EPIC and EPT enjoy a healthy speedup across all three problem configurations after maximizing the number of threads per node. However, some configurations lead to speedup peaks before all cores are utilized, as shown in Figure B.8a and Figure B.9a.

Speedup on Vilje is particularly bad for the propagate step across all configurations, with speedup decreasing rapidly as seen in Figure B.4 and in Figure B.1a.

The early peaks and the decreasing speedup can be explained by the access pattern of the propagate method, and the node configuration on the test platform.

In Listing 4.5, we see that the propagate method makes 12 memory accesses per lattice site, with no floating point operations. This leads to diminishing returns when the number of threads grows because much of the CPU time will be spent waiting for memory access. In comparison, the collide method makes 14 memory accesses and performs 242 floating point operations. In terms of the Roofline model, the application is memory bound because of its low operational intensity (floating point operations per byte retrieved). The Roofline model is further explained in Section 3.4.2.1.

The testing platforms all have nodes with two sockets, where each socket has its own private level 3 cache. The primary memory attached to the socket is shared. With a *compact* affinity setting, threads will be scheduled together on a socket (as opposed to *scattered*, where threads are evenly spread on sockets). A job running with a thread count of 8 on Vilje will fill up a single socket, saturating its memory bus. Increasing the thread count will populate the free socket, resulting in a rapid increase in speedup. The general pattern is shown in 5.1, and the pattern is mimicked by the results in Figures B.2, B.3, B.5, and B.6. The results were gathered with a compact affinity configuration.

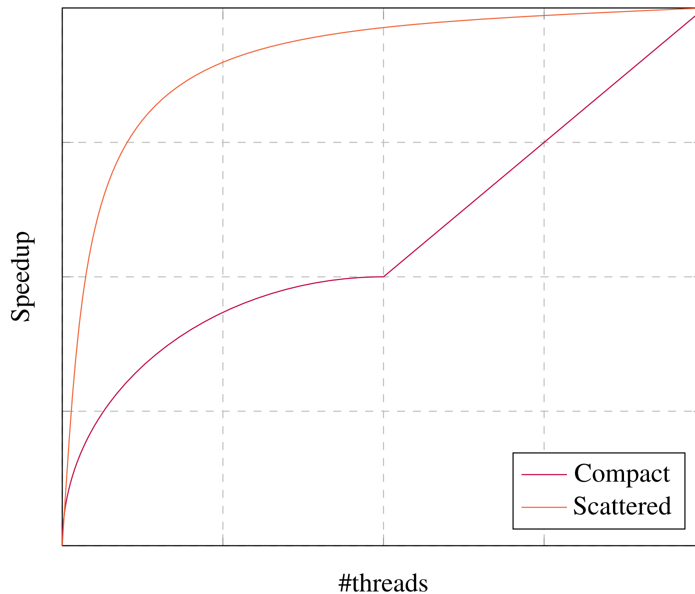


Figure 5.1: Saturation of sockets

To better utilize the available resources of a node, we measure running one process on each socket and only using threads within sockets. The speedup and efficiency of this configuration and a problem SCALE of 20 is shown in Figure 5.7. The efficiency of propagate and collide is maintained on both sockets, as opposed to getting decreased performance because of idle cores that are under utilizing the memory channels.

Because the application is memory bound, all the following results and models are executed with one rank per socket, and a problem size of SCALE=20.

Comparison of Workshare and Task implementations As mentioned in Section 3.1.1, the LBM application maintains a ghost array which is used to track solid points in the domain. There is less work to be done at solid points for the collision method, which can lead to an unbalanced amount of work if some threads carry more ghost points than others.

The motivation behind using tasks with OpenMP is to achieve load balance by creating tasks and deciding the task-thread mapping at run time. In this sub section, we compare traditional worksharing constructs with OpenMP tasks.

Figure 5.2 shows the wall time of a single iteration of LBM for two problems, Cylinder and Moffatt, running on two architectures, Vilje and EPIC, with two programming models, work sharing and task. The Cylinder problem with tasks is included as a control point. With tasks on Vilje we see an improvement of 31.2 % with the Cylinder problem, and an improvement of 26.5 % with the Moffatt problem. On EPIC the introduction of tasks seems to increase the time per iteration slightly. The fact that tasks has an impact dependent on the architecture indicate that the iteration time is controlled by factors like the memory hierarchy, processor features and thread scheduling or other things which are

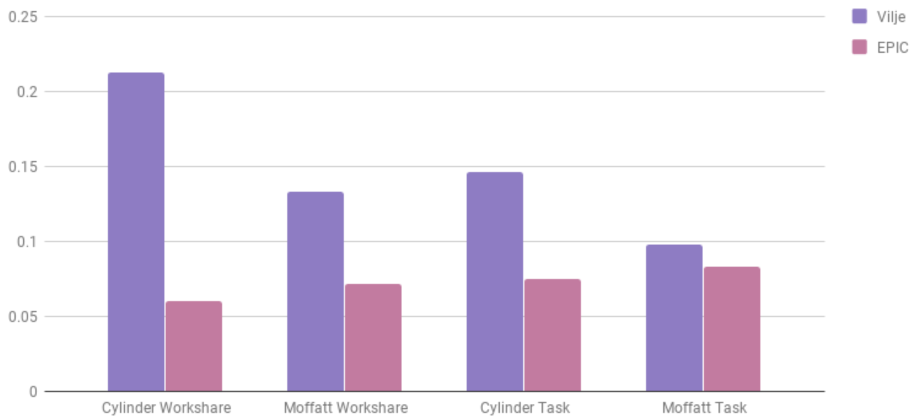
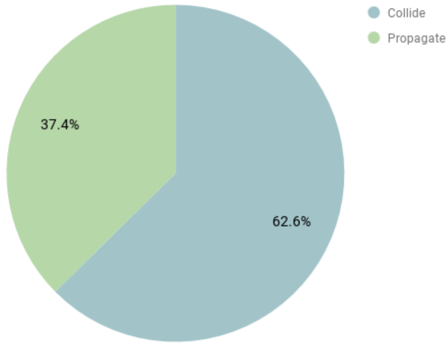


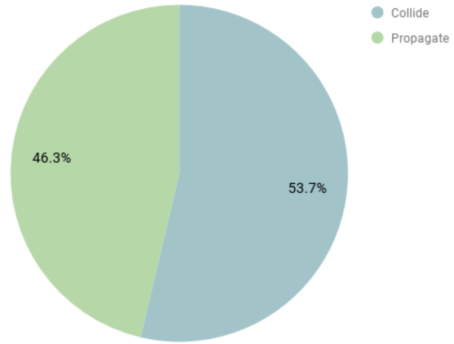
Figure 5.2: Single iteration time of workshare and task

out of the scope of this project.

With regards to performance modelling, the solution is to determine the architectural parameters of the machine under evaluation by running a small instance of the problem in order to capture the behaviour of the system.

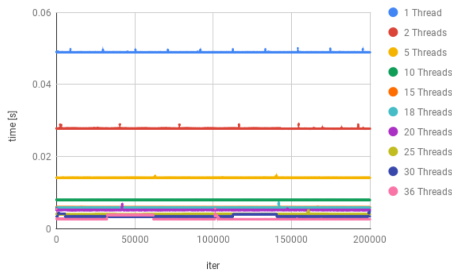


(a) Vilje, 8 threads

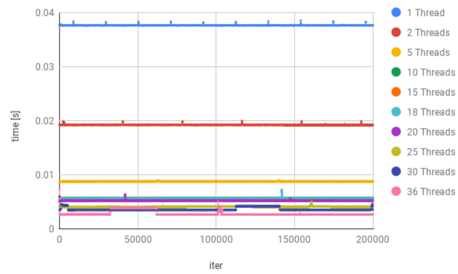


(b) EPIC, 18 threads

Figure 5.3: The workload of LBM's time integration phase split between Collision and Propagation

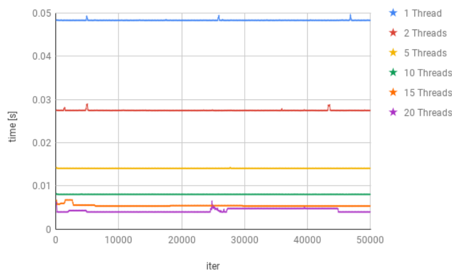


(a) EPIC Collide

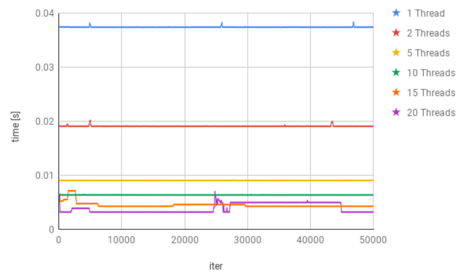


(b) EPIC Propagate

Figure 5.4: EPIC. The time per iteration on multiple numbers of threads.

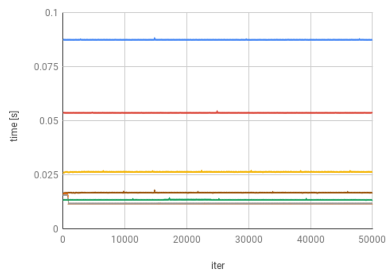


(a) EPT Collide

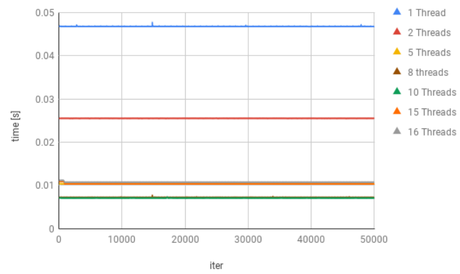


(b) EPT Propagate

Figure 5.5: EPT. The time per iteration on multiple numbers of threads.

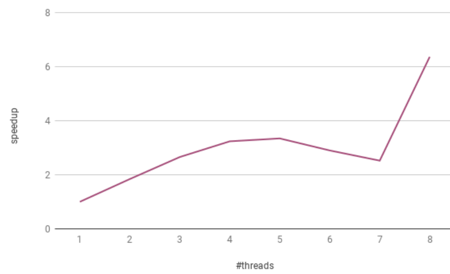


(a) VILJE Collide

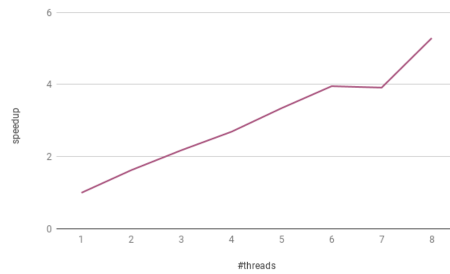


(b) VILJE Propagate

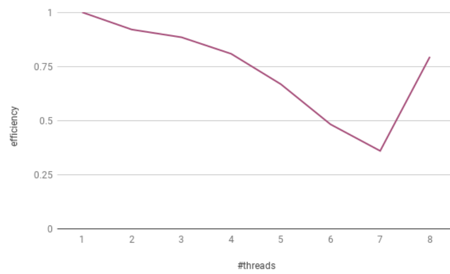
Figure 5.6: VILJE. The time per iteration on multiple numbers of threads.



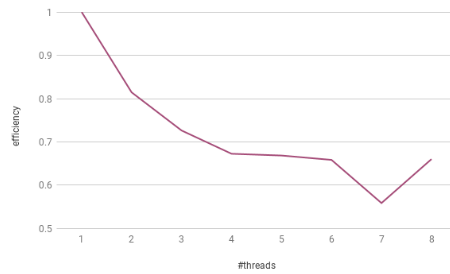
(a) Propagate Speedup



(b) Collide speedup



(c) Propagate Efficiency



(d) Collide Efficiency

Figure 5.7: Moffatt VILJE Speedup and Efficiency on SCALE 20 with 2 ranks

5.1.1.2 Inter Node Communication

In this section we discuss the distributed memory parallelism of the LBM application. As described in Section 4.1.2.2, the problem domain is split into a two dimensional grid of subdomains, where each part of the grid belongs to an MPI rank. In the following analysis, we assume a square number of ranks $N > 1$ and a square topology. The exact topology is dependent on the MPI implementation, as certain rank sizes (like primes greater than 2) will result in a one dimensional topology. However, the one dimensional case is a degenerate class of the two dimensional case and is easier to reason about. For non-square two dimensional topologies, the analysis is more specific to the rank layout as it requires knowledge about the width and height of the topology.

The communication pattern of the LBM application on 64 ranks is depicted in Figure 5.8. Each rank has two close neighbors and two far neighbors. Assuming a row major topology, the close neighbors are computed with Equations 5.3 and 5.4, and the far neighbors are computed with Equations 5.5 and 5.6. The domain is periodic in the horizontal direction, and therefore rank 0 must communicate with rank $N - 1$.

$$\text{neighbor}_{\text{west}} = (\text{rank} - 1) \mod N \tag{5.3}$$
$$\text{neighbor}_{\text{east}} = (\text{rank} + 1) \mod N \tag{5.4}$$

$$\text{neighbor}_{\text{north}} = (\text{rank} + \sqrt{N}) \tag{5.5}$$
$$\text{neighbor}_{\text{south}} = (\text{rank} - \sqrt{N}) \tag{5.6}$$

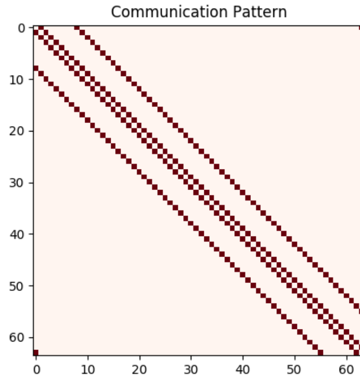


Figure 5.8: The communication pattern of the LBM application represented as an adjacency matrix, where red squares are neighbors. A square processor topology is assumed.

One message is sent for each direction, resulting in four messages per rank per iteration. The one exception is that the ranks placed in the top or bottom of the grid topology will not exchange messages north or south, meaning they only exchange three messages.

However, the exception is irrelevant for the execution time of the exchange routine because of the implicit synchronization in the routine (cf. Section 4.1.2.2). All ranks must wait for the slowest rank to finish communicating before execution can continue.

Each message consist of a row or column of lattice points, where each lattice point has a size of 64 bytes. The size of a subdomain column H_l (height local) and a subdomain row W_l (width local) is computed with Equation 5.7 and Equation 5.8, respectively. H_g and W_g represent the global height and width of the domain, while p_x and p_y is the width and height of the rank topology.

$$H_l = \frac{H_g}{p_y} \tag{5.7}$$

$$W_l = \frac{W_g}{p_x} \tag{5.8}$$

The number of bytes w to send in the horizontal directions is described by Equation 5.9.

$$w_{\text{east}} = w_{\text{west}} = 64H_l = 64\frac{H_g}{p_y} \tag{5.9}$$

Similarly, the number of bytes to send in the vertical directions is described by Equation 5.10. We add two elements to the width to cover the corner elements, as described in Section 4.1.2.2.

$$w_{\text{north}} = w_{\text{south}} = 64(W_l + 2) = 64(\frac{W_g}{p_x} + 2) \tag{5.10}$$

The time required for completing the exchange phase is dependent on the number of bytes per message, the number of messages and architecture specific parameters. We model the total running time of the border exchange routine as the maximum of horizontal communication across all ranks, plus the maximum of vertical communication across all ranks, because the horizontal communication must finish before the vertical. The model is described by Equation 5.11.

$$\text{T}_{\text{border_exchange}} \approx \max_{rank \in N}[\text{T}_{\text{east}}(\text{rank}) + \text{T}_{\text{west}}(\text{rank})] + \max_{rank \in N}[\text{T}_{\text{north}}(\text{rank}) + \text{T}_{\text{south}}(\text{rank})] \tag{5.11}$$

Note that if we are unable to send and receive at the same time, the time for each term will double.

Each term of the expression is a function of bytes to be exchanged, and the latency (α) and bandwidth (β) of the architecture. The latency and the bandwidth are parts of the Heterogeneous Hockney model, as described in Section 3.4.1.1. The expression is described in Equation 5.12, where i and j represent rank i and rank j. The indices implicitly decide which w to be used. For example, T_{east}(rank) would translate to $\text{T}(i, j)$ where i is "rank" and j is the eastern neighbor of "rank". In this case, w_{east} should be used.

$$\text{T}(i, j) \approx \alpha_{ij} + w_{\text{direction}}\beta_{ij}, i \neq j \tag{5.12}$$

5.1.2 SPH

The total computation time of SPH is composed of the computation time and the communication time. In Section 5.1.2.1, we discuss the computation part and in Section 5.1.2.2, we discuss the communication part.

5.1.2.1 Intra-Node Parallelism

This section studies the computation time of three different variations of the SPH program; the brute-force method described in Section 4.2.2.1, the method with cell-linked lists for particles described in Section 4.2.2.1, and the method with cell-linked lists for both particles and pairs described in Section 4.2.2.1.

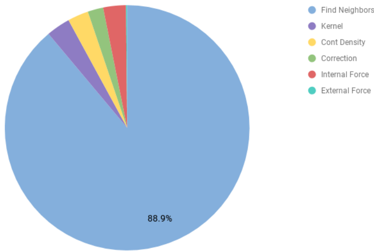
The computation time in the SPH program is dominated by the `time_step` method where the particle positions are updated. Profiles of the `time_step` method of the three implementations are shown in Figure 5.9.

Figure 5.9 shows that the `find_neighbors` method dominates the execution time of `time_step` in all three variations. For the brute_force version presented in Figures 5.9a and 5.9b, the dominance increases with SCALE, occupying 88.9% and 94.3% of the total execution time of `time_step` at SCALE 1 and SCALE 2, respectively. The share increases in proportion to the particle count, as a linear increase in particles result in a quadratic increase in the number of particle combinations in the nested for loop shown in Line 9 of Listing 4.6 (henceforth, the *neighbor loop*). The other methods in `time_step` consist of single loops that iterate through either particles or pairs. In contrast, the pair finding routine of the cell-linked particle list version visualized in Figures 5.9c and 5.9d, explores only particles in neighboring buckets as described in Section 4.2.2.1. Hence, the differences between SCALEs are minimal, at 89.3% and 88.9% at SCALE 1 and SCALE 2, respectively. In Figures 5.9e and 5.9f where particles and pairs are organized as cell-linked lists, the `find_neighbors` method is less dominant and decreasing with increasing SCALE, from 74.8% at SCALE 1 to 50.8% at SCALE 2. We attribute this effect to the absence of critical sections in this implementation as described in Section 4.2.2.1.

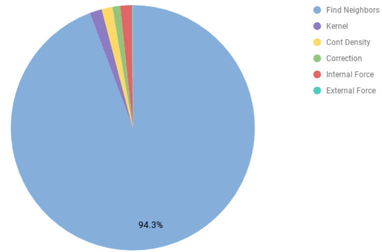
Profiling on Vilje shows a similar dominance from the `find_neighbors` method. 84.8%, 82.0% and 38.2% of the compute time is occupied by `find_neighbors` in brute-force, cell-linked particle list and cell-linked pair and particle lists, respectively (at SCALE 1 with 16 threads). For cell-linked pair and particle lists, the next largest term is `kernel` on 18.1%. In all cases, `find_neighbors` is the largest contributor to `time_step`'s overall execution time for all three implementations. Hence, the computation time of all three SPH implementations can be approximated as in Equation 5.13.

$$T_{\text{compute}} \approx T_{\text{find_neighbors}} \tag{5.13}$$

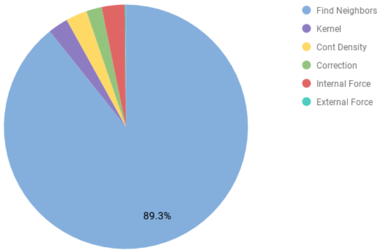
Note that the execution times of `kernel`, `cont_density`, `correction` and `int_force` are significant in cell-linked lists for pairs and particles. However, an analysis of these methods are beyond the scope of this study. In addition, we have only examined `time_step` as most compute time is spent here. However, profiling on the entire `time_integration` method described in Section 4.2.2, shows that `time_step` takes up at least 95.5% of the total execution time at SCALE 1 for the brute_force method. We argue



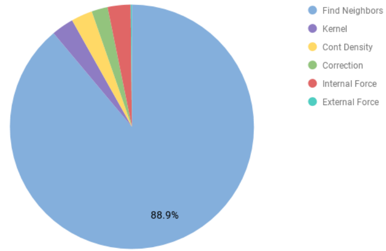
(a) Brute-force, SCALE 1.



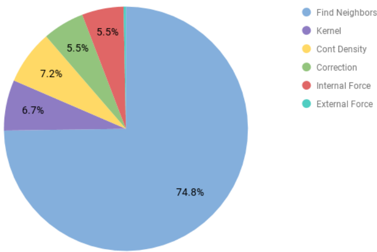
(b) Brute-force, SCALE 2.



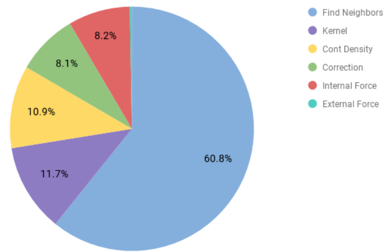
(c) Cell-linked list for particles, SCALE 1.



(d) Cell-linked list for particles, SCALE 2.

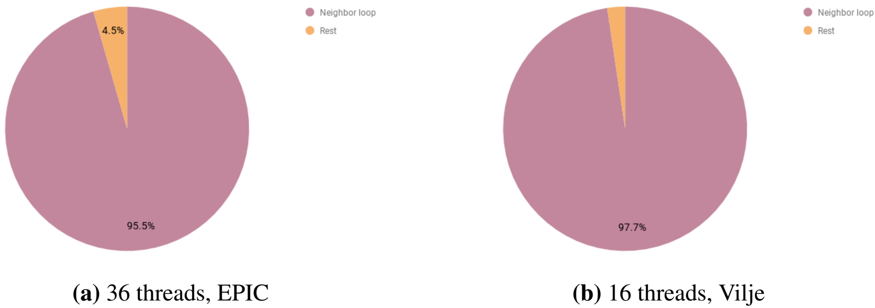


(e) Cell-linked list for particles and pairs, SCALE 1.



(f) Cell-linked list for particles and pairs, SCALE 2.

Figure 5.9: Distribution in execution time of methods in time_step, for three different SPH implementations. The numbers are collected by running on 36 threads on EPIC.



