PARALLEL EXPLICIT FEM ALGORITHMS USING GPU'S

A Dissertation
Presented to
The Academic Faculty

by

Seyed Parsa Banihashemi

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Civil and Environmental Engineering

Georgia Institute of Technology
December 2015

PARALLEL EXPLICIT FEM ALGORITHMS USING GPU'S

Approved by:

Dr. Kenneth M. Will, Co-Advisor
School of Civil and Environmental Engineering
Georgia Institute of Technology

Dr. Richard Vuduc, Co-Advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Arash Yavari
School of Civil and Environmental Engineering
Georgia Institute of Technology

Dr. Donald W. White
School of Civil and Environmental Engineering
Georgia Institute of Technology

Dr. Barry J. Goodno
School of Civil and Environmental Engineering
Georgia Institute of Technology

Date Approved: October 30, 2015

ACKNOWLEDGEMENTS

I start by expressing my respect and gratitude towards my advisor, Dr. Kenneth M. Will. Thank you for your patience, support and guidance through my time at Georgia Tech. You gave me the freedom and guidance to form the current research. You also gave me the opportunity to pursue my passion in the field of computer science. Your insight and critical thinking helped me shape my ideas, direct this research and conclude this dissertation.

I am deeply appreciative to Dr. Richard Vuduc, my respected co-advisor, who gave me the key points and ideas in the computational aspect of my research and spent a lot of time for me to help me to choose the right direction at numerous points during my research.

I would like to thank Dr. Donald W. White, Dr. Barry J. Goodno and Dr. Arash Yavari for their effort and time on as my committee members and the insights they offered me that led to a deeper understanding of my topic.

Financial support from the Georgia Tech Computer-Aided Structural Engineering (CASE) Center is gratefully acknowledged.

Thank you my friend and roommate Mr. Ehsan Hosseinian for sharing your point of view with me and helping me during my time in Atlanta. I would also like to thank my former officemates, Mr. Borja Zarco, Mr. Julian Diaz and Dr. Ben Deaton for being good friends and helping me on numerous occasions.

Finally, I cannot thank my family enough: My father, my mother and my sister Sarah. Your tremendous love, patience and support made all my life accomplishments possible. I love you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The Explicit Finite Element Method is a powerful tool in nonlinear dynamic finite element analysis. Recent major developments in computational devices, in particular, General Purpose Graphical Processing Units (GPGPU's) now make it possible to increase the performance of the explicit FEM.

This dissertation investigates existing explicit finite element method algorithms which are then redesigned for GPU's and implemented. The performance of these algorithms is assessed and a new asynchronous variational integrator spatial decomposition (AVISD) algorithm is developed which is flexible and encompasses all other methods and can be tuned based for a user-defined problem and the performance of the user's computer.

The mesh-aware performance of the proposed explicit finite element algorithm is studied and verified by implementation. The current research also introduces the use of a Particle Swarm Optimization method to tune the performance of the proposed algorithm automatically given a finite element mesh and the performance characteristics of a user's computer. For this purpose, a time performance model is developed which depends on the finite element mesh and the machine performance. This time performance model is then used as an objective function to minimize the run-time cost.

Also, based on the performance model provided in this research and predictions about the changes in GPU's in the near future, the performance of the AVISD method is predicted for future machines. Finally, suggestions and insights based on these results are proposed to help facilitate future explicit FEM development.

CHAPTER I

INTRODUCTION

1.1 Scope

The Explicit Finite Element Method is a powerful tool in nonlinear dynamic finite element analysis. Recent major developments in computational devices, in particular, General Purpose Graphical Processing Units (GPGPU's) now make it possible to increase the performance of the explicit FEM.

This dissertation investigates existing explicit finite element method algorithms which are then redesigned for GPU's and implemented. The performance of these algorithms is assessed and a new asynchronous variational integrator spatial decomposition (AVISD) algorithm is developed which is flexible and encompasses all other methods and can be tuned based for a user-defined problem and the performance of the user's computer.

The mesh-aware performance of the proposed explicit finite element algorithm is studied and verified by implementation. The current research also introduces the use of a Particle Swarm Optimization method to tune the performance of the proposed algorithm automatically given a finite element mesh and the performance characteristics of a user's computer. For this purpose, a time performance model is developed which depends on the finite element mesh and the machine performance. This time performance model is then used as an objective function to minimize the run-time cost.

Also, based on the performance model provided in this research and predictions about the changes in GPU's in the near future, the performance of the AVISD method is predicted for future machines. Finally, suggestions and insights based on these results are proposed to help facilitate future explicit FEM development.

1.2 Problem Statement

Dynamic analysis computes trajectories and mechanical properties of materials as a function of time. The solution methods used in this area are mainly classified as explicit and

implicit methods. The implicit methods are typically most applicable to linear problems and can have larger time steps than the explicit methods for these problems. However for nonlinear problems, implicit methods may require a large number of iterations. On the other hand, explicit methods require very small time steps but do not require iterations and the computational effort per time step has a linear relationship with respect to the number of elements. Explicit methods are typically much more suited where the strain rate is high as in blast or impact loadings.

Explicit methods are ideal for highly nonlinear phenomena such as crash, blast, impact, etc. The time-step requirement for these problems typically require a large number of time-steps, which is very costly. Practical nonlinear problems in this area may require several days to finish. Also with multiple loading conditions and mesh adaptivity, simulations longer than a few milliseconds are currently impractical for many problems.

This document delineates a research program focusing on introducing new explicit finite elements method (FEM) algorithms for Graphical Processing units (GPU's), including a GPU "spatial decomposition" algorithm [10] and GPU algorithms for the asynchronous variational integrator (AVI) method [51]. The AVI Spatial Decomposition algorithm (AVISD) is an algorithm that is very flexible and comprehensive and can be tuned for different situations.

Then for the first time, a self-tuning algorithm based on the input problem is introduced. The AVISD algorithm is adaptable and using an optimization method, it can be set to produce maximum performance. In order to minimize the time-cost function, there is a need for a formula for the time-performance model to be developed. This time-performance model is generated by running multiple tests under various conditions and using minimum sums of squares analyses, in order to find the constants of the performance model by regression analysis from benchmark problems. After that, the time-performance model is used as the objective function and the Particle Swarm Optimization method [42] is used to tune the parameters of the AVISD method. This way, a self-adapting mesh-aware algorithm is formed.

The computational devices, GPU's in particular, are changing rapidly and the computational throughput in the near future will be orders of magnitude greater than today. There are several problems that need more computational power such as adaptive dynamic analysis, with multiple load cases, and the simulation of longer time phenomena which are not practical with today's machines and algorithms. Since the performance of algorithms depend on the computer architecture, investment in the implementation of algorithms and designing new algorithms must be performed with a knowledge and foresight of the machines available in the near future.

In computational mechanics, no study has been carried out to predict the performance of explicit FEM algorithms on future computers. This research will use the performance relationships derived, along with the available predictions of the future computing devices proposed by computer scientists to further predict the performance of explicit FEM on future machines.

1.3 Background

1.3.1 Dynamic analysis of structures

Simulation of dynamic behavior of fluids and solids has long been of interest of engineers [6, 12]. The applications span a wide range of structural dynamics [4], material forming processes [80], wave propagation [64], etc. Finite element applications in structural dynamics also include problems in earthquake engineering [82, 50], stability analyses, crash [69], impact [77] and blast simulations [36].

Different simulation methods have been proposed for structural dynamics purposes mainly using the direct integration methods [7]. These methods are mainly categorized as implicit, explicit and hybrid implicit-explicit methods.

The implicit methods usually involve solving a system of simultaneous linear equations. These methods are capable of using larger time-steps, however, cases involving high nonlinearity require computing and assembling the matrices repeatedly during the iterations. Therefore, implicit methods are most often used in problems with low nonlinearity and larger simulation durations [57, 76].

The explicit methods, on the other hand, deal with each finite element independently. Each finite element's stresses and forces are derived directly from the position of its nodes and each node's acceleration is merely due to the effect of forces of the elements directly connecting to that node. This "local" formulation results in uncoupled equations. Since there is no need for iterations, these methods are very effective in nonlinear problems [46, 64, 67]. There have also been some efforts to mix the explicit and implicit methods into different hybrid implicit-explicit methods [27, 32].

The implicit methods require the storing of the global stiffness matrix, so the memory required is of order $\mathcal{O}(n^2)$ ($\mathcal{O}$ is the Big O notation) with respect to the number of elements, n. Also, the floating point operations needed to solve a linear system is of order $\mathcal{O}(n^3)$. On the other hand, the explicit algorithms do not need the assembly and storing of a stiffness matrix. The memory needed is linearly proportional to the number of elements, $\mathcal{O}(n)$, and since the equations are uncoupled and are solved once for each element, the floating point operations required to solve one time-step is also of order $\mathcal{O}(n)$.

The only drawback of the explicit methods is the stability time-step requirement[60]. This time-step is the minimum time needed for the stress waves to travel through one element, which can be in the order of microseconds in practical problems. This time-step requirement leads to many time-steps if the duration of the simulation is long. Therefore, the explicit methods have been used mainly in simulating nonlinear and short time phenomena such as crash, blast and impact.

As a common approach to solve a general problem by the explicit method, the time-step for all elements is chosen to be the minimum required time-step among all elements. This "naive" method is the simplest, the most popular and the easiest to implement. However, the computational effort is high, due to the choice of minimum time-step for all elements, and hence, higher update frequency. This means since all elements are updated with a small time-step, elements will advance in time in smaller steps and so larger number of time-steps are required. Here, the update frequency can be defined as the inverse of the time-step.

Different methods have been proposed to overcome this issue. Belytschko proposed sub-cycling methods, where neighbor elements could have different time-steps, but multiples of

each other [8]. Neal and Belytschko later offered another method where non-integer (but not totally arbitrary) time-steps were allowed [60]. This method is sometimes called the "domain decomposition method" and works best where large portions of adjacent elements have equal time-steps and the boundary between different size elements is small compared to the entire mesh. Therefore, this method is not efficient for problems that have a large number of elements of different size. Also, different regions and the boundary has to be defined by the user.

Gradual energy dissipation during simulation is a potential issue in dynamic analysis and it is aggravated in long term simulations since the energy is dissipated step by step and large number of steps can lead to relatively high dissipated energy. Variational Integrators which preserve momentum have been used by many including Veselov [89, 90], Wendlandt and Marsden [94], and, Marsden and West [58]. Kane [40] showed that the Newmark-beta method [61] is also a variational method and can be derived in that way. Lew and Marsden also studied variational time integrators [52] and developed an asynchronous variational integrator [51] which allows arbitrary time-steps for each element. This method is very interesting since the maximum possible time-step at each element will be used and the computational work is minimized. However, as will be explained in chapters 3 and 4, this method exhibits less available parallelism and shows a more inherent sequential nature, which is a drawback for parallel processing.

There have been some doubts concerning the stability of some of the explicit FEM methods. Daniel [16] stated that Neal and Belytschko's method is "statistically stable" but unstable for some time-steps smaller than the stability limit (the time-step dictated by the size of the element which guarantees passing the stress wave through one element during one time-step). Rangarajan and Lew [75] showed that the resonances are generally not an issue and can be solved by gradually changing the sizes of the elements. This is still an issue if the time-steps are held constant. Fong et al. [22] showed that the resonance instabilities in asynchronous variational integrators are not a problem in solid mechanics applications. Resonance instabilities are the type of instabilities that are caused by rapid changes in element sizes and there is numerical errors in the process of waves passing between the

small and the large elements.

In research studies on explicit FEM algorithms, each algorithm is only compared with the "naive" algorithm and to the knowledge of the author, none of these studies have compared their proposed algorithms with others. One of the main contribution of this research program is providing performance relationships for different algorithms and comparing the new AVISD algorithm with the most common existing algorithms.

1.3.2 Application of Parallel Processing in explicit FEM

Many finite element dynamic analyses require a very fine mesh and a large number of time-steps for better accuracy. This computational cost can be formidable and render the numerical methods useless in some cases. Today, there are many practical problems that take a significant amount of time even for a single dynamic loading condition [85].

Limited studies on adaptive explicit dynamic analysis have been conducted [73] since the run-times are very high, the algorithms' efficiency is mesh-dependent and the relationship of the algorithm run-time with the structure of an arbitrarily generated mesh is not well understood. In addition, since the explicit FEM is dominantly used for nonlinear problems, the superposition principle is not valid and the structure has to be solved under different load-combinations, which is impractical due to the high time cost of the simulations.

Also, the simulation duration of practical explicit analyses usually range between a millisecond and a second. Longer simulations often require more computational throughput than available today.

According to this computational demand, many researchers have long felt the need for higher computational capacity using parallel processing in particular. Parallel processing is the use of simultaneous computational units to solve a single problem and engineers have been taking advantage of this parallel processing for several decades [31]. Various numerical algorithms have been developed and optimized specifically for parallel machines [59]. The structural engineering community has in part taken advantage of parallel processing for different purposes [1, 23], mainly static and dynamic finite element analyses [81, 45, 9].

One of the newest and most rapidly developing branches of parallel processing is the use

of graphical processing units (GPU's) in scientific computing. General purpose graphical processing units (GPGPU) have recently turned into one of the most interesting areas of research in high performance computing (HPC), because the new GPU's are relatively cheaper, have a significantly higher computational throughput for single instruction-multiple data (SIMD) class of problems, and are now made with native double-precision computation ability. In 2009, NVIDIA introduced FERMI GPU's that significantly improved double precision performance[62]. Among the top ten fastest supercomputers available in 2013, five use GPU's to achieve higher performance, including the Tianhe-2 [86]

GPUs have been around since 1990, but the scientific computing capability and development kits have only been recently available. In years after 2003, general purpose computing became possible, however, deep hardware knowledge was necessary. After the Compute Unified Device Architecture language (CUDA, which is a computer language for GPU's) was introduced in 2006, the GPGPU development became more user-friendly and straightforward. However, CUDA only supports NVIDIA GPU's [65]. The CUDA language is based on the $\mathcal{C}$ language with compilers and libraries to develop code for GPGPU's. In 2008, Apple formed the Khronos group and gathered specialists from different CPU and GPU manufacturers and software specialists and introduced the Open Computing Language (OpenCL) [44]. OpenCL is based on the $\mathcal{C}$ language and is available on several CPU and GPU platforms including AMD, NVIDIA and Intel GPU's and Intel and AMD CPU's.

A few years ago, CPUs hit their clock frequency limit, which lead to multi-core systems. A CPU by nature is a very fast computation unit optimized for sequential computing. On the other hand, the GPUs contain hundreds to thousands of computing units. Nowadays, GPU computation throughput exceeds, by an order of magnitude, the throughput of current CPUs [68].

GPGPU has been used in finite elements in the past decade. In 2004, Wu et al. used GPUs for interactive 3D soft tissue modeling [97]. Goddeke tested a simple finite elements problem on a GPU [25]. They solved the Poisson's equation for a unit square with dirichlet boundary conditions and bilinear elements with different sizes. Komatitsh worked on elastodynamics of linear anisotropic materials [48]. In 2008, Taylor published a paper on

nonlinear finite elasticity using GPGPU for surgical simulations [84]. Also in 2008, Comas published a paper on soft tissue modeling [13]. In 2009, Goddeke accelerated problems in linear elasticity using GPUs [26]. In 2010, Komatitsch analyzed seismic wave propagations [47]. Various basic linear algebra problems have also been solved on GPUs [43, 21, 19, 70, 88]. In addition to these, many other studied using GPUs for finite element applications in the past five years [37, 56, 39, 74, 17, 20, 83, 5, 24, 41, 11, 49, 15].

All these studies with GPU's used a constant time-step scheme, i.e. the aforementioned naive method, which is highly inefficient if the finite element mesh is non-uniform.

Computer scientists predict that the computational capacity of the GPU devices will be orders of magnitude larger in the next decade. In 2010, Nickolls et al. mention that the GPU's will continue to scale in performance about 50 percent per year[62]. With the mentioned need for faster explicit FEM solvers and better algorithms, and the rapid growing of GPU's, studying the application of GPU's under new and more sophisticated algorithms and understanding these changes seem inevitable for engineers. In the process of the current research, the predicted performance of the AVISD method in the near future is studied.

Currently, the only explicit FEM algorithm method used on GPU's is the naive method, however, some work has been done to parallelize other methods for CPU's, such as the AVI method [38, 34] and the domain decomposition method [85]. Some of the most recent implicit and explicit FEM studies have been carried out on GPU's[18, 98, 3].

1.4 Research Objectives

In the current research, all explicit FEM algorithms including the naive method, the "spatial decomposition" algorithm [29], the "Domain Decomposition" algorithm[8], are examined and adapted for GPU's. The goal here is to identify the potential and flaws in each of these algorithms and be able to design a new reliable algorithm. A rudimentary parallel AVI algorithm [51] called the AVI coloring algorithm is also introduced.

These different algorithms are expected to perform differently according to

1. The size of the problem

2. The mesh's statistical properties (variance, distribution homogeneity, etc.)

3. Machine architecture

After identifying the potentials in each method, at the next stage, a new parallel AVI Spatial Decomposition algorithm (AVISD) is designed, which encompasses all other algorithms, is flexible and also versatile.

In order to understand the machine specifications and problem specifications' role in the performance of the problem, a performance model is generated and verified by implementation.

Having the generated performance model as an objective function and using Particle Swarm Optimization, the AVISD method can be tuned to a specific computer and mesh, before the start of the solution.

At the next stage, the predicted architecture of the future computation systems in the near future is studied, and by using the performance relationships derived, the performance of the AVISD method for the future architectures is examined.

In this work, the OpenCL language along with the C++ language have been chosen, because of the better potential for hybrid CPU-GPU applications and also the universality of the OpenCL-compatible hardware.

The result of the implementations of the designed algorithms lead to a software called "paraDyn", which is a multi-algorithm C++ based platform for explicit FEM.

1.5 Organization of Dissertation

Chapter 2 delineates the basics of an elasto-plastic nonlinear dynamic finite elements analysis. Chapter 3 explains existing and new parallel explicit FEM algorithm. Chapter 4 explains the new general parallel explicit AVI Spatial Decomposition algorithm called AVISD. In chapter 5, the time cost performance model is discussed and the tuning and optimization process of the AVISD algorithm is explained in detail. Chapter 6 explains some predictions of the future cost and trends of the explicit FEM based on the predictions of the future for GPU's and also the generated performance model in chapter 5. Finally in chapter 7, conclusions, the list of contributions of the current research program, some explanations about the paraDyn software and suggestions for future research are delineated.

CHAPTER II

DYNAMIC STRUCTURAL ANALYSIS

This chapter explains the details of an explicit elasto-plastic finite elements formulation, which is used during the course of this research.

2.1 Classical transient structural analysis

Transient dynamic analysis seeks to compute the trajectory and the relevant needed entities (stress, displacement, etc.) of the body under the condition of dynamic stability and conservation of mass and energy.

The finite element method (FEM) is a tool to simplify a continuum by discretization into finite number of sub-regions with simplified displacement space. In the current chapter, the details of the explicit dynamic analysis are illustrated and also the stages of the step-by-step solution of the explicit dynamic analysis by the use of the FEM are outlined in this chapter. In addition, the details of the numerical solution considering plasticity are explained.

2.1.1 Equilibrium Equation

The dynamic equilibrium equation (Newton's second law) to be solved in solid mechanics can be derived:

$$\rho \ddot{u}_i - \sigma_{ij,j} = f_i(t,x) \text{ in } \Omega,\ i,j = 1,2,3 \tag{1}$$

$$\sigma_{ij} n_j = g_i(t,x) \text{ on } \omega \tag{2}$$

In which u is the displacement, ρ is the mass density, σ is the stress tensor and f is the equivalent external and body force, Ω is the domain of the material, ω is the boundary of Ω, n is the unit normal to ω and g represents the traction on the boundary ω. Also, the "$,j$" subscript indicates differentiating w.r.t. x_j.

2.1.2 Galerkin Method

In order to solve equations 1 and 2, the Galerkin method can be used to approximate the solution by using the test function v_i and computing the weighted residuals.

Note that here, the damping is ignored for simplicity, however, damping effects can be included if needed, but all the problems throughout this thesis are solved without damping. By multiplying v_i by both sides of equations 1 and 2 and integrating the first one over Ω and the second one over ω and summing the results yields:

$$\int_{\Omega} \rho \ddot{u}_i v_i d\Omega - \int_{\Omega} \sigma_{ij,j} v_i d\Omega + \int_{\omega} \sigma_{ij} n_j v_i d\omega = \int_{\Omega} f_i v_i d\Omega + \int_{\omega} g_i v_i d\omega \tag{3}$$

By using the divergence theorem on the third term of equation 3, the result will be:

$$\int_{\omega} \sigma_{ij} n_j v_i d\omega = \int_{\Omega} (\sigma_{ij,j} v_i + \sigma_{ij} v_{i,j}) d\Omega \tag{4}$$

Substituting equation 4 into equation 3, yields:

$$\int_{\Omega} (\rho \ddot{u}_i v_i + \sigma_{ij} v_{i,j}) d\Omega = \int_{\Omega} f_i v_i d\Omega + \int_{\omega} g_i v_i d\omega \tag{5}$$

2.1.3 Space discretization

In order to solve equation 5, the finite elements method is normally used for the discretization in space.

The total domain is broken into sub-domains called elements:

$$\Omega = \cup \Omega_e, \qquad \Omega_i \cap \Omega_j = \emptyset \quad \text{for} \quad i \neq j \tag{6}$$

The elements are connected to each other by nodes. Each element's displacement can then be described based on the ones of its nodes:

$$u(x) = \sum_{i=1}^{n} N_i(x) u_i \tag{7}$$

In which N_i's are the so-called *shape functions* and n is the total number of nodes of the element.

The shape functions have the following properties:

$$\sum_{i=1}^{n} N_i(x) = 1 \tag{8}$$

$$N_i(x_j) = 0 \quad i \neq j \tag{9}$$

Now, in order to solve equation 5, v_i and u_i are expressed in terms of the shape function as follows:

$$\int_\Omega (\rho N_I N_J \ddot{u}_i^J + \sigma_{ij} N_{I,j}) v_i^I d\Omega = \int_\Omega f_i N_I v_i^I d\Omega + \int_\omega g_i N_I v_i^I d\omega \tag{10}$$

After dropping the v_i^I terms and some manipulations, the following equation is obtained:

$$\underset{\sim}{F} = \underset{\approx}{M}\underset{\sim}{A} \tag{11}$$

In which

$$M_{IJ} = \int_\Omega (\rho N_I N_J) d\Omega \tag{12}$$

$$\underset{\sim}{F} = \underset{\sim}{f_{int}} + \underset{\sim}{f_{ext}} \tag{13}$$

$$A^{iJ} = \ddot{u}_i^J(t) \tag{14}$$

and

$$f_{int}^{iI} = \int_\Omega \sigma_{ij} N_{I,j} d\Omega \tag{15}$$

$$f_{ext}^{iI} = \int_\Omega f_i N_I d\Omega + \int_\omega g_i N_I d\omega d\Omega \tag{16}$$

The domain of integrations is discretized over all elements and the integrations can be computed element by element. Equation 11 is defined at each point in the time domain.

2.1.4 Time discretization

The next task is to discretize the problem in the time domain. Here, the Newmark-beta method can be used because there is no numerical damping in this method, as will be explained in more detail in section 2.3.3. Numerical damping is caused by losing the total energy gradually during the course of simulation due to numerical errors. However, using other methods is possible and does not significantly affect the process. Also, the material damping is neglected in this formulation. By using the finite difference scheme and the

central difference method, which is a special case of the Newmark-beta method, it can be written:

$$\begin{aligned} \dot{u}_{n+\frac{1}{2}} &= (u_{n+1} - u_n)/\Delta t_{n+1} \\ \ddot{u}_{n+1} &= (\dot{u}_{n+\frac{1}{2}} - \dot{u}_{n-\frac{1}{2}})/\Delta t_{n+\frac{1}{2}} \end{aligned} \tag{17}$$

So,

$$\begin{aligned} u_{n+1} &= u_n + \dot{u}_{n+\frac{1}{2}} * \Delta t_{n+1} \\ \dot{u}_{n+\frac{1}{2}} &= \dot{u}_{n-\frac{1}{2}} + \ddot{u}_{n+1} * \Delta t_{n+\frac{1}{2}} \end{aligned} \tag{18}$$

These equations hold at each node and every degree of freedom respectively. For solving equations 18, the initial displacements and velocities (initial conditions) and also the accelerations are needed. The accelerations come from equation 11:

$$\underset{\sim}{\ddot{u}}_n = \underset{\approx}{M}^{-1} \underset{\sim}{F}_n \tag{19}$$

To calculate the internal forces, first the strain is calculated from the displacements, and then the new stresses are computed. After that, the nodal forces are computed. At each node, the internal forces of all element connected to that node are added together. This equivalent nodal force, added with the external forces and the body forces is the force that results due to the acceleration of the node.

The total force acting is the sum of the internal resisting forces, the external surface forces and the body forces:

$$F_n = f_n^{int} + f_n^{ext} + f_n^{body} \tag{20}$$

Equation 19 requires inverting the mass matrix. By assuming a diagonal mass matrix (lumped mass), the equations become uncoupled and the matrix inversion is not required. If the damping is included, in the same way, the diagonal damping matrix assumption is required to prevent a matrix inversion. This is one of the major assumptions of explicit FEM without which this method would not have been successful due to the computational effort required to invert the mass matrix. This assumption is very common [60] and some research has been done to show its accuracy and convergence[96].

Wu et al. illustrated two examples: a part impact problem, and a vehicle crash problem, in which the assumption of diagonal mass proved to be accurate. For the second problem, they showed that the computational cost of the consistent mass matrix was three times the computational cost of the diagonal mass matrix. In addition, the stability time-step for the consistent mass matrix was smaller than the time-step requirement for the diagonal mass matrix. In other words, the consistent mass matrix needed smaller time-steps to remain computationally stable[95].

The overall schematic of the time-steps are presented in Figure 2.1. The values of displacements and accelerations are known at the main steps and the velocities are calculated on half-steps. This method is usually referred to as the central difference method.

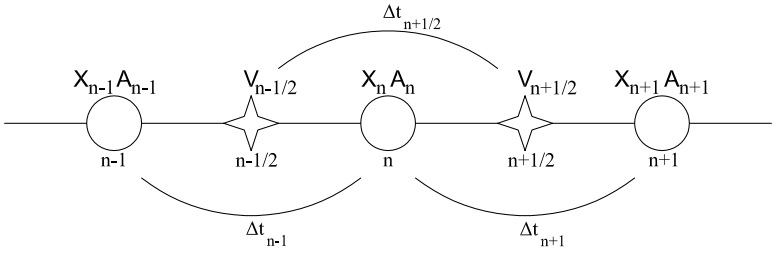

Figure 2.1: The time-line details. The circles represent starts and ends of time-steps and the stars represent the half-steps in which the velocities are calculated.

In Figure 2.1, X_n, V_n, and A_n represent the position, velocity and acceleration of the nodes at step n respectively. Here, if the time-steps are constant during the simulation, then Δt_{n-1}, $\Delta t_{n+1/2}$, and Δt_{n+1} will be equal. Figure 2.2 shows the flowchart of the explicit finite element analysis, which is very simple and straightforward.

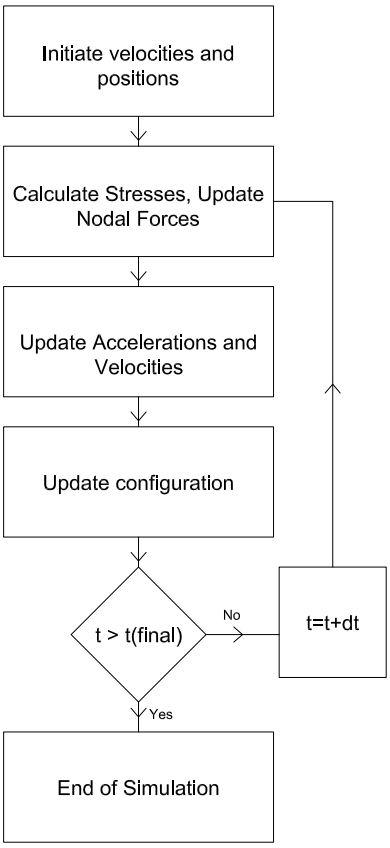

Figure 2.2: Explicit time discretization.

More details on the classical explicit FEM can be found in several books including the one by Wu and Gu [95]. The next chapter explains some details of the nonlinear FEM formulations used during the current research.

2.2 Plastic Finite Elements Analysis

Many of the practical applications of the explicit dynamics involve highly nonlinear behavior including large deformations, large rotations and also elasto-plastic material behavior. Although most of the formulations in finite elements analysis are derived from the principle of stationary energy and/or principle of stationary action, theses formulations can also be used when non-conservative behavior is involved.

When plastic behavior is present, the energy is not constant anymore, because the plastic behavior damps energy. So, the action is not stationary, but by assuming that the

time-steps are small enough and defining a plastic potential function, it is still possible to describe the internal resisting forces as the gradient of some potential function.

$$\underset{\sim}{f}^{\text{int}} = \overrightarrow{\nabla} V(u) \tag{21}$$

In which u is the displacement vector, V is the potential function, ∇ indicates the gradient operator, and f^{int} indicates the internal force vector. This will help us to continue using the methods that assume the existence of a potential function for the forces, including the variational methods, which will be discussed in more details in section 2.3. The index notation holds throughout this section.

The yielding limit is determined from the yield function, f, which is a function of the stress state.

$$f(\sigma_{ij}) = k(\kappa) \tag{22}$$

In which k is a material parameter from experimental results, and κ is the hardening parameter.

In general, it is common to assume that the strain is the sum of the elastic and the plastic parts:

$$\varepsilon_{ij} = \varepsilon^e_{ij} + \varepsilon^p_{ij} \tag{23}$$

The elastic part is derived from the Hooke's law. Since in the elastic case:

$$\sigma_{ij} = C_{ijkl}\varepsilon_{kl} \tag{24}$$

in which C is the elastic properties tensor, then:

$$\varepsilon^e_{ij} = \frac{d\sigma'_{ij}}{2\mu} + \frac{1-2\nu}{E}\delta_{ij}d\sigma_{kk} \tag{25}$$

Here, the *prime* superscript indicates the deviatoric stress terms.

To express the plastic strain terms, it is common [66] to assume that ε^p_{ij} is proportional to the stress gradient of a term called the *plastic potential*, Q:

$$\varepsilon^p_{ij} = d\lambda\frac{\partial Q}{\partial \sigma_{ij}} \tag{26}$$

in which $d\lambda$ is a dimensionless proportionality ratio. This equation is called the flow rule, which dictates the flow of the material after the yield point. The assumption of $f \equiv Q$ enables us to develop certain variational principles as stated at the beginning of this section. This is a common and valid assumption in most cases and such a formulation will be called the *associated* plasticity. By this assumption it can be written:

$$\varepsilon_{ij}^{p} = d\lambda \frac{\partial f}{\partial \sigma_{ij}} \tag{27}$$

These basics are enough for us to be able to calculate the stress-strain relationship.

$$\underset{\sim}{d\sigma} = \underset{\approx}{D_{ep}} \underset{\sim}{d\varepsilon} \tag{28}$$

$$\underset{\approx}{D_{ep}} = \underset{\approx}{D} - \frac{\underset{\sim}{d}\underset{\sim}{d}^{T}}{H' + \underset{\sim}{d}^{T}\underset{\sim}{a}} \tag{29}$$

In which

$$\underset{\sim}{a} = \frac{\partial f}{\partial \underset{\sim}{\sigma}} \tag{30}$$

$$\underset{\sim}{d} = \underset{\approx}{D}\underset{\sim}{a} \tag{31}$$

and $\underset{\approx}{D}$ is the elastic stress-strain matrix and H' is the hardening parameter:

$$H' = \frac{E_T}{1 - E_T/E} \tag{32}$$

and E_T is the instantaneous slope of the stress-strain relationship in the one-dimensional effective stress and strain space. The value of a depends on the flow rule chosen. The derivation and more details on this issue can be found in common plasticity textbooks [66].

2.3 Explicit FEM Algorithms

As stated before, there have been many attempts to reduce the computational work in explicit dynamics. Basically, these studies focus on avoiding element force calculation more frequently than needed.

In 1981, Belytschko et al. [8] offered partitioning of the mesh into different parts with integer ratio of the time-step. Neal and Belytschko [60] later lifted this assumption and allowed non-integer (but not arbitrary) ratio of time-steps. They showed that the plastic

analysis by their method maintained reasonable energy accuracy. These methods are usually referred to as the sub-cycling methods. Daniel [16] later stated that the sub-cycling methods are stable in a "probabilistic sense" and have been used successfully.

These methods are dependent on the user-defined sub-domains and sometimes are called the domain decomposition algorithms. Each domain maintains a particular constant time-step ratio and the boundary nodes and elements are updated according to the minimum time-step.

Halleux and Casadei [29], developed a "spatial decomposition" algorithm, which is not dependent on the user-defined domains. In 2003, Lew et al. introduced a variational based method which allowed arbitrary time-steps for each element.

The time-step requirements arise from the fact that during one step of the analysis, the stress wave must be able to travel through the element in order to keep the solution stable and valid. This required time-step depends on the speed of the wave, which is an inherent material property, and also the size of the element. So, the smallest dimension of the element divided by the speed of the wave gives the time wave requires to travel through the smallest dimension of the element. Lew et al. [51] suggest the following:

$$\Delta t_i^{\text{crit}} = s\frac{d_i}{c_i} \tag{33}$$

As will be explained below in more detail, the parameter s is a safety factor between 0 and 1, d_i is radius of the largest circle contained in the element and c_i is the speed of wave in the solid in element i:

$$c_i = \sqrt{\frac{\lambda_0 + 2\mu_0}{\rho}} \tag{34}$$

A safety factor is required to account for the effects of element distortions and shape irregularities in the stability time-step requirement. Also, the changes in the element size during simulation can cause the stability time-step to change. Lew et al. used a value of 0.1[51], and the same value is used in this research, although this value is believed to be very conservative.

In which λ and μ are the Lamé constants and ρ is the material density. The safety factor, s is usually enforced to account for possible future changes in the dimensions of the elements and also the slight changes in the speed of wave under different internal stress conditions. In cases of identity of materials used in each element, the only governing factor affecting the minimum time-step is the size of the elements.

There are mainly 4 different explicit FEM algorithms:

1. Naive algorithm [66]
2. Domain decomposition [60]
3. Spatial decomposition [29]
4. Asynchronous variational integrator (AVI) [51]

The *naive algorithm* is the best in terms of simplicity and ease of implementation and it works best when the elements are uniform which that is not always the case. A comparison has to be made to check at what point, in terms of a mesh's statistical properties such as mesh size variance, homogeneity, etc, this algorithm ceases to be the most efficient.

The *domain decomposition* algorithm is efficient when there are several user-defined regions with constant time-step for each region. This method is not automatic and needs data from the user and also is not useful if the mesh changes for any reason, including adaptive mesh resizing. Furthermore, the explicit methods proved to exhibit instability when there is sudden change in element size. Element size should be changed gradually to avoid instability [16]. For this reason, this method proved to be only "statistically stable", as mentioned in section 1.3.1. For these reasons, this method was not chosen to be implemented.

The *spatial decomposition* algorithm is automatic and does not require manual assignment of time-steps to the elements. It also accepts an arbitrary mesh. However, it is stated by Casadei et al[10] that this algorithm is not always efficient and its efficiency depends on the mesh. This efficiency is not well understood and needs to be further studied. Casadei et al. implemented and compared this algorithm with the naive algorithm by solving a bar

impact [30] problem. They reduced the run-time from 61.9 seconds to 6.75 seconds by using spatial partitioning.

The AVI method allows using an arbitrary time-step for each element, which makes it the best in terms of computational work in the sequential (single computational unit) case. But all the elements cannot be updated simultaneously and there is a chain of dependence between neighbor elements as stated by Lew et al. [51]. In the same paper, Lew et al. compared this method with the naive algorithm by solving a three-dimensional L-shaped beam and showed that the number of elemental updates required by AVI was one third of the naive algorithm using the Newmark method. It is not obvious under what mesh conditions this algorithm will perform better than the naive method and the spatial decomposition method, if there are enough parallel computational units available.

The term "Numerical Instability" is used to describe the case where accumulating error in a numerical simulation can make a solution process unstable resulting in the solution diverging from a correct solution, or by causing an error such as stack overflow, division by zero, etc, resulting in the program aborting.

