PETASCALE ADAPTIVE COMPUTATIONAL FLUID DYNAMICS

By

Min Zhou

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Mechanical Engineering

Approved by the
Examining Committee:

Dr. Kenneth E. Jansen, Thesis Adviser

Dr. Mark S. Shephard, Thesis Adviser

Dr. Christopher D. Carothers, Member

Dr. Charles A. Taylor, Member

Dr. Assad A. Oberai, Member

Dr. Onkar Sahni, Member

Rensselaer Polytechnic Institute
Troy, New York

August 2009
(For Graduation December 2009)

© Copyright 2009
by
Min Zhou
All Rights Reserved

CONTENTS

LIST OF TABLES iv

LIST OF FIGURES vi

ACKNOWLEDGMENT xi

ABSTRACT xiii

1. INTRODUCTION 1

1.1 Background and motivation 1

1.2 Framework of parallel adaptive simulation and the current status . . 4

1.3 Outline of thesis 8

2. NUMERICAL METHODS 10

2.1 Governing equations and flow solver 11

2.2 Parallel paradigm 15

2.3 Notion of a complete value 20

2.4 Communication steps and structures 21

2.5 Parallel implicit solve 23

3. PRE-SOLVER AND MESH ADAPTATION 26

3.1 Introduction to FMDB 26

3.1.1 Adjacency and Distributed Mesh Representation 27

3.2 Structure of *phParAdapt* 29

3.3 Open source attribute and modeler 31

3.4 Predictive load balance in mesh adaptation 33

3.4.1 Mesh metric field 34

3.4.2 Algorithm of predictive load balance 36

3.4.3 Applications of predictive load balance 38

3.4.3.1 AAA model 38

3.4.3.2 Straight pipe model with air bubbles 40

3.4.4 Discussions and summary 44

3.5 Multiple parts per processor 44

3.5.1 Design of multiple parts per processor 45

3.5.1.1 Partition redistribution . . . 46
3.5.1.2 Multiple parts per processor in *phParAdapt* . . . 49
3.5.2 Examples/Applications . . . 51
3.5.2.1 AAA model with $O(10^8)$ elements . . . 52
3.5.2.2 AAA model with $O(10^9)$ elements . . . 54
3.5.3 Discussion and summary . . . 58
4. PARTITION IMPROVEMENT ALGORITHMS . . . 60
4.1 Algorithm of MCOPI . . . 62
4.1.1 LIIPBMod . . . 62
4.1.2 HPS . . . 64
4.2 Results and discussions . . . 67
4.2.1 Graph/hypergraph-based partitions . . . 69
4.2.1.1 ParMetis on a bifurcation pipe model . . . 69
4.2.1.2 Zoltan PHG on an AAA model . . . 70
4.2.2 Balance improvement by MCOPI algorithms . . . 71
4.2.2.1 LIIPBMod . . . 71
4.2.2.2 HPS . . . 75
4.2.3 Performance improvement of the finite element analysis . . . 77
4.2.3.1 LIIPBMod on an AAA model . . . 77
5. DATA REORDERING ALGORITHM . . . 81
5.1 Algorithm of data reordering . . . 83
5.1.1 Linear finite element data access . . . 86
5.1.2 Adjacency-based reordering algorithm . . . 89
5.2 Results and discussions . . . 91
5.2.1 Relative effect of data reordering . . . 92
5.2.2 Effect of processor load (strong scaling) . . . 93
5.2.3 Effect of different computer architectures . . . 95
5.2.4 Cache performance modeling . . . 97
6. APPLICATION TO LARGE CARDIOVASCULAR SYSTEM . . . 104
6.1 Blood flow simulation in a 3-D "whole" body model . . . 105
7. CONCLUSIONS AND FUTURE WORK . . . 117
REFERENCES . . . 121

LIST OF TABLES

3.1 The average number of regions and vertices of a 133 million element mesh of an AAA model, along with the imbalance ratios. 53

3.2 Strong scaling results on an AAA model with 133M elements on Intrepid:IBM BG/P (execution time is measured in seconds). 54

3.3 The average number of regions and vertices of a 1.07 billion element mesh of an AAA model, along with the imbalance ratios. 55

3.4 Strong scaling results on an AAA model with $O(10^9)$ elements up to 163,840 cores on Intrepid:IBM BG/P (execution time is measured in seconds). 58

3.5 Strong scaling results on an AAA model with $O(10^9)$ elements up to 65,536 cores on Kraken: Cray XT5. 58

4.1 Average number of regions and vertices on each part along with the region imbalance and vertex imbalance ratios of different partitions on a bifurcation pipe model with 42M elements. 70

4.2 The comparison of region and vertex imbalance ratios before and after using LIIPBMod algorithm on a bifurcation pipe model with a $42M$ element mesh along with the time usage of ParMetis partitioner and LIIPBMod algorithm. 72

4.3 The comparison of region and vertex imbalance ratios before and after using LIIPBMod algorithm on a $1.07B$ element mesh on an AAA model along with time usage of partition and the number of iterations. 74

4.4 Strong scaling results on an AAA model with $O(10^9)$ elements up to 163,840 cores on Intrepid:IBM BG/P with LIIPBMod algorithm. 77

5.1 Definitions of symbols in the pseudo code. 88

5.2 The status of the queue and the nodes labeled on each pass through . . 91

5.3 Data reordering effect on different phases of implicit FEA. 93

5.4 Average number of regions and vertices per part of an 8.8 million region mesh of a bifurcation pipe model. 94

5.5 Time usage of FEA solver on a bifurcation pipe model on CCNI's BG/L along with performance improvement. 95

5.6 Time usage of FEA solver on an AAA model with and without using data reordering algorithm on different supercomputers along with performance improvement. 96

5.7 Summary of systems considered in this study. SPN refers to sockets per node, CPCN to cores per compute node and CPS to cores per socket. The tests reported in this paper use all the cores on each node. S refers to shared cache on a socket among its cores while P refers to private cache on cores of a socket. 98

5.8 Time usage of FEA solver on different meshes with and without using data reordering algorithm on NICS Kraken-XT5. 98

5.9 Comparison of cache access and cache misses for different meshes of a bifurcation model on NICS Kraken-XT5 with and without using data reordering. 100

5.10 Comparison of normalized cache access and cache misses for different meshes of a bifurcation model on NICS Kraken-XT5 with and without using data reordering algorithm. Case of 8.8 million region mesh is taken as the base to normalize, i.e., normalized by the first row in Table 5.9. The information is normalized again (in the parentheses) by the relative work load which is in the first column (of a given row) of this table. 100

5.11 Scaling factors (s_e) of FEA solver on different meshes with and without using data reordering algorithm on NICS Kraken-XT5. 101

LIST OF FIGURES

1.1 Key steps of parallel adaptive flow simulation 5

1.2 Anisotropic adaptivity, preserving boundary layer mesh structure in an artery model [66]. 8

2.1 Partition of a coarse unstructured mesh of arterial bypass (colors indicate different parts). 16

2.2 Solid dots indicate shared *dofs*. 17

2.3 Control relationships are established between multiple images of shared *dofs* (*dofs* are locally numbered on each part). Solid dot denotes an owner image whereas hollow ones indicate non-owners. 18

2.4 Schematic of partition-graph (arrows indicate interaction between two parts). 19

2.5 Communication steps involved with shared *dofs*. 22

3.1 Distributed mesh on three parts P_0, P_1 and P_2 [63]. 28

3.2 A demonstration of preprocessing and the relationship between the softwares. 29

3.3 The boundary and initial conditions applied on the bifurcation model . 33

3.4 A demonstration of the predictive load balance algorithm. 35

3.5 Ellipsoidal surface associated with the quadratic form 35

3.6 Geometry and a mesh of an AAA model. 38

3.7 (a) Number of regions and (b) memory usage on each part before and after mesh adaptation with and without using *PredLB* on an AAA model. Memory usage is measured in KB. 39

3.8 The number of regions on each part before and after mesh adaptation with predictive load balance on an AAA model. 40

3.9 The size field of an "air bubble". 41

3.10 A segment of a straight pipe model with distributed air bubbles colored by the magnitude of size field along with examples initial mesh and adapted mesh zoomed for 1/2 bubble. 42

3.11 The number of regions on each part of the 1024 parts before and after mesh adaptation with and without *PredLB* on a straight pipe model with 5 air bubbles. 42

3.12 The number of regions on each part of the 16,384 parts before and after mesh adaptation with and without *PredLB* on a straight pipe model with 160 air bubbles. 43

3.13 An example of multiple parts per processor. The solid lines represent the inter-processor part boundaries and the dashed lines represent the intra-processor part boundaries. 45

3.14 A distributed mesh and its corresponding part graph. (a) The mesh is partitioned into 3 parts and distributed on 3 processors (i.e., one part per processor). (b) Corresponding part graph. The circles represent the graph nodes. The red solid edges demonstrate the intra-processor graph edges and the green dashed ones demonstrate the inter-processor graph edges. 47

3.15 The graph and the partition given by (a) global partitioning and (b) local partitioning. 48

3.16 An example of dead lock in the communication 51

3.17 Number of regions on each part of a 133 million element mesh of an AAA model for a partition with 8,192 parts. 53

3.18 Number of regions on each part of a 1.07 billion element mesh of an AAA model for a partition with 163,840 parts. 56

3.19 Strong scaling performance for an AAA model with $O(10^9)$ elements up to 163,840 cores of Intrepid, IBM BG/P system. In this case, execution time is measured in terms of equation formation, equation solution and the total. 57

4.1 Partition before and after LIIPBMod, dashed lines are inter-part boundaries. 65

4.2 An example of the migration of mesh entities that causes bad partition boundary. 65

4.3 Algorithm of HPS 66

4.4 Rules to select mesh entities to be migrated from heavily loaded parts to empty parts. 67

4.5 Geometry model of a bifurcation pipe. 68

4.6 Comparison of normalized number of regions and vertices on each part of a 16,384 part mesh before and after LIIPBMod on a bifurcation pipe model. 73

4.7 Comparison of normalized number of vertices on each part of a 163,840 part partition before and after LIIPBMod on an AAA model. 75

4.8 Comparison of normalized number of vertices on each part of the partition given by graph-based partitioner and HPS on the bifurcation pipe model and the AAA model 76

4.9 Strong scaling performance for an AAA model with $O(10^9)$ regions up to 163,840 cores of Intrepid, IBM system with LIIPBMod. 78

4.10 Comparison of scaling factors of equation solution with and without LIIPBMod for an AAA model with $O(10^9)$ regions up to 163,840 cores of Intrepid, IBM system. 78

4.11 Analysis of parallel performance in terms of computation and communication time for an AAA model with $O(10^9)$ elements up to 163,840 cores of Intrepid, IBM BG/P system (communication time is composed of point-to-point and collective ones) [69]. 80

5.1 All the vertices that bound the five elements around a given vertex 29. . 83

5.2 Constant time operations are used to determine the adjacent vertices for a given vertex. 85

5.3 A 2D example of nodal and element labeling 89

5.4 FEA solver performance improvement due to current data reordering algorithm on a bifurcation pipe model on CCNI's BG/L. 95

5.5 FEA solver performance improvement using current data reordering algorithm on an AAA model on various supercomputers including IBM Blue Gene (CCNI-BGL and Intrepid-BGP), Cray XT (Bigben-XT3 and Kraken-XT5) and Sun Constellation Cluster (Ranger). 97

6.1 The CT dataset (left) and the model projected over the iso-surfaces (right) 105

6.2 Whole body model composed of 78 arteries along with labels for sections (left) and labels of main arteries in the magnified view (right). The inset shows the volumetric flow rate at inlet (the two dots correspond to instants of peak systole and early diastole). 106

6.3 The initial boundary layer mesh on the "whole" body model with 7.1M elements. 107

6.4 The adapted mesh (4th adaptation) on the "whole" body model with 42.8M elements. 108

6.5 The comparison of initial and adapted meshes (4th adaptation). 109

6.6 Surface distribution of the pressure over vessel walls obtained on the adapted mesh at peak systole, early diastole and end diastole. Note $1mmHg = 133.3Pa = 133.3N/m^2$. 110

6.7 Surface distribution of the wall shear stress over vessel walls obtained on the adapted mesh at peak systole, early diastole and the end diastole. Note $1dynes/cm^2 = 0.1Pa = 0.1N/m^2$. 111

6.8 Magnified view of surface distribution of the wall shear stress over aorta walls obtained on the adapted mesh at peak systole, early diastole and end diastole. Note $1dynes/cm^2 = 0.1Pa = 0.1N/m^2$. 111

6.9 Velocity magnitude on selected cut-planes obtained on the adapted mesh at four instants. See Figure 6.2 for the location of the six cut-planes. 112

6.10 Pressure and volumetric flow rate computed on the adapted mesh at various outlets in a "whole" body model using RCR boundary conditions. (Note: $1mmHg = 133.3Pa = 133.3N/m^2$ and $1cc/s = 10^{-6}m^3/s$.) 113

6.11 Comparison of temporal variation of pressure and velocity magnitude at the center of brachial artery and descending aorta between initial mesh and the adapted mesh. 114

6.12 Comparison of velocity magnitude along with contours for cut plane B and E on the initial and the adapted meshes at early diastole. 115

6.13 Comparison of wall shear stress on the initial and the adapted meshes at peak systole. 116

ACKNOWLEDGMENT

I would first like to express my sincere gratitude to my adviser, Professor Kenneth E. Jansen, for his support and guidance throughout this work. I would also like to express my appreciation to him for his insightful advice and consistent encouragement. His fine intuition and deep understanding of finite element and computational fluid dynamics have been inexhaustible source of inspiration during my work for this thesis. I thank him for giving me so much trust in the research and helping me build up the capability and the confidence. I still remember that many times, he calmed me down when I was stresses up and helped me out. I have had a great time and been productive in this harmonic and free environment and never been frustrated working in his group. I also appreciate his guidance with full of his heart and consideration. His support for my work, life and my career development are invaluable. It have been my extreme privilege to work with him in the past four years. I am also grateful to him for giving me the chance to utilize the world's leader-ship supercomputers including IBM Blue Gene/P at ANL. I also want to thank him for his great effort of reviewing this thesis to improve its overall quality.

