MPI-Based Scalable Computing Platform for Parallel Numerical Application

by

Abdulaziz Albaiz

B.S., Computer Engineering (2007)
M.S., Computer Engineering (2010)
King Fahd University of Petroleum and Minerals

Submitted to the School of Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Computation for Design and Optimization

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2014

Signature redacted

Author .....
School of Engineering
Signature redacted August 18, 2014

Certified by .....
John R. Williams
Professor, Department of Civil and Environmental Engineering
Thesis Supervisor

Signature redacted

Accepted by .....
Nicolas G. Hadjiconstantinou
Professor, Department of Mechanical Engineering
Co-Director, Computation for Design and Optimization Program

MPI-Based Scalable Computing Platform for Parallel Numerical Application

by

Abdulaziz Albaiz

Submitted to the School of Engineering
on August 18, 2014, in partial fulfillment of the
requirements for the degree of
Master of Science in Computation for Design and Optimization

Abstract

Developing parallel numerical applications, such as simulators and solvers, involves a variety of challenges in dealing with data partitioning, workload balancing, data dependencies, and synchronization. Many numerical applications share the need for an underlying parallel framework for parallelization on multi-core/multi-machine hardware. In this thesis, a computing platform for parallel numerical applications is designed and implemented. The platform performs parallelization by multiprocessing over MPI library, and serves as a layer of abstraction that hides the complexities in dealing with data distribution and inter-process communication. It also provides the essential functions that most numerical application use, such as handling data-dependency, workload-balancing, and overlapping communication and computation. The performance evaluation of the parallel platform shows that it is highly scalable for large problems.

Thesis Supervisor: John R. Williams
Title: Professor, Department of Civil and Environmental Engineering

Contents

List of Figures

Chapter 1

Introduction

1.1 Parallel Numerical Simulation

The fields of scientific computing and numerical modeling and simulation have grown rapidly in the past few decades. They became widely acceptable as a third mode of discovery, complimenting theory and physical experimentation. This growth is due to not only the development of efficient numerical methods and techniques that are used these days, but also the explosive advancements in computer technology that enabled such methods to be conducted fast and in large scale.

Computers have changed dramatically in the last ten years. The development of microprocessors has moved from single-core to multicore processors. All computers today have processors with several cores inside them. This has pushed the development of parallel and distributed programming, as it is the only way to take advantage of today's computing power efficiently.

1.2 Research Motivation

Parallel programming is usually looked at as challenging. It is "conceptually harder to undertake and to understand than sequential programming, because a programmer

has to manage the coexistence and coordination of multiple concurrent activities" [1].

This thesis addresses the challenges that lie ahead of building parallel computationally-intensive numerical applications, such as solvers and simulators, that are efficient for both small and large problems, and scale well with increasing computing resources. Problems related to data- and resource-management are particularly tackled, and a general parallel programming platform is designed and implemented to act as a foundation for building such numerical applications.

1.3 Thesis Objectives

The objective of this thesis is to design and implement a parallel and scalable programming library that serves as a development platform for different computational applications such as numerical simulators. This library is meant to add a layer of abstraction that hides some of the complexities that most parallel software developers have to tackle, such as data distribution, inter-process communication and synchronization, and workload partitioning.

1.4 Thesis Contributions

The main contributions of this thesis can be summarized in the following points:

- Overview over the requirements of building a parallel numerical simulation platform.
- Tackling the challenge of workload partitioning among different parallel processes to achieve better load-balancing.
- Implementation of efficient techniques for data distribution and synchronization between different processes, as well as using overlapped communication and processing to hide the overhead.

- Evaluating the platform's performance and scalability by building different numerical applications, such as a finite-difference solver, and compare different workloads and system configurations.

1.5 Thesis Organization

The Thesis is organized into the following chapters. Chapter 2 explores the challenges of parallel programming, and the requirements for parallelizing numerical applications. It also gives an overview of the current parallel and distributed computing technologies and libraries that utilize techniques such as multiprocessing and multithreading. Additionally, an overview of the general requirements of parallel numerical applications is presented to serve as a guidance for building the proposed parallel platform.

In chapter 3, the design and implementation of the parallel platform is described in detail. The chapter starts by describing the architecture of the platform, along with the execution flow of a parallel numerical application. After that, a detailed explanation of the internal design of the platform is presented. This includes the technical approach used for data management, synchronization and communication.

The performance of the proposed parallel platform is then evaluated for scalability in chapter 4, where two different types of numerical applications are built using the platform.

The thesis finally concludes in chapter 5, where the overall contributions of the thesis are summarized, with a list of potential improvements and unresolved issues as future work.

Chapter 2

Parallelizing Numerical Applications

In this chapter, an overview of the requirements and tools for parallelizing numerical applications is presented. The chapters starts by explaining the motivations for using parallel and concurrent programming in numerical applications, followed by a description of the challenges and issues that are faced in software parallelization. A literature review of the available parallel programming tools and techniques is then presented. The chapter concludes with a discussion of how different types of numerical applications could be parallelized, and how to overcome the parallelization challenges.

2.1 Motivations for Parallel Programming

2.1.1 Development of Computer Technology

The development of computer processors has changed dramatically in the past decade. In the past, processors were developed with only one core that could execute a single stream of instructions. Multi-tasking in these single-core processors was achieved using concurrent processing, where context-switching between different running streams of instructions, or *threads*, was used. The technology of single-core processors was advanced over the years by decreasing the transistors' size, and in-

creasing the clock frequency. In early 2000s, a ceiling in frequency scaling was reached because of the heat dissipation barrier that prevented processors from having faster clocks. Multicore processors started to appear where a single processor has two or more processing cores that can run independent threads in true parallel fashion. Today, most CPUs in desktop computers, office workstations, servers, and even smart phones are multicore. Desktop computers nowadays have processors with up to 6 cores, while larger computer systems, such as servers, may have processors with up to 15 cores.

In addition to the use of multicore processors, multiprocessor computers have been used for a long time. Most servers today have two processor sockets, each has a multicore processor. A typical server has up to 24 physical cores. Moreover, multi-machine systems have also been popular for many applications. A computer cluster is a multi-machine system where a number of similar but independent computers are interconnected using a high-speed network to act as one powerful system. These type of computers are commonly used in data centers and research facilities. [2]

One could plainly see that today's computer systems are multiprocessor systems. Utilizing such systems require writing software that can run efficiently in parallel over multiple processors and machines. While parallel programming has been used for a long time, the recent development of multiprocessor technology has resulted in a growing interest in parallel computing.

2.1.2 Large Numerical Problems

Numerical applications are used to model and simulate real-world behavior of a system as an alternative to performing actual, physical experiments. In many cases, numerical simulations are used because experimentation is not feasible, due to the scale or complexity of the problem. This is why many numerical problems are generally large in scale, and require a lot of computing resources to be run in a reasonable time.

A single computer does not always have the sufficient resources to run large numerical applications. Some numerical problems use very large data that cannot fit in a single computer's memory. Other problems could take infeasibly long time to be solved on a single computer. Hence, the only way to run such large numerical applications is to use parallel and distributed algorithms so that they could utilize the resources of multiprocessor and multi-computer systems, and be executed in reasonable time.

2.2 Challenges in Software Parallelization

Developing parallel and distributed programs introduces new challenges that are not usually encountered in serial programs. Depending on the type of the problem, some of these challenges could potentially limit the parallelizability of an application, which therefore results in low speedup when comparing the performance of the parallel application to the serial version.

2.2.1 Workload Balancing

Developing a parallel algorithm usually requires breaking a large problem into smaller, but similar sub-problems. Each sub-problem uses a small subset of the data, known as the *working set*, to perform certain computation. Usually, these computations are done in parallel over multiple processors, where each processor is assigned one or more of these sub-problems.

To achieve the best performance, the amount of computation done by each processor should be similar. This results in better workload-balancing, and therefore higher performance speedup. Unbalanced workload, where one or more processors are spending more time performing computations than other processors, impacts the performance of the application, as the processors that do less work will become idle at each synchronization point, waiting for the other processors to finish their compu-

tation, before they can start the next task. This results in poor resource utilization, and lower speedup.

Workload balancing does not depend only on the size of the working set data, but also on the type of processing and computation that needs to be performed on that data. In many cases, it is easy to split the input data into smaller working sets of the same size. However, the amount of computation that each processor will perform on these working sets is not always possible to control or even predict. Certain algorithms perform a fixed amount of computation on the input data, which results in high workload balancing. On the other hand, the amount of processing other algorithms perform depends completely on the input data, and therefore, it is highly likely to have unbalanced workload among processors. This presents a challenge when designing parallel applications as the performance could be significantly impacted by this imbalance.