In this chapter, different explicit FEM algorithms are explained and pros and cons of each are stated.

2.3.1 The naive algorithm

As stated earlier in this chapter, the size of the element governs the time-step size. In practical physical problems, occasionally, some reasons such as local accuracy, isolated nonlinearity or localized high stress gradients enforce the choice of smaller mesh sizes in a particular part of the domain, or having a mesh size gradient in a portion or throughout the physical domain.

As the first choice, it is possible to choose the smallest time-step among all the required time-steps as the global time-step. This way, all elements will have the same time-step, all minimum time-step requirements are satisfied, and no synchronization is required. This method follows the flowchart in Figure 2.3 for all elements concurrently.

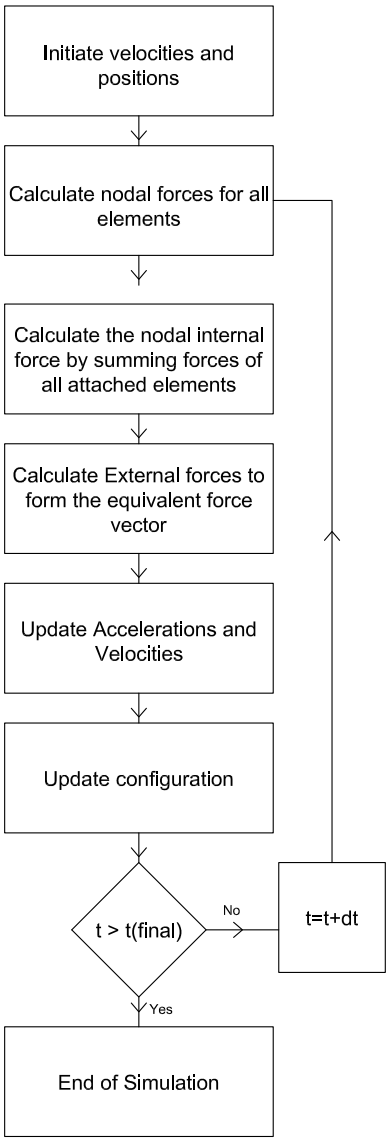

Figure 2.3: Naive explicit FEM simple flowchart.

The naive algorithm satisfies all the requirements and also allows very simple implementation and portability of the code. On the other hand, this choice requires all the elements to be updated as frequently as the smallest element. If the mesh is highly inhomogeneous and very few super-small elements exist, this method is also very inefficient.

2.3.2 Spatial decomposition algorithm

The spatial decomposition method categorizes elements into groups with time-steps ΔT, $\Delta T/2$, $\Delta T/4$,... . Each element's inherent required time-step is calculated based on equation 33. After that, a maximum time-step is chosen and all elements are grouped accordingly.

Casadei and Halleux made four observations [10]:

1. In the process of updating the elements, their intrinsic stability time-step (t_i^{crit}) must be respected.

2. If the acceleration or velocity of a node is updated, all elements attached to it are affected.

3. The critical time-step of all elements attached to a node must be respected in the process of updating the velocity or acceleration of that node.

4. The configuration (meaning the position of all attached nodes) of an element must be updated before updating that element (meaning calculating the internal and external forces of that element).

Based on these observations, they introduced rules that must be satisfied to keep the validity of the simulation results, which is the validity of the equilibrium equations and avoiding numerical damping (reduction of total energy due to numerical errors):

1. The velocity and acceleration of a node has to be updated at least as frequently as the critical requirement of all its attached elements:

$$\Delta t_i^{av} = \min(\Delta t_{n(i)}^{crit}) \tag{35}$$

   In which Δt_i^{av} is the time-step of updating node i's velocity and acceleration and $n(i)$ is the group of all elements attached to node i.

2. Updating an element (stresses, internal forces) must be done at least as frequently as all its attached nodes.

$$\Delta t_i = \min(\Delta t_{m(i)}^{av}) \tag{36}$$

$m(i)$ here is the list of all elements attached to node i and Δt_i is the new and more restricted time-step requirement of element i.

3. The displacements and position of the nodes must be updated at least as frequently as the restricted requirement of all its attached elements stated in rule 2.

$$\Delta t_i^{dx} = \min(\Delta t_{n(i)}) \tag{37}$$

Δt^{dx} is the time-step requirement on displacement and position update of node i.

The full algorithm is explained in the following in detail [10, 29]. A general review on the algorithm is provided here.

The set of possible time-steps, τ is defined as:

$$\tau = \left\{\Delta T, \Delta T/2, \cdots, \Delta T/(2^{d-1})\right\} \tag{38}$$

in which d is defined to be the size of the set τ. A cycling level is a smaller time-step within a macro step ΔT, with the size of the smallest time-step, $\Delta T/(2^{d-1})$. At first, the minimum and maximum Δt among all element is calculated. The authors state that if the $\Delta t_{max}/\Delta t_{min}$ ratio is larger than 1.7, the spatial decomposition algorithm is not feasible, because there is a need to use a smaller time-step for smaller elements than in the naive algorithm[10]. If spatial decomposition is used, the maximum number of cycling levels will be calculated as follows:

Initially, by having all time-steps dictated by stability for each element, d can be found [10]:

$$d = \text{ceiling}(\log_2(\Delta t_{max}/\Delta t_{min})) + 1 \tag{39}$$

Then each element K is assigned a stability time-step (δt_K) from the group:

$$\delta t_K \in \tau \tag{40}$$

and ΔT is the maximum time-step chosen. Also, it is defined:

$$\delta t_{min} = \Delta T/(2^{d-1}) \tag{41}$$

Then, the element stability frequency is defined as:

$$\phi_K = \delta t_K / \delta t_{min} \tag{42}$$

The nodal velocity and accelerations have to be updated at least as frequently as the connected elements' stability frequencies. So, the frequency of the node velocity/acceleration update will be:

$$\psi_i = \max \phi_K \quad \forall \text{ element } K \text{ connected to node } i \tag{43}$$

The elemental forces have to be updated at least as frequently as all their nodes. So, the frequency of updating elemental forces is:

$$\bar{\phi}_K = \max \psi_i \quad \forall \text{ node } i \in \text{element } K \tag{44}$$

And finally, the frequency of updating the nodal displacements has to be longer than the force frequency of all connected elements:

$$\bar{\psi}_i = \max \bar{\phi}_K \quad \forall \text{ element } K \text{ connected to node } i \tag{45}$$

ϕ_d is defined to be the maximum frequency possible. ϕ_d is equal to 2^{d-1}, because the maximum frequency will be the frequency associated with the smallest time-step:

$$\phi_d = (\Delta T)/(\Delta T/(2^{d-1})) = 2^{d-1} \tag{46}$$

At each macro step, there will be d micro-steps, because the smallest time-step is $\Delta T/$. The micro-step counter will be denoted by I and the macro-step counter by N. The action function, m, is the indicator of what updates are possible to be performed. The definition of m, the action function is:

$$m(I) = \phi_d / \kappa(I) \tag{47}$$

In which $\kappa(I)$ is the highest power of 2 factored in I.

The following is the pseudo-code for the spatial decomposition algorithm:

Algorithm 1 Spatial Decomposition Algorithm

1: $u \leftarrow u_0$, $v \leftarrow v_0$, $N \leftarrow 0$, $I \leftarrow 0$, $m \leftarrow 1$, $F^{int} \leftarrow 0$, $F^{ext} \leftarrow 0$

2: $N \leftarrow N + 1$

3: $I \leftarrow I + 1$

4: Compute $m(I)$ from equation 47

5: For all elements k for which $\bar{\phi}_K \geq m(I)$, update the elemental internal forces

6: For all nodes i for which $\psi_i \geq m(I)$, calculate the equivalent nodal forces and the new accelerations: $a_i \leftarrow (F_i^{ext} + F_i^{int})/m_i$

7: For all nodes i for which $\psi_i \geq m(I)$, calculate the new velocities: $v_i \leftarrow v_i + \Delta T/\psi_i * a_i$

8: For all nodes i for which $\bar{\psi}_i \geq m(I)$, calculate the new configuration: $x_i \leftarrow x_i + \Delta T/\bar{\psi}_i * v_i$

9: If $I < M$, go to step 3 (the next cycle)

10: If $N < N_{final}$, go to step 2 (the next step)

11: End

This algorithm is very sensitive to the ratio of element sizes. If the time-steps of different elements are close to each other, the use of the naive algorithm is more feasible. It is also possible to have two regions with two main ΔT and update the boundary elements and nodes between the two regions by the time-steps dictated by both regions. The boundary elements and nodes will be updated more frequently, but this idea will give the method more versatility and the ability to work with several user-defined connected regions with different time-steps, as shown in Figure 2.4:

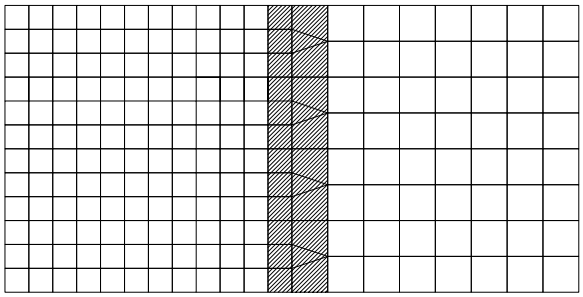

Figure 2.4: Adjacent domains with different sizes. The boundary elements are hashed.

2.3.3 Asynchronous variational integrator algorithm

The asynchronous variational integrator (AVI) method uses Hamilton's variational principle [51]. A variational integrator discretizes the Lagrangian. This approach preserves the local momenta and energy. Because of this, no numerical damping occurs during the solution. The Newmark-Beta method has been shown to be a variational integrator[51].

The main feature of the AVI algorithm is the ability to use arbitrary time-steps for each individual element. A brief description of the algorithm is provided here.

First, the deformation mapping is introduced:

$$x = \phi(X, t), \qquad X \in \Omega \tag{48}$$

x is the position at time t, X is the initial configuration and ϕ is the deformation map. Also, Ω is the physical domain of the problem. The boundary of the domain will be referred to as ω. $\dot{\phi}(X, t)$ and $\ddot{\phi}(X, t)$ then refer to the velocity and acceleration at each point X of the original configuration at time t.

The potential energy is defined as follows:

$$V(\phi, t) = \int_{\Omega} u dV - \int_{\Omega} \rho \boldsymbol{B}.\phi dV - \int_{\omega} \boldsymbol{T}.\phi d\omega \tag{49}$$

Where u is the strain energy density, ρ is the mass density, $\boldsymbol{B}$ is the body forces, $\boldsymbol{T}$ is the traction and $d\omega$ is the unit boundary surface. The kinetic energy is given by:

$$T(\dot{\phi}, t) = \int_{\Omega} \frac{1}{2} \rho \dot{\phi}^2 dV \tag{50}$$

And the Lagrangian is:

$$L(\phi, \dot{\phi}, t) = T - V \tag{51}$$

If the physical body's motion is defined between t_0 and t_f, the action is defined as:

$$\mathcal{A}(\phi) = \int_{t_0}^{t_f} L(\phi, \dot{\phi}, t) dt \tag{52}$$

In which $\mathcal{A}$ is called the *action function*. Hamilton's variational principle states that among all variations of ϕ between t_0 and t_f which are compatible with the boundary conditions, the one that keeps the action functional stationary is the acceptable deformation

map and the solution. This leads to the Euler-Lagrange equation:

$$\frac{\partial L}{\partial \phi} - \frac{d}{dt}\left[\frac{\partial L}{\partial \dot{\phi}}\right] = 0 \tag{53}$$

At this point, the discrete problem needs to be defined. In Figure 2.5, a 1D system with four elements is depicted. After that in Figure 2.6, the position of the system while it is held stationary through time is illustrated. Notice that the y-axis represents the time direction and each element has its own time-step.

Figure 2.5: A one dimensional system with 4 elements.

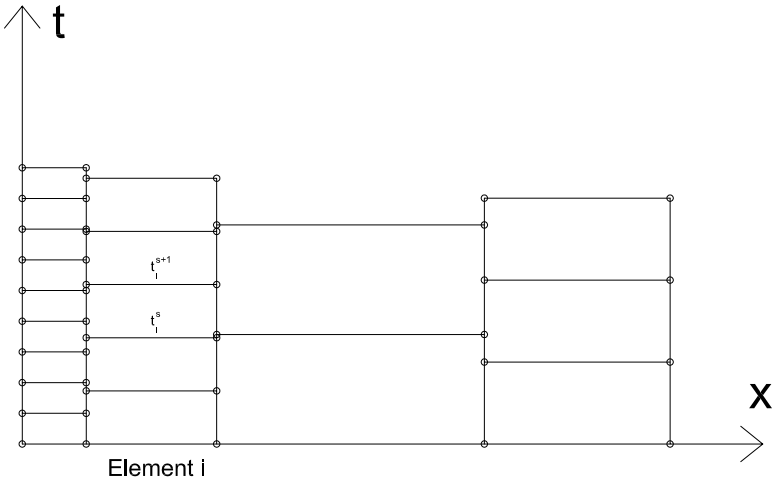

Figure 2.6: A one dimensional stationary system with 4 elements and different time-steps for each element.

As can be seen in Figure 2.6, the position of the nodes are constant through time. The position of each node is known at the "circled" point where the circled point are the starts and ends of the elemental time-steps. The position of a node between two circled points can be derived by linear interpolation between the circled points. Figure 2.7 represents the same system in motion.

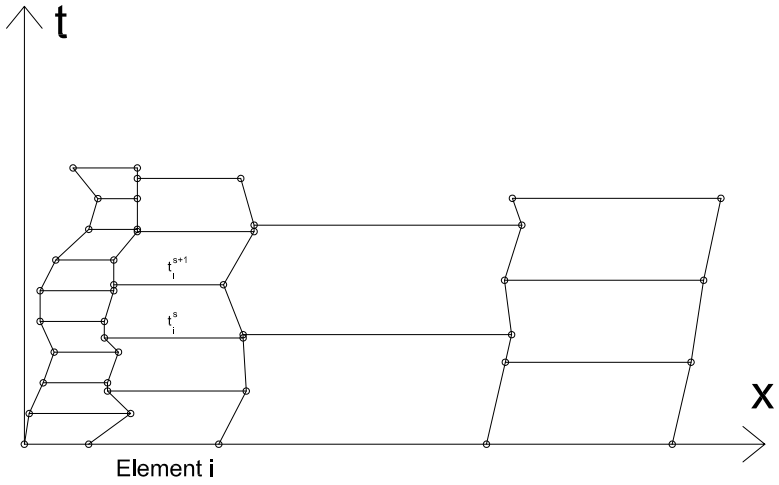

Figure 2.7: One dimensional system with 4 elements, constant velocity between two consecutive points

In the AVI method, the position of each node between two consecutive circled points in time is assumed to be changing linearly and therefore the velocity is constant in that period. Each time an element is updated, the velocity of all the nodes attached to it are updated.

The physical body will be meshed into finite elements connecting on nodes and the mesh shall be referred to as Ψ. The discrete deformation map is defined as:

$$\phi_h(X) = \sum_{i \in \Psi} x_i N_i(X) \tag{54}$$

Where x_i's are the nodal position values and N_i's are the shape functions. The Lagrangian is now defined as the sum of the Lagrangian of all elements.

$$L = \sum_{i \in \Psi} L_i \tag{55}$$

In order to discretize the problem in time, the incremental Lagrangian can be defined between t_i^j and t_i^{j+1} where t_i^j is the end of the j'th time-step for element i:

$$L_i^j \approx \int_{t_i^j}^{t_i^{j+1}} L_K dt \tag{56}$$

and the discrete action is:

$$S_d = \sum_{i \in \Psi} \sum_{1 \leq j < n_i} L_i^j \tag{57}$$

and n_i is the total number of time-steps for element i.

By using the principle of stationary action, the following equations are used for solving the physical system:

$$p_i^{s+1/2} - p_i^{s-1/2} = \boldsymbol{I}_i^s \tag{58}$$

in which

$$p_i^{s-1/2} = M_i \frac{x_i^s - x_i^{s-1}}{t_i^s - t_i^{s-1}} = M_i v_i^{s-1/2} \tag{59}$$

p_i^s represents the momentum of node i at time-step s and M_i represents the nodal mass. The terms $\boldsymbol{I}_i^s$ is also defined from:

$$\boldsymbol{P}_j^s = -(t_j^s - t_j^{s-1}) \frac{\partial V_j(x_j^s, t_j^s)}{\partial x_j^s} \tag{60}$$

where V is the potential energy and the term $\boldsymbol{P}_j^s$ represents the impulse of element j during time-step s on its nodes. The term $\boldsymbol{I}_i^s$ in equation 58 can be defined as the component of $\boldsymbol{P}^s$ relevant to node i. Each element, at the end of each time-step imposes an impulse on its nodes, changing their momentum.

This method was implemented by Lew et al. [51] based on a priority queue. Each element has a time-stamp. The element with the smallest time-stamp will be popped out of the queue, updated and then then scheduled in the queue with the new time-stamp.

The following is the pseudo-code for the AVI method:

This method minimizes the work needed for updating elements, however, the use of the priority queue, as in step 5 of algorithm 2, imposes some overhead. In chapters 3 and 4, more comments on the pros and cons of this method are provided.

2.4 Implementation and comparison of sequential algorithms

The sequential version of the naive algorithm, the spatial decomposition algorithm and the AVI algorithm are implemented and the time-performances are compared in this section.

The problem solved here is a quadrilateral with a constant mesh size gradient. The mesh is simply supported on the edges. The mesh is illustrated in Figure 2.8.

Algorithm 2 AVI algorithm

1: Initiate all momenta (p_i) and positions (x_i) to initial values.
2: Initiate all nodal time-stamps t_i to 0.
3: Calculate all intrinsic elemental time-steps dt_k.
4: Initiate all element time-stamps to 0.
5: Add all elements to the priority queue with time-ticket dt_k.
6: Pop an element from the priority queue (the one with smallest time-ticket), will be called element k. If priority queue is empty go to step 14.
7: Update positions: $x_i = x_i + v_i * (t - t_i)$ $\forall$ node $i \in$ element k
8: Update nodal time-stamps: $t_i = t$ $\forall$ node $i \in$ element k
9: If $t > t_f$, then go to step 13.
10: Update velocities: $v_i = v_i - 1/M_i * (t - t_k) \frac{\partial V_k}{\partial x_i}$ $\forall$ node $i \in$ element k
11: Update element time-stamp: $t_k = t_k + dt_k$
12: Push the element into priority queue with time ticket $t_k + dt_k$
13: Go to step 6
14: End of Simulation.

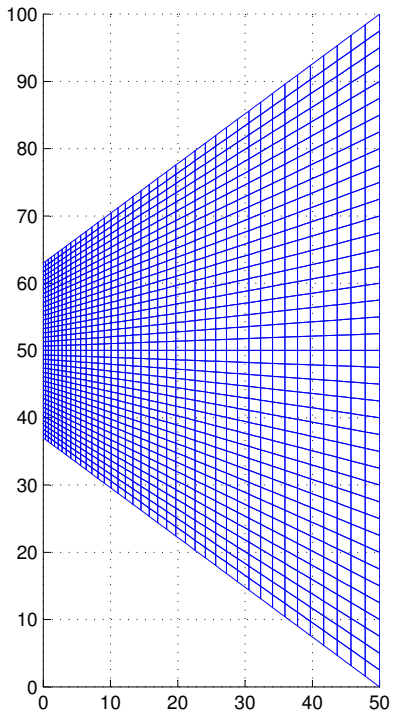

Figure 2.8: The quadrilateral mesh with a constant mesh size gradient

The problem will be solved for a duration of $0.00015s$. The maximum time-step requirement is $0.6\mu s$ and the minimum is $0.08\mu s$. The problem is solved by the three sequential algorithms mentioned above.

2.4.1 The naive algorithm

In this case, all the element time-steps will be the minimum time-step among all elements which is $0.08\mu s$. The run-time is $19.63s$.

2.4.2 The spatial decomposition algorithm

In this case, the maximum time-step will be $0.6\mu s$, and the number of sub-cycling levels is 4. So, all elements will have a time-step of $0.6\mu s$, $0.3\mu s$, $0.15\mu s$, or $0.075\mu s$ whichever is higher and also satisfies their time-step requirement. The run-time in this case is $11.26s$.

2.4.3 The AVI algorithm

In this algorithm, each element can have its unique time-step. The run-time is measured to be $7.96s$.

2.4.4 Conclusion

The the use of the three algorithms clearly shows that choosing the right algorithm can significantly reduce the time-step. So, the study on the best algorithm under different circumstances seems to be necessary. Also, it is not clear which algorithm will perform better if many processing units are available and further study on this is also required. The run-times can be seen in Figure 2.9.

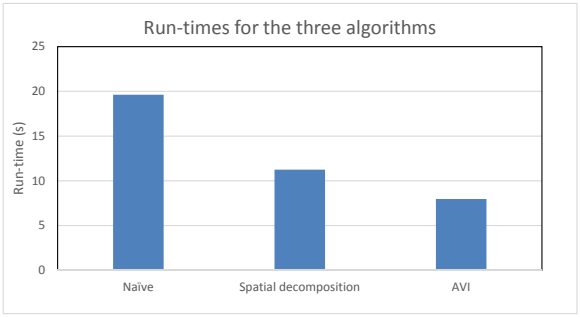

Figure 2.9: Run-time comparison of the three sequential algorithms.

CHAPTER III

PARALLEL PROCESSING IN EXPLICIT FINITE ELEMENTS

In this chapter, different parallel processing hardware and software are discussed, parallel explicit FEM algorithms are introduced and the current direction of computer development in the near future is reviewed.

3.1 Parallel systems

The central processing unit (CPU) and the graphical processing unit (GPU) were developed for different reasons and optimized for specific tasks. The CPU was originally designed to perform sequential computing. In the past decade, the CPU clock frequency has hit the maximum possible in terms of heat production. To overcome this limitation, the manufacturers have developed systems with more than one processor. The CPUs are still considered best for tasks that are mainly Multiple-Instruction Multiple-Data [87] (MIMD) tasks.

The GPU, on the other hand, was originally developed to handle processing graphics on the computer. Originally, the GPU consisted of tens of low-frequency processing units that could handle simple operations that did not require much memory bandwidth. Today, GPUs can have thousands of computational cores with clock frequency up to the GHz order[35]. They are capable of performing native double-precision floating point operations and can have several hundreds of gigabytes per second internal memory bandwidth. The GPUs, for now, still require explicit memory transfer operations between GPU device memory and the host memory (RAM). The GPUs are considered to be best at Single-Instruction Multiple-Data[79] (SIMD) instruction, which is, performing a single set of instructions on multiple data. In the next section, a brief description of the GPU architecture is provided.

3.2 GPU Architecture

A modern GPU consists of several work-groups. Each work-group can have a local cache, local memory (scratch-pad) and several work-items that work concurrently which are identified by the ID assigned to them. The work-group is optimized to access main GPU memory concurrently and the performance will be optimum if the memory access is "coherent", which means the pieces of memory needed by compute units with adjacent ID numbers are stored adjacently in memory. Figure 3.1 depicts a simplified view of the work-groups and work-items.

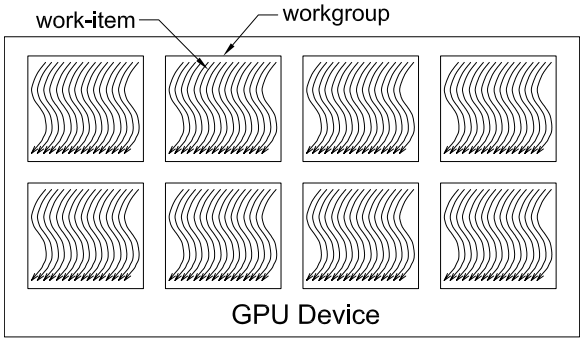

Figure 3.1: A GPU device containing several work-groups and work-items

The GPU device has a main memory. There are also smaller local memories for each work-group that can only be accessed by the work-items of that work-group. Also, each work-item has its own private memory and compute registers that is only accessible by the work-item. There can also be caches at the device level, and the work-group level. The bandwidth between the main memory and the work-groups is one of the important factors in the performance of the device. In Figure 3.2, a more detailed architecture of the device is illustrated.

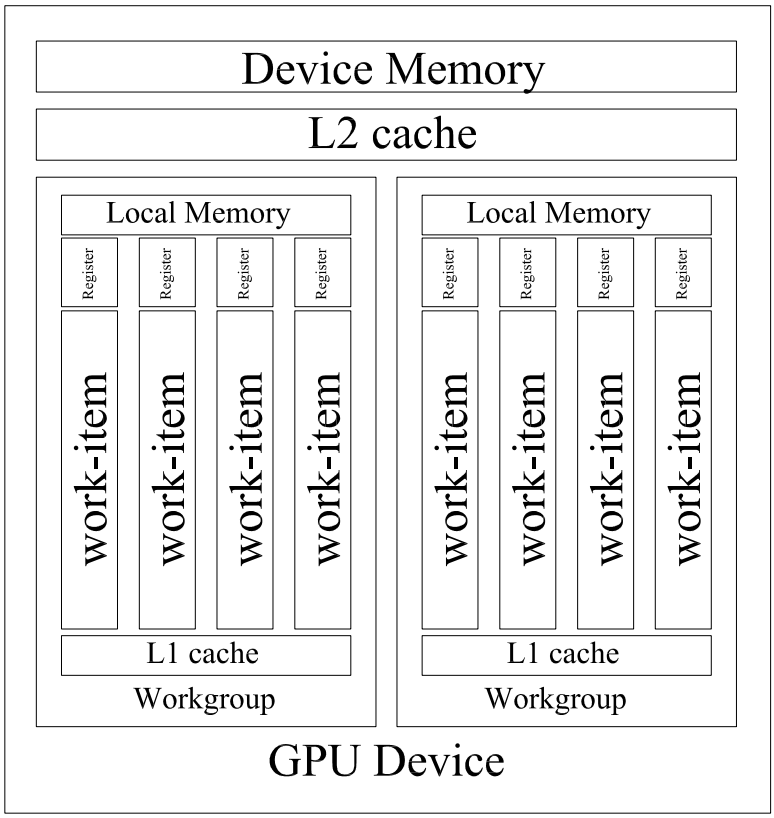

Figure 3.2: Simplified GPU micro-architecture

The set of instructions that is developed to be run by the GPU device is usually called a kernel. This kernel is set to execute by a predefined number of work-groups. The work-groups and work-items are assigned unique id's which are used to identify their share of the work.

3.3 Parallel Explicit FEM

The explicit finite elements method is composed of element by element force and stress updates and node-by-node velocity, acceleration and configuration updates. These tasks are highly repetitive and therefore very suitable to run on a GPU since updating of each element is completely independent of the others and also for the nodes. These properties and the high computational capacity available has made researchers to consider mass parallelization of explicit FEM on shared memory and also distributed memory machines. The three main parallel algorithms described in section 2.3 are implemented and compared in this thesis.

3.3.1 The naive algorithm

As stated in section 2.3.1, the naive algorithm chooses the same time-step for all elements in order to satisfy the stability time-step for all elements.

This choice provides maximum available parallelism, because all elements can always be updated at the same time and the processing overhead (deciding which elements must be updated at each step) is minimal. However, the workload is higher, because potentially, a lot of elements are updated more frequently than needed and the method is very inefficient due to unnecessary updating, so, The existence of a few "small elements" combined with many larger elements can potentially render it useless. This algorithm is ideal when all elements have approximately the same size, because in that case, maximum concurrency and minimum work are achieved simultaneously.

In section 2.3.1, it was shown that the sequential naive algorithm mainly proceeds in the following stages:

1. Calculate elemental forces
2. Calculate new nodal accelerations
3. Calculate new nodal velocities
4. Calculate new nodal positions

In the parallel algorithm, it is possible to calculate all elemental forces independently and concurrently, but one must refrain from adding them directly to the nodal equivalent forces. This is because the elemental forces are calculated concurrently and a race condition, which is simultaneous memory write by more than one computing unit, can result in an incorrect sum of nodal forces. First, one must ensure that the forces of all elements are calculated first and stored separately, and then for each node, sum of the forces are calculated by adding the resultant forces from all connecting elements. After this stage, the internal and external forces for each node are present and it is possible to proceed with stages 2, 3 and 4.

Before proceeding to the next time-step, one must make sure that the position of all the nodes are updated since. The calculation of the forces of the next time-step depends on the positions of the current time-step. Due to this requirement, all the memory transactions must finalize and synchronize at the end of stage 4. The steps of the parallel naive algorithm are shown below:

Algorithm 3 Parallel naive algorithm (For a single time-step)

1: In parallel: Calculate elemental forces
2: Synchronize†
3: In parallel: sum the forces of all nodes
4: In parallel: Calculate new nodal accelerations
5: In parallel: Calculate new nodal velocities
6: In parallel: Calculate new nodal positions
7: Synchronize†

† Synchronization process makes sure that all processors have finished their tasks up to the *Synchronize* command. The process ends when all instructions are finished by all processors and all the read/write operations are also complete.

3.3.2 Parallel spatial decomposition algorithm

Parallelizing the spatial decomposition algorithm, which was stated in Section 2.3.2, basically follows the same parallelization procedure. Algorithm 4 demonstrates the details of the process:

Algorithm 4 Parallel Spatial Decomposition Algorithm

1: Set values of u and v to the initial values for all nodes, $N \leftarrow 0$, $I \leftarrow 0$, $m \leftarrow 1$, initialize all F^{int} and F^{ext} to 0

2: $N \leftarrow N + 1$

3: $I \leftarrow I + 1$

4: Compute $m(I)$ from equation 47

5: In parallel: For all elements k for which $\bar{\phi}_K \geq m(I)$, update the elemental internal forces

6: Synchronize

7: In parallel: For all nodes i for which $\psi_i \geq m(I)$, calculate the equivalent nodal forces (sum the forces of all attached elements)

8: In parallel: For all nodes i for which $\psi_i \geq m(I)$, calculate the new accelerations: $a_i \leftarrow (F_i^{ext} + F_i^{int})/m_i$

9: In parallel: For all nodes i for which $\psi_i \geq m(I)$, calculate the new velocities: $v_i \leftarrow v_i + \Delta T/\psi_i * a_i$

10: Synchronize

11: For all nodes i for which $\bar{\psi}_i \geq m(I)$, calculate the new configuration: $x_i \leftarrow x_i + \Delta T/\bar{\psi}_i * v_i$

12: Synchronize

13: If $I < M$, go to step 3 (the next cycle)

14: If $N < N_{final}$, go to step 2 (the next step)

15: End

In algorithm 4, the parameters used are explained in section2.3.2.

3.4 Viability of Explicit FEM on GPU's

As part of the current research, software is developed to run the naive algorithm on a double-device machine. This means, the explicit solution can be carried out by the use of two devices working simultaneously. The CPUs and one GPU can be the two devices, or two GPUs. First, a brief explanation of the multi-device explicit FEM is presented. Then, the sequential version of the code is compared and verified with other published research.

This study will ensure that the GPU's are suitable for explicit FEM computations and also multi-device performance and the study of the effect of inter-device communications on the total runtime can provide us insight into whether the GPU computations can effectively be extended on more that one GPU. This will significantly make the current dissertation more interesting since the potential for mass parallel processing is evident.

3.4.1 Hybrid multi-device explicit FEM

Many computers today usually possess several CPU cores and a GPU. The computational power of each of the CPU and GPU is suitable for explicit FEM. If they are used simultaneously, speedup can be achieved and larger problems can be solved. The total memory available would be the memory of the RAM and the memory available on the GPU. Today, the GPGPU's (General purpose GPU's) have several gigabytes of device memory. Also, they both share the computation load. OpenCL, a C-based tool for development of kernels for GPUs provides the ability to run the same code on both GPUs and CPUs. It is possible to update some elements on the CPU and some on the GPU and communicate the needed information of the neighbor nodes at the end of the time-step. The algorithm is as follows:

Algorithm 5 The hybrid CPU-GPU naive algorithm

1: Initialize variables
2: Assign elements to two regions, identify the neighbor nodes (user defined). The regions and the neighbor nodes are illustrated in Figure 3.3.
3: $N \leftarrow 0$ //step counter
4: $N \leftarrow N + 1$
5: In parallel: Calculate Forces of all elements
6: Pack the force to be transfered ▷ As shown in Figures 3.3 and 3.4
7: Transfer the force to the other device. See Figure 3.4
8: In parallel: Sum up and calculate the equivalent forces for all nodes
9: In parallel: Calculate new nodal accelerations
10: In parallel: Calculate new nodal velocities
11: In parallel: Calculate new nodal positions
12: If $N < N_{final}$ GOTO step 4
13: End

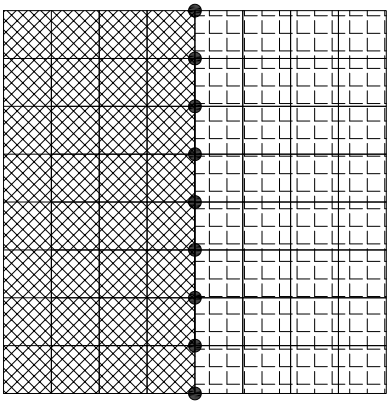

Figure 3.3: The two regions and the neighbor nodes

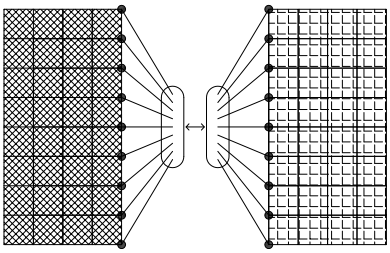

Figure 3.4: The forces on the boundary are packed and sent to the other side, so the total force on the boundary elements can be calculated

3.4.2 Verification

For verification purposes, the following problem is solved by the *naive method* extended to use more than one device and compared with other published papers. A $10inch$ x $10inch$ simply supported plate with $0.5in$ thickness subjected to a sudden uniform pressure of 300 psi with the following material properties (See Figure 3.5):

$\rho = 0.2589 * 10^{-3} lb.s^2.in^4$

$E = 10^7$ psi

$\nu = 0.3$

$\sigma_y = 30000$ psi

$E_T = 0$

In which ρ is the density, E is the modulus of elasticity, ν is the Poisson's ratio, σ_y is the yielding stress and E_T is the hardening parameter. The finite element used here is an 8-node Mindlin-Reissner plate element. In plastic analyses, a 6-layer mid-ordinate rule is applied. Also, to avoid shear locking, the reduced integration method for shear stresses is applied (selective reduced integration).

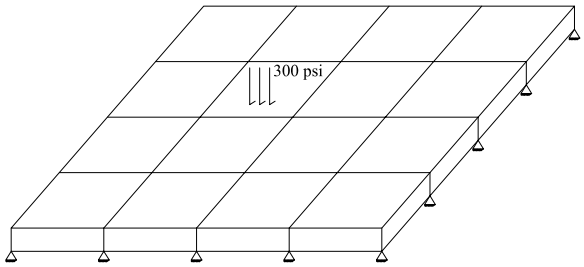

Figure 3.5: The simply supported plate with 16 Mindlin-Reissner plate elements

The elastic problem comparison is depicted in Figure 3.6. Owen et al.[67] and Huang et al. [33] used implicit dynamic analysis to solve the problem. The time-step used in the current work results is $1\mu s$ based on the stability time-step requirement. Other studies have also used this as a benchmark problem[33, 63, 67].

