Performance modeling of subdomain discretization techniques for SPH proxy applications

Fredrik Ås

June 22, 2021

Problem Description

This study aims to quantify the performance of two subdomain discretization techniques for a fluid proxy application and validate them experimentally with respect to parallel scaling.

Abstract

In this study we investigate the performance of two cell-grid discretization schemes for a smoothed-particle hydrodynamics (SPH) proxy application that is applied to the two-dimensional dam-break problem. The Cell+Table method uses persistent hash tables to implement grid cells. The Cell+List method uses volatile linked lists. We find that the Cell+Table outperforms the Cell+List method with a performance increase of more than 24 % and 30 % for two different test cases, respectively. We also model the SPH application using machine metrics derived from synthetic benchmarks, and find that both the computational cost and communication cost can be described according to a linear relationship. Furthermore, using the Roofline model to classify the SPH application, we find that it is memory bound which makes it suitable for subdomain discretization schemes that minimize memory traffic. We also provide a suite of benchmarking programs to measure peak FLOPS, system bandwidth, and inter-process communication cost.

Contents

Chapter 1

Introduction

Fluid simulations constitute an important part of scientific research and industry since they can be used to prototype and develop tools and model physical phenomena. A common challange with most fluid simulations however, is that they are computational intensive. Parallel methods can be used for speeding up the execution time of such simulations. In this study we investigate four aspects of a SPH proxy application with regards to parallel scaling:

Subdomain discretization

We develop and analyse the performance of two grid-cell schemes that we use to speed up local execution.

Application performance

We investigate central performance bottlenecks of the SPH application with regards to parallel scaling.

Performance modeling

We model the SPH application behaviour with regards to parallel resources.

Classification

We classify the SPH application based on its performance on three parallel systems and evaluate system performance based on a set of synthetic benchmarks.

The fundamental equation of modeling, proposed by Barker *et al.* [1], forms the basis of our analysis throughout this study. The equation models program performance as the sum of computational cost and communication cost, minus any overlapping execution that can be performed during communication,

$$T = T_{\text{Comp}} + T_{\text{Comm}} - T_{\text{Overlap}}. \tag{1.1}$$

We focus on the first two terms of Equation 1.1 in this study, namely T_{Comp} and T_{Comm}.

The chapters are organized as follows. Chapter 2 introduces the SPH model and the dam-break problem that forms the experimental basis for the performance analysis, as well as performance models, communication models, and architectural models that we used to analyse and validate our results. Chapter 3 presents implementation details for two different cell-grid discretization methods and synthetic benchmarks used to derive machine metrics. In Chapter 4 we present our experimental findinds, performance models, and model validations. Chapter 5 gives a summary of the main results and a set of recommendations based on our experimental results.

Chapter 2

Background

2.1 Smoothed Particle Hydrodynamics

Smoothed particle hydrodynamics (SPH) is a numerical method of simulating fluid using particles with fluid properties such as velocity, density, pressure, *etc.*. The method was first introduced by Gingold and Monaghan [2] and Lucy [3] in 1977, who used the method to simulate compressible flow problems in astrophysics. An extension to the SPH method was provided by Monaghan [4] in 1994, which included a mathematical model for weakly compressible SPH (WCSPH). A number of different SPH variations have been introduced so far. In this study we focus on the method developed by Ozbulut [5] *et al.* in 2014, which applies the WCSPH method to the two-dimensional dam-break problem. We will refer to this method as SPH throughout the study.

SPH simulates fluid using interpolation points, called particles, with fluid properties that are defined as a small fractions of a medium with finite volume. Two particles i and j may interact if they are within the interaction radius r_{ij}. The interaction radius is normalized by the smoothing length h and applied to a weighting function to determine the appropriate interaction force. The weighting function is defined as,

$$W(R) = \alpha_d \begin{cases} (3-R)^5 - 6(2-R)^5 + 15(1-R)5, & 0 \leq R < 1 \\ (3-R)^5 - 6(2-R)^5, & 1 \leq R < 2 \\ (3-R)^5, & 2 \leq R < 3 \\ 0 & R \geq 3 \end{cases} \tag{2.1}$$

where $R = r_{ij}/h$ is the normalized interaction radius between particle i and j. The α_d coefficient determines the dimensionality of the problem. For our implementation, α_d is set to $7/(478\pi h^2)$ which corresponds to a two-dimensional simulation.

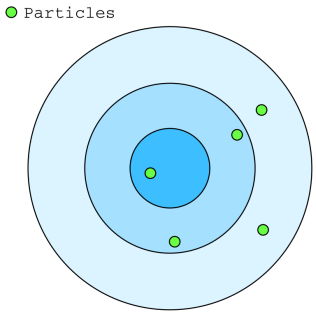


Figure 2.1 Particle interactions. Each field represents one step in the normalized distance between particle i and j.

2.2 The Dam-break Problem

In this study the SPH method is used to simulate the two-dimensional dam-break problem. Figure 2.2 shows the simulation for a few selected timesteps. At the start of the simulation, the dam is arranged to the left of the tank in a rectangular shape of width L and height T. The tank has a length B and infinite height so that there is no upper boundary to the tank. The simulation begins by applying the SPH kernel such that the dam breaks and hits the rightmost wall of the tank.

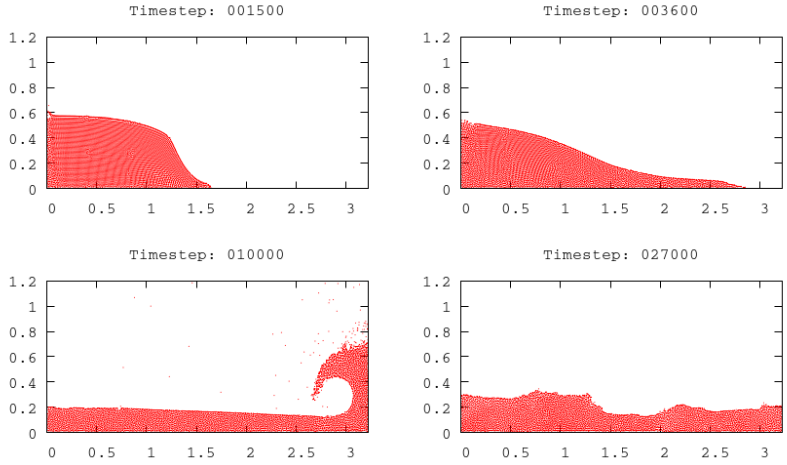


Figure 2.2 Two-dimensional dam break simulation.

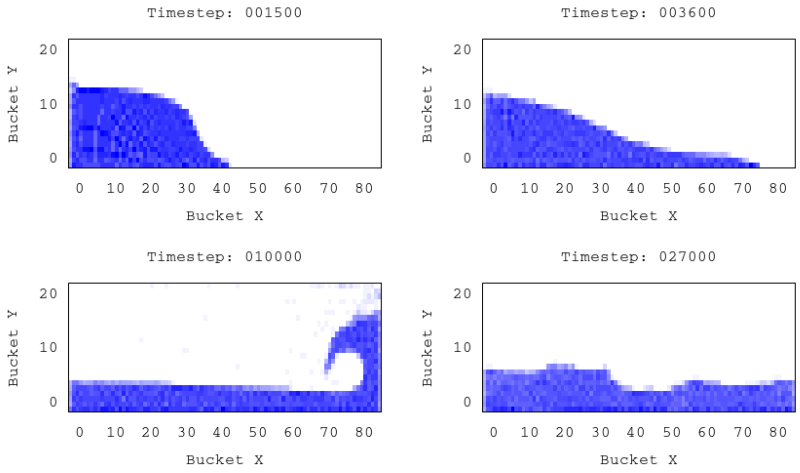


Figure 2.3 Visualization of cell density, *i.e.* number of particles per cell.

2.3 Application Scalability

In this section we present the analytical framework used to discuss application scalability and performance.

2.3.1 Parallel Speedup and Efficiency

Pacheco [6] defines *parallel speedup* as the relationship between an application's serial execution time T_{Serial} and parallel execution time T_{Parallel},

$$S = \frac{T_{\text{Serial}}}{T_{\text{Parallel}}}. \tag{2.2}$$

By incorporating the number of cores p into the model for parallel speedup, Pacheco defines parallel efficiency as

$$E = \frac{T_{\text{Serial}}}{p \times T_{\text{Parallel}}}. \tag{2.3}$$

We achieve *perfect parallel efficiency* in the ideal case where the serial execution time is reduced by a factor equal to the number of cores. In this case we have *linear speedup*, where running the application on p cores will be p times faster than its serial counterpart. The parallel efficiency in this case is

$$T_{\text{Parallel}} = T_{\text{Serial}}/p. \tag{2.4}$$

Intuitively, parallel efficiency is a measure of how much each core contributes the overall speedup of a parallel application. Low efficiency typically indicates that parallelism is introducing overhead that diminish performance gains.

2.3.2 Amdahl's Law

Amdahl's law [7] models the parallel speedup of an application for a fixed problem size. Consider an application consisting of a serial fraction s and a perfectly parallelizable fraction $1-s$ running on a system with p cores, then Amdahl defines parallel speedup as

$$S = \frac{s + (1-s)}{s + (1-s)/p} = \frac{1}{s + (1-s)/p}. \tag{2.5}$$

An application is scaled in strong scaling mode if we increase the number of cores while keeping the problem size constant. If the parallel efficiency for the application remains constant in strong scaling mode, we say that the application is strongly scalable.

2.3.3 Gustafson's Law

Gustafson's law [8] models the parallel speedup of an application where the problem size is scaled with the number of cores. Consider an application with a serial fraction s and a perfectly parallelizable fraction $1-s$ running on a system with p cores, then Gustafson defines the parallel speedup as

$$S = \frac{s + (1-s)p}{s + (1-s)} = s + (1-s)p. \tag{2.6}$$

An application is scaled in weak scaling mode if we increase both the number of cores and the problem size. If the parallel efficiency for the application remains constant in strong weak mode, we say that the application is weakly scalable.

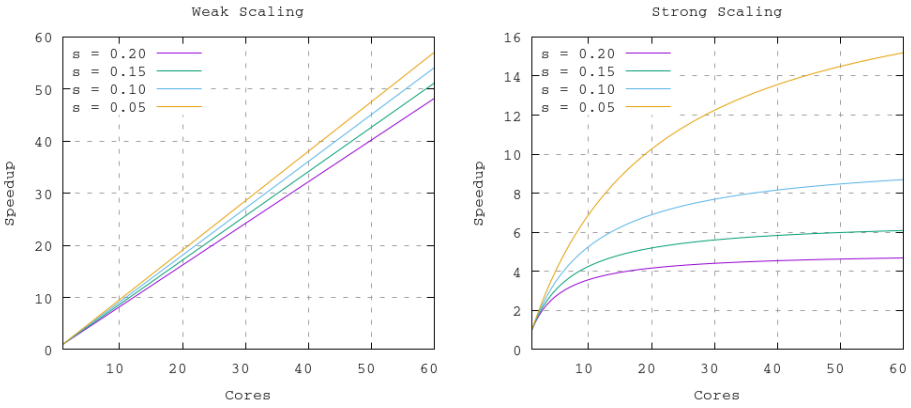


Figure 2.4 Weak scaling (left). Strong scaling (right).

2.4 Communication Models

2.4.1 The Hockney Model

The Hockney model [9] describes communication cost as the sum of the setup latency α and the inverse bandwidth β. The setup latency is the overhead of initializing the communication channel between the sender and receiver. The inverse bandwidth is the time it takes to send M bytes between the sender and receiver. The total cost of sending a message of size M can then be modeled according to the following equation,

$$T_{\text{Comm}} = \alpha + M\beta, \tag{2.7}$$

where M is the message size in bytes. The machine parameters α and β can be estimated by timing a series of message transmissions between the sender and receiver. We can adjust the number of round-trips N and the message size M such that either the setup latency of the inverse bandwidth becomes the dominating factor,

$$\{\alpha, \beta\} = \frac{t_1 - t_0}{2MN}. \tag{2.8}$$

The α and β parameters can be estimated in two ways. Both approaches uses a series of round-trip communication measurements to derive the average communication latency L by sending a message of size M back and forth between the sender and receiver N number of times,

Modern heterogeneous multi-node systems introduce machine parameters that may differ based on the locality of the sender and receiver. Lastovetsky [10] extends the Hockney model to account for heterogeneity by adding model parameters for each individual communication pair (i, j),

$$T = \alpha_{ij} + M\beta_{ij}. \tag{2.9}$$

The extended Hockney model provides a latency matrix that describes the communication cost between every communication pair on the system.

2.4.2 BSP

The bulk-synchronous parallel (BSP) model, proposed by Valiant [11], is a communication model for designing portable algorithms for parallel systems. The model describes communication overhead in terms of three parameters; a number of processors performing local computation or memory operations, a router that delivers point-to-point messages between processors, and facilities for synchronizing all or some of the processors at regular time units L. Computations are divided into a sequence of supersteps where each processor is assigned a task consisting of a combination of local computation and message transmissions. Global synchronization is performed at every unit L to determine if all processors have completed the current superstep. If the superstep is not completed, all processors continue with the next superstep, otherwise a period of L time units is allocated to continue the superstep. Every processor may send and receive at most h messages in each superstep. This is called a h-relation. With an inverse bandwidth g for point-to-point communication and a setup latency s, a total of $gh + s$ time units are required to realize such a h-relation. Since BSP is a synchronous model, messages that are sent in a superstep may only be used in the next superstep, even if there is remaining time in the current superstep. Another limitation of the BSP model is that is assumes hardware support for synchronizing processors at the end of every superstep.

2.4.3 LogP

The LogP model, developed by Culler *et al.* [12], is a parallel bridging model for designing and analyzing portable, parallel applications that run on distributed systems where processors communicate through point-to-point messages. The model is an improved version of the BSP model. The model predicts the communication latency in terms of latency (L), overhead (o), gap per message (g), and number of processors (P). The latency, L, is the upper-bound setup delay incurred by sending a single message between two points. The overhead, o, is defined as the time period a processor is engaged in sending or receiving a message and cannot perform other operations. The gap parameter, g, is the time delay between consecutive message transmissions. A message is sent as a small number of data units (words) m. The model assumes finite network capacity such that at most $\lceil L/g \rceil$ messages can be in transit between any processor at any time. If the limit is exceeded, the processors involved in the communication exchange is stalled until the message can be sent without exceeding the limit. Processors work asynchronously and the message latency is therefore considered unpredictable.

Lastovetsky *et al.* [10] describes large message transmissions with the LogP model as a series of small data-unit transfers with constant time delay between each transfer. The total communication latency, T, for transmitting a series of small messages, m, can then be expressed as

$$T = L + 2o + (m-1)g. \tag{2.10}$$

Alexandrov *et al.* [13] proposes the LogGP model which extends the original LogP model by incorporating large messages. The extended model introduces a gap-per-byte parameter, G, which is the latency of sending M bytes between two points. The model includes the original gap parameter, g, to represent the constant time delay between consecutive message transmissions m. The total communication latency as described by the extended LogGP model can then be expressed as

$$T = L + 2o + (m-1)g + (M-1)G. \tag{2.11}$$

Kielmann *et al.* [14] proposes another extension, the PLogP model, where all parameters except for the latency, L, are piecewise linear functions of the message size and their definitions differ slightly from the original LogP model.

2.5 Performance Models

2.5.1 Stream

Stream is a synthetic benchmark developed by McCalpin [15]. The benchmark measures the bandwidth between the top-level cache (L3 on most current systems) and main memory by timing four strided vector kernels with large operands. The kernels are listed in Equation 2.12. The timings of Triad kernel is used as the basis for the bandwidth results as well as to calculate the machine balance of the system.

$$\begin{aligned} \text{(Copy)} \quad & c_i = a_i \\ \text{(Scale)} \quad & b_i = kc_i \\ \text{(Add)} \quad & c_i = a_i + b_i \\ \text{(Triad)} \quad & a_i = b_i + kc_i \end{aligned} \tag{2.12}$$

McCalpin defines machine balance as the ratio of the number of peak float-point operations per cycle to the number of sustained memory operations per cycle. The machine balance of a system is a measure of its latency tolerance. High-throughput applications typically have high latency tolerance since they perform a high number of floating-point operations per memory operation. Conversely, low-throughput applications that have more memory traffic have low latency tolerance. A system with high machine balance is capable of performing a large number of floating-point operations per memory operation, which in turns makes the system inclined towards high-throughput applications. Extra thread-level parallelism typically yields increased latency tolerance in parallel applications on shared memory systems, albeit at the expense of increased machine balance.

2.5.2 Roofline

The Roofline model, developed by Williams *et al.* [16], is a visual performance model for comparing and identifying kernel optimizations on different systems. The Roofline performance equation is given by

$$P = \min\{\pi, \beta I\}, \tag{2.13}$$

where π is the system's theoretical peak floating-point performance per unit time and β is the system's theoretical peak bandwidth. I is the kernel's operational intensity, which describes the kernel's performance in terms of floating-point operations per byte. The kernel's operational intensity determines its constraints in relation to the target system, *i.e.* if it is memory bound, $I < \pi/\beta$, or compute bound, $I > \pi/\beta$.

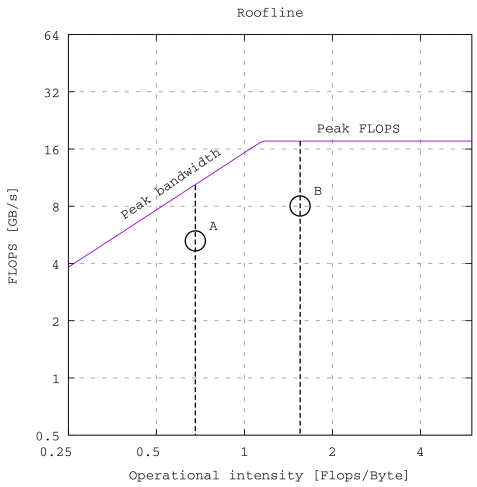


Figure 2.5 Roofline model for two applications on a theoretical system. Application A is memory bound. Application B is compute bound.

