ASYNCHRONOUS PARALLEL SOLVER FOR HYPERBOLIC PROBLEMS VIA THE SPACETIME DISCONTINUOUS GALERKIN METHOD

BY

AMIT MADHUKAR

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Mechanical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Professor Robert B. Haber

ABSTRACT

This thesis presents a parallel Space Time Discontinuous Galerkin (SDG) finite element method which makes use of the method's unstructured mesh generation and localized solution technique to achieve a high level of parallel scalability. Our SDG method is different from most traditional adaptive finite element methods in that the solution process generates fully unstructured spacetime grids that satisfy a special *causality constraint* ensuring that computations can occur locally on small cluster of spacetime elements. The resulting asynchronous solution scheme offers several desirable features: element-wise conservation of solution quantities, strong stability properties without the need for explicit stabilization, local mesh adaptivity operations and linear complexity in the number of spacetime elements.

In this thesis we propose an algorithm that effectively parallelizes the *Tent Pitcher* algorithm developed by [1] using the *POSIX Thread* (or Pthread) parallel execution model. Multiple software threads can simultaneously and asynchronously perform patch computations by advancing vertices in time. By enforcing the causality constraint on the time step, we can guarantee that each thread only performs calculations using data computed previously. Additionally, improvements to the adaptivity scheme allow for local mesh refinement and coarsening while maintaining globally conforming triangulation. Numerical tests show that our algorithm achieves high parallel scalability using shared-memory parallelization.

To my family

ACKNOWLEDGMENTS

First and foremost, I would like to express my gratitude to Professor Robert Haber who served as my advisor. I am indebted to him for his insight and guidance which has proven to be invaluable during my graduate studies. I am also thankful to Dr. Reza Abedi for taking the time to guide me through the intricacies of the SDG code, and to Dr. Volodymyr Kindratenko for providing the computational resources and expertise without which none of this would have been possible.

I would also like to thank my fellow students, Ian E. McNamara and Raj Kumar Pal, whose help and encouragement was greatly appreciated.

I would not have come this far were it not for the constant love and support of my family, for which I am deeply grateful.

Last, but by no means the least, I gratefully acknowledge financial support from the College of Engineering through the Carver Fellowship and the Graduate College through the Illinois Distinguished Fellowship program.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

In this thesis we develop a parallel solver for the spacetime discontinuous Galerkin (SDG) finite element method which takes advantage of the method's localized solution technique to achieve a high degree of parallel scalability. The SDG method has two main differences over traditional time-stepping methods that makes parallelism extremely effective. First, by constructing meshes that cover the entire spacetime domain, the SDG method does not require a fixed global time step on the mesh at each iteration. Instead, it imposes a special *causality constraint* to advance a single vertex forward in time, creating elements in *spacetime*. This constraint ensures that the solution in the new element depends only on information from previously computed elements. Thus, the solution for new elements can be computed locally and independent of other element computations. The process creates a fully unstructured and asynchronous mesh. Second, the SDG method performs mesh adaptivity operations on the same granularity as element computations, ensuring that the mesh can be locally refined or coarsened in response to *a posteriori* error estimates computed locally. This allows mesh adaptivity to occur simultaneously with element computations.

The resulting finite element method offers several desirable features over traditional time-marching methods. Linear and angular momentum are exactly balanced over every spacetime element. It is dissipative so no extra stabilization is required to prevents spurious oscillations when shocks are present. Adaptive meshing operations are computed exactly, without the need for error-prone projections. The computational complexity is linear in the number of of spacetime elements. These features allow for efficient parallelization of the existing serial SDG code developed earlier by [1, 2, 3].

1.1 Literature Survey

Since the early days of parallel computing, efforts have been made to efficiently parallelize finite element (FE) codes in an effort to reduce computation times. The *domain decomposition* technique is the traditional way to divide a FE problem into multiple, smaller subproblems that can be solved simultaneously [4, 5, 6]. In such a method, the decomposition must be done in a manner that minimizes dependence between the subproblems. This way, parallel processors can be assigned to perform computations on a subproblem with minimum communication between processors. Load balancing must also be performed to dynamically redistribute workload evenly among each processor. Popular decomposition schemes include graph-based techniques [7] and geometry-based techniques [8, 9]. One noteworthy method that has considerable research interest is the finite element tearing and interconnecting (FETI) method [9, 10]. This method requires fewer interprocessor communications than traditional domain decomposition schemes by partitioning the spatial domain into a set of totally disconnected subdomains. The global continuity across subdomain interfaces is enforced via Lagrange multipliers rather than explicit interprocessor dependence.

The solution to elliptic partial differential equations (PDEs) generally depend on all points of the domain, thus leading to global coupling and the need for domain decomposition techniques outlined above. Hyperbolic PDEs on the other hand have finite propagation speed of disturbances. This allows for more efficient parallelization schemes as coupling between elements in localized. The discontinuous Galerkin (DG) finite element method is particularly efficient for hyperbolic problems wherein discontinuous basis functions are used to formulate the Galerkin approximation [11, 12, 13]. Early works to parallelize the DG method have utilized the domain decomposition technique [14, 15], thus squandering the advantages of the underlying method. In this thesis, we have used the unstructured mesh generation capabilities inherent to the "Tent Pitching" method [1, 2] and developed a fully asynchronous parallel method.

1.2 Organization

The thesis is organized as follows. Chapter 2 provides background on the SDG method, defining quantities that will be used in later chapters. The main features of the asynchronous meshing strategy is presented here along with the governing equations of linearized elastodynamics which are formulated for the SDG method. In chapter 3, we detail the shared-memory parallel implementation of the SDG method. The operations associated with maintaining the front mesh is decoupled from the solution procedure. In this way are able to optimally distribute computational resources between these tasks. In chapter 4 we examine the performance of the parallel method using the metric of strong scaling efficiency. Finally, in chapter 5 we present some concluding remarks as well as outline areas for future development of the method such as for the case of adaptivity.

CHAPTER 2

THE SPACETIME DISCONTINUOUS GALERKIN FINITE ELEMENT METHOD

2.1 Introduction

In this chapter we provide background on the Spacetime Discontinuous Galerkin Finite Element Method as it applies to the solution of linear hyperbolic partial differential equations. Section 2.2 discusses the spacetime meshing scheme used by the method and introduces the concept of *causality*, a key requirement for the asynchronous parallel solver. Section 2.3 covers the one-field, linearized elastodynamics equations solved by the SDG method which is used to verify the capabilities of our parallel implementation (c.f. chapter 3). We provide only a brief overview of the SDG method in the following sections. Interested readers can refer to the works cited in this chapter for a more complete development.

2.2 Spacetime Meshing

The SDG method works on meshes constructed in *spacetime*. A four-dimensional spacetime mesh is required for a problem that involves evolving behavior in a three-dimensional spatial domain (3 spatial dimensions $\times$ time), for example. This contrasts with traditional *semi-discrete* methods where spatial dimensions are discretized independently of time.

The spacetime meshing algorithm, known as *Tent Pitcher* (developed by [1, 2]), produces a fully unstructured, *simplicial* spacetime mesh where the time step in each spacetime element only depends on the properties of the local, underlying space mesh. By weakly enforcing the governing equations over each spacetime element the SDG method eliminates the need for a time-marching procedure. Global time-marching schemes suffer from the following drawbacks - either the maximum possible time step is constrained by the worst quality

element in the space mesh [1], or a large coupled system has to be solved at each time increment. By construction, the Tent Pitcher algorithm avoids both these issues. In fact, by computing solutions within groups of spacetime elements, or *patches*, the computational cost is linear in the number of spacetime elements.

2.2.1 Causality and Progress Constraint

The characterizing feature of hyperbolic partial differential equations (PDEs) is the notion of *causality*. Disturbances in initial data of hyperbolic boundary value problems travel in a "wave-like" nature along *characteristics* of the equation with a bounded wave speed. The characteristics of a hyperbolic spacetime PDE represents the flow of information through space with time.

Points in spacetime are partially ordered[2] by this concept of causality. We say that a point P *depends* on another point Q if and only if changes to physical quantities (such as temperature, displacement, etc.) at point P influences Q. The *domain of influence* of P is the set of points that depend on P and the *domain of dependence* is the set of points that P depends on. If the governing equations are linear and the material properties are homogeneous and isotropic, as in the case considered here, the wave speed ω is a constant and the domains of influence and dependence are circular cones as depicted in Figure 2.1. We say that one spacetime element $\triangle$ depends on another spacetime element $\triangle'$ if any point $P \in \triangle$ depends on any point $Q \in \triangle'$, i.e. if Q is in the domain of influence of P.

We say that a face, or *facet*, F of a spacetime element is *causal* if it separates the cone of influence from the cone of dependence at every point on F (see Figure 2.1). Stated another way, causality requires each facet to be faster (or closer to horizontal, in spacetime) than the maximum wave speed, i.e. $||\nabla F|| \leq 1/\omega$. If a facet is causal, information only flows in one direction across that facet. The solution to the hyperbolic boundary value problem can be computed simultaneously in spacetime elements that obey the causality constraint and

[1] Work by [16] allows larger time steps than dictated by the worst quality element in explicit time-stepping methods by adaptively taking multiple stabilizing steps in each time increment

[2] A partially ordered set is a set P with a binary relation $R \subset P \times P$ satisfying the properties of *reflexivity*, *antisymmetry* and *transitivity* in set theory.

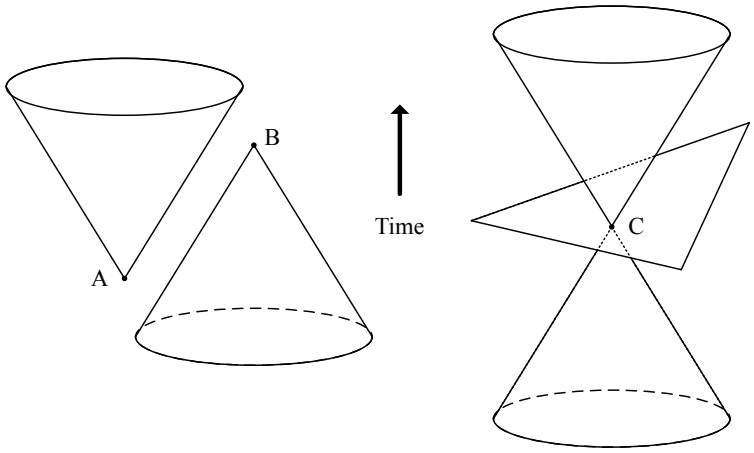

Figure 2.1: (Left) Points A and B are independent since B lies outside the cone of influence of A and A lies outside the cone of dependence of B. (Right) The facet centered on point C separates the cone of influence from the cone of dependence.

do not depend on each other.

Figure 2.2 illustrates the concept of causality for the simple case of 1d $\times$ time. The top part of the figure depicts an unstructured mesh generated by the Tent Pitcher algorithm where the arrows indicate the characteristics of the problem. In this case, characteristics propagate in both directions at a constant speed. The construction of element facets obeys the causality constraint so information flows only in one direction across each facet. We can thus label the earlier facets of an element as the *inflow* facets and the later ones as the *outflow* facets based on the direction of the flow of information. As long as the partial element ordering is observed, solutions to the boundary value problem can be computed locally and independently within causal elements. For example, if we consider all the level-1 elements, the solution within these elements only depends on the initial conditions along their bottom facets and boundary conditions along the left and right domain boundaries. As such, the solutions within the level-1 elements can be computed locally and concurrently. Any level-2 element can be solved as soon as its immediate level-1 neighbor has been solved, even if other level-1 element remains unsolved. It becomes clear from this simple example that a causal spacetime mesh enables asynchronous element-by-element solution with linear complexity.

