Volumetric Modeling Using a Generic Three-Dimensional Framework

by
Eric Torunski

23rd September 2004

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

© Copyright
2004, Eric Torunski

Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file   Votre référence

ISBN: 0-612-97458-8

Our file   Notre référence

ISBN: 0-612-97458-8

The author has granted a non-exclusive license allowing the Library and Archives Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

PAGINATION ERROR.

TEXT COMPLETE.

ERREUR DE PAGINATION.

LE TEXTE EST COMPLET.

Abstract

This thesis presents a three-dimensional modeling (3-MO) software framework that facilitates the creation of 3-dimensional modeling applications for parallel computers. 3-MO provides a generic volumetric data structure and functionality for inter-processor communication, and a simple interface which abstracts programming issues of parallel applications so that developers can easily write a parallel three-dimensional application. 3-MO can then be extended for use in a specific model, thus reducing the time needed to develop such applications.

This thesis presents two models that were implemented to use 3-MO. The first is a fire spread model, which models the spread of fire in a region of space. The second model simulates the explosive expansion of different elementary particles when subjected to a high-intensity laser beam.

Acknowledgements

I would like to start by expressing my thanks to my supervisors, Dr. Sack and Dr. Nussbaum for their time, help and patience. I have learned a lot from this thesis and from working with you. I am very thankful for all that you have done for me.

I also thank my parents and my older brother for their support and kindness. I thank you for always being there and lending a helping hand when needed.

Many thanks also go to my friends Nora Kapsalis and Geoff Bennett, Bogdan Gheorghe, Geoff Powell, Jen Prouse, Nicole McNeely, Éric Gervais, Véronique Audet, Miguel Vargas, Carolyn Gorrell, Lauren Gerber, Lindsay Freeman, Eric Cooper, Michael Heinemann, Tam Tran. They have all helped me at times to focus on my thesis, and other times to forget all about my thesis.

I would also thank my exchange supervisors at Universität Paderborn. My collaboration with Dr. Gitta Domik and Dr. Ulf-Peter Schröder was very helpful. I had a wonderful time while on exchange in Paderborn, Germany and learned a lot about life that could not be taught in any class or book.

Last but not least, I would like to thank Dr. Brabec and Christian Jungreuthmayer from the University of Ottawa for letting me use their ionized cluster particle model. I would especially like to thank Christian for spending the time to explain their software model and to answer any other questions that I had.

Contents

1	Introduction	1
2	Background	4
2.1	Cellular Automata	4
2.2	Neighbourhood Types	6
2.2.1	Von Neumann Neighbourhood Model	6
2.2.2	Moore Neighbourhood Model	7
2.2.3	Neighbourhood Depth	8
2.3	Three-dimensional Building Blocks and Data Structures	9
2.3.1	Voxel	9
2.3.2	Partition Block	10
2.4	Data Structures	12
2.4.1	Array	12
2.4.2	Octree	13
2.4.3	Other spatial data structures not used in the framework	14
2.4.3.1	K-d Trees	15
2.4.3.2	Binary Space Partition (BSP) trees	16
2.4.3.3	Other data structures	17
2.5	Parallel Cellular Automata Systems	17
2.5.1	Camel	17
2.5.2	Trend	18
2.5.3	CAPE	18
2.5.4	NEMO	19
2.5.5	Summary	21
2.6	Generic Programming	21
3	Parallel Computing Preliminary	24
3.1	Parallel Computing Models, Architectures and Topologies	24
3.1.1	Theoretical Parallel Computing Models	24
3.1.2	Bulk Synchronous Parallelism (BSP) model.	26
3.1.3	Coarse Grained Multicomputer (CGM) model.	27
3.1.4	Parallel Architectures and Topologies	28
3.2	Factors Affecting Parallel Performance	34
3.2.1	Algorithmic Factors	34
3.2.2	Communication Versus Computation	34
3.2.3	Data Related Factors	37
3.2.4	Implementation Related Factors	38
3.3	Mapping Schemes	38
3.3.1	Standard Mapping	40

3.3.2	Two-Dimensional Folding	42
3.4	Visualization	45
3.4.1	Slice Generation	45
3.4.2	Isosurfaces	47
4	Three Dimensional Modelling Framework Architecture	48
4.1	User-defined Components	48
4.2	Model Configuration	50
4.3	Principal Components	50
4.4	Program Flow	52
4.4.1	Initialization	53
4.4.2	Updating	55
4.4.3	Model Computation	59
4.4.4	Visualization by Slice Generation	60
4.4.4.1	Constructing a Slice	63
4.4.4.2	Combining Images	65
4.4.4.3	Slice Construction Algorithm Optimization	67
4.4.5	Termination and Clean-Up	68
5	Model Examples	69
5.1	Fire Model Description and Theory	70
5.1.1	Integration of the Fire Model and 3-MO	72
5.1.2	Fire Model Controls	74
5.2	Ionized Particle Cluster Explosion Model Description and Theory	81
5.2.1	Integration of the Cluster Explosion Model with 3-MO	82
5.2.2	Integration Issues	85
5.3	Results	86
5.3.1	Computer Environment	86
5.3.2	Test of Speed-up for Fire Model and Particle Explosion Model	87
5.3.2.1	FireModel Speed-up	87
5.3.2.2	Overpartitioning	92
5.3.2.3	Ionized Particle Explosion Speed-up	97
5.3.3	Block size test	101
5.3.4	Mapping Schemes	103
5.3.5	Slice Generation	105
5.4	Summary	107
5.5	Future work	108

List of Figures

1	The von Neumann neighbourhood in two dimensions (4 neighbours), and three dimensions (6 neighbours).	7
2	The Moore neighbourhood in two dimensions (8 neighbours) and three dimensions (26 neighbours).	8
3	Neighbourhood depth of 1 voxel on the left and 2 voxels on the right.	9
4	Neighbouring voxels.	10
5	Neighbouring voxels that lie on different processors.	11
6	Copies of Voxels Stored Locally on Neighbouring Volumes	12
7	A region divided into eight equal sub-regions.	13
8	Parallel Processors Sharing a Common Memory Space	25
9	Interconnected processors that have local memory.	26
10	The linear array and ring processor topologies.	29
11	The mesh and torus processor topologies.	30
12	The hypercube processor topology.	31
13	The tree processor topology.	32
14	The cluster processor topology.	33
15	A three-dimensional lattice of 12 processors, ordered by their id number.	40
16	Diagram of standard mapping in 2 dimensions.	41
17	Diagram of standard mapping in 3 dimensions.	42
18	Processor folding in 2 dimensions.	43
19	Processor folding in 3 dimensions.	44
20	Generating a two-dimensional slice from three-dimensional data.	46
21	3-MO's architecture.	51
22	Synchronized initialization	54
23	Synchronized updating in various directions.	57
24	Processors updating within a processor lattice and updating in a random order.	58
25	Visualization by slice generation.	60
26	Four cases of intersection between a plane and rectangloid	62
27	Combination of two images.	65
28	The bounding rectangloid of a plane intersecting a volume.	68
29	The fire model user interface.	73
30	Screenshot of a plane intersecting the data along the middle z coordinate.	75
31	An example of a plane that is perpendicular to the z axis.	75
32	Screenshot of a plane intersecting the data along the bottom, where the z coordinate equals 0.	76
33	Screenshot of a plane intersecting the data along the top, where the z coordinate equals 127.	77

34	Screenshot of a plane intersecting the data in the middle of the volume, with the y coordinate equal to 64.	78
35	An example of a plane that is perpendicular to the y axis.	78
36	Screenshot of a plane, perpendicular to the x axis, intersecting the middle of the data set with x coordinate equal to 64.	79
37	An example of a plane that is perpendicular to the x axis.	79
38	Screenshot of a plane that is not perpendicular to any axis, and passing through the centre of the data.	80
39	Electrons being freed by an electric field. The large dots are particles, and the small dots are electrons. This picture was taken from [14] . . .	82
40	Graphical representation of local fields to represent the global field. This picture is from a presentation given by Christian Jungreuthmayer at an HPCVL symposium at Carleton University, on November 15th, 2002.	83
41	Screenshots showing the first two iterations of the fire model. Green areas show voxels in the alive state, red in either burning1, burning2, or burning3 state, and blue in dead state.	88
42	Screenshots showing the third and fourth iterations of the fire model. Green areas show voxels in the alive state, red in either burning1, burning2, or burning3 state, and blue in dead state.	89
43	The speed-up of the fire model.	90
44	The average processor usage for different number of processors computing the fire model.	91
45	Time taken per processor for configurations of 2 to 8 processors.	92
46	Time taken per processor for 16 processors for fire model.	92
47	Half of the voxels on one of the eight processors are set to the burning state (shown in red). The other four processors map to the second plane of voxels.	93
48	Fire model run with a load imbalance on one of the processors.	94
49	Four of the eight processors have voxels in the burning state (shown in red). The other four processors map to the second plane of voxels.	95
50	Fire model run with a load imbalance on four of the processors.	95
51	An even distribution of voxels in the burning state (shown in red) among the eight processors. The other four processors map to the second plane of voxels.	96
52	Fire model run with an even load balance.	97
53	The speed-up of the particle model.	98
54	Average usage of processors for tests using varying number of processors.	99
55	Time taken per processor for configurations of 1 to 8 processors.	100
56	Time taken per processor for 16 processors for the particle model.	100

57	Time taken for fire model with the processor folding mapping scheme using different block sizes.	102
58	Time taken for fire model with the standard mapping scheme using different block sizes.	103
59	Comparison of communication and idle times of the fire model using standard mapping (on the left), and processor folding (on the right). . .	104
60	Time taken for generating slices using different numbers of processors.	105
61	Merge and idle times during image merging using 16 processors. . . .	107

1 Introduction

Volumetric modeling is used in a variety of disciplines such as computer graphics, mechanical design and medicine. Specific examples of modeling include simulating weather patterns, predicting lava flow in a volcanic eruption and simulating automobile collisions. Many of these models are constructed as a cellular automaton where the overall behaviour of the model arises from the computation of numerous small regions of data. The computation is localized which simplifies the design and implementation of a cellular automaton. There are many models that are based on three-dimensional cellular automata so it would be beneficial to create a generic three-dimensional cellular automaton framework. Such a framework could be used to in the creation of a modeling application thereby reducing the time and effort required to create such models.

One crucial aspect of computer modeling is that there is a large amount of data that must be computed or handled during the model's execution. This might be due to the accuracy level required by the model, or as a result of the sensing equipment that captures and generates a large amount of data. For example, the visible human project [32] contains images that are parallel slices of a human body taken at increments of one millimetre. There are 1871 pictures, each of which is 2048 by 1216 pixels, each made up of 24 bits of colour (8 red, 8 green, 8 blue). In total, the size of the data set is 13 GB. To iterate through each pixel of the data set would require several seconds. If a model requires a hundred iterations over the data, then a single computer could take several hours, or even days to compute some model.

Another limitation of computer modelling is that the model may also be complex. For example, Villasenor [31] describes a model of atomic particle interaction which has a relatively small amount of data, but must be modelled in very small time steps in order

to remain accurate. When using Villasenor's approach to model atomic particle interaction experiments, such as in [14], the model must replicate what happens in laboratory experiments over a certain timespan. Modelling for a period of time using small time steps requires the model to be iterated tens of thousands of times, and can take over a week to execute.

A parallel computer can be used in order to speed up the processing of large data sets and complex models. With a parallel computer, the time needed to run a program can be reduced by using more processors. However some programs see less and less of an increase in speed as more processors are used, so care must be taken in the design of the model so that it can take advantage of using a large number of processors.

This is of great benefit for models where the data is time sensitive, such as forest fire modelling. In forest fire modelling, a satellite image of an actual forest fire can be input as data for a fire spread model, in order to predict what effect different factors such as wind and rain will have on the spread of the fire. Obviously one would like to have the results of such a model as quickly as possible in order to be able to react in time. In these situations, a parallel computer could be used to compute the results as quickly as possible.

There have been many software libraries written to facilitate the development of programs. Some libraries have been designed to be generic so that they can be used in many various situations, such as the Standard Template Library[25] or NEMO [11], yet be efficient enough for high-performance applications. It would be helpful to use this approach to design a generic parallel three-dimensional cellular automaton to reduce the time for development of new applications. This thesis presents a three-dimensional MOdelling (3-MO) framework which addresses these goals.

3-MO allows for rapid application development by implementing a generic three-

dimensional cellular automaton on a beowulf cluster parallel computer. It is used to rapidly develop a model that takes advantage of parallel computing, yet abstracts the issues of parallel programming from the developer using the framework. It provides facilities for the developer to model initialize the model, implement the modelling logic, visualize the state of the model during computation, and post-processing facilities to handle the data that was computed, such as store the data, display an image, etc. All that the developer has to do is define the data types required for the model and implement the logic for the specific application. Applications that are created using the framework are fast, however, they may not be as fast as models that are specifically written and optimized for a given modelling application.

Chapter 2 presents background information for modelling and cellular automata, as well as data structures used in the framework for modelling three-dimensional data. It also introduces previous work in the area of modelling frameworks for parallel computers. Chapter 3 presents some preliminary information on the factors involved in parallel computing such as different parallel computer architectures, and different data mapping schemes. Chapter 4 presents the three-dimensional modelling framework that was designed and developed including: the architecture of the framework, specific functionality required for developing new models, as well as functionality for visualization of the data during execution of a model. Chapter 5 presents some three-dimensional models that were implemented using this framework as well as the performance analysis of these models. At the end of the section is a summary of 3-MO followed by a discussion of several directions for future research.

2 Background

This chapter presents background information on three-dimensional modeling. First, the cellular automata paradigm and different neighbourhood types used by automata are presented. Following that is an overview of data structures used by the framework to represent three-dimensional data. The chapter concludes by presenting some of the previous work that has been done in the area of parallel cellular automata systems and generic software frameworks.

2.1 Cellular Automata

Cellular automata were introduced by von Neumann in the 1950s to investigate self-reproduction. They are very effective in solving scientific problems because they capture the essential features of systems where a global behaviour arises from the collective effect of large numbers of locally interacting simple components [34]. One of the most famous cellular automata is the “game of life” developed by Conway [4].