2.2.2 Data Dependency and inter-process Communication

The parallelizability of an algorithms depends heavily on the data-dependency between processes. When a process requires data from other processes during runtime, inter-process communication and synchronization must take place, which are considered an overhead and therefore have an impact on the parallel algorithm performance. Algorithms that require minimal or no communication between processes are highly parallelizable, as each subtask is run independently of others. On the other hand, algorithms that have a lot of inter-process data dependency are usually more difficult to parallelize.

In serial programs, the two main metrics for measuring performance are time and memory usage. An algorithm's cost can be determined by the amount of time it takes to run, and the amount of memory it requires. Both of these metrics are usually represented as a function of the input size. In parallel programs, *communication* is considered as a third cost metric, since it is an overhead that has an impact on the

overall performance.

Most numerical applications, such as simulators and solvers, have some form of data-dependency, and therefore require inter-process communication at a certain point. For example, a parallel simulator with time-stepping loop usually requires inter-process communication to exchange data that is needed to perform the computations in the next time-step. This means that there is a synchronization point at the end of each time-step. Fortunately, most of these data dependencies are *spatially-local*. That is, only a small subset of data, such as the outer data point of spatially discretized sub-domains, needs to be communicated. Hence, the communication overhead, while still present, is relatively small.

2.3 Parallelization Techniques and Tools

The increasing interest in parallel and distributed computing has resulted in the introduction of many tools and frameworks for developing parallel applications for different computing environments. Before discussing these tools, it is important to first introduce the different parallel computing architectures, and the different types of parallelism.

2.3.1 Multiprocessor Memory Models

The most basic computer system model includes a single processor connected to a uniformly-accessible memory through a fixed-width bus. Adding more processors to a computer system could be done in a number of ways. The two main approaches that most computer nowadays use are shared-memory architecture, and distributed-memory architecture.

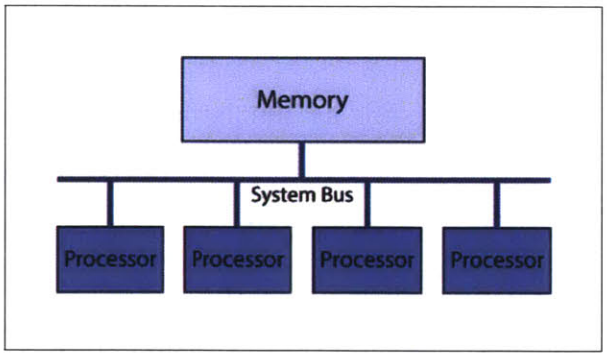

Figure 2-1: In shared-memory architecture, all processors are connected to the memory through the system bus

Shared-Memory Architecture

In shared-memory architecture, all processors in a computer system are connected directly to the memory through the main system bus. Since memory is shared, all processors have access the whole memory address space. Hence, changes in memory that are made by a program running in one processor can be observed immediately in other processors. Figure 2-1 shows an illustration of how processors and memory are connected.

The main advantage of shared-memory architecture is that synchronization and communication between processes is done through memory, and that the latest-written data is readily accessible by all processors. This reduces the overhead of communication. However, such architecture is not scalable, as only a limited number of processors can be connected to the system bus in one system.

Distributed-Memory Architecture

In distributed-memory architecture, each processor, or set of processors, is connected to a physically-separate memory, usually in a separate computer or server unit, as shown in figure 2-2. These computer units are interconnected through high-speed network to form a computer cluster. Because each unit has its own memory, the memory address space is not shared between processors, and therefore, synchronization

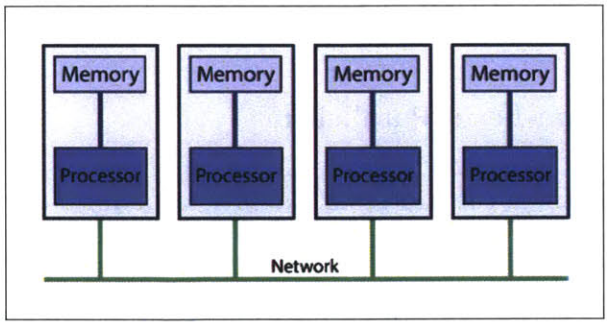

Figure 2-2: In distributed-memory architecture, processors are connected to separate memory modules, and are usually interconnected through a network

and communication are needed.

The overhead of communication and synchronization in distributed-memory architecture is higher, since it is done over the network. Even with high-speed networks, such as InfiniBand [3], that are used nowadays to interconnect computer clusters, the communication latency is still high compared to shared-memory architecture. However, the main advantage of distributed-memory architecture is its high scalability, as a large number of processors could be connected to build supercomputers.

Most computer cluster and supercomputers today use both architectures together; they are composed of a number of computer units that are interconnected using one or more high-speed networks, and each computer unit has two or more multicore processors. This hybrid approach takes advantage of both architectures, and hence provides high scalability.

2.3.2 Parallel Programming Models

There are several ways to write parallel programs. Today's processors are capable of performing several levels of parallelism. Some of them, such as data- and instruction-level parallelism, are built-in into the CPU architecture, and are performed transparently without the need for any specially programming. Task-level parallelism, on the other hand, requires actual parallel programming, where the developer has to write multiprocess or multithread application, and handle the needed

data distribution, communication, and synchronization. [4]

Data- and Instruction-Level Parallelism

Data-level parallelism is the processor's ability to perform operations on multiple data at the same time. This approach is called *Single-Instruction Multiple-Data* (SIMD). An instruction that performs an arithmetic or logical operation is executed over multiple data in the CPU at the same time.

Instruction-level parallelism, on the other hand, allows a single processor to execute multiple instructions at the same time, using different components of its datapath. Instructions that can be executed in parallel do not have any data-dependency between them. Most CPUs today can also perform out-of-order execution, where instructions can be executed in a different order than they were in memory, increasing the CPU's execution throughput. This, of course, is done only when the execution re-order does not affect the correctness of data. This type of parallelism is also done entirely by the processor, and does not require any changes in the code to take advantage of it.

Thread-Level Parallelism

Parallel programming usually refers to task-level parallelism, where the programmer has to write code that is executed concurrently and in parallel on one or more processors.

A process is the unit task that most operating systems deal with as an executing program. Within a process, one or more flows of instructions, or *threads*, are executed. Multiple processes and threads can be executed concurrently on a single processor.

Thread-level parallelism is the use of multiple threads of a single process to perform a certain task. Threads within a process share the same address space that belong to that process, and therefore can directly read and write data to that shared memory. Most processors today can execute multiple threads at the same time by using multiple

cores. Additionally, a single core can execute two threads at the same time using thread-level speculation, also known as *hyper-threading.*

In parallel applications that use multithreading, synchronization is crucial to avoid race-conditions, where two or more threads try to write to the same memory address, resulting in inconsistent data.

Because multithreaded programs run as a single process, they cannot be run on distributed-memory systems. Hence, shared-memory systems are the only suitable environment for multithreading.

Process-Level Parallelism

Process-level parallelism is a similar concept, but single-threaded processes are used as units of execution instead of threads. A multiprocess application launches a number of processes to perform a certain task. Unlike threads, each process has a separate memory address space, and processes do not have direct access to each other's memory space. Therefore, one or more inter-process communication (IPC) tools are used to communicate data between processes. Similar to multithreading, synchronization in multiprocessing is also needed when accessing shared resources, such as files on disk.

multiprocessing is suitable for both shared-memory and distributed-memory systems. Inter-process communication for processes that are running on a single system is usually done through memory using operating system tools, such as semaphores or pipes. When processes are running on multiple systems (in a distributed-memory environment), communication is usually performed over the network that interconnects these systems.

2.3.3 Parallel Programming Libraries

There are several libraries and frameworks for building multithreading and multiprocessing parallel applications. All modern operating systems provide Application-

Programming Interface (API) for creating and managing threads and processes. Additionally, a number of third-party libraries and standards are designed to provide more functionality and hide some of the system complexities.

Operating System API

The most direct way of writing parallel programs is to use the operating system's API for multithreading or multiprocessing. Operating systems provide means to create, control, synchronize and terminate processes and threads.

In Unix-based operating systems, the POSIX standard interface for threads and processes allows the programmer to make system calls to create processes and threads that are executed immediately in parallel to the main process or thread. It also provides a set of synchronization tools, such as mutexes, locks, and semaphores. The API for multithreading in POSIX is called *pthreads*.

Windows operating system also provides similar set of APIs and system calls for managing threads and processes. While the system calls are different under Windows system, the functions these APIs provides are essentially the same.

There are many third-party libraries that are written to hide the system-dependent APIs, and provide a standard way of creating and managing threads and libraries. This allow programmers to write portable code that can run on different systems. Boost library [5], for example, provides portable C++ multithreading functions that can be used to create and manage threads. Similarly, other multithreading libraries provide more advanced features, such as the use of worker-thread pools and job queues. The multithreaded library developed by Alhubail [6] for numerical application is an example.