I would like to thank my co-adviser, Professor Mark S. Shephard, for giving me the opportunity to work with him on the ITAPS project. His knowledge and intuition in mesh data base and mesh adaptation have open a door for me to an entirely new world. He gave me invaluable support in the project and helped me out when I encountered difficulties. I really appreciate his patience and guidance. I also thank him for his constant help to build my documentation skills and attitude as a professional researcher. Without his support, my thesis would not be possible. I would also like to express my appreciation to both Professor Shephard and Professor Jansen for leading the SCOREC, which provides computing resources that are incomparable to most of the other places. I was lucky to work in SCOREC and have opportunity of collaboration with various top researchers.

I would like to thank Dr. Onkar Sahni, for many useful discussions and fruitful suggestions over years. I appreciate the side-by-side work with him, which brought

my programing and trouble shooting skills to a new level. I also thank him for being my committee member. I would like to extend my appreciation to the other members of my committee. Professor Charles A. Taylor gave me a great opportunity to work with him and his research group and also provided interesting and challenging problem cases in field of cardiovascular flows. I would like to thank Professor Christopher D. Carothers and Assad A. Oberai for reviewing this thesis with great effort and the useful feedback.

I would like to thank the staffs in SCOREC, Adam Todorski, Tim Wickberg, Marge Verville for their help and support. I would also like to thank my colleagues, Dr. Xiaojuan Luo, Victor Marrero, Ting Xie, Alexander Ovcharenko, Joseph Olles, Farhad Behafarid, Michael Schrameyer, Dr. Alberto Figueroa (in Stanford University) and others for the help and the fruitful discussions. I would like to thank my friends who made this work interesting and worthwhile.

I would like to acknowledge the courtesy of Simmetrix Inc., ACUSIM Software and Kitware Inc. for providing software packages used in this work. I would like to express my acknowledgment to Dr. Karen D. Devine for the help with Zoltan partition package.

Last, but not least, I would like to thank my family members including my parents, my parents in-law, my brother, my sister in-law, my aunts, and my son, but especially my husband Da Ha, for their sweat love and ceaseless encouragement over the years. I would not go this far without their love and support.

ABSTRACT

In this study, we identify and resolve several bottlenecks facing unstructured, adaptive, implicit finite element methods march towards petascale simulations. With those obstacles resolved, our method demonstrates its capabilities with strong scalability on large scale supercomputers and its ability to solve problems of interest requiring numerically intensive computations in a reasonable time frame. The performance of our implicit solver is improved by two algorithms developed in this work. The first algorithm, multiple compute-object based partition improvement, incrementally improves the load balance, hence the scalability of both the equation formation and the equation solution of the finite element analysis (FEA). The second algorithm, data reordering, enables the effective usage of the memory subsystem by increasing the data locality, so as to accelerate the per-core performance of the FEA.

We present excellent strong scaling for several applications performed on various supercomputers including IBM Blue Gene (BG/L and BG/P), Cray (XT3 and XT5) and Sun Constellation Cluster. The applications involve the flow simulations of a bifurcation pipe model with relatively small meshes and cardiovascular flow of an abdominal aorta aneurysm model with a much bigger mesh (more than 1 billion elements). The other application involves the blood flow in a "whole" body model composed of 78 arteries; from the neck to the toes. The effectiveness of our methodologies and the algorithms developed in this work are investigated in those applications. With the ability to solve real-world problems having complex geometry/physics in a realistic time, this work provides a reliable and efficient computation tool that can be used by researchers for design and development purpose.

CHAPTER 1
INTRODUCTION

1.1 Background and motivation

Parallel computing is essential in large-scale computation of complex engineering problems involving fluid mechanics. Advanced algorithms capable of simulating real-world problems efficiently and powerful advanced parallel computing platforms have brought simulation and modeling to a new level. A range of physical applications including ones in fluid mechanics, solid mechanics, electromagnetic, biomechanics, molecular dynamics, to name a few, involve challenging and inherently large problems and require numerically intensive computations. One such example is patient-specific vascular surgical planning which requires the use of many millions of spatial degrees of freedom and many thousands of implicit time steps [69]. Problem of such size can only be solved on massively parallel computers. Another example is ultra-high resolution model of climate and weather. A credible kilometer scale atmospheric model would require at least a sustained ten petaflops computer (a petascale system is likely to contain 100,000+ cores) to provide scientifically useful climate simulations [95]. In the past three decades, both algorithms and hardware have continuously improved, allowing computational fluid dynamics (CFD) to gain wide acceptance. Obstacles do however remain before complete acceptance of CFD as an indispensable tool in certain applications (e.g., aerospace vehicle design and propulsion subsystem analysis) where they are still employed in a very limited fashion due to a prohibitive increase in the grid resolution for certifiable accuracy [15]. Petascale computing can open the door to affordable, routine, error quantification for simulation data. Simulations of increasingly complex phenomena with greater physics fidelity will be allowed by the rapid advances in high-performance computing technology. Over the last few decades, many interfaces have developed on parallel computers that makes the parallel implementation much easier, nevertheless computing at the petascale raises a number of significant challenges for parallel computational codes. A critical aspect for utilizing petascale resources is the scala-

bility of the underlying numerical methods, both strong scaling (i.e., speedup with respect to a fixed-size problem) and weak scaling (i.e., constant time usage for proportionally increased problem size and the number of processors). In general, the scalability achieved in parallel computations depends on the efficiency of the parallel algorithms as well as the hardware architecture of the processing units.

The challenges and the achievements towards petascale computing in a variety of numerical methods and applications have been discussed in previous work. Swaminarayan *et al.* [82] reports on molecular dynamics simulations on the roadrunner heterogeneous supercomputers. In the large-scale gyrokinetic particle simulation of microturbulence in magnetically confined fusion plasmas [23], the Gyrokinetic Toroidal Code (GTC) was developed and optimized on the IBM Blue Gene/L (BG/L) computer, achieving essential linear (weak) scaling on more than 30,000 processors. Oliker *et al.* [52, 56] discussed the performance of several scientific applications on the candidate petascale platforms. Out of the six application packages, two of them (GTC for magnetic fusion and CACTUS for astrophysics) attain impressive scalability characteristics all the way up to 32,768 processors on BG/L. The scaling of other applications (e.g., FLBM3D of Lattice Boltzmann fluid dynamics and PARATEC of material science, etc.) are still limited to a few thousand processors. The overall results indicate the potential of evaluated codes to effectively utilize petascale resources. However, re-engineering to incorporate the additional levels of parallelism is required to achieve the vast concurrency of ultrascale system. Bekas and co-authors [14] reported the extreme scalability challenges in micro-finite element simulations of human bone. A study of artificial as well as human bone specimens was conducted, which resulted in a large sparse systems of up to about 1.5 billion unknowns. The degradation of scalability is identified as the load imbalance due to the graph partitioner at this scale (with thousands of computing cores). More recently, Burstedde *et at.* [29] presented their work of adaptive mantle convection simulation on petascale supercomputers. Parallel adaptive mesh refinement and coarsening (AMR) is essential for their simulation to avoid too many elements when using uniform resolution. They demonstrated weak and strong scalability of parallel AMR on up to 62,464 cores on TACC's Ranger for problems with

up to 12.4 billion elements. Their parallel efficiency remains above 50% in scaling from 1 to 62,464 cores.

Despite the discussions in various application areas, the application area of focus in this thesis is computational fluid dynamics. Tezduyar and Sameh [88] provided a brief overview of the role of parallel computing in finite element computational fluid mechanics problems and addressed some of the computational challenges in 3D finite element computation of complex fluid problems on parallel architectures. Biswas and co-authors [15] discussed the impact of petascale computing on future NASA missions. In their work, three important application areas are considered, including aerospace analysis and design (Cart3D), propulsion subsystem analysis (INS3D) and hurricane prediction (fvGCM), as the representative set of challenges towards petascale computations. Degradation in performance was observed above several thousand processors. The potential benefits of petascale computing to NASA were outlined and the parallel scalability bottlenecks which have to be overcome to achieve its full potential were described. Sunderland *et al.* [81] analyzed the performance of several CFD codes for a range of datasets on some of the high performance computing architectures, but the codes do not scale well beyond 8,192 processors. Behara and Mittal [13] presents an implementation of a stabilized finite element formulation for 3-D unsteady incompressible flows on parallel computation. Their methodology is close to the focus of this thesis, however the study is still limited to O(100) processors.

Practical problems often involve large scale and complicated geometries and/or require time-critical decisions (such as planning of patient-specific surgical procedures based on cardiovascular flow simulations). Such problems place extreme demands on the flow solver. An implicit finite element flow solver, Parallel Hierarchic Adaptive Stabilized Transient Analysis (PHASTA), has demonstrated the ability to meet these performance requirements, by employing a partition of the mesh into a set of parts distributed over the processing cores for both the formulation and the solution of the finite element system [69, 77]. A near-perfect strong scaling is reported on a $O(10^9)$ element mesh using up to 163,840 processor cores [69]. PHASTA applies implicit methods for the PDE using unstructured meshes which allow for an effi-

cient solution strategy for many real-world problems. When the solution of interest is sensitive to small details of the complex geometry and/or if complex anisotropic solution features develop a wide range of length scales that cannot be determined *a priori*, adaptive unstructured meshes offer tremendous reduction in computational effort required for a given level of solution resolution. Furthermore, many such solutions are transient in nature but contain time scales whose resolution may or may not be of interest to the analysis, strongly motivating the consideration of implicit time integrators. Such problems are otherwise prohibitively expensive to consider with contrasting methods using structured grids (mesh grows too large), explicit techniques (time step becomes too small) and/or procedures with limited scalability (time-to-solution takes too long). To meet their full potential, the unstructured and implicit techniques must scale for extremely-large practical problems and also to yield dramatic compression in time-to-solution(s) for a fixed-size problem.

The objective of the current work is: (a) to identify the obstacles of current software utilized, including the pre-processing tools, towards petascale computation and (b) to resolve the issues for the full utilization of the leadership-class computing systems. In this work, we have focused our attention to methodologies for obtaining solutions to large-scale problem in realistic time. An overview of the framework of petascale adaptive computational fluid dynamics is presented in Section 1.2 along with the introduction to the main components in the framework.

1.2 Framework of parallel adaptive simulation and the current status

A parallel adaptive computational fluid dynamics framework evaluates the quality of the computed solution and parameters of interest by *a posteriori* error measure and reduces the discretization error based on these measures iteratively until the solution is acceptable. Figure 1.1 demonstrates the main components of a parallel adaptive flow simulation. An adaptive parallel analysis usually starts with an initial mesh (discretization of a physical domain) through *a priori* specification of mesh size information without precise knowledge of details of the solution. The pre-solver makes use of the geometric model, the attributes associated with

the model, and the mesh database to apply the correct problem description (e.g., initial and boundary conditions). This allows the pre-solver and flow solver to perform computations based on the current mesh which has already been partitioned and balanced for use on parallel computers. An error and/or correction indication scheme [9, 49, 66, 68, 90] based on the computed solution is used to specify the mesh size field [66, 68] which quantifies the desired mesh resolution over the domain. Next, local mesh modification [10, 21, 49, 66] is carried out to satisfy the specified mesh size field and the solution is transfered to the adapted mesh for the next stage of the computation. This process is continued and the mesh is improved iteratively to reduce the discretization error until the error reaches an acceptable level. Each com-

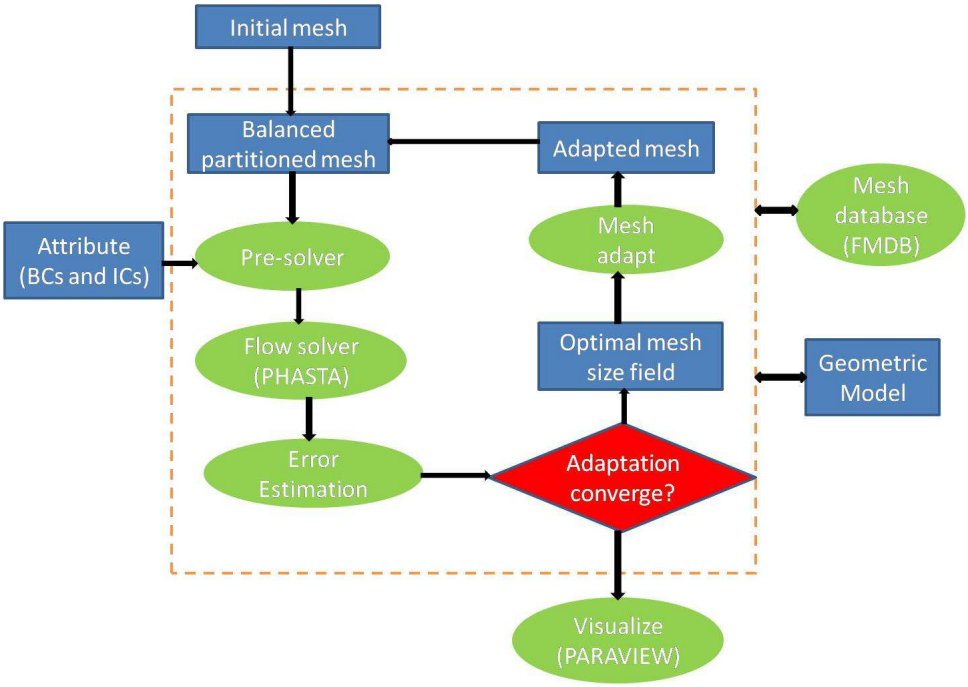


Figure 1.1: Key steps of parallel adaptive flow simulation

ponent in Figure 1.1 is indispensable and bottlenecks in any of them prevent moving forward to petascale simulations. The software available prior to this research was mature in serial as well as in parallel but with modest number of processing cores, however obstacles were encountered in extension to computations with O(100,000) processing cores. The goal of this work is to address those deficiencies and to resolve the problems to enable massively parallel computations. The second objective is to

improve the software performance, especially in high processing core count to utilize the supercomputing resources more effectively.