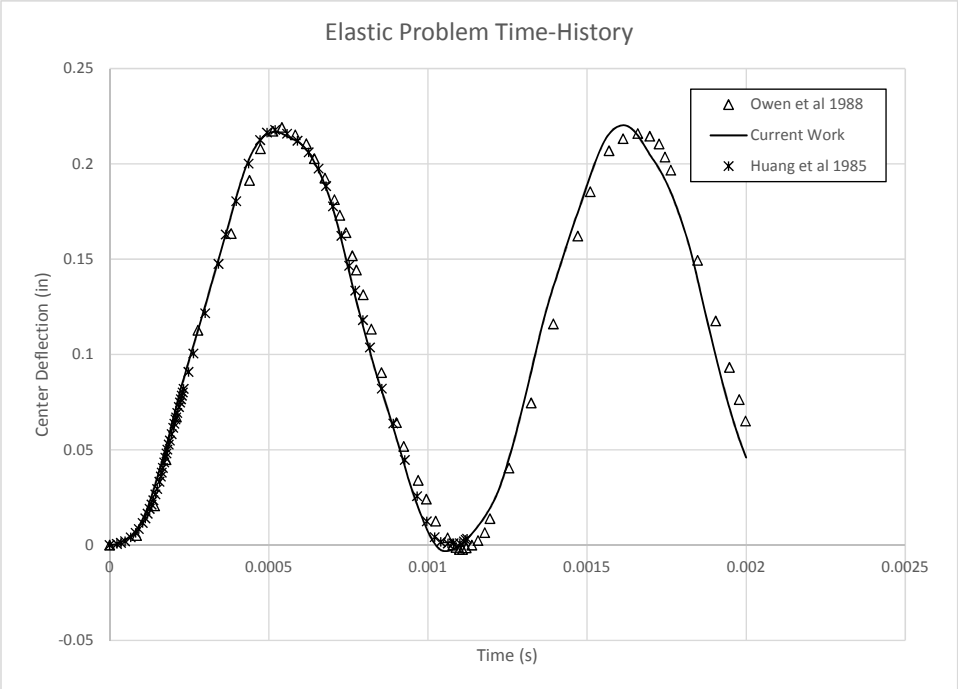

Figure 3.6: The elastic problem comparison with Owen et al. [67] and Huang et al. [33]

The plastic problem is compared and illustrated in Figure 3.7:

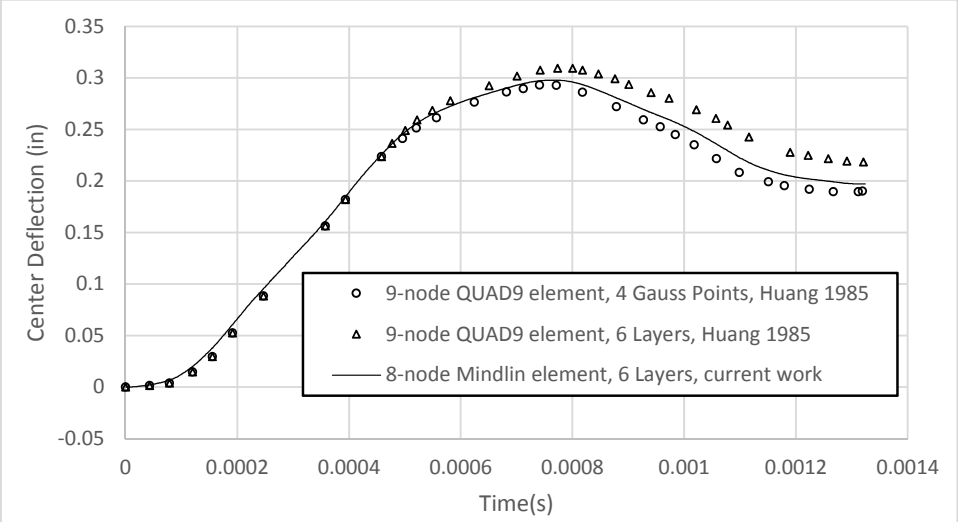

Figure 3.7: The plastic problem comparison with Huang et al. [33]

The results are compared with results provided by Huang et al.[33] who solved the problem by two methods: A) using 4 gauss points through the thickness, and B)using the mid-ordinate rule with 6 layers. Although Huang used QUAD9 elements and did not use a diagonal mass matrix as in the present study, the maximum displacements are within 5%.

3.4.3 Test cases

The problem chosen to be solved here is a plate meshed with 400x400 Mindlin plate elements (total 160,000 elements). A mid-ordinate rule is used to integrate through the thickness. The plate is simply supported. Three test cases are provided here. At each case, 2 devices (1CPU and 1GPU, or 2GPU's) are used to solve the problem. The percentage of elements given to device 1 (work-share of device 1) will be denoted by α which is always between 0 and 1. Figure 3.8 shows a simple physical domain and the work-shares of both devices.

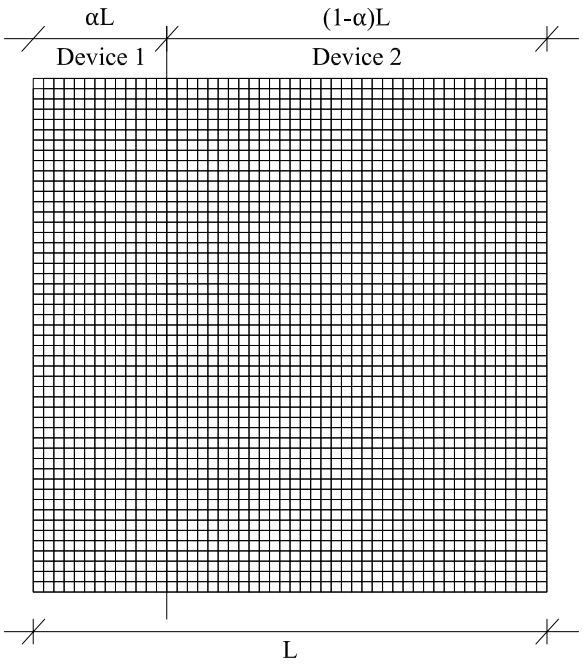

Figure 3.8: Work-share of the two devices

For more insight into the run-times, the timings have been provided for different stages of the computation:

- Stage 1: Calculating element-wise internal forces
- Stage 2: Calculating Nodal internal forces by summing elemental forces at each node
- Stage 3: Communicating the internal forces of the nodes on the boundary of computation regions
- Stage 4: Updating nodal velocities and the configuration

3.4.3.1 Case 1

◇ Device 1: A 6-core AMD phenom II X6 1045T CPU, 2.7GHz, 8GB RAM,

◇ Device 2: ATI Radeon HD 5830 GPU, 1GB DDR5 memory, engine clock: 800 MHz, memory clock: 1GHz, Memory Bandwidth: 128 GB/s, 1120 Stream Processing Units

Table 3.1: Run-times for running on Device 1 (CPU) only

Task	**Time Spent(s)**	**Runtime %**
Stage 1	17.55	74.7
Stage 2	1.52	6.5
Stage 3 (Communication)	0.00	0.0
Stage 4	4.41	18.8
Total	23.48	

Table 3.2: Run-times for running on Device 2 (GPU) only

Task	**Time Spent(s)**	**Runtime %**
Stage 1	19.50	96.1
Stage 2	0.20	1.0
Stage 3 (Communication)	0.00	0.0
Stage 4	0.59	2.9
Total	20.29	

Next, the problem is run in hybrid mode on both devices simultaneously. Figure 3.9 shows runtime of each device individually and the overall (hybrid) runtime for different work-share ratios using the previously defined work-share parameter α.

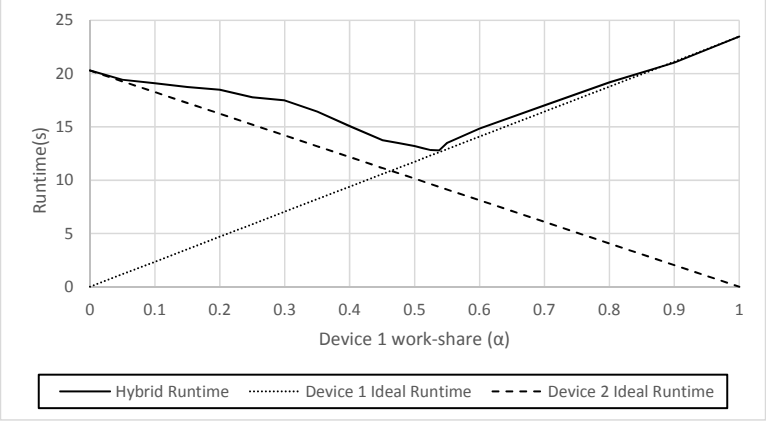

Figure 3.9: Case 1 Hybrid Performance

The hybrid simulation runtime is at least as much as maximum ideal runtimes of all devices, because the ideal device runtime is the time each device needs to finish its own fraction of the job without any interruption or synchronization. When more than one device is available, there is a need for synchronization and communication.

Furthermore, different devices with different architectures behave differently at various stages of the program. For instance, stages 2 and 4 need very low memory transfer from device memory to computing registers. This is ideal for GPUs and the speedup of GPU to CPU is more at these stages compared to stage 1, which is memory-bandwidth intensive. So, the optimal work balance ratio of stage 1 is different than stage 4, but a single work balance ratio is needed and so cannot reach perfect load balance.

In this case, the hybrid performance is still very good and the hybrid runtime is decreased by 60% compared to the runtime for device 2, which has the lower individual performance.

3.4.3.2 Case 2

In case 2, the same CPU and a different GPU is used:

- ◇ Device 1: A 6-core AMD phenom II X6 1045T CPU, 2.7GHz, 8GB RAM,
- ◇ Device 2: AMD Radeon 7970 GPU, 3GB DDR5 memory, engine clock: 925 MHz, Memory clock: 1375 MHz, memory bandwidth: 264 GB/s, 2048 Stream Processing Units.

Table 3.3: Run-times for running on Device 1 (CPU) only

Task	**Time Spent(s)**	**Runtime %**
Stage 1	18.585	75
Stage 2	1.58	7
Stage 3 (Communication)	0	0
Stage 4	4.51	18
Total	24.67	

Table 3.4: Run-times for running on Device 2 (GPU) only

Task	**Time Spent(s)**	**Runtime %**
Stage 1	5.190	92.4
Stage 2	0.18	3.2
Stage 3 (Communication)	0.00	0.0
Stage 4	0.24	4.3
Total	5.62	

Next, the problem is solved using both devices. Device 2 seems to be much faster. Since the work will be shared between the devices based on their speed, it is reasonable to split the work based on each device's individual runtime. So, Device 1 will roughly get 10% of the workload and Device 2 will get the remaining 90%.

Table 3.5: Hybrid runtime comparison of all computation steps

	Time Spent		
Task	**Dev1(CPU)**	**Dev2(GPU)**	**Max of Dev1 & Dev2**
Stage 1	3.89	5.43	5.43
Stage 2	0.37	0.16	0.37
Stage 3 (Communication)	0.03	0.03	0.03
Stage 4	0.99	0.21	0.99
		Total	6.82

Since both devices have to advance together, if a device finishes one of the stages faster, the other has to wait for it to finish, and then continue. So, the total runtime would be the sum of maximum time each device spent at each stage.

The Speedup in this case versus the results of the GPU-only run is negligible. There are two reasons for this: First, the perfect load balance is not achievable and second, the speed of the two devices are not comparable.

As the first reason, the GPU is much better at stages 2 and 4, because data coherence and little local memory bandwidth requirements are present. These stages are ideal for a

GPU. Stage 1 needs high bandwidth between device memory and computing registers, so, this stage has a different GPU to CPU speed ratio than other ones.

Table 3.6: Speed ratio for different stages

	GPU to CPU speed ratio
Stage 1	0.71
Stage 2	2.31
Stage 4	4.71

Since 80% of the work is done by the GPU, it is still performing faster on a per element basis.

Table 3.7: GPU to CPU per element/node speed ratio

	CPU time per element/node (s)	**GPU time per element/node (s)**	**Ratio**	**Stage workload / Total work**
Stage 1	121.56	42.42	2.87	0.80
Stage 2	11.56	1.25	9.25	0.05
Stage 4	30.94	1.64	18.86	0.15

In case 2, good speedup is not achievable from the hybrid action. This is mainly due to two reasons:

1. As stated above, different stages of the computation require different load balance ratios, so, the perfect speedup is not achievable. But since 80% of the work is due to stage 1, this is not a big problem and if the two devices are closer in terms of computational power, good speedups can be achieved.

2. This particular GPU is much faster and more capable than the used CPU device for this purpose. The two devices are very different in computational throughput capacity and therefore, a lot of the work has to be assigned to device 2, resulting in the same run-time as the run-time of device 2.

Figure 3.10 depicts the hybrid performance versus different work-shares.

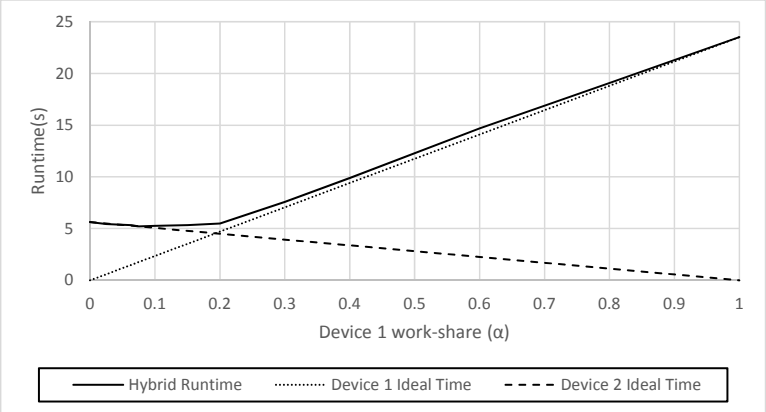

Figure 3.10: Case 2 Hybrid Performance

3.4.3.3 Case 3 (Two GPU's)

◇ Device 1: nVidia Telsa M2090 FERMI GPU, Processor core clock: 1.3 GHz, Memory:6 GB, Memory clock : 1.85 GHz, Number of processor cores: 512

◇ Device 2: (Same as device 1)

Table 3.8: Run-times for running on a single GPU

Task	**Time Spent(s)**	**Runtime %**
Stage 1	14.56	97
Stage 2	0.19	1.26
Stage 3 (Communication)	0	1
Stage 4	0.26	1.7
Total	15.01	

Next, both devices will be run in hybrid mode (Work shared equally):

Table 3.9: Hybrid runtime comparison of all computation steps

	Time Spent		
Task	**Dev1(GPU1)**	**Dev2(GPU2)**	**Max of Dev1 & Dev2**
Stage 1	7.37	7.36	7.37
Stage 2	0.10	0.10	0.1
Stage 3 (Communication)	0.002	0.002	0.002
Stage 4	0.14	0.15	0.15
		Total	7.62

As seen in this case, since the two devices are identical, their performance as a hybrid computing system is very good and the speedup by the 2 GPUs is 1.97. The slight loss of performance is due to the overhead of launching more kernels and communications. Figure 3.11 depicts the hybrid performance versus different work-shares.

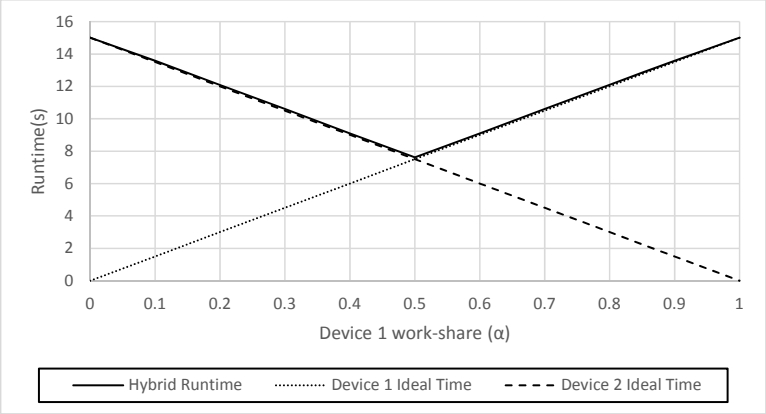

Figure 3.11: Case 3 Hybrid Performance

3.4.4 Hybrid Behavior Conclusion

As explained above, the hybrid action was tested for 3 different situations: 1) a GPU and a CPU, with almost equal computational capacity, 2)a CPU device and a much stronger GPU, and 3) two identical GPU's.

In the first case, the results were satisfying and the best-case individual runtime was reduced by 60%. Still, the ideal run-time was not reached. That is because different devices

were better at different parts of the computational stages. This prevented a perfect work-balance. However, the speedup was still satisfactory and practical.

In the second case, the GPU was much more powerful than the CPU when running the problem individually (4.5 times faster). So, the hybrid performance was not satisfactory, because when one worker is significantly faster, sharing the work was not worth the overhead.

In the last case, two identical GPUs behaved perfectly as a system, because they performed equally at different computation stages and almost perfect work balance was possible and the hybrid runtime was halved.

The results for case 2 encourage using CPU and GPU as a hybrid system for explicit FEM when the individual run-time of both devices are close. Solving small problems on both devices will help tuning the work-shares before starting a problem with large number of elements. Also, using more than identical GPU's proved very promising in case 3.

3.5 The future of Parallel Processing

In the next decade, predictions state that computers with the ability to compute 10^{18} floating point operations per second (Exaflop/second) will be introduced, which are orders of magnitude faster than today's computers [92]. These machines will be used in all areas of science including climate, materials, biology, etc. Many computer scientists believe that the architecture of these machines will dominantly look like GPUs [92], which are more energy efficient and more scalable [14, 92].

As mentioned in section 1.3.2, there is still need for computational capacity in explicit FEM such as mesh adaptivity, the ability to solve for multiple load cases, simulating longer phenomena, using finer mesh, etc. The next decade will bring an extraordinary opportunity for computational mechanics and in particular dynamic structural analysis.

Currently, there is no study of the future computer systems and the effect of computer architecture in computational mechanics. Since the performance of all algorithms depend on the computing hardware details, understanding the future of computers will help in designing new algorithms.

During the course of this research, using the predictions of computer scientists on future

computer hardware along with the performance relationships of the AVISD algorithm, the performance this algorithm in the future is predicted in Chapter 6.

Developing this understanding helps engineers in two aspects: First, the class of algorithms that are viable in the future are predicted. This will help focus future algorithm design to match the upcoming computational systems. Second, this gives us an estimation about the size and limitation of the problems that can be solved in the near future.

CHAPTER IV

NEW PARALLEL AVI ALGORITHMS

The AVI method proposed by Lew et al. [53], which was stated in Section 2.3.3, is inherently a sequential method. The method works based on a priority queue and one step at a time, the element with the lowest time-stamp is updated.

Kale and Lew [38] later stated that an element only needs to have a local minimum time-stamp to qualify for updating. They divided the mesh into sub-regions and one processor would update the elements of its own region. In their algorithm, a waiting list is also present for elements that are missing data from the neighbor regions and a mail-delivery system for communication is used. The algorithm is mainly developed for distributed machines and MPI is used for communications over the network. They showed that their system is scalable if there is a large number of elements in each region.

Huang et al. [34] in 2007 showed that if the maximum connectivity level at each node is d and number of elements is N_e, there is at most $N_e/(d+1)$ available parallelism. They used the same concept as Kale et al. in dividing the mesh into sub-regions, except they also used a shared memory parallelism at each node where a master node finds the updatable elements and then the local processors in parallel update them.

Subsequently, Huang et al. tried to group elements into zones and solve each zone on a separate CPU node. The time-step for each zone was constant and chosen to be the minimum required time-step of all its elements. This makes the required work to increase because some elements and occasionally most elements are updated with a time-step smaller than they need, however, the zoning process allows more concurrency and also the position and velocity of all nodes can now be updates concurrently. Their implementation showed a performance boost in the test cases studied.

Since the concurrency appears to be a big factor in GPU's, in the search for the most optimum algorithm, this method must be included.

However, in this research, another very similar method is used. In this work, an algorithm called the AVI Spatial Decomposition (AVISD) algorithm is developed that chooses some bins based on the time-step and puts all the elements in relevant bins (bin with biggest time-step smaller than inherent element time-step).

Grouping elements based on time-steps reduces the work load and also automatizes the binning process. Also, it gives the chance to change the size of the bins and the bin choice based on the state of the mesh. Possibly in some cases, having a single bin is enough and in others, bins with different sizes might provide a better performance. The mentioned reasons, among others, are the reasons behind choosing the bins based on time-step and not the physical placement of the elements. Further details on the implementation of the AVI Spatial Decomposition (AVISD) Algorithm is provided in section 4.4.

The parallel AVI methods are the focus of the major part of this thesis and the parallel spatial AVI algorithm is one of the major contributions of this research.

4.1 Parallel AVI Coloring Algorithm

An AVI coloring algorithm is introduced here. This method does not prove to be viable on a GPU, but this algorithm gives ideas and insights on how a more efficient and more realistic parallel AVI algorithm that will work well on a GPU can be designed.

Updating two non-neighbor elements can always be done in parallel, since they do not share nodes and also whether or not one is "updatable" does not depend on the other. Based on this fact, it is possible to group all the elements into a relatively small number of groups where no two members of each group are neighbors. Then, all elements of each group can be safely updated in parallel (if they are updatable). This grouping of objects is usually referred to as the coloring problem, where each group assumes a color and no two elements of the same color touch. An example is depicted in Figure 4.1.

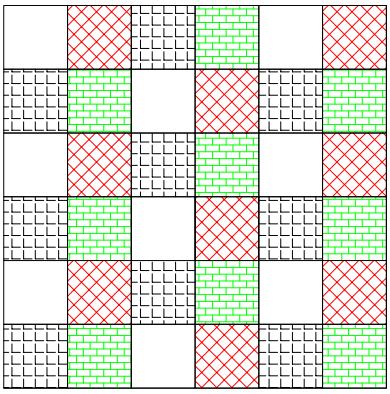

Figure 4.1: Quadrilateral mesh with connectivity degree 4 and 4 colors.

To check whether or not each element is updatable, a *future* time-stamp will be assigned to each node and a *now* time-stamp to each element. Each node's *future* time-stamp will start at minimum attached neighbor elements' updating time-step. Each element's *now* time-stamp will start at 0. Also, each node has a *last* time-stamp, which indicates the last time the node was updated. The *last* time-stamp is initiated to zero.

At each instance, the element is checked if the elemental time-step plus the *now* time-stamp was less than or equal to all its nodes' *future* time-stamp, then it will be updated and the *future* time-stamp of all its nodes are changed to *now* plus the element's time-step. Since the element with the smallest time-step will be updated every time, the maximum number of steps is $N_{max} = ceiling(t_{final}/\delta t_{min})$ and at each step, all colors will be checked sequentially.

The AVI coloring algorithm is now explained in more detail:

Algorithm 6 AVI Coloring Algorithm

1: All elements are grouped into minimum possible number of color groups (C_1, C_2, ..., C_d), which d is the connectivity degree of the mesh.

2: $\forall$ element k, the time-step δt_k is computed and the *now* time-stamp is zeroed:

$t_k^{now} \leftarrow 0$

3: $\forall$ node i, the *future* and the *last* time-stamp are initiated:

$t_i^{future} \leftarrow \min(\delta t_k)$ $\forall$ element k connected to node i

$t_i^{last} \leftarrow 0$

4: $\forall$ node i, $u_i \leftarrow u_i^0$, $v_i \leftarrow v_i^0$

5: $N_{max} \leftarrow ceiling(t_{final}/\delta t_{min})$

6: $N \leftarrow 0$ // The step counter

7: $N \leftarrow N + 1$

8: $I \leftarrow 0$ // The iterator through colors

9: $I \leftarrow I + 1$

10: Parallel region start $\forall$ element k in color list C_I:

11: $\forall$ element k, IF [$t_k^{now} + \delta t_k \leq \min(t_i^{future})$ $\forall$ node $i \in$ element k] THEN GOTO step 17

12: Update positions: $x_i \leftarrow x_i + v_i * (t_k^{now} + \delta t_k - t_i^{last})$ $\forall$ node $i \in$ element k

13: Compute stresses, forces and accelerations: $a_i \leftarrow -\frac{\partial V_k}{\partial x_i}/M_i$ $\forall$ node $i \in$ element k

14: Update velocities: $v_i \leftarrow v_i + a_i * \delta t_k$ $\forall$ node $i \in$ element k

15: $t_k^{now} \leftarrow t_k^{now} + \delta t_k$

16: $t_i^{last} \leftarrow t_k^{now}$, $t_i^{future} \leftarrow t_k^{now} + \delta t_k$ $\forall$ node $i \in$ element k

17: Synchronize

18: End parallel region

19: If $I < d$ GOTO step 9

20: If $N < N_{max}$ GOTO step 7

21: End of Simulation

4.2 AVI Coloring Algorithm Performance

The AVI coloring algorithm did not provide enough performance boost. In some cases, it even matched the results of the naive method.

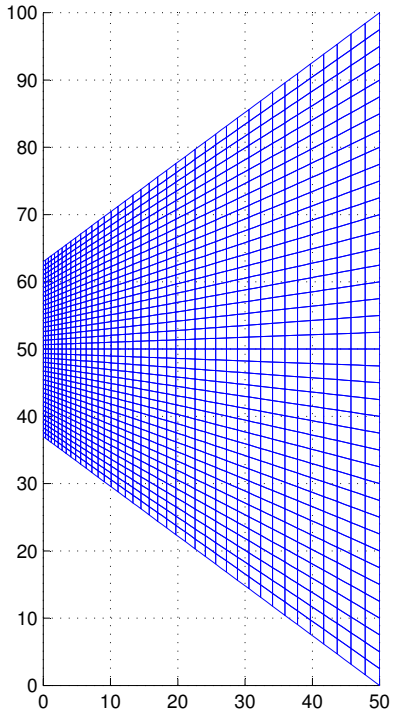

Figure 4.2: Testing AVI coloring algorithm

Here, a mesh of Mindlin-Reissner plate elements with 160,000 elements and simply supported on the corners is subjected to a uniform load and simulated for 0.005 seconds of physical time.

Table 4.1: Some GBT versus conventional terms

	naive	**Spatial**	**AVI Coloring**
runtime(s):	14.1	7.2	13.8

4.2.1 Discussion

As can be seen in Table 4.1, the current AVI coloring algorithm turns out to be extremely inefficient on a GPU. There are several reasons for this:

1. The process of checking all elements of a specific color is very inefficient, because most of the time, it's only the smallest elements (about 2% of all) are being updated while time is spent to check if any element is updatable.

2. As will be described in section 4.3, the dependency of elements on each other for getting updated will lead to only a few updatable elements most of the time. If only a handful of elements have the minimum time-step requirement, then at most steps it's only those elements that are being updated. Since there are hundreds to thousands of cores on a GPU, much of the computational potential in wasted.

3. Since the updated elements are not known beforehand, the data coherence is an issue. The elements that are updated simultaneously have the required data (Nodal position, stress, ...) at different parts of the memory and this makes the memory transfers much more costly.

4. Since each element is updated with its nodes and no neighbor elements are updated together, each node has to be updated once per attached element. This makes the process of updating the position and velocity of the nodes much more demanding.

4.3 Motivation for a new parallel AVI algorithm

The reasons for inefficiency of the coloring algorithm provided insight into the design of a proper GPU friendly AVI algorithm. In this section, some issues regarding the AVI algorithm and some possible solutions are provided.

4.3.1 Task dependency flow-chart

A task dependency flowchart is a flowchart that demonstrates how different computational jobs are predicated on each-other. This chart, like a Gantt chart shows the order which the tasks must be preformed to finish the simulation.

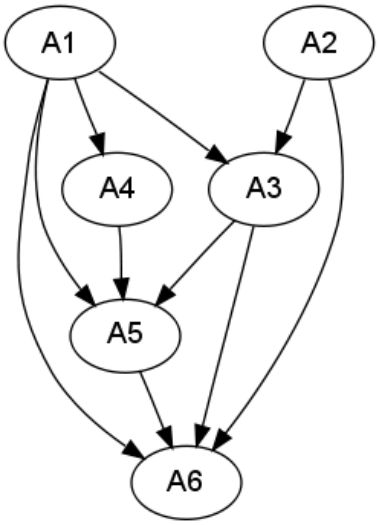

Figure 4.3: A sample Task Dependency Chart

In Figure 4.3, it is evident that the job is composed of 6 tasks A1 to A6. Each arrow defines a dependency and point from the prerequisite task to the next task.

The path A1-A4-A5-A6 is called the critical path, because it is the longest dependency path among all others. The critical path or alternatively called depth (D) is a defining factor in the parallel performance of problems. D Shows a path through which parallelizing is impossible.

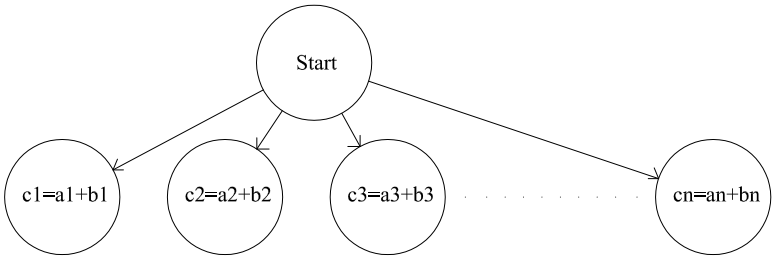

Figure 4.4: Task dependency flowchart for a vector addition

As seen in Figure 4.4, in some jobs such as adding two vectors together, the tasks (adding two corresponding single elements) are completely independent. Such problems are sometimes called embarrassingly parallel algorithms. In this case, $D = 1$.

On the other hand, a series of some jobs which each step's input depend on the output

of the previous step cannot be done in parallel. In this case, $D = N$ in which N is the total number of steps.

In practice, nearly all computational jobs are between the two cases above and D is a number between 1 and N. Brents Theorem[28] explains how the total number of steps required to finish a parallel job are related to problem parameters:

$$S = \frac{(N - D)}{p} + D \tag{61}$$

In which N is the total number of operations, D is the depth, p is the available number of identical parallel computing units and S represents the minimum number of steps it takes to finish the job.

This theorem states that the total work can be distributed among the processing units, but the depth is also a factor that makes the needed sequential steps larger no matter how many parallel computing units are available.

This concept is a motivation to introduce a balance between total work and concurrency. It can a good idea to decrease depth in cost of adding the total amount of operation, or work.

The process of the AVI method can also be described by a task dependency diagram such as the one in Figure 4.3.

The elements with the smallest time-steps are updated more frequently and among them, the elements with minimum time-step are updated at every step and therefore are located on the critical path.

In the AVI dependency flowchart, elements with smaller time-steps are the prerequisite for their physical neighbors (other elements with which they share nodes) and there is an arrow from each smaller element to its larger neighbor. This phenomenon makes the updating process of elements become a lot more sequential in nature and concurrency drops highly in most cases.

In extreme cases such as having a linear change in mesh size and inherent time-steps, updating each of the linearly varying elements that are connected to each-other is predicated

to updating its smaller neighbor. Thus, they form a queue for getting updated and the parallel behavior is completely gone. Although minimum work is spent on each element for updating it as each element is not updated with a frequency higher than it physically requires, the concurrency is minimum. As a result of low concurrency, many potential computing units will remain idle. If there is not enough work at each step to keep all computing units busy, the performance drops.

4.4 Parallel AVI Spatial Decomposition (AVISD) Algorithm

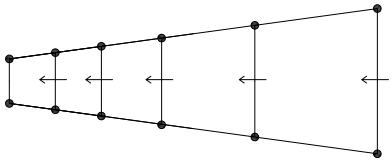

Figure 4.5: Dependency relations between elements for AVI algorithm. Each arrow is pointing to the prerequisite element

In Figure 4.5, the classical AVI method can only operate sequentially because of the dependency chain. If all elements are updated with the smallest required time-step (the naive method), the chain behavior will go away and more available parallelism is achieved. At the same time, elements that do not require the minimum time-step are updated more frequently. It is not readily evident which approach will be faster. In fact, there is a third approach in which the elements can be grouped spatially into different zones and update each zone concurrently.

In this new approach, all elements in each zone can be updated at the same time, even the neighbor elements. This fact allows less memory transfer because the shared nodes between two elements in the same node can be updated once, instead of once per each element.

These ideas lead to the AVI spatial decomposition (AVISD) algorithm. In this algorithm, the elements are grouped into bins based on their required time-step. All elements in a single bin are updated concurrently with the same time-step. Regarding this algorithm, the following points are worth mentioning:

1. The bins are chosen based on the time-step requirement, not based on the physical placement of the elements. Since patches of elements with the same time-step requirement could exist in different regions of the mesh, putting them in the same bin increases concurrency.

2. Since in practice the elements exist as patches of same or close sizes, binning of elements with the same size also means that those elements exist as neighborhoods, not just single separate elements

3. If some element are in a different bin than all their neighbors, the algorithm still works fine. The only issue would be the increased nodal updates. These updates are a minor part of the simulation. If these elements are a minority, which is true in nearly all practical applications, this does not compromise the efficiency of the algorithm.

4. The configuration of the bins, i.e. number of bins and size of each bin can potentially affect the runtime by a big factor. An appropriate optimization algorithm must be used to decide the best bin combination.

5. The choice of bin configuration depends on the problem. For example, for a problem with identical elements, choice of one bin is trivial, which is not the case with meshes with highly varying mesh sizes. The choice of bins are therefore a mesh dependent factor.

Here, the AVISD algorithm is explained assuming the bin configurations are a given. The choice of bins will be discussed later in section 5.8.

In the AVISD algorithm, there are b bins, which cover the time-step range.

Figure 4.6 shows the time-step configuration of a specific mesh. The elements are sorted based on the inherent time-step. The elements are put in five bins. The hatched areas indicate the amount of difference between the inherent element time-step and the assigned new time-step when the element is put in a specific bin.

The AVISD pseudo-code is presented here as Algorithm 7:

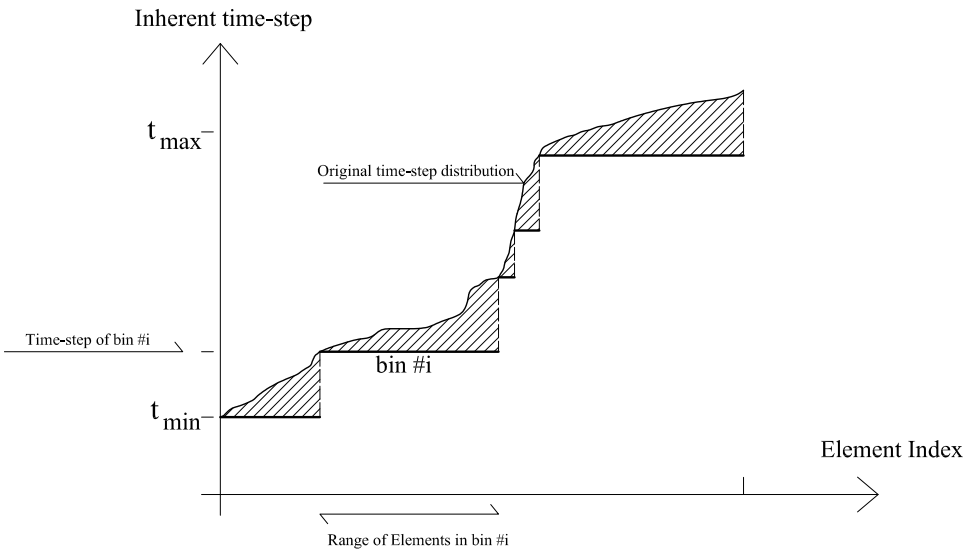

Figure 4.6: Sample bin configuration

Algorithm 7 AVI spatial decomposition (AVISD) Algorithm

$B \leftarrow \{t_1, t_2, \cdots, t_{b+1}\}$ //B is the bins timestep vector where t_i and t_{i+1} form the boundary of bin $\#i$ and so on. b is the number of bins
Put elements in Bins. Element e goes to bin $i \Leftrightarrow (t_e^{inherent} \geq t_i) \wedge (t_e^{inherent} < t_{i+1})$
Put all nodes attached to elements in bin i in group N_i
Find all neighbor bins (Bins that have common nodes) of bin i and put them in vector z_i
$T \leftarrow 0$
$\forall$ bin $i, \tau_i \leftarrow t_{b_i} \leftarrow 0$ ▷ τ_i is bin $\#i$'s next update time stamp
while $T > T_{Ultimate}$ **do** ▷ $T_{Ultimate}$ is the final simulation time
    $U \leftarrow \emptyset$ ▷ U is the set of updatable bins
    for all bins b_i **do**
        $Updatable \leftarrow 1$
        for all bins b_j **do**
            if $b_j \in z_i$ **then**
                if $\tau_j < z_i$ **then**
                $Updatable \leftarrow 0$
                EXIT FOR
        if $Updatable == 1$ **then**
            Add i to set U
    for $\forall I \in U$ **do**
        for $\forall$ node $i \in N_I$ **do**
            Update position of node i
        for $\forall$ element $i \in b_I$ **do**
            Update forces of element i
        for $\forall$ node $i \in N_I$ **do**
            Update velocities of node i

4.5 Verification of the AVISD Algorithm

In order to verify the AVISD method, different problems with 1 or more number of bins have been tested. The number of bins, as expected does not have any effect on the results.

