A PARALLEL GEOMETRIC MULTIGRID METHOD FOR FINITE ELEMENTS ON OCTREE MESHES APPLIED TO ELASTIC IMAGE REGISTRATION

A Dissertation
Presented to
The Academic Faculty

by

Rahul S. Sampath

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
Computational Science and Engineering

College of Computing
Georgia Institute of Technology
August 2009

A PARALLEL GEOMETRIC MULTIGRID METHOD
FOR
FINITE ELEMENTS ON OCTREE MESHES
APPLIED TO
ELASTIC IMAGE REGISTRATION

Approved by:

Dr. Richard Vuduc, Committee Chair
College of Computing
Georgia Institute of Technology

Dr. George Biros, Advisor
College of Computing
Georgia Institute of Technology

Dr. Allen Tannenbaum
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Hao Min Zhou
School of Mathematics
Georgia Institute of Technology

Dr. Christos Davatzikos
Department of Radiology
University of Pennsylvania

Date Approved: 18 June 2009

To my parents

Malini Sampath and V.S. Sampath

and my brothers

Balaji and Rajiv

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor, Dr. George Biros, for his guidance and support over the last 5 years. I would like to thank Dr. Christos Davatzikos, Dr. Allen Tannenbaum, Dr. Richard Vuduc and Dr. Hao Min Zhou for serving on my thesis committee. I am also grateful to Dr. Hari Sundar, Dr. Santi Swaroop Adavani and Dr. Ilya Lashuk for their support in developing Dendro. I would like to acknowledge the support for this work through the U.S. Department of Energy under grant DE-FG02-04ER25646, and the U.S. National Science Foundation grants CCF-0427985, CNS-0540372, DMS-0612578, OCI-0749285 and OCI-0749334. Computing resources on the TeraGrid systems were provided under the grants ASC070050N and MCA04T026. I am also grateful to the TeraGrid support staff and consultants at the National Center for Supercomputing Applications, Pittsburgh Supercomputing Center and Texas Advanced Computing Center for accomodating my various requests. I would also like to thank the PETSc team. Finally, I would like to thank my family and friends who have always supported me.

TABLE OF CONTENTS

CHAPTERS

APPENDICES

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The first component of this thesis is a parallel algorithm for constructing octree meshes for finite element computations. Prior to octree meshing, the linear octree data structure must be constructed and a constraint known as “2:1 balancing” must be enforced; parallel algorithms for these two subproblems are also presented. The second component of this thesis is a parallel geometric multigrid algorithm for solving elliptic partial differential equations (PDEs) using these octree meshes. The last component of this thesis is a parallel multiscale Gauss Newton optimization algorithm for solving the elastic image registration problem. The registration problem is discretized using finite elements on octree meshes and the parallel geometric multigrid algorithm is used as a preconditioner in the Conjugate Gradient (CG) algorithm to solve the linear system of equations formed in each Gauss Newton iteration.

The parallel octree meshing and multigrid algorithms have several physical and computer science applications such as in solid/fluid mechanics, heat/mass transfer, electromagnetism, image processing and unstructured mesh generation. Potential applications for the image registration algorithm include automatic identification of abnormalities in medical images, motion reconstruction from temporal sequences of images and planning of surgeries.

Several ideas were used to reduce the overhead for constructing the octree meshes. These include (a) a way to lower communication costs by reducing the number of synchronizations and reducing the communication message size, (b) a way to reduce the number of searches required to build element-to-vertex mappings, and (c) a compression scheme to reduce the memory footprint of the entire data structure. To our knowledge, the multigrid algorithm presented in this work is the only matrix-free multiplicative geometric multigrid implementation for solving finite element equations on octree meshes using thousands of processors.

The overall scheme is second-order accurate, for sufficiently smooth right-hand sides and material properties; and its complexity, for nearly uniform trees, is $\mathcal{O}(\frac{N}{n_p}\log\frac{N}{n_p})+\mathcal{O}(n_p\log n_p)$. Here, N is the number of octants and n_p is the number of processors. The proposed registration algorithm is also unique; it is a combination of many different ideas: adaptivity, parallelism, fast optimization algorithms, and fast linear solvers.

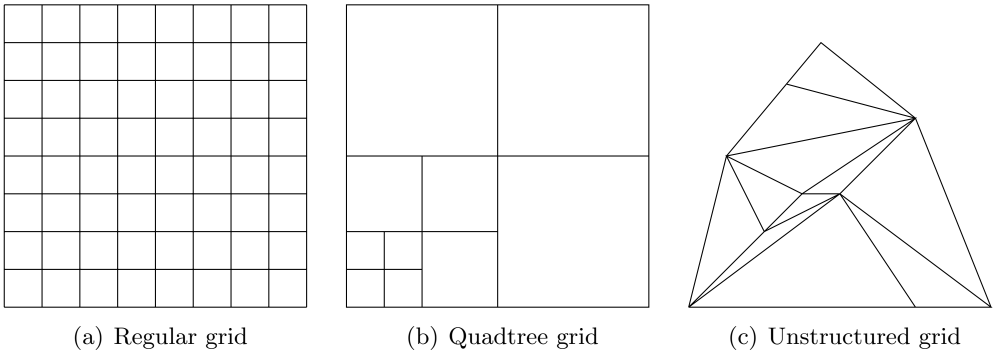
All the algorithms were implemented in C++ using the Message Passing Interface (MPI) standard and were built on top of the `PETSc` library from Argonne National Laboratory. The multigrid implementation has been released as an open source software: `Dendro`. Several numerical experiments were performed to test the performance of the algorithms. These experiments were performed on a variety of NSF TeraGrid platforms: on the Cray XT3 MPP system "*Bigben*" at the Pittsburgh Supercomputing Center (PSC), the Intel 64 Linux Cluster "*Abe*" at the National Center for Supercomputing Applications (NCSA), and the Sun Constellation Linux Cluster "*Ranger*" at the Texas Advanced Computing Center (TACC). Our largest run was a highly-nonuniform, 8- billion-unknown, elasticity calculation on 32,000 processors.

CHAPTER I

INTRODUCTION

The finite element method is a popular technique for solving partial differential equations (PDEs) numerically. Finite element methods require grid generation (or meshing) to generate function approximation spaces. Regular grids are easy to generate, but can be quite expensive when the solution of the PDE is highly localized. Localized solutions can be captured more efficiently using non-uniform or unstructured grids. However, the flexibility of unstructured grids comes at a price — they are difficult to construct in parallel, they are difficult to precondition, and they incur the overhead of explicitly constructing element-to-vertex connectivity information, they are unsuitable for matrix-free implementations and are generally cache inefficient because of random queries into this data structure [6, 55, 135]. Octree meshes seem like a promising alternative, at least for some problems [3, 13, 93]; they are more flexible than uniform grids, the overhead of constructing element-to-node connectivity information is lower than that of unstructured grids and they allow for matrix-free implementations (Figure 1). Constructing parallel octree meshes for finite element computations involves several challenges; the first part of this thesis addresses these challenges.

Besides grid-generation, an optimal solver is also necessary for good scalability. In



(a) Regular grid

(b) Quadtree grid

(c) Unstructured grid

Figure 1: Varying degrees of adaptivity: (a) Regular grid, (b) Quadtree grid and (c) Unstructured grid. Quadtrees are 2-D analogues of Octrees.

this thesis, we focus on elliptic PDEs and multigrid methods are known to be efficient for solving these type of PDEs. A distinguishing feature of multigrid algorithms is that their convergence rate does not deteriorate with increasing problem size [25, 61, 126]. Some multigrid implementations even obtain optimal complexity by combining this feature with an operation count linear in the number of unknowns.

Multigrid algorithms can be classified into two categories: (a) geometric and (b) algebraic; the primary difference being that algorithms of the former type use an underlying mesh for constructing coarser multigrid levels ("*coarsening*") and algorithms of the latter type use the entries of the fine-grid matrix for coarsening in a black-box fashion. Algebraic multigrid methods are gaining prominence due to their generality and the ability to deal with unstructured meshes. Geometric multigrid methods are less general, but have low overhead, are quite fast, and are easy to parallelize (at least for structured grids). For these reasons, geometric multigrid methods have been quite popular for solving smooth coefficient non-oscillatory elliptic PDEs on structured grids.

The main components of any multigrid algorithm include the construction of a sequence of coarse meshes and the construction of inter-grid transfer operations. Both of these operations are non-trivial to implement, particularly in parallel, for non-uniform meshes. In this thesis, we developed efficient parallel algorithms for performing these operations on octree meshes. To our knowledge there is no other work on octree-based, matrix-free, geometric multigrid solvers for finite element discretizations that has scaled to thousands of processors.

The multigrid algorithm developed in this thesis has several physical applications involving heat and mass transfer theory [35], solid and fluid mechanics [35, 56] and electromagnetism [54]. They can also be used in non-physical applications such as mesh generation [114] and image processing [47, 88]. In this thesis we have applied it to the image registration problem, which is an important image processing operation. Image registration, particularly nonlinear registration using non-parametric deformation models, is one of the challenging problems in image processing today. A few approaches to reduce the computational time for solving this problem have been proposed in the literature. These include adaptive schemes, fast optimization algorithms such as quasi-Newton and Gauss-Newton methods, fast linear solvers like multigrid and fast fourier transforms (FFTs) and parallelization. The final part of this thesis describes our approach to this problem, which uses a combination of all these ideas.

1.1 Related work

In this section, we will review some related work on constructing parallel octrees, enforcing the 2:1 balance constraint, meshing octrees and using them for finite element computations, multigrid and image registration.

1.1.1 Constructing octrees

The key component in constructing parallel octrees is the partitioning of the input in order to achieve good load balancing. The use of space-filling curves for partitioning data has been quite popular [62, 129, 134, 140]. The proximity preserving property of space-filling curves makes them attractive for data partitioning. Typical approaches to parallel octree construction use a top-down approach after the initial partition. The major hurdle in using a parallel top-down approach is avoiding overlaps. This typically requires significant synchronization and communication after constructing a portion of the tree [129, 134, 140]. In Section 2.2.2, we present an alternative bottom-up approach to constuct parallel linear octrees with little communication.

1.1.2 2:1 Balancing octrees

Balance refinement is a key pre-processing operation in order to use octrees for finite element computation. The only known parallel algorithms for this problem are presented in [16, 119, 129]. Bern et al. [16] proposed an algorithm for balancing quadtrees for EREW PRAM architectures; this cannot be easily adapted for distributed architectures. In addition, the balanced quadtree produced is suboptimal and can have up to 4 times as many cells as the optimal balanced quadtree. Tu et al. [129] propose a more promising approach, which was evaluated on billions of elements using thousands of processors. In Section 2.2.3 we present a way to decouple the problem of balancing and reduce communication costs.

1.1.3 Meshing octrees

Unstructured meshes are used extensively to solve problems with localized solutions and problems involving complex geometries. A pre-processing step in any unstructured finite element computation is the construction of the element-to-vertex connecitivity information; this operation is known as meshing. However, generating large unstructured meshes is a challenging task [111] and existing implementations do not scale well to many thousands of processors. Moreover, generic unstructured meshing schemes are not suitable for matrix-free implementations and tend to break down due to bad element quality during the remeshing step. On the contrary, octree-based unstructured hexahedral meshes can be constructed efficiently [17, 52, 106, 107, 110, 128] and the resulting quality of the elements is good. Scalable algorithms for parallel octree meshing are presented in [29, 129]. [129] describes an algorithm to mesh a parallel octree and [29] presents an algorithm to mesh a forest of parallel octrees. New parallel algorithms to mesh and compress octrees are presented in Chapter 3.

1.1.4 FEM using octrees

Examples of large scale finite element computations using parallel octrees can be found in [3, 29, 73, 129]. A characteristic feature of octree meshes is that they contain "*hanging*" vertices. Projection schemes are typically used to preserve the continuity of the solution at hanging vertices. Alternatively, one could modify the element shape functions for the elements that contain hanging vertices so that the continuity of the solution is automatically enforced. We discuss the latter approach in Section 3.3.

1.1.5 Multigrid

Multigrid methods for solving elliptic PDEs have been researched extensively in the past [11, 21, 22, 36, 57, 61, 109, 141, 142, 143, 144] and remain an active research area [1, 2, 13, 15, 53, 57, 71]. Here, we review some of the recent work on multigrid for adaptive meshes. In [18], a sequential geometric multigrid algorithm was used to solve two and three dimensional linear elastic problems using finite elements on non-nested unstructured triangular and tetrahedral meshes, respectively. The implementation of the intergrid transfer operations described in this work can be quite expensive for large problems and is non-trivial to parallelize. A sequential multigrid scheme for finite element simulations of non-linear problems on quadtree meshes was described in [71]. In addition to the 2:1 balance constraint, a specified number of "*safety layers*" of octants were added at each multigrid level to support their intergrid transfer operations. Projections were also required at each multigrid level to preserve the continuity of the solution, which is otherwise not guaranteed using their non-conforming discretizations. Projection schemes require two additional tree-traversals per MatVec, which we avoid in our approach. Multigrid algorithms for quadtree/octree meshes were also described in [13, 14, 93]. [93] created the multigrid hierarchy using a simple coarsening strategy in which only the octants at the finest level were coarsened at each stage. While that coarsening stategy ensures that the 2:1 balance constraint is automatically preserved after each stage of coarsening, the decrease in the number of elements after coarsening might be small. An alternate coarsening strategy that tries to coarsen all octants was used in [13, 14] and in the present work. [13] describes a sequential multigrid algorithm and the corresponding parallel extension is described in [14]. In [14] a sequential graph-based scheme was used to partition the meshes on a dedicated master processor and the resulting partitioned meshes were handed out to client processors, which performed the parallel multigrid solves. Hence, the scalability of their implementation was limited by the amount of memory available on the master processor. Moreover, the partitioning can be more expensive than the parallel computation of the solution. Simpler scalable partitioning schemes based on space-filling curves have been used in [28, 53, 85] and in the present work. All these algorithms work on adaptive hierarchical Cartesian grids, which are constructed by the recursive refinement of grid cells into a fixed number of congruent subcells. In the approach used in [28, 85], each refinement produced 3 subcells in each coordinate direction. In the approach used in the present work and in [53], each refinement produces 2 subcells in each coordinate direction and so the number of elements grow slower in this approach compared to the former approach. [28, 53] used the additive version of multigrid, which is simpler to parallelize compared to the multiplicative version of multigrid used in the present work. However, the multiplicative version is considered to be more robust than the additive version as far as convergence rates are concerned [12]. A parallel multiplicative multigrid algorithm for non-uniform meshes was presented in [78]; that work reported good scalability results on up to 512 processors. In [78], the smoothing at each grid was performed only in the refined regions and in a small neighborhood around the refined regions. In contrast, we chose to cover the entire domain at each grid and this allows us to use a simpler scheme to distribute the load across processors. A 3-D parallel algebraic multigrid method for unstructured finite element problems was presented in [2]. In that work, the authors used parallel maximal independent set algorithms for constructing the coarser grids and constructed the Galerkin coarse-grid operators algebraically using the restriction operators and the fine-grid operator. In [15], a calculation with over 11 billion elements was reported. The authors proposed a scheme for conforming discretizations and geometric multigrid solvers on semi-structured meshes. That approach is highly scalable for nearly structured meshes but it somewhat limits adaptivity because it is based on regular refinement. Additional examples of scalable approaches for non-uniform meshes include [1] and [83]. In those works, multigrid approaches for general elliptic operators were proposed. The associated constants for constructing the mesh and performing the calculations however, are quite large. A significant part of CPU time is related to the multigrid scheme. The high-costs related to partitioning, setup, and accessing generic unstructured grids, has motivated the use of octree-based data structures. A parallel, octree-based, geometric multigrid solver for finite element discretizations is described in Chapter 4.

1.1.6 Image registration

Image registration has been an active research area for the past two decades. [50] gives an exhaustive review of the classical general purpose registration methods. [145] and [88] focus on the more recent methods, the former focusses more on parametric registration methods and the latter focusses more on non-parametric registration methods. [66, 80, 82] focus on registration techniques commonly used in medical image processing.

Many of the early works on registration focussed on rigid or affine transformations.

These models are incapable of capturing the nonlinear deformations typically associated with medical images. Global polynomial models were proposed to tackle these nonlinear deformations. Local transformation models using piecewise polynomials [48] or weighted polynomials [49] were later introduced to handle local deformations better. Radial basis functions [41, 42] and B-splines [7, 76, 77, 113] were also introduced to improve the modelling of local deformations. Thin-plate splines [20, 97] and elastic splines [34, 138] were introduced to get physically meaningful deformations. [98] presented three different types of parametric registration approaches based on splines and anatomical point landmarks: thin-plate splines, radial basis functions with compact support and Gaussian elastic body splines. These parametric registration algorithms are computationally efficient because their search space is typically small, but they do not handle local deformations well. In contrast, non-parametric registration algorithms are well-equipped to deal with local deformations. In the non-parametric case, the transformation model comes directly from the discretization scheme such as finite differencing or finite elements. In the parametric case, the ill-posedness of the registration problem is addressed by the constraints on the displacements imposed by the transformation model; in the non-parametric case, the ill-posedness is addressed by adding an explicit penalty or regularization term to the objective function. Different choices for regularization give rise to different registration algorithms: diffusive registration [39], elastic registration [5, 63, 67, 136], fluid registration [26, 137] and curvature registration [40]. There are also examples of hybrid approaches that combine parametric and non-parametric registration methods: [105] used elastic registration on affine registered images to improve the accuracy of reconstruction. Some tools for rigid, affine, spline-based and demon's registration in two or more dimensions can be found in the open-source software: Insight Segmentation and Registration Toolkit (ITK) [68]. It provides implementations for different similarity metrics, various interpolation schemes and derivative-free and gradient-based optimizers.

The use of multigrid for image registration is fairly recent and some of the relevant works include [31, 60, 63, 67, 72]. [31] used a diffusive regularizer and a steepest-descent type optimization algorithm accelerated using a multigrid solver. [63] used a Full Approximation

Scheme (FAS) and d-linear image approximation for elastic registration. [67] presented a multigrid scheme using operator dependent prolongation for elastic image registration. [72] used a parallel multigrid algorithm for the optical flow problem with a diffusive regularizer. [60] presented a Gauss Newton algorithm and a multigrid scheme for solving the elastic registration problem using a regular grid discretization.

The use of octrees/quadtrees for image registration is also a fairly recent idea [58, 59, 75, 120, 121]. [58] presented a parametric registration algorithm using octree discretization and [59] used octrees for elastic image registration. [75] used quadtrees for affine image registration. [120] and [121] used a family of volumetric tensor product first order (linear) B-splines whose coefficients were defined on an octree and quadtree grid to model the transformation for the registration problem.

There is also little work on parallel image registration. [37, 89, 131] focussed on rigid registration using derivative-free optimization algorithms. A steepest-descent approach was used in [69] for parallel rigid registration of 2-D images to 3-D volumes. A parallel non-rigid registration algorithm using a B-spline transformation model was presented in [70]. [115] parallelized the 3-D demon's registration algorithm. [19] used a fixed point iteration combined with parallel FFTs to solve the 2-D elastic registration problem. [86, 87] also used a fixed point iteration to solve the 2-D elastic registration problem and used a parallel Conjugate Gradient (CG) method to solve the linear system within the nonlinear iteration. [72] presented a parallel multigrid scheme to solve the 3-D optical flow problem.

1.2 Contributions

The contributions of this thesis are summarized below.

Construction and 2:1 Balancing A parallel bottom-up algorithm for constructing linear octrees with little communication was developed. A hybrid algorithm was also developed to enforce the 2:1 balance constraint in parallel. This is required for using linear octrees for finite element computations. We introduced a way to decouple the 2:1 balancing problem and used it to reduce the number of synchronizations and the total communication message size compared to earlier approaches to the problem.

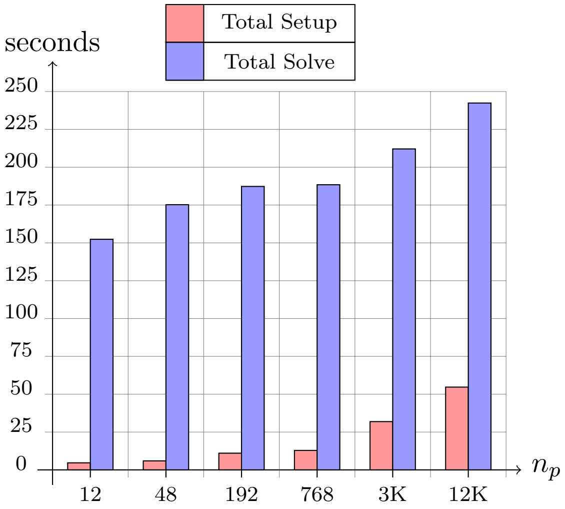


Figure 2: The results from an isogranular scalability experiment using `Dendro`. In this experiment, a linear elastostatic problem was solved on a set of octrees with a grain size (on the finest multigrid level) of approximately 80K elements per processor. The octrees were generated using a Gaussian distribution of points. A relative tolerance of 10^{-10} in the 2-norm of the residual was used. The time (in seconds) to setup and solve the problem in each case are reported. This experiment was performed on "Ranger".

This work was published in [119].

Meshing In this work, a parallel algorithm is presented to build data structures that store the element-to-vertex connectivity information, which is required for finite element computations. We use these data structures to build second-order accurate discretizations of PDEs. A compression scheme for the octree and the element connectivity is also presented that achieves a three-fold compression (a total of four words per octant). This work was published in [118].

Multigrid In this work, a matrix-free, geometric multigrid algorithm was designed and implemented for solving elliptic PDEs using finite elements on parallel octree meshes. The setup costs of our algorithm is low making it ideal for applications that require repeated solutions of linear systems of equations. This is significant for time dependent and nonlinear problems. The MPI-based implementation of our method, `Dendro`, has scaled to billions of elements on thousands of processors. Figure 2 shows an example of the performance of `Dendro`. This work was published in [101] and additional algorithmic details and numerical experiments were reported in [102]. Our implementation

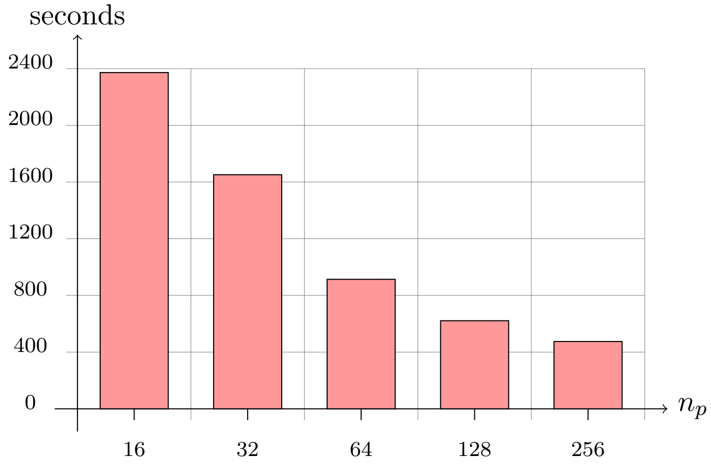


Figure 3: The time (in seconds) to register MR images of the brains of two different subjects using the octree-based multiscale Gauss Newton multigrid algorithm on varying number of processors. The original resolution of the images was 256 × 256 × 171 and the corresponding octree mesh had approximately 430K elements. This experiment was performed on "Ranger".

has also been released as an open source software [104].

Image Registration In this work, a multiscale Gauss Newton algorithm is presented for solving the elastic image registration problem. The linear system that is formed in each optimization iteration is solved using our octree-based matrix-free geometric multigrid algorithm. Our parallel implementation helps reduce the computation time for registration and also allows us to register images that are too large to fit on a single processor's memory. Figure 3 shows an example of the performance of this algorithm. This work has been submitted for publication [103].

1.3 Limitations

In this section, we list the limitations of this thesis.

Octree-based finite element discretization Our finite element discretization only results in a second-order accurate method. Our implementation does not directly support problems involving complex geometries; in principle, `Dendro` can be combined with fictitious domain methods [45, 94] to allow solution of such problems but the computational costs will increase and the order of accuracy may be reduced. Far-field and periodic boundary conditions are not supported in our implementation.

Multigrid Although the algorithm has been successfully applied to solve many problems with large jumps in the material properties, it can not be guaranteed to be robust for such problems. The convergence tends to deteriorate with increasing number and magnitude of discontinuities. We use a simple weighted partitioning heuristic to tackle the issue of load balancing across processors. However, load balancing is still a challenge and has not been fully addressed in this thesis.

Elastic registration We used the linear theory of elasticity, which is only valid for small deformations. Other regularization approaches may be more appropriate for large deformations. Further, we do not incorporate any biophysical information to additionally constrain the deformation. It has been suggested that incorporating such information will provide intelligent priors and reduce the ill-posedness of the registration problem [117]. Finally, our implementation does not use adaptive integration. Instead, we fix the order of the Gauss quadrature rule for integration a-priori. The use of adaptive integration can further reduce the computation costs for evaluating the objective function and gradient.

1.4 Future work

There are two important extensions for our octree framework: higher-order discretizations and integration with domain-decomposition methods such as the Hierarchical Hybrid Grids (HHG) scheme described in [15]. The former will result in improved accuracy with fewer elements and the latter will help solve problems involving complicated geometries with fewer elements. The last point stems from the fact that using a single octree to mesh a domain is more restrictive than allowing the use of multiple octrees, each of which is only responsible for a part of the entire domain.

In this thesis, we also applied our parallel octree framework to the elastic image registration. We anticipate a need to analyze very high resolution images in the future and we believe that scalable parallel registration algorithms are essential for such analysis. We also envision the use of the proposed framework in inverse biophysical applications in which high resolution images are used to estimate certain material parameters in biophysical models;

there is already some work in this direction [99, 108].

Our elastic registration framework will lay the foundation for two important extensions: (a) nonlinear elastic registration and (b) biophysically constrained registration. The former will be more appropriate for large deformations and the latter is a way to introduce informative priors for the registration.

1.5 Organization of the thesis

The rest of the thesis is organized as follows. Chapter 2 introduces some terminology related to octree data structures and describes parallel algorithms to construct and 2:1 balance linear octrees. Chapter 3 describes a parallel algorithm to mesh the 2:1 balanced linear octrees and describes the construction of finite element approximation spaces using these meshes. Chapter 4 presents a parallel geometric multigrid for solving elliptic PDEs using these octree-based finite element discretizations. Finally, Chapter 5 presents a parallel octree-based multiscale Gauss Newton Multigrid algorithm for intensity-based elastic image registration.

CHAPTER II

CONSTRUCTION AND 2:1 BALANCE-REFINEMENT OF OCTREES

This chapter presents an overview of the octree data structure and describes parallel algorithms to construct and 2:1 balance refine large linear octrees on distributed memory machines. Octrees are used in many problems in computational science and engineering: they can be used as algorithmic foundations for adaptive finite element methods [13, 73], adaptive mesh refinement methods [53, 93], and many-body algorithms [62, 124, 133, 139, 140]. These tree data structures have been in use for over three decades now [38, 100]. However, design and use of large scale distributed tree data structures that scale to thousands of processors is still a major challenge and is an area of active research even today [16, 30, 43, 53, 62, 124, 129, 133, 134, 139, 140].

Octrees are usually employed while solving the following two types of problems.

- *Searching:* Searches within a domain using d-trees (d-dimensional trees with a maximum of 2^d children per node), benefit from the reduction of the complexity of the search from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$ [44, 91].

- *Spatial decomposition:* Typical approaches for spatial decomposition include logically structured grids, block structured and overlapping grids, unstructured grids, and octrees. All methods have advantages and disadvantages. For example, structured grids are relatively easy to implement, have low memory requirements, and avoid indirect memory references. Structured grids however, limit adaptivity; for certain problems this limitation can result in excessively large systems of equations. Although unstructured meshes can conform to complex geometries and enable non-uniform discretizations, they incur the overhead of having to explicitly store element-node connectivity information and in general being cache inefficient because of random access [6, 55, 135]. Octrees offer a good balance between adaptivity and efficient performance for several applications like solid modeling [84], object representation [8, 27], visualization [43], image segmentation [116], adaptive mesh refinement [53, 93] and N-body simulations [62, 124, 133, 134, 139, 140].

Octree data structures used in discretizations of partial differential equations should satisfy a constraint known as the 2:1 balance constraint, which imposes a restriction on the relative sizes of adjacent octants[1] [16, 129]. One advantage of enforcing the 2:1 balance constraint is that it ensures that there is at most one *"hanging"* vertex on any edge or face; this makes it easier to construct conforming finite element approximation spaces on octree meshes. What makes the balance-refinement problem difficult and interesting is a property known as the *ripple effect*: An octant can trigger a sequence of splits whereby it can force an octant to split, even if it is not in its immediate neighborhood. Hence, balance-refinement is a non-local and inherently iterative process. Solving the balance-refinement problem in parallel, introduces further challenges in terms of synchronization and communication since the ripple can propagate across multiple processors.

Contributions. The salient contributions of this chapter are:

- A parallel bottom-up algorithm for coarsening octrees, which is also used for partitioning the input in our other algorithms.

- A parallel bottom-up algorithm for constructing linear octrees. We avoid the synchronization issues that are usually associated with parallel top-down methods.

- An algorithm for enforcing 2:1 balance refinement in parallel. The algorithm constructs the minimum number of nodes to satisfy the 2:1 constraint.[2] Its key feature is that it avoids parallel searches, which as we show in sections 2.2.3.6 and 2.2.3.7, are the main hurdles in achieving good isogranular scalability.

[1] A formal definition of the 2:1 balance constraint is given in section 2.1.2.

[2] There exists a unique least common balance refinement for a given octree [90].

Remark. The main parallel tools used in our algorithms are sample sorts (accelerated by bitonic sorts), and standard point-to-point/collective communication calls.[3] In the following sections we present several algorithms for which we give precise `work` and `storage` complexity. For some of the parallel algorithms we also give `time` complexity estimates; this corresponds to wall-clock time and includes work per processor and communication costs. The precise number depends on the initial distribution and the effectiveness of the partitioning. Thus the numbers for `time` are only an estimate under uniform distribution assumptions. If the `time` complexity is not specifically mentioned then it is comparable to that of a sample-sort, which runs in $\mathcal{O}\left(\frac{N}{n_p}\log\left(\frac{N}{n_p}\right) + n_p \log\left(n_p\right)\right)$ time for uniformly distributed points [51].

Organization of the chapter. In Section 2.1 we introduce some terminology that will be used in the rest of the thesis. In Section 2.2, we describe the various components of our construction and balance refinement algorithms. In Section 2.3, we present numerical experiments, including fixed size and isogranular scalability tests on different data distributions. Tables 1 and 2 summarize the notation that is used in the subsequent sections.

2.1 Background

An octree is a tree data structure in which every node[4] has a maximum of eight children. Octrees are analogous to binary trees (maximum of two children per node) in 1-D and quadtrees (maximum of four children per node) in 2-D. A node with no children is called a *leaf* and a node with one or more children is called an *interior node*. The only node with no parent is the *root* and all other nodes have exactly one parent. Nodes that have the same parent are called *siblings*. A node's children, grandchildren and so on and so forth are collectively referred to as the node's *descendants* and this node will be an *ancestor* of its descendants. A node along with all its descendants can be viewed as a separate tree in itself with this node as its root. Hence, this set is also referred to as a *subtree* of the original

[3] When we discuss communication costs, we assume a Hypercube network topology with $\Theta(n_p)$ Bisection Width.

[4] The term "*node*" is usually used to refer to the vertices of elements in a finite element mesh; but, in the context of tree data structures, it refers to the octants themselves.

Table 1: Symbols for terms

$\mathcal{L}(N)$	Level of octant N.
$\mathcal{L}^*$	Maximum level attained by any octant.
D_{max}	Maximum permissible depth of the tree. (Upper bound for $\mathcal{L}^*$).
$\mathcal{P}(N)$	Parent of octant N.
$\mathcal{B}(N)$	The block that is equal to or is an ancestor of octant N.
$\mathcal{S}(N)$	Siblings (sorted) of octant N.
$\mathcal{C}(N)$	Children (sorted) of octant N.
$\mathcal{D}(N)$	Descendant of octant N.
$\mathcal{FC}(N)$	First child of octant N.
$\mathcal{LC}(N)$	Last child of octant N.
$\mathcal{FD}\,(N, l)$	First descendant of octant N at level l.
$\mathcal{LD}\,(N, l)$	Last descendant of octant N at level l.
$\mathcal{DFD}(N)$	Deepest first descendant of octant N.
$\mathcal{DLD}(N)$	Deepest last descendant of octant N.
$\mathcal{A}(N)$	Ancestor of octant N.
$\mathcal{A}_{finest}\,(N, K)$	Nearest Common Ancestor of octants N and K.
$\mathcal{N}\,(N, l)$	List of all potential neighbors of octant N at level l.
$\mathcal{N}^s\,(N, l)$	A subset of $\mathcal{N}(N, l)$, with the property that all of these share the same common corner with N. This is also the corner that N shares with its parent.
$\mathcal{N}\,(N)$	Neighbor of N at any level.
$\mathcal{I}(N)$	Insulation layer around octant N.
N^p_{max}	Maximum number of points per octant.
n_p	Total number of processors.
A_{global}	Union of the list A from all the processors.
$\{\cdots\}$	A set of elements.
$\emptyset$	The empty set.

Table 2: Symbols for operations

$A \leftarrow B$	Assignment operation.
$A \oplus B$	Bitwise A XOR B.
$\{A\} \cup \{B\}$	Union of the sets A and B. The order is preserved, if possible.
$\{A\} \cap \{B\}$	Intersection of the sets A and B.
$A + B$	The list formed by concatenating the lists A and B.
$A - B$	Remove the contents of B from A.
$A[i]$	i^{th} element in list A.
len(A)	Number of elements in list A.
Sort(A)	Sort A in the ascending Morton order.
A.push_front(B)	Insert B to the beginning of A.
A.push_back(B)	Append B to the end of A.
Send(A ,r)	Send A to processor with rank = r.
Receive()	Receive from any processor.

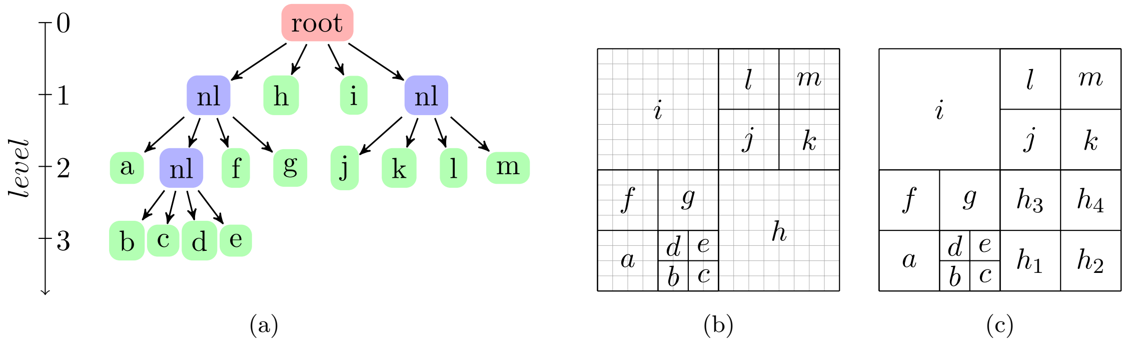


Figure 4: (a) Tree representation of a quadtree and (b) decomposition of a square domain using the quadtree, superimposed over a uniform grid, and (c) a balanced linear quadtree: result of balancing the quadtree.

tree. The depth of a node from the root is referred to as its *level*. As shown in Fig. 4(a), the root of the tree is at level 0 and every interior node is one level lower than its children.

Octrees[5] can be used to partition cuboidal regions (Figure 4(b)). These regions are referred to as the domain of the tree. A set of octants is said to be complete if the union of the regions spanned by them covers the entire domain. Alternatively, one can also define complete octrees as octrees in which every interior node has exactly eight child nodes. We will frequently use the equivalence of these two definitions.

There are many different ways to represent trees [32]. In this work, we will use a linearized representation of octrees known as *linear octrees*. In this representation, we discard the interior nodes and only store the complete list of leaves. This representation is advantageous for the following reasons.

- It has lower storage costs than other representations.
- The other representations use pointers, which add synchronization and communication overhead for parallel implementations.

To use a linear representation, a *locational code* is needed to identify the octants. A locational code is a code that contains information about the position and level of the octant in the tree. The following section describes one such locational code known as the

[5] All the algorithms described in this thesis are applicable to both octrees and quadtrees. For simplicity, we will sometimes use quadtrees to illustrate the concepts in this thesis and use the terms "octrees" and "octants", consistently, in the rest of the thesis.

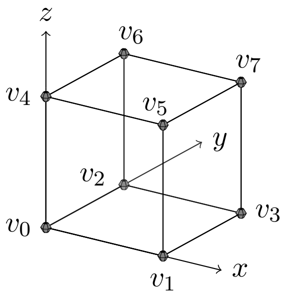


Figure 5: Orientation for an octant. By convention, v_0 is chosen as the anchor of the octant. The vertices are numbered in the Morton ordering.

Morton encoding.[6]

2.1.1 Morton encoding

In order to construct a Morton encoding, the maximum permissible depth, D_{max}, of the tree is specified *a priori*. Note that D_{max} is different from $\mathcal{L}^*$, the maximum level attained by any node. In general, $\mathcal{L}^*$ can not be specified a priori. D_{max} is only a weak upper bound for $\mathcal{L}^*$.

The domain is represented by an uniform grid of $2^{D_{max}}$ indivisible cells in each dimension (Fig. 4(b)). Each cell is identified by an integer triplet representing its x, y and z coordinates, respectively. Any octant in the domain can be uniquely identified by specifying one of its vertices, also known as its *anchor*, and its level in the tree (Fig. 6). By convention, the anchor of a quadrant is it's lower left corner and the anchor of an octant is it's lower left corner facing the reader (corner v_0 in Figure 5).

The Morton encoding for any octant is derived by interleaving[7] the binary representations (D_{max} bits each) of the three coordinates of the octant's anchor, and then appending the binary representation $((\lfloor(\log_2 D_{max})\rfloor + 1)$ bits) of the octant's level to this sequence of bits [16, 30, 125, 129]. Interesting properties of the Morton encoding scheme are listed in Appendix A. In the rest of the thesis the terms *lesser* and *greater* and the symbols $<$ and

[6] Morton encoding is one of many space-filling curves [30]. Our algorithms are generic enough to work with other space-filling curves as well. However, Morton encoding is relatively simpler to implement since, unlike other space-filling curves, no rotations or reflections are performed.

[7] Instead of bit-interleaving as described here, our implementation uses a multicomponent version (Appendix B) of the Morton encoding scheme.

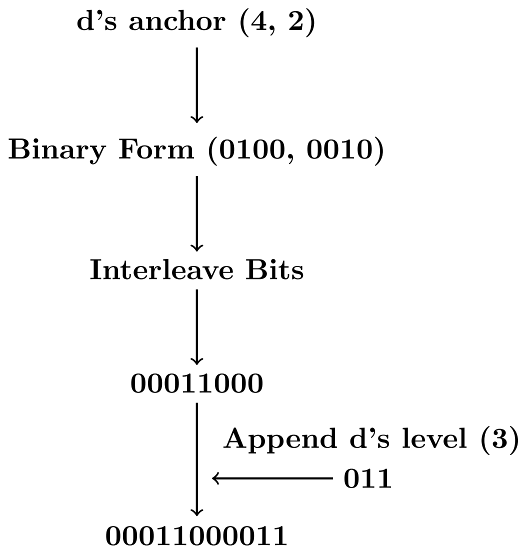


Figure 6: Computing the Morton id of quadrant "d" in the quadtree shown in Fig. 4(b). The anchor for any quadrant is it's lower left corner.

$>$ are used to compare octants based on their Morton ids, and *coarser* and *finer* to compare them based on their relative sizes, i.e., their levels in the octree.

2.1.2 Balance constraint

In many applications involving octrees, it is desirable to impose a restriction on the relative sizes of adjacent octants [65, 73, 129]. Generalizing Moore's [90] categorization of the general balance conditions, we have the following definition for the 2:1 balance constraint:

Definition 1 *A linear d-tree is k-balanced if and only if, for any $l \in [1, \mathcal{L}^*)$, no leaf at level l shares an m-dimensional face*[8] *($m \in [k, d)$) with another leaf, at level greater than $l + 1$.*

For the specific case of octrees we use 2-*balanced* to refer to octrees that are balanced across faces, 1-*balanced* to refer to octrees that are balanced across edges and faces, and 0-*balanced* to refer to octrees that are balanced across corners, edges and faces. The result of imposing the 2:1 balance constraint is that no octant can be more than twice as coarse as its adjacent octants. Similarly, 4:1 and higher constraints can be imposed. In this work, we will restrict the discussion to 2:1 balancing alone. Although, the algorithms presented

[8] *A corner is a 0-dimensional face, an edge is a 1-dimensional face and a face is a 2-dimensional face.*

here can be extended easily to satisfy higher balance constraints as well. An example of a 0-*balanced* quadtree is shown in Figure 4(c). The balance algorithm proposed in this work is capable of *k-balancing* a given complete linear octree, and since it is hardest to 0-*balance* a given octree we report all results for the 0-*balance* case.

2.2 Algorithms

We will first describe a key algorithmic component (Section 2.2.1) that forms the backbone for both our parallel octree construction and balancing algorithms. This is a partition heuristic known as *Block Partition* and is specifically designed for octrees. It has two main sub-components, which are described in Sections 2.2.1.1 and 2.2.1.2.

We then present the parallel octree construction algorithm in Section 2.2.2 and the parallel balancing algorithm in Section 2.2.3. The overall parallel balancing algorithm (Algorithm 11) is made up of several components, which are described in Sections 2.2.3.1 through 2.2.3.6.

2.2.1 Block partition

A simple way to partition the domain into an union of blocks would be to take a top-down approach and create a coarse regular grid, which can be divided[9] amongst the processors. However, this approach does not take load balancing into account since it does not use the underlying data distribution. Alternatively, one could use a space-filling curve to sort the octants and then partition them so that every processor gets an almost equal sized chunk of octants, contiguous in this order. This can be done by assigning the same weight to all the octants and then using Algorithm 2. However, this approach does not avoid overlaps.

Two desirable qualities of any partitioning strategy are load balancing, and minimization of overlap between the processor domains. We use a novel parallel bottom-up coarsening strategy to achieve these. The main intuition behind this partition algorithm (Algorithm 1) is that a coarse grid partition is more likely to have a smaller overlap between the processor domains as compared to a partition computed on the underlying fine grid. This algorithm

[9] If we create a regular grid at level l then the number of cells will be $n = 2^{dl}$, where d is the dimension. l is chosen in such a way that $n > p$.

comprises of 3 main stages:

1. Constructing a distributed coarse complete linear octree that is representative of the underlying data distribution.

2. Assigning weights to the octants in the coarse octree and partitioning them to achieve almost uniform load across the processors.

3. Projecting the partitioning computed in the previous step onto the original (fine) linear octree.

We sort the leaves according to their Morton ordering and then distribute them uniformly across the processors. We select the least and the greatest octant at each processor (e.g., octants a and h from Figure 7(a)) and complete the region between them, as described in Section 2.2.1.1, to obtain a list of coarse octants. We then select the coarsest cell(s) out of this list of coarse octants (octant e in Figure 7(a)). We use the selected octants at each processor and construct a complete linear octree as described in Section 2.2.1.2. The leaves of this complete linear octree are referred to as *Blocks*. This gives us a distributed coarse complete linear octree that is based on the underlying data distribution.[10]

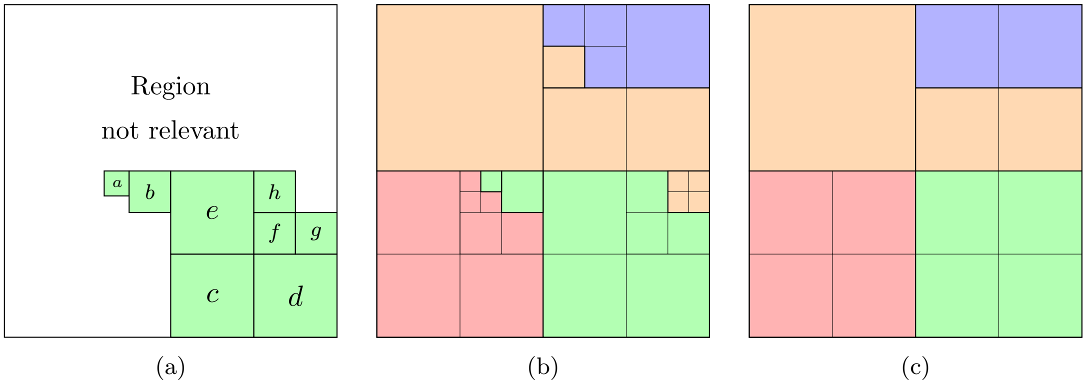
We compute the load of each of the blocks created above by computing the number of original octants that lie within it. The blocks are then distributed across the processors using Algorithm 2 so that the total weight on each processor is roughly the same.[11]

The original octants are then partitioned to align with the coarse block boundaries. Note that the domain occupied by the blocks and the original octants on any given processor is not the same, but it does overlap to a large extent. The overlap is guaranteed by the fact that both are sorted according to the Morton ordering and that the partitioning was based on the same weighting function (i.e., the number of original octants).

