MASSIVELY PARALLEL ALGORITHM FOR SOLVING THE EIKONAL EQUATION ON MULTIPLE ACCELERATOR PLATFORMS

by

Anup Shrestha

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

December 2016

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Anup Shrestha

Thesis Title: Massively Parallel Algorithm for Solving the Eikonal Equation on Multiple Accelerator Platforms

Date of Final Oral Examination: 14 October 2016

The following individuals read and discussed the thesis submitted by student Anup Shrestha, and they evaluated his presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Elena A. Sherman, Ph.D.	Chair, Supervisory Committee
İnanç Şenocak, Ph.D.	Co-Chair, Supervisory Committee
Steven M. Cutchin, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Elena A. Sherman, Ph.D., Chair of the Supervisory Committee. The thesis was approved by the Graduate College.

ACKNOWLEDGMENTS

I would like to thank my advisors Dr. Elena Sherman and Dr. İnanç Şenocak for their continuous guidance and support throughout the course of this Master's thesis. I would also like to thank Dr. Steven Cutchin for serving on my thesis supervisory committee. Many thanks to Boise State University and Jason Cook in particular for help with the local computing infrastructure and installation of various tools and libraries. I would also like to thank my colleagues Rey DeLeon and Micah Sandusky, from the High Performance Simulation Laboratory for Thermo-Fluids, for their help and willingness to answer my questions. Finally, I would like to thank my family for their continued support.

I am grateful for the funding during my graduate studies, provided by the National Science Foundation under grants number 1229709 and 1440638.

ABSTRACT

The research presented in this thesis investigates parallel implementations of the Fast Sweeping Method (FSM) for Graphics Processing Unit (GPU)-based computational platforms and proposes a new parallel algorithm for distributed computing platforms with accelerators. Hardware accelerators such as GPUs and co-processors have emerged as general-purpose processors in today's high performance computing (HPC) platforms, thereby increasing platforms' performance capabilities. This trend has allowed greater parallelism and substantial acceleration of scientific simulation software. In order to leverage the power of new HPC platforms, scientific applications must be written in specific lower-level programming languages, which used to be platform specific. Newer programming models such as OpenACC simplifies implementation and assures portability of applications to run across GPUs from different vendors and multi-core processors.

The distance field is a representation of a surface geometry or shape required by many algorithms within the areas of computer graphics, visualization, computational fluid dynamics and more. It can be calculated by solving the eikonal equation using the FSM. The parallel FSMs explored in this thesis have not been implemented on GPU platforms and do not scale to a large problem size. This thesis addresses this problem by designing a parallel algorithm that utilizes a domain decomposition strategy for multi-accelerated distributed platforms. The proposed algorithm applies first coarse grain parallelism using MPI to distribute subdomains across multiple nodes and then fine grain parallelism to optimize performance by utilizing accelerators. The results of the parallel implementations of FSM for GPU-based platforms showed speedup greater than $20\times$ compared to the serial version for some problems and the newly developed parallel algorithm eliminates the limitation of current algorithms to solve large memory problems with comparable runtime efficiency.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AMD – Advanced Micro Devices, Inc.

API – Application Programming Interface

ASCII – American Standard Code for Information Interchange

CPU – Central Processing Unit

CUDA – Compute Unified Device Architecture

DRAM – Dynamic Random Access Memory

FLOPS – Floating Point Operations per Second

FMM – Fast Marching Method

FSM – Fast Sweeping Method

GPGPU – General-Purpose computation on Graphics Processing Units

GPU – Graphics Processing Unit

HDF5 – Hierarchical Data Format 5

HPC – High Performance Computing

I/O – Input/Output

MIC – Many Integrated Cores

MIMD – Multiple Instruction Multiple Data

MISD – Multiple Instruction Single Data

MPI – Message Passing Interface

NetCDF – Network Common Data Formt

OpenACC – Open Accelerators

OpenCL – Open Computing Language

OpenGL – Open Graphics Library

OpenMP – Open Multi Processing

PCI – Peripheral Component Interconnect

POSIX – Portable Operating System Interface

SIMD – Single Instruction Multiple Data

SISD – Single Instruction Single Data

SM – Streaming Multiprocessors

XDR – eXternal Data Representation

CHAPTER 1

INTRODUCTION

1.1 Problem Context

Imagine a robot that wants to make its way from point A to point B in an environment with static and moving objects. For the robot to reach its destination efficiently and safely, it needs to find a collision free shortest path from its current location to the destination. To avoid collision the robot needs to know the distance between it and the surrounding objects. Using geometric primitives to calculate the distance between the robot and other objects each time it moves is very inefficient. Instead collision detection/avoidance algorithms utilize the concept of a distance field. Distance field is a gridded structure where each cell in the grid is a point in space, value of which represents the shortest distance between that point and the boundary of the closest object. Distance field can be signed or unsigned. Signed distance fields store the sign to distinguish whether the query point lies inside or outside of an object. The advantage of using distance field is that it can approximate the distance from any arbitrary point to the nearest object in $O(1)$ time, independent of the geometric complexity of the object [3].

Likewise, distance field is used to solve problems in numerous areas of research like computer graphics [1], computational fluid dynamics [29], computer vision and robotics [3, 11, 35], etc. However, distance field calculation for complex geometries and interactive applications (like motion planning) where distance field must be computed periodically, can be computationally expensive and therefore, fast computation methods still remain a topic of research. In general, speedup is achieved either by developing an algorithm with better performance or by executing the existing code on a machine with high processing power. In the last decade, the processing power of a single CPU has stalled and designers have shifted to a multi-core architecture where multiple CPUs are integrated into single circuit die. The idea here is to use parallelism by which higher data throughput may be achieved with lower voltage and frequency [24]. This means that executing the existing code on multi-core architecture results in no performance increase. Therefore, developers could no longer rely on hardware upgrades to increase the performance. Instead parallel algorithms are the key on taking advantage of the multi-core architecture. Further increase in performance could be achieved by leveraging the computational power of advanced hardware platforms such as the Graphical Processing Unit (GPU) or Intel Xeon Phi co-processor. They have adopted the multi-core architecture since their inception and can offer much higher throughput and combined processing power than the multi-core CPUs.

Orthogonal to performance increase, computation of distance field faces the issue of memory limitation when a domain is either large or has been refined into a finer grid for a high resolution. If the memory requirements of the domain exceeds the available system memory then the calculation cannot be performed. The solution is to store the grid using a file format that supports parallel I/O. With parallel I/O a part of the decomposed domain can be loaded into memory. This approach enables the development of an efficient and scalable algorithm for multi-core architecture. The following section reviews the available parallel computing platforms and its evolution.

1.2 Review of Processor Hardware and Accelerators

Moore's Law, a prediction by Gordon Moore that powered the information technology revolution since the 1960s, is nearing its end [39]. Moore's law states that "the number of transistors on a chip would double every two years" [39]. However the trend has stalled around 2005 with maximum clock frequencies around 2-3.5 GHz. This is due to the increased dynamic power dissipation, design complexity and increased power consumption by transistors [24]. Hence, the formula for increasing the number of transistors on a chip drastically changed in 2005 when Intel and AMD (Advanced Micro Devices, Inc.) both released dual-core processor designs with the release of Pentium D and Athlon X2 respectively. The idea was to use parallelism to achieve higher data throughput with lower voltage and frequency [24]. Since then CPU design has shifted to a multi-core architecture where two or more processor cores are incorporated into a single integrated circuit for enhanced performance, reduced power consumption and more efficient simultaneous processing of multiple tasks. Modern processors have up to 24 cores on a single chip (e.g. Intel Xeon E7-8890 v4 processor has 24 cores) [12] making it a shared-memory parallel computer. A parallel computer is a set of processors that are able to work cooperatively to solve a computational problem. It includes supercomputers, network of workstations and multi-core processor workstations. As a result of this trend in multi-core processors parallelism is becoming ubiquitous, and parallel programming is becoming a central part of software development.

On the other hand, in recent years GPUs have been gaining a lot of traction as a general processor platform for parallel computing because of its combination of high computational throughput, energy efficient architecture, high performance and low cost. GPUs are designed as a many-core architecture that offers thousands of cores on a single chip (e.g. NVIDIA's Tesla K40 has 2880 cores). These cores are light and simple with small cache size and low clock frequencies. GPUs are based on the streaming processor architecture [18] that is suitable for compute-intensive parallel tasks such as graphics rendering. With the evolution of computer graphics, the GPU hardware is becoming more powerful. The floating-point performance and memory bandwidth of today's GPUs are orders of magnitude faster than any CPU in market as shown in Figure 1.1. Other parallel processing hardwares include Intel's co-processors. The many integrated core (MIC) architecture of Intel's Xeon Phi is an example of a co-processor. It has anywhere from 57-61 cores running approximately at 1GHz clock frequency. Xeon Phi has 4 hardware threads per core and can provide up to a trillion floating point operations per second (TFLOP/s) performance in double precision. However the actual sustained performance are usually lower than the peak performance and depends on the numerical algorithm. The MIC and GPU architecture are collectively referred to as accelerators.

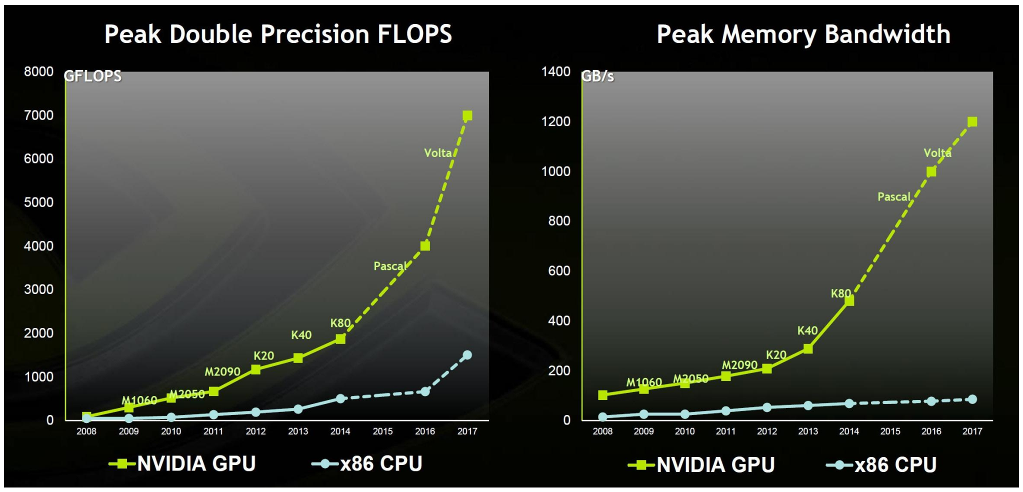

Figure 1.1: NVIDIA GPU performance trend from 2000 to 2017 A.D. [17]

1.3 Software Engineering for Scientific Computing

Scientific computing domain combines realistic mathematical models with numerical computing methods to study real world phenomenon using modeling and computer simulation. Such studies are carried out with the help of complicated software implementing highly sophisticated algorithms and concurrent techniques (using MPI, OpenMP) to improve performance and scalability. The amount of data and calculations processed by such studies require high performance computing (HPC) resources to run specialized software often written by research scientists without formal computer science background and limited exposure to proper software development processes. Hence, they are faced with software engineering challenges during the development process. Therefore, the code is often unmanaged, tightly-coupled, hard to follow and impossible to maintain or update. This leads researchers to spend more time maintaining code, then doing actual research.

The art of designing software and writing quality code has been developed over decades of practice and research. Although object-oriented programming techniques have increased productivity in software development, most of the research in this area is dedicated to sequential code for single processors (CPU). Modern parallel platforms like multicore/manycore, GPUs, distributed or hybrid, require new insight into development process of parallel applications due to their increased complexity. Programming massively parallel computers is at an early stage where the majority of the applications contain numeric computations, which were developed using relatively unstructured approaches [14]. At the same time GPUs are becoming increasingly popular for general purpose computing but their implementation complexity remains a major hurdle for their widespread adoption [16]. Current software engineering practices are not applicable to most of the present parallel hardware resulting in low quality software that is difficult to read and maintain. Therefore, it is necessary to develop advanced methods to support developers in implementing parallel code and also in re-engineering and parallelizing sequential legacy code.

There has been growing research for lowering the barrier for programming GPUs. CUDA programming model from NVIDIA made GPU programming mainstream and user-friendly. However, when compared to writing programs for general-purpose CPUs, developing code for GPUs is still complex and error-prone [16]. Recently, several directive based, accelerator programming models, such as OpenACC, OpenMP(v 4.0), etc were proposed. They offer higher level of abstraction and less programming effort for code restructuring and optimization. "In the directive-based accelerator programming models, a set of directives are used to augment information available to the designated compilers, such as guidance on mapping of loops on to GPU and data sharing rules" [16]. In such model the designated compiler hides most of the complex details of the underlying architecture to provide a very high-level abstraction of accelerator programming. Parallel regions of the code to be executed on GPU are annotated with specific directives. Then, the compiler automatically generates corresponding host+device code in the executable allowing an incremental parallelization of applications [16]. One of the many advantages of using directive-based parallel programming is that the structure of the code remains exactly the same as in the serial version. Hence, software engineering practices need not be changed for applications accelerated through the use of such languages. Another advantage of directive based parallel programming is portability allowing execution of code on any accelerated platform (such as GPUs, multicore CPUs, co-processors, etc) just by setting a compiler flag.

1.4 Thesis Statement

1.4.1 Objectives

The distance field computation can be generalized into a non-linear partial differential equation governed by the eikonal equation. The current state of the art sequential method for solving the eikonal equation is the Fast Sweeping Method (FSM) and there also exists parallel algorithms for FSM. However, the parallel algorithms have not been implemented for accelerators (i.e, GPUs and co-processors). Furthermore, none of the algorithms solve the issue of memory limitation hence, problems with large domain size cannot be solved using those implementations. Therefore, the prime objective of the present research is the "design and implementation of parallel algorithms for solving the eikonal equation to improve performance and scalability while requiring modest addition of programming effort to implement them". The research investigates and develops multi-level parallel computing strategies for accelerators using numerical methods to solve the eikonal equation. Numerous parallelization strategies including MPI, CUDA, OpenACC, MPI-CUDA and MPI-OpenACC implementations are shown. Challenges to achieving scalable performance are presented and relevant literature reviews are included in each chapter.

1.4.2 Procedures

The following tasks were accomplished as part of the research:

- Survey of the eikonal equation solver and methods.
- Implement serial version of FSM.
- Implement parallel versions of FSM using MPI, CUDA and OpenACC.
- Design and implement a hybrid parallel approach using combination of MPI-CUDA and MPI-OpenACC.
- Performance comparison between the sequential and parallel implementations.
- Design and implement a multi-level parallel FSM algorithm for multiple accelerator platforms using MPI, OpenACC and the NetCDF-4 file format and its parallel I/O capabilities.

The serial C version of the FSM is used as a benchmark for performance analysis. The following CPU-GPU high performance computing (HPC) platform was used for execution, performance and scaling analysis:

- `2U Dual Intel Xeon E5-2600`
  `Intel QuickPath Interconnect (QPI)`
  `Integrated Mellanox ConnectX-3 FDR (56Gbps) QSFP InfiniBand port`
  `(8×) 8 GB DDR3 Memory 1600 MHz`
- `Dual NVIDIA Tesla K20 ‘‘Kepler’’ M-class GPU Accelerator`
  `2496 CUDA Cores`
  `5GB Total Memory (208 GB/sec peak bandwidth)`

- `GeForce GTX TITAN PCI-E 3.0`

  `2688 CUDA Cores`

  `6GB GDDR5 Memory (288.38 GB/sec peak bandwidth)`

The implementation of these different tasks and the analysis of the results are presented in this thesis.

CHAPTER 2

BACKGROUND

2.1 Simulation Problem: Signed Distance Field

As mentioned in the introduction, distance field can be signed or unsigned. A signed distance field is a representation of the shortest distance from any point on the grid to the interface of an object represented as a polygonal model. The interface describes the isocontour of a shape in two or more dimensions and is defined by a signed distance function ϕ. Figure 2.1 shows an interface represented by ϕ in a two-dimensional ($\mathbb{R}^2$) grid. For a point $P \in \mathbb{R}^2$, the signed distance from P to the interface is positive if P is outside its boundary, negative if P is inside its boundary and zero if P lies on the interface itself. In Figure 2.1, point P has a positive distance value since it is located outside of the interface boundary. Figure 2.2 exhibits a slice of the signed distance field of complex three dimensional geometries.

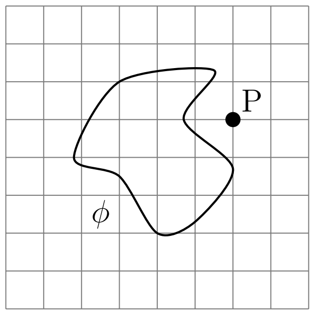

Figure 2.1: Signed distance calculation for a point on 2D grid with interface ϕ

2.1.1 Calculating Signed Distance Field

There are two general approaches for calculating the signed distance field. The first is geometric that includes brute-force, distance meshing [11], scan conversion [8], prism scan [32] etc. For example, in a brute-force approach, M line segments are drawn from a point on the grid to the boundary of the closest object. Then the minimum distance to all M line segments is calculated for each point of interest. The algorithmic complexity of the brute-force approach is $O(N * M)$ where N is the total number of grid points and M is the number of line segments. The implementation of such methods are complex and can be very slow compared to numerical methods.

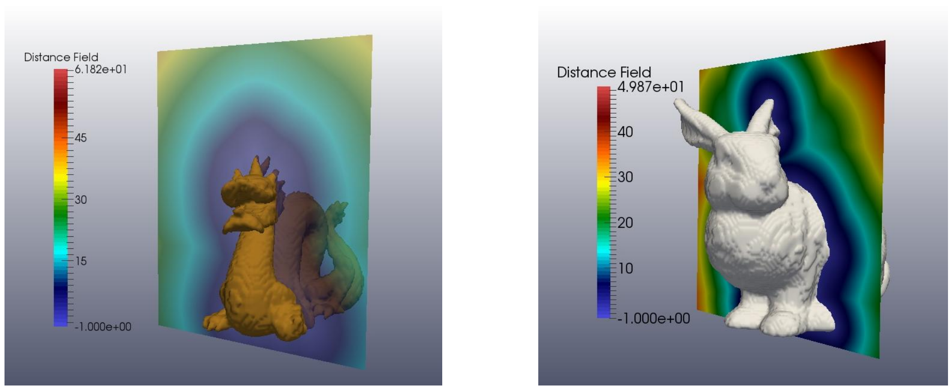

Figure 2.2: Distance field visualization of the Stanford Dragon (left) and the Stanford Bunny (right)

The second approach applies numerical methods to calculate the distance field. Fortunately mathematicians were able to generalize the distance field problem to a non-linear partial differential equation governed by the eikonal equation. Since, the brute force approach is computationally expensive and therefore, impractical to use in most applications, this research focuses on solving the eikonal equation.

2.1.2 The Eikonal Equation

The word "eikonal" is derived from the Greek word *eikon* meaning image and the solution of the eikonal equation determines the shape of an object. For example, the eikonal equation can describe the flux of secondary electrons in a scanning electron microscope. These secondary electrons are used to modulate appropriate devices to create a shape of the object by solving the eikonal equation [5].

Definition The *eikonal equation* is a first order, non-linear, hyperbolic partial differential equation of the form

$$\begin{aligned} |\nabla\phi(x)| = f(x),\ \text{for } x \in \Omega \subset \mathbb{R}^n, \\ \phi(x) = g(x),\ \text{for } x \in \Gamma \subset \Omega \end{aligned} \tag{2.1}$$

where, ϕ is the unknown, f is a given inverse velocity field, g is the value of ϕ at an interface Γ, Ω is an open set in $\mathbb{R}^n$, ∇ denotes the gradient and $|.|$ is the euclidean norm.

The eikonal equation is also fundamental to solving the interface propagation problem. To explain that problem consider Figure 2.3, which depicts an interface that is represented by the black curve that separates a red inside region from a blue outside region. Each point on the interface either moves towards the outside region or the inside region with either the same or different known speed values. The direction of propagation of the interface at each point, either inward or outward, is indicated by the sign of the speed values: negative for inside and positive for outside.

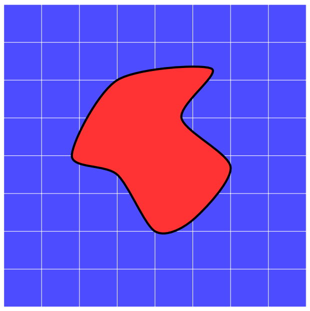

Figure 2.3: 2D shape with inside and outside regions separated by an interface

Such framework of propagating interface can be used to model different real world scenarios. For example, in a phase change problem, the red region can represent ice and the blue region can represent water. Then, the propagation of the boundary enclosing the red region can be determined either by the melting of ice, that shrinks the red region or the freezing of ice, that expands the red region. The goal is to track the interface as it evolves. Since, interfaces can be expanding and contracting at different points, $f(x)$ can be negative at some points and positive at others. The general solution of the eikonal equation is the shortest amount of time (or arrival time) the interface will take to propagate to that point given some known $f(x)$. The degenerate case when $f(x) = 1$ means all points propagate with the same speed (i.e., one grid cell per time interval). As a result, the solution of the eikonal equation for $f(x)$ can be interpreted as the shortest distance from the interface to that point. Hence the special case of the solution to the eikonal equation where $f(x) = 1$ is known as the signed distance function.