(a) 36 threads, EPIC

(b) 16 threads, Vilje

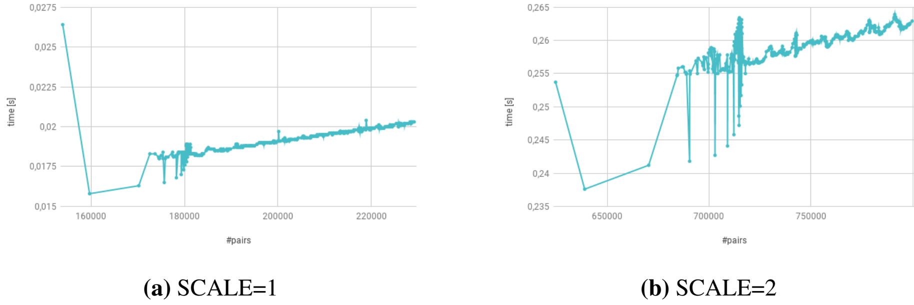
Figure 5.10: Distribution in execution time of different parts of the `find_neighbors` method (brute-force), SCALE=1.

that this percentage is less for cell-linked lists for pairs and particles as the only occurrences of critical sections are in methods other than `time_step`. These considerations are important because these methods may become the bottleneck if the compute time of `find_neighbors` becomes low enough so that the other methods become more substantial in terms of compute time. Detailed explanations are however beyond the scope of this thesis.

In addition, the number of particles mentioned in this section is the sum of actual particles, virtual particles and mirror particles (cf. Section 4.2.2) that are generated in the border_exchange phase as a result of distributed memory parallelism. This section, however, studies the effect of varying thread counts on one node, without considering internode communication. Therefore, there are no distributed memory parallelism and hence no mirror particles. However, the discussion in this section does not differentiate between different types of particles, and can therefore also be utilized when mirror particles are present. In addition, the graphs and equations used in the following discussions are functions of the particle count or the pair count. We therefore consider the findings of this section to be relevant for other flow problems, and not limited to the dambreak problem.

Sections 5.1.2.1, 5.1.2.1 and 5.1.2.1, analyze and discuss the compute time of the `find_neighbors` methods of the three implementations on different thread counts, SCALEs and Vilje, EPT and EPIC node architectures. EPIC and EPT were chosen out of the architectures on IDUN because they have the greatest and smallest number of cores per node. EPIC is treated as the point of reference.

Brute-Force A study of the computation time of the `find_neighbors` method shows the dominance of the neighbor loop, as shown in Figure 5.10. The neighbor loop executed on 16 threads on Vilje, visualized in Figure 5.10b, and 36 threads on EPIC, visualized in Figure 5.10a, dominates by 97.7% and 95.5%, respectively. As a nested loop iterating over particles combinations will occupy a larger percentage as the SCALE increases, this tendency will be valid and even more apparent on larger SCALEs. Therefore, we claim that a detailed analysis of the neighbor loop will be sufficient to predict the total execution time of the `find_neighbors` method. This relationship is described in Equation 5.14.



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

Figure 5.11: The relationship between number of pairs and the execution time of the neighbor loop in `find_neighbors` brute-force, 18 threads on EPIC, 200 000 timesteps.

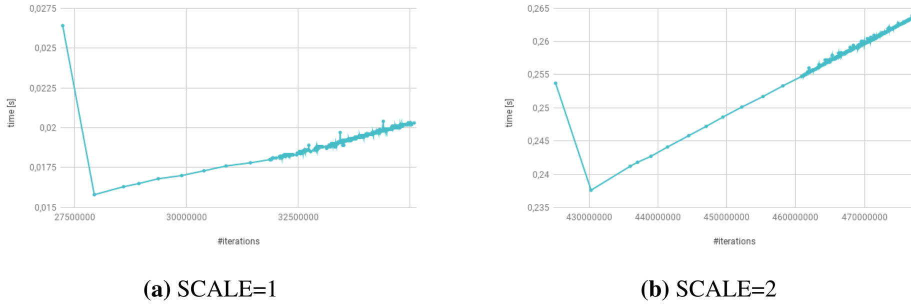
$$T_{\text{find_neighbors}} \approx T_{\text{neighbor_loop}} \tag{5.14}$$

The neighbor loop iterates through all particle combinations by comparing every particle by every other particle that has not been examined. This relationship between the number of iterations of the neighbor loop and the particle count is captured in Equation 5.15. When two particles are within each others interaction radius, the pair count is updated atomically, and a new pair is created and initialized. The particle count influences the overall computation time by affecting the number of iterations where pairs are tested for proximity, while the number of pairs can be associated with the number of times the body of the if statement in Lines 15-27 of Listing 4.6 where the critical section is located, is reached. Therefore, the execution time of the neighbor loop depends on both the number of pairs and the number of iterations of the neighbor loop. Figure 5.11 shows the relationship between execution time and the pair count. Figure 5.12 shows the relationship between execution time and the number of neighbor loop iterations.

$$N_{\text{loop_iter}} = \binom{N_{\text{particle}}}{2} = \frac{N_{\text{particle}} \cdot (N_{\text{particle}} - 1)}{2} \tag{5.15}$$

Figure 5.11 shows the execution time of the neighbor loop when the pair count per timestep increases at different SCALEs. The increase in the number of pairs, both due to variations within a SCALE as showed in Figure 5.11a and to variations caused by increasing the SCALE as shown in Figure 5.11b, results in an increase in compute time. Both graphs present an approximately linear relationship between the number of pairs and the overall execution time with a few outliers that are especially noticeable when the number of pairs are low.

Figure 5.12 presents the execution time with increasing iteration count at SCALE 1 and SCALE 2. When executing on 18 threads on EPIC at the two SCALE configurations, the iteration count seems to be more close-coupled with the execution time of the neighbor loop than the pair count, as the linear relationship between iterations and execution time is clearer (see Figures 5.11 and 5.12). We attribute this effect to the number of iterations being much larger than the number of pairs, meaning that most iterations proceed without entering the body of the if statement. In fact, a pair is detected for every 164 iterations at



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

Figure 5.12: The relationship between number of iterations of the body of the neighbor loop and the execution time in `find_neighbors` brute-force, 18 threads on EPIC, 200 000 timesteps.

SCALE 1, were 164 is the median over 200 000 timesteps. At SCALE 2 this number is even greater, at 630 $\frac{\text{iterations}}{\text{pair}}$, and increasing for even larger SCALEs. At SCALE 2 there are nearly $2^2 = 4$ times the particles at SCALE 1 as specified in Subsection 4.2.1. The factor is slightly less than 4 because the number of actual_particles increases by a factor of 4 while the virtual particle count increases approximately linearly (because the width of virtual particle generation boundary stays unchanged while the width of the tank doubles). As the particle count is 4 time greater at SCALE 2, the iterations count is approximately 16 times greater by Equation 5.15. As with the particle count, the number of pairs are increased by a factor of 4 when the SCALE is incremented to 2. This results in the number of $\frac{\text{iterations}}{\text{pair}}$ increasing by approximately $\frac{16}{4} = 4$ times (from 164 to 630 $\frac{\text{iterations}}{\text{pair}}$)).

Both Figure 5.11 and Figure 5.12 are obtained by measuring walltime for every timestep from 0 to 200 000, sorting by number of pairs and iterations, respectively, and reporting in averages of 500 timesteps. The duration of the simulation is set to 200 000 timesteps because an approximate equilibrium for the fluid in the dam break problem is reached at that point. Executions from timestep 0 to 20 000 will we presented in the rest of this subsection as these, in addition to Figures 5.11 and 5.12, are adequate to see a clear pattern of the total execution time.

Notice that the first data point in Figures 5.11 and 5.12 deviates significantly from the rest of the data points. The first data point in the graph is produced by timestep 0 and is therefore not an average of 500 timesteps. We attribute these and similar deviations on the first data point on graphs in Section 5.1.2.1, to a difference in the measuring method for timestep 0 along with start-up costs, and omit it from further discussion.

Figure 5.13 shows the relationship between execution time of the neighbor loop and the pair count on different number of threads on EPIC, and Figure 5.14 shows the relationship between execution time and iteration count on the same configuration. The linear relationship in Figure 5.14 between iterations and compute time is apparent on EPIC regardless of the number of threads being utilized, confirming the findings of Figure 5.12. The pair count in Figure 5.13 is however not as closely linked to the execution time. The graphs in this figure seem to fluctuate more than the graphs of Figure 5.11, because the fluctuations are greater on lower pair counts as is the case when only executing 20 000 timesteps.

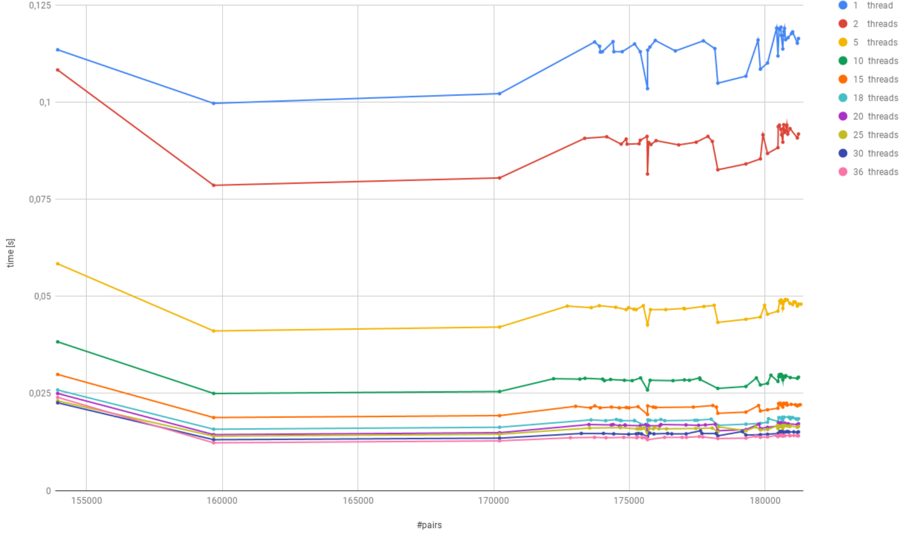


Figure 5.13: The relationship between number of pairs and execution time of the neighbor loop in `find_neighbors` brute-force. Executed on EPIC, 20 000 timestep, SCALE=1.

Note that the size of the particle list is independent of the thread count as no particles are generated or removed as a consequence of the introduction of thread parallelism. However, the number of particles varies slightly between executions of our program because of cumulative floating point errors causing differences in the virtual particle count. These deviations can be seen in Figures 5.13 and 5.14 where the data points, even though they are collected as averages of the same number of timesteps, are not aligned vertically.

Given Figures 5.12 and 5.14 we can identify a linear relationship between the neighbor loop and the iteration count as expressed in Equation 5.16.

$$\mathrm{T}_{\mathrm{neighbor_loop}}(\mathrm{N}_{\mathrm{thread}}, \mathrm{N}_{\mathrm{loop_iter}}) \approx \mathrm{slope}(\mathrm{N}_{\mathrm{thread}}) \cdot \mathrm{N}_{\mathrm{loop_iter}} + \mathrm{base}(\mathrm{N}_{\mathrm{thread}}) \tag{5.16}$$

In Equation 5.16, the term slope is the gradient given as $\frac{time}{loop_iter}$ and base is the intercept when disregarding the first data point. Both slope and base vary with the number of threads utilized.

Figure 5.15 presents the speedup and efficiency calculated for one of the iteration counts in Figure 5.14, chosen by finding an iteration number common for all threads. The speedup on EPIC increases for all 36 threads. Thus, executing on one more thread will result in a decrease in execution time. However, the efficiency graph deflects significantly on 5 threads. This is why the graphs of greater thread counts in Figure 5.14 are clustered closer together. All speedup graphs in Subsection 5.1.2.1 are presented with 1 thread as the baseline.

The similar linear pattern can be observed when executing on Vilje, shown in Figure 5.16. A comparison of EPIC and Vilje for equal thread counts are shown in Figure

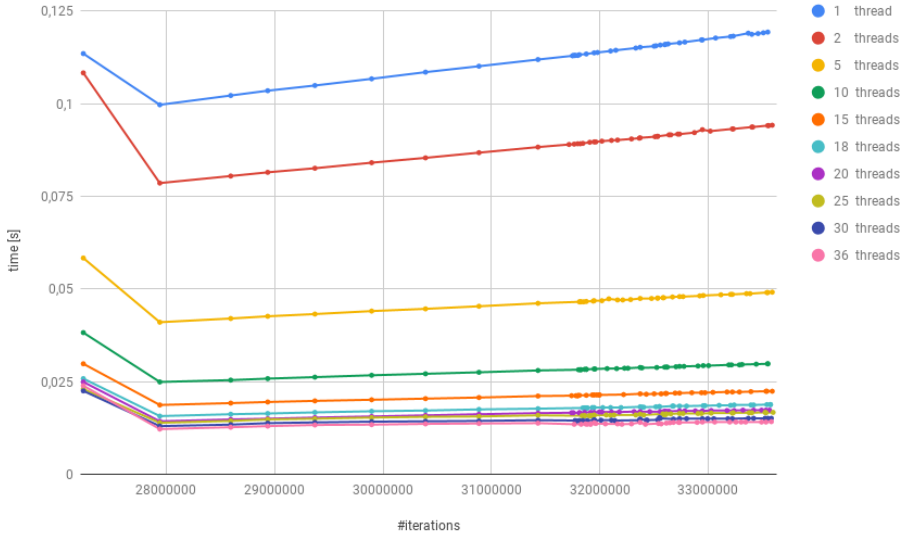
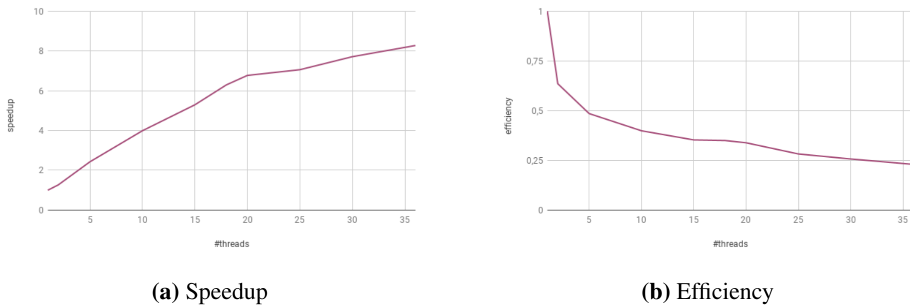


Figure 5.14: The relationship between number of iterations and execution time in `find_neighbors` brute-force. Executed on EPIC, 20 000 timesteps, SCALE=1.



(a) Speedup **(b)** Efficiency

Figure 5.15: Speedup and Efficiency of `find_neighbors` brute-force on EPIC, SCALE=1.

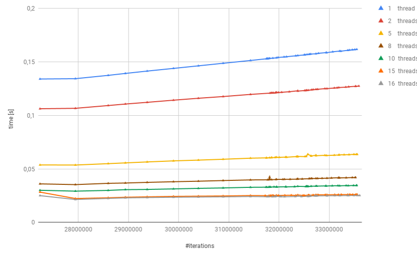


Figure 5.16: The relationship between the thread count and the execution time in `find_neighbors` brute-force on Vilje, SCALE=1 and 20 000 timesteps.

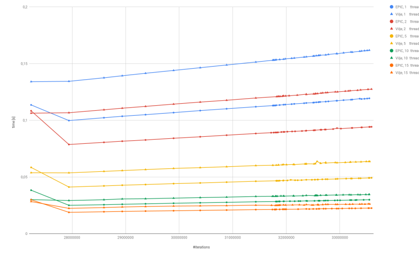
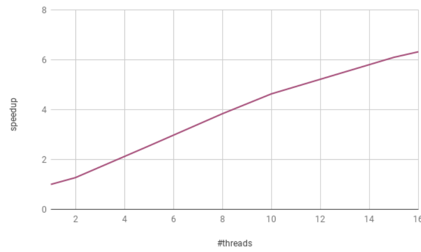
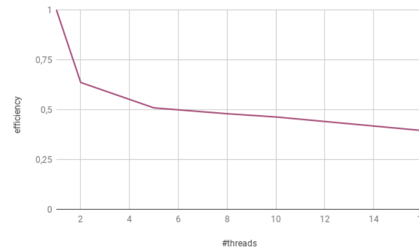


Figure 5.17: Comparison of execution time for various thread counts on Vilje and Epic for `find_neighbors` brute-force, SCALE=1 and 20 000 timesteps.



(a) Speedup



(b) Efficiency

Figure 5.18: Speedup and Efficiency of `find_neighbors` brute-force on Vilje, at SCALE=1.

5.17. The gradient of the curves of both clusters on equal thread counts are approximately similar and decreasing with increasing thread counts. The height of the graph, however, deviates on Vilje and EPIC within the same thread count, but the difference decreases with increasing thread counts. The speedup for Vilje, presented in Figure 5.18a, increases for all threads within a node, while the efficiency in Figure 5.18b has a significant deflection point when running on two threads, and another one on 5 threads.

The linear relationship is likewise visible on EPT, shown in Figure 5.19. A comparison of EPT and EPIC is visualized in Figure 5.20. Figure 5.20 indicates smaller variations in compute time between EPT and EPIC compared to EPT and Vilje. We believe this variation is a result of the processors on EPT and EPIC being nearly equal, while the processors on Vilje are of an older model (cf. Section 4.3.1). The speedup and efficiency on EPT, shown in Figure 5.21, are very similar to the results of Vilje.

Equation 5.16 can be extended as a result of the analysis of Figures 5.17 and 5.20 which demonstrate that the compute time of the neighbor loop varies with the architecture being utilized during program execution. Equation 5.17 describes this extension. The term arch in Equation 5.17 denotes the architecture.

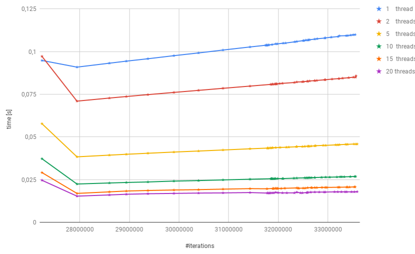


Figure 5.19: The relationship between number of threads and execution time in `find_neighbors` brute-force. Executed on EPT at SCALE=1, 20 000 timesteps.

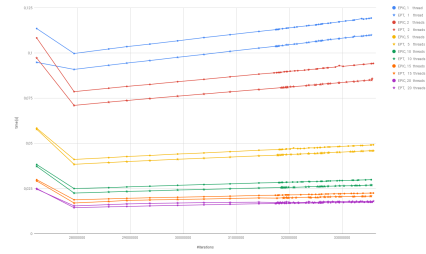
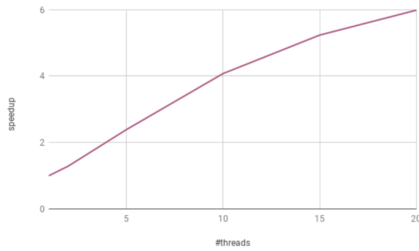
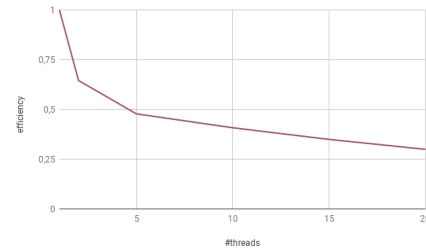


Figure 5.20: Comparison of execution time on various thread counts on EPT and Epic for `find_neighbors` brute-force, SCALE=1 and 20 2000 timesteps.

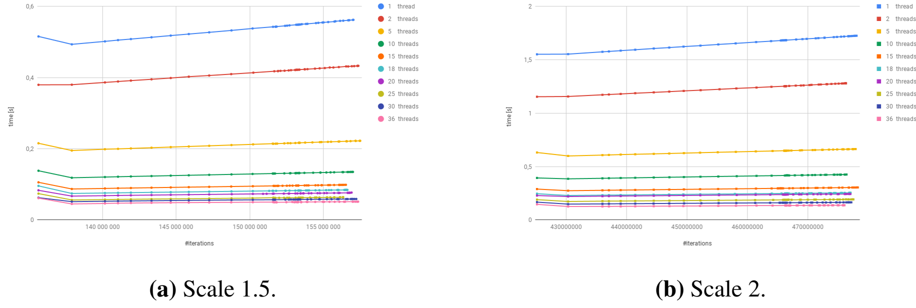


(a) Speedup



(b) Efficiency

Figure 5.21: Speedup and Efficiency of `find_neighbors` brute-force on EPT, on SCALE=1.



(a) Scale 1.5. **(b)** Scale 2.

Figure 5.22: The execution time of `find_neighbors` brute-force as the iterations count increases at SCALE=1.2 and SCALE=2, EPIC.

$$T_{\text{neighbor_loop}}(N_{\text{thread}}, \text{arch}, N_{\text{loop_iter}}) \approx \text{slope}(N_{\text{thread}}) \cdot N_{\text{loop_iter}} + \text{base}(N_{\text{thread}}, \text{arch}) \quad (5.17)$$

The execution time on EPIC at SCALE 1.5 and SCALE 2 are shown in Figure 5.22, and presented along with SCALE 1 in Figure 5.23. The height of the graphs at SCALE 1.5 and SCALE 2 are greater than those of SCALE 1 (for the same thread count) by a factor of $(1.5^2)^2 \approx 5$ and $(2^2)^2 = 16$, respectively, because the number of particles are approximately SCALE^2 times the particle count at SCALE 1. Following the graphs of same color, we observe that the gradients are approximately equal, while the length increases with increasing SCALE as the virtual particle counts varies to a greater extent with increasing SCALE. Hence, the compute time of the neighbor loop on SCALEs other than 1 can be predicted by the results from SCALE 1 (on the same thread count), together with the iteration count on the specific SCALE as Equation 5.18 describes. This makes it possible to predict the behaviour without executing the program on greater SCALEs. The speedup and efficiency graphs at SCALE 1.5 and SCALE 2 are shown in Figure 5.24 and 5.25, respectively.

$$T_{\text{neighbor_loop}}(N_{\text{thread}}, \text{arch}, N_{\text{loop_iter}}) \approx \text{slope}_{\text{SCALE1}}(N_{\text{thread}}) \cdot N_{\text{loop_iter}} + \text{base}_{\text{SCALE1}}(N_{\text{thread}}, \text{arch}) \quad (5.18)$$

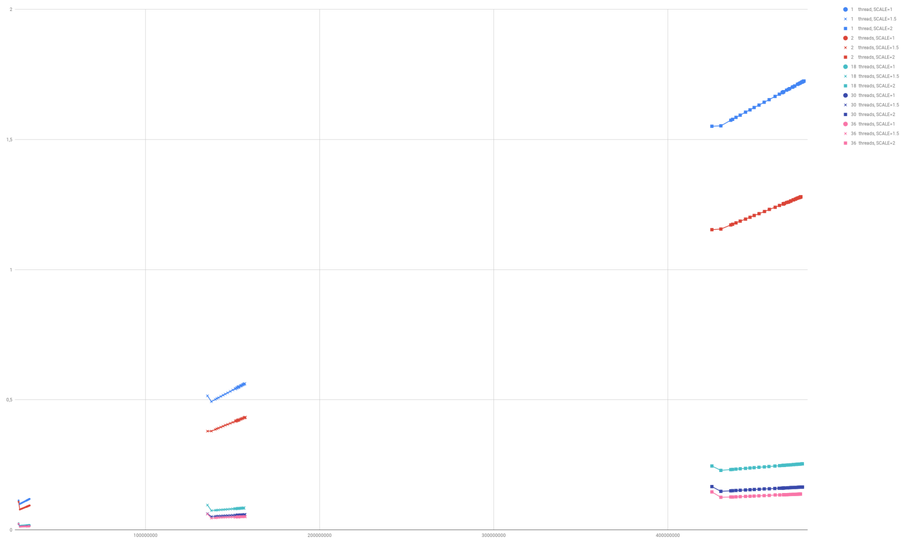
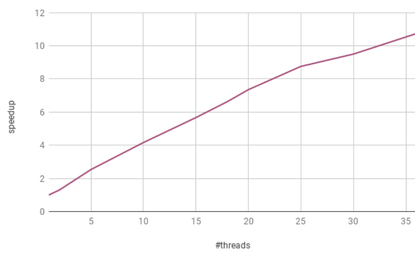
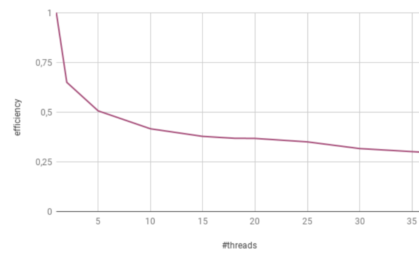


Figure 5.23: Comparison of the execution time of brute-force `find_neighbors` at various SCALEs on EPIC, 20 2000 timesteps.