2.5.3 DGEMM

The DGEMM benchmark measures the peak floating-point performance of a system by timing a level 3 BLAS kernel [17] of the form

$$\mathbf{C} = \alpha\mathbf{AB} + \beta\mathbf{C}. \tag{2.14}$$

General matrix-matrix multiplications are common in scientific applications and the DGEMM benchmark is therefore a useful metric for describing the system in terms of floating-performance. Both hardware and software optimizations are required to obtain a reasonable percentage of the

system's theoretical floating-point performance. There exist multiple libraries suitable for the task, *e.g.* OpenBLAS [18], which is an open-source implementation of the BLAS and LAPACK APIs that utilizes software and hardware optimizations for specific hardware architectures. By timing the execution of a DGEMM kernel with three matrices A ($M \times N$), B ($N \times Q$), and Q ($M \times Q$), we get the system performance by dividing the number of floating-point operations by the total execution time,

$$\frac{2MNQ}{t_1 - t_0}. \tag{2.15}$$

2.6 Architectural Models

2.6.1 Multiprocessors

A multiprocessor, as defined by Hennessy and Patterson [19], is a processor with multiple cores that are coordinated and controlled by a single operating system and share memory through a shared address space. Multiprocessors can be categorized by memory organization and interconnect strategy. Uniform memory access (UMA) multiprocessors, also called symmetric multiprocessors (SMP), typically have a small number of cores. With this memory organization the memory is centralized, and thus the memory access time is independent of the memory location relative to the core. Non-uniform memory access (NUMA) multiprocessors, also called distributed shared-memory multiprocessors, typically have a large number of cores compared to UMA multiprocessors. With this memory organization, memory is distributed among multiple cores, and thus the memory access time depends on the memory location relative to the core. The main advantage with NUMA architectures is that they offer higher memory bandwidth compared to UMA systems, and thus facilitate higher core counts. All multi-socket systems are NUMA architectures, however single-socket NUMA system are also quite common. Both UMA and NUMA architectures utilize thread-level parallelism through a shared address space. Therefore, a memory reference can be made by any core to any memory location as long as it has the correct access privileges. This is not the case for cluster and warehouse-scale computers, where multiple nodes are connected through a network. Such systems rely on message-passing protocols to transfer data between nodes, discussed later.

2.6.2 Memory Organization

A common type of memory organization in modern computers is dividing memory into a hierarchy where the smaller and faster (more expensive) memory typically sits close to the processor, while the larger and slower (less expensive) memory sits farther away from the processor. Registers and cache memory are embedded into the processor itself, while main memory is connected to the processor through a memory bus. Disk or flash storage is typically connected to RAM through an I/O bus. Figure 2.6 shows a hierarchical memory configuration. Registers are the fastest memory available to the processor. Data written to a register is typically available at the next processor clock cycle. The cache is commonly divided into three levels where L1 and L2 are local to a single core and L3 is shared between all cores on the same socket.

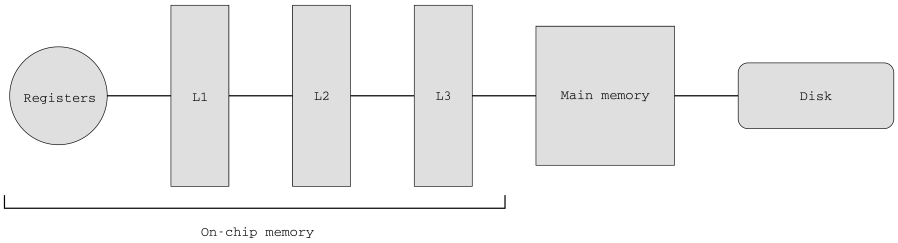


Figure 2.6 Memory hierarchy.

2.6.3 Locality

The principle of locality is the observation that subsequent memory references made by the processor tend to be close in space and time. There are two main types of reference locality: spatial and temporal. A piece of code that makes subsequent memory references to locations that are close to together within a short period of time is said to exhibit spatial locality. A piece of code that makes subsequent memory references to the same location within a short period of time is said to exhibit temporal locality. A third type of locality called network locality is typically seen on NUMA systems where the memory access time depends on the memory location relative to the processor.

2.6.4 Affinity

Affinity is the scheme of which threads or processes are mapped to individual cores. With a compact affinity scheme, consecutive threads are mapped to the same socket. With a scattered affinity scheme, consecutive threads are mapped to different sockets. Applications with a high degree of data sharing between consecutive threads may benefit from a compact affinity scheme due to increased cache accuracy. Applications with a low degree of sharing between threads may benefit from a scattered affinity scheme to avoid increased cache-miss rates caused by cache contention. In general, the performance benefits of a specific affinity scheme depends on both the memory access pattern of the application and the target architecture. Other affinity schemes are also possible, but they are beyond the scope of this study.

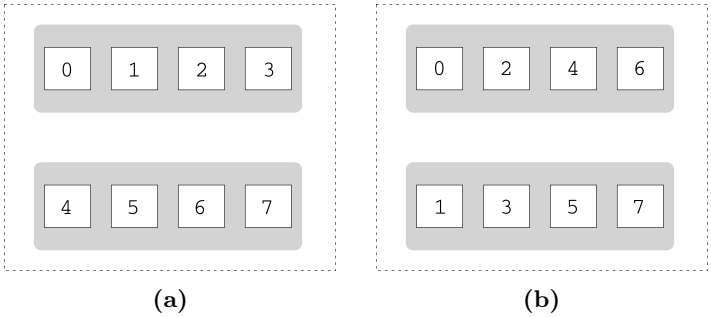


Figure 2.7 (a) Scattered affinity scheme. (b) Compact affinity scheme.

2.7 Programming Models

2.7.1 MPI

The message-passing interface (MPI) project [20] is a an open-source point-to-point communication protocol for distributed memory systems. There exists multiple implementations of the protocol, *e.g.* Intel MPI, MPICH, and OpenMPI. OpenMPI and MPICH are open-source implementations while Intel MPI is developed by Intel. We hereby assume OpenMPI as the implementation in question when discussing implementation specific details of the MPI protocol.

MPI enables data-level parallelism by giving each process, or rank, its own designated memory area. Upon initialization, all ranks are associated with a default communicator, `MPI_COMM_WORLD`. A communicator specifies a group of ranks that can transmit messages to each other. The default communicator holds all ranks that were created upon program initialization. Custom communicators may be used, *e.g.*, to specify a subset of ranks to be included in a collective communication routine or synchronization barrier. Since ranks are not aware of each other and only have access to their own separate memory areas, data must be explicitly transmitted between ranks through point-to-point communication. This also means that the problem domain of the application needs to be decomposed and distributed among ranks at program initialization.

Unlike in thread-level parallelism, race conditions cannot occur due to data-level parallelism since memory is not shared between ranks. Deadlocks may occur however, *e.g.*, if a rank for some reason fails to enter a synchronization barrier. (Usually caused by a bug introduced by the programmer.)

2.7.2 OpenMP

OpenMP [21] is a industry standard API for shared-memory programming. OpenMP enables thread-level parallelism in the form of compiler directives that are inserted by the programmer to tell the compiler to execute the code in parallel. All directives starts with `#pragma omp`, followed by a construct and other options needed to execute the code in parallel. A common construct in the `parallel` construct, used to execute loops in parallel.

2.8 Data Structures

2.8.1 Structures

A structure, or struct for short, is a native C data type that groups a collection of variables, called members, together into a single unit. Member names do not conflict with other variable names since they are uniquely identified in conjunction with the struct tag. The compiler may insert padding to align struct members to their natural address boundaries. The order in which struct members are declare therefore affects the size of the struct, as shown in Figure 2.8.

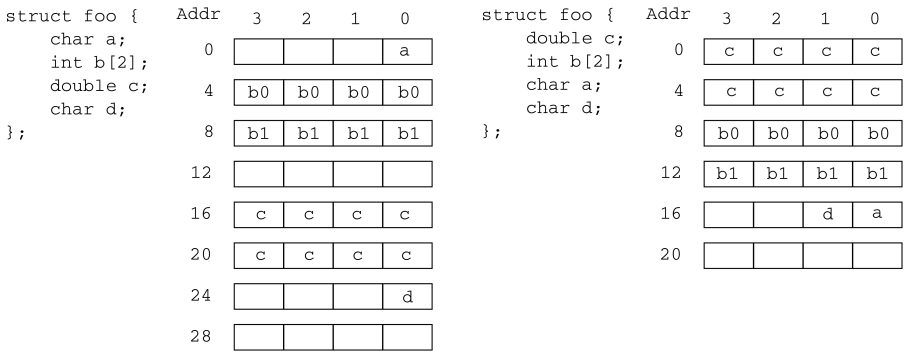


Figure 2.8 Memory layout of a C struct. The struct to the left is 32 bytes; the struct to the right is 24 bytes.

2.8.2 Linked Lists

A linked list is a linear collection of elements that are linked together in the order they are inserted. In C, a linked list is typically implemented by series of structs that are linked together by pointers. For a singly linked list, each struct contains a pointer to the next struct in the list, and other optional members for storing data, as well as a head pointer that points to the start of the list. A doubly linked list has an extra pointer in the opposite direction between each struct in the list. This may increase the performance of certain list functions. The insertion order in a linked list is preserved since the position of every struct depends only on the insertion index. The linked list data structure forms the basis for many other data structures such as stacks, queues, and associative arrays.

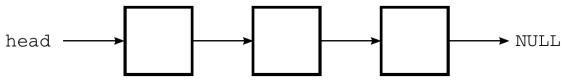


Figure 2.9 Singly linked list.

2.8.3 Hash Tables

A hash table is a collection of key-value elements where every value is uniquely identified by its corresponding key. In C, a hash table can be implemented using an array of linked lists. A hash function is used to select a list to append a new entry. A hash table does not preserve insertion order since the hash function determines the placement of each key-value element.

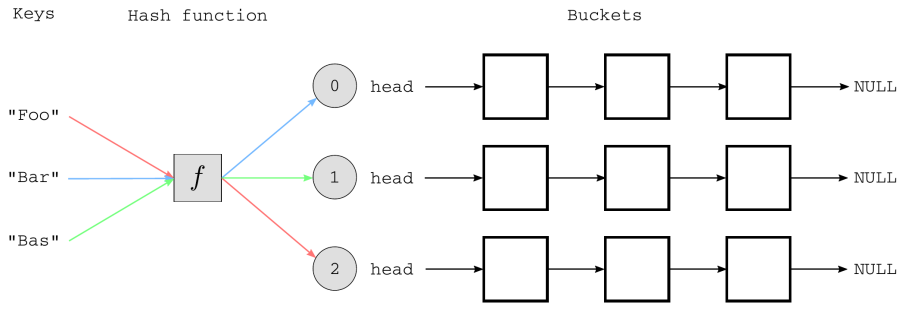


Figure 2.10 Hash table.

Table 2.1 Average asymptotic data-structure performance.

Data type	Search	Insert	Delete
Array	$O(n)$	$O(n)$	$O(n)$
Singly linked list	$O(n)$	$O(1)$	$O(1)$
Hash table	$O(1)$	$O(1)$	$O(1)$

2.9 Statistical Analysis

In this section we define various statistical methods that we use to evaluate our measurements.

2.9.1 Mean

The mean of a dataset is the average value of all the elements in the dataset. There are different approached to compute the mean of a dataset, the most common one being the arithmetic mean (AM). The arithmetic mean is the sum of all the elements in the dataset divided by its size, *i.e.*

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} x_i. \tag{2.16}$$

A second method for computing the average of dataset is the geometric mean (GM). The geometric mean is the nth rooth of the product of all dataset elements, *i.e.*

$$\bar{y} = \left(\prod_{i=1}^{n} x_i\right)^{1/n}, \quad x_i > 0. \tag{2.17}$$

Another common method is called the harmonic mean (HM). The harmonic mean is the size of the dataset divided by the reciprocal sum of each dataset element, *i.e.*

$$\bar{y} = n \left(\sum_{i=1}^{n} \frac{1}{x_i} \right)^{-1}, \quad x_i > 0. \tag{2.18}$$

The inequality AM $\geq$ GM $\geq$ HM holds true for the same dataset.

2.9.2 Coefficient of Determination

We commonly use residuals to analyze the deviation between a predicted value and an observed value, or to get a qualitative measure of the dispersion around the dataset mean. The total sum of squares describes the dispersion around the dataset mean, *i.e.*

$$SS_{\text{Total}} = \sum_{i=n}^{n} (y_i - \bar{y})^2. \tag{2.19}$$

The standard deviation of a dataset is defined as the the root of the total sum of squares divided by its size,

$$\sigma = \sqrt{\frac{SS_{\text{Total}}}{n}}. \tag{2.20}$$

The residual sum of squares gives a measure of the deviation between the dataset and dataset prediction, *i.e.*

$$SS_{\text{Residual}} = \sum_{i=n}^{n} (y_i - \hat{y}_i)^2. \tag{2.21}$$

The coefficient of determination can then be defined as the relationship between Equation 2.21 and Equation 2.19,

$$R^2 = 1 - \frac{SS_{\text{Residual}}}{SS_{\text{Total}}}. \tag{2.22}$$

The inequality $SS_{\text{Residual}} \leq SS_{\text{Total}}$ holds such that R^2 will always be in the range $[0.0, 1.0]$. A high value (close to 1.0) implies little dispersion between the prediction model and the dataset, while a low value indicates dispersion between observed and predicted values.

Chapter 3

Methodology

3.1 Problem Domain

Three variables decide the problem size of the application, namely the dam width L, the dam height T, and the tank length B. These variables are multiplied by a scaling factor, Scale, which allows for re-sizing the problem domain by adjusting a single parameter. The tank has no roof that particles can collide with. Instead, the height of the tank, which is set to $1.5T$, constitutes a virtual upper boundary that particles are stored in if they move outside the domain.

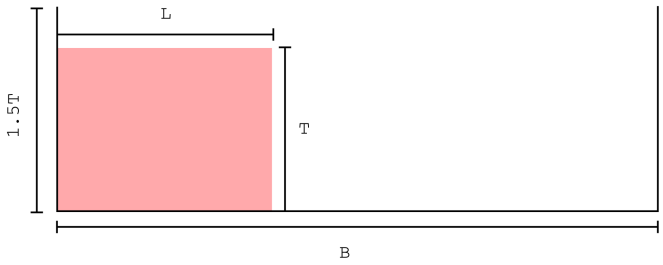


Figure 3.1 Problem dimensions.

The resolution coefficient, Delta, determines the distribution density, *i.e.* the number of particles, within the initial dam geometry. The total number of particles, $N_x \times N_y$, is given by Equation 3.1.

$$\begin{aligned} N_x &= 1 + L/\text{Delta} \\ N_y &= 1 + T/\text{Delta} \end{aligned} \tag{3.1}$$

Each subdomain is discretized into cells. The cell size, R_{Cell}, is set equal to the particle interaction radius, R_{Particle}. Thus, any particle may interact with particles in 9 cells, as shown in Figure 3.2.

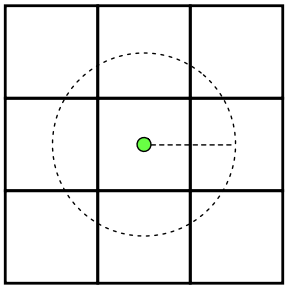


Figure 3.2 Particle interaction radius.

The number of ranks, Size, is limited by the cell size R_{Cell} and the tank width B, such that

$$1 \leq \text{Size} \leq \text{ceil}\left\{ \frac{B}{R_{\text{Cell}}} \right\}. \tag{3.2}$$

With Scale set to 1.0, the maximum number of ranks is 81, with Scale set to 2.0, the maximum number of ranks is 162, and so forth. Thus, the maximum number of possible ranks equals the number of horizontal cells in the tank. Increasing the number of ranks even further would not be possible since that would require two or more ranks to share the same cell. Implementation specific parameters are discussed in the following sections. Figure 3.3 shows the relationship between the scale, the maximum number of ranks, and the number of particles.

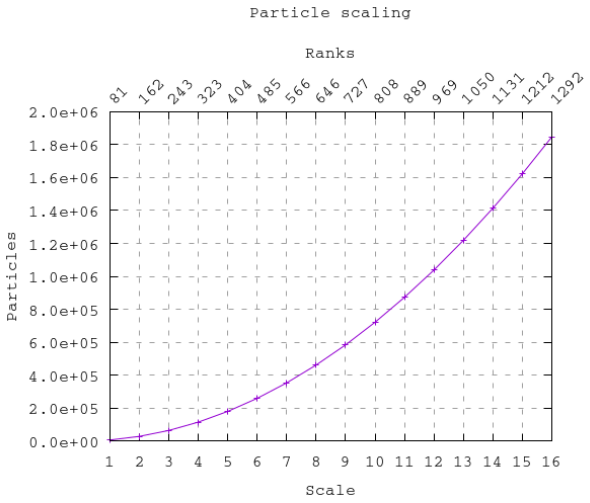


Figure 3.3

The tank is vertically decomposed among ranks in increasing order according to Figure 3.4a. This decomposition scheme is the most beneficial for this specific problem domain since it assures that the workload is distributed evenly among every rank. A horizontal distribution would lead to an unbalanced workload distribution since most of the simulation takes place in the lower portion of the tank, leaving several ranks with little or no work as the dam stabilizes. Another disadvantage with a horizontal decomposition for this specific problem domain, is that each subdomain would be smaller compared to the same rank count with a vertical decomposition.

Each rank has one neighbor east and one neighbor west, except for the first and last ranks which have only one one neighbor due to the tank boundaries. The communication pattern for this rank topology is shown in Figure 3.5. We use `MPI_PROC_NULL` as a placeholder for the source and destination for the edges of the tank. `MPI_PROC_NULL` is a dummy value that, when used with point-to-point communication routines, returns as soon as possible without modifying the source or destination buffers in any way. This simplifies the implementation details concerning communication around the tank boundaries.