Definition The *signed distance* function for any given closed set S of points, is

$$d_S(p) = \min_{x \in S} ||p - x||$$

that is, d_S is the distance to the closest point in S. If S divides space into a well-defined regions, then the signed distance function is

$$\phi_S = \left\{ \begin{array}{r} -d_S(p) : p \text{ is inside,} \\ d_S(p) : p \text{ is outside} \end{array} \right\}$$

Following the derivation in [4], the eikonal equation for calculating the signed distance is written as follows:

$$|\nabla \phi| = 1 \tag{2.2}$$

The following section presents few of the well known algorithms aimed at solving Equation (2.2).

2.2 Solution Algorithms

Some of the well known methods for solving the interface propagation problem are Dijkstra's algorithm [33], Fast Marching Method [30] and Fast Sweeping Method [40]. These methods improve upon the previous work and provide a different perspective on the problem. The following subsections explore these methods in more detail.

2.2.1 Dijkstra's Algorithm

Dijkstra's single source shortest path algorithm [33], based on discrete structures, can also be adapted to solve this problem [9]. It is a graph search algorithm, developed by Edsger Dijkstra in 1959, that finds the shortest path from one node to another. It is ubiquitous; everywhere from network routing to a car's navigation system [30]. In the context of distance field, this is in essence what we want to find; the shortest distance from a source point to an object. This method is a very intuitive way to solve this problem with the following basic steps:

1. Start with a source node and initialize the cost to be 0.
2. Initialize tentative cost at all other nodes to ∞.
3. Mark the start node as accepted and keep track of the list of accepted nodes.
4. Calculate the cost of reaching the neighbor nodes which are one node away.
5. Update the cost if it is less than previous.
6. Mark the node with the least cost as accepted.
7. Repeat until all nodes are accepted.

The runtime complexity of this algorithm is $O(E * N * log(N))$ where N is the total number of nodes and E is the maximum number of edges associate with a single node. Figure 2.4, depicts the shortest path (shown in red) found by the Dijkstra's algorithm and also indicates that "this method cannot converge to the solution of continuous eikonal problem" [30]. It cannot find the diagonal (shown in blue), which is more accurate solution to the shortest distance problem. Therefore, Sethian in

1996 created a numerical method called fast marching method [30], similar to the Dijkstra's algorithm for solving the continuous eikonal problem.

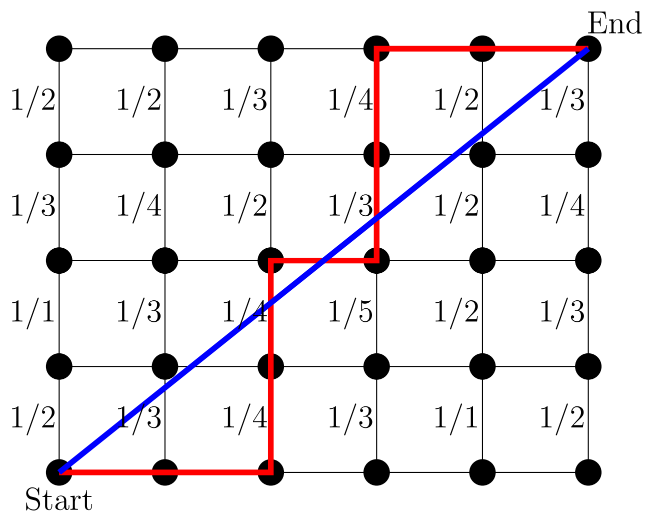

Figure 2.4: Dijkstra's algorithm finding the shortest path (shown in red). Actual shortest distance is the diagonal (shown in blue).

2.2.2 Fast Marching Method (FMM)

FMM is one of the popular techniques for solving the eikonal equation. It is also a one-pass algorithm similar to Dijkstra's but uses upwind difference operators to approximate the gradient [31]. The grid points are updated in order of increasing distance with the aid of a difference formula. Starting with the initial position of the interface, FMM systematically marches outwards one grid point at a time, relying on entropy-satisfying schemes to produce the correct solution [31]. It needs a sorting algorithm that is $O(log(N))$ where N is the number of grid points. Therefore, the FMM has an algorithmic complexity of order $O(Nlog(N))$. The advantage of FMM is that it allows for calculation of narrow bands of data near the surface. The disadvantages of this method are that it is difficult to implement, has an additional $log(N)$ factor added by the sorting algorithm and is challenging to parallelize.

2.2.3 Fast Sweeping Method (FSM)

FSM is another popular iterative algorithm for computing the numerical solution for Equation (2.1) on a rectangular grid in any number of spatial dimensions [40].

It uses nonlinear upwind difference scheme and alternating sweeping orderings of Gauss-Seidel iterations on the whole grid until convergence [40]. The advantage of FSM is that it is straightforward and has linear runtime complexity $O(N)$ for N grid points.

Consider a three-dimensional domain discretized into a grid with NI, NJ, NK nodes in the x-, y-, and z- directions, respectively. Let Γ be a two dimensional interface describing the initial location from which the solution propagates. Then the Godunov upwind differencing scheme on the interior nodes used by the FSM as represented in [29] is:

$$\left(\frac{\phi_{i,j,k} - \phi_{xmin}}{dx}\right)^2 + \left(\frac{\phi_{i,j,k} - \phi_{ymin}}{dy}\right)^2 + \left(\frac{\phi_{i,j,k} - \phi_{zmin}}{dz}\right)^2 = f^2_{i,j,k} \qquad (2.3)$$

for $i \in \{2, \ldots, NI-1\}$, $j \in \{2, \ldots, NJ-1\}$, and $k \in \{2, \ldots, NK-1\}$ where, $\phi_{xmin} = min(\phi_{i-1,j,k}, \phi_{i+1,j,k})$, $\phi_{ymin} = min(\phi_{i,j-1,k}, \phi_{i,j+1,k})$, and $\phi_{zmin} = min(\phi_{i,j,k-1}, \phi_{i,j,k+1})$. In this thesis, the implementations of FSM will only be solved for computing the signed distance field i.e., Equation (2.2). Therefore, the right hand side of Equation (2.3) is set to 1 (i.e., $f = 1$). Equation (2.3) then becomes

$$\left(\frac{\phi_{i,j,k} - \phi_{xmin}}{dx}\right)^2 + \left(\frac{\phi_{i,j,k} - \phi_{ymin}}{dy}\right)^2 + \left(\frac{\phi_{i,j,k} - \phi_{zmin}}{dz}\right)^2 = 1 \qquad (2.4)$$

Gauss-Seidel iterations with alternating sweeping orderings are performed to solve Equation (2.1), until the solution converges for every point. The convergence for all points is guaranteed by the characteristic groups formed by an interface containing initial values. According to Zhao [40], there is a finite amount of interface characteristic groups, so a finite amount of sweeps is required for point solution to converge. The number of iterations to ensure convergence depends on the number of dimensions in a grid. Approximately $2^n \in \mathbb{R}^n$ ($2^2 \in \mathbb{R}^2$ and $2^3 \in \mathbb{R}^3$) sweeps are required for n dimensions. The fast sweeping algorithm consists of two main phases: initialization and sweeping.

1. Initialization: The grid is initialized by first setting the known interface boundary values in the grid. As soon as the interface points are initialized, all the other points in the grid are set to large positive values. During the sweeping phase of the algorithm, interface points do not change, while the rest of the points get updated to new smaller values using characteristic groups of the interface points.
2. Sweeping: The sweeping phase of the algorithm performs several Gauss-Seidel iterations. These iterations do not change interface boundary points that are fixed during the initialization phase. Sweeping iterations update the distance values for points that were not fixed at initialization phase. New distance value for a point is solved by selecting the minimum value between the current and the calculated value. This assures that the solution for a point will remain non-increasing. Alternating sweeping ordering is used to ensure that the information is being propagated along all ($4 \in \mathbb{R}^2$ and $8 \in \mathbb{R}^3$) classes of characteristics. In [40], Zhao showed that this method converges in 2^n sweeps in $\mathbb{R}^n$ and that the convergence is independent of the size of the grid.

   The nodes with the red dots in Figure 2.5, indicate an updated solution during the calculation of the first sweep ordering in two dimensions. The possible sweep directions to guarantee convergence in three dimensions are listed below:

   (a) i = 1:NI, j = 1:NJ, k = 1:NK
   (b) i = NI:1, j = 1:NJ, k = NK:1
   (c) i = 1:NI, j = 1:NJ, k = NK:1
   (d) i = NI:1, j = 1:NJ, k = 1:NK
   (e) i = NI:1, j = NJ:1, k = NK:1
   (f) i = NI:1, j = NJ:1, k = 1:NK
   (g) i = NI:1, j = NJ:1, k = 1:NK
   (h) i = 1:NI, j = NJ:1, k = NK:1

In Algorithm 1, line 1 represents the initialization phase of FSM. The for-loop in

Algorithm 1: Fast Sweeping Method [40] in 3D

```
1  initialize($\phi$);
2  for $iteration = 1$ to $max\_iterations$ do
3      for $ordering = 1$ to $2^3$ do
4          change_axes($ordering$);
5          for $i = 1$ to $NI$ do
6              for $j = 1$ to $NJ$ do
7                  for $k = 1$ to $NK$ do
8                      update($\phi_{i,j,k}$);
9                  end
10             end
11         end
12     end
13 end
```

line 2 increases the accuracy of the solution. The for-loop in line 3 represents the sweeping phase, where line 4 rotates the axis to perform sweeping in a different direction and line 8 solves Equation (2.4). Although there are several methods for solving the eikonal equation, this thesis focuses on the FSM by Zhao [40] because of its straightforward implementation and linear time algorithmic complexity. However, the performance gained from Algorithm 1, may not be sufficient for practical use in application domains like real-time rendering and motion planning or solve large problems that require large memory. Therefore, researchers are always investigating faster methods and designing parallel algorithms to improve performance of existing methods. The following sections discusses the parallel algorithms that improves the performance of FSM.

2.3 Parallel Algorithms for FSM

The goal of parallelization is to increase the efficiency of an algorithm by performing some computations simultaneously. Although the serial implementation is relatively straightforward parallelization of FSM is challenging. The data dependencies of each sweeps, where updating of a node requires the updated values of the previous

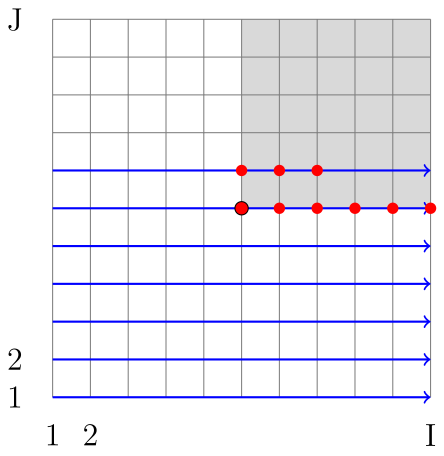

Figure 2.5: The first sweep ordering of FSM for a hypothetical distance function calculation from a source point (center) [7]

nodes, makes it challenging to design an efficient parallel algorithm without causing a decay in the convergence rate. Nonetheless, research efforts produced algorithms for parallelizing FSM.

2.3.1 Parallel Algorithm of Zhao

Zhao presented a parallel implementation of FSM [41] by leveraging the causality of the partial differential equation, i.e., when a grid value reaches the smallest possible value it is the solution and will not be changed in later iterations. This simple causality allowed implementation of different sweeping ordering in parallel, i.e., each sweep is assigned to different threads (or processes) and is computed simultaneously. Each of the threads generate a separate temporary solution which is reduced to a single solution by taking the minimum value from all the temporary solutions. For example, in $\mathbb{R}^2$ there are four sweepings which are assigned to four different threads. After each thread is done with its sweeping the four solutions ϕ_1, ϕ_2, ϕ_3, and ϕ_4 are synchronized using

$$\phi = min(\phi_1, \phi_2, \phi_3, \phi_4)$$

to get the new solution. The diagram of this process is shown in Figure 2 where the direction of the arrow represents the direction of the sweep.

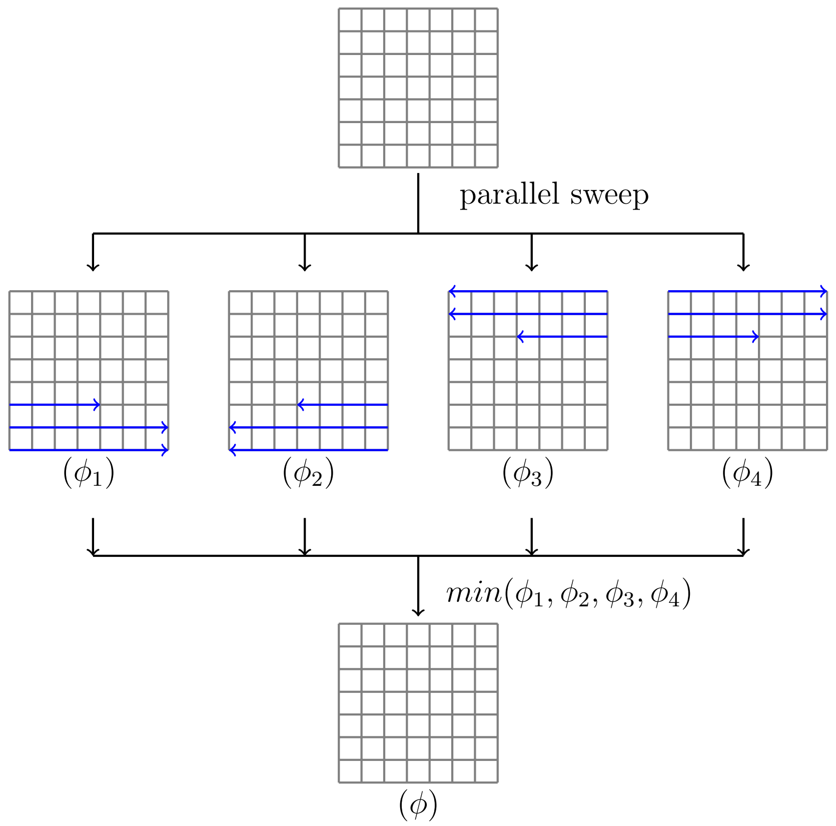

Figure 2.6: Parallel FSM of Zhao in 2D

This method sets a lower upper bound on the degree of parallelization that can be exploited. For example, it instantiates up to four threads in two spatial dimensions and eight threads in three spatial dimensions. The algorithm can be implemented using either OpenMP in a shared memory environment or MPI in a cluster environment. In order to increase the parallelization, Zhao also proposed a domain decomposition approach for parallelizing FSM that could potentially utilize any arbitrary number of threads [41]. However, the performance of the domain decomposition approach plateaus with the increase in the number of threads. This is due to the communication overhead of passing the boundary values back and forth among the threads. Hence, this approach is not discussed in this thesis.

The advantage of the parallel sweeping approach is its simplicity. The drawbacks are 1) more iterations are required for the algorithm to converge, 2) it requires 4 (in 2D) and 8 (in 3D) times more memory resources, 3) it creates synchronization overhead and is not suitable for massively parallel architectures such as GPUs. A pseudo-code of Zhao's parallel algorithm is shown in Algorithm 2.

Algorithm 2: Parallel Fast Sweeping Method of Zhao [41] in 3D

```
1  initialize($\phi$);
2  for $iteration = 1$ to $max\_iterations$ do
3      parallel for $ordering = 1$ to $2^3$ do
           /* change the direction of the sweep ordering based on the
              thread number                                          */
4          change_axes($ordering$);
5          for $i = 1$ to $NI$ do
6              for $j = 1$ to $NJ$ do
7                  for $k = 1$ to $NK$ do
8                      update($\phi_{i,j,k}$);
9                  end
10             end
11         end
12     end
13     $\phi = min(\phi_1, \ldots, \phi_8)$;
14 end
```

In Algorithm 2 the block of code in lines $4 - 11$ is executed in parallel by each of the 2^3 threads. After the distance values are calculated by each thread the solution at each point in the grid is reduced by taking the minimum value calculated for that point which is represented by line 13 in Algorithm 2.

2.3.2 Parallel Algorithm of Detrixhe et al.

The FSM is intrinsically challenging for fine-grain parallelism due to the sequential nature of the Gauss-Seidel iterations that require updated values of previous points to update the current point. Due to this dependency there is a limitation on the points that can be updated simultaneously. Detrixhe et al. [7] presented a parallel approach for FSM that utilizes Cuthill-McKee ordering to distribute the points on the grid to available processors. This approach follows the same procedure as [40] but chooses the direction of sweeps that allows for sets of nodes to be updated simultaneously.

Detrixhe et al. [7] classify each node on the grid to a level based on the sum of the node's coordinates. In 2D, the level of a node (i, j) is defined as $level = i+j$ as shown in Figure 2.7(a) and in 3D, the level of a node (i, j, k) is defined as $level = i+j+k$ as shown in 2.7(b). In $\mathbb{R}^n$, this ordering allows for a $(2n+1)$-point stencil to be updated independently of any other points within a single level.

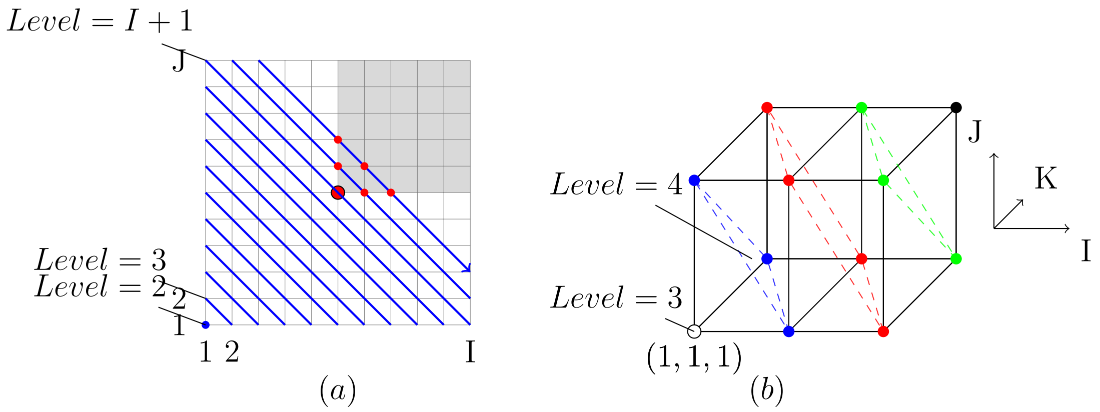

Figure 2.7: Cuthill-McKee ordering of nodes into different levels [7] in two dimensions (left) and three dimensions (right)

Although this approach is not straightforward as Zhao's parallel algorithm [41], it offers significant advantages such as

- The same, as in the serial implementation, number of iterations required for the computation to converge.
- No extra memory resources are required comparing to Algorithm 2.
- No synchronization overhead.
- The level of parallelism is not limited by the number of threads.

A pseudo-code of the algorithm is shown in Algorithm 3. It is an extended version, described for problems in $\mathbb{R}^3$ based on the algorithm presented in [7] that only describes for problems in $\mathbb{R}^2$. This method is not specific to a type of parallel architecture. Detrixhe et al. decided to implement it with OpenMP on shared memory [7]. However, since the performance of this method does not plateau with the increase in number of threads, it is well suited for massively parallel architectures (such as GPUs). These architectures have high number of hardware cores and less overhead of spawning threads than on a CPU architecture.

Algorithm 3: Parallel Fast Sweeping Method from Detrixhe et al. [7]

```
initialize($\phi$);
for $iteration = 1$ to $max\_iterations$ do
    for $ordering = 1$ to $2^3$ do
        change_axes($ordering$);
        for $level = 3$ to $NI + NJ + NK$ do
            $NI_1 = max(1, level - (NJ + NK))$;
            $NI_2 = min(NI, level - 2)$;
            $NJ_1 = max(1, level - (NI + NK))$;
            $NJ_2 = min(NJ, level - 2)$;
            parallel
            /* Each combination of $(i, j)$ generated from the for loops
               below is executed in parallel                          */
            for $i = NI_1$ to $NI_2$ do
                for $j = NJ_1$ to $NJ_2$ do
                    $k = level - (i + j)$;
                    if $k > 0$ and $k <= NK$ then
                        update($\phi_{i,j,k}$);
                    end
                end
            end
        end
    end
end
```

2.3.3 Hybrid Parallel Algorithm

This section investigates a hybrid approach to further improve the performance of FSM algorithm by combining Algorithms 2 and 3. The idea is to use Zhao's approach to distribute the sweep directions to different processes while using Detrixhe et al's approach to perform iterative calculation in each sweep. A pseudo-code representation of the algorithm is shown in Algorithm 4.