OpenMP

OpenMP [7] is a programming interface for shared-memory systems. Unlike operating system APIs, OpenMP provides a simple way to create threads through the use

of programming directives in the code. OpenMP simplifies converting serial programs to parallel ones, as only few changes need to be made.

Parallelization using OpenMP is done by marking sections of the code using C++ preprocessor directives. These directives are also used to specify the shared and private variables in the code. At run time, OpenMP creates a number of threads to perform these parallel tasks.

OpenMP provides a simple and flexibly way to create multithreaded applications, and hides the complexity of dealing with the operating system low-level APIs. However, there are a number of cons for using OpenMP, such as its limited scalability and the lack of error handling.

Message-Passing Interface (MPI)

Message-Passing Interface (MPI) [8] is a popular standard for parallel programming. It has became the de facto standard for writing multiprocessing applications. MPI, as its name suggest, provides means for inter-process communication through message-passing, which are done within a single system through memory, or between multiple system through the network. This flexibility makes MPI suitable for distributed-memory systems.

A parallel application written using MPI can run on a single or multiple systems. MPI provides means for communicating data between processes, as well as tools for synchronization and process-management. Each processes is assigned an MPI process number, known as rank, that identifies it. Communication between processes could be performed point-to-point, where one processes sends data to another process, or collectively, where all the processes exchange data at the same time such that each process has the same data. Processes could also be grouped into subsets, called *worlds*, such that processes within one world could perform collective communications together.

MPI also hides the underlying communication and synchronization layers. The

medium of communication, whether through the main memory or the interconnecting network, is automatically picked by MPI based on where the communicating processes reside. This provides better performance when running multiple processes in the same system.

2.3.4 Hybrid Parallel Programming Model

A common approach to write large parallel applications is to use a hybrid combination of multiprocessing and multithreading. This two-level parallelism allows the application to take advantage of both the scalability of distributed-memory systems, and the performance of shared-memory systems. Multiprocessing is used such that each computer (or node) in the cluster is assigned one or two processes. Within each process, multithreading is used to create parallel threads that will be executed in different cores of the system processor.

MPI and OpenMP are used together to achieve this hybrid model of parallel programming. MPI is used to create and manage processes that will run on different nodes, while OpenMP is used to parallelize tasks within one process using multithreading. Figure 2-3 shows how processes and threads are created in such hybrid approach.

The parallel computing platform that is designed in this thesis uses MPI for process-level parallelism and data communication. Numerical applications built using the platform utilize OpenMP to achieve thread-level parallelism.

2.4 Numerical Applications

To build a programming platform for parallel numerical applications, it is important to understand how these applications could be parallelized, and what types of data dependency they have.

The goal is to provide a simple yet sufficient platform that has the required func-

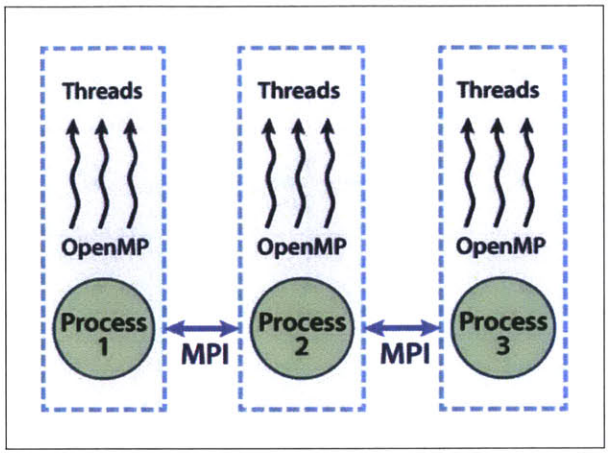

Figure 2-3: Hybrid model for parallel programming using MPI for multiprocessing and OpenMP for multithreading

tionality for writing numerical simulators and solvers that can be executed on parallel systems. As explained earlier in this chapter, the main challenges in parallelizing a program are balancing the workload, and handling data-dependency.

Numerical simulation methods could be classified based on their spatial data representation into two categories: *mesh-based* methods, and *mesh-free* methods.

2.4.1 Mesh-Based Numerical Methods

Mesh-based methods are the methods that use grids to represent their data points. Each cell of the grid represents one data point in the system. Examples of mesh-based methods are finite-difference and finite-volume methods. Throughout this thesis, finite-difference method will be used as the numerical application that represents mesh-based methods.

Data Dependency in Finite-Difference Method

In finite-different methods, the differential equations are solved by approximating the derivatives using Taylor expansion. The approximations are usually the sum of terms that represent the values at a set of points in the domain. The number of

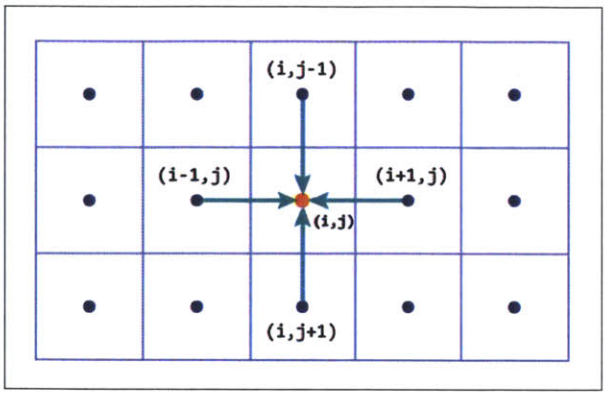

Figure 2-4: Second-order finite-difference in 2D uses a 5-point stencil

points used depends on the approximation order.

One of the common approximations that are used for 1D second-derivatives is given by the equation

$$\frac{\partial^2 f(x)}{\partial x^2} \approx \frac{f(x-\Delta x) - 2f(x) + f(x+\Delta x)}{\Delta x^2}$$

This central-difference second-order approximation uses 3 points to compute the approximate value of the derivatives.

The diffusion equation in 2D, for example, has a Laplacian term that can be approximated as

$$\begin{aligned}\Delta f(x,y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \approx & \frac{f(x-\Delta x, y) - 2f(x,y) + f(x+\Delta x, y)}{\Delta x^2} \\ & + \frac{f(x, y-\Delta y) - 2f(x,y) + f(x, y+\Delta y)}{\Delta y^2}\end{aligned}$$

This approximation uses 5 data points to approximate the derivative at a given point, as shown in figure 2-4. The 5-point stencil is used in this case. Higher-order approximations use more data points and larger stencils.

It is clear that the computation of data points require the use of other nearby data points. Fortunately, this type of dependency is spatially-local, i.e. the computation

depends only on points that are in close proximity. This limited-dependency makes it easier to parallelize such algorithms, as only a small part of the grid, namely the outer points of each sub-domain, need to be communicated to other process, while internal points can be computed directly.

Workload-Balancing in Finite-Difference Method

The amount of computation that is done at each data point in finite-difference method is almost the same. Data points near the boundaries may have different computations, but these are only a small fraction of the total number of data points. Moreover, data-points could be partitioned such that each process has similar number of boundary points, resulting in higher load-balancing. As long as the input size (number of data-points) that each process has is the same, the workload in finite-difference method will be highly balanced.

2.4.2 Mesh-Free Numerical Methods

Mesh-free methods do not use spatial grid for data representation. Instead, computations are performed directly on data-points. In some of these methods, data-points are not stationary in space, as their location is updated in every time-step. Smoothed-Particle Hydrodynamics (SPH) [9] is an example of mesh-free simulation, where data-points are represented as particles that interact with each other within a given spatial domain.

Data Dependency in Particle Simulation

In SPH, the computation of particle data, such as velocity and location, depends on the forces that act on that particle. The forces are usually computed using the data points of the neighboring particles within a certain radius, as figure 2-5 illustrates. This, again, represents a spatially-local data dependency, as computation on each particle depends only on a small subset of data points that resides within a given

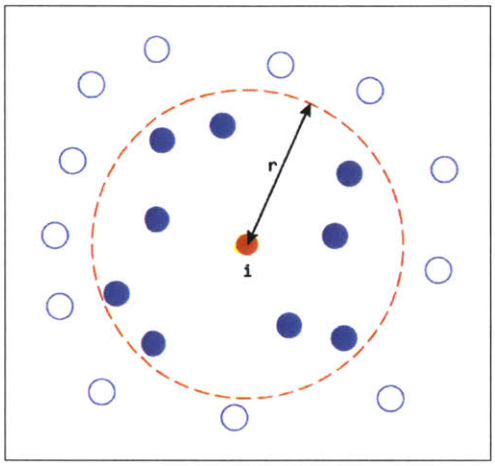

Figure 2-5: In SPH Simulation, forces are computed using data from neighboring particles within a certain radius

spatial volume. However, because SPH is a mesh-free method, data points are not sorted by their spatial location. This raises an issue when trying to resolve data-dependency in this method.