Algorithm 1 lists all the steps described above and Figures 7(b) and 7(c) illustrate a sample input to Algorithm 1 and the corresponding output, respectively.

[10] Refer to the Appendix C for an estimate of the number of blocks produced.

[11] Some of the coarse blocks could be split if it facilitates achieving better load balance across the processors.



(a) (b) (c)

Figure 7: `(a)` A minimal list of quadrants covering the local domain on a processor, and `(b)` A Morton ordering based partition of a quadtree across 4 processors, and `(c)` the coarse quadrants and the final partition produced by using the quadtree shown in `(b)` as input to `Algorithm 1`.

Algorithm 1 PARTITIONING OCTANTS INTO LARGE CONTIGUOUS BLOCKS (PARALLEL) - `BlockPartition`

```
Input:    A distributed sorted list of octants, $F$.
Output:   A list of the blocks, $G$.  $F$ is re-distributed,
          but the relative order of the octants is preserved.
Work:     $\mathcal{O}(n)$, where $n =$ len($F$).
Storage:  $\mathcal{O}(n)$, where $n =$ len($F$).
Time:     Refer to the Appendix C.

$T \leftarrow$ CompleteRegion($F[1]$, $F[\texttt{len}(F)]$) ( Algorithm 3 )
$C \leftarrow \{x \in T \mid \forall y \in T,\ \mathcal{L}(x) \leq \mathcal{L}(y)\}$
$G \leftarrow$ CompleteOctree($C$) ( Algorithm 4 )
for each $g \in G$
    weight($g$) $\leftarrow$ len($F_{global} \cap \{g, \{\mathcal{D}(g)\}\}$)
end for
Partition($G$) ( Algorithm 2 )
$F \leftarrow F_{global} \cap \{\{g, \{\mathcal{D}(g)\}\},\ \forall\ g \in G\}$
```

Algorithm 2 Partitioning a distributed list of octants (parallel) - `Partition`

Input: A distributed list of octants, W.

Output: The octants re-distributed across processors so that the total weight on each processor is roughly the same. The relative order of the octants is preserved.

Work: $\mathcal{O}(n)$, where $n =$ len(W).
Storage: $\mathcal{O}(n)$, where $n =$ len(W).

```
$S \leftarrow$ Scan( weight($W$) )
if rank = $(n_p - 1)$
  TotalWeight $\leftarrow \max(S)$
  Broadcast(TotalWeight)
end if
$\bar{w} \leftarrow \frac{\texttt{TotalWeight}}{n_p}$
$k \leftarrow (\texttt{TotalWeight}) \mod n_p$
$Q_{tot} \leftarrow \emptyset$
for $p \leftarrow 1$ to $n_p$
    if $p \leq k$
        $Q \leftarrow \{x \in W \mid (p-1).(\bar{w}+1) \leq S(x) < p.(\bar{w}+1)\}$
    else
        $Q \leftarrow \{x \in W \mid (p-1).\bar{w}+k \leq S(x) < p.\bar{w}+k\}$
    end if
    $Q_{tot} \leftarrow Q_{tot} + Q$
    Send($Q$, $(p-1)$)
end for
$R \leftarrow$ Receive()
$W \leftarrow W - Q_{tot} + R$
```

Algorithm 3 Constructing a minimal linear octree between two octants (sequential) - `CompleteRegion`

Input: `Two octants,` a `and` $b > a$.
Output: R, `the minimal linear octree between` a `and` b.
Work: $\mathcal{O}(n \log n)$, `where` $n =$ `len`(R).
Storage: $\mathcal{O}(n)$, `where` $n =$ `len`(R).

```
$W \leftarrow \mathcal{C}(\mathcal{A}_{finest}(a,b))$
for each $w \in W$
    if $(a < w < b)$ AND $(w \notin \{\mathcal{A}(b)\})$
        $R \leftarrow R + w$
    else if $(w \in \{\{\mathcal{A}(a)\}, \{\mathcal{A}(b)\}\})$
            $W \leftarrow W - w + \mathcal{C}(w)$
    end if
end for
Sort(R)
```

2.2.1.1 *Constructing a minimal linear octree between two octants*

Given two octants, a and $b > a$, we wish to generate the minimal number of octants that span the region between a and b according to the Morton ordering. The algorithm (Algorithm 3) first calculates the nearest common ancestor of the octants a and b. This octant is split into its eight children. Out of these, only the octants that are either greater than a and lesser than b or ancestors of a are retained and the rest are discarded. The ancestors of either a or b are split again and we iterate until no further splits are necessary. This produces the minimal coarse complete linear octree (Figure 8(b)) between the two octants a and b (Figure 8(a)). This algorithm is based on the Properties 2 and 3 of the Morton ordering, which are listed in Appendix A.

2.2.1.2 *Constructing complete linear octrees from a partial set of octants*

In order to construct a complete linear octree from a partial set of octants (e.g. Figure 8(c)), the octants are initially sorted based on the Morton ordering. Algorithm 8 is subsequently used to remove overlaps, if any. Two additional octants are added to complete the domain (Figure 8(d)). The first is the coarsest ancestor of the least possible octant (the deepest first descendant of the root octant, Property 6), which does not overlap the least octant in the input. This is also the first child of the nearest common ancestor of the least octant in

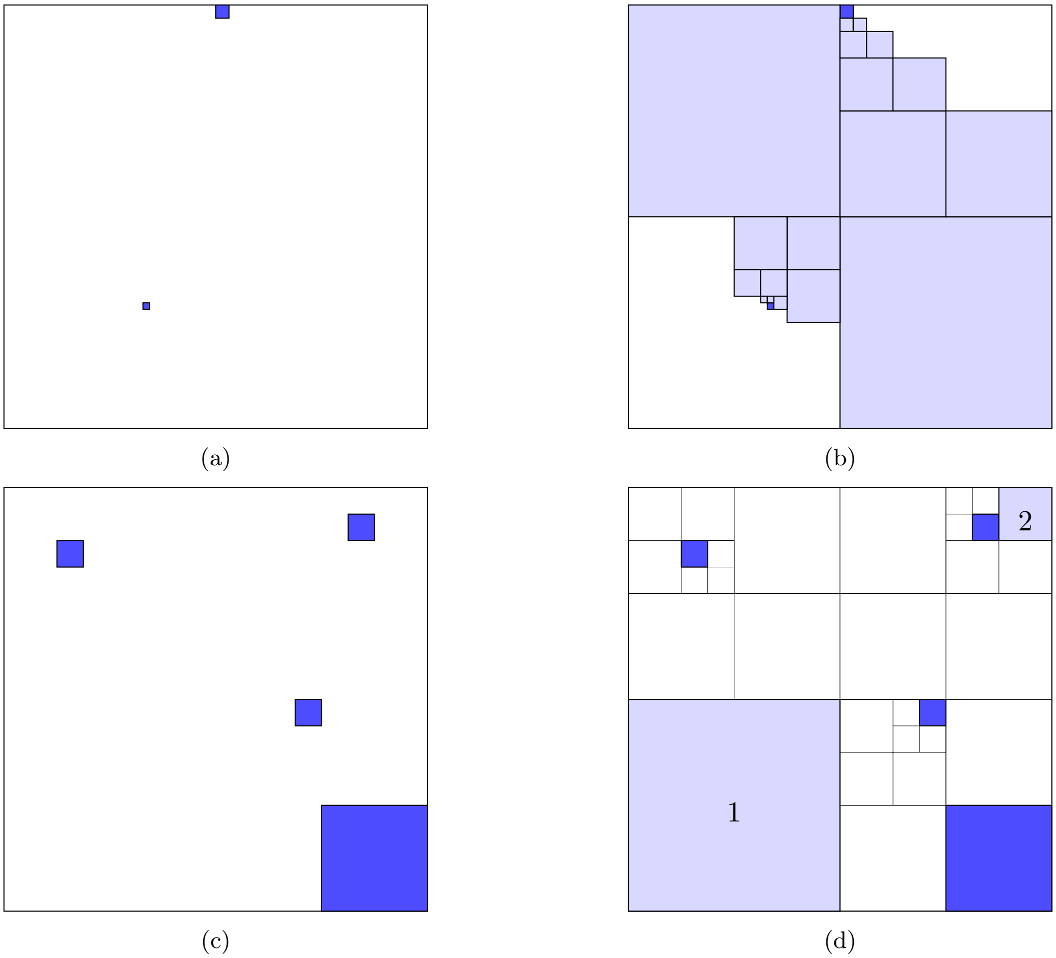


Figure 8: (b) The minimal number of octants between the cells given in (a). This is produced by using (a) as an input to `Algorithm 3`. (d) The coarsest possible complete linear quadtree containing all the cells in (c). This is produced by using (c) as an input to `Algorithm 4`. The figure also shows the two additional octants added to complete the domain. The first one is the coarsest ancestor of the least possible octant (the deepest first descendant of the root octant), which does not overlap the least octant in the input. This is also the first child of the nearest common ancestor of the least octant in the input and the deepest first decendant of root. The second is the coarsest ancestor of the greatest possible octant (the deepest last descendant of the root octant), which does not overlap the greatest octant in the input. This is also the last child of the nearest common ancestor of the greatest octant in the input and the deepest last decendant of root.

Algorithm 4 Constructing a complete linear octree from a partial (incomplete) set of octants (parallel) - `CompleteOctree`

Input: A distributed sorted list of octants, L.
Output: R, the complete linear octree.
Work: $\mathcal{O}(n \log n)$, where $n =$ len(R).
Storage: $\mathcal{O}(n)$, where $n =$ len(R).

```
RemoveDuplicates($L$)
$L \leftarrow$ Linearise($L$) ( Algorithm 8 )
Partition($L$) ( Algorithm 2 )
if rank = 0
   $L$.push_front($\mathcal{FC}$ ($\mathcal{A}_{finest}$ ($\mathcal{DFD}$( root ), $L[1]$)))
end if
if rank = $(n_p - 1)$
   $L$.push_back($\mathcal{LC}$ ($\mathcal{A}_{finest}$ ($\mathcal{DLD}$( root), $L$[len ($L$)])))
end if
if rank > 0
   Send($L[1]$ ,(rank$-1$) )
end if
if rank < $(n_p - 1)$
   $L$.push_back(Recieve() )
end if
for $i \leftarrow 1$ to (len($L$) $- 1$)
    $A \leftarrow$ CompleteRegion ($L[i], L[i+1]$) ( Algorithm 3 )
    $R \leftarrow R + L[i] + A$
end for
if rank = $(n_p - 1)$
   $R \leftarrow R + L$[len($L$)]
end if
```

the input and the deepest first decendant of root. The second is the coarsest ancestor of the greatest possible octant (the deepest last descendant of the root octant, Property 8), which does not overlap the greatest octant in the input. This is also the last child of the nearest common ancestor of the greatest octant in the input and the deepest last decendant of root. The octants are distributed across the processors to get a weight-based uniform load distribution. The local complete linear octree is subsequently generated by completing the region between every consecutive pair of octants as described in Section 2.2.1.1. Each processor is also responsible for completing the region between the first octant owned by that processor and the last octant owned by the previous processor, thus ensuring that a global complete linear octree is produced.

2.2.2 Constructing linear octrees in parallel

Octrees are usually constructed by using a top-down approach: starting with the root octant, cells are split iteratively based on some criteria, until no further splits are required. This is a simple and efficient sequential algorithm. However, it's parallel analogue is not so. We use the case of point datasets to discuss some shortcomings of a parallel top-down tree construction. Formally, the problem might be stated as: Construct a complete linear octree in parallel from a distributed set of points in a domain with the constraint that no octant should contain more than (N^p_{max}) number of points. Each processor can independently construct a tree using a top-down approach on its local set of points. Constructing a global linear octree requires a parallel merge. Merging however, is not straightforward.

1. Consider the case where the local number of points in some region on every processor was less than (N^p_{max}), and hence all the processors end up having the same level of coarseness in the region. However, the total number of points in that region could be more than (N^p_{max}) and hence the corresponding octant should be refined further.

2. In most applications, we would also like to associate a unique processor to each octant. Thus, duplicates across processors must be removed.

3. For linear octrees overlaps across processors must be resolved.

4. Since there might be overlaps and duplicates, not all the work done by the processors can be accounted as useful work. This is a subtle yet important point to consider while analyzing the algorithm for load-balancing.

Previous work [62, 129, 134, 140] on this problem has addressed these issues; however, all the existing algorithms involve many synchronization steps and thus suffer from a sizable overhead, resulting in suboptimal isogranular scalability. Instead, we propose a bottom-up approach for constructing octrees from points. The crux of the algorithm is to distribute the data across the processors in such a way that there is uniform load distribution across processors and the subsequent operations to build the octree can be performed by the processors independently, i.e., requiring no additional communication.

Algorithm 5 Constructing a complete linear octree from a distributed list of points (parallel) - `Points2Octree`

```
Input:     A distributed list of points, $L$ and a parameter, $(N^p_{max})$,
           which specifies the maximum number of points per octant.
Output:    Complete linear Octree, $B$.
Work:      $\mathcal{O}(n \log n)$, where $n =$ len($L$).
Storage:   $\mathcal{O}(n)$, where $n =$ len($L$).

$F \leftarrow$ [Octant($p, D_{max}$), $\forall p \in L$]
Sort($F$)
$B \leftarrow$ BlockPartition($F$) ( Algorithm 1 )
for each $b \in B$
    if NumberOfPoints($b$) $> N^p_{max}$
        $B \leftarrow B - b + \mathcal{C}(b)$
    end if
end for
```

First, all points are converted into octants at the maximum depth and then partitioned across the processors using the algorithm described in Section 2.2.1. This produces a contiguous set of coarse blocks (with their corresponding points) on each processor. The complete linear octree is generated by iterating through the blocks and by splitting them based on number of points per block.[12] This process is continued until no further splits are required. This procedure is summarized in Algorithm 5.

2.2.3 Balancing linear octrees in parallel

Balance refinement is the process of refining (subdividing) nodes in a complete linear octree, which fail to satisfy the balance constraint described in Section 2.1.2. The nodes are refined until all their descendants, which are created in the process of subdivision, satisfy the balance constraint. These subdivisions could in turn introduce new imbalances and so the process has to be repeated iteratively. The fact that an octant can affect octants not immediately adjacent to it is known as the *ripple effect*.

We use a two-stage balancing scheme: first we perform local balancing on each processor, and follow this up by balancing across the inter-processor boundaries. We first use the parallel bottom-up coarsening and partitioning algorithm (described in section 2.2.1) to

[12] Refer to the Appendix D on how to sample the points in order to construct the coarsest possible octree.

construct coarse blocks on each processor and to distribute the underlying octants. By construction, the domains covered by these blocks are disjoint and the union of these blocks covers the entire domain. We use the blocks as a means to minimize the number of octants that need to be split due to inter-processor violations of the 2:1 balancing rule.

2.2.3.1 Local balancing

There are two approaches for balancing a complete octree. In the first approach, every node constructs the coarsest possible neighbors satisfying the balance constraint, and subsequently duplicates and overlaps are removed [16]. We describe this approach in Algorithm 6. In an alternative approach, the nodes search for neighbors and resolve any violations of the balance constraint [127, 129]. The main advantage of the former approach is that constructing nodes is inexpensive, since it does not involve any searches. However, this could produce a lot of duplicates and overlaps making the linearizing operations expensive. Another disadvantage of this approach is that it cannot handle incomplete domains, and can only operate on subtrees. The advantage of the second approach is that the list of nodes is complete and linear at any stage in the algorithm. The drawback, however, is that searching for neighbors is an expensive operation. Our algorithm uses a hybrid approach: it keeps the number of duplicates and overlaps to a minimum and also reduces the search space thereby reducing the cost of the searching operation. The complete linear octree is first partitioned into coarse blocks using the algorithm described in Section 2.2.1. The descendants of any block, which are present in the fine octree, form a linear subtree with this block as its root. This block-subtree is first balanced using the approach described in Section 2.2.3.2; the size of this tree will be relatively small, and hence the number of duplicates and overlaps will be small too. After balancing all the blocks, the inter-block boundaries in each processor are balanced using a variant of the *ripple propagation* algorithm [129] described in Section 2.2.3.4. The performance improvements from using the combined approach are presented in Section 2.3.2.

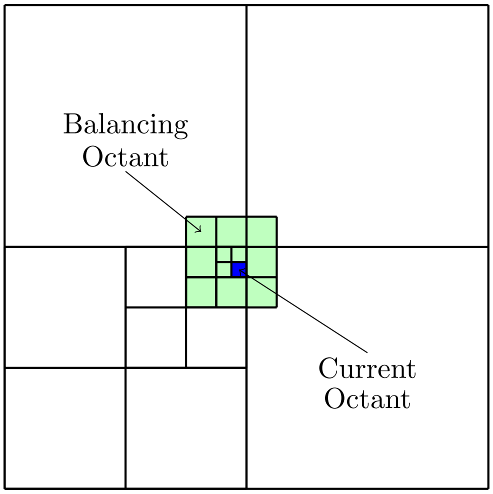


Figure 9: The minimal list of balancing quadrants for the current quadrant is shown. This list of quadrants is generated in one iteration of `Algorithm 6`.

2.2.3.2 Balancing a local block

In principle, Algorithm 6 can be used to construct a complete balanced subtree of this block for each octant in the initial unbalanced linear subtree. Note that these balanced subtrees may have substantial overlap. Hence, Algorithm 8 is used to remove these overlaps. Lemma 1 shows that this process of merging these different balanced subtrees results in a complete linear balanced subtree. However, this implementation would be inefficient due to the number of overlaps, which would in turn increase the storage costs and also make the subsequent operations of sorting and removing duplicates and overlaps more expensive. Instead, we interleave the two operations: constructing the different complete balanced subtrees and merging them. The overall scheme is described in Algorithm 7.

We note that a list of octants forms a balanced complete octree, if and only if for every octant all its neighbors are at the same level as this octant or one level finer or one level coarser. Hence, the coarsest possible octants in a complete octree that will be balanced against this octant are the siblings and the neighbors at the level of this octant's parent. Starting with the finest level and iterating over the levels up to but not including the level of the block, the coarsest possible (without violating the balance constraint) neighbors (Figure 9) of every octant at this level in the current tree (union of the initial unbalanced linear subtree and newly generated octants) are generated. After processing all the octants at any given level, the list of newly introduced coarse octants is merged with the previous list of octants at this level and duplicate octants are removed. The newly created octants are included while working on subsequent levels. Algorithm 8 still needs to be used in the end to remove overlaps, but the working size is much smaller now compared to the earlier case (Algorithm 6). To avoid redundant work and to reduce the number of duplicates to be removed in the end, we ensure that no two elements in the working list at any given level are siblings of one another. This can be done in a linear pass on the working list for that level as shown in Algorithm 7.

Lemma 1 *Let T_1 and T_2 be two complete balanced linear octrees with n_1 and n_2 number of potential ancestors respectively, then*

$$T_3 = (T_1 \cup T_2) - \left(\sum_{i=1}^{n_1} \{\mathcal{A}(T_1[i])\}\right) - \left(\sum_{j=1}^{n_2} \{\mathcal{A}(T_2[j])\}\right)$$

is a complete linear balanced octree.

Proof. $T_4 = (T_1 \cup T_2)$ is a complete octree. Now,

$$\left(\left(\sum_{i=1}^{n_1} \{\mathcal{A}(T_1[i])\}\right) + \left(\sum_{j=1}^{n_2} \{\mathcal{A}(T_2[j])\}\right)\right) = \left(\sum_{k=1}^{n_3} \{\mathcal{A}(T_4[k])\}\right)$$

So, $T_3 = \left(T_4 - \left(\sum_{k=1}^{n_3} \{\mathcal{A}(T_4[k])\}\right)\right)$ is a complete linear octree.
Now, suppose that a node $N \in T_3$ has a neighbor $K \in T_3$ such that $\mathcal{L}(K) \geq (\mathcal{L}(N) + 2)$. It is obvious that exactly one of N and K must be present in T_1 and the other must be present in T_2. Without loss of generality, assume that $N \in T_1$ and $K \in T_2$. Since T_2 is complete, there exists at least one neighbor of K,$L \in T_2$, which overlaps N. Also, since T_2 is balanced $\mathcal{L}(L) = \mathcal{L}(K)$ or $\mathcal{L}(L) = (\mathcal{L}(K) - 1)$ or $\mathcal{L}(L) = (\mathcal{L}(K) + 1)$. So, $\mathcal{L}(L) \geq (\mathcal{L}(N) + 1)$. Since L overlaps N and since $\mathcal{L}(L) \geq (\mathcal{L}(N) + 1)$, $L \in \{\mathcal{D}(N)\}$. Hence, $N \notin T_3$. This contradicts the initial assumption. Therefore, T_3 is also balanced. □

2.2.3.3 Searching for neighbors

A leaf needs to be refined if and only if the level of one of its neighbors is at least 2 levels finer than its own. In terms of a search this presents us two options: search for coarser

Algorithm 6 Constructing a complete balanced subtree of an octant, given one of its descendants (sequential)

Input: An octant, N, and one of its descendants, L.
Output: Complete balanced subtree, R.
Work: $\mathcal{O}(n \log n)$, where $n =$ len(R).
Storage: $\mathcal{O}(n)$, where $n =$ len(R).

```
$W \leftarrow L$, $T \leftarrow \emptyset$, $R \leftarrow \emptyset$
for $l \leftarrow \mathcal{L}(L)$ to $(\mathcal{L}(N) + 1)$
    for each $w \in W$
        $R \leftarrow R + w + \mathcal{S}(w)$
        $T \leftarrow T + \{\mathcal{N}(\mathcal{P}(w), l - 1) \cap \{\mathcal{D}(N)\}\}$
    end for
    $W \leftarrow T$, $T \leftarrow \emptyset$
end for
Sort($R$)
RemoveDuplicates($R$)
$R \leftarrow$ Linearise($R$) ( Algorithm 8 )
```

Algorithm 7 Balancing a local block (sequential) - BalanceSubtree

Input: An octant, N, and a partial list of its descendants, L.
Output: Complete balanced subtree, R.
Work: $\mathcal{O}(n \log n)$, where $n =$ len(R).
Storage: $\mathcal{O}(n)$, where $n =$ len(R).

```
$W \leftarrow L$, $P \leftarrow \emptyset$, $R \leftarrow \emptyset$
for $l \leftarrow D_{max}$ to $(\mathcal{L}(N) + 1)$
    $Q \leftarrow \{x \in W \mid \mathcal{L}(x) = l\}$
    Sort($Q$)
    $T \leftarrow \{x \in Q \mid \mathcal{S}(x) \notin T\}$
    for each $t \in T$
        $R \leftarrow R + t + \mathcal{S}(t)$
        $P \leftarrow P + \{\mathcal{N}(\mathcal{P}(t), l - 1) \cap \{\mathcal{D}(N)\}\}$
    end for
    $P \leftarrow P + \{x \in W \mid \mathcal{L}(x) = l - 1\}$
    $W \leftarrow \{x \in W | \mathcal{L}(x) \neq l - 1\}$
    RemoveDuplicates($P$)
    $W \leftarrow W + P$, $P \leftarrow \emptyset$
end for
Sort($R$)
$R \leftarrow$ Linearise($R$) ( Algorithm 8 )
```

Algorithm 8 Removing overlaps from a sorted list of octants (sequential) - Linearise

```
Input:      A sorted list of octants, $W$.
Output:     $R$, an octree with no overlaps.
Work:       $\mathcal{O}(n)$, where $n =$ len($W$).
Storage:    $\mathcal{O}(n)$, where $n =$ len($W$).

for $i \leftarrow 1$ to (len($W$) $- 1$)
    if $(W[i] \notin \{\mathcal{A}(W[i+1])\})$
        $R \leftarrow R + W[i]$
    end if
end for
$R \leftarrow R + W[$len($W$)$]$
```

neighbors or search for finer neighbors. It is much easier to search for coarser neighbors than it is to search for finer neighbors. If we consider the 2D case, only 3 neighbors coarser than the current cell need to be searched for. However, the number of potential neighbors finer than the cell is extremely large, (in 2D it is $2 \cdot 2^{D_{max}-l} + 3$, where l is the level of the current quadrant), and therefore not practical to search. In addition, the search strategy depends on the way the octree is stored; the pointer-based approach is more popular [16, 127], but has the overhead that it has to be rebuilt every time octants are communicated across processors. In the proposed approach the octree is stored as a linear octree in which the octants are sorted globally in the ascending Morton order, allowing us to search in $\mathcal{O}(\log n)$.

In order to find neighbors coarser than the current cell, we use the approach illustrated in Figure 10. First, the finest cell at the far corner (marked as "Search Corner" in Figure 10) is determined. This is the corner that this octant shares with its parent. This is also the corner diagonally opposite to the corner common to all the siblings of the current cell.[13] The neighbors (at the finest level) of this cell (N) are then selected and used as the search keys. These are denoted by $\mathcal{N}^s(N, D_{max})$. The maximum lower bound[14] for the given search key is determined by searching within the complete linear octree. In a complete linear octree, the maximum lower bound of a search key returns its finest ancestor. If the search result is at a level finer than or equal to the current cell then it is guaranteed that

[13] We do not need to search in the direction of the siblings.

[14] The greatest cell lesser than or equal to the search key is referred to as its maximum lower bound.

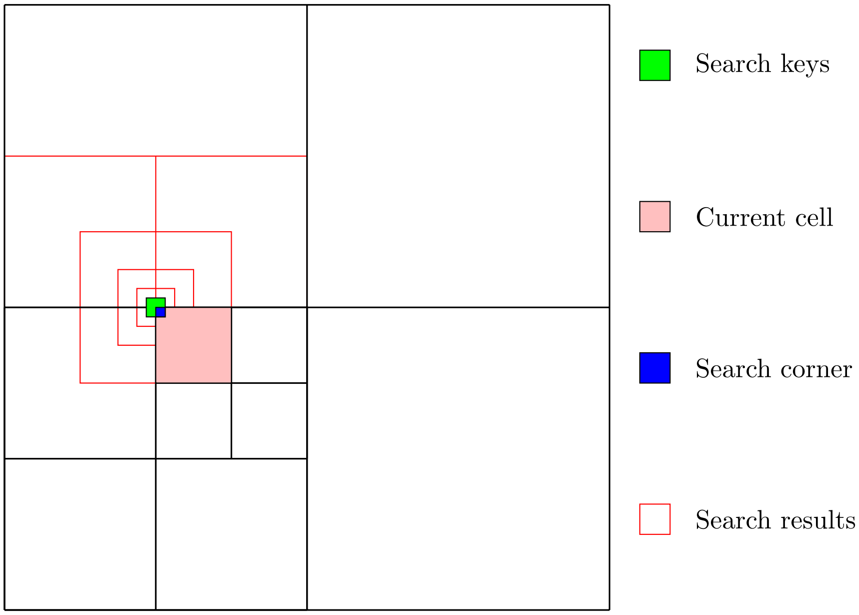


Figure 10: To find neighbors coarser than the current cell, we first select the finest cell at the far corner. The far corner is the one that is not shared with any of the current cell's siblings. The neighbors of this corner cell are determined and used as the search keys. The search returns the greatest cell lesser than or equal to the search key. The possible candidates in a complete linear quadtree, as shown, are ancestors of the search key.

no coarser neighbor can exist in that direction. This idea can be extended to incomplete linear octrees (including multiply connected domains). In this case, the result of a search is ignored if it is not an ancestor of the search key.

2.2.3.4 Ripple propagation

A variant (Algorithm 9) of the *prioritized ripple propagation* algorithm first proposed by Tu et al. [127], modified to work with linear octrees, is used to balance the boundary leaves. The algorithm selects all leaves at a given level (successively decreasing levels starting with the finest), and searches for neighbors coarser than itself. A list of balancing descendants[15] for neighbors that violate the balance condition are stored. At the end of each level, any octant that violated the balance condition is replaced by a complete linear subtree. This subtree can be obtained either by using the sequential version of Algorithm 4 or by using

[15] Balancing descendants are the minimum number of descendants that will balance against the octant that performed the search.

Algorithm 10, which is a variant of Algorithm 7. Both the algorithms perform equally well.[16]

One difference with earlier versions of the ripple propagation algorithm is that our version works with incomplete domains. In addition, earlier approaches [16, 127, 129] have used pointer-based representations of the local octree, which incurs the additional cost of constructing the pointer-based tree from the linear representation and also increases the memory footprint of the octree as 9 additional pointers[17] are required per octant. The work and storage costs incurred for balancing using the proposed algorithm to construct n balanced octants are $\mathcal{O}(n \log n)$ and $\mathcal{O}(n)$, respectively. This is true irrespective of the domain, including domains that are not simply connected.

2.2.3.5 Insulation against the ripple-effect

An interesting property of complete linear octrees is that a boundary octant cannot be finer than its internal neighbors[18] (Figure 11(a)) [127]. So, if a node (at any level) is internally balanced then to balance it with all its neighboring domains, it is sufficient to appropriately refine the internal boundary leaves.[19] The interior leaves need not be refined any further. Since the interior leaves are also balanced against all their neighbors, they will not force any other octant to split. Hence, interior octants do not participate in the remaining stages of balancing.

Definition 2 *For any octant, N, in the octree, we refer to the union of the domains occupied by its potential neighbor's at the same level as N ($\mathcal{N}(N, \mathcal{L}(N))$) as the insulation layer around octant N. This will be denoted by $\mathcal{I}(N)$.*

Observe that the phenomenon with interior octants described above is only an example of a more general property: No octant outside the insulation layer (Definition 2) around

[16] We indicate which algorithms are parallel and which are sequential. In our notation the sequential algorithms are sometimes invoked with a distributed object: it is implied that the input is the local instance of the distributed object.

[17] One pointer to the parent and eight pointers to its children.

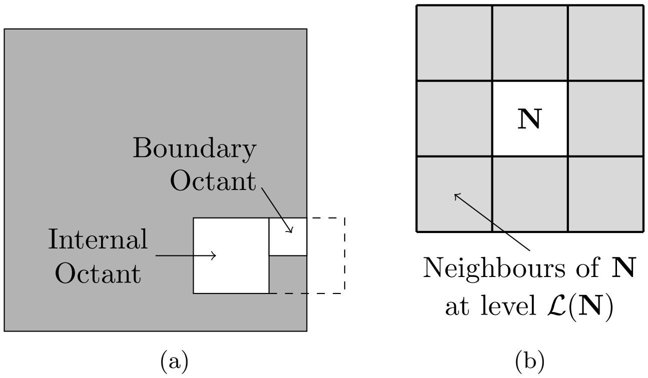
[18] A neighbor of a boundary octant that does not touch the boundary is referred to as an internal neighbor of the boundary octant.

[19] We refer to the descendants of a node that touch its boundary from the inside as its internal boundary leaves.

Algorithm 9 RIPPLE PROPAGATION ON INCOMPLETE DOMAINS (SEQUENTIAL) - `Ripple`

Input: L, a sorted incomplete linear octree.
Output: W, a balanced incomplete linear octree.
Work: $\mathcal{O}(n \log n)$, where $n =$ len(L).
Storage: $\mathcal{O}(n)$, where $n =$ len(L).

```
W ← L
for l ← D_max to 3
    T, R ← ∅
    for each w ∈ W
        if 𝓛(w) = l
            K ← search_keys(w) ( Section 2.2.3.3 )
            (B, J) ← maximum_lower_bound (K, W)
                     (J is the index of B in W)
            for each (b, j) ∈ (B, J) | (∃ k ∈ K | b ∈ {𝒜(k)})
                T[j] ← T[j] + ({𝒩^s (w, (l − 1))} ∩ {𝒟(b)})
            end for
        end if
    end for
    for i ← 1 to len(W)
        if T[i] ≠ ∅
            R ← R+ CompleteSubtree(W[i], T[i]) ( Algorithm 10 )
        else
            R ← R + W[i]
        end if
    end for
    W ← R
end for
```



(a)

(b)

Figure 11: (a) A boundary octant cannot be finer than its internal neighbors, and (b) an illustration of an insulation layer around octant **N**. No octant outside this layer of insulation can force a split on **N**.

Algorithm 10 Completing a local block (sequential) - `CompleteSubtree`

Input: An octant, N, and a partial list of its descendants, L.
Output: Complete subtree, R.
Work: $\mathcal{O}(n \log n)$, where $n =$ len(R).
Storage: $\mathcal{O}(n)$, where $n =$ len(R).

```
$W \leftarrow L$
for $l \leftarrow D_{max}$ to $\mathcal{L}(N) + 1$
    $Q \leftarrow \{x \in W \mid \mathcal{L}(x) = l\}$
    Sort($Q$)
    $T \leftarrow \{x \in Q \mid \mathcal{S}(x) \notin T\}$
    for each $t \in T$
        $R \leftarrow R + t + \mathcal{S}(t)$
        $P \leftarrow P + \mathcal{S}(\mathcal{P}(t))$
    end for
    $P \leftarrow P + \{x \in W \mid \mathcal{L}(x) = l - 1\}$
    $W \leftarrow \{x \in W \mid \mathcal{L}(x) \neq l - 1\}$
    RemoveDuplicates($P$)
    $W \leftarrow W + P$, $P \leftarrow \emptyset$
end for
Sort($R$)
$R \leftarrow$ Linearise($R$) ( Algorithm 8 )
```

octant N can force N to split (Figure 11(b)). This property allows us to decouple the problem of balancing and allows us to work on only a subset of nodes in the octree and yet ensure that the entire octree is balanced.

2.2.3.6 Balancing inter-processor boundaries

After the intra-processor, and inter-block boundaries are balanced, the inter-processor boundaries need to be balanced. Unlike the internal leaves (Section 2.2.3.5), the octants on the boundary do not have any insulation against the ripple-effect. Moreover, a ripple can propagate across multiple processors. Most approaches to perform this balance have been based on extensions of the sequential ripple algorithm to a parallel case by performing parallel searches. Although this approach works well for small problems on a small number of processors, it shows suboptimal isogranular scalability [129]. The main reason is iterative communication. Although there are many examples of scalable parallel algorithms that involve iterative communication, they overlap communication with computation to reduce

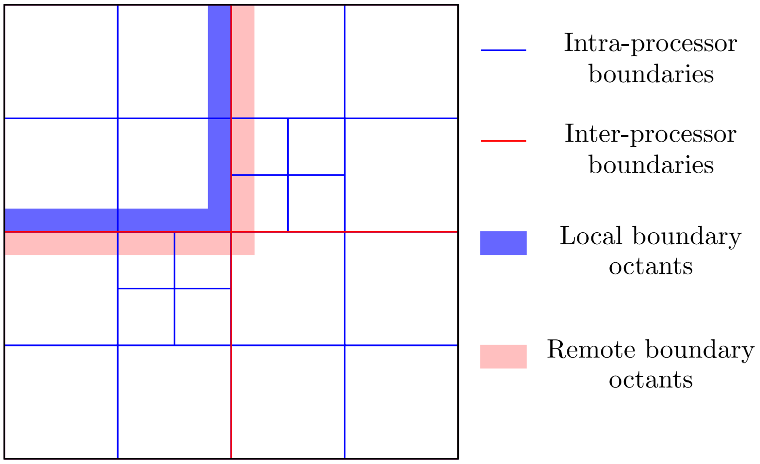


Figure 12: A coarse quadtree illustrating inter and intra processor boundaries. First, every processor balances each of its local blocks. Then, each processor balances the cells on its intra-processor boundaries. The octants that lie on inter-processor boundaries are then communicated to the respective processors and each processor balances the combined list of local and remote octants.

the overhead associated with communication [51, 112]. Currently, there is no method that overlap communication with computation for the balancing problem. Thus, any algorithm that uses iterative parallel searches for balancing octrees will have high synchronization costs.

In order to avoid parallel searches, the problem of balancing is decoupled. In other words, each processor works independently without iterative communication. To achieve this, two properties are used: (1) the only octants that need to be refined after the local balancing stage are the ones whose insulation layer is not contained entirely within the same processor. We will refer to them as the unstable octants. and (2) an artificial insulation layer (Property **??**) for these octants can be constructed with little communication overhead (Section 2.2.3.7).

Note that although it is sufficient to build an insulation layer for octants that truly touch the inter-processor boundary, it is non-trivial to identify such octants. Moreover, even if it was easy to identify the true inter-processor boundary octants all unstable octants must participate in subsequent balancing as well. Hence, the insulation layer is built for all unstable octants as they can be identified easily. Since most of the unstable octants

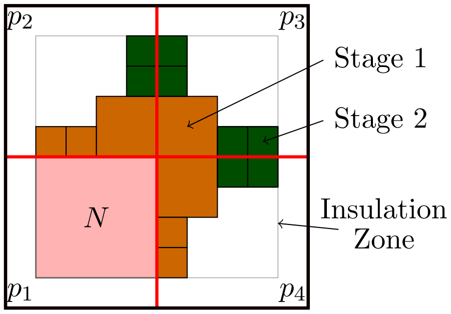


Figure 13: Communication for inter-processor balancing is done in two stages: First, every octant on the inter-processor boundary (Stage 1) is communicated to processors that overlap with its insulation layer. Next, all the local inter-processor boundary octants that lie in the insulation layer of a remote octant (N) received from another processor are communicated to that processor (Stage 2).

do touch the inter-processor boundaries, we will simply refer to them as inter-processor boundary octants in the following sections.

The construction of the insulation layer for the inter-processor boundary octants is done in two stages (Figure 13): First, every local octant on the inter-processor boundary (Figure 12) is communicated to processors that overlap with its insulation layer. These processors can be determined by comparing the local boundary octants against the global coarse blocks. In the second stage of communication, all the local inter-processor boundary octants that overlap with the insulation layer of a remote octant received from another processor are communicated to that processor. Octants that were communicated in the first stage are not communicated to the same processor again. For simplicity, Algorithm 11 only describes a naïve implementation for determining the octants that need to be communicated in this stage. However, this can be performed much more efficiently using the results of Lemma 2 and Lemma 3. After this two-stage communication, each processor balances the union of the local and remote boundary octants using the ripple propagation based method (Section 2.2.3.4). At the end only the octants spanning the original domain spanned by the processors are retained. Although there is some redundancy in the work, it is compensated by the fact that we avoid iterative communications and also the communication message size is smaller than any alternative parallel search-based approach. Section 2.2.3.7 gives a detailed analysis of the communication cost involved.

Lemma 2 *If octants a and $b > a$ do not overlap, then there can be no octant $c > b$ that overlaps a.*

Proof. If a and c overlap, then either $a \in \{\mathcal{A}(c)\}$ or $a \in \{\mathcal{D}(c)\}$. Since $c > a$, the latter is a direct violation of Property 3 and hence is impossible. Hence, assume that $c \in \{\mathcal{D}(a)\}$. By Property 8, $c \leq \mathcal{DLD}(a)$. Property 9 would then imply that $b \in \{\mathcal{D}(a)\}$. Property 4 would then imply that a and b must overlap. Since, this is not true our initial assumption must be wrong. Hence, a and c can not overlap. □

Lemma 3 *Let N be an inter-processor boundary octant belonging to processor q. If the $\mathcal{I}(N)$ is contained entirely within processors q and p, then the inter-processor boundary octants on processor p that overlap with $\mathcal{I}(N)$ and that were not communicated to q in the first stage, will not force a split on N.*

Proof. Note that at this stage both p and q are internally balanced. Thus, N will be forced to split if and only if there is a true inter-processor boundary octant, a, on p touching an octant, b, on q such that $\mathcal{L}(a) > (\mathcal{L}(b)+1)$ and when b is split it starts a cascade of splits on octants in q that in turn force N to split. Since every true inter-processor boundary octant is sent to all its adjacent processors, a must have been sent to q during the first stage of communication. □

Algorithm 11 gives the pseudo-code for the overall parallel balancing.

2.2.3.7 Communication costs for parallel balancing

Although not all unstable octants are true inter-processor boundaries, it is easier to visualize and understand the arguments presented in this section if this subtle point is ignored. Moreover, since we only compare the communication costs associated with the two approaches (upfront communication versus iterative communication) and since the majority of unstable octants are true inter-processor boundary octants it is not too restrictive to assume that all unstable octants are true inter-processor boundary octants.

Let us assume that prior to parallel balancing there are a total of N octants in the global octree. The octants that lie on the inter-processor boundary can be classified based

Algorithm 11 BALANCING COMPLETE LINEAR OCTREES (PARALLEL)

```
Input:      A distributed sorted complete linear octree, $L$.
Output:     A distributed complete balanced linear octree, $R$.
Work:       $\mathcal{O}(n \log n)$, where $n =$ len($L$).
Storage:    $\mathcal{O}(n)$, where $n =$ len($L$).
Time:       Refer to Section 2.2.3.7.

1.  $B \leftarrow$ BlockPartition($L$) ( Algorithm 1 )
2.  $C \leftarrow \emptyset$
3.  for each $b \in B$
4.      $C \leftarrow C +$ BalanceSubtree($b$, $\{\{\mathcal{D}(b)\} \cap L\}$) ( Algorithm 7 )
5.  end for
6.  $D \leftarrow \{x \in C \mid \exists\ z \in \{\mathcal{I}(x)\} \mid \mathcal{B}(z) \neq \mathcal{B}(x)\}$
      ( intra-processor boundary octants )
7.  $S \leftarrow$ Ripple($D$) ( Algorithm 9 )
8.  $F \leftarrow (C - D) \cup S$
9.  $G \leftarrow \{x \in S \mid \exists\ z \in \{\mathcal{I}(x)\} \mid$ rank($z$) $\neq$ rank($x$) $\}$
      ( inter-processor boundary octants )
10. for each $g \in G$
11.     for each $b \in B_{global} - B$
12.         if $\{b \cap \mathcal{I}(g)\} \neq \emptyset$
13.             Send($g$, rank($b$))
14.         end if
15.     end for
16. end for
17. $T \leftarrow$ Receive()
18. for each $g \in G$
19.     for each $t \in T$
20.         if $\{g \cap \mathcal{I}(t)\} \neq \emptyset$
21.             if $g$ was not sent to rank($t$) in Step 10
22.                 Send($g$, rank($t$))
23.             end if
24.         end if
25.     end for
26. end for
27. $K \leftarrow$ Receive()
28. $H \leftarrow$ Ripple($G \cup T \cup K$)
29. $R \leftarrow \{x \in \{H \cup F\} \mid \{B \cap \{x, \{\mathcal{A}(x)\}\}\} \neq \emptyset\}$
30. $R \leftarrow$ Linearise($R$) ( Algorithm 8 )
```

on the *degree of the face*[20] that they share with the inter-processor boundary. We use N_k to represent the number of octants that touch any m-dimensional face ($m \in [0, k]$) of the inter-processor boundary.

Note that all vertex boundary octants are also edge and face boundaries and that all edge boundary octants are also face boundary octants. Therefore we have, $N \geq N_2 \geq N_1 \geq N_0$, and for $N \gg n_p$, we have $N \gg N_2 \gg N_1 \gg N_0$.

Although it is theoretically possible that an octant is larger than the entire domain controlled by some processors, it is unlikely for dense octrees. Thus, ignoring such cases we can show that the total number of octants of a d-tree that need to be communicated in the first stage of the proposed approach is given by

$$N_u = \sum_{k=1}^{d} 2^{d-k} N_{k-1}. \tag{1}$$

Consider the example shown in Figure 14. The domain on the left is partitioned into two regions, and in this case all boundary octants need to be transmitted to exactly one other processor. The addition of the additional boundary, in the figure on the right, does not affect most boundary nodes, except for the boundary octants that share a corner, i.e., a 0-dimensional face with the inter processor boundaries. These octants need to be sent to an additional 2 processors, and that is the reason we have a factor of 2^{d-k} in Equation 1. For the case of octrees, additional communication is incurred because of edge boundaries as well as vertex boundaries. Edge boundary octants need to be communicated to 2 additional processors whereas the vertex boundary octants need to be communicated to 4 additional processors (7 processors in all).

Now, we analyze the cost associated with the second communication step in our algorithm. Consider the example shown in Figure 13. Note that all the immediate neighbors of the octant under consideration (octant on processor 1 in the figure), were communicated during the first stage. The octants that lie in the insulation zone of this octant and that were not communicated in the first stage are those that lie in a direction normal to the

[20] *A corner is a 0-degree face, an edge is a 1-degree face and a face is a 2-degree face.*

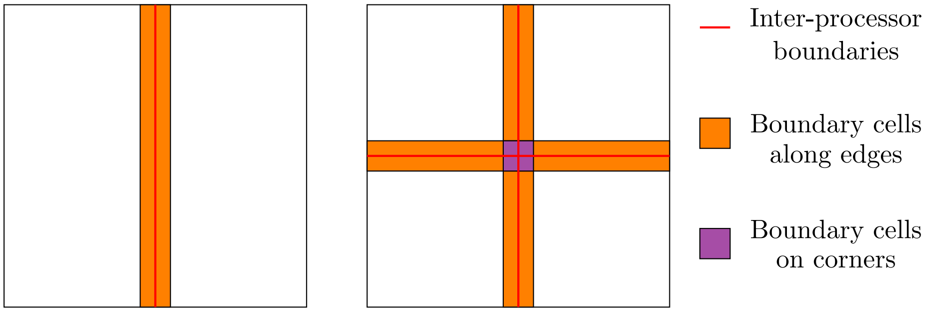

Inter-processor boundaries
Boundary cells along edges
Boundary cells on corners

Figure 14: Cells that lie on the inter-processor boundaries. The figure on the left shows an inter-processor boundary involving 2 processors and the figure on the right shows an inter-processor boundary involving 4 processors.

inter-processor boundary. However, most octants that lie in a direction normal to the inter-processor boundary are internal octants on other processors. As shown in Figure 13, the only octants that lie in a direction normal to one inter-processor boundary and are also tangential to another inter-processor boundary are the ones that lie in the shadow of some edge or corner boundary octant. Therefore, we only communicate $\mathcal{O}(N_1 + N_0)$ octants during this stage. Since $N \gg n_p$ and $N_2 \gg N_1 \gg N_0$ for most practical applications, the cost for this communication step can be ignored.