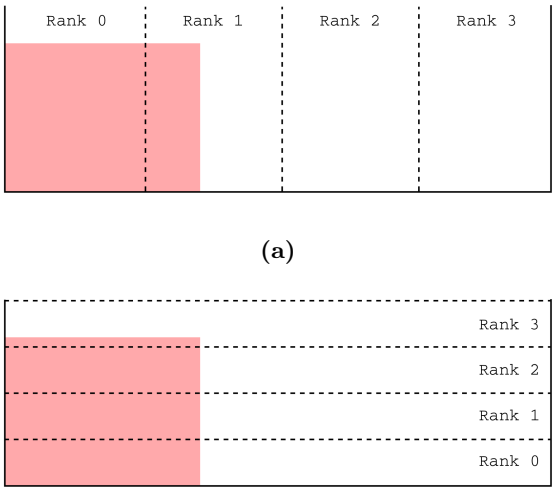


Figure 3.4 Two domain-decomposition schemes. (a) Vertical decomposition. (b) Horizontal decomposition.

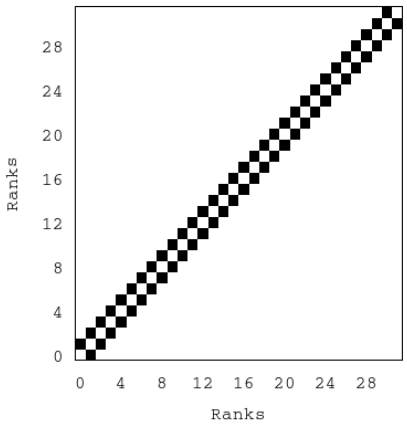


Figure 3.5 Rank communication pattern where every rank has two adjacent neighbors.

Table 3.1 SPH applicaton parameters.

Variable	Description
B	Tank length
T	Dam height
L	Dam width
H	Smoothing length
R_{Particle}	Interaction radius
R_{Cell}	Cell size
Scale	Scaling coefficient
Delta	Resolution coefficient
Size	Number of ranks
Rank	Rank id
Idx	Particle id

3.2 Subdomain Discretization

The subdomain of each rank is discretized into a cell grid to minimize the cost of searching for particle interactions. With no cell grid we would have to parse $n-1$ particles for every particle in the subdomain, resulting $n(n-1)$ comparisons. With a cell grid we are able to reduce the number of comparison significantly since we only need to check 9 other cells for ever cell in the local grid. Figure 3.6 shows the discretization geometry for a single rank. Extra virtual buffers, called ghost buffers, are added to provide storage for particles that are not shown in the simulation. These buffers serve two main purposes. First, they provide storage for particles that are imported from adjacent domains. Second, they provide storage for virtual particles around the tank boundaries. Particles are classified according to three categories:

Field particles

Particles that are part of the medium that make up the dam in the fluid simulation.

Virtual particles

Particles stored in ghost buffers around the floor and walls of the tank to provide a bounce-back effect when field particles interact with the tank boundaries.

Ghost particles

Particles that have been temporary imported from adjacent domains to enable interactions between particles across ranks.

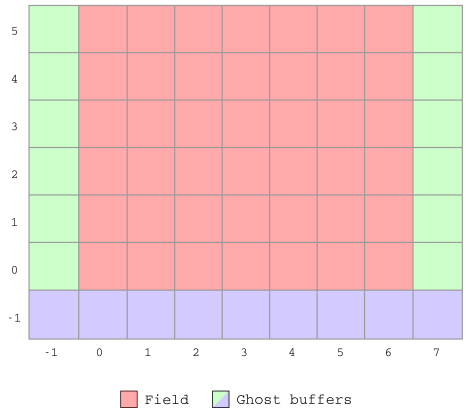


Figure 3.6 Subdomain discretization.

Ragunathan and Valstad [22] has implemented a subdomain discretization scheme, which we refer to as the Cell+List method, that uses volatile linked lists to represent grid cells. We use the Cell+List method as the basis for comparison for our Cell+Table method, which uses persistent hash tables to represent grid cells. For both of these method we use the same SPH kernel implementation originally implemented by Ragunathan and Valstad. Both discretization scheme are presented in the following sections.

3.3 Grid Model: Cell+List

3.3.1 Program Overview

The Cell+List program overview is outlined in Figure 3.7. The program consists of three main steps. Step 1 takes care of particle initialization and domain decomposition. In step 2, particles are serialized and prepared for inter-process communication and kernel execution. Step 3 involves creating particle pairs, mapping particles to the cell grid, and executing the SPH kernel.

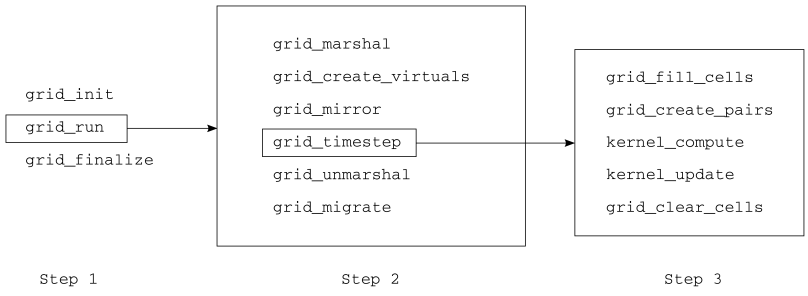


Figure 3.7 Cell+List program overview.

3.3.2 Cell Implementation

With the Cell+List method every cell is implemented as a linked list of particles. Upon program initialization, particles are first initialized and placed in a hash table. Then, at every iteration of the main loop, all local particles are serialized from the hash table into a flat pointer array according to their type. Field particles first, then ghost particles, followed by virtual particles. This step is required for the SPH kernel to execute correctly. The array is then parsed and every particle is mapped to a cell in the local grid. The cell grid is then processed for particle interactions and the resulting particle pairs are stored in a particle-pair array for the SPH kernel. When the SPH kernel has been applied, the hash table is updated with the new particle positions. The reason for storing particles in a local hash table is to avoid array fragmentation when particles migrate to other ranks.

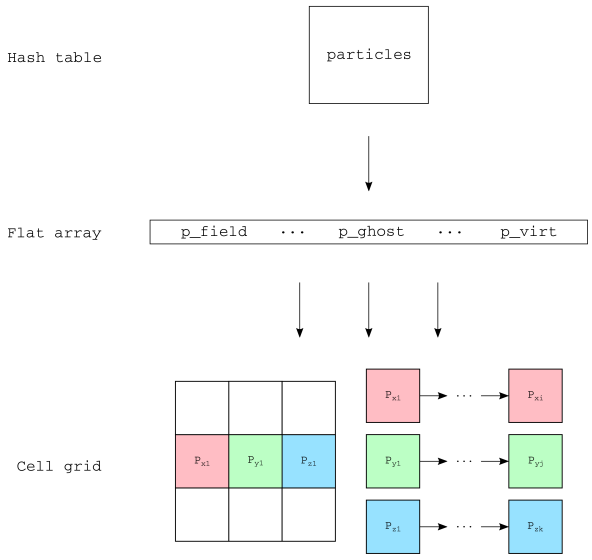


Figure 3.8 Cell+List method. Particles are serialized from a local hash table to flat pointer array. Particles are then placed into cells represented by linked lists.

Listing 3.1 Code for mapping particles to cells.

```
#pragma omp parallel
{
   #pragma omp for
   for (int i = 0; i < n_total; ++i)
   {
      particle_t *particle = &list[i];
      int_t cellpos[2];
      grid_pos2cell(particle->x, cellpos);
      particle->local_idx = i;
      particle->bucket_x = cellpos[0];
      particle->bucket_y = cellpos[1];
```

```
   }

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

      if ((particle->x[0] < subdomain[0] - RADIUS) ||
         (particle->x[0] > subdomain[1] + RADIUS))
         continue;

      int_t x = list[i].bucket_x;
      int_t y = list[i].bucket_y;
      int_t bid = y * N_BUCKETS_X + x;

      omp_set_lock(&(lock[bid]));

      bucket_t *bucket = CELL(x, 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;
         CELL(x, y) = new_bucket;
      }

      omp_unset_lock(&(lock[bid]));
   }
}
```

3.3.3 Virtual Particles

Virtual particles are generated by field particles that are within the interaction radius of the tank boundaries. For the tank walls this happens when a particle is within a horizontal distance of R_{Particle} from the wall. For the floor this happens when a particle is within a vertical distance of $R_{\text{Particle}}/2$ from the floor. The purpose of generating virtual particles is to created an a bounce-back effect when field particles hit the tank boundaries. The Cell+List method uses Neumann boundary conditions, which means that virtual particles are clones of the particles that created them, and hence possess identical particle properties. No field particle generates more than five virtual particles since only one virtual particle is created for every interaction with a ghost-buffer cell around the tank boundaries.

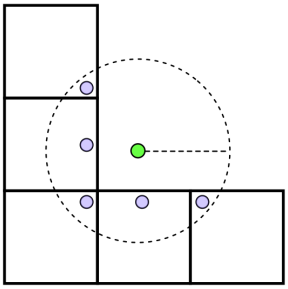


Figure 3.9 Virtual particles are created for a nearby field particle.

Listing 3.2 Routine for creating virtual particles.

```
void grid_create_virtuals(void)
{
   n_virt = 0;
   real_t boundary = RADIUS / 2;

   /* No particle adds more than 5 virtual particles */
   resize_list(n_field * 5);

   #pragma omp parallel for shared(n_virt)
   for (int_t k = 0; k < n_field; k++)
   {
      int_t current_n_virt;

      /* Horizontal mirror left */
      if (X(k) < boundary)
      {
         #pragma omp atomic capture
         current_n_virt = n_virt++;

         int_t gk = n_field + current_n_virt;
         X(gk) = -X(k), VX(gk) = -VX(k);
         Y(gk) = Y(k), VY(gk) = VY(k);
         P(gk) = P(k), RHO(gk) = RHO(k), M(gk) = M(k);
         TYPE(gk) = -2, HSML(gk) = H;
      }

      /* Horizontal mirror right */
      if (X(k) > B - boundary)
      {
         #pragma omp atomic capture
         current_n_virt = n_virt++;

         int_t gk = n_field + current_n_virt;
         X(gk) = 2 * B - X(k), VX(gk) = -VX(k);
         Y(gk) = Y(k), VY(gk) = VY(k);
         P(gk) = P(k), RHO(gk) = RHO(k);
         M(gk) = M(k);
```

```
         TYPE(gk) = -2, HSML(gk) = H;
      }

      /* Vertical mirror bottom */
      if (Y(k) < boundary)
      {
         #pragma omp atomic capture
         current_n_virt = n_virt++;

         int_t gk = n_field + current_n_virt;
         X(gk) = X(k), VX(gk) = VX(k);
         Y(gk) = -Y(k), VY(gk) = -VY(k);
         P(gk) = P(k), RHO(gk) = RHO(k), M(gk) = M(k);
         TYPE(gk) = -2, HSML(gk) = H;
      }

      /* Lower left corner */
      if (X(k) < boundary && Y(k) < boundary)
      {
         #pragma omp atomic capture
         current_n_virt = n_virt++;

         int_t gk = n_field + current_n_virt;
         X(gk) = -X(k), VX(gk) = -VX(k);
         Y(gk) = -Y(k), VY(gk) = -VY(k);
         P(gk) = P(k), RHO(gk) = RHO(k), M(gk) = M(k);
         TYPE(gk) = -2, HSML(gk) = H;
      }

      /* Lower right corner */
      if (X(k) > B - boundary && Y(k) < boundary)
      {
         #pragma omp atomic capture
         current_n_virt = n_virt++;

         int_t gk = n_field + current_n_virt;
         X(gk) = 2 * B - X(k), VX(gk) = -VX(k);
         Y(gk) = -Y(k), VY(gk) = -VY(k);
         P(gk) = P(k), RHO(gk) = RHO(k), M(gk) = M(k);
         TYPE(gk) = -2, HSML(gk) = H;
      }
   }
}
```

3.3.4 Ghost Particles

For each rank, the particle array is scanned and searched for particles that are within the interaction radius to adjacent domains. The number of export particles are counted and exchanged between all ranks. The export particles are then placed into buffers and transmitted. The import particles are stored to the right in the flat particle array.

Listing 3.3 Particle mirroring.

```
int_t w_idx = 0, e_idx = export_west;
int_t priv_w_idx, priv_e_idx;
```

```
#pragma omp parallel for private(priv_w_idx, priv_e_idx)
for (int_t k = 0; k < (n_field + n_virt); k++)
{
   /* Export west */
   if (((X(k) - subdomain[0]) < CELL_SIZE) && rank > 0)
   {
      #pragma omp atomic capture
      priv_w_idx = w_idx++;
      memcpy(&(transfer[priv_w_idx]), &(list[k]), sizeof(particle_t));
   }
   /* Export east */
   if (((subdomain[1] - X(k)) < CELL_SIZE) && rank < size - 1)
   {
      #pragma omp atomic capture
      priv_e_idx = e_idx++;
      memcpy(&(transfer[priv_e_idx]), &(list[k]), sizeof(particle_t));
   }
}

n_mirror = import_east + import_west;

resize_list(n_field + n_virt + n_mirror);

MPI_Sendrecv(transfer, export_west * sizeof(particle_t), MPI_BYTE, west, 0,
   list + n_field + n_virt + import_west, import_east * sizeof(particle_t),
   MPI_BYTE, east, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Sendrecv(transfer + export_west, export_east * sizeof(particle_t),
   MPI_BYTE, east, 0, list + n_field + n_virt,
   import_west * sizeof(particle_t), MPI_BYTE, west, 0, MPI_COMM_WORLD,
   MPI_STATUS_IGNORE);
```

3.3.5 Particle Pairs

Creating particles pairs involves traversing 9 linked lists (cells) for every particle in the subdomain. If the Idx of the current particle is less than the Idx of the other particle, then we check if they are within interaction radius. If they are, we add them to the pair list. The counter for particle pairs is incremented atomically to avoid race conditions.

Listing 3.4 Routine for creating particle pairs.

```
void cell_create_pairs(int x, int y, bucket_t **buckets, particle_t *particle,
   int_t *n_pairs, int_t *interactions)
{
   if (x >= N_BUCKETS_X || x < 0 || y >= N_BUCKETS_Y || y < 0 ||
      particle == NULL)
      return;

   bucket_t *current = CELL(x, y);
   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].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;
  }
}
```

3.3.6 Particle Serialization

With the Cell+List method, particles are marshaled in and out of the local hash table between iterations. At the beginning of every timestep, particles are serialized into a flat pointer array. At the end of every timestep, the hash table is updated with the new particle positions.

Listing 3.5 Routines for marshaling particles in and out of the local hash table.

```
void table_marshal(particle_t *list)
{
   int_t n_total = table->size;
   particle_t *pointerlist[n_total];
   table_serialize(&(pointerlist[0]));
   for (int_t i = 0; i < n_total; i++)
      memcpy(&(list[i]), pointerlist[i], sizeof(particle_t));
}

void table_unmarshal(particle_t *list, int_t size)
{
   for (int_t k = 0; k < size; k++)
   {
      particle_t *pi = &(list[k]);
      particle_t *pj;
      table_lookup(&pj, pi->idx);
      memcpy(pj, pi, sizeof(particle_t));
   }
}
```

3.3.7 Particle Migration

The migration step for the Cell+List method is very similar to the mirror step and is therefore not listed here. The main difference between the two is that in the migration routine particles

are deleted from the pointer array; in the mirror routine particles are just transmitted without modifying the local buffer.

3.4 Grid Model: Cell+Table

3.4.1 Program Overview

The Cell+Table method is outlined in Figure 3.10. The three main steps are as follows. Step 1 initializes field particles and virtual particles. Virtual particles are placed in ghost cells surrounding the tank boundaries. In step 2, particles are serialized and prepared for inter-process communication and kernel execution. Step 3 involves creating particle pairs and executing the SPH kernel.

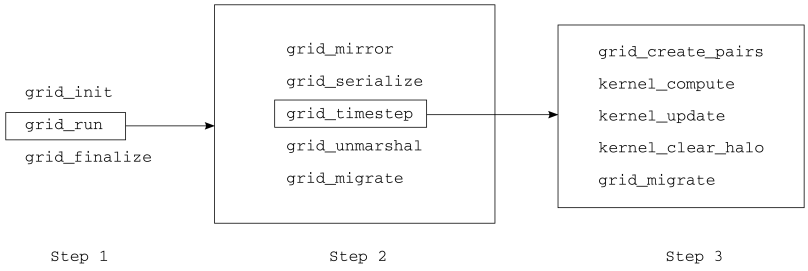


Figure 3.10 Cell+Table program overview.

3.4.2 Cell Implementation

The domain geometry of the Cell+Table method is similar to that of the Cell+List method, but instead of representing cells using linked lists that are re-constructed at every timestep, the Cell+Table method represents cells using persistent hash tables. Using this method, particles are stored in grid cells between timesteps and may migrate to new cells when their position is updated. When that happens, the particle in question is removed from its current cell and inserted into a a new cell according to its new cell coordinates. Like the Cell+List method, particles are serialized into a flat pointer array for the SPH kernel.

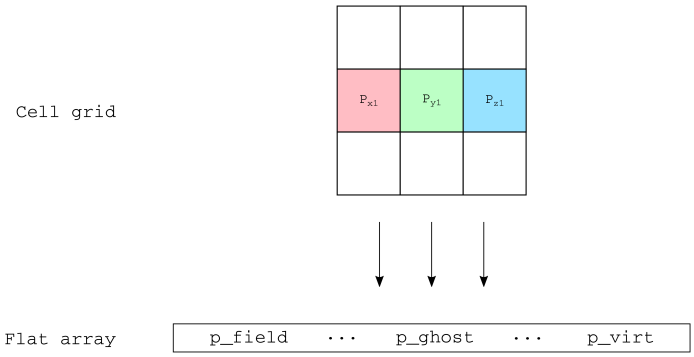


Figure 3.11 Cell+List discretized subdomain.

3.4.3 Virtual Particles

The Cell+Table method uses static virtual particles as tank boundaries, whereas the Cell+List method generates new virtual particles at every timestep when field particles are within the interaction radius to the tank boundaries. With this approach, virtual particle are created only once at program initialization and remain in ghost buffers throughout the lifetime of the application. This method uses Dirichlet boundary conditions, which means that virtual particles are initialized with constant velocities.