Cellular automata are used to model many different systems in mathematics, physics, chemistry and biology [8, 24, 27]. It has also been found that these cellular automata can be easily parallelized [9].

A cellular automaton is characterized by the following properties:

1. They consist of a regular discrete lattice of cells.
2. The evolution takes place in discrete time steps.
3. Each cell is characterized by a state taken from a set of finite states.
4. Each cell evolves according to the same set of rules, which depends only on the state of the cell and a finite number of other cells, typically neighbours.
5. The neighbourhood relation is local and uniform throughout the model.

A cellular automaton is typically used in two dimensions, however the properties of cellular automata can apply to n dimensional lattices.

A classification of cellular automata models, as described in [10], is formally presented here. Let time be divided into a series of equal intervals, each associated with one iteration of a cellular automaton. Let t be the time index associated with a given iteration and δ be the interval between two successive iterations.

Let a portion of 2-D space be divided into a series of rectangular cells. Each cell can then be uniquely identified by a row index i and a column index j .

1. *Model I . The independent model:* $g_{i,j}(t+\delta)$ is a random variable not related to $g_{i,j}(t)$. This model appears to be of limited use in the context of volumetric modelling.
2. *Model II. The functional dependent model:* $g_{i,j}(t+\delta)=F(g_{i,j}(t))$, where F is a function. The value of a cell at time $(t+\delta)$ depends only on the value of the cell at time t . This model assumes total locality.
3. *Model III. The historic model:* $g_{i,j}(t+\delta)=F(g_{i,j}(t), g_{i,j}(t-\delta), g_{i,j}(t-2\delta), \dots, g_{i,j}(t-k\delta))$, where F is a function. The value of a cell depends on several previous values at that location. For example, if $\delta=1$ then the value of cell $g_{i,j}$ at time $t+1$ depends on the values the cell had during the last k time units. The model is based on local information only but takes into account the history of the cell.
4. *Model IV. The multivariate model:* $g_{i,j}(t+\delta)=F(u_{i,j}(t), v_{i,j}(t), w_{i,j}(t), \dots, z_{i,j}(t))$, where F is a function based on several attributes associated with the cell $g_{i,j}$ at time t .
5. *Model V. The geographical model:* $g_{i,j}(t+\delta)= F(g_{i,j}(t, Neighbourhood))$, where F is a function defined for cell $g_{i,j}$ at time t using attribute values present at all cells in the neighbourhood of $g_{i,j}$. In its most general form, the neighbourhood is arbitrarily defined. See Section 2.2 for different neighbourhood types.

In typical cellular automaton applications, for each time step the automaton iterates over all of the cells and applies a transition function to each one. This transition func-

tion computes the cell's next state based on the cell's current state and the state of the cells in its neighbourhood. The transition function and neighbourhood are specific to the application and must be defined by the programmer implementing the automaton. Cellular automata are ideally suited for parallel computation since the computations usually occur locally, and so the data can be divided and distributed among processors to be computed in parallel, with little communication required between the processors.

2.2 Neighbourhood Types

If c is a cell in the automaton then the neighbourhood of c , written as $N(c)$, is the set of cells in the automaton that are associated with c . For cellular automata, the type of neighbourhood used depends on the needs of the model. The cells are shaped like an axis-parallel rectangle, so different configurations of cells may be required to approximate a neighbourhood for a given application. For example, it is hard to approximate a circular neighbourhood using rectangles. The circle can be more closely approximated by adding more rectangles but it will never be a perfect circle. The level of accuracy of the neighbourhood depends on the individual model. Two common types of neighbourhoods are the von Neumann and Moore neighbourhoods[35].

2.2.1 Von Neumann Neighbourhood Model

In two dimensions, the von Neumann neighbourhood consists of 4 neighbours. They are those neighbours that share an edge with the cell. In three dimensions, the von Neumann neighbourhood comprises of the 6 voxels that share a face with a voxel. A voxel is a three-dimensional axis-parallel rectangular shape, which corresponds to a

cell in a two-dimensional cellular automaton. Figure 1 gives an illustration of the von Neumann neighbourhood in two and three dimensions.

Figure 1: The von Neumann neighbourhood in two dimensions (4 neighbours), and three dimensions (6 neighbours).

2.2.2 Moore Neighbourhood Model

The Moore neighbourhood consisting those neighbours that share a vertex with a cell. In two dimensions, the Moore neighbourhood consists of 8 cells. In three dimensions, it consists of 26 voxels: those who share a vertex with a voxel. Figure 2 shows the Moore neighbourhood in two and three dimensions.

Figure 2: The Moore neighbourhood in two dimensions (8 neighbours) and three dimensions (26 neighbours).

2.2.3 Neighbourhood Depth

The size of the neighbourhood can also be different depending on the model. The new value of a voxel might depend on the value of its nearest neighbour to the left, right, etc., or the value of the two nearest neighbours to the left, to the right, etc.. Let the depth of the neighbourhood be defined as the number of neighbours in each direction (left, right, up, down, etc.) that are required for the computation of the next value of a voxel. For example, if the new value of a voxel depends on its direct neighbours (those voxels that share an edge or corner with it), then the depth of the neighbourhood is 1. If it depends on the two neighbouring voxels to the left, right, etc., then the depth is 2, and so on. Figure 3 illustrates different neighbourhood depths.

Figure 3: Neighbourhood depth of 1 voxel on the left and 2 voxels on the right.

The neighbourhood depth is specific to the model and therefore is left as a parameter of 3-MO that must be specified by the programmer. This value defines the size of the neighbourhood buffer for the volumes.

2.3 Three-dimensional Building Blocks and Data Structures

2.3.1 Voxel

A voxel is defined as an axis-parallel rectangular region of three-dimensional space that is indivisible. It is the equivalent of a cell in two dimensions. A voxel is the atomic data element of 3-MO. It is used to represent the information about the model at a specific region of space. This information can be considered to be uniform throughout the voxel, or considered to be specific to a certain point within the voxel. Figure 4 shows an example of a small region of neighbouring voxels.

Figure 4: Neighbouring voxels.

Let us define v_{ijk} to be a voxel corresponding to a region of space at the coordinates (i, j, k) . The value of voxel v_{ijk} may be time dependent, so let $v_{ijk}(t)$ denote the value of the voxel at time t . As well, the neighbourhood of v is expressed as $N(v_{ijk})$.

2.3.2 Partition Block

A partition block is used as a container to hold all voxels within an axis-parallel three-dimensional rectangle. Partition blocks can be considered the basic work unit in 3-MO. The data set for the model is divided into blocks of voxels, and then the blocks get distributed among the processors.

All partition blocks within the model have the same size, which must be specified by the programmer. 3-MO has been designed so that the block size can be easily changed because it has an impact on the speed of computation of the model. The reader is referred to Section 3.2.2 for an in-depth discussion of the impact on block size on computation time.

When calculating the value of a voxel in successive iterations of the model, the new value depends on the current value and the values of its neighbours. These neighbours may be in the same block, or might be in another block on a different processor, as shown in Figure 5. Furthermore, for the sake of simplicity, we will assume that the neighbourhood of voxel is connected to that voxel, ie. if voxel a is the neighbourhood of voxel b , then there exists a line (not necessarily straight) from a to b which only intersects voxels that are also in the neighbourhood of b .

Figure 5: Neighbouring voxels that lie on different processors.

If the neighbouring voxels do not lie on the same processor, then their values must be transmitted to their neighbours across the network, which takes much more time than if it were stored on the same processor. As an optimization, if the neighbour of a voxel in the block lies on another processor then a copy of that neighbour is stored in the block, as shown in Figure 6. This way, the voxels are only transmitted to neighbouring blocks once, where they can be stored for quick retrieval.

Figure 6: Copies of Voxels Stored Locally on Neighbouring Volumes

2.4 Data Structures

3-MO uses several data structures for storing and representing three-dimensional data. The following section presents the two data structures used by 3-MO and the reasons for their inclusion. It also presents other possible data structures that could have been used, and the reason why they were not used.

2.4.1 Array

An array is simply a continuous area of computer memory. All other locations within this array are accessed by some index that is used to compute the offset from the beginning of the array. Although arrays are one-dimensional in the computer's memory, they can be used to represent a multi-dimensional data by using multi-dimensional mapping.

Arrays are well-suited for implementing the volume data structure since they are indexed, and the indices correspond conceptually to the discrete divisions between the voxels of the model. The indices allow for constant-time retrieval of data. As well,

there are no gaps, or holes, in the array, which conceptually corresponds to the partition block.

2.4.2 Octree

An octree [17] is a tree data structure for storing 3-dimensional data within a finite region of space. Each node in the tree represents a subsection of the region. This subsection is then partitioned into 8 smaller subsections for the children of the node, as shown in Figure 7. The region is divided into two parts along each of the x, y and z axes. Thus, each node has 8 children.

Figure 7: A region divided into eight equal sub-regions.

Octrees are well suited for representing data that are arranged in cells. If there are neighbouring regions which have similar data values, then these regions can be combined into a single, larger cell. In this way, octrees can be used to decrease the amount of space needed to store the data. Octrees can also be used to efficiently represent regions where there are gaps in the data, since gaps can be considered as regions of space containing nothing, and thus combined. However, if the tree is balanced, then retrieving

the value of a given cell takes $O(\log n)$, where n is the number of nodes in the octree since there must be a search done on the tree. If the tree is unbalanced, then a search takes $O(n)$ in the worst case.

If there is no consecutive repetition between neighbouring voxels, then using an octree can actually use more space to store the data than using an array, since few of the cells can be combined, and the octree needs internal nodes to maintain the octree structure.

Since we assume that the entire data set of the model will be divided into partition blocks to be distributed among the processors, each processor will have only a fraction of the total data set. For a parallel machine with p processors, assuming that the volumes are evenly distributed, then each processor should only have $1/p$ of the total number of blocks. An octree can efficiently store the partition blocks on a processor since there will be large gaps in the data set. Also, if there is an even distribution of the volumes, then the octree that stores them will be balanced, in which case the time for retrieving volumes from the octree is $O(\log n)$. For this reason, the octree is a good data structure for storing and retrieving partition blocks on a given processor.

2.4.3 Other spatial data structures not used in the framework

There exist many other data structures for spatial data. They each have their own advantages and disadvantages, however the data structures already presented are sufficient for use in the framework. This section presents briefly presents other data structures, and why they are not used in 3-MO.

2.4.3.1 K-d Trees K-d trees are a generalization of the binary search tree to k dimensions, where k is the dimensionality of the stored objects. When inserting an element, the inserted element is compared to elements in the tree according to a specific coordinate. If a node is at level i in the k -d tree, then the element at that node is compared to according to its $(i \text{ modulo } k)^{\text{th}}$ coordinate. For simplicity, we will assume that k is constant for a given tree, i.e. that all elements within a k -d tree have the same dimensionality.

For example, if the point p (3, 7, 6) is inserted into the tree, the algorithm first compares the point p to the point stored at the root,. If the tree is empty, then p becomes the element stored at the root. Otherwise, since the root is at level 0, the points are compared by their first coordinate. If the root's point is (5, 3, 9), then the 3 from p is compared to the 5 of the root's point. Since 3 is less than 5, p is recursively inserted into the root's left subtree.

The spatial divisions created by the nodes of a k-d tree are based on the elements stored in the tree. The divisions define axis-parallel subspaces which separate all elements in the node's left and right subtrees. Since the tree is a binary tree whose splits are determined by the order of insertion of the data, it is possible that the points are inserted in sorted order, in which case the tree will have the shape of a linked list. Therefore, the time for insertion and retrieval is $O(n)$. If the tree is balanced, then the time for insertion and retrieval is $O(\log(n))$.

K-d trees are well suited for range queries, where the problem is to retrieve all elements in a set whose data are within a certain range. In one dimension, one possible query is to find all elements that are greater than N_1 , and smaller than N_2 . In the case of one dimension, the K-d tree is a normal binary search tree. In two dimensions, the query would be to find all elements that are within a given rectangle. In three dimensions, to

find all elements within a given box, etc. 3-MO does not perform any range queries, so the extra features of the k-d tree are not needed and is therefore not used. Also, the worst case time for insertion and retrieval is $O(\log(n))$, which is slower than the octree.

2.4.3.2 Binary Space Partition (BSP) trees Binary spatial partition trees are similar to K -d trees, except that the elements are not necessarily divided by axis-parallel subspaces. The root of a tree has a hyperplane associated with it (in two dimensions the hyperplane is a line, in three dimensions a plane). The elements in the left and right subtrees are on different sides of the hyperplane at a given node. There is no specific way to choose the dividing hyperplanes, but they can be computed based on the elements to be stored in the tree. This requires that the algorithm have prior knowledge of the elements to be inserted, so that the hyperplanes can be chosen so that they do not intersect the elements. The hyperplanes can also be chosen so that they produce an even split of the elements, thereby giving a balanced binary tree.

For example: in computer games, BSP trees are used for scene management. Objects such as houses, trees, rocks, etc. are created by an artist and then placed in the world. When all of the elements are placed within the world, then the BSP tree is computed.

BSP trees are well suited for culling, or removing, objects from consideration by an algorithm. The framework does not need culling for the purposes of modelling so BSP trees are not used. The BSP trees could be used to store the voxels, but they have a retrieval time of $O(\log(n))$, which is the same as the octree. However, octrees are simpler to implement since the dividing planes are always axis-parallel.

2.4.3.3 Other data structures There are other data structures used for spatial data that are not used in this thesis. These include EXCELL, GRID files, hierarchical grid files, and BANG files. For a discussion of these data structures, the user is referred to [20].

2.5 Parallel Cellular Automata Systems

There has been some work done to develop software frameworks for writing parallel cellular automata that abstract issues of parallelization from the user. These frameworks are all designed for two-dimensional cellular automata, and they differ in operations they can be performed on cellular automata, and how a developer uses the framework to build an application.

2.5.1 Camel

In [8] and [27], the authors describe CAMEL (Cellular Automata environMent for systems modeling). The Camel system is a software framework for developing two-dimensional cellular automata for parallel computers where the processors are organized in a grid-type layout. The system is meant to abstract the details of parallel programming from the user. The user has only to define the cell's data members, the transition function, and the neighbourhood used by the automaton. The neighbours are specified by the offset of each neighbour with respect to a given cell.