The minimum number of search keys that need to be communicated in a search-based approach is given by

$$N_s = \sum_{k=1}^{d} 2^{k-1} N_{k-1}. \tag{2}$$

Again considering the example shown in Figure 14, each boundary octant in the figure shown on the left, generates 3 search keys, out of which one lies on the same processor. The other two need to be communicated to the other processor. The addition of the extra boundary, in the figure on the right, does not affect most boundary nodes, except for the boundary octants that share a corner, i.e., a 0-dimensional face with the inter processor boundaries. These octants need to be sent to an additional processor, and that is the reason we have a factor of 2^{k-1} in Equation 2. It is important to observe the difference between the communication estimates for upfront communication, 1, with that of the search-based approach, 2. For large octrees,

$$N_u \approx N_2,$$

while,

$$N_s \approx 4N_2.$$

Note, that in arriving at the communication estimate for the search-based approaches, we have not accounted for the additional octants created during the inter-processor balancing. In addition, iterative search-based approaches are further affected by communication lag and synchronization. Our approach in contrast requires no subsequent communication.

In conclusion, the communication cost involved in the proposed approach is lower than that of search-based approaches.[21]

2.3 Results

The performance of the described algorithms is evaluated by a number of numerical experiments, including fixed-size and isogranular scalabilty analysis. The algorithms were implemented in C++ using the MPI library. A variant of the sample sort algorithm was used to sort the points and the octants, which incorporates a parallel bitonic sort to sort the sample elements as suggested in [51]. `PETSc` [10, 9] was used for profiling the code. All tests were performed on the Pittsburgh Supercomputing Center's TCS-1 terascale computing HP AlphaServer Cluster comprising of 750 SMP ES45 nodes. Each node is equipped with four Alpha EV-68 processors at 1 GHz and 4 GB of memory. The peak performance is approximately 6 Tflops, and the peak performance for the top-500 LINPACK benchmark is approximately 4 Tflops. The nodes are connected by a Quadrics interconnect, which delivers over 500 MB/s of message-passing bandwidth per node and has a bisection bandwidth of 187 GB/s. In our tests, we have used 4 processors per node wherever possible.

We present results from an experiment that we conducted to highlight the advantage of using the proposed two-stage method for intra-processor balancing. Also, we present fixed-size and isogranular scalability analysis results.

[21] We are assuming that both the approaches use the same partitioning of octants.

Table 3: Input and output sizes for the construction and balancing algorithms for the scalability experiments on Gaussian, Log-Normal, and Regular point distributions. The output of the construction algorithm is the input for the balancing algorithm. All the octrees were generated using the same parameters: $D_{max} = 30$ and $N^p_{max} = 1$; differences in the number and distributions of the input points result in different octrees for each case. The maximum level of the leaves for each case is listed. Note that none of the leaves produced were at the maximum permissible depth (D_{max}). This depends only on the input distribution. Regular point distributions are inherently balanced, and so we report the number of octants only once.

Problem size	Gaussian				Log-Normal				Regular		
	Points	Balancing		Max. Level ($\mathcal{L}^*$)	Points	Balancing		$\mathcal{L}^*$	Points	Leaves	$\mathcal{L}^*$
		Leaves before	Leaves after			Leaves before	Leaves after				
1M	180K	607K	0.99M	14	180K	607K	0.99M	13	0.41M	0.99M	7
2M	361K	1.2M	2M	15	361K	1.2M	2M	14	2M	2M	7
4M	720K	2.4M	3.9M	14	720K	2.4M	3.9M	15	2.4M	4.06M	8
8M	1.5M	4.9M	8.0M	16	1.5M	4.9M	8.1M	16	3.24M	7.96M	8
16M	2.9M	9.7M	16M	16	2.9M	9.7M	16M	16	16.8M	16.8M	8
32M	5.8M	19.6M	31.9M	17	5.8M	19.6M	31.8M	17	19.3M	32.5M	9
64M	11.7M	39.3M	64.4M	18	11.7M	39.3M	64.7M	17	25.9M	63.7M	9
128M	23.5M	79.3M	0.13B	19	23.5M	79.4M	0.13B	19	0.13B	0.13B	9
256M	47M	0.16B	0.26B	19	47M	0.16B	0.26B	19	0.15B	0.26B	10
512M	94M	0.32B	0.52B	20	94M	0.32B	0.52B	20	0.17B	0.34B	10
1B	0.16B	0.55B	0.91B	21	0.16B	0.55B	0.91B	20	1.07B	1.07B	10

2.3.1 Test data

Data of different sizes were generated for three different spatial distributions of points; Gaussian, Log-normal and Regular. The Regular distribution corresponds to a set of points distributed on a Cartesian grid. Datasets of increasing sizes were generated for all three distributions so that they result in balanced octrees with octants ranging from 10^6(1M) to 10^9(1B). All of the experiments were carried out using the same parameters: $D_{max} = 30$ and $N^p_{max} = 1$. Only the number and distribution of points were varied to produce the various octrees. The fixed size scalability analysis was performed by selecting the 1M, 32M, and 128M Gaussian point distributions to represent small, medium and large problems. We provide the input and output sizes for the construction and balancing algorithms in Table 3. The output of the construction algorithm is the input for the balancing algorithm.

2.3.2 Comparison between different strategies for local balancing

In order to assess the advantages of using a two-stage approach for local balancing over existing methods, we compared the runtimes on different problem sizes. Since the comparison was for different local-balancing strategies, it does not involve any communication and hence was evaluated on a shared memory machine. We compared our two-stage approach, discussed in Section 2.2.3.1, with two other approaches; the first approach is the prioritized ripple propagation idea applied on the entire local domain [129], and the second approach is to use ripple propagation in 2 stages, where the local domain is first split into coarser blocks[22] and ripple propagation is applied first to each local block and then repeated on the boundaries of all local blocks. Fixed size scalability analysis was performed to compare the above mentioned three approaches with problem sizes of 1, 4, 8, and 16 million octants. The results are shown in in Figure 15. All three approaches demonstrate good fixed size scalability, but the proposed two-stage approach has a lower absolute runtime.

2.3.3 Scalability analysis

In this Section, we provide experimental evidence of the good scalability of our algorithms. We present both fixed-size and isogranular scalability analysis. Fixed size scalability was performed for different problem sizes to compute the speedup when the problem size is kept constant and the number of processors is increased. Isogranular scalability analysis is performed by tracking the execution time while increasing the problem size and the number of processors proportionately. By maintaining the problem size per processor (relatively) constant as the number of processors is increased, we can identify communication problems related to the size and frequency of the messages as well as global reductions and problems with algorithmic scalability.

One of the important components in our algorithms is the sample sort routine, which has a complexity of $\mathcal{O}(\frac{N}{n_p}\log\frac{N}{n_p} + n_p^2 \log n_p)$ if the samples are sorted using a serial sort. This causes problems when $\mathcal{O}(N) < \mathcal{O}(n_p^3)$ as the serial sort begins to dominate and results in poor scalability. For example, at $n_p = 1024$ we would require $\frac{N}{n_p} > 10^6$ to obtain

[22] The same partitioning strategy as used in our two-stage algorithm was used to obtain the coarser blocks.

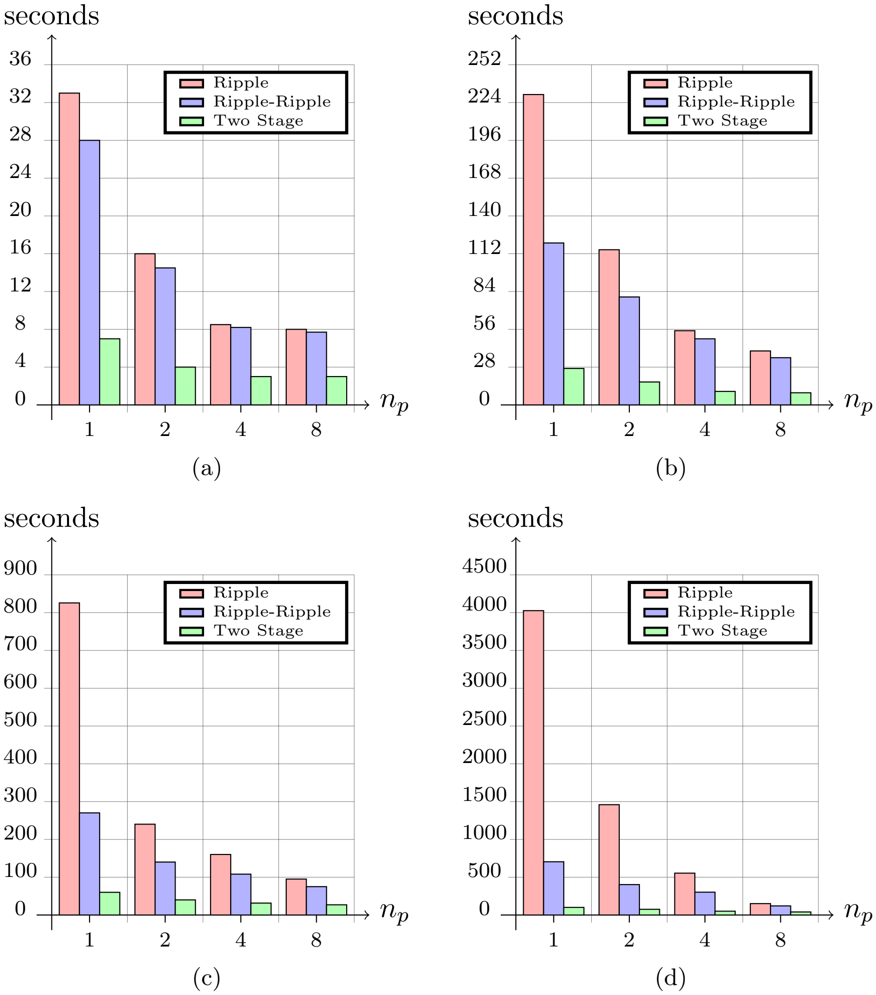


Figure 15: Comparison of three different approaches for balancing linear octrees `(a)` for a Gaussian distribution of `1M` octants, `(b)` for a Gaussian distribution of `4M` octants, `(c)` for a Gaussian distribution of `8M` octants, and `(d)` for a Gaussian distribution of `16M` octants.

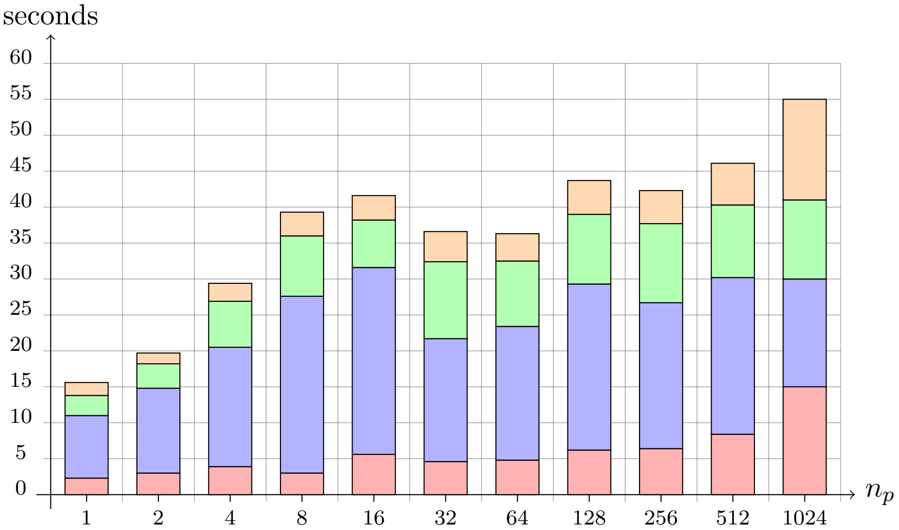
good scalability. This presents some problems as it becomes difficult to fit arbitrarily large problems on a single processor. A solution, previously proposed in [51], is to sort the samples using the parallel bitonic sort; this reduces the complexity of the overall parallel sort function to $\mathcal{O}(\frac{N}{n_p}\log\frac{N}{n_p}+n_p\log n_p)$. Our implementation uses this approach.

Isogranular scalability analysis was performed for all three distributions with an output size of roughly 1M octants per processor, for processor counts ranging from 1 to 1024. Wall-clock timings, speedup, and efficiency for the isogranular analysis for the three distributions are shown in Figures 16, 17, and 18.

Since the regularly spaced distribution is inherently balanced, the input point sizes were much greater for this case than those for Gaussian and Log-normal distributions. Both the Gaussian and Log-normal distributions are imbalanced; and in Table 3, we can see that, on average, the number of unbalanced octants is three times the number of input points and the number of octants doubles after balancing. For the regularly spaced distribution, we observe that in some cases the number of octants is the same as the number of input points (2M, 16M, 128M and 1B). These are special cases where the resulting grid is a perfect regular grid. Thus, while both the input and output grain sizes remain almost constant for the Gaussian and LogNormal distributions, only the output grain size remains constant for the Regular distribution. Hence, the trend for the regular distribution is a little different from those for the Gaussian and LogNormal distributions.

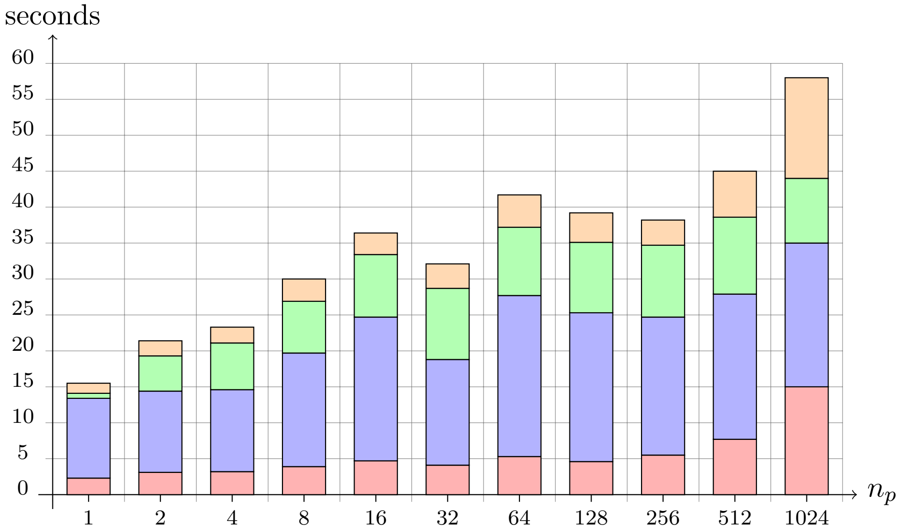
The plots demonstrate the good isogranular scalability of the algorithm. We achieve near optimal isogranular scalability for all three distributions (50s per 10^6 octants per processor for the Gaussian and Log-normal distributions and 25s for the regularly spaced distribution.).

Fixed size scalability tests were also performed for three problem set sizes, small (1 million points), medium (32 million points), and large (128 million points), for the Gaussian distribution. These results are plotted in Figures 19, 20 and 21.



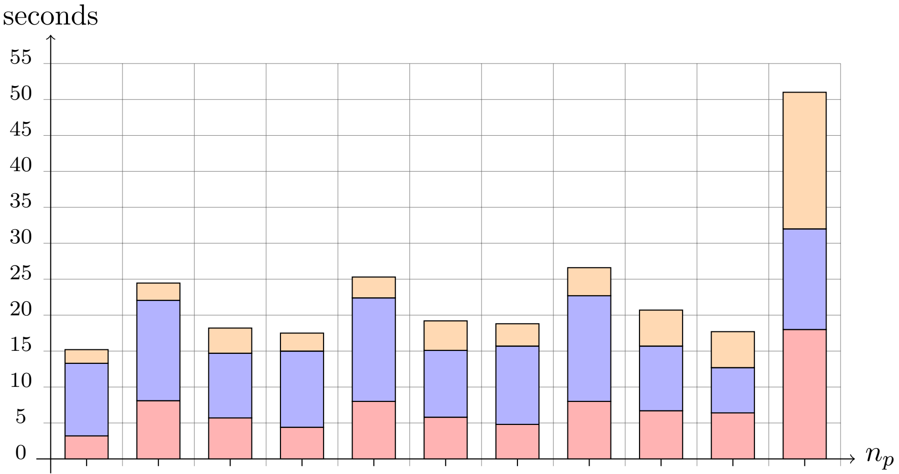
		1	2	4	8	16	32	64	128	256	512	1024
Construction	🟥	2.3	3.0	3.9	3.0	5.6	4.6	4.8	6.2	6.4	8.4	15
Internal Balance	🟦	8.7	11.8	16.6	24.6	26.0	17.1	18.6	23.1	20.3	21.8	15
Boundary Balance	🟩	2.8	3.4	6.4	8.4	6.6	10.7	9.1	9.7	11.0	10.1	11
Communication in Balance	🟧	1.8	1.5	2.5	3.3	3.4	4.2	3.8	4.7	4.6	5.8	14

Figure 16: Isogranular scalability for a Gaussian distribution of `1M` octants per processor. From left to right, the bars indicate the time taken (in seconds) for the different components of our algorithms for increasing processor counts. The bar for each processor is partitioned into `4` sections. From top to bottom, the sections represent the time taken (in seconds) for `(1)` communication (including related pre-processing and post-processing) during balance refinement `(Algorithm 11)`, `(2)` balancing across intra and inter processor boundaries `(Algorithm 9)`, `(3)` balancing the blocks `(Algorithm 7)`, and `(4)` construction from points `(Algorithm 5)`.



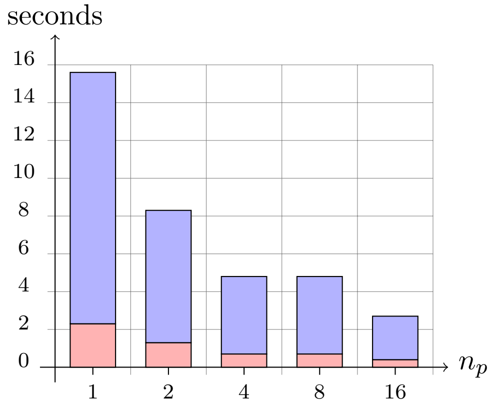
		1	2	4	8	16	32	64	128	256	512	1024
Construction	🟥	2.3	3.1	3.2	3.9	4.7	4.1	5.3	4.6	5.5	7.7	15
Internal Balance	🟦	11.1	11.3	11.4	15.8	20.0	14.7	22.4	20.7	19.2	20.2	20
Boundary Balance	🟩	0.7	4.9	6.5	7.2	8.7	9.9	9.5	9.8	10.0	10.7	9
Communication in Balance	🟧	1.4	2.1	2.2	3.1	3.0	3.4	4.5	4.1	3.5	6.4	14

Figure 17: Isogranular scalability for a Log-normal distribution of `1M` octants per processor. From left to right, the bars indicate the time taken (in seconds) for the different components of our algorithms for increasing processor counts. The bar for each processor is partitioned into `4` sections. From top to bottom, the sections represent the time taken (in seconds) for `(1)` communication (including related pre-processing and post-processing) during balance refinement `(Algorithm 11)`, `(2)` balancing across intra and inter processor boundaries `(Algorithm 9)`, `(3)` balancing the blocks `(Algorithm 7)`, and `(4)` construction from points `(Algorithm 5)`.



		1	2	4	8	16	32	64	128	256	512	1024
Construction	🟥	3.2	8.1	5.7	4.4	8.0	5.8	4.8	8.0	6.7	6.4	18
Internal Balance	🟦	10.1	13.96	9.0	10.6	14.4	9.3	10.9	14.7	9.0	6.3	14
Boundary Balance	🟩	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Communication in Balance	🟧	1.9	2.4	3.5	2.5	2.9	4.1	3.1	3.9	5.0	5.0	19

Figure 18: Isogranular scalability for a Regular distribution of `1M` octants per processor. From left to right, the bars indicate the time taken (in seconds) for the different components of our algorithms for increasing processor counts. The bar for each processor is partitioned into `4` sections. From top to bottom, the sections represent the time taken (in seconds) for `(1)` communication (including related pre-processing and post-processing) during balance refinement `(Algorithm 11)`, `(2)` balancing across intra and inter processor boundaries `(Algorithm 9)`, `(3)` balancing the blocks `(Algorithm 7)` and `(4)` construction from points `(Algorithm 5)`. While both the input and output grain sizes remain almost constant for the Gaussian and LogNormal distributions, only the output grain size remains constant for the Uniform distribution. Hence, the trend seen in this study is a little different from those for the Gaussian and LogNormal distributions.



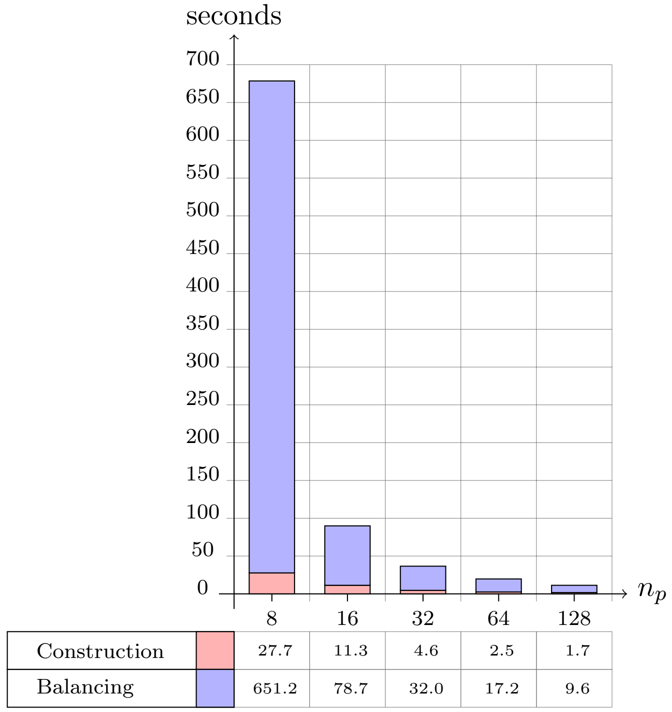
Construction	🟥	2.3	1.3	0.7	0.7	0.4
Balancing	🟦	13.3	7.0	4.1	4.1	2.3

Figure 19: Fixed size scalability for a Gaussian distribution of `1M` octants. From left to right, the bars indicate the time taken (in seconds) for the different components of our algorithms for increasing processor counts. The bar for each processor is partitioned into `2` sections. The top and bottom sections of each column represent the total time taken (in seconds) for `(1)` balance refinement `(Algorithm 11)` and `(2)` construction `(Algorithm 5)`, respectively.

2.4 Summary

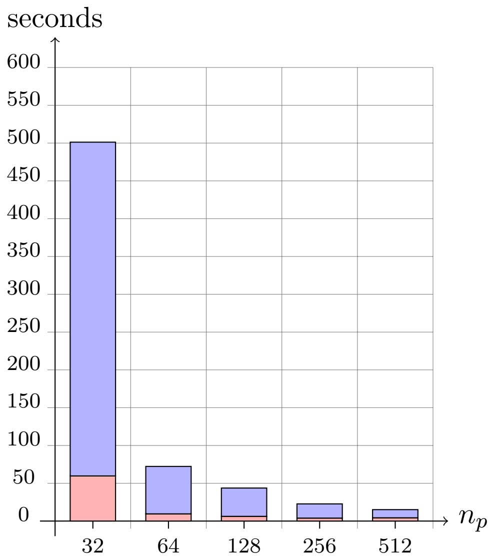
In this chapter, we presented new parallel algorithms for constructing and balancing large linear octrees on distributed memory machines. We also tested MPI-based scalable parallel implementations for both the algorithms. Our algorithms have several important features:

- Experiments on three different types of input distributions demonstrate that the algorithms are insensitive to the underlying data distribution.
- Our algorithms avoid iterative communications and thus are able to achieve low absolute runtime and good scalability.
- Experiments demonstrate that the proposed two-stage intra-processor balancing algorithm has a significantly lower running time compared to alternate approaches.
- We demonstrated scalability up to 1024 processors: we were able to construct and balance octrees with over 1 billion octants in less than one minute.



		8	16	32	64	128
Construction	🟥	27.7	11.3	4.6	2.5	1.7
Balancing	🟦	651.2	78.7	32.0	17.2	9.6

Figure 20: Fixed size scalability for a Gaussian distribution of 32M octants. From left to right, the bars indicate the time taken (in seconds) for the different components of our algorithms for increasing processor counts. The bar for each processor is partitioned into 2 sections. The top and bottom sections of each column represent the total time taken (in seconds) for (1) balance refinement (Algorithm 11) and (2) construction (Algorithm 5), respectively.



Construction	🟥	59.7	9.5	6.2	3.9	4.3
Balancing	🟦	441.8	62.8	37.5	18.8	10.8

Figure 21: Fixed size scalability for a Gaussian distribution of `128M` octants. From left to right, the bars indicate the time taken (in seconds) for the different components of our algorithms for increasing processor counts. The bar for each processor is partitioned into `2` sections. The top and bottom sections of each column represent the total time taken (in seconds) for `(1)` balance refinement `(Algorithm 11)` and `(2)` construction `(Algorithm 5)`, respectively.

CHAPTER III

OCTREE MESHING FOR FINITE ELEMENT COMPUTATIONS

In order to use the 2:1 balanced linear octrees described in the previous chapter for finite element computations, we need additional data structures to store element-to-vertex mappings. This chapter presents a parallel algorithm for constructing these data structures; this process is referred to as *"meshing"*. We also present a compression scheme to compress the linear octree and the element-to-vertex connectivity information. Further, we describe how to construct finite element shape functions using these data structures and describe how we implement a typical finite element matrix-vector multiplication (`MatVec`[1]) without actually assembling the matrix. We focus on reducing (1) the time to build these data structures; (2) the memory overhead associated in storing them; and (3) the time to perform finite element calculations using these data structures.

We avoid using multiple passes (projections) to enforce conformity; instead, we perform a single traversal by mapping each octant to one of eight pre-computed element types, depending on the configuration of hanging vertices for that element. Our data structure does not allow efficient random queries in the octree, but such access patterns are not necessary for finite element calculations.

The memory overhead associated with unstructured meshes arises from the need to store the element connectivity information. In regular grids such connectivity is not necessary as the indexing is explicit. For general unstructured meshes one has to explicitly store the indices that point to the element vertices. In octrees we still need to store this information, but it turns out that instead of storing eight integers (32 `bytes`), we only need to store 12 `bytes`. We use the Golomb-Rice encoding scheme to compress the element connectivity information and to represent it as a Uniquely Decodable Code (UDC) [79]. In addition,

[1] A MatVec is a function that takes a vector as input and returns another vector, the result of applying the matrix on the input vector.

the linear octree is stored in a compressed form that requires only one byte per octant (the level of the octant).

Finally, we employ overlapping of communication and computation to efficiently handle octants shared by several processors or *"ghost"* octants.[2] In addition, the Morton-ordering offers reasonably good memory locality.

Contributions. In a nutshell, the contributions in this chapter are the following:

- We present a parallel algorithm to build element-to-nodal connectivity information efficiently. We use apriori communication of ghost elements to do this instead of the more expensive explicity parallel searches. We also introduce the "4-way" search strategy to reduce the number of searches required for meshing.

- We present a compression scheme for the octree and the element connectivity that achieves a three-fold compression (a total of four words per octant).

- We present a lookup-table-based conforming discretization scheme that requires only a single traversal for the evaluation of a partial differential operator.

- Our implementation supports looping over some ghost elements as well, which allows us to avoid 1 communication step in every finite element Matvec.

Limitations. Some of the limitations of our implementation are listed below:

- Our current implementation only results in a second-order accurate method. A higher-order method can be obtained either by extending the meshing algorithm to support higher order discretizations.

- Problems with complex geometries are not directly supported in our implementation; in principle, the algorithms described here can be combined with fictitious domain methods [45, 94] to allow solution of such problems but the computational costs will increase and the order of accuracy will be reduced.

[2] Every octant is owned by a single processor. However, the values of unknowns associated with octants on interpocessor boundaries need to be shared among several processors. We keep multiple copies of the information related to these octants and we term them ghost octants.

Algorithm 12 Octree Meshing And Compression

```
Input:  A distributed sorted complete balanced linear octree, L
Output: Compressed Octree Mesh and Compressed Octree.

1. Embed L into a larger octree, O, and add boundary octants.
2. Identify "hanging" vertices.
3. Exchange "Ghost" octants.
4. Build lookup tables for first layer of octants. (Section 3.1.1)
5. Perform 4-way searches for remaining octants.
   (Section 3.1.2)
6. Store the levels of the octants and discard the anchors.
7. Compress the mesh (Section 3.2).
```

- Far-field and periodic boundary conditions are not supported in our implementation.

Remark. Our algorithms have $\mathcal{O}(n \log n)$ work and $\mathcal{O}(n)$ storage complexity. For typical distributions of octants (and work per octant), the parallel time complexity of our scheme is $\mathcal{O}(n/n_p \; \log(n/n_p) + n_p \log n_p)$, where n is the final number of leaves and n_p is the number of processors. In contrast to existing implementations, our methods avoid iterative communications and thus, achieve low absolute runtime and excellent scalability. Our algorithm has scaled to four billion octants on 4096 processors on a Cray XT3 ("Big Ben") at the Pittsburgh Supercomputing Center.

Organization of the chapter. The rest of this chapter is organized as follows. In Section 3.1 we describe the construction of element-to-vertex mappings and describe the octree and mesh compression schemes in Section 3.2. In Section 3.3 we describe how we perform the finite element computation. In Section 3.4 we present performance results that demonstrate the efficiency of our implementation.

3.1 Computing the element to vertex mapping

In this section, we describe how we construct the data structures required to perform the finite element Matvecs efficiently. The data structure is designed to be cache efficient by using a Morton ordering based element traversal, and by reducing the memory footprint

Algorithm 13 FINDING THE CHILD NUMBER OF AN OCTANT

Input: The anchor (x,y,z) and level (d) of the octant and the maximum permissible depth of the tree ($\mathcal{L}_{max}$).
Output: c, the child number of the octant.

```
$l \leftarrow 2^{(\mathcal{L}_{max}-d)}$
$l_p \leftarrow 2^{(\mathcal{L}_{max}-d+1)}$
$(i,j,k) \leftarrow (x,y,z) \bmod l_p$
$(i,j,k) \leftarrow (i,j,k)/l$
$c \leftarrow (4k+2j+i)$
```

using compressed representations for both the octree and the element-to-vertex connectivity tables. The algorithm for generating the mesh given a distributed, sorted, complete, balanced linear octree is outlined in Algorithm 12.

In the subsequent sections, we use the term *"child number"* to refer to an octant's configuration with respect to its parent. It is also the octant's position relative to its siblings in a list sorted in the Morton ordering. The child number of an octant is a function of the coordinates of its anchor and its level in the tree. Algorithm 13 is used to compute the child number of an octant. For convenience, the vertices of a given element are numbered according to the Morton ordering. Hence, an octant with a child number equal to k will share its k-th vertex with its parent. An example is shown in Figure 22(a). There are only 8 possible child number configurations.

Since all vertices, except boundary vertices, can be uniquely associated with an octant (the octant with its anchor at the same coordinate as the vertex) we use an interleaved representation where a common index is used for both the elements and the vertices. Since the input balanced octree does not have any octants corresponding to the positive boundary vertices, we embed the input octree in a larger octree with maximum depth $D_{max}+1$; here, D_{max} is the maximum depth of the input octree. All elements on the positive boundaries in the input octree add a layer of octants, with a single linear pass ($\mathcal{O}(n/p)$) followed by a parallel sort ($\mathcal{O}(n/p \log n/p)$)). Since the input octree is already sorted we only sort the extra octants and append them to the original octree.

The second step in the the computation of the element-to-vertex mapping is the identification of hanging vertices. Vertices that exist at the center of a face of another octant are called face-hanging vertices. Vertices that are located at the center of an edge of another octant are called edge-hanging vertices. Octants that are the 0 or 7 children of their parent (a_0, a_7) can never be hanging (Figure 22(a)). Octants that are $3, 5, 6$ children of their parent (a_3, a_5, a_6) can only be *face hanging*, and their status is determined by a single negative search.[3] The remaining octants $(1, 2, 4$ children) are *edge hanging* and identifying their status requires three searches.

After identifying hanging vertices, we repartition the octree using the algorithm described in Section 2.2.1 and all octants touching the inter-processor boundaries are communicated to the neighbouring processors. These octants will be referred to as ghost elements on the processors that receive them and their anchors are called ghost vertices. In our implementation of a typical finite element MatVec, we do not loop over ghost elements recieved from a processor with greater rank and we do not write to ghost vertices. However, we do support writing to ghost values if the need arises. We also support looping over ghost elements recieved from a processor with lower rank. This framework gives rise to a subtle special case for *singular blocks.* A singular block is a block (output of the partition algorithm), which is also a leaf in the underlying fine octree (input to the partition algorithm). If the singular block's anchor is hanging, it might point to a ghost vertex and if so this ghost vertex will be owned by a processor with lower rank. This ghost vertex will be the anchor of the singular block's parent. We tackle this case while partitioning by ensuring that any singular block with a hanging anchor is sent to the processor to which the first child of the singular block's parent is sent. We also send any octant that lies between (in the Morton ordering) the singular block and its parent to the same processor in order to ensure that the relative ordering of the octants is preserved.

After exchanging ghosts, we perform independent sequential searches on each processor to build the element-to-vertex mappings. We present two methods for the same, an exhaustive approach (Section 3.1.1) that searches for all the 8 vertices for each element, and a more efficient approach that utilizes the mapping of its negative face neighbours (Section

[3] By *"negative"* searches we refer to searches in the $-x$ or $-y$ or $-z$ directions. We use *"positive searches"* to refer to searches along the positive directions.

3.1.2) to construct its own mapping.

3.1.1 Exhaustive searches to compute mapping

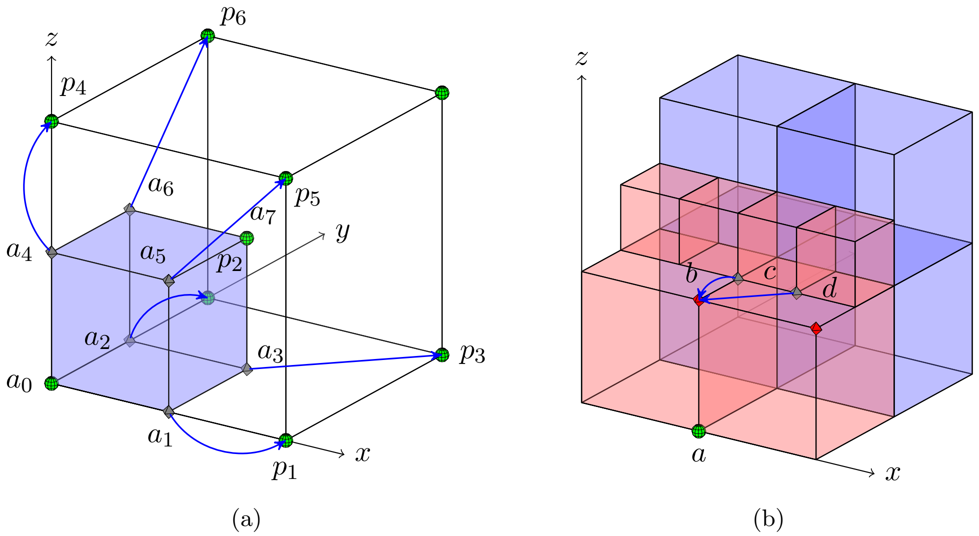
The simplest approach to compute the element-to-vertex mapping would be to search for the vertices explicitly using a parallel search algorithm, followed by the computation of global-to-local mappings that are necessary to manage the distributed data. However, this would incur expensive communication and synchronization costs. To reduce these costs, we chose to use *a priori* communication of "ghost" octants[4] followed by independent local searches on each processor that require no communication. A few special cases that can not be identified easily during a priori communication are identified during the local searches and corrected later.

For any element, all vertices except the anchor are in the positive direction. The exhaustive search strategy is as follows: Generate search keys at the location of the eight vertices of the element at the maximum depth and search for them in the linear octree. Since the linear octree is sorted, the search can be performed in $\mathcal{O}(\log n)$. If the search result is not hanging, then the lookup table is updated. If we discover a hanging vertex instead, then a secondary search is performed to recover the correct non-hanging index. As shown in Figure 22(a), hanging vertices are always mapped to the corresponding vertices of their parent[5]. Unfortunately, secondary searches can not be avoided despite the identification of hanging vertices prior to searching. This is because only the element whose anchor is hanging knows this information and the other elements that share this vertex must first search and find this element in order to learn this information.

Using exhaustive searches we can get the mapping for most elements, but certain special cases arise for ghost elements. This case is illustrated in Figure 22(b), where the ghost elements are drawn in red and the local elements are drawn in blue. Consider searching for the $+z$ neighbor of element a. Since a is a ghost, and we only communicate a single layer of ghost octants across processor boundaries, the vertex b will not be found. In such cases, we set the mapping to point to one of the hanging siblings of the missing vertex (c or d in this

[4] The use of blocks makes it easy to identify "ghost" octants.

[5] The 2:1 balance constraint ensures that the vertices of the parent can never be hanging.



(a)

(b)

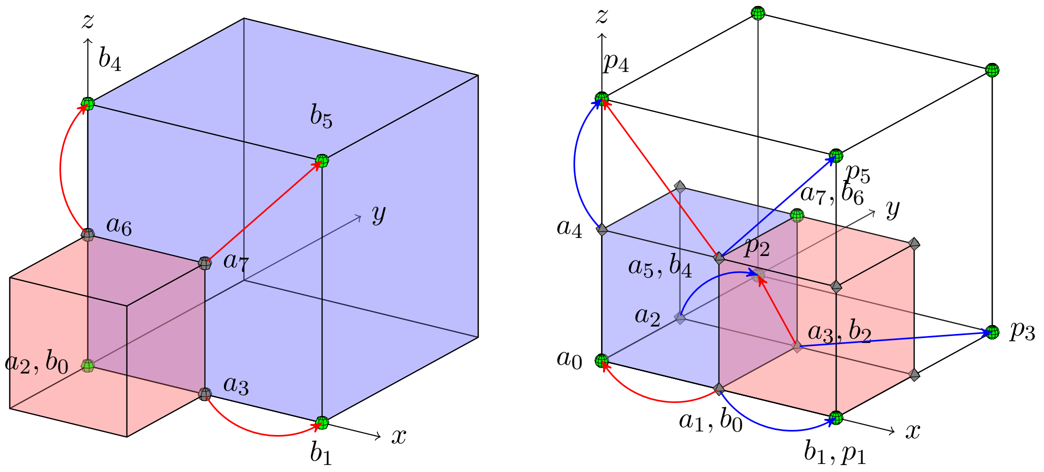
Figure 22: (a) Illustration of nodal-connectivities required to perform conforming FEM calculations using a single tree traversal. Every octant has at least 2 non-hanging vertices, one of which is shared with the parent and the other is shared amongst all the siblings. The octant shown in blue (a) is a child 0, since it shares its zero vertex (a_0) with its parent. It shares vertex a_7 with its siblings. All other vertices, if hanging, point to the corresponding vertex of the parent octant instead. Vertices, a_3, a_5, a_6 are face hanging and point to p_3, p_5, p_6, respectively. Similarly a_1, a_2, a_4 are edge hanging and point to p_1, p_2, p_4. (b) The figure explains the special case that occurs during exhaustive searches of ghost elements. Element anchored at a, when searching for vertex b, will not find any vertex. Instead, one of the hanging siblings of b, (c, d) which are hanging will be pointed to. Since hanging vertices do not carry any information, the information for b will be replicated to all its hanging siblings while updating the ghosts.

case). The most likely condition under which b in not found is when the $+z$ neighbor(s) is smaller. In this case, we know that at least one of the siblings will be hanging. Although the lookup table for this element is incorrect, we make use of the fact that the lookup table points to a non-existent vertex, and the owner of the true vertex simply copies its data value to the hanging vertex locations, thereby ensuring that the correct value is read. This case can only happen for ghost elements and need not be done for local elements. In addition, we occasionally observe cases where neither the searched vertex nor any of its siblings are found. Such cases are marked and at the end of the lookup table construction, a parallel search is done to obtain the missing vertices directly from the processors that own them.

3.1.2 Four-way searches to compute mapping

The exhaustive search explicitly searches for all vertices and in many cases is the only way to find the correct element-to-vertex mapping. However, it requires a minimum of 7 and a maximum of 13 searches per element. In order to reduce the constants associated with the exhaustive search, we use the exhaustive search only for the first layer of octants (octants that do not have neighbours in the negative x, y and z directions) on each processor. For all other octants, the lookup table information can be copied from the elements in the negative directions. Each element in the negative x, y and z directions that shares a face with the current element, also shares 4 vertices. Therefore, by performing negative searches along these directions, we can obtain the lookup information for 7 out of the 8 vertices of an element. Only the last vertex, labeled a_7 in Figure 22(a), cannot be obtained using a negative search and a positive search is required.

In order to get the mapping information using negative searches, we perform the search in the negative direction and check if the current element is a sibling of the element obtained via the negative search. If the element found by the search is not a sibling of the current element, then the lookup information can be copied via a mapping. For the example shown in Figure 23(a), given the element b and searching in the $-y$ direction, we find a, then the mapping is $(b_0, b_1, b_4, b_5) = (a_2, a_3, a_6, a_7)$. Corresponding mappings are $(b_0, b_2, b_4, b_6) = (a_1, a_3, a_5, a_7)$, and $(b_0, b_1, b_2, b_3) = (a_4, a_5, a_6, a_7)$, for negative searches along the x and z



(a)

(b)

Figure 23: Computing element-to-vertex mapping using negative searches. (a) If the found octant (a) is not a sibling of the current octant (b), then the element-to-vertex mapping can be copied via the mapping $b_0 \leftarrow a_2$, $b_1 \leftarrow a_3$, $b_4 \leftarrow a_6$, and $b_5 \leftarrow a_7$. (b) In case the found octant (a) is a sibling of the current octant (b), then the mapping depends on whether or not the vertex in question is hanging. If the vertex is not hanging, then the same mapping as used in (a) can be applied. If the vertex is hanging, then the corresponding indices for the found element are directly copied. For the case shown, $(b_0, b_2, b_4, b_6) \leftarrow (a_0, a_2, a_4, a_7) = (p_0, p_2, p_4, a_7)$.

axes, respectively. Unfortunately, the mapping is a bit more complex if the negative search returns a sibling of the current element. If the vertex in question is not hanging, then we can copy its value according to the above mentioned mapping. However, if the vertex in question is hanging, then instead of the mapping, the corresponding indices from element a are copied. This case is explained in Figure 23(b), where we observe that if vertex a_1, b_0 is hanging, we need to use $b_0 = a_0$ and use $b_0 = a_1$ if it is not hanging.

3.2 Mesh compression

One of the major problems with unstructured meshes is the storage overhead. In the case of the octree, this amounts to having to store both the octree and the lookup table. In order to reduce the storage costs associated with the octree, we compress both the octree and the lookup table. The sorted, unique, linear octree can be easily compressed by retaining only the offset of the first element and the level of subsequent octants. Storing the offset for each octant requires a storage of three integers (12 `bytes`) and a `byte` for storing the level. Storing only the level represents a 12`x` compression as opposed to storing the offset for every octant.

It is much harder to compress the element-to-vertex mapping, which requires eight integers for each element. In order to devise a good compression scheme, we first estimate the distribution of indices. The following lemma helps us analyze the distribution of the indices of the vertices of a given element.