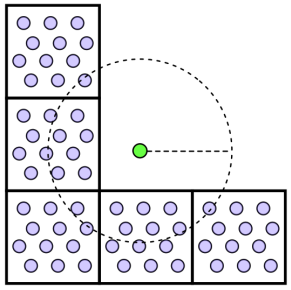


Figure 3.12 Virtual particles are created at program initialization and persists throughout the simulation.

3.4.4 Ghost Particles

For each rank, the east and west cell column is scanned and the number of export particles are counted. The particle counts are then exchanged in both directions between all rank neighbors to get the number of expected import particles. The export particles are then serialized and transmitted. Import particles are received in a flat array and inserted into the east and west

ghost buffers while awaiting kernel execution. When kernel execution is completed, the ghost particles are not needed anymore and are removed from the ghost cells.

Listing 3.6 Particle mirroring.

```
for (int_t y = 0; y < domain.n_cells_y; y++)
{
   tlhash_t *c1 = CELL(0, y);
   tlhash_t *c2 = CELL(domain.n_cells_x - 1, y);

   if (c1->size > 0 && rank != 0)
   {
      tlhash_values(c1, (void **)&(buff_west[n1]));
      n1 += c1->size;
   }
   if (c2->size > 0 && rank != size - 1)
   {
      tlhash_values(c2, (void **)&(buff_east[n2]));
      n2 += c2->size;
   }
}

#pragma omp parallel for
for (int_t i = 0; i < n_export_west; i++)
   send_west[i] = *buff_west[i];
#pragma omp parallel for
for (int_t i = 0; i < n_export_east; i++)
   send_east[i] = *buff_east[i];

int_t n_import = n_import_east + n_import_west;

particle_t *recv = (particle_t *)malloc(sizeof(particle_t) * n_import);

MPI_Request req_west, req_east;
if (n_export_west > 0)
   MPI_Isend(send_west, n_export_west * sizeof(particle_t), MPI_BYTE, west,
      0, MPI_COMM_WORLD, &req_west);
if (n_export_east > 0)
   MPI_Isend(send_east, n_export_east * sizeof(particle_t), MPI_BYTE, east,
      0, MPI_COMM_WORLD, &req_east);
if (n_import_west > 0)
   MPI_Recv(recv, n_import_west * sizeof(particle_t), MPI_BYTE, west, 0,
      MPI_COMM_WORLD, MPI_STATUS_IGNORE);
if (n_import_east > 0)
   MPI_Recv(&recv[n_import_west], n_import_east * sizeof(particle_t),
      MPI_BYTE, east, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
if (n_export_west > 0)
   MPI_Wait(&req_west, MPI_STATUS_IGNORE);
if (n_export_east > 0)
   MPI_Wait(&req_east, MPI_STATUS_IGNORE);

for (int_t i = 0; i < n_import; i++)
{
   particle_t *p = &(recv[i]);
   p->type = P_GHOST;
   grid_add_particle(p);
}
```

3.4.5 Particle Pairs

Creating particle pairs involves scanning every cell in the local domain and, for every particle in the current cell, check for particle interactions in the surrounding cells and the current cell. If at least one of the two particles is a field particle and the Idx of the current particle is less than the Idx of the other particle, then we check if the distance between the particles are less than R_{Cell}. If they are, we add them to the pair list. The number of particle interactions and particle pairs is incremented in an atomic fashion to avoid race conditions.

Listing 3.7 Routine for creating particle pairs.

```
void cell_create_pairs(tlhash_t *cell1, int_t x, int_t y)
{
   if (x < -1 || x > domain.n_cells_x || y < -1 || y >= domain.n_cells_y)
      return;

   tlhash_t *cell2 = CELL(x, y);

   int_t size1 = cell1->size;
   int_t size2 = cell2->size;
   particle_t *particles1[size1];
   particle_t *particles2[size2];

   tlhash_values(cell1, (void **)particles1);
   tlhash_values(cell2, (void **)particles2);

   for (int_t i = 0; i < size1; i++)
   {
      particle_t *p1 = particles1[i];
      for (int_t j = 0; j < size2; j++)
      {
         particle_t *p2 = particles2[j];
         if (p1->type == P_VIRT && p2->type == P_VIRT)
            continue;
         if (p1->idx < p2->idx)
         {
            particle_t *p2 = particles2[j];
            real_t dx = p1->x[0] - p2->x[0];
            real_t dy = p1->x[1] - p2->x[1];
            real_t d = sqrt(dx * dx + dy * dy);
            if (d < RADIUS)
            {
               #pragma omp atomic update
               p1->interactions++;
               #pragma omp atomic update
               p2->interactions++;

               int_t i;
               #pragma omp atomic capture
               i = domain.n_pairs++;

               domain.pairs[i].ip = p1;
               domain.pairs[i].jp = p2;
               domain.pairs[i].r = distance;
               domain.pairs[i].q = distance / H;
               domain.pairs[i].w = 0.0;
               domain.pairs[i].dwdx[0] = domain.pairs[i].dwdx[1] = 0.0;
```

```
            }
        }
    }
}
```

3.4.6 Particle Serialization

Particles are serialized from the cell-grid into a flat pointer array. This includes field particles, ghost particles, and virtual particles. Field particles are placed first, followed by ghost particles and virtual particles. The particles are arranged in this way to be able to differentiate between the different particle types in the SPH kernel.

Listing 3.8 Routine for serializing particles from the cell-grid into a flat pointer array.

```
void grid_categorize(void)
{
   grid_resize_particles();

   int_t n_total = domain.n_field + domain.n_virt + domain.n_ghost;
   int_t n_field = 0, n_virt = 0, n_ghost = 0;

   memset(domain.particles, 0, n_total * sizeof(particle_t *));

   for (int_t y = -1; y < domain.n_cells_y; y++)
   {
      for (int_t x = -1; x <= domain.n_cells_x; x++)
      {
         tlhash_t *cell = CELL(x, y);
         particle_t *particles[cell->size];
         tlhash_values(cell, (void **)particles);
         for (int_t i = 0; i < cell->size; i++)
         {
            particle_t *p = particles[i];
            if (p->type == P_FIELD)
            {

               p->local_idx = n_field;
               domain.particles[n_field] = p;
               n_field++;
            }
            else if (p->type == P_VIRT)
            {
               p->local_idx = domain.n_field + n_virt;
               domain.particles[domain.n_field + n_virt] = p;
               n_virt++;
            }
            else
            {
               p->local_idx = domain.n_field + domain.n_virt + n_ghost;
               domain.particles[domain.n_field + domain.n_virt + n_ghost] = p;
               n_ghost++;
            }
         }
      }
```

```
   }
}
```

3.4.7 Particle Migration

Particle migration happens in two steps. First, all field cells are scanned to check if any particles have moved outside their current cells. If that is the case, the particles in question are moved to their new cell destinations. The new cell may be a ghost-buffer cell, which means that the particle has moved outside its current subdomain and will be migrated to an adjacent rank. The particle migration procedure is almost identical to the particle mirroring procedure. The only difference is that in the particle migration routine, particles are removed from the cell grid after the migration has completed.

Listing 3.9 Routine for migrating particles to new cells.

```
void grid_update(void)
{
   int_t n_cells = domain.n_cells_x * domain.n_cells_y;
   for (int_t k = 0; k < n_cells; k++)
   {
      int_t cx = k % domain.n_cells_x;
      int_t cy = k / domain.n_cells_x;

      tlhash_t *current = CELL(cx, cy);
      particle_t *particles[current->size];
      tlhash_values(current, (void **)particles);

      for (int_t j = 0; j < current->size; j++)
      {
         particle_t *p = particles[j];
         if (p->type == P_FIELD)
         {
            int_t cellpos[2];
            grid_pos2cell(p->x, cellpos);
            int_t x = cellpos[0];
            int_t y = cellpos[1];

            grid_clamp(cellpos);

            /* Particle has moved outside its current cell */
            if ((x != cx) || (y != cy))
            {
               tlhash_t *next = CELL(x, y);
               tlhash_remove(current, &(p->idx), sizeof(int_t));
               tlhash_insert(next, &(p->idx), sizeof(int_t), (void *)p);
               /* Particle has moved into halo and does no longer
                  belong to this domain */
               if (x <= -1 || x >= domain.n_cells_x)
                  domain.n_field--;
            }
         }
      }
   }
}
```

3.5 Hardware

In this section we present the compute resources used to conduct benchmarks and experiments. The node topologies of Idun-E5 and Idun-Gold are shown in Appendix B.

3.5.1 Fram

Fram is supercomputer hosted at the Arctic University of Norway (UIT). It consists of 1004 dual-socket nodes (32256 cores) interconnected with an InfiniBand network. Each node has 2 × Intel Xeon E5-2683 v4 CPUs. The system has a theoretical peak performance of 1.1 petaFLOPS.

3.5.2 Idun

Idun is a computing cluster hosted at the Norwegian University of Science and Technology (NTNU). It consists of several sub-clusters. We use two clusters in this study, namely Idun-E5 and Idun-Gold, described below.

Idun-E5

27 nodes (540 cores). Each node has 2 × Intel Xeon E5-2630 v2 CPUs.

Idun-Gold

12 nodes (336 cores). Each node has 2 × Intel Xeon Gold 6132 CPUs.

Table 3.2 CPU specifications.

Fram	Cores	16 (32)
	Base Frequency	2.10 GHz
	L3 cache (shared)	40 MB
	Max Bandwidth	76.8 GB/s
	Max FLOPS	NA
Idun-E5	Cores	10 (20)
	Base Frequency	2.20 GHz
	L3 cache (shared)	25 MB
	Max Bandwidth	68.3 GB/s
	Max FLOPS	NA
Idun-Gold	Cores	14 (28)
	Base Frequency	2.60 GHz
	L3 cache (shared)	19 MB
	Max Bandwidth	119.2 GB/s
	Max FLOPS	NA

3.6 Simulation Setup

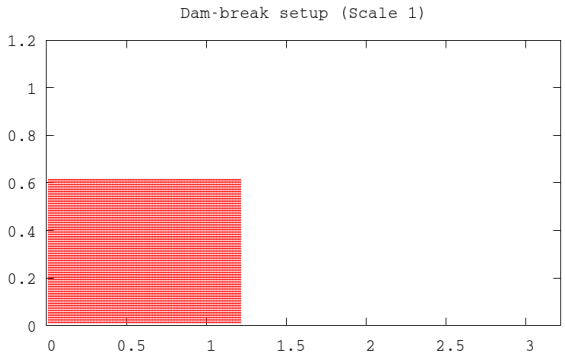
We use two different simulation setups when measuring program performance. The dam-break simulation has already been presented in Section 2.2. With this simulation we get violent surface flows during the first half of the simulation before the dam eventually calms down. With the other simulation setup, the still simulation, we get a simulation that has very little dynamic movement in the dam.

3.6.1 The Still Simulation

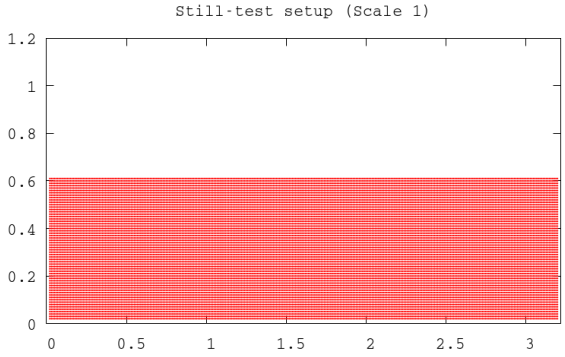
The still simulation is similar to the original dam break simulation, but we adjust the width of the dam to almost completely fill the tank,

$$L = B - R_{\text{Cell}},$$

leaving only half a cell width between the dam and the tank on each side. The dam is vertically positioned from a height of $R_{\text{Cell}}/2$ to provide sufficient fluid motion as the dam hits the tank. These parameters yield the initial configuration as shown in Figure 3.13. We note that the still simulation yields a significantly larger particle count compared to the dam-break simulation with the same scale, having approximately 2.5 times as many particles.



(a)



(b)

Figure 3.13 The two simulation setups we use to measure application performance. (a) Dam-break simulation, Scale 1, 7381 particles. (b) Still simulation, Scale 1, 19140 particles.

3.7 Program Metrics

The SPH program metrics are measured using the dam-break simulation to get an overview of how the SPH application performs under dynamic conditions. We use these metrics to expose the main characteristics of the SPH application and its potential bottlenecks.

3.7.1 Execution Time

To get an overview of the performance characteristics of the application, we measure T_{Comp} and T_{Comm} by timing the SPH kernel and each individual communication routine. We also look at number of particles transmitted between all ranks. We measure the migration counts and mirror counts separately as we expect a rather large difference between the two. The measurements are reported as the averages over a 1000 timesteps across all ranks over 128 K timesteps.

3.7.2 Workload

The workload distribution is measured by running the dam-break simulation for 128 K timesteps. We perform more frequent measurements during the first half of the simulation since we expect more rapid changes in the workload distribution during the first half of the simulation. The workload is computed by reducing the local particle count to the master rank (rank 0). The workload distribution is then reported as the fraction of the total particle count per timestep. We measure the workload distribution for 4 different scales using 4 ranks. We use a relatively low rank count for illustration purposes since it makes the distribution measurements more distinguishable.

3.7.3 Idle Time

The idle time of the SPH application is the total fraction of the execution time where one or more ranks are have no particles in their domains and, consequently, have no work to do. We measure the idle time on two ways. First, we measure the number of timesteps required before all ranks have at least one particle in their domain. Second, we measure the total time spent waiting in an idle state as a fraction of the total execution time.

3.8 System Metrics

We measure systems metrics using synthetic benchmarks that are tuned to each target system. In this section we present the implementation and configuration of three benchmarks, namely the DGEMM benchmark, the Stream benchmark, and the Ping-pong benchmark.

3.8.1 DGEMM benchmark

We measure the floating-point performance per node according to Equation 2.15 using a level 3 BLAS routine. We use the CBLAS library to perform matrix-matrix multiplications for this benchmark, as it provides hardware optimizations for a range of different CPUs, including the ones we use in this study. The result is computed as the average FLOPS over 10 consecutive benchmark executions.

Listing 3.10 DGEMM benchmark using the CBLAS library.

```
t_start = walltime();

cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
   M, Q, N, alpha, mat_a, N, mat_b, Q, beta, mat_c, Q);

return walltime() - t_start;
```

3.8.2 Stream Benchmark

The bandwidth of each system is measured with the Stream benchmark by recording the execution time of the Triad routine from Equation 2.12 over 10 benchmark executions. These measurements are then used together with the combined memory requirement of the Triad routine to get the total bandwidth of the system. We run the benchmark with an increasing

number of threads with both scattered and compact affinity schemes since we expect to see the effects of memory-channel saturation. The Stream benchmark requires each array operand to be roughly four times larger than the combined cache memory in order to measure the bandwidth accurately. We therefore use an operand size of 640 MB on all three systems, which is roughly 6.3, and 8.2 times larger than the total L3 cache memory on Fram, Idun-E5, and Idun-Gold, respectively. Our Stream-benchmark implementation is listed in Appendix A.

Listing 3.11 Stream benchmark.

```
t_start = walltime();
#pragma omp parallel for
for (int i = 0; i < STREAM_ARRAY_SIZE; i++)
   a[i] = b[i] + scalar * c[i];
t_time = walltime() - t_start;
if (j > 0)
{
   timings[TRIAD][MIN] = MIN(timings[TRIAD][MIN], t_time);
   timings[TRIAD][MAX] = MAX(timings[TRIAD][MAX], t_time);
   timings[TRIAD][AVG] += t_time;
}
```

3.8.3 Ping-pong Benchmark

We derive the Hockney parameters from Equation 2.9 by performing a series of Ping-pong benchmarks. We make the assumption that the cost of sending a message between two ranks is the same both ways. Thus, we need only benchmark $n(n-1)/2$ communication pairs. Every two ranks execute exclusively to avoid communication-channel interference. The setup latency is derived by measuring the cost of sending a small message (1 byte) back and fourth 1000 times between every communication pair. The inverse bandwidth is derived by measuring the cost of sending a large message (100 MB) 10 times back and fourth between every communication pair. Thus, for a message size of M bytes and N benchmark iterations, the model parameters can be derived by adjusting the N and M such that either the setup latency or the inverse bandwidth becomes the domination factor. Our Ping-pong-benchmark implementation is listed in Appendix A.

Listing 3.12 Ping-pong benchmark.

```
double t_start;

start = MPI_Wtime();
if (rank == src)
{
   for (int_t k = 0; k < num_bench; k++)
      MPI_Ssend(send_buff, message_size, MPI_CHAR, dst,
         0, MPI_COMM_WORLD);
   for (int_t k = 0; k < num_bench; k++)
      MPI_Recv(recv_buff, message_size, MPI_CHAR,
         dst, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
if (rank == dst)
{
   for (int_t k = 0; k < num_bench; k++)
      MPI_Recv(recv_buff, message_size, MPI_CHAR,
         src, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
   for (int_t k = 0; k < num_bench; k++)
      MPI_Ssend(send_buff, message_size, MPI_CHAR, src,
         0, MPI_COMM_WORLD);
}

return (MPI_Wtime() - t_start) / (2 * num_bench * message_size * sizeof(char));
```

3.9 Cell Performance

We compare the speedup and parallel efficiency of the Cell+Table and Cell+List methods. For this purpose we look at the average program runtime across all ranks for a fixed number of timesteps using the still simulation. With this simulation setup we get a performance measure of each cell implementation when there is little dynamic movement in the dam, and thus the difference in program performance will mostly depend on the cell-grid scheme.

We also compare the grid performance using the dam-break simulation. With this simulation setup we get a performance measure of the cell-grid scheme when there is a lot of dynamic movement in dam during the first half of the simulation. We measure the performance in total grid execution time across all ranks. With more dynamic movement in the dam, we expect to get an indication of the cost of migrating particles to individual cells for Cell+Table method.

Chapter 4

Results and Discussion

4.1 Program Metrics

In this section we present performance metrics for the SPH application that is independent of the cell-grid implementation scheme. These results include the kernel execution time, communication cost, and workload distribution. We compare the performance of the Cell+Table and Cell+List methods separately in Section 4.3.

4.1.1 Execution Time

Figure 4.1 shows the total number of particles transmitted between all ranks for particle migration and particle mirroring. Only exported particles are counted. Figure 4.2 shows the kernel execution time and communication time.