Algorithm 4: Hybrid Parallel Fast Sweeping Method using Algorithm 2 and 3

```
1  initialize($\phi$);
2  for $iteration = 1$ to $max\_iterations$ do
3      parallel for $ordering = 1$ to $2^3$ do
           /* change the direction of the sweep ordering based on the
              thread number                                          */
4          change_axes($ordering$);
5          for $level = 3$ to $NI + NJ + NK$ do
6              $NI_1 = max(1, level - (NJ + NK))$;
7              $NI_2 = min(NI, level - 2)$;
8              $NJ_1 = max(1, level - (NI + NK))$;
9              $NJ_2 = min(NJ, level - 2)$;
10             parallel
               /* Each combination of $(i, j)$ generated from the for loops
                  below is executed in parallel                      */
11             for $i = NI_1$ to $NI_2$ do
12                 for $j = NJ_1$ to $NJ_2$ do
13                     $k = level - (i + j)$;
14                     if $k > 0$ and $k <= NK$ then
15                         update($\phi_{i,j,k}$);
16                     end
17                 end
18             end
19         end
20     end
21     $\phi = min(\phi_1, \ldots, \phi_8)$;
22 end
```

Although it might be intuitive to think that this algorithm will perform better than Algorithms 2 and 3 since, the algorithm employs multi-level parallelism (coarse-grain and fine-grain). However, it still retains the same disadvantages that were discussed in Section 2.3.1 that could impact the performance. There is also the added performance overhead associated with communication and reduction of the final solution from the results of different processes. Therefore, it is important to compare the performances between the various implementation of these algorithms. Chapter 4 will present and analyze this data.

To further improve the performance of the FSM solver implemented using these algorithms, the FSM solver can be implemented in various specialized hardware accelerators and architecture models. The following section discusses parallel architectures and different types of hardware accelerators available today.

2.4 Hardware Accelerators

Accelerators refer to specialized computer hardwares that have higher potential to increase performance of certain computational functions compared to a general purpose processor. Accelerators such as GPUs and Intel co-processors are specialized for performing floating point operations. These hardware accelerators allow greater parallelization of tasks due to their high number of cores with the added advantage of reduced overhead of instruction control to improve the execution of certain programs. The Flynn's taxonomy, a classification system proposed by Michael J. Flynn, classifies these architectures as Single Instruction, Multiple Data (SIMD) architecture. This classification system is based upon the number of concurrent instruction streams and data streams available in the architecture and is used as a reference in designing modern processors. There are four classifications in Flynn's taxonomy and they are as follows:

- Single Instruction, Single Data (SISD): This architecture model is found in sequential computer where a single operation is performed at a time, using a single data stream. The traditional uniprocessor machines and older personal computers are examples of SISD architecture.
- Single Instruction, Multiple Data (SIMD): This architecture model is found in modern processors that have multiple cores where a single instruction is carried out by multiple cores but on different data streams. SIMD architectures exploit data parallelism to increase performance of certain programs. GPUs and multi-core processors are examples of SIMD architecture.
- Multiple Instruction, Single Data (MISD): This architecture model is very uncommon and no commercial implementation can be found.

- Multiple Instruction, Multiple Data (MIMD): This architecture model is found in systems with number of autonomous processors that can concurrently execute different instructions on different data. Distributed systems is an example of MIMD architecture.

Although modern CPUs are multi-core and often feature parallel SIMD units; the use of accelerators still yields benefits. The two types of hardware accelerators is discussed in more detail in the following subsections.

2.4.1 Graphics Processing Unit (GPU)

GPU is a single-chip processor especially designed for performing calculations related to 3D computer graphics such as lighting effects, object transformations and 3D motion. These are embarrassingly parallel calculations, that exhibit massive data parallelism, for which GPUs are designed to perform extremely well. The multi-billion dollar gaming industry exerts tremendous economic pressure for the ability to perform massive number of floating point calculations per video frame in advance games and hence, drives the development of the GPUs [15]. As a result, GPUs are designed as a throughput oriented device i.e., it is optimized for the execution throughput of massive number of threads. GPUs require large number of executing threads to hide the long-latency memory accesses. Thus, minimizing the control logic required for each execution thread. [15] “GPUs have small cache size to help control the bandwidth requirements of these applications so multiple threads that access the same memory data do not need to all go to the DRAM. Hence, more chip area is dedicated to the floating-point calculations” [15].

With the modern GPU hardware and the introduction of CUDA in 2006 [20] GPUs are not only used as a graphics rendering engine but also for general purpose computations. As a result, the research community is taking advantage of this performance gain by successfully implementing various computationally demanding complex problems in GPU-aware languages. This effort in general-purpose computing using GPU (known as GPGPU), has positioned the GPU as a compelling alternative to traditional microprocessors in high-performance computer systems of the future [23].

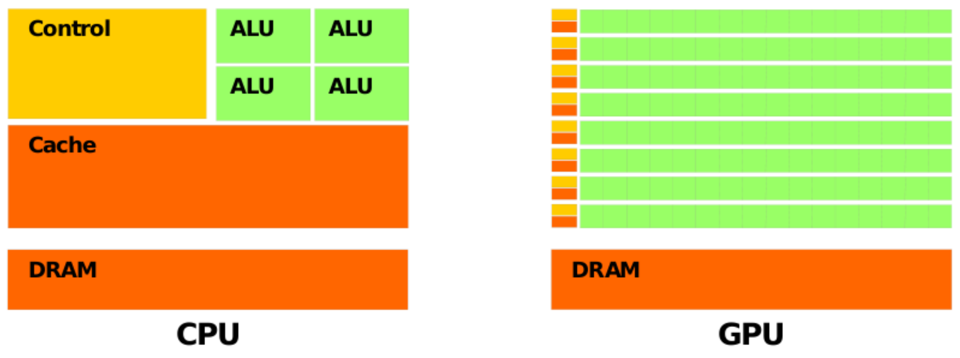

Figure 2.8: CPU and GPU fundamental architecture design. [19]

Figure 2.8 depicts the general design diagrams of CPUs and GPUs. CPUs have few cores with high clock frequencies along with larger areas allocated to cache and a control unit. Such design allows CPUs to perform a single task efficiently. Whereas, GPUs have hundreds of cores with low clock frequencies, small cache area and simple control units. Thus, GPUs are designed to perform several multiple similar tasks simultaneously. Therefore, GPUs do not perform well on tasks where different instructions are carried out sequentially. As a consequence, CPU and GPU are used alongside of each other where the sequential code are executed on the CPU and numerically intensive parallel code on the GPU.

A schematic of the CPU/GPU execution is illustrated in Figure 2.9 [21]. The application is run on the CPU then the computationally extensive part is handed to the GPU. During this time the CPU can choose to wait for the GPU to finish execution or perform other tasks while the GPU is still running. Once the GPU completes the function CPU copies the results and resumes its operations. There are various model for programming GPUs which is discussed in Section 2.5.

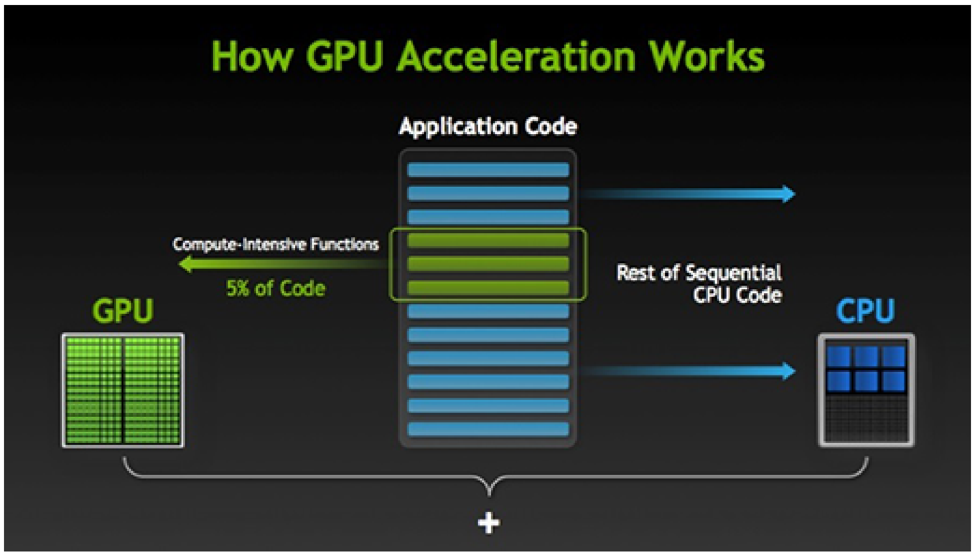
How GPU Acceleration Works
Application Code
Compute-Intensive Functions
5% of Code
Rest of Sequential CPU Code
GPU
CPU
+

Figure 2.9: A schematic of the CPU/GPU execution [21].

2.4.2 Intel Xeon Phi Co-processor

Intel's Xeon Phi co-processor, exemplifies the many integrated core (MIC) architecture and is another example of a hardware accelerator. It contains up to 61 small, low-power processor cores on a single chip to provide a high degree of parallelism and better power efficiency [28]. One advantage Intel's co-processors have over GPUs is that it can operate independently of CPUs and they don't require specialized programming [6]. Another advantage is that the co-processor has "x86-compatible cores with wide vector processing units and uses standard parallel programming models, such as MPI, OpenMP, and hybrid MPI+OpenMP, which makes code development simpler" [28]. Each of the cores support four hardware threads, i.e. each core can concurrently execute instructions from four threads or processes. This keeps the execution units busy by reducing the vector pipeline and memory access latencies [25]. The on chip memory on an Intel Xeon Phi is not shared with the host processor. Thus, copying of data to and from the host and co-processor memories is necessary. This communication between the co-processor and the host processor goes through the PCI Express bus [28] that causes the added latency to data transfers.

The Xeon Phi coprocessor runs a complete micro operating system based on Linux kernel and supports MPI and OpenMP. There are three common execution models for programming and execution of applications for Xeon Phi coprocessors and can be categorized as follows [26]:

- Offload execution mode: In this mode the application starts execution on the host node and the instructions of code segments annotated with offload directives are sent to the coprocessor for execution. This is also known as heterogeneous programming mode.
- Coprocessor native execution mode: Since, Intel Xeon Phi runs a micro Linux operating system the users can view it as a compute node and can execute code directly on the coprocessor.
- Symmetric execution mode: In this mode the application executes on both the host and the coprocessor. Communication between the host and the coprocessor is usually done through message passing interface.

Due to the increasing popularity of accelerators and co-processors, various programming models have been developed by different communities to make programming easier for these devices. Although NVIDIA GPU hardware is used as the chosen accelerator, but the algorithms themselves are general and not specific to GPUs. The following section discusses some of the available frameworks and models for programming accelerators.

2.5 Programming Models

Graphics processors were very difficult to program for general purpose computations due to limited specialized set of APIs that were only designed for graphics applications. That meant programmers had to use OpenGL or Direct3D API to program GPUs. Handful of people could master the skills necessary to use GPUs to increase performance. Initially, GPGPU failed to become a widespread programming phenomenon. However, it was sufficient to inspire hardware manufacturers such as

NVIDIA to add features to facilitate the ease of parallel programming using GPUs [15].

As a result, NVIDIA made changes to its hardware and released CUDA (Compute Unified Device Architecture) in 2007 [15] that made GPGPU implementation easier and faster to learn. Currently there are other platforms like OpenCL, OpenMP and OpenACC that provide different constructs for running code on various accelerators including GPU and Intel Xeon Phi coprocessor. However, none of the above platforms address multi-GPU parallelism across different nodes therefore, multiple GPU implementation must be explicitly performed by the developer [37]. Libraries such as MPI or POSIX can be associated with CUDA and OpenACC in order to benefit from a GPU cluster. The programming models used and discussed in this research are CUDA, OpenACC and MPI.

2.5.1 Compute Unified Device Architecture (CUDA)

CUDA is a parallel computing platform and a library interface developed by NVIDIA to harness massively parallel computing architecture of modern NVIDIA GPUs. The CUDA interface is a proprietary of NVIDIA and can only be used with NVIDIA GPUs. "The CUDA architecture is built around a scalable array of multithreaded streaming multiprocessors (SMs)" [19] each containing a group of execution cores. For example, Tesla K40 has 2880 cores distributed over 15 SMs, each of those SMs have 192 cores [28]. Before using CUDA for GPU programming, it is important to understand some basic CUDA concepts [27]. CUDA-enabled GPUs run in a memory space separate from the host processor. Hence, data must be transferred from the host to the device for performing calculations by using one of the data transfer mechanisms provided by CUDA. Explicit data transfers can be done using API such as `cudaMemcpy()`, whereas implicit transfers are done using pinned or mapped memory. The code that is executed on the CUDA device is written in a subroutine called a *kernel*. "It is important to note that kernels are not functions, as they cannot return a value"[27]. Kernel calls are asynchronous thus, the programmer need to explicitly call synchronization API such as `cudaThreadSynchronize()` to act as a barrier.

A thread is the basic unit of work on the GPU [27]. As illustrated in Figure 2.10, CUDA follows a thread hierarchy arranged in a 1D, 2D or 3D computational grid composed of 1D, 2D or 3D thread blocks. An executional configuration of a CUDA kernel encloses the configuration information, that defines the arrangement of threads, between triple angle brackets `<<< >>>`. "When a kernel is invoked by the host CPU of a CUDA program, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity" [19]. The threads of a thread block execute concurrently on one multiprocessor in a SIMD manner, i.e., all threads execute the same instruction but with different data streams. Furthermore, multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks finish execution, new blocks are launched on the available multiprocessors [19]. In addition to thread blocks, CUDA defines a wrap, which is a collection of 32 threads. At any time an SM can execute a single warp.

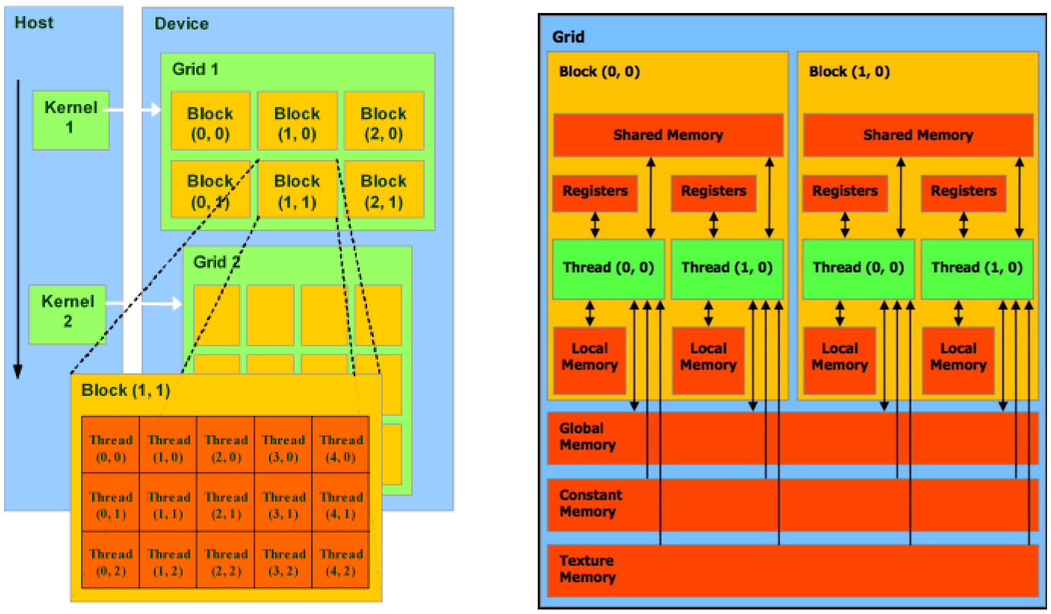

Figure 2.10: CUDA Thread Hierarchy Model (left), CUDA Memory Hierarchy Model (right) [19].

Figure 2.10 depicts the memory hierarchy of CUDA which is similar to a conventional multiprocessor. The fastest and the closest to the core are the local *registers*. The next closest and fastest memory is the *shared* memory. All the threads from a single grid block can access shared memory and can synchronize together. Threads from different grid blocks cannot synchronize. Thus, the exchange of data is only possible through the *global* memory. The shared memory is similar to the L1 cache on a CPU, as it provides very fast access and stores data that is accessed frequently by multiple threads [37]. The only downside is that the shared memory must be maintained by the programmer explicitly. The next memory in this hierarchy is the *global* memory, i.e., the memory space that can be addressed by any thread from any grid block but has a high latency cost. Therefore, coalesced memory access is crucial when accessing global memory as it can hide the latency cost. Global memory accesses can be coalesced in a single transaction if thread blocks are created as a multiple of 16 threads i.e., half warp [28]. Note, that for each GPU architecture there is a maximum number of threads per block that can be run per multiprocessor. Therefore along with the parallelization of the code, optimization of the memory accesses using shared memory and the coalesced access to global memory is also a significant challenge in the development process.

The next available type of memory is called *constant* memory, which is stored on chip. It is used for allocating fixed data, i.e. data that won't change during the execution of a kernel. And the final type of memory in CUDA is the texture memory. The *texture* memory is also stored on chip and optimized for 2D spatial locality. When coalesced read cannot be achieved, it is preferred to use texture memory than global memory [37]. To illustrate a CUDA kernel using CUDA-C, consider the code in Listing 2.1 where the host code invokes a a CUDA kernel that adds two vectors.

```
/* CUDA kernel */
__global__ void add_vector(int n, float *x, float *y)
{
        /* Calculate unique 1D index from 2D thread block */
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        if (i < n) y[i] = x[i] + y[i];
}

int main(void)
{
        ⋮

        /* Allocate memory on GPU */
        cudaMalloc(&d_x, n * sizeof(float));
        cudaMalloc(&d_y, n * sizeof(float));

        /* Copy data from host to GPU */
        cudaMemcpy(d_x, x, n * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_y, y, n * sizeof(float), cudaMemcpyHostToDevice);

        /* Execute the code on the GPU using 2D grid block */
        add_vector<<<(n+255)/256, 256>>>(n, d_x, d_y);

        /* Copy data from GPU to host */
        cudaMemcpy(y, d_y, n * sizeof(float), cudaMemcpyDeviceToHost);

        ⋮
}
```

Listing 2.1: An example of a CUDA kernel using the C programming language that adds two vectors. The text in red represents the API and reserved keywords in CUDA.

2.5.2 Open Accelerators (OpenACC)

OpenACC is a set of compiler directives (pragmas) used for programming accelerators developed by The Portland Group (PGI), Cray, CAPS and NVIDIA [10]. It maintains an open industry standard for compiler directives used for parallel computing similar to OpenMP. OpenACC provides portability across a wide variety of platforms, including GPU, multi-core, and many-core processors [22]. To ensure future portability OpenACC supports multiple levels of parallelism and a memory hierarchy [28]. OpenACC directives are added to code written in C, C++ and Fortran enabling the compiler to generate executables for supported architectures. The table

2.1 lists the supported platforms by various commercial OpenACC compilers (PGI, PathScale and Cray).

OpenACC directives provide an easy and powerful way of leveraging the advantages of accelerator computing while keeping the code compatible with CPU only systems [20]. A typical scenario of a computing system would be the CPU as the host and the GPU as the accelerator. If an accelerator is not present then the compiler ignores the directives and generates machine code that runs on the CPU. The host and the accelerator have separate memory spaces, therefore, OpenACC provides directives that handles the transfer of the data between the host and the accelerator. Thus, abstracting the details of the underlying implementation and significantly simplifying the tasks of parallel programming and code maintenance. The common syntax for writing an OpenACC pragma in C/C++ is

```
#pragma acc directive-name [clause [[,] clause]...]  newline
```

Table 2.1: Platforms supported by OpenACC compilers [22].

Accelerator	PGI	PathScale	Cray
x86 (NVIDIA Tesla)	Yes	Yes	Yes
x86 (AMD Tahiti)	Yes	Yes	No
x86 (AMD Hawaii)	No	Yes	No
x86 multi-core	Yes	Yes	No
ARM multi-core	No	Yes	No

The OpenACC compiler knows that the directives followed by the `acc` keyword belongs to the OpenACC API. It is applied to a block of code contained within curly braces ({. . . }) [28]. Here is a simple example in C, parallelized using OpenACC, that adds two vectors.

```
void add_vector(int n, float *x, float *y)
{
        #pragma acc kernels
        for(int i = 0; i < n; ++i) {
                y[i] = x[i] + y[i];
        }
}
```

Listing 2.2: An example of a loop parallelized using OpenACC. The text in red represents the OpenACC keywords.

Here the `#pragma acc` line indicates that it is an OpenACC compiler directive and simply suggests the compiler to attempt to generate parallel code for the targeted accelerator. Comparing the two code fragments in Listings 2.1 and 2.2, it is clear that OpenACC requires fewer code modifications, is simple to program and easier to understand than CUDA. For reasons of portability, simplicity and code maintenance using OpenACC for acceleration is advantageous than using CUDA.

2.5.3 Message Passing Interface (MPI)

MPI is a standard specification for message passing libraries based on the consensus of the MPI Forum, that consists of vendors, developers and users. The goal of MPI is to provide a portable, efficient and flexible interface standard for writing message passing programs. In message passing programming model, data located on the address space of one process is moved to the address space of another process through cooperative operations [2]. Thus, enabling developers to communicate between different processes to create parallel programs. MPI was originally designed for networks of workstations with distributed memory. However, as architecture changed to shared memory systems, MPI adapted to handle a combination of both types of underlying memory architectures seamlessly [2]. As a result, MPI is widely used and considered as an industry standard for writing message passing applications.