Lemma 4 *The Morton ids of the vertices of a given element are greater than or equal to the Morton id of the element.*

Proof. Let the anchor of the given element be (x, y, z) and let its size be h. In that case the anchors of the 8 vertices of the element are given by $(x, y, z), (x + h, y, z), (x, y + h, z), (x + h, y + h, z) \cdots$. By the definition of the Morton ordering all of these except (x, y, z) are greater than the Morton id of the element. The vertex at (x, y, z) is equal to the Morton id of the element. □

Corollary 1 *Given a sorted list of Morton ids corresponding to the combined list of elements and vertices of a balanced linear octree, the indices of the 8 vertices of a given element in this list are strictly greater than the index of the element. Moreover, if the vertices are listed in the Morton order, the list of indices is monotonically increasing. If we store offsets in the sorted list, then these offsets are strictly positive.*

Based on these observations we can estimate the expected range of offsets. Let us consider a balanced octree, O, with n octants and with maximum possible depth $D_{\max}$. Consider an element in the octree, o_i, whose index is i, $0 \leqslant i < n$. The offset of the anchor of this element is either i (if the anchor is not hanging) or $n_0 < i$. The indices for the remaining 7 vertices do not depend on octants with index less than i. In addition since the indices of the 7 vertices are monotonically increasing, we can store offsets between two consecutive vertices. That is, if the indices of the 8 vertices of an element, o_i, are $(n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7)$, we only need to store $(n_0 - i, n_1 - n_0, n_2 - n_1, n_3 - n_2, n_4 - n_3, n_5 - n_4, n_6 - n_5, n_7 - n_6)$. To efficiently store these offsets, we need to estimate how large these offsets can be. We start with a regular grid, i.e., a balanced octree with all octants at $D_{\max}$. Note that any octree that can be generated at the same value of $D_{\max}$ can be obtained by applying a series of local coarsening operations to the regular grid. Since we only store the offsets it is sufficient to analyze the distribution of the offset values for one given direction, say for a neighbor along the x-axis. The expression for all other directions are similar.

For $D_{\max} = 0$, there is only one octant and correspondingly the offset is 1. If we introduce a split in the root octant, $D_{\max}$ becomes 1, the offset increases by 2 for one octant. On introducing further splits, the offset is going to increase for those octants that lie on the boundaries of the original splits, and the general expression for the maximum offset can be written as offset $= 1 + \sum_{i=1}^{D_{\max}} 2^{d \cdot i - 1}$, for a d-tree. In addition, a number of other large offsets are produced for intermediate split boundaries. Specifically for a regular grid at maximum depth $D_{\max}$, we shall have $2^{d \cdot (D_{\max} - x)}$ octants with an offset of $1 + \sum_{i=1}^{x} 2^{d \cdot i - 1}$. As can be clearly seen from the expression, the distribution of the offsets is geometric. With the largest number of octants having small offsets.

For the case of general balanced octrees, we observe that any of these can be obtained from a regular grid by a number of coarsening operations. The only concern is whether the coarsening can increase the offset for a given octant. The coarsening does not affect octants that are greater than the current octant (in the Morton order). For those which are smaller, the effect is minimal since every coarsening operation reduces the offsets that need to be stored.

Golomb-Rice coding [46, 95] is a form of entropy encoding that is optimal for geometric distributions, that is, when small values are vastly more common than large values. Since, the distribution of the offsets is geometric, we expect a lot of offsets with small values and fewer occurrences of large offsets. The Golomb coding uses a tunable parameter M to divide an input value into two parts: q, the result of a division by M, and r, the remainder. In our implementation, the remainder is stored as a `byte`, and the quotient as a `short`. On an average, we observe one large jump in the indices, and therefore the amortized cost of storing the compressed lookup table, is 8 `bytes` for storing the remainders, 2 `bytes` for the quotient, one `byte` for storing a flag to determine which of the 8 vertices need to use a quotient, and one additional `byte` for storing additional element specific flags. Storing the lookup explicitly would require 8 `ints`, and therefore we obtain a `3x` compression in storing the lookup table.

3.3 Finite element computation on octrees

In this section, we describe the evaluation of a `MatVec` with the global finite element "stiffness" matrix. A key difference between our `MatVec` and earlier approaches [129] is that the *"hanging vertices*[6]*"* are not stored explicitly. A method to eliminate hanging vertices in locally refined quadrilateral meshes and yet ensure inter-element continuity by the use of special bilinear quadrilateral elements was presented in [132]. We have extended that approach to three dimensions.

The following properties of 2:1 balanced linear octrees helps us reduce the total number of permissible hanging configurations. Figure 22(a) illustrates these properties.

[6] *They do not represent independent degrees of freedom in a FEM solution.*

- Every octant has at least 2 non-hanging vertices:
    - The vertex that is common to both this octant and its parent.
    - The vertex that is common to this octant and all its siblings.
- An octant can have a face-hanging vertex only if the remaining vertices on that face are one of the following:
    - Edge hanging vertices.
    - The vertex that is common to both this octant and its parent.

After factoring in the above constraints, there are only 18 potential hanging-vertex configurations for each of the 8 child number configurations.

Below, we list some of the properties of the shape functions defined on octree meshes.

- No shape function is rooted at hanging vertices.
- The shape functions are trilinear.
- The shape functions assume a value of 1 at the vertex at which they are rooted and a value of 0 at all other non-hanging vertices in the octree.
- The support of a shape function can spread over more than 8 elements.
- If a vertex of an element is hanging, then the shape functions rooted at the other non-hanging vertices in that element do not vanish on this hanging vertex. Instead, they will vanish at the non-hanging vertex that this hanging vertex is mapped to. If the i-th vertex of an element/octant is hanging, then the index corresponding to this vertex will point to the i-th vertex of the parent[7] of this element instead. For example, in Figure 22(a) the shape function rooted at vertex a_0 will not vanish at vertices a_1, a_2, a_3, a_4, a_5 or a_6. It will vanish at vertices p_1, p_2, p_3, p_4, p_5, p_6 and a_7. It will assume a value equal to 1 at vertex a_0.

[7] The 2:1 balance constraint ensures that the vertices of the parent can never be hanging.

- A shape function assumes non-zero values within an octant if and only if it is rooted at some non-hanging vertex of this octant or if some vertex of the octant under consideration is hanging, say the i-th vertex, and the shape function in question is rooted at the i-th non-hanging vertex of the parent of this octant. Hence, for any octant there are exactly eight shape functions that do not vanish within it and their indices will be stored in the vertices of this octant.

- The finite element matrices constructed using these shape functions are mathematically equivalent to those obtained using projection schemes such as in [73, 129, 130].

3.3.1 Overlapping communication with computation

Every octant is owned by a single processor. However, the values of unknowns associated with octants on inter-processor boundaries need to be shared among several processors. We keep multiple copies of the information related to these octants and we term them "*ghost*" octants. In our implementation of the finite element MatVec, each processor iterates over all the octants it owns and also loops over a layer of ghost octants that contribute to the vertices it owns. Within the loops, each octant is mapped to one of the above described hanging configurations. This is used to select the appropriate element stencil from a list of pre-computed stencils. Although a processor needs to read ghost values from other processors, it only needs to write data back to the vertices it owns and does not need to write to ghost vertices.[8] Thus, there is only one communication phase within each MatVec, which we can overlap with a computation phase:

1. Initiate non-blocking MPI sends for information stored on ghost-vertices.

2. Loop over the elements in the interior of the processor domain. These elements do not share any vertices with other processors. We identify these elements during the meshing phase itself.

3. Receive ghost information from other processors.

[8] This is possible because our meshing scheme also builds the element-to-vertex connectivity mappings for the appropriate ghost elements. Although, this adds an additional layer of complexity to our meshing algorithm, it saves us one communication per MatVec.

4. Loop over remaining elements to update information.

3.4 Performance evaluation

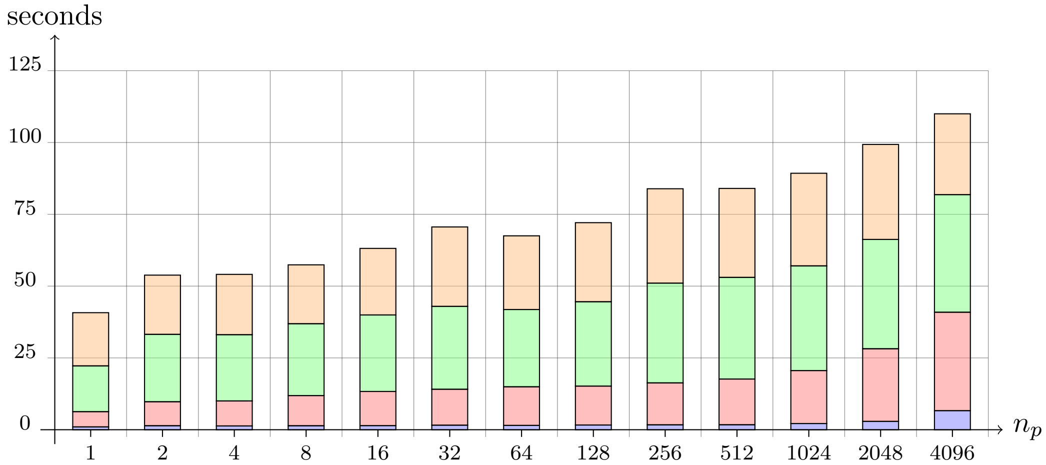
In this section we present numerical results for the tree construction, balancing, meshing and matrix vector multiplication for a number of different cases. The algorithms were implemented in C++ using the MPI library. `PETSc` [10, 9] was used for profiling the code. We consider two point distribution cases: a regular grid one, to directly compare with structured grids; and a Gaussian distribution which resembles a generic non-uniform distribution. In all examples we discretized a variable-coefficient linear elliptic operator. We used piecewise constant coefficients for both the Laplacian and Identity operators. Material properties for the element were stored in an independent array, rather than within the octree data structure.

First we tested the performance of the code on a sequential machine and compared it to a regular grid implementation with direct indexing (the vertices are ordered in lexicographic order along the coordinates). The results are presented in Table 4. We report construction times, and the total time for 5 matrix vector multiplications. Overall the code performs quite well. Both the meshing time and the time for performing the `MatVecs` are not sensitive to the input point distribution used to construct the octrees. The `MatVec` time is only 50% more than that for a regular grid with direct indexing, about five seconds for four million octants.

In the second set of experiments we tested the isogranular scalability of our code. Again, we considered two point distributions, a uniform one and a Gaussian. The size of the input points, the corresponding linear and balanced octrees, the number of vertices, and the runtimes for the two distributions are reported in Figures 24 and 25. All the runs were performed on a Cray XT3 MPP system equipped with 2068 compute nodes (two 2.6 GHz AMD Opteron and 2 GBytes of RAM per node) at the Pittsburgh Supercomputing Center. We observe excellent scalability for the construction and balancing of octrees, meshing and the matrix-vector multiplication operation. For example, in Figure 24 we observe the expected complexity in the construction and balancing of the octree (there is a slight

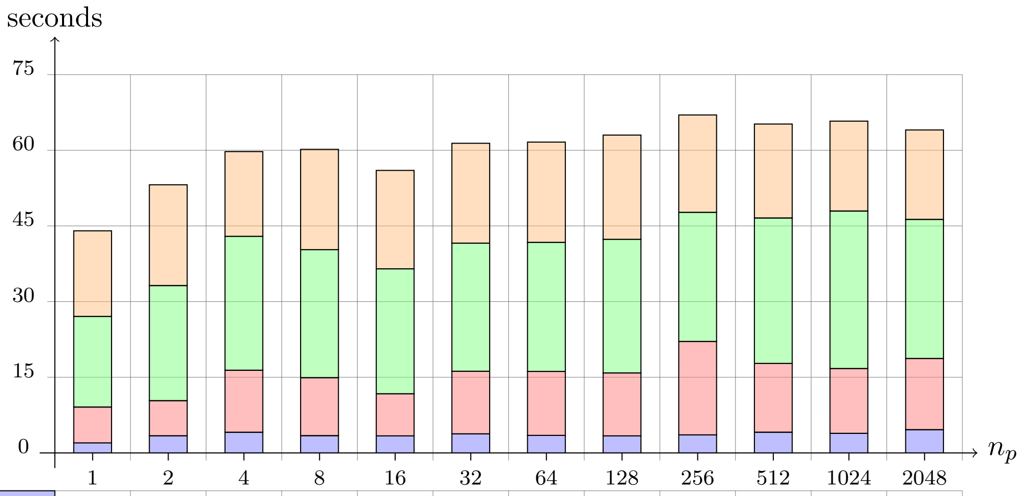
Table 4: The time (in seconds) to construct (**Meshing**) and perform 5 matrix-vector multiplications (**MatVec**) on a single processor for increasing problem sizes. Results are presented for Gaussian distribution and for uniformly spaced points. We compare with matrix-vector multiplication on a regular grid (no indexing) having the same number of elements and the same discretization (trilinear elements). We discretize a variable coefficient (isotropic) operator. The runs took place on a 2.2 GHz, 32-bit Xeon box. The sustained performance is approximately 400 MFlops/sec for the structured grid. For the uniform and Gaussian distribution of points, the sustained performance is approximately 280 MFlops/sec.

Problem Size	Regular Grid	Octree Mesh			
		Uniform		Gaussian	
	MatVec	Meshing	MatVec	Meshing	MatVec
256K	1.08	4.07	1.62	4.34	1.57
512K	2.11	8.48	3.18	8.92	3.09
1M	4.11	17.52	6.24	17.78	6.08
2M	8.61	36.27	11.13	37.29	12.33
4M	17.22	73.74	24.12	76.25	24.22



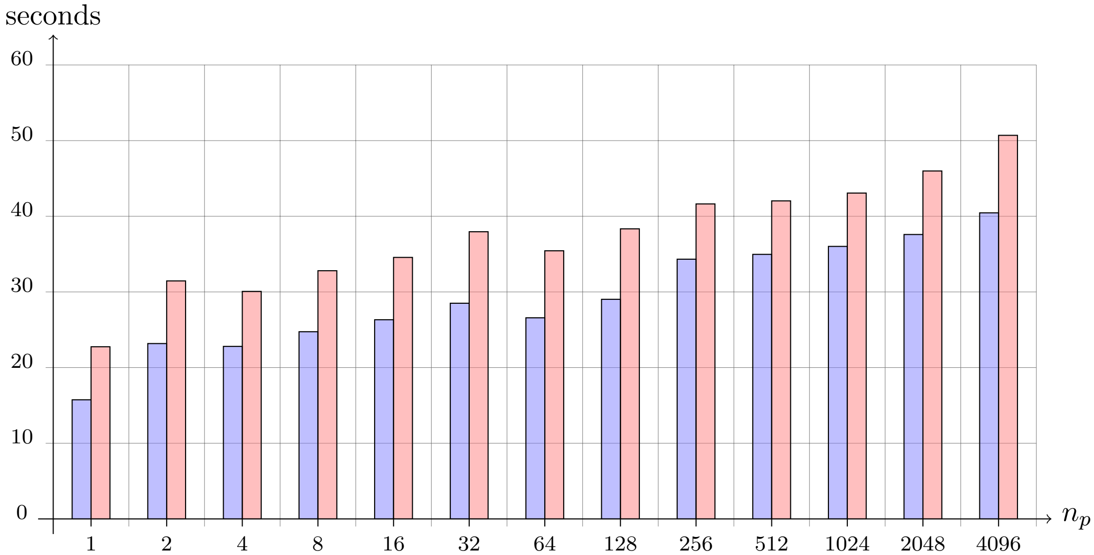
	1	2	4	8	16	32	64	128	256	512	1024	2048	4096
Octree Construction	0.99	1.41	1.32	1.40	1.44	1.60	1.51	1.65	1.72	1.75	2.17	2.92	6.68
Octree Balancing	5.34	8.39	8.75	10.55	11.94	12.57	13.52	13.62	14.67	15.99	18.53	25.42	34.44
Meshing	16.01	23.57	23.19	25.15	26.77	28.99	27.03	29.52	34.91	35.56	36.63	38.23	41.14
MatVec (5)	18.61	20.72	21.09	20.59	23.29	27.79	25.78	27.65	33.01	31.12	32.40	33.24	28.27
Points	180K	361K	720K	1.47M	2.89M	5.8M	11.7M	23.5M	47M	94M	188M	376M	752M
Unbalanced Octants	607K	1.2M	2.4M	4.9M	9.7M	19.6M	39.3M	79.3M	158M	315M	635M	1.26B	2.52B
Balanced Octants	996K	2M	4M	8M	16M	31.9M	64.4M	131M	257M	519M	1.04B	2.05B	4.16B
Independent Vertices	660K	1.3M	2.7M	5.2M	10.5M	21.5M	42M	87.8M	172M	339M	702M	1.36B	2.72B

Figure 24: Isogranular scalability for Gaussian distribution of `1M` octants per processor. From left to right, the bars indicate the time taken (in seconds) for the different components of our algorithms for increasing processor counts. The bar for each processor is partitioned into `4` sections. From top to bottom, the sections represent the time taken (in seconds) for `(1)` performing 5 Matrix-Vector multiplications, `(2)` Construction of the octree-based mesh, `(3)` balancing the octree and `(4)` construction from points.



	1	2	4	8	16	32	64	128	256	512	1024	2048
Octree Construction	2.01	3.44	4.15	3.47	3.42	3.83	3.52	3.42	3.62	4.16	3.93	4.67
Octree Balancing	7.21	7.06	12.46	11.64	8.46	12.57	12.86	12.65	18.78	13.83	13.03	14.31
Meshing	18.21	23.12	26.91	25.72	25.11	25.74	25.93	26.85	25.94	29.21	31.64	27.94
`MatVec` (5)	17.19	20.25	17.02	20.16	19.77	20.07	20.14	20.94	19.56	18.88	18.06	17.97
Points	1M	2.05M	3.94M	8M	15.8M	31.8M	63.5M	127.3M	248.8M	504.5M	1B	1.96B
Unbalanced Octants	1.06M	2.09M	4.14M	8.29M	16.1M	32.3M	64.3M	128M	250.5M	505.6M	1B	2B
Balanced Octants	1.06M	2.09M	4.14M	8.29M	16.1M	32.3M	64.3M	128.5M	250.5M	505.6M	1B	2B
Vertices	1.07M	2.15M	4.14M	8.29M	16.1M	32.3M	64.3M	128.5M	250.5M	505.6M	1B	2B

Figure 25: Isogranular scalability for uniformly spaced points with `1M` octants per processor. From left to right, the bars indicate the time taken (in seconds) for the different components of our algorithms for increasing processor counts. The bar for each processor is partitioned into `4` sections. From top to bottom, the sections represent the time taken (in seconds) for `(1)` performing 5 Matrix-Vector multiplications, `(2)` Construction of the octree-based mesh, `(3)` balancing the octree and `(4)` construction from points.



	1	2	4	8	16	32	64	128	256	512	1024	2048	4096
4-way	16.01	23.57	23.19	25.16	26.77	28.99	27.03	29.52	34.91	35.56	36.63	38.23	41.14
Exhaustive	23.14	31.99	30.58	33.36	35.15	38.60	36.04	38.99	42.34	42.75	43.80	46.77	51.56

Figure 26: Comparison of meshing times (in seconds) using exhaustive search with using a hybrid approach where only the first layer of octants uses exhaustive search and the rest use the 4-way search to construct the lookup tables. The test was performed using a Gaussian distribution of 1 million octants per processor. It can be seen that the 4-way search is faster than the exhaustive search and scales upto 4096 processors.

growth due to the logarithmic factor in the complexity estimate) and we observe a roughly size-independent behavior for the matrix-vector multiplication. The results are even better for the uniform distribution of points in Figure 25, where the time for 5 matrix-vector multiplications remains nearly constant at approximately 20 seconds.

Finally, we compared the meshing time for the two search strategies presented in Section 3.1. The improvement in meshing time as a result of using the 4-way search is shown in Figure 26, for the Gaussian distribution.

3.5 Summary

In this chapter, we presented a parallel algorithm for meshing linear octrees. Our mesh data structure is interfaced with `PETSc` [10, 9], thus allowing us to use its linear and non-linear solvers. Our data structure supports second order accurate finite element discretizations of partial differential equations. We presented results that verify the overall scalability of our code. The overall meshing time was approximately one minute for problems with four billion elements using 4096 processors. Thus, our scheme enables efficient execution of applications that require frequent remeshing.

CHAPTER IV

GEOMETRIC MULTIGRID ON OCTREES

The previous chapters described how to construct octree meshes and discretize partial differential equations using finite element shape functions. In this chapter, we present a parallel geometric multigrid algorithm for solving the resulting finite element equations efficiently. Although several sequential and parallel multigrid implementations are available [2, 10, 64], to our knowledge there is no work on octree-based, matrix-free, geometric multigrid solvers for finite element discretizations that has scaled to thousands of processors. In addition to the components described in the previous chapters, this method includes a global coarsening algorithm and a matrix-free implementation for the intergrid transfer operations. The coarsening algorithm is used to construct a sequence of coarser 2:1 balanced octrees starting with an arbitrary 2:1 balanced fine-grid octree. The intergrid transfer operations are used to restrict the residuals from the fine grid to the coarse grid and interpolate the solution of the coarse grid problems to the fine grid.

The overall scheme is second-order accurate, for sufficiently smooth right-hand sides and material properties; and its complexity, for nearly uniform trees, is $\mathcal{O}(\frac{N}{n_p}\log\frac{N}{n_p}) + \mathcal{O}(n_p \log n_p)$, where N is the number of octants, and n_p is the number of processors. Our implementation, `Dendro`, uses the Message Passing Interface (MPI) standard and is built on top of the `PETSc` library from Argonne National Laboratory. `Dendro` has been released as an open source software that can be downloaded from [104].

Salient features. The main features of the proposed multigrid algorithm are listed below.

- In our global coarsening approach we construct a sequence of coarse octrees starting with an arbitrary fine octree. An alternative approach would be to use regular refinements of a coarse octree to construct a sequence of octrees. Although global coarsening poses more difficulties with partitioning and load balancing compared to global refinement, it is more natural for typical PDE applications in which only some discrete representation (e.g., material properties defined at certain points) is available.

- We do not impose any restrictions on the number of multigrid levels or the size of the coarsest mesh. We automatically reduce the number of processors at the coarser levels if the grain size becomes too small and manage the different partitions and communicators in a seamless fashion.
- Transferring information between successive multigrid levels in parallel is a challenging task because the coarse and fine grids may have been partitioned across processors in a completely different way. A scalable, matrix-free implementation of the intergrid transfer operators is one of the main components of the multigrid algorithm.
- The setup costs of our algorithm is low making it ideal for applications that require repeated solutions of linear systems of equations. This is significant for time dependent and nonlinear problems.
- The MPI-based implementation of our multigrid method, `Dendro`, has scaled to billions of elements on thousands of processors even for problems with large contrasts in the material properties.

Limitations. Some of the limitations of the proposed methodology are listed below:

- The method is not robust for problems with large jumps in the material properties.
- The problems of load balancing across processors has not been fully addressed in this work.

Organization of the chapter. In Section 4.1, we present a symmetric variational problem and describe a V-cycle multigrid algorithm to solve the corresponding discretized system of equations. It is common to work with discrete, mesh-dependent, inner products in these derivations so that inverting the Gram matrix[1] can be avoided [11, 22, 23, 24, 142, 143, 144].

[1] Given an inner-product and a set of vectors, the Gram matrix is defined as the matrix whose entries are the inner-products of the vectors.

However, we do not impose any such restrictions. Instead, we show (Section 4.1.5) how to avoid inverting the Gram matrix for any choice of the inner-product. In Section 4.2, we describe a matrix-free implementation for the multigrid method. In Section 4.3, we present numerical experiments for the Laplace and Navier (linear elasticity) operators that demonstrate the scalability of our method. Our largest run was a highly-nonuniform, 8-billion-unknown, elasticity calculation on 32,000 processors.

4.1 A finite element multigrid formulation

4.1.1 Variational problem

Given a domain $\Omega \subset \mathcal{R}^3$ and a bounded, symmetric bilinear form, $a(u,v)$, that is coercive on $H^1(\Omega)$ and $f \in L^2(\Omega)$, we want to find $u \in H^1(\Omega)$ such that u satisfies

$$a(u,v) = (f,v)_{L^2(\Omega)} \quad \forall v \in H^1(\Omega) \tag{3}$$

and the appropriate boundary conditions on the boundary of the domain, $\partial\Omega$. This problem has a unique solution [24].

4.1.1.1 Galerkin approximation

In this section, we derive a discrete set of equations that need to be solved to find an approximate solution for Equation 3. First, we define a sequence of nested *finite* dimensional spaces, $V_1 \subset V_2 \subset \cdots \subset H^1(\Omega)$, all of which are subspaces of $H^1(\Omega)$. Here, V_k corresponds to a fine mesh and V_{k-1} corresponds to the immediately coarser mesh. Then, the discretized problem is to find an approximation of u, $u_k \in V_k$, such that

$$a(u_k,v) = (f,v)_{L^2(\Omega)} \quad \forall v \in V_k. \tag{4}$$

The discretized problem has a unique solution and the sequence $\{u_k\}$ converges to u [24].

Let $(\cdot,\cdot)_k$ be an inner-product defined on V_k. By using the linear operator $A_k : V_k \rightarrow V_k$ defined by

$$(A_k v, w)_k = a(v,w) \quad \forall v,w \in V_k, \tag{5}$$

the discretized problem can be restated as follows: Find $u_k \in V_k$, which satisfies

$$A_k u_k = f_k \tag{6}$$

where $f_k \in V_k$ is defined by

$$(f_k, v)_k = (f, v)_{L^2(\Omega)} \quad \forall v \in V_k \tag{7}$$

The operator A_k is a symmetric (self-adjoint) positive operator w.r.t $(\cdot, \cdot)_k$. (In the following sections, we use italics to represent an operator (or vector) in the continuous form and use bold face to represent the matrix (or vector) corresponding to its co-ordinate basis representation.)

Let $\left\{\phi_1^k, \phi_2^k, \ldots, \phi_{\#(V_k)}^k\right\}$ be a basis for V_k. Then, we can show the following:

$$\boxed{\begin{array}{rcl} \mathbf{A_k} & = & (\mathbf{M_k^k})^{-1}\tilde{\mathbf{A}}_\mathbf{k} \\ \mathbf{f_k} & = & (\mathbf{M_k^k})^{-1}\tilde{\mathbf{f}}_\mathbf{k} \\ \mathbf{M_k^k}(i,j) & = & (\phi_i^k, \phi_j^k)_k \\ \tilde{\mathbf{A}}_\mathbf{k}(i,j) & = & a(\phi_i^k, \phi_j^k) \quad \forall i, j = 1, 2, \ldots, \#(V_k) \\ \tilde{\mathbf{f}}_\mathbf{k}(j) & = & (f, \phi_j^k)_{L^2(\Omega)} \quad \forall j = 1, 2, \ldots, \#(V_k) \end{array}} \tag{8}$$

In Equation 8, $\mathbf{M_k^k}$ is the Gram or mass matrix.

4.1.2 Prolongation

The prolongation operator is a linear operator

$$P : V_{k-1} \rightarrow V_k \tag{9}$$

defined by

$$Pv = v \quad \forall v \in V_{k-1} \subset V_k. \tag{10}$$

This is a standard prolongation operator and has been used previously [24, 25]. The variational form of Equation 10 is given by

$$(Pv, w)_k = (v, w)_k \quad \forall v \in V_{k-1}\ ,\ w \in V_k. \tag{11}$$

In the Appendix, we show that

$$\boxed{\mathbf{P}(i,j) = \phi_j^{k-1}(p_i).} \tag{12}$$

In equation 12, p_i is the fine-grid vertex associated with the fine-grid finite element shape function, ϕ_i^k and ϕ_j^{k-1} is a coarse-grid finite element shape function.

4.1.3 Coarse-grid problem

The coarse-grid problem can be stated as follows: Find $v_{k-1} \in V_{k-1}$ that satisfies

$$A_{k-1}^G v_{k-1} = f_{k-1}^G \tag{13}$$

where, A_{k-1}^G and f_{k-1}^G are defined by the "*Galerkin*" condition (Equation 14) [25].

$$\boxed{\begin{aligned} A_{k-1}^G &= P^* A_k P \\ f_{k-1}^G &= P^*(A_k v_k - f_k), \\ &\quad \forall v_{k-1} \in V_{k-1}, v_k \in V_k \end{aligned}} \tag{14}$$

Here, P is the prolongation operator defined in Section 4.1.2 and P^* is the Hilbert adjoint operator[2] of P with respect to the inner-products $(\cdot,\cdot)_k$ and $(\cdot,\cdot)_{k-1}$.

4.1.4 Restriction

Since the restriction operator must be the Hilbert adjoint of the prolongation operator, we define the restriction operator $R : V_k \rightarrow V_{k-1}$ as follows:

$$(Rw, v)_{k-1} = (w, Pv)_k = (w, v)_k \quad \forall v \in V_{k-1}, w \in V_k \tag{15}$$

In the Appendix, we show that

$$\boxed{\mathbf{R} = (\mathbf{M}_{\mathbf{k-1}}^{\mathbf{k-1}})^{-1}\mathbf{M}_{\mathbf{k}}^{\mathbf{k-1}}} \tag{16}$$

where,

$$\mathbf{M}_{\mathbf{k}}^{\mathbf{k-1}}(i,j) = (\phi_i^{k-1}, \phi_j^k)_k = \mathbf{M}_{\mathbf{k-1}}^{\mathbf{k}}(j,i). \tag{17}$$

[2] P is a bounded linear operator from one Hilbert space, V_{k-1}, to another, V_k, and hence it has an unique, bounded, linear Hilbert adjoint operator with respect to the inner-products considered [74].

Algorithm 14 Two-Grid Correction Scheme

```
1. Relax $\nu_1$ times on Equation 60 with an initial guess, $u_k^0$.
   (Pre-smoothing)
2. Compute the fine-grid residual using the solution vector, $v_k$, at the
   end of the pre-smoothing step: $\mathbf{r_k} = \tilde{\mathbf{f}}_\mathbf{k} - \tilde{\mathbf{A}}_\mathbf{k}\mathbf{v_k}$.
3. Compute: $\mathbf{r_{k-1}} = \mathbf{P}^T\mathbf{r_k}$. (Restriction)
4. Solve for $\mathbf{e_{k-1}}$ in Equation 61. (Coarse-grid correction)
5. Correct the fine-grid approximation: $\mathbf{v_k^{new}} = \mathbf{v_k} + \mathbf{Pe_{k-1}}$.
   (Prolongation)
6. Relax $\nu_2$ times on Equation 60 with the initial guess, $v_k^{new}$.
   (Post-smoothing)
```

4.1.5 A note on implementing the operators

The fine-grid operator, A_k, the coarse-grid operator, A_{k-1}^G, and the restriction operator, R, are expensive to implement using Equations 8, 14 and 16, respectively. Instead of using these operators, we can solve an equivalent problem using the matrices $\tilde{\mathbf{A}}_\mathbf{k}$, $\tilde{\mathbf{A}}_\mathbf{k-1}$ and $\mathbf{P}^T$ (Equations 8 and 12). We state the algorithm for the two-level case in Algorithm 14. This scheme can be extended to construct the other standard multigrid schemes, namely the V, W and FMV cycles [24, 25].

4.2 Implementation

In Section 4.2.1, we describe an algorithm for constructing coarse octrees starting with an arbitrary 2:1 balanced fine-grid octree. This sequence of octrees gives rise to a sequence of nested finite element spaces that can be used in the multigrid algorithm presented in Section 4.1. In Section 4.2.2, we describe the matrix-free implementation of the restriction and prolongation operators derived in Section 4.1. Finally, we end this section with a note on variable-coefficient operators.

4.2.1 Global coarsening

Starting with the finest octree, we iteratively construct a hierarchy of complete, balanced, linear octrees such that every octant in the k-th octree is either present in the $k + 1$-th octree or all of its eight children are present (Figures 27(a) - 27(c)).

We construct the k-th octree from the $k+1$-th octree by replacing every set of eight siblings by their parent. This algorithm is based on the fact that in a sorted linear octree, each of the 7 successive elements following a "Child-0" element is either one of its siblings or a descendant of its siblings. Let i and j be the indices of any two successive Child-0 elements in the $k+1$-th octree. We have the following 3 cases: (a) $j < (i+8)$, (b) $j = (i+8)$ and (c) $j > (i+8)$. In the first case, the elements with indices in the range $[i, j)$ are not coarsened. In the second case, the elements with indices in the range $[i, j)$ are all siblings of each other and are replaced by their parent. In the last case, the elements with indices in the range $[i, (i+7)]$ are all siblings of each other and are replaced by their parent. The elements with indices in the range $[(i+8), j)$ are not coarsened. The pseudocode for the sequential implementation of the coarsening algorithm is given in Algorithm 15.

One-level coarsening is an operation with $\mathcal{O}(N)$ work complexity, where N is the number of leaves in the $k+1$-th octree. It is easy to parallelize and has an $\mathcal{O}(\frac{N}{n_p})$ parallel time complexity, where n_p is the number of processors. The main parallel operations are two circular shifts; one clockwise and another anti-clockwise. The message in each case is just 1 integer: (a) the index of the first Child-0 element on each processor and (b) the number of elements between the last Child-0 element on any processor and the last element on that processor. While we communicate these messages in the background, we simultaneously process the elements in between the first and last Child-0 elements on each processor. The pseudocode for the parallel implementation of the coarsening algorithm is given in Algorithm 16.

The operation described above may produce 4:1 balanced octrees[3] instead of 2:1 balanced octrees. Hence, we balance the result using the algorithm described in Chapter 2. Although there is only one level of imbalance that we need to correct, the imbalance can still affect octants that are not in its immediate vicinity. This is a result of the "*ripple effect*". Even with just one level of imbalance, a ripple can still propagate across many processors.

[3] The input is 2:1 balanced and we coarsen by at most one level in this operation. Hence, this operation will only introduce one additional level of imbalance resulting in 4:1 balanced octrees.

Algorithm 15 Sequential Coarsening

Input: A sorted, complete, linear fine octree (F).
Output: A sorted, complete linear coarse octree (C).
Note: This algorithm can also be used with a contiguous subset of F, provided the first element of this subset is a Child-0 element and the last element of this subset is either the last element of F or the element that immediately precedes a Child-0 element. The output in this case will be the corresponding contiguous subset of C.

```
$C \leftarrow \emptyset$
$\mathcal{I}_1 \leftarrow 0$
while $(\mathcal{I}_1 < \textbf{len}(F))$
        Find $\mathcal{I}_2$ such that $\textbf{Child}-\textbf{Number}(F[\mathcal{I}_2]) = 0$ and
        $\textbf{Child}-\textbf{Number}(F[k]) \neq 0 \quad \forall \quad \mathcal{I}_1 < k < \mathcal{I}_2$.
        if no such $\mathcal{I}_2$ exists
           $\mathcal{I}_2 \leftarrow \textbf{len}(F)$
        end if
        if $\mathcal{I}_2 \geq (\mathcal{I}_1 + 8)$
           $C.\textbf{push\_back}(\textbf{Parent}(F[\mathcal{I}_1]))$
           if $\mathcal{I}_2 > (\mathcal{I}_1 + 8)$
              $C.\textbf{push\_back}(F[\mathcal{I}_1 + 8], F[\mathcal{I}_1 + 9], \ldots, F[\mathcal{I}_2 - 1])$
           end if
        else
           $C.\textbf{push\_back}(F[\mathcal{I}_1], F[\mathcal{I}_1 + 1], \ldots, F[\mathcal{I}_2 - 1])$
        end if
        $\mathcal{I}_1 \leftarrow \mathcal{I}_2$
end while
```

Algorithm 16 Parallel Coarsening
(As executed by processor P)

Input: A distributed, globally sorted, complete, linear fine octree (F).
Output: A distributed, globally sorted, complete, linear coarse octree (C).
Note: We assume that $\mathbf{len}(F) > 8$ on each processor.

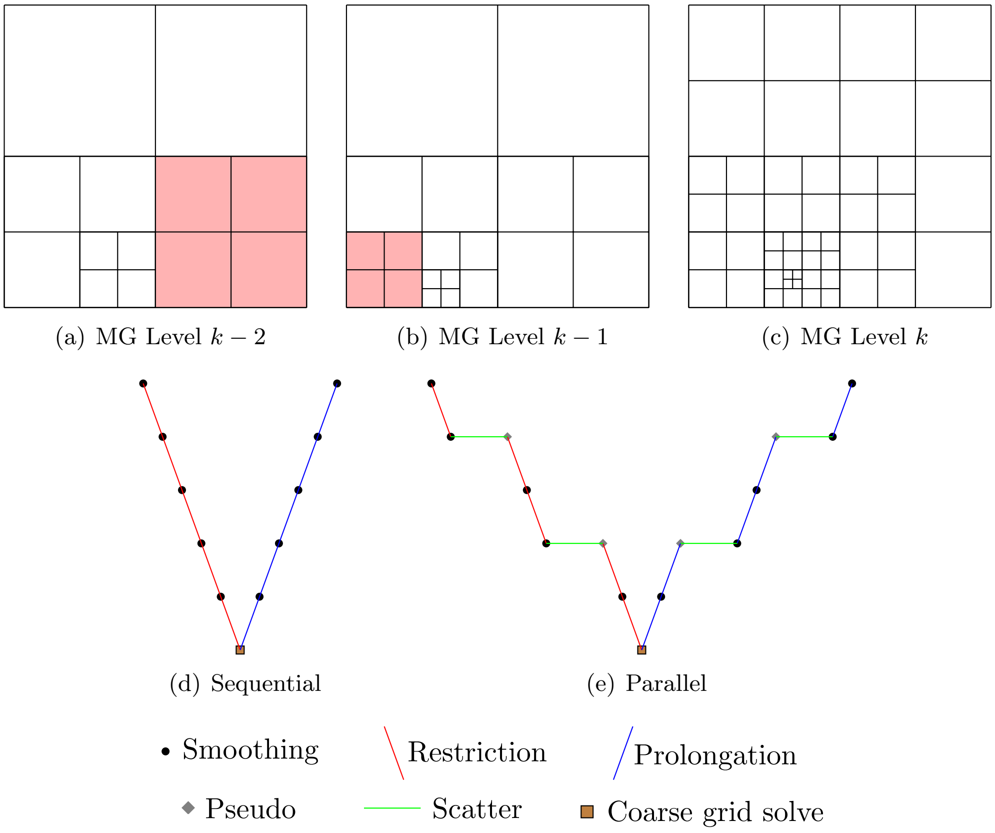
```
$C \leftarrow \emptyset$
Find $\mathcal{I}_f$ such that $\mathbf{Child-Number}(F[\mathcal{I}_f]) = 0$ and
$\mathbf{Child-Number}(F[k]) \neq 0 \quad \forall \quad 0 \leq k < \mathcal{I}_f$.
**if** no such $\mathcal{I}_f$ exists on P
  $M_f \leftarrow -1$ ; $M_l \leftarrow -1$
**else**
  Find $\mathcal{I}_l$ such that $\mathbf{Child-Number}(F[\mathcal{I}_l]) = 0$ and
  $\mathbf{Child-Number}(F[k]) \neq 0 \quad \forall \quad \mathcal{I}_l < k < \mathbf{len}(F)$.
  $M_f \leftarrow \mathcal{I}_f$ ; $M_l \leftarrow (\mathbf{len}(F) - \mathcal{I}_l)$
**end if**
**if** P is not the first processor
  Send $M_f$ to the previous processor (P-1)
  using an non-blocking MPI send.
**end if**
**if** P is not the last processor
  Send $M_l$ to the next processor (P+1)
  using an non-blocking MPI send.
**else if** $M_f > -1$
      $\mathcal{I}_l \leftarrow \mathbf{len}(F)$
**end if**
**if** $M_f > -1$
  Coarsen the list $\{F[\mathcal{I}_f], F[\mathcal{I}_f + 1], \ldots, F[\mathcal{I}_l - 1]\}$
  and store the result in $C$. **(Algorithm 15)**
**end if**
**if** P is not the first processor
  Receive $\mathcal{I}_p$ from the previous processor (P-1).
  Process octants with indices $< \mathcal{I}_f$. **(Algorithm 17)**
**end if**
**if** P is not the last processor
  Receive $\mathcal{I}_n$ from the next processor (P+1).
  Process octants with indices $\geq \mathcal{I}_l$. **(Algorithm 18)**
**end if**
```

Algorithm 17 Coarsening the first few octants on processor P (Subcomponent of Algorithm 16)

```
if $\mathcal{I}_p \geq 0$ and $M_f \geq 0$
    if $(\mathcal{I}_p + \mathcal{I}_f) \geq 8$
        $\mathcal{I}_c \leftarrow \max(0, (8 - \mathcal{I}_p))$
        $C$.push_front($F[\mathcal{I}_c], F[\mathcal{I}_c + 1], \ldots, F[\mathcal{I}_f - 1]$)
    else
        $C$.push_front($F[0], F[1], \ldots, F[\mathcal{I}_f - 1]$)
    end if
else
    if $M_f < 0$
        if $\mathcal{I}_p < 0$ or $\mathcal{I}_p \geq 8$
            $C \leftarrow F$
        else
            $\mathcal{I}_c \leftarrow (8 - \mathcal{I}_p)$
            $C$.push_front($F[\mathcal{I}_c], F[\mathcal{I}_c + 1], \ldots, F[\mathcal{I}_f - 1]$)
        end if
    else
        $C$.push_front($F[0], F[1], \ldots, F[\mathcal{I}_f - 1]$)
    end if
end if
```

Algorithm 18 Coarsening the last few octants on processor P (Subcomponent of Algorithm 16)

```
if $\mathcal{I}_n \geq 0$ and $M_l \geq 0$
    if $(\mathcal{I}_n + M_l) \geq 8$
        $C$.push_back(Parent($F[\mathcal{I}_l]$))
        if $M_l > 8$
            $C$.push_back($F[\mathcal{I}_l + 8], F[\mathcal{I}_l + 9], \ldots, F[$len$(F) - 1]$)
        end if
    else
        $C$.push_back($F[\mathcal{I}_l], F[\mathcal{I}_l + 1], \ldots, F[$len$(F) - 1]$)
    end if
else
    if $M_l \geq 0$
        $C$.push_back(Parent($F[\mathcal{I}_l]$))
        if $M_l > 8$
            $C$.push_back($F[\mathcal{I}_l + 8], F[\mathcal{I}_l + 9], \ldots, F[$len$(F) - 1]$)
        end if
    end if
end if
```


(a) MG Level $k-2$

(b) MG Level $k-1$

(c) MG Level k

(d) Sequential

(e) Parallel

Figure 27: `(a)-(c)` Quadtree meshes for three successive multigrid levels. The shaded octants in (a) and (b) were not coarsened because doing so would have violated the 2:1 balance constraint. `(d)` A V-cycle where the meshes at all multigrid levels share the same partition and `(e)` A V-cycle where not all meshes share the same partition. Some meshes do share the same partition and whenever the partition changes a pseudo mesh is added. The pseudo mesh is only used to support intergrid transfer operations and smoothing is not performed on this mesh.

The sequence of octrees constructed as described above has the property that non-hanging vertices in any octree remain non-hanging in all the finer octrees as well. Hanging vertices on any octree could either become non-hanging on a finer octree or remain hanging on the finer octrees too. In addition, an octree can have new hanging as well as non-hanging vertices that are not present in any of the coarser octrees.

4.2.2 Intergrid transfer operations

To implement the intergrid transfer operations in Algorithm 14, we need to find all non-hanging fine-grid vertices that lie within the support of each coarse-grid shape function. This is trivial on regular grids, but for non-uniform grids it can be quite expensive; especially for parallel implementations. Fortunately, for a hierarchy of octree meshes constructed as described in Section 4.2.1, these operations can be implemented quite efficiently.

As seen in Section 4.1.5, the restriction matrix is the transpose of the prolongation matrix. We do not construct these matrices explicitly; we implement a matrix-free scheme using MatVecs. The MatVecs for the restriction and prolongation operators are very similar. In both cases, we loop over the coarse and fine grid octants simultaneously. For each coarse-grid octant, the underlying fine-grid octant could either be the same as itself or be one of its eight children (Section 4.2.1). We identify these cases and handle them separately. The main operation within the loop is selecting the coarse-grid shape functions that do not vanish within the current coarse-grid octant and evaluating them at the non-hanging fine-grid vertices that lie within this coarse-grid octant. These form the entries of the restriction and prolongation matrices (Equation 12).

4.2.2.1 Alignment of grids

To parallelize the intergrid transfer operations, we need the coarse and fine grid partitions to be "*aligned*". By aligned we require the following two conditions to be satisfied:

- If an octant exists both in the coarse and fine grids, then the same processor must "*own*" this octant on both the meshes.
- If an octant's children exist in the fine-grid, then the same processor must own this octant on the coarse mesh and all its 8 children on the fine mesh.

In order to satisfy these conditions, we first compute the partition on the coarse-grid and then impose it on the finer grid. In general, it might not be possible or desirable to use the same partition for all the multigrid levels. For example, one problem with maintaining a single partition across all multigrid levels is that the coarser multigrid levels might be too sparse to be distributed across all the processors. As explained in Section 4.2.4, we enforce a minimum grain size (elements per processor) for all grids and this limits the number of processors used for each grid. Another reason to use different partitions for the different grids is to get better load distribution across the processors for the smoothing operation.