In the realm of computational/numerical methods to solve partial differential equations, a physical problem is transformed into a set of mathematical equations by providing boundary conditions and initial conditions along with constitutive properties and geometric models. The pre-solver software in use, *phParAdapt*, takes the discretization of the domain (mesh), geometric model and the attribute which supplies additional information (e.g., boundary conditions and material properties) and output a set of files which are ready to be used by finite element analysis tools (e.g., PHASTA in the current study). The process of *phParAdapt* is aided by databases and functions that can answer basic queries about the topological adjacencies of a given mesh and its association with geometric model and inter-part boundaries. *phParAdapt* is compatible with the *Simmetrix* [3] commercial mesh database, that is not available on all petascale class supercomputers. Therefore, one additional contribution of this work is to support the compatibility of *phParAdapt* with an open source data base, the Flexible distributed Mesh DataBase (FMDB), which is a distributed mesh data management system capable of shaping its data structure dynamically based on the user-requested mesh representation [72] (see Section 3.1 for a brief introduction of FMDB). The geometric models (representing the physical domain) is constructed using solid model kernels [1, 2, 4] that support boundary representations including topological data and shape information. The boundary conditions, initial conditions and the constitutive properties are specified through an attribute system. The importance of the open source code is discussed in Section 3.3, along with the introduction of an open source geometric modeler (*meshmodel*) and the design of an attribute system.

It is important to note that one distinguishing feature of adaptive simulations is that the resolution and/or its distribution in the problem is/are expected to have dramatic changes. Typically, initial simulations only capture the coarsest scales of the problem which then provide solutions and associated error fields that drive the adaptivity to resolve better the smaller scales. This dramatic change in the problem size makes solving the problem from start to finish on the same number of processors

impractical. To enable adaptive simulations to efficiently change the size of their partition, requires *phParAdapt* to account for multiple parts on each processor. The details of this development are provided in Section 3.5.

The finite element analysis (FEA) software is another key component in the parallel flow simulation framework. The FEA software utilized, PHASTA, was the first massively parallel unstructured grid LES/DNS code [35, 36, 37] and has been applied to flows ranging from validation benchmarks to cases of practical interest. The practical cases of interest not only involve complicated geometries (such as human arterial system) but also complex physics (such as fluid turbulence). PHASTA has been shown to be an effective tool using implicit techniques for bridging a broad range of time and length scales in various flows with strong scalability over various large-scale systems including IBM Blue Gene systems, Cray XT systems and Sun Constellation Cluster [69, 77]. It has also effectively applied recent anisotropic adaptive algorithms [49, 67, 68] along with advanced numerical methods. The performance improvement algorithms developed in this work are focused on enabling PHASTA to achieve high scalability by addressing several scaling issues. To improve the partition (domain decomposition), two multiple compute-object based partitioning improvement (MCOPI) algorithms have been developed to decrease the imbalance of degrees of freedom while maintaining the good balance in regions (see Section 4). Furthermore, an adjacency-based data reordering algorithm [99] for acceleration of finite element computations is devised to improve the per-core performance (see Chapter 5).

Since there are no reliable a *a priori* methods to control the approximation errors, adaptive methods must be applied where the mesh resolution is determined in a local fashion based on the spatial distribution of the solution and errors associated with its numerical approximation. Moreover, the reliability and accuracy of simulations is also a strong function of the mesh quality and configuration. The function of mesh adaptation (modification) is to convert a given mesh into the desired mesh consistent with the mesh improvement information. A mesh modification based procedure has been developed that accepts a mesh size field as input and applies mesh refinement, coarsening and swapping operations on the mesh to match

the specified mesh size [76, 94], including anisotropic meshes [42, 43, 44] while preserving boundary layers [67, 68] (see Figure 1.2). These tools have been developed and are being extended by others within our research group but are being applied at unprecedented scale in this thesis. The mesh modification procedures operate

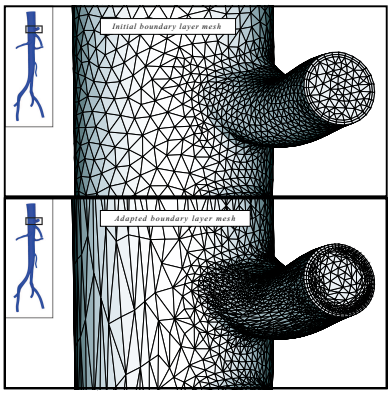


Figure 1.2: Anisotropic adaptivity, preserving boundary layer mesh structure in an artery model [66].

in parallel on distributed mesh using the ability to migrate group of mesh entities between different processors to allow the individual mesh modifications to be performed locally on any processor [21]. A predictive load balancing algorithm is developed and added to the parallel mesh adaptation tool to balance the number of mesh entities in the adapted mesh (details in Section 3.4) enabling a mesh with over 1 billion regions to be created.

To summarize, efficient adaptive simulations require that each single component in the framework is powerful, whereas, a bottleneck in any of the components will break the loop and prevent moving forward to petascale simulation.

1.3 Outline of thesis

The organization of this dissertation is as follow. Chapter 2 gives an introduction to the numerical methods used in this work. Chapter 3 focuses on the addressing bottlenecks in the pre-processing of the finite element solver and the methodologies to resolve the problems. In this chapter, the importance of open source software in the petascale computation is discussed along with the implementation of an open source attribute system. The need for multiple parts per processor as well as its

implementation in the pre-solver is also presented in this chapter. The last section in Chapter 2 describes a predictive load balance algorithm. Chapter 4 and 5 demonstrate the flow solver performance improvement. Two distinct algorithms (partition improvement and data reordering) are developed in the current work, both of which accelerate the FEA. In Chapter 4, two multiple compute-object based partitioning improvement algorithms are introduced to improve load balance and scalability of both the equation formation and the equation solution by incremental redistribution of mesh entities. Chapter 5 presents an adjacency-based data reordering algorithm which improves the overall cache utilization so as to accelerate the on-processor calculations. Chapter 6 demonstrates the results obtained, based on the contributions of this work, by applying it to problem cases in cardiovascular flows. Chapter 7 concludes the study by summarizing the results obtained, which clearly illustrate the effectiveness of the current work, and also discusses the future work.

CHAPTER 2
NUMERICAL METHODS

The finite element method is a standard analysis tool for solving complex sets of partial differential equations (PDEs) over general domains. In a large number of cases, due to numerically intensive computations, the ability to solve problems of practical interest requires the use of massively parallel computers so long as the formation and solution of the finite element systems are computationally efficient and have a high degree of scalability, particularly in situations with tight constraints on time to solution. An implicit finite element flow solver, PHASTA, has demonstrated the ability to meet these performance requirements. PHASTA employs a partition of the mesh into a set of parts distributed over the processing cores for both the formation and implicit solution of the finite element system [69, 77]. PHASTA has been shown [38, 96] to be an effective tool for a broad range of time and length scales in various flows including turbulent ones (based on URANSS, DES, LES, DNS). It has also been effectively applied with anisotropic adaptive algorithms [67, 68] and advanced numerical models of flow physics [34, 85, 86].

The computational work involved in implicit methods (e.g., finite element or volume methods) for partial differential equations mainly consists of two components: (a) formation/assembly of the linearized algebraic system of equations and (b) computation of the solution to the linear system of equations. In the first component, entity-level evaluations over the mesh (e.g., for finite elements, element-wise integration based on numerical quadrature) are performed to form the system of equations, $\mathbf{Ax} = \mathbf{b}$. The resulting system is sparse but involves a large number of unknowns. The second work component involves computation of a solution to the formed system of equations. Here, pre-conditioned iterative procedures are suitable for large, sparse systems (e.g., GMRES [65, 73]).

In the PHASTA solver, the Navier-Stokes equations (conservation of mass, momentum and energy) plus any auxiliary equations (as needed for turbulence models or level sets in two-phase flow) are discretized in space and time. Discretization

in space is carried out with a stabilized finite element method. This step leads to a weak form of the governing equations, where the solution (and weight function) are first interpolated using hierarchic, piecewise polynomials [38, 97], and followed by the computation of integrals appearing in the weak form using Gauss quadrature. Implicit integration in time is then performed using a generalized-α method [38]. The resulting non-linear algebraic equations are linearized to yield a system of equations which are solved using iterative procedures, e.g., GMRES is applied to the linear system of equations $\mathbf{Ax} = \mathbf{b}$ (where, $\mathbf{b}$ is the right-hand-side or residual-vector of the weak form and $\mathbf{A}$ is the left-hand-side or linearized tangent-matrix of $\mathbf{b}$ with respect to unknown solution coefficients $\mathbf{x}$). After a detailed description of the physics model and mathematical formulations used in PHASTA are given in Section 2.1, we focus our attention on the details of the parallelization paradigm and associated procedures that enable our implicit, unstructured flow-solver to achieve strong scalability.

2.1 Governing equations and flow solver

The governing equations for blood flow, assuming Newtonian constitutive behavior and rigid blood vessel walls, are the transient incompressible Navier-Stokes equations:

$$u_{i,i} = 0 \tag{2.1}$$

$$\rho \dot{u}_i + \rho u_j u_{i,j} = -p_{,i} + \tau_{ij,j} + f_i \tag{2.2}$$

where u_i is the ith component of velocity, p the pressure, f_i the prescribed body force or source term, such as gravity, which is typically neglected in arterial flow analysis, and τ_{ij} the viscous stress tensor given by:

$$\tau_{ij} = \mu(u_{i,j} + u_{j,i}) \tag{2.3}$$

where μ is the dynamic viscosity.

The above system of equations is supplemented with an appropriate set of boundary conditions that are prescribed on the model boundary of the blood vessels.

No-slip boundary conditions are imposed on the rigid and impermeable vessel walls. A time varying velocity profile, based on the physiological values, is prescribed at the inlet. And a lumped parameter boundary condition [91] is applied at the outlets.

Finite element methods are based on the weak form of the governing equations (Eq. 2.1 and 2.2) which is obtained by taking the $L^2(\Omega)$-inner product of the entire system with weight functions. Integration by parts is then performed to move the spatial derivatives off some terms and onto the corresponding weight functions. Specifically, the diffusive term, pressure term and continuity equation are all integrated by parts to reduce continuity requirements, since otherwise we could have second derivatives on our solution space. The pressure term is integrated by parts to provide symmetry with the continuity equation which in turn is integrated by parts to provide discrete conservation of mass. This process leads to an integral equation often referred to as the weak form.

To derive the finite element discretization from the strong form of the governing equation, discrete weight and solution function spaces must be introduced. Let $\bar{\Omega} \subset \mathbf{R}^N$ represents the closure of the physical spatial domain, $\Omega \cup \Gamma$, in N dimensions; only $N = 3$ is considered. The boundary is decomposed into portions with natural boundary conditions, Γ_h, and essential boundary conditions, Γ_g, i.e., $\Gamma = \Gamma_g \cup \Gamma_h$. In addition, $H^1(\Omega)$ represents the usual Sobolev space of functions with square-integrable values and derivatives on Ω. Subsequently Ω is discretized into n_{el} finite elements, $\bar{\Omega}_e$, as defined above. With this, we can define the discrete trial solution and weight spaces for the semi-discrete formulation as

$$\mathcal{S}_h^k = \{\mathbf{u} | \mathbf{u}(\cdot, t) \in H^1(\Omega)^N, t \in [0, T], \mathbf{u}|_{x \in \bar{\Omega}_e} \in P_k(\bar{\Omega}_e)^N, \mathbf{u}(\cdot, t) = \hat{\mathbf{g}} \text{ on } \Gamma_g\}, \quad (2.4)$$

$$\mathcal{W}_h^k = \{\mathbf{w} | \mathbf{w}(\cdot, t) \in H^1(\Omega)^N, t \in [0, T], \mathbf{w}|_{x \in \bar{\Omega}_e} \in P_k(\bar{\Omega}_e)^N, \mathbf{w}(\cdot, t) = \mathbf{0} \text{ on } \Gamma_g\}, \quad (2.5)$$

$$\mathcal{P}_h^k = \{p | p(\cdot, t) \in H^1(\Omega), t \in [0, T], p|_{x \in \bar{\Omega}_e} \in P_k(\bar{\Omega}_e)\} \quad (2.6)$$

where $P_k(\bar{\Omega}_e)$ is the space for all polynomials defined on $\bar{\Omega}_e$, complete up to order $k \geq 1$. Let us emphasize that the local approximation space, $P_k(\bar{\Omega}_e)$, is the same for both the velocity and pressure variables (although this is not necessary, it is

computationally convenient, especially when working with higher-order discretizations). This equal-order interpolation is possible due to the stabilized nature of the formulation to be introduced below. These spaces represent discrete subspaces of the space in which the weak form is defined.