On the other hand, the bottom part of Figure 2.2 depicts a noncausal mesh corresponding

to a traditional time-marching scheme where each element depends on the other. In this case all elements must be solved together. Such a meshing scheme does not take advantage of the causal property inherent to hyperbolic PDEs. A parallel implementation would not be particularly useful for this situation.

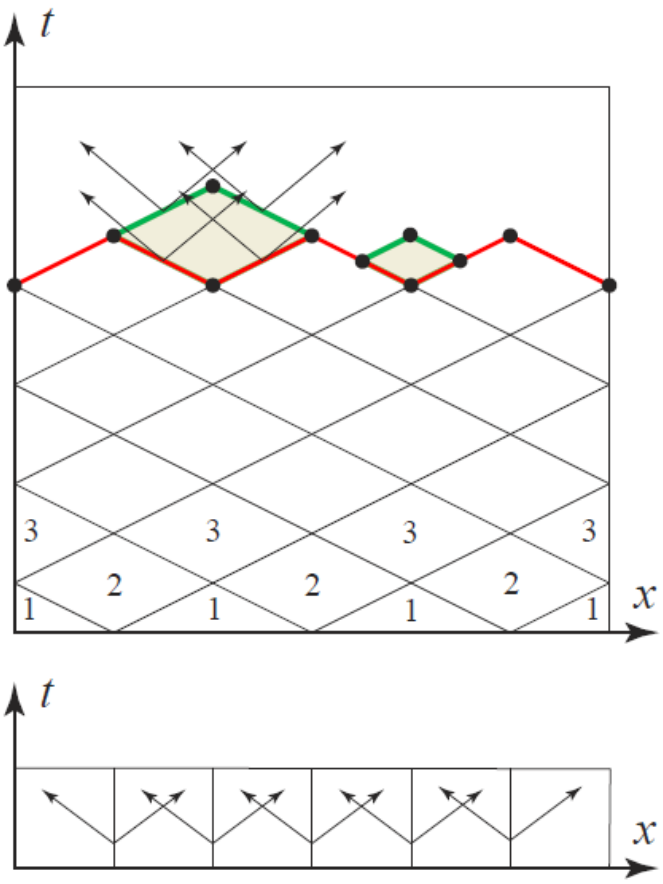

Figure 2.2: 1d × time domain meshed with causal (top) and noncausal (bottom) elements. Reproduced from [17].

To construct an efficient spacetime mesh, the Tent Pitcher algorithm groups elements in *patches* each containing a constant number of spacetime elements. All the elements in a patch must be solved simultaneously as facets of the elements within a patch are not causal. The spacetime mesh can be solved in a patch-by-patch manner that respects the partial ordering of points in the spacetime analysis domain. If we assume polynomial basis function of bounded degree the resulting system of equations in each patch has bounded size and can therefore be solved in constant time. We say that the solution strategy is *efficient* if the

total computation time is proportional to the number of elements in the mesh.

In addition to the causality condition, it was determined that another so called *progress* constraint is needed to construct the spacetime mesh. Without this constraint, there may be cases where the construction of causal elements would make it impossible for subsequently constructed elements to obey the causality constraint, thus making progress impossible. Practically, this places a further limit on the maximum time-step that can be taken when constructing a new patch (see section 2.2.2).

We refer readers to the paper by [18] for a more detailed discussion on the causality and progress constraint.

2.2.2 Advancing Front Spacetime Meshing

The patch-by-patch solution strategy described in section 2.2.1 respects the partial ordering of patches in spacetime. Our goal is to now incrementally construct the spacetime mesh that respects both the causality and progress constraint. This is achieved by the *advancing front* meshing procedure. For any time, the *front* τ is the graph of a continuous time function $\tau : \mathbb{E}^d \to \mathbb{R}$, such that within every triangle, τ is linear and $||\nabla\tau|| \leq 1/\omega$ ([18]), i.e. the front is a *maximal* set of points such that no two points of τ influence each other. The front is a d-dimensional piecewise linear terrain, a subset of the spacetime domain $\mathbb{E}^d \times \mathbb{R}$. Each point P on the front τ can be written as $P = (p, \tau(p))$ where p is the spatial projection of P and $\tau(p)$ is the temporal projection.

Given an initial triangulated front $\tau : \mathbb{E}^d \to \mathbb{R}$, Tent Pitcher selects an arbitrary local minimum vertex $P = (p, \tau(p))$ and moves it forward in time to a new point $P' = (p, \tau'(p))$, thus also advancing the local neighborhood. The volume between the new and old front is called a *tent* and the edge corresponding to the time increment, i.e. PP', is called the *tentpole*. The advancing front method is depicted for the case of 2d $\times$ time in Figure 2.3. The tent is decomposed into spacetime elements (or patch) sharing the noncausal edge PP' and thus, must be solved as a coupled system. However, since the causality constraint is placed on the maximum time step taken when advancing the front, each patch constructed by the advancing front method depends only on elements adjacent to its inflow boundary,

in which the solution has already been computed. Therefore, the solution within the a new patch can be computed as soon as the patch is constructed.

The algorithm advances local minimum vertices since this guarantees a finite amount of progress can be made. A list of all local minima are maintained as a time ordered *heap*. When deciding the next vertex to pitch over, Tent Pitcher chooses the first entry in this heap, corresponding to the global minimum vertex. This heuristic approach to selecting the next pitchable vertex has been found to perform better than others.

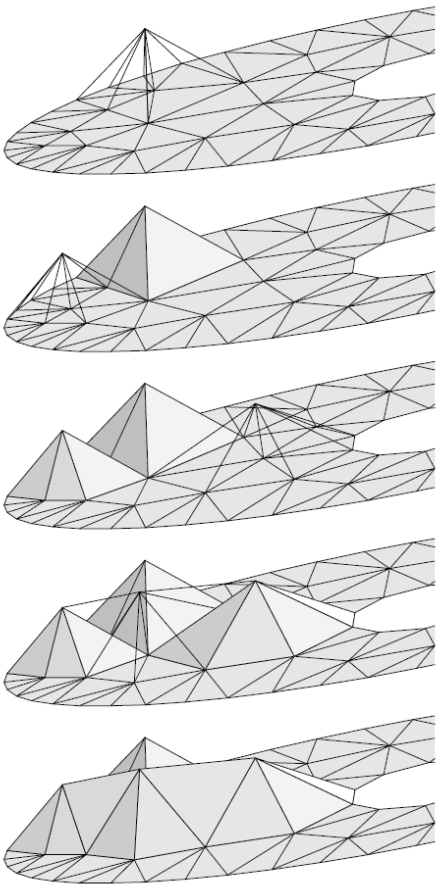

Figure 2.3: Pitching patches (or *tents*) in spacetime using the advancing front algorithm. Newly formed patches are shown in wireframe whereas solid surfaces are patches solved by the SDG solver. Note the partial ordering of patches in accordance to causality. The sequence is advancing from top to bottom while time is increasing upwards. Reproduced from [3]

The advancing front procedure thus allows us to *decouple* the patch generation process from the solution process. This feature has been used in the parallel implementation de-

scribed in Chapter 3 where the meshing and solution tasks have been separated into distinct units of work which can be carried out simultaneously.

Another unique capability of the advancing front algorithm is the inherent capability of localized mesh adaptivity operations. The solution within each patch produces an *a posteriori* estimate of the numerical error incurred during the solution. This error estimate can be used to drive an adaptive meshing algorithm as presented in [19]. Adaptive refinement and coarsening of the individual patches can occur locally and independently of each other where dictated by the error. Adaptive operations can be confined within the patch (or at worst, to a small cluster of patches) and the solution only needs to be recomputed within that region. Additionally, patch generation and adaptive operations can take place simultaneously with the SDG solution procedure in causal patches. This is an incredibly powerful feature of the SDG method that can be exploited by parallel execution. We plan to extend our current parallel implementation (discussed in chapter 3 for the non-adaptive case) to make use of an adaptive meshing procedure in future works.

2.3 Linearized Elastodynamics

In this section, we first discuss some common numerical methods that have been used for linearized elastodynamics and then present the spacetime discontinuous Galerkin formulation that has been implemented in the parallel algorithm.

2.3.1 Comparison of Numerical Methods

Semi-discrete finite elements have traditionally been favored for solving elastodynamics problems. However, in addition to the shortfalls in meshing (discussed in section 2.2) these methods suffer from additional issues when solving elastodynamics problems. These include failure to preserve the invariants of the mechanical system and the inability to accurately capture strong discontinuities, such as shocks, without the use of additional stabilization techniques. In fact, the implicit Newmark family of methods, one of the most widely used semi-discrete algorithm in structural dynamics, are not designed to conserve energy and also

fail to conserve momentum [20]. However, recent work by [21, 22] make use of *variational time integrators* that require multiple nonlinear iterations to simultaneously achieve energy and momentum conservation in time-stepping methods.

The time-discontinuous Galerkin methods, first introduced by [23], discretize space and time simultaneously with space-time finite elements that are continuous in space but have discontinuities in time. These discontinuities correspond to the boundaries of constant-time slabs where jump conditions are used to enforce the appropriate level of continuity between adjacent slabs. Works by [24, 25] find that the time-discontinuous Galerkin methods have better convergence rates and stability properties than the traditional time-stepping algorithms. The higher computational cost associated with the discontinuous Galerkin methods is compensated by the ability to use a larger time step in each iteration.

Spacetime discontinuous Galerkin methods were first applied for elastodynamics by [26]. Similar to time-discontinuous Galerkin methods the SDG method discretizes both space and time with space-time elements however discontinuities are allowed between all element boundaries. This results in the unstructured meshes detailed in earlier sections. The appeal of this method for the case of elastodynamics is the element level conservation property as achieved in [27]. Additionally, in shock-capturing problems, no extra stabilization is needed to suppress spurious oscillations. When implemented on causal spacetime meshes, we have observed that the computational cost is linear in the number of elements and the underlying method has a rich parallel structure.

2.3.2 Spacetime Discontinuous Galerkin Formulation of Linearized Elastodynamics

The method presented here is adapted from work by [27] and later generalized in [28] for the three-field case. This has been implemented in the SDG solver used by our parallel algorithm. We use the notation of differential forms and exterior calculus on manifolds to develop the spacetime continuum theory for linearized elastodynamics. This notation, though unconventional, is very well suited for spacetime mechanics formulations. For example, it provides a coordinate-free way to express fluxes across interfaces with arbitrary orientations in space-

time and circumvents problems relating to frame invariance that arise with traditional tensor notation. We then apply a Bubnov-Galerkin weighted residual procedure to formulate the finite element method. In this section, we present only a brief, qualitative description of the single-field formulation as presented in [27] where the displacement field $\mathbf{u}$ is the sole primary unknown field.

Consider a flat spacetime manifold $\mathcal{D} \subset \mathcal{M} := \mathbb{E}^d \times \mathbb{R}$. The balance of linear momentum is written in forms notation as

$$\int_{\partial\mathcal{Q}} \boldsymbol{M} - \int_{\mathcal{Q}} \rho\boldsymbol{b} = \mathbf{0} \;\forall\; \mathcal{Q} \subset \mathcal{D} \;, \tag{2.1}$$

where $\mathcal{Q}$ is any spacetime domain with a suitably regular boundary. $\boldsymbol{M}$ is the spacetime d-form that delivers the flux of linear momentum across any spacetime d-manifold, ρ is the mass density, and $\boldsymbol{b}$ is the $(d+1)$-form for body force per unit mass. The spacetime flux of linear momentum is obtained as

$$\boldsymbol{M} := \boldsymbol{p} - \boldsymbol{\sigma} \;, \tag{2.2}$$

where $\boldsymbol{p}$ and $\boldsymbol{\sigma}$ are the d-forms for linear momentum density and stress respectively. These d-forms have vector and tensor coefficients: $\boldsymbol{p} = \mathbf{p}\star\mathrm{d}t$, and $\boldsymbol{\sigma} = \boldsymbol{\sigma} \wedge \star\mathbf{dx}$ respectively. Note that we have used bold italicized typeface to denote forms and bold upright typeface for their vector and tensor coefficients. The d-form $\star\mathbf{dx} := \mathbf{e}_i\star\mathrm{d}x^i$ where the *Hodge star* operator is used.