Since data need to be distributed such that particles within the same spatial region are stored together, the use of a *virtual grid* that sorts data points spatially is an efficient way of achieving this. This grid is meant only to keep particles ordered by their location, by assigning particles that belong to the same set of cells (or sub-domain) to a single process. This also makes the search for neighboring particles (that are needed to perform the computations) faster as only particles within the surrounding cells are used.

Workload-Balancing in Particle Simulation

Because of the dynamic nature of mesh-free methods, and how data points, such as particles, can move freely within the domain, load-balancing comes as a more challenging issue. Data is distributed among processes based on the particles' initial location using the virtual grid. When particles start to move during the simulation, they move from one sub-domain to another. This may cause an imbalance in the

workload of processes as one process may end up performing the computations on a large number of data points, while another process has a much smaller set of data points to compute. This presents a challenge in maintaining load-balancing between processes, which will be looked at in more detail in the next chapter.

2.5 Conclusion

There are a variety of tools for developing parallel applications. However, developing a parallel platform that targets numerical applications narrows down the scope into a smaller subset of tools, as these applications have certain requirements that need to be met.

Both mesh-based and mesh-free numerical methods have data dependency that is spatially-local. Hence, sorting and distributing data points by their spatial location is the most suitable way to achieve better parallelism.

Chapter 3

Design and Implementation of the Parallel Platform

This chapter describes the technical details in designing and implementing the parallel numerical application platform. The overall architecture of the platform is overviewed, showing how numerical simulations and solvers could be built seamlessly on top of the platform. This is followed by a detailed description of the different components and their implementation, such as the data structures used to represent data points, the algorithms used for workload partitioning and data exchange, and the handling of data-dependency.

3.1 Platform Architecture

One of the main objectives of building the parallel platform is to hide the underlying complexities of data-distribution, communication and synchronization, and to add a layer of abstraction that simplifies the process of building numerical applications on top of the platform. At the same time, it is important that the platform be flexible enough to allow different types of numerical applications to use it, and not limit the functionality of these applications.

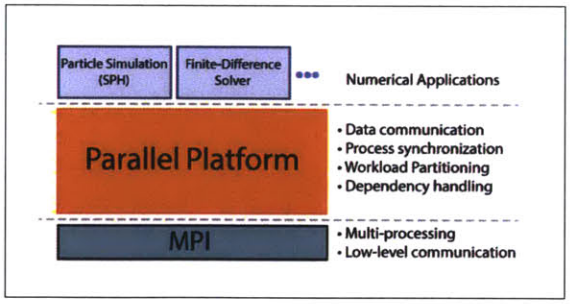

Figure 3-1: The conceptual architecture and abstraction layers of numerical applications that uses the parallel platform

3.1.1 Abstraction Layers

Figure 3-1 shows the general architecture of numerical applications built using the parallel platform. As described in the previous chapter, MPI is the most popular multiprocessing API that is used nowadays. MPI is used here by the parallel platform to provide the necessary multiprocessing functionalities, such as process management, and low-level inter-process communication.

The parallel platform handles data management, workload distribution, data dependency, and high-level inter-process communication. All these functions are required by most numerical applications. Different numerical methods can then be built at a layer on top of the platform, where the implementation is mainly focused on the physics and computation, rather than data management and parallelization. This form of abstraction makes building parallel numerical applications much simpler.

3.1.2 Parallel Execution Flow

A typical numerical application that is built on top of the parallel library would run as multiple processes using MPI. The flow diagram in figure 3-2 shows the general execution flow of these processes. The application starts by reading and distributing the input data. Depending on the type of application, input data could either come from files, or generated by the application. The partitioning of data is done spatially, and the distribution is performed by the parallel platform, where each process handles

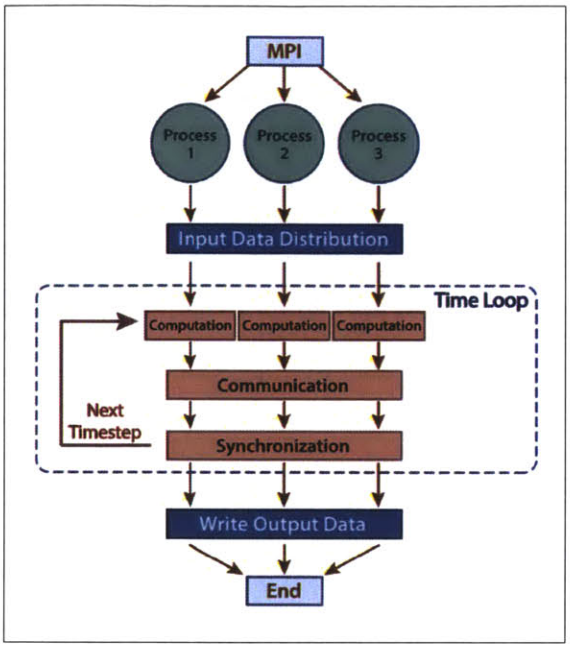

Figure 3-2: Execution flow of a parallel numerical application using the parallel platform

a subset of the input data that resides in its spatial sub-domain.

Once the input data in each process is ready to be processes, the application enters the time loop, where each process performs computations on its sub-domain data independently. After that, data is exchanged between processes. This is where data-dependency between processes is handled. Each process sends the parts of its data that other processes depend on, and receives parts of other processes data that it depends on. This inter-processes communication is also handled by the parallel platform. Processes are synchronized to ensure all data-exchanges are complete before proceeding to the next time-step. The subsequent time-steps are executed in a similar manner, where computations are done first, then communications and synchronization are performed.

At the end of the last time-step, the output data is written to files by all processes. The output writing procedure is flexible to allow parallel-writing, where multiple processes write to the same outfile file. The applications ends after the output data

is written.

In some applications, output data needs to be written at each time-step, or every given number of time-steps. In this case, output-writing will take place inside the time loop, after synchronization is done.

All the steps of this execution flow, except for the *computation* step, are performed by the parallel platform. This signifies the importance of this platform, as it hides most of these tasks that are usually repeatedly-implemented in each numerical application.

3.2 Data Management and Distribution

Data points are stored in arrays in the memory of each process. A virtual grid is used, for both mesh-based and mesh-free methods, to describe the spatial domain. This is important for both data partitioning, and handling data-dependency.

3.2.1 Virtual Grid

Because of the spatial-locality of dependent data, data is distributed over processes based on the location of data points. In mesh-base methods, such as finite-difference, the domain is discretized spatially, and therefore, the mesh itself is used for data distribution. On the other hand, mesh-free methods, such as particle simulations, deals with moving data points that aren't bound to a spatial location. In order to minimize data dependency between processes, data points that reside within the same spatial volume should be assigned to the same process. This raises a challenge since particles are not stationary and will change their location during the simulation time-steps.

To resolve this issue, a virtual grid is used. The purpose of the grid is to store data points that belong to the same spatial volume in contiguous memory arrays, and assigning these arrays to the same processes. As particles change locations during

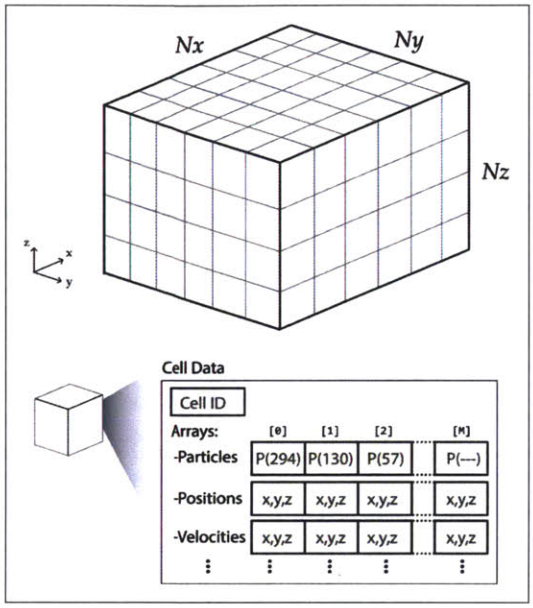

Figure 3-3: Data arrays are allocated such that each cell holds the data of the data points inside it

the simulation, these arrays are updated to keep their spatial-locality consistent.

The platform utilizes a uniform virtual grid that serves multiple purposes. Each unit of the grid is called a *cell*. Particle data are assigned to cells based on their location, i.e. a particle belongs to a specific cell if its location is within the cell's boundaries. Data arrays are partitioned over multiple processes using cell numbers. Each process is assigned a sub-domain that is represented as a sequence of cells and their corresponding particle data. A process is responsible for managing and processing the particles that belongs to its sub-domain.