Hence, we allow certain multigrid levels to be partitioned differently than others.[4] When a transition in the partitions is required, we duplicate the octree in question and let one of the duplicates share the same partition as that of its immediate finer multigrid level and let the other one share the same partition as that of its immediate coarser multigrid level. We refer to one of these duplicates as the "*pseudo*" mesh (Figure 27(e)). The pseudo mesh is only used to support intergrid transfer operations (Smoothing is not performed on this mesh). On these multigrid levels, the intergrid transfer operations include an additional step referred to as "*Scatter*", which just involves re-distributing the values from one partition to another. We also want to reduce the number of pseudo meshes in order to lower setup costs and to lower communication costs by avoiding Scatter operations. Hence, we do the following checks to avoid pseudo meshes if possible:

- If a fine grid can use more processors than its immediate coarse grid, we first check the increase in average grain size if the fine grid used the same number of processors as the coarse grid. If this increase is small, we restrict the number of processors on the fine grid to be the same as that for the coarse grid.
- We check the load imbalance on each fine grid if it were to use the same partition as its immediate coarse grid. If this imbalance is below a user-specified threshold, we use the same partition for the fine grid and its immediate coarse grid.

4.2.2.2 Matvecs for restriction and prolongation

One of the challenges with implementing the MatVec for the intergrid transfer operations is that as we loop over the octants we must keep track of the pairs of coarse and fine grid vertices that were visited already. In order to implement this MatVec efficiently, we make use of the following observations.

- Every non-hanging fine-grid vertex is shared by at most eight fine-grid elements, excluding the elements whose hanging vertices are mapped to this vertex.

[4] It is also possible that some processors are idle on the coarse-grids, while no processor is idle on the finer grids.

- Each of these eight fine-grid elements will be visited only once within the Restriction and Prolongation MatVecs.
- Since we loop over the coarse and fine elements simultaneously, there is a coarse octant associated with each of these eight fine octants. These coarse octants (maximum of eight) overlap with the respective fine octants.
- The only coarse-grid shape functions that do not vanish at the non-hanging fine-grid vertex under consideration are those whose indices are stored in the vertices of each of these coarse octants. Some of these vertices may be hanging, but they will be mapped to the corresponding non-hanging vertex. So, the correct index is always stored immaterial of the hanging state of the vertex.

We compute and store a mask for each fine-grid vertex. Each of these masks is a set of eight bytes, one for each of the eight fine-grid elements that surround this fine-grid vertex. When we visit a fine-grid octant and the corresponding coarse-grid octant within the loop, we read the eight bits corresponding to this fine-grid octant. Each of these bits is a flag to determine whether or not the respective coarse-grid shape function contributes to this fine-grid vertex. The overhead of using this mask within the actual MatVecs includes (a) the cost of a few bitwise operations for each fine-grid octant and (b) the memory bandwidth required for reading the eight-byte mask. The latter cost is comparable to the cost required for reading a material property array within the finite element MatVec (for a variable coefficient operator). The restriction and prolongation MatVecs are operations with $\mathcal{O}(N)$ work complexity and have an $\mathcal{O}(\frac{N}{n_p})$ parallel time complexity. Algorithm 19 lists the sequence of operations performed by a processor for the restriction MatVec. For simplicity, we do not overlap communication with computation in the pseudocode. In the actual implementation, we overlap communication with computation as described in Section 4.2.2.4. The following section describes how we compute these masks for any given pair of coarse and fine octrees.

Algorithm 19 PARALLEL RESTRICTION MATVEC
(AS EXECUTED BY PROCESSOR P)

Input: Fine vector (F), masks (M), pre-computed stencils (R_1) and (R_2), fine octree (O_f), coarse octree (O_c).
Output: Coarse vector (C).

```
1.  Exchange ghost values for $F$ and $M$ with other processors.
2.  $C \leftarrow 0$.
3.  for each $o^c \in O_c$
4.     Let $c^c$ be the child number of $o^c$.
5.     Let $h^c$ be the hanging type of $o^c$.
6.     Step through $O_f$ until $o^f \in O_f$ is found s.t.
       Anchor($o^f$) = Anchor($o^c$).
7.     if Level($o^c$) = Level($o^f$)
8.       for each vertex, $V_f$, of $o^f$
9.          Let $V_f$ be the $i$-th vertex of $o^f$.
10.         if $V_f$ is not hanging
11.           for each vertex, $V_c$, of $o^c$
12.              Let $V_c$ be the $j$-th vertex of $o^c$.
13.              If $V_c$ is hanging, use the corresponding
                 non-hanging vertex instead.
14.              if the $j$-th bit of $M(V_f, i) = 1$
15.                $C(V_c) = C(V_c) + R_1(c^c, h^c, i, j)F(V_f)$
16.              end if
17.           end for
18.         end if
19.      end for
20.    else
21.        for each of the 8 children of $o^c$
22.           Let $c^f$ be the child number of $o^f$, the child of $o^c$
              that is processed in the current iteration.
23.           Perform steps 8 to 19 by replacing $R_1(c^c, h^c, i, j)$
              with $R_2(c^f, c^c, h^c, i, j)$ in step 15.
24.        end for
25.    end if
26. end for
27. Exchange ghost values for $C$ with other processors.
28. Add the contributions recieved from other processors
    to the local copy of $C$.
```

4.2.2.3 Computing the "masks" for restriction and prolongation

Each non-hanging fine-grid vertex has a maximum[5] of 1758 unique locations at which a coarse-grid shape function that contributes to this fine vertex could be rooted. Each of the vertices of the coarse-grid octants that overlap with the fine-grid octants surrounding this fine-grid vertex, can be mapped to one of these 1758 possibilities. It is also possible that some of these vertices are mapped to the same location. When we compute the masks described earlier, we want to identify these many-to-one mappings and only one of them is selected to contribute to the fine-grid vertex under consideration.

Now, we briefly describe how we identified these 1758 cases. We first choose one of the eight fine-grid octants surrounding a given fine-grid vertex as a reference element. Without loss of generality, we pick the octant whose anchor is located at the given fine vertex. Now the remaining fine-grid octants could either be the same size as the reference element, or be half the size or twice the size of the reference element. This simply follows from the 2:1 balance constraint. Further, each of these eight fine-grid octants could either be the same as the overlapping coarse-grid octant or be any of its eight children. Moreover, each of these coarse-grid octants that overlap the fine-grid octants under consideration could belong to any of the 8 child number types, each of which could further be of any of the 18 hanging configurations. Taking all these possible combinations into account, we can locate all the possible non-hanging coarse-grid vertices around a fine-grid vertex. Note that the child numbers, the hanging vertex configurations, and relative sizes of the eight fine-grid octants described above are not mutually independent. Each choice of child number, hanging vertex configuration and size for one of the eight fine-grid octants imposes numerous constraints on the respective choices for the other elements. Listing all possible constraints is unnecessary for our purposes; we simply assume that the choices for the eight elements under consideration are mutually independent. This computation can be done offline and results in a weak upper bound of 1758 unique non-hanging coarse-grid locations around any fine-grid vertex.

[5] This is a weak upper bound.

We cannot compute the masks offline since this depends on the coarse and fine octrees under consideration. To do this computation efficiently, we employ a "*PreMatVec*" before we actually begin solving the problem; this is only performed once for each multigrid level. In this PreMatVec, we use a set of 16 bytes per fine-grid vertex; 2 bytes for each of the eight fine-grid octants surrounding the vertex. In these 16 bits, we store the flags for each of the possibilities described above. These flags contain the following information.

- A flag to determine whether or not the coarse and fine grid octants are the same (1 bit).
- The child number of the current fine-grid octant (3 bits).
- The child number of the corresponding coarse-grid octant (3 bits).
- The hanging configuration of the corresponding coarse-grid octant (5 bits).
- The relative size of the current fine-grid octant with respect to the reference element (2 bits).

Using this information and some simple bitwise operations, we can compute and store the masks for each fine-grid vertex. The PreMatVec is an operation with $\mathcal{O}(N)$ work complexity and has an $\mathcal{O}(\frac{N}{n_p})$ parallel time complexity.

4.2.2.4 Overlapping communication with computation

Finally, we overlap computation with communication for ghost values even within the Restriction and Prolongation MatVecs. However, unlike the finite element MatVec the loop is split into three parts because we cannot loop over ghost octants since these octants need not be aligned across grids. Hence, each processor loops only over the coarse and the underlying fine octants that it owns. As a result, we need to both read as well as write to ghost values within the MatVec. The steps involved are listed below:

1. Initiate non-blocking MPI sends for ghost-values from the input vector.
2. Loop over some of the coarse and fine grid elements that are present in the interior of the processor domains. These elements do not share any vertices with other processors.

3. Receive the ghost-values sent from other processors in step 1.

4. Loop over the coarse and fine grid elements that share at least one of its vertices with a different processor.

5. Initiate non-blocking MPI sends for ghost-values in the output vector.

6. Loop over the remaining coarse and fine grid elements that are present in the interior of the processor domains. Note in step 2, we only iterated over some of these elements. In this step, we iterate over the remaining elements.

7. Receive the ghost-values sent from other processors in step 5.

8. Add the values received in step 7 to the existing values in the output vector.

4.2.3 Handling variable-coefficient operators

One of the problems with geometric multigrid methods is that their performance deteriorates with increasing contrast in material properties [25, 36]. Section 4.1.5 shows that the direct coarse-grid discretization can be used instead of the Galerkin coarse-grid operator provided the same bilinear form, $a(u, v)$, is used both on the coarse and fine multigrid levels. This poses no difficulty for constant coefficient problems. For variable-coefficient problems, this means that the coarser grid MatVecs must be performed by looping over the underlying finest grid elements, using the material property defined on each fine-grid element. This would make the coarse-grid MatVecs quite expensive. A cheaper alternative would be to define the material properties for the coarser grid elements as the average of those for the underlying fine-grid elements. This process amounts to using a different bilinear form for each multigrid level and hence is a clear deviation from the theory. This is one reason why the convergence of the stand-alone multigrid solver deteriorates with increasing contrast in material properties. Coarsening across discontinuities also affects the coarse grid correction, even when the Galerkin condition is satisfied. Large contrasts in material properties also affect simple smoothers like the Jacobi smoother. The standard solution is to use multigrid as a preconditioner to the Conjugate Gradient (CG) method. We have conducted numerical experiments that demonstrate this for the Poisson problem. The method works well for smooth coefficients but it is not robust in the presence of discontinuous coefficients.

4.2.4 Minimum grain size required for good scalability

For good scalability of our algorithms, the number of elements in the interior of the processor domains must be significantly greater than the number of elements on the inter-processor boundaries. This is because communication costs are proportional to the number of elements on the inter-processor boundaries, and by keeping the number of such elements small we can keep our communication costs low. We use a heuristic to estimate the minimum grain size necessary to ensure that the number of elements in the interior of a processor is greater than those on its surface. In order to do this, we assume the octree to be a regular grid. Consider a cube that is divided into N^3 equal parts. There are $(N-2)^3$ small cubes in the interior of the large cube and $N^3-(N-2)^3$ small cubes touching the internal surface of the large cube. In order for the number of cubes in the interior to be more than the number of cubes on the surface, N must be $>= 10$. Hence, the minimum grain size per processor is estimated to be 1000 elements.

4.2.5 Summary

The sequence of steps involved in solving the problem defined in Section 4.1.1.1 is summarized below:

1. A "sufficiently" fine[6] 2:1 balanced complete linear octree is constructed using the algorithms described in Chapter 2.

2. Starting with the finest octree, a sequence of 2:1 balanced coarse linear octrees is constructed using the global coarsening algorithm (Section 4.2.1).

3. The maximum number of processors that can be used for each multigrid level without violating the minimum grain size criteria (Section 4.2.4) is computed.

4. Starting with the coarsest octree, the octree at each multigrid level is meshed using the

[6] Here the term sufficiently is used to mean that the discretization error introduced is acceptable.

algorithm described in Chapter 3. As long as the load imbalance across processors is acceptable and as long as the number of processors used for the coarser grid is the same as the maximum number of processors that can be used for the finer grid without violating the minimum grain size criteria, the partition of the coarser grid is imposed on to the finer grid during meshing. If either of the above two conditions is violated then the octree for the finer grid is duplicated; one of them is meshed using the partition of the coarser grid and the other is meshed using a fresh partition. The process is repeated until the finest octree has been meshed.

5. A restriction PreMatVec (Section 4.2.2) is performed at each multigrid level (except the coarsest) and the masks that will be used in the actual restriction and prolongation MatVecs are computed and stored.

The discrete system of equations is solved using the Conjugate Gradient algorithm preconditioned with the multigrid scheme.

4.3 Numerical experiments

In this section, we consider solving for $\mathbf{u}$ in Equation 18 and u in Equations 19, 20, 21 and 22. Equation 18 represents a 3-dimensional, linear elastostatics (vector) problem with isotropic and homogeneous Lamé moduli (μ and λ) and homogeneous Dirichlet boundary conditions. Equations 19 through 22 represent 3-dimensional, linear Poisson (scalar) problems with inhomogeneous material properties and homogeneous Neumann boundary conditions.

$$
\begin{aligned}
\mu\Delta\mathbf{u} + (\lambda+\mu)\nabla \text{ Div } \mathbf{u} &= \mathbf{f} \text{ in } \Omega \\
\mathbf{u} &= \mathbf{0} \text{ in } \partial\Omega \\
\mu = 1;\ \lambda = 4;\ \Omega &= [0,1]^3
\end{aligned}
\tag{18}
$$

$$
\begin{aligned}
-\nabla\cdot(\epsilon\nabla u) + u &= f \text{ in } \Omega \\
\hat{n}\cdot\nabla u &= 0 \text{ in } \partial\Omega \\
\epsilon(x,y,z) &= \left(1 + 10^6\left(\cos^2(2\pi x) + \cos^2(2\pi y) + \cos^2(2\pi z)\right)\right)
\end{aligned}
\tag{19}
$$

$$
\epsilon(x,y,z) = \begin{cases} 10^7 & \text{if } 0.3 \le x,y,z \le 0.6 \\ 1.0 & \text{otherwise} \end{cases}
\tag{20}
$$

$$
\epsilon(x,y,z) = \begin{cases} 10^7 & \text{if the index of the octant} \\ & \text{containing } (x,y,z) \text{ is divisible by} \\ & \text{some given integer } K \\ 1.0 & \text{otherwise} \end{cases}
\tag{21}
$$

$$
\epsilon(x,y,z) = \begin{cases} 10^7 & \text{if } (x,y,z) \in [0,0.5)\times[0,0.5)\times[0,0.5) \\ & \cup\,[0.5,1.0]\times[0.5,1.0]\times[0,0.5) \\ & \cup\,[0,0.5)\times[0.5,1.0]\times[0.5,1.0] \\ & \cup\,[0.5,1.0]\times[0,0.5)\times[0.5,1.0] \\ 1.0 & \text{otherwise} \end{cases}
\tag{22}
$$

We discretized these problems on various octree meshes generated using Gaussian and log-normal distributions.[7] Figures 28(a) and 28(b) respectively show samples of the Gaussian and log-normal distributions that were used in all our experiments. The number of elements in these meshes range from about 25 thousand to over 2 billion and were solved on up to 32000 processors on the Teragrid system: "Ranger" (63K Barcelona cores with Infiniband). Details for this system can be found in [122]. Our C++ implementation uses MPI, `PETSc` [10, 9] and `SuperLU_Dist` [81]. The runs were profiled using `PETSc`.

In this Section, we present the results from four sets of experiments: `(A)` a convergence

[7] In the following experiments, the octrees were not generated based on the underlying material properties.

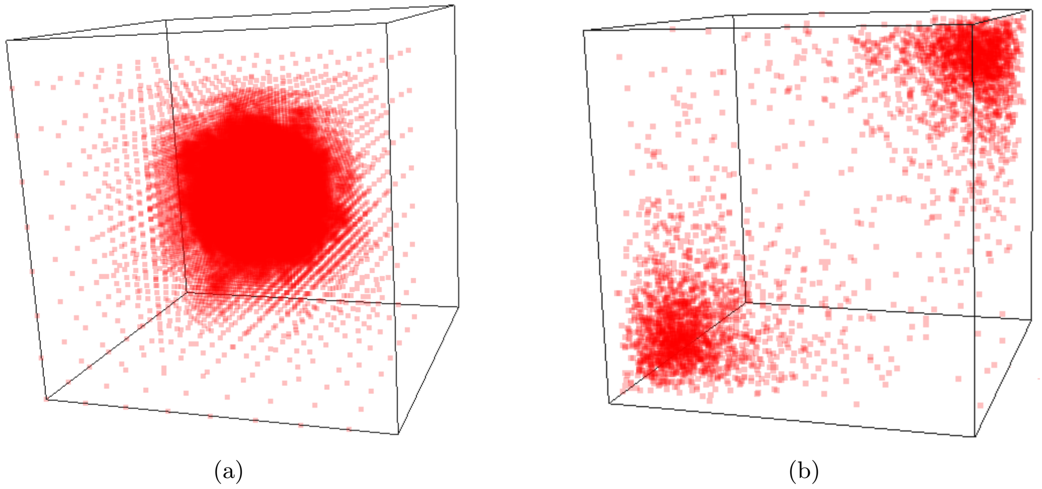


Figure 28: Samples of the point distributions used for the numerical experiments: (a) A Gaussian point distribution with mean at the center of the unit cube and (b) A log-normal point distribution with mean near one corner of the unit cube and it's mirror image about the main diagonal.

test, (B) a robustness test, (C) isogranular scalability, and (D) fixed size scalability. The parameters used in the experiments are listed below:

- For experiment (A), we set $u = \cos(2\pi x)\cos(2\pi y)\cos(2\pi z)$ and constructed the corresponding force (f).
- For experiments (B) through (D), we used a random solution (u) to construct the force (f).
- A zero initial guess was used in all experiments.
- One multigrid V-cycle was used as a preconditioner to the Conjugate Gradient (CG) method in all experiments. This is known to be more robust than the stand alone multigrid algorithm for variable-coefficient problems [123].
- The damped Jacobi method was used as the smoother at each multigrid level.
- SuperLU_Dist [81] was used to solve the coarsest grid problem in all cases.
- In order to minimize communication costs, the coarsest grid used fewer processors than the finer grids. This keeps the setup cost for SuperLU_Dist low.

Table 5: L^2 norm of the error between the true solution and its finite element approximation for the variable coefficient problem (Equation 19). The sequence of meshes used in this experiment were constructed by using a base discretization of ≈ 0.25M elements generated using a Gaussian point distribution followed by successive uniform refinements of the coarse elements of this mesh.

Max. Element Size (h_{max})	1/16	1/32	1/64	1/128	1/256
L^2 norm of the error	3.98×10^{-3}	9.62×10^{-4}	2.46×10^{-4}	6.18×10^{-5}	1.56×10^{-5}

4.3.1 Convergence test

In the first experiment, a base discretization of approximately ≈ 0.25M elements generated using the Gaussian distribution was used to solve the variable-coefficient problem (Equation 19). We measured the L^2 norm of the error as a function of the maximum element size (h_{max}) by uniformly refining the coarse elements[8] in the base mesh. In Table 5, we report the L^2 norm of the error between the true solution and its finite element approximation for the sequence of meshes constructed as described above. A second order convergence is observed just as predicted by the theory.

4.3.2 Robustness test

In the second experiment, we tested the robustness of the multigrid solver in the presence of strong jumps in the material properties. We discretized Equations 21 and 22 on an uniform octree with about 2M elements and measured the convergence rate for different values of K. Six multigrid levels were used for these problems. In Table 6, we report the number of iterations that were required to reduce the 2-norm of the residual in Equation 21 by a factor of 10^{-8} for different values of K; the number of jumps decreases as K increases. It is apparent that the solver is quite sensitive to the number of jumps. However, there are other factors that determine the overall performance of the solver. For example, it only takes 7 iterations to solve Equation 22 to the same tolerance; although there are more number of jumps in Equation 22 than Equation 21 for $\log_2 K = 10, 13, 16$ or 19. While the fine grid material properties in Equation 22 are represented exactly on all coarser grids, the fine grid material properties in Equation 21 are not represented accurately on any of the coarse

[8] Any element whose length is greater than h_{max}.

Table 6: The number of iterations required to reduce the 2-norm of the residual in Equation 21 by a factor of 10^{-8} for different values of K, a parameter that controls the frequency of jumps. A regular grid with 128 elements in each dimension was used for this experiment.

$\log_2 K$	1	4	7	10	13	16	19
Its.	119	18	25	55	66	43	17

grids. This would explain why coarse grid correction works better for Equation 22 than for Equation 21. The results of this experiment show that the current scheme is not robust in the presence of discontinuous coefficients.

4.3.3 Parallel scalability results

We tested the scalability of our implementation on the TeraGrid system: Ranger. In all the fixed-size (strong) and iso-granular (weak) scalability results, the reported times for each component are the maximum values for that component across all the processors. Hence, in some cases the total time[9] is lower than the sum of the individual components. We also report the theoretical predictions[10] for the total setup and solve times. This was computed using the asymptotic complexity estimates for the setup ($\mathcal{O}(\frac{N}{n_p}\log\frac{N}{n_p}) + \mathcal{O}(n_p \log n_p)$) and solve ($\mathcal{O}(\frac{N}{n_p}) + \mathcal{O}(\log n_p)$) times. The coefficients in the expressions for the complexity were computed so that the sum of squares of the deviation between the theoretical estimates and the actual data is minimized. While determining these coefficients, we skipped the last data point (corresponding to the greatest number of processors) in each experiment. This was done so that we could use our model to predict the value for the last data point and compare our predictions with the observed results. The number of multigrid levels and the total number of meshes generated for each case is also reported. Note that due to the addition of auxiliary meshes, the total number of meshes is greater than the number of multigrid levels. The setup cost includes the time for constructing the mesh for all the multigrid levels (including the finest), constructing and balancing all the coarser multigrid levels and setting up the intergrid transfer operators by performing one PreMatVec at each multigrid level. The time to create the work vectors for the MG scheme and the time to

[9] This is reported in bold face.

[10] This is reported within parenthesis just below the total setup and solve times.

build the coarsest grid matrix are also included in the total setup time, but are not reported individually since they are insignificant. "Scatter" refers to the process of transferring the vectors between two different partitions of the same multigrid level during the intergrid transfer operations, required whenever the coarse and fine grids do not share the same partition. The time spent in applying the Jacobi preconditioner, computing the inner-products within CG, and solving the coarsest grid problems using LU are all accounted for in the total solve time, but are not reported individually since they are insignificant. When we report `MPI_Wait()` times, we refer to synchronization for non-blocking operations during the Restriction, Prolongation and Finite Element MatVecs.

4.3.3.1 Isogranular (weak) scalability

Isogranular scalability analysis was performed by tracking the execution time while increasing the problem size and the number of processors proportionately. The results from isogranular scalability experiments on the octrees generated from Gaussian point distributions are reported in Tables 7, 8 and 9. Tables 7 and 8 report the results for the constant coefficient elasticity (Equation 18) problem for two different grain sizes and Table 9 reports the results for the variable-coefficient Poisson problem (Equation 19). The results from an isogranular scalability experiment for solving the variable-coefficient Poisson problem (Equation 19) on octrees generated from log-normal point distributions are reported in Table 10. There is little variation between the Gaussian distribution case and the log-normal distribution case. For the Gaussian distribution cases, the coarsest octant at the finest multigrid level was at level three; the level of the finest octant at the finest multigrid level for each case is reported in the tables. The octrees considered here are extremely non-uniform—roughly five-orders of magnitude variation in the leaf size. It is also quite promising that the setup costs are smaller than the solution costs, suggesting that the method is suitable for problems that require the construction and solution of linear systems of equations numerous times. The increase in running times for the large processor cases can be primarily attributed to poor load balancing. This is evident from `(a)` the imbalance in the number of elements per processor and `(b)` the time spent in calls to `MPI_Wait()`. These numbers are reported in

Table 7: Isogranular scalability for solving the constant coefficient linear elastostatics problem on a set of octrees with a grain size (on the finest multigrid level) of $30K$ (approx) elements per CPU (n_p) generated using a Gaussian distribution of points. A relative tolerance of 10^{-10} in the 2-norm of the residual was used. 11 iterations were required in each case, to solve the problem to the specified tolerance. The size of the problem is indicated in the "Elements" row, the "Max/Min elements" row gives the load imbalance across processors, the "MG levels" row indicates the number of multigrid levels (it differs from the number of "Meshes" because our algorithm duplicates meshes to allow for incompatible partitioning), "R+P" indicates restriction and prolongation costs, and "LU" is the coarse-grid solve. In the "Theory row", we report an estimate of the time required using the asymptotic analysis complexity using constants fitted by the runs on 12 – 12288 processors. The fine-level input octrees are highly non-uniform. The largest octants are at tree-level three and the smallest octants are at a tree-level reported in the "Finest Octant's level" row. All timings are reported in seconds.

CPUs	**12**	**48**	**192**	**768**	**3072**	**12288**	**32000**
Coarsening	0.33	0.56	0.95	1.66	2.18	4.39	1.90
Balancing	0.77	0.99	1.23	1.84	4.48	12.66	9.67
Meshing	0.973	1.44	1.82	3.32	15.09	32.89	21.67
R-setup	0.092	0.125	0.122	0.14	0.172	0.173	0.355
Total Setup (Theory)	**2.41** (4.94)	**3.22** (4.97)	**3.87** (5.25)	**6.51** (7.02)	**23.4** (15.48)	**53.21** (54.86)	**47.14** (148.39)
LU	0.189	0.4	0.017	0.171	0.543	0.015	0.0025
R + P	2.53	3.62	4.23	5.36	8.55	8.96	11.97
Scatter	3.11	6.49	8.59	13.13	16.98	21.15	27.88
FE Matvecs	51.63	57.73	60.20	64.58	68.78	69.87	66.91
Total Solve (Theory)	**54.82** (55.87)	**63.74** (61.73)	**66.14** (67.39)	**73.5** (73.60)	**80.96** (79.79)	**85.28** (86.06)	**89.77** (90.33)
Meshes	11	16	19	24	25	28	29
MG Levels	8	11	12	14	14	15	16
Elements	337.8K	1.34M	5.29M	21.15M	84.5M	338.3M	880.3M
Max/Min Elements	1.89	1.79	2.04	2.82	3.9	3.08	3.12
MPI_Wait	21.32	24.32	29.58	32.65	39.8	39.19	95.75
Finest Octant's Level	12	15	16	18	18	19	19

Table 8: Isogranular scalability for solving a linear elastostatics problem on a set of octrees with a grain size (on the finest multigrid level) of $80K$ (approx) elements per processor generated using a Gaussian distribution of points. A relative tolerance of 10^{-10} in the 2-norm of the residual was used. 11 iterations were required in each case, to solve the problem to the specified tolerance. All timings are reported in seconds.

CPUs	**12**	**48**	**192**	**768**	**3072**	**12288**	**32000**
Coarsening	0.85	1.27	2.18	3.09	4.17	6.28	4.01
Balancing	1.76	2.08	2.96	3.87	7.19	13.33	12.97
Meshing	2.41	3.21	6.53	7.4	21.06	33.41	34.03
R-setup	0.27	0.32	0.323	0.32	0.35	0.54	0.63
Total Setup (Theory)	**4.64** (10.24)	**5.95** (10.36)	**11.04** (10.69)	**12.85** (12.34)	**31.89** (20.29)	**54.69** (57.18)	**70.14** (144.99)
LU	0.025	0.02	0.517	0.048	4.92	0.019	0.041
R + P	6.22	7.89	12.87	13.8	14.87	56.2	30.16
Scatter	3.9	6.1	13.83	15.26	20.63	62.61	50.76
FE Matvecs	145.8	166.73	174.27	173.64	186.11	221.01	169.81
Total Solve (Theory)	**152.36** (152.35)	**175.2** (169.13)	**187.27** (185.13)	**188.38** (201.06)	**212.04** (217.24)	**242.37** (232.69)	**208.06** (244.25)
Meshes	13	16	22	25	27	29	30
MG Levels	10	11	12	14	15	16	16
Elements	986.97K	3.97M	15.87M	63.4M	253.8M	1.01B	2.64B
Max/Min Elements	1.66	1.9	2.44	2.43	2.73	2.88	2.89
MPI_Wait	53.95	80.23	97.86	88.1	97.83	173.82	118.1
Finest Octant's Level	14	15	17	18	19	20	20

Table 9: Isogranular scalability for solving the variable-coefficient Poisson problem (Equation 19) on the set of octrees with a grain size (on the finest multigrid level) of 0.25M elements (approx.) per processor (n_p) generated using a Gaussian distribution of points. The iterations were terminated when the 2-norm of the residual was reduced by a factor of 10^{-10}. 5 iterations were required in each case. All timings are reported in seconds.

CPUs	**1**	**4**	**16**	**64**	**256**	**1024**	**4096**
Coarsening	0.02	0.09	2.79	3.41	4.48	5.44	6.75
Balancing	0.34	2.32	4.66	5.23	6.18	7.96	8.92
Meshing	2.66	7.44	7.12	8.12	20.43	19.54	29.14
R-setup	0.48	1.04	0.85	0.88	1.01	0.99	1.04
Total Setup	**3.59**	**11.11**	**13.07**	**15.32**	**30.29**	**30.58**	**44.95**
LU	1.16	0.27	1.11	2.69	5.4	11.69	10.99
R + P	2.07	5.61	5.26	6.75	6.99	7.26	10.22
Scatter	0	0	0.11	0.32	2.55	3.65	6.79
FE Matvecs	20.37	43.92	37.46	39.97	40.64	40.96	52.53
Total Solve	**24.19**	**49.98**	**43.53**	**48.19**	**53.06**	**60.37**	**73.46**
Elements	239.4K	995.4K	3.97M	16.0M	64.4M	256.8M	1.04B
Vertices	151.7K	660.1K	2.68M	10.52M	42.0M	172.4M	702.9M
Meshes	4	7	8	9	15	17	19
MG Levels	4	7	7	7	8	9	10
Finest Octant's Level	8	14	14	16	18	19	21

Table 10: Isogranular scalability for solving the variable-coefficient Poisson problem (Equation 19) on a set of octrees with a grain size (on the finest multigrid level) of 25K elements (approx.) per processor (n_p) generated using a log-normal distributions of points located on two diagonally opposite corners of the unit cube. The iterations were terminated when the 2-norm of the residual was reduced by a factor of 10^{-10}. The levels of the coarsest and finest octants at the finest multigrid level are reported in the table. All timings are reported in seconds.

CPUs	**1**	**4**	**16**	**64**	**256**	**1024**
Coarsening	0.015	0.036	0.18	0.43	0.58	0.76
Balancing	0.03	0.16	0.4	0.72	0.89	5.99
Meshing	0.28	0.59	0.76	1.26	2.73	6.13
R-setup	0.059	0.092	0.08	0.102	0.12	0.14
Total Setup	**0.95**	**0.92**	**1.33**	**3.07**	**5.15**	**14.13**
LU	0.55	0.56	0.19	0.07	0.75	0.96
R + P	0.24	0.58	0.57	1.14	0.99	1.32
Scatter	0	0	0.08	0.46	0.88	1.39
FE Matvecs	2.05	3.8	3.47	5.42	4.45	5.52
Total Solve	**3.09**	**4.96**	**4.38**	**6.48**	**6.69**	**8.48**
CG Its.	5	5	5	7	6	7
Meshes	3	4	6	11	15	15
MG Levels	3	4	5	7	8	8
Finest Octant's Level	9	13	13	13	15	16
Coarsest Octant's Level	3	3	4	4	5	5
Elements	24.6K	99.3K	362.6K	1.42M	5.64M	22.4M
Vertices	17.4K	68.2K	243.3K	952.2K	3.79M	14.9M

Table 11: Fixed-size scalability for solving the variable-coefficient Poisson problem (Equation 19) on an octree with 31.9M elements generated from a Gaussian distribution of points. 8 multigrid levels were used. 5 iterations were required to reduce the 2-norm of the residual by a factor of 10^{-10}. 468 Matvecs, 72 of which are on the finest grid, were required. All timings are reported in seconds.

CPUs	**32**	**64**	**128**	**256**	**512**	**1024**
Coarsening	9.02	5.81	4.08	2.73	1.85	1.44
Balancing	15.02	9.03	5.91	3.83	2.43	1.73
Meshing	30.69	24.81	9.25	7.99	5.94	4.15
R-setup	3.64	1.97	0.94	0.56	0.3	0.19
Total Setup	**51.14**	**37.52**	**17.32**	**13.89**	**10.39**	**7.82**
LU	1.82	2.08	1.59	1.70	1.71	1.77
R + P	24.59	12.06	7.30	4.18	2.11	1.35
Scatter	0.25	1.62	0.61	0.89	1.45	1.46
FE Matvecs	159.0	77.94	41.28	25.11	10.93	5.99
Total Solve	**181.9**	**91.59**	**48.94**	**31.23**	**15.28**	**9.98**
Meshes	10	11	12	14	15	15

Tables 7 and 8 (in the introduction).[11] Load balancing is a challenging problem due to the following reasons:

- We need to make an accurate a-priori estimate of the computation and communication loads. It is difficult to make such estimates for arbitrary distributions.
- For the intergrid transfer operations, the coarse and fine grids need to be aligned. It is difficult to get good load balance for both grids, especially for non-uniform distributions.
- Partitioning each multigrid level independently to get good load balance for the smoothing operations would require the creation of an auxiliary mesh for each multigrid level and a scatter operation for each intergrid transfer operation at each multigrid level. This would increase the setup costs and the communication costs.

4.3.3.2 Fixed-size (strong) scalability

Fixed-size scalability was performed on the octrees generated from Gaussian and log-normal point distributions to compute the speedup when the problem size is kept constant and the

[11] We only report the Max/Min elements ratios for the finest multigrid level although the trend is similar for other multigrid levels as well.

Table 12: Fixed-size scalability for solving the variable-coefficient Poisson problem (Equation 19) on an octree with 22.4M elements generated using a log-normal distribution of points located on two diagonally opposite corners of the unit cube. 8 multigrid levels were used. 5 iterations were required to reduce the 2-norm of the residual by a factor of 10^{-10}. All timings are reported in seconds.

CPUs	**32**	**64**	**128**	**256**	**512**	**1024**
Coarsening	5.9	4.57	2.78	1.67	1.12	0.84
Balancing	9.9	6.52	4.12	2.51	1.75	1.70
Meshing	20.17	14.28	6.61	5.39	6.4	6.17
R-setup	2.29	1.47	0.64	0.34	0.25	0.14
Total Setup	**33.51**	**24.36**	**12.59**	**9.03**	**10.36**	**9.71**
LU	0.59	1.87	1.3	0.58	0.95	0.84
R + P	13.73	9.42	4.17	2.56	2.44	1.34
Scatter	0.2	0.57	0.37	0.69	1.51	1.31
FE Matvecs	99.76	63.01	27.35	14.45	12.44	5.88
Total Solve	**113.77**	**73.86**	**32.48**	**17.41**	**16.62**	**9.1**
Meshes	10	11	12	14	15	15

Table 13: Fixed-size scalability for solving the variable-coefficient Poisson problem (Equation 19) on an octree with 5.64M elements generated using a log-normal distribution of points located on two diagonally opposite corners of the unit cube. 8 multigrid levels were used. 5 iterations were required to reduce the 2-norm of the residual by a factor of 10^{-10}. All timings are reported in seconds.

CPUs	**32**	**64**	**128**	**256**	**512**	**1024**
Coarsening	2.75	2.11	0.99	0.59	0.41	0.35
Balancing	4.56	2.95	1.36	0.91	0.96	1.09
Meshing	8.46	4.95	2.68	2.68	2.61	2.59
R-setup	0.66	0.35	0.21	0.16	0.18	0.087
Total Setup	**15.54**	**9.59**	**5.52**	**4.55**	**5.21**	**7.43**
LU	1.08	0.89	0.82	0.17	0.69	0.79
R + P	4.57	2.89	1.68	0.95	0.81	0.72
Scatter	0.32	0.58	0.46	0.84	1.45	1.44
FE Matvecs	28.69	14.74	8.68	4.36	3.35	2.47
Total Solve	**35.61**	**18.49**	**11.45**	**5.65**	**5.73**	**5.12**
Meshes	11	12	13	15	15	15

number of processors is increased. The results from fixed size scalability experiments for the solving the variable-coefficient problem (Equation 19) on an octree with 32M (approx) elements generated from Gaussian point distribution are reported in Table 11. This experiment was repeated on octrees with 6M and 22M (approx) elements generated from log-normal point distributions and the corresponding results are reported in Tables 13 and 12, respectively. The results for the Gaussian and log-normal distributions are similar. We observe good speed-ups for the setup phase on up to 256 processors and the speed-ups begin to deteriorate beyond that. We believe that the surface computation (e.g. meshing for ghost elements) begins to dominate beyond 256 processors. Note that the number of meshes also grow with the number of processors. This is another reason why we don't observe ideal speed-ups for the setup phase. The speed-ups for the solve phase, although not ideal, seem to be quite good. Poor load balancing, which affects isogranular scalability on large processor counts, seems to be another factor that affects the speed-ups for the setup and solve phases in the fixed-size scalability experiments.

4.4 Conclusions

In this chapter, we described a parallel geometric multigrid method for solving elliptic partial differential equations using finite elements on octree-based discretizations. The features of the described method are summarized below:

- We automatically generate a sequence of coarse meshes from an arbitrary 2:1 balanced fine octree. We do not impose any restrictions on the number of meshes in this sequence or the size of the coarsest mesh. We do not require the meshes to be aligned and hence the different meshes can be partitioned independently to satisfy any user-defined constraint such as a limit on the load imbalance. Although, the process of constructing coarser meshes from a fine mesh is harder than iterative global refinements of a coarse mesh to generate a sequence of fine meshes; this is more practical since the fine mesh can be defined naturally depending on modeling restrictions, and/or physics of the problem as opposed to the coarse mesh, which is purely an artifact of the numerical method. It is also natural and more desirable to be able to control the fine mesh in an adaptive algorithm rather than controlling the coarse mesh.

- We demonstrated good scalability of our implementation and can solve problems with billions of elements on thousands of processors in less than 10 minutes. However, load balancing remains an open problem and this begins to affect our iso-granular scalability beyond a thousand processors. This is a difficult problem to tackle because there are many competing factors: Restriction, prolongation, scatters and MatVecs.
- Finally, we demonstrated that our implementation works well even on problems with variable coefficients.

There are two important extensions for the present work: higher-order discretizations and integration with domain-decomposition methods such as the Hierarchical Hybrid Grids (HHG) scheme described in [15]. The former will result in improved accuracy with fewer elements and the latter will help solve problems involving complicated geometries with fewer elements. The last point stems from the fact that using a single octree to mesh a domain is more restrictive than allowing the use of multiple octrees, each of which is only responsible for a part of the entire domain.

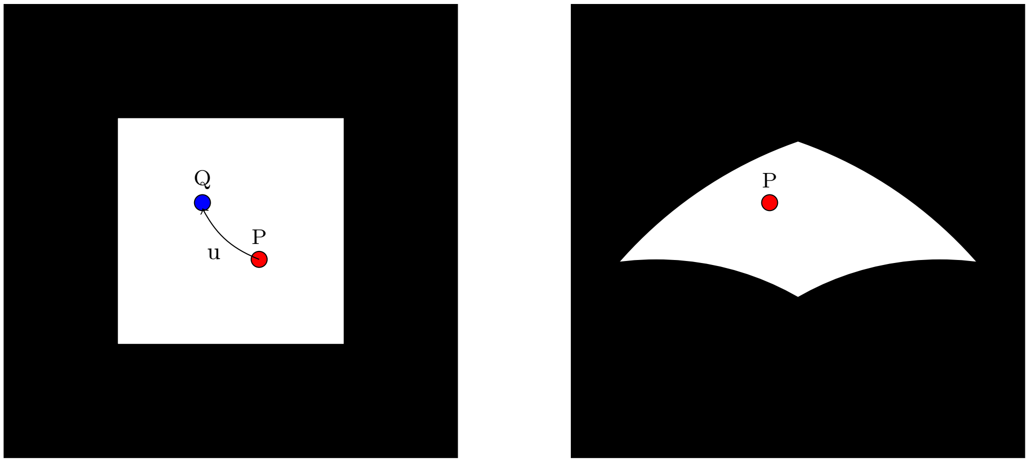
CHAPTER V

ELASTIC REGISTRATION USING OCTREES

In this chapter, we present a parallel algorithm for intensity-based elastic image registration. This is one of the most challenging problems in image processing. It is the process of overlaying two or more images of the same scene taken at different times, from different viewpoints, and/or by different sensors [145]. Specifically, we want to find a suitable transformation or mapping such that a transformed image becomes similar to another image [88]. This is illustrated in Figure 29. It is a pre-processing step for the following applications:

- Integrating complementary information contained in images of the same subject obtained using different modalities.
- Aligning temporal sequences of images to compensate for motion of the subject.
- Image guidance during surgery.
- Aligning images from multiple subjects in statistical studies.
- Comparing images taken at different stages of progression of a disease such as in tumor growth.

The image registration problem can be viewed as a non-convex optimization problem; such problems typically have multiple local optima [92]. Also, it is an ill-posed problem and we need to impose a regularization that constrains the displacement field. The choice of regularization is an important feature that can be used to distinguish between different registration algorithms. In this work, we will use the elastic deformation energy as the regularization and hence this procedure is known as "*elastic registration*". This choice is motivated by the fact that many biological materials that are imaged are elastic. In particular, we will use the linear theory of elasticity to derive the deformation energy. The optimality condition for the elastic registration problem is a nonlinear partial differential equation



Moving Image

Fixed Image

Figure 29: Illustration of the image registration problem. The point “Q” on the “moving” image and the point “P” on the “fixed” image have the same physical coordinates but they represent different material points. The points “P” on the fixed and moving images represent the same material points but have different physical coordiantes. We need to find the displacement “u” between the points “P” and “Q” on the moving image.

(PDE) and the corresponding linearization suffers from indefiniteness and ill-conditioning.

Registration methods can be broadly classified into two types: (a) Landmark/feature-based and (b) Intensity-based. In the former approach, few distinct features in the images are identified and the registration is performed by first establishing point correspondences between the features in the source and target images, followed by an interpolation or approximation scheme to map the remaining regions of the images. In contrast, intensity-based approach work directly with intensity values in the pixels/voxels. Typically, landmark-based approaches have the drawback of requiring manual intervention to select landmarks. Since medical images seldom have distinct features, it is difficult and error prone to identify landmarks. For this reason, intensity-based approaches are easier to automate. However, intensity-based approaches lack the anatomical information contained in features and are generally more computationally intensive than feature-based approaches. Hybrid approaches that combine intensity-based and feature-based criteria have also been proposed. In the present work, we adopt an intensity-based approach.

There are many different registration algorithms and all of them require the following components: (a) a metric to measure the similarity between any given pair of images, (b) a model for the desired transformation, (c) an interpolation scheme and (d) an optimizer. Some of the commonly used similarity metrics are (a) Sum of squared differences (SSD), (b) correlation coefficient, (c) correlation ratio and (d) mutual information [33, 96]. SSD is used when the registered images differ only by a Gaussian noise. Correlation coefficient is used when the registered images have a linear intensity relationship. Correlation ratio is used when the registered images have some nonlinear intensity relationship. Mutual information only assumes a probabilistic relationship between the intensities of the registered images. SSD and correlation coefficient are used for single modality[1] registration and the other metrics are used for multiple modality registration. We use the SSD metric in this work. The transformation models can be classified into two types: (a) Parametric and (b) Non-parametric. Parametric models such as rigid/affine or spline-based deformations are computationally efficient because of their small search space; and they have limited flexibility for the same reason. In the non-parametric case, the transformation model comes directly from the discretization scheme such as finite differencing or finite elements. They have greater flexibility, but are more expensive and need explicit regularization to constrain the deformation. D-linear and cubic spline interpolation are typically used in registration algorithms to interpolate the discrete images. D-linear interpolation is not suitable for gradient-based optimization algorithms for image registration because these algorithms require the gradient of the image and d-linear image approximations are not continuously differentiable. On the other hand, spline approximations can be differentiated and hence are typically used in gradient-based algorithms. In this work, we use piecewise tricubic polynomial approximations to the images. The advantage of this interpolation scheme is that the coefficients are simply the image intensity and its first derivatives at the grid points, which can be computed quickly using finite differences and its easy to parallelize. Different types of optimizers have been used for image registration; these range from derivative-free approaches [37, 89, 131] to quasi-Newton [58, 59] and inexact-Newton type algorithms [60]. In this work, we use the Gauss-Newton algorithm for solving the optimization problem.

[1] Modality refers to the technique used to acquire the images. Some popular modalities include ultrasound, magnetic resonance imaging and computed tomography.

Elastic image registration is a computationally intensive problem that involves the numerical solution of large linear systems of equations several times. To reduce the computation time for registration, we considered a combination of various techniques. To reduce the amount of processed data we used a non-uniform discretization scheme. Such a discretization scheme would also be useful for further extensions of the work that will include adaptive mesh refinement proceedures. Due to the sheer size of the problem and storage limitations we considered matrix-free schemes and parallel implementations. Since, generic unstructured meshes are not suitable for matrix-free schemes, we considered octree discretizations instead. Moreover, generic unstructured meshing schemes tend to break down due to bad element quality during the remeshing step. We used iterative solvers since direct solvers do not work with matrix-free schemes and moreover they are known to not scale well. The convergence rates of most iterative solvers deteriorate with the condition number of the matrix to be inverted. The matrices that need to be inverted to solve the registration problem suffer from ill-conditioning and so it was very important to address this problem. Multigrid methods are known to be robust for such ill-conditioned systems; so, we used the geometric multigrid algorithm described in Chapter 4 to solve the linear system formed in each Gauss Newton iteration. Since this has a low setup cost it was ideal for this application in which we need to setup and solve numerous linear systems of equations. We demonstrate the performance of our method on synthetic as well as clinical images. Also, we demonstrate the scalability of our implementation on up to 2048 processors on the Sun Constellation Linux Cluster "Ranger" at the Texas Advanced Computing Center (TACC).