Since we use the single-field formulation for linear elastodynamics, the relations between $\mathbf{u}$ and the one-forms for velocity and strain, $\boldsymbol{v}$ and $\boldsymbol{E}$, are strictly enforced on element interiors. The one-forms combine to give the spacetime *strain-velocity* one-form

$$\mathbf{d}\boldsymbol{\varepsilon} := \boldsymbol{v} + \boldsymbol{E} \tag{2.3}$$

where $\boldsymbol{d}$ is the exterior derivative

The *strain-velocity* relation is expressed as

$$\widetilde{\nabla}\mathbf{v} - \dot{\mathbf{E}} = \mathbf{0} \tag{2.4}$$

in which $\widetilde{\nabla}$ is the symmetric part of the spatial gradient operator. Equation (2.4) is valid in regions where the strain-velocity $\boldsymbol{\varepsilon}$ is continuous, but it is incomplete in regions where $\boldsymbol{\varepsilon}$ suffers jumps. We need the *kinematic compatibility* relations to couple the strain displacement and strain fields across jumps.

Let $\mathcal{S}$ be the space of symmetric, second-order tensor fields with components in the space of test functions on $\mathcal{D}$. The first compatibility relation requires that for all $\mathbf{T} \in \mathcal{S}$ and for all open regions $\mathcal{Q} \subset \mathcal{D}$ with suitably regular boundaries,

$$(\mathbf{d}\boldsymbol{\varepsilon} \wedge \boldsymbol{T})|_{\mathcal{Q}\setminus \Gamma^{\mathrm{J}}_{\boldsymbol{\varepsilon}}} = \mathbf{0} \tag{2.5a}$$

$$[(\boldsymbol{\varepsilon}^* - \boldsymbol{\varepsilon}) \wedge \boldsymbol{T}]|_{\partial\mathcal{Q}\cup(\mathcal{Q}\cap\Gamma^{\mathrm{J}}_{\boldsymbol{\varepsilon}})} = \mathbf{0}\ , \tag{2.5b}$$

where $\boldsymbol{T} := \mathbf{T}\wedge\mathbf{i}\star\mathbf{dx}$ and $\Gamma^{\mathrm{J}}_{\boldsymbol{\varepsilon}}$ is the jump set of $\boldsymbol{\varepsilon}$. Equations (2.5a) and (2.5b) are components of the same equation, where $\mathbf{d}\boldsymbol{\varepsilon}|_{\mathcal{D}\setminus\Gamma^{\mathrm{J}}_{\boldsymbol{\varepsilon}}}$ and $(\boldsymbol{\varepsilon}^* - \boldsymbol{\varepsilon})|_{\partial\mathcal{Q}\cup(\mathcal{Q}\cap\Gamma^{\mathrm{J}}_{\boldsymbol{\varepsilon}})}$ represent the *diffuse* and *jump* parts of the exterior derivative of $\boldsymbol{\varepsilon}$. The quantity $\boldsymbol{\varepsilon}^*$ is the *target* value of the strain-velocity on $\partial\mathcal{Q}$, which is discussed later.

The second kinematic compatibility condition is the *displacement-velocity relation*. This says for all $\mathcal{Q} \subset \mathcal{D}$

$$[(\mathbf{du} - \boldsymbol{v}) \wedge \star\mathrm{d}t]|_{\mathcal{Q}\setminus\Gamma^{\mathrm{J}}_{\mathbf{u}}} = \mathbf{0} \tag{2.6a}$$

$$[(\mathbf{u}^* - \mathbf{u})\star\mathrm{d}t]|_{\partial\mathcal{Q}\cup(\mathcal{Q}\cap\Gamma^{\mathrm{J}}_{\mathbf{u}})} = \mathbf{0} \tag{2.6b}$$

where $\Gamma^{\mathrm{J}}_{\mathbf{u}}$ is the jump set of $\mathbf{u}$, $\mathbf{u}|_{\partial\mathcal{Q}}$ is the interior trace of $\mathbf{u}$ on $\partial\mathcal{Q}$ and $\mathbf{u}^*$ is a target value of the displacement on $\Gamma^{\mathrm{J}}_{\mathbf{u}}$. Equations (2.6a) and (2.6b) are again the diffuse and jump parts of a single equation involving the complete exterior derivative of $\mathbf{u}$.

We localize the momentum balance equation (2.1) and apply the Stokes theorem for differential forms, $\int_{\partial\mathcal{Q}} \boldsymbol{M} = \int_{\mathcal{Q}} \mathbf{dM}$. This yields $\forall\ \mathcal{Q} \subset \mathcal{D}$,

$$(\mathbf{d}\boldsymbol{M} - \rho\boldsymbol{b})|_{\mathcal{Q}\setminus\Gamma^{\mathrm{J}}_{\boldsymbol{\varepsilon}}} = \mathbf{0} \tag{2.7a}$$

$$(\boldsymbol{M}^* - \boldsymbol{M})|_{\partial\mathcal{Q}\cup(\mathcal{Q}\cap\Gamma^{\mathrm{J}}_{\boldsymbol{\varepsilon}})} = \mathbf{0}\ . \tag{2.7b}$$

where (2.7a) is the diffuse part and (2.7b) is the jump part of the *equation of motion*. $\Gamma_{\varepsilon}^{\mathrm{J}}$ is the jump set of $\boldsymbol{M}$ on $\mathcal{Q}$. In (2.7b), $\boldsymbol{M}|_{\partial\mathcal{Q}\cup(\mathcal{Q}\cap\Gamma_{\varepsilon}^{\mathrm{J}})}$ denotes the interior trace of $\boldsymbol{M}$ on $\partial\mathcal{Q}\cup(\mathcal{Q}\cap\Gamma_{\varepsilon}^{\mathrm{J}})$, while $\boldsymbol{M}^*$ is the target spacetime momentum flux.

The equations (2.5), (2.6) and (2.7) are strictly enforced on element interiors. On the element boundaries, the solution field may suffer jumps. Thus, the exterior derivatives in these equations must be interpreted in the sense of distribution theory. All three exterior derivatives generate jump conditions across element boundaries (c.f. equations (2.5b), (2.6b) and (2.7b)). These jump conditions are weakly enforced along with the quantities on the interior of the elements in the SDG formulation. Rather than write a single jump equation across element interfaces, we use separate jump conditions from each side of the boundary that equate various interior trace quantities to the *interface target values* which are denoted by the * superscript. The target values are determined depending the spacetime orientation of the facet and depending on the problem being solved. These can be selected to be Riemann solution values or functions of prescribed boundary/initial conditions. For the case of 1-field and 3-field elastodynamics, the expressions for these target values are given in [27] and [28] respectively. The use of target values for the jump terms greatly improves accuracy and stability of the method.

The per-element weighted residual statement of the SDG method can be written as follows. We denote weighting functions and associated derived quantities with a $\hat{}$ decoration. The unknown displacement field $\mathbf{u}$ and the corresponding weighting function $\hat{\mathbf{u}}$ lie in the polynomial space $\mathcal{U}$ constructed on the interior of spacetime element $\mathcal{Q}$ of order $k_{\mathcal{Q}}$.

Problem 1 (One-field Weighted Residuals Statement) *Find* $\mathbf{u}\in\mathcal{U}$ *such that for every element* $\mathcal{Q}$

$$\begin{aligned}\int_{\mathcal{Q}}\mathbf{i}\hat{\boldsymbol{\varepsilon}}\wedge(\mathbf{d}\boldsymbol{M}-\rho\boldsymbol{b})\\ +\int_{\partial\mathcal{Q}}\left[\mathbf{i}\hat{\boldsymbol{\varepsilon}}\wedge(\boldsymbol{M}^*-\boldsymbol{M})+(\boldsymbol{\varepsilon}^*-\boldsymbol{\varepsilon})\wedge\mathbf{i}\hat{\boldsymbol{M}}+(\mathbf{u}^*-\mathbf{u})\wedge\hat{\boldsymbol{f}}_{\mathrm{I}}\right]=0\\ \forall\ \hat{\mathbf{u}}\in\mathcal{U}^{\mathcal{Q}}\ ,\end{aligned}\tag{2.8}$$

where $\boldsymbol{v}:=\dot{\mathbf{u}}\mathrm{d}t$, $\mathbf{i}\hat{\boldsymbol{\varepsilon}}:=\dot{\hat{\mathbf{u}}}$, $\boldsymbol{\sigma}:=\mathbf{C}\left(\widetilde{\nabla}\mathbf{u}\right)$, $\mathbf{i}\hat{\boldsymbol{M}}:=\mathbf{C}\left(\widetilde{\nabla}\hat{\mathbf{u}}\right)\wedge\mathbf{i}\star\mathbf{dx}$ and $\mathbf{C}$ is the positve

fourth-order elasticity tensor field.

The corresponding weak form is obtained by applying the Stoke's theorem to the weighted residual statement (2.8).

Problem 2 (One-field Weak Statement) *Find* $\mathbf{u} \in \mathcal{U}$ *such that for every element* $\mathcal{Q}$

$$\begin{aligned} &- \int_{\mathcal{Q}} (\mathbf{d}\mathbf{i}\hat{\boldsymbol{\varepsilon}} \wedge \boldsymbol{M} + \mathbf{i}\hat{\boldsymbol{\varepsilon}} \wedge \rho \boldsymbol{b}) \\ &+ \int_{\partial \mathcal{Q}} \left[\mathbf{i}\hat{\boldsymbol{\varepsilon}} \wedge \boldsymbol{M}^* + (\boldsymbol{\varepsilon}^* - \boldsymbol{\varepsilon}) \wedge \mathbf{i}\hat{\boldsymbol{M}} + (\mathbf{u}^* - \mathbf{u}) \wedge \hat{\boldsymbol{f}}_{\mathrm{I}} \right] = 0 \\ &\qquad \forall\ \hat{\mathbf{u}} \in \mathcal{U}^{\mathcal{Q}} \ . \end{aligned} \tag{2.9}$$

The solution to Problem 2 is solved within each element in the patch constructed using the advancing front method (c.f. section 2.2.2). The data from the inflow facets are used to construct the target values for the jump terms across the element boundaries. This formulation has optimal convergence rate of $\mathcal{O}(h^{p+1})$ for interpolation fields of polynomial order p ([28]).

CHAPTER 3

NONADAPTIVE SHARED-MEMORY PARALLEL IMPLEMENTATION

3.1 Introduction

This chapter discusses the shared memory parallelization of the SDG code. In the shared memory model, computations are performed on processor cores that share a common physical memory space. All data is immediately accessible from every core without explicit communication between them. This allows for efficient parallelization of existing serial code without the need for message passing or load balancing across distributed nodes.

An ideal program for parallelization is one that can be divided into separate subproblems of equal size that require little to no communication between the subproblems. Problems of this type are commonly called *embarrassingly parallel*. The SDG method satisfies this criteria implicitly since the solution of a new patch depends only on previously computed data. Additionally the localized patch-by-patch solution technique limits the number of degrees of freedom per solve, ensuring much smaller matrix-vector operations when compared to traditional finite element methods which need to solve large, global matrix equations. For a front mesh of constant degree, each patch solve operation takes constant time. These features make the SDG method uniquely well-suited for parallel implementation.