The validity of the AVISD method depends on this fact since the bins must not effect the energy levels and the physics of the problem. If the inherent minimum required time-step of each element in a particular bin is greater than or equal to the bin time-step, a stable behavior is expected and the results must be the same up to engineering significant precision.

In order to verify the AVISD solution method with another program, a 1600 Mindlin plate element model with linearly varying element sizes is compared with an explicit solution in Abaqus using shell elements.

The model and the mesh are shown in Figure 4.7. The edges of the plate are simply supported. The material constants and the thickness used are shown below:

Material: Steel, E=29000 Ksi, ν=0.3, Thickness=1 in.

The load is a uniform instantaneous pressure of 1 psi applied over the entire surface and normal to the plate that is constantly applied throughout the simulation. In Abaqus, as is the case for the AVISD method, an explicit dynamic analysis has been performed and the vertical displacement of a particular node of the mesh is measured.

The red dot in Figure 4.7 indicate the point where the transverse displacement is compared between Abaqus and the current study. Figure 4.8 shows a comparison of the displacement at this point over time between Abaqus and the current study.

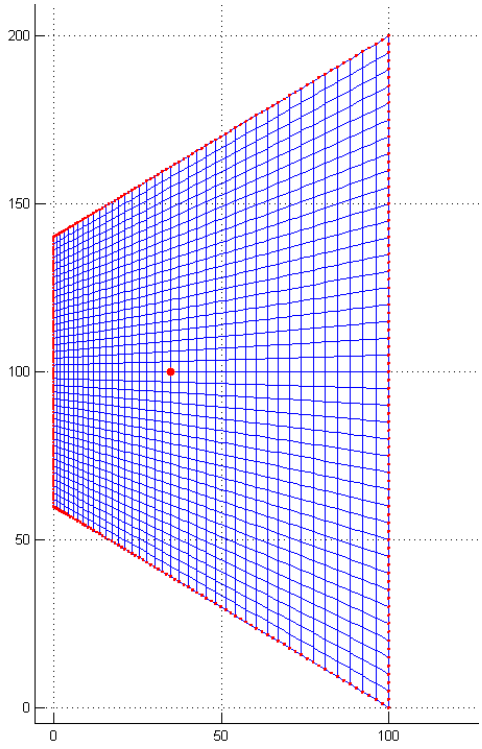

Figure 4.7: 1600 Mindlin plate elements run with paraDyn, the AVISD algorithm and 5 equal sized bins, versus Abaqus shell elements. The displacement will be measured at the red circle.

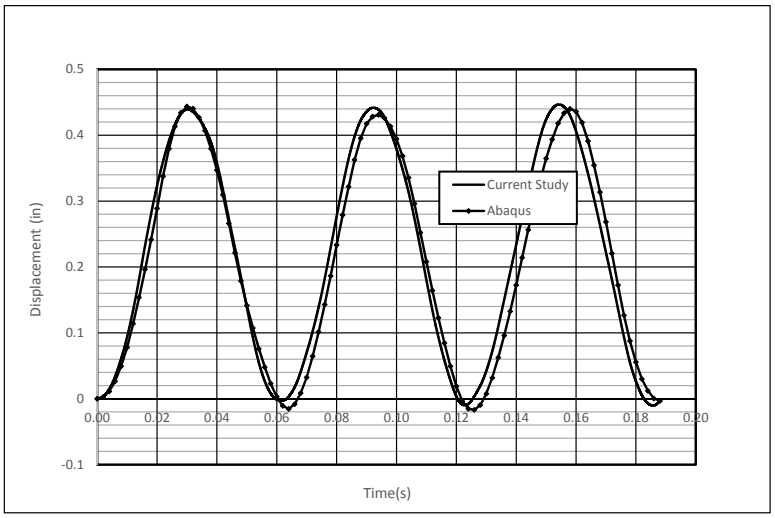

Figure 4.8: Model displacement comparison with Abaqus. The solid line is current research and the line with diagonal markers is Abaqus solution. (Inches units)

Figures 4.9 and 4.10 are the deformed shapes from Abaqus and the current study (TECPLOT visualization). As can be seen in Figure 4.8, the results are very close. The possible reasons for the small differences in the results can be attributed to the different element formulation. The current study uses 8-node Mindlin-Reissner plate elements with reduced integration while Abaqus uses 4-node S4RS shell elements with the shear stiffness given by the user as an input.

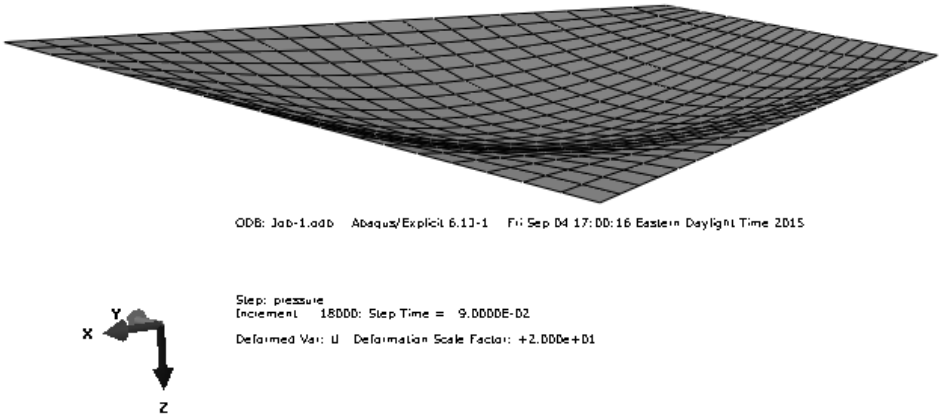

Figure 4.9: The deformed shape of the plate from Abaqus software.

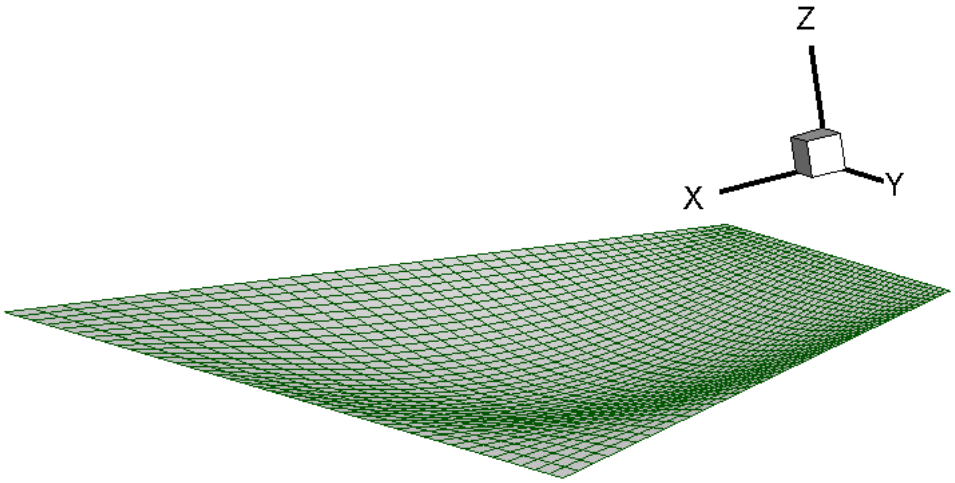

Figure 4.10: The deformed shape of the plate from current study, illustrated by the TECPLOT software.

4.6 Generality and flexibility of the AVISD Algorithm

In the previous sections, the details of the AVISD method were elaborated. At this point some interesting aspects of the AVISD method are worth mentioning.

In the AVISD method, one has the flexibility to choose the bin configuration as desired. The AVISD then compares to the other algorithms as follows:

1. Choosing one bin: This choice will lead to the naive method. So the naive method can be considered a special case of the AVISD method

2. Choosing doubling bin sizes:

$$b = \{\Delta T, 2\Delta T, 4\Delta T, ...\} \tag{62}$$

   This choice will lead to a method like the classical spatial decomposition method. So again, the spatial decomposition method is another special case of the AVI method.

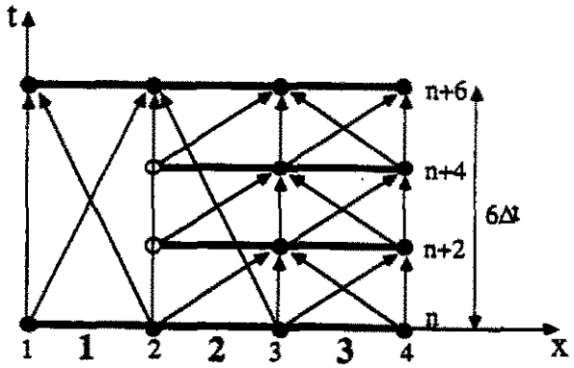

Figure 4.11: Neal and Belytschko (1989)[60]

Figure 4.11 shows bar elements, with different time-steps. The "y" direction shows advancement through time. element 1 is updated with time-steps three times the time-steps of elements 2 and 3, because of different time-step requirement.

3. In meshes with clear regions of different sizes, each region with a specific size can be chosen to be a bin. This encompasses the methods that Belytschko et. al.[60, 27] have used to perform explicit FEM with spatially different time-steps, see Figure 4.11. Belytschko's method can only handle certain ratios of time-steps between the two regions, while AVISD can basically work with any arbitrary time-step ratio.

4. Compared to the algorithm offered by Huang et. al. [34], the AVISD algorithm is well suited to run on a GPU and also, the time-step dependence of the bins give this method a big advantage over Huang's method since it is possible to have more elements that are not spatially located nearby in the same bin and increase concurrency and also there is a potential to be more work-efficient because if the elements are grouped spatially, some large elements can be grouped with few small ones and have to be updated much more frequently.

5. Choosing as many bins as elements. This introduces the classical sequential AVI method, however, it minimizes work since no element is updated more than the necessary stability time-step requirement. So the classical sequential AVI method is also

a special case of the AVISD.

According to these advantages, the AVISD algorithm proves to be the prevailing algorithm among all, however, there is another issue here. A bin configuration that will optimize the performance must be chosen. This bin configuration depends on the problem. In the next chapter, an optimization process will be introduced.

CHAPTER V

MESH-AWARE PERFORMANCE ANALYSIS

In this section, the goal is to adapt the solution method to an arbitrary mesh to gain maximum performance for any given mesh. As seen in the previous chapters, the finite element mesh configuration has a key role in the time performance of a dynamic analysis, especially in an explicit FEM problem.

5.1 Motivation

It is important to incorporate mesh configuration information to get the best time performance. In order to do this, the following point must be considered:

1. The classical "naive" method is the first, simplest and most natural method that can be used in explicit FEM. However, it is inefficient in most cases because all elements have to be updated as frequently as the smallest element.

2. The spatial decomposition method only works for certain mesh configurations. For example if there are two patches of element with the ratio of the sizes are 1.6-1.7, then according to Halleux et. al.[29], this method is less efficient than the naive method. This is a very simple case. There could potentially be many other cases that this method is not efficient. This shows the importance of deciding the solution method based on the mesh configuration.

3. As explained in section 4.3, the AVI method on a GPU is susceptible to a very low concurrency. This also affects the performance of a regular AVI method.

To the knowledge of the author, there have been no research on automatizing the choice of algorithm based on mesh analysis, or a self-adapting algorithm. The original goal of this research was to compare three different algorithm and introduce performance charts based on mesh statistical properties.

In the process of evolving the proposed AVI method, the design of the AVISD algorithm led to a method that can imitate all the other methods by the choice of bin configuration. In order to get the best bin configuration, an optimization algorithm is needed to get the desired bin configuration to minimize the runtime.

The key to achieving this goal is to describe the runtime in a performance model. This way the time cost associated with each bin choice can be found and the time performance model function can be chosen as the objective function.

5.2 Nature of the performance model

One of the goals of this research is to explain what factors take part in the performance of the discussed explicit FEM GPU algorithms. The factors that are suspected to take part in the performance are the machine parameters, the mesh parameters, the type of finite elements used, and the structure of the code engine.

The machine parameters include the computational throughput (GFLOPS/s) and device internal memory bandwidth (GBytes/s). The mesh configuration on the other hand can affect the runtime. In addition, different finite elements have different computational price.

A good performance model is a mathematical relationship that can describe and explain the role of different factors in the runtime and the more concise the relationship the more useful it will be.

From another point of view, a performance model can be described by distinguishing the factors contributing to it. Different factors contributing to the runtime can potentially be:

1. Cost of updating a single element. Since the process of updating all elements is a repetitive constant process, this cost will probably have a linear relation with the number of elements.

2. Each function that is run on the device is called a kernel. There is cost merely associated with preparations to run a kernel and so the kernel overhead cost is another factor contributing to the runtime. This cost is charged per kernel and can be related to the nature of the kernel.

3. At specific points in time, some process on the host are performed, such as input/output to the hard drive. This cost is charged per step. Note that several kernels could run per step and this cost is different than the kernel overhead. The charge associated with the Input/Output (I/O) might not be much compared to kernel runs if kernels are big enough. Also, buffering the I/O can lead to the I/O cost being overlapped by computation. So there is a chance that this cost does not have a role in the performance model.

   In addition, the I/O cost depends on the demand of the user, not the type and performance of the algorithm. The more output variables are requested and the more frequently they are requested, the higher the cost will be and this cost does not affect the choice of algorithm, or tuning the algorithm.

5.3 AVI spatial decomposition (AVISD) Algorithm Performance model

In order to track the relationship between the mentioned factors, different simulations with various bin configurations, element types and mesh configurations have been performed.

If there is a set of b number of bins as follows:

$$B = \{t_1, t_2, \cdots, t_b, t_{b+1}\} \tag{63}$$

With t_i being the bin time-step and c_i elements with inherent time-steps: $t_i \leq dt < t_{i+1}$ in bin $\#i$, element counts in bins as follows:

$$C = \{c_1, c_2, \cdots, c_b\} \tag{64}$$

And obviously:

$$\text{Total Number of elements} = \sum_{i=1}^{b} c_i \tag{65}$$

The number of updates that bin $\#i$ needs is:

$$U_i = T_{ult}/t_i \tag{66}$$

in which U_i is the number of updates done on bin $\#i$ and T_{ult} is the ultimate simulation time.

So, the total number of element updates needed will be:

$$u = \sum_{i=1}^{b} c_i U_i = \sum_{i=1}^{b} c_i (T_{ult}/t_i) = T_{ult} \sum_{i=1}^{b} \frac{c_i}{t_i} \tag{67}$$

in which u is the total element updates needed during the simulation.

If every bin update is done with a single kernel run, then the number of total kernel executions will be:

$$k = \sum_{i=1}^{b} U_i = \sum_{i=1}^{b} \frac{T_{ult}}{t_i} = T_{ult} \sum_{i=1}^{b} \frac{1}{t_i} \tag{68}$$

in which k is the number of kernel executions.

Assuming that the runtime R is linearly related to u and k :

$$R = \alpha_1 k + \alpha_2 u \tag{69}$$

In which α_1 and α_2 are constant and can be acquired by linear regression. The regression coefficients indicate the validity of the assumption for equation 69.

5.4 Explaining the coefficients

Coefficients α_1 and α_2 are all dependent on the code, the finite element type, and the machine architecture. In this section, the nature of these coefficients is explained the parameters contributing to them are discussed.

5.4.1 Cost of a single element update

The cost of element force update at each step takes between 55% to 95% of the total work, depending on the element type and the complexity of the element internal force formulation.

The unit work in the process of updating the forces is to update the force of a single element during a single time-step. This work is handled by the smallest computational unit as a sequential process. The process consists of loading the data needed (Nodal positions, previous stress values, previous plastic strain, material properties and the integration points' values and weights) computing the internal force vector, and writing the needed values back to the host memory for output (The internal force vector, the new stresses, the new plastic strains).

Here, a few factors that contribute to the run-time:

1. **The number of available cores, Device internal clock frequency:** These factors determine the capacity of the device to perform computations when the input needed data are available.

2. **The memory bandwidth between the device memory and the device computing registers:** Determines the amount of data per unit time that can be fed into the computational units

3. **Device available cache memory:** The more cache available to the device the more data can be stored by a core without the need to get reloaded again and unnecessarily increase bandwidth demand.

4. **Device available number of registers:** The number of used registers per computing unit determined the number of cores that can be utilized simultaneously. If a lot of registers are used by a single core, not all cores can be summoned.

The percentage of available computational units can be summoned is called the *Kernel Occupancy*. This factor can depend on the nature of the algorithm, as well as the implementation.

A simplistic performance model of an algorithm can usually be described as:

$$T = \frac{W}{\rho} + \frac{M}{\beta} \tag{70}$$

in which W is the total work in the number of floating point operations (FLOPS), ρ is the ideal computational throughput(FLOPS/sec), M is the amount of memory input and output(I/O) (Bytes), and β is the memory bandwidth (Bytes/sec).

In the case of this research, W and M are the work load and memory load associated with updating one element, and ρ and β are parameters associated with the device computational capacity and internal memory bandwidth. These indicate that the runtime T is closely associated with parameter α_1 in equation 69.

5.4.2 Kernel overhead

Each kernel run entails loading costs that mainly include accessing the device driver software. This cost varies based on the device and also based on the operating system. If a

kernel is large enough, this cost will be amortized by the large runtime of the kernel. In equation 69, α_1 is the term representing the kernel overhead cost. This cost is charged per kernel run and is almost independent of the problem.

5.4.3 Simulator driver code costs

The main loop of the problem has several tasks:

1. Identify the updatable loops
2. Check if the final simulation time is reached
3. Read needed output from GPU memory and store them

The cost associated with these stages are constant. In the AVISD algorithm, these costs are independent of the choice of bins. The cost associated with these steps are charged per simulator driver loop. These costs can overlap with GPU kernel runs and do not add to the total cost of the problem.

5.5 Test cases and results

For each test case, the runtime has been measured for numerous simulations with different bin sizes and configurations and different durations of simulation time. Various needed parameters such as the runtime, number of kernels, number of steps, number of element updates and bin configurations are recorded for each simulation in a configuration file.

After that, using the these information, the relationship between these parameters were evaluated using regression analysis and the regression parameters, the relevant graphs and the correlation coefficient are reported. For each case, the regression R^2 factor was higher than 98%.

5.5.1 Case 1

This test case is composed of 160,000 8-node Mindlin plate elements. Figure 5.1 demonstrates the mesh configuration for case 1.

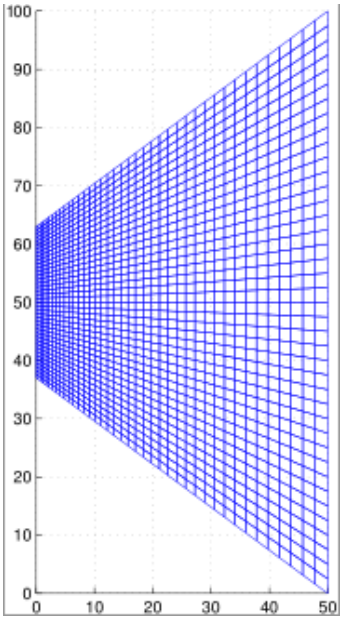

Figure 5.1: Case 1 Mesh configuration

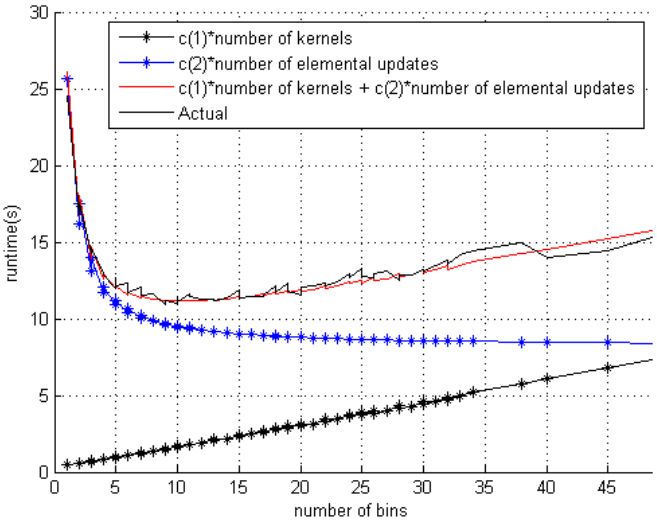

Figure 5.2: Case 1 Performance Chart

Figure 5.2 demonstrates different factors contributing to the runtime. The sizes of all the bins are equal. $c(1)$ in the figure is equivalent to α_1. The first term, ("c(1)*" number of kernels), demonstrate the cost of executing kernels. This cost increases as the number of

kernels increase. $c(2)$ in the figure is equivalent to α_2. The second term, ("c(2)*" the number of elemental updates), shows the cost of updating elements. This cost term decreases as the number of bins increase, because the time-step of the chosen bin for each element is close to the required stability time-step of the element and the element is updated at a larger time-step, so, less work needs to be done. The next chart chart in Figure 5.2 is the sum of the two, which is the cost predicted by the performance model. The last term, called "Actual" in the legend, is the actual run-time measured. As see in the Figure, the run-time predicted by the performance model is very close to the actual measured run-time of the implemented algorithm.

The regression results are listed in Table 5.1

Table 5.1: Case 1 regression results

α_1	α_2
5×10^{-4}	1.5×10^{-7}

5.5.2 Case 2

This test case is composed of 160,000 8-node Mindlin plate elements. Figure 5.3 demonstrates the mesh configuration for case 2.

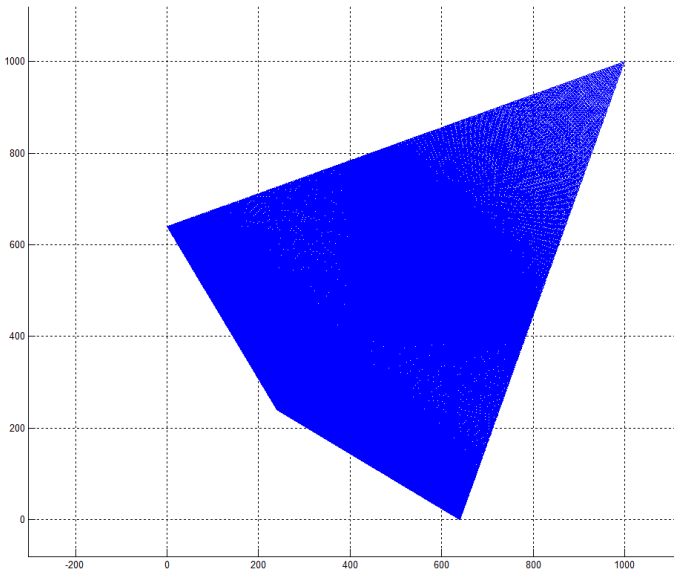

Figure 5.3: Case 2 Mesh configuration

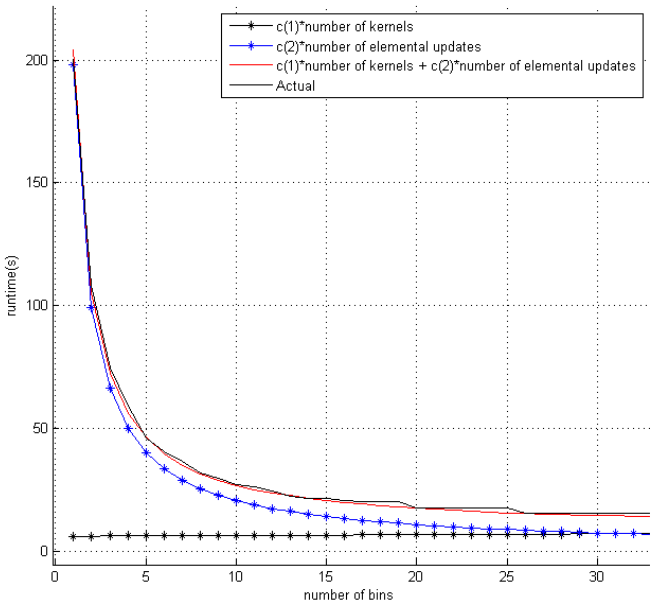

Figure 5.4: Case 2 Performance Chart

Figure 5.4 demonstrates the different factors contributing to the runtime. The parameters are the same as explained for Figure 5.2.

The regression results are listed in Table 5.2

Table 5.2: Case 2 regression results

α_1	α_2
5×10^{-4}	1.5×10^{-7}

5.5.3 Case 3

This test case is composed of 129,000 3-node CST 2D elements. Figure 5.5 demonstrates the mesh configuration for case 3.

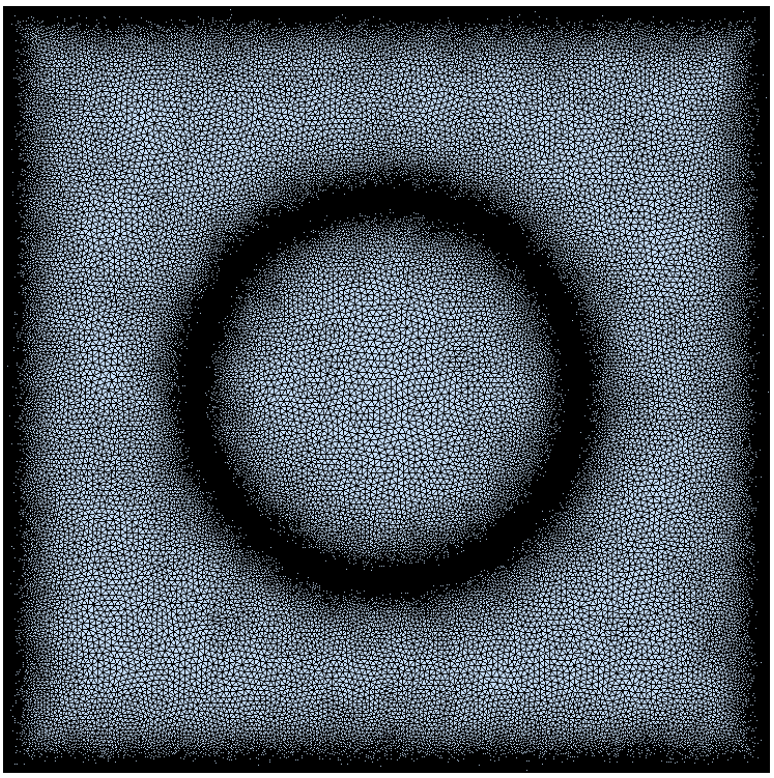

Figure 5.5: Case 3 Mesh configuration

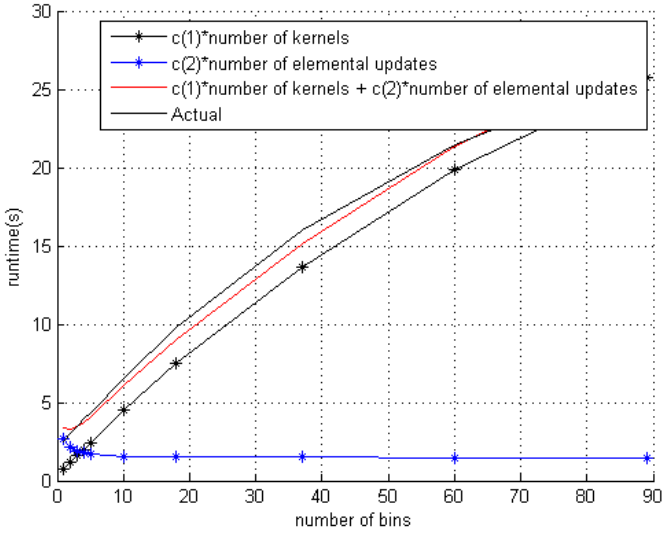

Figure 5.6: Case 3 Performance Chart

Figure 5.6 demonstrates the different factors contributing to the runtime. The parameters are the same as explained under Figure 5.2.

The regression results are listed in Table 5.3

Table 5.3: Case 3 regression results

α_1	α_2
4.48×10^{-4}	9.99×10^{-9}

5.5.4 Case 4

This test case is composed of 148,000 3-node CST 2D elements. Figure 5.7 demonstrates the mesh configuration for case 4.

Figure 5.8 demonstrates different factors contributing to the runtime. The parameters in the figures are as explained under Figure 5.2.

The regression results are listed in Table 5.4

Table 5.4: Case 4 regression results

α_1	α_2
4.48×10^{-4}	9.99×10^{-9}

5.6 Analysis and Discussion

As seen in the above 4 test cases, the kernel cost is nearly constant and the cost per element update depends on the complexity of the element. The CST element needs only 1 gauss point while the Mindlin plate element needs 9 gauss points at each layer and can have 6 to 8 layers to capture the nonlinear behavior.

The optimal number of bins is very dependent on the mesh. One cannot readily determine the optimal number of bins just based on the element type. This fact is evident by comparing case 1 and case 2.

Also, the simpler the element, the less number of bins appear to be required, since the work per element is lower and the ratio of the kernel cost to element cost is higher, resulting in relative more cost of each kernel compared to element updates. So for simpler elements,

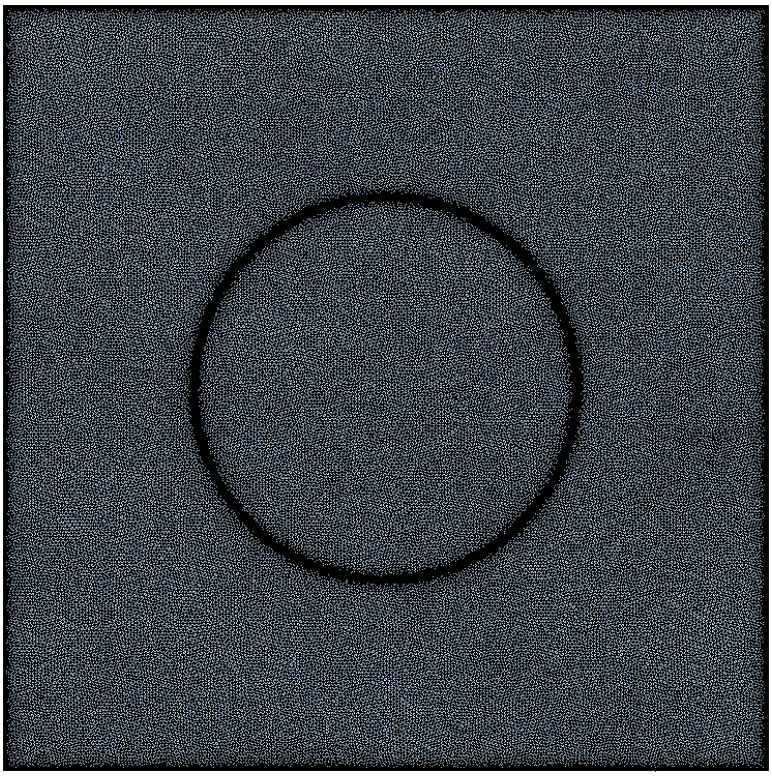

Figure 5.7: Case 4 Mesh configuration

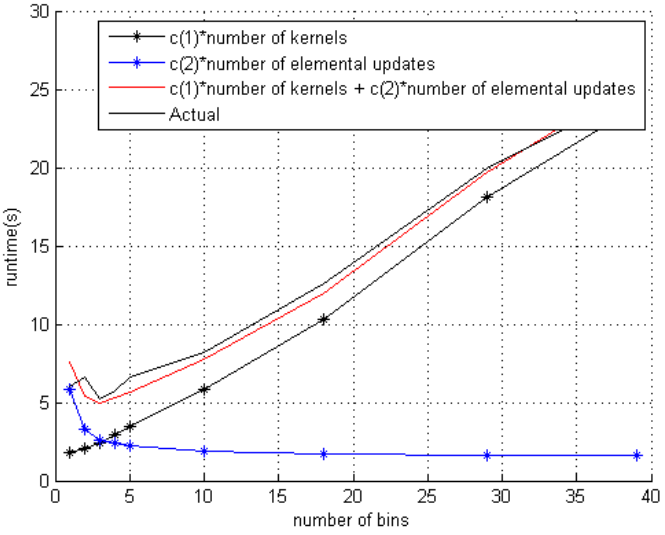

Figure 5.8: Case 4 Performance Chart

more concurrency is needed to reach the optimal behavior. A more extensive discussion on the optimization of the AVISD algorithm will be provided later in this chapter.

5.7 Comparing the naive, AVISD and Spatial Decomposition Algorithm

Here, as a case study, the following problem's cost is illustrated to show the advantage of the AVISD method to the other algorithms.

There are 600,000 Mindlin plate elements, in three groups, with the required stability time-step for each group as $1\mu s$, $1.5\mu s$ and $2.25\mu s$ respectively. The following shows the time-cost of each algorithm.

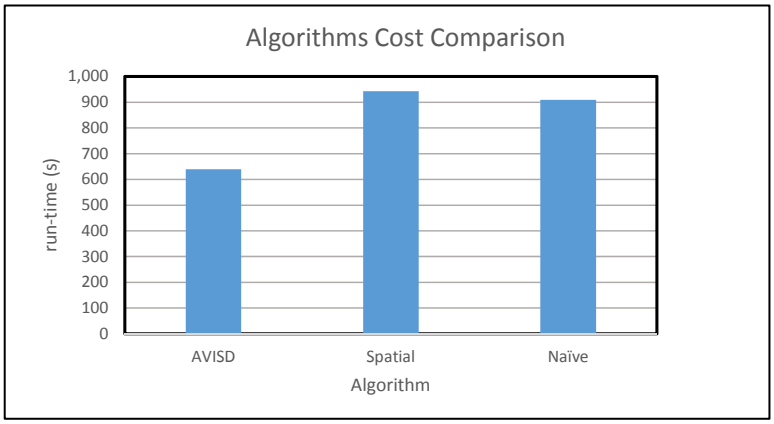

Figure 5.9: Comparing the run-time for three different algorithms.

As evident in Figure 5.9, in this case, as predicted by Casadei et. al. [10], because of the ratio of the time-steps of the groups being near 1.6, the spatial decomposition algorithm works even worse than the naive algorithm, while the AVISD algorithm is still effective and costs reasonably.

5.8 Designing a Self-tuning Algorithm

To the author's knowledge, so far there have been not been studies on:

1. The effect of mesh size distribution on the performance of a finite element algorithm
2. Designing algorithms that can handle an arbitrary mesh and perform well for each case

3. Comparing algorithms for different meshes and suggest the best algorithm based on the mesh.

One of the main objectives and contributions of this research is addressing the mentioned problems. At the first stage, there is a need to be able to incorporate the mesh parameters in the performance model.

The naive algorithm and the spatial decomposition algorithms cannot be tuned for each mesh. Their performance varies depending on different meshes, however, the performance in these cases can only be measured, not optimized.

The proposed AVISD algorithm, on the other hand, is dependent of the choice of bins and there is actually a "best" performance related to the choice of the bin configuration. The runtime cost model of equation 69 can be used to find the bin combination that minimizes the runtime.

5.9 Choice of an optimization method

At this point, an optimization method must be chosen. The chosen method must be able to accept a "Black Box" objective function that can only provide values given inputs, because the cost function as a function of the bin configuration is an extremely nonlinear and problem-dependent function.

One of the choices to solve such a problem is Particle Swarm Optimization (PSO), an iterative computational method that can be used in various optimization practices. In the PSO method, there is no need to compute the gradient of the objective function. This property is especially useful in problems that the gradient cannot easily or possibly be computed.

This advantage comes at a cost. There is a need to tune the parameters of the method or it will not be as effective or as stable as needed for practical purposes. Many have tried to improve this method over the years [78, 55]. During the past several years, PSO has been used in many applications and proved to give cheaper and faster results compared to many other methods [99, 54, 93].

A brief review of the PSO method is presented in the next section.

5.10 Particle Swarm Optimization

5.10.1 Background

In 1995, Particle Swarm optimization was invented by James Kennedy, Russell Eberhart [42] and is a stochastic population-based optimization method. Kennedy and Eberhart were originally working on simulating the behavior of birds around sources of food. Later they realized that their method is well suited for optimizing nonlinear functions. This method is very similar to evolutionary techniques such as the Genetic Algorithm. It starts with a random initial population and updates through generations to get the optima, however, there are no crossovers or mutations involved.

5.10.2 Particle Swarm Optimization General Formulation

The PSO method starts with randomly generated instances of the independent variable called particles. During the iterations of the optimization process, each particle is expected to move closer and closer to the target. During the iterations, each particle keeps a history of the best instance of itself, called the personal best. Also at each step, the global personal best is recorded too.

Each particle has a velocity that changes its current state. The velocity of each particle depends on its personal best, the global personal best, its previous velocity, predefined constants and randomly generated values.

Assuming n is the number of particles, m is the number of iterations, P is the set of particles and F is the objective function we have:

$$P = \{P_1, P_2, \cdots, P_m\} \tag{71}$$

In which P_i's are individual particles.