The mirror count starts relatively high compared to the corresponding migration count and converges towards approximately 500 particles as the dam stabilizes. We see the same trend with the particle migration count, albeit with a much lower initial migration count that converges towards zero. With particle mirroring, more ranks corresponds to higher transmission counts. This can be explained by the communication topology and the fact that every new rank introduces extra subdomain boundaries where particles are mirrored to adjacent domains. This does not affect particle migration in the same way since only particles that cross the subdomain boundary are subject to migration. This leads to much higher transmission counts for particle mirroring.

The communication cost constitutes a very low fraction of the kernel execution time. In the case where the domain is split between a high number of ranks, *e.g.* 41 or 32, such that the width the subdomain is only 1 or 2 cells, the communication time represents less than 5 % of the total kernel execution time. Thus, the communication cost is not a limiting performance factor, even for sparse simulations where particle counts are low.

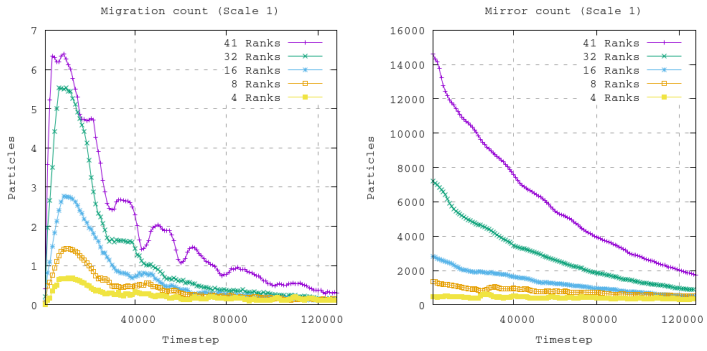


Figure 4.1 Number of particles transmitted as averages of 1000 timesteps. Migration (left). Mirroring (right). Scale 1, 7381 particles, 128 K timesteps.

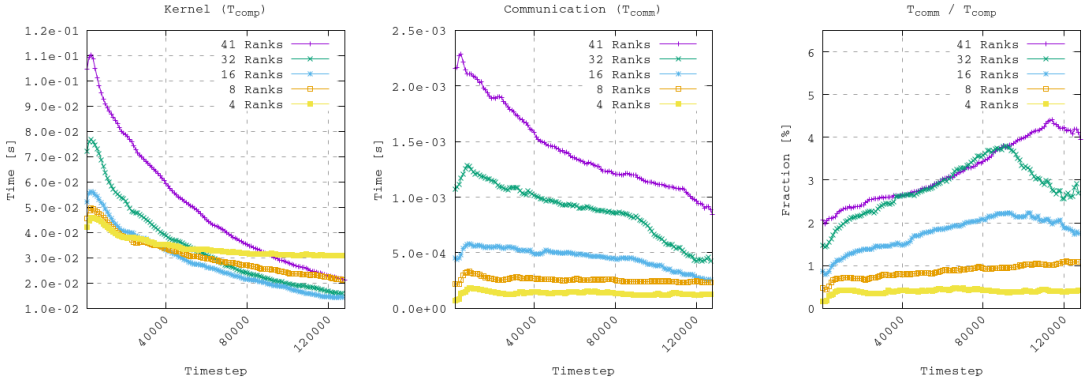


Figure 4.2 Kernel execution vs. communication cost as averages of 1000 timesteps. Kernel execution time (left). Communication time (center). Fraction of communication time to kernel execution time (right). Scale 1, 7381 particles, 128 K timesteps.

4.1.2 Workload

Figure 4.3 displays the workload distribution among 4 ranks for Scale 1, 2, 3 and 4, with the simulation running for 128 K timesteps. The domain is decomposed vertically among ranks, giving each consecutive rank a rectangular subdomain. Consequently, as the dam breaks and starts moving from left to right, the workload is distributed unevenly until the dam has entered the domain of the rightmost rank. Initially, most of the work will be divided among the leftmost ranks. As the dam breaks and starts moving, particles migrate across subdomains and the workload gets distributed among multiple ranks. Finally, when the simulation has reached equilibrium, the workload distribution per rank will be close to 50 %. For Scale 1 we see the that the entire workload is divided among the first two ranks at timestep 1000. As the simulation progresses and particles start moving into adjacent subdomains, the workload of the two

rightmost ranks increases at the same time as the workload of the two leftmost ranks decreases. At around 128 K timesteps the workload is distributed evenly across all ranks. For the other scale factors we see a similar wave effect as the dam propagates through the tank.

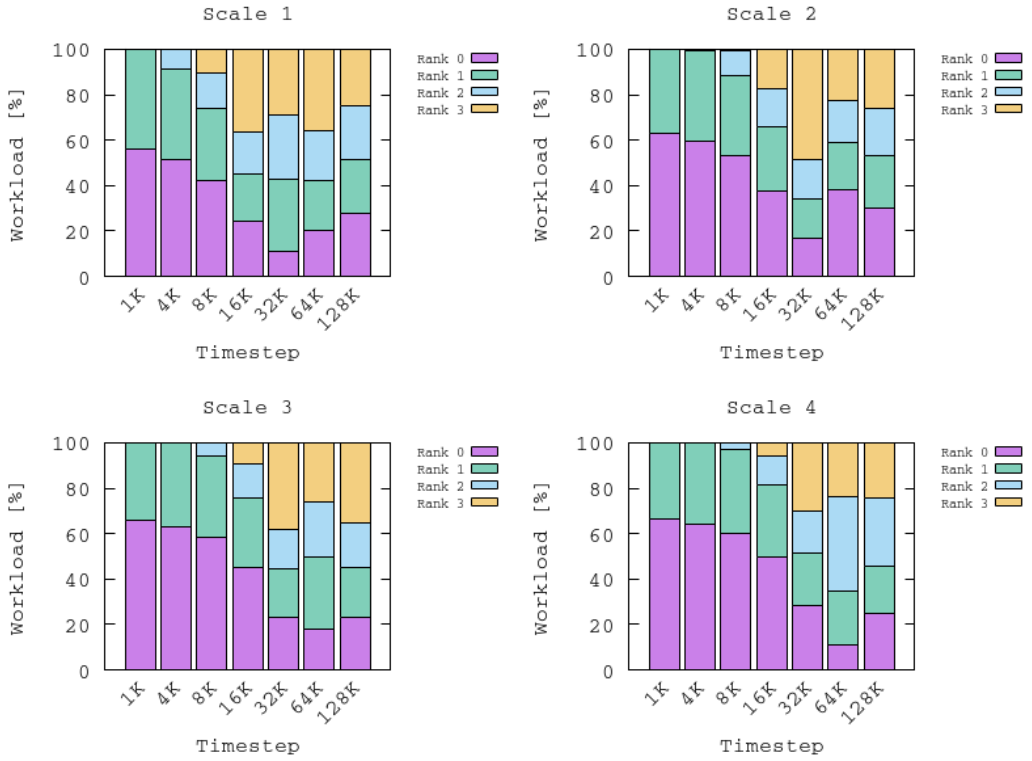


Figure 4.3 Workload distribution for different scales. 4 ranks, 128 K timesteps.

4.1.3 Idle Time

Figure 4.4 shows the relationship between simulation timesteps and idle time. The left figure shows the number of timesteps required to reach zero idle time for different scales factors. The right figure shows the corresponding idle time as the percentage of the total execution time. The staircase pattern is caused by particles entering a new subdomain for the first time. The reduction in idle time equals the workload distribution per rank, which in this case is 1/8 (12.5 %) for 8 ranks. The average increase in the number of timesteps required before all ranks have some degree of work increases by approximately 30 % per scale increment. If c_r is a constant denoting the first timestep all r ranks have work to do for a scale factor s, then the expected idle time can be approximated by the following equation,

$$T(c_r, s)_{\text{Idle}} = c_r 1.3^s. \tag{4.1}$$

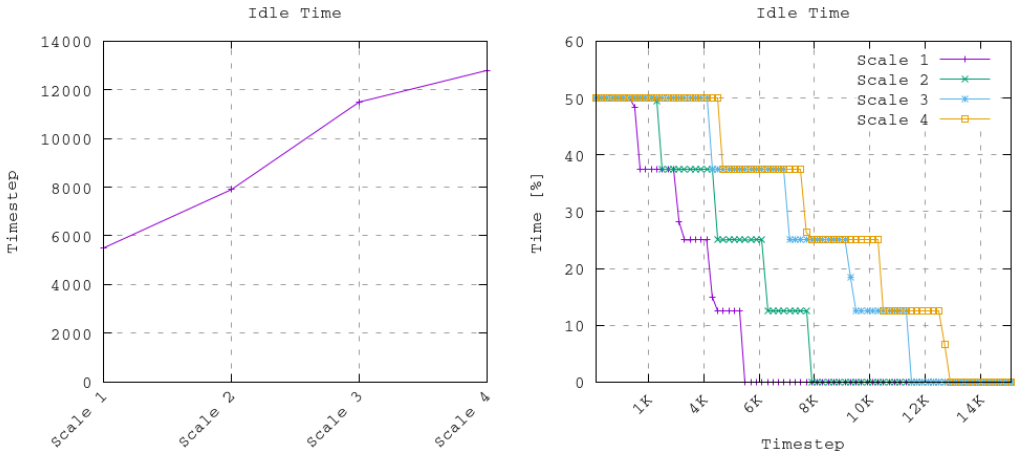


Figure 4.4 Total fraction of program execution time spent idle. 14 K timesteps, 8 ranks. Left: Number of timesteps required before all ranks have work to do. Right: Idle time as percentage of total execution time.

4.2 System Metrics

In this section we present performance metrics for Fram, Idun-E5, and Idun-Gold.

4.2.1 DGEMM Benchmark

Table 4.1 lists the DGEMM benchmark results. We see that Idun-Gold achieves the highest node performance of the three, with a compute capacity that is 17 % higher compared to Fram. Idun-E5 has the lowest node performance. Adjusting the results to single-thread performance, we see that the performance per thread correlates with the CPU clock frequency listed in Table 3.2.

Table 4.1 DGEMM benchmark results.

System	GFLOPS per node	GFLOPS per core
Fram	428.9	13.4
Idun-E5	357.2	17.9
Idun-Gold	504.2	18.0

4.2.2 Stream Benchmark

Figure 4.5 shows the results from the Stream benchmark. For Fram, Idun-E5, and Idun-Gold, the measured bandwidth constitutes 61.9 %, 70.1 %, and 72.6 % of the theoretical peak bandwidth reported from the hardware manufacturer. Using a compact affinity scheme, we see that the memory channel is saturated as the number of threads starts to fill the first socket. As additional threads start to execute on the second socket, we see a linear increase in bandwidth for every new thread. Running the benchmark with scattered affinity, the increase in bandwidth happens more rapidly due to both memory channels being utilized. The saw-tooth pattern occurs on every odd-numbered thread count and is caused by one memory channel being slightly more saturated than the other, thereby causing a slight decrease in measured bandwidth.

Table 4.2 Stream benchmark results.

System	Affinity	Min [GB/s]	Max [GB/s]	Avg [GB/s]
Fram	Compact	123.4	123.8	123.6
	Scatter	122.8	123.2	123.1
Idun-E5	Compact	98.4	98.8	98.6
	Scatter	95.9	96.1	96.0
Idun-Gold	Compact	167.4	167.8	167.6
	Scatter	156.4	158.2	157.8

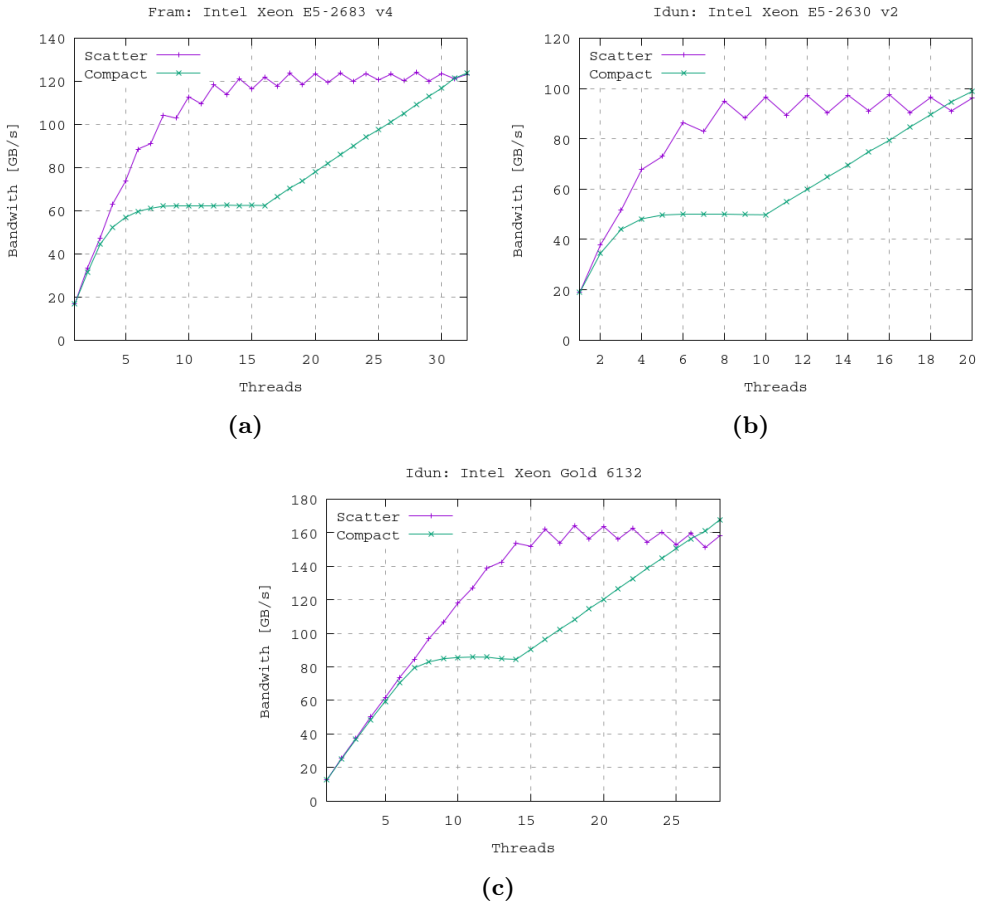


Figure 4.5 Stream benchmark results. (a) Fram. (b) Idun-E5. (c) Idun-Gold.

4.2.3 Ping-pong Benchmark

The Hockney parameters derived using the Ping-pong benchmark are listed in Table 4.3. The values listed here are the maximum average values for each rank affinity: intra socket, inter socket, and inter node. The raw individual metrics for every communication pair is shown in Figure 4.6 (setup latency) and Figure 4.7 (inverse bandwidth). The setup latency dominates the inverse bandwidth until a certain number of particles is reached. This limit is encountered when the message size is around α/β bytes, which corresponds to around 20–30 particles depending on the system. Since the Hockney parameters were derived using message sizes up to 100 MB, the validity of communication-cost prediction is restricted to transmission loads smaller or equal to 100 MB. Since we know that the SPH application has transmission loads significantly lower than 100 MB, even for high rank counts, the validity of the model applies to all test cases conducted in this study.

Table 4.3 COMMS1 parameter measurements.

		Intra socket	Inter socket	Inter node
Fram	α [s]	2.13e-06	2.96e-06	9.66e-06
	β [s/byte]	2.95e-10	3.1e-10	1.84e-10
Idun-E5	α [s]	6.42e-07	1.45e-06	3.17e-06
	β [s/byte]	2.85e-10	3.05e-10	3.43e-10
Idun-Gold	α [s]	5.87e-07	1.69e-06	5.44e-06
	β [s/byte]	2.45e-10	2.69e-10	1.79e-10

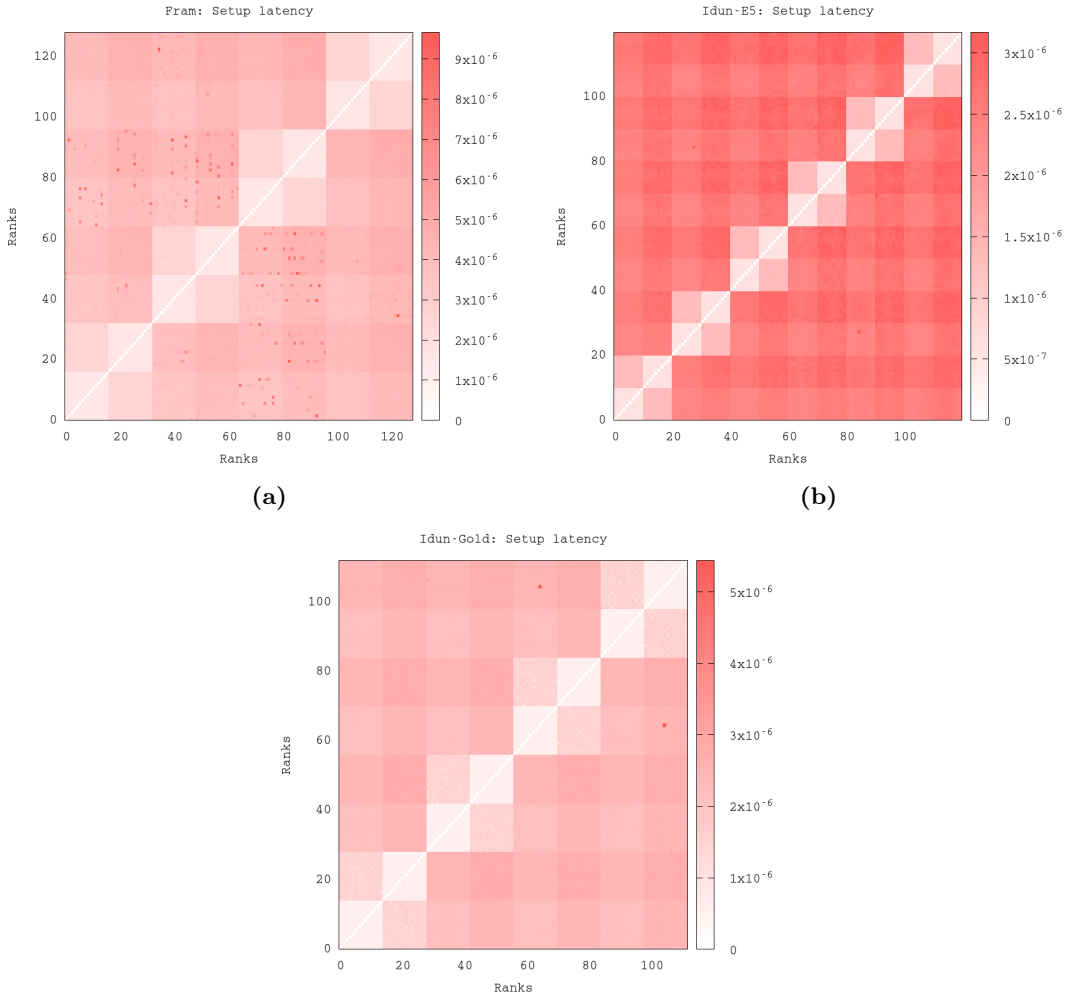


Figure 4.6 Setup latency . (a) Fram, 128 ranks. (b) Idun-E5, 120 ranks. (c) Idun-Gold, 112 ranks.