The stabilized formulation used in the present work is based on that described by Taylor *et al.* [84] generalized to include the higher-order basis functions. Given the spaces defined above, we first present the semi-discrete Galerkin finite element formulation applied to the weak form of (2.1, 2.2) as:

Find $\mathbf{u} \in \mathcal{S}_h^k$ and $p \in \mathcal{P}_h^k$ such that

$$\mathcal{B}_G(w_i, q; u_i, p) = 0$$

$$\begin{aligned}\mathcal{B}_G(w_i, q; u_i, p) = &\int_\Omega \{w_i \left(\rho \dot{u}_i + \rho u_j u_{i,j} - f_i\right) + w_{i,j} \left(-p\delta_{ij} + \tau_{ij}\right) - q_{,i} u_i\} dx \\ &+ \int_{\Gamma_h} \left\{w_i \left(p\delta_{in} - \tau_{in}\right) + q u_n\right\} ds\end{aligned} \tag{2.7}$$

for all $\mathbf{w} \in \mathcal{W}_h^k$ and $q \in \mathcal{P}_h^k$. The boundary integral term arises from the integration by parts and is only carried out over the portion of the domain with natural boundary conditions. Since all the weight coefficients are arbitrary, this gives us a separate equation for each of the i components (and for each of the basis functions). The standard Galerkin method is well known to be unstable for equal-order interpolation of the velocity and pressure, additional stabilization terms are introduced as follows:

Find $\mathbf{u} \in \mathcal{S}_h^k$ and $p \in \mathcal{P}_h^k$ such that

$$\mathcal{B}(w_i, q; u_i, p) = 0$$

$$\begin{aligned}\mathcal{B}(w_i, q; u_i, p) = &\mathcal{B}_G(w_i, q; u_i, p) \\ &+ \sum_{e=1}^{n_{el}} \int_{\bar{\Omega}_e} \left\{\tau_M (u_j w_{i,j} + q_{,i}) \mathcal{L}_i + \tau_C w_{i,i} u_{j,j}\right\} dx \\ &+ \sum_{e=1}^{n_{el}} \int_{\bar{\Omega}_e} \left\{w_i \rho \overset{\Delta}{u}_j u_{i,j} + \bar{\tau} \mathcal{L}_j w_{i,j} \mathcal{L}_k u_{i,k}\right\} dx\end{aligned} \tag{2.8}$$

for all $\mathbf{w} \in \mathcal{W}_h^k$ and $q \in \mathcal{P}_h^k$. We have used $\mathcal{L}_i$ to represent the residual of the ith momentum equation,

$$\mathcal{L}_i = \rho \dot{u}_i + \rho u_j u_{i,j} + p_{,i} - \tau_{ij,j} - f_i \tag{2.9}$$

The second line in the stabilized formulation, (2.8), represents the typical SUPG stabilization added to the Galerkin formulation for the incompressible set of equations (see Franca and Frey [27]). The first term in the third line of (2.8) was introduced by Taylor *et al.* [84] to overcome the lack of momentum conservation introduced as a consequence of the momentum stabilization in the continuity equation. If we define

$$\overset{\Delta}{u}_i = \frac{-\tau_M \mathcal{L}_i}{\rho}, \tag{2.10}$$

then $\overset{\Delta}{u}_i$ may be identified with a modified, conservation-restoring, advective velocity. This term must itself be stabilized since it is an advective type term which will lead to advective instabilities (second term on the last line of(2.8)). The stabilization parameters for continuity and momentum are defined as

$$\tau_M = \frac{1}{\sqrt{2c_1/{\Delta_t}^2 + u_i g_{ij} u_j + c_2 \nu^2 g_{ij} g_{ij}}} \tag{2.11}$$

$$\tau_C = \frac{\rho}{8\tau_M \mathrm{tr}(g_{ij})} \tag{2.12}$$

and the stabilization of the new advective term is defined in direct analogy with the advective portion of τ_M as

$$\bar{\tau} = \frac{\tau_M}{\sqrt{\mathcal{L}_i g_{ij} \mathcal{L}_j}} \tag{2.13}$$

where c_1 and c_2 are defined based on considerations of the one-dimensional, linear advection-diffusion equation using a linear finite element basis and $g_{ij} = \xi_{k,i}\xi_{k,j}$ is the covariant metric tensor related to the mapping from global to element coordinates. It should be noted that for tetrahedral elements, this mapping depends on the orientation of the element, and therefore must be corrected to create an invariant element length-scale by permuting the possible choices of orientation. This term

may be identified with the element length-scale, and is hence a mesh dependent parameter.

To summarize, we use the SUPG stabilized formulation for the transient incompressible Navier-Stokes equations, governing blood flow, that are discretized by linear finite elements, both for the pressure and the velocity field. Now, to develop a discrete system of algebraic equations, the weight functions w_i and q, the solution variables u_i and p, and their time derivatives are expanded in terms of the finite element basis functions. Gauss quadrature of the spatial integrals results in a system of first-order, nonlinear differential-algebraic equations. Finally this system of non-linear ordinary differential-algebraic equations is discretized in time via a generalized-α time integrator (see Jansen et al. [38]) resulting in a non-linear system of algebraic equations. This system is in turn linearized with Newton's method which yields a linear algebraic system of equations that is solved (at each time step) and the solution is updated for each of the Newton iterations. The linear algebra solver of Shakib [74] is used to solve the linear system of equations.

2.2 Parallel paradigm

The aim of the current parallelization procedures is to achieve scaling in mesh-based computations with a massive level of parallelism involving hundreds of thousands of cores. All computations are based on a decomposition or partition of the mesh into parts with equal work load. The term part is used to denote a set of mesh entities and partition is used to indicate collection of all parts (i.e., together all the parts within a partition comprise the aggregate mesh). For a mesh with fixed element topology and order, balanced parts within a partition implies that each part contains as close to the average number of mesh entities as possible (see Figure 2.1). For other cases such as ones with mixed element topology or order, weights reflecting the work load for every individual element are assigned to create parts with a balanced load.

Graph and hypergraph based partitioning schemes (e.g., ParMetis[41], PHG[16], PT-SCOTCH [58]) are currently used as they are suitable for unstructured mesh calculations that possess irregular structures and connections. Such partitioning

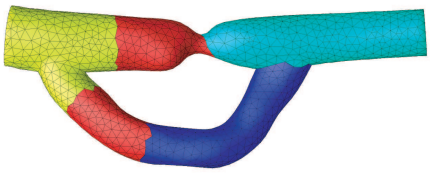


Figure 2.1: Partition of a coarse unstructured mesh of arterial bypass (colors indicate different parts).

schemes not only balance the work load but also minimize the amount of communication required (or shared mesh entities) between parts. One important point to consider during partitioning is that the computational load (in any part) during the system formation stage (i.e., during formation of **A** and **b**) depends on the number and type of the elements present in the part whereas in the system solution stage it depends on the number of degrees-of-freedom (*dofs*), or unknowns in the system of equations on that part. For example, in the case of linear finite elements of all the same topology, work involved in equation formation is proportional to the number of mesh elements in the part while during equation solution the work is proportional to the number of mesh vertices in the part since the unknowns are associated with the vertices. Element-based partitioning is currently used as it is natural for element-integration/equation-formation stage making it highly scalable. So long as the *dof* balance is also preserved, this partitioning also maintains the scalability of the iterative linear solve. Typically, element balance (with sufficient load per part) and minimization of amount of communications during partitioning results in a reasonable *dof* balance as well. In element-based partitioning, each element is uniquely assigned to a single part but in turn leads to shared *dofs* or unknowns in the system of equations. Here, a shared *dof* (or unknown) is defined as one that appears on an inter-part boundary, see solid dots in Figure 2.2, whereas a non-shared *dof* resides solely on one part (e.g., interior to a part). Therefore, the communication effort related to shared *dofs* is peer-to-peer, i.e., required only among connected neighboring parts, and grows with the number of *dofs* on the surface/boundary that is shared in segments with one or more neighboring parts. Note that in situations where the number of mesh regions per part is relatively small (in the order of few thousand),

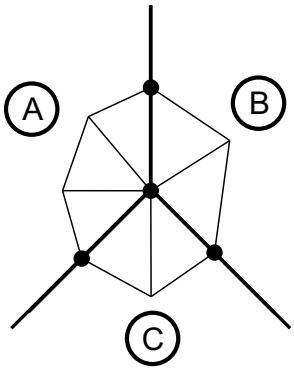


Figure 2.2: Solid dots indicate shared *dofs*.

significant imbalance in *dofs* can result, since the balance of *dofs* is not explicitly requested. Furthermore, the percentage of shared *dofs* increases in situations where a fixed-size problem is spread over more and more parts as is the case during strong scaling studies, eventually becoming detrimental to scaling. The MCOPI algorithms introduced in Chapter 4 addresses this issue by improving an existing partitioning through incremental redistribution of mesh entities.

We now introduce a concept of the *partition-graph* that is currently used to describe the parts and their interactions within a partition. Each partition-graph vertex represents a part whereas each partition-graph edge represents interaction between a pair of parts sharing *dofs*. This is done in a distributed way such that a part contains information only in terms of its portion of the work (or sub-mesh) along with its interaction with neighboring parts. The interaction between neighboring parts is defined based on shared *dofs* , as shown in Figure 2.2, where every shared *dof* resides as an image on each part sharing it. Only one image among all images of a shared *dof* is assigned to be the *owner* thereby making all other images explicitly declared to be non-owners, see Figure 2.3. This process insures that the sum total of *dofs* based on owner images over all the parts within a partition is independent of the partitioning and is equal to the number of (unique) *dofs* in the aggregate mesh. Such a control relationship among images of shared *dofs* allows owner image of each shared *dof* to be "in-charge" for data accumulation and update and in turn limits communications to exist only between owner and non-owners (i.e., non-owner images do not communicate to each other). Furthermore, data exchange is done

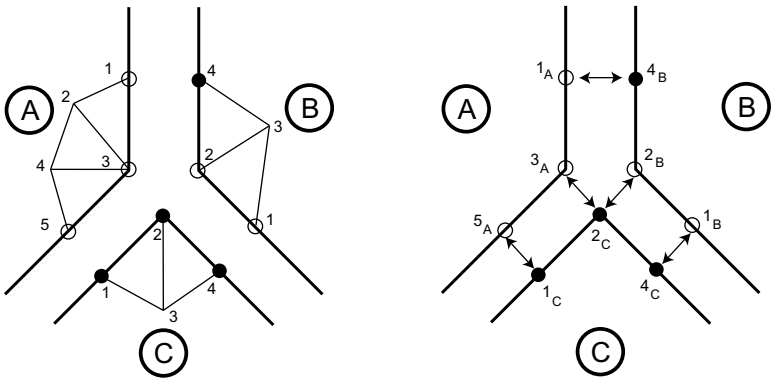
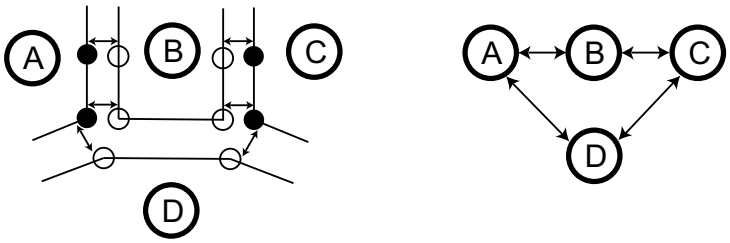


Figure 2.3: Control relationships are established between multiple images of shared *dofs* (*dofs* are locally numbered on each part). Solid dot denotes an owner image whereas hollow ones indicate non-owners.

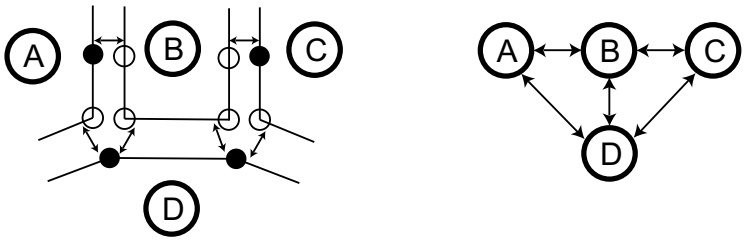
only for vector entries (e.g., in **b**) as this is sufficient to advance the computations (details are provided in the next section). Thus, a partition-graph edge connects only those pair of parts that involves owner image(s), as depicted in Figure 2.4. Typically for the three-dimensional unstructured meshes each part contains on the order of 40 partition-graph edges connecting it with other neighboring parts.

In summary, the following attributes are possessed by the current approach to attain scalability to over one hundred thousand cores (details of associate procedures of the current parallel paradigm are provided in the next section):

- One partitioning is maintained among all compute stages of the solver including equation-formation and equation-solution and thus, no re-distribution of data is required in between.
- The critical step of balancing computations and communications is followed throughout the analysis phase with the help of a distributed partition-graph that describes interaction of every part with its neighbors. Communications are dominated by point-to-point operations that are bounded to neighboring parts only whereas collective operations are required to compute norms during iterative solves within the linear algebra routines.
- There is no information required in a global view such as global numbering of *dofs* or equation number; as each part handles elements (and *dofs*) allocated to it in a local manner and exchanges data associated with shared *dofs* through



(a) Parts B and D do not interact (as no owner image is involved).



(b) All pairs of parts sharing *dofs* interact (as at least an owner image is involved).

Figure 2.4: Schematic of partition-graph (arrows indicate interaction between two parts).

communication structures defined in terms of local images.