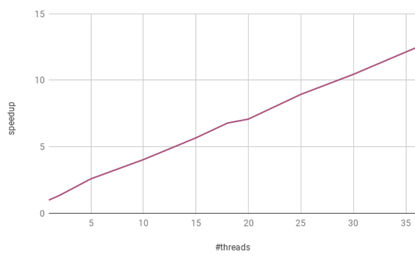


(a) Speedup

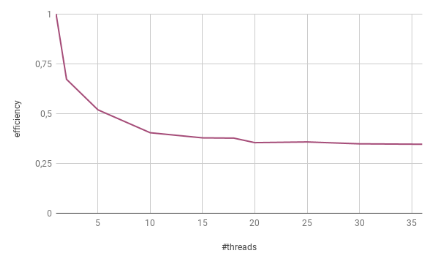


(b) Efficiency

Figure 5.24: Speedup and Efficiency of brute-force `find_neighbors` on EPIC, at SCALE=1.5.

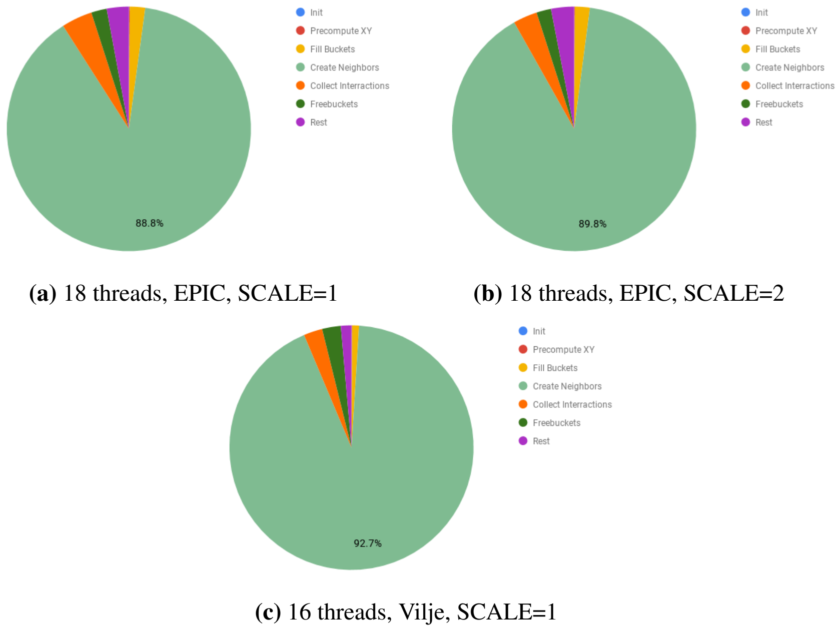


(a) Speedup



(b) Efficiency

Figure 5.25: Speedup and Efficiency of brute-force `find_neighbors` on EPIC, at SCALE=2.



(a) 18 threads, EPIC, SCALE=1

(b) 18 threads, EPIC, SCALE=2

(c) 16 threads, Vilje, SCALE=1

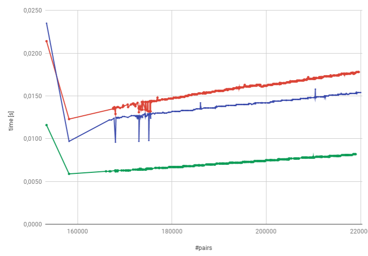
Figure 5.26: Distribution in execution time of different parts of the `find_neighbors` method (cell-linked particle list)

Cell-Linked List for Particles Profiling results of the `find_neighbors` method of cell-linked particle lists are shown in Figures 5.26a and 5.26b for EPIC at SCALE 1 and 2, respectively, and Figure 5.26c for Vilje at SCALE 1. The figures show that the `create_neighbors`[1] method dominates the execution time of `find_neighbors` for all three executions. The share increases slightly, from 88.8% to 89.8%, when the SCALE is incremented from 1 to 2 on EPIC. On one node on Vilje at SCALE 1, this number is 92.7%. As a result of Figure 5.26, we consider `create_neighbors` as the only bottleneck of the `find_neighbors` method and omit other components of the method from further discussion. This relationship is given in Equation 5.19.

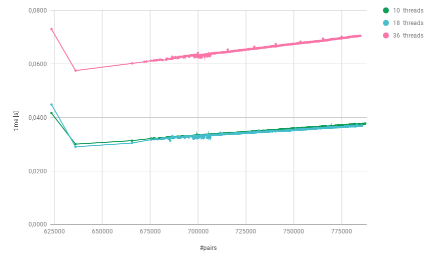
$$T_{\text{find_neighbors}} \approx T_{\text{create_pairs}} \tag{5.19}$$

The `create_neighbors` method iterates through all particles and for each of them examines the neighboring buckets (cf. Section 4.2.2.1) for proximity. With small bucket sizes (set approximately to the interaction radius × interaction radius in our implementation), the number of particles inspected for closeness is remarkably decreased compared to the brute-force implementation. Because the probability of detecting a pair is increased, the impact of pairs will be more prominent on the execution time. In fact, a pair is found for every 2.7 proximity check performed at SCALE 1 (2.7 is the median over all timesteps). At SCALE 2, there is a hit for every 2.8 check. The small difference in the hit rate on

[1] We use the terms `create_neighbors` and `create_pairs` interchangeably.

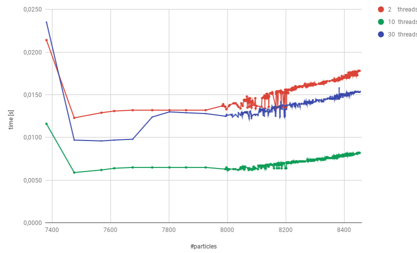


(a) SCALE=1

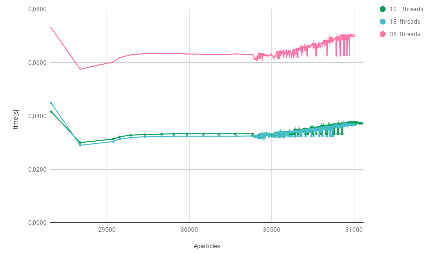


(b) SCALE=2

Figure 5.27: The relationship between number of pairs and the execution time of the `create_pair` method in the cell-linked particle implementation, 200 000 timesteps, EPIC.



(a) SCALE=1



(b) SCALE=2

Figure 5.28: The relationship between number of particles and the execution time of the `create_pair` method in the cell-linked particle implementation, 200 000 timesteps, EPIC.

SCALE 1 and SCALE 2 may be caused by changes in virtual particles and pairs between executions or by measuring uncertainty. As the bucket size is kept constant while increasing the SCALE, the number particles examined for closeness is constant if the particle density within the buckets is approximately unchanged. Nevertheless, the probability of detecting a pair is much higher than for brute-force. Figure 5.27 shows how the execution time is effected by the number of pairs when executing the program until timestep 200 000.

Apart from some fluctuations on lower pair counts at SCALE 1 (see Figure 5.27a), the graphs of all thread counts in Figure 5.27 are approximately linear. The particle count, however, seen in Figure 5.28, is not as dominating as pairs in terms of the execution time.

Figure 5.29 presents how the execution time behaves as the pair count is increased for various thread counts. The figure shows that the execution time remains approximately constant when the pair count increases. However, the graph does only present the execution times for the lower end of the x-axis in Figure 5.27, and the linearly increasing pattern is therefore is not visible. An interesting remark in Figures 5.29 and 5.27 is that execution time is the lowest with 10 threads. This observation can also be seen in Figure 5.30 showing the speedup and efficiency graphs for a cross-section of Figure 5.29. Increasing the thread count above 10 threads will lead to an increase in the overall execution time

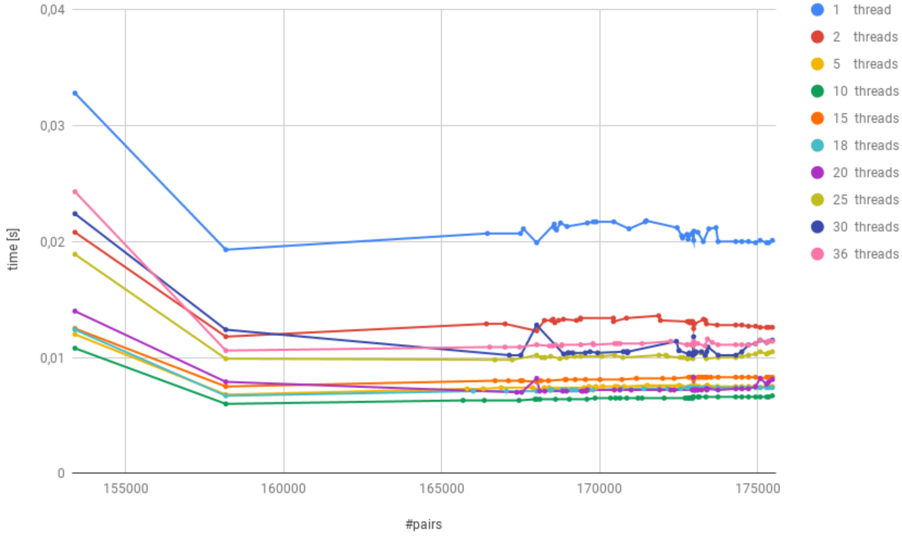


Figure 5.29: The relationship between number of pairs and the execution time of the `create_pair` method in `find_neighbors` cell-linked particles. Executed on EPIC, 20 000 timestep, SCALE=1.

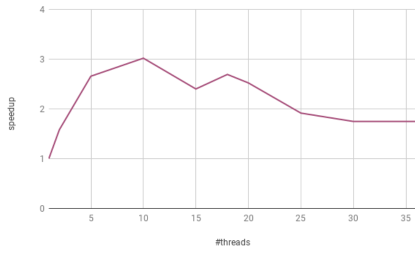
of `create_pairs`. Note that the standard deviation in the data points presented in this section (out of 30 sample executions) is of order 10^{-3}. Thus, the execution times of the graphs clustered together within this deviation may vary from execution to execution.

Given Figure 5.29, the compute time of the `create_pairs` method can be modeled as in Equation 5.20.

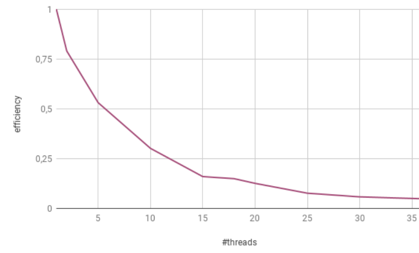
$$T_{\text{create_pairs}}(N_{\text{thread}}, N_{\text{pairs}}) \approx \text{slope}(N_{\text{thread}}) \cdot N_{\text{pairs}} + \text{base}(N_{\text{thread}}) \tag{5.20}$$

Figure 5.31 shows the relationship between pairs and compute time on Vilje, and Figure 5.32 compares the result with EPIC on equal thread counts. In Figure 5.31 the linearly increasing pattern observed in Figure 5.27 is visible. In addition, the gradient is decreasing as the thread count increases. The impact of utilizing processors of an older model is visible in Figure 5.32 as was the case with brute_force. The speedup and efficiency on Vilje is presented in Figure 5.33. The global maximum on the speedup in Figure 5.33a is, as with EPIC, reached when executing on 10 threads. In addition, the efficiency on both EPIC and Vilje drop significantly as the thread count is increased.

The linear, slightly increasing pattern is also detectable on EPT. This is shown in Figure 5.34, and shown together with EPIC in Figure 5.35. The maximum speedup when executing on 10 threads, and the dropping efficiency is also present on EPT as shown in Figure 5.36. The execution on EPT and Vilje show that the execution time vary with the architecture being utilized. The height of the graphs are different for each architecture while the gradient is approximately the same. This is described in Equation 5.21.



(a) Speedup



(b) Efficiency

Figure 5.30: Speedup and Efficiency of `find_neighbors` cell-linked particle lists on EPIC, cell-linked particle list, SCALE=1.

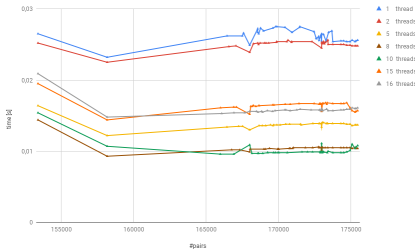


Figure 5.31: The relationship between the thread count and the execution time in `create_pairs` on Vilje, SCALE=1 and 20 000 timesteps.

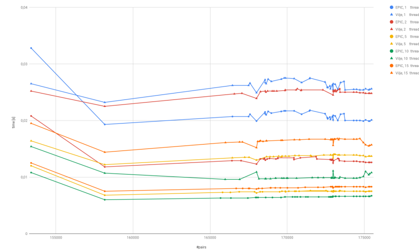
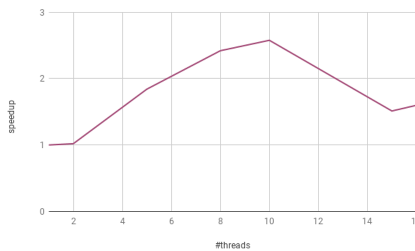
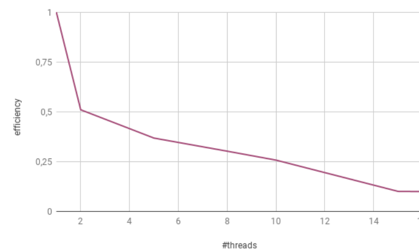


Figure 5.32: Comparison of execution time for various thread counts on Vilje and Epic for `create_pairs`, SCALE=1 and 20 000 timesteps.



(a) Speedup



(b) Efficiency

Figure 5.33: Speedup and Efficiency of `find_neighbors` cell-linked particle lists on Vilje, at SCALE=1.

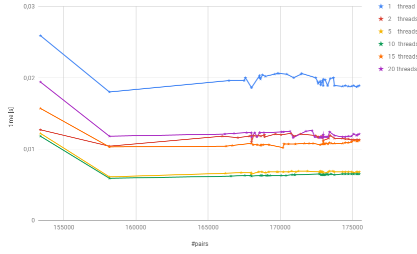


Figure 5.34: The relationship between the thread count and the execution time in `create_pairs` on EPT, SCALE=1 and 20 000 timesteps.

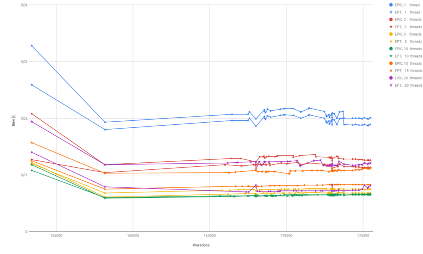
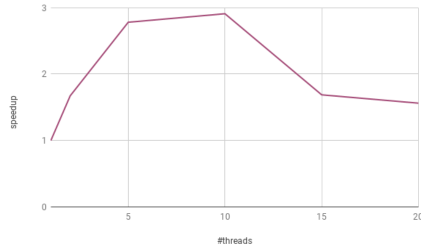
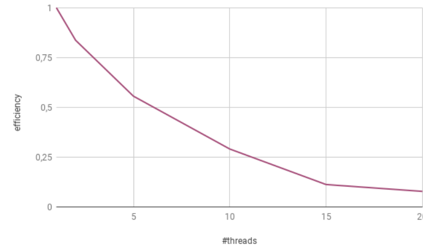


Figure 5.35: Comparison of execution time for various thread counts on EPT and Epic for `create_pairs`, SCALE=1 and 20 000 timesteps.



(a) Speedup



(b) Efficiency

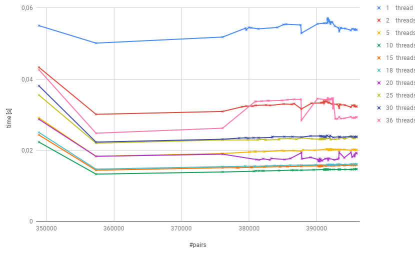
Figure 5.36: Speedup and Efficiency of `find_neighbors` cell-linked particle lists on EPT, at SCALE=1.

$$T_{\text{create_pairs}}(N_{\text{thread}}, \text{arch}, N_{\text{pairs}}) \approx \text{slope}(N_{\text{thread}}) \cdot N_{\text{pairs}} + \text{base}(N_{\text{thread}}, \text{arch}) \quad (5.21)$$

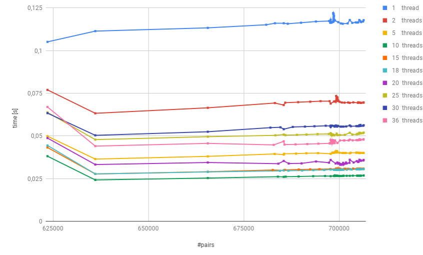
Figure 5.37 shows the execution time at SCALE 1.5 and SCALE 2. As previously discussed, the ratio between a check for proximity and a detection of a pair is approximately constant from one SCALE to the next if the particle density inside a bucket is nearly constant. As a result, the maximum of speedup graph at Scale 1.5 and SCALE 2 shown in Figures 5.38a and 5.39a, respectively, remain at 10 threads.

A comparison of different SCALEs shown in Figure 5.40 shows that a regression line at SCALE 1 can be sufficient in order to describe the execution time of SCALEs greater than 1. By Figure 5.27 the linearity of the graphs will be more evident as more timesteps are executed and more pair counts with respective execution times are obtained. Equation 5.22 captures this relationship.

$$T_{\text{create_pairs}}(N_{\text{thread}}, \text{arch}, N_{\text{pairs}}) \approx \text{slope}_{\text{SCALE1}}(N_{\text{thread}}) \cdot N_{\text{pairs}} + \text{base}_{\text{SCALE1}}(N_{\text{thread}}, \text{arch}) \quad (5.22)$$

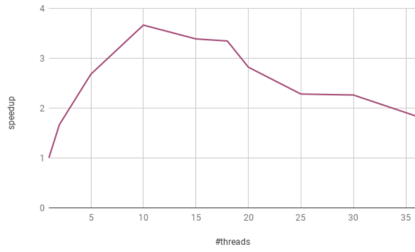


(a) Scale 1.5.

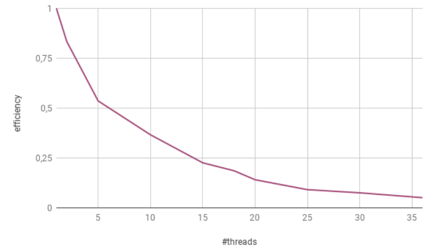


(b) Scale 2.

Figure 5.37: The execution time of `find_neighbors` cell-linked particle lists as the iterations count increases at SCALE=1.2 and SCALE=2, EPIC.

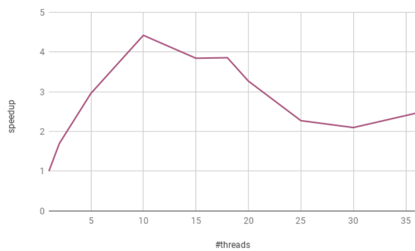


(a) Speedup

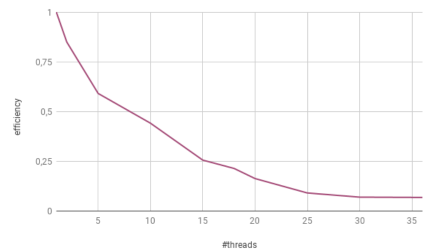


(b) Efficiency

Figure 5.38: Speedup and Efficiency of `find_neighbors` cell-linked particle lists, at SCALE=1.5.



(a) Speedup



(b) Efficiency

Figure 5.39: Speedup and Efficiency of `find_neighbors` cell-linked particle lists on EPIC, on SCALE=2.