The virtual grid, as shown in figure 3-3, consists of $N_x \times N_y \times N_z$ cells. The number of cells and cell size depend on the type of numerical application that is used. Data points are assigned to the cells that they reside in spatially. In particle simulation, for example, each cell holds the data of the particles that are contained within that cell.

3.2.2 Data Representation and structure

different numerical applications use different types of data points to perform the computations. In finite-difference method, for example, each cell represent a single data point with one or more properties. In particle simulation, on the other hand, each cell contain a different number of particles, and this number could dynamically change during the simulation. Moreover, each particle isn't represented by only one value, but a set of properties. Therefore, it is important to design the platform such that it allows for flexible data allocation for these different types of data.

To achieve such flexibility, data is represented as arrays that are allocated at the start of the application. Each array represent a property for the data points. The size of the array is set depending on the type of application. Cells of the virtual grid point to the corresponding addresses of these arrays that represent the starting location of the data points contained in these cells. An example of this data structured is shown in figure 3-3. Each cell carries the information related to the particles included in that cell, and that information is composed of a number of arrays, representing different properties of the particles.

The list of cells within the domain is also represented as an array. This array holds the cell information, such as the cell number, and pointers to the data arrays that belong to that cell, as shown in figure 3-4. Because data arrays are of fixed size, all the data arrays are continuous in memory, i.e. the data of a given cell starts immediately after the end of the data of the previous cell. This property reduces the time needed to allocate the arrays, as one allocation operation per property, with the total size of all the cells, is needed, instead of allocating a smaller array for each cell. This is also practical for applications where there is only one data point per cell, such as finite-difference method, as only one data array is allocated, instead of a scattered set of single-element arrays.

As the simulation proceeds, data points are moved from one cell to another based on their new spatial location. For example, particle locations are updated based

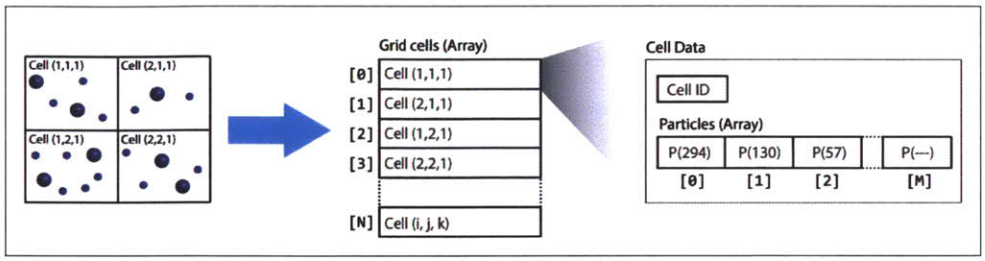

Figure 3-4: Particle data in cells

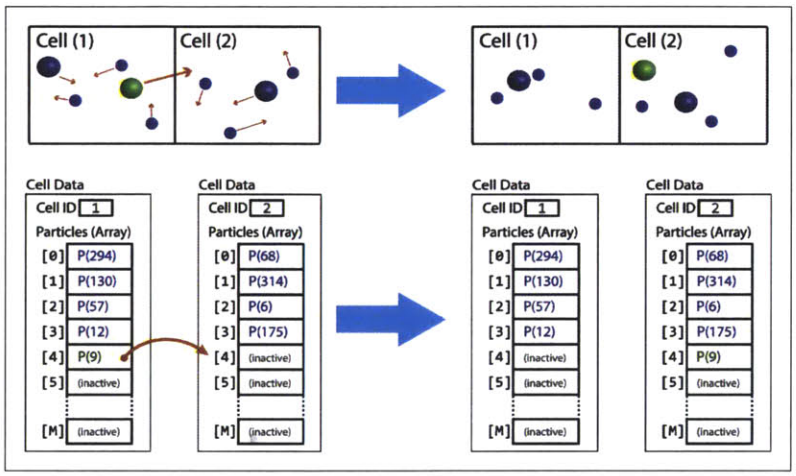

Figure 3-5: Particle motion between grid cells

on the computations that are performed. The new locations are used to find the containing cell of each particle. If the new particle location is in a different cell, its data is moved to the new cell, as shown in figure 3-5.

3.2.3 Time-Variant Data

There are two types of data arrays: time-variant and time-invariant. The difference is that time-variant data is updated in every time-step, while time-invariant data is fixed throughout the simulation. Particle locations, in particle simulation, for example, are time-variant data, while particle masses are time-invariant.

Every time-step, time-variant data is computed by using the values from the previous time-step. Hence, some form of temporary array is needed to keep the

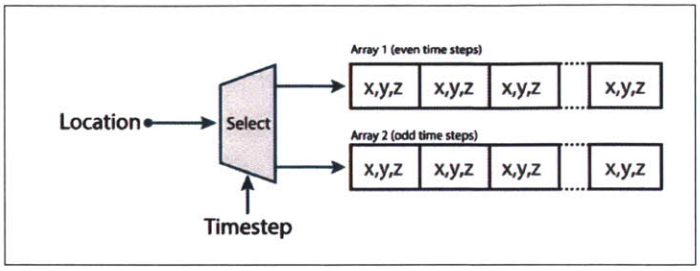

Figure 3-6: Time-variant data is allocated as two arrays, and are accessed based on the time-step

new computed values until the time-step is over, before writing it to data array and over-writing the previous-time-step data.

To achieve this, allocation of time-variant arrays is done by allocation two memory spaces of the same size. One is used to hold the data of the previous time-step, and the other is used to write the new computed values of the current time-step. At the end of each time-step, the points of these two arrays are swapped, rendering the current time-step data to be treated as the data from the previous time-step, and new values of the new times-step are written into the other array. This is illustrator is figure 3-6, where the particle location data array is accessed using the time-step as a selector to decide which array represent current data, and which one represent previous data.

The allocation of separate two arrays has an advantage when using multithreading in the application. Only one of the two arrays will be modified at each time-step, while the other array is read-only. This ensure that accessing these arrays is thread-safe, as threads can read data from any array elements, such as neighbor-particle data, but only one thread will be writing data to a given array element.

3.2.4 Data Partitioning

The use of the virtual grid to represent data in a spatially-ordered form enables data to be partitioned easily over multiple processes by simply breaking the domain

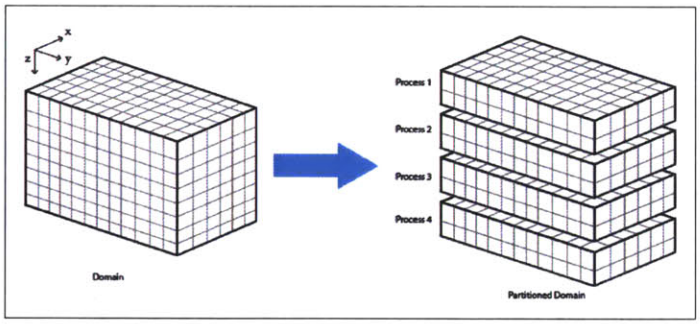

Figure 3-7: Data is partition by splitting the domain into a set of sub-domains

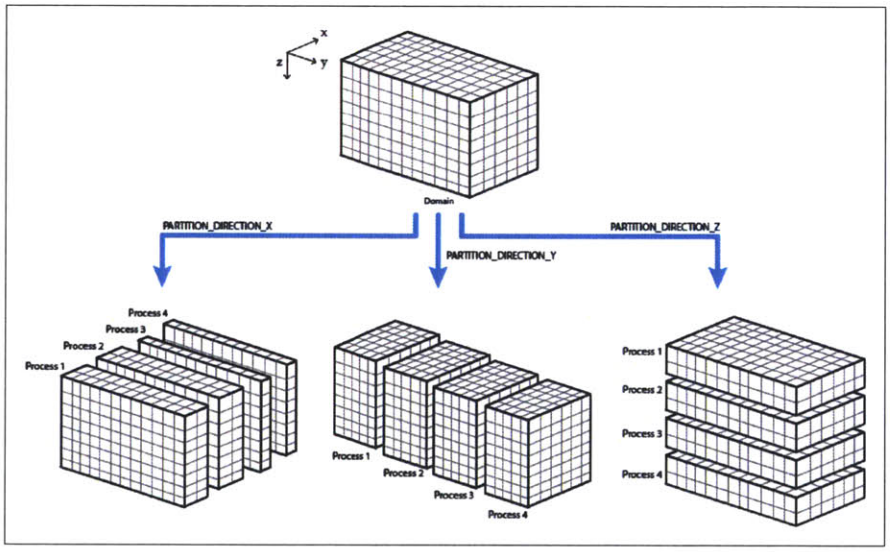

Figure 3-8: Domain partitioning in different directions

into smaller sub-domains, as shown in figure 3-7. Each sub-domain consist of a set of cells with their corresponding data arrays. Each process will have a partition of data that resides within the spatial boundaries of its sub-domain.