- Control relationships are established between copies/images of each shared *dof* residing on multiple parts (i.e., two or more parts). Such a relationship allows owner image of each shared *dof* to be in-charge for data accumulation and update and thus, limits communications to exist only between owner and non-owners (non-owners do not communicate to each other). Furthermore, exchange of data is done at a part-level (not for each and every pair of owner and non-owner) and only for vector entries (e.g., in $\mathbf{b}$). This leads to effective and organized communications.

All the steps in the numerical analysis phase operate from a decomposition of the mesh. Typically, with one part per processor core (or process), each core executes a copy of the analysis code to handle the computational work and interactions corresponding to its part. Collectively, all the cores have the same information as in the serial or one-part case but no one core holds or has knowledge of the

entire tangent matrix, **A**, nor the residual vector **b**. Thus, to be able to progress the computations in parallel, interactions between shared *dofs* are completed via communications.

2.3 Notion of a complete value

We will consider a vector-entry (or matrix-entry) to be complete when it has the same value in a partitioned (or parallel) case as it would have in an one-part case. For instance, after numerical integration on a local part, values in rows of **b** for shared *dofs* are individually incomplete on each part (referred to as *on-part value*) because their contributions are distributed among their images (in finite element methods this is due to the compact support of basis or shape functions used). On the other hand, values for non-shared *dofs* are complete. Thus, an inter-part assembly process is required to obtain complete values related to shared *dofs*. Under current procedures, complete values are obtained only for vector entries (e.g., in **b**) as it is sufficient to advance the computations. To obtain a complete value for a vector entry associated with shared *dof*, an inter-part assembly process is defined as: $\sum_{p_j \in \mathcal{R}_i} \tilde{v}_{i_{p_j}}$ (where, i indicates a vector entry associated with a particular shared *dof* , $\mathcal{R}_i$ is the set of residence parts of that shared *dof*, i.e., parts where its images exits, and $\tilde{v}_{i_{p_j}}$ is used to denote contribution from an image on part p_j). This operation is carried out via inter-processor communications unlike intra-part assembly operation where information is available locally. For each shared *dof* only an image, that we defined as owner image, takes charge of this accumulation operation. Thus, $v_{i_{p_{owner}}}$ (where, $p_{owner} \in \mathcal{R}_i$ is the owner part) is obtained as a complete value and then it is used to update all the non-owner images (on other parts $\in \mathcal{R}_i - p_{owner}$). For example, based on the case depicted in Figure 2.3; $v_{4_B} = \tilde{v}_{1_A} + \tilde{v}_{4_B}$ or $v_{2_C} = \tilde{v}_{3_A} + \tilde{v}_{2_B} + \tilde{v}_{2_C}$. In practice, this process is carried out at a part-level and not for each and every shared *dof* individually. This is achieved through communication stage where pre-processed communication traces and tasks defined between pairs of parts are used.

2.4 Communication steps and structures

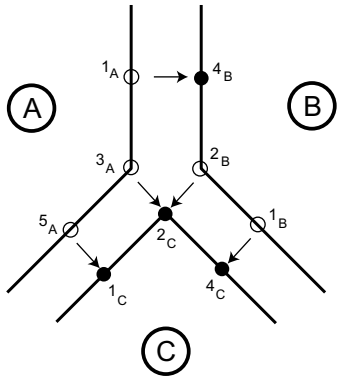
To understand the control of communications, we first define a communication stage:

Definition 2.4.1 Communication stage*: is a step where all cores make the necessary point-to-point communications to obtain complete values. There are two types of communication stage. Type 1 where data is accumulated at owners to obtain complete values, i.e., owners receive from their non-owner images. Type 2 where complete values are copied from owners to update their non-owner images. These two types of communications allow us to obtain complete values on all images of shared dofs. Figure 2.5 depicts the two communication stages.*

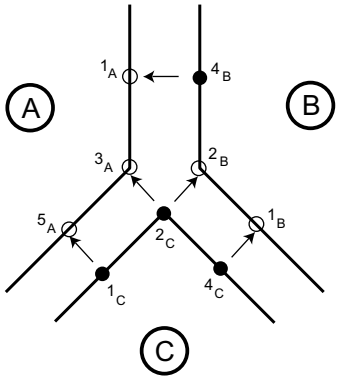
In a communication stage, two cores communicate with each other if they contain interacting pair of parts or are connected through a partition-graph edge (as shown in Figure 2.4). Therefore, from the perspective of a single processor core, say p_i, a communication stage can be described as a sequence of communication tasks, where each communication task is defined as:

Definition 2.4.2 Communication task*: is a point-to-point operation (either send or receive) that is performed on a given processor core, p_i, to communicate with another one, p_k. On a given p_i, it is denoted as $T_{p_k}^{p_i}$ such that $p_k \in \mathcal{I}(p_i)$, where $\mathcal{I}(p_i)$ denotes the set of parts that p_i must interact/communicate with (determined based on partition-graph edges). Note that a communication task will include owner images on one side (referred to as the owner side) and non-owner images on the other side (referred to as non-owner side) (i.e., in $T_{p_k}^{p_i}$ either p_i or p_j is the owner side and the other is the non-owner side).*

As defined earlier, during a type 1 communication stage the owner side of a communication task receives and non-owner side sends as shown in Figure 2.5(a) (where arrows indicate the direction of data flow). On the other hand, in a type 2 communication stage owner side of a communication task sends and non-owner side receives (see Figure 2.5(b)). Therefore, in any communication task the role of point-to-point operation is reversed (i.e., either send or receive) from one type of



(a) Type 1 : Non-owners send while owners receive and accumulate.



(b) Type 2 : Owners send whereas non-owners receive and update.

Figure 2.5: Communication steps involved with shared *dofs*.

communication stage to another. This also tells us that for a communication task $T_{p_k}^{p_i}$ defined on p_i, there exists a matching and opposite communication task $T_{p_i}^{p_k}$ on p_k. For instance, T_B^A and T_A^B in Figure 2.5 form the matching and opposite communication tasks, where A is the non-owner side and B is the owner side (similarly, pair T_C^A and T_A^C, or pair T_C^B and T_B^C). The two matching and opposite communication tasks form a send and receive pair.

One more important point to notice is that a communication task deals with the whole segment of *dofs* shared between the owner and non-owner sides. Instead of performing point-to-point operations between each and every pair of owner and

non-owner, a communication trace is defined:

Definition 2.4.3 Communication trace*: is the segment of shared dofs involved in a communication task. A communication trace is defined in a matching order from both the owner and non-owner sides.*

For example, in Figure 2.5 a communication trace for communication task T_A^C is defined as $\{1_C, 2_C\}$ whereas for the matching and opposite communication task T_C^A it is defined as $\{5_A, 3_A\}$ (i.e., 1_C is owner image for 5_A and similarly 2_C is owner image for 3_A). In practice, the communication traces and tasks are pre-computed and stored for each part while the partition of the mesh remains fixed during the analysis phase. This leads to effective and organized point-to-point communications (where, non-blocking calls are used). The pre-processing (in *phParAdapt*) of communication traces and tasks is made possible by an underlying distributed mesh database library (for example, see [72]).

2.5 Parallel implicit solve

In a transient, non-linear problem, multiple linearizations are performed for each time step to obtain a converged non-linear solution at that step before advancing to the next time step. Each linearization includes two primary work components of equation-formation and equation-solution. To allow the advancement of computations in parallel, we employ the communication steps and structures defined above.

Each processor core first performs interpolation and numerical integration over the (interior and boundary) elements on its local part to form the linearized equations, i.e., the tangent matrix ($\mathbf{A}$) and the residual vector ($\mathbf{b}$). At this point, the entries associated to shared *dofs* in $\mathbf{A}$ and $\mathbf{b}$ are incomplete and contain the contributions from elements local to the part. As a next step two communication stages (one each of type 1 and type 2) are carried out to obtain complete values (only) in the residual vector ($\mathbf{b}$) on each core. Although one could elect to communicate the on-part entries of matrix ($\mathbf{A}$) to make them complete, our approach does not; limiting communications to vector entries only (such as in $\mathbf{b}$). The following step in an implicit solve involves finding the solution update vector ($\mathbf{x}$).

Krylov iterative solution techniques are currently used to find $\mathbf{x}$ (e.g., GMRES [65, 73]). These techniques employ repeated products of $\mathbf{A}$ with a series of vectors (say, $\mathbf{p}$) to construct orthonormal basis vectors to approximate $\mathbf{x}$. In this series, the outcome of any matrix-vector product is another vector $\mathbf{q}$ (=$\mathbf{Ap}$) which is used to derive the subsequent vector in the series while the first vector in the series is derived from the residual vector ($\mathbf{b}$) which contains complete values at this point of the solve. Even though $\mathbf{A}$ contains only on-part values for shared *dofs*, it is still possible to perform the basic kernel of (sparse) matrix-vector product, i.e., $\mathbf{q} = \mathbf{Ap}$, provided vector $\mathbf{p}$ contains complete values (as is the case for the first vector derived from vector $\mathbf{b}$). Note that due to the distributive property of $\mathbf{Ap}$ product, the resulting vector $\mathbf{q}$ will contain on-part (incomplete) values. In other words, values from local $\mathbf{Ap}$ product can be assembled to obtain complete values in $\mathbf{q}$, i.e., $x_{p_i}z + x_{p_j}z = (x_{p_i} + x_{p_j})z$ (where, local products $x_{p_i}z$ and $x_{p_j}z$ results in on-part values because x_{p_i} and x_{p_j} are considered as on-part values in $\mathbf{A}$ associated with two images of a shared *dof* and z is a complete value in vector $\mathbf{p}$; such a logic can be applied similarly to cases with more than two images of a shared *dof*). Therefore, after the local $\mathbf{Ap}$ product we can apply our two-pass communication strategy to obtain complete values in $\mathbf{q}$ (provided $\mathbf{p}$ contained complete values). Before proceeding to the next product in the series, it is important to note that computation of norms is required to perform orthonormalization. In this step, the norm of vector $\mathbf{q}$, and its dot-product with the previous vectors in the series, are computed. To compute a norm or dot-product, first a local dot-product is computed (requiring no communication) but then, to obtain a complete dot-product, a sum across all cores is needed. A collective communication (of allreduce type) is used to carry out the global summation. It is important to point out that while computing a local dot-product value, only the owner image of each shared *dof* takes active part to correctly account for its contribution in the complete (or global) dot-product. Successive $\mathbf{Ap}$ products are carried out along with communications to obtain complete values and to carry out orthonormalization. This leads to an orthonormal basis of vectors which is used to find an approximate update vector $\mathbf{x}$ and marks the end of a non-linear iteration step. Algorithm 1 presents the primary steps of a parallel implicit solve.

Algorithm 1 Pseudocode for a parallel implicit solve ([a] indicates steps with local computations (only), [b] denotes steps involving point-to-point communications and [c] requires collective communication)

CC: SYSTEM FORMATION
[a]Perform entity-level evaluations to form on-part **A** and **b**

CC: EQUATION SOLUTION
[b]Apply two-pass communications to obtain complete values (only) in **b**
[c]Normalize **b** into **p** (used as first vector in the series of **Ap** products)
while (convergence or max. iteration not reached) **do**
  [a]Perform local **Ap** product to obtain **q** with on-part values (**p** is the previous vector in the series)
  [b]Apply two-pass communications to obtain complete values in **q**
  [c]Orthonormalize **q** with respect to previous vectors in the series
end while
[b]Approximate **x** based on the orthonormal basis of vectors (computed above) and apply the update

CHAPTER 3
PRE-SOLVER AND MESH ADAPTATION

Chapter 1 reviewed the status of the software prior to this research and also discussed the existing obstacles to overcome to allow effective computations on massively parallel computing systems. In this chapter, attention is focused on addressing and resolving the issues in the pre-processing/adaptivity-update phase, including features in the mesh adaptation/modification tool, *MeshAdapt* [42, 43, 44], and PHASTA pre-solver software, *phParAdapt*.

Open source software is one of the key components needed to enable us to take advantage of the latest supercomputers, since commercial software currently in use is not presently supported on all the platforms. After a brief introduction to the mesh data base (FMDB) utilized in this thesis (Section 3.1) and a description of the structure of *phParAdapt* (Section 3.2), a discussion of the significance of open source software and an implementation of an attribute system is given in Section 3.3. Additionally, since adaptivity often causes a dramatic increase in the number of elements in one region (or decrease in the elements in another region), it is expected that it will result in an unbalanced mesh. Sometimes, this imbalance can be so severe that the processors exhaust available memory. In such situations, a predictive load balancing algorithm is required with the objective of achieving close to an equal number of mesh entities on each part after adapting mesh, which is introduced in Section 3.4. Finally, to efficiently apply these tools to adaptive simulations where the number of regions (and therefore the number of parts) may grow by three orders of magnitude, *phParAdapt* must be able to handle multiple parts per processor. In Section 3.5, the extension of *phParAdapt* to incorporate multiple parts on each processor is presented.

3.1 Introduction to FMDB

The process of preprocessing and mesh modification are greatly aided by a database and functions that can answer basic queries about the topological ad-

jacencies of a given mesh and its association with the geometric model and the interior partition boundaries. In the studies described herein, we have made use of the Flexible distributed Mesh DataBase (FMDB) which is a distributed mesh data management system that is capable of shaping its data structures dynamically based on the user-requested mesh representation [72]. Before describing the algorithms it is useful to introduce some nomenclature.