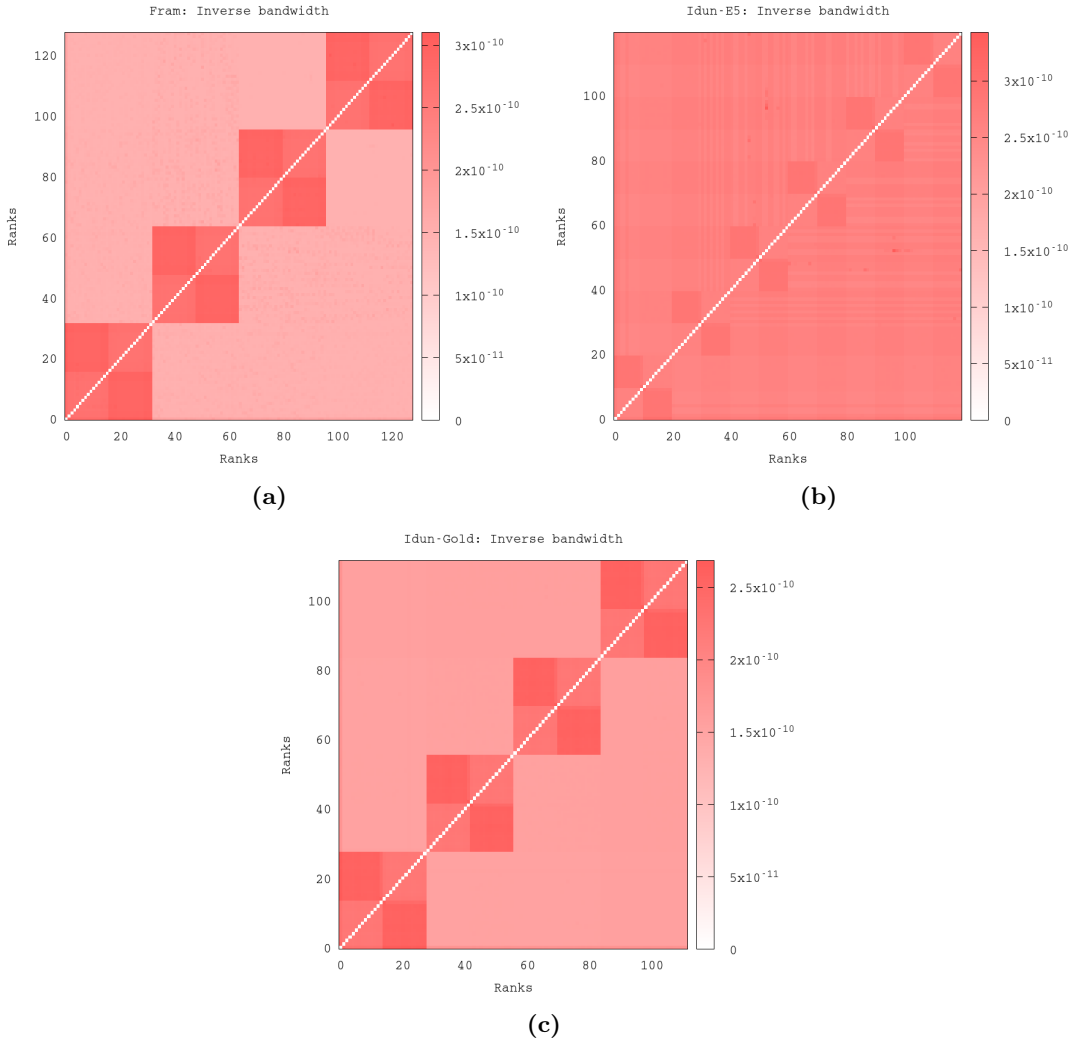


Figure 4.7 Inverse bandwidth. (a) Fram, 128 ranks. (b) Idun-E5, 120 ranks. (c) Idun-Gold, 112 ranks.

4.2.4 Roofline

Using the peak bandwidth from the Stream benchmark and peak FLOPS from the DGEMM benchmark, we get the Roofline ceilings for Fram, Idun-E5, and Idun-Gold, as shown in Figure 4.8. We get the theoretical operational intensity for the SPH kernel by counting the number of floating-point operations involving particle pairs, which is $89n$ for n particle pairs. The total memory requirement for n particle pairs is $56n$. Since both the number of floating-point operations and the memory requirement per particle scale by the same factor n, we can remove n from the equation. The operational intensity for the SPH application then becomes $I = 89/56$.

Plotting the SPH kernel in the Roofline model shows that the SPH kernel is memory bound on all three systems.

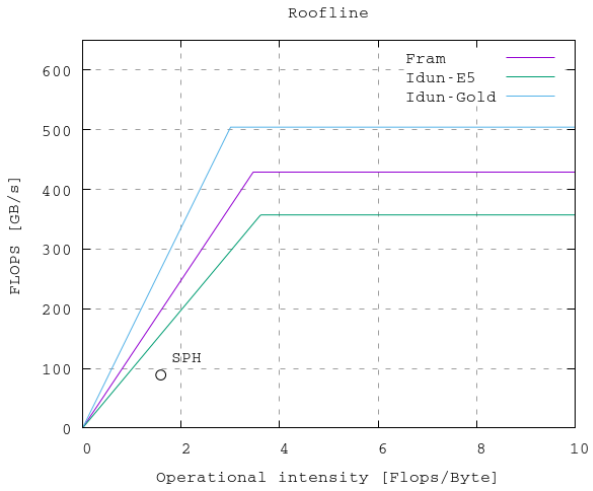


Figure 4.8

4.3 Cell Performance

In this section we present and compare performance metrics for the Cell+Table and Cell+List methods.

4.3.1 Still Test

For the still simulation, the Cell+Table method achieves a speedup of 30.1 % to 65.1 %, respectively, compared to the Cell+List method.

The parallel efficiency declines rapidly as the rank count is increased for both methods. The efficiency of the Cell+Table method declines from 86.9 % with 4 ranks, to 9.1 % with 32 ranks. The efficiency for the Cell+List is almost identical, starting with an efficiency of 80.1 % for 4 ranks, declining to 6.7 % with 32 ranks.

The communication cost constitutes only a very small fraction of the total execution time, even for high rank counts. We know that the two main performance bottlenecks of the SPH application is particle-pair creation and kernel execution. Therefore, increasing the rank count above a certain point only marginally contributes to performance increase, which causes a rapid decline in parallel efficiency. The performance inprovement of the Cell+Table method over the Cell+List method comes from the low maintanance cost using persistent hash tables and less memory traffic when scanning cells for particle pairs.

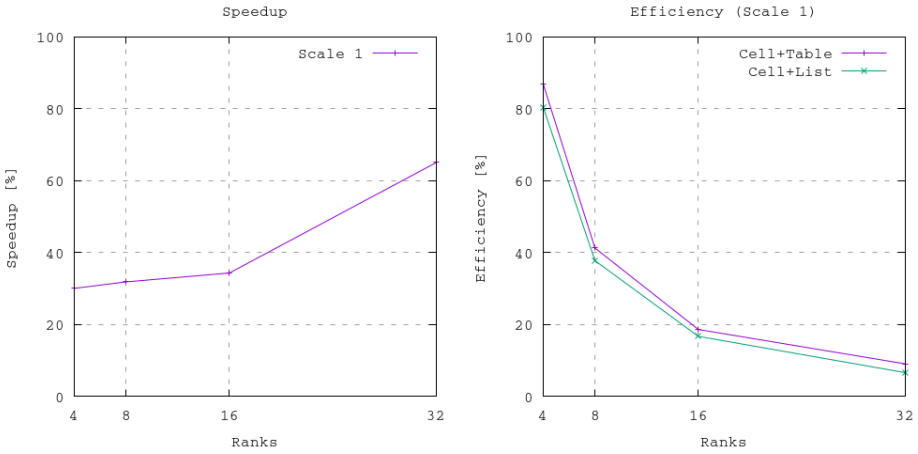


Figure 4.9 Speedup and efficiency for Cell+Table vs. Cell+List. Scale 1, 7381 particles.

4.3.2 Dam-break Test

For the dam-break test, Cell+Table achieves a speedup of 24.4 % and 28.6 %, respectively, compared to the Cell+List method.

For the Cell+Table method we see that overhead of migrating particles to new cells introduces some overhead during the early stages of the simulation. After around 20 K to 30 K timesteps, the number of particles that migrate between cells decreases, causing the overall grid cost to go down as there is less movement in the dam. For the Cell+List however, we observe that the overhead of re-building the cell grid at every timestep incurs a constant overhead that persists throughout the simulation, also after the dam has calmed down.

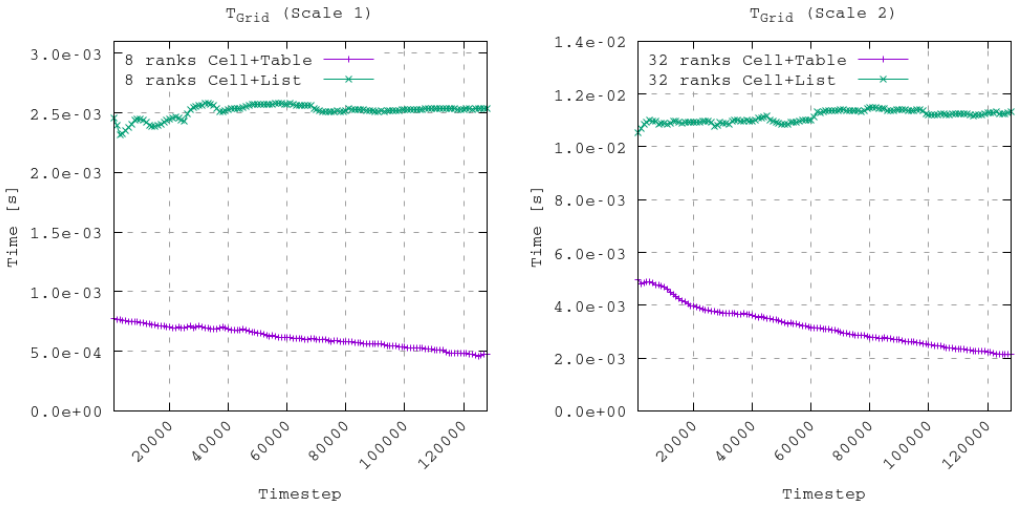


Figure 4.10 Grid maintenance cost. 128 K timesteps.

4.4 Model Validation

4.4.1 Communication Cost

The communication cost is predicted with the Hockney model (Equation 2.9) using the machine parameters for setup latency and inverse bandwidth derived using the Ping-pong benchmark. We validate the Hockney model using the average maximum measurements for setup latency and inverse bandwidth for each rank affinity. The prediction error of the communication cost is shown in Figure 4.11. We see that the prediction error converges towards zero as the total number of export particles declines. This suggests that the machine metric for the inverse bandwidth introduces some inaccuracy when the fluid motion is violent and the transmission load changes frequently during the first 40 K timesteps of the simulation. This error can be accounted by modelling the prediction error and extending Equation 2.9,

$$T_{\text{Comm}} = a_{ij} + M\beta_{ij} - \epsilon(M), \tag{4.2}$$

where $\epsilon(M)$ is the expected error prediction for M bytes. The definition of $\epsilon(M)$ may vary depending on the target system. For our measurements we suggest an exponential error function to reduce the error prediction when there is a lot of movement in the dam during the first half of the simulation. However, since the communication cost constitutes a very small fraction of the total execution time, the extra effort spent modeling the (very small) prediction error may not be worth it.

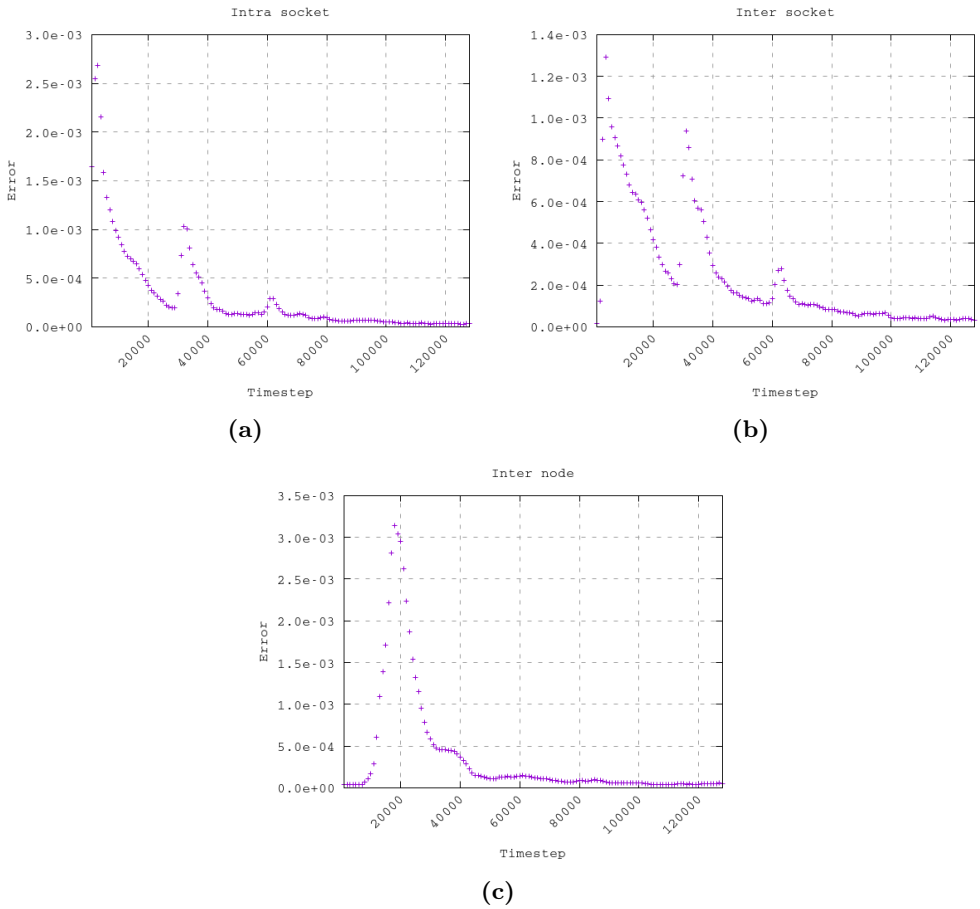


Figure 4.11 Fram communication cost prediction error. (a) Intra socket. (b) Inter socket. (c) Inter node.

4.4.2 Computational Cost

Modeling the computational cost involves predicting the number of particles and particles pairs in the local domain. From Figure 4.12 we see that there is linear relationship between the number of particles currently in the domain and the number of generated particle pairs. We see a similar relationship between the number of local particles and exported particles. We therefore end up with a linear model for both cases,

$$p(n) = (mn + r), \tag{4.3}$$

where n is the number of particles in the local domain. Table 4.4 lists the parameters for each model.

Table 4.4 Particle prediction coefficients.

Function	Coefficients	Values	Asymptotic Standard Error
$p(n)_{\text{Pair}}$	m	19.451	0.4994 (2.567 %)
	r	-589.96	299.6 (50.77 %)
$p(n)_{\text{Export}}$	m	0.112845	0.004522 (4.008 %)
	r	-4.02341	2.713 (67.42 %)

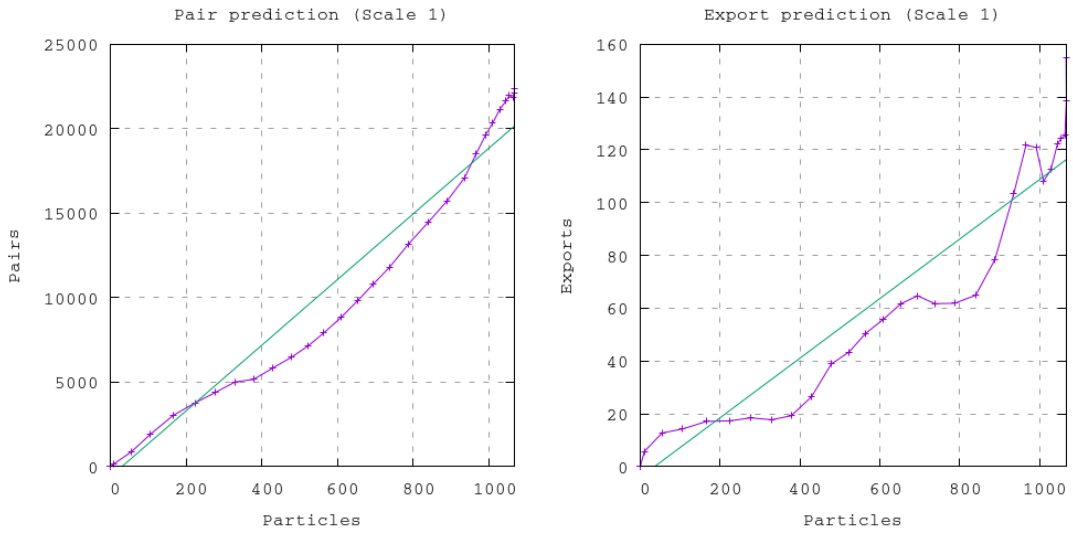


Figure 4.12

4.4.3 Memory Cost

The two main sources of memory usage comes from the two arrays holding particles and particles pairs, and the cell-grid data structure. The required memory for n particles is

$$M(n)_{\text{Particles}} = 2nm, \tag{4.4}$$

where m is the size of a single particle in bytes. The memory requirement for particles pairs is based on two assumptions. First, we assume that every particle interacts with every other particle in neighboring cells. This is not practically possible since peripheral particles may have only a few or no interactions. Thus it is a pessimistic assumption. Second, measurements show that no cell contains more than 32 particles throughout the lifetime of the simulation. Based on these two assumptions we can define an upper limit memory requirement for particle pairs,

$$M(n)_{\text{Pairs}} = \frac{nmk}{2}, \tag{4.5}$$

where $k = 32 \times 9 = 288$ is an upper limit to the number of possible interactions for every particle in the domain. We divide k by 2 since we treat the particle pairs (i, j) and (j, i) as the same interaction.