Contributions. The main contributions of this work are as follows:

- We use parallel octree discretizations to reduce the computation time for the registration problem. This implementation will also allow us to register images that are too large to fit on a single processor's memory.

- We use a multigrid algorithm to solve the linear system that arises in each optimization iteration. In particular, we use a matrix-free geometric multigrid algorithm, which has lower setup costs compared to its algebraic counterpart. This is significant because we need to setup the linear system of equations for the multigrid scheme in every optimization iteration.

Limitations. There are a few limitations in the proposed framework:

- The linear theory of elasticity is only valid for small deformations. Other regularization approaches may be more appropriate for large deformations.
- We do not incorporate any biophysical information to additionally constrain the deformation. It has been suggested that incorporating such information will provide intelligent priors and reduce the ill-posedness of the registration problem [117].
- We do not use adaptive integration in our present implementation. Instead, we fix the order of the Gauss quadrature rule for integration a-priori. The use of adaptive integration can further reduce the computation costs for evaluating the objective function and gradient.

Organization of the chapter. The rest of this chapter is organized as follows. We give the mathematical problem formulation in Section 5.1 and describe the octree discretization and image interpolation schemes in Sections 5.2 and 5.3, respectively. Section 5.4 describes the linear and nonlinear solvers used to solve the optimization problem. Finally, in Section 5.5 we present the results from using the proposed algorithm for registering synthetic as well as clinical images.

5.1 Problem description

Given two images, $S(x)$ and $T(x)$, we want to find a displacement field, $u(x)$, that is a solution to the following minimization problem:

$$\begin{aligned} \min_u \mathcal{J}(u) &= \frac{1}{2}\int [S - T(u)]^2 \, dx + \frac{\gamma}{2} a(u,u) \\ a(u,v) &= -\int v \cdot (\Delta u + (\lambda+1)\nabla div\, u) \; dx \end{aligned} \tag{23}$$

S is referred to as the fixed image, T is referred to as the moving image and γ is the regularization parameter. $a(u, u)$ represents the elastic potential energy due to the deformation and λ is the Lamé constant. In this framework, we are trying to find a displacement field that simulataneously minimizes the dissimilarity between the fixed and moving images and the associated deformation energy. The optimality condition for this problem is given by Equation 24:

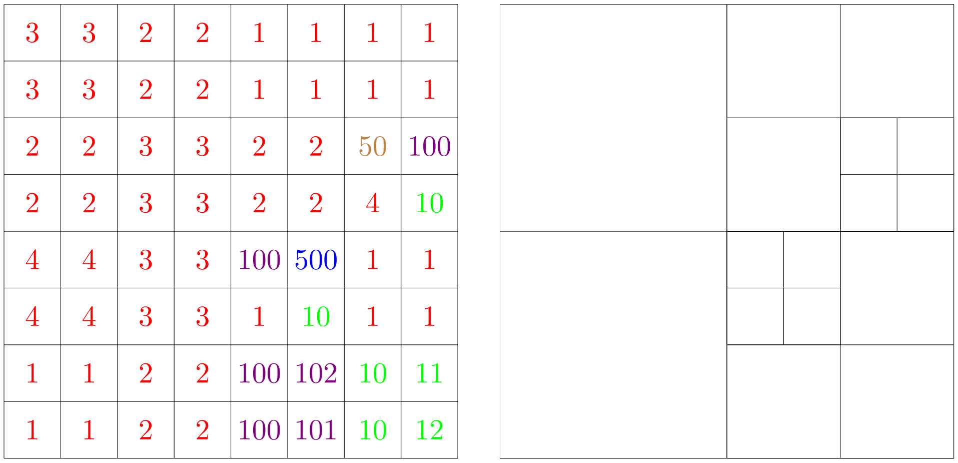
$$g(u) \cdot v = \gamma \, a(u, v) - \int (S - T(u)) \, \nabla T(u) \cdot v \, dx = 0 \; \forall v \tag{24}$$

where, $g(u)$ represents the gradient of the objective function. This is the weak form of a nonlinear coupled partial differential equation. We need to specify appropriate boundary conditions to solve this equation; we chose to enforce homogeneous Dirichlet boundary conditions in this work.

5.2 Octree discretization

In this section, we describe how we construct the parallel octree discretization from the high resolution fixed and moving images discretized on an uniform grid. We assume that the number of elements in each dimension of the uniform grid is a power of 2. Note that we only use the octree discretization for the displacements; we continue to use the original uniform grid representation for the given images. To make the distinction between the two discretizations clear, we use the term "octants" to refer to elements in the octree discretization and the term "voxels" to refer to elements in the original regular grid discretization.

We first construct two parallel complete linear octrees; one for the fixed image and another for the moving image. The uniform grids for the input images can be viewed as octrees in which all the octants are at the same octree-level. We coarsen these octrees by replacing every set of 8 siblings by their parent as long as the maximum difference between the image values of the siblings is less than an user-specified threshold (δ). After coarsening, we set the image value of the parent to be the average of the values of its children and repeat the process until further coarsening is not possible. This is illustrated in Figure 30. Next, we merge the two octrees and linearize the result using the algorithm described in Chapter



3	3	2	2	1	1	1	1
3	3	2	2	1	1	1	1
2	2	3	3	2	2	50	100
2	2	3	3	2	2	4	10
4	4	3	3	100	500	1	1
4	4	3	3	1	10	1	1
1	1	2	2	100	102	10	11
1	1	2	2	100	101	10	12

(a) Regular Grid

(b) Linear Octree

Figure 30: (a) A regular grid image and (b) the linear octree constructed by coarsening the regular grid image.

2.[2] We then use the algorithm described in Chapter 2 to enforce the 2:1 balance constraint. We construct an octree mesh from the 2:1 balanced octree using the algorithm described in Chapter 3. We discretize the optimization problem given in Equation 23 using trilinear finite elements on this octree mesh as described in Chapter 3.

5.3 Interpolation

Since the input images are discrete, we need to construct continuous approximations to the images that can be used to evaluate their intensity values at arbitrary locations within the domain. Such evaluations are necessary for computing the integrals in Equation 23 and Equation 24 numerically. In this work, we use a gradient-based optimization algorithm and to compute the gradient, $g(u)$, (Equation 24) we also need derivatives of the approximation of the moving image, $T(x)$. Although computationally efficient, piecewise trilinear image approximations are not suitable for our purposes as they are not continuously differentiable. Instead, we used higher order polynomial approximations. In this section, we describe how we construct continuously differentiable approximations to the images using piecewise tricubic polynomials. First, we define 32 tricubic polynomials in the reference voxel's local

[2] Our implementation is capable of simultaneously registering multimodal images. For simplicity, we only explain the monomodal case. For the multimodal case, an octree is constructed for each modality and all the octrees are merged in the end.

coordinates, $(\xi, \eta, \zeta) \in [-1,1] \times [-1,1] \times [-1,1]$ using combinations of 4 cubic polynomials in each of the variables ξ, η and ζ. A generic cubic polynomial in the variable ξ is given in Equation 25.

$$P(\xi) = \sum_{i=0}^{3} a_i \xi^i \tag{25}$$

The 4 cubic polynomials can be constructed by computing the coefficients, a_i, in Equation 25 such that exactly one of $P(-1)$, $P(1)$, $\frac{dP(-1)}{d\xi}$ and $\frac{dP(1)}{d\xi}$ is 1 and the rest are 0; each combination gives one unique polynomial. These 4 cubic polynomials are listed in Equation 26.

$$\begin{aligned} P_0^0(\xi) &= \frac{2 - 3\xi + \xi^3}{4} \\ P_0^1(\xi) &= \frac{1 - \xi - \xi^2 + \xi^3}{4} \\ P_1^0(\xi) &= \frac{2 + 3\xi - \xi^3}{4} \\ P_1^1(\xi) &= \frac{-1 - \xi + \xi^2 + \xi^3}{4} \end{aligned} \tag{26}$$

Using tensor products, we define:

$$\begin{aligned} N_{ijk}^{lmn}(\xi, \eta, \zeta) &= P_i^l(\xi) P_j^m(\eta) P_k^n(\zeta) \\ i, j, k &\in \{0, 1\} \\ l, m, n &\in \{0, 1\} \\ l + m + n &\leq 1 \end{aligned} \tag{27}$$

The approximations to the images within each voxel are then given by Equation 28. We used the second order accurate central difference scheme to compute the derivatives of T in Equation 28.

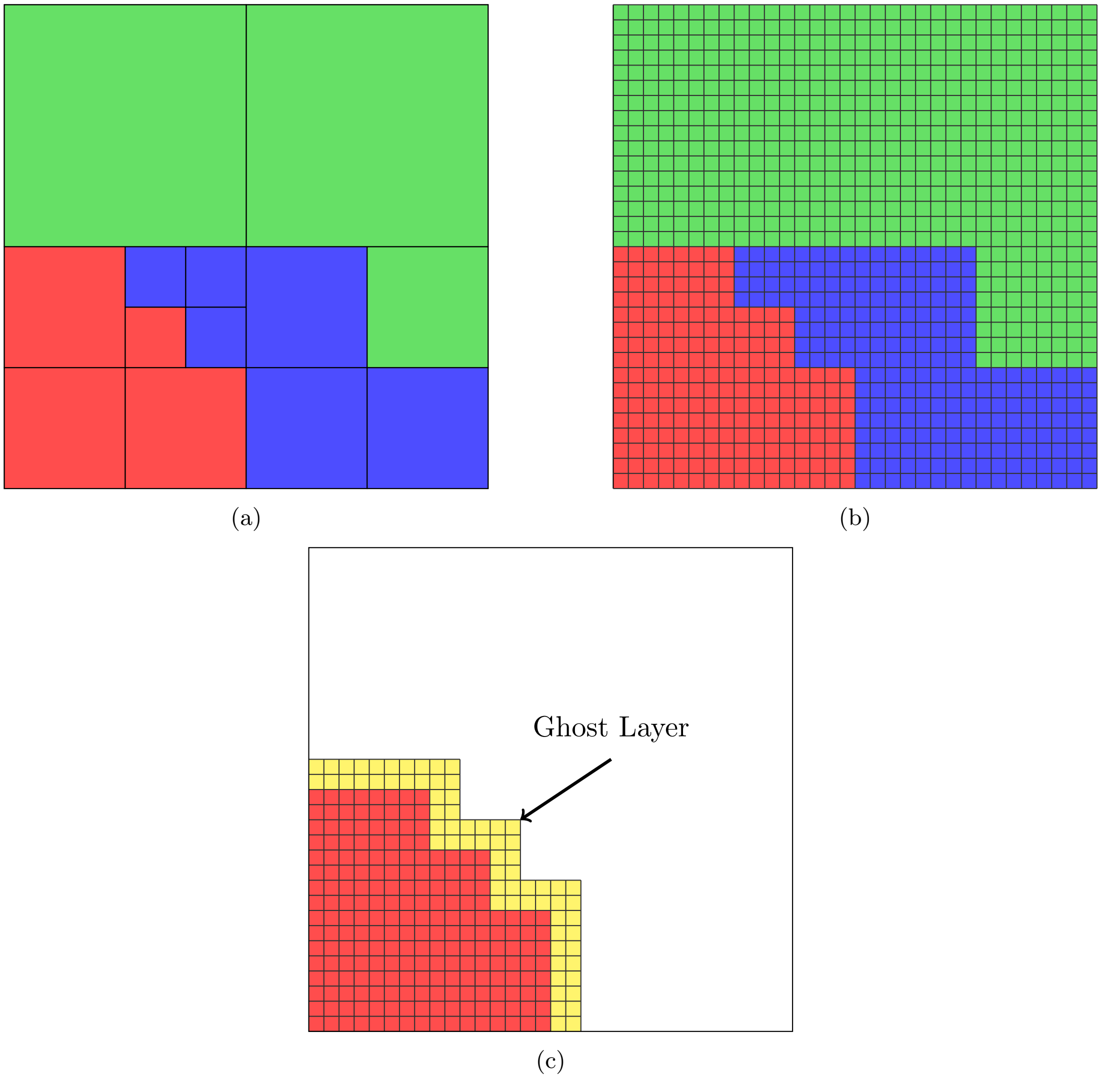
$$
\begin{aligned}
T(x,y,z) &= \sum_{i,j,k=0}^{1} \sum_{\substack{l,m,n=0\\ l+m+n\leq 1}}^{1} \widehat{T}_{ijk}^{lmn} N_{ijk}^{lmn}(\xi,\eta,\zeta) \\
\widehat{T}_{ijk}^{lmn} &= \left(\frac{h}{2}\right)^{l+m+n} \left.\frac{\partial^{(l+m+n)}T}{\partial x^l \partial y^m \partial z^n}\right|_{\substack{x=x0+ih\\ y=y0+jh\\ z=z0+kh}} \\
\xi &= -1+\frac{2(x-x0)}{h} \\
\eta &= -1+\frac{2(y-y0)}{h} \\
\zeta &= -1+\frac{2(z-z0)}{h} \\
(x,y,z) &\in [x0,x0+h)\times[y0,y0+h)\times[z0,z0+h) \qquad (28)
\end{aligned}
$$

5.3.1 Image partition

In our parallel implementation, we partition the images across the processors. The displacements computed during the solve may be such that some processors may need to access portions of the images owned by other processors in order to perform interpolation. To reduce the communication costs associated with this operation we impose the partition for the finest octree onto the images as well; hence, the portion of the images owned by each processor is aligned with the part of the finest octree owned by that processor. We then expand this initial partition of the images to include a layer of image voxels that is owned by other processors; the thickness of this layer is a few voxels and it is a parameter that can be controlled. This is illustrated in Figure 31. For the case of small deformations, most of the points, where the images and their gradients need to be evaluated, generated by each processor will lie within the portion of the images owned by that processor. We communicate the remaining points to the processors that own the respective image voxels, the recieving processor will evaluate the image and gradients at these points and communicate the results back to the processor that generated these points.

5.4 Solvers

In this section, we describe three techniques used to accelerate the solution of the elastic registration problem. First, we describe a Gauss Newton method that generates good search


(a)

(b)

(c)

Figure 31: Illustration of image partition: (a) An octree distributed on 3 processors, (b) the input image aligned with the octree and (c) the part of the input image owned by the first processor. In this example, the ghost layer recieved from other processors is 2 voxels thick.

directions for the optimization and thereby keeping the number of optimization iterations small. Next, we describe a multigrid scheme to efficiently solve the linear system of equations that arise in each optimization iteration. Finally, we describe a grid continuation scheme that solves a sequence of optimization problems on increasingly finer grids to reduce the overall computational cost and to improve the convergence of the algorithm.

5.4.1 Gauss Newton approximation

In this section, we describe how we solve Equation 24 using a Newton-type method. In the Newton's algorithm, the search directions, p, are generated by inverting the Hessian, H, in Equation 29:

$$\begin{aligned} H(u_{n-1})p_{n-1} &= -g(u_{n-1}) \\ u_n &= u_{n-1} + \alpha_{n-1}p_{n-1} \end{aligned} \tag{29}$$

where, α_{n-1} satisifies the Armijo backtracking condition [92].

$$\begin{aligned} w \cdot H(u)p &= \gamma\, a(w,p) + \int w \cdot (\nabla T(u) \otimes \nabla T(u))\, p\, dx \\ &- \int (S - T(u))\, w \cdot \left(\nabla^2 T(u)\right) p\, dx\ \forall p, w \end{aligned} \tag{30}$$

The true Hessian for the registration problem, given by Equation 30, can be shown to be indefinite and so the Newton step might not be a descent direction. Instead, we use a Gauss-Newton approximation (Equation 31) for the Hessian.

$$w \cdot H(u)p = \gamma\, a(w,p) + \int w \cdot (\nabla T(u) \otimes \nabla T(u))\, p\, dx\ \forall p, w \tag{31}$$

The approximate Hessian is derived by dropping the terms involving the second derivative of the “moving” image from the true Hessian. The approximate Hessian has an elasticity part and an image part. We evaluate the integral for the elasticity part exactly using precomputed stencils and we use numerical integration to approximate the integral for the image part. To reduce the computational costs, we only use the values of $\nabla T(u)$ evaluated

Algorithm 20 Gauss Newton Algorithm

```
Repeat until convergence:
  1. Given $u$, use interpolation to compute $T(u)$ and $\nabla T(u)$.
  2. Evaluate the objective function and the gradient of the objective
     function.
  3. Compute the Newton step, $p$, in Equation 30 using the multigrid solver.
  4. Use line search to scale the step:
       • Let $w$ be the Newton step.
       • Set $\alpha = 1$
       • While $J(u + \alpha w) > (J(u) + c_1 \alpha \nabla J(u)^T w)$
           – set $\alpha = c_2 \alpha$.   $(c_2 \in (0, 1))$
     A trust region approach could be used instead of line search.
  5. Update $u$ as shown in Equation 30.
  6. Declare convergence if one of the following holds:
       • The step-length is smaller than a given tolerance, which is
         typically some fraction of the voxel size.
       • Reduction in gradient is sufficient.
```

at the non-hanging vertices of the octree in our numerical integration rule. This results in a block diagonal approximation to the image part of the Hessian matrix. This approximate Hessian is positive definite, but can be highly ill-conditioned. We tackle the problem of ill-conditioning using the multigrid method, which is known to be robust for ill-conditioned systems. The Gauss Newton algorithm is listed in Algorithm 20.

5.4.2 Multigrid preconditioner

We solve Equation 29 using a Conjugate Gradient (CG) algorithm preconditioned using one multigrid V-cycle. Here, we use the multigrid algorithm described in Chapter 4. We use a 3×3 block-Jacobi smoother at each level and use a direct solver (LU) at the coarsest grid. To form the coarse grid operator, we first copy the values of $\nabla T(u)$ from the fine grid vertices to the coarse grid vertices by injection. We then assemble the coarse grid operators by using the coarse grid discretization for the elasticity part and nodal integration for the

Algorithm 21 Grid Continuation

```
1. Construct a sequence of coarse images from the given image by
   averaging.
2. Solve the optimization problem on the coarsest grid using zero guess.
3. Interpolate the solution to the next finer grid and use it as an
   initial guess for the optimization problem on that grid.
4. Repeat the last step until the finest grid is reached.
```

image part as decribed earlier.

5.4.3 Grid continuation

To reduce the overall computational cost and to improve the convergence of the algorithm we use a multiscale optimization algorithm. This idea has been used in other works as well [4, 60, 63]. In this approach, we solve a sequence of optimization problems on increasingly finer grids. The solution of each optimization problem is used as an initial guess for the subsequent optimization on the next finer level. The coarser grid iterations are cheaper than the fine grid iterations and they help escape local minima by aligning the coarse-level details. We also use the initial guess to construct an octree corresponding to the deformed moving image instead of the original moving image. The multiscale optimization algorithm is listed in Algorithm 21.

5.5 Results

In this section, we present the results from using the multiscale Gauss Newton algorithm to register synthetic as well as clinical images. We performed four sets of experiments:

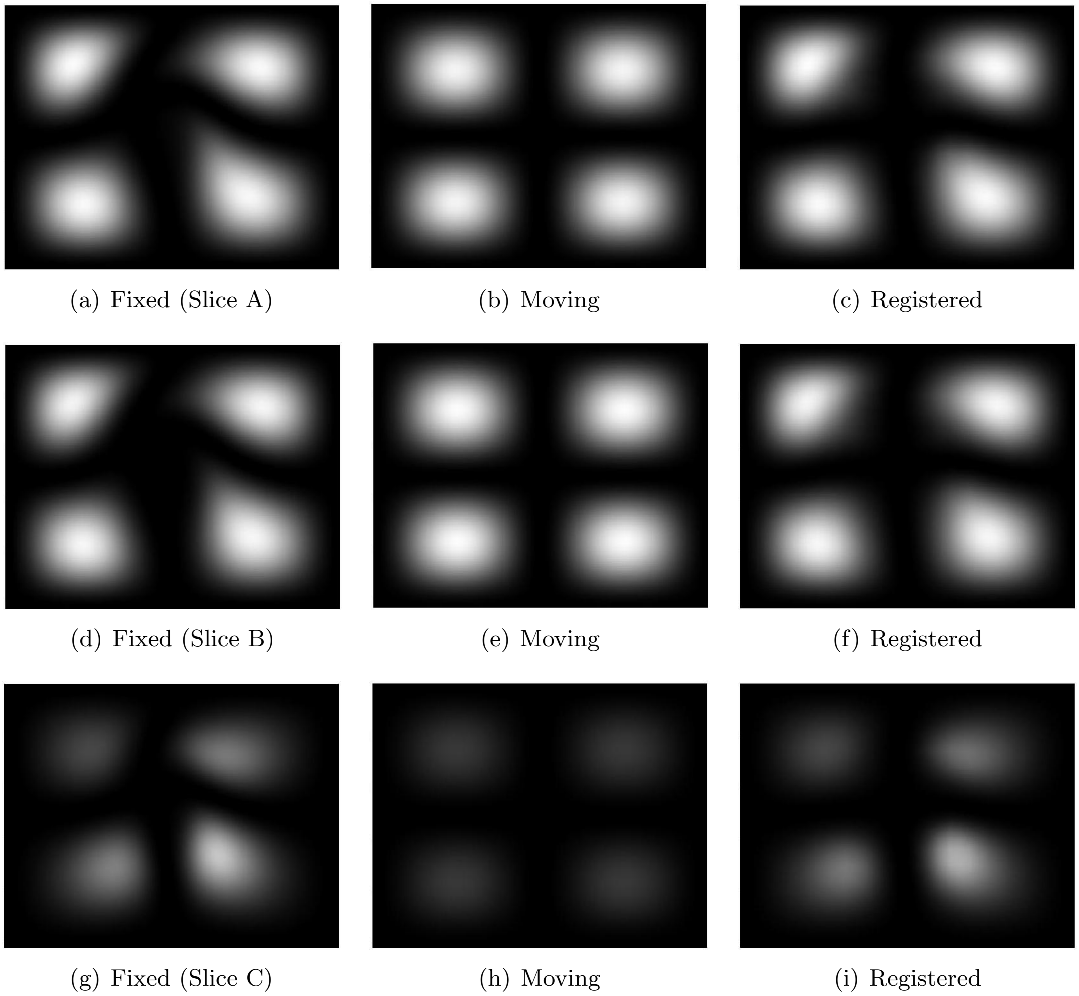
- We tested the performance of the algorithm on two synthetic examples.
- We tested the performance of the algorithm on clinical MR images of brains.
- We tested the effect of the thresholding parameter (δ) on the registration accuracy and the corresponding reduction in problem size. This parameter controls the number of elements in the octree; For a given image, using higher values of δ would result in coarser octrees compared to those constructed using lower values of δ.

- We tested the fixed-size and isogranular scalability of our MPI-based implementation of the proposed algorithms on the Sun Constellation Linux Cluster "Ranger" at the Texas Advanced Computing Center (TACC).

The Newton iterations in all these experiments were terminated when the maximum step length was less than $0.1 \times h$, where h is the regular grid spacing. In these experiments, we measured the performance of registration for different regularization parameters (γ) using the following metrics: (A) the reduction in absolute value of the mismatch between the registered and fixed images, (B) the number of optimization iterations, (C) the relative reductions in objective function, and (D) the relative reduction in the 2-norm of the gradient. We also computed the determinants of the Jacobians of the recovered deformations at a small number (7) of points within each element and report the maximum and minimum values across all elements; values closer to 1 indicate small deformations, values greater than 1 indicate expansion, values lesser than 1 indicate compression and negative values indicate non-physical deformations such as those involving intersecting or flipped elements.

5.5.1 Synthetic examples

We first present two synthetic example problems of resolution $256 \times 256 \times 256$ in Figures 32 and 35. In the former example, the moving image was chosen to be $255 \sin^2(2\pi x) \sin^2(2\pi y) \sin^2(2\pi z)$ and the fixed image was generated by applying 3 successive synthetic diffeomorphic displacement field to this image. We found the maximum and minimum values for the determinants of the Jacobians of the first deformation to be 1.12 and 0.84, respectively; the corresponding values for the second deformation were 1.28 and 0.64, respectively and the corresponding values for the third deformation were 1.70 and 0.08, respectively. In the latter example, the moving image is a sphere and the fixed image is a partial torus. In the figures, we only show a few selected slices in which the differences between the fixed and moving images are easily noticeable. We report the results from solving these registration problems using different regularization parameters in Figures 33 and 36, respectively. We show the corresponding recovered deformations for these two examples in Figures 34 and 37, respectively. We also report the corresponding number of Gauss Newton iterations and relative reductions in


(a) Fixed (Slice A)

(b) Moving

(c) Registered

(d) Fixed (Slice B)

(e) Moving

(f) Registered

(g) Fixed (Slice C)

(h) Moving

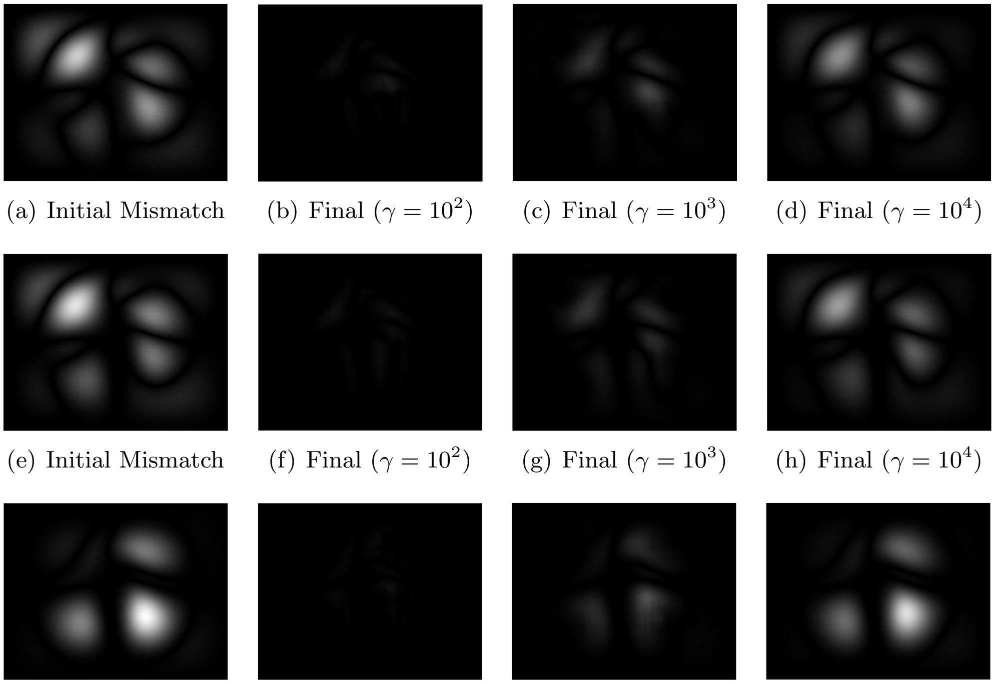
(i) Registered

Figure 32: First synthetic example. The moving image was chosen to be 255 $\sin^2(2\pi x)$ $\sin^2(2\pi y)$ $\sin^2(2\pi z)$ and the fixed image was generated by applying 3 successive synthetic diffeomorphic displacement field to this image. The resolution of each image was $256 \times 256 \times 256$. Each row shows a z-crosssectional slice of the fixed, moving and corresponding registered (deformed moving) images using the regularization parameter: $\gamma = 1000$.

objective function and gradient for each level of the multiscale algorithm in Tables 14 and 15, respectively.

5.5.2 Clinical examples

Next, we tested the performance of the algorithm on clinical MR images of resolution $256 \times 256 \times 256$. Figure 38 shows a few z-crosssectional slices of the fixed, moving and registered MR images of the brain of the same subject taken at different times. We show the result of registration for different choices of the regularization parameter in Figure 39.



(a) Initial Mismatch (b) Final ($\gamma = 10^2$) (c) Final ($\gamma = 10^3$) (d) Final ($\gamma = 10^4$)

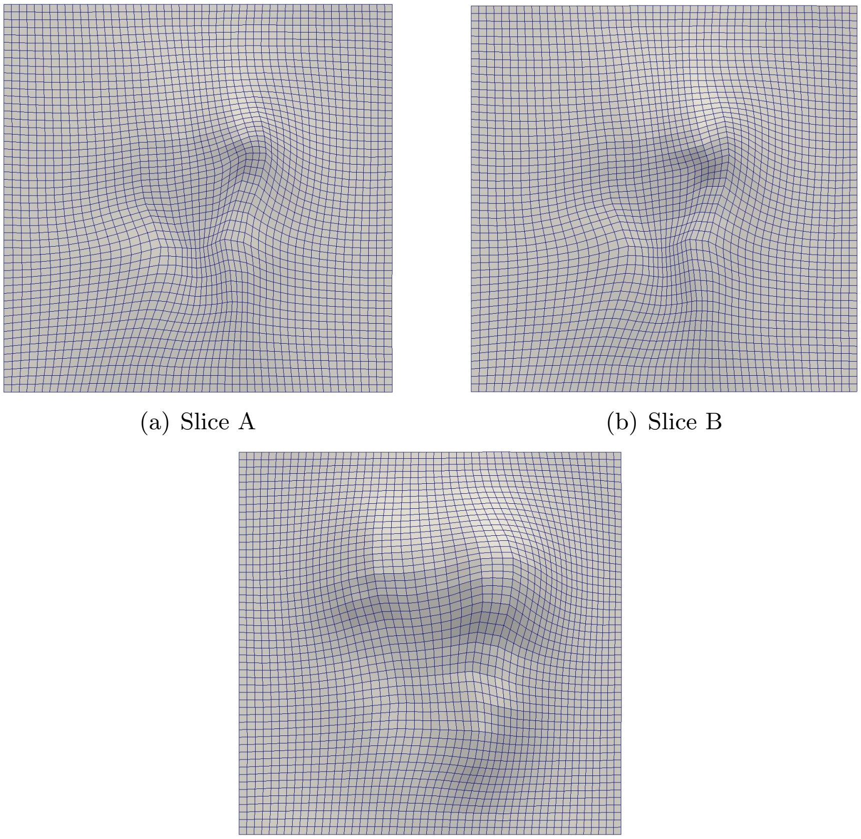
(e) Initial Mismatch (f) Final ($\gamma = 10^2$) (g) Final ($\gamma = 10^3$) (h) Final ($\gamma = 10^4$)

(i) Initial Mismatch (j) Final ($\gamma = 10^2$) (k) Final ($\gamma = 10^3$) (l) Final ($\gamma = 10^4$)

Figure 33: Results from using the proposed methology on the images shown in Figure 32. Each row shows a z-crosssectional slice of the initial mismatch between the fixed and moving images and the final mismatch between the registered and fixed images for different regularization parameters (γ).

Table 14: Performance of the optimizer for the synthetic images shown in Figure 32. J is the objective function and g is the 2-norm of the gradient. γ is the regularization parameter. Approximately 10^4 elements were used in the finest grid. The Newton iterations were terminated when the maximum step-length was less than $0.1 \times h$, where h is the regular grid spacing for that level. The maximum and minimum values of the determinants of the Jacobian of the recovered deformation for $\gamma = 10^2$ was found to be 1.89 and -0.14, respectively. The maximum and minimum values of the determinants of the Jacobian of the recovered deformation for $\gamma = 10^3$ was found to be 1.56 and 0.35, respectively. The maximum and minimum values of the determinants of the Jacobian of the recovered deformation for $\gamma = 10^4$ was found to be 1.21 and 0.85, respectively.

γ	Level 1 (Coarsest)			Level 2			Level 3			Level 4			Level 5 (Finest)		
	Its.	$\frac{J}{J_0}$	$\frac{g}{g_0}$	Its.	$\frac{J}{J_0}$	$\frac{g}{g_0}$	Its.	$\frac{J}{J_0}$	$\frac{g}{g_0}$	Its.	$\frac{J}{J_0}$	$\frac{g}{g_0}$	Its.	$\frac{J}{J_0}$	$\frac{g}{g_0}$
10^2	16	0.07	0.06	8	0.67	0.03	12	0.92	0.01	15	0.98	0.01	9	0.99	0.01
10^3	4	0.30	0.07	3	0.92	0.04	2	0.99	0.18	3	0.99	0.06	3	0.99	0.05
10^4	2	0.78	0.14	2	0.98	0.03	2	0.99	0.05	2	0.99	0.04	2	0.99	0.04



(a) Slice A

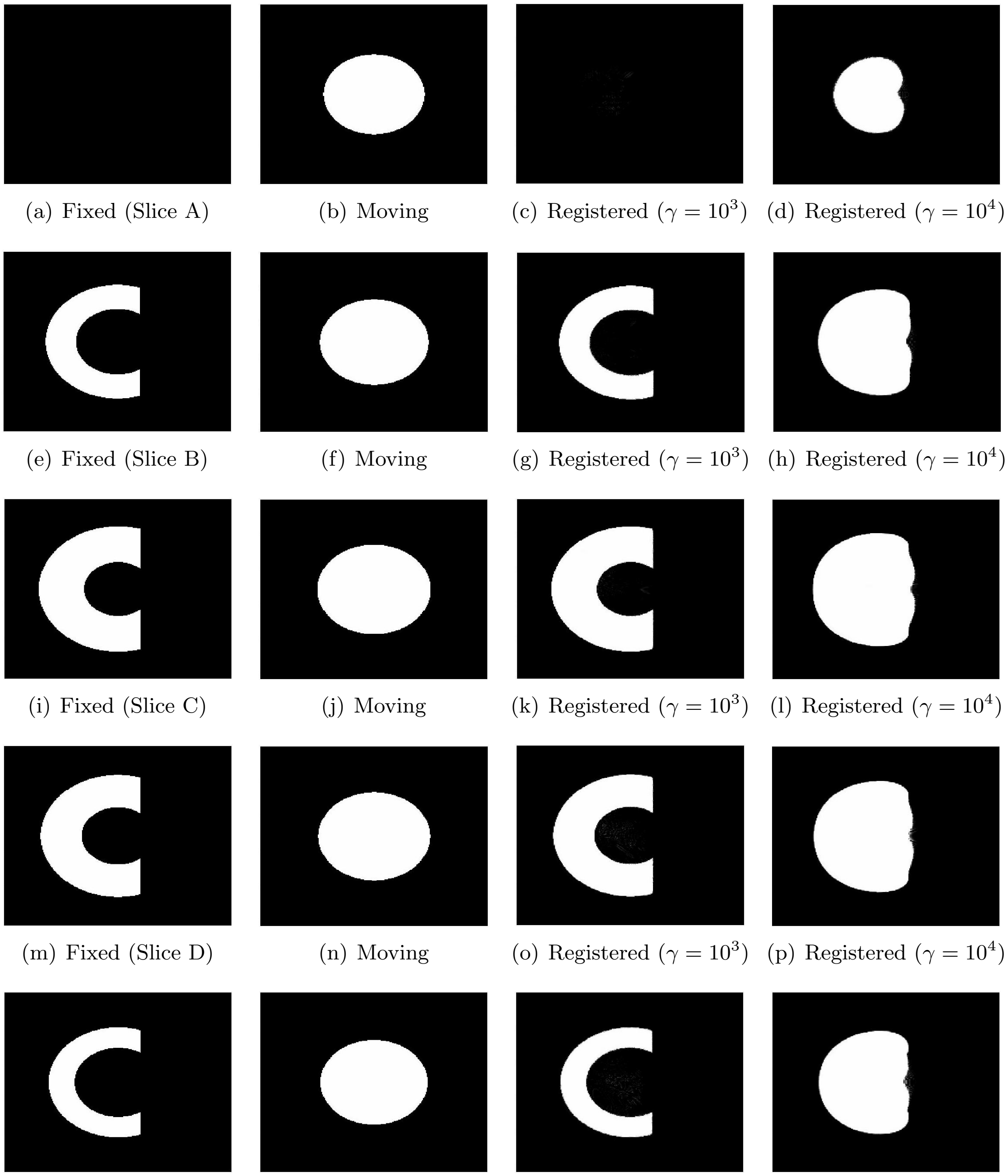
(b) Slice B

(c) Slice C

Figure 34: The z-crosssectional slice of the reconstructed deformation for the example shown in Figure 32.

Table 15: Performance of the optimizer for the synthetic images shown in Figure 35. J is the objective function and g is the 2-norm of the gradient. γ is the regularization parameter. Approximately 2.9×10^5 elements were used in the finest grid. The Newton iterations were terminated when the maximum step-length was less than $0.1 \times h$, where h is the regular grid spacing for that level or after 30 iterations. The maximum and minimum values of the determinants of the Jacobian of the recovered deformation for $\gamma = 10^3$ was found to be 84.82 and -11.94, respectively. The maximum and minimum values of the determinants of the Jacobian of the recovered deformation for $\gamma = 10^4$ was found to be 3.33 and -0.42, respectively. The maximum and minimum values of the determinants of the Jacobian of the recovered deformation for $\gamma = 10^5$ was found to be 1.07 and 0.97, respectively.

γ	Level 1 (Coarsest)			Level 2			Level 3			Level 4			Level 5 (Finest)		
	Its.	$\frac{J}{J_0}$	$\frac{g}{g_0}$	Its.	$\frac{J}{J_0}$	$\frac{g}{g_0}$	Its.	$\frac{J}{J_0}$	$\frac{g}{g_0}$	Its.	$\frac{J}{J_0}$	$\frac{g}{g_0}$	Its.	$\frac{J}{J_0}$	$\frac{g}{g_0}$
10^3	17	0.21	0.06	30	0.55	0.12	21	0.81	0.16	30	0.88	0.18	30	0.91	0.19
10^4	5	0.75	0.17	8	0.93	0.08	11	0.97	0.09	30	0.98	0.06	30	0.99	0.16
10^5	2	0.97	0.15	1	0.99	-	2	0.99	0.09	2	0.99	0.07	2	0.99	0.07



(a) Fixed (Slice A) (b) Moving (c) Registered ($\gamma = 10^3$) (d) Registered ($\gamma = 10^4$)

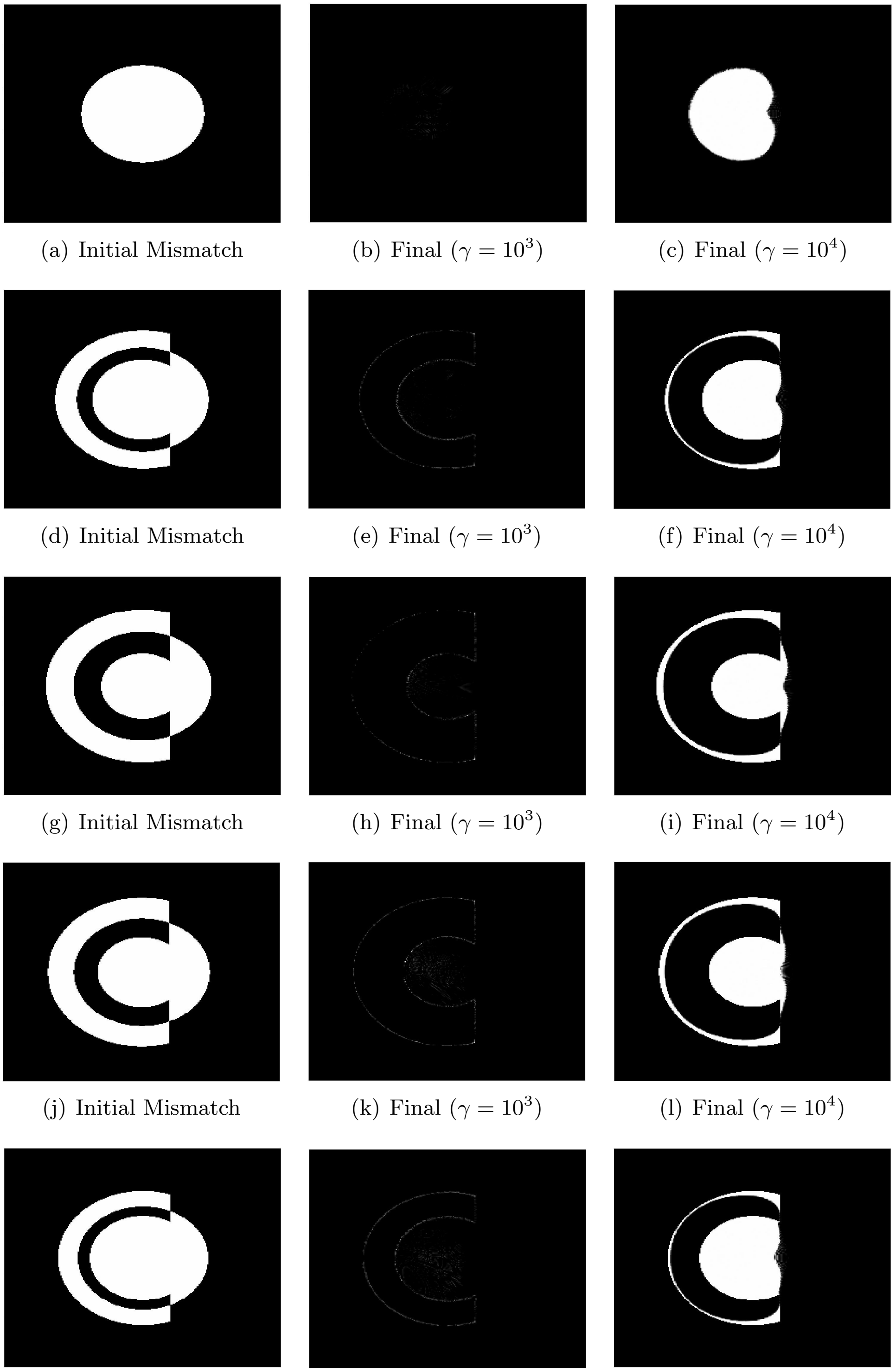
(e) Fixed (Slice B) (f) Moving (g) Registered ($\gamma = 10^3$) (h) Registered ($\gamma = 10^4$)

(i) Fixed (Slice C) (j) Moving (k) Registered ($\gamma = 10^3$) (l) Registered ($\gamma = 10^4$)

(m) Fixed (Slice D) (n) Moving (o) Registered ($\gamma = 10^3$) (p) Registered ($\gamma = 10^4$)

(q) Fixed (Slice E) (r) Moving (s) Registered ($\gamma = 10^3$) (t) Registered ($\gamma = 10^4$)

Figure 35: Second synthetic example. The moving image is a sphere and the fixed image is a partial torus. The resolution of each image was 256 × 256 × 256. Each row shows a z-crosssectional slice of the fixed, moving and corresponding registered (deformed moving) images for the regularization parameters: $\gamma = 10^3$ and $\gamma = 10^4$.