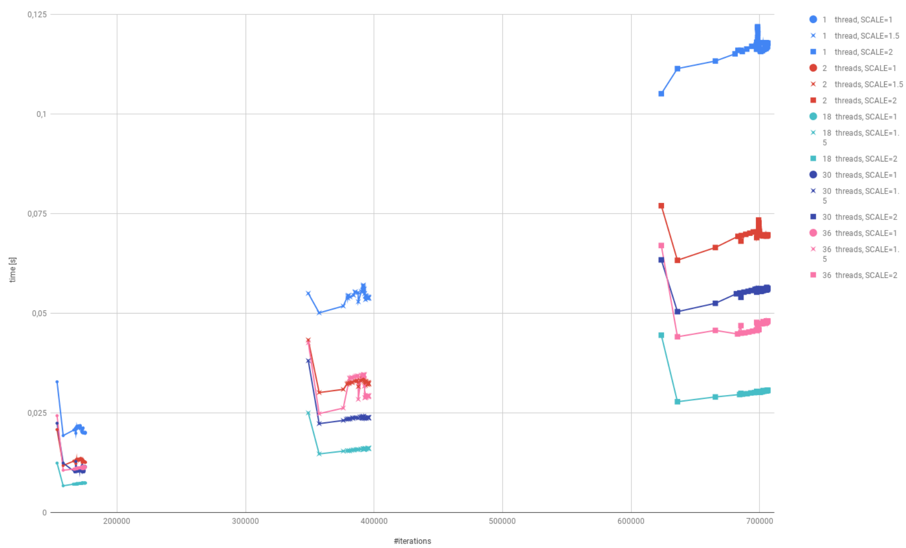
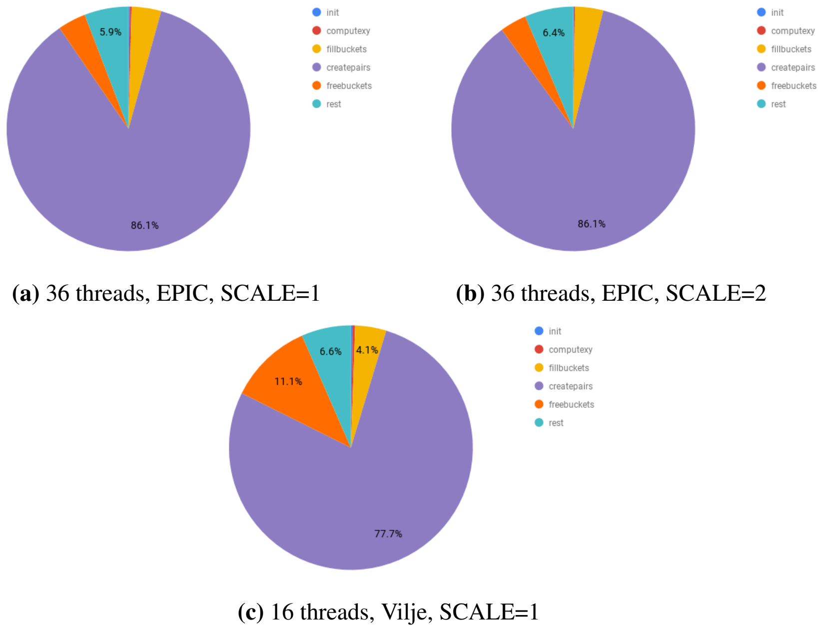


Figure 5.40: Comparison of the execution time of `create_pairs` in `find_neighbors` cell-linked particle lists at various SCALEs on EPIC, 20 2000 timesteps.



(a) 36 threads, EPIC, SCALE=1

(b) 36 threads, EPIC, SCALE=2

(c) 16 threads, Vilje, SCALE=1

Figure 5.41: Distribution in execution time of different parts of the `find_neighbors` method (cell-linked pair and particle lists)

Cell-Linked List for Particles and Pairs Figure 5.41 shows the profiling results of the `find_neighbors` method for cell-linked pair and particle lists. The figure show that the `create_pair` subroutine dominates the execution time in `find_neighbors`. In addition, `create_pairs`'s share of the compute time is equal (86.1%) at both SCALE 1 and SCALE 2 on 36 threads on EPIC, as shown in Figures 5.41a and 5.41b, respectively. On 16 threads on Vilje, the share is smaller (77.7%). However, the `create_pairs` subroutine dominates the compute time in all three cases and is therefore analyzed further in this section. Equation 5.23 presents this relationship.

$$T_{\text{find_neighbors}} \approx T_{\text{create_pairs}} \tag{5.23}$$

In the `find_neighbors` method of the cell-linked pair and particle list implementation, the critical section present in the cell-linked particle list implementation is avoided (cf. Section 4.2.2.1). When a thread detects a pair, no other thread will do computation that may interrupt this thread from creating the pair. Hence, the time spent inside the body of the if statement in Line 47 of Listing A.1 is decreased compared to the cell-linked particle method. However, in order to avoid any critical sections, all pairs across buckets are duplicated. This increases a thread's probability of executing the body of the if statement. In addition, the proximity check is carried out for all particles in all buckets. Thus, the compute time of the `create_pairs` subroutine is dependent on the particle count, the real pair count and the created pair count (real pairs and duplicates).

Figure 5.42 shows how the compute time varies with the number of particles. The

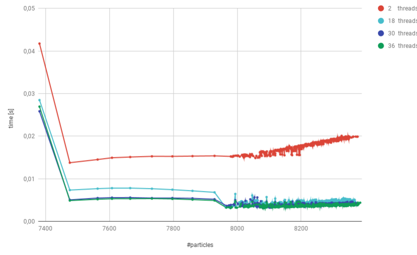
execution time is greater on lower particle counts, which occur in the beginning on an execution (that is, on low timesteps), as a result of start-up costs. As the timestep increases, the graphs stabilize. In fact, the the graphs of higher thread counts can be treated as constants after the effect of the start-up cost is invisible, which means that adding particles within a given SCALE does not increase the execution time significantly. In the brute-force implementation discussed previously, adding a particle means that all other particles in the domain have to be compared with this particle. Hence, the execution time increases by the square of the particle count. In both cell-linked implementations, proximity checks are solely performed between particles within neighboring buckets. However, in cell-linked particles, when a pair is detected on higher thread counts, contention on the atomically updated pair count increases significantly, ”wasting” computation power. When the thread count in the cell-linked particle and pair lists implementation is increased, however, the contention on the pair counter is not present and the threads can ”maximize” their computation power. Note that the execution time for each particle count is of a smaller magnitude than the ones of the other two implementations. This and other factors may contribute to the more visible fluctuations in Figures 5.42a and 5.42b.

Figures 5.44 and 5.43 show how the execution time varies with the number of real pairs (henceforth referred to as actual pairs) and created pairs, respectively, during 200 000 timesteps. Similar to Figure 5.42, all graphs except the one utilizing two threads reach an approximately constant value as the timestep increases. The graphs in both Figures 5.44 and 5.43 have a similar stabilizing pattern and are therefore equally good candidates for the prediction of compute time. However, the number of particles is easier to retrieve than the number of pairs as the pairs are organized in buckets, exclusively, while the particles are located in a one dimensional array in addition to buckets. Furthermore, the smooth out effect of the particle graph is visible on as little as 20 000 timeteps, while the same amount of timesteps in both of the pair graphs show no such effect. Therefore, we choose the particle count to describe the computation time of the `create_pairs` subroutine. Note that the approximate iteration count could have been used to predict the computation time. Each particle is compared to all particles in the 9 neighboring buckets. Hence, the iteration count could be described as the number of buckets times the average number of particles in each bucket. This way, the bucket size would be included in the model. However, the effect of changing the bucket size is not examined in this thesis and is therefore omitted from further discussion.

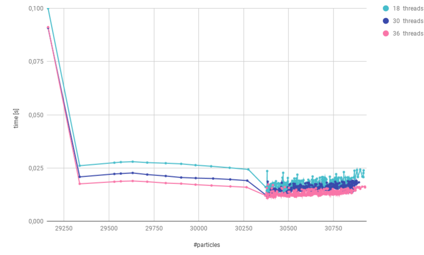
The execution time as the number of particles is increased on EPIC at SCALE 1 is presented in Figure 5.45. The figure shows that all graphs are clustered together and smoothed out as the number of particles per timestep increases. The compute time on each thread count can therefore be approximated to a constant as given in Equation 5.24. In Equation 5.24, the term *base* is constant for a given thread count at SCALE 1. Note that the equation omits the linearly increasing compute time for lower thread counts and considers all computation times to be constant. This choice was made in order to simplify the model.

$$\mathrm{T}_{\mathrm{create_pairs}}(\mathrm{N}_{\mathrm{threads}}) \approx \mathrm{base}(\mathrm{N}_{\mathrm{threads}}) \tag{5.24}$$

The speedup and efficiency on EPIC at SCALE 1 is visualized in Figure 5.46. The speedup shown in Figure 5.46a is increasing for most thread counts. Hence, the compute

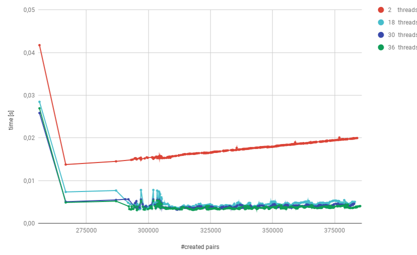


(a) SCALE=1

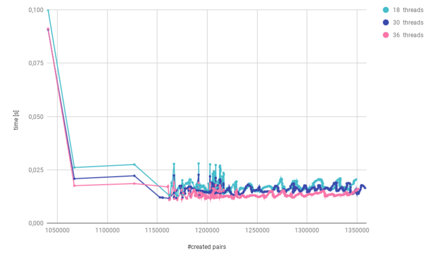


(b) SCALE=2

Figure 5.42: The relationship between number of particles and the execution time of the `create_pair` method in the cell-linked pair and particle list implementation, 200 000 timesteps, EPIC.

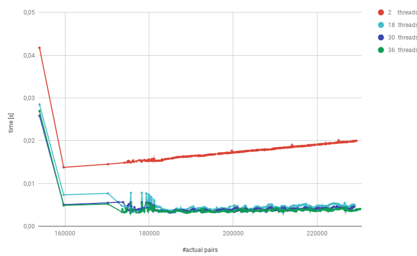


(a) SCALE=1

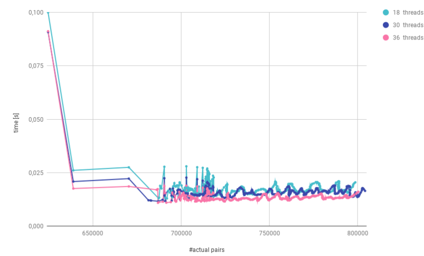


(b) SCALE=2

Figure 5.43: The relationship between number of created pairs and the execution time of the `create_pair` method in the cell-linked pair and particle list implementation, 200 000 timesteps, EPIC.



(a) SCALE=1



(b) SCALE=2

Figure 5.44: The relationship between number of actual pairs and the execution time of the `create_pair` method in the cell-linked pair and particle lists implementation, 200 000 timesteps, EPIC.

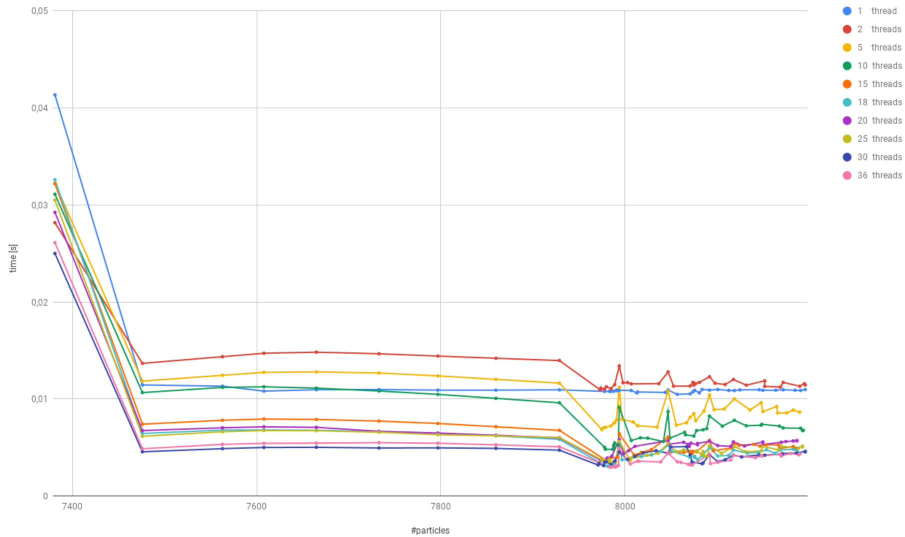
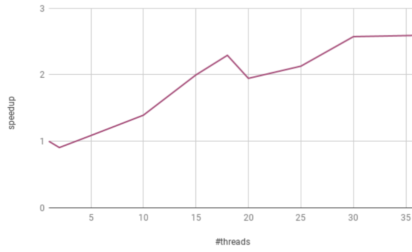


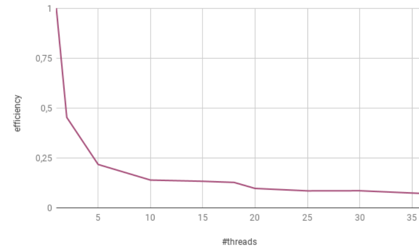
Figure 5.45: The relationship between number of pairs and the execution time of the `create_pair` method in `find_neighbors` cell-linked particle and pair lists. Executed on EPIC, 20 000 timestep, SCALE=1.

time will decrease as the number of threads is increased within a node on EPIC. This is not the case for the previously discussed cell-linked particle implementation. We attribute this effect to the critical section updating the number of pairs being removed. Furthermore, the gradient of the speedup graph is low, which may indicate that the subroutine is memory bound. Note that there is a local maximum on 18 threads, which may be a result of measurement uncertainty as the standard deviation is 0.001 (drawn out of a sample size of 30). An complete analysis of this effect will however be omitted from further discussion.

Figures 5.47 and 5.48 show the compute time as a function of the particle count on Vilje, and a comparison between Vilje and EPIC, respectively. As on EPIC, the tail of the graphs on Vilje are approximately constant. The comparison shows that the compute time on Vilje is greater than EPIC for lower threads counts, and lower than EPIC for greater thread counts. These differences may be cause by measurement uncertainties as mentioned above. The fluctuations, however, are mostly similar. The speedup and efficiency of the `create_pairs` subroutine on Vilje at SCALE 1 is shown in Figure 5.49. The speedup in Figure 5.49a increases at a greater rate than the one of EPIC in Figure 5.46a. In addition, the speedup increases as the thread count is increased. The compute time on EPT is presented in Figure 5.50 and combined with the compute time on EPIC in 5.51. Figure 5.51 indicates that the computation time on EPT is slightly greater than on EPIC on equal thread counts. The speedup and efficiency on EPT is shown in 5.52. The speedup curve in Figure 5.52a drops at 2 threads. This effect is also seen in the speedup graph of EPIC in Figure 5.46a. We attribute this effect to increased overhead as a result of the introduction of threads (cf. Section 3.4.3.2). Furthermore, the speedup decreases slightly at 20 threads.



(a) Speedup



(b) Efficiency

Figure 5.46: Speedup and Efficiency on EPIC, cell-linked particle and pair lists, SCALE=1.

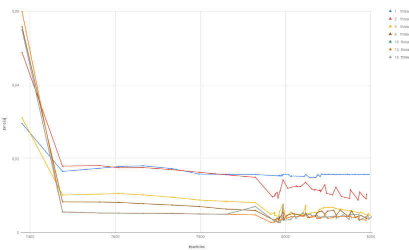


Figure 5.47: The relationship between the thread count and the execution time in `create_pairs` in `find_neighbors` cell-linked particle and pair lists on Vilje, SCALE=1 and 20 000 timesteps.

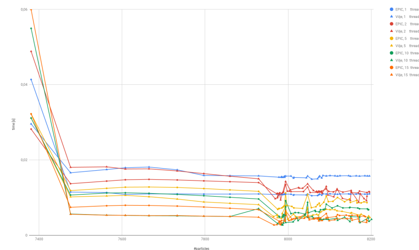


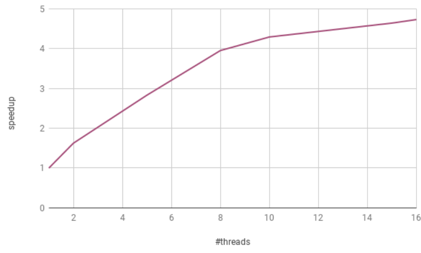
Figure 5.48: Comparison of execution time for various thread counts on Vilje and Epic for `create_pairs` in `find_neighbors` cell-linked particle and pair lists, SCALE=1 and 20 000 timesteps.

However, as we have not seen this effect on either Vilje or the more similar in terms of processor type, EPIC, we attribute this deviation to measuring uncertainty. An Equation combining these findings is expressed in Equation 5.25.

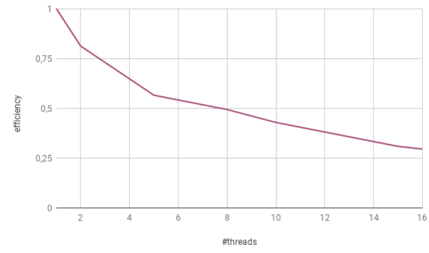
$$T_{\text{create_pairs}}(N_{\text{threads}}, \text{arch}) \approx \text{base}(N_{\text{threads}}, \text{arch}) \tag{5.25}$$

Equation 5.25 estimates the compute time of `create_pairs` to be approximately equal to a constant given by the thread count and the architecture.

The compute times at SCALE 1.5 and SCALE 2 on EPIC are shown in Figure 5.53. The respective speedup and efficiency graphs are shown in Figure 5.54 and 5.55. These figures correspond with the smooth out effect observed at SCALE 1. A comparison at SCALE 1, 1.5 and 2 is illustrated in Figure 5.56. The figure shows that the execution time increases proportionally to the number of particles at different SCALEs. Hence, the execution time increases approximately by a factor of SCALE^2 compared to SCALE 1. Note that Figure 5.56 shows that the execution time varies with the number of particles. However, we estimate the execution time on a specific SCALE as a constant. As the number of particles across SCALEs vary to a much larger degree compared to within the same SCALE (i.e. variations in the number of virtual particles), this effect can be



(a) Speedup



(b) Efficiency

Figure 5.49: Speedup and Efficiency of `find_neighbors` cell-linked pair and particle lists on Vilje, at SCALE=1.

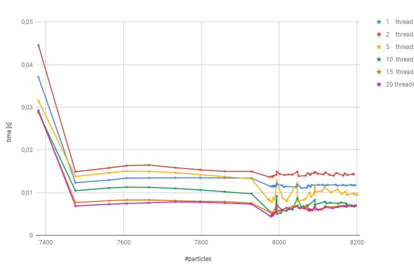


Figure 5.50: The relationship between the thread count and the execution time in `create_pairs` on EPT, SCALE=1 and 20 000 timesteps.

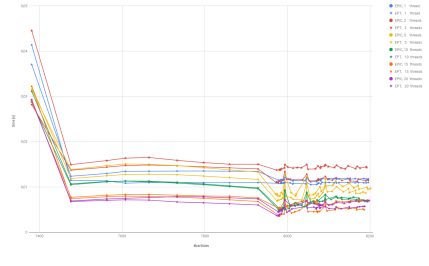
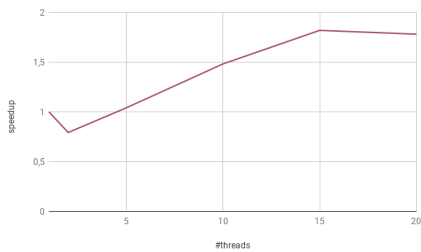
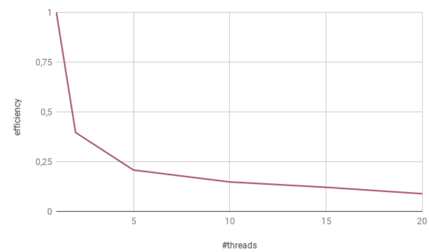


Figure 5.51: Comparison of execution time for various thread counts on EPT and Epic for for `create_pairs` in `find_neighbors` cell-linked particle and pair lists, SCALE=1 and 20 000 timesteps.

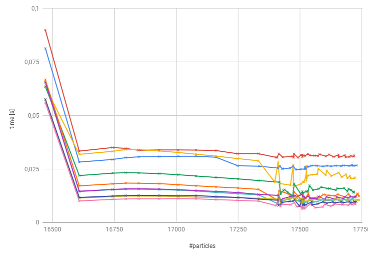


(a) Speedup

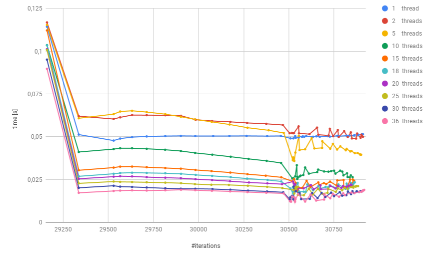


(b) Efficiency

Figure 5.52: Speedup and Efficiency for `create_pairs` in `find_neighbors` cell-linked particle and pair lists on EPT, on SCALE=1.

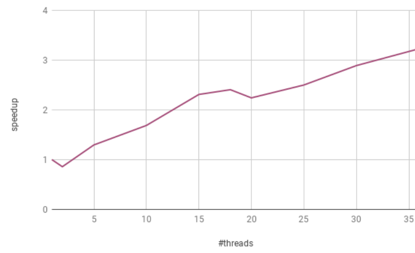


(a) Scale 1.5.

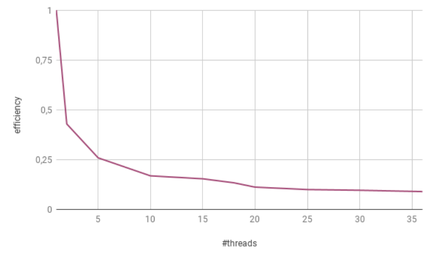


(b) Scale 2.

Figure 5.53: The execution time of `find_neighbors` cell-linked particle and pair lists as the iterations count increases at SCALE=1.2 and SCALE=2, EPIC.



(a) Speedup

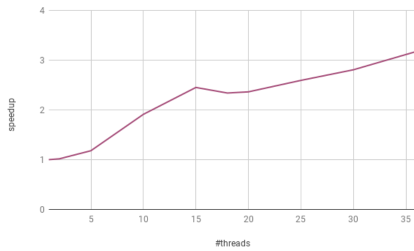


(b) Efficiency

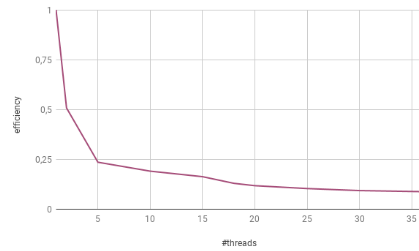
Figure 5.54: Speedup and Efficiency of `find_neighbors` cell-linked particle and pair lists on EPIC, at SCALE=1.5.

disregarded. Equation 5.26 reevaluates Equation 5.25 by including SCALE.

$$T_{\text{create_pairs}}(N_{\text{threads}}, \text{arch}, \text{SCALE}) \approx \text{base}_{\text{SCALE1}}(N_{\text{threads}}, \text{arch}) \cdot \text{SCALE}^2 \quad (5.26)$$



(a) Speedup



(b) Efficiency

Figure 5.55: Speedup and Efficiency of `find_neighbors` cell-linked particle and pair lists on EPIC, on SCALE=2.