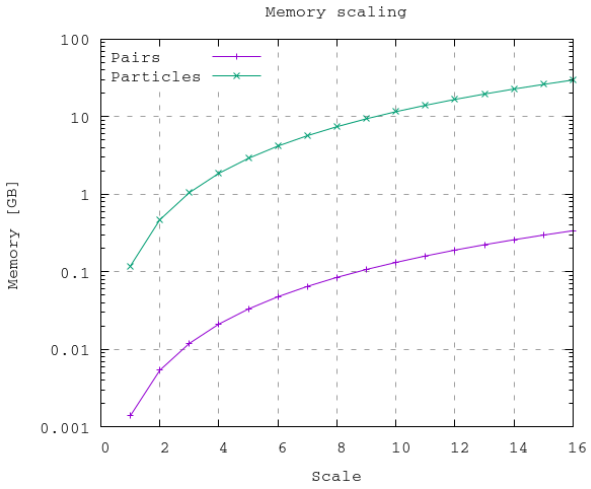


Figure 4.13 Memory usage for particles and particle pairs by scale factor.

Chapter 5

Conclusion

In this study we have implemented two different subdomain discretization techniques; the Cell+Table method that implements grid cells using persistent hash tables, and the Cell+List method that implements grid cells using volatile linked lists. We find that the Cell+Table outperforms the Cell+List method with a performance increase of more than 24 % and 30 % for two different test cases, respectively. We have found that implementing grid cells using persistent hash tables can yield significant performance increase based on two observations. First, the lower bound of the operational intensity for the SPH proxy application is relatively small. This shows that the ratio of the number of floating-point operations to the number of memory operations for the SPH kernel is low, and that the SPH kernel is memory bound for the systems used in this study. Since the gap between floating-point performance and memory latency has shown no signs of getting smaller over recent years, it is reasonable to assume that the SPH kernel will be memory bound on most modern computing systems for the foreseeable future. Representing grid cells using persistent hash tables will therefore yield less memory traffic and consequently higher application performance compared to the alternative approach of using volatile linked lists as cell representations. Second, we also see that the cost of migrating particles between individual cells is low enough so that the Cell+Table method outperforms the Cell+List method for violent surface flows. Based on these observations, we recommend using the Cell+Table method as the cell-grid scheme for SPH applications with similar characteristics as the SPH application used in this study.

We have developed a suite of benchmarks for measuring machine metrics that can be used to model application performance. These benchmarks include the DGEMM benchmark (using the CBLAS library) for measuring peak system FLOPS, the Stream benchmark for measuring system bandwidth, and the Ping-pong benchmark for deriving the Hockney parameters for setup latency and inverse bandwidth.

We have found that the average particle transmission load between individual ranks quickly declines as the fluid simulation calms down. Even during the first half of the simulation when there is a lot of dynamic fluid motion, the communication cost constitutes a small fraction of the total execution time. (Less than 5 % for the dam-break simulation with Scale 1 and 41 ranks.) The program routines for finding particle pairs and computing the SPH kernel constitute

the main bottlenecks of the SPH application, and should therefore be the main concern of optimization.

For modeling field particles and particle pairs, and field particles and transmission counts, we propose a linear estimate that is independent of Scale. Furthermore, we find tha the Hockney model accurately describes the communication cost of inter-rank communication with low prediction-error rates using machine metrics from the Ping-pong benchmark. We also note that the setup latency is the dominating cost factor up to a certain limit defined by α_{ij}/β_{ij}.

Based on our experiments we provide the following summary and recommendations for running the SPH application on parallel systems:

- The operational intensity of the SPH application has a small lower bound and therefore benefits from implementing grid cells using persistent hash tables in order to reduce memory traffic.
- The SPH application benefits from subdomain discretization for simulations that have non-trivial amount of particles. Since a relatively large number of particles is needed for the simulation to behave realistically according to the fluid medium, cell-grid discretization will reduce the overall execution time significantly.
- There is a linear correspondence between the number of field particles and the number of particle pairs and exported particles. This can be used to analyze the workload per rank ahead of time when running on large parallel systems.
- The number of ranks is limited by the relationship between the cell size and the tank width. Therefore the SPH application will benefit from thread-level optimizations to reduce the pair-creation and kernel-execution routines.
- For simulations with little dynamic movement, the communication cost can be factored out, and the setup latency becomes the dominating factor.
- A compact affinity scheme should be used since ranks are organized in increasing order and every rank communicates with its left and right neighbors.

Bibliography

[1] K. J. Barker, K. Davis, A. Hoisie, *et al.*, "Using performance modeling to design large-scale systems," *Computer*, vol. 42, no. 11, pp. 42–49, Nov. 2009. DOI: 10.1109/MC.2009.372.

[2] R. A. Gingold and J. J. Monaghan, "Smoothed particle hydrodynamics — theory and application to non-spherical stars," *Monthly Notices of the Royal Astronomical Society*, vol. 181, pp. 375–389, Nov. 1977. DOI: 10.1093/mnras/181.3.375.

[3] L. B. Lucy, "A numerical approach to the testing of the fission hypothesis," *Astronomical Journal*, vol. 82, pp. 1013–1024, Dec. 1977. DOI: 10.1086/112164.

[4] J. Monaghan, "Simulating free surface flows with sph," *J. Comput. Phys.*, vol. 110, no. 2, pp. 399–406, Feb. 1994. DOI: 10.1006/jcph.1994.1034.

[5] M. Ozbulut, M. Yildiz, and O. Goren, "A numerical investigation into the correction algorithms for SPH method in modeling violent free surface flows," *International Journal of Mechanical Sciences*, vol. 79, pp. 56–65, 2014. DOI: 10.1016/j.ijmecsci.2013.11.021.

[6] P. Pacheco, *An Introduction to Parallel Programming*, 1st. Morgan Kaufmann, Inc., 2011.

[7] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560.

[8] J. L. Gustafson, "Reevaluating Amdahl's law," *Commun. ACM*, vol. 31, no. 5, pp. 532–533, May 1988. DOI: 10.1145/42411.42415.

[9] R. W. Hockney, "The communication challenge for MPP: Intel paragon and Meiko CS-2," *Parallel Computing*, vol. 20, no. 3, pp. 389–398, 1994. DOI: 10.1016/S0167-8191(06)80021-9.

[10] A. Lastovetsky, V. Rychkov, and M. O'Flynn, "Accurate heterogeneous communication models and a software tool for their efficient estimation," *IJHPCA*, vol. 24, pp. 34–48, Feb. 2010. DOI: 10.1177/1094342009359012.

[11] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990. DOI: 10.1145/79173.79181.

[12] D. Culler, R. Karp, D. Patterson, *et al.*, "Logp: Towards a realistic model of parallel computation," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '93, Association for Computing Machinery, 1993, pp. 1–12. DOI: 10.1145/155332.155333.

[13] A. Alexandrov, M. F. Ionescu, K. E. Schauser, *et al.*, "Loggp: Incorporating long messages into the logp model—one step closer towards a realistic model for parallel computation," in *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '95, Association for Computing Machinery, 1995, pp. 95–105. DOI: 10.1145/215399.215427.

[14] T. Kielmann, H. Bal, and K. Verstoep, "Fast measurement of logp parameters for message passing platforms," vol. 1800, Jan. 2000, pp. 1176–1183. DOI: 10.1007/3-540-45591-4_162.

[15] J. McCalpin, "Memory bandwidth and machine balance in high performance computers," *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, vol. 2, Dec. 1995.

[16] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. DOI: 10.1145/1498765.1498785.

[17] J. J. Dongarra, J. Du Croz, S. Hammarling, *et al.*, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, Mar. 1990. DOI: 10.1145/77626.79170.

[18] Q. Wang, X. Zhang, Y. Zhang, *et al.*, "AUGEM: Automatically generate high performance dense linear algebra kernels on x86 CPUs," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12. DOI: 10.1145/2503210.2503219.

[19] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th. Morgan Kaufmann, Inc., 2011.

[20] The MPI Forum, *MPI: A message passing interface standard*, version 3.1, 2015.

[21] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998. DOI: 10.1109/99.660313.

[22] J. Valstad and J. Ragunathan, "Performance modeling of CFD application scalability using co-design methods," Master's Thesis, 2018.

Appendix A

Listing A.1 SPH header file.

```
#ifndef COMMON_H
#define COMMON_H

#include <getopt.h>
#include <math.h>
#include <mpi.h>
#include <omp.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define MAX_ITERATION_DEFAULT 200000
#define CHECKPOINT_WRITE 100
#define CHECKPOINT_CUTOFF 128000

static const int OPTION_PRINT_DEBUG_INFO = 0;

typedef int64_t int_t;
typedef double real_t;

#define PI 3.141592654
#define TABLE_BUCKETS 16
#define PARTICLE_CAP 65536
#define PAIR_CAP 65536

#define MIN(X, Y) ((X) < (Y) ? (X) : (Y))
#define MAX(X, Y) ((X) > (Y) ? (X) : (Y))
#define CELL(x, y) (grid[((y) + 1) * (domain.n_cells_x + 2) + (x) + 1])

#define debug_printf(format, ...) do { \
   if (OPTION_PRINT_DEBUG_INFO) fprintf(stderr, format, __VA_ARGS__); \
} while (0)

enum { P_FIELD, P_VIRT, P_GHOST };

enum { C_MIGRATE, C_MIRROR, C_FIELD, C_PAIRS, C_PAIRS_PREDICT, NUM_COUNTERS };

```

```
41  enum { T_PAIRS, T_GRID, T_COMP, T_COMM, T_COMM_PREDICT, NUM_TIMERS };
42
43  /* Defined in sph.c */
44  extern int size, rank, east, west;
45  extern int_t n_field, n_global;
46  extern float SCALE;
47
48  /*****************************************************************************
49   *
50   * Benchmarking
51   *
52   *****************************************************************************/
53
54  #define BENCH_CHECKPOINT 1000
55
56  /*****************************************************************************
57   *
58   * Kernel parameters
59   *
60   *****************************************************************************/
61
62  /* Defined in sph.c */
63  extern real_t B; /* Tank width */
64
65  static const real_t density = 1e3;
66  static const real_t dt = 1e-4;
67  static const real_t sos = 50.0;
68
69  #define T (0.6 * SCALE) /* Dam height */
70  #define L (1.2 * SCALE) /* Dam width */
71  #define DELTA (0.01) /* Resolution (0.01) */
72  #define H (0.94 * DELTA * 1.4142135623)
73
74  #define CELL_SIZE (3.0 * H)
75  #define RADIUS (3.0 * H) /* Interaction radius */
76
77  #define X(k) ((domain.particles[(k)])->x[0])
78  #define Y(k) ((domain.particles[(k)])->x[1])
79  #define VX(k) ((domain.particles[(k)])->v[0])
80  #define VY(k) ((domain.particles[(k)])->v[1])
81  #define M(k) ((domain.particles[(k)])->mass)
82  #define RHO(k) ((domain.particles[(k)])->rho)
83  #define P(k) ((domain.particles[(k)])->p)
84  #define TYPE(k) ((domain.particles[(k)])->type)
85  #define HSML(k) ((domain.particles[(k)])->hsml)
86  #define INTER(k) ((domain.particles[(k)])->interactions)
87  #define INDVXDT(k, i) ((domain.particles[(k)])->indvxdt[(i)])
88  #define EXDVXDT(k, i) ((domain.particles[(k)])->exdvxdt[(i)])
89  #define DVX(k, i) ((domain.particles[(k)])->dvx[(i)])
90  #define DRHODT(k) ((domain.particles[(k)])->drhodt)
91  #define AVRHO(k) ((domain.particles[(k)])->avrho)
92  #define WSUM(k) ((domain.particles[(k)])->w_sum)
93
94  /*****************************************************************************
95   *
96   * Structures
97   *
98   *****************************************************************************/
```

```
 99
100 typedef struct world_t world_t;
101 typedef struct particle_t particle_t;
102 typedef struct domain_t domain_t;
103 typedef struct pair_t pair_t;
104 typedef struct bucket_t bucket_t;
105
106 struct world_t
107 {
108    int_t n_cells_x;
109    int_t n_cells_y;
110    int_t origin[2]; /* Cell origin */
111    int_t n_field;
112 };
113
114 struct domain_t
115 {
116    int_t n_cells_x;
117    int_t n_cells_y;
118    int_t origin[2]; /* Cell origin */
119    int_t n_field;
120    int_t n_virt;
121    int_t n_ghost;
122    particle_t **particles;
123    int_t particle_cap;
124    pair_t *pairs;
125    int_t pair_cap;
126    int_t n_pairs;
127 };
128
129 struct particle_t
130 {
131    int_t idx;
132    int_t local_idx;
133    int_t interactions;
134    int_t type;
135    int_t cell[2];
136    real_t x[2]; /* 2D position */
137    real_t v[2]; /* 2D velocity */
138    real_t mass; /* Mass */
139    real_t rho; /* Density */
140    real_t p; /* Pressure */
141    real_t hsml;
142    /* Differentials */
143    real_t indvxdt[2];
144    real_t exdvxdt[2];
145    real_t dvx[2];
146    real_t drhodt;
147    /* Density correction */
148    real_t avrho;
149    real_t w_sum;
150 };
151
152 struct pair_t
153 {
154    particle_t *ip;
155    particle_t *jp;
156    real_t r; /* Distance between particles */
```

```
   real_t q; /* Normalized distance */
   real_t w; /* Edge weight */
   real_t dwdx[2]; /* Influence on velocity */
};

struct bucket_t
{
   particle_t *particle;
   bucket_t *next;
};

#endif /* COMMON_H */
```

Listing A.2 Benchmark routines for measuring SPH metrics.

```
#include "common.h"

void bench_add_count(double *c_acc, int type, double count)
{
   c_acc[type] += count;
}

void bench_write_count(char *path, double *c_acc, int acc_size, long timestep)
{
   static int delta = 0;

   if (timestep == 0)
      return;

   delta = timestep - delta;

   if (rank == 0)
      MPI_Reduce(MPI_IN_PLACE, c_acc, acc_size, MPI_DOUBLE, MPI_SUM, 0,
          MPI_COMM_WORLD);
   else
      MPI_Reduce(c_acc, c_acc, acc_size, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

   if (rank == 0)
   {
      FILE *fp = fopen(path, "a");
      fprintf(fp, "%ld ", timestep);
      for (int i = 0; i < acc_size; i++)
         fprintf(fp, "%lf ", c_acc[i] / (double)delta);
      fprintf(fp, "\n");
      fclose(fp);
   }

   for (int i = 0; i < acc_size; i++)
      c_acc[i] = 0;

   delta = timestep;
}

void bench_write_rank_count(char *path, int src, double *c_acc, int acc_size, long
    timestep)
{
   static int delta = 0;