In which *Pbest* is the vector of personal bests for each particle and *gbest* is the best answer among all particles. c_1 and c_2 are constants defined by user and $rand()$ is a random generating function that produces uniform values between 0 and 1. They can affect the convergence and stability. It is worth mentioning that the PSO method does not guarantee finding the best answer possible. In the end, *gbest* is the best answer that was achieved and the output of the process.

Algorithm 8 Particle swarm optimization to minimize objective function F

1: $P \leftarrow$ Random values ▷ Initiation
2: $v \leftarrow 0$
3: $Pbest \leftarrow P$
4: $[f_{min}, p] \leftarrow \min(F(P))$ ▷ f_{min}: minimum value of $F(P)$ and p the minimizing element
5: $gbest \leftarrow p$
6: **for** $iter = 1 : m$ **do**
7: **for** $i = 1 : n$ **do**
8: $v_i = v_i + c_1 * rand() * (gbest - P_i) + c_2 * rand() * (pbest - P_i)$
9: $P_i = P_i + v_i$
10: $[f_{min}, p] \leftarrow \min(F(P_i, Pbest_i))$
11: $Pbest_i = p$ ▷ the minimizing factor in particle i among all iterations
12: $[f_{min}, p] \leftarrow \min(F(P, gbest))$
13: $gbest \leftarrow p$ ▷ the minimizing factor among all particles and all iterations
14: Output $gbest$

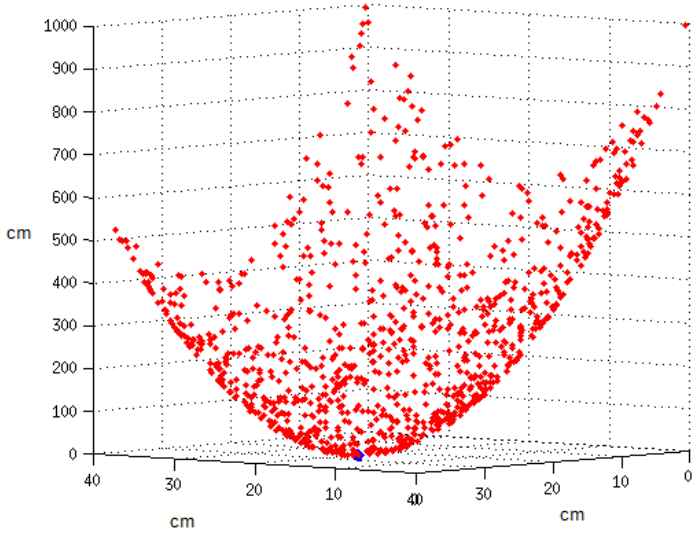

Figure 5.10: An example of finding the minimum by the PSO method [2]

Figure 5.10 shows randomly generated particles. Each particle will move toward the minimum value and the speed depends on the distance with the global minimum. All the particles together will find the minimum. Finding the minimum is not guaranteed, but with enough particles, enough iterations and proper PSO constants, one increases the chances of finding the absolute optimum point in the objective function.

5.10.3 Using PSO to find the best bin combination in the AVISD algorithm

Here the goal is to find the perfect bin configuration to minimize the runtime for the AVI binning algorithm. If as many bins as there are finite elements are chosen, each element will be updated by it's exact required time-step, not any smaller time-step. So the work required to update each element will be minimum. On the other hand, more bins will lead to more kernel runs and to pay the kernel overhead has to be paid. The number of bins and also the size of each bin are the needed parameters and the process of finding them is nonlinear.

The input here is the inherent time-step of each element. Each problem has a different bin combination. In order to represent the bins, each bin size is introduced as an independent variable. This way, the bins will always keep their order and the maximum and minimum ranges of the bins do not switch places.

The PSO method can be used to find the suitable bin combination that reduces the runtime. The objective function in the AVI binning algorithm is the runtime. Each particle in this method would be a set of bin combinations.

The problem input is:

$$E = \{\delta t_1, \delta t_2, \cdots, \delta t_n\} \tag{72}$$

In which E represents the set of time-steps of all elements, δt_i is the inherent stability time-step of bin i and n is the number of finite elements.

A bin combination can be represented as follows:

$$T = \{\tau_1, \tau_2, \cdots, \tau_b\} \tag{73}$$

In which T is the set of time-step sizes, b is the number of bins and τ_i is the size of bin i.

By definition:

$$t_{\min} = \min E \tag{74}$$

Then the ranges of bin i become:

$$t_{1i} = (\sum_{j=1}^{i-1} \tau_j) + t_{\min} \tag{75}$$

$$t_{2i} = t_{1i} + \tau_i \tag{76}$$

Here, t_{1i} and t_{2i} are the lower and upper bounds of bin i. Any element with the inherent time-step in that range belongs to this bin.

When using PSO, each particle represents one instance of E defined in equation 72. The PSO algorithm searching for optimum bin combination is represented in Algorithm 9.

Algorithm 9 Searching for the optimum bin combination using Particle Swarm Optimization

$Solution \leftarrow (0, \emptyset, \infty)$ ▷ The output contains the number of bins and the bin configuration bin pair, corresponding to the optimum solution. Here the initial values are: $Solution.b = 0, Solution.P = \emptyset, Solution.cost = \infty$

$P \leftarrow$ Random values of size $b_{\max}$ ▷ Initiation

$v \leftarrow 0$

$Pbest \leftarrow P$

$[f_{min}, p] \leftarrow \min(F(P))$ ▷ f_{min}: minimum value of $F(P)$ and p the minimizing element

$gbest \leftarrow p$

for $iter = 1 : m$ **do**

    for $i = 1 : n$ **do**

        $v_i = v_i + c_1 * rand() * (gbest - P_i) + c_2 * rand() * (pbest - P_i)$

        $P_i = P_i + v_i$

        $[f_{min}, p] \leftarrow \min(F(P_i, Pbest_i))$

        $Pbest_i = p$ ▷ the minimizing factor in particle i among all iterations

    $[f_{min}, p] \leftarrow \min(F(P, gbest))$

    $gbest \leftarrow p$ ▷ the minimizing factor among all particles and all iterations

if $F(gbest) < F(Solution.P)$ **then** ▷ replace the solution if the new found point has a lower cost

    $Solution.P = gbest$

    $Solution.b = ib$

Output $Solution$

The method demonstrated in algorithm 8 is used for maximum number of bins expected. That means given the maximum possible number of bins (can come from an educated guess, or just a large number that makes the ratio of number of elements to the number of bins greater than or equal to number of available cores), the best combination of the bins is obtained.

If for example, $b_{\max}$ is chosen to be 100 and the best bin configuration is composed of 12 bins, automatically during the iterations, the bins will go out of the $\{t_{min} \cdots t_{max}\}$ range and only 12 of the bins will contain any elements and the rest 88 will be empty. After the optimization process is finished, the bins are post-processed and the empty ones are deleted. This way, without initially knowing the best number of bins, the optimum number of bins is reached automatically.

By each bin choice, each element must be put in its relevant bin and compute the number of elements in each bin and form the vector of bin sizes in equation 64. Then by computing u from equation 67 and computing k from equation 68, the cost function of equation 69 can be computed. This process is equivalent to the function F is algorithm 9.

5.10.4 The choice of Initial Population

The common procedure in choosing the initial particles in the PSO method is to choose a completely random initial values. A series of random population will lead to a wide searching range for the optimum value. As the objective function becomes more complicated and the number of variables increase, having some reasonable guesses between the initial population becomes inevitable.

Since the objective variable is a vector of bin sizes, not a single number, having a few educated guesses beside random guesses will aid in reaching the answer more efficiently.

Choosing different bin counts with equal sizes (1 bin, 2 equal bins, 3 equal bins, up to half of the number of particles, which is 10) is the approach used here. This gives the solver the chance to find the results of equally sized bins with different number of bins and hold a balance between regular bins and random bins. Also, a gradually doubling of the bin sizes is used as another guess, since this choice of bins have proved to be very good at some

problems in the original spatial decomposition algorithm[29]. The rest of the bin sizes are chosen at random.

In the current work, half of the initial guesses are chosen regularly and half randomly, along with one matching the spatial decomposition algorithm.

5.10.5 Tuning of the PSO method

The parameters C_1 and C_2, the size of initial population and the number of PSO iterations are the 4 parameters that can be changed and the PSO method here can be tuned based on them.

By using simulated annealing, these parameters were defined to minimize the efforts to reach the optimized value faster and in a more stable fashion. The simulated annealing used here is changing variables one by one slightly seeking for the best combination. The chosen parameters are:

- $C_1 = 10^{-6}$
- $C_2 = 2 * 10^{-6}$
- Number of particles (Size of population): 20
- Number Iterations: 100

5.10.6 Statistical Analysis of the PSO method in AVISD

Since the PSO method does not guarantee finding the "best" bin configuration, there is a need to analyze the accuracy, sensitivity and stability of this method. In order to verify these factors, several test cases have run, each with a population of 50. That is, the PSO method is run 50 times for each case and the average and standard deviation of the predicted cost will be declared, along with the cost of the naive method.

Here, 4 cases are studied and the results are illustrated. Throughout this dissertation, all 2D 3-node elements and 3D 4-node elements are generated using the open-source "distmesh" software[71].

5.10.6.1 Case 1

- 265626 linear tetrahedral (LTH) Elements.
- Fully supported on all faces.
- Constant uniform step load in the beginning of simulation, as a body force.

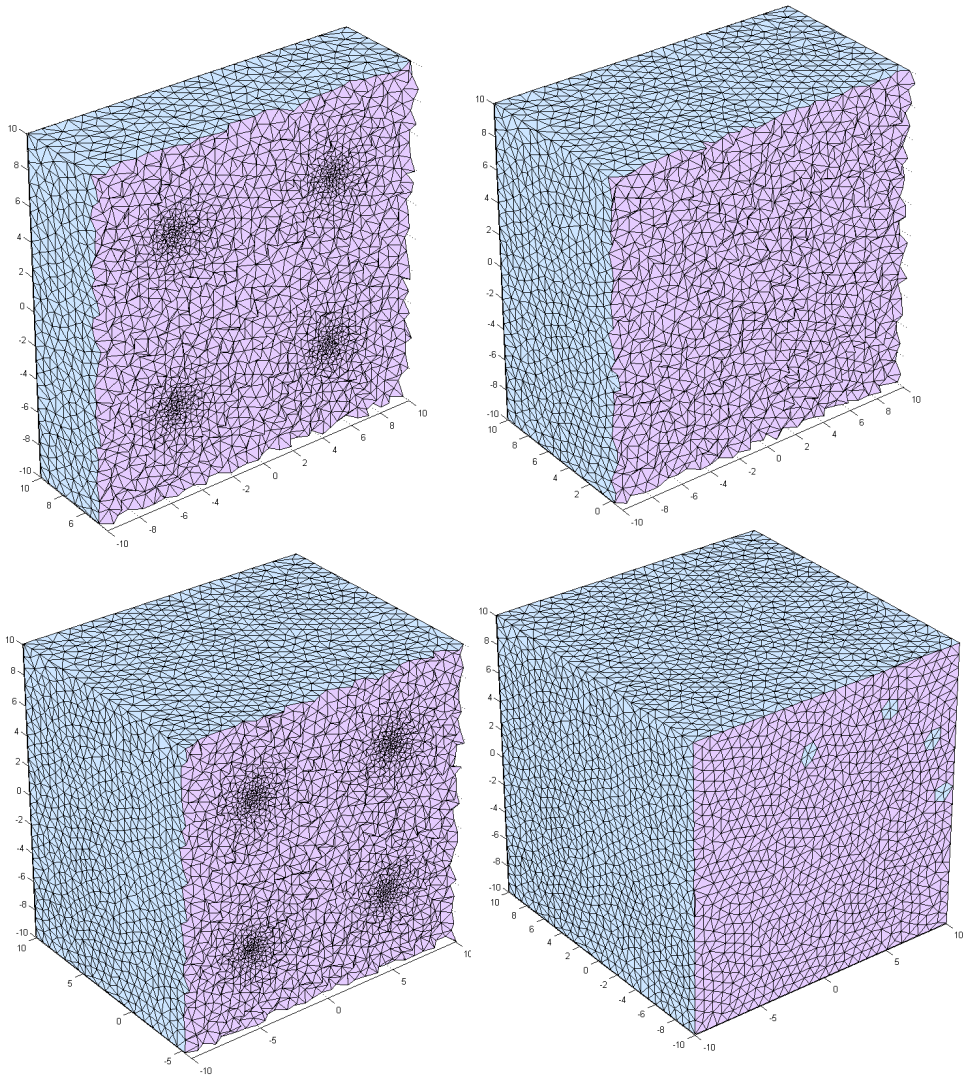

Figure 5.11: Case 1 LTH mesh.

In Figure 5.11, the mesh is depicted for case 1. 8 points are chosen and the mesh has been made finer across these 8 nodes.

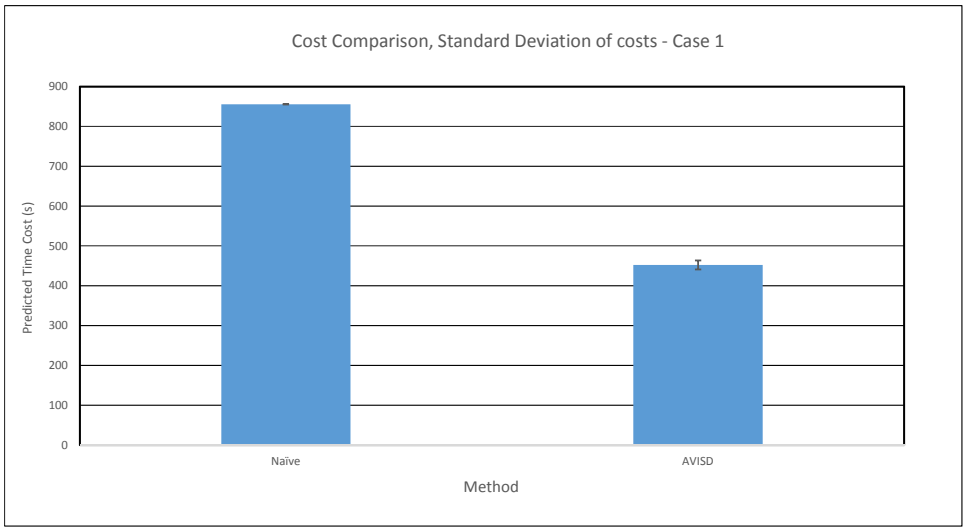

Figure 5.12: Case 1 results.

In Figure 5.12, as is the case for Figures 5.14, 5.16 and 5.18, the standard deviation is marked on the chart as an I-shape on the bar. The naive method includes 0 standard deviation as there is no uncertainty in it, however, there is a nonzero standard deviation of about 15, which is 3% of the total cost 450, has been measured. These result are measured as part of actual sampling of size 50. In this figure, we see different cross-sections of the same mesh at 25%, 50%, 75% and 100% of the depth in X-direction.

5.10.6.2 Case 2

- 418780 2D constant strain triangles (CST) Elements.
- Simply supported on all edges.
- Plane stress behavior
- Constant uniform body force step load in the beginning of simulation in X direction.

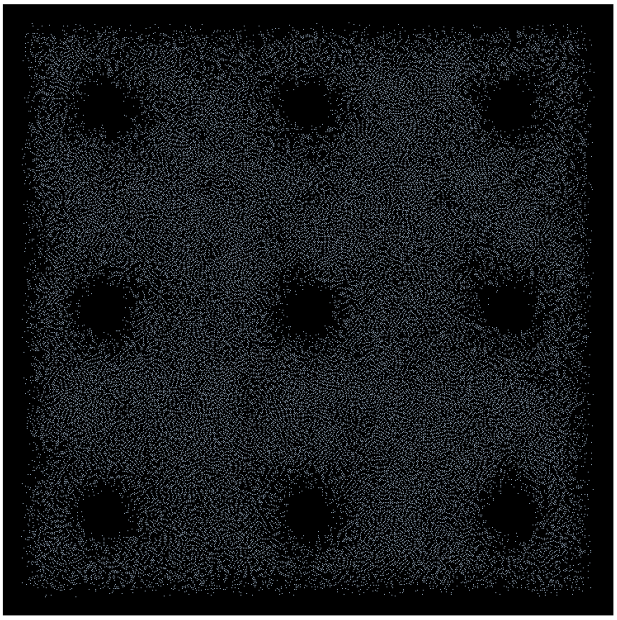

Figure 5.13: Case 2 Mesh.

In Figure 5.13, the mesh is depicted for case 2.

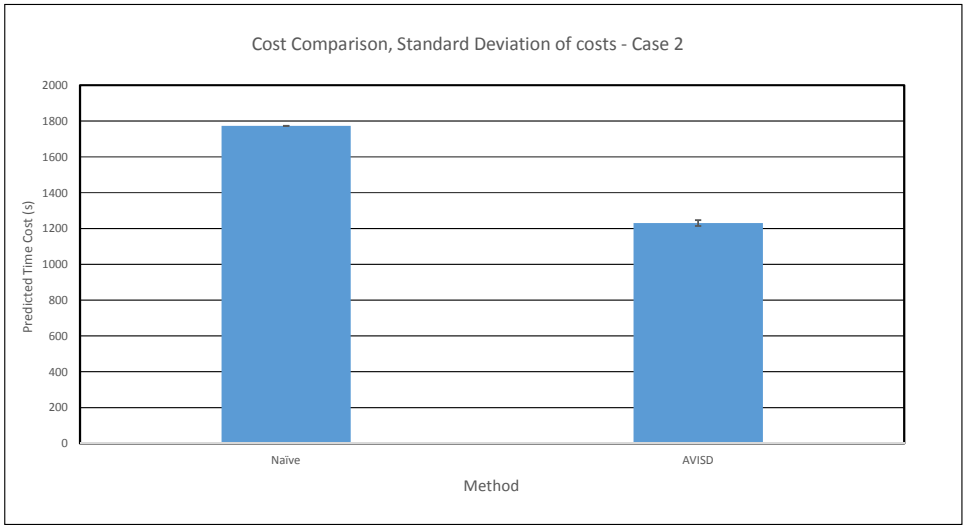

Figure 5.14: Case 2 results.

Figure 5.14 compares the naive method with the AVISD method in cost. The AVISD

method's cost is averaged among 50 samples and the standard deviation is marked over the bar with an I-shape symbol indicating the range.

5.10.6.3 Case 3

- 160,000 MINDLIN plate Elements.
- Simply supported on all edges.
- Constant uniform pressure load in the beginning of simulation applied normal to the plane.

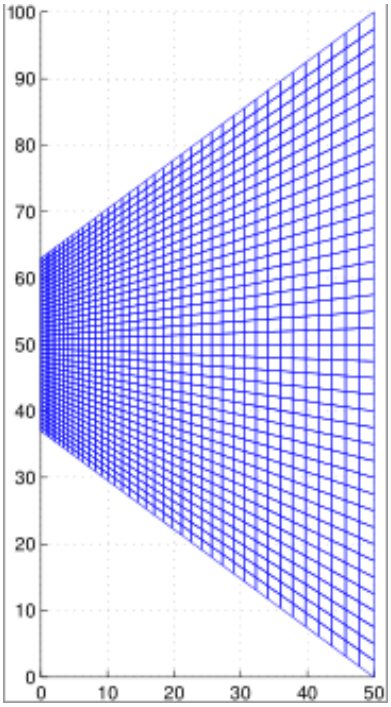

Figure 5.15: Case 3 Mesh.

In Figure 5.15, the mesh is depicted for case 3.

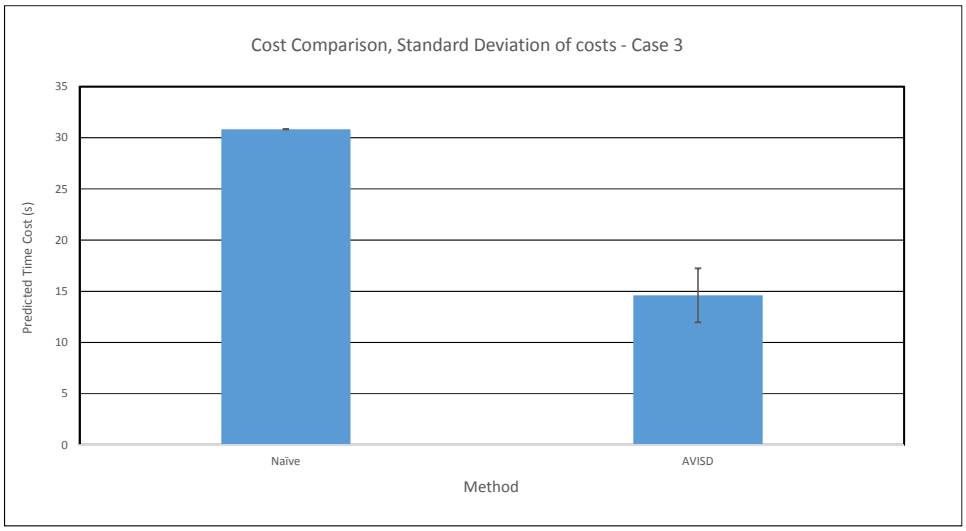

Figure 5.16: Case 3 results.

Figure 5.16 compares the naive method with the AVISD method in cost. The AVISD method's cost is averaged among 50 samples and the standard deviation is marked over the bar with an I-shape symbol indicating the range.

5.10.6.4 Case 4

- 160,000 MINDLIN plate Elements.
- Simply supported on all edges.
- Constant uniform pressure load in the beginning of simulation applied normal to the plane.

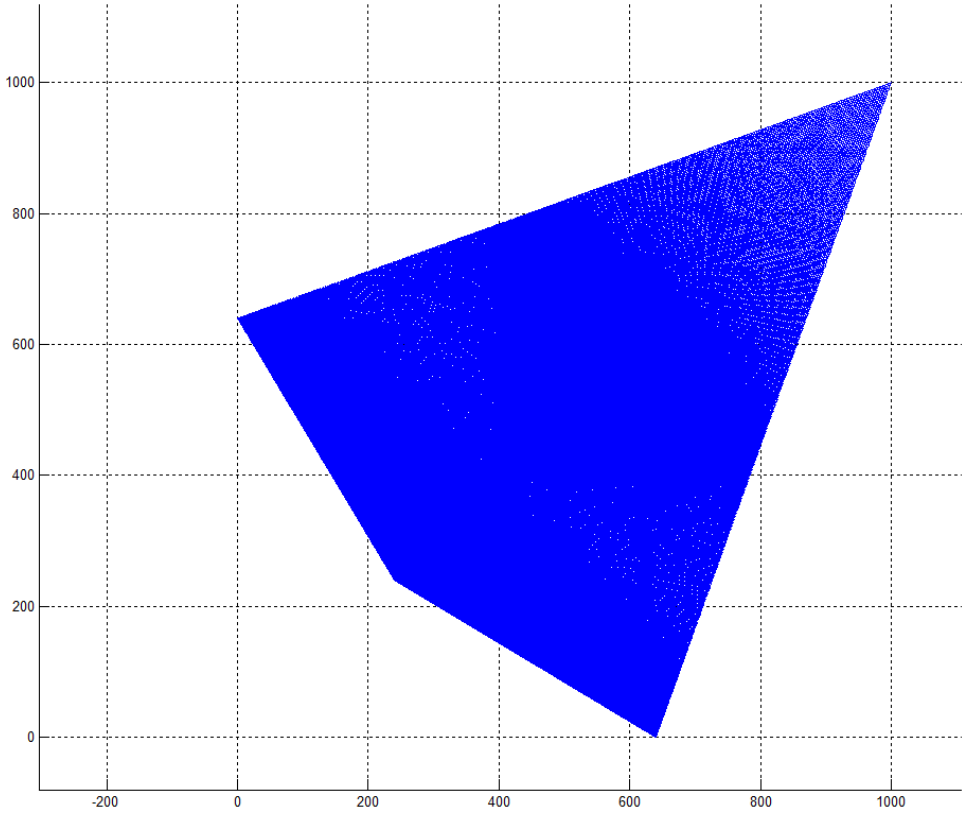

Figure 5.17: Case 4 Mesh.

In Figure 5.17, the mesh is depicted for case 4.

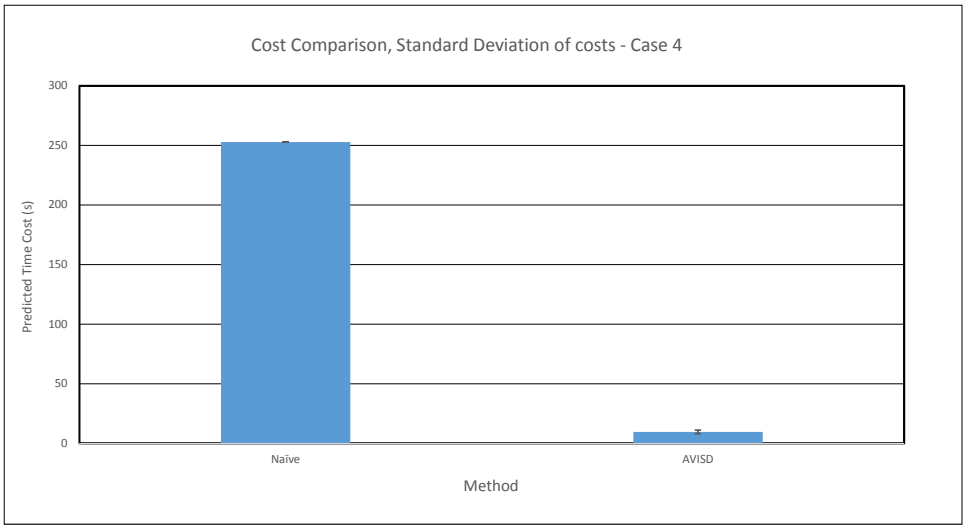

Figure 5.18: Case 4 results.

Figure 5.18 compares the naive method with the AVISD method in cost. The AVISD method's cost is averaged among 50 samples and the standard deviation is marked over the bar with an I-shape symbol indicating the range.

As can be seen in the mentioned figures, the standard deviation of the predicted cost for the AVISD method using the PSO method is negligible compared to the total cost. This shows that the use of the PSO method here is dependable and stable in this application.

5.11 Assessing the effectiveness of the PSO algorithm for the AVISD method

At this stage, the performance of the PSO method in finding an acceptable bin combination is examined. For different examples, the time cost of the PSO method is compared with the cost using uniform bin choices. Also, the bin configurations are visualized.

5.11.1 Test cases

5.11.1.1 Case 1

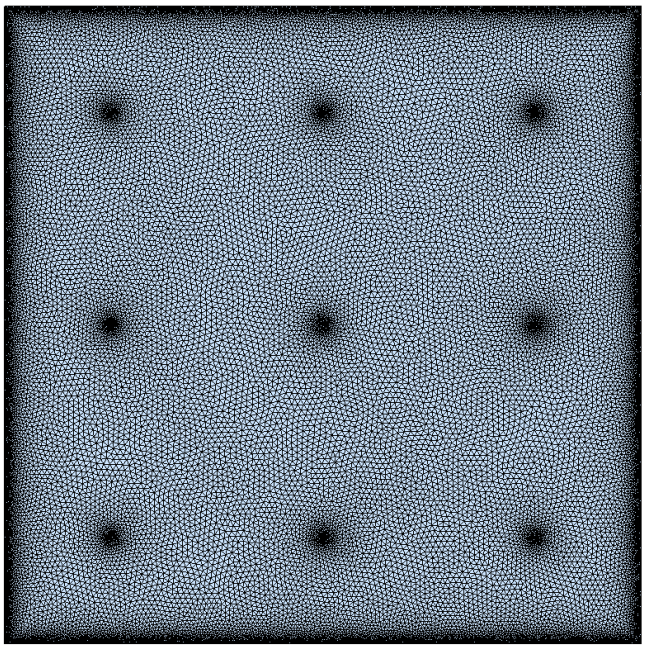

Figure 5.19: Case 1 Mesh. A coarser mesh (54000 elements) is demonstrated so the details of the mesh are more distinguishable. The mesh is uniform everywhere except for the edges and 9 dots, where the mesh gradually becomes much finer. This is a good example to test the performance of the model in a situation where there are multiple mesh concentration and in different forms (Local in the dots, distributed along the edges).

Case 1 is consisted of 418,780 2D CST elements, as shown in Figure 5.19. 9 points are chosen and the mesh made finer around those nodes.

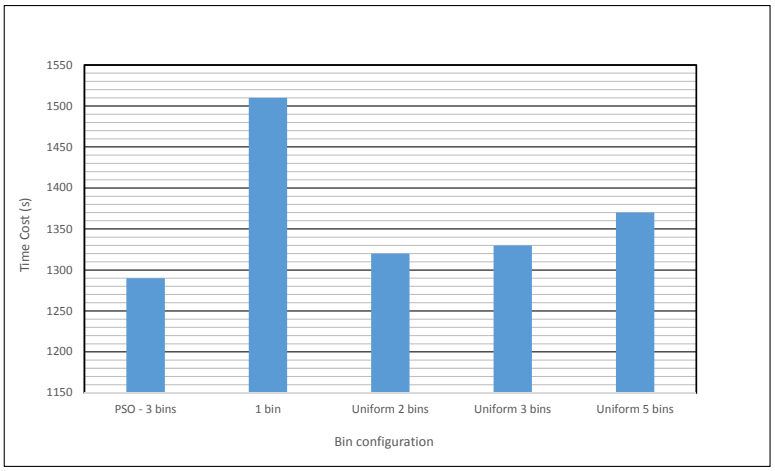

Figure 5.20: Case 1 time cost comparison

The time costs are compared in Figure 5.20. The costs are measured from actual simulations. The choice of 3 bins with the naive method is taking less run-time compared to all other cases.

Figures 5.21, 5.22 and 5.23 demonstrate the choice of bins by different methods. The PSO method automatically has chosen the number of bins and the size of each bin.

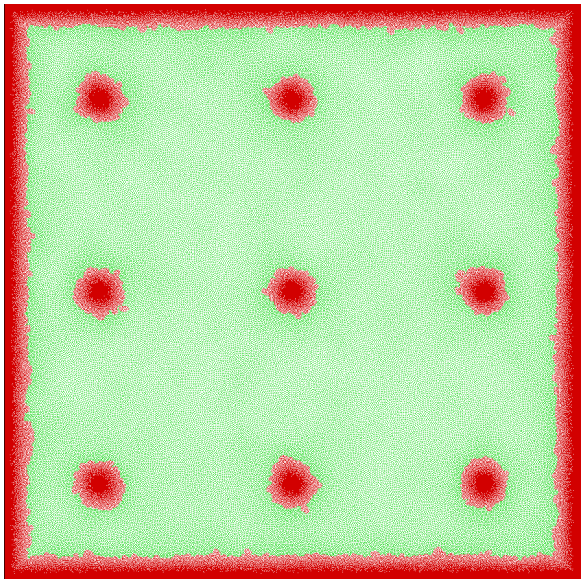

Figure 5.21: Case 1 - Two uniform bins. Each color represents a specific bin, where elements in that neighborhood are members of it.

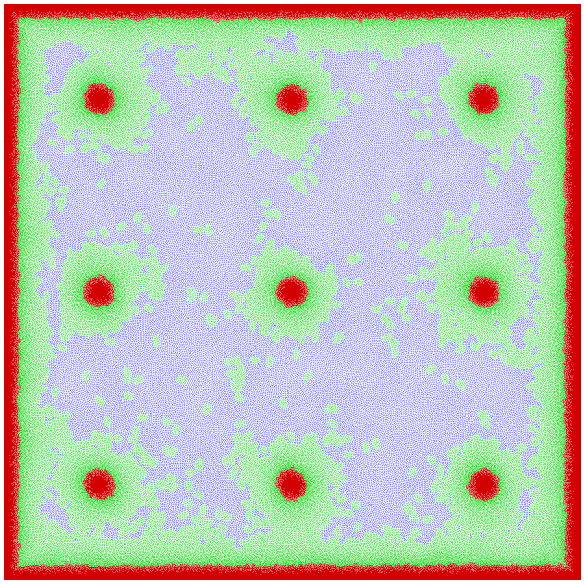

Figure 5.22: Case 1 - Three uniform bins

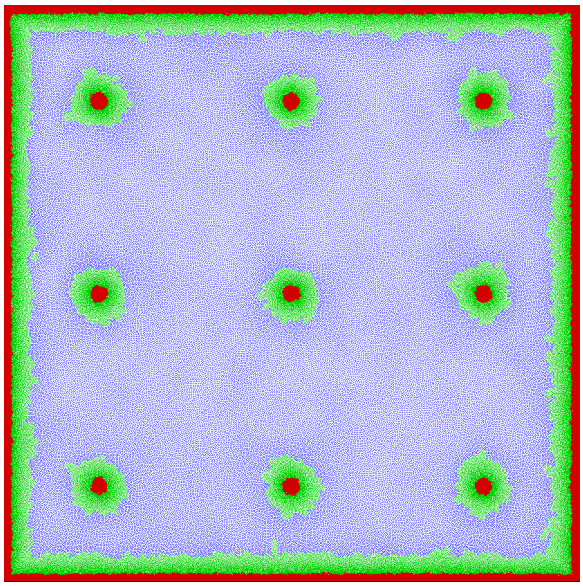

Figure 5.23: Case 1 - Three PSO chosen bins

5.11.1.2 Case 2

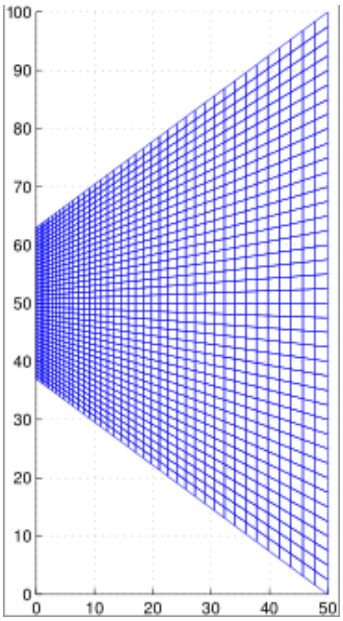

Figure 5.24: Case 2 Mesh

Case 2 is consisted of 160,000 Mindlin plate elements, as shown in Figure 5.24. The time costs are compared in Figure 5.25. Again, the time-costs are measured from actual simulations. The PSO method is taking less time compared to uniformly chosen bins.

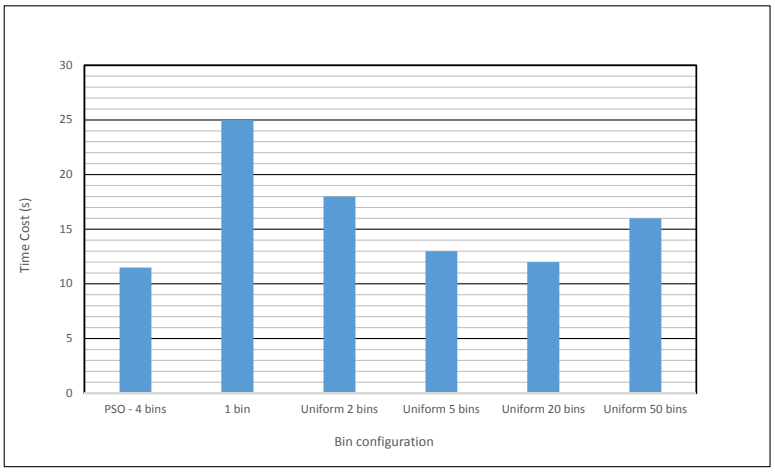

Figure 5.25: Case 2 time cost comparison

Figures 5.26, 5.27 and 5.28 demonstrate the choice of bins by different methods. The

PSO method automatically has chosen the number of bins and the size of each bin.