These parameters are specified in another language called CARPET (CelluLAR Programming EnvironmenT), which is a high-level language based on the C language with additional constructs to describe the rules of the transition function. The programming

environment parses the CARPET instructions and produces *C* language code with pre-processor directives for configuring the framework.

This system is designed for two-dimensional cellular automata and for image processing, and has a load-balancing scheme to ensure that there is not a uneven workload among the processors.

2.5.2 Trend

The Trend [3] programming environment is a set of tools that allows a user to define data types, modelling logic, neighbourhood types, etc. There is an editor where a user can use a graphical user interface to specify the data types used for each cell and the neighbourhood type. There is also a text editor where the programmer can input the transition function by using the Trend language. This language is a C-type programming language designed specifically for use in cellular automata. There is also a simulator window where the user can run the simulation, and view the state during its execution. The user can pause the simulation at any time, and even backtrack through the time steps. However, the Trend environment was designed for the simulation of rule sets to model artificial life and self-replication. It is not designed for numeric computation as in fluid dynamics, nor is it designed to run on a parallel machine.

2.5.3 CAPE

CAPE [21] is a cellular automaton environment implemented on a parallel computer. It is designed to simulate a two-dimensional cellular automaton. It is designed to be generic so that it can be used for many different modelling applications, and provides

a generic GUI to view the current state of the model. The GUI displays any model by querying the state of each cell from a database. When writing an application, the program must communicate with the database to maintain the current state of the model. CAPE is implemented in FORTRAN, and does not extend the language by adding additional language constructs. One thing missing from CAPE is that it lacks a load balancing strategy. This can lead to an inefficient use of parallelism if there are regions of the data that are not active in the simulation. Also, CAPE is designed only for two-dimensional cellular automata applications.

2.5.4 NEMO

In [11], the authors describe NEMO (NEighbourhood MOdeling). It is a software framework that is meant to simplify the process of programming parallel cellular automata and other kinds of processing. The framework abstracts issues of communication, synchronization and load balancing so that programmers who are not familiar with parallel programming can write an application that takes advantage of parallelism. It allows for modeling two-dimensional cellular automata, neighbourhood analysis (for example: image processing) and propagation modelling, all within the same framework. In order to develop such applications, the developer just has to implement a few functions needed by the framework.

Neighbourhood analysis involves executing one, or a series of distinct, single pass neighbourhood functions on a raster. Propagation modelling is designed to process applications which operate on the active border principle, in which only a subset of cells need to be processed at a given time. Although propagation can be emulated with cellular automata and vice versa, it is better to separate the two types to increase

performance.

NEMO uses a two-dimensional mapping scheme, called two-dimensional folding, as a load-balancing strategy. This strategy partitions the data evenly among the processors, with an optimization where certain neighbouring data elements are mapped to the same processor. This speeds up the computation of a model since there is a decreased amount of communication among processors. For an in-depth discussion of two-dimensional folding, the reader is referred to Section 3.3.

In order to write an application that uses NEMO, the programmer must implement five functions:

1. **Modelling.** The modelling function is what implements the modelling logic. It computes the value of a given cell depending on the cell's current value and the values of those cells in the neighbourhood.
2. **Initialization.** This function initializes the parameters needed by the model, such as neighbourhood size, rasters needed, neighbourhood function, cell and time comparison functions, and when to stop the simulation.
3. **Simulation time comparison.** This function is required by the propagation driver to impose an ordering on the cells so that they can be placed in a priority queue. The cells are processed by removing them from the priority queue, applying the modelling function, and then re-inserting them in the queue.
4. **Cell comparison.** This function is required by the propagation driver to compare two copies of the same cell to determine which has the minimum time. If a cell *A* is a neighbour of two cells which are on different processors that are processed, then those processors will each have a copy of *A*. The application must have a method of determining which copy of *A* has the minimum time, and therefore should be kept.
5. **Termination.** This function is used by the propagation driver when the application has finished. It is responsible for releasing any resources used, closing files, etc.

For more information on NEMO, the reader is referred to [22].

The approach used by NEMO is an elegant one for abstracting the issues of parallelism from the programmer. It also allows a programmer to quickly develop parallel applications since it addresses issues that are common to many parallel applications, so all that is left to be written are the functions that are specific to a given model. These functions are not meant to use any aspect of parallelism, such as communication, so they can be written by someone who is not familiar with parallel programming. As well, NEMO does not use language extensions, as opposed to CAMEL and Trend, so the programmer does not have to learn any new language constructs. NEMO also has a built-in load balancing scheme, which is not present in CAPE. Due to NEMO's simplicity, this approach is also used and extended by 3-MO.

2.5.5 Summary

None of these systems are designed to model cellular automata in three dimensions. As well, some of the systems are based on the *C* programming language, but extend the language by adding special constructs for modelling the cellular automaton. With these extended languages come specific programming tools to develop the automata using the extended language. This approach requires extra time for the developer to learn the extensions and get acquainted with the development tools.

2.6 Generic Programming

One of the goals of this framework is to simplify the development of parallel cellular automata, while abstracting the issues involved with programming parallel applications. The framework must also be generic so that it can be used to implement a wide variety

of modelling applications, yet be written efficiently so that the program runs as fast as possible.

In other parallel cellular automata frameworks such as CAMEL [8], the framework is implemented using the *C* programming language, but extends the language to add some automaton-specific constructs to describe the particular model. Other frameworks, such as Trend [3], have a completely new language to describe the setup of the cellular automata framework. This approach needs a new compiler specifically for the Trend language. This allows for the framework's implementation to be completely separate from the automaton's implementation, however it requires that the programmer learn the Trend language.

This framework was implemented in C++ because it enables software to be written in a generic fashion and be compiled into machine language for fast execution on a computer. To tailor the framework to a specific model, a programmer just has to implement the modelling logic, and define the data types using C++ code. The framework is then compiled with the modelling code to produce the actual program. The programmer does not have to learn any extra language constructs, he or she can program the model-specific code using standard syntax, as if it is going to be run on a traditional desktop computer with a single processor.

There are several ways of writing generic code in C++: either with virtual functions or templates. If a variable is declared as a pointer to a certain object type, then that variable can also be assigned pointers to subclasses of that object. The subclass inherits the functions and members of the parent class, however when calling those functions, the functions in the parent class will be invoked. To ensure that the functions of the subclass are invoked, the function must be declared as virtual. This causes the compiler to add a virtual function table to the objects to keep track of which class' function to

call. When a virtual function is called, the program looks at the virtual function table and then calls the appropriate function. This table look-up slows down the performance of the framework, as well as adds to the data size of the object.

The C++ language has templates which also enable code to be written in a generic fashion. Using templates, code is written for an abstract data type, that is not directly specified. In order to use templated code, the programmer must pass in the data type for the code as a parameter when declaring a variable. The compiler can then link the appropriate function, thereby eliminating the need for a look-up into a table. Using this approach, software can be written for a generic data type, yet be compiled and linked as normal code, without the need for an extra step of looking for the appropriate function in a table.

There have been several generic libraries developed in C++ that use templates. The Standard Template Library [25], Boost [2], and Blitz [30] are good examples of what can be achieved using templates. They demonstrate that by using templates, software can be written in a generic manner, without having a decrease in performance by using virtual functions and virtual function tables.

3 Parallel Computing Preliminary

This chapter presents the reader with information and concepts that are relevant to parallel three-dimensional cellular automata. This section begins with concepts of parallel computing models and architectures. Following that is an introduction of different factors that affect the performance of parallel programs, as well as different mapping schemes for the data, and their impact on the performance. This chapter concludes by discussing techniques for visualizing three-dimensional data.

3.1 Parallel Computing Models, Architectures and Topologies

This section presents different parallel models and their properties. It also presents different types of interconnection networks for a parallel machine. Cellular automata have characteristics that are similar to one type of parallel computer architecture, however the parallel computer used in the implementation of 3-MO has a different parallel architecture. It is important to understand the differences in these architectures, and how that impacts the design of 3-MO. The first models of computation are shared memory systems, followed by distributed memory models. For descriptions of parallel computation models, this thesis uses [22] as a source of information.

3.1.1 Theoretical Parallel Computing Models

Sequential machines with a single processor are based on the random access machine (RAM) model of computation. The RAM model of computation consists of a processor and local memory, where the cost of accessing any location within that memory is the same, which takes constant time. In parallel computing, there are four variants of the

parallel random access machine (PRAM) model of computation. The PRAM model of computation is purely a theoretical model used for algorithmic analysis. The PRAM model allows for an unlimited number of processors that share a common memory space for storing and retrieving data, as shown in Figure 8. In this common memory space, it takes an equal amount of time for any processor to access any location in the memory. The four variants of this model of computation are the different combinations of how processors can read to, and write from, memory at any cycle of computation.

Figure 8: Parallel Processors Sharing a Common Memory Space

- Exclusive Read Exclusive Write (EREW) - This variant allows only one processor to read from, and write to, a given memory location.
- Concurrent Read Exclusive Write (CREW) - This variant allows only one processor to write to a memory location, but any number of processors can read from it.

- Exclusive Read Concurrent Write (ERCW) - This variant allows only one processor to read from a memory location, but any number of processors can write to it.
- Concurrent Read Concurrent Write (CRCW) - This variant allows any number of processors to read from, and write to, a given memory location.

There are several different models of parallel computation where processors do not have a global memory space, but rather each processor has its own local memory.

3.1.2 Bulk Synchronous Parallelism (BSP) model.

The BSP model is a distributed network model where each processor has local memory that is separate from other processor's memory, as shown in Figure 9. In other words, it uses a distributed memory model. Processors interact with each other by sending and receiving messages over the network.

Figure 9: Interconnected processors that have local memory.

The processors perform local computations and then are synchronized for exchanging messages with each other. This model uses three parameters to measure the complexity and efficiency of the algorithms:

1. The number of processors, P .
2. The time it takes to synchronize the processors, L .
3. The ratio between the local computation and communication, G

The complexity of an algorithm is the total time it take to execute all three steps.

3.1.3 Coarse Grained Multicomputer (CGM) model.

The CGM model of computation proposes a generic multi-computer $M_\alpha(m, n)$, which has m processors with data size n , and the following properties:

1. $M_\alpha(m, n)$ has P processors, $p_1, \dots, p_m$, connected by some arbitrary interconnection network or bus system.
2. Every processor, p_i has $\Theta(n/p)$ local memory.
3. $n/p \geq p^\alpha$, for some small, but fixed, constant $\alpha > 0$
4. The total data size, n , is a large number where m , the number of processors, is not necessarily large.

The term “Coarse Grained” refers to the fact that we assume that the size of $O(n/p)$ is considerably larger than $O(1)$. The definition of considerably larger is $(n/p) > p$. All algorithms consist of alternating local computation and global communication rounds. Each communication round consists of a single h -relation with $h=O(n/p)$, i.e. each processor sends $O(n/p)$ data and receives $O(n/p)$ data. It is required that all information

sent from a given processor to another processor in one communication round is packed into one message.

The CGM model combines local and global computation and communication. The model assumes that several global communication schemes are available and global functions are available. The complexity of the algorithms developed using the CGM is measured by the number of these global steps as well as the complexity of the local computation.

3.1.4 Parallel Architectures and Topologies

There are several different classes of parallel computers. Typically, they are classed according to how they process instructions and data. These two different factors gives rise to four different combinations:

1. Single Instruction Single Data (SISD). This class of computers is typical of personal computers. There is one processor that works on one datum at a time.
2. Single Instruction Multiple Data (SIMD). This class of computer has all processors executing the same instruction on different data at the same time.
3. Multiple Instruction Single Data (MISD). This class of computer has all processors executing their own instructions on the same data element.
4. Multiple Instruction Multiple Data (MIMD). This class of computer has all processors executing their own instructions on their own separate data elements.

The MIMD architecture has become the most common architecture for parallel computers. There are also several topologies possible for parallel computers. What separates the topologies is how the processors are connected to each other. The following are some of the most common topologies:

1. Ring and Linear Array. This class of topology has p processors, where one of the processors is only connected to two other processors. In a linear array, the processor i is connected to processor $i-1$, and $i+1$, except for processors 0, and p , which are only connected to processor 1, and $p-1$, respectively. A ring topology is similar to a linear array, except processor 0 and p are also connected, as shown in Figure 10.

LINEAR ARRAY

RING

Figure 10: The linear array and ring processor topologies.

2. **Mesh and Torus.** This class of topology has p processors arranged in a grid of $m \times n$ processors, where $m \times n = p$. In a two-dimensional mesh, each processor p is connected to four other processors: to the left, right, top and bottom. In other words, the rows and columns are linear arrays. Those processors which lie on the boundary of the processor grid have less connectivity than those processors which lie inside the boundary. A torus is similar to a mesh, except processors which lie on the boundary of the mesh are connected to processors on the opposite boundary of the mesh. In other words, the connections “wrap” around to the other side of the mesh. An illustration of both architectures are shown in Figure 11. The mesh architecture corresponds to the architecture of a two-dimensional cellular automaton. The processors have a local computation, and each processor can communicate with its neighbours.

Figure 11: The mesh and torus processor topologies.

3. Hypercube. This class of topology has p processors, where p is a power of 2 ($p = 2^d$), and d is the dimension of the hypercube. Each processor is connected to d other processors, which are determined based on their id number. The d neighbours of processor i are those processors whose id number differ from i by exactly one bit. For example, if there are 8 processors, then the neighbours of processor 1 (or 0001 in binary) are: 0000, 0011, 0101, or 0, 2, and 5. Figure 12 shows a picture of the hypercube topology.

Figure 12: The hypercube processor topology.

4. Tree. This class of topology has p processors, which form a tree with d levels. The levels are numbered from 0 to $d-1$. A processor at level i is connected to its parent at level $i-1$, and to its children at level $i+1$ by bi-directional links. The only exceptions are the root, which has no parent, and the leaf nodes, which have no children. Figure 13 shows a picture of the tree topology.

Figure 13: The tree processor topology.