An MPI program is executed by specifying the number of processes, executing their own code, in an MIMD fashion. CUDA and OpenACC models are also able to execute in MIMD manner through streams. The discussion of CUDA streams is beyond the scope of this thesis. Each process in MPI is identified with consecutive integers starting at 0, according to their relative rank in a group. “The processes communicate via calls to MPI communication primitives. Typically, each process executes in its own address space, although shared-memory implementations are also possible” [34]. The basic communication mechanism is the point-to-point communication, where data is transmitted between a sender and a receiver. Then, there is collective communication where data is transmitted to group of processes specified by the user. MPI also supports both blocking and non-blocking communications between processes. Non-blocking communication allows developers to increase performance by overlapping communication and execution between processes. Along with predefined data types, MPI also permits user-defined data types for heterogeneous communication. A simple example of a C code using MPI, where process 0 sends a message to process 1, is presented below:

```
int main(int argc, char *argv[])
{
        char msg[15];
        int myRank;
        int tag = 0;

        MPI_Init(&argc, &argv);

        MPI_Request req;
        MPI_Status status;

        /* find process rank */
        MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

        MPI
        if (myRank == 0) {
                strcpy(msg, "Hello World!");
                MPI_Isend(msg, strlen(msg)+1, MPI_CHAR, 1, tag,
                MPI_COMM_WORLD, &req);
                MPI_Wait(&req, &status);
        } else if (myRank == 1) {
                MPI_Irecv(msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &req);
                MPI_Wait(&req, &status);
        }

        MPI_Finalize();

        return 0;
}
```

Listing 2.3: An example of MPI communication. The text in red represents the MPI keywords and API.

2.6 Parallel FSM Implementations

2.6.1 Common Implementation Details

Most of the code for the distance field solver is identical across all parallel implementations and are not considered as part of the FSM algorithm. There are number of steps carried out before and after the execution of the FSM algorithm, including

1. Open and parse the input file that contains the information about the geometry. The file is saved with a `.vti/.nc` extension.
2. Boundary values are added to each dimension of the domain and set to some default values.
3. The FSM algorithm is executed.
4. The distance values of the points that lie inside of an object are set to `-1`.
5. The default boundary values set in the second step are adjusted by copying the values of the nearest neighbor.

The implementations using various programming models of the parallel algorithms described in Section 2.3 are presented here. The code presented in the implementations, follow the coding guidelines outlined in Appendix A developed as part of this research for adhering to proper software engineering practices.

2.6.2 MPI Implementation of Zhao's Method

The parallel sweeping algorithm for FSM described in Section 2.3.1 Algorithm 2 is implemented here using MPI, with pseudocode shown in Listing 2.4.

```
int index, totalNodes;
// specifies the sweeping directions
int sweeps[8][3] = { { 1, 1, 1 },
                     { 0, 1, 0 },
                     { 0, 1, 1 },
                     { 1, 1, 0 },
                     { 0, 0, 0 },
                     { 1, 0, 1 },
                     { 1, 0, 0 },
                     { 0, 0, 1 } };

// temporary array to store the reduced solution
double * tmp_distance;

if(my_rank == MASTER) {
// if there are more than 1 PEs then the
// temporary array is required to store the
// final distance values after min reduction
  if (npes > 1) {
    tmp_distance = (double *) malloc( totalNodes * sizeof(double) );
  }
}

for (int s = my_rank; s < 8; s += npes) {

  int iStart = (sweeps[s][0]) ? 1 : pf->z;
  int iEnd = (sweeps[s][0]) ? pf->z + 1 : 0;

  int jStart = (sweeps[s][1]) ? 1 : pf->y;
  int jEnd = (sweeps[s][1]) ? pf->y + 1 : 0;

  int kStart = (sweeps[s][2]) ? 1 : pf->x;
  int kEnd = (sweeps[s][2]) ? pf->x + 1 : 0;

  for (int i = iStart; i != iEnd; i = (sweeps[s][0]) ? i + 1 : i - 1) {
    for (int j = jStart; j != jEnd; j = (sweeps[s][1]) ? j + 1 : j - 1) {
      for (int k = kStart; k != kEnd; k = (sweeps[s][2]) ? k + 1 : k - 1) {
        index = i * max_xy + j * max_x + k;
        pf->distance[index] = solveEikonal(pf, index);
      }
    }
  }
}

MPI_Barrier(MPI_COMM_WORLD);
// reduce the solution if there are more than 1 PE
if( npes > 1 ) {
  MPI_Reduce(pf->distance, tmp_distance, totalNodes, MPI_DOUBLE,
             MPI_MIN, MASTER, MPI_COMM_WORLD);

  if( my_rank == MASTER ) {
    free( pf->distance );
    pf->distance = tmp_distance;
  }
}
```

Listing 2.4: Code fragment that implements the sweeping stage of the parallel FSM algorithm by Zhao [41], Algorithm 2, using MPI.

2.6.3 CUDA Implementation of Detrixhe et al.'s Method

The parallel algorithm for FSM described in Section 2.3.2 Algorithm 3 is implemented here using CUDA, with pseudocode shown in Listing 2.5. The CUDA kernel in this implementation is executed for each level of the Cuthill McKee ordering for all sweeping directions. Since, each level has different number of mesh points, the kernel is executed with different configuration of two dimensional thread block and grid arrangements. However, when the number of mesh points is equal to or greater than 256, the kernel is executed with (16×16) thread block and two dimension grid size that encompasses the total number of mesh points.

```
for (int swCount = 1; swCount <= 8; ++swCount) {
  int start = (swCount == 2 || swCount == 5 ||
               swCount == 7 || swCount == 8) ? totalLevels : meshDim;
  int end = (start == meshDim) ? totalLevels + 1 : meshDim - 1;
  int incr = (start == meshDim) ? true : false;

  // sweep offset is used for translating the 3D coordinates
  // to perform sweeps from different directions
  sw.xSweepOff = (swCount == 4 || swCount == 8) ? sw.xDim + 1 : 0;
  sw.ySweepOff = (swCount == 2 || swCount == 6) ? sw.yDim + 1 : 0;
  sw.zSweepOff = (swCount == 3 || swCount == 7) ? sw.zDim + 1 : 0;
  for (int level = start;
           level != end;
           level = (incr) ? level + 1 : level - 1) {
    int xs = max(1, level - (sw.yDim + sw.zDim));
    int ys = max(1, level - (sw.xDim + sw.zDim));
    int xe = min(sw.xDim, level - (meshDim - 1));
    int ye = min(sw.yDim, level - (meshDim - 1));

    int xr = xe - xs + 1, yr = ye - ys + 1;
    int tth = xr * yr; // Total number of threads needed

    dim3 bs(16, 16, 1);
    if (tth < 256) {
      bs.x = xr;
      bs.y = yr;
    }

    dim3 gs(iDivUp(xr, bs.x), iDivUp(yr, bs.y), 1);

    sw.level = level;
    sw.xOffSet = xs;
    sw.yOffset = ys;

    fast_sweep_kernel <<<gs, bs>>> (dPitchPtr, sw);
    cudaThreadSynchronize();
  }
}
```

Listing 2.5: Host code fragment that implements the sweeping stage of the parallel FSM algorithm by Detrixhe et al. [7], Algorithm 3, and calls the CUDA kernel that executes parallel operations on the GPU.

```
__global__ void fast_sweep_kernel(cudaPitchedPtr dPitchPtr, SweepInfo s) {
  int x = (blockIdx.x * blockDim.x + threadIdx.x) + s.xOffSet;
  int y = (blockIdx.y * blockDim.y + threadIdx.y) + s.yOffset;

  if (x <= s.xDim && y <= s.yDim) {     int z = s.level - (x + y);
    if (z > 0 && z <= s.zDim) {
      int i = abs(z - s.zSweepOff);
      int j = abs(y - s.ySweepOff);
      int k = abs(x - s.xSweepOff);

      char *devPtr = (char *)dPitchPtr.ptr;
      size_t pitch = dPitchPtr.pitch;
      size_t slicePitch = pitch * (s.yDim + 2);
      double *c_row = (double *)((devPtr + i * slicePitch) + j * pitch);
      double center = c_row[k];
      double left   = c_row[k-1];
      double right  = c_row[k+1];
      double up     = ((double *)
                      ((devPtr + i * slicePitch) + (j-1) * pitch))[k];
      double down   = ((double *)
                      ((devPtr + i * slicePitch) + (j+1) * pitch))[k];
      double front  = ((double *)
                      ((devPtr + (i-1) * slicePitch) + j * pitch))[k];
      double back   = ((double *)
                      ((devPtr + (i+1) * slicePitch) +j * pitch))[k];

      double minX = min(left, right);
      double minY = min(up, down);
      double minZ = min(front, back);
      c_row[k] = solve_eikonal(center, minX, minY, minZ, s.dx, s.dy, s.dz);
    }
  }
}
```

Listing 2.6: CUDA kernel that maps each thread to a grid point and calls the function that solves Equation (2.2) **on that point.**

CUDA provides various APIs for allocating memory on the GPU. The most commonly used API is `cudaMalloc` that allocates contiguous chunk of memory of the specified size. Allocating memory using `cudaMalloc` for multi-dimensional arrays can create a bottleneck during memory transactions due to unaligned data elements and uncoalesced memory access. To address this issue and lower the memory access latency the data elements must be properly aligned and must ensure coalesced memory access pattern. Therefore, a better API to allocate memory for a multi-dimensional array is `cudaMalloc3D`. It allocates linear memory that may be padded to ensure that hardware alignment requirements are met. It leads to fewer memory transactions during non-sequential access, thus reducing memory access time. However, using `cudaMalloc3D` is not straightforward as `cudaMalloc` as shown in Listing 2.7. Although it requires additional lines of code and a different API for transferring data between the host and the device than `cudaMalloc`, it dramatically increased the performance of the code by greater than 100× compared to the `cudaMalloc` implementation.

```
cudaPitchedPtr hostPtr, devicePtr;
hostPtr = make_cudaPitchedPtr(pf->distance,
                              max_x * sizeof(double),
                              max_x, max_y);

cudaExtent dExt = make_cudaExtent(max_x * sizeof(double), max_y, max_z);

// allocate memory on the device using cudaMalloc3D
cudaMalloc3D(&devicePtr, dExt);

// copy the host memory to device memory
cudaMemcpy3DParms mcp =  0 ;
mcp.kind = cudaMemcpyHostToDevice;
mcp.extent = dExt;
mcp.srcPtr = hostPtr;
mcp.dstPtr = devicePtr;

cudaMemcpy3D(&mcp);
```

Listing 2.7: CUDA-C code fragment that copies a three dimensional array from the host to the GPU.

2.6.4 OpenACC Implementation of Detrixhe et al.'s Method

The pseudocode for the parallel algorithm for FSM described in Section 2.3.2, Algorithm 3, implemented using OpenACC is shown in Listing 2.8.

```
int start, end, incr;

for (int sn = 1; sn <= 8; ++sn) {
  SweepInfo s = make_sweepInfo(pf, sn);
  start = (sn == 2 || sn == 5 || sn == 7 || sn == 8) ?
          s.lastLevel : s.firstLevel;

  if (start == s.firstLevel) {
    end = s.lastLevel + 1;
    incr = 1;
  } else {
    end = s.firstLevel - 1;
    incr = 0;
  }
  for (int level = start;
           level != end;
           level = (incr) ? level + 1 : level - 1) {
    // s - start, e - end
    int xs, xe, ys, ye;

    xs = max(1, level - (s.yDim + s.zDim));
    ys = max(1, level - (s.xDim + s.zDim));
    xe = min(s.xDim, level - (s.firstLevel - 1));
    ye = min(s.yDim, level - (s.firstLevel - 1));

    int i, j, k, index;
    int xSO = s.xSweepOff;
    int ySO = s.ySweepOff;
    int zSO = s.zSweepOff;

    #pragma acc kernels {
      #pragma acc loop independent
      for (int x = xs; x <= xe; x++) {
        #pragma acc loop independent
        for (int y = ys; y <= ye; y++) {
          int z = level - (x + y);
          if (z > 0 && z <= pf->z) {
            i = abs(z - zSO);
            j = abs(y - ySO);
            k = abs(x - xSO);
            index = i * max_xy + j * max_x + k;
            pf->distance[index] = solveEikonal(pf, index);
          }
        }
      } // end of acc kernels
    }
  }
}
```

Listing 2.8: Code fragment that implements the sweeping stage of the parallel FSM algorithm by Detrixhe et al. [7], Algorithm 3, using OpenACC.

2.6.5 MPI/CUDA Implementation of Hybrid Parallel FSM

The pseudocode for the hybrid parallel algorithm for FSM described in Section 2.3.3, Algorithm 4, implemented using combination of MPI and CUDA, is shown in Listing 2.9. Further comments are provided in the code that describes the function of each code block.

```
int my_rank, npes;
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &npes);

// temporary array to store the reduced solution
double *tmp_distance;
int totalNodes = max_x * max_y * max_z;

if (my_rank == MASTER) {
  // if there are more than 1 PEs then the temporary array
  // is required to store the final distance values after min reduction
  if (npes > 1) {
    tmp_distance = (double *) malloc(totalNodes * sizeof(double));
  }
}

cudaPitchedPtr hostPtr, devicePtr;
hostPtr = make_cudaPitchedPtr(pf->distance,
                              max_x * sizeof(double),
                              max_x, max_y);
cudaExtent dExt = make_cudaExtent(max_x * sizeof(double), max_y, max_z);

cudaMalloc3D(&devicePtr, dExt);

_cudaMemcpy3D(hostPtr, devicePtr, dExt, cudaMemcpyHostToDevice);

// Each rank does a different sweep
for (int swCount = my_rank+1; swCount <= 8; swCount+=npes) {
  int start = (swCount == 2 || swCount == 5 ||
               swCount == 7 || swCount == 8) ? totalLevels : meshDim;
  int end = (start == meshDim) ? totalLevels + 1 : meshDim - 1;
  int incr = (start == meshDim) ? true : false;

  // sweep offset is used for translating the 3D coordinates
  // to perform sweeps from different directions
  sw.xSweepOff = (swCount == 4 || swCount == 8) ? sw.xDim + 1 : 0;
  sw.ySweepOff = (swCount == 2 || swCount == 6) ? sw.yDim + 1 : 0;
  sw.zSweepOff = (swCount == 3 || swCount == 7) ? sw.zDim + 1 : 0;

  for (int level = start;
           level != end;
           level = (incr) ? level + 1 : level - 1) {
    int xs = max(1, level - (sw.yDim + sw.zDim));
    int ys = max(1, level - (sw.xDim + sw.zDim));
    int xe = min(sw.xDim, level - (meshDim - 1));
    int ye = min(sw.yDim, level - (meshDim - 1));

    int xr = xe - xs + 1, yr = ye - ys + 1;
    int tth = xr * yr; // Total number of threads needed

    dim3 bs(16, 16, 1);
    if (tth < 256) {
      bs.x = xr;
      bs.y = yr;
    }

    dim3 gs(iDivUp(xr, bs.x), iDivUp(yr, bs.y), 1);

    sw.level = level;
    sw.xOffSet = xs;
    sw.yOffset = ys;

    fast_sweep_kernel <<<gs, bs>>> (dPitchPtr, sw);
    cudaThreadSynchronize();
  }
}

_cudaMemcpy3D(devicePtr, hostPtr, dExt, cudaMemcpyDeviceToHost);

MPI_Barrier(MPI_COMM_WORLD);

// The solution is reduced by taking the minimum from all
// differenent sweeps done by each process
MPI_Reduce(pf->distance, tmp_distance, totalNodes, MPI_DOUBLE,
           MPI_MIN, MASTER, MPI_COMM_WORLD);

if (my_rank == MASTER) {
        free(pf->distance);
        pf->distance = tmp_distance;
}
```

Listing 2.9: Code fragment that implements the sweeping stage of the hybrid parallel FSM algorithm using MPI and CUDA.

2.6.6 MPI/OpenACC Implementation of Hybrid Parallel FSM

The pseudocode for the hybrid parallel algorithm for FSM described in Section 2.3.3, Algorithm 4, implemented using combination of MPI and OpenACC, is shown in Listing 2.10. Further comments are provided in the code that describes the function of each code block.

```
int my_rank, npes;
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &npes);

// temporary array to store the reduced solution
double *tmp_distance;
int totalNodes = max_x * max_y * max_z;

if (my_rank == MASTER) {
  // if there are more than 1 PEs then the temporary array
  // is required to store the final distance values after min reduction
  if (npes > 1) {
    tmp_distance = (double *) malloc(totalNodes * sizeof(double));
  }
}

int start, end, incr, sn;

for (sn = 1; sn <= 8; ++sn) {
  SweepInfo s = make_sweepInfo(pf, sn);
  start = (sn == 2 || sn == 5 || sn == 7 || sn == 8) ?
          s.lastLevel : s.firstLevel;

  if (start == s.firstLevel) {
    end = s.lastLevel + 1;
    incr = 1;
  } else {
    end = s.firstLevel - 1;
    incr = 0;
  }

  for (int level = start;
           level != end;
           level = (incr) ? level + 1 : level - 1) {
    // s - start, e - end
    int xs, xe, ys, ye;

    xs = max(1, level - (s.yDim + s.zDim));
    ys = max(1, level - (s.xDim + s.zDim));
    xe = min(s.xDim, level - (s.firstLevel - 1));
    ye = min(s.yDim, level - (s.firstLevel - 1));

    int x, y, z, i, j, k, index;
    int xSO = s.xSweepOff;
    int ySO = s.ySweepOff;
    int zSO = s.zSweepOff;

    #pragma acc kernels {
      #pragma acc loop independent
      for (x = xs; x <= xe; x++) {
        #pragma acc loop independent
        for (y = ys; y <= ye; y++) {
          z = level - (x + y);
         if (z > 0 && z <= pf->z) {
           i = abs(z - zSO);
           j = abs(y - ySO);
           k = abs(x - xSO);
           index = i * max_xy + j * max_x + k;
           pf->distance[index] = solveEikonal(pf, index);
         }
       }
     }
   } // end of acc kernels
}

MPI_Barrier(MPI_COMM_WORLD);

// The solution is reduced by taking the minimum from all
// differenent sweeps done by each process
MPI_Reduce(pf->distance, tmp_distance, totalNodes, MPI_DOUBLE,
MPI_MIN, MASTER, MPI_COMM_WORLD);

if (my_rank == MASTER) {
  free(pf->distance);
  pf->distance = tmp_distance;
}
```

Listing 2.10: Code fragment that implements the sweeping stage of the hybrid parallel FSM algorithm using MPI and OpenACC.

CHAPTER 3

PARALLEL FSM ALGORITHM FOR DISTRIBUTED PLATFORMS WITH ACCELERATORS

The chief objective of this thesis is to design a scalable parallel FSM algorithm for distributed computing environment with accelerators. The current state of the art parallel approach by Detrixhe et al. is not applicable to problems that exceeds the available memory on the system. Therefore in order to solve a large problem, the memory must be upgraded to fit the problem size; which is not a viable solution. To address this issue a scalable domain decomposition strategy is required. In the following section an algorithm is presented based on Cuthill McKee ordering presented in Algorithm 3 [7], that uses coarse grain parallelism using MPI for distributing the subdomains (or blocks) across multiple nodes and fine grain parallelism to optimize performance by utilizing accelerators.

3.1 Method

The methods of solving the eikonal equation discussed so far are applicable for problems that require at most memory space of a system. However, these methods do not extend to problems with large domain sizes that exceed memory resources. In such cases, decomposition strategy is required to partition the domain into smaller subdomains and execute them separately. Zhao [41] also proposed a domain decomposition method for a parallel implementation where the domain is split into rectangular blocks. This decomposition allows simultaneous computation of FSM on each block. Although this method could potentially solve large problems in a distributed environment, Zhao's approach limits the distance that the values can propagate in a single iteration, thereby increasing the number of iterations required for the algorithm to converge [7].

In [7], Detrixhe et al. utilized Cuthill McKee ordering that allowed sets of nodes to be updated simultaneously without any domain decomposition. This approach resulted in a better performance and converged at the same rate as the serial implementation. Similar approach can be applied on a coarser level as a domain decomposition strategy for a distributed environment.

3.1.1 Domain Decomposition

To simplify the illustrations the figures are shown in two dimensions on rectangular grids whereas the algorithms are explained in three dimensions. A three dimensional Cartesian volume is decomposed based on the size of the domain in the x-, y- and z- direction forming smaller blocks. This is shown in two dimensions in Figure 3.1, where the domain is decomposed into three blocks in the x-direction and two blocks in the y-direction. The blocks are outlined in blue lines.

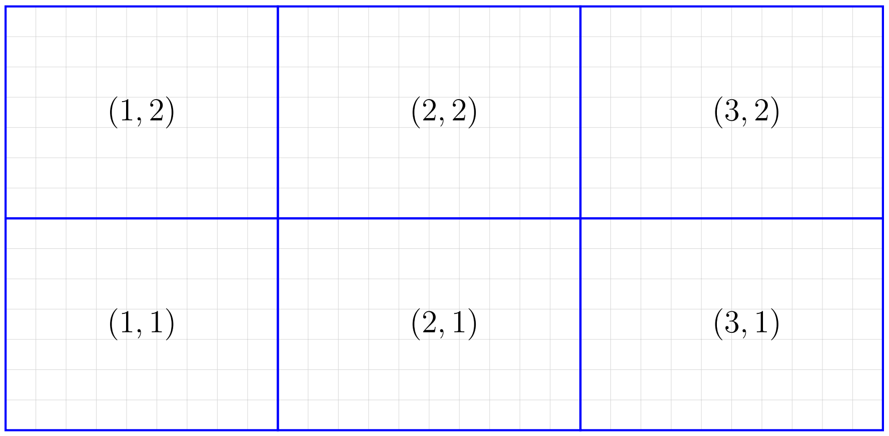

Figure 3.1: A 3×2 decomposition of the domain. Each block is labeled by a two dimensional coordinate system starting at index 1.