Within the shared memory model, *threads* can be used to implement parallelism. A thread is defined as "an independent stream of instructions that can be scheduled to run as such by the operating system". We utilize the POSIX Threads, or *Pthreads*, execution model to implement the parallelism in the SDG Method. The Pthreads framework allows a program to launch and maintain a number of threads that can run simultaneously and independently. Care has to be taken to ensure that the instructions carried out by each thread do not access memory locations used by other threads. One method to overcome this is to use locks to

guard critical sections of the code. This however introduces serializations within parallel sections of the code. Based on the patch-by-patch solution technique of the SDG method, we have implemented an element level locking scheme that enforces lock-free operation.

In this chapter, we first discuss the main structure of our shared-memory implementation. We then present the locking scheme employed to guarantee the threads run without race-conditions and low overhead. Finally, we present a few optimizations that improved performance on the machine architecture considered.

3.2 Thread Pool Model

The parallel SDG method is implemented using a *thread pool* software design pattern wherein a predetermined number of threads are spawned at the start of the simulation and can be used to solve different tasks concurrently. The number of threads spawned is based on the computational resources available at runtime. The tasks to be performed by the threads, such as a patch solve, are stored in a *task queue* which is maintained by a *master thread*. The threads request tasks from the queue, perform the task, and then request more work. A simple algorithmic description for this model is as follows:

Main Thread

1. Spawn thread pool of worker threads and initialize task queue
2. Wait for threads to exit
3. Join threads, terminate program

Worker Threads

1. Read task from task queue
2. If work is finished:
    2.1. Exit to main thread

3. Else:

    3.1. Remove task from task queue

    3.2. Perform task

    3.3. Add new task to task queue, if any

The advantage of using the thread pool model rather than spawning one thread to run one task is to prevent the overhead present in thread creation. Additionally, as long as the task queue is not empty, worker threads can perform tasks simultaneously. Care must be taken to ensure that the task performed by each thread has no data dependencies with other tasks. The drawback with this model is that the task queue becomes a shared resource and access to it must be protected. This can cause contention when the number of worker threads is large.

We point readers to the book by [29] for a more detailed description of parallel execution models.

3.3 SDG Thread Pool Model

In this section, we discuss how the thread pool model has been adapted to the SDG method. As noted before, the patch-by-patch solution technique is embarrassingly parallel, a unique feature of the SDG method. This is crucial to ensure that the worker threads can perform concurrently without the need for synchronization.

The main tasks of the SDG Method are performed by two pools of threads: the *Physics* and *Geometry* thread pool. Each thread pool has a separate task queue from which work is performed, the *Physics task queue* and the *Geometry task queue*. The Geometry thread pool advances the front by selecting a local minimum vertex and constructs a space-time patch over that vertex whereas the Physics thread pool invokes the SDG finite-element method to compute the solution within that patch. By utilizing the two thread pool structure, the construction of patches happens independently of the solution procedure. Thus, the compute-intensive tasks performed by the Physics thread pool can be assigned a larger

portion of the available system resources. Figure 3.1 depicts the simplified structure of the thread pool model utilized in the SDG code. Note that each Physics thread has a private task queue to reduce contention for cases where the number of Physics threads are large.

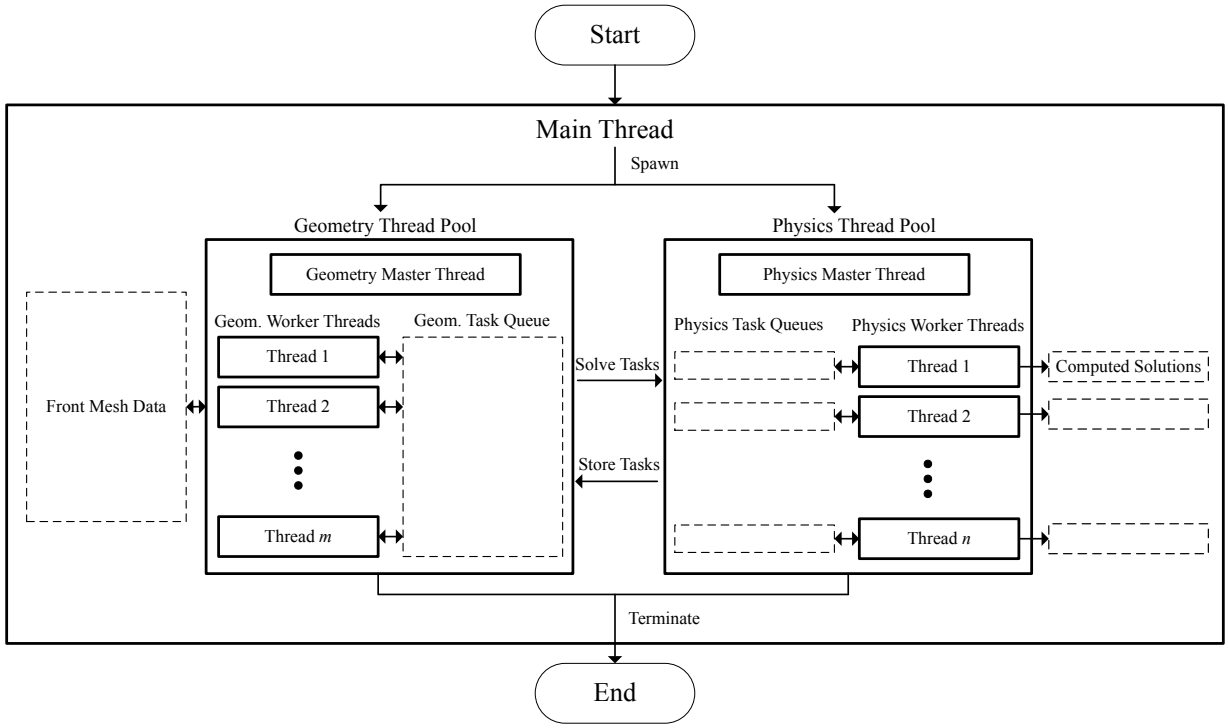

Figure 3.1: Thread pool model for SDG code. Geometry and Physics thread pools perform tasks from their respective task queues and can generate additional tasks.

The *Main thread* is responsible for reading the supplied input files and setting up the initial front mesh. The two thread pools are launched by the Main thread and the progress of the simulation is monitored here. After completion, the worker threads are joined and terminated from the Main thread. The control flow of the main thread is depicted in Figure 3.2.

3.3.1 Physics Thread Pool

The Physics thread pool is responsible for the patch-by-patch solution procedure over patches constructed by the Geometry thread pool. The flowchart detailing the operations of the Physics thread pool is given in Figure 3.3. For each patch of elements, the Physics thread pool computes a solution for Problem 2, c.f. section 2.3.2. The embarrassingly parallel

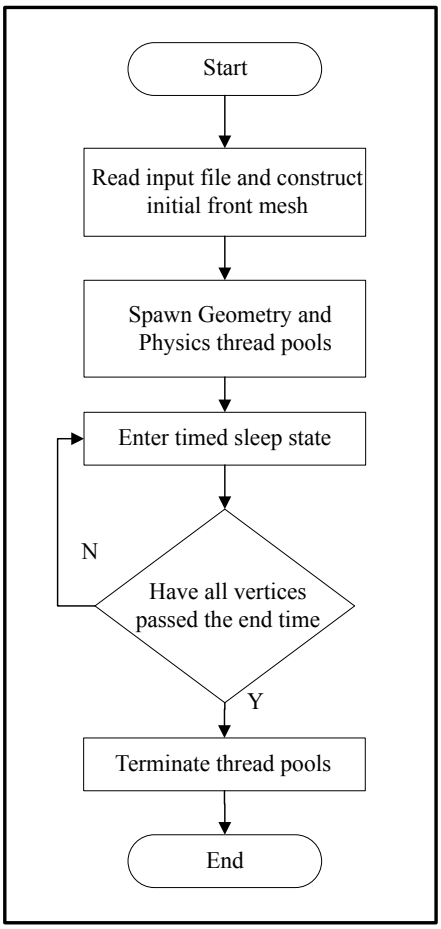

Figure 3.2: Flowchart of operations performed by the Main thread

nature of the SDG method ensures the Physics pool can complete tasks as long as there are tasks available as it only depends on boundary data and solution data from previously computed patches. As such, the Physics pool is constructed to be as large as possible, subject to the availability of tasks supplied by the Geometry pool. This unique feature of the SDG method allows for the high degree of parallel scalability observed in our code, presented in section 4.2.

In general, the Physics thread would compute an *a posteriori* estimate of the numerical error incurred during the solution of a patch. If the error in any element is above some threshold, the patch is rejected and the solution must be recomputed after refinement has occurred. However, for the non-adaptive case considered here, the error is not relevant and the computed solution is passed to the Geometry pool which updates the front mesh.

One of the limitations of the thread pool model is the contention on the task queue, espe-

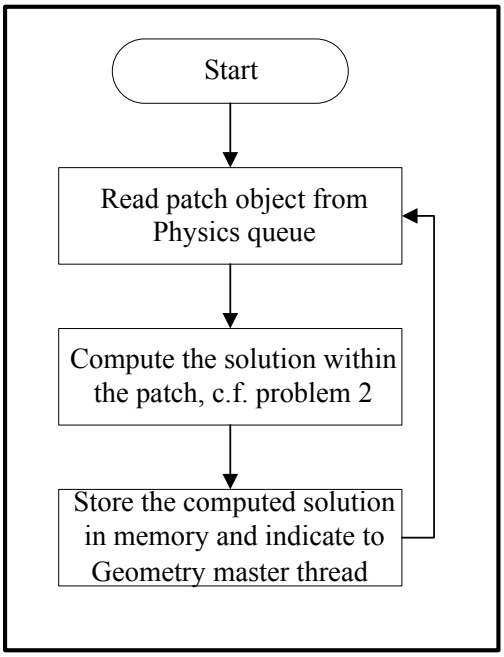

Figure 3.3: Flowchart of operations performed by the Physics threads

cially as the number of worker threads becomes very large. To circumvent this, each Physics thread has a separate task queue. Each queue has a maximum size that is heuristically determined to minimize the possibility of the queue emptying.

3.3.2 Geometry Thread Pool

The Geometry thread pool advances the front mesh by constructing space-time patches over local minima vertices, as outlined in section 2.2. These patches are then solved by the Physics pool. After a solution has been computed, the Geometry pool updates the front mesh and stores the solution for post-processing.