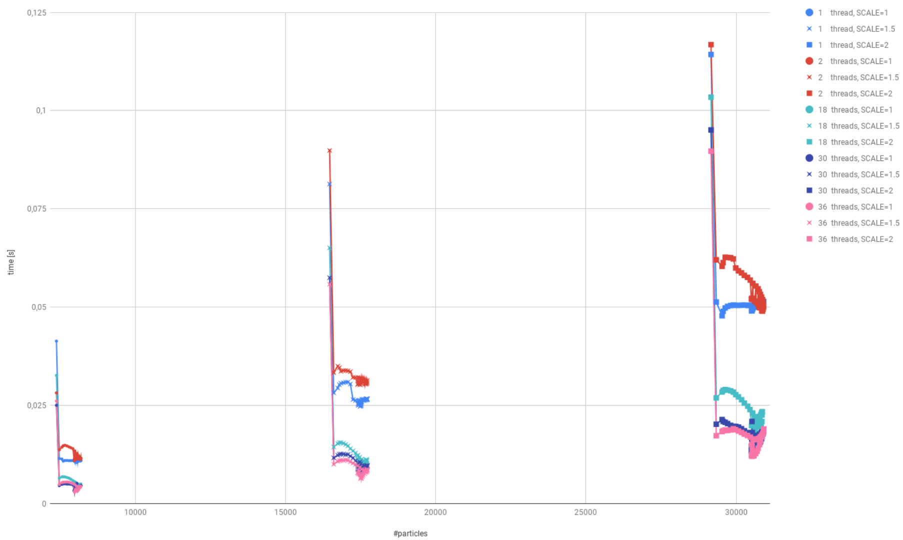


Figure 5.56: Comparison of the execution time of `create_pairs` in `find_neighbors` cell-linked particle and pair lists at various SCALEs on EPIC, 20 2000 timesteps.

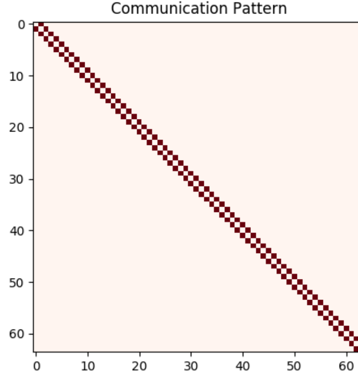


Figure 5.57: The communication pattern of the SPH program. A red square indicates exchanged between the corresponding ranks.

5.1.2.2 Inter Node Parallelism

In this section we will discuss the distributed memory parallelism of the SPH program. Contrary to the tripartite intra node discussion of Section 5.1.2.1, this section is relevant for all three implementations of the program, because the communication pattern and the number of bytes communicated for each of the variations are the same.

There are two methods in the SPH program were communication is carried out; the `border_exchange` method and the `migrate_particles` method (see Section 4.2.2). In both methods, the data units being exchanged are particles. The total communication time is a combination of the communication time of these two methods as captured in Equation 5.27. Both `border_exchange` and `migrate_particles` have the same, one dimensional, near neighbor communication pattern shown in Figure 5.57. As each rank only sends data to and receives data from its neighbor rank in east and west, given by Equation 5.28, the communication pattern is clustered on both sides of the diagonal in Figure 5.57.

$$\mathrm{T_{communication}} = \mathrm{T_{migrate_particles}} + \mathrm{T_{border_exchange}} \tag{5.27}$$

$$\mathrm{neighbor(rank)_{east}} = (\mathrm{rank} + 1) \mod \mathrm{N} \tag{5.28}$$

$$\mathrm{neighbor(rank)_{west}} = (\mathrm{rank} + \mathrm{N_{rank}} - 1) \mod \mathrm{N} \tag{5.29}$$

The communication is performed in two stages during the execution of `border_exchange` and `migrate_particles`. In the first stage, each rank transfers an integer indicating the number of particles that is going to be transferred during the next stage. In the second stage, the particles are transferred (cf. Section 4.2.2). This is described in Equations 5.30 and 5.31. The term bexchange in Equation 5.30 denotes border_exchange. The terms border_exchange_count and border_exchange_the_particles denotes the first and second stage of the border_exchange, respectively.

$$T_{\text{border_exchange}} = T_{\text{bexchange_count}} + T_{\text{bexchange_the_particles}} \tag{5.30}$$

$$T_{\text{migrate_particles}} = T_{\text{migrate_count}} + T_{\text{migrate_the_particles}} \tag{5.31}$$

All ranks communicate concurrently during a data exchange phase. Hence, the rank performing the most time consuming communication determines the phase's communication time. The sends and receives are carried out by the MPI library function `MPI_Sendrecv`, described in Section 3.2.1. In our MPI implementation, the communication time of one `MPI_Sendrecv` corresponds to the communication time of one synchronized send (`MPI_Ssend`). However, depending on the implementation, the `MPI_Sendrecv` construct may correspond to one or two synchronized sends. Thus, the communication time of each stage of `border_exchange` and `migrate_particles` can be described as in Equations 5.32 and 5.33, respectively.

$$T_{\text{bexchange_stage}} \approx s \cdot \max_{rank \in \mathrm{N}} (T(\text{rank},\text{neighbor(rank)}_{\text{east}}) + T(\text{rank},\text{neighbor(rank)}_{\text{west}}))) \tag{5.32}$$

$$T_{\text{migrate_stage}} \approx s \cdot \max_{rank \in \mathrm{N}} (T(\text{rank},\text{neighbor(rank)}_{\text{east}}) + T(\text{rank},\text{neighbor(rank)}_{\text{west}}))) \tag{5.33}$$

The term N in both Equations 5.32 and 5.33 is defined as the number of ranks. The term s is captured in Equation 5.34, and the term stage is either the particle count transfer or the particle transfer (i.e. stage $\in \{\text{count}, \text{the_particles}\}$). Hence, $T(\text{rank},\text{neighbor(rank)}_{\text{east/west}}$ is defined in terms of one synchronized send.

$$s = \begin{cases} 1 & \text{if } T_{\text{Sendrecv}} = T_{\text{Ssend}} \\ 2 & \text{otherwise.} \end{cases} \tag{5.34}$$

The Heterogeneous Hockney model, described in Section 3.4.1.1, can be used to determine the communication time of a data transfer in terms of latency (α) and bandwidth (β). The number of bytes exchanged, is the product of the particle count being exchanged and the size of a particle, as given by Equation 5.35. The exchange time between neighboring ranks, introduced in Equations 5.32 and 5.33, is described in Equation 5.36 by applying the Heterogeneous Hockey model on data transfers.

$$N_{\text{bytes}} = N_{\text{data}} \cdot \text{bytes}_{\text{data}} \tag{5.35}$$

$$T(\text{rank},\text{neighbor(rank)}_{\text{east/west}}) \approx \alpha(\text{rank},\text{neighbor(rank)}_{\text{east/west}}) + N_{\text{bytes}} \cdot \beta(\text{rank},\text{neighbor(r)}_{\text{east/west}}) \tag{5.36}$$

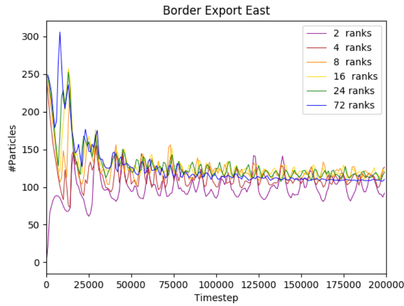
In the first stage of the communication (i.e. the count transfer), a `MPI_LONG` is transferred. The size of a `MPI_LONG` is on most MPI standards equal to 4B. The size of the particles sent during the second stage varies between 160B in the brute-force implementation, and 184B in the two cell-linked list versions. The increase in size for the cell-linked

implementations is due to the introduction of bucket indices defining the x and y coordinates of the bucket that the particle resides in.

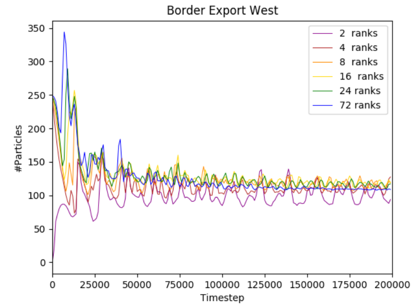
In addition to the data size ($bytes_{data}$), the number of data elements (N_{data}) transferred need to be determined in order to identify N_{bytes}. The number of data elements sent during the first stage is one `MPI_LONG`. Figure 5.58 presents the maximum number of particles being sent to east and west across all ranks during the second stage of `border_exchange`, for different rank counts at SCALE 1. The figure shows that the number of particles transferred, smooths out when timestep increases and the system reaches an approximate equilibrium. This approximately constant value is similar, independent of the rank count, because the height of the dam is constant at a given SCALE. The fluctuations increase along with increasing rank counts because the domain is being decomposed into more narrow rectangles and new subdomain boundaries are created. These new and potentially taller boundaries of the wave increases the maximum number of particles being transferred between the ranks that borders the tallest subdomain boundary. However, this maximum will stop increasing when a subdomain boundary is placed at the tallest point on the wave. Note that, for easier modeling purposes, the effects over several timesteps were smoothed out by averaging 1000 timesteps.

Figure 5.59 shows how the number of particles being exchanged increases with SCALE. In fact, the particle count increases approximately by a factor of SCALE compared to SCALE 1, on equal rank counts. The linear increase occurs because the height of the halos that are transferred between ranks increase by SCALE (cf. Figure 4.6 of Section 4.2.2), while the width is not affected as the interaction radius is not changed (cf. Section 4.2.2). In addition, the particle density within the halo is not changed significantly because there is a physical boundary on how many particles that can be placed within a certain region. Furthermore, the leveling effect as the timestep increases, observed in Figure 5.58, is also visible in Figure 5.59, because the dam reaches an approximate equilibrium. Hence, it is possible to model the maximum number of particles sent across all ranks in each timestep by the constant value derived from the right end of Figure 5.59. Then, at SCALE 1 with 72 ranks, the difference between this constant and the global maximum is about 210 particles as shown in Figure 5.58b, which may lead to an inaccuracy in the predicted communication time. However, we claim that this error is negligible because the difference is small, and not present when the dam has reached equilibrium.

Figure 5.60 shows the number of particles transferred to the neighbors in `migrate_particle`. Compared to the border exchange case in Figure 5.58, the number of particles being sent in `migrate_particle` is considerably less. This is because each increment in timestep does only result in a small change in a particle's position. However, the number increases with increasing rank counts as the subdomain border may be located at a place were the wave is taller, as was the case during `border_exchange`. In addition, the number approaches zero as timestep increases and the movements of the dam slows down. Figure 5.61 shows the case for several SCALEs. As with `border_exchange`, the number of particles being transferred increases by a factor of SCALE compared to SCALE=1. Note, however, that this number is of a much smaller magnitude than in `border_exchange`, and we claim that this number will be dominated by the latency. The number of data transfers to west is slightly greater than the number of transfers to east in both `border_exchange` `migrate_particle`, because the dam is placed on

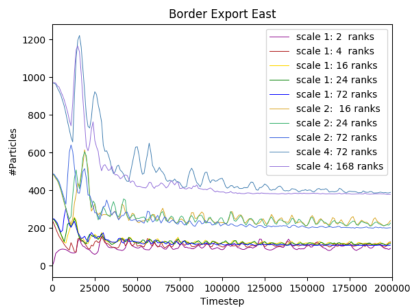


(a) Border exchange east.

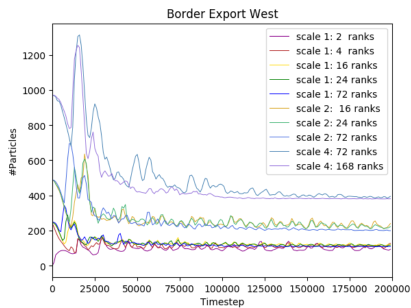


(b) Border exchange west

Figure 5.58: Maximum of all ranks on each timestep, averages of 1000 timesteps. From 0 to 200 000 timesteps, SCALE=1.

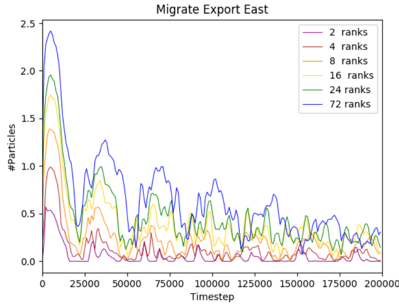


(a) Border exchange east.

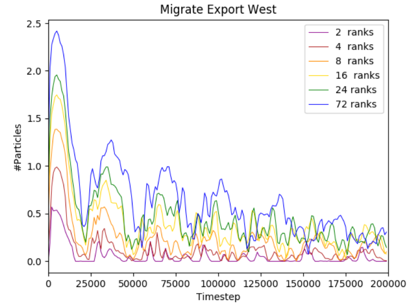


(b) Border exchange west

Figure 5.59: Maximum of all ranks on each timestep, averages of 1000 timesteps. From 0 to 200 000 timesteps, SCALE=1, SCALE=2 and SCALE=4.

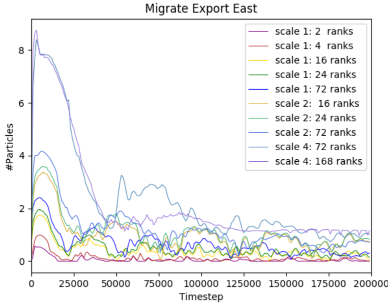


(a) Migrate east.

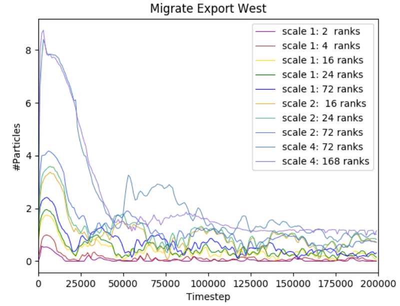


(b) Migrate west

Figure 5.60: Maximum of all ranks on each timestep, averages of 1000 timesteps. From 0 to 200 000 timesteps, SCALE=1.



(a) Migrate east.



(b) Migrate west

Figure 5.61: Maximum of all ranks on each timestep, averages of 1000 timesteps. From 0 to 200 000 timesteps, SCALE=1, SCALE=2 and SCALE=4.

the left side of the domain before the dam breaks. Equation 5.37 describes the number of particles being exchanged during `migrate_particle` and `border_exchange`.

$$N_{\text{data}} \approx N_{\text{data, equilibrium, SCALE=1}} \cdot \text{SCALE} \tag{5.37}$$

In order to utilize the results from Figures 5.58-5.61, Equations 5.32 and 5.33 can be reevaluated to be the maximum of the its individual components. Even though this may result in a larger communication cost, it will simplify the modeling process. Equations 5.38-5.39 and 5.40-5.41 modify Equations 5.32-5.33 and 5.36.

$$T_{\text{bexchange_stage}} \approx s \cdot T_{\text{bexchange_stage_east}} + s \cdot T_{\text{bexchange_stage_west}} \tag{5.38}$$

$$T_{\text{migrate_stage}} \approx s \cdot T_{\text{migrate_stage_east}} + s \cdot T_{\text{migrate_stage_west}} \tag{5.39}$$

$$\mathrm{T}_{\text{bexchange_stage_east/west}} \approx \max_{rank \in \mathrm{N}} \alpha(\text{rank,neighbor(rank)}_{\text{east/west}}) + \max_{rank \in \mathrm{N}} \mathrm{N}_{\text{bytes, east/west}} \cdot \max_{rank \in \mathrm{N}} \beta(\text{rank,neighbor(rank)}_{\text{east/west}} \quad (5.40)$$

$$\mathrm{T}_{\text{migrate_stage}} \approx \max_{rank \in \mathrm{N}} \alpha(\text{rank,neighbor(rank)}_{\text{east/west}}) + \max_{rank \in \mathrm{N}} \mathrm{N}_{\text{bytes, east/west}} \cdot \max_{rank \in \mathrm{N}} \beta(\text{rank,neighbor(rank)}_{\text{east/west}} \quad (5.41)$$

In Equations 5.38 and 5.39, the terms $\mathrm{T}_{\text{bexchange_stage_east/west}}$ $\mathrm{T}_{\text{migrate_stage_east/west}}$ denote the maximum transfer time in `border_exchange` and `migrate_particles`. respectively.

Finally, there is an important restriction on the number of ranks than can be utilized. In order to prevent particles from interacting with particles that are not in the subdomain of one of the neighboring ranks, the subdomain width (i.e. $fracBN$) has to be larger than the interaction radius. The number of ranks are constrained by Equation 5.42.

$$N < \frac{\mathrm{B}}{\text{interation_radius}} = \frac{\mathrm{B}}{\text{scale_k} \cdot \mathrm{H}} \quad (5.42)$$

5.2 Architectural Model

Knowledge of a platforms interconnection is necessary to understand how the communication step affects our application models. Different levels of latency and bandwidth is expected in a large cluster. For example, the latency between cores on a single socket should be lower than for cores on different sockets. Cores on different nodes adds another layer of connections.

To measure the parameters of the interconnect, a small benchmark was created to perform ping pong tests on all pairs of nodes. The benchmark is detailed in Listing 5.1.

```
double
time_pingpong ( int source, int peer, int n_tests, int msg_size, MPI_Comm pair_comm↩
    )
{
    double start, end;
    MPI_Barrier ( pair_comm );
    start = MPI_Wtime();
    if ( source == rank )
    {
        for ( int i=0; i<n_tests; i++ )
            MPI_Ssend ( message, msg_size, MPI_CHAR, peer, 0, MPI_COMM_WORLD );
        for ( int i=0; i<n_tests; i++ )
            MPI_Recv ( message, msg_size, MPI_CHAR,
                peer, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE
            );
    }
    else if ( peer == rank )
    {
        for ( int i=0; i<n_tests; i++ )
            MPI_Recv ( message, msg_size, MPI_CHAR,
                source, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE
            );
        for ( int i=0; i<n_tests; i++ )
            MPI_Ssend ( message, msg_size, MPI_CHAR, source, 0, MPI_COMM_WORLD );
    }
    end = MPI_Wtime();
    return (end-start)/(2.0*n_tests*msg_size);
}
```

Listing 5.1: Ping pong benchmark

The function `time_pingpong` takes two ranks as parameters, the number of tests to be performed, the message size and a custom communicator. The custom communicator is a communicator which only holds the two ranks which are currently executing the function. This enables us to create a barrier with only the relevant ranks. The two ranks exchange `n_tests` number of messages and the time required is measured with `MPI_Wtime()`. After the all messages are sent and received, the average time spent per test is returned.

By adjusting the parameters `n_tests` and `msg_size`, we can retrieve either the latency or the inverse bandwidth. For example, when the number of tests is low (< 30) and the message size is large (> 64 MB), then the bandwidth will be the dominating factor of the time spent per message. The unit of the inverse bandwidth is [s/byte].

By setting the number of tests high (>10 000) and the message size low (< 1 byte), the latency will be the dominating factor of the time spent per message. The unit of the latency is [s].

In our experiments, the `time_pingpong` function is used to find the inverse bandwidth and latency for all pairs of nodes. The results for Vilje, EPIC and EPT are shown in Figures 5.62-5.67.

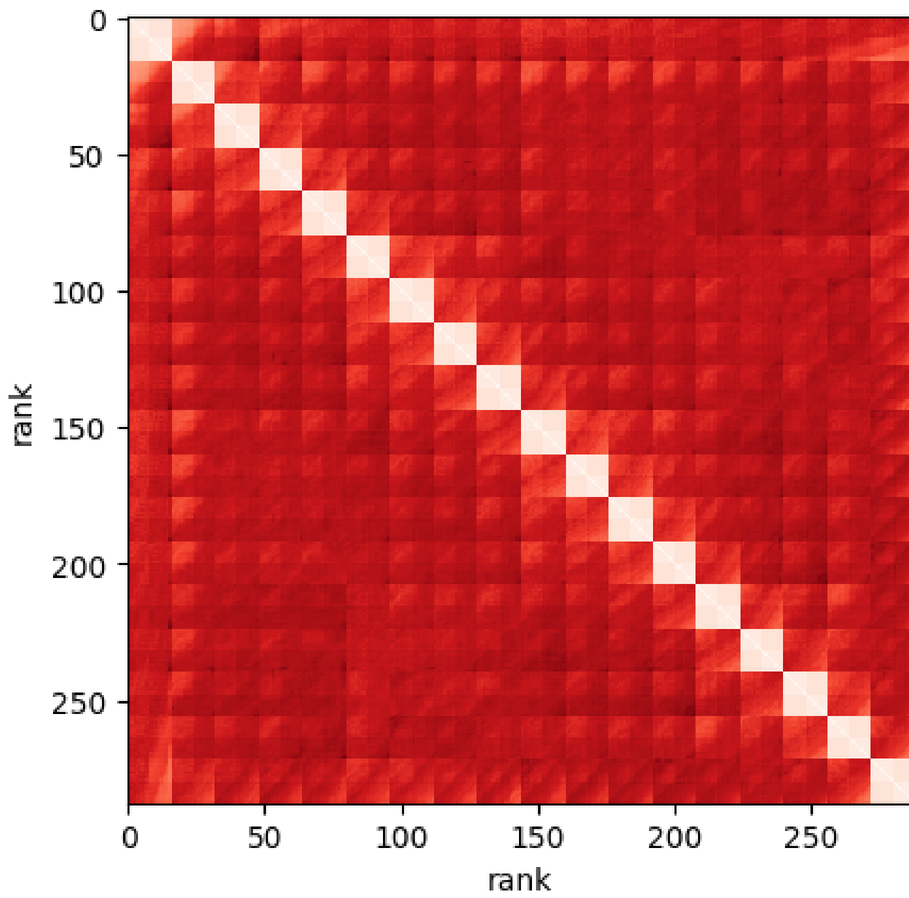


Figure 5.62: Vilje, Latency, 18 Nodes, 16 ranks per node. min:9.612437e-07 max:1.662683e-05

Figure 5.62 shows the latency result of a ping pong test with 18 nodes on Vilje, running 16 ranks per node. As predicted, there are multiple visible levels of latencies. The light blocks along the diagonal represent communication within a one node. Within the light blocks, there is two shades of light red, which indicate the difference between intra-socket communication and inter-socket communication.

The latency of intra-socket communication on Vilje is about $1.04 \cdot 10^{-6}$s, while the inter-socket latency is about $1.74 \cdot 10^{-6}$s. Inter node communication is in the order of 10^{-5}s. These values represents three classes of architectural parameters which can be used as part of the heterogeneous Hockney model for Vilje. Table 5.1 shows latencies for EPIC and EPT. Inverse bandwidths for all architectures are listed in Table 5.2.

	Intra Socket	Inter Socket	Inter Node
Vilje	$7.15 \cdot 10^{-7}$s	$1.52 \cdot 10^{-6}$s	$1.66 \cdot 10^{-5}$s
EPT	$5.29 \cdot 10^{-7}$s	$1.43 \cdot 10^{-6}$s	$7.59 \cdot 10^{-6}$s
EPIC	$5.35 \cdot 10^{-7}$s	$2.62 \cdot 10^{-6}$s	$8.12 \cdot 10^{-6}$s

Table 5.1: Latency parameters for our test platforms

	Intra Socket	Inter Socket	Inter Node
Vilje	$1.66 \cdot 10^{-10}$s/byte	$6.07 \cdot 10^{-10}$s/byte	$5.54 \cdot 10^{-9}$s/byte
EPT	$2.60 \cdot 10^{-10}$s/byte	$3.84 \cdot 10^{-10}$s/byte	$4.13 \cdot 10^{-9}$s/byte
EPIC	$2.54 \cdot 10^{-10}$s/byte	$4.46 \cdot 10^{-10}$s/byte	$8.81 \cdot 10^{-9}$s/byte

Table 5.2: Inverse bandwidth parameters for our test platforms

18 ranks is equal to one rack unit (*IRU*), which consists of two half IRUs. The half IRUs are connected by a separate connection, however, this is not apparent in our measurements.