The blocks are labeled using a coordinate system that starts at 1 as shown in Figure 3.1. Each block is assigned to an MPI process running on a distributed system. The mapping of a block to an MPI process is calculated by converting the coordinate into a linear 0-based index. The formula to convert a three dimensional 1-based coordinate into a one dimensional 0-based index, is given in Equation 3.1.

$$process = (x-1) + width * (y-1) + width * height * (z-1) \tag{3.1}$$

where x, y, z represents the coordinate of the block, width is the number of blocks in the x-direction and height is the number of blocks in the y-direction.

Each process loads its assigned block into memory through the parallel I/O file format discussed in Section 3.2. Similar to [41], there could be different implementations for this domain decomposition algorithm. Computations in each block can be done simultaneously or sequentially. Simultaneous computation degrades the convergence speed while sequential computation degrades performance. Hence, to improve efficiency through parallelization without degrading the convergence speed, the same approach as [7] can be applied on a coarser grid. The blocks of the coarser grid are ordered using the Cuthill McKee ordering (as illustrated in two dimensions in Figure 3.2) where simultaneous computations are performed on blocks of the same level.

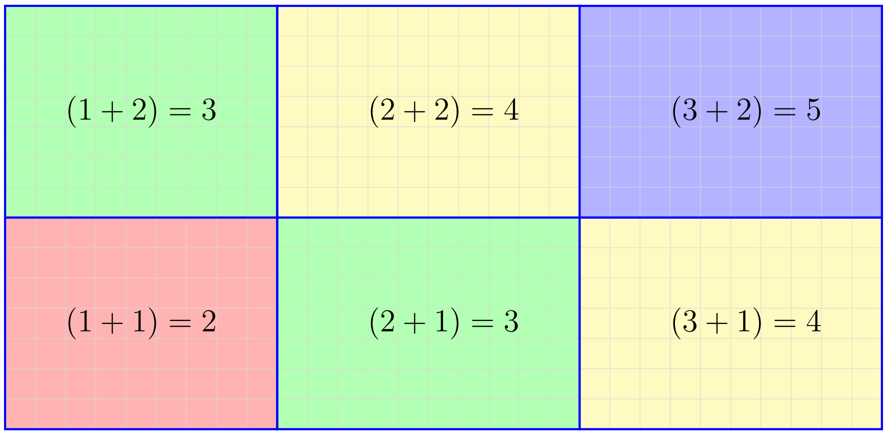

Figure 3.2: Cuthill McKee ordering of the decomposed blocks. The sum of the coordinates and the colors represent the levels of the Cuthill McKee ordering.

In this approach first ghost cells are added to each block as shown in Figure

3.3. Then the block with the lowest level performs the first sweep iteration of FSM while the other blocks are blocked as illustrated in Figure 3.4. The execution of the sweeps is implemented using the Cuthill McKee ordering as described in Section 2.3.2. After completion of the sweep the updated values along the shared boundaries are transferred to the blocks on the next level as illustrated in Figure 3.5. This process is repeated until the block with the highest level completes the sweep iteration. Similarly the axes are configured to repeat the same process for all sweep directions listed in Section 2.2.3. The algorithm and its implementation is explained in detail in the following section.

3.1.2 Algorithm and Implementation

In $\mathbb{R}^3$, a pseudocode for the distributed parallel implementation is given in Algorithm 5 where the domain is split into 3D blocks and the code is executed with the same number of MPI processes as the number of blocks. Launching the code in such manner is an implementation decision for better performance as it reduces parallel I/O operations (each process does one read and one write access). On the other hand, if fewer processes were launched, each process moving to unexecuted block would have to write its previous block of data into file and read the new block of data into memory each time for every iteration. This would heavily impact the performance of the algorithm.

The variable *blockDim* on line 1 of Algorithm 5 stores the dimensions of the decomposed (coarser) grid. A three dimensional index is calculated by each process using its rank and the size of the block dimensions that associates each MPI process with a block. For example, if the domain is divided into blocks by 2 in the x- and y-direction and 1 in the z- direction then there would be four blocks and four processes. The assignment of an MPI process to a block is shown in Listing 3.1

After calculating the block coordinates, each MPI process calculates the position and the size of the block relative to the entire domain. The calculations involved in

Algorithm 5: Parallel FSM for Distributed Computing Platforms with Accelerators

```
/* launch the program with same number of MPI processes as the
   total number of blocks                                          */
int $blockDim[3]$ ;                 /* Dimensions of the coarser grid */
int $mpiRank$ ;                             /* Rank of the MPI process */
/* Block coordinate assigned to each MPI process                   */
int $blockIdX = mpiRank \% blockDim[0] + 1$;
int $blockIdY = (mpiRank / blockDim[0]) \% blockDim[1] + 1$;
int $blockIdZ = mpiRank / (blockDim[0] * blockDim[1]) + 1$;
Calculate block size and offset for each MPI process
All MPI processes allocate memory including ghost layers
All MPI processes load their assigned block into their memory
/* Start FSM and propagation                                       */
initialize($\phi$);
for $iteration = 1$ to $max\_iterations$ do
    for $ordering = 1$ to $2^3$ do
        change_axes($ordering$);
        if $blockIdX == 1$ and $blockIdY == 1$ and $blockIdZ == 1$ then
            fsm($\phi$) ;                                /* Algorithm 3 */
        end
        else
            if $blockIdX - 1 > 0$ then
                receive YZ plane from west block;
            end
            if $blockIdY - 1 > 0$ then
                receive XZ plane from south block;
            end
            if $blockIdZ - 1 > 0$ then
                receive XY plane from bottom block;
            end
            fsm($\phi$) ;                                /* Algorithm 3 */
        end
        if $blockIdX + 1 <= blockDim[0]$ then
            send YZ plane to east block;
        end
        if $blockIdY + 1 <= blockDim[1]$ then
            send XZ plane to north block;
        end
        if $blockIdZ + 1 <= blockDim[2]$ then
            send XY plane to top block;
        end
    end
end
```

this process are shown in Listing 3.2 which is represented by line 6 of Algorithm 5.

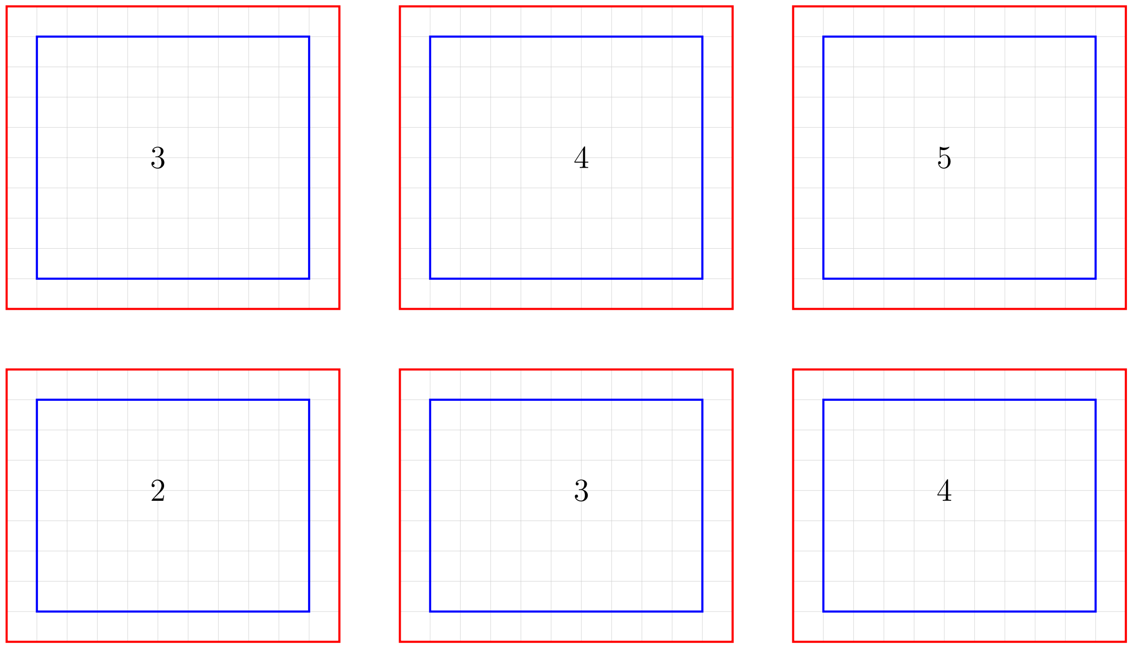

Figure 3.3: Addition of the ghost cell layers on each block. The separation of the blocks represents that each block is managed by a different MPI process that might be running on a separate node with enough memory resources.

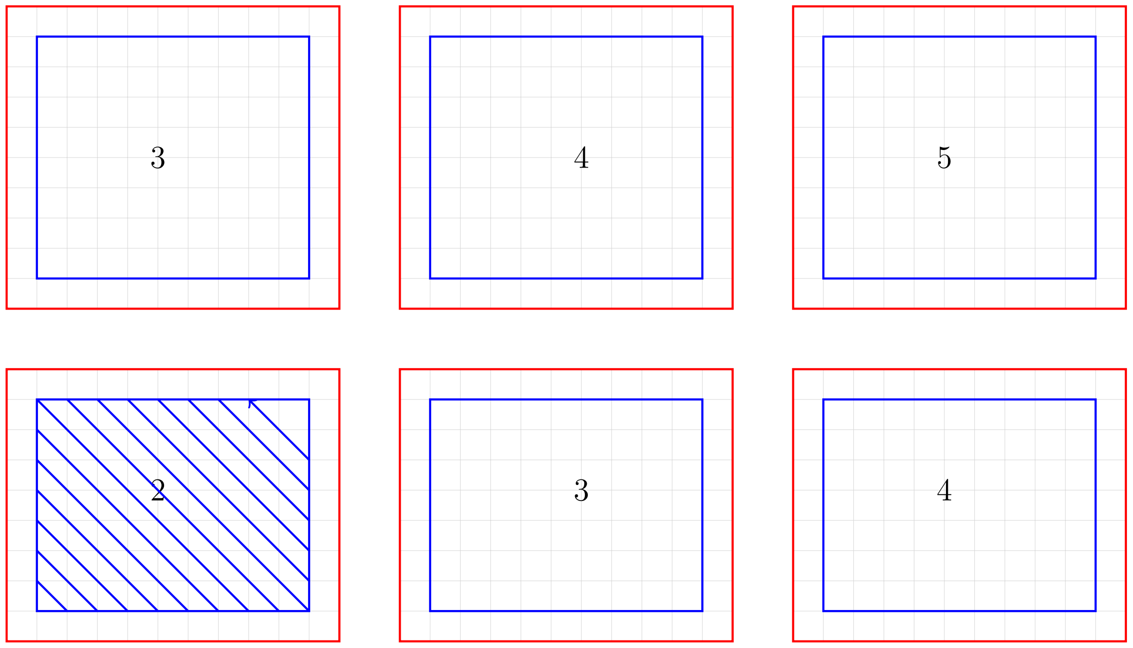

Figure 3.4: The first sweep iteration of FSM by the block in level two of the Cuthill McKee ordering.

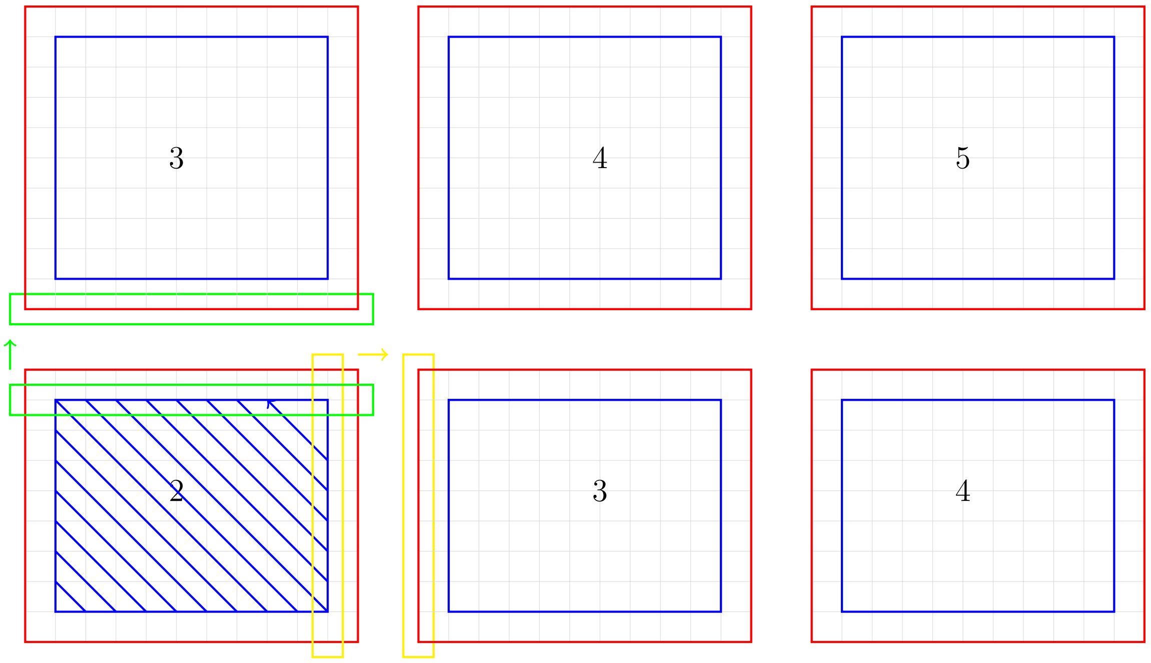

Figure 3.5: Exchange of ghost cell values from level two block on to level three blocks.

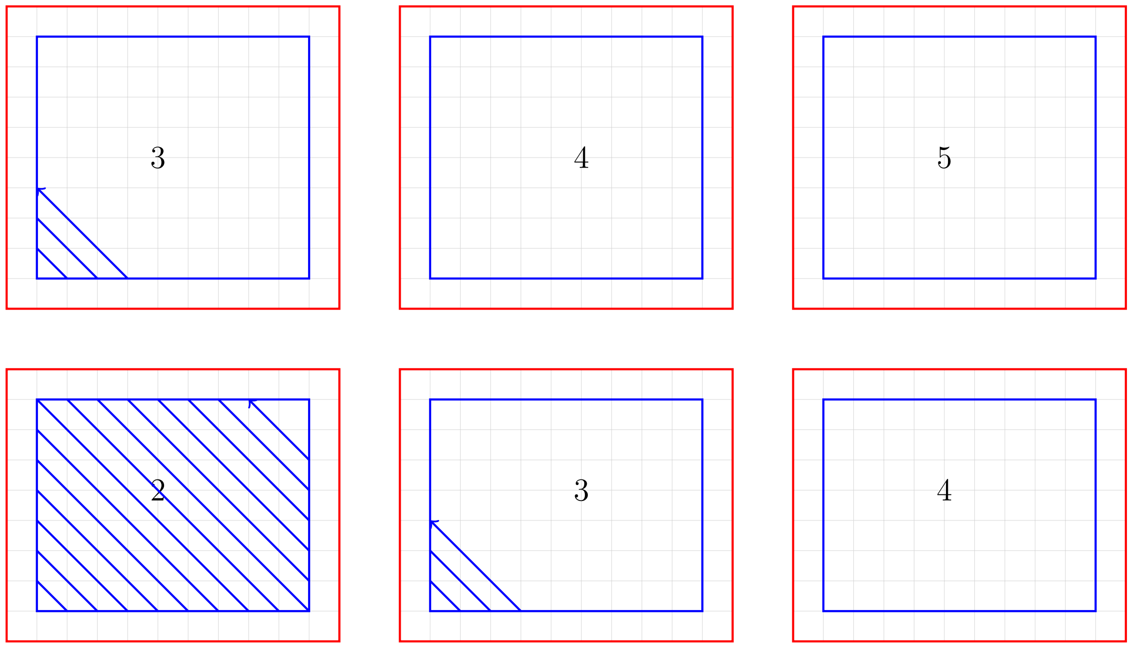

Figure 3.6: Continuation of the first sweep iteration of FSM on to the next next level. The two blocks on level three are executed simultaneously.

```
blockDim[0] = 2; blockDim[1] = 2; blockDim[2] = 1;
```

```
/* MPI Rank 0 */
blockIdX = 0 % 2 + 1       = 1;
blockIdY = (0 / 2) % 2 + 1 = 1;
blockIdZ = 0 / (2 * 2) + 1 = 1;

/* MPI Rank 1 */
blockIdX = 1 % 2 + 1       = 2;
blockIdY = (1 / 2) % 2 + 1 = 1;
blockIdZ = 1 / (2 * 2) + 1 = 1;

/* MPI Rank 2 */
blockIdX = 2 % 2 + 1       = 1;
blockIdY = (2 / 2) % 2 + 1 = 2;
blockIdZ = 2 / (2 * 2) + 1 = 1;

/* MPI Rank 3 */
blockIdX = 3 % 2 + 1       = 2;
blockIdY = (3 / 2) % 2 + 1 = 2;
blockIdZ = 3 / (2 * 2) + 1 = 1;
```

Listing 3.1: An example of MPI process to block mapping calculation.

```
int dim[3];      /* Dimensions of the original grid */
int blockDim[3]; /* Dimensions of the coarser grid */
int blockIdX, blockIdY, blockIdZ /* Block coordinate */

/* Calculating the size of the block */
int BOX_X = (blockDim[0] == 1) ? dim[0] : (dim[0] / blockDim[0]) + 1;
int BOX_Y = (blockDim[1] == 1) ? dim[1] : (dim[1] / blockDim[1]) + 1;
int BOX_Z = (blockDim[2] == 1) ? dim[2] : (dim[2] / blockDim[2]) + 1;

int start[3], count[3];
start[2] = (x - 1) * BOX_X;
start[1] = (y - 1) * BOX_Y;
start[0] = (z - 1) * BOX_Z;

count[2] = min(dim[0] - 1, start[2] + BOX_X) - start[2];
count[1] = min(dim[1] - 1, start[1] + BOX_Y) - start[1];
count[0] = min(dim[2] - 1, start[0] + BOX_Z) - start[0];

/* Including the ghost cells */
start[2] = (start[2] == 0) ? 0 : start[2] - 1;
start[1] = (start[1] == 0) ? 0 : start[1] - 1;
start[0] = (start[0] == 0) ? 0 : start[0] - 1;

count[2] = (start[2] == 0) ? count[2] + 1 : count[2] + 2;
```

```
count[1] = (start[1] == 0) ? count[1] + 1 : count[1] + 2;
count[0] = (start[0] == 0) ? count[0] + 1 : count[0] + 2;
```

Listing 3.2: Code segment that calculates the position and size of the block assigned to each MPI process.

Each MPI process then allocates memory for their block and loads the information from the file in to their memory. The allocated memory is referred as the symbol ϕ. Similar to Algorithm 3, ϕ is initialized and the iterations and ordering loops are executed as shown in lines 10 and 11 of Algorithm 5. For each sweeping ordering appropriate rotation of the axis is performed and then the block at coordinate (1,1,1) executes using the parallel FSM algorithm (i.e. Algorithm 3). All other blocks are blocked and waiting to receive the updated values on their ghost cells. Once the block that executed FSM completes, it sends the appropriate ghost layer to the neighboring blocks. The sending and receiving of the ghost layers is done by checking whether a block exists on the sending side (east, north, top) or the receiving side (west, south, bottom) respectively as shown in code block from lines 17-25 and 28-36 of Algorithm 5.

Ability to perform parallel I/O operations on a single file is what makes it feasible to solve this problem. Each process needs to load its assigned dataset from the same file, process the data and write to file simultaneously without having to wait on other processes. However, it is a daunting task to perform parallel I/O operations on a regular text file. Therefore, this thesis uses netCDF [38] as its input file format. The netCDF version four supports parallel I/O features using HDF5 as the underlying file format. It provides a rich set of interfaces for parallel I/O access to organize and store data efficiently. One of the advantages of using netCDF is its ability to query the dataset for specific locations without having to load the entire file into the memory. More detail about netCDF is provided in the following section.

3.2 Network Common Data Form (NetCDF)

NetCDF defines a self-describing, portable, array-oriented file format with a simple interface for creating, storing, accessing and sharing of scientific data. It is called self-describing because its dataset includes information that defines the data it contains. The data represented by netCDF is machine independent, i.e. data can be accessed on any machine even if integers, characters and floating point numbers are stored differently on that architecture [38]. Machine independence is achieved by representing the data in a well-defined format similar to XDR (eXternal Data Representation) but with extended support for efficiently storing arrays of non-byte data [13]. The data in netCDF is accessed through an interface that provides a library of functions for storing and retrieving data in the form of arrays. An array can represent any number of dimensions and contain items which all have the same data type. A single value in netCDF is represented as a 0-dimensional array. The netCDF interface provides an abstraction that allows direct access to the values of the dataset, without knowing the details of how the data are stored. Hence, the implementation of how the data is stored can be changed without affecting existing programs. The Unidata, community that develops and maintains netCDF, supports netCDF interfaces for C, C++, Fortran 90, Fortran 95, Fortran 2003 and Fortran 2008. The details about the file format model of a netCDF file is outlined in the following subsection.