The main goal of the Geometry pool is to supply the Physics pool with enough work. This is achieved through a *dynamic scheduling system* performed by the Geometry master thread as depicted in Figure 3.4. A simple round-robin technique is used to monitor the number of patch solve tasks in each Physics task queue. If any of these queues are not full the Geometry master thread adds a patch generation task to the Geometry queue. A patch generation task involves the standard advancing-front spacetime meshing algorithm where local minima vertices are advanced in time to form spacetime patches. Since there are

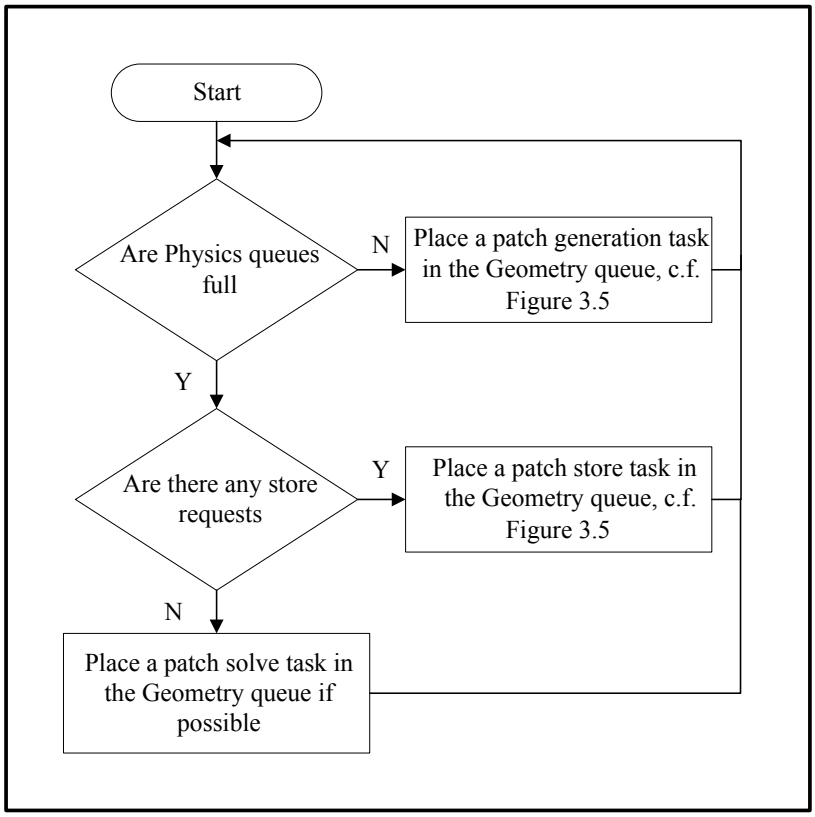

Figure 3.4: Flowchart of operations performed by the Geometry master thread

multiple Geometry threads in the pool, a locking scheme is needed to ensure that the patches generated have no data dependencies with each other. This has been achieved by placing a lock on all elements belonging to the patch footprint, as discussed in section 3.4. The newly constructed patch is then placed in the appropriate Physics queue. On completion of each pass over the Physics queues, the Geometry master thread checks for any patches solved by the Physics pool. The solved patches are streamed to a separate queue for each Physics thread (as depicted in Figure 3.1) If any exist, a patch store task is placed on the Geometry task queue. This instructs a Geometry thread to update the front mesh with the computed solution within the patch and store the solution in long-term storage for post-processing. The local minima time heap is then modified based on the new terrain. The patch generation and store operations performed by each Geometry thread in detailed in Figure 3.5.

This scheduling scheme prioritizes patch store operations subject to the condition that the Physics queues are full.

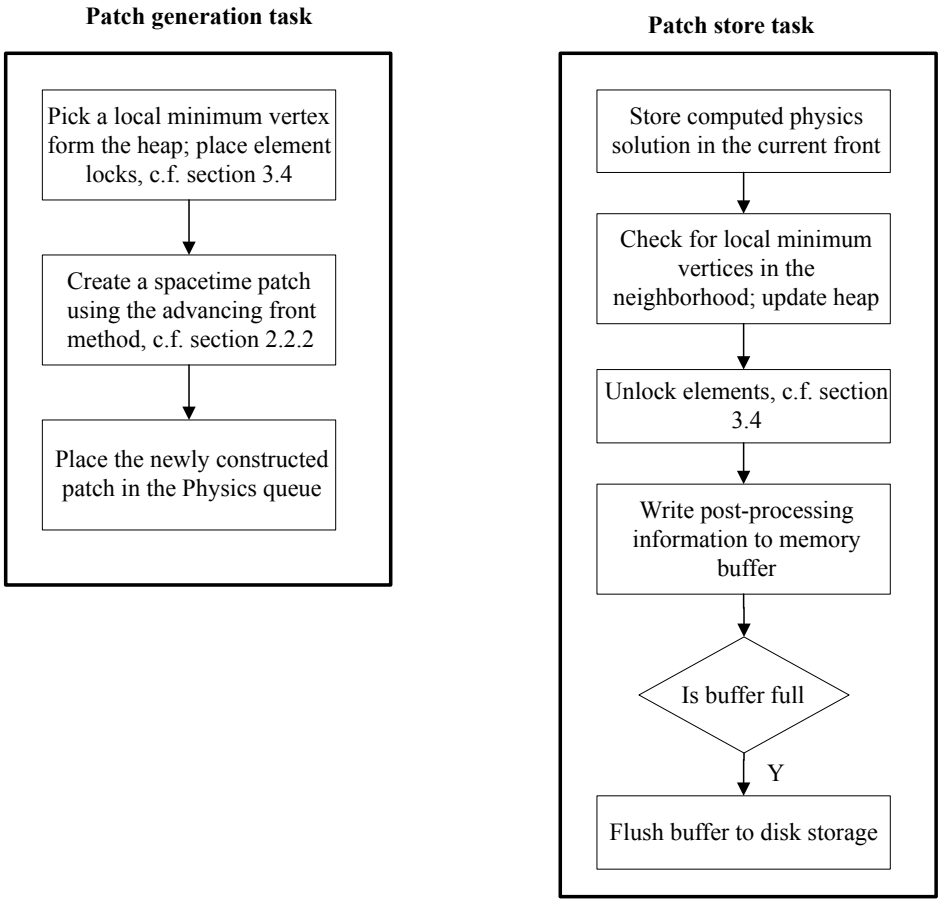

Figure 3.5: Flowchart of tasks performed by the Geometry worker threads: (left) patch generation and (right) patch storage

Due to the imbalance in workload of the Geometry and Physics thread pools, it is very possible for the Geometry pool to idle, wasting computational resources. This is especially the case when running on systems where there is very little scope for parallelization, such as a personal computer. Since we make use of thread affinity to improve cache performance (see section 3.5.1), this leads to a load-balancing problem. In such a situation,the Geometry master thread allows a subset of the Geometry threads to perform patch solve operations,thus balancing the computational workload across both thread pools. The actual number of geometry threads allowed to solve patches depends on the system architecture and should be chosen to let the remaining Geometry threads satisfy the demand of the Physics thread pool. There is a limit on the number of patch solve requests placed in the Geometry queue.

The other extreme is when the number of Physics threads is much larger than Geometry

threads or when the cost of the patch solve is very small (for linear interpolation functions, for example). In this scenario patches are solved at a faster rate than what can be produced by the Geometry pool. This results in idling of Physics threads which reduces parallel scalability. The number of Geometry threads must be increased in this case to satisfy the demand of the Physics threads.

3.4 Thread Synchronization and Locking Schemes

The use of the shared-memory parallelization model allows for extremely efficient communication between threads. The trade-off is the need for synchronization between threads to ensure that shared data is protected from simultaneous modification. Synchronization introduces undesirable overhead in multi-threaded programs where threads are forced to wait, or *block*, to access shared resources. In order to achieve maximum parallel scalability, the parallel SDG code has been designed to eliminate these expensive blocking locks. This has been achieved partly due to the independence of patch solves guaranteed by the SDG method and partly due to careful redesign of the serial code.

Managing the shared front mesh data is one place where contention can occur between threads. By design, the Geometry thread pool manages all operations involving manipulating the front mesh. Multiple Geometry threads read and modify data related to the front mesh simultaneously. The first safeguard needed is during a patch generation operation when a vertex is selected as the base of a new patch (known as a tentpole vertex). The vertices in elements surrounding the tentpole vertex are locked by setting an internal flag, referred to as a *vertex lock*, which signals that these vertices are now part of an active patch footprint. Additionally, vertices in any elements adjacent to the patch footprint are also locked. This feature has been preemptively added to accommodate mesh adaptivity, although that capability is beyond the scope of this thesis. The yellow-shaded elements in Figure 3.6 indicate the extent of the vertex locks placed during the construction of a patch centered around the vertex A. Any element which contains locked vertices can no longer be used as part of a new patch footprint, such as the case for a subsequent pitch over vertex B, due to the conflict in vetrex $C1$ and $C2$. The setting of vertex locks is implemented as

an atomic operation [1] which prevents unpredictable outcomes when two threads try to set the locks of the same vertex simultaneously. The vertex lock guarantees patches generated by the Geometry thread pool use memory locations that are only accessed by one thread at a time and can thus be freely modified by that thread without the need for expensive synchronization. Once the patch has been solved by the Physics thread pool and the front mesh has been updated, the vertex lock can be removed.

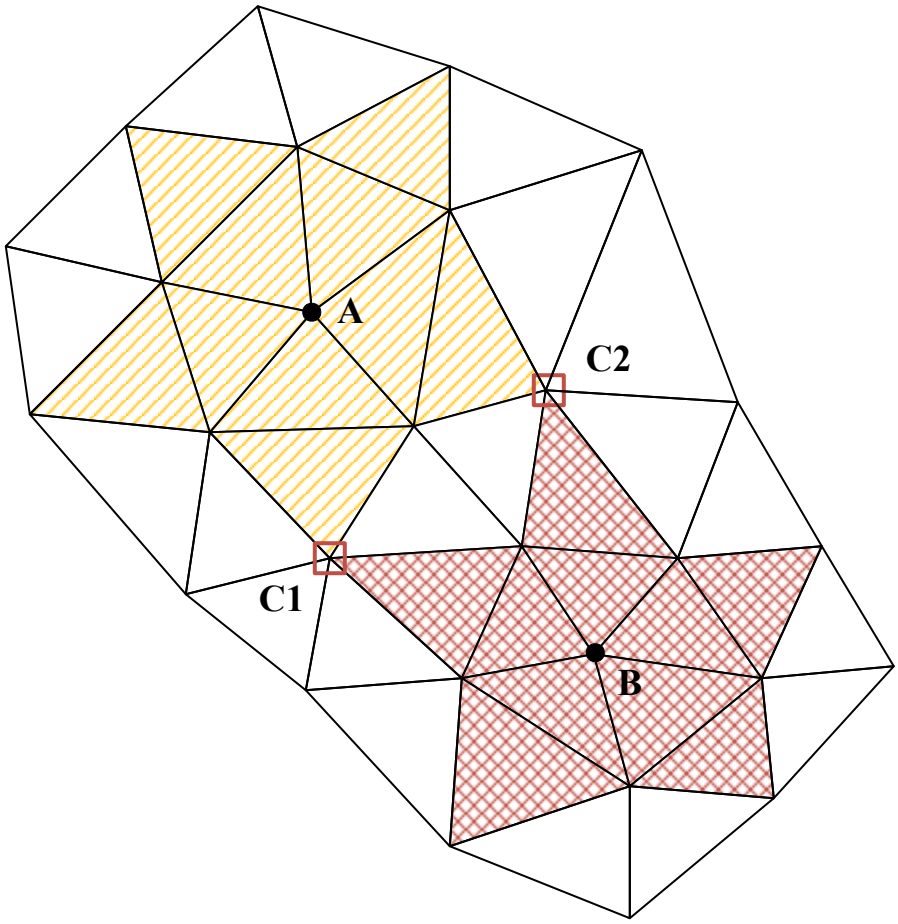

Figure 3.6: Space mesh depicting vertex locking scheme. A pitch over vertex A places locks on the vertices in the yellow-shaded (single-hatched) elements. This prevents a subsequent pitch over vertex B which would require locks on vertices in the red-shaded (cross-hatched) elements due to the conflicts in vertices C1 and C2.

Multi-threaded programs have the additional need for *thread safety*. This is where manipulations of shared data by multiple threads at the same time is possible without data

[1] An atomic operation is one that is indivisible. When one thread performs an atomic operation, all other threads see it as completing instantaneously. Using an atomic operation enforces synchronization without much overhead. However, only simple operations can be made atomic.

races. The standard data structures used by serial programs (such as arrays, queues, etc.) are not thread safe. Modifications to these structures by multiple threads would lead to unpredictable results due to reordering of allocated memory during a write operation, for example. In the adaptive version of the SDG code, thread safety becomes all the more necessary. As such, we have used the *concurrent vector* from the Intel ® Thread Building Blocks (TBB) library as storage for mesh data. The concurrent vector has been designed to be thread safe. Multiple threads can concurrently insert or remove elements without invalidating existing indices. Storage of physics data is done using Blitz++, a high-performance library for scientific computing. By default, Blitz uses expensive mutual exclusion locks, or *mutexes*, to guarantee thread safety. We have used atomic operations provided from the TBB library to replace the mutexes, yielding a more efficient implementation.