Nomenclature

M	an abstract model of the mesh.
$\{M\{M^d\}\}$	a set of topological entities of dimension d in model M.
M_i^d	the ith entity of dimension d in model M. $d = 0$ for a vertex, $d = 1$ for an edge, $d = 2$ for a face, and $d = 3$ for a region.
$\{M_i^d\{M^q\}\}$	a set of entities of dimension q in model M that are adjacent to M_i^d.
$M_i^d\{M^q\}_j$	the jth entity in the set of entities of dimension q in model M that are adjacent to M_i^d.
$\{M_i^d\{D\}\}$	a set of degrees of freedom associated with mesh entity M_i^d
$M_i^d\{D\}_j$	j^{th} degree of freedom associated with mesh entity M_i^d.
N_i	the number of entities of dimension i in the mesh. Shorthand for $\|\{M\{M^i\}\}\|$. N_0, N_1, N_2 and N_3 are the number of vertices, edges, faces and regions, respectively, in the mesh.
Examples	
$\{M\{M^2\}\}$	the set of all the faces in the mesh.
$\{M_3^1\{M^3\}\}$	the mesh regions adjacent to mesh edge M_3^1.
$M_1^3\{M^1\}_2$	the 2nd edge adjacent to mesh region M_1^3.

3.1.1 Adjacency and Distributed Mesh Representation

Adjacencies describe how mesh entities connect to each other. For an entity of dimension d, the adjacency $\{M_i^d\{M^q\}\}$ returns all the mesh entities of dimension q, which are on the closure of the entity for a downward adjacency ($d > q$), or for which the entity is part of the closure for an upward adjacency ($d < q$). There are many options in the design of the mesh data structure in terms of the entities and adjacencies stored. If a mesh representation stores all *0* to d level entities explicitly,

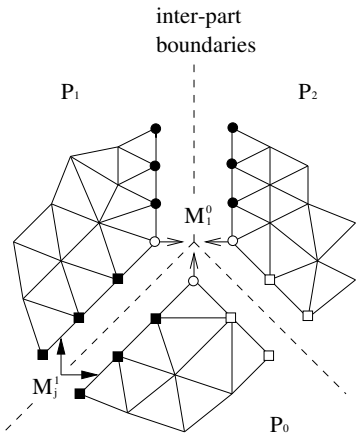


Figure 3.1: Distributed mesh on three parts P_0, P_1 and P_2 [63].

it is a *full* representation, otherwise, it is referred to as a *reduced* representation. *Completeness of adjacency* indicates the ability of a mesh representation to provide any type of adjacencies requested without involving an operation dependent on the mesh size such as the global mesh search or mesh traversal. Regardless of whether the representation is full or reduced, if all adjacency information is obtainable in O(1) operations, the representation is complete, otherwise, it is incomplete. We assume full and complete mesh representations throughout this thesis. Implementations with reduced complete representations using all the same overall algorithms are possible, with the addition of complexities within the mesh database API functions [72].

A distributed mesh is a mesh divided into parts for distribution over a set of processors for specific reasons, for example, parallel computation. Each part is treated as a serial mesh with the addition of mesh part boundaries to describe groups of mesh entities that are on inter-part boundaries. Mesh entities on inter-part boundaries are duplicated on parts where they are used in adjacency relations. Mesh entities that are not on any inter-part boundary exist on a single part. Figure 3.1 depicts a mesh that is distributed on 3 parts. Vertex M_1^0 is common to three parts and exists on each part, several mesh edges like M_j^1 are common to two parts. The dashed lines represent inter-part boundaries that consist of mesh vertices and edges (including mesh faces in 3D) duplicated on multiple parts.

3.2 Structure of *phParAdapt*

There are two modes of operation of *phParAdapt*, the preprocessing mode (demonstrated in Figure 3.2) and the mesh adapt mode. In the mesh adapt mode, *phParAdapt* takes the solution and errors from the finite element analysis (FEA) tool associated with the current resolution mesh, computes the discretization error and transfers the information to mesh size field which quantifies the desired resolution of the domain discretization. *phParAdapt* then calls *MeshAdapt* to perform the mesh modification procedures to alter the mesh to satisfy the specified mesh size field. If the application requires the solution to continue from the last available timestep, that solution field must be transfered to the new, adapted mesh. To accomplish this, a call back mechanism is used to locally transfer the existing solution to the adapted mesh as each mesh modification occurs.

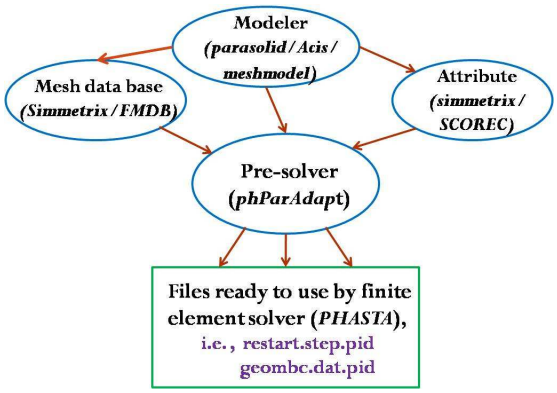


Figure 3.2: A demonstration of preprocessing and the relationship between the softwares.

In the preprocessing mode, *phParAdapt*, takes the mesh data base, solid model and attributes, the latter supplies additional analysis information about the problem definition (e.g., boundary conditions and initial conditions specified on the solid model) and gives a set of files that are ready to be used by finite element analysis tools (see Figure 3.2). In the specific case of PHASTA, those files include *restart* files which include the solution's initial conditions and *geombc* files which contain the boundary conditions, geometric information including vertex coordinates and connectivity, etc. The equation formation phase of the finite element analysis computes the element-level integrals which requires a mapping from each local vertex of

an element to a global vertex, the mapping is commonly referred to as connectivity. The relationship between parts required by point-to-point communication in the flow solver (refer to Section 2.4 for detailed discussion) is also set up in the preprocessing phase and included in the *geombc* files as well. Algorithm 2 demonstrates the main components of *phParAdapt* in the preprocessing mode.

Algorithm 2 The main components of *phParAdapt* in the preprocessing mode.

1: Load in the mesh
2: Redistribute the mesh if required (attach solution to the mesh before the redistribution if there is a solution associated with the existing mesh, which provides the initial condition.)
3: CC: SET UP PEER TO PEER COMMUNICATION REQUIRED BY SOLVER.
4: set up the neighborhood for each part
5: traverse the mesh entities on inter-part boundaries, for each mesh entity not owned by the current part, add the info. (e.g. entity, owners entity, owner's pid) to the *ilwork* if *ilwork* already exist between current part and the owner part, otherwise create an *ilwork* and add the information.
6: communication of *ilwork* between neighbors by point-to-point communication.
7: CC: SET UP BC AND IC ON EACH PART
8: attach natural boundary condition assigned on model by attributes
9: attach essential boundary condition assigned on model by attributes
10: if there is a solution field available for solution transfer, apply as the initial condition, otherwise apply the initial condition assigned on the model. Default value (zero) will be used if neither is specified.
11: setup the connectivity based on mesh database
12: write files ready to use by PHASTA solver, i.e., *restart.step.pid*, *geombc.dat.pid*

In order to set up the peer-to-peer communication required by FEA, on each part, *phParAdapt* has to identify its off-part mesh entities and setup the communication tasks and communication traces (see Section 2.4 for the definition), denoted as *ilwork*, for receiving and sending relevant information to the respective part where the information resides. As noted above, prior implementations of *phParAdapt* performed well for modest processor counts but the use of communications from each to every other processor at this stage introduced a bottleneck. This mode of communication is expensive and it is easy to run out the buffer size if non-blocking send and receive are used in the communication. As part of this thesis research, this mode of communication is replaced by the communication between neighbors only

(step 4 to 6 in Algorithm 2), where an extra step of setting up the neighborhood is required. This setup step is observed to be small as the overall efficiency is significantly improved. In an example of preprocessing for a case with 1 billion regions on 1024 parts, it takes about 30 minutes of wall clock time, including file IO (step 12 in Algorithm 2), to finish the preprocessing on 1024 opteron processing cores on NICS Kraken-XT4 if communication based on each to every other processor is used. The same test case takes less than 3 minutes under the new, neighborhood only, communication approach. Moreover, the pre-solver failed for the communication based on each to every other processor if this same 1 billion region mesh distributed on more than 2048 parts. On the other hand, the neighborhood based communication enables *phParAdapt* successfully set up cases up to 163,840 parts. This is one example where the software is efficient for computation in parallel with a modest number of processing cores, however has drawbacks in the computation with a large number of processors.

The relationship between the components of software associated in the preprocessing is demonstrated in Figure 3.2. *phParAdapt* is based on a mesh data base (*Simmetrix*/FMDB) and requires a geometric modeler (Parasolid/ACIS/discrete/ *meshmodel*) and an attribute system (*Simmetrix*/SCOREC) and provides output for the FEA. The march to petascale computers sometimes involves hardware or operating systems that are not supported by some of the commercial libraries. Therefore, the usefulness of open source alternatives to these commercial libraries is discussed in next section.

3.3 Open source attribute and modeler

The geometric based simulation environment consists of following components: the *geometric model* which houses the topology and the shape description of the computation domain, the *mesh* which describes the discretization of the domain, the *attributes* which provides the rest of the information needed to define the problem. The geometric based FEA input are provided by the pre-solver which reads the geometric model, the initial/adaptive discretization (mesh) of the domain and the attributes (providing extra information like boundary and initial conditions), and

reduces the information to only what is required by the flow solver. The previous section presents the issues related to *phParAdapt* in detail. In order to make use of all the supercomputers and various computing platforms, open source software is a critical issue. To achieve this objective, the algorithms and code within *phParAdapt* have been made compatible with the open source mesh data base (FMDB). Similar developments have been made to the the mesh modification tool (*MeshAdapt*). To maintain the capability to associate boundary conditions directly with the geometric model also requires the development of an open source geometric modeler and the development of an attribute system.

The commercial 3-D modeler systems, for example Parasolid^{TM} [2], ACIS^{TM} [1] Pro/Engineer^{TM} [5], are mature and provide complex geometric definitions, but they are not supported on all the supercomputers (e.g., Blue Gene). In addition to the geometric model, attributes defines the other information required by the simulation, like the material properties, boundary conditions, initial conditions, etc. While attributes could in principle be applied to mesh entities, for adaptive simulations they are most efficient when applied to geometric model entities. Once the attribute is associated with a model entity, it applies to all the mesh entities classified on this model entity. Commercial software, like Simmetrix, support a broad range of attribute types, including integers, vectors, tensors, and strings [3], but same as geometric modelers, they are not available on all the platforms.

SCOREC *meshmodel* creates the model topology from the mesh classification information. Once we have a mesh created on the CAD model (e.g., Parasolid), the mesh entity classification (to a model entity) is used to find a *meshmodel* as a surrogate to the CAD model. Then the model topology is saved for adaptation and other usage. The disadvantages of *meshmodel* are its fixed degree of geometric resolution and its lack of geometric details, while its advantage is that it can be made available on platforms of interest. Another alternative, OpenCascade [4] is a CAD/CAM/CAE kernel and development platform for 3-D modeling application which is available in open source. Effort is planned in the near future for making SCOREC tools compatible with OpenCascade.

While complex attribute systems have been developed in the past [50], a rela-

tively simpler attribute system associated with a *meshmodel* was developed in this research. Each of the attribute cases uses the following data structure,

```
structure Attribute{
  string  ConditionType;
  int     tag;    # tag of the model entity
  int     dim;    # dimension of model entity, 3,2,1,0 for region,
                  # face, edge, and vertex respectively
  double* value; # the condition values, the vector in format of
                  # value[0]:magnitute, value[1]~value[3]:direction
 }
```

Figure 3.3 provides an example of the way how information can be specified on model entities, where essential boundary condition of velocity is specified on the inlet and the no-slip walls, and pressure is weakly specified on the outlets.

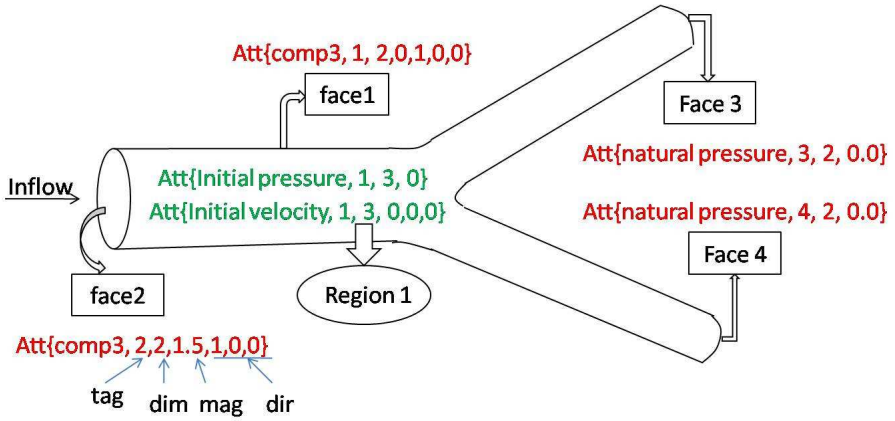


Figure 3.3: The boundary and initial conditions applied on the bifurcation model

3.4 Predictive load balance in mesh adaptation

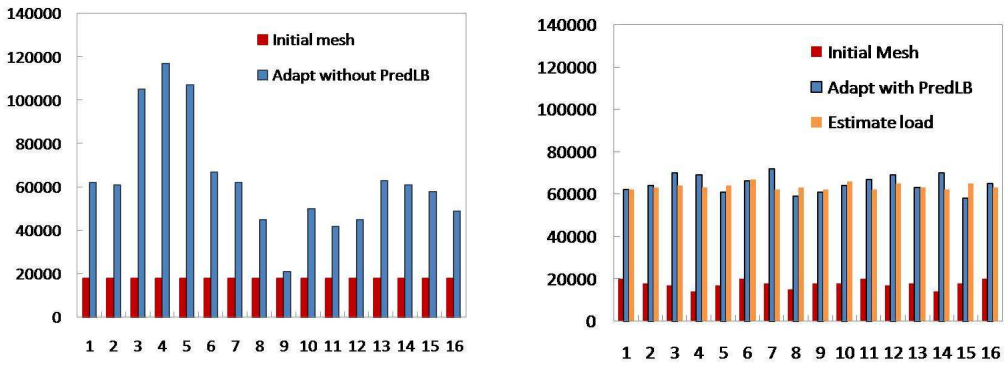
Another key component that enables petascale computations is the high resolution discretization of the physical domain with many millions, or even many billions of spatial degrees of freedom. Parallel mesh adaptation is a very promising approach to efficiently distribute degrees of freedom at this scale. As demonstrated

in Figure 1.1, discretization error based on the current computed solution is evaluated and the error information is transformed into a mesh size field that quantifies the desired mesh resolution over the domain. Mesh modification procedures are then carried out to alter the mesh to satisfy the specified mesh size field. The concept of the mesh metric field is discussed in Section 3.4.1. During the process of parallel mesh modification, without predictive load balance, some processors may have nearly all their entities scheduled for mesh modification which may exceed the limit of the physical memory of these processors and would either slow down the process or even kill the process. Flaherty *et al.* [24, 25] introduced a predictive load balancing algorithm to improve the entity balance after the refinement of the unstructured mesh adaptation process. A weight is specified to each of the current element based on the 3-D subdivision patterns after the edges have been marked for splitting required by mesh size field and a weight-based graph partitioning is performed before the refinement. A similar technique has also been used by Oliker *et al.* in [51]. This predictive balancing algorithm only improves the balance in the refinement stage of unstructured mesh adaptation, moreover, a greater imbalance may occur if more subdivisions (more iterations of refinement) will be performed than predicted by the original edge marking.