3.2.1 File Format

NetCDF stores data in an array-oriented dataset, which contains dimensions, variables and attributes. The dataset is physically categorized into two parts: file header and array data. The "file header first defines a number of dimensions, each with a name and a length". The dimensions of a netCDF file defines the shapes of variables in the dataset [13]. Following the dimensions each variable of the dataset is described in the file header by its name, attributes, data type, array size, and data offset.

The data part of a netCDF file stores the values for each variable in an array, one variable after another, in their defined order [13]. The latest version of NetCDF known as NetCDF-4 offers features including compound types and parallel I/O access. The parallel read/write access to netCDF files is supported using the underlying Hierarchical Data Format 5 (HDF5) library. HDF5 provides data model, file format, API, library and tools to manage extremely large and complex data collections [36]. Refer to Appendix B for an example of an ASCII representation of a netCDF file.

CHAPTER 4

RESULTS: PARALLEL PERFORMANCE ANALYSIS

4.1 Simulation Problem: Complex Geometry

The complex geometry of the Hagerman landscape shown in Figure 4.1 is used as a case study to evaluate the serial and parallel implementations of FSM algorithms. The two slices in Figure 4.1 represent the distance field values calculated at those locations. Various grid resolutions of the complex geometry was generated as listed in Tables 4.1 and 4.2 for analyzing the performance of all the parallel implementations of FSM.

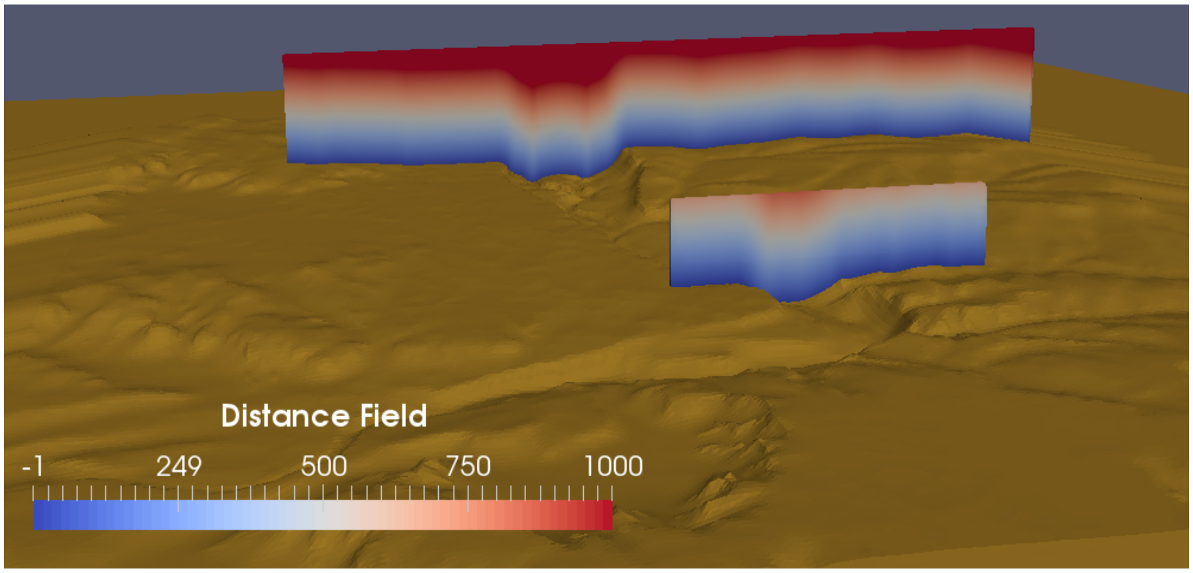

Figure 4.1: Surface representation of a complex terrain. The two slices depict the visualization of the distance field from a complex terrain.

The specifications of the hardware and the compilers used for compiling and executing the implementations are listed below:

- `Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz`
  `x86_64 GNU/Linux`
- `NVIDIA Tesla K20 ‘‘Kepler’’ M-class GPU Accelerator`
  `2496 CUDA Cores, 5GB Total Memory (208 GB/sec peak bandwidth)`
- `Compilers:`

```
  C: GCC v4.8.1              CUDA: nvcc v6.5.12
MPI: openmpi v1.8.4       OpenACC: PGI v15.7-0
```

Further details about the distributed computing cluster is provided in Section 1.4.2. The performance was calculated by averaging ten executions for each experiment. The runtime on the host and the GPU were calculated using the code snippet shown in Listings D.1 and D.2 respectively. The results of the calculations were validated by visualizing the results using the Paraview application.

4.2 Benchmark Serial FSM (Algorithm 1)

Table 4.1: Benchmark runtime of the serial implementation of FSM.

Grid Resolution	Mesh Points ($\times 10^6$)	Runtime (s)
$100 \times 100 \times 100$	1	0.65
$100 \times 500 \times 100$	5	3.38
$200 \times 500 \times 100$	10	6.32
$500 \times 500 \times 100$	25	15.72
$500 \times 500 \times 200$	50	31.27
$500 \times 500 \times 400$	100	64.90

Algorithm 1 is the FSM where the sweeps and the update of the grid points in each sweep is executed in a sequential order. The serial algorithm was implemented using the C programming language and chosen as the benchmark case with which all other performance measures were compared. The runtime for the serial implementation of FSM is shown in Table 4.1.

4.3 Zhao's Parallel FSM (Algorithm 2)

Algorithm 2 is the parallel sweeping algorithm where the sweeps (2^3) are executed by different number of processes. It is implemented using the `openmpi v1.8.4` library compiled with the `GCC v4.8.1` compiler. The program is executed with 2, 4 and 8 MPI processes on a single node. Different number of processes helped to determine the impact of the communication overhead among the processes on the overall performance of the implementation. It can be predicted that the increase in number of processes, as well as the increase in the number of mesh points increases the communication overhead. However, if the computation time takes precedence over the collective communication time then increasing the number of processes results in higher performance.

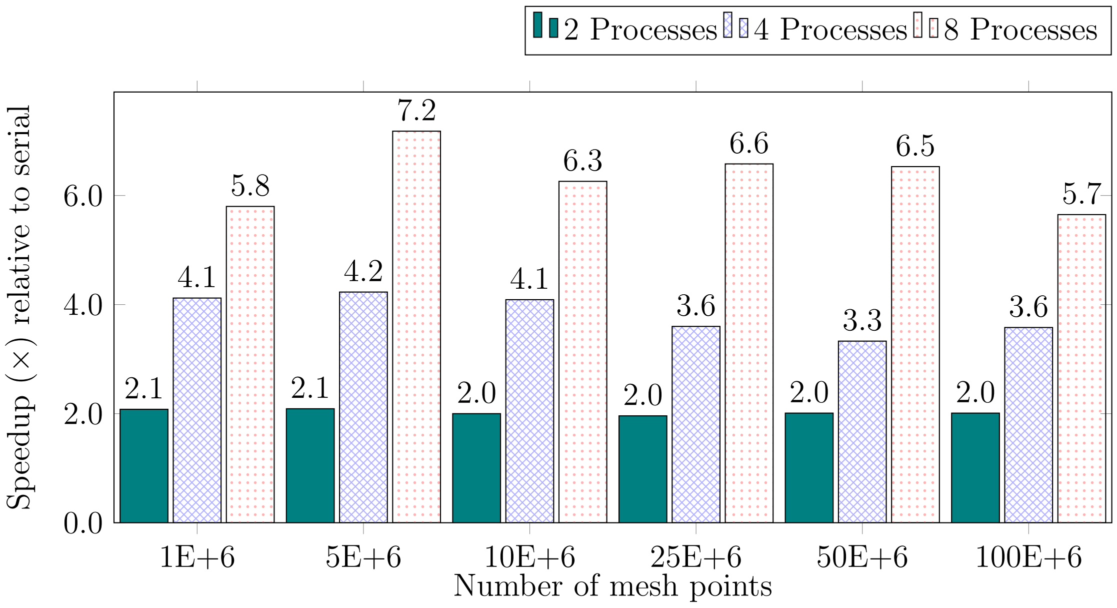

Figure 4.2: Acceleration of FSM calculation on a single node with multiple cores using MPI implementation of Algorithm 2. Speedup relative to serial implementation is plotted for various physical domain sizes and different number of processes.

This is supported by the results depicted in Figure 4.2 where execution of the implementation using 8 MPI processes results in the highest speedup relative to the serial implementation for all domain sizes. In contrast there are cases where performance decreased when increasing the number of mesh points. For example, in Figure 4.2 for 8 MPI processes increasing the number of mesh points from 5×10^6 to 10×10^6 decreased the performance by 0.9. This is because in general the percentage of time spent for communication was higher for cases with lower speedup than for cases with higher speedup as shown in Figure 4.4.

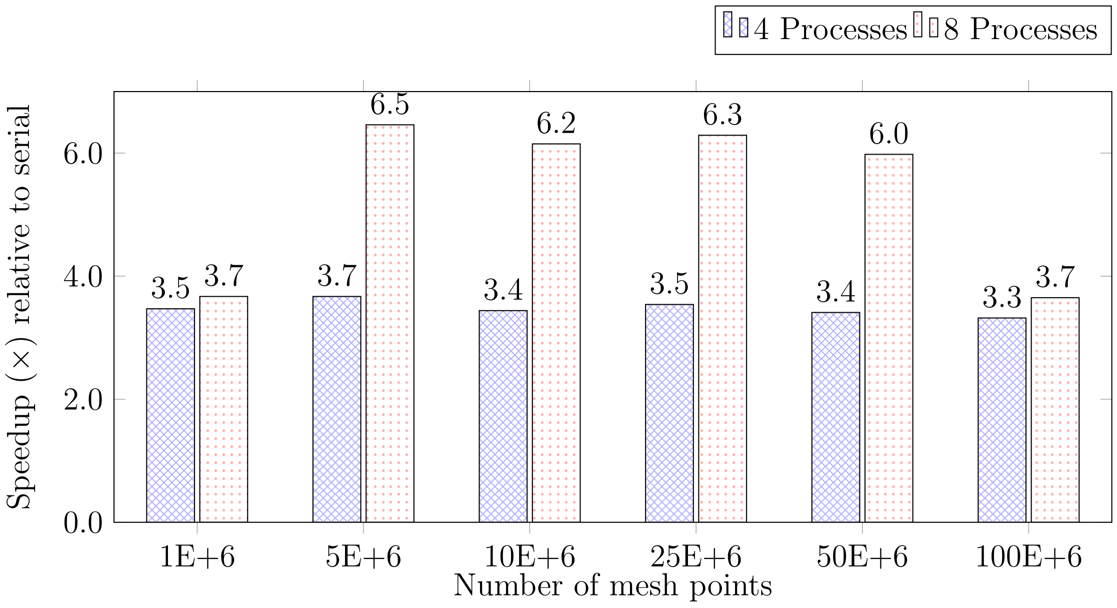

Figure 4.3: Acceleration of FSM calculation on multiple nodes (2 processes/node) with multiple cores using MPI implementation of Algorithm 2. Speedup relative to serial implementation is plotted for various physical domain sizes and different number of processes.

Furthermore, Algorithm 2 requires 8× the memory when executed with 8 processes, thus the available memory on a single node might not be sufficient to execute with 8 processes. In such cases, either the number of processes must be decreased which also reduces the speedup, or it must be executed on multiple nodes. The communication overhead on multiple nodes is much higher than a single node due to the high communication latency between network connections. Therefore, to study the impact of network communication on performance, Algorithm 2 is also executed on a multi-node environment. The results in Figure 4.3 show lower speedup than single node implementation across all possible scenarios. Both implementations have similar trend except the apparent difference in performance when increasing the number of mesh points from 1×10^6 to 5×10^6 and 50×10^6 to 100×10^6. The case with 1×10^6 mesh points spent 51% of its overall execution time for collective communication where as only 29% of the overall execution time was spent for collective communication in the case with 5×10^6 mesh points. Similarly, the case with 100×10^6 mesh points spent 57% of its overall execution time for collective communication where as only 29% of the overall execution time was spent for collective communication in the case with 50×10^6 mesh points. The percentage of time spent on communication versus computation on every mesh size is shown in Figure 4.4. Hence, it can be concluded that performance on multi-node environment is adversely affected by communication overhead for problems with both relatively small and large domain size.

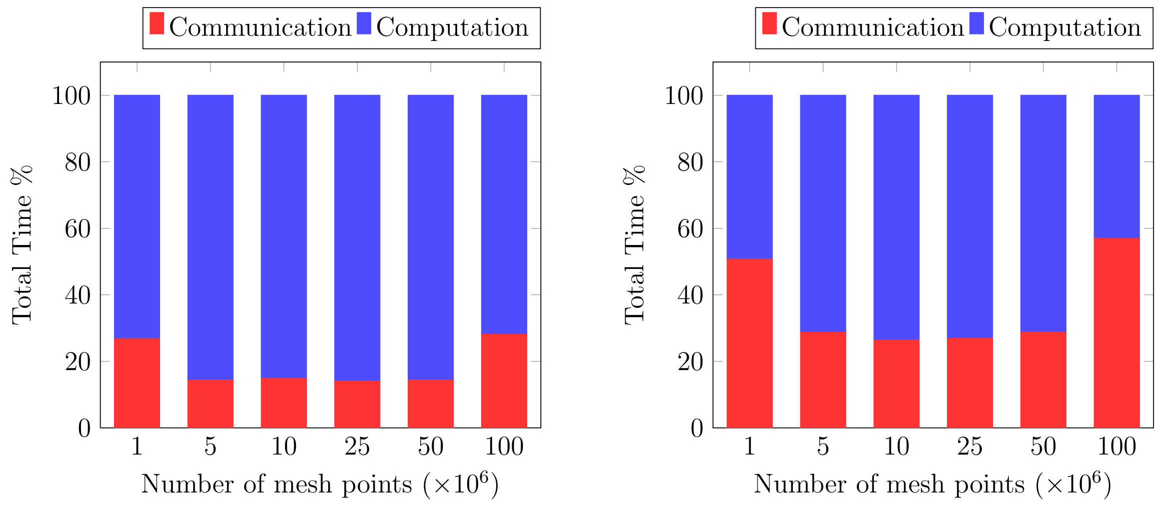

Figure 4.4: Percent of time spent in computation and communication in a single node (left) and multi-node (right) environment by Algorithm 2 implemented using MPI.

4.4 Detrixhe et al.'s Parallel FSM (Algorithm 3)

In Algorithm 3 the mesh points are ordered using the Cuthill-McKee ordering scheme where each mesh point is classified into different levels based on the sum of the point's coordinates. This ordering allows a set of mesh points within a single level to be updated independently of any other points. The algorithm was implemented using CUDA and OpenACC. The CUDA version was implemented using the `cudaMalloc3D` API instead of `cudaMalloc` API for allocating memory on the GPU. The `cudaMalloc3D` API allocates linear memory that may be padded to ensure that hardware alignment requirements are met, leading to fewer memory transactions during non-sequential access. This resulted in a performance increase of greater than 100× compared to the `cudaMalloc` implementation.

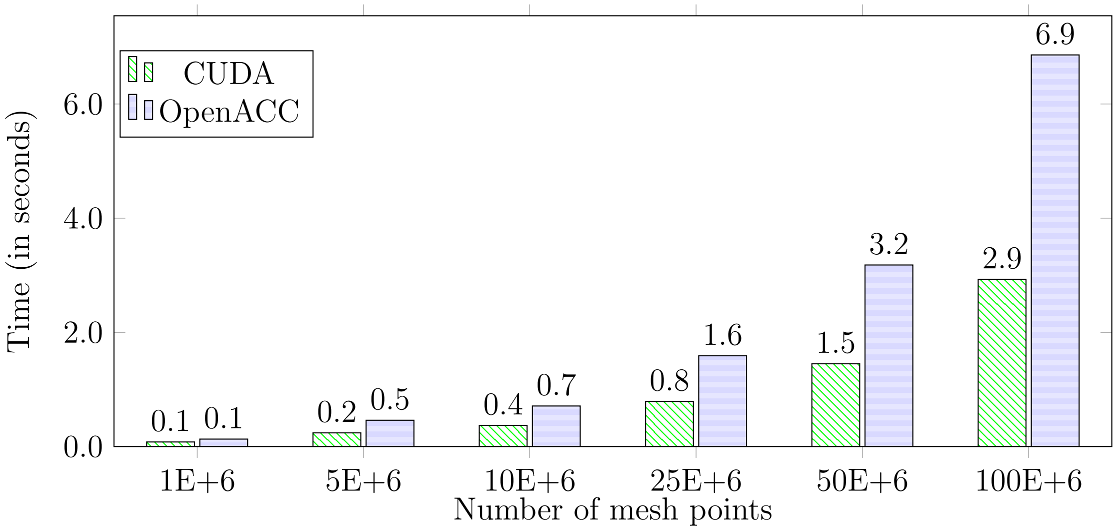

Figure 4.5: Runtime of FSM calculation on GPU using CUDA and OpenACC implementation of Algorithm 3. The runtime of CUDA and OpenACC implementation is plotted for various physical domain sizes.

The bar chart in Figure 4.5, illustrates the runtime for the parallel FSM (Algorithm 3) which follows similar trend to the results in Table 4.1, i.e. doubling of the mesh points approximately doubled the runtime of the CUDA and OpenACC implementations. Figure 4.6, shows the speedup achieved by the CUDA and OpenACC implementation of Algorithm 3 relative to the serial implementation. It can

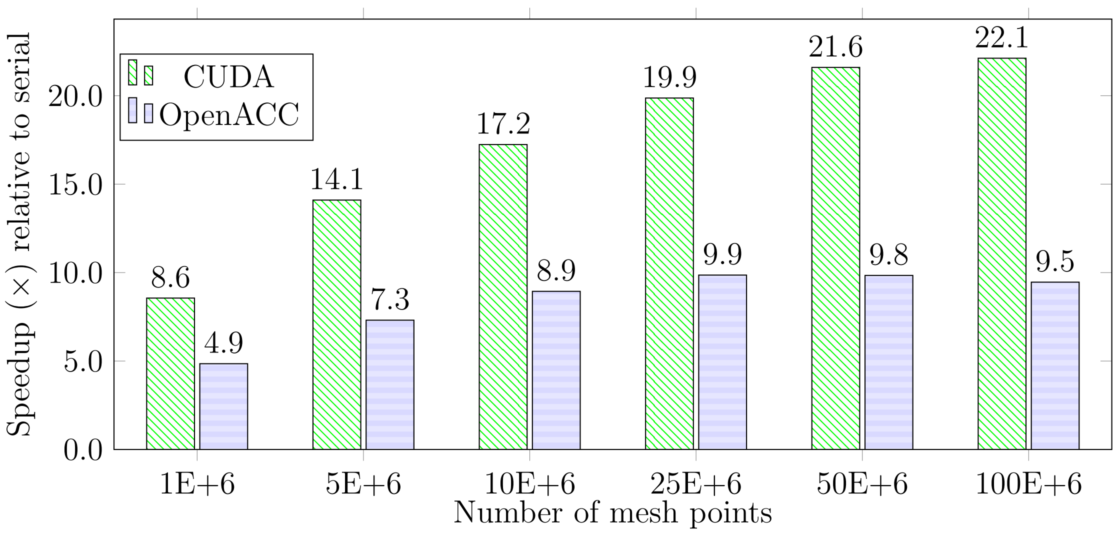

Figure 4.6: Acceleration of FSM calculation on GPU using CUDA and OpenACC implementation of Algorithm 3. GPU speedup relative to serial CPU implementation is plotted for various physical domain sizes. The CUDA implementation results in higher speedup than OpenACC implementation.

be thus concluded from the results that higher speedup was achieved by CUDA than OpenACC implementation.

However, it should be noted that CUDA is a lower-level programming model than OpenACC. CUDA requires a higher learning curve and thorough understanding of the underlying GPU hardware. On the other hand, with few modifications to the serial code OpenACC is able to achieve decent speedup to justify using GPUs. In theory, OpenACC can achieve the same acceleration speedup as CUDA but requires thorough understanding of the OpenACC parallel model and data constructs for data locality for optimization. Since, the objective of using OpenACC is to ensure simplicity and portability of the implementation. In addition, only minimal effort is dedicated to optimizing the OpenACC implementation.

Furthermore, CUDA achieves higher performance with the increase in the mesh points whereas the performance of OpenACC falters around 25×10^6 mesh points. This plateau in performance of OpenACC for large domain size could be caused by the unnecessary data transfers performed by OpenACC due to the absence of any data transfer constructs for optimization.

4.5 Hybrid (Zhao and Detrixhe et al.) Parallel FSM (Algorithm 4)

Algorithm 4 is the hybrid approach combining the parallel FSM of Zhao [41] (Algorithm 2) and Detrixhe et al. [7] (Algorithm 3). It was implemented to employ multi-level (coarse-grain and fine-grain) parallelism. In this approach, the parallel sweeps are distributed to different processes for coarse-grain parallelism, while multiple mesh points are updated simultaneously for fine-grain parallelism. Due to the multi-level parallelism, it is intuitive to think that this approach will perform better than both Algorithms 2 and 3. However, the algorithm's performance might have the same disadvantages as of Algorithm 2 that are discussed in Section 2.3.1. Furthermore, there is also the added overhead of transferring the data to and from the GPU and the collective communication that is required for assembling the final solution from different processes.