Once the patch has been created by the Geometry thread in the above manner, the solution of the patch by the Physics thread pool can proceed lock-free.

3.5 Optimizations

We employ many standard techniques to optimize the performance of the parallel SDG code. We must note that some of these improvements are machine dependent and may need to be carefully tuned to yield optimal results.

3.5.1 Thread Affinity Mapping, Non-uniform Memory Access and Hyper-threading

Thread or *processor affinity mapping* is the explicit binding of execution threads to specific hardware resources. This may be on the granularity of individual cores or even hardware threads. By default, threads are not tied to a particular resource but rather, the available resources are divided in some way by the *scheduler*. This is usually done in a way that ensures fairness among the current tasks such that no task will be denied access to the resources for extended periods of time. The scheduler "swaps" the currently executing task for another at regular intervals. This is done at significant expense as the state of

the currently executed task must be saved and the new task must be loaded. The fast cache memory must be evacuated to make space for the new task and data may need to be fetched from the slower Random Access Memory (RAM) at greater frequency due to thread swapping. This phenomenon is exacerbated when the hardware resources are not sufficient for the number of tasks. Thread affinity mapping is used to override the default scheduler when the performance properties of the program are well known. By doing so, each execution thread is granted exclusive ownership of hardware resources, improving memory performance.

Another feature of modern-day multi-threaded processors is the *non-uniform memory access* (NUMA) architecture where memory is physically distributed but logically shared. Due to the distributed nature, memory access time depends on the location of memory relative to the processor. The thread affinity mapping should take into account the NUMA architecture of the system and store data in such a way that the memory access times are minimized.

For the SDG code, we employ affinity mapping for the main thread as well as the Geometry and Physics thread pools. The initial front mesh is constructed by the main thread and stored in memory closest to the core where the thread is mapped. Thus, we map the Geometry thread pool to the same core to take advantage of the proximity to the memory where the front mesh is stored. The Physics thread pool is mapped to the remaining cores of the system, prioritizing cores that have closest location to the memory that is utilized by the Geometry thread. Based on the architecture of the system, additional cores may be required for the Geometry pool in order to satisfy the demand from the Physics pool.

Since the tasks performed by the Geometry threads involve reading and writing to memory locations as well as to disk, idle periods will occur during execution as data is collected. As a result, we choose to map multiple Geometry threads to the same core to take advantage of the *hyper-threading* or *out-of-order execution* feature available on modern processors. Hyper-threading is where each physical core is represented by two virtual cores that share the workload between them. Since these virtual cores share resources, when one task stalls, the other task can continue without any disruption. Each physics thread on the other hand is mapped to a single core since there is very little idle time during the patch solve operation.

3.5.2 Reducing Contention on Mesh Data Structures

Data stored in the mesh data structures is used extensively during simulations. Since we make use of the shared-memory model, all threads have immediate access to this data. Through the use of vertex locks and thread-safe containers (section 3.4) multiple threads can access mesh data without issues. However, this imposes large contention on the memory bandwidth when the number of threads increases. To reduce the contention on the mesh data structures, we create a copy of the patch footprint at the point of patch creation. This allows the patch to be created without reference to the original memory location. When the patch is given to the physics thread to solve, the data associated with the patch object is private to that thread. This potentially allows the data to be moved closer to the core on which the Physics thread is mapped. Currently, we utilize the serialization operation offered by the Boost C++ Library due to its convenience. Further optimization will replace this library call with a specialized copy command which would reduce overhead.

3.5.3 Parallel Streaming of Output Data

As part of a patch store operation solution data is stored as a binary file on disk storage for post-processing purposes. Output to disk is buffered on RAM until a large enough file can be written at once. For the parallel code, each geometry thread streams output data to a different location in memory and disk in order to remove the need for synchronization. The parallel streams are merged at the end of the solution process.

CHAPTER 4

NUMERICAL RESULTS

This chapter presents numerical results for the nonadaptive shared-memory parallel SDG method. The parallel scalability of the overall code has been verified on a system with 48 physical cores, the specifications for which has been detailed in section 4.1. We achieve very good scalability even for 48 cores. These results have been presented in section 4.2. Additionally, we achieve greater scalability for higher polynomial order of the solution field.

4.1 Experimental Set-up

The implemented method is verified using the one-field elastodynamics equations (c.f. Section 2.3.2) for the case of two spatial dimensions $\times$ time. The degrees of freedom per spacetime element n can be determined as

$$n = 3\frac{(p+d)!}{p!d!}, \tag{4.1}$$

where p is the polynomial order of the function interpolating the solution field and d is the total dimensions of the problem ($d = 3$ for two spatial dimensions). The factor 3 is to account for the three components of a single vector field. Since solutions are usually computed over patches of elements, we have the total number of degrees of freedom per patch as

$$N = Kn, \tag{4.2}$$

where K is the number of elements per patch. Generally, K is a function of the degree of a vertex in the front.

4.1.1 Computing Platform

We have used the Innovative Systems Lab (ISL) at the National Center for Supercomputing Applications (NCSA) to develop and benchmark our code. The testing configuration consists of a single Dell PowerEdge R920 node with four Intel Xeon E7 4860V2 CPUs, each with 12 cores operating at 2.6 GHz, for a total of *48 computing cores*. Each core has 32 KB of L1 and 256 KB of L2 cache memory. Each CPU has a total of 30 MB of L3 cache that is shared across its 12 cores. The system contained a total of 3 TB of RAM. This exceptionally large amount of system RAM prevents paging to disk that may occur when the memory is full. However, since the SDG method performs computations on localized patches rather than on the entire mesh, this is not a problem even on systems with much smaller memory. Each physical core is hyper-threaded to improve parallel execution. By virtue of the NUMA architecture (c.f section 3.5.1) each CPU core has slightly different latency when accessing data stored on memory associated with another CPU.

4.1.2 Thread Affinity Mapping

Given the system topology as described above, we first have to determine the optimal mapping of worker threads to physical cores. We propose three different mapping strategies. The first is where none of the threads are explicitly mapped to the CPU cores and the scheduler decides the order in which threads are executed. This case is used as the baseline in Figure 4.1. For the second strategy the worker threads are mapped in such a way that all cores on a given CPU socket [1] are saturated before threads are assigned to the next socket. The final strategy involves round-robin distribution of threads across the sockets. Figure 4.1 compares the execution times for these mapping strategies. We find that using thread affinity mapping is not always beneficial. If we distribute threads across sockets, as in the case of the third strategy, the performance degrades due to increased latency and reduced cache performance when communicating across sockets. The optimal mapping strategy is the second one which minimizes communication across sockets. This has been chosen as the

[1] A CPU socket is the physical housing of a CPU, connecting it to the other components of the computer. In the test system, each CPU is placed in a separate socket. Additional latency exists when communicating between sockets

default mapping strategy for the results presented later in the chapter.

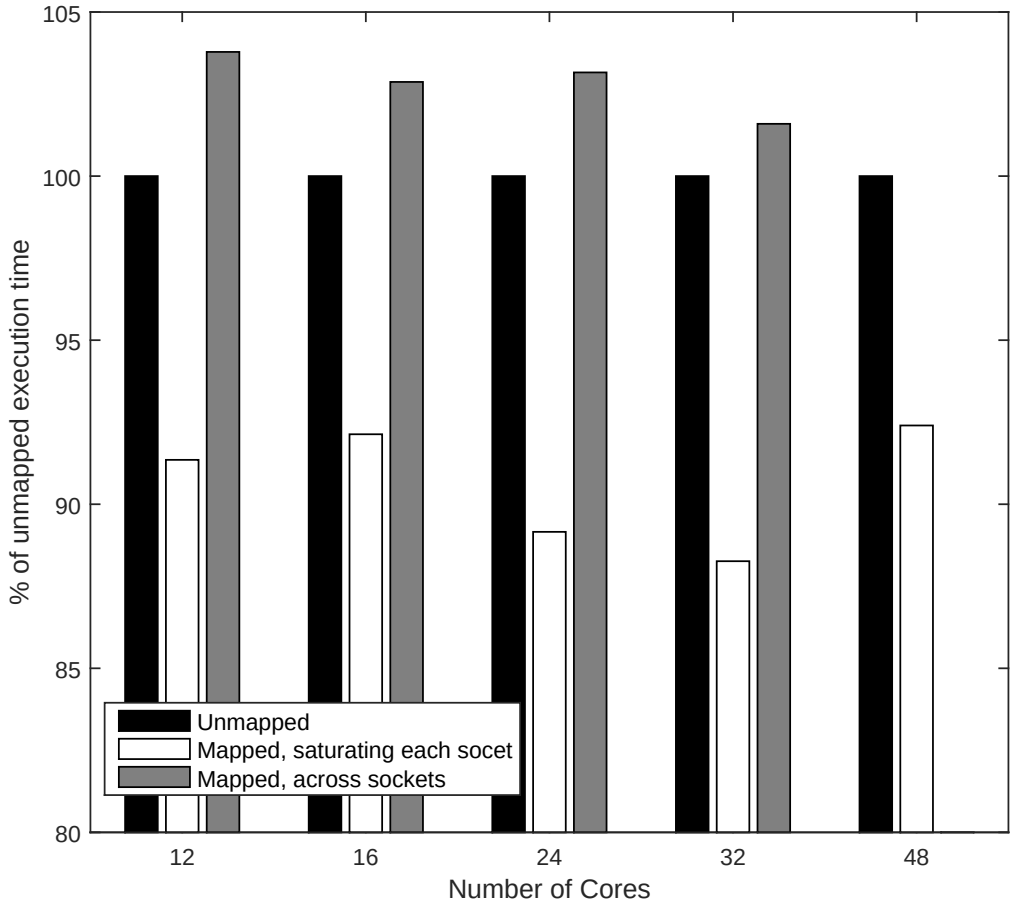

Figure 4.1: Comparison of execution time for different thread affinity mapping schemes relative to the unmapped case.

The other consideration when mapping the worker threads is whether to utilize the hyper-threading capability of the CPU cores. If we map more than one thread to a single core, these threads will, for the majority of the time, execute serially. However when one thread stalls, the other thread can be executed without any disruption. This reduces the idle time in each core, effectively improving parallel performance. We determine that optimal performance is achieved when the Geometry threads are hyper-threaded (2 Geometry threads are mapped to one core) and the Physics threads are not. This is due to the nature of tasks performed by each of these threads, as discussed in chapter 3. Additionally, we require a much smaller

number of Geometry threads to satisfy the Physics threads. For the fully saturated system, it we determine that *two* Geometry threads are sufficient to provide enough worker for the Physics threads. Thus, we always use two Geometry threads mapped onto a single core for the given system topology. We map the Geometry threads onto the same core where the main thread is executed. This ensures the least amount of latency when accessing data associated with the initial mesh.

The final thread affinity mapping for the given system topology is as follow:

Core 1

- Main Thread
- Hyper-threaded Geometry threads

Core 2 - 48

- Single Physics thread, subject to saturating the current socket

4.1.3 Test Problem

The test problem considered is a unit square plate as given in Figure 4.2(a). A uniform pressure P is applied to the top edge of the plate while the bottom edge is fixed. The left and right edge have periodic boundary conditions. The pressure is applied as a step function at time zero. The initial front mesh is produced by triangulating the domain, producing roughly 8000 elements. A coarser mesh of 500 elements is given in Figure 4.2(b). The simulation is run for a fixed number of spacetime patch solves to ensure that the unit of work is constant across all runs. Quadratic ($p = 2$) interpolating functions are used for the solution fields.