The choice of how the domain is partitioned could make a difference in the performance of the simulation. Partitioning the domain in the direction of a certain axis, such as the use of sub-domains composed of z-layers, results in different data distribution among processes. Figure 3-8 shows how a given domain could be partitioned into different sub-domains based on the selected partitioning direction.

Depending on the type of application, a certain partitioning direction can provide

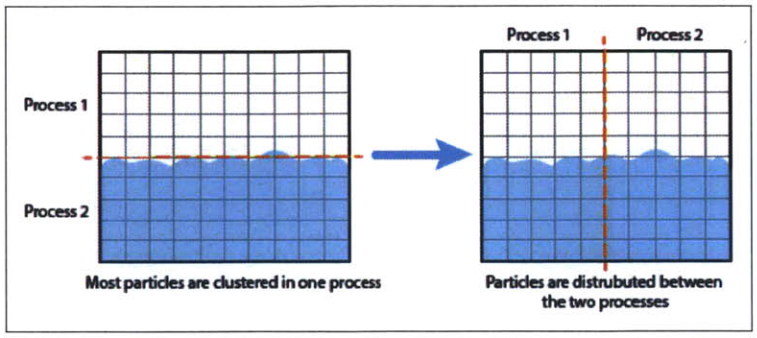

Figure 3-9: Load balancing using different domain partitioning directions

better workload balancing. An example of this is the simulation of fluid in a container, like the one shown in figure 3-9. In this example, partitioning the domain horizontally will result in assigning most of the data points to one process, while leaving the other process with very little data. On the other hand, vertical partitioning leads to much better load-balancing, as each process will have approximately the same data workload.

The partitioning direction choice is left as an option for the numerical application that is built on top of the platform, as this option is problem-dependent and cannot be easily predicted using the input data.

3.3 Data-Dependency and inter-process Communication

After the initial setup, where data arrays are allocated, populated with the input data, and distributed among processes, the numerical applications starts the main time loop. In each time-step, each process performs the computations on its data arrays, then all processes begin the communication process, where they exchange parts of the data based on their data dependency.

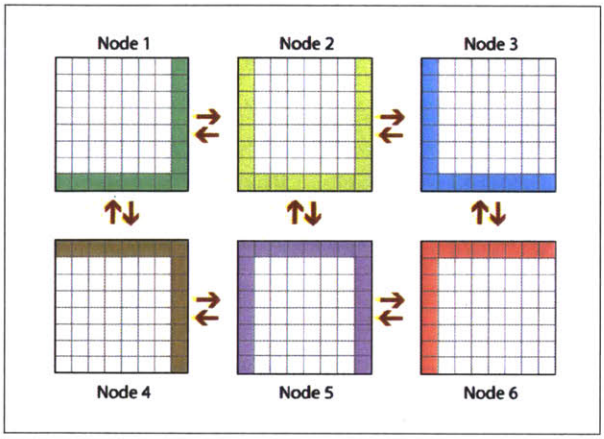

Figure 3-10: Exchange of ghost region data between processes

3.3.1 Ghost Regions

In the previous chapter, it was shown that data-dependency in most numerical methods is spatially-local. Because the domain partitioning performed by the parallel platform is done using the virtual grid, which is a spatial organization of data points, data-dependency between processes can be represented as the set of cells of the sub-domain that resides spatially at the edge of that sub-domain.

The implementation of such dependency in the parallel platform is accomplished by using neighboring cells. A given particle that belongs to a certain cell is surrounded by other particles that belong to either the same cell, or one of the directly neighboring cells. In the case of 3D grid, a cell has 26 direct neighbors.

When a cell and all its neighbors belong to the same processor, the computation of data can be done directly, as all the needed data are available and accessible in that processor's memory. This is the case for most cells that belong to the *interior* of the sub-domain. On the other hand, when a cell has neighbors that belong to one or more different processors, the data of these neighboring cells need to be retrieved first before any computation could be done. These kind of cells are called the *ghost region*. These ghost-regions are the parts of data that need to be exchanged between processes in each time-step. Figure 3-10 shows an example of six sub-domains, their

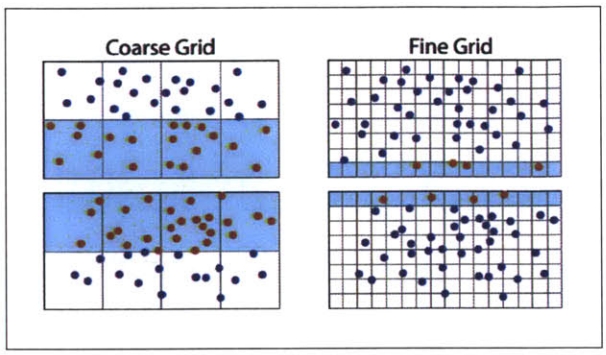

Figure 3-11: Smaller cell size results in smaller ghost regions, and therefore, less inter-process communication

ghost regions, and the data exchange that each process, or node, needs to perform.

For time-variant data arrays, the exchange of ghost region data is done only for the newly computed data arrays, that represent the current time-step. The actual exchange of ghost region data is done for each destination process. The ghost region cell data is collected into a continuous large array, then sent to the destination process. This is performed by each process to every other destination that has dependency on its data.

The size of the ghost region depends on the number of cells in the domain, the number of processes (or sub-domains), and the partitioning direction. The use of a larger grid, or smaller cells, result in smaller ghost regions, which means less data to be exchange, as figure 3-11 illustrates. However, there is a trade-off between the cell size and the simulation time-step size. Smaller cells may limit the movement of data points in the domain, as no data point is allowed to travel across more than one cell per time-step.

3.3.2 Computation and Communication Overlap

inter-process communication over MPI takes place in different mediums, depending on where the communicating processes reside. If the two processes are on the same computer, the communication takes place over memory. If, on the other hand,

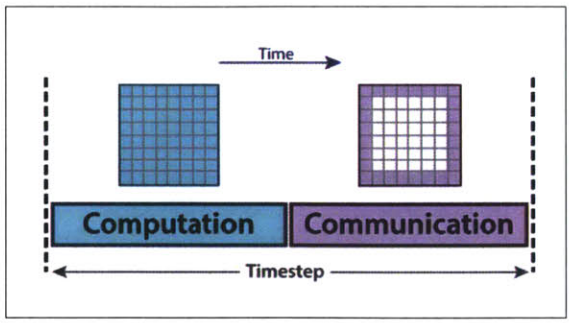

Figure 3-12: Without overlapping, exchange of ghost regions takes place after all the computations are complete

the communicating processes are running on two different nodes of a computer cluster, they communicate through the interconnecting network between these nodes. Either way, communication is costly when compared to computation, since it is an I/O-bound operation.

The execution flow described earlier in this chapter shows that communication of ghost regions happens after the computations are fully performed on data. The simulation time of a single time-step, as shown in figure 3-12, is composed of the computation time, and the communication time. Larger ghost regions, or larger number of communicating processes, will result in higher communication cost.

One way to improve the performance and reduce the required time for performing a time-step is to overlap communication and computation. To do that, the computation part starts by performing the necessary computations on the ghost region cells first. Since ghost regions are generally small compared to the sub-domain size, this part of the computation would be completed in a short time. At this point, communication is initiated in a separate thread, while the main thread continues to perform the computations on the remaining cells of the sub-domain. This overlap of communication and computation reduces the time required for each time-step, and can result in much better performance.

Figure 3-13 illustrates the time line for a single time-step with overlapping communication and computation. Communication is initiated as soon as the computations

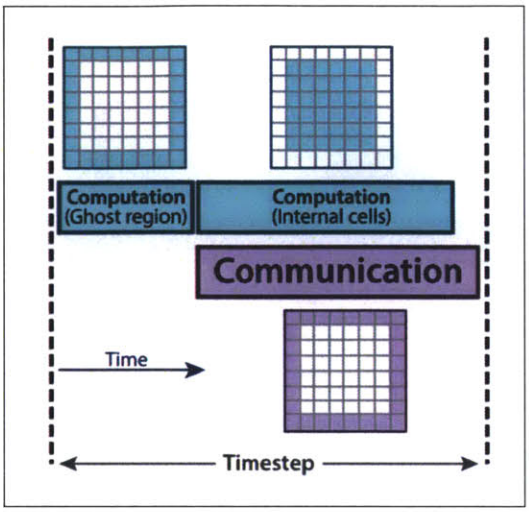

Figure 3-13: By performing the computations on the ghost regions first, communication of ghost regions could start earlier and overlap with the remaining of the computations

are done for the ghost region cells.