(a) Initial Mismatch

(b) Final ($\gamma = 10^3$)

(c) Final ($\gamma = 10^4$)

(d) Initial Mismatch

(e) Final ($\gamma = 10^3$)

(f) Final ($\gamma = 10^4$)

(g) Initial Mismatch

(h) Final ($\gamma = 10^3$)

(i) Final ($\gamma = 10^4$)

(j) Initial Mismatch

(k) Final ($\gamma = 10^3$)

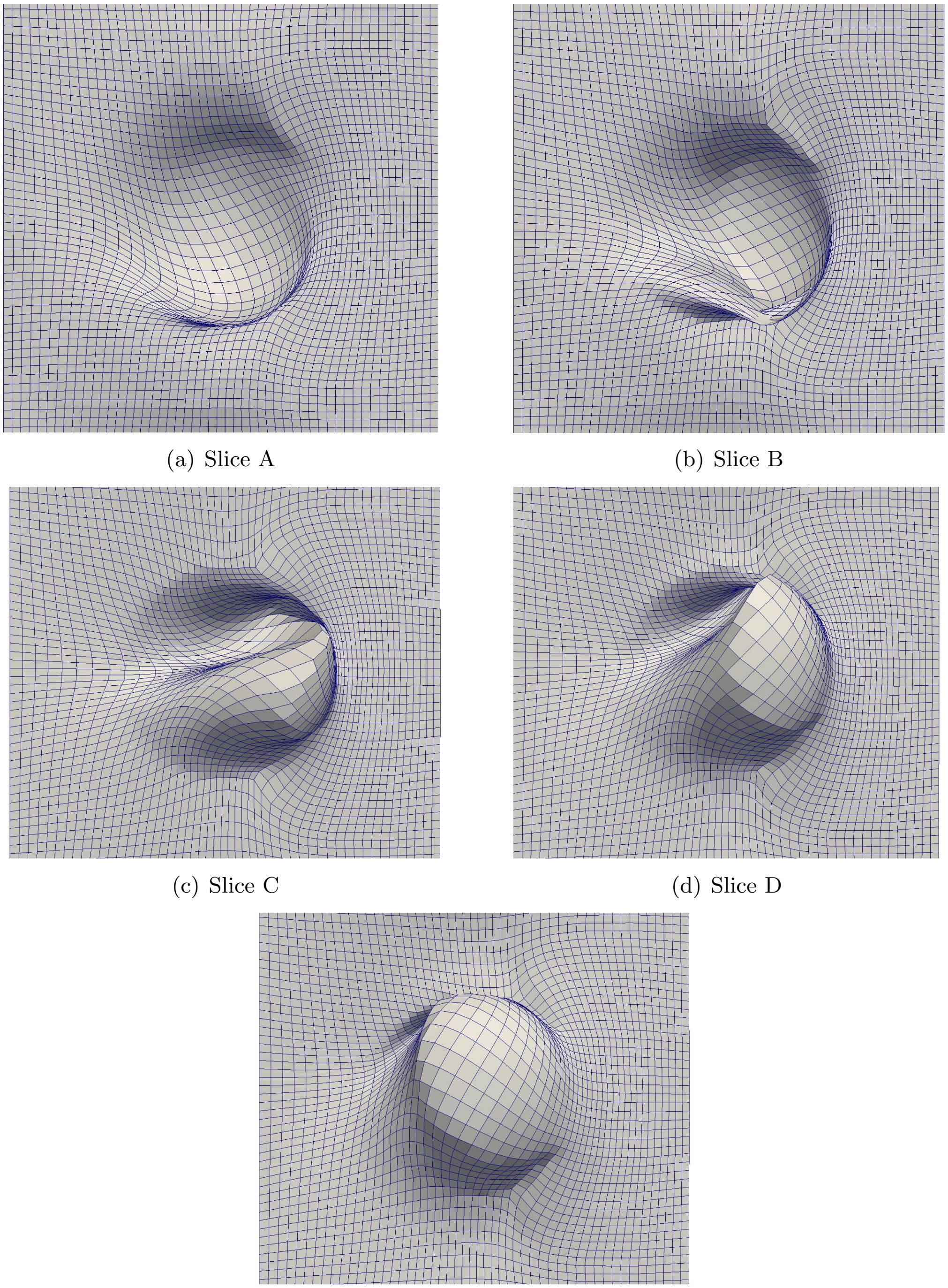
(l) Final ($\gamma = 10^4$)

(m) Initial Mismatch

(n) Final ($\gamma = 10^3$)

(o) Final ($\gamma = 10^4$)

Figure 36: Results from using the proposed methology on the images shown in Figure 35. Each row shows a z-crosssectional slice of the initial mismatch between the fixed and moving images and the final mismatch between the registered and fixed images for different regularization parameters (γ).



(a) Slice A

(b) Slice B

(c) Slice C

(d) Slice D

(e) Slice E

Figure 37: The z-crosssectional slice of the reconstructed deformation for the example shown in Figure 35.

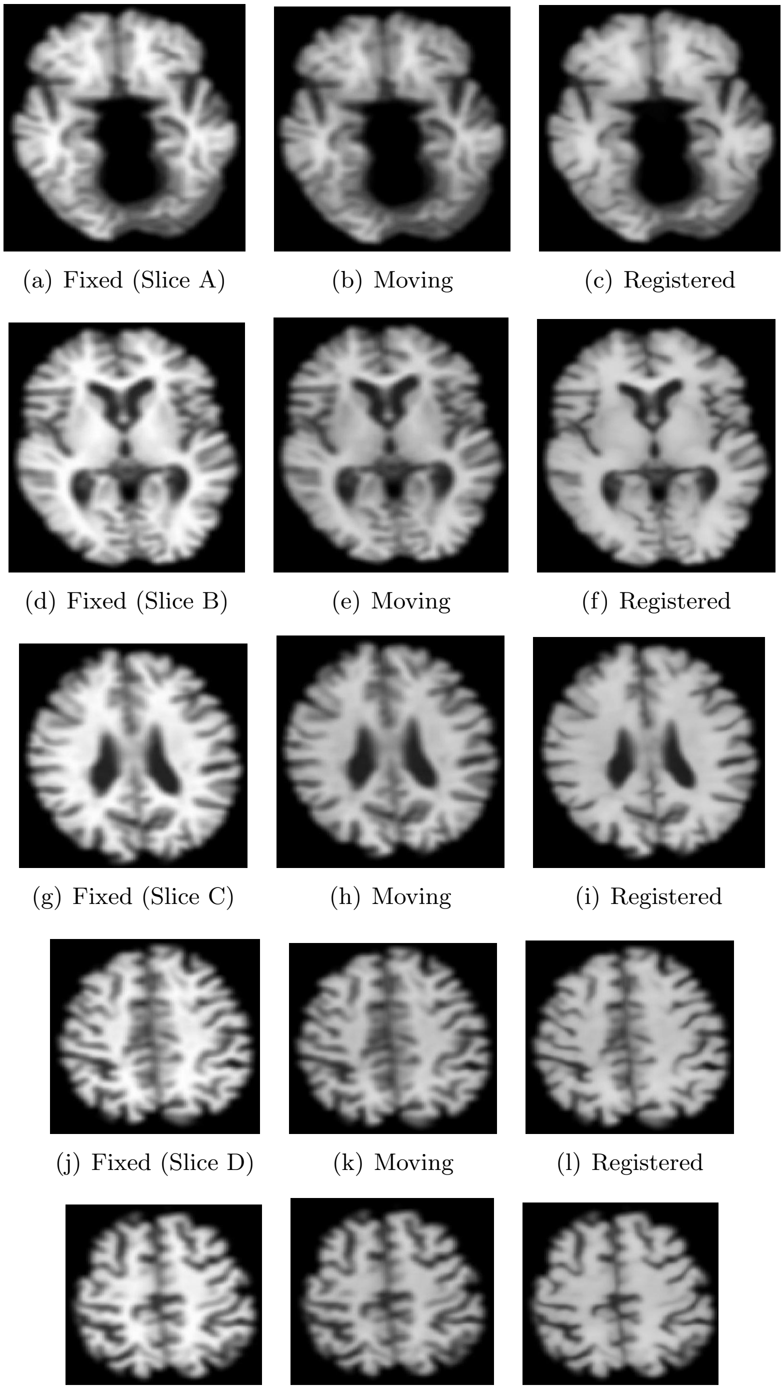
Table 16: Performance of the optimizer for the MR images shown in Figure 38. J is the objective function and g is the 2-norm of the gradient. γ is the regularization parameter. Approximately 1.28×10^6 elements were used in the finest grid. The Newton iterations were terminated when the maximum step-length was less than $0.1 \times h$, where h is the regular grid spacing for that level. The maximum and minimum values of the determinants of the Jacobian of the recovered deformation for $\gamma = 200$ was found to be 2.79 and 0.087, respectively. The maximum and minimum values of the determinants of the Jacobian of the recovered deformation for $\gamma = 500$ was found to be 1.65 and 0.49, respectively. The maximum and minimum values of the determinants of the Jacobian of the recovered deformation for $\gamma = 1000$ was found to be 1.35 and 0.67, respectively.

γ	Level 1 (Coarsest)			Level 2			Level 3			Level 4			Level 5 (Finest)		
	Its.	$\frac{J}{J_0}$	$\frac{g}{g_0}$	Its.	$\frac{J}{J_0}$	$\frac{g}{g_0}$	Its.	$\frac{J}{J_0}$	$\frac{g}{g_0}$	Its.	$\frac{J}{J_0}$	$\frac{g}{g_0}$	Its.	$\frac{J}{J_0}$	$\frac{g}{g_0}$
200	4	0.67	0.34	11	0.69	0.16	11	0.77	13.1	11	0.89	1.6e-2	12	0.97	8.9e-3
500	6	0.73	0.29	4	0.82	0.36	4	0.85	0.195	4	0.94	0.055	4	0.98	0.038
1000	3	0.78	0.21	4	0.87	0.16	4	0.92	0.106	27	0.97	3.1e-3	15	0.99	6.3e-3

The corresponding number of Gauss Newton iterations and relative reductions in objective function and gradient for each level of the multiscale algorithm is reported in Table 16. Another clinical example is shown in Figure 40; this is an example of inter-subject registration. The mismatch between the fixed and moving images before and after registration is shown in Figure 41. This experiment was repeated for different registration parameters and for each case the respective performance metrics like the number of optimization iterations and relative reductions in objective function and gradient are reported in Table 17. The reconstructed deformation for the inter-subject registration example is shown in Figure 42. We also computed the determinants of the Jacobians of the deformation at the centers of each voxel and this is shown in Figure 43.

5.5.3 Effect of the thresholding parameter

In Figure 44, we show the effect of the thresholding parameter, δ, on the registration accuracy for the example shown in Figure 40. We also report the relative reduction in mismatch between the registered and fixed images and the corresponding number of octants in the finest octree in Table 18. It is quite promising that the thresholding parameter has little effect on the registration accuracy but has a significant effect on the number of octants. This suggests that we can significantly reduce the computation time without sacrificing registration accuracy.



(a) Fixed (Slice A) (b) Moving (c) Registered

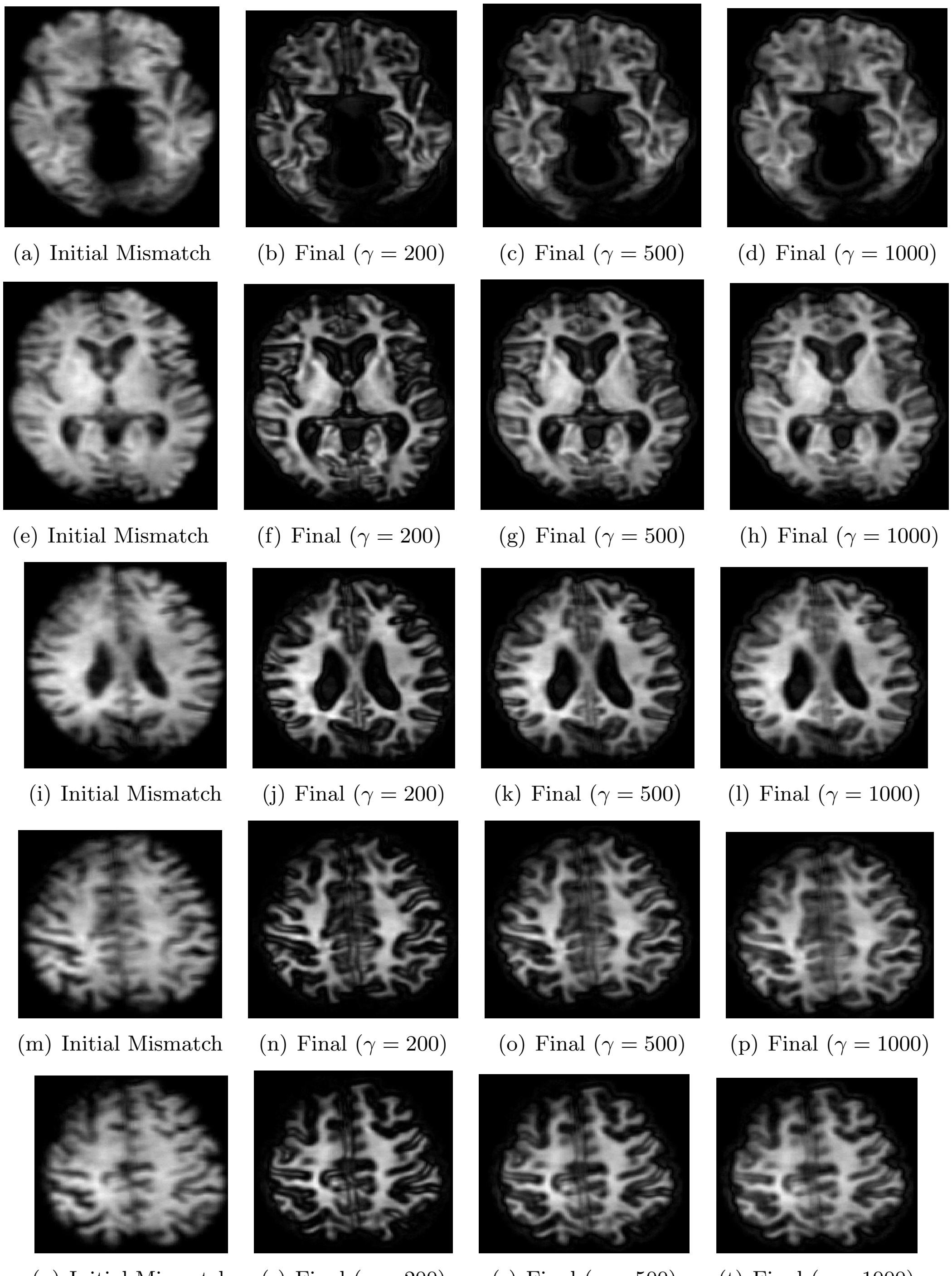
(d) Fixed (Slice B) (e) Moving (f) Registered

(g) Fixed (Slice C) (h) Moving (i) Registered

(j) Fixed (Slice D) (k) Moving (l) Registered

(m) Fixed (Slice E) (n) Moving (o) Registered

Figure 38: Skull stripped MR images of the brain of the same subject taken at different times and the corresponding registered (deformed moving) image for $\gamma = 200$. Each row shows a z-crosssectional slice of the fixed, moving and registered images.



(a) Initial Mismatch (b) Final ($\gamma = 200$) (c) Final ($\gamma = 500$) (d) Final ($\gamma = 1000$)

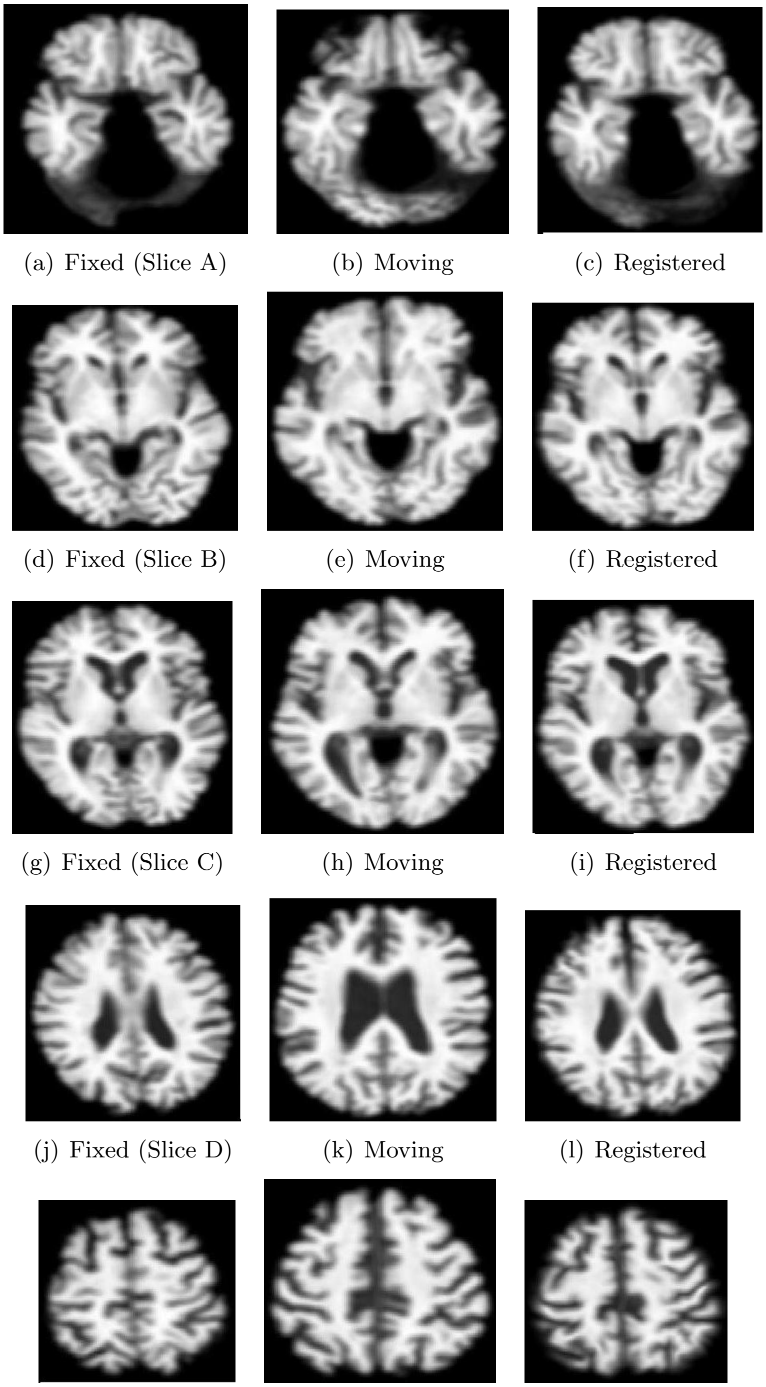
(e) Initial Mismatch (f) Final ($\gamma = 200$) (g) Final ($\gamma = 500$) (h) Final ($\gamma = 1000$)

(i) Initial Mismatch (j) Final ($\gamma = 200$) (k) Final ($\gamma = 500$) (l) Final ($\gamma = 1000$)

(m) Initial Mismatch (n) Final ($\gamma = 200$) (o) Final ($\gamma = 500$) (p) Final ($\gamma = 1000$)

(q) Initial Mismatch (r) Final ($\gamma = 200$) (s) Final ($\gamma = 500$) (t) Final ($\gamma = 1000$)

Figure 39: Results from using the proposed methology for registering the example shown in Figure 38. Each row shows a z-crosssectional slice of the initial mismatch between the fixed and moving images and the final mismatch between the registered and fixed images for different regularization parameters (γ).



(a) Fixed (Slice A) (b) Moving (c) Registered

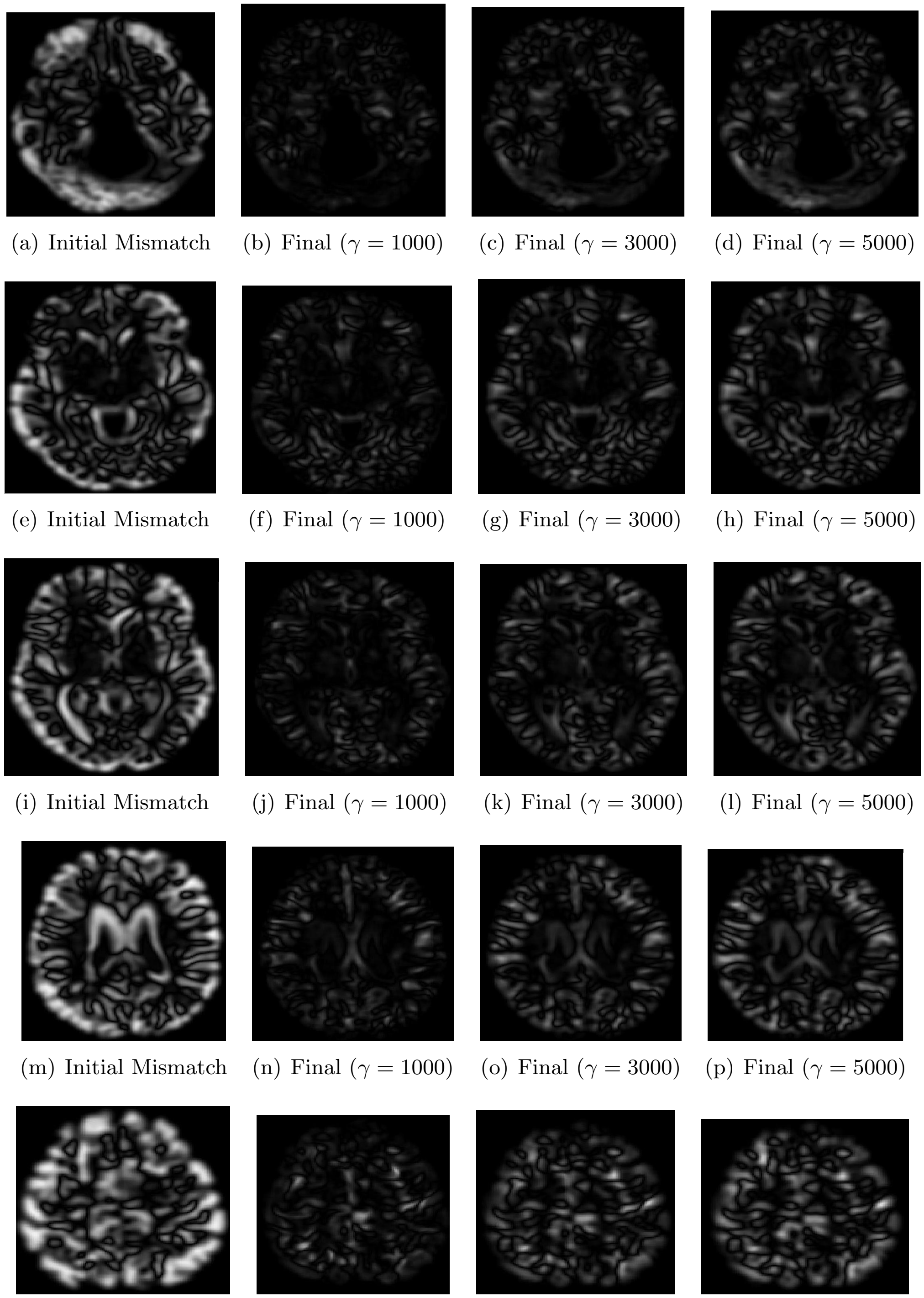
(d) Fixed (Slice B) (e) Moving (f) Registered

(g) Fixed (Slice C) (h) Moving (i) Registered

(j) Fixed (Slice D) (k) Moving (l) Registered

(m) Fixed (Slice E) (n) Moving (o) Registered

Figure 40: Skull stripped MR images of the brains of two different subjects and the corresponding registered (deformed moving) image for $\gamma = 3000$. Each row shows a z-crosssectional slice of the fixed, moving and registered images.



(a) Initial Mismatch (b) Final ($\gamma = 1000$) (c) Final ($\gamma = 3000$) (d) Final ($\gamma = 5000$)

(e) Initial Mismatch (f) Final ($\gamma = 1000$) (g) Final ($\gamma = 3000$) (h) Final ($\gamma = 5000$)

(i) Initial Mismatch (j) Final ($\gamma = 1000$) (k) Final ($\gamma = 3000$) (l) Final ($\gamma = 5000$)

(m) Initial Mismatch (n) Final ($\gamma = 1000$) (o) Final ($\gamma = 3000$) (p) Final ($\gamma = 5000$)

(q) Initial Mismatch (r) Final ($\gamma = 1000$) (s) Final ($\gamma = 3000$) (t) Final ($\gamma = 5000$)

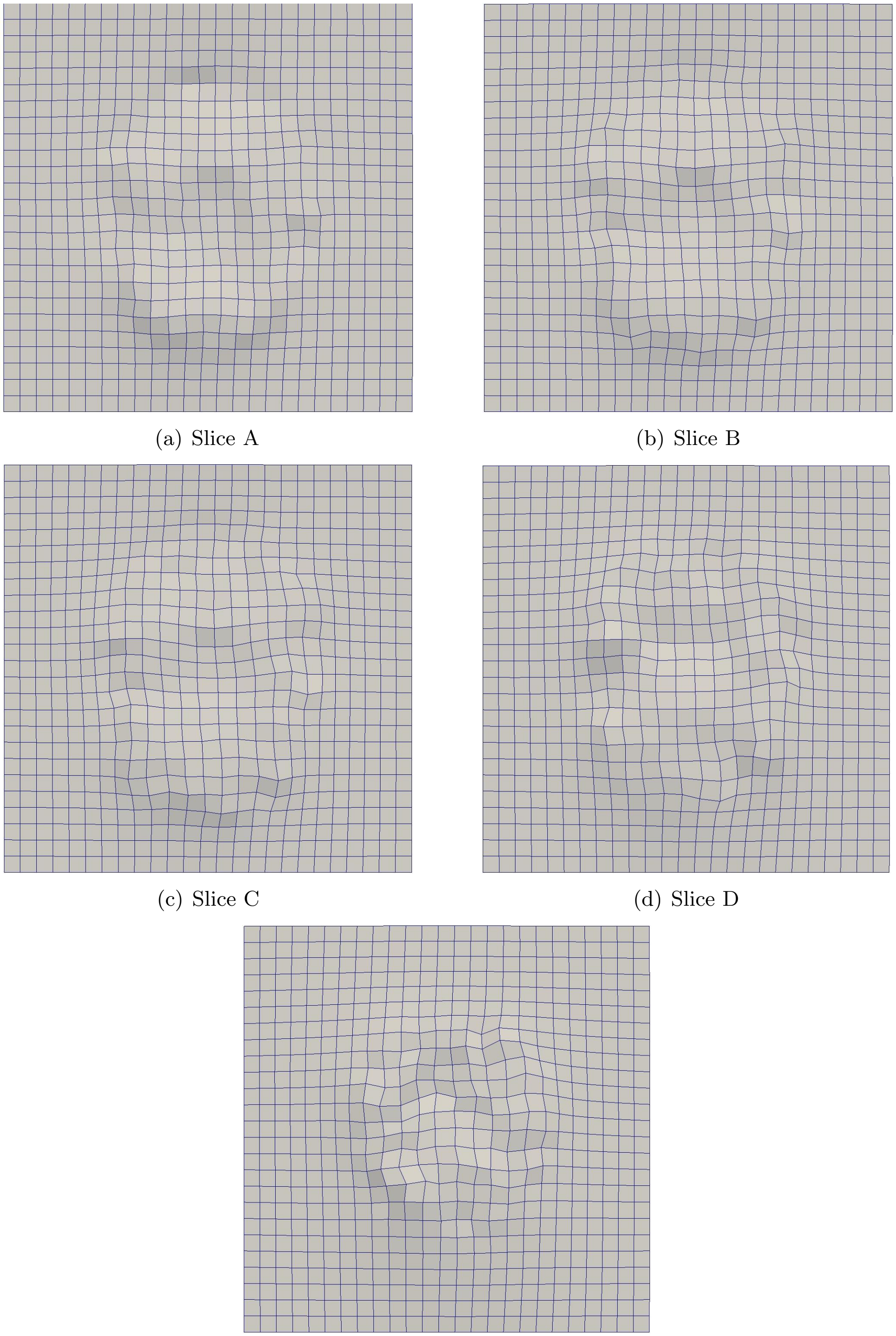
Figure 41: Results from using the proposed methology for registering the example shown in Figure 40. Each row shows a z-crosssectional slice of the initial mismatch between the fixed and moving images and the final mismatch between the registered and fixed images for different regularization parameters (γ).

Table 17: Performance of the optimizer for registering the MR images shown in Figure 40. J is the objective function and g is the 2-norm of the gradient. γ is the regularization parameter. The threshold parameter, δ, was set equal to 10 in this experiment. Approximately 1.4×10^6 elements were used in the finest grid. The Newton iterations were terminated when the maximum step-length was less than $0.1 \times h$, where h is the regular grid spacing for that level. We computed the determinant of the Jacobians of the deformation at 7 points within each voxel. The maximum and minimum values of the determinants of the Jacobian of the recovered deformation for $\gamma = 1000$ was found to be 11.15 and -1.34, respectively. The maximum and minimum values of the determinants of the Jacobian of the recovered deformation for $\gamma = 3000$ was found to be 3.83 and 0.53, respectively. The maximum and minimum values of the determinants of the Jacobian of the recovered deformation for $\gamma = 5000$ was found to be 2.19 and 0.74, respectively.

γ	Level 1 (Coarsest)			Level 2			Level 3			Level 4			Level 5 (Finest)		
	Its.	$\frac{J}{J_0}$	$\frac{g}{g_0}$	Its.	$\frac{J}{J_0}$	$\frac{g}{g_0}$	Its.	$\frac{J}{J_0}$	$\frac{g}{g_0}$	Its.	$\frac{J}{J_0}$	$\frac{g}{g_0}$	Its.	$\frac{J}{J_0}$	$\frac{g}{g_0}$
1e3	8	0.26	0.06	9	0.69	0.08	20	0.70	0.01	43	0.84	4e-3	14	0.95	6e-3
3e3	3	0.44	0.13	3	0.82	0.20	7	0.84	0.04	24	0.92	0.01	33	0.98	3e-3
5e3	2	0.54	0.25	2	0.86	0.33	9	0.88	0.02	9	0.95	0.01	12	0.98	5e-3

Table 18: Effect of the thresholding parameter (δ) used for octree construction for the example shown in Figure 40. The regularization parameter, γ, was set equal to 3000 in this experiment. We report the sum of the square of the mismatch between the fixed and registered images normalized by the sum of the square of the mismatch between the fixed and moving images. We also report the number of octants in the finest octree for each case.

δ	Relative SSD	Number of Elements
0	0.151	1.678×10^7
10	0.152	1.37×10^6
20	0.152	1.18×10^6
30	0.153	9.4×10^5
40	0.154	6.7×10^5
50	0.155	4.3×10^5



(a) Slice A

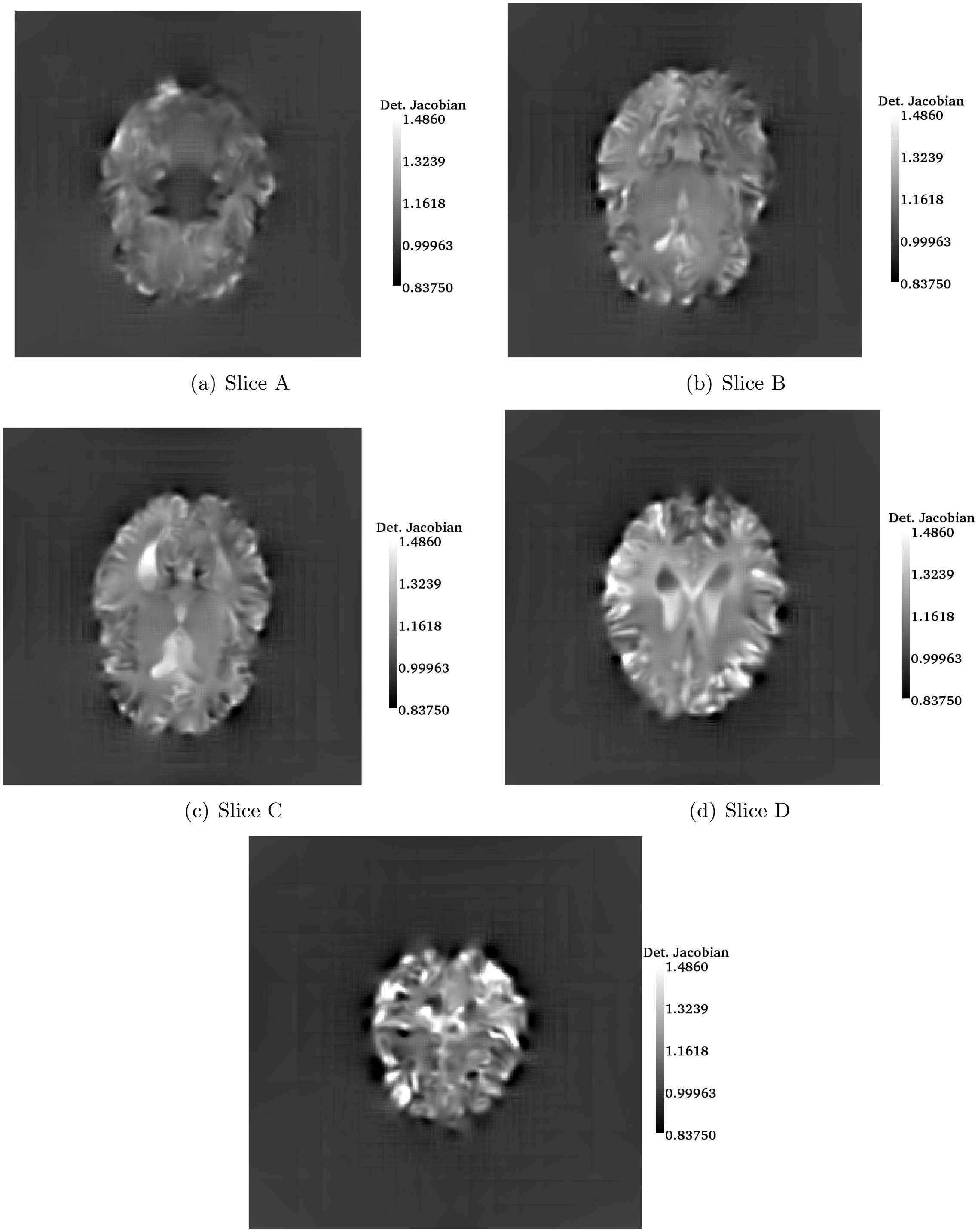
(b) Slice B

(c) Slice C

(d) Slice D

(e) Slice E

Figure 42: The z-crosssectional slice of the reconstructed deformation for the example shown in Figure 40.



(a) Slice A

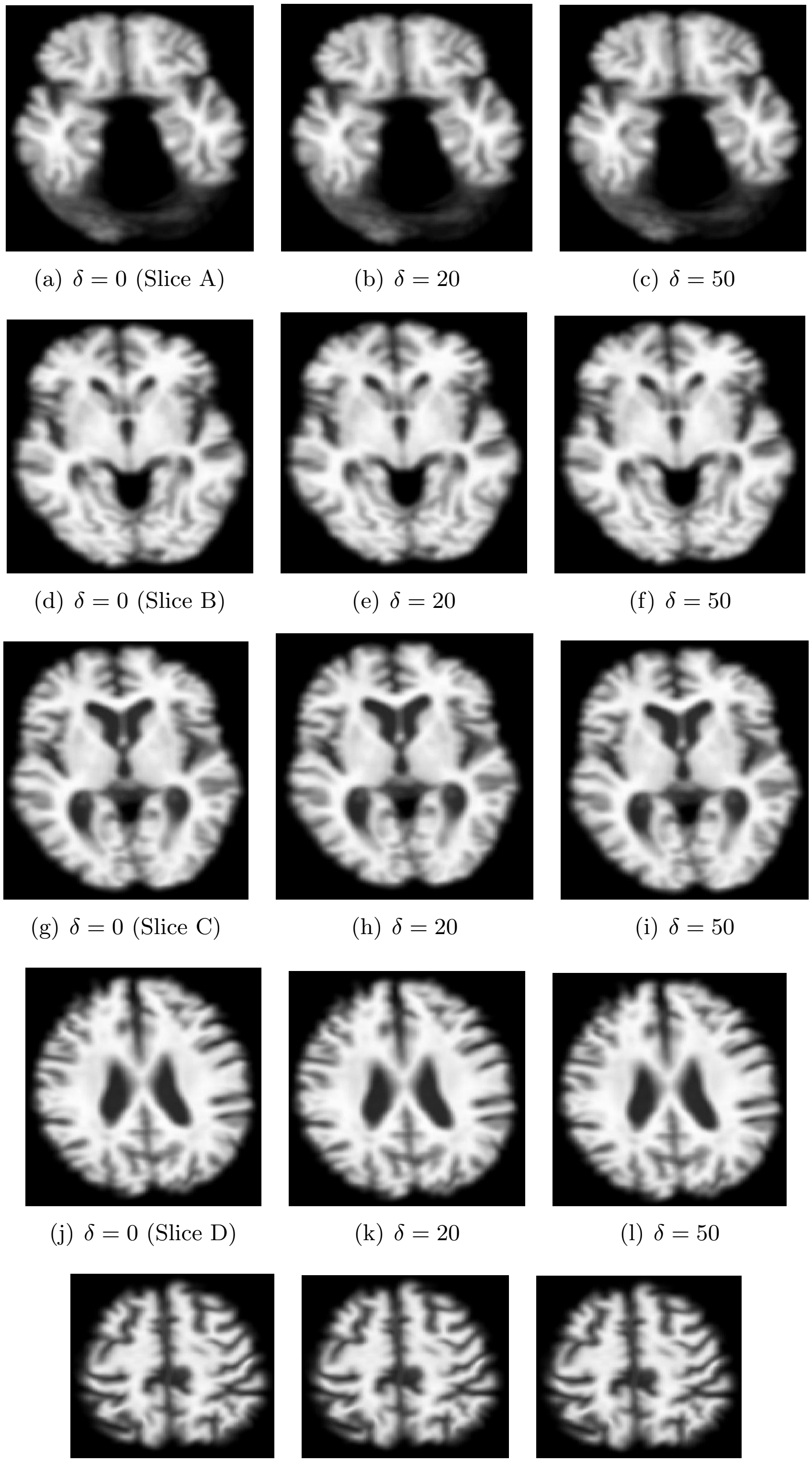
(b) Slice B

(c) Slice C

(d) Slice D

(e) Slice E

Figure 43: Determinants of the Jacobians at the centers of voxels for the example shown in Figure 40.



(a) $\delta = 0$ (Slice A)

(b) $\delta = 20$

(c) $\delta = 50$

(d) $\delta = 0$ (Slice B)

(e) $\delta = 20$

(f) $\delta = 50$

(g) $\delta = 0$ (Slice C)

(h) $\delta = 20$

(i) $\delta = 50$

(j) $\delta = 0$ (Slice D)

(k) $\delta = 20$

(l) $\delta = 50$

(m) $\delta = 0$ (Slice E)

(n) $\delta = 20$

(o) $\delta = 50$

Figure 44: Effect of the thresholding parameter (δ) used for octree construction for the example shown in Figure 40. Each row shows a z-crosssectional slice of the registered (deformed moving) images for $\gamma = 3000$.

5.5.4 Parallel scalability

Finally, we tested the parallel scalability of our implementation on the Sun Constellation Linux Cluster “Ranger” at the Texas Advanced Computing Center (TACC). We report the results from the fixed-size scalability and isogranular scalability experiments in Tables 19 and 20, respectively. In the fixed-size scalability experiment, we track the execution time to solve a fixed problem on different processor counts. In the isogranular scalability experiment, we track the execution time while increasing the problem size and the number of processors proportionately. Overall the scalability of the algorithms is reasonable but the overhead associated with building the image patches needs to be reduced.

5.6 Summary

In this chapter, we presented a parallel algorithm for solving the elastic image registration problem using octrees. We used a Gauss Newton optimization algorithm and solved the linear system that arises in each optimization iteration using a multigrid preconditioned Conjugate Gradient algorithm. We also used a multiscale optimization approach to speed up the computation and to escape local minima. We demonstrated the performance of the algorithm on synthetic as well as clinical images.

Table 19: Fixed-size scalability for the example shown in Figure 40 using $\gamma = 3000$ and $\delta = 50$. Approximately 4.3×10^5 elements were used in the finest grid. The time spent in evaluating the objective function is reported in the row labelled "Objective". The time spent in evaluating the gradient is reported in the row labelled "Gradient". Interpolation at the Gauss points were required to evaluate the objective function and the gradient. 4-th order Gauss quadrature rule was used so there are 64 Gauss points per element. Interpolation at vertices were required to build the approximate Hessian using under-integration. The total time spent in the optimization function is reported in the row labelled "Gauss Newton". The total time spent in the linear solves in each Newton iteration is reported in the row labelled "KSP solve". The time spent in the Matvecs for the elasticity operator is reported in the row labelled "Elas-Matvec"; this is only used to evaluate the objective and the gradient and not for the Hessian. The time spent in the Hessian Matvecs is reported in the row labelled "Hess-Matvec". The time required to update the Hessian is reported in the row labelled "Update-Hess"; this includes the time spent in interpolations. The time spent in building the image patches on each processor and gathering the image values from the Petsc DA ordering to the local ordering is reported in the row labelled "Build Patches". The time spent in setting up the multigrid solver is reported in the row labelled "MG-Setup"; this includes the time spent in coarsening and balancing all the coarser octrees for the multigrid and meshing for all the multigrid levels. The total runtime is the sum of the times spent in setting up the Multigrid, building the image patches and the Gauss Newton iterations. These rows are printed in boldface. All timings are reported in seconds. This experiment was performed on the Teragrid system "Ranger" [122]. Although, the last run was submitted to 512 processors only 374 processors could be used because the grain size (elements per processor) for the finest octree was too small.

Processors	16	32	64	128	256	374
Objective	35.28	153.04	42.68	35.81	16.22	11.19
Gradient	28.37	20.12	10.27	8.66	6.85	6.94
Interpolation at Gauss points	861.60	606.03	273.44	157.15	83.38	72.51
Interpolation at Vertices	5.41	3.84	1.68	1.04	0.57	0.50
Gauss Newton	**2.37e3**	**1.65e3**	**912.70**	**620.77**	**474.32**	**446.13**
KSP Solve	1.46e3	1.01e3	622.79	443.01	379.48	354.93
Elas-Matvec	43.2	62.4	42.17	28.75	16.45	15.28
Hess-Matvec	1.30e3	892.8	523.52	345.91	303.73	266.74
Update-Hess	654.30	462.77	212.53	126.68	70.31	62.33
Build Patches	**92.95**	**92.08**	**87.02**	**80.54**	**78.76**	**83.03**
MG-Setup	**3.02**	**3.77**	**3.53**	**2.92**	**5.03**	**3.79**

Table 20: Isogranular scalability for synthetic examples. The row labelled "N" gives the number of voxels in each dimension of the images and the corresponding number of octants at the finest octree is reported in the row labelled "Octants". The time spent in evaluating the objective function is reported in the row labelled "Objective". The time spent in evaluating the gradient is reported in the row labelled "Gradient". Interpolation at the Gauss points were required to evaluate the objective function and the gradient. 4-th order Gauss quadrature rule was used so there are 64 Gauss points per element. Interpolation at vertices were required to build the approximate Hessian using under-integration. The total time spent in the optimization function is reported in the row labelled "Gauss Newton". The total time spent in the linear solves in each Newton iteration is reported in the row labelled "KSP solve". The time spent in the Matvecs for the elasticity operator is reported in the row labelled "Elas-Matvec"; this is only used to evaluate the objective and the gradient and not for the Hessian. The time spent in the Hessian Matvecs is reported in the row labelled "Hess-Matvec". The time required to update the Hessian is reported in the row labelled "Update-Hess"; this includes the time spent in interpolations. The time spent in building the image patches on each processor and gathering the image values from the Petsc DA ordering to the local ordering is reported in the row labelled "Build Patches". The time spent in setting up the multigrid solver is reported in the row labelled "MG-Setup"; this includes the time spent in coarsening and balancing all the coarser octrees for the multigrid and meshing for all the multigrid levels. The total runtime is the sum of the times spent in setting up the Multigrid, building the image patches and the Gauss Newton iterations. These rows are printed in boldface. All timings are reported in seconds. This experiment was performed on the Teragrid system "Ranger" [122].

Processors	4	32	256	2048
N	64	128	256	512
Octants	1.7×10^5	1.08×10^6	5.04×10^6	5.614×10^6
Objective	2.04	1.19	3.41	2.71
Gradient	3.06	2.59	2.02	1.27
Interpolation at Gauss points	50.39	43.71	39.41	15.39
Interpolation at Vertices	0.37	0.31	0.28	0.18
Gauss Newton	**101.99**	**96.29**	**84.67**	**54.37**
KSP Solve	50.85	52.40	42.65	33.43
Elas-Matvec	2.75	1.85	2.60	2.41
Hess-Matvec	46.50	46.86	32.15	19.28
Update-Hess	35.08	30.76	29.15	13.34
Build Patches	**3.46**	**4.49**	**9.05**	**28.51**
MG-Setup	**1.68**	**3.02**	**4.59**	**7.45**

APPENDIX A

PROPERTIES OF MORTON ENCODING

Property 1 *Sorting all the leaves in the ascending order of their Morton ids is identical to a preorder traversal of the leaves of the octree. If one connects the centers of the leaves in this order, one can observe a Z-pattern in the Cartesian space. The space-filling Z-order curve has the property that spatially nearby octants tend to be clustered together. The octants in Figures 4(b) and 4(c) are all labeled according to this order. Depending on the order of interleaving the coordinates, different Z-order curves are obtained. The two possible Z-curves in 2-D are shown in the Figure 45. Similarly, in 3-D six different types of Morton ordering are possible.*

Property 2 *Given three octants, $a < b < c$ and $c \notin \{\mathcal{D}(b)\}$:*

$$a < d < c, \quad \forall d \in \{\mathcal{D}(b)\}.$$

Property 3 *The Morton id of any node is less than those of its descendants.*

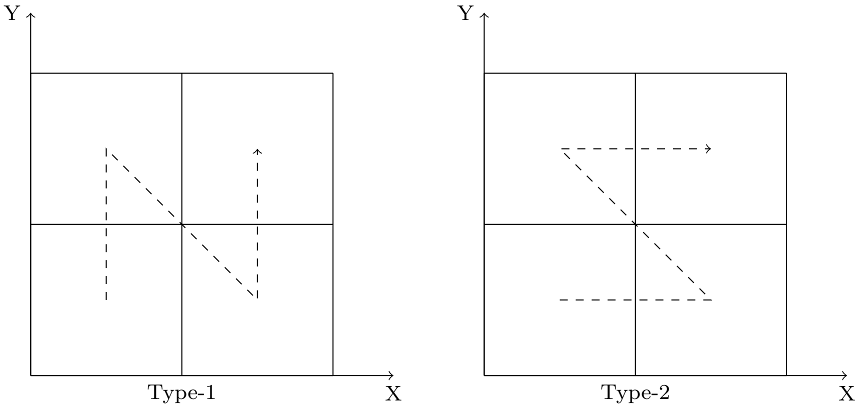


Figure 45: Two types of z-ordering in quadtrees.