The hybrid parallel Algorithm 4 is implemented using combination of MPI/CUDA and MPI/OpenACC. The implementations are executed on a single node with a single GPU for various domain sizes with different number of processes. The results of the speedup achieved relative to the serial version is shown in Figures 4.7 and 4.8. The runtime for the 100×10^6 case with 8 processes could not be measured due to lack of required memory resources on the GPU. Therefore, the results for that specific case is not shown in the graphs.

Similar results were seen where CUDA implementation performed better than OpenACC implementation. In contrast to the MPI implementation, executing on 8 processes did not yield higher speedup for the hybrid implementation. This behavior is the result of the acceleration produced by the fine-grain parallelism using Algorithm

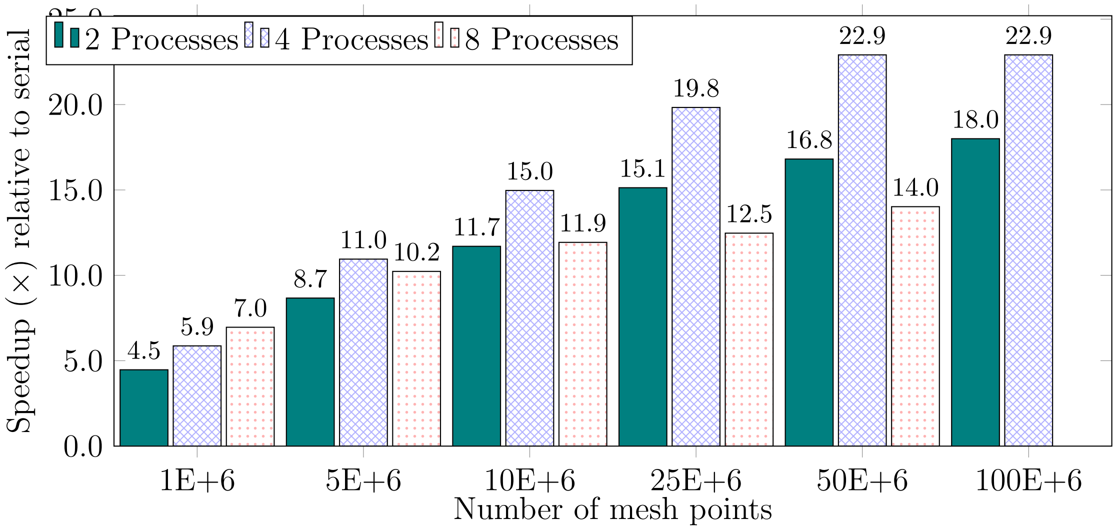

Figure 4.7: Acceleration of FSM calculation on a single node multi-core processor, single GPU using MPI/CUDA implementation of Algorithm 4. Speedup relative to serial implementation is plotted for various physical domain sizes and different number of processes.

3 that reduced the computation time. As a result, the speedup depended on the communication time. Hence, using 4 processes yielded more performance gain than using 8 processes.

4.6 Parallel FSM Algorithm for Distributed Platforms with Accelerators (Algorithm 5)

There are several limitations of the algorithms discussed thus far. Either the parallel efficiency is limited by the number of sweep directions or it cannot be extended to problems that exceed the available memory on the GPU or the node. Therefore to overcome these limitations the hybrid parallel Algorithm 5 is designed to utilize a domain decomposition strategy to partition the domain into arbitrary number of smaller subdomains. Each subdomain is managed by a process and executed in parallel as explained in Section 3.1.1. This approach avoids the use of extra memory, does not require more iterations for convergence, coarse-grain parallelism is not limited by the number of dimensions of the domain and can solve problems

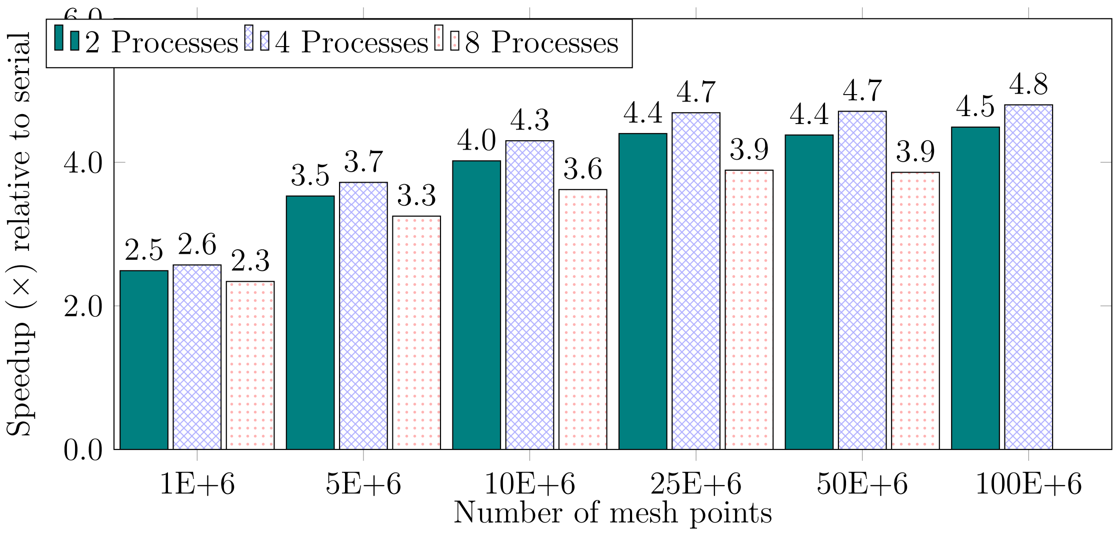

Figure 4.8: Acceleration of FSM calculation on a single node multi-core processor, single GPU using MPI/OpenACC implementation of Algorithm 4. Speedup relative to serial implementation is plotted for various physical domain sizes and different number of processes.

with domain size that exceed the available memory on the GPU or the node.

Tables 4.2 and 4.3 list the grid resolutions of the input and the corresponding number of mesh points in column 1 and 2 respectively. The remaining columns list the decomposition using the following format $x \times y \times z$ and the execution time for each combination of the grid resolution and the decomposition. The cells where runtime could not be measured due to lack of required memory resources for execution within those parameters are marked as Not Applicable (N/A). The algorithm is implemented using MPI and OpenACC. The decision to use OpenACC instead of CUDA is motivated by the flexibility, portability and simplicity in implementation provided by OpenACC.

Firstly, the implementation was executed on a distributed platform without enabling the OpenACC directives. The simultaneous execution of subdomains on the same level is the only parallelism expressed in this implementation. The maximum number of subdomains that can be executed simultaneously for decomposition of $1 \times 1 \times 1$ is 1, $2 \times 2 \times 1$ is 2, $2 \times 2 \times 2$ is 3 and $3 \times 2 \times 2$ is 4. The results of the runtime are shown in Table 4.2.

Table 4.2: Runtime for hybrid parallel Algorithm 5 w/o OpenACC directives.

Grid Resolution	Mesh Points	Runtime (s) for each Decomposition			
	($\times 10^6$)	$1 \times 1 \times 1$	$2 \times 2 \times 1$	$2 \times 2 \times 2$	$3 \times 2 \times 2$
$500 \times 500 \times 400$	100	96.67	54.35	37.92	27.34
$1000 \times 1000 \times 500$	500	N/A	894.69	559.00	185.93
$2000 \times 2000 \times 500$	2000	N/A	2390.81	1267.82	903.19

Finally, the implementation was executed with the OpenACC directives on a distributed platform with accelerators. Same execution parameters were chosen as before and the results of the runtime are shown in Table 4.3.

Table 4.3: Runtime for hybrid parallel Algorithm 5 with OpenACC directives.

Grid Resolution	Mesh Points	Decomposition Runtime (s)			
	($\times 10^6$)	$1 \times 1 \times 1$	$2 \times 2 \times 1$	$2 \times 2 \times 2$	$3 \times 2 \times 2$
$500 \times 500 \times 400$	100	8.15	18.76	17.04	13.32
$1000 \times 1000 \times 500$	500	N/A	99.33	69.84	66.92
$2000 \times 2000 \times 500$	2000	N/A	N/A	N/A	217.42

Theoretically the runtime for the $1 \times 1 \times 1$ decomposition of $500 \times 500 \times 500$ grid resolution in Tables 4.2 and 4.3 should match the runtime for $500 \times 500 \times 500$ grid resolution in Table 4.1 and Figure 4.5. However, due to different implementations and the added instructions due to MPI API calls the runtime is higher in Tables 4.2 and 4.3. The results also indicate that it might not always be possible to execute a problem with specific decomposition parameters on the GPU. OpenACC provides the flexibility of turning the directives off for serial computation of such cases which is not readily available with CUDA. The final conclusion from the runtime is that the

OpenACC implementation performed better than non-OpenACC implementation for every case that could be executed.

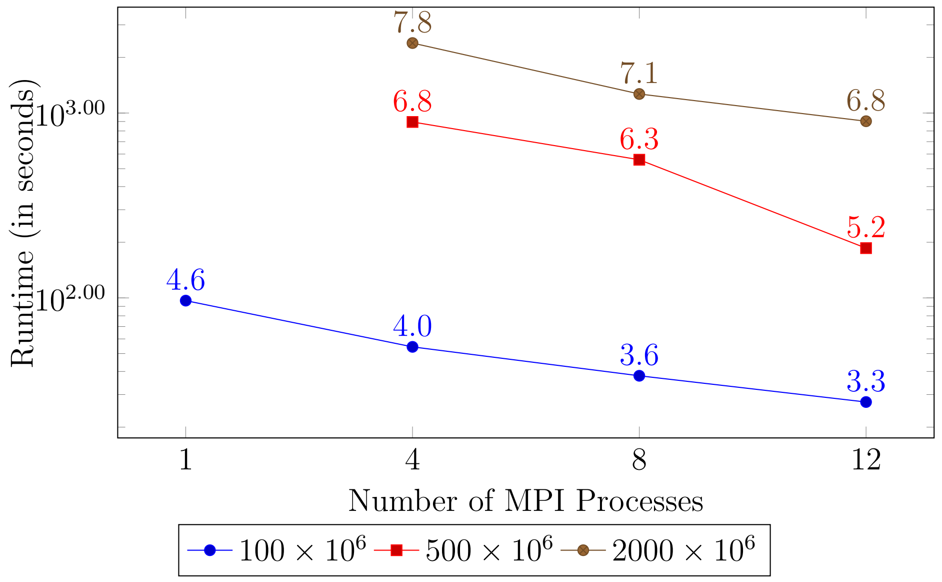

Figure 4.9: Result of increasing the number of processes by changing the decomposition parameter using Algorithm 5 w/o OpenACC directives.

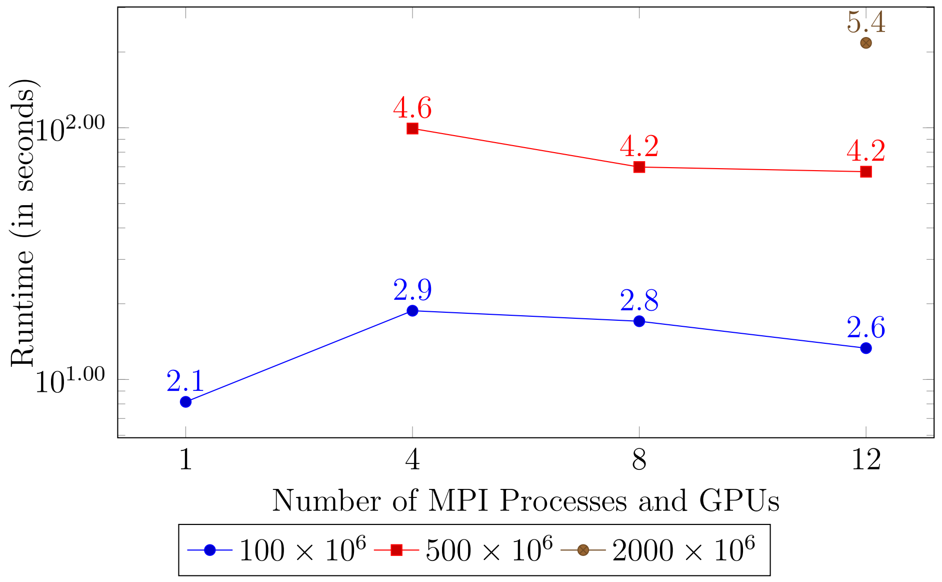

Figure 4.10: Result of increasing the number of processes by changing the decomposition parameter using Algorithm 5 with OpenACC directives.

Figures 4.9 and 4.10 show the scaling effect of this algorithm. The results demonstrate higher performance gain when increasing the size of the domain while keeping the number of processes constant. Likewise for each grid resolution the execution time is decreasing with the increase in number of processes. This is true for all cases where the algorithm is not executed on an accelerator. However, for the 100×10^6 case in Figure 4.10 executed on an accelerator the highest performance is achieved by 1 process and 1 GPU, since there is no added communication overhead. This concludes that there is an optimum execution parameter depending on the domain size and the decomposition that results in the highest performance.

4.7 Results Summary

In this chapter the runtime and speedup of various algorithms using different programming models were presented. The CUDA implementation of the Detrixhe et al. Algorithm 3 performed relatively better than any other implementations that could be executed on the available memory of the GPU and the node. However, there is a trade off between performance and portability. Therefore, if portability is the main goal with decent performance boost then OpenACC implementation of Algorithm 3 is a better option that CUDA implementation. However, Algorithm 3 fails to extend to large problems and therefore cannot always be used. In such cases a better option would be to use the MPI/OpenACC implementation of Algorithm 5.

4.8 Threats to Validity

Even though the experiments were performed with utmost care and accuracy there are several circumstances that might affect observing the same results in different experiments. For example, depending on the size of the problem and the number of processors used, the memory access latency could be higher for processes that accesses memory on a different partition. The effects of these circumstances could be substantial enough to generate different trends in the results. Furthermore, the results and conclusions are based on a single case study of a complex geometry (Hagerman). Although similar results were obtained for different geometries shown in Figure 2.2

however, a thorough analysis of the different grid resolutions is left for future work. Thus, the speedup trend for different problems is encouraging, but must be confirmed with additional experiments.

CHAPTER 5

CONCLUSIONS

5.1 Summary

The overall objective of this Master's thesis was achieved by implementing the existing parallel FSM algorithms using different programming models (MPI, CUDA, OpenACC) for various architectures (multi-core, GPU, distributed and their hybrid), and designing a new parallel FSM algorithm for distributed computing platforms with accelerators. The new parallel FSM algorithm was designed to address problems limited by memory resources. The design features multi-level parallelism that utilizes a domain decomposition strategy. The decomposition is divided in each dimension that breaks the large domain into smaller subdomains to be processed on a distributed platform. These subdomains are ordered using Cuthill-McKee ordering for applying a coarse-grain parallelism by simultaneous execution of the subdomains that lie on the same level of the ordering scheme. The same ordering scheme is applied to the mesh points of each subdomain for fine-grain parallelism. The fine-grain parallelism is implemented for an accelerator (GPU) platform using the OpenACC programming model.

The results presented in Chapter 4 show that CUDA based implementations consistently achieved better performance than OpenACC. However, CUDA is a lower-level programming model that is only compatible with NVIDIA GPUs, thus OpenACC tends to be a better option. OpenACC is a higher-level construct that makes programming accelerators simple and provides portability across various platforms and vendors (AMD, ATI, etc.). Furthermore, OpenACC directives can be ignored by platforms with no accelerators, in which case the program simply runs normally on CPUs. The tasks undertaken to accomplish the objectives were presented in detail in this thesis. Finally, the parallel implementations of the parallel FSM, Algorithm 3, for GPU-based platforms produced a speedup greater than $20\times$ compared to the serial version for some problems and the newly developed parallel algorithm can solve problems with any memory requirement (i.e., problems requiring large amount of memory resources) with comparable runtime efficiency.

5.2 Future Work

While the new algorithm fulfills the objective of this thesis work, many improvements could be made, some of which are listed below.

- There are various techniques within the OpenACC programming model that helps in optimizing the performance of the generated code. Therefore, investigation and testing of these techniques to this implementation would highly increase chances of achieving performance closer to the CUDA implementation.
- Although OpenACC is portable across different types of accelerators, the only accelerator used in this thesis is an NVIDIA GPU. Therefore, the next step would be to use Intel Xeon Phi as the targeted accelerator for executing the OpenACC generated parallel code. A performance comparison between the Intel Xeon Phi and the GPU executions can be conducted.
- Current implementation of the algorithm is only able to execute, if the number of processes launched during execution and the total number of subdomains in the decomposition are equal. This requires large amount of resources to execute problems with higher decomposition values. Although this issue can be easily addressed by executing the subdomains in serial on a single node, the coarse grain parallelism is lost, which will negatively impact the overall performance. Further research is required to optimize the single node version

for better parallel efficiency.

REFERENCES

[1] Jakob Andreas Barentzen and Niels Jorgen Christensen. *Manipulation of volumetric solids with applications to sculpting.* PhD thesis, Technical University of Denmark, 2001.

[2] Blaise Barney. Message passing interface (mpi). https://computing.llnl.gov/tutorials/mpi/, 6 2016.

[3] J. Bender, C. Duriez, F. Jaillet, and G. Zachmann. Continuous collision detection between points and signed distance fields. *Workshop on Virtual Reality Interaction and Physical Simulation VRIPHYS*, 2014.

[4] Robert Bridson. *Fluid Simulation for Computer Graphics.* CRC press, 2008.

[5] Anna R. Bruss. The eikonal equation: some results applicable to computer vision. *Shape from shading*, pages 69–87, 1989.

[6] Rachel Courtland. What intels xeon phi coprocessor means for the future of supercomputing. *IEEE Spectrum*, 7 2013.

[7] Miles Detrixhe, Frdric Gibou, and Chohong Min. A parallel fast sweeping method for the eikonal equation. *Journal of Computational Physics*, 237:46–55, 2013.

[8] K. Erleben and Dohlmann H. *Signed Distance Fields Using Single-Pass GPU Scan Conversion of Tetrahedra*, volume 3. 2008.

[9] Karthik S. Gurumoorthy, Adrian M. Peter, Birmingham Hang Guan, and Anand Rangarajan. A fast eikonal equation solver using the schrodinger wave equation. *arXiv preprint arXiv:1403.1937*, 2014.

[10] Mark Harris. Openacc: Directives for gpus. https://devblogs.nvidia.com/parallelforall/openacc-directives-gpus/, 3 2012.

[11] Kenneth E. Hoff III, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. Fast computation of generalized voronoi diagrams using graphics hardware. *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pages 277–286, 1999.

[12] Intel. Meet the *Intel*® *Xeon*® Processor 37 v4 Family, 2016. http://www.intel.com/content/www/us/en/processors/xeon/xeon-processor-e7-family.html.

[13] Li Jianwei, Wei keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Andrew Siegel Robert Latham, Brad Gallagher, and Michael Zingale. Parallel netcdf: A high performance scientific i/o interface. pages 39–39. IEEE, In Supercomputing, 2003 ACM/IEEE Conference, 2003.

[14] Michael F. Kilian. Can oo aid massively parallel programming. *In Proceedings of the Dartmouth Institute for Advanced Graduate Study in Parallel Computation Symposium*, pages 246–256, 1992.

[15] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors A Hands-on Approach*. Elsevier Inc., 1 edition, 2010.

[16] Seyong Lee and Jeffrey S. Vetter. Early evaluation of directive-based gpu programming models for productive exascale computing. *In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 23, 2012.

[17] Hassan Mujtaba. Nvidia pascal gpus double precision performance rated at over 4 tflops, 16nm finfet architecture confirmed volta gpu peaks at over 7 tflops, 1.2 tb/s hbm2. http://wccftech.com/nvidia-pascal-volta-gpus-sc15/, 2015.

[18] NVIDIA. Nvidia. cuda compute unified device architecture programming guide. http://www.nvidia.com/object/cuda programming tools.html, 2008.

[19] NVIDIA. *CUDA C Programming Guide*, 2015.

[20] NVIDIA. Nvidia accelerated computing: Cuda faq. https://developer.nvidia.com/cuda-faq, 2016.

[21] NVIDIA. What is GPU-Accelerated Computing?, 2016. http://www.nvidia.com/object/what-is-gpu-computing.html.

[22] OpenACC-standard.org. Openacc directives for accelerators: Faq. http://www.openacc.org/faq-questions-inline, 2016.

[23] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[24] Jeff Parkhurst, John Darringer, and Bill Grundmann. From single core to multi-core: preparing for a new exponential. *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 67–72, 2006.

[25] Reza Rahman. Intel xeon phi core micro-architecture. 5 2013.

[26] Reza Rahman. Intel xeon phi programming environment. 5 2013.

[27] Farber Rob. *CUDA Application Design and Development*. Elsevier, 2011.

[28] Inanc Senocak and Haoqiang Jin. *Introduction to Scientific and Technical Computing*, chapter 16, pages 219–234. CRC Press, 2 edition, 2016.