5. **Cluster.** This class of topology has p processors, which are all interconnected by a network router. This network is typically an Ethernet network running at 100 or 1000 Mb/s. The processors can communicate with any other processor in the cluster. One processor can communicate with any another processor to which it is connected in the network switch. However, processors can only receive messages from a single processor at a time. For example: processors 0 and 1 can communicate at the same time as processors 2 and 3, but processors 0 and 1 cannot both communicate with processor 2 at the same time. Depending on the number of processors, there may be a network switch hierarchy for inter-processor communication, and therefore messages between processors on different network switches would travel through the network link between the two switches. This means that processors on different switches could not all communicate at the same time. Figure 14 shows an example of the cluster architecture.

Figure 14: The cluster processor topology.

3.2 Factors Affecting Parallel Performance

3-MO was designed to facilitate the implementation of three-dimensional cellular automata. One goal of the design and implementation was to make 3-MO as efficient as possible. There are several factors that affect the efficiency of parallel cellular automata, which are described in the following subsection.

3.2.1 Algorithmic Factors

Since 3-MO is designed in a generic fashion, it does not implement any specific modelling logic. When a programmer wishes to use 3-MO to implement a model, the algorithm used to compute the specific model has an impact on the runtime. The efficiency of the algorithm cannot be taken into account in this design of 3-MO, since the algorithm that will be used is unknown.

3.2.2 Communication Versus Computation

Computing the value of a voxel for the next time step depends on the current value of the voxel and the value of its neighbours. If a voxel lies on the boundary of a block, but not on the boundary of the entire data set, then a neighbouring voxel may reside in a block that is processed by another processor.

The neighbouring voxel of v must be retrieved prior to computing the next value of v . Communicating with other processors is slower than reading from a processor's local memory so every time that a voxel must be communicated, it increases the time taken to run the model. To minimize this communication time, the values of neighbouring voxels are transmitted once and stored locally by the volume for faster retrieval.

There exists also a trade-off between communication time and computation time when choosing the size of the blocks in the model [7]. Using a smaller block size gives a finer distribution of the data, but increases the number of voxels that must be transmitted during updating. The time required to transmit the neighbouring voxel data to other processors depends on the surface area of the block, which can be expressed as:

$$(2 \times x \times y) + (2 \times y \times z) + (2 \times x \times z),$$

where x , y , and z are the number of voxels in the corresponding dimensions. This assumes that the model is using the van Neumann neighbourhood. There are 6 sides to a cube and rectangloid, so each term represents one side. However, the opposite sides have the same size, which is why there are 3 terms, each multiplied by two. The Moore neighbourhood takes slightly more time to transmit, where the number of voxels that are transmitted can be expressed as:

$$(2 \times x \times y) + (2 \times y \times z) + (2 \times x \times z) + (4 \times x) + (4 \times y) + (4 \times z) + 8$$

The first three terms are the voxels which lie on the outer border of the volume (same as the previous expression), the middle three terms are the edges of the volume, and the last term is the 8 corners of the volume.

The amount of time needed for computation depends on the number of voxels in the volume data structure, and the computational complexity of the modelling algorithm. The number of voxels can be expressed as:

$$(x \times y \times z),$$

but the computational complexity of the modelling algorithm is unknown, since the algorithm is unknown. The computational complexity of the algorithm will be denoted by C , and depends on the amount of data input to the algorithm, which is $(x \times y \times z)$. Therefore, the complexity is denoted as $C(x, y, z)$.

There is also a set-up cost for sending data across the network. For instance, the TCP/IP protocol sends data in packets across the network. Each packet has a header which contains information about the packet. This header requires a certain amount of time to be transmitted along with the data contained within the packet. This set up time is denoted as S .

For a sequence of communication followed by modeling, the time needed is:

$$S + T \times ((2 \times x \times y) + (2 \times y \times z) + (2 \times x \times z)) + C(x, y, z).$$

where T is the time to transmit a single voxel. If the block size is large, then the time for communication is small. However, there might be an imbalance in the workload of the processors, and the processors will be used inefficiently. Conversely, if the block size is small, then there will be a greater time for communication.

If V is the total number of blocks in the model, i.e. the entire data set, then the number of voxels that lie on the face of all blocks is $V \times 2 \times ((x \times y) + (x \times z) + (y \times z))$. So as the number of blocks increases, so does the number of voxels that are stored locally by neighbouring blocks. This also means that there will be an increase in the time needed to update neighbouring blocks, since more voxels are transmitted. However, the number of voxels that are used in the computation remains the same so the time for computation remains the same.

Herein lies the problem of choosing the correct size for the partition blocks. To minimize the time for modeling, one should choose as large a block size as possible so

the ratio of volume to surface area is minimized. This means that the amount of time spent communicating data between processors is minimized. On the other hand, this can increase the imbalance in the computational workload among the processors.

It is important to find the balance where the time taken for communication balances out the difference in computation time caused by an imbalance in workload among processors. It would be valuable to be able to compute this point, however since the 3-MO is designed to be generic, the running time of the algorithms that will be used in 3-MO is unknown. For this reason, the size of the partition block is one of the parameters of 3-MO and must be set by the programmer. Through trial and error, the programmer can try different block sizes until the optimal speed is obtained.

3.2.3 Data Related Factors

In parallel applications, it is important to have a good distribution of the data. To use parallelism efficiently, the processors should have an equal workload, so that they finish at the same time, as opposed to some processors finishing early and having to wait for other processors to complete their computations. For this reason, the data should be distributed in such a way that all the processors finish computing at roughly the same time. For some applications, this can be done by having an even distribution of the data.

In other applications, some data can take longer to compute than others. For example, in modeling the flow of lava down a mountain, if there is no lava in a certain portion of terrain, there is no flow to compute. In this situation, it is best to distribute the portions of terrain which have lava flowing on them among the processors so that the computation is balanced. This distribution of the data is decided by the mapping scheme (see Section 3.3).

3.2.4 Implementation Related Factors

There are many design decisions involved in implementing a software framework. First of all, one must decide which language to use for writing a parallel application. There are many languages that are suitable for writing parallel applications, including Fortran, C/C++ and Java. Even within a language such as C++, there are different compilers available that produce code that runs faster than that produced by another compiler. For Fortran, only GCC is freely available but it only supports Fortran77 syntax. Java has many freely available tools for development, however the Java language is typically much slower than Fortran and C++. It seems like C++ offers the most flexibility as to availability of tools, speed of code produced, as well as ease of development, which is why it was chosen as the language of implementation for the framework.

Another implementation factor is that there are several ways in C++ of creating generic data structures and algorithms, with some being faster than others. Using templates in C++ should produce faster code than using pointers and virtual functions, since functions in templated classes are statically linked whereas virtual functions are accessed through a look-up table. For this reason, templates are used whenever possible in the implementation.

3.3 Mapping Schemes

In order to take advantage of parallelism, the data should be distributed among the processors so that the computation among the processors is even. How the data gets distributed among the processors depends on the mapping scheme. The cellular automata behaves like a mesh architecture, whereas 3-MO runs on a cluster computer. In the mesh architecture, a processor can only communicate with the processors with which it has

a communication link. In a cluster computer, any processor can communicate with any other processor. The mapping scheme imposes a logical mesh topology on the cluster so that the cluster behaves like a mesh.

In propagation modelling (see Section 2.5.4), there is only a small portion of the model that needs to be computed in the first few time steps. For example, the diffusion of a chemical or gas into the air starts at a specific location and moves outward, or sound waves travelling outward from a speaker. For the first few generations, only regions near the point of origin will change their values. If this active region lies completely within one of the volumes, then there will be an uneven workload among the processors, and there will be a decreased benefit from using parallelism. It would be best to have all the volumes that are part of this active region to be evenly distributed among the processors. This maintains the workload balance, and gains the most from the advantages of parallelism.

In three dimensional models, the shape of this active region depends on the model. For example, if the model is computing the flow of heat rising from a fire, then the active region is the hot air rising in a cylindrical shape above the source of the fire. If the model is computing the air turbulence caused by a bullet moving at high velocity, then the active region will be a cone-shape, since there is a shock wave caused by the bullet moving at high speed through the air. If the model is computing the explosion of a cluster of ionized particles under radiation from a laser, as in [13, 14], then the active region is sphere since each particle has an electric field that pushes against the other particles, thus moving the particles out wards from the cluster.

A mapping scheme is needed to introduce the notion of neighbours between the processors in a cluster parallel computer. This is because all processors are connected to a switch. This is helpful since a cellular automaton has the notion of neighbours,

and neighbourhoods. In the mapping scheme, the processors of the cluster are logically mapped to a three-dimensional grid computer architecture.

To achieve this mapping, the processors are logically organized in a lattice configuration. This lattice has a set width, height and depth, the product of which is less than or equal to the number of processors present in the machine. The processors are then arranged in the lattice in order of their id numbers, first along the x-axis, then y-axis and finally the z-axis, as shown in Figure 15.

Figure 15: A three-dimensional lattice of 12 processors, ordered by their id number.

3.3.1 Standard Mapping

The standard mapping scheme uses the dimensions of the volume, the dimensions of the data set and maps each volume's three-dimensional position within the data to a specific processor id. They are mapped to processors according to each volume's offset in the x,

y, and z directions. Figures 16 and 17 illustrate standard mapping in 2 and 3 dimensions, respectively.

If T_x , T_y and T_z are the a block's top left offsets in the x, y, and z directions, and D_x , D_y and D_z are the volume sizes in the x, y, and z dimensions, then the mapping is as follows:

$$i = \frac{T_x}{D_x}, \quad j = \frac{T_y}{D_y}, \quad k = \frac{T_z}{D_z}$$

The mappings are then used to look-up the processor id in the lattice in the lattice as follows:

$$Px = i \times \text{mod}(Lx), \quad Py = j \times \text{mod}(Ly), \quad Pz = k \times \text{mod}(Lz),$$

where Lx , Ly , and Lz are the dimensions of the processor lattice in the x, y, and z-dimensions, respectively. The id of the processor is then located at (Px, Py, Pz) in the processor lattice.

Figure 16: Diagram of standard mapping in 2 dimensions.

Figure 17: Diagram of standard mapping in 3 dimensions.

The benefit of this approach is that it is simple to implement since it produces a cyclical mapping of the blocks. It also produces an even distribution of the volumes so that, as long as the dimensions of the processor lattice are greater than 1, no two neighbouring volumes will map to the same processor. Although this approach depends first on the x dimension, and then y and z, any approach that produces a cyclical mapping of this nature will have the same characteristics.

3.3.2 Two-Dimensional Folding

Another mapping scheme, called two-dimensional folding [11], is similar to the idea of cyclical mapping, except that the order of processors in the mapping is reversed for neighbouring processor lattices. In other words, the volumes at the origin, with offset (0,

0, 0) use the same processor lattice as in standard mapping, but the neighbouring lattices are mirrored along the common edge. Figure 18 shows how a 2×2 processor lattice is mirrored according to two-dimensional folding, and Figure 19 shows three-dimensional folding.

Figure 18: Processor folding in 2 dimensions.

Figure 19: Processor folding in 3 dimensions.

This approach produces an even distribution of volumes, as with cyclical mapping, but it also ensures that neighbouring blocks get mapped to neighbouring processors, as defined by the mapping scheme. It also adds an optimization that if a processor is on a side of the processor lattice, then any block that maps to it will also have a neighbour of that block map to the same processor. In Figure 18, all processors lie on the face of the processor lattice, so all blocks have a neighbour mapped to the same processor. This reduces the communication time since these neighbouring blocks can be updated without communicating over the network.

3.4 Visualization

During the computation of the model, one may wish to know the current state of the model. This may be done either by inspecting the actual numerical values of each voxel, or by showing a visual representation. This thesis describes two common methods of visualizing voxels in three-dimensional models.

3.4.1 Slice Generation

If one would like to view the state of a voxel which lies within the data set, one could either query the specific values or state of the voxel, or generate a two-dimensional slice that intersects the desired voxel. This approach is similar to the notion of cutting an apple into two halves to see the core of the apple. The reader is referred to Figure 20 for an illustration of slice generation.

To give a sense of navigating through the data, the user should be able to view a slice of any orientation, either completely horizontal, completely vertical, or at some angle specified by the user. For an example of a program that displays three-dimensional data by generating an arbitrary slice, the reader is referred to the Visible Human viewer [33].

Figure 20: Generating a two-dimensional slice from three-dimensional data.

Assuming that the data is distributed among the processors, determining which voxels intersect an arbitrary plane can be computed in parallel. Each processor computes the intersection of its local blocks with the plane, resulting in a partial image. Each processor's image is then combined by the root processor to produce the complete image. This is the approach that is implemented by the 3-MO (see Section 4.4.4 for more information and Figure 27 for an illustration). The processors' images are computed in parallel and also combined in parallel to form the complete image. This approach takes full advantage of parallelism.

3.4.2 Isosurfaces

An isosurface is a surface generated by projecting parallel rays into the three-dimensional data from the user's point of view. Parallel rays are cast through every pixel in the image, and stop when they first intersect a voxel that satisfies some stopping condition, such as element type, colour, density, etc. The word isosurface itself means "equal surface", so the surface generated is made up of voxels that all equal in some way, such as their element type is wood, or their colour is green, or their density is greater than 5, etc. By putting together the intersecting voxels, the outer surface of the model can be generated.

This approach produces good results for viewing three dimensional objects, but it does not show anything about the data within the outer layers. 3-MO does not implement isosurface visualization since it was decided that its two-dimensional slice generation gives a visualization of the data throughout the data set, and not just the outer layer. For more information on systems that implement isosurface generation in parallel, the reader is referred to [16].

4 Three Dimensional Modelling Framework Architecture

The following section presents the architecture of 3-MO. It introduces the main components, as well as how to configure 3-MO for use in a model and what functions must be provided by the programmer. This section also defines the scope of applicability of 3-MO, as in which models are good candidates for using 3-MO. It also explains the flow of execution that these programs follow, as well as features available in each phase of the program flow.

4.1 User-defined Components

A programmer must use the object-oriented features of C++ to define two classes that will represent the objects needed by the model: the Datum class and Modelling class. The Datum class is responsible for storing the data that represents the state of the model for a given voxel, such as temperature, magnetic field strength, etc.