```

```
42    if (timestep == 0)
43       return;
44
45    delta = timestep - delta;
46
47    if (rank == src)
48    {
49       FILE *fp = fopen(path, "a");
50       fprintf(fp, "%ld ", timestep);
51       for (int i = 0; i < acc_size; i++)
52          fprintf(fp, "%lf ", c_acc[i] / (double)delta);
53       fprintf(fp, "\n");
54       fclose(fp);
55    }
56
57    for (int i = 0; i < acc_size; i++)
58       c_acc[i] = 0;
59
60    delta = timestep;
61 }
62
63 void bench_add_timing(double *t_acc, int type, double time)
64 {
65    t_acc[type] += time;
66 }
67
68 void bench_clear_timing(double *t_acc, int acc_size)
69 {
70    for (int i = 0; i < acc_size; i++)
71       t_acc[i] = 0;
72 }
73
74 void bench_write_rank_timing(char *path, int src, double *t_acc, long timestep)
75 {
76    static int delta = 0;
77
78    if (timestep == 0)
79       return;
80
81    delta = timestep - delta;
82
83    if (rank == src)
84    {
85       FILE *fp = fopen(path, "a");
86       fprintf(fp, "%ld ", timestep);
87       for (int i = 0; i < NUM_TIMERS; i++)
88          fprintf(fp, "%lf ", t_acc[i] / (double)delta);
89
90       double error = ((t_acc[T_COMM_PREDICT] - t_acc[T_COMM]) / (double)delta);
91
92       fprintf(fp, "%lf ", error);
93       fprintf(fp, "\n");
94       fclose(fp);
95    }
96
97    for (int i = 0; i < NUM_TIMERS; i++)
98       t_acc[i] = 0;
99
```

```
100     delta = timestep;
101  }
102
103  void bench_write_timing(char *path, double *t_acc, long timestep)
104  {
105     static int delta = 0;
106
107     if (timestep == 0)
108        return;
109
110     delta = timestep - delta;
111
112     if (rank == 0)
113        MPI_Reduce(MPI_IN_PLACE, t_acc, NUM_TIMERS, MPI_DOUBLE, MPI_SUM, 0,  ↩
               MPI_COMM_WORLD);
114     else
115        MPI_Reduce(t_acc, t_acc, NUM_TIMERS, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
116
117     if (rank == 0)
118     {
119        FILE *fp = fopen(path, "a");
120        fprintf(fp, "%ld ", timestep);
121        for (int i = 0; i < NUM_TIMERS; i++)
122           fprintf(fp, "%lf ", t_acc[i] / (double)delta);
123
124        /* Total time, excluding T_COMM_PREDICT */
125        double sum = 0;
126        for (int i = 0; i < NUM_TIMERS - 1; i++)
127           sum += t_acc[i] / (double)delta;
128        fprintf(fp, "%lf ", sum);
129
130        fprintf(fp, "\n");
131        fclose(fp);
132     }
133
134     for (int i = 0; i < NUM_TIMERS; i++)
135        t_acc[i] = 0;
136
137     delta = timestep;
138  }
139
140  void bench_write_idle(char *path, double *t_acc, double *t_idle, long timestep)
141  {
142     static int delta = 0;
143
144     if (timestep == 0)
145        return;
146
147     delta = timestep - delta;
148
149     /* Total runtime */
150     if (rank == 0)
151        MPI_Reduce(MPI_IN_PLACE, t_acc, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
152     else
153        MPI_Reduce(t_acc, t_acc, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
154
155     /* Total idle time */
156     if (rank == 0)
```

```
157         MPI_Reduce(MPI_IN_PLACE, t_idle, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
158      else
159         MPI_Reduce(t_idle, t_idle, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
160
161      MPI_Barrier(MPI_COMM_WORLD);
162
163      if (rank == 0)
164      {
165         FILE *fp = fopen(path, "a");
166         fprintf(fp, "%ld %lf %lf\n", timestep, *t_acc / delta, *t_idle / delta);
167         fclose(fp);
168      }
169
170      *t_acc = *t_idle = 0;
171      delta = timestep;
172   }
173
174   void bench_write_workload(char *path, long n_field, long timestep)
175   {
176      if (timestep == 0)
177         return;
178
179      long counts[size];
180
181      if (rank == 0)
182      {
183         counts[0] = n_field;
184         for (int r = 1; r < size; r++)
185            MPI_Recv(&(counts[r]), 1, MPI_LONG, r, 0, MPI_COMM_WORLD, ↩
                 MPI_STATUS_IGNORE);
186      }
187      else
188      {
189         MPI_Send(&n_field, 1, MPI_LONG, 0, 0, MPI_COMM_WORLD);
190      }
191
192      MPI_Barrier(MPI_COMM_WORLD);
193
194      if (rank == 0)
195      {
196         FILE *fp = fopen(path, "a");
197
198         long sum = n_field;
199         for (int r = 1; r < size; r++)
200            sum += counts[r];
201
202         fprintf(fp, "%.0lfK %ld ", timestep / (double)1000, sum);
203         for (int r = 0; r < size; r++)
204         {
205            fprintf(fp, "%ld ", counts[r]);
206         }
207         fprintf(fp, "\n");
208
209         fclose(fp);
210      }
211   }
```

Listing A.3 Ping-pong benchmark.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <inttypes.h>
#include <sys/time.h>
#include <mpi.h>

typedef double real_t;
typedef uint64_t int_t;

enum { T_ALPHA, T_BETA };

int rank, size;

MPI_Group world_group;

real_t *matrix;
real_t *vector;

char *send_buff;
char *recv_buff;

int *pairs;
int num_pairs = 0;

int_t MSG_SIZE_MAX = 100 * 1e6; /* 100 MiB */
int_t NUM_BENCH_MIN = 5;
int_t NUM_BENCH_MAX = 1000;

/* Root matrix */
#define MAT(k, x, y) matrix[(k) * size * size + size * (y) + (x)]

/* Local vector */
#define VEC(k, x) vector[(k) * size + (x)]

/* Communication pairs */
#define TUPLE(i, j) pairs[2 * (i) + (j)]

void create_tuples(void)
{
   num_pairs = (size * (size - 1) / 2);
   pairs = (int *)calloc(num_pairs * 2, sizeof(int));
   int k = 0;
   for (int i = 0; i < size - 1; i++)
   {
      for (int j = i + 1; j < size; j++)
      {
         TUPLE(k, 0) = i;
         TUPLE(k, 1) = j;
         k++;
      }
   }
}

void initialize_buffers(void)
{
   create_tuples();
   if (rank == 0)
```

```
59       matrix = (double *)calloc(2 * size * size, sizeof(double));
60    vector = (double *)calloc(2 * size, sizeof(double));
61    send_buff = (char *)calloc(MSG_SIZE_MAX, sizeof(char));
62    recv_buff = (char *)calloc(MSG_SIZE_MAX, sizeof(char));
63    double j = 0.0;
64    for (int_t i = 0; i < MSG_SIZE_MAX; i++)
65       send_buff[i] = ++j;
66 }
67
68 void comms1(int src, int dst, int_t message_size, int_t num_bench, int k)
69 {
70    double start, time;
71
72    start = MPI_Wtime();
73    if (rank == src)
74    {
75       for (int_t k = 0; k < num_bench; k++)
76          MPI_Ssend(send_buff, message_size, MPI_CHAR, dst,
77             0, MPI_COMM_WORLD);
78       for (int_t k = 0; k < num_bench; k++)
79          MPI_Recv(recv_buff, message_size, MPI_CHAR,
80             dst, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
81    }
82    if (rank == dst)
83    {
84       for (int_t k = 0; k < num_bench; k++)
85          MPI_Recv(recv_buff, message_size, MPI_CHAR,
86             src, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
87       for (int_t k = 0; k < num_bench; k++)
88          MPI_Ssend(send_buff, message_size, MPI_CHAR, src,
89             0, MPI_COMM_WORLD);
90    }
91    time = MPI_Wtime() - start;
92
93    if (rank != dst)
94       VEC(k, dst) = time / (2 * num_bench * message_size * sizeof(char));
95 }
96
97 void create_communicator(int src, int dst, MPI_Comm *COMM_PAIR)
98 {
99    int pair[2] = { src, dst };
100   MPI_Group group;
101   /* Create group */
102   MPI_Group_incl(world_group, 2, pair, &group);
103   /* Create communicator */
104   MPI_Comm_create_group(MPI_COMM_WORLD, group, 0, COMM_PAIR);
105 }
106
107 void run_comms1(int_t message_size, int_t num_bench, int k)
108 {
109    if (rank == 0)
110    {
111       if (k == T_ALPHA)
112          printf("Computing alpha...\n");
113       else
114          printf("Computing beta...\n");
115    }
116
```

```
117    int state = 0;
118
119    for (int i = 0; i < num_pairs; i++)
120    {
121       int src = TUPLE(i, 0);
122       int dst = TUPLE(i, 1);
123
124       MPI_Barrier(MPI_COMM_WORLD);
125
126       if (rank == src || rank == dst)
127       {
128          MPI_Comm COMM_PAIR;
129          create_communicator(src, dst, &COMM_PAIR);
130          MPI_Barrier(COMM_PAIR);
131          comms1(src, dst, message_size, num_bench, k);
132          MPI_Comm_free(&COMM_PAIR);
133       }
134
135       if (rank == 0)
136       {
137          switch (state)
138          {
139          case 0:
140             if (i / (double)num_pairs > 0.25)
141             {
142                state = 1;
143                printf("25%%\n");
144             }
145             break;
146          case 1:
147             if (i / (double)num_pairs > 0.50)
148             {
149                state = 2;
150                printf("50%%\n");
151             }
152             break;
153          case 2:
154             if (i / (double)num_pairs > 0.75)
155             {
156                state = 3;
157                printf("75%%\n");
158             }
159             break;
160          default:
161             break;
162          }
163       }
164    }
165
166    if (rank == 0)
167       printf("Done!\n");
168 }
```

Listing A.4 Stream benchmark.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <float.h>
```

```
 4  #include <limits.h>
 5  #include <sys/time.h>
 6  #include <omp.h>
 7
 8  #include "common.h"
 9
10  #ifndef STREAM_ARRAY_SIZE
11  #define STREAM_ARRAY_SIZE 10000000
12  #endif
13
14  static real_t a[STREAM_ARRAY_SIZE];
15  static real_t b[STREAM_ARRAY_SIZE];
16  static real_t c[STREAM_ARRAY_SIZE];
17
18  /* MIN, MAX, AVG */
19  static double timings[4][3] = {
20     [COPY] = { FLT_MAX, 0.0, 0.0 },
21     [SCALE] = { FLT_MAX, 0.0, 0.0 },
22     [ADD] = { FLT_MAX, 0.0, 0.0 },
23     [TRIAD] = { FLT_MAX, 0.0, 0.0 }
24  };
25
26  double BYTES_TRIAD = 0;
27
28  #define MIN(a, b) ((a) < (b) ? (a) : (b))
29  #define MAX(a, b) ((a) > (b) ? (a) : (b))
30
31  double walltime(void)
32  {
33     static struct timeval t;
34     gettimeofday(&t, NULL);
35     return ((double)t.tv_sec + (double)t.tv_usec * 1.0E-6);
36  }
37
38  int clock_ticks()
39  {
40     const int M = 20;
41
42     double t1, t2, time_value[M];
43
44     /* Collect a sequence of 'M' unique time values from the system */
45     for (int i = 0; i < M; i++)
46     {
47        t1 = walltime();
48        while(((t2 = walltime()) - t1) < 1.0E-6);
49        time_value[i] = t1 = t2;
50     }
51
52     /* Determine the minimum difference between these 'M' values. This result
53       will be the estimate (in microseconds) for the clock granularity. */
54     int min_delta = 1000000;
55     for (int i = 1; i < M; i++)
56     {
57        int delta = (int)(1.0E6 * (time_value[i] - time_value[i - 1]));
58        min_delta = MIN(min_delta, MAX(delta, 0));
59     }
60
61    return min_delta;
```

```
}

void run_stream(void)
{
   double t_start, t_latency;

   BYTES_TRIAD = 3 * sizeof(real_t) * (STREAM_ARRAY_SIZE * 1.0E-9);

   #pragma omp parallel for
   for (int j = 0; j < STREAM_ARRAY_SIZE; j++)
   {
     a[j] = 1.0;
     b[j] = 2.0;
     c[j] = 0.0;
  }

   #pragma omp parallel for
   for (int j = 0; j < STREAM_ARRAY_SIZE; j++)
   a[j] = 2.0 * a[j];

   real_t scalar = 3.0;

   for (int j = 0; j < NUM_BENCH; j++)
   {
      /* COPY */
      t_start = walltime();
      #pragma omp parallel for
      for (int i = 0; i < STREAM_ARRAY_SIZE; i++)
         c[i] = a[i];
      t_latency = walltime() - t_start;
      if (j > 0)
      {
         timings[COPY][MIN] = MIN(timings[COPY][MIN], t_latency);
         timings[COPY][MAX] = MAX(timings[COPY][MAX], t_latency);
         timings[COPY][AVG] += t_latency;
      }

      /* SCALE */
      t_start = walltime();
      #pragma omp parallel for
      for (int i = 0; i < STREAM_ARRAY_SIZE; i++)
         b[i] = scalar * c[i];
      t_latency = walltime() - t_start;
      if (j > 0)
      {
         timings[SCALE][MIN] = MIN(timings[SCALE][MIN], t_latency);
         timings[SCALE][MAX] = MAX(timings[SCALE][MAX], t_latency);
         timings[SCALE][AVG] += t_latency;
      }

      /* ADD */
      t_start = walltime();
      #pragma omp parallel for
      for (int i = 0; i < STREAM_ARRAY_SIZE; i++)
         c[i] = a[i] + b[i];
      t_latency = walltime() - t_start;
      if (j > 0)
      {
```

```
120            timings[ADD][MIN] = MIN(timings[ADD][MIN], t_latency);
121            timings[ADD][MAX] = MAX(timings[ADD][MAX], t_latency);
122            timings[ADD][AVG] += t_latency;
123         }
124
125         /* TRIAD */
126         t_start = walltime();
127         #pragma omp parallel for
128         for (int i = 0; i < STREAM_ARRAY_SIZE; i++)
129            a[i] = b[i] + scalar * c[i];
130         t_latency = walltime() - t_start;
131         if (j > 0)
132         {
133            timings[TRIAD][MIN] = MIN(timings[TRIAD][MIN], t_latency);
134            timings[TRIAD][MAX] = MAX(timings[TRIAD][MAX], t_latency);
135            timings[TRIAD][AVG] += t_latency;
136         }
137      }
138
139      for (int i = 0; i < 4; i++)
140         timings[i][AVG] /= ((int)NUM_BENCH - 1);
141
142      int threads = 0;
143      #pragma omp parallel
144      {
145         #pragma omp master
146         threads = omp_get_num_threads();
147      }
148   }
```

Appendix B

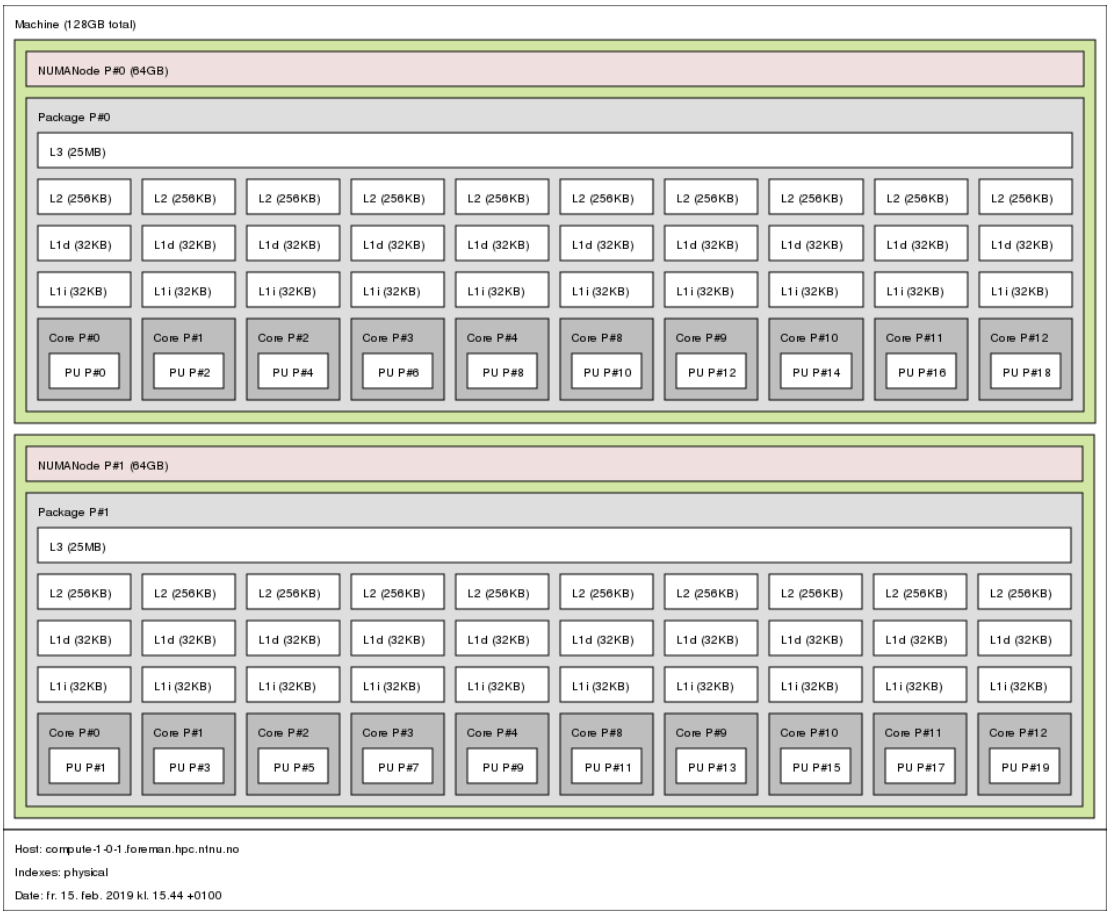


Figure B.1 Idun-E5 node topology.

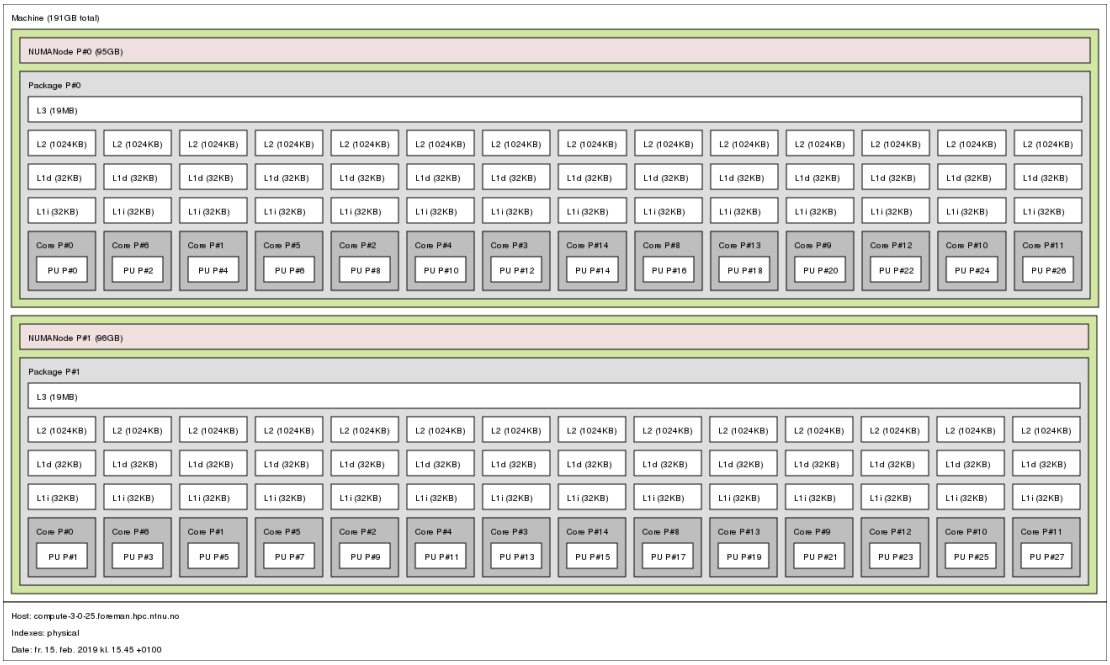


Figure B.2 Idun-Gold node topology.