Property 4 *Two distinct octants overlap if and only if one is an ancestor of the other.*

Property 5 *The Morton id of any node and of its first child*[1] *are consecutive. It follows from Property 3 that the first child is also the child with the least Morton id.*

Property 6 *The first descendant at level l, denoted by $\mathcal{FD}\,(N, l)$, of any node N is the descendant at level l with the least Morton id. This can be arrived at by following the first child at every level starting from N. $\mathcal{FD}\,(N, D_{max})$ is also the anchor of N and is also referred to as the* deepest first descendant*, denoted by $\mathcal{DFD}(N)$, of node N.*

Property 7 *The range $(N, \mathcal{DFD}(N)]$ only contains the first descendants of N at different levels and hence there can be no more than one leaf in this range in the entire linear octree.*

Property 8 *The* last descendant *at level l, denoted by $\mathcal{LD}\,(N, l)$, of any node N is the descendant at level l with the greatest Morton id. This can be arrived at by following the* last child[2] *at every level starting from N. $\mathcal{LD}\,(N, D_{max})$ is also referred to as the* deepest last descendant*, denoted by $\mathcal{DLD}(N)$, of node N.*

Property 9 *Every octant in the range $(N, \mathcal{DLD}(N)]$ is a descendant of N.*

[1] *the child that has the same anchor as the parent*

[2] *child with the greatest Morton id*

APPENDIX B

MULTICOMPONENT MORTON REPRESENTATION

Every Morton id is a set of 4 entities: The three co-ordinates of the anchor of the octant and the level of the octant. We have implemented the node as a C++ class, which contains these 4 entities as its member data. To use this set as a locational code for octants, we define two primary binary logical operations on it: a) Comparing if 2 ids are equal and b) Comparing if one id is lesser than the other.

Two ids are equal if and only if all the 4 entities are respectively equal. If two ids have the same anchor then the one at a coarser level has a lesser Morton id. If the anchors are different, then we can use Algorithm 22 to determine the lesser id. The Z-ordering produced by this operator is identical to that produced by the scalar Morton ids described in section 2.1.1. The other logical operations can be readily derived from these two operations.

Algorithm 22 Finding the lesser of two Morton ids (sequential)

```
Input:  Two Morton ids, $A$ and $B$ with different anchors.
Output: $R$, the lesser of the two Morton ids.

$X_i \leftarrow (A_i \oplus B_i)$,  $i \in \{x, y, z\}$
$e \leftarrow \arg\max_i \left(\lfloor \log_2(X_i) \rfloor\right)$
if $A_e < B_e$
    $R \leftarrow A$
else
    $R \leftarrow B$
end if
```

APPENDIX C

ANALYSIS OF THE BLOCK PARTITIONING ALGORITHM

Assume that the input to the partitioning algorithm is a sorted distributed list of N octants. Then, we can guarantee coarsening of the input if there are more than eight octants[1] per processor. The minimum number of octants on any processor, n_{min}, can be expressed in terms of N and the imbalance factor[2], c, as follows:

$$n_{min} = \frac{N}{1 + c(n_p - 1)}.$$

This implies that the coarsening algorithm will coarsen the octree if,

$$\begin{aligned} n_{min} = \frac{N}{1 + c(n_p - 1)} &> 2^d, \\ \implies N > 2^d(1 + c(n_p - 1)). \end{aligned}$$

The total number of blocks created by our coarsening algorithm is $\mathcal{O}(p)$. Specifically, the total number of blocks produced by the coarsening algorithm, N_{blocks}, satisfies:

$$p \leq N_{blocks} < 2^d p.$$

If the input is sorted and if $c \approx 1$, then the communication cost for this partition is $\mathcal{O}(\frac{N}{n_p})$.

[1] 2^d cells for a d-tree.

[2] The imbalance factor is the ratio between the maximum and minimum number of octants on any processor.

APPENDIX D

SPECIAL CASE DURING CONSTRUCTION

We can not always guarantee the coarsest possible octree for an arbitrary distribution of N points and arbitrary values of N^p_{max}, especially when $N^p_{max} \approx \frac{N}{n_p}$. However, if every processor has at least two *well-separated* [1] points and if $N^p_{max} = 1$, then the algorithm will produce the coarsest possible octree under these constraints. However, this is not too restrictive because the input points can always be sampled in such a way that the algorithm produces the desired octree. Besides, the maximum depth of the octree can also be used to control the coarseness of the resulting octree. In all our experiments, we used $N^p_{max} = 1$ and we always got the same octree for different number of processor counts (Table 3).

[1] Convert the points into octants at D_{max} level. If there exists at least one coarse octant between these two octants, then the points are considered to be well-separated.

APPENDIX E

A_K IS A SYMMETRIC POSITIVE OPERATOR W.R.T. $(\cdot,\cdot)_K$

Since V_k is a finite-dimensional normed space, every linear operator on V_k is bounded, in particular A_k is bounded. Since V_k is a finite-dimensional space, it is complete with respect to any norm defined on that space and in particular with respect to the norm induced by the inner-product under consideration. Hence, the space V_k along with the respective inner-product $(\cdot,\cdot)_k$ forms a Hilbert space [74]. Hence, A_k has a unique Hilbert-adjoint operator; in fact, as Equation 32 shows A_k is also self-adjoint. Equation 5, the coercivity of $a(u,v)$ and the symmetricity of $a(u,v)$ and $(\cdot,\cdot)_k$ together lead to Equation 32.

$$\begin{aligned}(A_kv,v)_k &= a(v,v) > 0 \quad \forall v \neq 0 \in V_k \\ (A_kw,v)_k &= a(v,w) = (A_kv,w)_k = (w,A_kv)_k \quad \forall v,w \in V_k \end{aligned} \tag{32}$$

APPENDIX F

THE PROLONGATION MATRIX

Since the coarse-grid vector space is a subspace of the fine-grid vector space, any coarse-grid vector, v, can be expanded independently in terms of the fine and coarse-grid basis vectors.

$$v = \sum_{n=1}^{\#(V_{k-1})} v_{n,k-1}\phi_n^{k-1} = \sum_{m=1}^{\#(V_k)} v_{m,k}\phi_m^k \tag{33}$$

In Equation 33, $v_{n,k}$ and $v_{n,k-1}$ are the coefficients in the basis expansions for v on the fine and coarse grids, respectively. If we choose the standard finite element shape functions, then for each ϕ_i^k there exists a unique $p_i \in \Omega$ such that

$$\phi_j^k(p_i) = \delta_{ij} \quad \forall i,j = 1,2,\ldots,\#(V_k) \tag{34}$$

In Equation 34, δ_{ij} is the Kronecker delta function and p_i is the fine-grid vertex associated with ϕ_i^k. Equations 33 and 34 lead to

$$v_{i,k} = \sum_{j=1}^{\#(V_{k-1})} v_{j,k-1}\phi_j^{k-1}(p_i) \tag{35}$$

We can view the prolongation operator as a MatVec with the input vector as the coarse-grid nodal values (co-efficients in the basis expansion using the finite element shape functions as the basis vectors) and the output vector as the fine-grid nodal values. The matrix entries are then just the coarse-grid shape functions evaluated at the fine-grid vertices (Equation 36).

$$\boxed{\mathbf{P_1}(i,j) = \phi_j^{k-1}(p_i).} \tag{36}$$

An equivalent formulation is to satisfy Equation 10 in the variational sense by taking an inner-product with an arbitrary fine-grid test function. This formulation also produces the vector of fine-grid nodal values as a result of a MatVec with the vector of coarse-grid nodal values and the matrix is defined by Equation 37.

$$\boxed{\mathbf{P_2} = (\mathbf{M_k^k})^{-1}\mathbf{M_{k-1}^k}} \tag{37}$$

where,

$$\mathbf{M}_{\mathbf{k-1}}^{\mathbf{k}}(i,j) = (\phi_i^k, \phi_j^{k-1})_k. \tag{38}$$

Since the two formulations are equivalent, we have

$$\mathbf{P_1} = \mathbf{P_2}. \tag{39}$$

APPENDIX G

DERIVATION OF THE GALERKIN CONDITION

Define the functional

$$F^k(v_k) = \frac{1}{2}(A_k v_k, v_k)_k - (f_k, v_k)_k \quad \forall v_k \in V_k \tag{40}$$

Since A_k is a symmetric positive operator w.r.t $(\cdot,\cdot)_k$, the solution u_k to the Equation 6 satisfies

$$u_k = \arg\min_{\forall v_k \in V_k} F^k(v_k) \tag{41}$$

This is simply the Ritz FEM formulation. In the multigrid scheme, we want to find

$$v_{k-1} = \arg\min_{w_{k-1} \in V_{k-1}} F^k(v_k + Pw_{k-1}) \tag{42}$$

Here, P is the prolongation operator defined in Section 4.1.2.

$$\begin{aligned}
F^k(v_k + Pw_{k-1}) = & \\
& \tfrac{1}{2}((A_k v_k + A_k P w_{k-1}), (v_k + P w_{k-1}))_k - (f_k, v_k + P w_{k-1})_k \\
= & \tfrac{1}{2}(A_k v_k, v_k)_k + \tfrac{1}{2}(A_k P w_{k-1}, v_k)_k + \tfrac{1}{2}(A_k v_k, P w_{k-1})_k \\
& + \tfrac{1}{2}(A_k P w_{k-1}, P w_{k-1})_k - (f_k, v_k)_k \\
& - \tfrac{1}{2}(f_k, P w_{k-1})_k - \tfrac{1}{2}(f_k, P w_{k-1})_k \\
= & F^k(v_k) + \tfrac{1}{2}(A_k P w_{k-1}, v_k)_k + \tfrac{1}{2}((A_k v_k - f_k), P w_{k-1})_k \\
& - \tfrac{1}{2}(f_k, P w_{k-1})_k + \tfrac{1}{2}(P^* A_k P w_{k-1}, w_{k-1})_{k-1}
\end{aligned} \tag{43}$$

Here, P^* is the Hilbert adjoint operator of P with respect to the inner-products considered. Since, A_k is symmetric with respect to $(\cdot,\cdot)_k$ and since the vector spaces are real we have,

$$\frac{1}{2}(A_k P w_{k-1}, v_k)_k = \frac{1}{2}(P w_{k-1}, A_k v_k)_k = \frac{1}{2}(A_k v_k, P w_{k-1})_k \tag{44}$$

Hence, we have

$$F^k(v_k + P w_{k-1}) = F^k(v_k) + F_G^{k-1}(w_{k-1}) \tag{45}$$

with F_G^{k-1} defined by

$$F_G^{k-1}(v_{k-1}) = \frac{1}{2}(A_{k-1}^G v_{k-1}, v_{k-1})_{k-1} - (f_{k-1}^G, v_{k-1})_{k-1}. \tag{46}$$

A_{k-1}^G and f_{k-1}^G are defined by Equation 14 (The "*Galerkin*" condition). Equations 42 and 45 together lead to

$$v_{k-1} = \arg \min_{w_{k-1} \in V_{k-1}} F_G^{k-1}(w_{k-1}) \tag{47}$$

Equation 48 shows that A_{k-1}^G is symmetric with respect to $(\cdot,\cdot)_{k-1}$ and Equation 49 shows that it is also positive.

$$(A_{k-1}^G u, v)_{k-1} = (A_k P u, P v)_k = (P u, A_k P v)_k = (u, A_{k-1}^G v)_{k-1} \quad \forall u, v \in V_{k-1} \tag{48}$$

$$\begin{aligned} (A_{k-1}^G u, u)_{k-1} &= (A_k P u, P u)_k \quad \forall u \in V_{k-1} \\ \forall u \in V_{k-1}, \quad &\exists \quad w_u \in V_k \;\mid\; P u = w_u \\ \Rightarrow (A_{k-1}^G u, u)_{k-1} &= (A_k w_u, w_u)_k \geq 0 \quad \forall u \in V_{k-1} \end{aligned} \tag{49}$$

Hence, the solution v_{k-1} to Equation 13 satisfies Equation 47.

APPENDIX H

RESTRICTION MATRIX

Any fine-grid vector, w, and coarse-grid vector, v can be expanded in terms of the fine and coarse grid basis vectors respectively

$$w = \sum_{m=1}^{\#(V_k)} w_m \phi_m^k \quad \text{and} \quad v = \sum_{n=1}^{\#(V_{k-1})} v_n \phi_n^{k-1} \tag{50}$$

Now, let

$$R\phi_m^k = \sum_{l=1}^{\#(V_{k-1})} \mathbf{R}(l,m)\phi_l^{k-1} \quad \forall m = 1,2,\ldots,\#(V_k) \tag{51}$$

Using the definition of the restriction operator (Equation 15), we have

$$\begin{gathered}(R\phi_m^k, \phi_n^{k-1})_{k-1} = \sum_{l=1}^{\#(V_{k-1})} \mathbf{R}(l,m)(\phi_l^{k-1}, \phi_n^{k-1})_{k-1} = (\phi_m^k, \phi_n^{k-1})_k, \\ \forall m = 1,2,\ldots,\#(V_k) \quad \text{and} \quad \forall n = 1,2,\ldots,\#(V_{k-1}).\end{gathered} \tag{52}$$

Thus,

$$\boxed{\mathbf{R} = (\mathbf{M_{k-1}^{k-1}})^{-1}\mathbf{M_k^{k-1}}} \tag{53}$$

where,

$$\mathbf{M_k^{k-1}}(i,j) = (\phi_i^{k-1}, \phi_j^k)_k = \mathbf{M_{k-1}^k}(j,i) \tag{54}$$

APPENDIX I

AN EQUIVALENT MULTIGRID SCHEME

The coarse-grid operator defined in Equation 14 is expensive to build. Here, we will show that this operator is equivalent to the coarse-grid version of the operator defined in Equation 5. This operator can be implemented efficiently using a matrix-free scheme. Using Equations 8, 37, 14, and 16 we have

$$\begin{aligned}\mathbf{A^G_{k-1}} &= (\mathbf{M^{k-1}_{k-1}})^{-1}\mathbf{\tilde{A}^G_{k-1}} \\ \mathbf{\tilde{A}^G_{k-1}} &= \mathbf{M^{k-1}_{k}}(\mathbf{M^k_k})^{-1}\mathbf{\tilde{A}_k}(\mathbf{M^k_k})^{-1}\mathbf{M^k_{k-1}} \qquad (55)\end{aligned}$$

Since $V_{k-1} \subset V_k$, we can expand the coarse-grid basis vectors in terms of the fine-grid basis vectors as follows:

$$\phi_j^{k-1} = \sum_{i=1}^{\#(V_k)} \mathbf{c}(i,j)\phi_i^k \quad \forall j = 1, 2, \ldots, \#(V_{k-1}) \qquad (56)$$

By taking inner-products with arbitrary fine-grid test functions on either side of Equation 56, we have

$$(\phi_l^k, \phi_j^{k-1})_k = \sum_{i=1}^{\#(V_k)} \mathbf{c}(i,j)(\phi_l^k, \phi_i^k)_k,\ \forall j = 1, 2, \ldots, \#(V_{k-1}),\ \forall l = 1, 2, \ldots, \#(V_k) \qquad (57)$$

This leads to

$$\mathbf{c^{k-1}_k} = (\mathbf{M^k_k})^{-1}\mathbf{M^k_{k-1}} \qquad (58)$$

Using Equations 17, 55, 56, and 58 we can show that

$$\boxed{\mathbf{\tilde{A}_{k-1}} = \mathbf{\tilde{A}^G_{k-1}}\ ;\ \mathbf{A_{k-1}} = \mathbf{A^G_{k-1}}} \qquad (59)$$

Note that the fine-grid problem defined in Equation 6, the corresponding coarse-grid problem (Equation 13) and the restriction operator (Equation 16) all require inverting a mass-matrix. This could be quite expensive. Instead, we solve the following problem on the fine-grid

$$\tilde{A}_k u_k = \tilde{f}_k \qquad (60)$$

and solve the following corresponding coarse-grid problem

$$\mathbf{\tilde{A}_{k-1} e_{k-1}} = \mathbf{M^{k-1}_{k-1} f^{G}_{k-1}} = \mathbf{\tilde{R} r_k} = \mathbf{r_{k-1}} \tag{61}$$

for the coarse-grid representation of the error, e_{k-1}, using the fine-grid residual, r_k, after a few smoothing iterations. Here, $\tilde{R}$ is the modified restriction operator, which can be expressed as

$$\mathbf{\tilde{R}} = \mathbf{M^{k-1}_{k-1} R (M^{k}_{k})}^{-1} \tag{62}$$

Note, that this operator is the matrix-transpose of the prolongation operator derived using the variational formulation.

$$\boxed{\mathbf{\tilde{R}} = \mathbf{P_2}^T} \tag{63}$$

Since, $\mathbf{P_1} = \mathbf{P_2}$, we can use $\mathbf{P_1}^T$ instead of $\mathbf{\tilde{R}}$.

REFERENCES

[1] Adams, M. F., Bayraktar, H. H., Keaveny, T. M., and Papadopoulos, P., "Ultrascalable implicit finite element analyses in solid mechanics with over a half a billion degrees of freedom," in *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, ACM/IEEE, 2004.

[2] Adams, M. and Demmel, J. W., "Parallel multigrid solver for 3D unstructured finite element problems," in *SC '99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, ACM Press, 1999.

[3] Akcelik, V., Bielak, J., Biros, G., Epanomeritakis, I., Fernandez, A., Ghattas, O., Kim, E. J., Lopez, J., O'Hallaron, D. R., Tu, T., and Urbanic, J., "High resolution forward and inverse earthquake modeling on terascale computers," in *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, ACM, 2003.

[4] Akcelik, V., Biros, G., and Ghattas, O., "Parallel multiscale Gauss-Newton-Krylov methods for inverse wave propagation," in *SC '02: Proceedings of the 2002 IEEE/ACM Conference on Supercomputing*, IEEE, 2002.

[5] Alexander, D., Gee, J., and Bajcsy, R., "Elastic matching of diffusion tensor MRIs," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 1, pp. 244 – 249, 1999.

[6] Anderson, W., Gropp, W., Kaushik, D., Keyes, D., and Smith, B., "Achieving high sustained performance in an unstructured mesh CFD application," in *SC '99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, 1999.

[7] Arganda-Carreras, I., Sorzano, C. O., Marabini, R., Carazo, J. M., de Solorzano, C. O., and Kybic, J., "Consistent and elastic registration of histological sections using vector-spline regularization," in *Computer Vision Approaches to Medical Image Analysis (CVAMIA)*, vol. 4241 of *LNCS*, pp. 85 – 95, 2006.

[8] Ayala, D., Brunet, P., Juan, R., and Navazo, I., "Object representation by means of nonminimal division quadtrees and octrees," *ACM Transactions on Graphics*, vol. 4, no. 1, pp. 41 – 59, 1985.

[9] Balay, S., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H., "PETSc users manual," Tech. Rep. ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[10] Balay, S., Buschelman, K., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H., "PETSc home page," 2001. http://www.mcs.anl.gov/petsc (Accessed on March 20, 2009).

[11] Bank, R. E. and Dupont, T., "An optimal order process for solving finite element equations," *Mathematics of Computation*, vol. 36, no. 153, pp. 35 – 51, 1981.

[12] Bastian, P., Hackbusch, W., and Wittum, G., "Additive and multiplicative multi-grid A comparison," *Computing*, vol. 60, no. 4, pp. 345 – 364, 1998.

[13] Becker, R. and Braack, M., "Multigrid techniques for finite elements on locally refined meshes," *Numerical Linear Algebra with Applications*, vol. 7, pp. 363 – 379, 2000.

[14] Becker, R., Braack, M., and Richter, T., "Parallel multigrid on locally refined meshes," in *Reactive Flows, Diffusion and Transport*, pp. 77 – 92, Springer Berlin Heidelberg, 2007.

[15] Bergen, B., Hulsemann, F., and Rude, U., "Is 1.7×10^{10} unknowns the largest finite element system that can be solved today?," in *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, IEEE Computer Society, 2005.

[16] Bern, M. W., Eppstein, D., and Teng, S.-H., "Parallel construction of quadtrees and quality triangulations," *International Journal of Computational Geometry and Applications*, vol. 9, no. 6, pp. 517 – 532, 1999.

[17] Berti, G., "Image-based unstructured 3-D mesh generation for medical applications," in *European Congress on Computational Methods in Applied Sciences and Engineering*, 2004.

[18] Bittencourt, M. and Feij'oo, R., "Non-nested multigrid methods in finite element linear structural analysis," in *Virtual Proceedings of the 8th Copper Mountain Conference on Multigrid Methods (MGNET)*, 1997.

[19] Bohme, M., Hagenau, R., Modersitzki, J., and Siebert, B., "Non-linear image registration on PC-clusters using parallel FFT techniques," tech. rep., University of Lubeck, 2002.

[20] Bookstein, F. L., "Principal warps: Thin plate splines and the decomposition of deformations," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, no. 6, pp. 567 – 585, 1989.

[21] Braess, D. and Hackbusch, W., "A new convergence proof for the multigrid method including the V-cycle," *SIAM Journal on Numerical Analysis*, vol. 20, no. 5, pp. 967 – 975, 1983.

[22] Bramble, J. H., Pasciak, J. E., and Xu, J., "The analysis of multigrid algorithms for nonsymmetric and indefinite elliptic problems," *Mathematics of Computation*, vol. 51, no. 184, pp. 389 – 414, 1988.

[23] Bramble, J. H., Pasciak, J. E., and Xu, J., "Parallel multilevel preconditioners," *Mathematics of Computation*, vol. 55, no. 191, pp. 1 – 22, 1990.

[24] Brenner, S. C. and Scott, L. R., *The mathematical theory of finite element methods*, vol. 15 of *Texts in Applied Mathematics*. Springer-Verlag, 1994.

[25] Briggs, W. L., Henson, V. E., and McCormick, S. F., *A multigrid tutorial (2nd ed.)*. Society for Industrial and Applied Mathematics, 2000.

[26] Bro-Nielsen, M. and Gramkow, C., "Fast fluid registration of medical images," in *Visualization in Biomedical Computing (VBC'96)*, vol. 1131 of *LNCS*, pp. 267 – 276, 1996.

[27] Brunet, P. and Navazo, I., "Solid representation and operation using extended octrees," *ACM Transactions on Graphics*, vol. 9, no. 2, pp. 170 – 197, 1990.

[28] Bungartz, H.-J., Mehl, M., and Weinzierl, T., "A parallel adaptive cartesian PDE solver using space-filling curves," in *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference* (Nagel, E. W., Walter, V. W., and Lehner, W., eds.), vol. 4128 of *LNCS*, pp. 1064 – 1074, Springer-Verlag, 2006.

[29] Burstedde, C., Ghattas, O., Gurnis, M., Stadler, G., Tan, E., Tu, T., Wilcox, L. C., and Zhong, S., "Scalable adaptive mantle convection simulation on petascale supercomputers," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, IEEE Press, 2008.

[30] Campbell, P. M., Devine, K. D., Flaherty, J. E., Gervasio, L. G., and Teresco, J. D., "Dynamic octree load balancing using space-filling curves," Tech. Rep. CS-03-01, Williams College Department of Computer Science, 2003.

[31] Clarenz, U., Droske, M., and Rumpf, M., "Towards fast nonrigid registration," in *Inverse Problems, Image Analysis and Medical Imaging, AMS Special Session Interaction of Inverse Problems and Image Analysis*, pp. 67 – 84, AMS, 2002.

[32] Corman, T., Leiserson, C., and Rivest, R., *Introduction to Algorithms.* MIT Press, 1990.

[33] Crum, W., Hartkens, T., and Hill, D., "Non-rigid image registration: Theory and practice," *The British Journal of Radiology*, vol. 77, pp. S140 – S153, 2004.

[34] Davis, M. H., Khotanzad, A., Flamig, D. P., and Harms, S. E., "Coordinate transformation in 3D image matching by a physics based method-elastic body splines," in *International Symposium on Computer Vision*, pp. 218 – 222, 1995.

[35] Deen, W. M., *Analysis of transport phenomena.* Topics in Chemical Engineering, Oxford University Press, 1998.

[36] Dendy, J. E., "Black box multigrid," *Journal of Computational Physics*, vol. 48, pp. 366 – 386, 1982.

[37] El-Ghazawi, T. A., Chalermwat, P., and Moigne, J. L., "Wavelet-based image registration on parallel computers," in *SC '97: Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, 1997.

[38] Finkel, R. A. and Bentley, J. L., "Quad trees: A data structure for retrieval on composite keys," *Acta Informatica*, vol. 4, pp. 1 – 9, 1974.

[39] Fischer, B. and Modersitzki, J., "Fast diffusion registration," in *AMS Contemporary Mathematics, Inverse Problems, Image Analysis, and Medical Imaging*, vol. 313, pp. 117 – 129, 2002.

[40] Fischer, B. and Modersitzki, J., “Curvature based image registration,” *Journal of Mathematical Imaging and Vision*, vol. 18, pp. 81 – 85, 2003.

[41] Fornefett, M., Rohr, K., and Stiehl, H., “Radial basis functions with compact support for elastic registration of medical images,” *Image and Vision Computing*, vol. 19, pp. 87 – 96, 2001.

[42] Fornefett, M., Rohr, K., and Stiehl, H. S., “Elastic registration of medical images using radial basis functions with compact support,” in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 1, pp. 402 – 407, 1999.

[43] Freitag, L. and Loy, R., “Adaptive, multiresolution visualization of large data sets using a distributed memory octree,” in *SC ’99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, 1999.

[44] Frisken, S. and Perry, R., “Simple and efficient traversal methods for quadtrees and octrees,” *Journal of Graphics Tools*, vol. 7, no. 3, pp. 1 – 11, 2002.

[45] Glowinski, R., Pan, T.-W., Hesla, T. I., Joseph, D. D., and Periaux, J., “A fictitious domain method with distributed lagrange multipliers for the numerical simulation of particulate flow,” *Contemporary Mathematics*, vol. 218, pp. 121 – 137, 1998.

[46] Golomb, S., “Run-length encodings,” *IEEE Transactions on Information Theory*, vol. 12, no. 3, pp. 399 – 401, 1966.

[47] Gorelick, L., Galun, M., and Brandt, A., “Shape representation and classification using the Poisson equation,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, no. 12, pp. 1991 – 2005, 2006.

[48] Goshtasby, A., “Piecewise cubic mapping functions for image registration,” *Pattern Recognition*, vol. 20, pp. 525 – 533, 1987.

[49] Goshtasby, A., “Image registration by local appoximation methods,” *Image and Vision Computing*, vol. 6, no. 4, pp. 255 – 261, 1988.

[50] gottesfeld brown, L., “A survey of image registration techniques,” *ACM Computing Surveys*, vol. 24, no. 4, pp. 325 – 376, 1992.

[51] Grama, A., Gupta, A., Karypis, G., and Kumar, V., *An Introduction to Parallel Computing: Design and Analysis of Algorithms.* Addison Wesley, second ed., 2003.

[52] Greaves, D. M. and Borthwick, A. G. L., “Hierarchical tree-based finite element mesh generation,” *International Journal for Numerical Methods in Engineering*, vol. 45, no. 4, pp. 447 – 471, 1999.

[53] Griebel, M. and Zumbusch, G., “Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves,” *Parallel Computing*, vol. 25, no. 7, pp. 827 – 843, 1999.

[54] Griffiths, D. J., *Introduction to electrodynamics.* Prentice-Hall, 1999.

[55] Gropp, W., Kaushik, D., Keyes, D., and Smith, B., "Performance modeling and tuning of an unstructured mesh CFD application," in *SC2000: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, 2000.

[56] Gurtin, M. E., *An introduction to continuum mechanics*, vol. 158 of *Mathematics in Science and Engineering.* Academic Press, 2003.

[57] Haber, E. and Heldmann, S., "An octree multigrid method for quasi-static Maxwell's equations with highly discontinuous coefficients," *Journal of Computational Physics*, vol. 223, no. 2, pp. 783 – 796, 2007.

[58] Haber, E., Heldmann, S., and Modersitzki, J., "An octree method for parametric image registration," *SIAM Journal on Scientific Computing*, vol. 29, no. 5, pp. 2008 – 2023, 2007.

[59] Haber, E., Heldmann, S., and Modersitzki, J., "Adaptive mesh refinement for nonparametric image registration," *SIAM Journal on Scientific Computing*, vol. 30, no. 6, pp. 3012 – 3027, 2008.

[60] Haber, E. and Modersitzki, J., "A multilevel method for image registration," *SIAM Journal on Scientific Computing*, vol. 27, no. 5, pp. 1594 – 1607, 2006.

[61] Hackbusch, W., *Multigrid methods and applications*, vol. 4 of *Springer Series in Computational Mathematics.* Springer-Verlag, 1985.

[62] Hariharan, B., Aluru, S., and Shanker, B., "A scalable parallel fast multipole method for analysis of scattering from perfect electrically conducting surfaces," in *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pp. 1 – 17, IEEE Computer Society Press, 2002.

[63] Henn, S. and Witsch, K., "Iterative multigrid regularization techniques for image matching," *SIAM Journal on Scientific Computing*, vol. 23, no. 4, pp. 1077 – 1093, 2001.

[64] Henson, V. E. and Yang, U. M., "BoomerAMG: A parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, no. 1, pp. 155 – 177, 2002.

[65] Herzen, B. V. and Barr, A. H., "Accurate triangulations of deformed, intersecting surfaces," in *SIGGRAPH '87: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 103 – 110, ACM Press, 1987.

[66] Hill, D. L. G., Batchelor, P. G., Holden, M., and Hawkes, D. J., "Medical image registration," *Physics in Medicine and Biology*, vol. 46, pp. R1 – R45, 2001.

[67] Homke, L., "A multigrid method for anisotropic PDEs in elastic image registration," *Numerical Linear Algebra with Applications*, vol. 13, pp. 215 – 229, 2006.

[68] Ibanez, L., Schroeder, W., Ng, L., and Cates, J., *The ITK Software Guide.* Kitware, Inc., second ed., 2005.

[69] Ino, F., Kawasaki, Y., Tashiro, T., Nakajima, Y., Sato, Y., Tamura, S., and Hagihara, K., "A parallel implementation of 2-D/3-D image registration for computer-assisted surgery," in *11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, 2005.

[70] Ino, F., Ooyama, K., and Hagihara, K., "A data distributed parallel algorithm for nonrigid image registration," *Parallel Computing*, vol. 31, pp. 19 – 43, 2005.

[71] Jones, A. and Jimack, P., "An adaptive multigrid tool for elliptic and parabolic systems," *International Journal for Numerical Methods in Fluids*, vol. 47, pp. 1123 – 1128, 2005.

[72] Kalmoun, E. M., Kostler, H., and Rude, U., "3-D optical flow computation using a parallel variational multigrid scheme with application to cardiac C-arm CT motion," *Image and Vision Computing*, vol. 25, pp. 1482 – 1494, 2007.

[73] Kim, E., Bielak, J., Ghattas, O., and Wang, J., "Octree-based finite element method for large-scale earthquake ground motion modeling in heterogeneous basins," *AGU Fall Meeting Abstracts*, 2002.

[74] Kreyszig, E., *Introductory functional analysis with applications.* John Wiley and Sons, Inc., 1989.

[75] Kruger, S. and Calway, A., "Image registration using multiresolution frequency domain correlation," in *British Machine Vision Conference*, pp. 316 – 325, 1998.

[76] Kybic, J. and Unser, M., "Multiresolution spline warping for EPI registration," in *Proceedings of SPIE*, pp. 571 – 579, 1999.

[77] Kybic, J. and Unser, M., "Fast parametric elastic image registration," *IEEE Transactions on Image Processing*, vol. 12, no. 11, pp. 1427 – 1442, 2003.

[78] Lang, S., "Parallel-adaptive simulation with the multigrid-based software framework UG," *Engineering with Computers*, vol. 22, pp. 157 – 179, 2006.

[79] Lelewer, D. A. and Hirschberg, D. S., "Data compression," *ACM Computing Surveys*, vol. 19, no. 3, pp. 261 – 296, 1987.

[80] Lester, H. and Arridge, S. R., "A survey of hierarchical non-linear medical image registration," *Pattern Recognition*, vol. 32, pp. 129 – 149, 1999.

[81] Li, X. S. and Demmel, J. W., "SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems," *ACM Transactions on Mathematical Software*, vol. 29, pp. 110 – 140, June 2003.

[82] Maintz, J. B. A. and Viergever, M. A., "A survey of medical image registration," *Medical Image Analysis*, vol. 2, no. 1, pp. 1 – 36, 1998.

[83] Mavriplis, D. J., Aftosmis, M. J., and Berger, M., "High resolution aerospace applications using the NASA Columbia Supercomputer," in *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, IEEE Computer Society, 2005.

[84] Meagher, D., "Geometric modeling using octree encoding," *Computer Graphics and Image Processing*, vol. 19, pp. 129 – 147, 1982.

[85] Mehl, M., "Cache-optimal data-structures for hierarchical methods on adaptively refined space-partitioning grids," Sept. 2006.

[86] Modersitzki, J., Lustig, G., Schmitt, O., and Obeloer, W., "Elastic registration of brain images on large pc-clusters," *Future Generation Computer Systems*, vol. 18, pp. 115 – 125, 2001.

[87] Modersitzki, J., ObelSer, W., Schmitt, O., and Lustig, G., "Elastic matching of very large digital images on high performance clusters," in *Proceedings of the 7th International Conference on High-Performance Computing and Networking*, vol. 1593 of *LNCS*, pp. 141 – 149, 1999.

[88] Modersitzki, J., *Numerical Methods for Image Registration.* Numerical Mathematics and Scientific Computation, Oxford Univ Press, 2004.

[89] Moigne, J. L., Campbell, W. J., and Cromp, R. F., "An automated parallel image registration technique based on the correlation of wavelet features," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 40, no. 8, pp. 1849 – 1864, 2002.

[90] Moore, D., "The cost of balancing generalized quadtrees.," in *Symposium on Solid Modeling and Applications*, pp. 305 – 312, 1995.

[91] Narasimhan, S., Mundani, R.-P., and Bungartz, H.-J., "An octree and a graph-based approach to support location aware navigation services," in *Proceedings of the 2006 International Conference on Pervasive Systems & Computing, (PSC 2006)*, pp. 24 – 30, CSREA Press, 2006.

[92] Nocedal, J. and Wright, S. J., *Numerical Optimization.* Springer series in operations research, Springer, 2006.

[93] Popinet, S., "Gerris: A tree-based adaptive solver for the incompressible Euler equations in complex geometries," *Journal of Computational Physics*, vol. 190, pp. 572 – 600, 2003.

[94] Ramière, I., Angot, P., and Belliard, M., "A general fictitious domain method with immersed jumps and multilevel nested structured meshes," *Journal of Computational Physics*, vol. 225, no. 2, pp. 1347 – 1387, 2007.

[95] Rice, R. F., "Some practical universal noiseless coding techniques," Tech. Rep. JPL Publication 79-22, Jet Propulsion Laboratory, Pasadena, California, 1979.

[96] Roche, A., Malandain, G., Ayache, N., and Prima, S., "Towards a better comprehension of similarity measures used in medical image registration," in *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, vol. 1679 of *LNCS*, pp. 555 – 567, 1999.

[97] Rohr, K., Stiehl, H., Sprengel, R., Beil, W., Buzug, T., Weese, J., and Kuhn, M., "Point-based elastic registration of medical image data using approximating thin-plate splines," *Visualization in Biomedical Computing*, pp. 297 – 306, 1996.

[98] Rohr, K., "Spline-based elastic image registration," *Proceedings in Applied Mathematics and Mechanics*, vol. 3, pp. 36 – 39, 2003.

[99] Sainte-Marie, J., Chapelle, D., Cimrman, R., and Sorine, M., "Modeling and estimation of the cardiac electromechanical activity," *Computers and Structures*, vol. 84, pp. 1743 – 1759, 2006.

[100] Samet, H., "The quadtree and related hierarchical data structures," *ACM Computing Surveys*, vol. 16, no. 2, pp. 187 – 260, 1984.

[101] Sampath, R. S., Adavani, S. S., Sundar, H., Lashuk, I., and Biros, G., "Dendro: Parallel algorithms for multigrid and AMR methods on 2:1 balanced octrees," in *SC '08: Proceedings of the 2008 IEEE/ACM Conference on Supercomputing*, IEEE, 2008.

[102] Sampath, R. S. and Biros, G., "A parallel geometric multigrid method for finite elements on octree meshes," 2008. Submitted for publication.

[103] Sampath, R. S. and Biros, G., "Parallel elastic registration using a multigrid preconditioned Gauss Newton Krylov solver, grid continuation and octrees," 2009. Submitted for publication.

[104] Sampath, R. S., Sundar, H., Adavani, S. S., Lashuk, I., and Biros, G., "Dendro home page," 2008. http://www.cc.gatech.edu/csela/dendro (Accessed on March 20, 2009).

[105] Schmitt, O., Modersitzki, J., Heldmann, S., Wirtz, S., and Fischer, B., "Image registration of sectioned brains," *International Journal of Computer Vision*, vol. 73, no. 1, pp. 5 – 39, 2007.

[106] Schneiders, R., "An algorithm for the generation of hexahedral element meshes based on an octree technique," in *Proceedings of the 6th International Meshing Roundtable*, pp. 183 – 194, 1997.

[107] Schneiders, R., Schindler, R., and Weiler, F., "Octree-based generation of hexahedral element meshes," in *Proceedings of the 5th International Meshing Roundtable*, pp. 205 – 216, 1996.

[108] Sermesant, M., Moireau, P., Camara, O., Sainte-Marie, J., Andriantsimiavona, R., Cimrman, R., Hill, D., Chapelle, D., and Razavi, R., "Cardiac function estimation from MRI using a heart model and data assimilation: Advances and difficulties," *Medical Image Analysis*, vol. 10, pp. 642 – 656, 2006.

[109] Shapira, Y., "Multigrid for locally refined meshes," *SIAM Journal on Scientific Computing*, vol. 21, no. 3, pp. 1168 – 1190, 1999.

[110] Shephard, M. and Georges, M., "Automatic three-dimensional mesh generation by the finite octree technique," *International Journal for Numerical Methods in Engineering*, vol. 26, pp. 709 – 749, 1991.

[111] Shewchuk, J. R., "Tetrahedral mesh generation by Delaunay refinement," in *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*, pp. 86 – 95, Association for Computing Machinery, June 1998.

[112] Somani, A. K. and Sansano, A. M., "Minimizing overhead in parallel algorithms through overlapping communication/computation," tech. rep., Institute for Computer Applications in Science and Engineering (ICASE), 1997.

[113] Sorzano, C. O. S., Thevenaz, P., and Unser, M., "Elastic registration of biological images using vector-spline regularization," *IEEE Transactions on Biomedical Engineering*, vol. 52, no. 4, pp. 652 – 663, 2005.

[114] Spekreijse, S. P., "Elliptic grid generation based on Laplace equations and algebraic transformations," *Journal of Computational Physics*, vol. 118, no. 1, pp. 38 – 61, 1995.

[115] Stefanescu, R., Pennec, X., and Ayache, N., "Grid-enabled non-rigid registration of medical images," *Parallel Processing Letters*, vol. 14, no. 2, pp. 197 – 216, 2004.

[116] Strasters, K. and Gerbrands, J., "3-dimensional image segmentation using a split, merge and group-approach," *Pattern Recognition Letters*, vol. 12, pp. 307 – 325, May 1991.

[117] Sundar, H., *Spatio-temporal deformation analysis of cardiac MR images.* PhD thesis, University of Pennsylvania, 2008.

[118] Sundar, H., Sampath, R. S., Adavani, S. S., Davatzikos, C., and Biros, G., "Low-constant parallel algorithms for finite element simulations using linear octrees," in *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, ACM Press, 2007.

[119] Sundar, H., Sampath, R. S., and Biros, G., "Bottom-up construction and 2:1 balance refinement of linear octrees in parallel," *SIAM Journal on Scientific Computing*, vol. 30, no. 5, pp. 2675 – 2708, 2008.

[120] Szeliski, R. and Lavallee, S., "Matching 3-D anatomical surfaces with nonrigid deformations using octree-splines," *International Journal of Computer Vision*, vol. 18, no. 2, pp. 171 – 186, 1996.

[121] Szeliski, R. and Shum, H.-Y., "Motion estimation with quadtree splines," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 18, no. 12, pp. 1199 – 1210, 1996.

[122] TACC, "Ranger's system architecture." http://www.tacc.utexas.edu (Accessed on March 20, 2009).

[123] Tatebe, O. and Oyanagi, Y., "Efficient implementation of the multigrid preconditioned Conjugate Gradient method on distributed memory machines," in *Supercomputing '94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, pp. 194 – 203, ACM, 1994.

[124] Teng, S.-H., "Provably good partitioning and load balancing algorithms for parallel adaptive N-body simulation," *SIAM Journal on Scientific Computing*, vol. 19, no. 2, pp. 635 – 656, 1998.

[125] Tropf, H. and Herzog, H., "Multidimensional range search in dynamically balanced trees," *Angewandte Informatik*, vol. 2, pp. 71 – 77, 1981.

[126] Trottenberg, U. and Oosterlee, C. W. and Schuller, A., *Multigrid*. Academic Press Inc., 2001.

[127] Tu, T. and O'Hallaron, D. R., "Balance refinement of massive linear octree datasets," *CMU Technical Report*, vol. CMU-CS-04, no. 129, 2004.

[128] Tu, T. and O'Hallaron, D. R., "Extracting hexahedral mesh structures from balanced linear octrees," in *Proceedings of the 13th International Meshing Roundtable*, pp. 191 – 200, 2004.

[129] Tu, T., O'Hallaron, D. R., and Ghattas, O., "Scalable parallel octree meshing for terascale applications," in *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, IEEE Computer Society, 2005.

[130] Tu, T., Yu, H., Ramirez-Guzman, L., Bielak, J., Ghattas, O., Ma, K.-L., and O'Hallaron, D. R., "From mesh generation to scientific visualization: An end-to-end approach to parallel supercomputing," in *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ACM Press, 2006.

[131] Wachowiak, M. P. and Peters, T. M., "Parallel optimization approaches for medical image registration," in *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, vol. 3216 of *LNCS*, pp. 781 – 788, Springer-Verlag, 2004.

[132] Wang, W., "Special bilinear quadrilateral elements for locally refined finite element grids," *SIAM Journal on Scientific Computing*, vol. 22, pp. 2029 – 2050, 2001.

[133] Warren, M. S. and Salmon, J. K., "Astrophysical N-body simulations using hierarchical tree data structures," in *SC '92: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, pp. 570 – 576, IEEE Computer Society Press, 1992.

[134] Warren, M. S. and Salmon, J. K., "A parallel hashed octree N-body algorithm," in *SC '93: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, 1993.

[135] White, B. S., McKee, S. A., de Supinski, B. R., Miller, B., Quinlan, D., and Schulz, M., "Improving the computational intensity of unstructured mesh applications," in *ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing*, pp. 341 – 350, ACM Press, 2005.

[136] Wirtz, S., Fischer, B., Modersitzki, J., and Schmitt, O., "Super-fast elastic registration of histologic images of a whole rat brain for three-dimensional reconstruction," *Medical Imaging*, vol. 5370, pp. 328 – 334, 2004.

[137] Wollny, G. and Kruggel, F., "Computional cost of nonrigid registration algorithms based on fluid dynamics," *IEEE Transactions on Medical Imaging*, vol. 21, no. 8, pp. 946 – 952, 2002.

[138] Worz, S. and Rohr, K., "Physics-based elastic registration using non-radial basis functions and including landmark localization uncertainties," *Computer Vision and Image Understanding*, vol. 111, pp. 263 – 274, 2008.

[139] Ying, L., Biros, G., and Zorin, D., "A kernel-independent adaptive fast multipole algorithm in two and three dimensions," *Journal of Computational Physics*, vol. 196, no. 2, pp. 591 – 626, 2004.

[140] Ying, L., Biros, G., Zorin, D., and Langston, H., "A new parallel kernel-independent fast multipole method," in *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, IEEE Computer Society, 2003.

[141] Yserentant, H., "On the convergence of multi-level methods for strongly nonuniform families of grids and any number of smoothing steps per level," *Computing*, vol. 30, pp. 305 – 313, 1983.

[142] Yserentant, H., "The convergence of multilevel methods for solving finite-element equations in the presence of singularities," *Mathematics of Computation*, vol. 47, no. 176, pp. 399 – 409, 1986.

[143] Zhang, S., "Optimal-order nonnested multigrid methods for solving finite element equations. I. On quasi-uniform meshes," *Mathematics of Computation*, vol. 55, no. 191, pp. 23 – 36, 1990.

[144] Zhang, S., "Optimal-order nonnested multigrid methods for solving finite element equations. II. On nonquasiuniform meshes," *Mathematics of Computation*, vol. 55, no. 192, pp. 439 – 450, 1990.

[145] Zitova, B. and Flusser, J., "Image registration methods: A survey," *Image and Vision Computing*, vol. 21, pp. 977 – 1000, 2003.