Appendix C contains a variety of heat maps on smaller scales than those in this section.

5.3 Performance Models

5.3.1 LBM

This section summarizes the models conceived in Sections 5.1.1.1, 5.1.1.2 and 5.2.

The total execution time of the application is expressed as the computation time and the communication time, as seen in Equation 5.43.

$$\mathrm{T_{total}} = \mathrm{T_{compute}} + \mathrm{T_{communicate}} \tag{5.43}$$

In Section 5.1.1.1, we learned that the execution time per iteration of LBM is close to constant. Meaning $\mathrm{T_{compute}}$ can be approximated by measuring the constants for the methods collide (c_c) and propagate (c_p), and multiplying by the number of iterations one wishes to execute for, as shown in Equation 5.44. In Equation 5.44, the terms c_c and c_p are dependant on the thread count, the architecture and the number of lattice points.

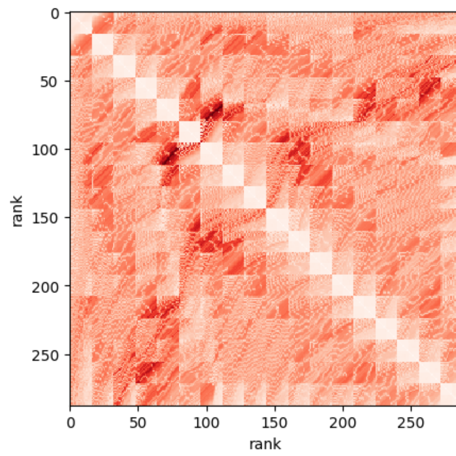


Figure 5.63: Vilje, Beta Inverse, 18 Nodes, 16 ranks per node. min:1.625543e-10 max:5.536014e-09

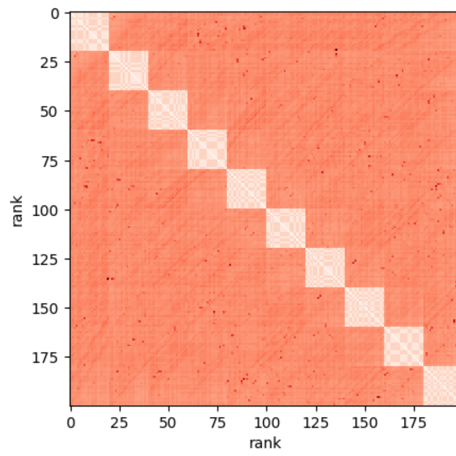


Figure 5.64: EPT, Latency, 10 Nodes, 20 ranks per node. min:5.296946e-07 max:7.599199e-06

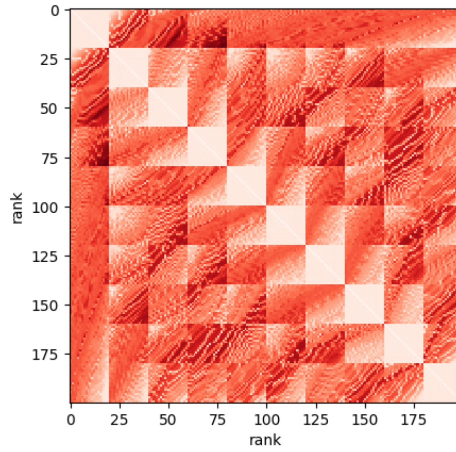


Figure 5.65: EPT, Beta Inverse, 10 Nodes, 20 ranks per node. min:2.59738e-10 max:4.134643e-09

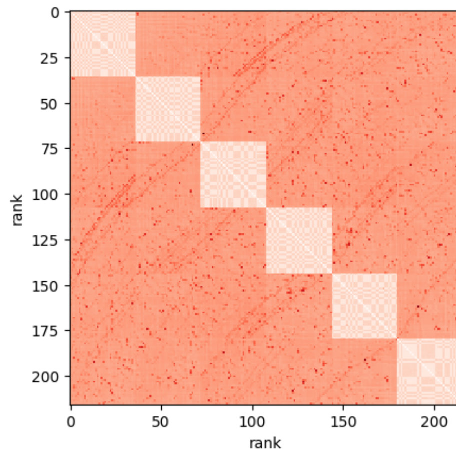


Figure 5.66: EPIC, Latency, 6 Nodes, 36 ranks per node, min:5.356431e-07 max:8.119893e-06

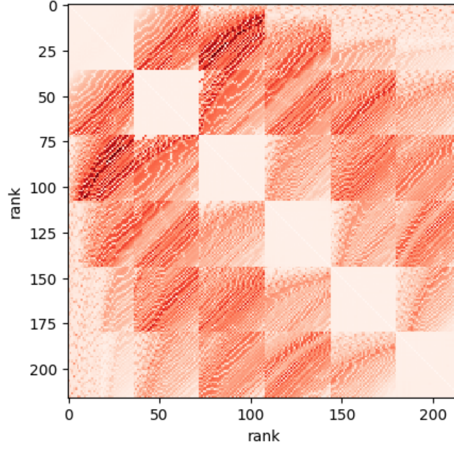


Figure 5.67: EPIC, Beta Inverse, 6 Nodes, 36 ranks per node, min:2.17188e-10 max:8.808944e-09

$$\mathrm{T}_{\text{compute}} \approx (c_c + c_p)N_{\text{it}} \tag{5.44}$$

The communication time is the time to perform a border exchange, and is modelled in Equation 5.45.

$$\mathrm{T}_{\text{border_exchange}} \approx \max_{\text{rank} \in N}[\mathrm{T}_{\text{east}}(\text{rank}) + \mathrm{T}_{\text{west}}(\text{rank})] + \max_{\text{rank} \in N}[\mathrm{T}_{\text{north}}(\text{rank}) + \mathrm{T}_{\text{south}}(\text{rank})] \tag{5.45}$$

Where each term is based on the heterogeneous Hockney model, as shown in Equation 5.46. For example, T_{east}(rank) would translate to $\mathrm{T}(i, j)$ where i is "rank" and j is the eastern neighbor of "rank".

$$\mathrm{T}(i, j) \approx \alpha_{ij} + w_{\text{direction}}\beta_{ij}, i \neq j \tag{5.46}$$

Based on our results, we can make four recommendations.

1. The effectiveness of tasks compared to worksharing constructs must be determined for the architecture.

2. To avoid saturating the memory channels of a socket, the application should run with one rank per socket.

3. In order to estimate the total execution time of N_{it} iterations, run for a small amount of iterations and capture the constants for propagate and collide. The total time is then estimated by multiplying N_{it} by the sum of the constants.

4. The total execution time is dominated by the computational step, because the number of messages in the communication step is small, the number of bytes per message low, and the latency and bandwidth of the interconnects we have used are favorable for fast communication.

Note that in order to predict the application's execution time on an unknown machine, the unknown machine must be available for test runs so that it's possible to establish the constants c_c and c_p. However, one can still obtain an understanding of the application's characteristics through the model.

5.3.2 SPH

This section summarizes the model equations described in Sections 5.1.2.1 and 5.1.2.2.

Three implementation of the SPH program have been analyzed; the brute-force implementation where all particle combinations are checked for proximity (cf. Section 4.2.2.1), the cell-linked particle implementation were the particles are placed in buckets and only neighboring buckets are examined for proximity (cf. Section 4.2.2.1), and the cell-linked particle and pair implementation were both particles and pairs are located in buckets to avoid the critical sections present in both of the other implementations (cf. Section 4.2.2.1).

The total execution time of a program consist of the communication time and the computation time, in addition to any overlap between them. In our SPH application, there is no overlap between communication and computation. Hence, the total execution time can be described as in Equation 5.47.

$$T_{\text{total_execution_time}} = T_{\text{communication_time}} + T_{\text{computation_time}} \tag{5.47}$$

The computation time is dominated by the `find_neighbors` method in all three implementations. The neighbor loop (i.e. a nested loop iterating over all particle combinations) dominates the computation time of the `find_neighbors` method in the brute-force implementation, while the `create_pairs` method and subroutine dominate `find_neighbors` in cell-linked particle lists and cell-linked particle and pair lists, respectively. In the brute-force implementation, the compute time is represented through the number of iterations of the neighbor loop as the number of iterations is much greater than the pair count, and therefore closely coupled with the execution time. There is a linear relationship between the iteration count and the execution time. In the cell-linked particle list method, the pair count is a more accurate representation of the execution time because the probability of detecting a pair (i.e. the critical section) during a proximity check is high. However, the critical section prevents the speedup from increasing beyond a socket. There is a linear relationship between the pair count and the computation time. The cell-linked particle and pair lists method is not bounded by any critical sections and is not significantly effected by the increase in particle count within a SCALE. Hence, the computation in cell-linked particle and pair lists can be represented as a constant. Furthermore, the brute-force implementation and the cell-linked particle and pair lists implementation can be executed on threads filling the entire node without a decrease in speedup.

The computation times of the brute-force implementation, the cell-linked particle list implementation and the cell-linked particle and pair lists implementation are captured in Equations 5.48, 5.49 and 5.50, respectively.

$$\begin{aligned} T_{\text{computation_time_bf}} &\approx \mathrm{T}_{\text{neighbor_loop}}(\mathrm{N}_{\text{thread}}, \text{arch}, \mathrm{N}_{\text{loop_iter}}) \\ &\approx \text{slope}_{\text{SCALE1}}(\mathrm{N}_{\text{thread}}) \cdot \mathrm{N}_{\text{loop_iter}} + \text{base}_{\text{SCALE1}}(\mathrm{N}_{\text{thread}}, \text{arch}) \end{aligned} \tag{5.48}$$

$$T_{\text{computation_time_clpl}} \approx \text{T}_{\text{create_pairs}}(\text{N}_{\text{thread}}, \text{arch}, \text{N}_{\text{pairs}}) \approx \text{slope}_{\text{SCALE1}}(\text{N}_{\text{thread}}) \cdot \text{N}_{\text{pairs}} + \text{base}_{\text{SCALE1}}(\text{N}_{\text{thread}}, \text{arch}) \tag{5.49}$$

$$T_{\text{computation_time_clppl}} \approx \text{T}_{\text{create_pairs}}(\text{N}_{\text{threads}}, \text{arch}, \text{SCALE}) \approx \text{base}_{\text{SCALE1}}(\text{N}_{\text{threads}}, \text{arch}) \cdot \text{SCALE}^2 \tag{5.50}$$

In Equations 5.48, 5.49 and 5.50, the term loop_iter denotes the number of iterations of the neighbor loop, the term arch denotes the architecture, the term slope denotes the gradient of the linear relationship and base denotes the intercept. In addition, the abbreviations bf, clpl and clppl denote brute-force implementation, the cell-linked particle implementation and the cell-linked particle and pair lists implementation, respectively. Both base and slope are drawn by executing the program for about 20 000 timesteps at SCALE 1 on a given architecture with a given thread count.

In contrast to the computation time, the communication time of the SPH application is equivalent for all implementations. There are two methods in the SPH application were communication is performed; `border_exchange` and `migrate_particles`. This is captured in Equation 5.51. The communication in both of these methods are one dimensional and performed in two stages. During the first stage, an integer representing the number of particles that will be transferred during the second stage is communicated. The particles are sent during the second stage. This is shown in Equations 5.52 and 5.53. The term s in Equations 5.52 and 5.53 is 1 if the execution time of one `MPI_Sendrecv` corresponds to one `MPI_Ssend`, and 2 otherwise. The four different communication stages in the SPH application are approximated in Equations 5.54-5.57.

$$\text{T}_{\text{communication}} = \text{T}_{\text{migrate_particles}} + \text{T}_{\text{border_exchange}} \tag{5.51}$$

$$\text{T}_{\text{border_exchange}} = \text{s} \cdot \text{T}_{\text{bexchange_count}} + \text{s} \cdot \text{T}_{\text{bexchange_the_particles}} \tag{5.52}$$

$$\text{T}_{\text{migrate_particles}} = \text{s} \cdot \text{T}_{\text{migrate_count}} + \text{s} \cdot \text{T}_{\text{migrate_the_particles}} \tag{5.53}$$

$$\text{T}_{\text{bexchange_count}} \approx \max_{rank \in \text{N}} \alpha(\text{rank,neighbor(rank)}_{\text{east}}) + 4B \cdot \max_{rank \in \text{N}} \beta(\text{rank,neighbor(rank)}_{\text{east}} + \max_{rank \in \text{N}} \alpha(\text{rank,neighbor(rank)}_{\text{west}}) + 4B \cdot \max_{rank \in \text{N}} \beta(\text{rank,neighbor(rank)}_{\text{west}} \tag{5.54}$$

$$\text{T}_{\text{bexchange_the_particles}} \approx \max_{rank \in \text{N}} \alpha(\text{rank,neighbor(rank)}_{\text{east}}) + \max_{rank \in \text{N}} \text{N}_{\text{bytes, east}} \cdot \max_{rank \in \text{N}} \beta(\text{rank,neighbor(rank)}_{\text{east}} + \max_{rank \in \text{N}} \alpha(\text{rank,neighbor(rank)}_{\text{west}}) + \max_{rank \in \text{N}} \text{N}_{\text{bytes, west}} \cdot \max_{rank \in \text{N}} \beta(\text{rank,neighbor(rank)}_{\text{west}} \tag{5.55}$$

$$\mathrm{T_{migrate_count}} \approx \max_{rank \in \mathrm{N}} \alpha(\mathrm{rank,neighbor(rank)_{east}}) + 4B \cdot \max_{rank \in \mathrm{N}} \beta(\mathrm{rank,neighbor(rank)_{east}} \\ + \max_{rank \in \mathrm{N}} \alpha(\mathrm{rank,neighbor(rank)_{west}}) + 4B \cdot \max_{rank \in \mathrm{N}} \beta(\mathrm{rank,neighbor(rank)_{west}} \tag{5.56}$$

$$\mathrm{T_{migrate_the_particles}} \approx \max_{rank \in \mathrm{N}} \alpha(\mathrm{rank,neighbor(rank)_{east}}) \\ + \max_{rank \in \mathrm{N}} \mathrm{N_{bytes,\ east}} \cdot \max_{rank \in \mathrm{N}} \beta(\mathrm{rank,neighbor(rank)_{east}} \\ + \max_{rank \in \mathrm{N}} \alpha(\mathrm{rank,neighbor(rank)_{west}}) \\ + \max_{rank \in \mathrm{N}} \mathrm{N_{bytes,\ west}} \cdot \max_{rank \in \mathrm{N}} \beta(\mathrm{rank,neighbor(rank)_{west}} \tag{5.57}$$

In Equations 5.54-5.57, the term $\mathrm{N}_{bytes,east}$ denote the number of bytes transferred to the neighbor in east when the dam has reach an approximate equilibrium. This number is retrieved by multiplying the number of particles that are transferred, by the size of a particle. The term N is the number of ranks, and α and β are constants denoting the latency and bandwidth in the Heterogeneous Hockney model (cf. Section 3.4.1.1). The the_particle and count subscripts denote the two stages of the communication. The factor 4B found in Equations 5.54 and 5.56 are is the size of an `MPI_LONG`.

Note that the number of ranks is constrained by the interaction radius as described in Equation 5.58.

$$N < \frac{\mathrm{B}}{\mathrm{interaction_radius}} = \frac{\mathrm{B}}{\mathrm{scale_k} \cdot \mathrm{H}} \tag{5.58}$$

Based on these results, we can make 6 recommendations:

1. As the number of ranks that can be utilize on a given SCALE is limited, we have to utilize as many threads as possible in order to obtain maximum speedup.
2. Brute-force: In order to estimate slope and base on a given thread count and architecture, run the application on that thread count and architecture until a linear pattern between neighbor loop iterations ($\mathrm{N_{loop_iter}}$) in each timestep and execution time is observed.
3. Cell-linked particle list: In order to estimate slope and base on a given thread count and architecture, run the application on that thread count and architecture until a linear pattern between the number of pairs ($\mathrm{N_{pairs}}$) in each timestep and execution time is observed.
4. Cell-linked pair and particle list: In order to estimate base on a given thread count and architecture, run the application on that thread count and architecture until the fluctuations in execution time per particle count is almost non-existent.

5. The number of particles transferred during communication can be estimated by running the application until the number of particles transmitted per timestep stabilizes.

6. The brute-force implementation and the cell-linked pair and particle list implementation should be executed with 1 rank per node, while the cell-linked particle method should be executed with 1 rank per socket.

Note that the computation models created in this section estimate the compute time in terms of the most compute intensive method. Hence, there may be some variations in the accuracy of the models. This is especially true for the cell-linked particle and pair lists implementation. Here, the modeled subroutine does only represent about $\frac{4}{5}$ of the compute time of `find_neighbors`, and the remaining methods of `time_step` account for about $\frac{1}{3}$ of `time_step`'s execution time. The methods of `time_step` does not contain any critical sections. Methods outside of the `time_step` method, however, do contain critical sections, and may therefore represent significant parts of the execution time once the bottleneck on `find_neighbours` is removed. In addition, we need to be able to execute the application on the thread count and architecture that we would like to model in order to get accurate results. However, the findings of Sections 5.1.2.1 and 5.1.2.2 may still be useful as they expose important characteristics about the application and how it performs, not only for the Dambreak problem, but also for other problems. Furthermore, the communication models include a parameter representing the number of transferred particles. This parameter is difficult to predict accurately. However, the communication time is much larger than the computation time as a transfer of 3079 particles (were each particle is 184B) is needed to exceed a computation time of 0.005s (with the slowest interconnect; inter-node on EPIC). Hence, we claim that the communication models do not need to be as accurate as the computation models for the SPH application.

5.4 Parameter Tabels

arch	the architecture that the application executes on.
B	The global width of the tank.
N	The number of ranks.
interaction radius	If two particles are within each others interaction radius, a pair is created.
$N_{threads}$	The number of threads.
`find_neighbors`	The most compute intensive method in the SPH application.
`time_step`	The method where most of the computation takes place.
H	The smoothing length, the interaction radius is determined by this length.
scale_k	A factor of the interaction radius.
neighbor loop	The nested loop that iterates over all particle combinations.
`time_integration`	The method in the SPH application that iterates through timesteps.

timestep	The evolution of the SPH application is discretized into small units of time called timestep.
SCALE	A variable used to change the size of the domain.
$N_{particles}$	The number of particles present in a timestep.
N_{pairs}	The number of pairs present in a timestep.
N_{loop_iter}	The number of iterations of the neighbor loop during a timestep.
N_{data}	The number of data units transferred during communication.
$N_{data,\ equilibrium,\ SCALE=1}$	The number of data units transferred when the dam has reach an approximate equilibrium, at SCALE=1.
N_{bytes}	The number of bytes transferred during communication.
$bytes_{data}$	The size of the data unit in bytes.
α	Latency in the Hockney model.
β	Bandwidth in the Hockney model.
the brute-force implementation	The naive detect neighbor implementation that examines all particle combinations for proximity.
the cell-linked particle list implementation	Only the nine neighboring buckets are examined during proximity checks. The creation of a pair includes a critical section.
the cell-linked particle and pair lists implementation	Only the nine neighboring buckets are examined during proximity checks. No critical section during pair creation. Pairs across buckets are duplicated.
s	This term describes the correspondence between the execution time of `MPI_Sendrecv` and `MPI_Ssend`. It is 1 if they are equal and 2 otherwise.
stage	Can be either *count* (first stage: the particle count is transferred) or *the_particles* (second stage: the particles are transferred).
the_particles	Subscript that denotes the second stage of the communication (i.e. particles are transferred).
count	Subscript that denotes the first stage of the communication (i.e. the particles count is transferred).

border_exchange	Exchanges the halo of each subdomain in order to accommodate for distributed memory parallelism.
migrate_particles	Sends particles which have moved out of this subdomain, and receives particles which have entered the subdomain.

Table 5.4: SPH Parameter table for chapter 5.

SCALE	Size of domain: $(\text{WIDTH} \cdot \text{SCALE}) \cdot (\text{HEIGHT} \cdot \text{SCALE})$
N	Number of ranks
rank	Rank ID
N_{it}	Number of iterations/time steps
width_a	width of domain with scale a
height_a	height of domain with scale a
c_c	constant approximation of iteration time in collide method
c_p	constant approximation of iteration time in propagate method
$\text{T}_{\text{collide}}$	Time for the collide step
$\text{T}_{\text{propagate}}$	Time for the propagate step
$\text{T}_{\text{compute}}$	Time for the compute step
$\text{T}_{\text{communicate}}$	Time for the communication step
T_{total}	Time to perform compute and communication steps
neighbor	Rank of neighbor
α	Latency
β	Inverse Bandwidth
H_l	Height of local subdomain
W_l	Width of local subdomain
H_g	Height of global domain
W_g	Width of global domain
p_x	Number of ranks in the x-direction
p_y	Number of ranks in the y-direction
w	Number of bytes
$\text{T}_{\text{border_exchange}}$	Time to complete border exchange
$\text{T}_{\text{direction}}(\text{rank})$	Time for rank to complete an exchange with its neighbor in "direction"
$\text{T}(i, j)$	Time for rank i and j to complete an exchange

Table 5.3: LBM Parameter table for chapter 5.

Chapter 6

Validation

In order to evaluate the predictive power of our models with regards to scalability, we run a set of experiments on an architecture not used during development, Archer, the UK national supercomputer.

Archer consists of 4920 compute nodes, with 64 GB of memory per node. Each node contains two Intel Xeon 12 core processors, which is a different core density than the ones already seen in Vilje, EPIC and EPT. For the purpose of validation, this is positive because the platform has not been a part of the design of the models.

In this chapter, we test the optimal configuration for scalability in terms of threads and ranks, as recommended by our performance models from Chapter 5. In Section 6.1, we describe the experiments we performed, and their results. In Section 6.2, we evaluate the predictive power of our models based on the experimental results.

6.1 Test procedure and results

In order to test our predictions, we ran both our applications with multiple thread/rank configurations to see if the scalability is predicted by our models.

6.1.1 LBM

In our model, whether worksharing or tasking is faster depends on the architecture. Therefore, we test Moffatt on a single node on SCALE=20 for both methods. On Archer, this results in worksharing being slightly faster (0.01s per iteration) than tasking. Based on this, the following tests are performed with worksharing.

The performance model for LBM indicate that running one rank per socket is the optimal configuration. To test this, we ran LBM Moffatt on SCALE = 40 on 2, 4, 8 and 16 nodes for two cases. The "good case" has one rank per socket, the "bad case" has one rank per node. The result is illustrated in Figure 6.1, the baseline is the best result on two nodes. As predicted, using one rank per socket is optimal.