3.3.3 Neighbor Search

In particle simulations, such as SPH, neighbor particles are used to perform computation on the current particle. Similarly in mesh-based methods, the neighbor cells are used as data points to perform computations on the current data point.

At each time-step, data of ghost regions are exchanged between processes. After this exchange, each process will have data arrays of all the cells surrounding its sub-domain. Hence, the data of all the neighboring cells of the process' sub-domain cells are locally accessible. Hence, neighbor search can be performed immediately to find all the surrounding cells and their data points, and use them for the computation at that time-step.

Figure 3-14 illustrates an example of two processes that have exchange their ghost region data. Each process now has the necessary data required to perform neighbor search for any of its sub-domain cells. Once the computations are done, updated ghost region data is exchanged again, as described earlier, and the time-step advances.

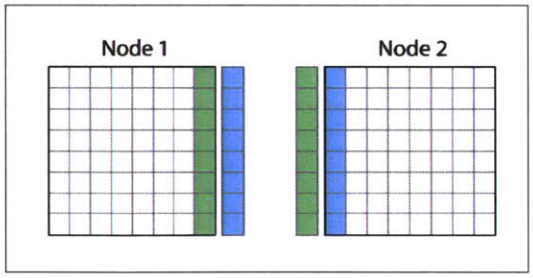

Figure 3-14: Once ghost-region data is exchanged, each process can perform neighbor search for all its cells

Depending on the type of numerical application, more than one layer of cells may need to be exchanged as ghost region. The *depth* of the ghost region is determined by the requirements of the computations that are performed. For example, higher-order finite-difference approximations require the use of several data points in each direction. The implementation of such solver would require exchanging several layers of cells between processes, increasing the communication cost.

3.4 Conclusion

The proposed parallel platform provides most of the necessary parallelization functionalities to the numerical applications that are built on top of it. These functions, as explained in this chapter, include data management and inter-process communication. In the next chapter, actual numerical applications will be built using the parallel platform to evaluate its performance.

Chapter 4

Testing and Performance Evaluation

Two parallel numerical applications will be implemented on top of the parallel platform. The first is a finite-difference solver, which is an example of a mesh-based method. The other application is a particle simulation, representing an example of a mesh-free method. The objective is to evaluate the performance and scalability of the parallel platform for these two applications.

4.1 Finite-difference Solver

The finite-difference solver application uses the heat equation in 3D grid using second-order central-difference approximation.

4.1.1 Solver Setup

The heat differential equation that will be implemented in the solves is given as

$$\frac{\partial u}{\partial t} - k\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) = 0$$

where $u(x, y, z, t)$ represent the temperature at a given point (x, y, z) at time t. The domain is assumed to be 3D cube of length 1 on each side.

The initial condition is set to as a function $u(x, y, z, 0) = f(x, y, z)$, and the boundary conditions at each side are also set to the same function value, that is:

$$u(0, y, z, t) = f(0, y, z) \qquad u(1, y, z, t) = f(1, y, z)$$
$$u(x, 0, z, t) = f(x, 0, z) \qquad u(x, 1, z, t) = f(x, 1, z)$$
$$u(x, y, 0, t) = f(x, y, 0) \qquad u(x, y, 1, t) = f(x, y, 1)$$

The second-order finite-difference approximation for the Laplacian operator over a discretized domain uses a 7-point stencil, and can be expressed as

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \approx \frac{u_{i-1,j,k} - 2u_{i,j,k} + u_{i+1,j,k}}{\Delta x^2} + \frac{u_{i,j-1,k} - 2u_{i,j,k} + u_{i,j+1,k}}{\Delta y^2} + \frac{u_{i,j,k-1} - 2u_{i,j,k} + u_{i,j,k+1}}{\Delta z^2}$$

The first derivative with respect to time is approximated using forward Euler, resulting in the time integration equation:

$$\begin{aligned}
\frac{u^{t+1}_{i,j,k} - u^t_{i,j,k}}{\Delta t} &= -k\left(\frac{u^t_{i-1,j,k} - 2u^t_{i,j,k} + u^t_{i+1,j,k}}{\Delta x^2} + \frac{u^t_{i,j-1,k} - 2u^t_{i,j,k} + u^t_{i,j+1,k}}{\Delta y^2} + \frac{u^t_{i,j,k-1} - 2u^t_{i,j,k} + u^t_{i,j,k+1}}{\Delta z^2}\right) \\
\implies u^{t+1}_{i,j,k} &= -k\Delta t\left(\frac{u^t_{i-1,j,k} - 2u^t_{i,j,k} + u^t_{i+1,j,k}}{\Delta x^2} + \frac{u^t_{i,j-1,k} - 2u^t_{i,j,k} + u^t_{i,j+1,k}}{\Delta y^2} + \frac{u^t_{i,j,k-1} - 2u^t_{i,j,k} + u^t_{i,j,k+1}}{\Delta z^2}\right) + u^t_{i,j,k}
\end{aligned}$$

The simulation is considered converged when the change in the vector u over a time-step, that is, $\|u^{t+1} - u^t\|$, is smaller than a threshold $\epsilon = 10^{-6}$.

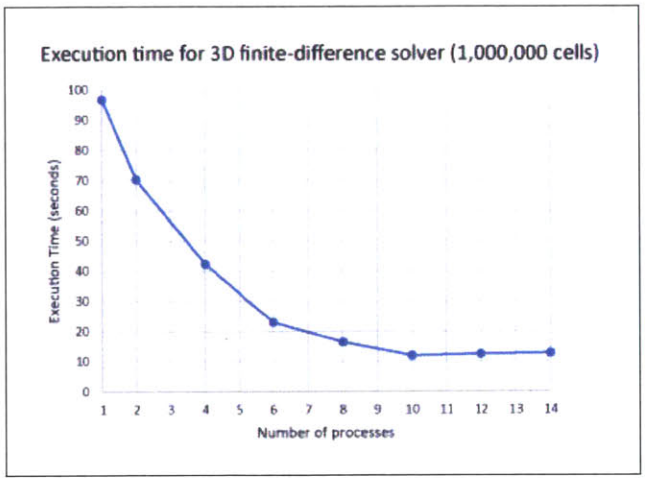

Figure 4-1: Finite-difference solver execution time for 1,000,000 cells

4.1.2 Performance Analysis

The finite-difference solver is run for two difference scenarios, one is using a domain with $100 \times 100 \times 100 = 1{,}000{,}000$ cells, and the other uses $200 \times 200 \times 200 = 8{,}000{,}000$ cells.

The first scenario, with 1,000,000 cells, is run multiple times with different number of parallel processes, and the solver run time is collected for each run. The plot in figure 4-1 shows the execution time for each case. As expected, running more parallel processes results in decreased execution time, compared to the 1-process run.

It is also noticed that at some point, adding more parallel processes does not effect the performance anymore. In reality, having too many processes could have a negative impact on performance. There are several reasons for that. First, the communication overhead increases as the number of processes increase, since more inter-process communication is needed for the ghost regions. Additionally, having smaller sub-domains in each process results in less computation time, and hence most of the time-step is spent on communication rather than communication.

In the second scenario, where a finer grid of 8,000,000 cells is used, we notice that the impact of having too many processes is delayed, as there is more data for

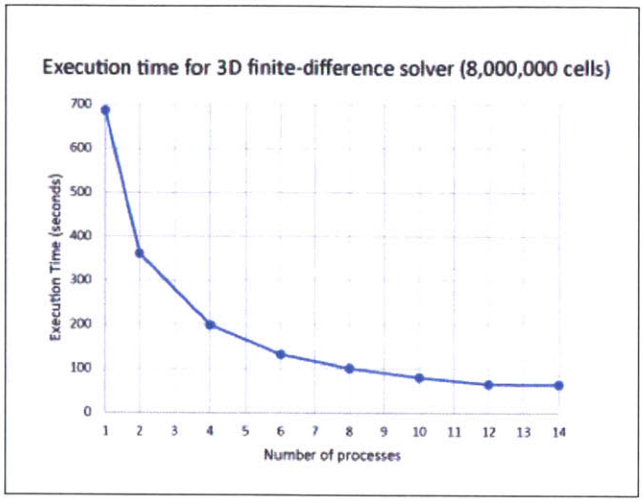

Figure 4-2: Finite-difference solver execution time for 8,000,000 cells

computation per process than the previous scenario, as seen in figure 4-2. However, there isn't much improvement in performance as the number of parallel processes exceeds 10 processes.

To measure the scalability of this solver, the speedup for each case is computed as

$$\text{Speedup}(n) = \frac{\text{Execution time of n processes}}{\text{Execution time of 1 process}}$$

. The ideal speedup of n parallel processes is n.

The plot in figure 4-3 shows the execution-time speedup for both 1,000,000-cell and 8,000,000-cell scenarios. It is noticed that the parallel platform scales well as the number of processes increases up to 10 processes. After that, the communication overhead starts to affect the performance negatively, which results in longer execution time.