To be used by 3-MO, the Datum class must include the following three functions:

1. `getNullElement()`
2. `initialize()`
3. `interpolateValue()`

The `getNullElement()` function is a static function that returns a datum object that is used only for combining images. The object returned by this function is used for comparison against other voxels to determine if a voxel is a placeholder or not.

The `initialize()` function is a member function that takes a three-dimensional coordinate as a parameter. The function is used as a sort of constructor, used to initialize a voxel to some model-specific value. For more on voxel initialization, the is referred to Section 4.4.1.

The `interpolateValue` function takes an array of 8 Datum values and a 3-d point. The 3-d point has x, y, and z values that are reals, and not integers. There are at most 8 possible combinations of floor and ceiling of the x, y, and z values. The array of 8 Datum values are the Datum objects located at these 8 possible combinations. The purpose of this function is for the developer to implement some sort of interpolation function for a given 3-d point in space. This can be nearest neighbour, trilinear interpolation, etc. This function returns the interpolated Datum object.

The second class that the developer must be implemented is the Modelling class. This class implements the modelling functionality and must include the following two methods:

1. `model()`
2. `finalize()`.

The `model()` method takes a block object and iteration number as parameters. It is meant to implement the modelling logic for a block of voxels in a given time step. Since the block stores its neighbouring voxels, all of the data needed to compute the next generation of values for the voxels is available to the modelling function. The model function can also store intermediate data of the model to disk. The `finalize` function takes only a block object as a parameter, and is meant to implement any functionality that might be needed at the end of the model such as computing some final values, or saving data to disk, etc.

4.2 Model Configuration

3-MO was designed in a generic fashion so that it can be used for any model that operates like a three-dimensional cellular automaton. To implement a specific model, there are several parameters that must be configured. These parameters are all defined within a Configuration class must specify the following parameters:

1. The size of the data set. The developer specifies the total number of voxels in the x, y, and z directions.
2. The size of the blocks. These parameter specify the size of the blocks in the x, y and z directions. The reader is referred to Section 3.2.2 for a discussion on the impact of the block size on the model performance.
3. The dimensions of the processor lattice. These parameters are needed for mapping the blocks to processors.
4. The type of neighbourhood used in the model. This can be either the van Neumann or Moore neighbourhood.
5. The names of the Modeller and Datum classes. These parameters are needed by the C++ compiler to compile and link the appropriate object types for 3-MO. The program is statically linked since there was no apparent need to use dynamic linking, although it should be possible to do so.
6. The mapping scheme that is used. This can be either processor folding or cyclical mapping. For more information on mapping schemes, the reader is referred to Section 3.3.

By specifying these specific parameters, 3-MO can be fine-tuned for any specific modelling application.

4.3 Principal Components

This section introduces the principal components and then gives a description of each component, what it does and how the component interacts with other components. 3-

MO has three main components: a model controller, computation node and modeller, as shown in Figure 21.

The model controller is a single object which is on the root processor, and is responsible for maintaining synchronization between the processors. When writing an application which uses 3-MO, the programmer only has to interact with the model controller. This interaction consists of calling functions of the model controller, which include initialization, model computation, visualization and termination. This way, the issues of parallelization are hidden from the programmer.

Figure 21: 3-MO's architecture.

The computation node object is an object that controls the program flow on the other processors (those other than the root processor). Each of the other processors has one

of these computation nodes. Each computation node has its own octrees to store the partition blocks that are sent to that particular processors.

3-MO is based on an event-driven paradigm, where the model controller sends messages to the other computation nodes to tell them what will be the next step in the application. The user may want to view different visualizations of the data before computing the next generation of the model. Therefore, once the user has decided how the application is to continue, the application calls a function of the model controller, which in turn sends a message to the computation nodes. The computation nodes then execute the appropriate functionality using their local data, depending on the message. This functionality is implemented by the modeller object, as mentioned in Section 4.1.

The model controller does very little in terms of computation since it just sends messages to the other computation nodes. Instead of having the model controller just sending and receiving messages, it was designed to have its own local computation node that performs some of the model computation. This local computation node is given a workload that is comparable to all the other computation nodes. This way, the root processor, where the model controller resides, is not idle while waiting for the results from the other computation nodes.

4.4 Program Flow

Since 3-MO models a cellular automaton, the program's flow must be similar to that of a cellular automaton. Specifically, there is an initialization phase, a modelling phase, and a termination phase. During the initialization phase, the voxels of the model are set to some initial value. In the modelling phase, the program iterates over the data and computes the modelling logic, or can present a visualization of the data to the user. In

the termination phase, the program can free any resources used and the results that are computed can be displayed to the user or saved to a disk.

Conceptually, the programs that can be used in 3-MO must have the following flow:

1. Initialization
2. A number of iterations of: (The number of iterations is set by the programmer):
  - Model Computation. This phase consists of two parts: updating and modelling. In order to ensure that the partition blocks have up-to-date neighbourhood information, they automatically update information with their neighbours before computation. Once all the blocks on a processor have been updated, the application iterates over the all the voxels to compute their new values according to the modelling logic. See Section 4.4.3 for a more detailed explanation.
  - Visualizing the current state of the model. The application generates a visualization of the data and presents it to the user.
3. Termination and clean-up

This is a high-level overview of the operations carried by each of the nodes. The following subsections give an in-depth explanation of each of the phases and presents features provided by 3-MO.

4.4.1 Initialization

Before modelling, the programs must create the partition blocks and their respective voxels, initialize their values, and distribute them among the processors. This initialization consists of iterating over each voxel and setting it to an initial value. This initial value is specific to a given model and therefore cannot be automatically set by 3-MO. The initialization is therefore left to the developer who must write an initialize function

that is defined in the Datum class. See Section 4.1 for an overview of components that must be defined by the user.

There are two ways that the voxels can be initialized: centralized or distributed. If the initial values of the voxels are being read from a file, or retrieved from a database, then it might not be possible to retrieve more than one voxel at a time. In order to avoid problems with synchronization, the blocks can be initialized sequentially by the model controller. The model controller then maps each block to a processor and transmits them over the network.

Figure 22: Synchronized initialization

During this centralized initialization phase, all of the computational nodes wait for some communication from the model controller. As shown in Figure 22, when a block

has been initialized, the model controller maps the block to a computation node and signals the corresponding computation node that a block is going to be transmitted. The model controller then transmits the block to the computation node, which then stores it in its local memory and goes back into a waiting state. The model controller then repeats the steps for the next block until all blocks have been initialized and transmitted. After initialization, the model controller sends a message to the computation nodes that the blocks have all been created, and both the computation nodes and the model controller exit the initialization phase.

In the distributed initialization phase, the computation nodes construct and initialize the blocks in parallel. The computation nodes iterate through all of the blocks in the data set, and if a block's coordinates map to that computation node, then it constructs and initializes the voxels of that block. This initialization is done by setting each voxel's value to a value that is supplied by the application prior to computation. This initial value is specific to a given model.

4.4.2 Updating

In 3-MO, there are two types of neighbourhoods supported: the von Neumann and Moore neighbourhoods (see Section 2.2). Although there are two types of neighbourhoods supported, the von Neumann neighbourhood is really just a subset of the Moore neighbourhood. Therefore, it would be good if the code for updating the voxels of the von Neumann neighbourhood were shared by both neighbourhood types. Updating the Moore neighbourhood would start by updating the voxels in the von Neumann neighbourhood, and then updating the remaining voxels that are not in the von Neumann neighbourhood. This makes the implementation simpler since code is shared.

The main problem with the updating is to maintain synchronization among processor nodes. It is important to avoid deadlocks, where computation node A is waiting for communication from computation node B, while at the same time, B is waiting for communication from node A. They are both waiting for the other to communicate and will only continue when they have received data. One way to avoid this deadlock is to only update one block at a time. This guarantees that only one block is waiting for communication at a time, but this reduces the advantages of parallelism.

One approach to updating is to use asynchronous communication. Asynchronous communication involves the sending of messages between processors such that the receiving processor does not have to explicitly wait for the message. The message gets stored in a temporary buffer on the receiver's computer to be retrieved when the receiver wishes to check for new messages. One drawback of this approach is that it is possible to have a buffer overload if too many messages get stored in the temporary buffer.

Another approach is to use synchronous communication. Synchronous communication involves the sending of messages between processors such that the receiving processor explicitly waits for the message. Waiting for a message can introduce inefficiency in a program since the receiving processor must wait for the sending processor to finish sending before continuing. However, it is possible to organize the updating so that processors are always either sending or receiving, and not waiting. It was decided by the author to use synchronized communication for updating since there is less danger of having a buffer overrun, and there is no need for a processor to continually check to see if it has received any new messages. At any point of updating, each processor knows whether it is sending or receiving a message, and with which processor it is communicating.

One way to update the blocks while avoiding deadlocks is to synchronize the di-

rection in which the blocks update their data. If the computation nodes update all of their blocks first in the x direction, followed by the y, and z directions, then a deadlock will be avoided since the volumes on the edge of the data set do not have to wait for any communication. Figure 23 illustrates the notion of updating blocks in a common direction.

Figure 23: Synchronized updating in various directions.

Each computation node has several blocks that are mapped to it. However, the computation nodes can only update one block at a time, since they only have one processor. At any given time, there can only be n blocks being updated at any given time, where n is the number of computation nodes. To choose which blocks are updated at any given time, there is an advantage in updating a section of neighbouring blocks at once, instead of updating the volumes in a random manner as shown in Figure 24. By updating a

section of neighbouring blocks, each computation node has a block to update, and a neighbouring processor will, at the same time, have that block's neighbour to update (assuming it has a neighbour). When a computation node sends voxels to its neighbour, the neighbour implicitly knows to which block the voxels belong to. They come from the neighbour of the partition block that is currently being updated.

Figure 24: Processors updating within a processor lattice and updating in a random order.

In deciding how to define this neighbourhood of n blocks, recall that the processor lattice, defined in Section 3.3, has n processors, and defines a neighbourhood relationship among them. This neighbourhood relationship can be used to determine which blocks to update first. If the processor lattice has p_x , p_y and p_z processors in the x , y and z directions, respectively, then the first p_x , p_y and p_z blocks in the x , y and z directions get updated first. Using this configuration of blocks, each computation node has a block

to update, and the processors implicitly knows which voxels are being transmitted.

Once one group of blocks has been updated, the computation nodes can then go to the next group of blocks, and not have to visit the first group again. Once all of the blocks have been updated, the computation nodes return to a waiting state where they wait for the next command from the model controller.

4.4.3 Model Computation

When computing an iteration of the model, each computation node iterates through all of its local partition blocks and invokes the model function with the block and iteration number as a parameters. This model function iterates over all voxels within the block and computes their values for the next generation, according to the user-specified modelling logic. The voxels for the next generation are written to a second buffer, since the old values are needed for the computation of other voxels.

Once the modeller has finished the computation on all of the voxels within a block, it uses a double buffer technique to copy the voxels from the temporary buffer back to the block. As an optimization, the computation node exchanges a pointer to the second buffer with the block's current buffer. This way, copying the voxels is done in a single step instead of copying each voxel to the original buffer. The computation node then continues iterating over its local blocks until they have all been processed.

One of the parameters passed to the modelling function is the iteration number of the model. If the user wishes to write intermediate data to disk, say every n^{th} iteration, then this is done by checking if n iterations have passed since the last time the data was written, or since the start of the program.

4.4.4 Visualization by Slice Generation

The user might like to see the state of the data during its computation. For 3-dimensional data, the voxels can be seen by viewing a two-dimensional plane that intersects them, as shown in Figure 25. The user should be able to change the orientation of the plane so that the user can view various portions of the data, or see a region of the data from different angles.

Figure 25: Visualization by slice generation.

To generate this image, we need to compute which voxels intersect the plane. Since the blocks are on different processors, the image will have to be generated in sections and then assembled into a picture to display.

It is easy to generate the intersection of a plane and a block when the plane is perpendicular to one of the x , y , or z -axes. In this case, one of the dimensions is held fixed, and the algorithm iterates through the other two dimensions. For example, if the plane is perpendicular to the z axis, and the plane intersects the z axis at the point $(0, 0, 5)$,

then this plane can be expressed as the equation $z=5$. Then, the image can be generated by iterating through all voxels whose z -coordinate is equal to 5. Combining the image from the processors is simple since they can be ordered by their x and y coordinates. This approach requires $O(n^2)$ time where n is the maximum number of voxels in the x and y directions.

An arbitrary plane can be specified in two ways:

1. Two non-parallel vectors, $P = av_1 + bv_2$
2. A normal vector and a point that lies on the plane (also known as the Cartesian form) $Ax+By+Cz+D=0$.

The two forms are equivalent, and interchangeable. 3-MO uses the Cartesian form because if two of the coordinates are known (x and y , or x and z , etc...) then the value of the third coordinate can be computed from the plane's equation.

If n is the maximum size of the data in the x , y , and z -axes, then the size of the intersection of the plane and the voxels is $O(n^2)$. However, the plane might only intersect a corner of the data set, as demonstrated in Figure 28. It would be best if the algorithm for generating the intersection is output-sensitive, where it would depend only on the number of voxels that intersect the plane instead of the dimensions of the entire data set.

As a first idea, it is possible to calculate the points of intersection of the plane and the outer edges of a partition block. This way, the algorithm can compute the two-dimensional boundary of the intersection and then iterate through this region. However, a problem arises because a region of intersection may have a different number of intersections with the sides of the block. In this thesis, we define that a plane intersects a block only if the plane intersects some region inside the block. In other words, if a plane intersects the block along one of the boundary edges or corner points, but does

not intersect any region within the block, then it does not intersect the block. In this case, there is nothing to visualize since the intersection is either a two-dimensional image, or a single point. Also, since it intersects along a line or at a point the plane will also intersect the blocks which share that edge or point. The plane must intersect one of those blocks somewhere within the volume of the block, and the algorithm will pick that block to calculate the intersecting voxels.

If the plane does intersect some region within the block, then the plane can have 3, 4, 5 or 6 points of intersection with the boundary edges, depending on the orientation of the plane. Figure 26 displays the possible cases:

Figure 26: Four cases of intersection between a plane and rectangloid

4.4.4.1 Constructing a Slice Once the region of intersection is known, all that is needed to do is to iterate through this region and map the voxels to a plane to form a two-dimensional image. What is not known is how to iterate through the region, i.e. which dimensions to iterate through. If the image construction algorithm iterates through two dimensions that are parallel to the plane's normal vector, then it will not produce an image. In essence, it will be looking at the plane from the side.