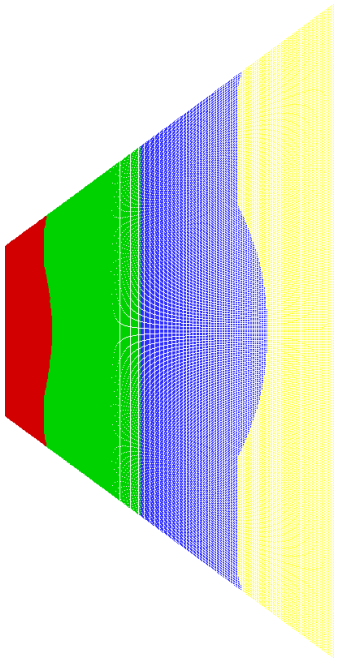

Figure 5.26: Case 2 - Four uniform bins

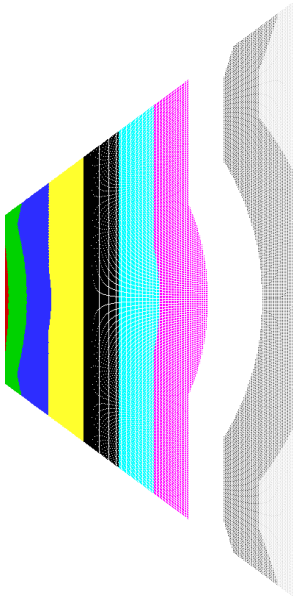

Figure 5.27: Case 2 - Ten uniform bins

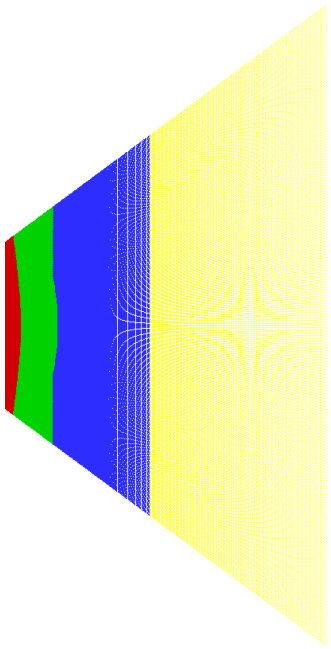

Figure 5.28: Case 2 - Four PSO chosen bins

5.11.1.3 Case 3

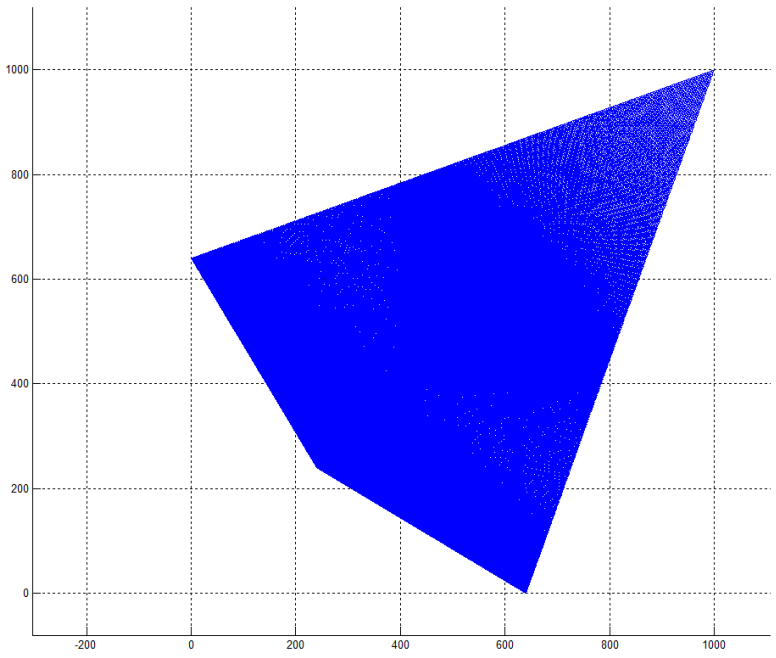

Figure 5.29: Case 3 Mesh

Case 3 is consisted of 160,000 Mindlin plate elements, as shown in Figure 5.29.

The time costs are compared in Figure 5.30. The PSO method has chosen 7 bins and is doing better than other heuristically chosen cases. The measured time is from actual simulations.

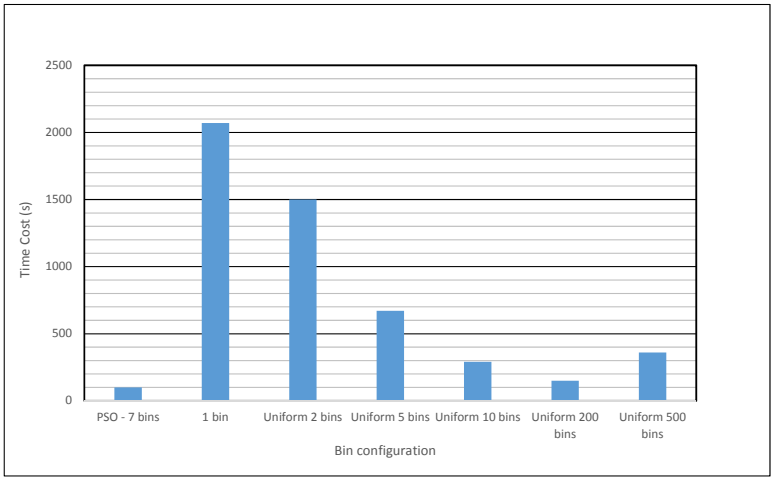

Figure 5.30: Case 3 time cost comparison

5.11.2 Analysis of the Results

In all cases, the PSO chosen bin configuration works better and faster than any other heuristically chosen bin configuration. The number of bins are automatically determined and there is no need for a trial and error. That is to say, in heuristic method of choosing the bins, one might choose different number of bins with uniform sizes and measure the run-time, while with the PSO method, the bin-sizes do not have to be uniform and the choice of bins are chosen before the simulation starts, just once.

5.12 Assessing the accuracy of the performance model.

At this point, there is a need to compare the predicted time cost with the actual implementation costs for different mesh composed of various elements and determine the effectiveness of the performance model. For this purpose, tens of simulations with various mesh configurations, element counts, element types and simulation durations have been performed. Then, the percentage of error in the prediction of the run-time is reported for each case.

5.12.1 MINDLIN elements

The test cases are described in a Table 5.5 and Figures 5.31, 5.32 and 5.33 represent the mesh types as described in Table 5.5.

Table 5.5: MINDLIN element test cases estimated time cost versus measured run-time

Test Case	Mesh Type	No.of.Elems	Duration (s)	Predicted Cost (s)	Actual (measured) Cost (s)	Error %
1	Type 2	160000	0.0001	13	13	0.67
2	Type 3	160000	0.0001	97	100	-2.88
3	Type 1	40000	0.0005	5	5	-0.76
4	Type 1	90000	0.0005	11	10	5.87
5	Type 1	160000	0.0005	19	16	13.74
6	Type 2	160000	0.0005	65	62	4.77
7	Type 3	160000	0.0005	481	489	-1.73
8	Type 1	40000	0.001	10	9	9.26
9	Type 1	90000	0.001	21	19	10.52
10	Type 1	160000	0.001	37	33	10.99
11	Type 2	160000	0.001	132	133	-0.57
12	Type 3	160000	0.001	1005	1027	-2.16
13	Type 1	40000	0.002	20	18	9.26
14	Type 1	90000	0.002	42	38	10.52
15	Type 1	160000	0.002	74	65	12.34
16	Type 2	160000	0.002	261	252	3.41
17	Type 3	160000	0.002	1924	1934	-0.54
18	Type 1	40000	0.01	99	94	5.22
19	Type 1	90000	0.01	212	228.609	-7.67
20	Type 1	160000	0.01	371	321	13.41
21	Type 2	160000	0.01	1298	1316	-1.39
22	Type 3	160000	0.01	10260	10255	0.04

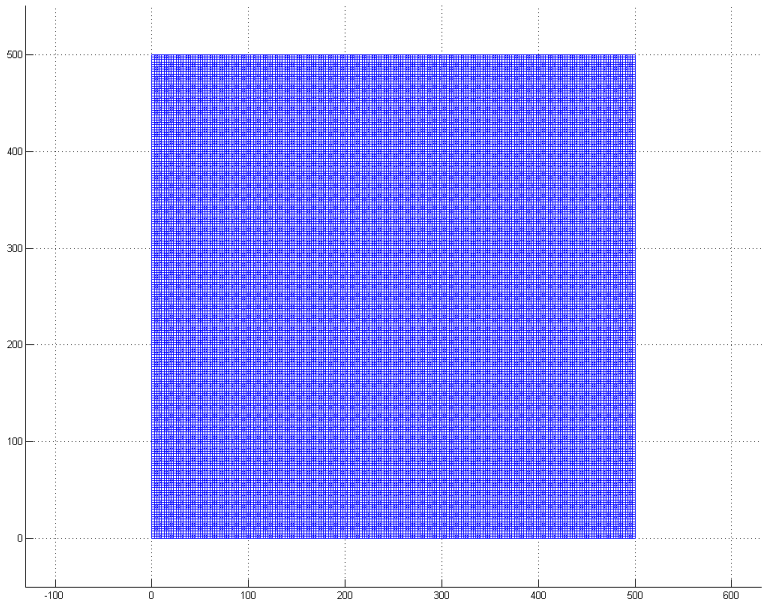

Figure 5.31: Type 1 8-node MINDLIN plate element Mesh

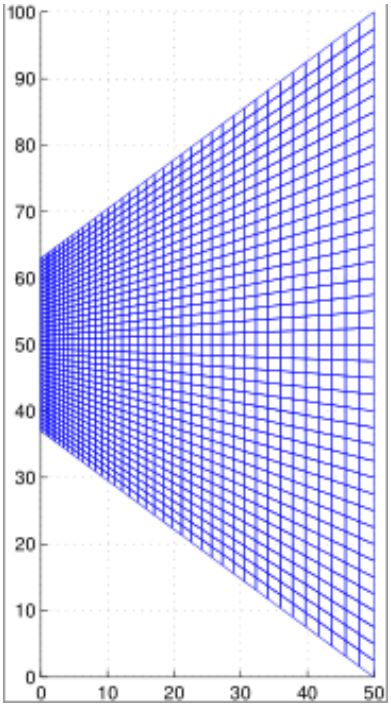

Figure 5.32: Type 2 8-node MINDLIN plate element Mesh

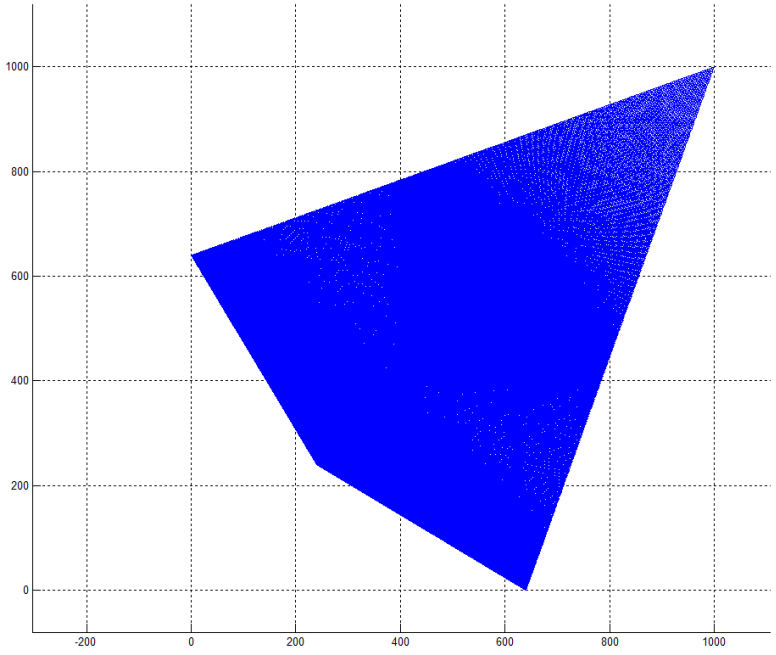

Figure 5.33: Type 3 8-node MINDLIN plate element Mesh

The diagram of the errors for each of the above cases is illustrated in Figure 5.34.

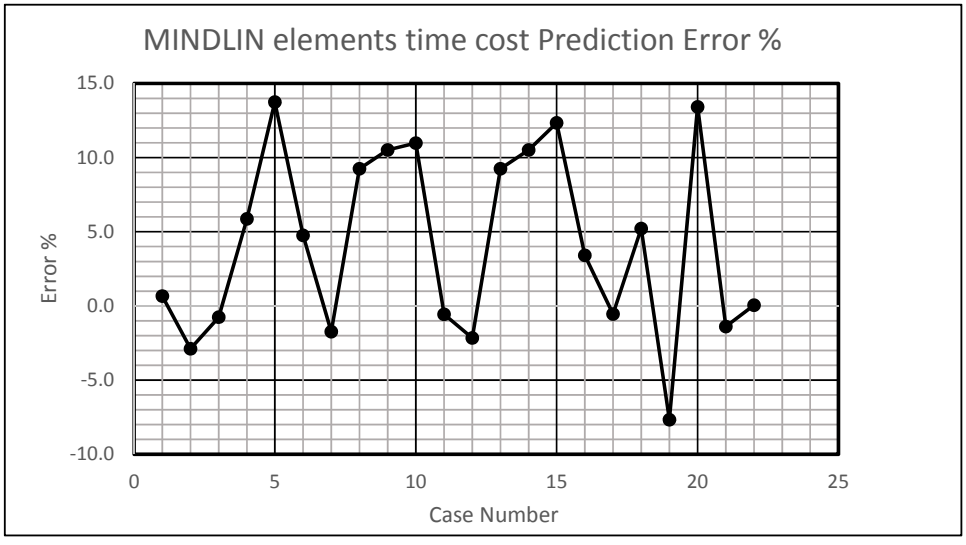

Figure 5.34: Error in time cost estimation for each Mindlin element test case.

Figure 5.34 shows the error percentage between the performance model prediction and the actual measured run-time from simulation.

5.12.2 CST elements

The test cases are described in a Table 5.6 and Figures 5.35, 5.36, 5.37, 5.38, 5.39, 5.40, 5.41, 5.42, 5.43, 5.44 and 5.45 represent the mesh types in Table 5.6.

Table 5.6: CST element test cases estimated time cost versus measured run-time

Case No.	Mesh Type	No.of.Elems	Duration(s)	Predicted(s)	Measured(s)	Error %
1	Type 1	110909	0.001	46	47	-2.80
2	Type 2	101465	0.001	122	116	4.71
3	Type 3	35136	0.001	24	26	-6.15
4	Type 4	132111	0.001	52	53	-2.53
5	Type 4	137960	0.001	27	26	4.65
6	Type 4	123332	0.001	25	23	6.70
7	Type 5	134514	0.001	66	68	-3.59
8	Type 8	137199	0.001	109	100	7.96
9	Type 6	132716	0.001	212	205	3.16
10	Type 9	119311	0.001	83	76	8.89
11	Type 10	169904	0.001	47	48	-2.89
12	Type 10	129563	0.001	33	31	5.48
13	Type 10	102280	0.001	24	24	1.94
14	Type 10	74859	0.001	17	18	-4.54
15	Type 10	57692	0.001	13	13	-1.71
16	Type 10	36941	0.001	9	9	-3.39
17	Type 10	20604	0.001	5	5	6.00
18	Type 5	28890	0.001	17	18	-6.35
19	Type 5	54537	0.001	27	30	-9.34
20	Type 5	81661	0.001	40	43	-6.17
21	Type 5	103946	0.001	48	51	-5.45
22	Type 5	107898	0.001	44	47	-6.15

Table 5.6 (continued).

23	Type 5	134989	0.001	66	69	-4.72
24	Type 5	158164	0.001	78	82	-5.64
25	Type 5	185306	0.001	92	97	-5.18
26	Type 5	219037	0.001	112	113	-1.30
27	Type 5	267193	0.001	140	143	-2.44
28	Type 5	332211	0.001	179	189	-5.30
29	Type 5	418780	0.001	252	276	-9.40
30	Type 5	28890	0.005	85	87	-2.81
31	Type 5	54537	0.005	137	147	-7.14
32	Type 5	81661	0.005	202	216	-6.71
33	Type 5	103946	0.005	242	256	-5.89
34	Type 5	107898	0.005	221	230	-3.92
35	Type 5	134989	0.005	330	347	-5.30
36	Type 5	158164	0.005	388	408	-5.12
37	Type 5	185306	0.005	461	488	-5.85
38	Type 5	219037	0.005	558	568	-1.86
39	Type 5	267193	0.005	698	698	-0.05
40	Type 5	332211	0.005	904	933	-3.22
41	Type 5	418780	0.005	1264	1366	-8.08
42	Type 2	19436	0.001	22	23	-3.61
43	Type 2	32818	0.001	37	38	-3.83
44	Type 2	64442	0.001	68	64	5.47
45	Type 2	101820	0.001	122	115	5.75
46	Type 2	132708	0.001	169	177	-4.44
47	Type 2	180862	0.001	234	250	-6.92
48	Type 2	261516	0.001	356	364	-2.11
49	Type 2	337831	0.001	500	522	-4.49

Table 5.6 (continued).

50	Type 2	409217	0.001	615	601	2.30
51	Type 2	19436	0.005	111	108	2.69
52	Type 2	32818	0.005	183	177	3.28
53	Type 2	64442	0.005	339	320	5.47
54	Type 2	101820	0.005	652	701	-7.46
55	Type 2	132708	0.005	848	867	-2.30
56	Type 2	180862	0.005	1169	1210	-3.55
57	Type 2	261516	0.005	1779	1855	-4.29
58	Type 2	337831	0.005	2497	2608	-4.44
59	Type 2	409217	0.005	3076	3006	2.27
60	Type 11	23868	0.001	5	6	-9.40
61	Type 11	34784	0.001	7	7	3.96
62	Type 11	56819	0.001	12	12	-1.49
63	Type 11	95531	0.001	20	20	1.84
64	Type 11	132530	0.001	31	29	5.51
65	Type 11	170182	0.001	41	40	2.81
66	Type 11	23868	0.005	27	29	-5.76
67	Type 11	34784	0.005	36	39	-7.03
68	Type 11	56819	0.005	59	61	-3.19
69	Type 11	95531	0.005	102	104	-2.09
70	Type 11	132530	0.005	153	148	3.55
71	Type 11	170182	0.005	206	205	0.38
72	Type 1	110909	0.008	366	380	-3.91
73	Type 2	101465	0.008	1040	1123	-8.01
74	Type 3	35136	0.008	196	204	-4.11
75	Type 4	132111	0.008	414	423	-2.27
76	Type 4	137960	0.008	218	211	3.28

Table 5.6 (continued).

77	Type 4	123332	0.008	197	192	2.64
78	Type 5	134514	0.008	525	551	-4.93
79	Type 8	137199	0.008	869	834	4.05
80	Type 6	132716	0.008	1694	1762	-4.03
81	Type 7	141547	0.008	1086	1035	4.72
82	Type 9	119311	0.008	667	638	4.39
83	Type 10	169904	0.008	373	380	-1.84
84	Type 10	129563	0.008	262	255	2.81
85	Type 10	102280	0.008	196	192	1.94
86	Type 10	74859	0.008	138	140	-1.64
87	Type 10	57692	0.008	102	104	-1.71
88	Type 10	36941	0.008	70	72	-3.39
89	Type 10	20604	0.008	43	45	-5.75

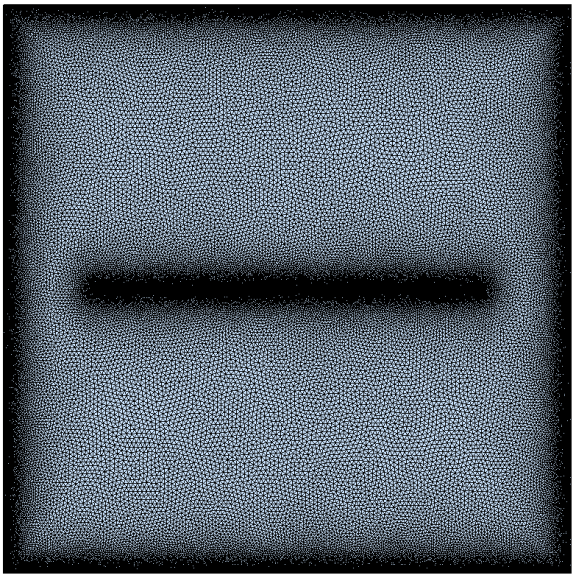

Figure 5.35: Type 1 2D CST element Mesh. Square mesh with finer mesh at a line in the middle and on the edges.

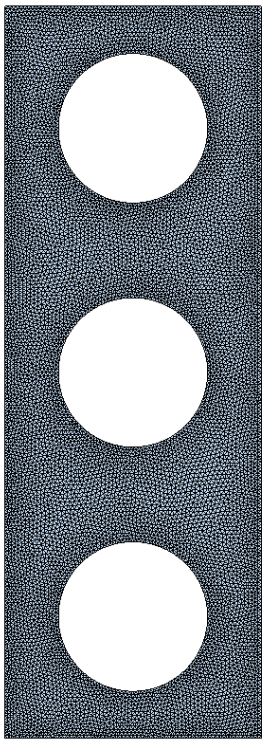

Figure 5.36: Type 2 2D CST element Mesh. Rectangular plate with three interior holes.

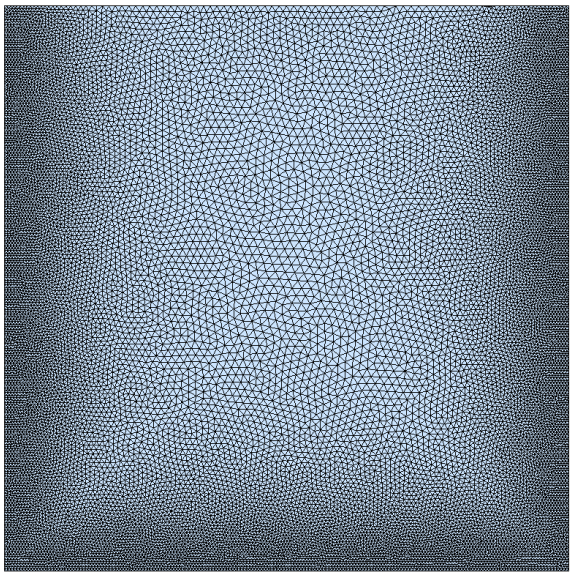

Figure 5.37: Type 3 2D CST element Mesh. Square mesh with finer mesh on three edges

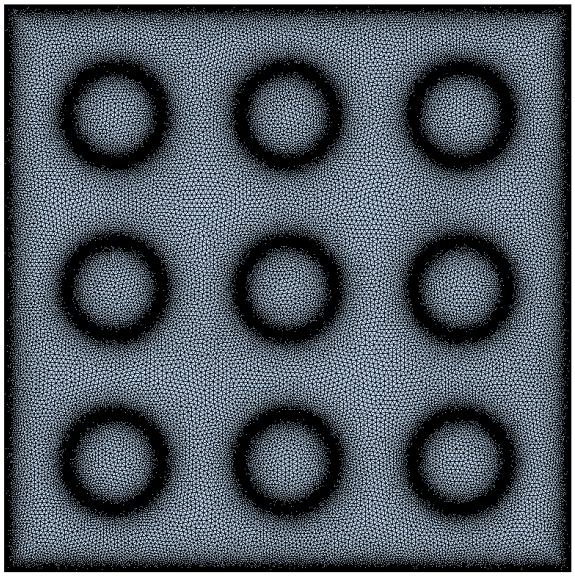

Figure 5.38: Type 4 2D CST element Mesh. Square mesh with finer mesh around 9 circles and the edges.

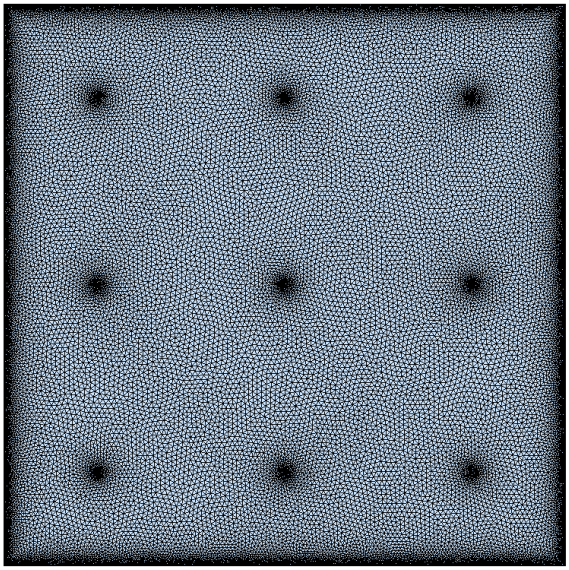

Figure 5.39: Type 5 2D CST element Mesh. Square mesh with finer mesh at 9 points.

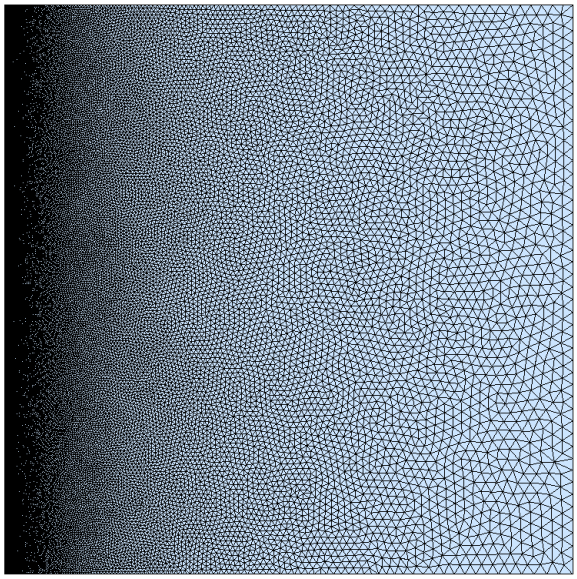

Figure 5.40: Type 6 2D CST element Mesh. Square mesh with a linear gradient of the change in element size.

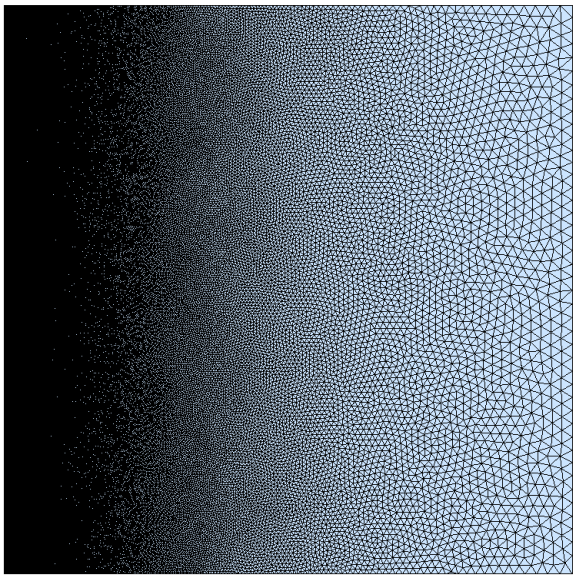

Figure 5.41: Type 7 2D CST element Mesh. Square mesh with a quadratic gradient of the change in element size.

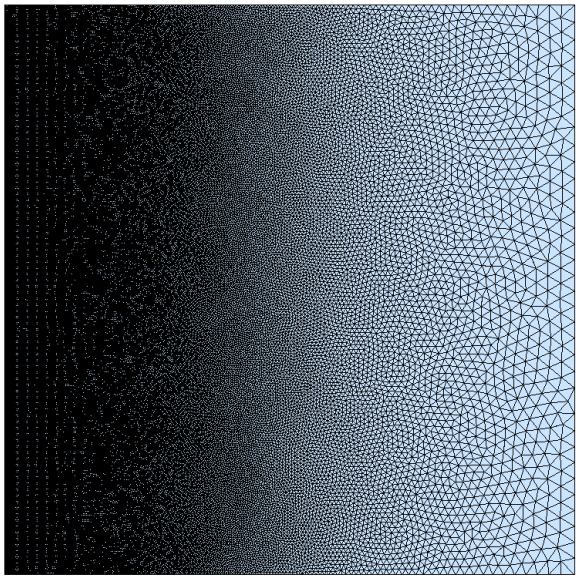

Figure 5.42: Type 8 2D CST element Mesh. Square mesh with a cubic gradient of the change in element size.

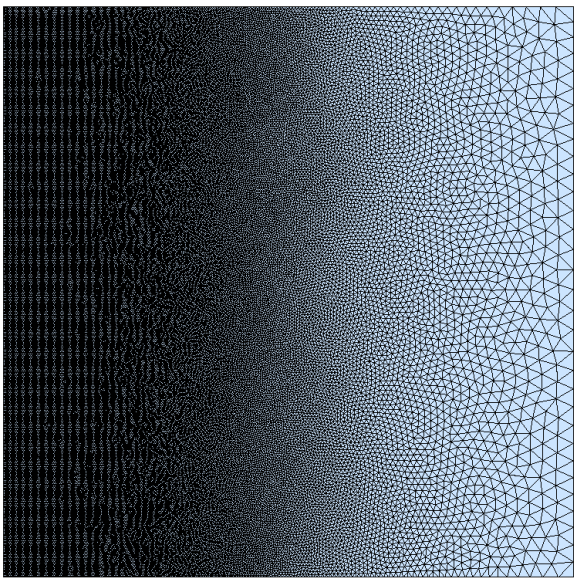

Figure 5.43: Type 9 2D CST element Mesh. Square mesh with a quartic gradient of the change in element size.

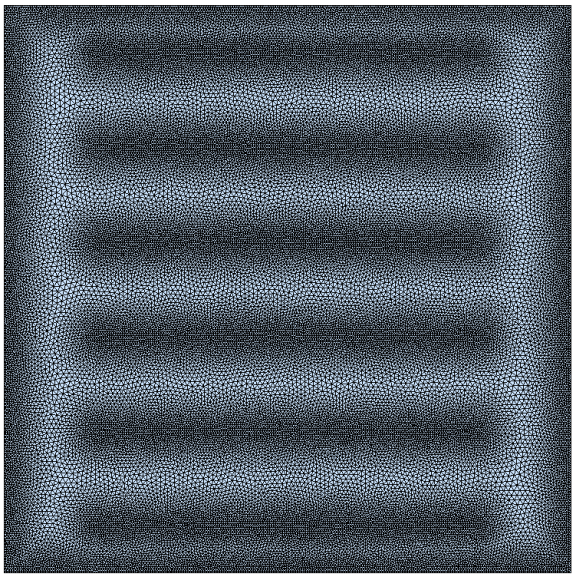

Figure 5.44: Type 10 2D CST element Mesh. Square mesh with finer mesh on 6 interior lines.

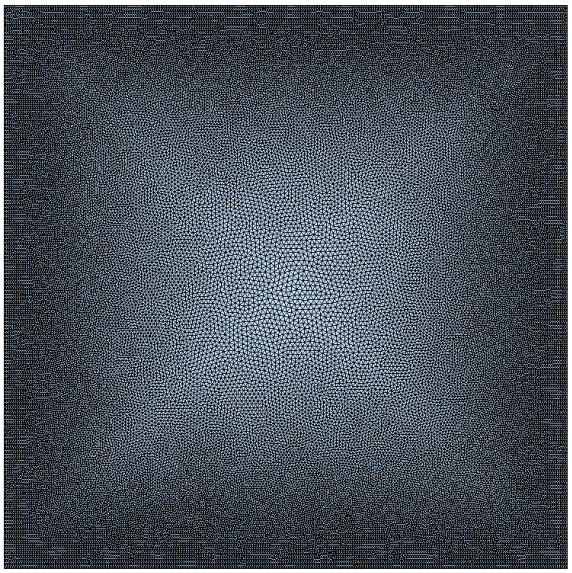

Figure 5.45: Type 11 2D CST element Mesh. Square mesh with mesh finer on all edges.

The diagram of the errors for each case are illustrated in Figure 5.46.

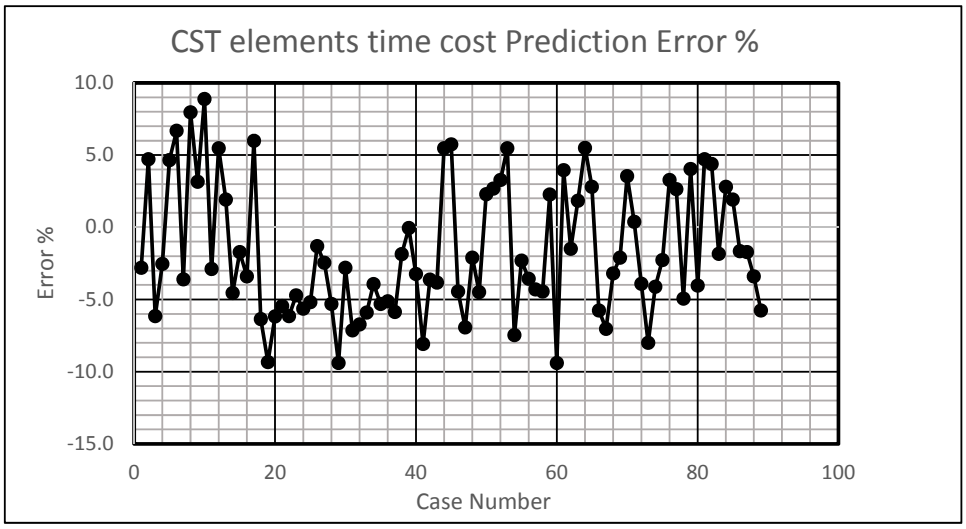

Figure 5.46: Error in time cost estimation for each CST element test case.

5.12.3 LTH elements

The test cases are described in a Table 5.7 and Figures 5.47, 5.48, 5.49 and 5.50 represent the mesh types as described in Table 5.7.

Table 5.7: LTH element test cases estimated time cost versus measured run-time

Case No.	Mesh Type	No.of.Elems	Duration(s)	Predicted(s)	Measured(s)	Error %
1	Type 1	123615	0.0005	7	7	0.65
2	Type 1	164019	0.0005	10	10	-4.68
3	Type 1	316541	0.0005	19	19	1.99
4	Type 2	66337	0.0005	5	5	6.81
5	Type 2	105118	0.0005	9	9	-4.30
6	Type 2	141207	0.0005	11	11	-2.35
7	Type 2	273742	0.0005	23	23	-2.06
8	Type 2	288718	0.0005	24	24	-2.07
9	Type 3	37495	0.0005	4	4	1.67
10	Type 3	62283	0.0005	6	6	-4.15
11	Type 3	99602	0.0005	8	9	-7.65
12	Type 3	265626	0.0005	22	23	-5.28
13	Type 3	295911	0.0005	25	25	0.57
14	Type 4	25188	0.0005	2	2	-8.05
15	Type 4	326845	0.0005	25	24	6.05
16	Type 1	123615	0.001	14	13	8.39
17	Type 1	164019	0.001	19	19	0.36
18	Type 1	316541	0.001	39	37	4.72
19	Type 1	363788	0.001	173	165	4.77
20	Type 2	25178	0.001	5	5	4.24
21	Type 2	66337	0.001	11	10	6.81
22	Type 2	105118	0.001	17	16	7.67
23	Type 2	141207	0.001	21	22	-2.32
24	Type 2	273742	0.001	45	45	0.13
25	Type 2	288718	0.001	47	46	2.14

Table 5.7 (continued).

26	Type 3	265626	0.001	45	46	-1.97
27	Type 3	295911	0.001	50	53	-5.13
28	Type 4	326845	0.001	51	50	2.56
29	Type 1	164019	0.005	95	89	7.13
30	Type 1	316541	0.005	194	190	1.95
31	Type 2	66337	0.005	53	49	9.02
32	Type 2	105118	0.005	86	82	5.01
33	Type 2	141207	0.005	107	108	-0.48
34	Type 2	273742	0.005	225	229	-1.64
35	Type 2	288718	0.005	235	233	0.85
36	Type 3	37495	0.005	41	41	-0.73
37	Type 3	62283	0.005	57	58	-0.98
38	Type 3	99602	0.005	83	89	-6.58
39	Type 3	131935	0.005	111	121	-8.19
40	Type 3	265626	0.005	225	245	-7.96
41	Type 3	295911	0.005	241	256	-6.01
42	Type 4	326845	0.005	254	236	7.74
43	Type 1	164019	0.01	191	177	7.72
44	Type 1	316541	0.01	387	368	5.30
45	Type 1	363788	0.01	1725	1603	7.59
46	Type 2	66337	0.01	107	98	9.11
47	Type 2	105118	0.01	172	168	2.55
48	Type 2	141207	0.01	215	215	0.02
49	Type 2	288718	0.01	470	463	1.52
50	Type 3	37495	0.01	81	81	0.43
51	Type 3	62283	0.01	115	121	-5.01
52	Type 3	99602	0.01	166	176	-5.54

Table 5.7 (continued).