The predictive load balance ($PredLB$) algorithm introduced herein use *a priori* estimate of number of mesh regions after mesh adaptation. The algorithm has been added to mesh modification with the goal of achieving about equal number of regions in the adapted mesh on each part. Figure 3.4 demonstrates the number of mesh regions after mesh modification with and without using $PredLB$. Section 3.4.2 describes the algorithm of predictive load balance added to SCOREC mesh adapt tools [43, 44], $MeshAdapt$, and Section 3.4.3 presents the applications of $PredLB$ to demonstrate the usefulness of the algorithm.

3.4.1 Mesh metric field

Anisotropic mesh adaptation [44, 66, 68] requires a way to define the desired element size distribution over the domain. Mesh metric tensors are used to define an desired anisotropic mesh size field in the anisotropic mesh adaptation. The



(a) Without *PredLB* (b) With *PredLB*

Figure 3.4: A demonstration of the predictive load balance algorithm.

concept of a mesh metric field is used to represent the desired size field as a tensor over the domain. The mesh metric tensor at any point P in the domain is defined as a symmetric positive definite matrix $\mathbf{T}(P)$. The associated quadratic form $< \mathbf{x}, \mathbf{Tx} >= 1$, whose geometric representation will be an ellipsoidal surface (refer to Figure 3.5), defines a mapping of an ellipsoid in the physical space into a unit sphere in the transformed/metric space. In other words, any vector x at point P assumes a unit value in the space where any distance is measured under the metric $\mathbf{T}(P)$. The

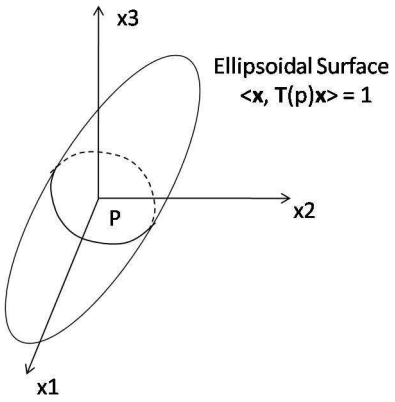


Figure 3.5: Ellipsoidal surface associated with the quadratic form

ideal goal of the mesh adaptation algorithm is to yield a mesh with regular elements

in the metric space where each edge e must satisfies the following equality:

$$< e, \mathbf{T}e >= 1 \tag{3.1}$$

A mesh with all its edges satisfying this relationship is referred as a *unit mesh*. However, the fact that we can not pack unit regular tetrahedral to satisfy constant unit mesh metric field over a domain indicates that a perfect matching is not possible. Therefore, a relaxed criteria is inevitable. Refer [42, 44] for details on relaxed criteria and element quality measures in the transformed space and [66] for the computation of mesh size field. In $MeshAdapt$, the mesh metric tensor $\mathbf{T}(P)$ is decomposed [42] to three unit directions (e_1, e_2, e_3) and the desired length h_1, h_2, and h_3 in each corresponding direction, the desired element volume (tetrahedral element) at point P is then computed as $V = h_1 h_2 h_3/6$.

3.4.2 Algorithm of predictive load balance

To avoid running out of the physical memory on certain processing cores in parallel mesh adaptation with nearly all their entities scheduled for mesh modification (especially refinement), a predictive load balance algorithm is developed to achieve about equal number of elements in the adapted mesh on each processor. Better balance may be obtained while moving smaller number of mesh entities between processors as compared to migration after mesh adaptation. To accomplish the goal of $PredLB$, weights have to be specified to graph nodes and a repartitioning on a weighted graph is performed. The weight is used to estimate the physical memory usage which is proportional to the number of elements to be generated.

Pain *et al.* [57] use the ratio of the element volume of the initial coarse mesh to the volume of desired element in the adapted mesh to estimate the number of elements to be generated in the mesh refinement. A similar idea combined with the concept of the mesh metric field specifies the weight on the graph node for $PredLB$. Given the initial mesh and mesh metric field attached to the vertex, for each graph node, the desired element volume at each vertex P is computed as $h_1(P)h_2(P)h_3(P)/6$. The average volume of the desired elements within the current element volume is estimated as $\sum_{p=1}^{n_{en}} h_1(p)h_2(p)h_3(p)/6n_{en}$, where n_{en} is

the number of nodes per element, which is 4 for tetrahedral mesh. The number of adapted elements within each of the current element is estimated by the volume of the current mesh element over the average volume of the desired mesh elements expressed as,

$$num = \frac{R_volume}{\sum_{p=1}^{n_{en}} h_1(p)h_2(p)h_3(p)/6n_{en}} \tag{3.2}$$

These numbers are specified as weights to the current elements which are defined as the graph nodes.

Some graph based partitioner in use through Zoltan interface (e.g., ParMetis) takes single precision integer as the object weight, which is constraining. To avoid rounding of fractional weights (including to zero), the smallest predicted weight is normalized to 1, which makes either the maximum weight or the summation on weights internal to ParMetis easily overflow a 4 byte integer. An artificial clip on either side of the "integer converter" is required to resolve this problem. Equation 3.3, 3.4 and 3.5 are the modifications to num needed to scale them in to a good range. Instead of the smallest weight, a larger value is used to normalize the predictive weight to keep the average from growing too high, and all those ones whose weights are less than one are promoted up to one to avoid zeros.

The following formulas are used to rescale num so as to guarantee the weight specified to the graph nodes to be in a good range of (4 byte) integers.

$if\ (max(num)/min(num) > maxvwgt)$

$$nvwgt = \frac{num}{max(num)/maxvwgt}, \tag{3.3}$$

else

$$nvwgt = \frac{num}{min(num)}, \tag{3.4}$$

$If\ (nvwgt < 1)$

$$nvwgt = nvwgt + 1, \tag{3.5}$$

where $maxvwgt$ is the prescribed maximum weight, $min(num)$ and $max(num)$ are the minimum and the maximum value of num, respectively, over the partition.

3.4.3 Applications of predictive load balance

In this section, examples of mesh adaptation on two geometric models are considered to study the performance of predictive load balance. The first one is on a patient specific abdominal aortic aneurysm (AAA) model (shown in Figure 3.6) and the second test is on a straight pipe model with air bubbles distributed in the pipe. The first test is performed on SCOREC opteron cluster, Borg, and the second example is performed on IBM Blue Gene/L at Rensselaer's CCNI.

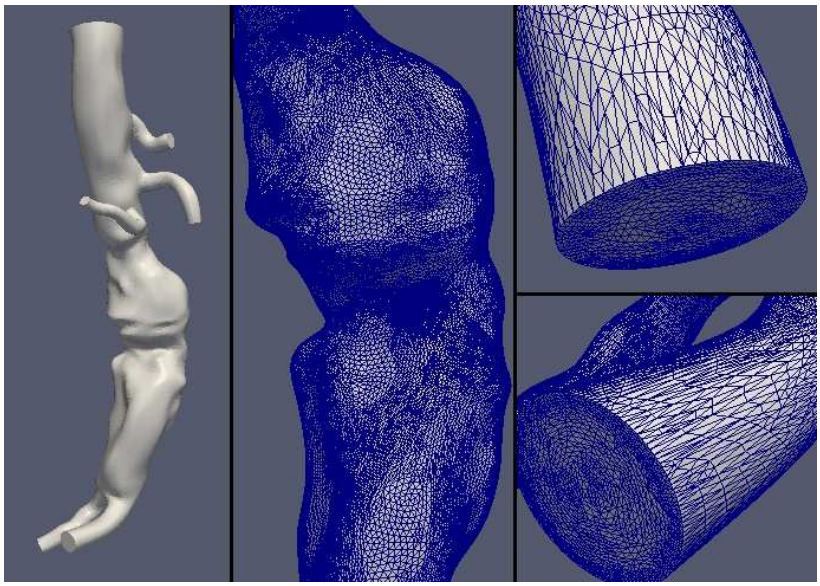
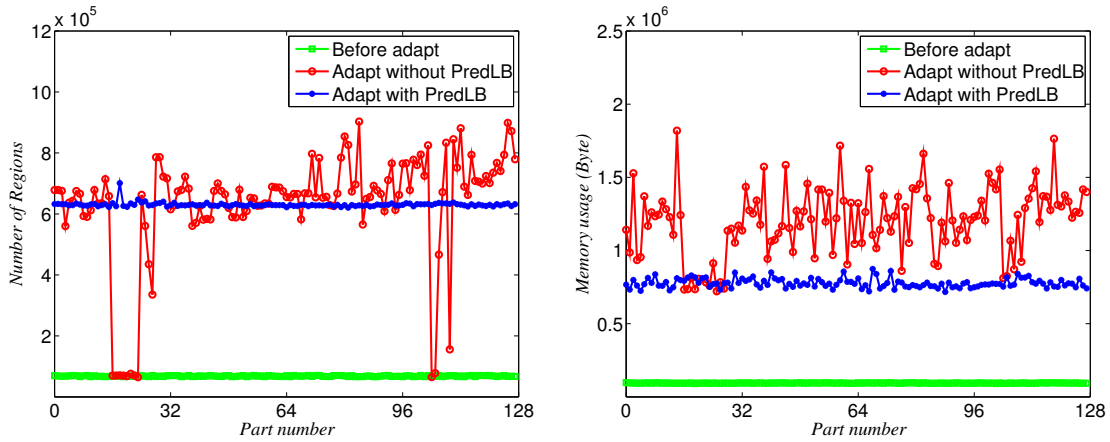


Figure 3.6: Geometry and a mesh of an AAA model.

3.4.3.1 AAA model

In the first example, an anisotropic adaptation on an AAA model is considered. The initial mesh with 8.7 million regions in total is distributed on 128 parts. The solution based anisotropic adaptation is carried out on the initial mesh to increase the local mesh resolution in order to capture the flow features better. A mesh with 80.5 million regions in total is obtained and the adaptation is performed on 128 AMD opteron processing cores at SCOREC with one part per core. Figure 3.7(a) demonstrates the number of regions on each part before and after mesh adaptation and compares that with and without the use of $PredLB$. Without $PredLB$, we see peaks (high number of regions) in the region distribution which uses more memory



(a) Number of regions on each part before and after adaptation

(b) Memory usage (high water mark) on each part before and after adaptation

Figure 3.7: (a) Number of regions and (b) memory usage on each part before and after mesh adaptation with and without using *PredLB* on an AAA model. Memory usage is measured in KB.

than other processors. By *PredLB*, the number of regions on each part becomes more even (with lesser variation). However, on part 18, the number of regions is a little higher than others. This is due to an underestimate of the adapted number of regions by *PredLB* algorithm in the coarsening process. Figure 3.7(b) shows the memory usage (the high water mark) before and after mesh adaptation with and without using the *PredLB* algorithm. The memory usage after adaptation is much higher without *PredLB*. The time usage of *PredLB*, including the migration of mesh entities, is 107 seconds, while the total mesh adaptation time is 68 minutes without using *PredLB* and 44 minutes with *PredLB*. In this specific example, coarsening is performed, but refinement is still the dominant part. The *PredLB* algorithm attempts to balance the number of regions in the adapted mesh and it improves the work load in the refinement procedures as well, therefore, the total time usage of mesh adaptation is reduced even though *PredLB* itself takes 107 seconds. In the coarsening heavy mesh adaptation procedures, *PredLB* may make balance of the work load worse, since it attempts to move mesh entities to the processors with a lot of coarsening which have lower numbers of mesh entities in the adapted

mesh. The objective of *PredLB* discussed in this thesis is to balance the number of mesh entities in the adapted mesh without considering the work load. Accounting for work load during predictive load balance is left for future work.

Another example of anisotropic mesh adaptation on the same AAA model is performed on 200 AMD opteron processing cores at SCOREC. The initial mesh is again with 8.7 million regions distributed on 200 parts and the adapted mesh has 187 million regions. Figure 3.8 shows the number of regions on each part before and after mesh adaptation. Without *PredLB* the test runs out of memory making it impossible to compare.