However, if the algorithm iterates through the region along a perpendicular dimension, then it will produce an image. A perpendicular image is better since it covers more of the image area. In essence, the algorithm will have to visit a fewer number of volumes to produce the image. This is used by the algorithm to pick which two dimensions to iterate through in order to generate the image.

It is necessary to iterate through two dimensions, since the algorithm is using the equation of a plane in Cartesian form, which has three unknown values (x , y , and z). By iterating through two dimensions, two of the unknowns are generated by iterating, and the third calculated directly from the equation of the plane. The three values define a point which lies on the plane, and thus the voxel that contains that point must intersect the plane. Another way of thinking of what the algorithm is doing is that it through one of the sides of the volume and casts axis-parallel rays into the block and computes their intersection with the plane. The corresponding voxel is then copied to the image that represents the intersection.

Determining the orientation of a plane can be done by computing the dot product of the normal vector of the plane with each of the x , y , and z -axis unit vectors. For example, if the normal vector of a plane is (nx, ny, nz) , then the dot product between the normal vector and the x axis is $(1 \times nx) + (0 \times ny) + (0 \times nz) = nx$. Likewise, the dot products between the plane's normal vector and the y , and z axes are ny and nz , respectively. The

axis that produces the largest dot product is closest to being perpendicular to the plane. The algorithm will then iterate through the other two dimensions to generate the image.

The algorithm now has all the information needed to choose which dimensions to iterate through. Let $d1$ and $d2$ be those two dimensions which will be iterated through. The algorithm is then:

Algorithm 1 Computing the intersection of a plane and a volume

---

```
For  $a = d1_{\min}$  to  $d1_{\max}$  do  
    For  $b = d2_{\min}$  to  $d2_{\max}$  do  
         $c = \text{plane.computeMissingValue}(a, b)$   
         $v = \text{volume.getVoxel}(a, b, c)$   
        slice.setVoxel(  $a, b, v$ )
```

In most cases, the coordinates of the point of intersection (a, b, c) are not integers. In order to retrieve the proper voxel, the `getVoxel` function calls a programmer-specified interpolation function. This interpolation function takes the eight nearest voxels, and the point of intersection so that the user can specify a method for deciding what the value of the voxel should be. For example, if the point of intersection is (a, b, c) , then the 8 nearest voxels are all combinations of floor and ceiling of $(a, b, \text{ and } c)$. The user can implement different methods such as the nearest neighbour, trilinear interpolation, or whatever is needed by the model.

In the case where there are 3, 5 or 6 points of intersection between the plane and the boundary of the voxels, it is possible that the image generated is incomplete. For example, if there are 3 points of intersection, then the image produced will be a triangle instead of a rectangle, which is the shape of the final image. In this case, a “null” value

is placed where the voxels are in the missing region of the image so that they can be overwritten by actual values when combining images.

4.4.4.2 Combining Images Once the computation node has computed the intersection of all its voxels with the plane, the images are combined in parallel with the images of other computation nodes. The images are combined in a binary fashion, where two computation nodes are paired to combine their images, as shown in Figure 27.

Two Images from different computation nodes

Combined to make a new image. In the next round, this image will be combined with yet another image.

Figure 27: Combination of two images.

To combine two images, one computation node sends its image to another computation node. The receiving computation node then iterates through the voxels of the image, and if the voxel is not the null voxel, then it copies the voxel to its own image. If a voxel has been copied at a given location, and another voxel maps to that same location, then it overwrites the previous voxel that was there.

At the start of the combination, each computation node knows the total number of computation nodes in the framework. For any given round of combination, let N be total number of computation nodes left in the combination. At the start, N is the total number of computation nodes. The algorithm for combining the images is as follows:

Algorithm 2 Recombining Images for Computation Nodes

```

I = computation node's id;
while ( $N \geq 2$ )
{
    if ( $I \geq N$ ) quit recombining // the image has already been copied
    else if ( $N$  is odd) and ( $I = \lfloor N/2 \rfloor$ ) //if there is an odd number of computation
        nodes, then one must sit out a round
        do nothing
    else if ( $I > \lfloor N/2 \rfloor$ ) send myImage to computation node ( $I - \lfloor N/2 \rfloor$ ) //if the id is
        in the upper half of remaining nodes, it sends
    else // if the id is in the lower half of remaining nodes, it receives
    {
        img = receive image from computation node ( $I + \lfloor N/2 \rfloor$ )
        myImage = combine( myImage, img )
    }
     $N = \lceil N/2 \rceil$ 
}

```

If there is an odd number of computation nodes, then the computation node with middle id number sits out for that round of combination. The overall process of combination takes $O(\log(n))$ rounds in the worst case, where n is the number of computation nodes.

Once all of the combination rounds are complete, then model controller's computation node will have the complete image, since the model controller has id 0. It is then up to the programmer to use this data, either writing the values to a disk, or by displaying a representation of the image on a graphical interface of some kind.

4.4.4.3 Slice Construction Algorithm Optimization Instead of iterating through two dimensions of the entire volume, the algorithm is optimized to only iterate through the region of intersection, as shown in Figure 28. The first step is to compute a bounding axis-parallel region, which is the smallest axis-parallel rectangloid that contains all points of intersection between the plane and the voxels. This region is computed by finding the minimum and maximum coordinates of the intersected voxels, $(x_{\min}, y_{\min}, z_{\min})$ and $(x_{\max}, y_{\max}, z_{\max})$. The other corner points of the region are the 6 other combinations of min and max x , y and z coordinates.

This optimization makes the algorithm output-sensitive, since the area that is iterated through is proportional to the number of voxels that intersect the plane, and not to the number of voxels in the volume.

Figure 28: The bounding rectangloid of a plane intersecting a volume.

This optimized algorithm runs in $O(n^2)$ time in the worst case, where n is the maximum number of voxels in the region of intersection along the x , y and z directions. This algorithm is therefore optimal in the worst case since the area of intersection is $\Omega(n^2)$.

4.4.5 Termination and Clean-Up

When the modelling iterations have finished, a model might need to release some resources. If the resources are kept by the computation node's modeller object, they are released in the modeller's destructor. The computation nodes also iterate through all the blocks and invokes the `finalize` function with each block as a parameter. This `finalize` function can be used to either display a visualization of the data to the user, or write the data to disk, etc.

5 Model Examples

Two models were implemented to show that the three-dimensional modelling framework can be tailored for individual models, and also to test the performance of 3-MO. The first model is a simple fire model developed by Brinch Hansen [9], which models the spread of fire in 2-dimensions. This model was picked because it is simple to implement, and it can be used to test the visualization functionality of 3-MO. For this thesis, the fire model was extended to model the spread of fire in 3-dimensions. There is no practical reason for extending the fire model to three dimensions, other than a proof of concept, and to test the speedup of a three-dimensional model using 3-MO.

A more complicated and realistic model is the cluster explosion by laser pulse model, described in [13, 14, 31]. It models the interaction of a laser beam with a cluster of atomic particles in three dimensions and was originally implemented by Christian Jungreuthmayer at the University of Ottawa. This model was chosen to show that it is possible to integrate pre-existing models with 3-MO, as well as to compare the performance of a pre-existing parallel model, and to a similar model using 3-MO.

The model was previously written as a parallel application in Fortran, and was designed to run on a cluster parallel computer. The Fortran model uses a data set size of $120 \times 120 \times 180$ to model the electric and magnetic fields. Each voxel has 9 variables to store the state of the voxel (x, y, and z vector components for each of the electric field, magnetic field, and time-averaged magnetic fields), giving a total of 288 bytes per voxel. In order for the computed values for the electric and magnetic fields to remain accurate throughout the model, the model must use a small time increment in each iteration, around 1×10^{-14} seconds. With such a small time step, there must be many iterations to simulate the movement of the particles in the cluster expansion; somewhere in the order

of 10^5 iterations.

The model must also compute the trajectory of the particles in the simulation. In order to compute results that can be compared with those observed in the laboratory, there must be tens of thousands of particles (10000 - 150000). With so many iterations used in the modelling process, a parallel computer is needed to compute the results in a reasonable amount of time.

The program flow of the Fortran model was similar to that of 3-MO; there was an initialization phase, a modelling phase, and a final clean-up phase. This made it a good candidate to be integrated into 3-MO, however it had to be refactored in order to implement the interface required by 3-MO.

5.1 Fire Model Description and Theory

In [9], Brinch Hansen proposes a model to simulate the spread of fire in two dimensions. This model illustrates how cellular automata can be used to simulate natural behaviour. This model uses the van Neumann neighbourhood to compute the next state of a cell. This model was extended to three dimensions so that it can be implemented in 3-MO.

In this model, each cell has one of five states: alive, burning1, burning2, burning3 or dead. The alive state represents an area where the vegetation is alive. The three burning states represent different stages of burning: beginning, middle and end. The dead state represents an area where all the vegetation in an area is dead. An area that is dead cannot burn since all of the combustible material has been burned. To compute the transitions between states, the model uses the following rules:

1. If a cell is in the alive state, then it has some small probability, p , at each iteration

of changing its state to burning. This probability, p , is to be specified by the user. The value of p should be quite low, since forest fires usually have a single point of ignition. If there n voxels in the data set, then the probability of ignition should be $\frac{1}{n}$ because the expected value of modeling an iteration is that there is a single point of ignition of the fire in the data set. This simulates the ignition of a fire.

2. If a cell is alive, and has at least one neighbour that is in a burning state, then it is defined to have a 50% probability of changing state to burning. This simulates the spread of fire from one voxel to a neighbouring voxel. The probability of 50% is determined as a property of the model, but can be altered to change the length of time that a voxel is burning.
3. There are three burning states for a cell: burning1, burning2, and burning3. If a cell is in the burning1 or burning2 state, then it is defined to have a 50% probability of changing to the next burning state (burning2 or burning3, respectively). This simulates the burning of combustible material, until it has all been burned.
4. If a cell is in the burning3 state, then it is defined to have a 50% probability of changing to the dead state in the next iteration. This simulates the fire dying.
5. If a cell is in the dead state, then it has some small probability of changing to the alive state. This probability is to be specified by the user. This simulates the growth of new vegetation after the fire.

At the start of the model, every cell is set to the alive state. This model can then be run as long as desired by the user. The fire starts in several places at once, with those starting points being distributed uniformly throughout the voxels. The fire then grows outwards from these sources, while new fires are simultaneously starting in unburned areas.

This model produces fairly idealized results of fire propagation, but it is sufficient to demonstrate the simplicity of cellular automata. Due to this simplicity, the framework was expected to have a good speedup. The reader is referred to Section 5.3 for the results.

5.1.1 Integration of the Fire Model and 3-MO

As stated in Section 4.1, there are two C++ classes that must be implemented in order to use 3-MO: the Datum and Modeller class. In this application, the Datum class has only one instance variable which is needed to keep track of its current state. There were 5 possible states: *alive*, *burning1*, *burning2*, *burning3*, and *dead*. The class also had accessor methods for getting and setting the state of a cell. The Modeller class implements the modelling logic for the fire model. It implements the `model()` and `finalize()` functions, as described in Section 4.1.

A user interface was developed by the author to demonstrate the model, as well as slice generation. A picture of the user interface is shown in Figure 29. The large green and red area shows the state of the model after three generations. The areas in green are in the alive state, and those in red are in one of the burning states.

Figure 29: The fire model user interface.

The interface allows the user to control the flow of the model. The user can view the state of the model by viewing the intersection of a plane and the data. Once the user has finished viewing the current state of the model, the user can make the model continue to computing the next iteration of the model. The user can continue computing and viewing the state of the model until they wish to quit the program.

5.1.2 Fire Model Controls

There are controls on the right side of the interface to specify the plane that intersects the data. The voxels that intersect this plane are what will be displayed to the user. The front and side view windows are used to specify a point that lies on the plane. Using the slide bars, the user moves the point throughout the data set. For more information of visualization by slice generation, the user is referred to Section 4.4.4.

Below the “Front View” and “Side View” windows are the “Vertical Angle” and “Horizontal Angle” controls. With these controls, the user can specify the normal vector of the plane. These angles are shown both numerically, and graphically to give the user an intuitive feel of the orientation of the plane. In effect, the user is manipulating the plane’s normal vector using three dimensional polar coordinates. However, the normal vector is unit length, so that leaves two parameters that are set by the user using these controls.

To show the results of constructing slices by intersecting a plane with a volume, the following data set was constructed. Using $128 \times 128 \times 128$ voxels, the voxels were initialized according to their z coordinate. Those with a z-coordinate of 0 were assigned the colour green, and those with z-coordinate 127 were assigned the colour red. The voxels in between were assigned colours that are linearly interpolated between green

and red, according to their z-coordinate. Figure 30 shows a screenshot of a plane that is perpendicular to the z axis, intersecting the volume along the middle z value ($z = 64$). One would expect that image be a solid darker green colour. Figure 31 gives an example of the orientation of the plane.

Figure 30: Screenshot of a plane intersecting the data along the middle z coordinate.

Figure 31: An example of a plane that is perpendicular to the z axis.

By decreasing the z coordinate to 0, the colour should be a light green colour, as shown in Figure 32.

Figure 32: Screenshot of a plane intersecting the data along the bottom, where the z coordinate equals 0.

Likewise, increasing the z coordinate to 127 should produce an image whose colour is bright red, as shown in Figure 33.

Figure 33: Screenshot of a plane intersecting the data along the top, where the z coordinate equals 127.

Now for a more interesting demonstration of a plane intersecting colours, Figure 35 shows a plane that is perpendicular to the y axis. One would expect that this image would have bright green on the top, and bright red on the bottom, since the plane has a constant y coordinate. Figure 34 shows the resulting image.

Figure 34: Screenshot of a plane intersecting the data in the middle of the volume, with the y coordinate equal to 64.

Figure 35: An example of a plane that is perpendicular to the y axis.

Figure 36 shows the plane intersecting in the middle of the data set, and perpendicular to the x axis, as shown in Figure 37.