[29] Inanc Senocak, Micah Sandusky, Rey DeLeon, Derek Wade, Kyle Felzien, and Marianna Budnikova. An immersed boundary geometric preprocessor for arbitrarily complex terrain and geometry. *Journal of Atmospheric and Oceanic Technology*, 32(11):2075–2087, 2015.

[30] James Albert Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93(4):1591–1595, 1996.

[31] James Albert Sethian. *Level set methods and fast marching methods: evolving interfaces in computational geometry, fluid mechanics, computer vision, and materials science*, volume 3. Cambridge university press, 1999.

[32] C. Sigg, Peikert R., and Gross M. ”signed distance transform using graphics hardware. *In Proceedings of IEEE Visualization*, pages 83–90, 2003.

[33] Steven S. Skiena. *The algorithm design manual*, volume 1. Springer Science & Business Media, 1998.

[34] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference.* The MIT Press, 1 edition, 1196.

[35] Avneesh Sud, Naga Govindaraju, Russel Gayle, Ilknur Kabul, and Dinesh Manocha. Fast proximity computation among deformable models using discrete voronoi diagrams. *ACM Transactions on Graphics*, 25(3):1144–1153, 2006.

[36] The HDF Group. Hierarchical Data Format, version 5, 1997-2016. http://www.hdfgroup.org/HDF5/.

[37] Julien C. Thibault. Implementation of a cartesian grid incompressible navier-stokes solver on multi-gpu desktop platforms using cuda. Master's thesis, Boise State University, 3 2009.

[38] Unidata. Network Common Data Form (netCDF), version 4.4.1 [software], 2016. http://doi.org/10.5065/D6H70CW6.

[39] M Mitchell Waldrop. The chips are down for moore's law. *Nature News*, 2016.

[40] Hongkai Zhao. A fast sweeping method for eikonal equations. *Mathematics of computation*, 74(250):603–627, 2004.

[41] Hongkai Zhao. Parallel implementations of the fast sweeping method. *Journal of Computational Mathematics-Internation Edition*, 25(4):421, 2007.

APPENDIX A

CODING STYLE GUIDE

This chapter discusses recommended practices, coding styles and conventions for writing and organizing a C program. “Good programming style begins with the effective organization of code. By using a clear and consistent organization of the components of your programs, you make them more efficient, readable, and maintainable.” - Steve Oualline, C Elements of Style. Coding style and consistency matters for writing efficient code and increasing productivity. A code that is easy to read is also easy to understand, allowing programmers to focus on substance rather than spending time figuring out what the code does. Therefore, a code with “good style” is defined as that which is

- Organized
- Easy to read and understand
- Efficient
- Maintainable and Extensible

The guidelines presented here are based on recommended software engineering techniques, industry standards, recommended practices from the experts in the field and local conventions. Various parts of the guidelines have been extracted from the NASA C Style Guide, The C Programming Language and the Linux Kernel Coding Style.

A.1 Program Organization

A C program consists of multiple routines that are grouped together into different files based on their functionality called module or source files. The prototype of the functions of each module file that are shared between different modules are put in a separate file called a header file. Furthermore, usually with large programs there is also a build file (e.g. Makefile) that automates the process of compiling and linking all the files of the program. During compilation, the compiler generates object files for each source file that gets linked together to generate the final executable. Hence, it is easy to see that the number of files in a C program could increase significantly therefore, a proper organization schema is a must to maintain consistency and good style. A recommended schema for organizing a C program is shown in Fig. A.1.

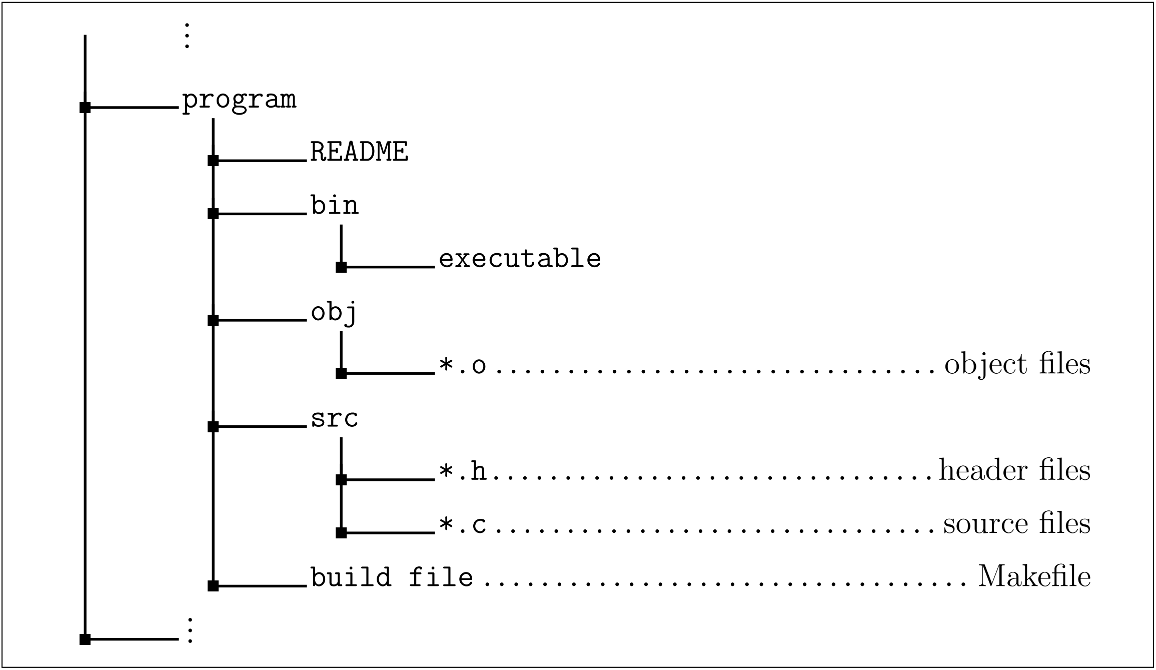
program
README
bin
executable
obj
*.o .... object files
src
*.h .... header files
*.c .... source files
build file .... Makefile

Figure A.1: Program Organization Schema

The following sections discusses the different files in a C program and recommends a good coding style to follow.

A.1.1 README File

A README file should be named "README" and succinctly explain what the program does and how the different files are organized. It should contain all the important information about the program as following

- Program name followed by a brief description
- Version, Contact, and License information

- Program Organization
- Requirements and Dependencies
- Instructions for compiling and installing the program
- Usage information for running the program

An example template of a README file is shown in Fig. A.2.

A.1.2 Header Files

A program with good style uses module files to group logically related functions into one file. Header files make it possible to share information between modules that need access to external variables or functions. It also encapsulates the implementation details in the module files. Follow the following guidelines when writing a header file.

- Save the header file with a ".h" extension and use an all lowercase filename that best describes the relation of its content. Avoid names that could conflict with system header files.
- Start header files with an include guard. It prevents the compiler from processing the same header file more than once.
- Use the header filename in all uppercase and append "_H" to define the header.
- Use specialized (e.g., doxygen) comments in order to generate documentation automatically.
- Follow the header file template in Fig. A.3 for the prologue.
- Only include other header files if they are needed by the current header file. Do not rely on users including things.
- Group the header file includes logically from local to global and arrange them in an alphabetical order for each group.
- Do not use absolute path in #include directive.
- Use #include <system_file>for system include files.
- Use #include "user_file" for user include files.

```
Example Program Version #.# MM/DD/YYYY
Copyright (C) <year> <name of author>
Email: <email of author>

Synopsis
----------
A brief introduction and/or overview that explains what the program is.

License
---------
This program is free software; you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by the
Free Software Foundation; either version 2 of the License, or (at your
option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
General Public License for more details.

Program Organization
----------------------
Directory structure similar to Fig. A.1.

Requirements/Dependencies
---------------------------
List all requirements including hardware (GPUs, CPU architecture)
and software (operating systems, external modules, libraries).

Installation
--------------
Provide instructions on how to compile and run the code.

Usage
-------
<program_name> <args> [options]

Contributors
--------------
<name of contributor> - <email address>
```

Figure A.2: README file template

- Put data definitions, declarations, typedefs and enums used by more than one program in a header file.
- Only include those function prototypes that need to be visible to the linker.
- End the header file with the following comment on the same line as the end of the include guard.

  ```
  #endif /* END <FILENAME>_H */
  ```

An example template of a C header file is shown in Fig. A.3.

A.1.3 Source Files

A source file contains the implementation of logically related functions, constants, types, data definitions and declarations. Follow the following guidelines when writing a source file.

- Save the source file with a ".c" extension and use an all lowercase filename that best describes the relation of its content or the same as the header file whose functions it is implementing. If the source file contains the main function then name it "`main.c`".
- Start with the file prologue. Follow the source file template in Fig. A.3 for the prologue.
- Then, include header files that are necessary.
- Group the header file includes logically from local to global and arrange them in an alphabetical order for each group.
- Do not use absolute path in #include directive.
- Use #include <system_file>for system include files.
- Use #include "user_file" for user include files.
- Use specialized (e.g., doxygen) comments in order to generate documentation automatically.
- Make the local functions static and follow the guidelines listed in Section A.2.6.

```
#ifndef FILENAME_H
#define FILENAME_H

/**
 * @file <filename>.h
 * @brief Brief explanation of the header file.
 *
 * Here typically goes a more extensive explanation of what the header
 * defines.
 *
 * @author Last Name, First Name Middle Initial
 * @date DD MON YYYY
 * @see http://website/
 *
 * @note Something to note.
 * @warning Warning.
 * @bug No known bugs.
 *
 */

#include "local1.h"
#include "local2.h"

#include "external.h"

#include <system.h>

/**
 * @brief Brief explanation
 *
 * @param[in]     arg1 explain the argument
 * @param[in,out] arg2 explain the argument
 * @return void
 *
 */
void functionPrototype(int arg1, char *arg2);

#endif /* END FILENAME_H */
```

Figure A.3: C header file template

- Format the file as described in Section A.2.

An example template of a C source file is shown in Fig. A.4.

A.2 Recommended C Coding Style and Conventions

A.2.1 Naming

Names for files, functions, constants or variables should be descriptive, meaningful and readable. However, over complicating a variable name is also not recommended. For example, a variable that holds a temporary value can always be named 'tmp' instead of 'temporaryCounterVariable'. As a general rule, the scope of a variable is directly proportional to the length of its name. For example, an important variable that is used in most places should have a very descriptive name. Conversely, local variable names should be short, and to the point. A random integer loop counter, should probably be called "i", since this is very common and there is no chance of it being mis-understood. Furthermore, the names of functions, constants and typedefs should be self descriptive, as short as possible and unambiguous. Finally, the names should always be written in a consistent manner as shown in Table A.1. Here, for consistency the recommended naming convention to use is *lowerCamelCase* unless specified explicitly.

Table A.1: Recommended Naming Convention

Name	Convention
file name	should start with letter and be a noun
function name	should start with letter and be a verb
variable name	should start with letter
constant name	should be in uppercase words separated by underscores
type name	should start with letter and be a noun
enumeration name	should be in uppercase words separated by underscores
global name	prefix using `g_`

```
/**
 * @file <filename>.c
 * @brief Brief explanation of the source file.
 *
 * Here typically goes a more extensive explanation of
 * what the source file does.
 *
 * @author Last Name, First Name Middle Initial
 * @date DD MON YYYY
 *
 * @warning Warning.
 * @bug No known bugs.
 *
 */

#include "header.h"

#include <system.h>

/**
 * @brief Brief explanation
 *
 * Add more details here if needed.
 *
 * @param[in]     arg1 explain the argument
 * @param[in,out] arg2 explain the argument
 * @return void
 */
void function(int arg1, char *arg2)
{
    do sth ...
}

/**
 * @brief Brief explanation
 *
 * Add more details here if needed.
 *
 * @param[in] argc command-line arg count
 * @param[in] argv command-line arg values
 * @return int
 */
int main(int argc, char *argv)
{
    return 0;
}
```

Figure A.4: C source file template

A.2.2 Indentation

Indentation defines an area where a block of control starts and ends. Proper indentation makes the code easier to read. Always use **tabs** to indent code rather than spaces. The rationale being spaces can often be out by one, and lead to misunderstandings about which lines are controlled by if-statements or loops. An incorrect number of tabs at the start of line is easier to spot. Tabs are usually 8 characters however, for scientific code where use of multiple nested for-loops is the norm, having a large indentation is not recommended. Use **4** characters deep indent which will still make the code easy to read and not flushed too far to the right.

A.2.3 Braces

The placement of the braces is a style choice and different styles have their own benefits. However, for consistency and the benefit of minimizing the number of lines without any loss of readability follow the "K&R" style for brace placement. Their preferred way is to put the opening brace last on the line, and put the closing brace first.

```
if (true) {
    do sth ...
}
```

This applies to all non-function statement blocks `(if, switch, for, while, do)`. However, there is one special case, namely functions: they have the opening brace at the beginning of the next line, thus:

```
int function(int x)
{
        ...
}
```

Note that the closing brace is empty on a line of its own, except in the cases where it is followed by a continuation of the statement, i.e., a while in a do-statement or an else in an if-statement, like this:

```
do {
        ...
} while (condition);
```

```
if (x == y) {
        ...
} else if (x > y) {
        ...
} else {
        ...
}
```

Do not unnecessarily use braces where a single statement will do.

```
if (condition)
        action();
```

```
if (condition)
        doThis();
else
        doThat();
```

This does not apply if only one branch of a conditional statement is a single statement; in the latter case use braces in both branches:

```
if (condition) {
    doThis();
    doThat();
} else {
    otherwise();
}
```

A.2.4 Spaces

Spaces should be used to separate components of expressions, to reveal structure and make clear intention. Good spacing reveals and documents programmer's intentions whereas poor spacing can lead to confusion.

- Use a space after keywords like:

  `if, switch, case, for, do, while`.

- Use one space on each side of binary and ternary operators like:

  `= + - < > * / | & ^ <= >= == != ?  :  %`

- Use a space (or newline) after commas `,` and semicolons `;`.

- Do not use space after unary operators like:

  `& * + - ~ !  sizeof typeof alignof __attribute__ defined`

- Do not use space around the primary operators like:

  `->, ., and [].`

- Do not use space before the prefix and after the postfix increment and decrement unary operators like:

  `++ --`

- Do not leave trailing whitespace at the ends of lines.

A.2.5 Commenting

Comments when used properly can provide important information that cannot be discerned by just reading the code. However, if used improperly it can make code difficult to understand. When writing comments do not try to explain how the code works, let the code speak for itself. In general comments should explain what the code does and possibly why it does it. The preferred style for multi-line comments is specialized, doxygen style comment block, so that documentation can be generated automatically.

```
/**
 * \brief Brief description.
 *        Brief description continued.
 *
 * This is the preferred style for multi-line
 * comments using the doxygen style comment block.
 * Two column of asterisks on the left side of the
 * first line. Then, a single column of asterisks
 * on the left side, ending with almost-blank line.
 *
 * \author Name
 * @see functions()
 */
```

Also, add comments to data throughout the file, where they are being declared or defined. Use one data declaration per line (i.e., no commas for multiple data declarations), leaving space for small comment on each item, explaining its purpose. If more than one short comment appears in a block of code or data definition, start all of them at the same tab position and end all at the same position.

```
void someFunction()
{
    doWork();     /* Does something */
    doMoreWork(); /* Does something else} */
}
```

A.2.6 Functions

Design functions to be short that does just one thing and does that well. Each function should be preceded by a function prologue that gives a short description of what the function does and how to use it. Avoid duplicating information clear form the code. An example function prologue is shown in Fig A.5. The recommended conventions for declaring a function

- The function's return type, name and arguments should be on the same line, unless the number of arguments do not fit on a single line. In that case move the rest of the arguments to the next line aligned with the arguments on the line above.
- The name of the function should be descriptive and follow the naming conventions defined in Table A.1.
- Do not default to int; if the function does not return a value then it should be given return type void.
- The opening brace of the function body should be alone on a line beginning in column 1.
- Declare each parameter, do not default to int.
- Comments for parameters and local variables should be tabbed so that they line up underneath each other.
- In function prototypes, include parameter names with their data types.

A.2.7 Structs, Macros and Enums

- Structs

  Structures are very useful feature in C, that enhances the logical organization

```
/**
 * @brief Brief explanation
 *
 * Add more details here if needed.
 *
 * @param[in]     arg1 explain the argument
 * @param[in,out] arg2 explain the argument
 * @return void
 */
void function(int arg1, char *arg2)
{
    do sth ...
}
```

Figure A.5: C function template

of the code and offers consistent addressing. The variables declared in a struct should be ordered by type to minimize any memory wastage because of compiler alignment issues, then by size and then by alphabetical order.

```
struct point3D {
    int x;
    int y;
    int z;
}
```

- Macros

  Using the #define preprocessor command to define constants is a convenient technique. It not only improves readability, but also provides a mechanism to avoid hard-coding numbers. Use uppercase letters to name constants and align the various components as shown in the example below.

  ```
  #define NULL  0
  #define FALSE 0
  #define TRUE  1
  ```

  On the other hand avoid using macro functions. They are generally a bad idea with potential side-effects without any advantages.

- Enums

  Enumeration types are used to create an association between constant names and their values. Use uppercase letters to name the enum type and the constants. Place one variable identifier per line and use aligned braces and indentation to improved readability.

```
enum POSITION {                          enum POSITION {
        LOW,                                     LOW    = -1,
        MIDDLE,                                  MIDDLE = 0,
        HIGH                                     HIGH   = 1
};                                       };
```

APPENDIX B

NETCDF FILE FORMAT

Listing B.1: ASCII text representation of a netCDF file.

```
netcdf example_netcdf {
dimensions:
x = 10 ;
y = 5 ;
z = 2 ;
variables:
double Variable(z, y, x) ;
data:
Variable =
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0,
2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0,
2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0,
2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0,
2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0 ;
}
```

APPENDIX C

METHOD THAT SOLVES THE EIKONAL EQUATION

```
static double solveEikonal(Phi *pf, int index)
{

  double dist_new = 0;
  double dist_old = pf->distance[index];

  double dx = pf->dx;
  double dy = pf->dy;
  double dz = pf->dz;

  double minX = min(pf->distance[index - 1], pf->distance[index + 1]);
  double minY = min(pf->distance[abs(index - max_x)], pf->distance[abs(index + max_x)]);
  double minZ = min(pf->distance[abs(index - max_xy)], pf->distance[abs(index + max_xy)]

  double m[] = { minX, minY, minZ };
  double d[] = { dx, dy, dz };

  // sort the mins
  int i, j;
  double tmp_m, tmp_d;
  for (i = 1; i < 3; i++) {
    for (j = 0; j < 3 - i; j++) {
      if (m[j] > m[j + 1]) {
        tmp_m = m[j];
        tmp_d = d[j];
        m[j] = m[j + 1];
        d[j] = d[j + 1];
        m[j + 1] = tmp_m;
        d[j + 1] = tmp_d;
      }
    }
  }

  // simplifying the variables
  double m_0 = m[0], m_1 = m[1], m_2 = m[2];
  double d_0 = d[0], d_1 = d[1], d_2 = d[2];
  double m2_0 = m_0 * m_0, m2_1 = m_1 * m_1, m2_2 = m_2 * m_2;
  double d2_0 = d_0 * d_0, d2_1 = d_1 * d_1, d2_2 = d_2 * d_2;

  dist_new = m_0 + d_0;
  if (dist_new > m_1) {
    double s = sqrt(-m2_0 + 2 * m_0 * m_1 - m2_1 + d2_0 + d2_1);
    dist_new = (m_1 * d2_0 + m_0 * d2_1 + d_0 * d_1 * s) / (d2_0 + d2_1);

    if (dist_new > m_2) {

      double a = sqrt(-m2_0 * d2_1 - m2_0 * d2_2 + 2 * m_0 * m_1 * d2_2 - m2_1 * d2_0 -
                      m2_1 * d2_2 + 2 * m_0 * m_2 * d2_1 - m2_2 * d2_0 - m2_2 * d2_1 +
                      2 * m_1 * m_2 * d2_0 + d2_0 * d2_1 + d2_0 * d2_2 + d2_1 * d2_2);

      dist_new = (m_2 * d2_0 * d2_1 + m_1 * d2_0 * d2_2 + m_0 * d2_1 * d2_2 +
                 d_0 * d_1 * d_2 * a) / (d2_0 * d2_1 + d2_0 * d2_2 + d2_1 * d2_2);
    }
  }
  return min(dist_old, dist_new);
}
```

Listing C.1: Serial code fragment that solves the eikonal equation on a signle mesh point.

APPENDIX D

TIMING CODE SNIPPETS

```
#include <sys/time.h>

/* The argument now should be a double (not a pointer to a double) */
#define GET_TIME(now)                                                   \
{                                                                       \
        struct timeval t;                                               \
        gettimeofday(&t, NULL);                                         \
        now = t.tv_sec + t.tv_usec / 1000000.0;                         \
}
```

Listing D.1: Code snippet of a macro function used for timing the results on the host side. The function returns time with microsecond accuracy.

```
// time cuda code
cudaEvent_t start, stop;
float elapsedTime;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

/* CUDA CODE */

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime, start, stop);
```

Listing D.2: Code snippet used for timing the CUDA section.