53	Type 3	131935	0.01	222	242	-8.18
54	Type 3	265626	0.01	434	480	-9.48
55	Type 4	295911	0.01	503	529	-4.94
56	Type 4	326845	0.01	510	492	3.71

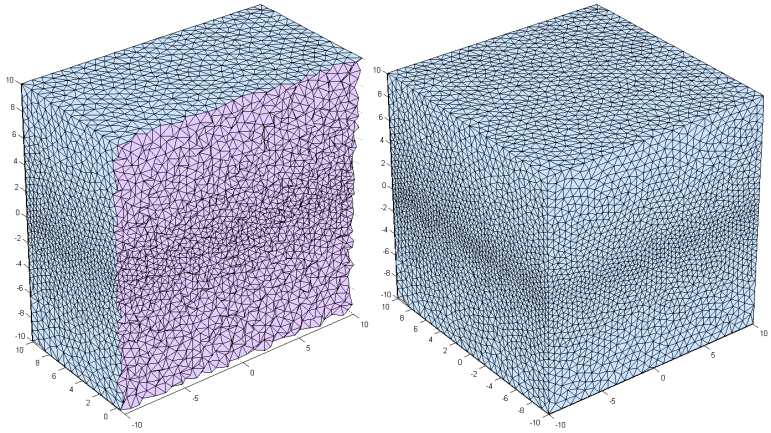

Figure 5.47: Type 1 3D LTH element Mesh. Cube with mesh finer across a plane.

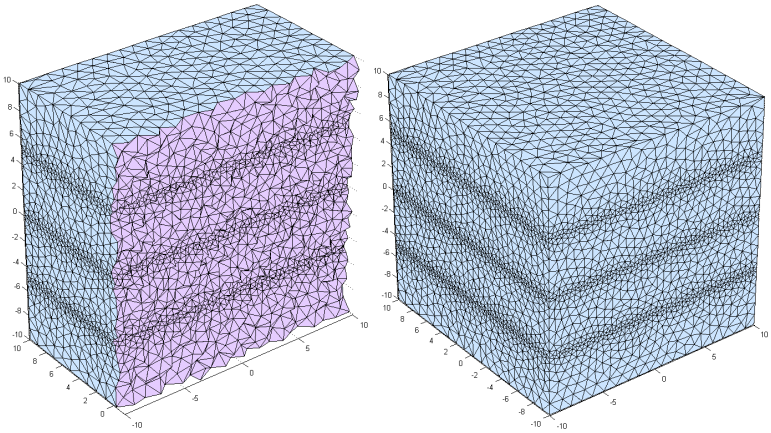

Figure 5.48: Type 2 3D LTH element Mesh. Cube with mesh finer across three parallel planes.

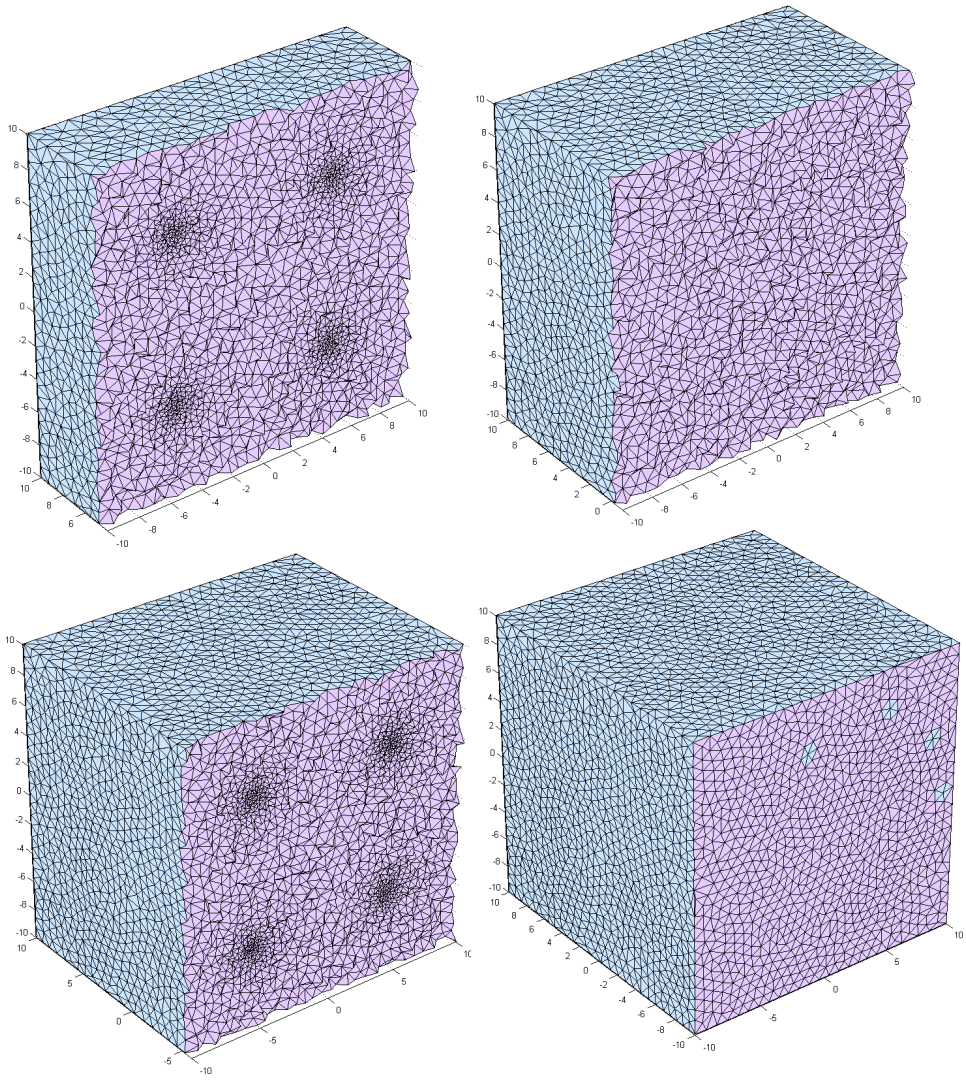

Figure 5.49: Type 3 3D LTH element Mesh. Cube with mesh finer at 8 interior nodes.

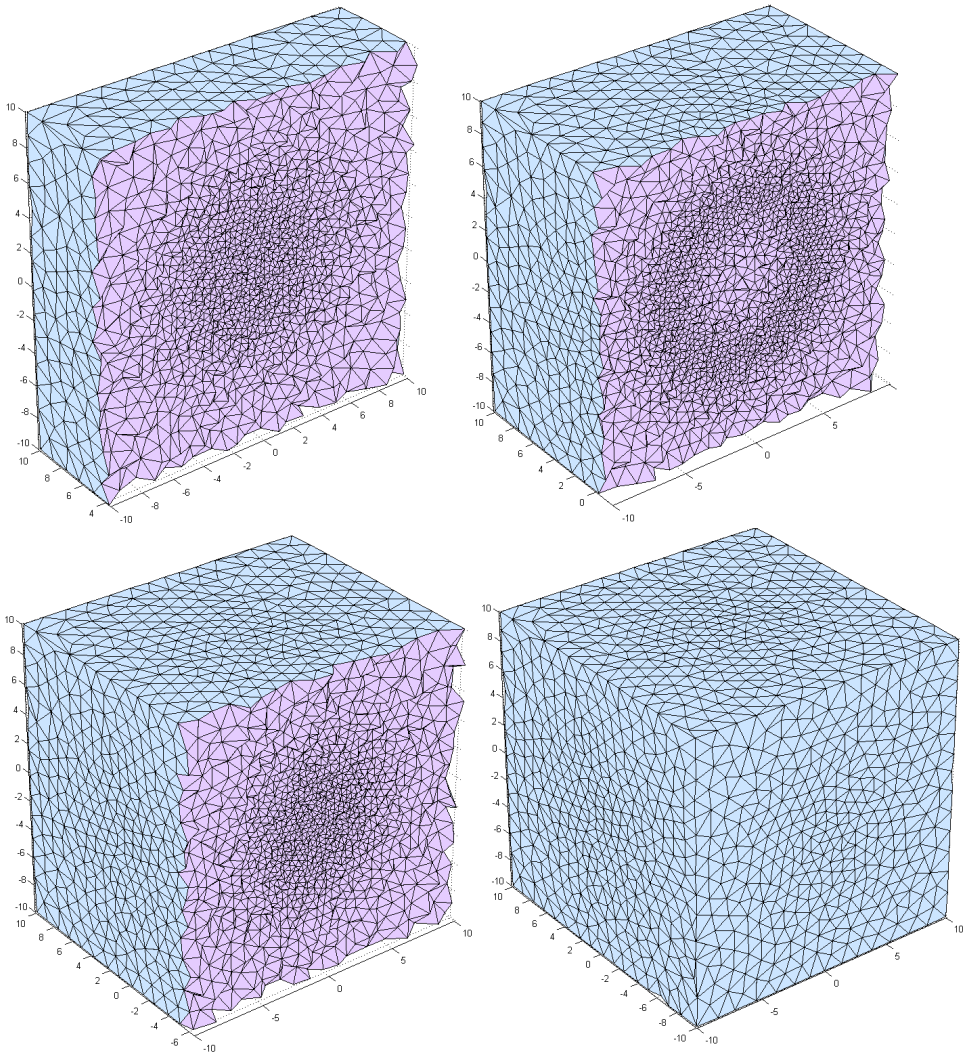

Figure 5.50: Type 4 3D LTH element Mesh. Cube with mesh finer around a spherical region inside the cube.

The diagram of the errors for each case are illustrated in Figure 5.51.

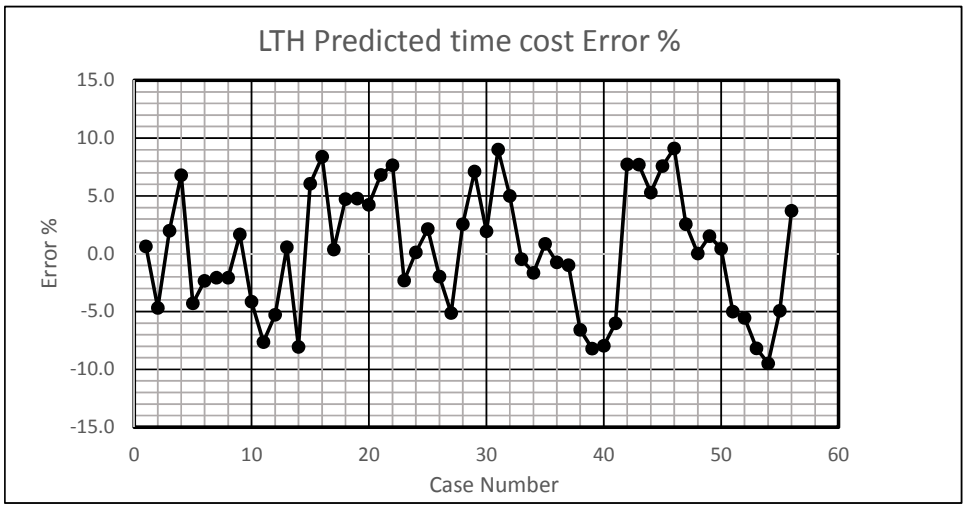

Figure 5.51: Error in time cost estimation for each LTH element test case.

5.12.4 Discussion

In all three elements, the performance model predicts the time cost within 15% of the actual measured results for MINDLIN elements and 10% for CST and LTH elements. This change is negligible since the performance improvement by the AVISD method, as demonstrated in the figures for the test cases in section 5.11, is much higher.

The difference between the predicted and actual costs is due to the several factors:

- Data coherence: It is not possible in a finite elements mesh to number all elements perfectly so all the data needed by computing cores with consecutive id's are located in the same order. This causes some incoherence in data that is aleatoric in nature which affects the run-time.

- The GPU driver does not guarantee a constant kernel cost and this cost can vary slightly, up to 20% in author's experience on a Windows 7 OS.

The time performance model is merely a means to optimize the bin combination. It is a means to compare different combinations and its accuracy per-se is not required for satisfactory results. The 10-15% difference between the predicted and actual run-times is considered reasonable.

5.13 A more Comprehensive Performance Model

Assuming that in the course of an Explicit FEM simulation, the elements are updated in groups with a constant time-step for each group (As is the case for the naive algorithm, the Spatial Decomposition Algorithm and the AVISD algorithm), the number of groups is b, the number of total element updates can be computed according to equation 67 and the number of kernels is computed by equation 68. Substituting all of them into equation 69 and obtaining α_2 from equation 70 we get:

$$T = \Omega(T_{ult}\sum_{i=1}^{b}\frac{1}{t_i}) + (\frac{W}{\xi\rho} + \frac{M}{\beta})(T_{ult}\sum_{i=1}^{b}\frac{c_i}{t_i}) \tag{77}$$

So,

$$\frac{T}{T_{ult}} = \Omega\sum_{i=1}^{b}\frac{1}{t_i} + (\frac{W}{\xi\rho} + \frac{M}{\beta})\sum_{i=1}^{b}\frac{c_i}{t_i} \tag{78}$$

In which Ω is the kernel overhead explained in section 5.4.2. W and M are also the work and memory requirement associated with updating a single element and ρ and β are the computational throughput and memory bandwidth of the device as mentioned in section 5.4.1.

The factor ξ is a reducing factor which incorporates kernel occupancy. Kernel occupancy is a measure of the utilization of the resources of a compute unit on a GPU. Lower occupancy does not necessarily mean poorer performance. Volkov[91] showed that some algorithms work better in lower occupancy. In a practical problem, the lowest occupancy Volkov used was 30%. Lower occupancies can affect the performance of a kernel.

Occupancy is affected by different factors. These factors are the local memory used, number of scalar graphical registers (SGPRs) and the number of vector graphical registers (VGPRs).

Decreasing the number of registers means loading less variables into computing registers. In the current application, this means less variables can be retained and some variables need to be overwritten and reloaded again multiple times and this leads to a higher cache-miss, which in turn will lead to a higher memory transfer requirement and higher time cost.

Next, another previously mentioned parameter called Kernel Occupancy is studies, measured and its effect of the performance is studied. On the following machine: AMD Radeon

Table 5.8: Kernel occupancy percentage for different elements

Element Type:	MINDLIN	CST	LTH
K. Occupancy:	10%	30%	20%

7970 GPU, 3GB DDR5 memory, engine clock: 925 MHz, Memory clock: 1375 MHz, memory bandwidth: 264 GB/s, 2048 Stream Processing Units, as measured by OpenCL profiler, the kernel occupancy for the following elements are as follows:

The above values in Table 5.8 are measured by OpenXL, an OpenCL code profiler. From equation 78, we get:

$$\alpha_1 = \frac{w}{\xi\rho} + \frac{m}{\beta} \tag{79}$$

in which w is the computational work in flop units needed for updating one element and m is the memory transfer need for one element. Now the value of ξ in equation 79 is measured, by having all machine parameters and measuring run-times.

For an AMD 7970, rho is around $240 GFlop/s$ for one 4-way SIMD operation and β is $264 GB/s$.

For updating each element, three kernels are executed for updating positions, forces, and velocities.

For any of the three MINDLIN, CST and LTH elements, one test case is run and the results are provided here in Tables 5.9, reftab:kernels˙cost˙LTH and 5.11.

Table 5.9: CST element costs

UNIT:	flop	flop	KB	KB			FLOP	Bytes
Item:	VALUInsts	SALUInsts	Fetch Size	Write Size	nelem	nnode	w	m
kernel 1	29	12	2979	1489	148096	76240	30.2	59
kernel 2	226	18	69154	39490	148096	76240	244	734
kernel 3	109	43	21031	1192	148096	76240	152	291
						Sum:	437	1084

Table 5.9 shows the work in FLOPS and memory transfer in Bytes that is needed by

every element, for each of the three Kernels for the AVISD method for 2D CST elements.

Table 5.10: LTH element costs

UNIT:	flop	flop	KB	KB	0	0	FLOP	Bytes
Item:	VALUInsts	SALUInsts	Fetch Size	Write Size	nelem	nnode	w	m
kernel 1	31	12	26524	15838	160000	481616	43	88
kernel 2	8084	714	804242	882888	160000	481616	8798	10545
kernel 3	114	30	85163	17192	160000	481616	144	213
						Sum:	8985	10845

Table 5.10 shows the work in FLOPS and memory transfer in Bytes that is needed by every element, for each of the three Kernels for the AVISD method for 3D LTH elements.

Table 5.11: Mindlin element costs

UNIT:	flop	flop	KB	KB	0	0	FLOP	Bytes
Item:	VALUInsts	SALUInsts	Fetch Size	Write Size	nelem	nnode	w	m
kernel 1	29	13	1198	665.97	123615	21885	42	85
kernel 2	419	17	43253	37534	123615	21885	436	654
kernel 3	280	127	28480	798.56	123615	21885	407	1338
						Sum:	885	2077

Table 5.11 shows the work in FLOPS and memory transfer in Bytes that is needed by every element, for each of the three Kernels for the AVISD method for MINDLIN plate elements.

Table 5.12: Work and memory transfer amount for each tested element type.

	CST	LTH	Mindlin
w(FLOP)	437	8985	885
m(Bytes)	1084	10845	2077

Table 5.12 summarize the demand on memory and computational units for a single element. According to the equation 79, the predicted α_1 value for various elements with

$\xi = 1$ are shown in Table 5.13:

Table 5.13: Work and memory transfer amount for each tested element type.

	CST	LTH	Mindlin
α_1	5.95e-9	1.16e-8	7.9e-8

The values from regression analysis of the test cases are shown in Table 5.14:

Table 5.14: Regression analysis cost-per-element values for each element type.

	CST	LTH	Mindlin
α_1	6.19e-9	1.14e-8	1.43e-7

By comparing Table 5.13 with Table 5.14, it is evident that the costs yielded by regression analysis are very close to the one by analyzing the algorithm using the profiler and the formulation of equation 79, except for the MINDLIN plate element. By having Table 5.8 in mind, the role of low kernel occupancy becomes evident. By choosing $\xi = 3$, the $alpha_1$ value for the MINDLIN plate element becomes: $1.55 * 10^{-7}$, which means not all the computational units can be summoned at the same time on this particular GPU for this particular algorithm and implementation. Using a higher kernel occupancy might not help the run-time much because then there will be more cache misses and more memory bandwidth will be required.

The cost of the PSO optimization is very small (15-20 seconds) compared to the actual simulation time which can be up to hours. It is possible to perform the optimization process and update the time-step requirements every few time-steps to ensure that changes in the mesh, which can be due to deformations and fracture, are considered in the stability time-step requirement. The stability time-step depends on the size of the element and the deformations during the simulation change the element size.

5.14 Defining Benchmark problems, machine-specific tuning

In order to find the system parameters α_1 and α_2, a tuning process can be designed to compute them. A few small tests and time-measurements can find the parameters that

define the performance model.

The test cases in section 5.12, can be used to capture these parameters. The number of element updates and kernels for each case is output by the program. These numbers will be used in a bilinear regression analysis to compute the parameters according to equation 69.

By defining these benchmark problems, on each machine, relevant parameters according to the current machine architecture and specifications are computed and the AVISD method can work efficiently based on any machine's specific details and be tuned specifically for that machine.

5.15 Testing on different platforms

In this section, various GPU's and operating systems are tested to show benchmark problems computing the constants in the performance model as shown in Table 5.15.

Table 5.15: Specifications of the test platforms

System	OS	GPU
1	Windows 7 PC	AMD Radeon HD 7970
2	Linux, RedHat Enterprise Server 6.3	NVIDIA GeForce GTX Titan
3	Linux, Ubuntu server 15.04	AMD Radeon HD R9 280X

The benchmark problems used here are the same as described in Tables 5.5, 5.6 and 5.7.

Table 5.16: Performance model constants for each system and each element

	Element					
	Mindlin		CST		LTH	
System	α1	α2	α1	α2	α1	α2
1	1.4E-07	5.5E-04	6.2E-09	3.5E-04	1.1E-08	3.7E-04
2	2.7E-07	2.9E-04	7.1E-09	2.2E-04	1.7E-08	1.8E-04
3	1.4E-07	3.5E-04	4.8E-09	1.4E-04	7.8E-09	2.3E-04

In Table 5.16, the values for $alpha_1$ and $alpha_2$ according to equation 69 are displayed. These values are computed based on the benchmark problems. The results indicate that the kernel overhead is much lower on Linux platforms compared to Windows.

Up to this stage, the program has computed the constants to the performance model. At this point for specific test cases, the measured run-time for each platform is reported:

Table 5.17: The measured run-time for three test cases for the three platforms

		runtime (s)		
Case	Mesh	Platform 3	Platform 1	Platform 2
1	Mindlin Type 2	670	1120	590
2	CST Type 5	966	1002	723
3	LTH Type 3	455	544	326

Table 5.17 shows the measured run-times. The mesh types in this table are illustrated in Figures 5.32, 5.39 and 5.49.

CHAPTER VI

FUTURE OF PARALLEL EXPLICIT FEM

In 2010, Nickolls et al. predicted that the GPUs will continue to scale in performance about 50 percent per year[62]. Similarly, the GPU bandwidth is predicted to grow 25% every year[72]. This predicted exponential growth is very promising and leads to the prediction that in the near future, GPU's can become a major computational tool available to engineers.

It is possible to use the current performance models presented in Chapter 5 to predict, to some extent, the performance of the AVISD method in the near future. This provides insight into the machine parameters that must improve and which improvements in future computers can maximize and optimize the application of GPU's in an explicit FEM application.

6.1 Evolution of GPU systems

As mentioned before, Nickolls predicted an exponential growth in GPU computational throughput[62]. In the time period 2010-2014, this prediction can be evaluated. All major NVIDIA GPU's up to the latest major release are included in Table 6.1, and visualized in Figure 6.1. As seen in the figure, the prediction appears to be valid for the time period shown.

Table 6.1: Single Precision throughput of NVIDIA GPUs over time since 2010.

		Throughput(GFLOPS)	
Model	**Launch**	**Actual**	**Predicted**
GeForce GTX 480	Mar-10	1345	1345
GeForce GTX 590	Mar-11	2488	2013
GeForce GTX 680	Mar-12	3090	3016
GeForce GTX Titan	Feb-13	4500	4381
GeForce GTX Titan Z	Mar-14	8122	6726
Tesla K80	Nov-14	8740	8860

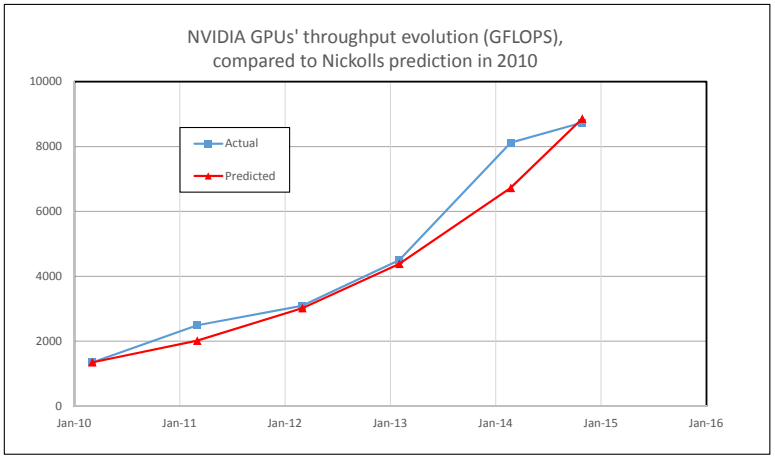

Figure 6.1: Evolution of NVIDIA GPUs compared to the predicted values by Nickolls[62]

6.2 Future systems performance model

According to equation 78, the parameters ρ, β and Ω play a key role and represent the machine specifications' effect on the run-time. Earlier in Chapter 5, discussions about ρ and β were made. The computational throughput, ρ, is expected to double every two years and β, the data bandwidth between device memory and device computing registers is expected to grow 25% every year.

The parameter Ω depends on a variety of factors such as the graphics driver software, GPU hardware, memory bandwidth between GPU and host, etc. No predictions could be found or made by the author at this point, but various rates of change in Ω can be examined

and the behavior based on different rates can be demonstrated. At this point, there is a need to generate the time costs for future years based on predictions and compare the naive method and the AVISD method, since all methods were shown to be a special case of the AVISD method, AVISD method here will only be compared with the naive method to check if there is a need for complicated algorithms in the future, or if a method as simple as the naive method is sufficient.

The parameter Ω describes the cost for each kernel run. As it decreases, more kernels can be run with less penalty. This allows having more bins and so doing less extra work such as updating elements more frequently than needed.

When Ω becomes very small, the behavior gets closer being work-optimal. As more bins are used, less unnecessary updates are made. In the following figures, the effects of changes in Ω on the number of bins in the future is examined.

The meshes used here to represent the MINDLIN, CST and LTH elements are demonstrated respectively in Figures 5.32, 5.39 and 5.49.

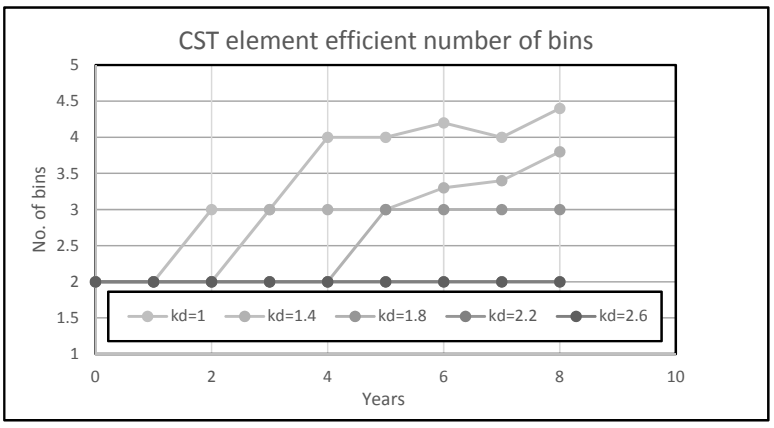

Figure 6.2: Number of efficient bins for a CST mesh, over time

k_d is the rate the kernel overhead decreases over time. $k_d = x$ means Ω , is halved every x years.

Figure 6.2 shows the efficient number of bins, with different values of k_d, predicted by paraDyn software for a 2D CST element mesh. With low values of k_d, more number of bins are required for efficient behavior, which means the naive is no longer the efficient method

and more complicated binning is required to achieve the best performance.

This is favorable because the more number of bins used, the more work efficient the process is. This phenomena is explained more during this chapter by introducing the "Inefficiency" parameter.

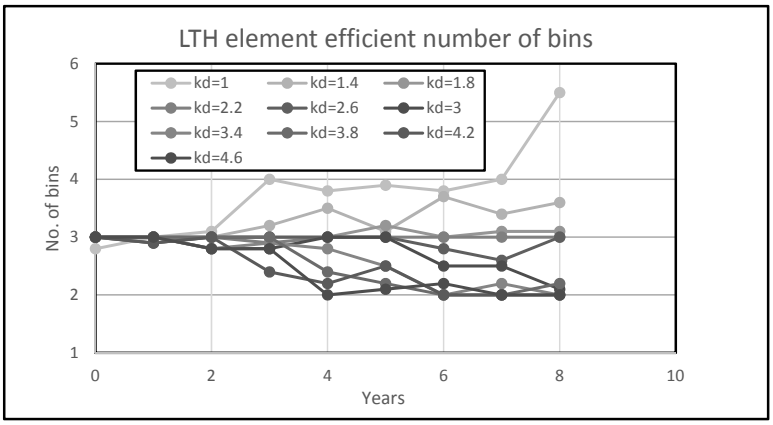

Figure 6.3: Number of efficient bin for an LTH mesh, over time

Figure 6.3 shows the efficient number of bins, with different values of k_d, predicted by paraDyn software, for an LTH 3D mesh. For k_d values lower than 2.6, the number of bins decreases, which means the AVISD method gets closer to the naive method and inefficiency increases.

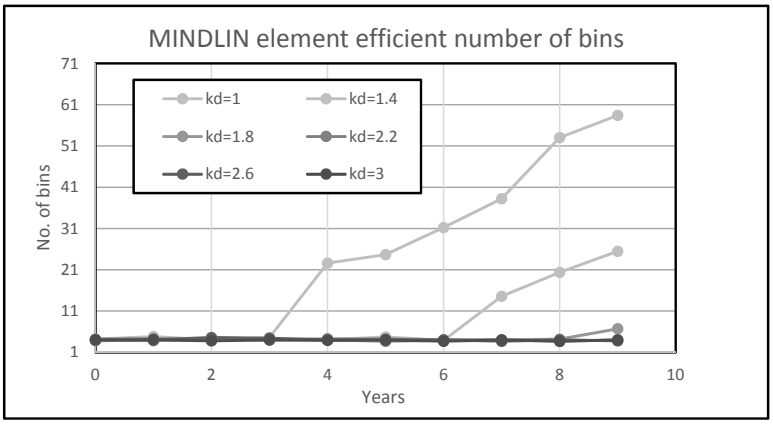

Figure 6.4: Number of efficient bin for a MINDLIN mesh, over time

Figure 6.4 shows the efficient number of bins, with different values of k_d, predicted by

paraDyn software, for a MINDLIN mesh.

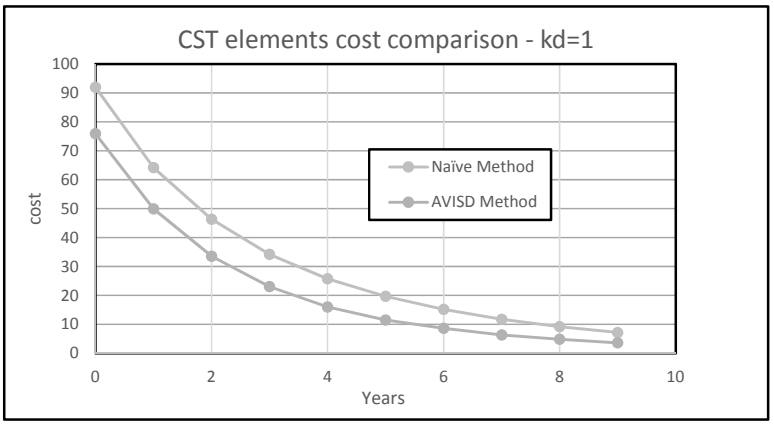

Figure 6.5: Comparing the Naive method cost with the AVISD method, CST element, $k_d = 1$

As seen in Figure 6.5, for $k_d = 1$, the AVISD method maintains its lead over the naive method strongly and even the run-time ratio is increased.

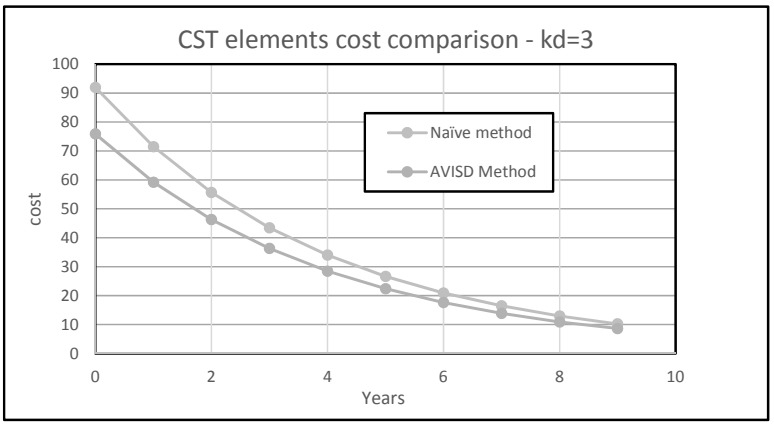

Figure 6.6: Comparing the Naive method cost with the AVISD method, CST element, $k_d = 3$

In Figure 6.6, it is evident that for $k_d = 3$, the two method get closer and closer over time and the performance difference becomes negligible.

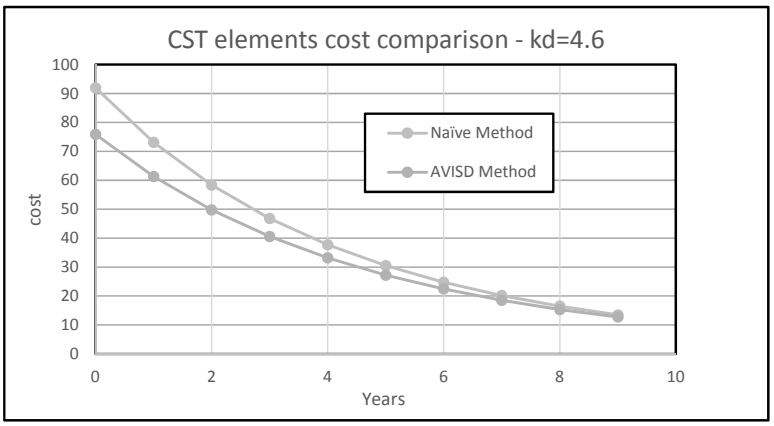

Figure 6.7: Comparing the Naive method cost with the AVISD method, CST element, $k_d = 4.6$

As seen in Figure 6.7, for $k_d = 4.6$, the results of the two methods are even closer. A slow decrease in

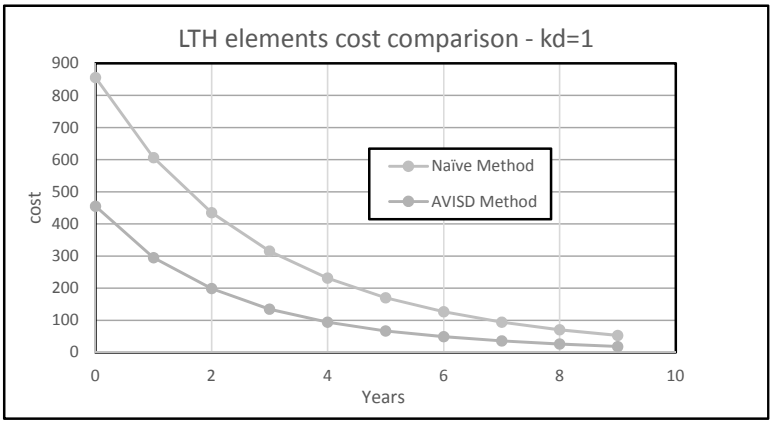

Figure 6.8: Comparing the Naive method cost with the AVISD method, LTH element, $k_d = 1$

Figure 6.8, for $k_d = 1$, shows the performance of the AVISD method compared to the naive method for a 3D LTH mesh.

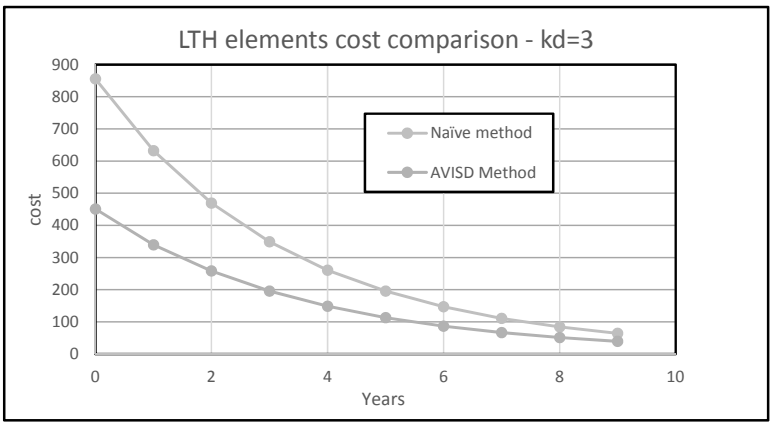

Figure 6.9: Comparing the Naive method cost with the AVISD method, LTH element, $k_d = 3$

Figure 6.9, for $k_d = 3$, shows the performance of the AVISD method compared to the naive method for a 3D LTH mesh.

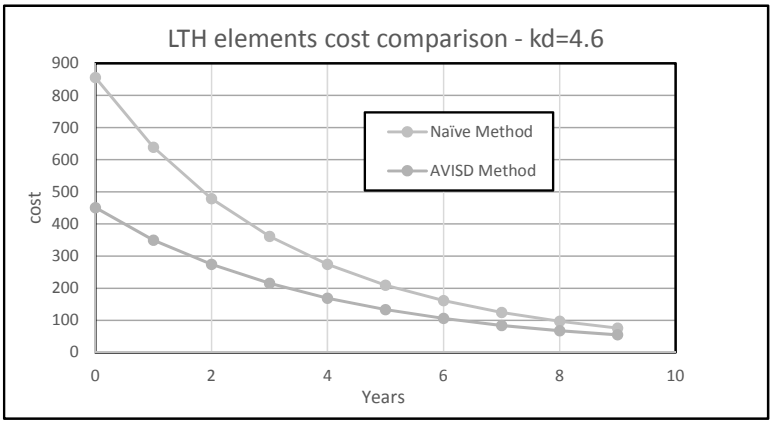

Figure 6.10: Comparing the Naive method cost with the AVISD method, LTH element, $k_d = 4.6$

Figure 6.10, for $k_d = 4.6$, shows the performance of the AVISD method compared to the naive method for a 3D LTH mesh.