Figure 36: Screenshot of a plane, perpendicular to the x axis, intersecting the middle of the data set with x coordinate equal to 64.

Figure 37: An example of a plane that is perpendicular to the x axis.

Figure 38 shows a the plane intersecting the data, that is is not perpendicular to any of the three axes. The orientation of the plane can be seen in the plane controls at the bottom left corner of the picture.

Figure 38: Screenshot of a plane that is not perpendicular to any axis, and passing through the centre of the data.

5.2 Ionized Particle Cluster Explosion Model Description and Theory

This model simulates the interaction of a high-powered laser beam with atomic particles. It has been observed that when a cluster of atoms is subjected to a high-powered laser, the atoms undergo ...“a violent break-up of the cluster, involving the emission of energetic particles and photons...The mechanism of absorption and dissipation of such large amounts of energy in the clusters is still not understood” [13].

There are two theories which attempt to explain why the cluster expands so rapidly: the hydrodynamic expansion model, and the ionization ignition-Coulomb explosion model. The hydrodynamic expansion model proposes that some of the atom's electrons are freed by the laser energy, but most of the electrons absorb the energy and are not freed. With this absorption of energy, the cluster of atoms then expands due to hydrodynamic pressure.

The ionization ignition-Coulomb explosion model proposes that the electric field of the laser causes the electrons to be freed from the atoms with enough energy to leave the cluster (see Figure 39). Since the remaining atoms are positively charged, they repel each other thus causing the expansion. Since there is the presence of an electric field, there should be a greater number of electrons that escape in the direction of the electric field.

Figure 39: Electrons being freed by an electric field. The large dots are particles, and the small dots are electrons. This picture was taken from [14]

The major difference between these two theories is that the hydrodynamic explosion model predicts that the electrons will escape in a uniformly distributed manner. The ionization ignition-Coulomb model predicts that the distribution will be uneven with a greater number of electrons escaping in the direction of the electric field in the laser pulse. There are several research papers [13, 14] describing experiments done to try and measure this distribution of electrons in order to verify which model is correct.

5.2.1 Integration of the Cluster Explosion Model with 3-MO

To model this interaction, Jungreuthmayer implemented a parallel Fortran application on a cluster parallel computer that computes electric and magnetic fields as described in [31]. The model defines an evenly spaced lattice to store the local electric and magnetic field values, as illustrated in Figure 40. In reality, these fields are continuous in space,

so care must be taken that the distance between these cells are small enough so that the error between any interpolated values and the actual field intensity values are acceptable.

Figure 40: Graphical representation of local fields to represent the global field. This picture is from a presentation given by Christian Jungreuthmayer at an HPCVL symposium at Carleton University, on November 15th, 2002.

The Fortran model divided the data into sub-regions and distributed them among the processors. Each processor had one block of the data for which it computed the model. The only caveat was that the model had two types of data: electric and magnetic field values defined at evenly-spaced points in space (voxels), and particles that could move freely about in space. 3-MO was designed as a cellular automata which only has voxels and not particles.

3-MO had to be adapted to incorporate modeling particles. The partition blocks had to be changed to store particles that were located within one of the block's voxels. The

particles move freely in space, and therefore have an x , y , z coordinate as a member variable to store their position. There can also be a different number of particles in each partition block, whereas they all have the same number of voxels.

The neighbour update function had to be modified to account for the fact that a particle moves in space, and might move from intersecting a voxel in one partition block, to intersecting a voxel on a neighbouring partition block. The update function had to be changed so that it sends neighbouring voxel information, and possibly particles that have crossed from one partition block to another.

For each iteration in the model in this model, there are several steps of the model that must be computed:

1. Iterate over all particles and model their interaction with the laser. This interaction might cause the release of an electron.
2. Iterate over all particles and compute their movement. These particles may have an electric charge which causes a force between other charged particles and the electric and magnetic fields. This force must be incorporated when computing the movement of the particles.
3. The movement of a charged particle induces an electric current which affects the local electric field. Therefore, the model iterates over all particles and computes the electric current produced by their movement. This electric current is represented as a lattice similar to the electric and magnetic fields. This current is only used within an iteration, so its values do not need to be stored from one iteration to the next.
4. An electric current produces an electric field, so the model iterates over the voxels, and computes its effect on the corresponding electric field points.
5. The magnitude of the electric field has an effect on the magnetic field so the model must iterate over the electric field points and compute their corresponding magnetic field points.

As described in Section 4.1, there are two C++ classes that must be implemented to create an application using 3-MO: the Datum and Modeller classes. The Datum class

represents a voxel in the model, with 9 variables: x , y , and z vector components for electric, magnetic and time averaged magnetic fields. The Modeller class implements the modelling logic in the `model()` function, but this logic was already implemented in the Fortran. In this case, the Modeller class is just a wrapper class that calls the appropriate Fortran functions.

5.2.2 Integration Issues

This model was implemented in Fortran, but 3-MO is written in C++. The Fortran model is quite large (around 20000 lines of code) and would take a long time to rewrite it in C++. However, it is possible to mix Fortran and C++ code into one program, but there was not much documentation on how to do this.

Another complication was that the Fortran model was not designed to be integrated into the framework. The source code had to be reorganized so that the initialization, modelling, and finalization portions of the program were organized into corresponding functions which could then be called by the framework.

The language syntax was also a problem. In Fortran, all variables must be declared before any commands. Since the variables were all declared before the initialization code, it was hard to determine where each of variables were used. Some were just used in the initialization section, some just in the looping section, and some in both. In separating the two program portions into different functions, a long time was needed to determine which variables were initialized in the initialization section, and then used in the looping section. These variables were the parameters for the model.

It was also necessary to find which variables were used to represent the state of the model. Although these variables were only used in the looping section, they needed

to be stored outside of the modelling function to make sure that the state of the model is kept in between iterations of the model. In the end, there are approximately 100 parameters that are passed into the looping function.

5.3 Results

This section presents different tests that were run to test the behaviour of different aspects of 3-MO. Those aspects include the speed-up of applications using 3-MO, the effect of partition block size on the run time of a model, and a comparison of the run times of a model using 3-MO using two different mapping schemes. The time taken for slice generation was also measured to get an idea of the time required, and of the speed-up.

Testing the speed-up of applications using 3-MO show how the application behaves as a different number of processors are used. In some applications, the speed-up stops increasing as more processors are used. It is interesting to see if models using 3-MO behave in this manner. Section 3.2.2 introduces the idea that there is a trade-off between computation and communication. Testing the effect of partition block size on the application runtime will show that this is indeed true.

5.3.1 Computer Environment

The chosen model of computation was a Multiple Instruction Multiple Data (MIMD), using a distributed memory architecture. The machine used to develop 3-MO was a Beowulf Cluster comprising of 32 dual 2.0 GHz Xeon processors nodes and 32 dual 1.7 GHz Xeon processors. All of the processors are interconnected by a Cisco 6509

network switch using a Gigabit Ethernet network.

5.3.2 Test of Speed-up for Fire Model and Particle Explosion Model

Both models that were previously introduced were run several times on a different number of processors to test the speed-up of the respective applications. The following subsections describe how the tests were run, how the models were configured and present the results of those tests.

5.3.2.1 FireModel Speed-up The fire model was run several times using a different number of processors in order to measure the speed-up. For each different number of processors, the application was run three times in order to get a better sampling. Within one test of the application, there was one iteration of updating and modelling. That is to say, the application was initialized, then the computation nodes updated the neighbourhood information of their local partition blocks, the model was run using the data, and then the application terminated.

Figures 41 and 42 show the first 4 iterations of the fire model to illustrate how the fire spreads.

Figure 41: Screenshots showing the first two iterations of the fire model. Green areas show voxels in the alive state, red in either burning1, burning2, or burning3 state, and blue in dead state.

Figure 42: Screenshots showing the third and fourth iterations of the fire model. Green areas show voxels in the alive state, red in either burning1, burning2, or burning3 state, and blue in dead state.

The fire model uses a data set of $512 \times 512 \times 512$ voxels, and a block size of $32 \times 32 \times 32$ voxels.

The tests were run using 1 up to 32 processors. The time was measured on each proces-

sor, measuring how much time was spent communicating, modeling, and sitting idle. The graph in Figure 43 shows the speed-up of the model using different numbers of processors.

Figure 43: The speed-up of the fire model.

The results show an almost linear speed-up for the framework. This is expected since there is an equal balance of work among the processors.

Figure 44 shows the average time taken for communication, modeling and sitting idle for the processors during the test. Most of the time is spent computing the model, and a small fraction of the time is spent on communication. Idle Time is shown in the graph, however it is so small compared to the modelling time, that it is not visible in the graph. As shown in Figure 43, the speed-up is nearly linear, so the total time taken for a given number of processors is half the time taken for half the number of processors.

Figure 44: The average processor usage for different number of processors computing the fire model.

Figures 45 and 46 show the time taken to run the program using different numbers of processors. It shows the time taken per processor for communication, modeling, and sitting idle. As previously mentioned, most of the time was spent modeling, and a small fraction of the time spent on communication. The idle times for the processors are included in the graph, however they are not visible when compared to the modeling and communication times. The results are almost all the same among the tests, and among processors. This is because there is an equal amount of work to do among the processors, regardless of what state each voxel is. If model were a bit more sophisticated, and the amount of computation more dependant on the data, then there would be a larger variation in modelling times between the processors.

Figure 45: Time taken per processor for configurations of 2 to 8 processors.

Figure 46: Time taken per processor for 16 processors for fire model.

5.3.2.2 Overpartitioning The fire model was configured to be run on 8 processors, and a data size of $256 \times 256 \times 256$ voxels, and the voxels from the point (0, 0, 0) to (127,

127, 63) were initialized to be in the burning state. The block size was first chosen to be $128 \times 128 \times 128$, so that half of the voxels on one of the processors would start in the burning1 state, and all other voxels would be in the alive state, as shown in Figure 47.

Figure 47: Half of the voxels on one of the eight processors are set to the burning state (shown in red). The other four processors map to the second plane of voxels.

In the fire model, there is much less work to be done if a voxel is in any of the burning states. The voxel has a 50% chance of changing state to the burning2 state. This involves generating a random number between 0 and 1, and if it is above 0.5, then the voxel changes state.

If a voxel is in the alive state, then it has some small probability, p , of changing into the burning1 state, or if it has any neighbours in the burning state, then it has a 50% probability of changing into the burning1 state. Checking the values of the neighbours of a voxel incurs a slow-down, since the values must be retrieved. Therefore, the processors whose voxels are all set to the alive state should take more time than the processor whose voxels are all set to the burning1 state. This configuration of the

fire model is not unreasonable, since fire generally starts in a single area and spreads outwards. So it is quite likely that one portion of the area is burning, while the rest is not.

As shown in Figure 48, processor 0 takes less time to compute its results than all the other processors. This is because half of its voxels are initialized to the burning1 state at the start of the computation. In this case, the standard deviation in the modelling times on the processors is 546.85 milliseconds.

Figure 48: Fire model run with a load imbalance on one of the processors.

The model was then reconfigured to have a block size of $64 \times 64 \times 64$ voxels. Since half of the voxels on one of the processors were previously initialized to the burning state, if the block size is decreased by 8, then 4 of the processors should now have partition blocks that have all their voxels initialized to the burning1 state, as shown in Figure 49.

Figure 49: Four of the eight processors have voxels in the burning state (shown in red). The other four processors map to the second plane of voxels.

As seen in Figure 50, four of the eight processors now have a lesser workload than the other 4 processors. There is a difference of more than 300 milliseconds in the modelling time between the processors that have voxels in the burning state, and with those that have none. The standard deviation between the modelling times is 189.51 milliseconds.

Figure 50: Fire model run with a load imbalance on four of the processors.

Now, by cutting the block size in half again, to $32 \times 32 \times 32$ voxels, there should be an even number of voxels in the burning state among the processors, and therefore there should be an even workload among all of the processors.

Figure 51: An even distribution of voxels in the burning state (shown in red) among the eight processors. The other four processors map to the second plane of voxels.

As seen in Figure 52, this is indeed the case. The times taken for modelling seem more uneven than in the two previous graphs, however by looking on the scale, the difference in times is much smaller than in the previous graphs. In fact, the standard deviation in all of the modelling times is only 35.82 milliseconds.

Figure 52: Fire model run with an even load balance.

So by overpartitioning the data among the processors, we get a reduction in the load imbalances between the processors. This shows that overpartitioning can be used as a load balancing scheme.

5.3.2.3 Ionized Particle Explosion Speed-up The ionized particle explosion model was also tested for its speed-up on the same cluster computer as for the fire model test. The particle data available for the model uses a data set of size $120 \times 120 \times 180$ voxels. The model was tested on configurations of 1 up to 16 processors, and was run 3 times for each configuration. The block size was $15 \times 15 \times 15$ voxels for all the tests. For timing the tests, the time was recorded on each processor, broken down by communication time, modeling time and idle time. The application was initialized, then computed one round of updating and modeling, and was then terminated. Figure 53 shows the speed-up in graphical format.

Figure 53: The speed-up of the particle model.

The reader will notice that the speed-up of the framework is close to optimal. As the number of processors increases, there is a greater gap between the actual speed-up and the theoretical speed-up. This is due to the imbalance in workloads between processors that currently have particles within their region of space and those without particles, as is shown in Figure 56. When a volume has no particles within its region of space, there is no movement to compute so it sits idle while waiting for other processors to finish.

Figure 54 shows the average usage of the processors during the tests. The application has near-optimal speed-up so the overall time taken decreases by half as the number of processors double. The time was spent mostly on computing the model, and a small fraction spent on communication. The idle time is also shown, however it is so small that is not visible for all numbers of processors. The communication time is measured by timing the entire time taken for updating the neighbouring voxels, so it includes the

time for sending and retrieving the information, and for copying it into the appropriate partition block. This is why the graph shows that there is communication for running the model on one processor.

Figure 54: Average usage of processors for tests using varying number of processors.

Figures 55 and 56 show the time taken for communication, modeling and sitting idle for each processor during the test. Most of the time is spent on modeling, and a small fraction of the time spent on communication.