4.2 Scaling Studies and Discussion

In this section we investigate the *parallel scalability* of the implemented parallel SDG code. This is an indication of how efficiently the code takes advantage of increasing number of

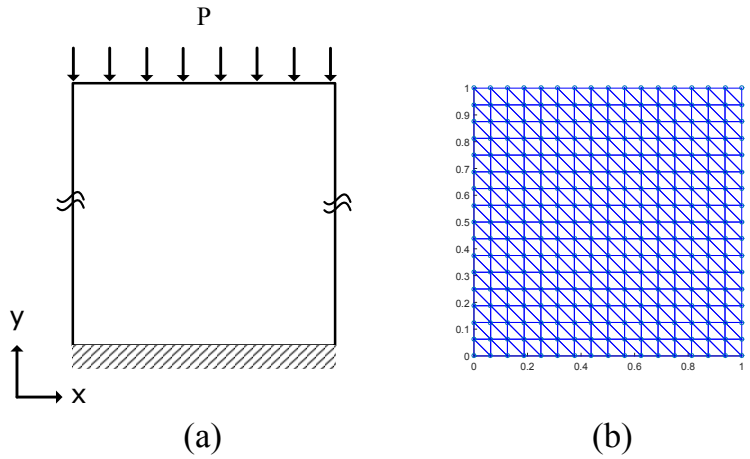

Figure 4.2: Test problem used to verify the parallel scalability of the method. (a) Schematic of unit plate loaded with uniform pressure, (b) coarse triangulated front mesh containing 500 elements

parallel processing elements, or cores in this case. This can be measured by the *strong scaling efficiency* and is defined as

$$\varepsilon_s = \frac{T_1}{N \times T_N} \times 100\%, \tag{4.3}$$

where T_1 is the time needed to complete a task using 1 processing element and T_N is the time needed to complete the same task using N processing elements. For strong scaling, the problem size stays fixed. For a perfectly scaling problem, ε_s is 100 %. In general, it is hard to get good scaling for large number of cores as the communication cost increases in proportion to the number of cores.

Before we can measure the scaling efficiency of our program, the definition of the baseline or serial performance, T_1 must be discussed. Since we use two separate thread pools in our parallel method, T_1 is not straight forward to define. If we consider our thread affinity mapping (c.f. section 4.1.2) the 1 core case would correspond to only one Geometry thread running in serial. The Geometry threads are equipped to solve patches if there is no demand from the Physics thread pool. If we do not launch any Physics threads, the Geometry thread automatically switches between generating, solving and storing patches, in other words, acting like a truly serial algorithm. This is the *serial baseline*. If, on the other hand, we have two Geometry threads on a single core though hyper-threading, only one of these

can solve patches. The other thread would perform purely geometry related tasks. This is the *hyper-threaded baseline*.

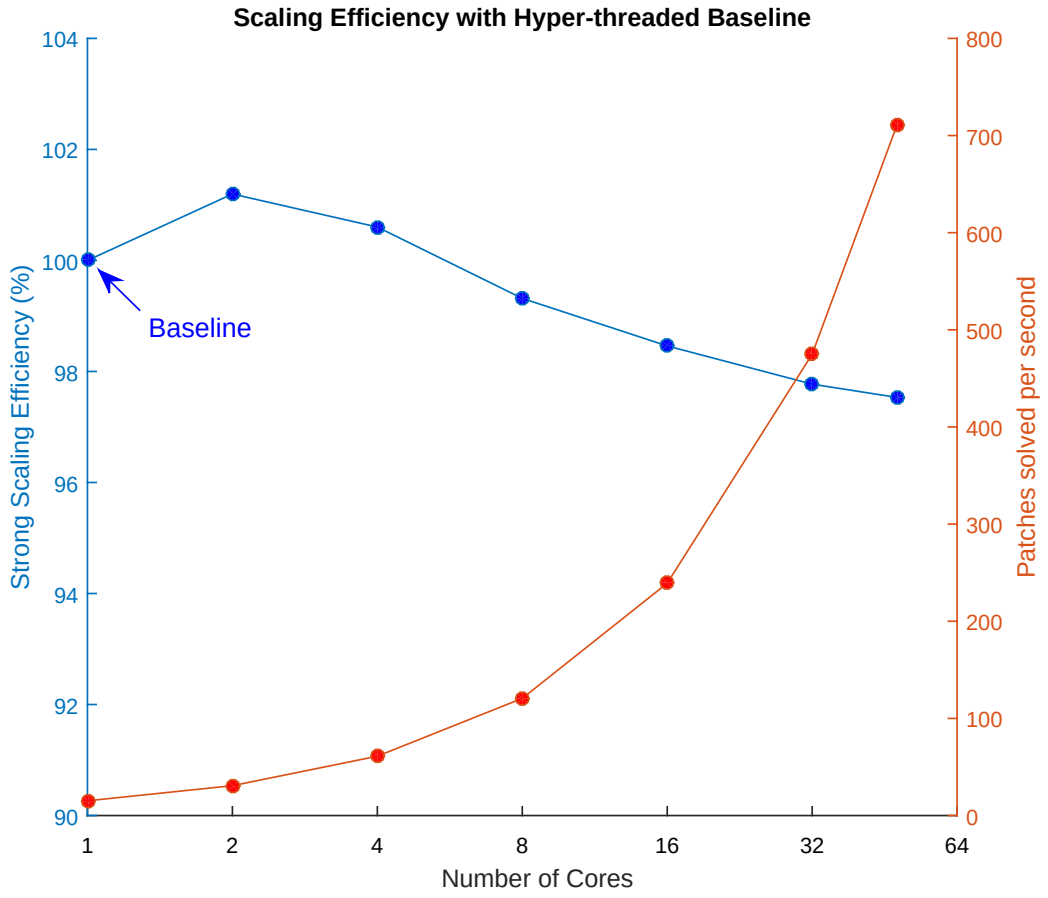

Figure 4.3: Strong scaling efficiency of the parallel SDG code using the hyper-threaded baseline plotted on the left axis. Number of patches solved per second is plotted on the right axis.

We test the strong scaling efficiency of the parallel SDG method using the hyper-threaded baseline as shown in Figure 4.3. It is immediately noticeable that we achieve efficiency greater than 100%, or *super-linear scaling* for the case of 2 and 4 cores. This is due to the choice of baseline as described earlier. The single core case consists of one Geometry thread that can solve patches, albeit less efficiently than a dedicated Physics thread as it shares a core with another Geometry thread through hyper-threading. When we go to the 2 core case, we are more than doubling the performance as we have added a Physics thread on the second core which is dedicated to solving patches. The efficiency drops off

as we add more cores but is still over 98% when we are at the fully saturated system. The drop off can be explained by the increased cost of synchronization and communication as the number of cores increases. The SDG method inherently has minimal communication during execution however synchronization between threads is still required. This involves the various atomic operations that are required to ensure threads execute without errors due to race conditions. Adding more cores also reduces cache performance. By this we mean that the high performance cache memory becomes filled more easily and data must be read in from the slower system RAM. Another potential source of degradation to performance is the inability of the Geometry thread pool to provide enough work to keep the Physics threads busy. This would cause system resources to go idle while the Physics threads wait for patches from the Geometry pool. This is likely to happen when the number of Physics threads greatly outnumbers the Geometry threads. Based on our experience, 2 hyper-threaded Geometry threads are sufficient for 47 Physics threads.

In Figure 4.4 we compare the efficiency obtained for both the serial and hyper-threaded Geometry pool cases. The definition of T_1 in equation 4.3 for this case is that for the serial geometry thread, indicated as "baseline" in the figure. The results form Figure 4.3 (blue curve) is simply shifted higher in Figure 4.4 due to the different choice of baseline when defining the scaling efficiency. As expected, the case where there is only a single thread in the Geometry thread pool (black curve) performs worse than when we have hyper-threading. Here the geometry thread can barely provide enough work to keep the Physics threads busy, especially for over 16 cores. This results in periods where Physics threads are idle.

Increasing the order for the interpolating polynomial fields increases the scaling efficiency as presented in Figure 4.5. This result may seem contradictory as higher order polynomial functions should increase communication costs, thus reducing parallel scalability. However, as we have shown the SDG method exhibits linear complexity in the number of patches solved. Increasing the polynomial order would increase the time per patch solve but would not add any further communication or synchronization costs due to the independence of patch solves. The relatively small size of each patch limits the total number of degrees of freedom during the solution stage, ensuring these operations can take place from the cache memory, even as the polynomial order increases. Finally, the increase in time spent per patch

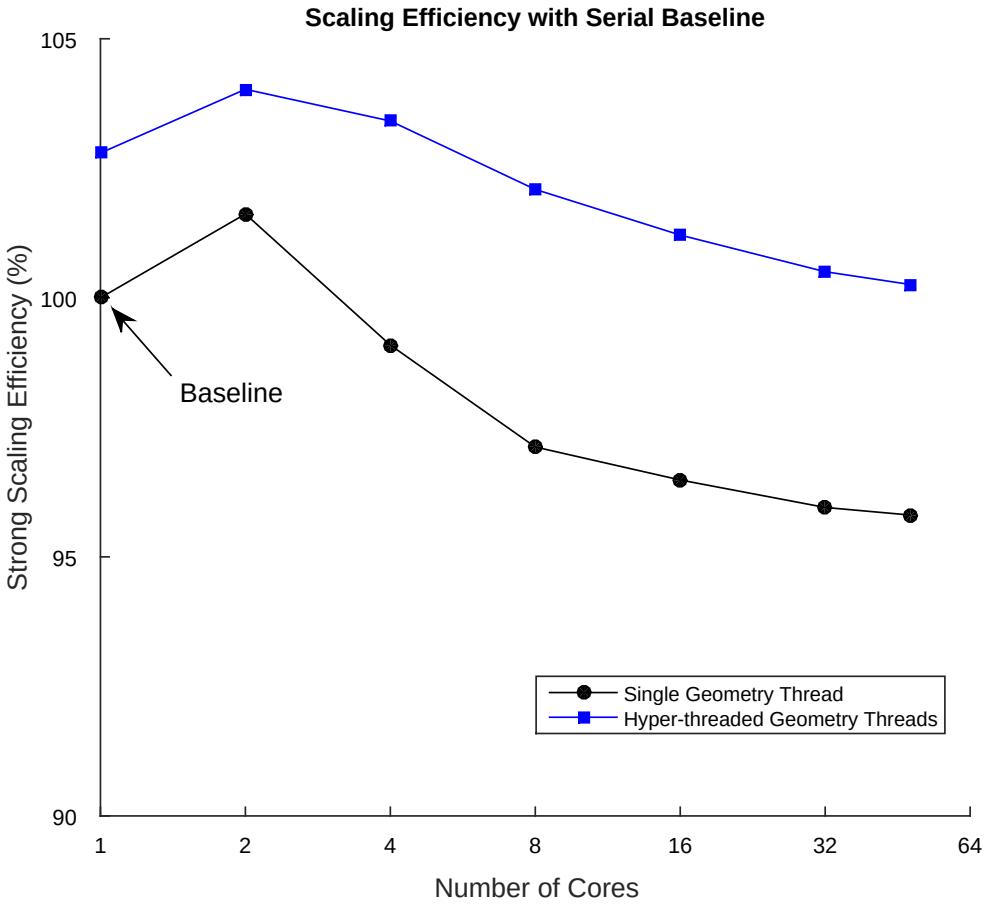

Figure 4.4: Comparison of strong scaling efficiency using two different number of threads for the Geometry thread pool. The definition of the baseline T_1 in equation 4.3 is taken to be the execution time for the 1 threaded Geometry pool case.

solve allows the Geometry thread pool to more easily keep the Physics threads satisfied with work.