One could also observe from the speedup plot that the parallel platform scales better for larger data set. The 8,000,000-cell scenario has better speedup for small number of processes, and is only slightly affected by the communication overhead once the number of parallel processes exceeds 10.

This could be extrapolated to conclude that the parallel platform performs better

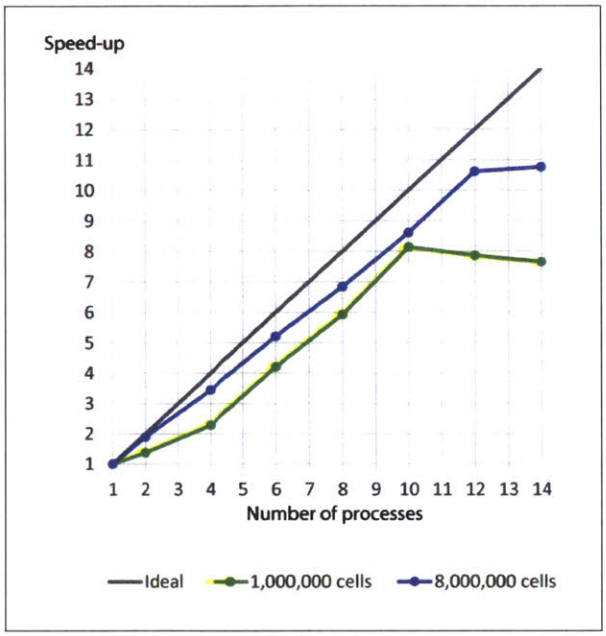

Figure 4-3: Finite-difference solver execution-time speedup

for larger problems. The two test-cases were relatively small, and hence there was not sufficient computations per time-step to keep the processes busy. However, for much larger problems, it is expected that the application will scale well for a large number of processes.

4.2 Particle Simulation

The particle simulation application is a simple motion-simulation of particles within a spatial domain. Particles are represented by their location, velocities, and accelerations.

4.2.1 Simulation Setup

The main equations that will be used in this application are the simple motion equations. Each particles has 4 properties: Particle number n, position p, velocity v, and acceleration a. Arrays are initialized with random data, but a seeded random

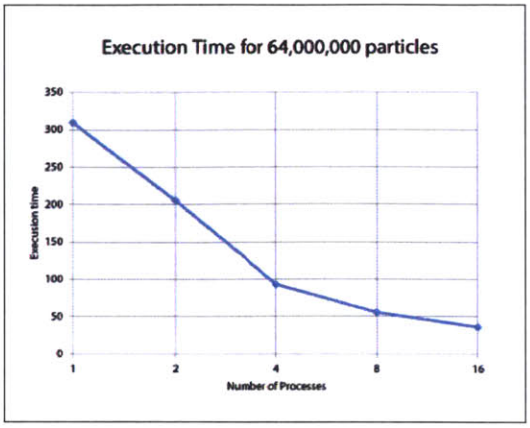

Figure 4-4: Particle simulation execution time for 64,000,000 particles

generator is used to ensure that the same data is reproduced for every run.

The acceleration a is time-invariant. It is fixed throughout the simulation. Position p and velocity v are computed at each time step using the following equations

$$v^{t+1} = v^t + \Delta t * a$$

$$p^{t+1} = p^t + \Delta t * v^t$$

The boundaries of the domain are reflective, i.e. when a particle reaches the boundary, its motion direction is reversed so that it moves back towards the domain's center.

4.2.2 Performance Analysis

The simulation is run with a $100 \times 100 \times 100$ cell virtual grid, with 64 particles per cell. Different number of processes is used to evaluate the scalability. Fixed time-steps are used, and the simulation is run for 60 time-steps.

The execution time is measured for each run. Figure 4-4 shows the execution time needed by each run at different number of processes. Similar to the behavior that was noticed before, the performance improves as the number of processes is increased.

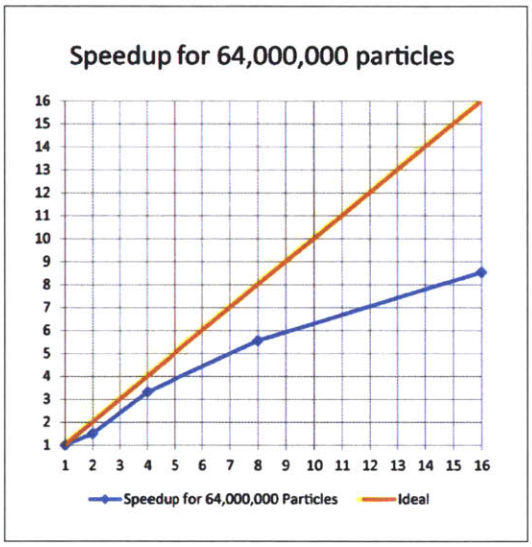

Figure 4-5: Speedup for particle simulation with 64,000,000 particles

At some point, however, doubling the number of processes has insignificant affect on the simulation run time. There are two reasons for this. First, the communication overhead starts to appear as more processes are added. The other reason is that in particle simulation, unlike finite-difference, the workload is not balanced. Some processes may contain more particles than others, resulting in an imbalance that impacts performance.

The measured speedup can also be seen in figure 4-5. For the above-mentioned reasons, the increase of number of processes beyond a certain point impacts the speedup negatively. Using less processes is more suitable for this problem, as the grid size and number of particles is relatively small.

4.3 Conclusion

The performance of the parallel platform shows high scalability for large problem. The speedup of execution time is found to be high when the number of processes is increased. For problems that do not have enough computation per time-step, the communication overhead takes over and causes a decrease in performance as the

number of parallel processes is increased. However, for larger problem, the parallel platforms shows scalable performance.

Chapter 5

Conclusion

This chapter concludes the thesis by outlining the main contributions achieved. Additionally, some of the open problems and potential improvements are discussed for future work.

5.1 Summary of the Contributions

In this thesis, the following work has been achieved:

1. A highly-scalable parallel framework for numerical applications is designed and implemented.

2. A number of techniques for workload partitioning and data distribution and inter-process synchronization are implemented.

3. Different numerical applications, such as finite-difference solver, are implemented on top of the framework, and used to evaluate the framework's performance and scalability.

5.2 Future Work

Many issues in the proposed framework are open for further research. Additionally, there are many potential performance improvements that could also be implemented. Some of these are:

- **Adaptive data-partitioning:** The static data-partitioning approach used in the framework is based on the the initial location of data points, and it does not take into account the possible changes that could happen to their location during the simulation. Dynamic data re-partitioning could be used to improve load-balancing, and therefore, the overall performance. Research in this area involves finding ways to automatically detect when data-repartitioning is needed, which data-partitioning approach is best for the current state of data-points, and how often should data be repartitioned to achieve maximum performance and hide the repartitioning cost.

- **Use of non-restrictive spatial domain:** The current implementation of the platform assumes a bounded spatial domain, where all the particles or data points are within that space. Additionally, the data-structures used in the implementation represent fully-allocated arrays at each cell. This kind of implementation, while suitable for dense problems with many data points per cell, may not be ideal for sparse problems. Hence, a potential improvement for the parallel platform is to design data structures that are more suitable for sparse, unbounded spatial domains.

- **Dynamic addition and removal of MPI processes:** The performance and scalability of the parallel libraries depends highly on the problem size and the amount of computation performed at each time-step. During the performance evaluation, it was noticed that communication overhead could slow down the application if the time-step computations are not large. One way to provide

better scalability is to implement dynamic addition and removal of MPI processes as the simulation progresses. The platform should be able to measure the amount of computation and communication that is done in each time-step, and estimate the optimal number of MPI processes that are needed to achieve maximum scalability.

Bibliography

[1] W. Hasselbring, "Programming languages and systems for prototyping concurrent applications," *ACM Computing Surveys (CSUR)*, vol. 32, no. 1, pp. 43–79, 2000.

[2] B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice Hall, second ed., 2004.

[3] G. F. Pfister, "An introduction to the Infiniband architecture," in *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pp. 617–632, IEEE Press, 2001.

[4] F. Gebali, *Algorithms and Parallel Computing.* Wiley, 2011.

[5] R. Demming, *Introduction to the Boost C++ libraries.* Amsterdam, The Netherlands: Datasim Education BV, 2010.

[6] M. Alhubail, "A thread-based parallel programming library for numerical algorithms," Master's thesis, Massachusetts Institute of Technology, June 2014.

[7] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[8] Message Passing Interface Forum, "MPI: A message-passing interface standard version 3.0," tech. rep., Knoxville, TN, USA, September 2012.

[9] 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, no. 3, pp. 375–389, 1977.