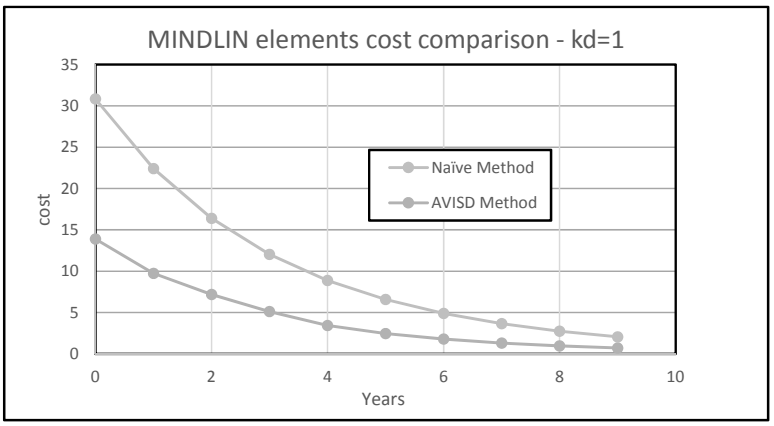

Figure 6.11: Comparing the Naive method cost with the AVISD method, MINDLIN element, $k_d = 1$

Figure 6.11, for $k_d = 1$, shows the performance of the AVISD method compared to the naive method for a MINLIN plate mesh.

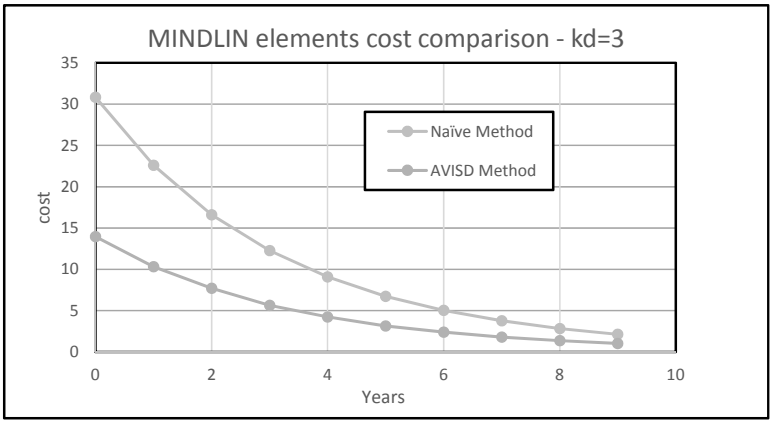

Figure 6.12: Comparing the Naive method cost with the AVISD method, MINDLIN element, $k_d = 3$

Figure 6.12, for $k_d = 3$, shows the performance of the AVISD method compared to the naive method for a MINLIN plate mesh.

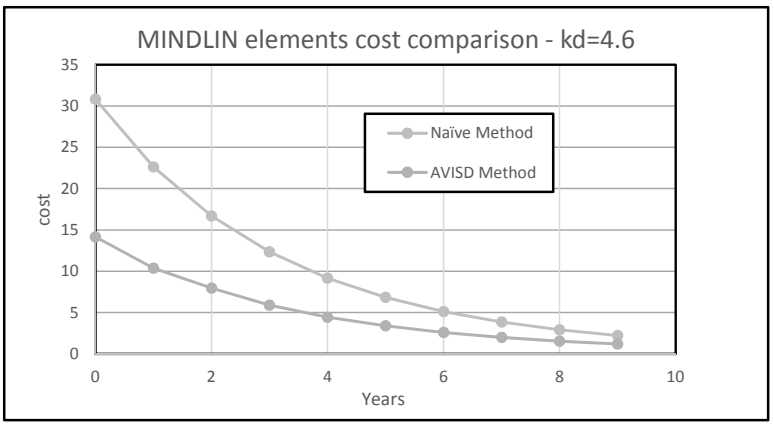

Figure 6.13: Comparing the Naive method cost with the AVISD method, MINDLIN element, $k_d = 4.6$

Figure 6.13, for $k_d = 4.6$, shows the performance of the AVISD method compared to the naive method for a MINLIN plate mesh.

6.3 Work efficiency analysis

Work inefficiency here, is defined as follows:

$$u_{min} = \sum_{i=1}^{No.of.Elems} \frac{t_f}{dt_i} \tag{80}$$

Where dt_i is the time-step of element i and t_f is the total simulation time.

Now, inefficiency I is defined:

$$I = \frac{u}{u_{min}} \tag{81}$$

In which $u_{\min}$ is the minimum number of updates and u is the real number of updates defined in equation 67.

The change in inefficiency values I through time is demonstrated as follows:

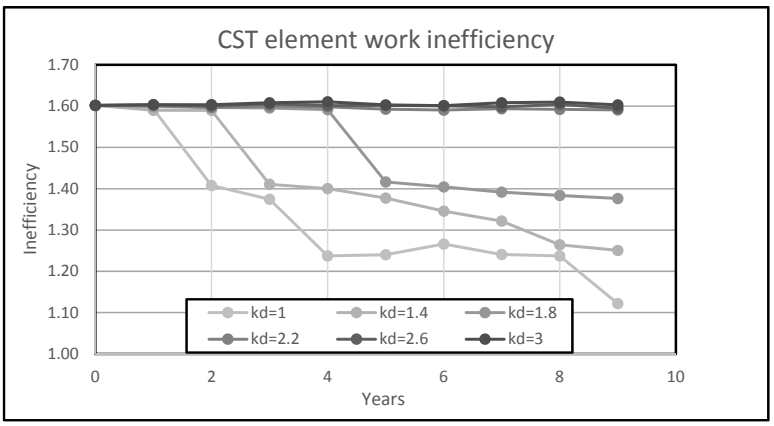

Figure 6.14: Inefficiency changes for a CST mesh, over time

Figure 6.14 shows inefficiency for different values of k_d for a 2D CST mesh.

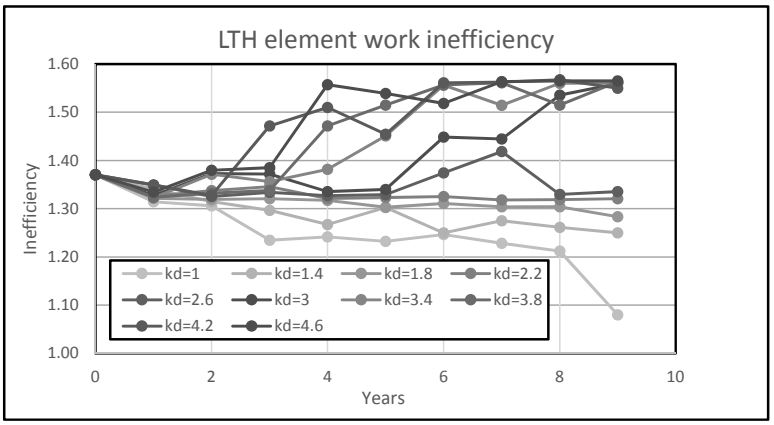

Figure 6.15: Inefficiency changes for a LTH mesh, over time

Figure 6.15 shows inefficiency for different values of k_d for a 3D LTH mesh.

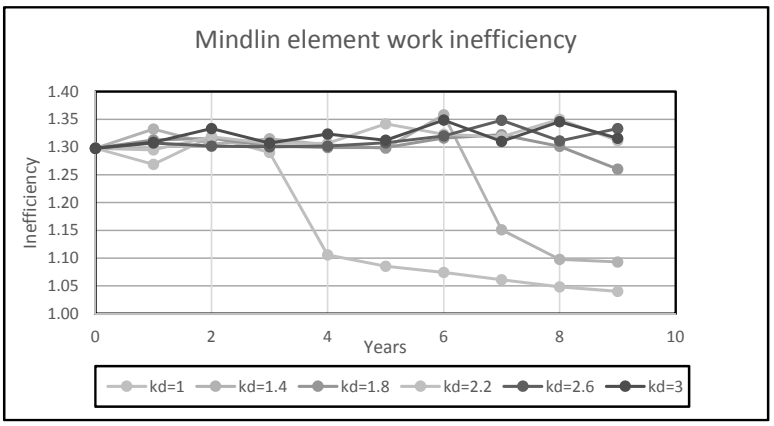

Figure 6.16: Inefficiency changes for a MINDLIN mesh, over time

Figure 6.16 shows inefficiency for different values of k_d for a MINDLIN element mesh.

6.4 Suggestions

As seen in Figures 6.2, 6.3 and 6.4, the smaller k_d, meaning the faster the kernel cost drops, the larger the number of efficient bins become. Thus the kernel cost drops, having more bins results in increased performance.

This fact can be confirmed in Figures 6.14, 6.15 and 6.16. The work inefficiency drops faster for smaller numbers of k_d.

Figures 6.5, 6.6, 6.7, 6.8, 6.9, 6.10, 6.11, 6.12 and 6.13 show the larger k_d, the closer AVISD method's cost gets to the naive method's cost. This confirms the results shown in the inefficiency charts.

According to the above results, although the behavior also depends on the mesh type and configuration, a k_d value of at least 2.2 is required to maintain the current inefficiency and not lose more efficiency.

Also, if k_d is more that that, meaning the kernel cost does not drop as much as other machine parameters, at some point in the future, the naive method can potentially become the most efficient solution. This is not a desirable result because the naive method does the most unnecessary work as shown in Chapter 3. In an explicit dynamic analysis with a GPU, the role of kernel overhead is one of the primary factors in the performance and as

mentioned above, improvement in the kernel cost is needed to maximize the performance gained from a GPU device.

Finally, the suggestion are:

- The change in kernel cost, which depends on the architecture and GPU driver software has to keep up with the growth of the computational power of the GPU in order to prevent further increase in "inefficiency". For this purpose, a value of k_d near 2.5 is needed, which means the kernel cost must be halved every 2.5 years in order to maintain the level of utilization of the GPU. An increase in the "Inefficiency" factor means updating elements more than required and therefore performing more computations than needed.

- If the kernel cost cannot keep up with the computational power, especially for $k_d > 4$, the naive method can become the lead method and the use of complicated algorithms is not required in most cases, especially in simple elements such as CST and LTH. In elements with more complicated internal force function, this issue is less critical. The underlying software that is needed for the kernel run is the major contributor to a high kernel overhead.

- Based on the studies in Section 5.15, Linux platforms have less software overhead and less kernel overhead as a result, exhibit better performance.

- The memory transfer cost is a major cost. Providing higher memory bandwidth can then significantly increase the performance of an Explicit FEM analysis.

CHAPTER VII

CONCLUSIONS AND RECOMMENDATIONS

7.1 Summary and conclusions

The present research demonstrates the potential of Graphical processing units for the explicit finite elements method (FEM) which is a computationally demanding problem in computational mechanics.

All of the currently known explicit FEM algorithms were studied, examined and GPU-friendly algorithms were designed, introduced, and tested on GPU's. Finally, the reasons behind the success of some algorithms over others on a GPU system were studied and at the end, the AVISD (Asynchronous Variational Integrator Spatial Decomposition) algorithm, which is a versatile and flexible algorithm that encompasses all current algorithms, is introduced.

The flexibility of this algorithm makes it possible to tune this algorithm based on any underlying hardware, the finite element type, or mesh configuration. For this reason, the AVISD algorithm is further examined and a general cost performance model is introduced.

After this, using the Particle Swarm Optimization (PSO) method and using the cost model as the objective function, an optimization process is designed to first tune the performance model's constants based on the machine, and then tune the algorithm based on the mesh configuration to theoretically achieve maximum speedup for any problem.

Furthermore, the performance of this algorithm is examined for the hardware advances predicted for the near future and the cost trends are demonstrated.

The conclusions can be summarized as:

1. Explicit FEM analysis on a GPU is highly dependent on high concurrency and for this purpose, updating elements in large groups is essential.

2. Data coherence also plays a major role and using data that are stored physically adjacent to each other is important. For this purpose, updating elements that are

physically adjacent is essential.

3. By using the AVISD method, all current methods are concentrated in one single flexible method.

4. A performance model is presented and tested to be able to formulate and predict the cost of simulation.

5. Using the Performance model and the Particle Swarm Optimization method, the AVISD method can be tuned for any computer and any given problem. This is the first work in Explicit FEM that offers problem dependent algorithm optimization for GPU's. The AVISD method can be developed for any given element and the performance model can be tuned for any machine.

6. The size of the problem that can be solved on a GPU depends directly on the device memory

7.2 Advancements and Contributions to the State-of-the-Art

The main contributions of this dissertation are

- Designing a GPU version of the "Spatial Decomposition" algorithm.
- Introducing the "AVI coloring" algorithm.
- Designing the first mesh-aware and machine-aware explicit FEM algorithm that encompasses all other algorithms, called the Asynchronous Variational Integrator Spatial Decomposition (AVISD) algorithm.
- proposing the first mesh-aware and machine-aware performance model for an explicit FEM analysis.
- Employing the Particle Swarm Optimization method to tune and adapt the AVISD algorithm.
- Predicting the cost of the proposed algorithm in the near future using the derived performance model.

7.3 The paraDyn software

During the course of this thesis, all the code is developed as a piece of software named paraDyn, by the author of this dissertation, which can perform explicit FEM analysis using the various algorithms mentioned in this thesis.

The code is developed using C++ language and the device-side codes are developed using the OpenCL language. The software accepts the simulation specifications through multiple input files such as simulation details, element details and material properties. Requested outputs can be generated as text files and animation of the results is done through generating input to be post-processed using the TECPLOT software.

7.4 Recommendations for further research

As the current study is limited, the following topics are recommended for further research:

1. Extending the application to multi-GPU machines with shared memory and/or distributed memory machines. As shown in section 3.4.1, the studied GPU's showed potential for a scalable Explicit FEM analysis on a shared memory system, although further research is needed.

2. Extending the performance model to encompass multi-GPU applications and include network bandwidth and latency. The communication cost is another factor that can become more critical if the data has to be constantly transferred during simulation over a Local Area Network (LAN).

3. Investigating the renumbering of the nodes based on the nonlinear behavior of each problem during the simulation to maximize performance. In this research, the nodes are numbered in the time-step order. This will help elements in the same bin have a close element index. However, further research is needed to maximize data coherence based on the physical placement of the elements as well as the time-step.

4. Research adaptive processes and predicting the nonlinear regions in an adaptive process to maximize performance. Computing nonlinear forces are usually more costly and accounting for the cost difference can lead to a better performance algorithm.

5. Introducing graph partitioning algorithms for the AVISD method to minimize communication costs in a multi-GPU application. Each bin or element can be represented by a graph node and the communication cost can be the weight on the graph. Suitable partitioning algorithms according to the graph model can help reduce the amount of communication in a multi-device solution.

6. Research on possible parallelization over the time domain for linear and possibly nonlinear problems.

7. Investigate the influence of non-diagonal damping on performance.

8. More research is needed to properly address the physical effects and errors caused by a diagonal damping matrix and how to derive a proper diagonal damping matrix.

REFERENCES

[1] Adeli, H. and Kamal, O., *Parallel Processing in Structural Engineering*. Routledge, 1993.

[2] Ahmadzadeh, R., "Particle Swarm optimizaion Package for Matlab," 2014.

[3] Alhadeff, A., Celes, W., and Paulino, G. H., "Mapping Cohesive Fracture and Fragmentation Simulations to Graphics Processor Units," *International Journal for Numerical Methods in Engineering*, vol. 103, pp. 859–893, Sept. 2015.

[4] Asgari, B., Osman, S. A., and Adnan, A., "Three-dimensional finite element modelling of long-span cable-stayed bridges.," *IES Journal Part A: Civil & Structural Engineering*, vol. 6, pp. 258–269, Nov. 2013.

[5] Bahcecioglu, T. and Kurc, O., "Nonlinear dynamic finite element analysis with GPU," in *Fourteenth International Conference on Computing in Civil and Building Engineering*, 2012.

[6] Bathe, K. J., *Finite element procedures*. Prentice Hall: Englewood Cliffs, N.J., 1996.

[7] Bathe, K. J. and Wilson, E. L., "Stability and accuracy analysis of direct integration methods," Jan. 1973.

[8] Belytschko, T., "Partitioned and Adaptive Algorithms for Explicit Time Integration," in *Nonlinear Finite Element Analysis in Structural Mechanics SE - 29* (Wunderlich, W., Stein, E., and Bathe, K.-J., eds.), pp. 572–584, Springer Berlin Heidelberg, 1981.

[9] Carter, W. T., Sham, T. L., and Law, K. H., "A parallel finite element method and its prototype implementation on a hypercube," *Computers and Structures*, vol. 31, pp. 921–934, 1989.

[10] Casadei, F. and Halleux, J., "Spatial Time Step Partitioning in Explicit Fast Transient Dynamics," *Report EUR 23062 EN*, 2008.

[11] Chetverushkin, B., Shilnikov, E., and Davydov, A., "Numerical simulation of the continuous media problems on hybrid computer systems," *Advances in Engineering Software*, vol. 60-61, pp. 42–47, June 2013.

[12] Chung, J. and Lee, J. M., "A new family of explicit time integration methods for linear and non-linear structural dynamics," *International Journal for Numerical Methods in Engineering*, vol. 37, pp. 3961–3976, Dec. 1994.

[13] Comas, O., Taylor, Z., Allard, J., Ourselin, S., Cotin, S., and Passenger, J., "Efficient Nonlinear FEM for Soft Tissue Modelling and Its GPU Implementation within the Open Source Framework SOFA," in *Biomedical Simulation SE - 4* (Bello, F. and Edwards, P., eds.), vol. 5104 of *Lecture Notes in Computer Science*, pp. 28–39, Springer Berlin Heidelberg, 2008.

[14] Czechowski, K., Battaglino, C., McClanahan, C., Iyer, K., Yeung, P.-K., and Vuduc, R., "On the Communication Complexity of 3D FFTs and Its Implications for Exascale," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, (New York, NY, USA), pp. 205–214, ACM, 2012.

[15] D'Amato, J. and Vénere, M., "A CPU GPU framework for optimizing the quality of large meshes," *Journal of Parallel and Distributed Computing*, vol. 73, pp. 1127–1134, Aug. 2013.

[16] Daniel, W. J. T., "A study of the stability of subcycling algorithms in structural dynamics," *Computer Methods in Applied Mechanics and Engineering*, vol. 156, pp. 1–13, Apr. 1998.

[17] Dick, C., Georgii, J., and Westermann, R., "A real-time multigrid finite hexahedra method for elasticity simulation using CUDA," *Simulation Modelling Practice and Theory*, vol. 19, pp. 801–816, Feb. 2011.

[18] Duarte, L., Celes, W., Pereira, A., M. Menezes, I., and Paulino, G., "PolyTop++: an efficient alternative for serial and parallel topology optimization on CPUs & GPUs," *Structural and Multidisciplinary Optimization*, pp. 1–15, 2015.

[19] Dziekonski, A., Lamecki, A., and Mrozowski, M., "Hybrid GPU-CPU Multilevel Preconditioner for Solving Large Systems of FEM Equations," in *GPU Technology conference*, p. 2011, 2011.

[20] Dziekonski, A., Sypek, P., Lamecki, A., and Mrozowski, M., "Finite Element Matrix Generation on a Gpu," *Progress In Electromagnetics Research*, vol. 128, no. May, pp. 249–265, 2012.

[21] Elble, J. M., Sahinidis, N. V., and Vouzis, P., "GPU computing with Kaczmarz's and other iterative algorithms for linear systems.," *Parallel computing*, vol. 36, pp. 215–231, June 2010.

[22] Fong, W., Darve, E., and Lew, A., "Stability of asynchronous variational integrators," *Journal of Computational Physics*, vol. 227, pp. 8367–8394, Sept. 2008.

[23] Fulton, R., Goehlich, D., and Ou, R., "Structural Dynamics Methods for Parallel Supercomputers," *Research Report to MSC*, 1987.

[24] Georgescu, S., Chow, P., and Okuda, H., "GPU Acceleration for FEM-Based Structural Analysis," *Archives of Computational Methods in Engineering*, vol. 20, pp. 111–121, Apr. 2013.

[25] Göddeke, D., Strzodka, R., and Turek, S., "Accelerating Double Precision FEM Simulations with GPUs," in *Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique*, Sept. 2005.

[26] Göddeke, D., Wobker, H., Strzodka, R., Mohd-Yusof, J., McCormick, P., and Turek, S., "Co-Processor Acceleration of an Unmodified Parallel Solid Mechanics Code with {FEASTGPU}," *International Journal of Computational Science and Engineering (IJCSE)*, vol. 4, pp. 254–269, Nov. 2009.

[27] Gravouil, A. and Combescure, A., "Multi-time-step explicitimplicit method for nonlinear structural dynamics," *International Journal for Numerical Methods in Engineering*, no. July 1999, pp. 199–225, 2001.

[28] Gustafson, J., "Brents Theorem," in *Encyclopedia of Parallel Computing SE - 80* (Padua, D., ed.), pp. 182–185, Springer US, 2011.

[29] Halleux, J. and Casadei, F., *Spatial Time Step Partitioning in EUROPLEXUS*. JRC EUR Report N. 22464 EN, 2006.

[30] Hallquist, J. O. and Benson, D. J., *DYNA-3D: Users Manual (Nonlinear Dynamic Analysis of Sol- ids in Three Dimensions). Revision 2*. University of California, Lawrence Livermore National Laboratory, Report UCID-19592, 1986.

[31] Hockney, R. W. and Jesshope, C. R., *Parallel Computers: Architecture, Programming and Algorithms*. Bristol: Adam Hilger Ltd, 1981.

[32] Honda, R., Sakai, H., and Sawada, S., "Non-iterative time integration scheme for non-linear dynamic FEM analysis," *Earthquake Engineering & Structural Dynamics*, vol. 33, pp. 111–132, Jan. 2004.

[33] Huang, H. and Hinton, E., "Elasto-plastic dynamic analysis of plate and shell-structures using a new nine node element," *Material nonlinearity in vibration problems*, pp. 41–60, 1985.

[34] Huang, J.-C., Jiao, X., Fujimoto, R. M., and Zha, H., "DAG-guided Parallel Asynchronous Variational Integrators with Super-elements," in *Proceedings of the 2007 Summer Computer Simulation Conference*, SCSC '07, (San Diego, CA, USA), pp. 691–697, Society for Computer Simulation International, 2007.

[35] Itoh, S. I., Nakata, S., Hirokawa, Y., and Taku, "High Performance Computing of Meshless Time Domain Method on Multi-GPU Cluster," *Journal of Physics: Conference Series*, vol. 574, no. 1, p. 12106, 2015.

[36] Jayasooriya, R., Thambiratnam, D. P., and Perera, N. J., "Blast response and safety evaluation of a composite column for use as key element in structural systems," *Engineering Structures*, vol. 61, pp. 31–43, Mar. 2014.

[37] Joldes, G. R., Wittek, A., and Miller, K., "Real-Time Nonlinear Finite Element Computations on GPU - Application to Neurosurgical Simulation.," *Computer methods in applied mechanics and engineering*, vol. 199, pp. 3305–3314, Dec. 2010.

[38] Kale, K. G. and Lew, A. J., "Parallel asynchronous variational integrators," no. October 2006, pp. 291–321, 2007.

[39] Kandasamy, V., "Parallel FEM Simulation Using GPUs," in *23rd European Conference Forum Bauinformatik*, 2011.

[40] Kane, C., Marsden, J. E., Ortiz, M., and West, M., "Variational integrators and the Newmark algorithm for conservative and dissipative mechanical systems," *International Journal for Numerical Methods in Engineering*, vol. 49, no. 10, pp. 1295–1325, 2000.

[41] Karatarakis, a., Metsis, P., and Papadrakakis, M., "GPU-acceleration of stiffness matrix calculation and efficient initialization of EFG meshless methods," *Computer Methods in Applied Mechanics and Engineering*, vol. 258, pp. 63–80, May 2013.

[42] Kennedy, J. and Eberhart, R., "Particle swarm optimization," *Neural Networks, 1995. Proceedings., IEEE International Conference on*, vol. 4, pp. 1942–1948 vol.4, 1995.

[43] Kerr, A., Campbell, D., and Richards, M., "QR decomposition on GPUs," *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pp. 71–78, 2009.

[44] Khronos Group, "The Khronos Group Releases OpenCL 1.0 Specification," *(Press release)*, Dec. 2008.

[45] King, R., "Implementation of an Element-by-Element Solution Algorithm for the Finite Element Method on a Coarse-Grained Parallel Computer," *Computer Methods in Applied Mechanics and Engineering*, vol. 65, pp. 47–59, 1987.

[46] Kolman, R., Cho, S., and Park, K., "Explicit Time Integrations for Finite Element Computations of Wave Propagation Problems," *Výpočty konstrukcí metodou konečných prvk*, pp. 2–5, 2013.

[47] Komatitsch, D., Erlebacher, G., Göddeke, D., and Michéa, D., "High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster," *Journal of Computational Physics*, vol. 229, pp. 7692–7714, Oct. 2010.

[48] Komatitsch, D., Michéa, D., and Erlebacher, G., "Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA," *Journal of Parallel and Distributed Computing*, vol. 69, pp. 451–460, May 2009.

[49] Lang, J. and Rünger, G., "Dynamic Distribution of Workload between CPU and GPU for a Parallel Conjugate Gradient Method in an Adaptive FEM," *Procedia Computer Science*, vol. 18, pp. 299–308, Jan. 2013.

[50] Lee, K. S., "A New Methodology for Nonlinear Dynamic Analysis of Reinforced-Concrete Buildings Including Their Pile Foundations," *Advances in Structural Engineering*, vol. 17, pp. 67–82, Jan. 2014.

[51] Lew, a., Marsden, J. E., Ortiz, M., and West, M., "Asynchronous Variational Integrators," *Archive for Rational Mechanics and Analysis*, vol. 167, pp. 85–146, Apr. 2003.

[52] Lew, A., Marsden, J. E., Ortiz, M., and West, M., "Variational time integrators," *International Journal for Numerical Methods in Engineering VO - 60*, no. 1, p. 153, 2004.

[53] Lew, a., Marsden, J. E., Ortiz, M., and West, M., "Variational time integrators," *International Journal for Numerical Methods in Engineering*, vol. 60, pp. 153–212, May 2004.

[54] Lin, H.-I., "A Fast and Unified Method to Find a Minimum-Jerk Robot Joint Trajectory Using Particle Swarm Optimization," *Journal of Intelligent & Robotic Systems*, vol. 75, no. 3-4, pp. 379–392, 2014.

[55] Liu, H., Cai, Z., and Wang, Y., "Hybridizing particle swarm optimization with differential evolution for constrained numerical and engineering optimization," *Applied Soft Computing Journal*, vol. 10, no. 2, pp. 629–640, 2010.

[56] Macio, P., Paszewski, P., and Banaś, K., "3D finite element numerical integration on GPUs," *Procedia Computer Science*, vol. 1, pp. 1093–1100, May 2010.

[57] Mani, K. and Mavriplis, D., "Spatially non-uniform time-step adaptation for functional outputs in unsteady flow problems," *48th AIAA Aerospace Sciences Meeting Including ...*, pp. 1–22, 2010.

[58] Marsden, J. E. and West, M., "Discrete mechanics and variational integrators.," *Acta Numerica*, vol. 10, p. 357, May 2001.

[59] Miranker, W. and Linger, W., "Parallel method for the Numerical Integra- tion of Ordinary Differential Equations," *Mathematics of Computation*, vol. 21, pp. 303–320, 1967.

[60] Neal, M. O. and Belytschko, T., "Explicit-explicit subcycling with non-integer time step ratios for structural dynamic systems," *Computers & Structures*, vol. 31, pp. 871–880, Jan. 1989.

[61] Newmark, N., "A method of computation for structural dynamics," *Journal of Engineering Mechanics, ASCE, 85 (EM3)*, pp. 67–94, 1959.

[62] Nickolls, J. and Dally, W., "The GPU computing era," *IEEE micro*, pp. 56–69, 2010.

[63] Nielsen, C. V., Zhang, W., Alves, L. M., Bay, N., and Martins, P. a. F., *Modeling of Thermo-Electro-Mechanical Manufacturing Processes*. SpringerBriefs in Applied Sciences and Technology, London: Springer London, 2013.

[64] Noh, G. and Bathe, K.-J., "An explicit time integration scheme for the analysis of wave propagations," *Computers & Structures*, vol. 129, pp. 178–193, Dec. 2013.

[65] NVIDIA Corporation, "NVIDIA CUDA Programming Guide, Version 5.5," *http://docs.nvidia.com/cuda/cuda-c-programming-guide*, 2013.

[66] Owen, D. R. J. and Hinton, E., *Finite elements in plasticity: theory and practice*. Pineridge Press, 1980.

[67] Owen, D. and Li, Z., "Elastic-plastic dynamic analysis of anisotropic laminated plates," *Computer Methods in Applied Mechanics and Engineering*, vol. 70, pp. 349–365, Oct. 1988.

[68] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C., "GPU Computing," 2008.

[69] Papadakis, L., Schober, A., and Zaeh, M. F., "Numerical investigation of the influence of preliminary manufacturing processes on the crash behaviour of automotive body assemblies," *International Journal of Advanced Manufacturing Technology*, vol. 65, pp. 867–880, Mar. 2013.

[70] Papadrakakis, M., Stavroulakis, G., and Karatarakis, A., "A new era in scientific computing: Domain decomposition methods in hybrid CPUGPU architectures," *Computer Methods in Applied Mechanics and Engineering*, vol. 200, pp. 1490–1508, Mar. 2011.

[71] Persson, P.-O. and Strang, G., "A Simple Mesh Generator in MATLAB," *Siam Review*, vol. 46, no. 2, pp. 329–345, 2004.

[72] Power, J., Li, Y., Hill, M. D., Patel, J. M., and Wood, D. A., "Toward GPUs being mainstream in analytic processing," 2015.

[73] Probert, E. J., Hassan, O., Morgan, K., and Peraire, J., "Adaptive explicit and implicit finite element methods for transient thermal analysis," *International Journal for Numerical Methods in Engineering*, vol. 35, no. 4, pp. 655–670, 1992.

[74] Rakić, P., Milašinović, D., Živanov, v., Suvajdžin, Z., Nikolić, M., and Hajduković, M., "MPICUDA parallelization of a finite-strip program for geometric nonlinear analysis: A hybrid approach," *Advances in Engineering Software*, vol. 42, pp. 273–285, May 2011.

[75] Rangarajan, R., Lew, A., and Buscaglia, G. C., "A discontinuous-Galerkin-based immersed boundary method with non-homogeneous boundary conditions and its application to elasticity," *Computer Methods in Applied Mechanics and Engineering*, vol. 198, pp. 1513–1534, Apr. 2009.

[76] Rao, A. R. M., Rao, T. V. S. R. A., and Dattaguru, B., "Comparative efficiencies of three parallel algorithms for nonlinear implicit transient dynamic analysis," *Sadhana*, vol. 29, no. February, pp. 57–81, 2004.

[77] Sha, Y. and Hao, H., "A simplified approach for predicting bridge pier responses subjected to barge impact loading," *Advances in Structural Engineering*, vol. 17, pp. 11–23, Jan. 2014.

[78] Shi, Y. and Eberhart, R., "A modified particle swarm optimizer," *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360)*, pp. 69–73, 1998.

[79] Siegel, H. J., "Partitionable SIMD computer system interconnection network universality," in *Proc. 16th Annual Allerton Conference on Communications, Control, and Computing*, pp. 586–595, 1978.

[80] Soltani, B., Mattiasson, K., and Samuelsson, A., "Implicit and dynamic explicit solutions of blade forging using the finite element method," *Journal of materials Processing Technology*, vol. 45, pp. 69–74, 1994.

[81] Storaasli, O., Ransom, J., and Fulton, R., "Structural Dynamic Analysis on a Parallel Computer: The Finite Element Machine," *Computers and Structures*, vol. 26, no. 4, pp. 551–559, 1987.

[82] Tabatabaiefar, R., Hamid, S., Fatahi, B., and Samali, B., "Seismic Behavior of Building Frames Considering Dynamic Soil-Structure Interaction.," *International Journal of Geomechanics*, vol. 13, pp. 409–420, Aug. 2013.

[83] Tahmasebi, P., Sahimi, M., Marietho z, G., and Hezarkhani, A., "Accelerating geostatistical simulations using graphics processing units (GPU)," *Computers & Geosciences*, vol. 46, pp. 51–59, Sept. 2012.

[84] Taylor, Z. A., Cheng, M., and Ourselin, S., "High-Speed Nonlinear Finite Element Analysis for Surgical Simulation Using Graphics Processing Units," 2008.

[85] Terrier, J. M. and Bobineau, J. P., "Zoom in On the Details: Multi-Domain Technology for Impact and Crash Simulation," *Tech Briefs Media Group*, 2013.

[86] Top500.org, *Top 500 list.* http://www.top500.org/lists/2013/11/ , November 2013.

[87] Trienekens, H. W. J. M., "Parallel Branch and Bound on an MIMD System ; CU-CS-354-87," *Paper 340, Computer Science Technical Reports.*, 1987.

[88] Verschoor, M. and Jalba, A. C., "Analysis and performance estimation of the Conjugate Gradient method on multiple GPUs," *Parallel Computing*, vol. 38, pp. 552–575, Oct. 2012.

[89] Veselov, A. P., "Integrable discrete-time systems and difference operators," *Functional Analysis and Its Applications*, vol. 22, no. 2, pp. 83–93, 1988.

[90] Veselov, A. P., "Integrable Lagrangian correspondences and the factorization of matrix polynomials," *Functional Analysis and Its Applications*, vol. 25, no. 2, pp. 112–122, 1991.

[91] Volkov, V., "Better performance at lower occupancy," 2010.

[92] Vuduc, R. and Czechowski, K., "What GPU Computing Means for High-End Systems," *Micro, IEEE*, vol. 31, pp. 74–78, July 2011.

[93] WANG, J.-C. and SUNG, Y.-H., "A Novel Fast Mode Decision Algorithm for H.264/AVC Using Particle Swarm Optimization," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E96.A, no. 11, pp. 2154–2160, 2013.

[94] Wendlandt, J. M. and Marsden, J. E., "Mechanical integrators derived from a discrete variational principle," *Physica D: Nonlinear Phenomena*, vol. 106, pp. 223–246, Aug. 1997.

[95] Wu, S. R. and Gu, L., *Introduction to the Explicit Finite Element Method for Nonlinear Transient Dynamics.* Wiley, 2012.

[96] Wu, S. R., "Lumped mass matrix in explicit finite element method for transient dynamics of elasticity," *Computer Methods in Applied Mechanics and Engineering*, vol. 195, no. 4447, pp. 5983–5994, 2006.

[97] Wu, W. and Heng, P. A., "A hybrid condensed finite element model with GPU acceleration for interactive 3D soft tissue cutting," *Computer Animation and Virtual Worlds*, vol. 15, pp. 219–227, July 2004.

[98] Zegard, T. and Paulino, G., "Toward GPU accelerated topology optimization on unstructured meshes," *Structural and Multidisciplinary Optimization*, vol. 48, no. 3, pp. 473–485, 2013.

[99] Zhao, J.-g. and Niu, L., "Particle swarm optimization-based Fast Relevance Vector Machine for forecasting dissolved gases content in power transformer oil," no. 1, pp. 290–293, 2013.

VITA

Seyed Parsa Banihashemi was born on September 22, 1985 in Tehran, Iran where he grew up. After graduating high school in 2003, he attended University of Tehran. He obtained his bachelor's degree in Civil and Environmental Engineering in 2007. He continued his studies in Sharif University of Technology in Tehran where he pursued a Master's degree in Structural Engineering. After graduating in 2010, he traveled to the United States to attend Georgia Institute of Technology doctorate program in Civil Engineering. In 2013, he got his Master's degree in Computational Science and Engineering (CSE) from the Georgia Institute of Technology, while pursuing his PhD studies with an emphasis on computational mechanics and a minor in CSE.