These results clearly show the excellent parallel capabilities of the SDG method. The use of two separate thread pools allows us to decouple the patch solve procedure from that of the mesh related operations, enabling us to optimally distribute physical resources between these tasks. We determine the optimal distribution of worker threads that is able to achieve parallel scalability of over 98% for the case of a fully saturated system.

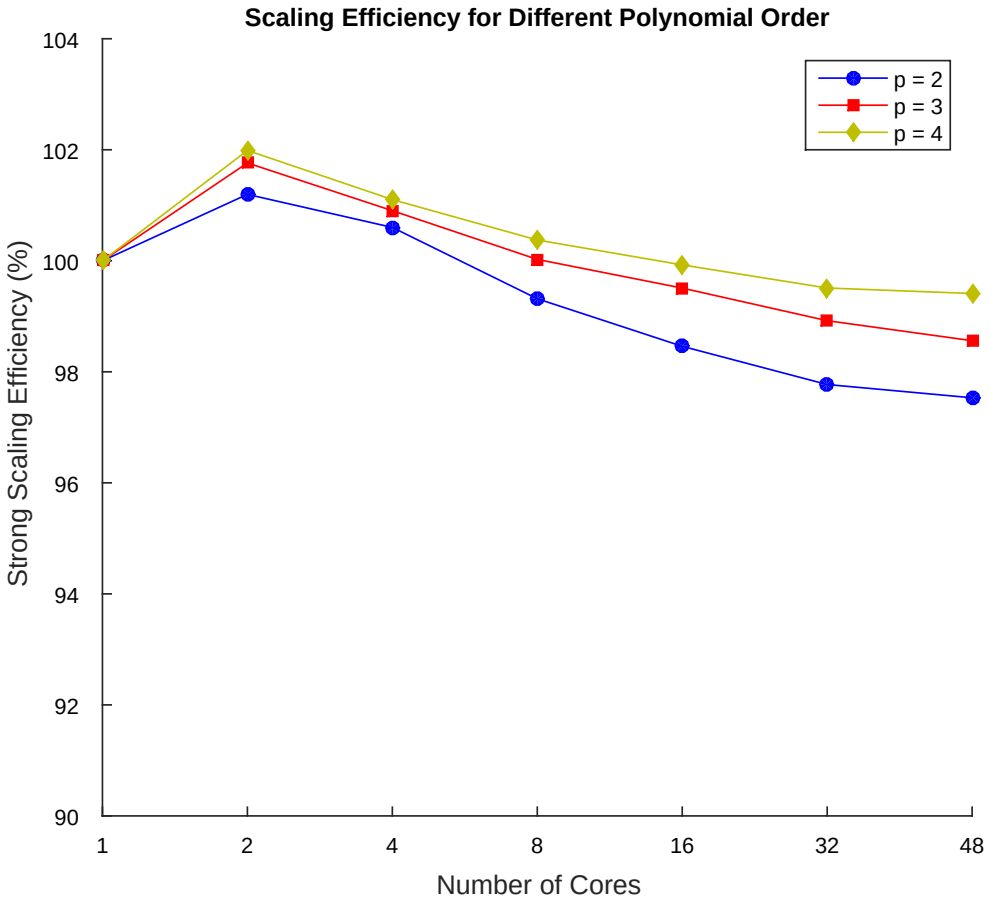

Figure 4.5: Comparison of execution time for different thread affinity mapping schemes relative to the unmapped case.

CHAPTER 5

CONCLUSION AND FUTURE EXTENSION

In this thesis, we have implemented a parallel solver for the SDG finite element method that has scaling efficiency of over 95% for the case of shared-memory parallelism on a cluster of up to 48 processors, even achieving super-linear scaling in some cases. This high degree of parallel scalability was possible in part due to the "embarrassingly parallel" nature of the SDG method and in part due to careful design of the parallel algorithm to eliminate most synchronization overheads and communication costs between parallel threads. We have entirely decoupled mesh-generation from the solution procedure, allowing us to dynamically distribute available computing resources between these operations, ensuring that idle time is minimized. The resulting method is fully scalable for much larger systems than the one considered here.

Future extensions of this work will add the capability of mesh adaptivity to the parallel solver. Adaptive refinement and coarsening occurs on the same granularity as the SDG solution procedure and these operations can take place simultaneously [19]. This is very conducive to a parallel implementation as it avoids the need for a global re-meshing step which interrupts the solution procedure. The adaptive SDG method is very well suited for problems involving tracking shocks or other sharp solution features.

Another possible extension is to the case of distributed-memory parallelism. In this case, memory is physically distributed across computing nodes and explicit communication is required to transfer data across these nodes. Again, the asynchronous nature of the SDG method provides a clear path to an efficient algorithm for this case. The existing shared-memory method already requires newly constructed patches to contain within them all the data necessary to compute the solution within the patch. By distributing pieces of the front mesh across different nodes, such as in the case of a domain decomposition technique,

causal patches generated on each node can be solved independently from other nodes. Communication would only be required for the case where vertices are on the boundary of a domain.

REFERENCES

[1] J. Erickson, D. Guoy, J. Sullivan, and A. Üngör, "Building spacetime meshes over arbitrary spatial domains," *Engineering with Computers*, vol. 20, no. 4, pp. 342–353, 2005.

[2] A. Üngör and A. Sheffer, "Pitching tents in space-time: Mesh generation for discontinuous galerkin method," *International Journal of Foundations of Computer Science*, vol. 13, no. 2, pp. 201–221, 2002.

[3] R. Abedi, B. Petracovici, and R. Haber, "A spacetime discontinuous galerkin method for linearized elastodynamics with element-wise momentum balance," *Computer Methods in Applied Mechanics and Engineering*, vol. 195, no. 2528, pp. 3247 – 3273, 2006.

[4] J. Malone, "Automated mesh decomposition and concurrent finite element analysis for hypercube multiprocessor computers," *Computer Methods in Applied Mechanics and Engineering*, vol. 70, no. 1, pp. 27–58, 1988.

[5] J.-C. Luo and M. Friedman, "A parallel computational model for the finite element method on a memory-sharing multiprocessor computer," *Computer Methods in Applied Mechanics and Engineering*, vol. 84, no. 2, pp. 193–209, 1990.

[6] D. Goehlich, L. Komzsik, and R. Fulton, "Application of a parallel equation solver to static fem problems," *Computers and Structures*, vol. 31, no. 2, pp. 121–129, 1989.

[7] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[8] K. Danielson, S. Hao, W. Liu, R. Uras, and S. Li, "Parallel computation of meshless methods for explicit dynamic analysis," *International Journal for Numerical Methods in Engineering*, vol. 47, no. 7, pp. 1323–1341, 2000.

[9] C. Farhat and F. X. Roux, "A method of finite element tearing and interconnecting and its parallel solution algorithm," *International Journal for Numerical Methods in Engineering*, vol. 32, no. 6, pp. 1205–1227, 1991.

[10] C. Farhat, M. Lesoinne, P. Letallec, K. Pierson, and D. Rixen, "Feti-dp: A dual-primal unified feti method part i: A faster alternative to the two-level feti method," *International Journal for Numerical Methods in Engineering*, vol. 50, no. 7, pp. 1523–1544, 2001.

[11] B. Cockburn and C.-W. Shu, "The runge-kutta discontinuous galerkin method for conservation laws v: Multidimensional systems," *Journal of Computational Physics*, vol. 141, no. 2, pp. 199–224, 1998.

[12] F. Bassi and S. Rebay, "High-order accurate discontinuous finite element solution of the 2d euler equations," *Journal of Computational Physics*, vol. 138, no. 2, pp. 251–285, 1997.

[13] H. Atkins and C.-W. Shu, "Quadrature-free implementation of discontinuous galerkin method for hyperbolic equations," *AIAA Journal*, vol. 36, no. 5, pp. 775–782, 1998.

[14] Y. Xia, H. Luo, M. Frisbey, and R. Nourgaliev, "A set of parallel, implicit methods for a reconstructed discontinuous galerkin method for compressible flows on 3d hybrid grids," *Computers and Fluids*, vol. 98, pp. 134–151, 2014.

[15] H. Luo, L. Luo, A. Ali, R. Nourgaliev, and C. Cai, "A parallel, reconstructed discontinuous galerkin method for the compressible flows on arbitrary grids," *Communications in Computational Physics*, vol. 9, no. 2, pp. 363–389, 2011.

[16] K. Eriksson, C. Johnson, and A. Logg, "Explicit time-stepping for stiff odes," *SIAM Journal on Scientific Computing*, vol. 25, no. 4, pp. 1142–1157, 2003.

[17] R. Abedi and R. Haber, "Riemann solutions and spacetime discontinuous galerkin method for linear elastodynamic contact," *Comput. Methods Appl. Mech. Engrg.*, vol. 270, pp. 150–177, 2014.

[18] R. Abedi, S.-H. Chung, J. Erickson, Y. Fan, M. Garland, D. Guoy, R. Haber, J. Sullivan, S. Thite, and Y. Zhou, "Spacetime meshing with adaptive refinement and coarsening," *Proceedings of the Annual Symposium on Computational Geometry*, pp. 300–309, 2004.

[19] S. Thite, "Adaptive spacetime meshing for discontinuous galerkin methods," *Computational Geometry: Theory and Applications*, vol. 42, no. 1, pp. 20–44, 2009.

[20] J. Simo, N. Tarnow, and K. Wong, "Exact energy-momentum conserving algorithms and symplectic schemes for nonlinear dynamics," *Computer Methods in Applied Mechanics and Engineering*, vol. 100, no. 1, pp. 63–116, 1992.

[21] C. Kane, J. Marsden, M. Ortiz, and M. West, "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.

[22] A. Lew, J. Marsden, M. Ortiz, and M. West, "Variational time integrators," *International Journal for Numerical Methods in Engineering*, vol. 60, no. 1, pp. 153–212, 2004.

[23] T. Hughes and G. Hulbert, "Space-time finite element methods for elastodynamics: Formulations and error estimates," *Computer Methods in Applied Mechanics and Engineering*, vol. 66, no. 3, pp. 339–363, 1988.

[24] X. Li and N.-E. Wiberg, "Structural dynamic analysis by a time-discontinuous galerkin finite element method," *International Journal for Numerical Methods in Engineering*, vol. 39, no. 12, pp. 2131–2152, 1996.

[25] M. Grote, A. Schneebeli, and D. Schtzau, "Discontinuous galerkin finite element method for the wave equation," *SIAM Journal on Numerical Analysis*, vol. 44, no. 6, pp. 2408–2431, 2006.

[26] L. Yin, A. Acharya, N. Sobh, R. B. Haber, and D. A. Tortorelli, "A space-time discontinuous galerkin method for elastodynamic analysis," in *Discontinuous Galerkin Methods: Theory, Computation and Applications*, B. Cockburn, G. E. Karniadakis, and C.-W. Shu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 459–464.

[27] R. Abedi, B. Petracovici, and R. Haber, "A space-time discontinuous galerkin method for linearized elastodynamics with element-wise momentum balance," *Computer Methods in Applied Mechanics and Engineering*, vol. 195, no. 25-28, pp. 3247–3273, 2006.

[28] B. S. Miller, R. H. Kraczek, and D. Johnson, "Multi-field spacetime discontinuous galerkin methods for linearized elastodynamics," *Computer Methods in Applied Mechanics and Engineering*, vol. 199, no. 14, pp. 34–47, 2009.

[29] R. P. Garg and I. A. Sharapov, *Techniques for optimizing applications - high performance computing.* Upper Saddle River, NJ: Prentice Hall PTR.