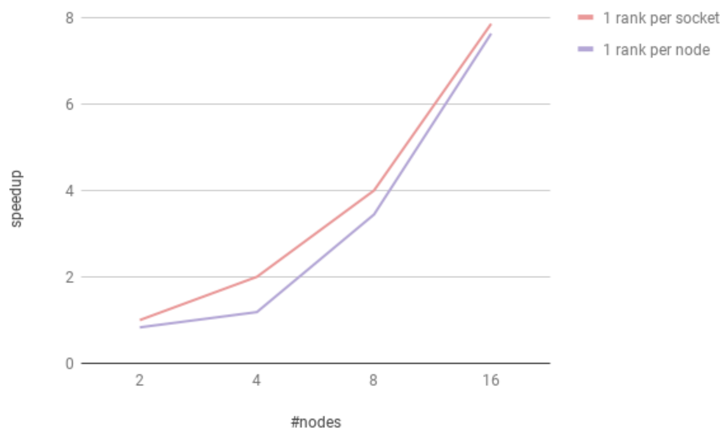


Figure 6.1: LBM Moffatt speedup with SCALE=40 on Archer.

6.1.2 SPH

Our performance model for SPH indicate that running one rank per node is the optimal configuration, because the speedup only grows as the thread count increases, and the number of ranks we can have is bounded by the SCALE parameter. This is tested by running two variants of SPH, brute force and cell linked list for pairs and particles, on SCALE 1 with 2, 4, 8 and 16 nodes. The result is shown in Figure 6.2. The baseline is the best result on two nodes. Both variants achieve speedups with one rank per node, indicating that the sockets are not saturated even with full thread parallelism. The cell list variant is predictably faster than the brute force variant.

6.2 Evaluation

Through our analysis of the proxy applications, we have achieved a set of performance models that can predict the characteristics of our proxy applications. The modelled effects are shown to be accurate, even when running on Archer, which is an architecture outside of our of testing platforms. We opted not to test any of the communication characteristics as our analysis indicated that it would not be the dominating effect on this scale.

Latency and Bandwidth heat maps for Archer can be found in Figure C.9 and Figure C.10, respectively.

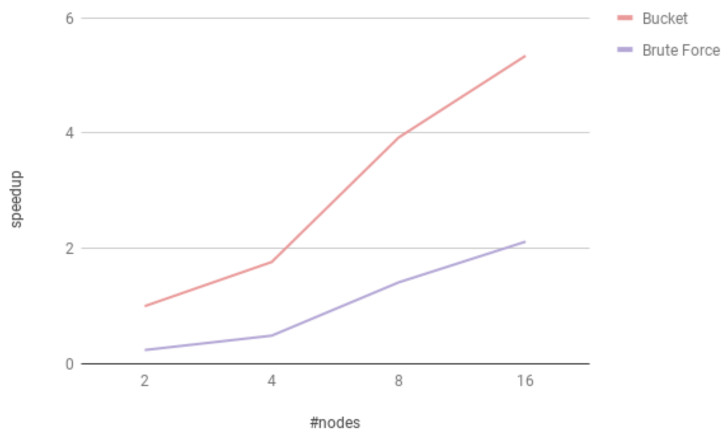


Figure 6.2: SPH speedup with SCALE=1 on Archer.

Chapter 7

Conclusion and Future Work

In this thesis we have developed and analyzed two CFD proxy applications in six variants on four machines. From these we have created performance models of the interactions between software and hardware which are common between architectures.

We have discovered that the effect of using tasking constructs over worksharing constructs for LBM is dependent on the system architecture. We have learned that the LBM application is memory bound which can lead to diminishing returns in speedup if one rank per node is used instead of one rank per socket.

In the SPH application we have discovered that models based on the number of loop iterations, pairs or particles is sufficient to create models which accurately describe the application behaviour. Surprisingly, the execution time grows predictably with the size of the problem, even though the dambreak problem starts with an unbalanced work load.

The communication models of both LBM and SPH are similar and their impact is small compared to the computational models. Both applications only exchange small amounts of data per iteration. Combined with the fast interconnects on Vilje, EPIC, EPT and Archer, the applications are well suited for running with large problem sizes on a large number of cores.

7.1 Future work

There are plenty of interesting ways this work could continue.

Domínguez *et al.* (2011) describes and evaluates many algorithms for neighbor finding in SPH. Performance of our proxy application could improve by implementing some their recommendations.

Replacing the linked lists with arrays in the cell linked list methods would be interesting to investigate, especially in terms of cache performance.

Because the neighbor finding routine in the "cell-linked list for particles and pairs" method only accounts for about $\frac{3}{4}$ of the `time_step` method, it would be interesting to model the other methods of the `time_step` method as well.

The neighbor finding routine in SPH could be further improved by offloading it to a GPGPU.

The CUDA version of LBM performs extremely well, and it would be intriguing to create performance models for it. In the current CUDA version, all data is transferred back from the GPU for each border exchange. Implementing a CUDA-aware MPI version of LBM would probably be faster and more efficient, because data can be transferred from GPU device memory directly to the interconnect (requires GPUDirect) without copying to host memory first.

Bibliography

Alexandrov, A., Ionescu, M., Schauser, K., and Scheiman, C. (1997). Loggp: Incorporating long messages into the logp model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79.

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), pages 483–485, New York, NY, USA. ACM.

Asanović, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. (2006). The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

B. Lucy, L. (1977). A numerical approach to the testing of the fission hypothesis. *Astronomical Journal*, 82:1013–1024.

Banerjee, T., Hackl, J., Shringarpure, M., Islam, T., Balachandar, S., Jackson, T., and Ranka, S. (2016). Cmt-bone — a proxy application for compressible multiphase turbulent flows. *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*.

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

Bhatnagar, P. L., Gross, E. P., and Krook, M. (1954). A model for collision processes in gases. i. small amplitude processes in charged and neutral one-component systems. *Phys. Rev.*, 94:511–525.

Chen, S. and Doolen, G. D. (1998). Lattice boltzmann method for fluid flows. *Annual Review of Fluid Mechanics*, 30(1):329–364.

Chen, S., Doolen, G. D., and Eggert, K. G. (1994). Lattice-boltzmann fluid dynamics; a versatile tool for multiphase and other complicated flows. *Los Alamos Science*, 22:98–109.

Cicotti, P., Mniszewski, S. M., and Carrington, L. (2014). An evaluation of threaded models for a classical md proxy application. *2014 Hardware-Software Co-Design for High Performance Computing*.

Culler, Karp, P. S. S. S. u. E. (1993). Logp - towards a realistic model of parallel computation. *Sigplan Notices*, 28(7):1–12.

Dickson, J., Wright, S., Maheswaran, S., Herdmant, A., Miller, M., and Jarvis, S. (2016). Replicating hpc i/o workloads with proxy applications. In *Proceedings of the 1st Joint International Workshop on parallel data storage & data intensive scalable computing systems*, PDSW-DISCS '16, pages 13–18. IEEE Press.

Domínguez, J. M., Crespo, A. J. C., Gómez-Gesteira, M., and Marongiu, J. C. (2011). Neighbour lists in smoothed particle hydrodynamics. *International Journal for Numerical Methods in Fluids*, 67(12):2026–2042.

Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. S. (1990). A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17.

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

Frisch, U., Hasslacher, B., and Pomeau, Y. (1986). Lattice-gas automata for the navier-stokes equation. *Physical Review Letters*, 56(14):1505–1508.

Gingold, R. A. and 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.

Gustafson, J. L. (1988). Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533.

Hardy, J., Pomeau, Y., and de Pazzis, O. (1973). Time evolution of a two-dimensional classical lattice system. *Phys. Rev. Lett.*, 31:276–279.

Karlin, I., Bhatele, A., Keasler, J., Cohen, J., Devito, Z., Haque, R., Laney, D., Luke, E., Wang, F., Richards, D., Schulz, M., and Still, C. H. (2012). Exploring traditional and emerging parallel programming models using a proxy application. *Presented at: 27th IEEE International Parallel & Distributed Processing Symposium, Boston, MA, United States*.

Kielmann, T., Bal, H. E., and Verstoep, K. (2000). Fast measurement of logp parameters for message passing platforms. In Rolim, J., editor, *Parallel and Distributed Processing*, pages 1176–1183, Berlin, Heidelberg. Springer Berlin Heidelberg.

Lastovetsky, A., Rychkov, V., and O'Flynn, M. (2010). Accurate heterogeneous communication models and a software tool for their efficient estimation. *International Journal of High Performance Computing Applications*, 24(1):34–48.

Lawson, G., Sosonkina, M., and Shen, Y. (2015). Changing cpu frequency in comd proxy application offloaded to intel xeon phi co-processors. *Procedia Computer Science*, 51(1):100–109.

Mattson, W. and Rice, B. M. (1999). Near-neighbor calculations using a modified cell-linked list method. *Computer Physics Communications*, 119(2):135–148.

McCalpin, J. (1995). Memory bandwidth and machine balance in high performance computers.

McNamara, G. and Zanetti, G. (1988). Use of the boltzmann equation to simulate lattice-gas automata. *Physical Review Letters*, 61(20):2332–2335.

Moffatt, H. K. (1964). Viscous and resistive eddies near a sharp corner. *Journal of Fluid Mechanics*, 18(1):1–18.

Monaghan, J. (1994). Simulating free surface flows with sph. *Journal of Computational Physics*, 110(2):399–406.

Monaghan, J. J. and Kos, A. (1999). Solitary waves on a cretan beach. *Journal of Waterway, Port, Coastal, and Ocean Engineering*, 125(3):145–155.

OpenMP Architecture Review Board (2015). OpenMP application program interface version 4.5. `https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`, [accessed 2018-06-27].

Ozbulut, M., Yildiz, M., and 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.

Pacheco, P. (2011). *An introduction to parallel programming*. Morgan Kaufmann, Amsterdam Boston.

Rabenseifner, R. (2003). Hybrid parallel programming on hpc platforms. In *proceedings of the Fifth European Workshop on OpenMP, EWOMP*, volume 3, pages 185–194.

Valiant, L. (1990). A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111.

Wilkinson, B. (2005). Parallel programming : techniques and applications using networked workstations and parallel computers.

Williams, S., Waterman, A., and Patterson, D. (2009). Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65.

Appendices

Appendix A

Selected Source Code

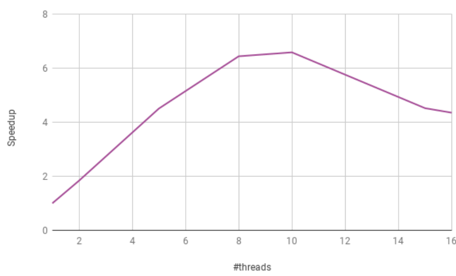
```
#pragma omp for
for (int x = 0; x < N_BUCKETS_X; ++x) {
    for (int y = 0; y < N_BUCKETS_Y; ++y) {
        pair_t* base_pair = &pairs[BID(x, y)];
        bucket_t* base_bucket = buckets[BID(x, y)];
        particle_t* base_particle;
        if (base_bucket != NULL) { //kan kanskje fjernes
            base_particle = base_bucket->particle;
            if (base_particle == NULL) {
                continue;
            }
        }
        else
            continue;

        while(base_bucket != NULL && base_bucket->particle != NULL) {
            base_particle = base_bucket->particle;

            int noffsets[8][2] = {
                                 {-1, 1}, // Top left
                                 {0, 1}, // Top center
                                 {1, 1},  // Top right
                                 {1, 0},  // Right
                                 {1, -1}, // Bottom right
                                 {0, -1}, // Bottom center
                                 {-1, -1}, // Bottom left
                                 {-1, 0},  // Left
            };

            for (int i = 0; i < 9; ++i) {
                bucket_t* current_bucket;
                if (i == 0) {
                    current_bucket = base_bucket->next;
                } else {
                    int bx = x+noffsets[i-1][0];
                    int by = y+noffsets[i-1][1];
                    if (bx >= N_BUCKETS_X  bx < 0
                        by >= N_BUCKETS_Y  by < 0) continue;
                    current_bucket =
                        buckets[BID(bx, by)];
                }
                while (current_bucket != NULL && current_bucket->particle != NULL) ↩
                     {
                    particle_t* current_particle = current_bucket->particle;
```

```
44                          double distance_squared =
45                              pow(base_particle->x[0] - current_particle->x[0], 2) +
46                              pow(base_particle->x[1] - current_particle->x[1], 2);
47                          if (distance_squared <= RADIUS*RADIUS) {
48 
49                              if (base_pair->ip == NULL) {
50                                  base_pair->i = base_particle->local_idx;
51                                  base_pair->j = current_particle->local_idx;
52                                  base_pair->ip = base_particle;
53                                  base_pair->jp = current_particle;
54                                  base_pair->r = sqrt(distance_squared);
55                                  base_pair->q = base_pair->r / H;
56                                  base_pair->w = 0.0;
57                                  base_pair->dwdx[0] = base_pair->dwdx[1] = 0.0;
58                              } else {
59                                  base_pair->next = (pair_t*)malloc(sizeof(pair_t));
60                                  base_pair->next->i = base_particle->local_idx;
61                                  base_pair->next->j = current_particle->local_idx;
62                                  base_pair->next->ip = base_particle;
63                                  base_pair->next->jp = current_particle;
64                                  base_pair->next->r = sqrt(distance_squared);
65                                  base_pair->next->q = base_pair->next->r / H;
66                                  base_pair->next->w = 0.0;
67                                  base_pair->next->dwdx[0] = base_pair->next->dwdx[1] = ↩
                                        0.0;
68                                  base_pair->next->next = NULL;
69                                  base_pair = base_pair->next;
70                              }
71 
72                              base_particle->interactions++;
73                              if (i == 0) {
74                                  current_particle->interactions++;
75                              }
76 
77                          }
78                          current_bucket = current_bucket->next;
79                      }
80                  }
81                  base_bucket = base_bucket->next;
82              }
83          }
84  }
```

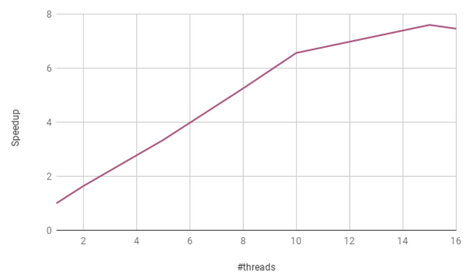
Listing A.1: Create neighbors for the lockless bucket method. The method iterates through all of the particles in the current bucket (i=0), and then the particles neighboring buckets (i=1..9). A pair is added to a linked list of pairs if the distance between the particles is lower than the interaction radius. The interaction attribute of a particle is only updated if the particle belongs to the base bucket. This is to avoid counting the interaction twice.

Appendix B

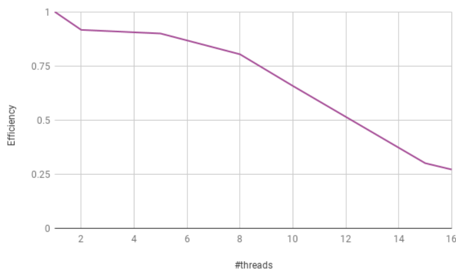
LBM Speedup and Efficiency graphs



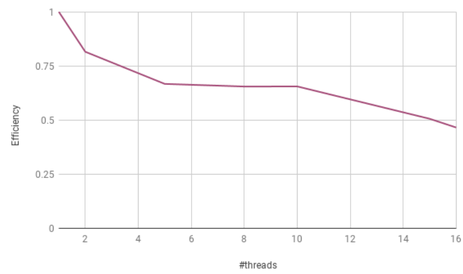
(a) Vilje Moffatt Propagate Speedup



(b) Vilje Moffatt Collide Speedup

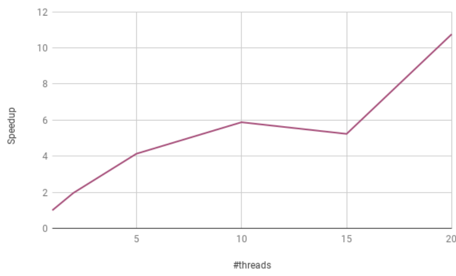


(c) Vilje Moffatt Propagate Efficiency

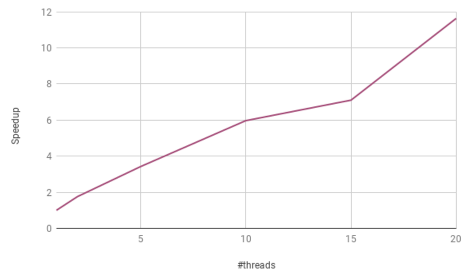


(d) Vilje Moffatt Collide Efficiency

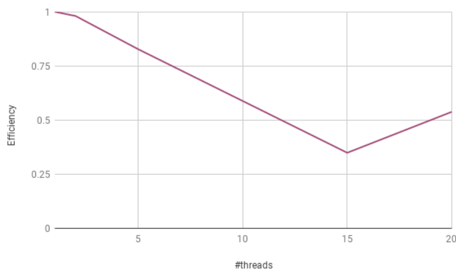
Figure B.1: Vilje Moffatt peedup and efficiency



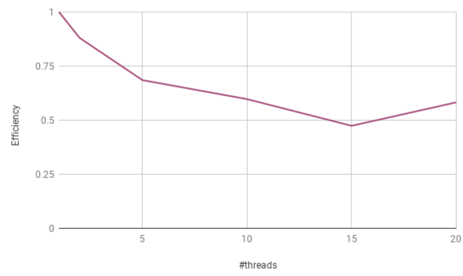
(a) EPT Moffatt Propagate Speedup



(b) EPT Moffatt Collide Speedup

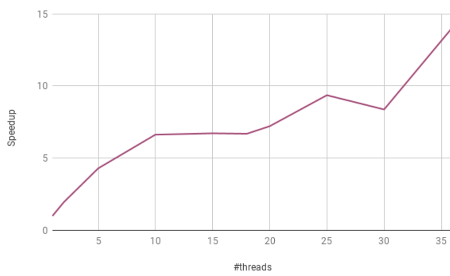


(c) EPT Moffatt Propagate Efficiency

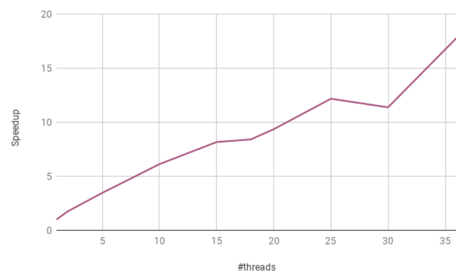


(d) EPT Moffatt Collide Efficiency

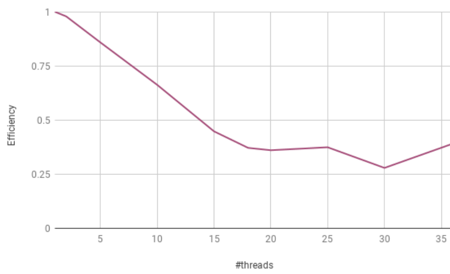
Figure B.2: EPT Moffatt speedup and efficiency



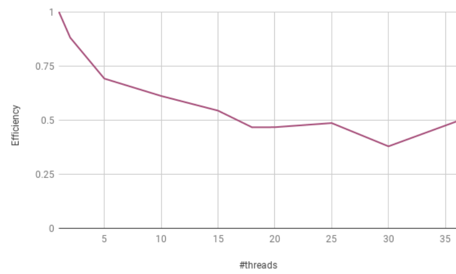
(a) EPIC Moffatt Propagate Speedup



(b) EPIC Moffatt Collide Speedup

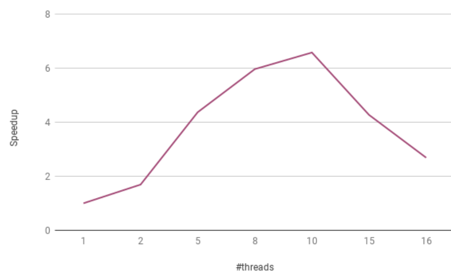


(c) EPIC Moffatt Propagate Efficiency

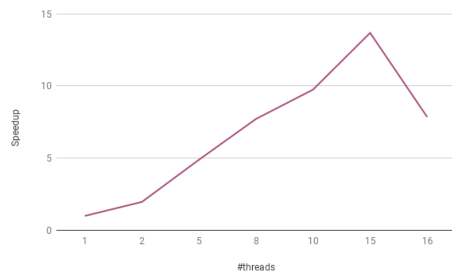


(d) EPIC Moffatt Collide Efficiency

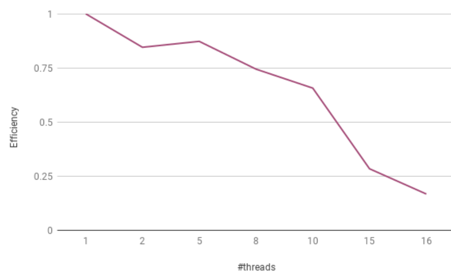
Figure B.3: EPIC Moffatt speedup and efficiency



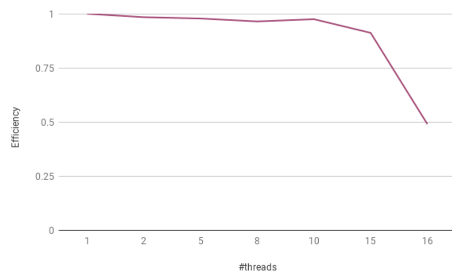
(a) Vilje Cylinder Propagate Speedup



(b) Vilje Cylinder Collide Speedup

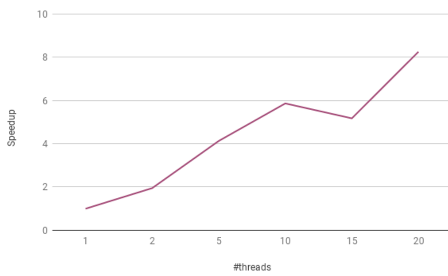


(c) Vilje Cylinder Propagate Efficiency

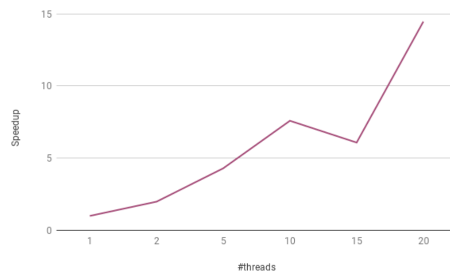


(d) Vilje Cylinder Collide Efficiency

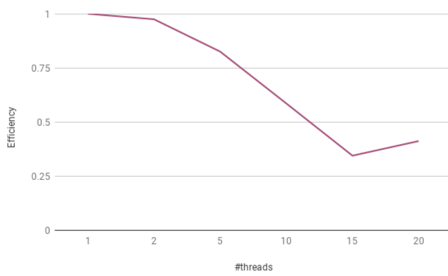
Figure B.4: Vilje Cylinder speedup and efficiency