Figure 55: Time taken per processor for configurations of 1 to 8 processors.

Figure 56: Time taken per processor for 16 processors for the particle model.

Using 3-MO, the particle model actually ran around five times slower than the original Fortran code when using one processor. This is due to the fact that in the original Fortran code, the data are stored in an array and are retrieved by indexing into the array by the modelling logic. In order to adapt the code to use 3-MO, the arrays were removed from the Fortran code, and were replaced by the partition blocks in 3-MO. When the Fortran code needs to access a data value, it calls a C++ function that was added to 3-MO to return the specified voxel. Since the model is using function calls instead of array indexing to retrieve data, it incurs a significant performance slow-down. Also, much of the time spent computing is in updating the values of the electric and magnetic fields. There are $9 \times 120 \times 120 \times 180$ field values (23.3 million), but only 1000 particles, so there was not a large imbalance of workload between the processors, nor a large idle time compared to modeling time.

The two models implemented did not have a great deal of workload imbalance between the processors, although the particle explosion model did have a small amount. The main reason for this is that the two models implemented simply did not have a large imbalance in computation time. The models were simply chosen to show that 3-MO can be used to implement normal models. If it turns out that a model has a large imbalance in workload, then the programmer can make the partition block sizes smaller, so that there is a finer granularity of the data distribution and even out the workload imbalances, as shown in Section 5.3.2.2.

5.3.3 Block size test

As mentioned in Section 3.2.2, the size of the volumes has an impact on the time needed for a round of modeling. To show that this is indeed true, tests were run on the fire model

using different block sizes. The block sizes tested were $4 \times 4 \times 4$, $8 \times 8 \times 8$, $16 \times 16 \times 16$, $32 \times 32 \times 32$ and $64 \times 64 \times 64$ voxels. The total data size was $512 \times 512 \times 512$ voxels, and the test was run using 8 processors. Figure 57 shows the time taken for communication, modeling and sitting idle.

Figure 57: Time taken for fire model with the processor folding mapping scheme using different block sizes.

There is a greater amount of time taken for small block sizes. This increase in time is due to the time increase in communication. The time needed for computation stays the same for different volume sizes. This is due to the fact that, although the volume sizes change, the overall number of voxels on one processor remains the same, as does the time needed to iterate over the data set. The idle time of the processors is also shown in the graph, but it is so small compared to the modeling and computation time that it is not visible.

5.3.4 Mapping Schemes

The block size test was done using the same configuration as in Section 5.3.3, except that it uses the standard mapping scheme, introduced in Section 3.3. Figure 58 shows the results of the test.

Figure 58: Time taken for fire model with the standard mapping scheme using different block sizes.

There is a difference of approximately 40 seconds, or 20%, between the two mapping schemes when using a block size of $4 \times 4 \times 4$. This is due to the fact that, for neighbouring blocks that reside on the same computer, updating neighbouring voxels is done within the processor's memory, and not by transmitting the voxels across the network, and therefore much faster. For a block size of $8 \times 8 \times 8$, there is a difference of 5 seconds, or 20% between the two mapping schemes. When using a block size of $16 \times 16 \times 16$, the difference drops to 0.8 seconds, or 15%, and the difference drops to

10% for a block size of $32 \times 32 \times 32$. As the block size increases, the difference in time for the two mapping schemes decreases. This is because the size of the entire data set remains the same at $512 \times 512 \times 512$ voxels, and the number of processors is still 8. So as the block size increases, the number of blocks per processor decreases, and so the advantage of the processor folding mapping scheme also decreases.

Figure 59: Comparison of communication and idle times of the fire model using standard mapping (on the left), and processor folding (on the right).

Figure 59 shows a comparison of communication and idle times of processors using the standard and processor folding mapping schemes. In the graph, the idle times are multiplied by 100, so that they are visible compared to the communication times. Using

the processor folding scheme, both the communication times and idle times are less than using the standard mapping scheme.

5.3.5 Slice Generation

The fire model uses slice generation to view the state of the model (see Figure 29). This generation is done in parallel, so there should be a speed-up in slice generation as more and more processors are used in the process. To test this speed-up, the fire model was used with a data size of $512 \times 512 \times 512$ voxels, and with a block size of $32 \times 32 \times 32$ voxels. The slice generation was performed on a different number of processors: 1, 2, 4, 8, 16, 32, and 32. The results are shown in Figure 60.

Figure 60: Time taken for generating slices using different numbers of processors.

The results show that as more processors are added, the time for merging slices decreases, since more processors are being used. Also, the time that the processors are sitting idle increases. This is due to the fact that in each round of merging slices, as described in Section 4.4.4.2, half of the processors finish the merging process and have transferred their portion of the image to another processor.

Looking at Figure 61, the time for merging is highest for the root processor, i.e. processor number 0, and the idle time is least of all processors. This is because the root processor remains in the merging process until the end. Processor number 1 has a lesser merge time and greater idle time than processor 0, because processor 1 stopped merging at the second-to-last round. Likewise, processors 2 and 3 have similar times for merging and for sitting idle, and both processors have smaller merge times and greater idle times than processor 1, since they stopped the round previous to when processor 1 stopped. The pattern continues for processors 4 through 7, and again for processors 8 to 15.

Figure 61: Merge and idle times during image merging using 16 processors.

5.4 Summary

The particle model had good speed-up, and the fire model had near-optimal speed-up for up to 16 processors. 3-MO is faster when using larger volume sizes, and the processor folding mapping scheme is faster than the standard mapping scheme when using a small number of processors and a small block size. For larger block sizes, and larger number of processors, the mapping schemes give comparable times for communication.

Although 3-MO is relatively fast in computing models, it is possible that programs written from scratch for one specific model are faster, since they can be optimized for the specific problem. 3-MO is designed for the rapid development of parallel models, however 3-MO's source code can then be optimized by the programmer for the specific application. Another point is that models that are written from the start to use 3-MO will be faster than pre-existing programs that are simply plugged into 3-MO. The Fortran model of the ionized particle cluster explosion had to be refactored so that it could be incorporated with 3-MO, while making as few changes as possible to the Fortran code. In the process of refactoring the model, there were inefficiencies introduced by the fact that retrieving data was no longer done by indexing into an array, but by calling a function. If the ionized particle cluster explosion model were implemented using 3-MO from the start, then such inefficiencies would not have been introduced to the program.

The slice generation for data visualization works quite well, but it would help if it gave feedback to the user to show the current orientation of the plane. In the fire model, this was not implemented to give more space in the user interface to display the model data. Generating the slices is quite fast when compared to modeling (325 milliseconds compared to 40 seconds, respectively), and has good speed-up up to 16 processors.

5.5 Future work

Using generative programming[6, 29, 30], it is possible to extend this modelling framework for modelling applications in n dimensions. The question is whether or not there are any modelling applications that require more than 3 dimensions? Another extension to the framework might be to allow for volumes to contain data types other than voxels, such as particles, or sample points. In implementing the cluster explosion model, the framework had to be manually extended to incorporate data types for modelling particles. Is there a way of designing the framework so that it can handle an unknown number of data types? Also, in the modelling phase, there is typically one communication phase which precedes a modelling phase. It might be necessary, as in the case in the cluster explosion model, to compute some parts of the data, like the electric fields, and then update them to other processors, before other parts of the data can be computed, such as the magnetic fields. Can this functionality be added in a simple, generic fashion?

Another feature to implement is iso-surface visualization. This visualization technique is good for viewing 3-dimensional models. Another area of research is to see if there is a way of implementing an abstract parallel framework for visualization. Using 3-MO's approach for abstracting parallel modelling, is there a way of abstracting parallel visualization, so that a programmer can quickly implement a visualization technique?

This thesis presented two mapping schemes, which give a similar distribution of the data on a cluster computer. These mapping schemes might not produce an even distribution of the data on other architectures. It might be handy to implement other mapping schemes that are optimal for other architectures, so that the programmer can then choose the right one for the desired architecture. Another feature might be to

implement a mapping scheme centered around the source of change in the model. As in the case of modelling the outward propagation of a phenomenon, it would be possible to create a mapping where the volumes mapped to a processor “radiate” outward from the source of the phenomenon. In other words, if a sphere were centered on the source, and the surface of the sphere were divided into p sections, the volumes that map to a specific processor are those that lie within the “shadow” of rays that emanate from the center of the sphere, and intersect the portion of the sphere’s surface area corresponding to the processor.

References

- [1] M. Burton, W. Griswold. *Static Data Structures: Reconciling Template Metaprogramming and Generic Programming*, <http://citeseer.nj.nec.com/583387.html>, 2002
- [2] *Boost*, <http://www.boost.org>, 2002
- [3] H. Chou, *The Trend Cellular Automata Programming Environment for Artificial Life, Parallel Computing and Simulation Research*, 2001
- [4] E. F. Codd, *Cellular Automata*, Academic Press, New York, 1968
- [5] M. Chen, *Volume Modelling*, Volume Graphics, Springer, 2000, pp. 29-48.
- [6] K. Czarnecki, U. Eisenecker, *Generative Programming*, Addison-Wesley, Boston, 2000
- [7] I. Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995
- [8] S. Di Gregorio, R. Rongo, W. Spataro, G. Spezzano, D. Talia, *A Parallel Cellular Tool for Interactive Modeling and Simulation*, IEEE Computational Science & Engineering, 1996, pp. 33-43
- [9] P.B. Hansen, *Parallel Cellular Automata: A Model Program for Computational Science*, Concurrency: Practice and Experience, 1993, pp. 425-448.
- [10] C. Hecker, D. Roytenberg, J.-R. Sack, Z. Wang. *System development for parallel cellular automata and its applications*, Future Generation Computer Systems, 1999 pp. 235-247
- [11] D. Hutchinson, M. Lanthier, A. Maheshwari, D. Nussbaum, D. Roytenberg, J.-R. Sack, *Parallel Neighbourhood Modelling*, ACM-GIS, 1996, pp. 25-34
- [12] D. Hutchinson, M. Lanthier, A. Maheshwari, D. Nussbaum, D. Roytenberg, J.-R. Sack, *Parallel Neighbourhood Modelling: Research Summary*, In SPAA 1996 Symposium on Parallel Algorithms and Architectures, (Padova, Italy, June 24-26 1996). pp. 204-207
- [13] V. Kumarappan, M. Krishnamurthy, D. Mathur, *Asymmetric emission of high-energy electrons in the two-dimensional hydrodynamic expansion of large xenon clusters irradiated by intense laser fields*, Physical Review, April 2003, Vol. 67, No. 4, pp. 043204-1 to 043204-8

- [14] V. Kumarappan, M. Krishnamurthy, D. Mathur, *Asymmetric High-Energy Ion Emission from Argon Clusters in Intense Laser Fields*, Physical Review Letters, Aug 2001, vol. 87, no. 8, pp. 085005-1 to 085005-4
- [15] M. Lanthier, D. Nussbaum, J. R. Sack, *A Parallel Implementation of Geometric Shortest Path Algorithms*, Parallel Computing, Vol. 29, 2003, pp. 1445-1479.
- [16] S. Lombeyda, Michael Aivazis, *Parallel Isosurface Calculation and Rendering of Large Data Sets in IRIS Explorer*, Proceedings of the IEEE Visualization '98, October 1998, pp. 167-174, ACM Press
- [17] D. Meagher, *Geometric Modelling Using Octree Encoding*, Computer Graphics and Image Processing, Vol 19, No. 2, 1982, pp. 129-147
- [18] S. Molnar, M. Cox, D. Ellsworth, Henry Fuchs. *A Sorting Classification of Parallel Rendering*, IEEE Computer Graphics and Applications, 1994, pp. 23-31
- [19] J. von Neumann. *Theory of Self Reproducing Automata*. University of Illinois Press, 1966
- [20] J. Nievergelt, P. Widmayer, *Spatial Data Structures: Concepts and Design Choices*, Handbook of Computational Geometry, Elsevier Science, 2000, pp. 725-764.
- [21] M. Norman, J. R. Henderson, I. G. Main, D. J. Wallace, *The Use of the CAPE Environment in the Simulation of Rock Fracturing*, Concurrency: Practice and Experience, Wiley, Vol. 3, 1991, pp 687-698
- [22] D. Nussbaum, *Parallel Spatial Modelling*. PhD Thesis, School of Computer Science, Carleton University, 2000
- [23] D. Roytenberg, *Developing Parallel G.I.S. on the Alex AVX II Computer*, Master's Thesis, School of Computer Science, Carleton University, 1996
- [24] M. Smith, *Cellular Automata Methods in Mathematical Physics*, PhD Thesis, MIT, 1982
- [25] A. Stepanov, M. Lee, *The Standard Template Library*, <http://citeseer.nj.nec.com/stepanov95standard.html>, no. X3J16/94-0095, 1994
- [26] D. Talia. *Cellular automata thrive on parallel systems*. Scientific Computing World, No. 42, pp 21-22, October 1998,
- [27] S. Telford, *A Cellular Automata Simulation Environment for Modelling Soil Bioremediation*, HPCN Europe, pp. 1143-1146, 1999

- [28] T. Toffoli, N Margolus, *Cellular Automata Machines, a New Environment for Modeling*, the MIT Press, 1987
- [29] T. Veldhuizen, *C++ Templates as Partial Evaluation*, Partial Evaluation and Semantic-Based Program Manipulation, 1999, pp 13-18
- [30] T. Veldhuizen, D. Gannon, *Active Libraries: Rethinking the roles of compilers and libraries*, Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, Philadelphia, 1998
- [31] J. Villasenor, *Rigorous charge conservation for local electromagnetic field solvers*, Computer Physics Communications, No 69, 1992, pp 306-316
- [32] *Visible Human Project*,
http://www.nlm.nih.gov/research/visible/visible_human.html, 1986
- [33] *Visible Human Server*, <http://visiblehuman.epfl.ch>, 1998
- [34] S. Wolfram. *Computational Theory of Cellular Automata*, Comm. Math. Phys, Vol 96, 1984, pp 15- 57
- [35] S. Wolfram, *Mathworld*, <http://mathworld.wolfram.com/CellularAutomaton.html>
- [36] S. Wolfram, *Twenty Problems in the Theory of Cellular Automata*, Physica Scripta, 1985, pp. 170-183