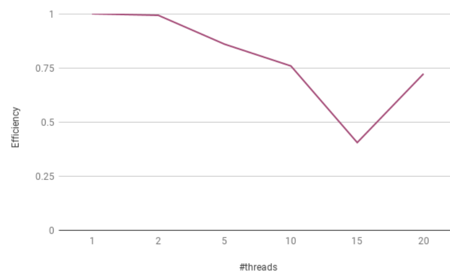
(a) EPT Cylinder Propagate Speedup



(b) EPT Cylinder Collide Speedup

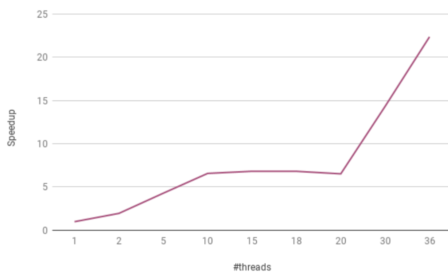


(c) EPT Cylinder Propagate Efficiency

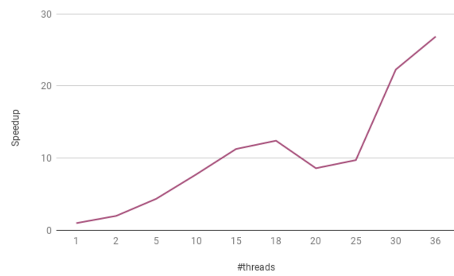


(d) EPT Cylinder Collide Efficiency

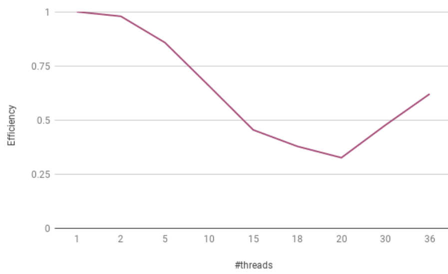
Figure B.5: EPT Cylinder speedup and efficiency



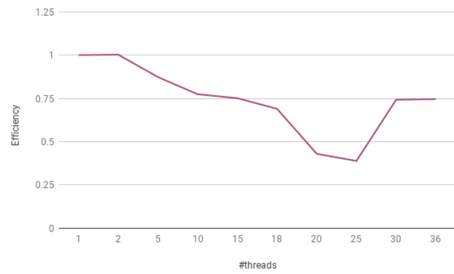
(a) EPIC Cylinder Propagate Speedup



(b) EPIC Cylinder Collide Speedup

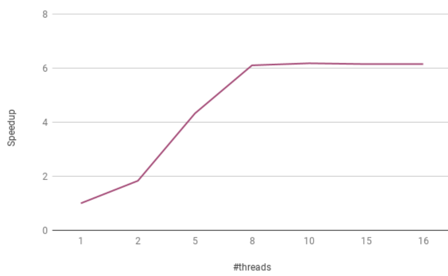


(c) EPIC Cylinder Propagate Efficiency

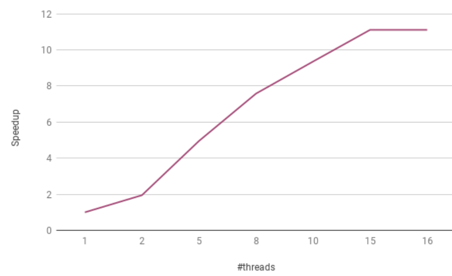


(d) EPIC Cylinder Collide Efficiency

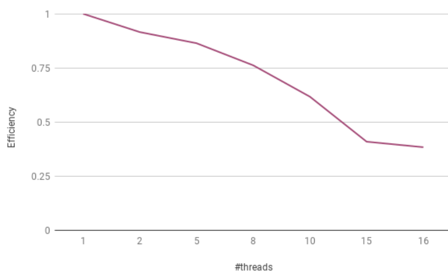
Figure B.6: EPIC Cylinder speedup and efficiency



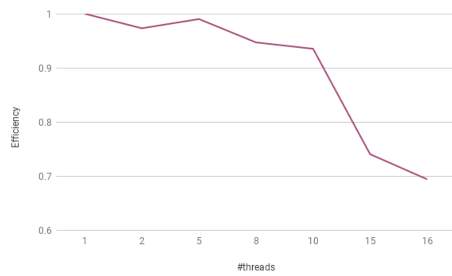
(a) Vilje Moffatt Task Propagate Speedup



(b) Vilje Moffatt Task Collide Speedup

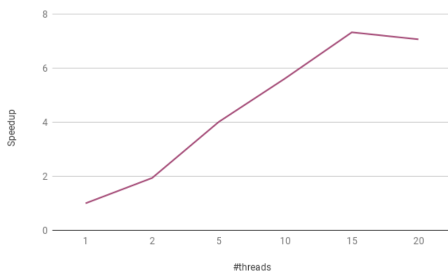


(c) Vilje Moffatt Task Propagate Efficiency

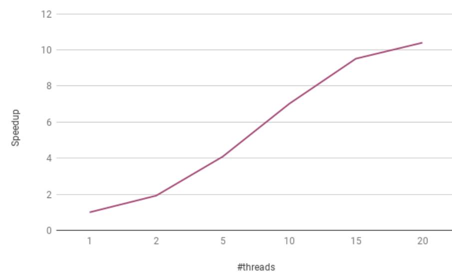


(d) Vilje Moffatt Task Collide Efficiency

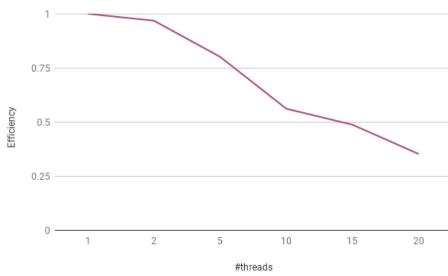
Figure B.7: Vilje Moffatt Task speedup and efficiency



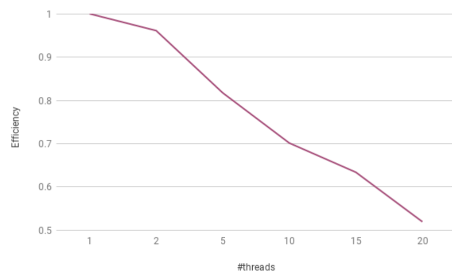
(a) EPT Moffatt Task Propagate Speedup



(b) EPT Moffatt Task Collide Speedup

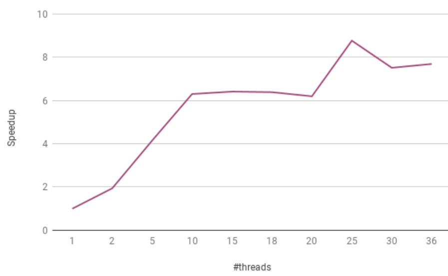


(c) EPT Moffatt Task Propagate Efficiency

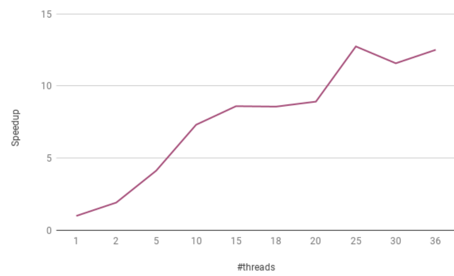


(d) EPT Moffatt Task Collide Efficiency

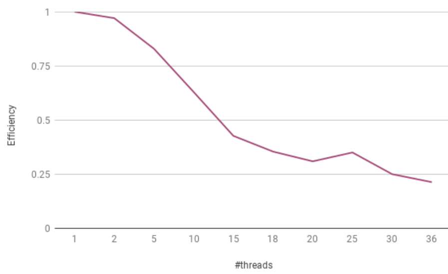
Figure B.8: EPT Moffatt Task speedup and efficiency



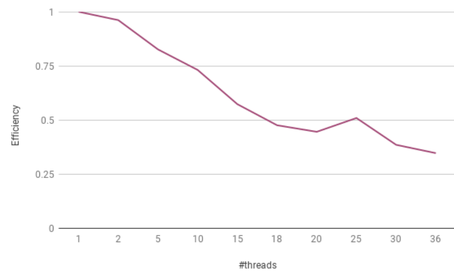
(a) EPIC Moffatt Task Propagate Speedup



(b) EPIC Moffatt Task Collide Speedup



(c) EPIC Moffatt Task Propagate Efficiency



(d) EPIC Moffatt Task Collide Efficiency

Figure B.9: EPIC Moffatt Task speedup and efficiency

Appendix C

Latency and Inverse Bandwidth Heatmaps

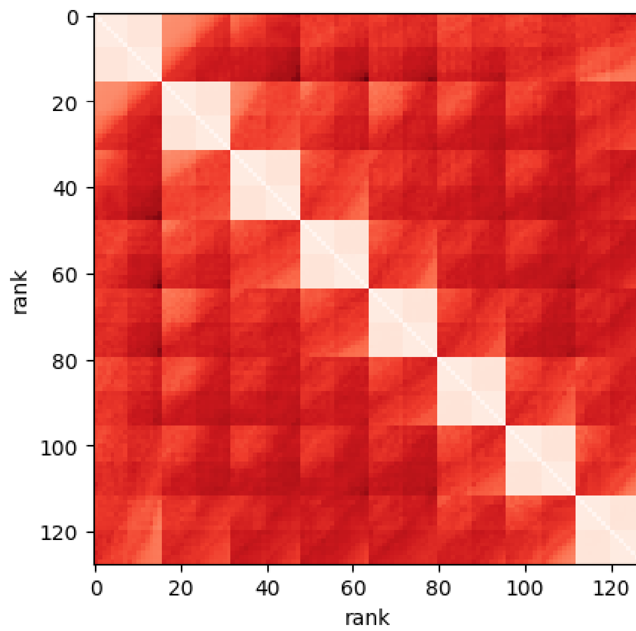


Figure C.1: Vilje, Latency, 8 Nodes, 16 ranks per node. min:9.750016e-07 max:1.789225e-05

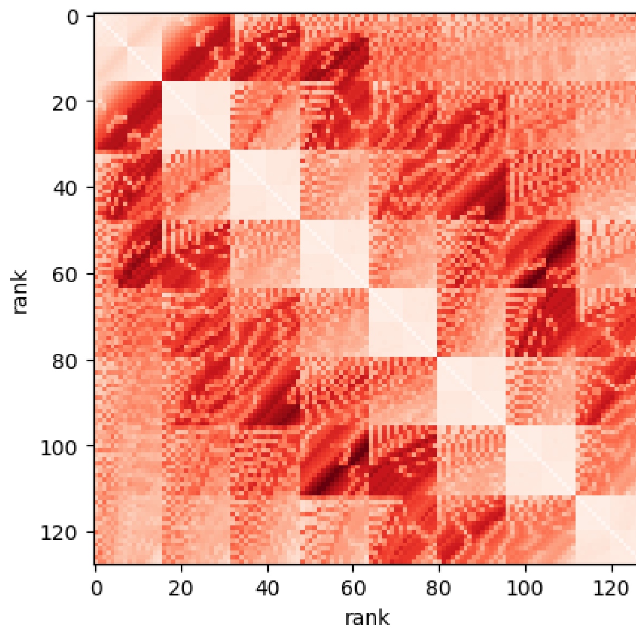


Figure C.2: VILJE, Beta Inverse, 8 Nodes, 16 ranks per node. min:1.662708e-10 max:3.360967e-09

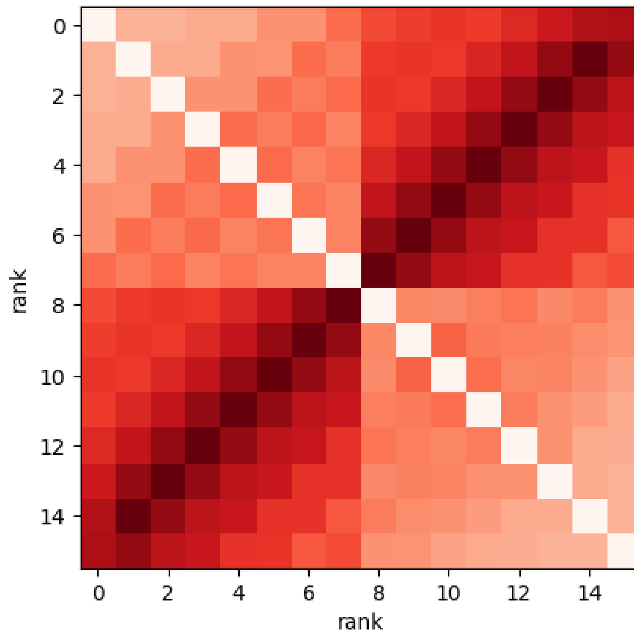


Figure C.3: Vilje, Beta Inverse, 1 Node, 16 ranks per node. min:1.669952e-10 max:6.072257e-10

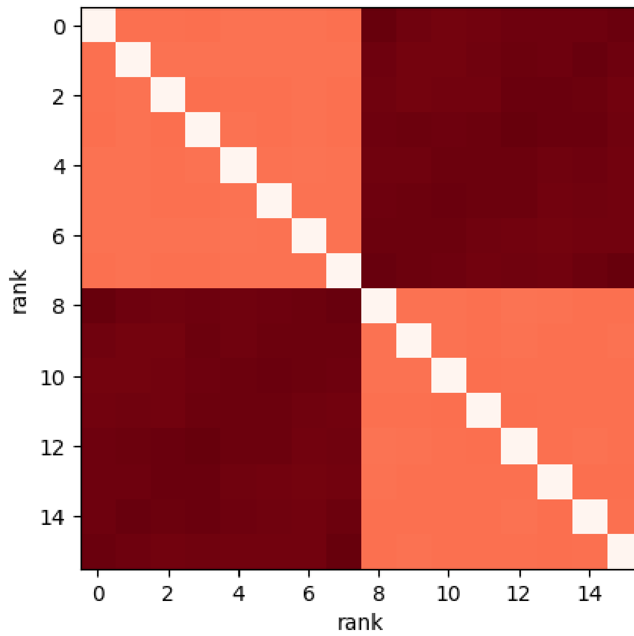


Figure C.4: Vilje, Latency, 1 Node, 16 ranks per node. min:7.157778e-07 max:1.521244e-06

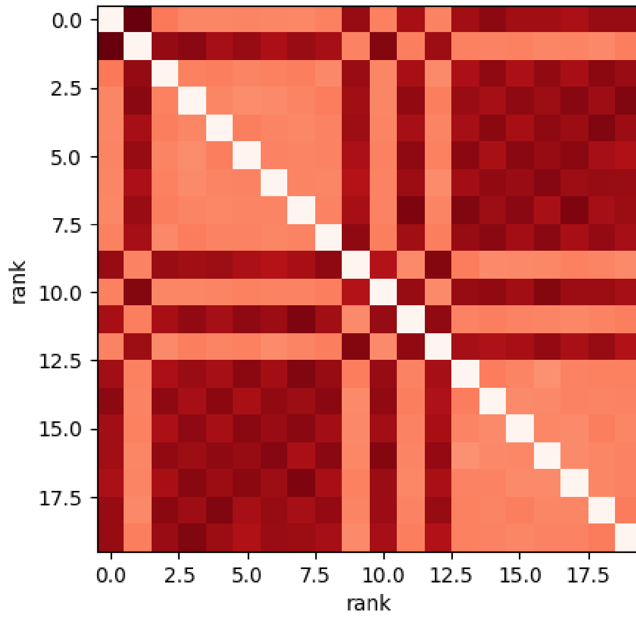


Figure C.5: EPT, Latency, 1 Nodes, 20 ranks per node. min:5.405903e-07 max:1.430154e-06

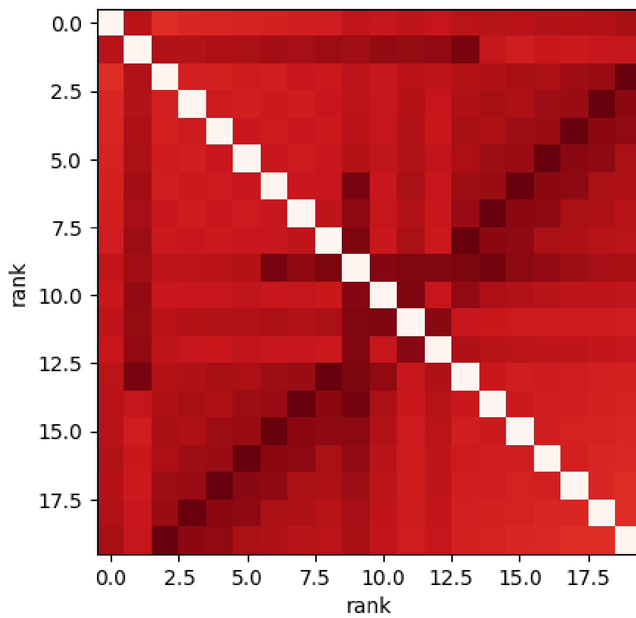


Figure C.6: EPT, Beta Inverse, 1 Nodes, 20 ranks per node. min:2.602089e-10 max:3.849905e-10

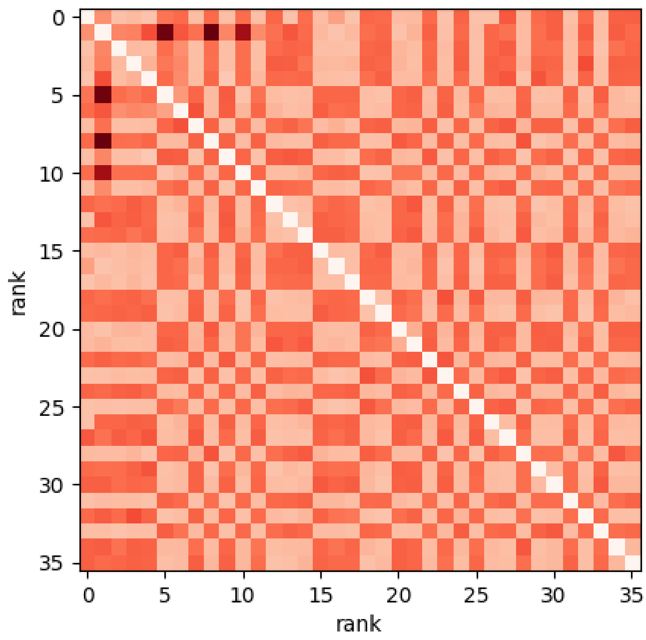


Figure C.7: EPIC, Latency, 1 Nodes, 36 ranks per node. min:5.350947e-07 max:2.622306e-06

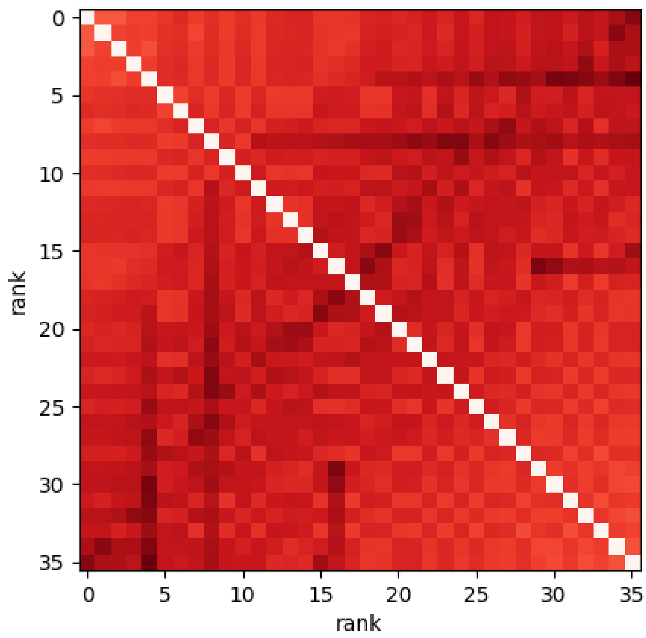


Figure C.8: EPIC, Beta Inverse, 1 Nodes, 36 ranks per node. min:2.435713e-10 max:4.464687e-10

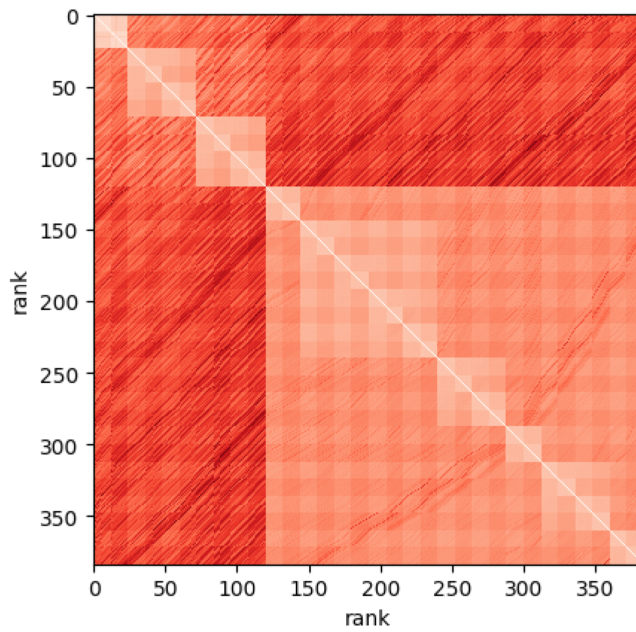


Figure C.9: Archer, Latency, 16 Nodes, 24 ranks per node. min:1.286399e-06 max:8.888543e-06

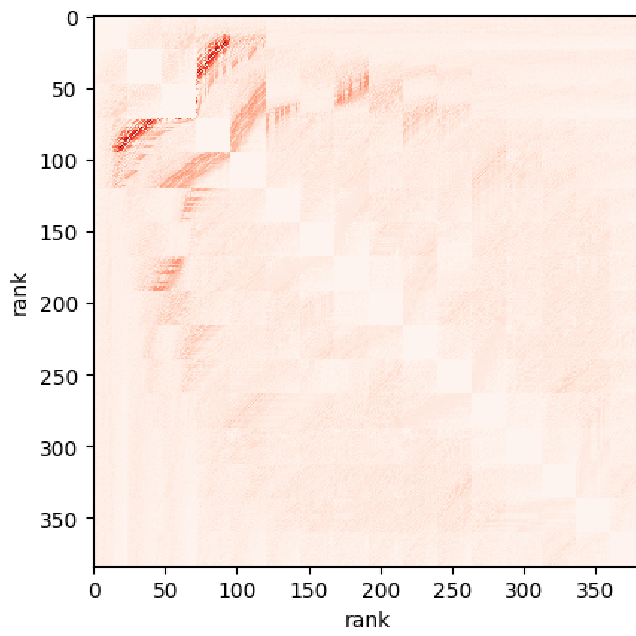


Figure C.10: Archer, Beta Inverse, 16 Nodes, 24 ranks per node.
min:1.371288e-10 max:2.839602e-08