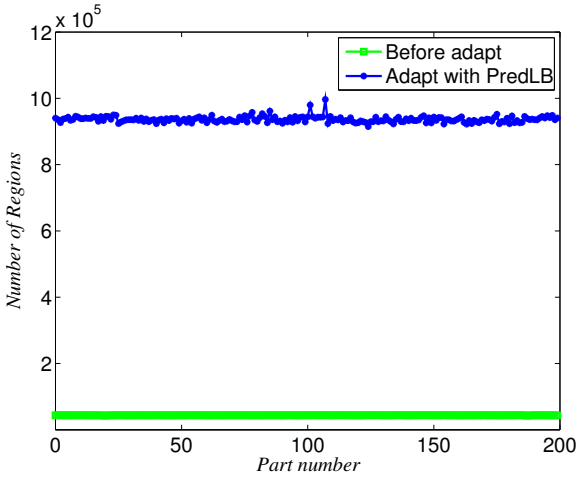


Figure 3.8: The number of regions on each part before and after mesh adaptation with predictive load balance on an AAA model.

3.4.3.2 Straight pipe model with air bubbles

The second example is on a straight pipe model with air bubbles distributing in the pipe, which is a physically relevant model (e.g., two phase flow simulation). The size field which defines the "air bubbles" in the model is shown in Figure 3.9. The blue color demonstrates small values of size field which implies a fine mesh (or high resolution) and the red color demonstrates relative large size field which implies a coarse mesh. In multi-phase flow simulation, meshes with high resolution at phase boundaries are needed due to the complex physical phenomena. The description of the size field of an air bubble is as follow:

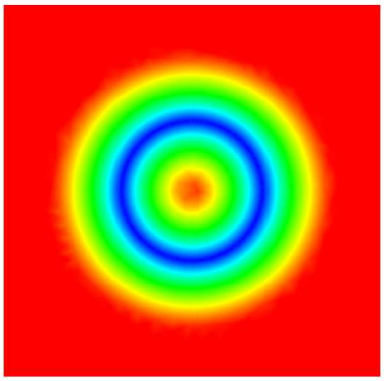


Figure 3.9: The size field of an "air bubble".

$for\ (0 < R < R_0)$

$$h_0 = h_1 = h_2 = c_1 * (1.0 - e^{-abs(R-R_0)*L}) + c_2, \tag{3.6}$$

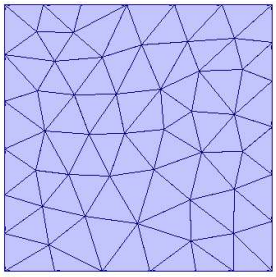
and else

$$h_0 = h_1 = h_2 = c_1 + c_2, \tag{3.7}$$

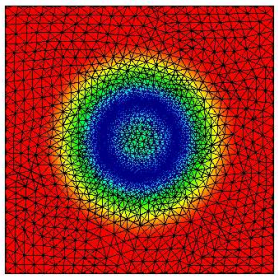
where R_0 is the radius of the air bubble and R is the distance of a point to the center of the nearest air bubble, c_1, c_2 and L are constants. The selected definition $(c_1 + c_2)$ of the mesh size field outside the air bubble regions gives a smooth transition in the mesh size.

A segment of the straight pipe model with air bubbles is shown in Figure 3.10(a). The mesh adaptation starts with an initial uniform tetrahedral element mesh and obtains non-uniform mesh with air bubbles. Figures 3.10(b) and 3.10(c) are examples of initial mesh and adapted mesh zoomed for 1/2 bubble colored by the magnitude of size field. To demonstrate the usefulness of $PredLB$ algorithm, the first test on this straight pipe model compares the number of regions on each part in the adapted mesh with and without using $PredLB$. The initial mesh is well balanced on 1024 parts with 1.24 million as the total number of mesh regions which is adapted to a 36.6 million region mesh with 5 air bubbles distributed in the straight pipe. The number of regions before and after the mesh adaptation with and without using $PredLB$ is demonstrated in Figure 3.11. Without $PredLB$, the number of regions on some parts are much higher than others (red line) after mesh

(a) A segment of the straight pipe model with air bubbles



(b) An example of initial mesh



(c) An adapted mesh zoomed for 1/2 bubble

Figure 3.10: A segment of a straight pipe model with distributed air bubbles colored by the magnitude of size field along with examples initial mesh and adapted mesh zoomed for 1/2 bubble.

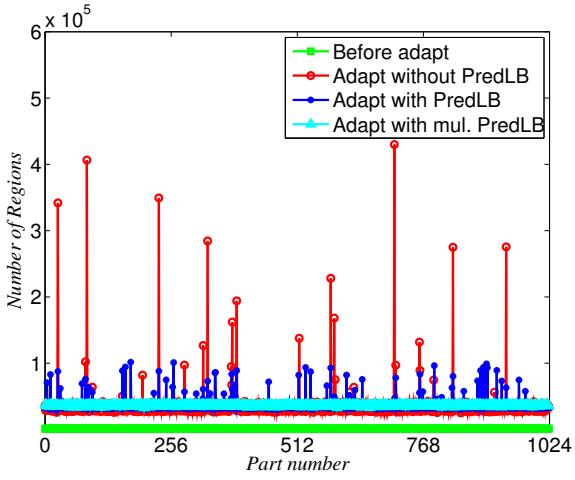


Figure 3.11: The number of regions on each part of the 1024 parts before and after mesh adaptation with and without *PredLB* on a straight pipe model with 5 air bubbles.

adaptation. The balance of mesh regions in the adapted mesh is improved by using *PredLB* (blue line), however, it is still not optimal. Because, unlike the case on AAA model discussed in the previous section, an analytical size field is utilized in this test, which is updated (recomputed using Eq. 3.6 and Eq. 3.7) after each iteration for newly created mesh entities. It is not easy to accurately estimate the number of mesh regions in the final mesh due to a very coarse initial mesh, which is the situation in this study. If the mesh adaptation is performed on supercomputers with relatively smaller memory per core (e.g., BG/L), then the *PredLB* can be used multiple times (i.e., used each time in the first several iterations) to further improve the balance in the final mesh (cyan line in Figure 3.11), hence avoid running out of memory in the whole adaptation process.

In a much bigger test, a uniform initial mesh with 17.2 million regions is well balanced on 16,384 parts and adapted to 1.06 billion regions using 16,384 IBM Blue Gene/L cores with multiple predictive load balance. 160 air bubbles are distributed in the pipe in the adapted mesh. Figure 3.12 shows the number of regions on each part before and after adaptation with multiple *PredLB*. The process runs out of memory without *PredLB*.

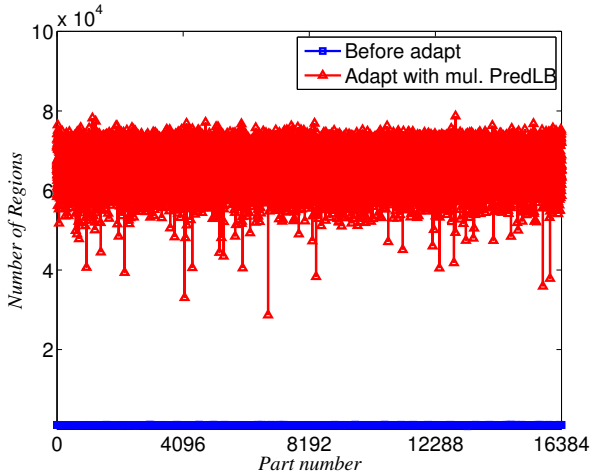


Figure 3.12: The number of regions on each part of the 16,384 parts before and after mesh adaptation with and without *PredLB* on a straight pipe model with 160 air bubbles.

3.4.4 Discussions and summary

A predictive load balance algorithm based on the concept of mesh metric field is developed for mesh adaptation with the objective of obtaining close to equal number of mesh regions on each part in the adapted part. The ratio of the element volume of the initial mesh to the volume of the desired element in the adapted mesh is used to estimate the number of elements to be generated. Non-uniform weights are assigned to graph nodes and weighted graph-based repartition is carried out to achieve the goal of predictive load balance. Examples on two geometry models, abdominal aortic aneurysm (AAA) and straight pipe with air bubbles are used to demonstrate the usefulness of the algorithm. The number of mesh regions in the adapted mesh is more balanced when applying $PredLB$, hence the amount of peak memory required is much smaller. By using $PredLB$ multiple times, 17.2 million region uniform mesh is successfully adapted to a 1.06 billion region mesh with 160 air bubbles in a straight pipe on 16,384 BG/L processing cores.

3.5 Multiple parts per processor

In the adaptive simulations, thousands or millions of degrees of freedom may increase to the order of billions, consequently the number of processing cores required is in the range of hundreds to thousands or even millions. The assumption of one part per processor makes sense for the flow solver on a given mesh but it is clear that when adaptivity changes the size of the mesh by one, two, and even three orders of magnitude the same number of processors cannot be used for every run that will be done for a given problem definition. The original one part per processor for the pre-solver software is a limit which constrains the computation to a fixed number of processors in the whole loop of the simulation, which is not practical for adaptive computations. In such a scenario, the processors will eventually run out of memory after several iterations if the total number of regions continues to grow while attempting to further reduce the errors on a fixed number of processors. Conversely, it is not efficient or even possible to start with too many processors at the beginning of a loop and the *a priori* estimation of the number of regions (and therefore processors) is not possible. Multiple parts per processor is able to

deal with this problem. Allowing multiple parts on each processor is useful in other situations as well. One such case is that the simulation may be performed on different architectures with different amount of per core memory, hence the number of processors required will be different even for a fixed-size problem. For example, each processing core at TACC Ranger has 2GB memory, while the one at Intrepid (BG/P) has 512MB. The simulation which nearly fits in 2,048 processors on Ranger requires 8,192 cores on Intrepid. Preprocessing with multiple parts per processor helps this partition re-distribution.

The objective of current work is to support changing (especially increasing) the number of parts in a partition when necessary. FMDB was extended by its developers to support multiple parts per processor. This section describes the capability of *phParAdapt* to allow multiple parts on each processor based on the design of FMDB.

3.5.1 Design of multiple parts per processor

Figure 3.13 is an example of a partition with multiple parts on each processor. The solid line is the inter-processor part boundaries, while the dashed line is the intra-processor part boundaries. The pair of processor rank and a local part ID defines a unique part (ID of a part). The number of parts on each processor is arbitrary and could be same or different on different processors.

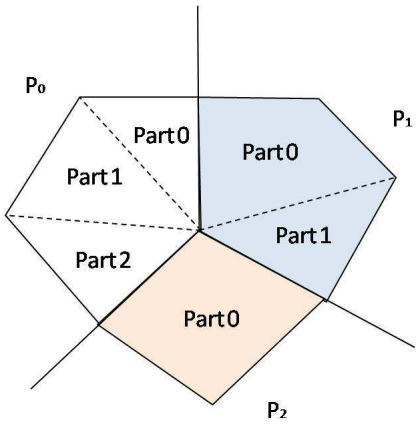
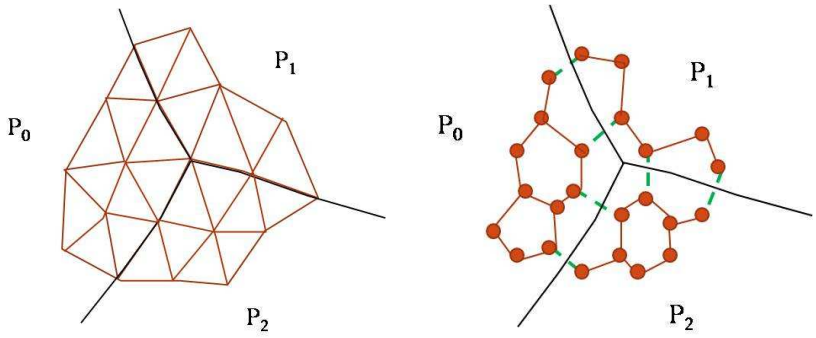


Figure 3.13: An example of multiple parts per processor. The solid lines represent the inter-processor part boundaries and the dashed lines represent the intra-processor part boundaries.

3.5.1.1 Partition redistribution

Changing the number of parts of a partition is achieved by partition redistribution algorithms. Graph [39, 40, 41, 58, 61, 71, 70, 93] and hypergraph [16] based partitioning schemes are currently used as they are more suitable for unstructured mesh calculations that process irregular structures and connections. Such partitioning schemes are well known to produce a partition of a mesh into parts that are well balanced, in terms of the defined part-object type, while requiring minimum inter-part communications. FMDB is interfaced with the Zoltan partitioning library [22] which provides a suite of such load balancing schemes including ParMetis [41], PT-SCOTCH [58], PHG [16]. In the application of these methods, the graph nodes and edges used by graph/hyper-graph based partitioning algorithms need to be defined. In the case of a 3-D flow analysis, the defined graph nodes, the objects to be partitioned into parts, are the mesh regions which correspond to the finite elements or entity groups (a group of mesh entities to be partitioned that need to stay together). This selection is natural for element-integration/equation-formation stage of the FEA making it highly scalable. So long as the balance of the vertices corresponding to degrees of freedom ($dofs$), or unknowns in the system, is also preserved, this partitioning also maintains the scalability of the iterative linear solver. While it is possible to create a full set of graph edges associated with a full set of adjacencies, testing indicated that creating all the graph edges based on a full set of adjacencies requires a substantially larger time without producing a better partition. Therefore, a subset of graph edges defined by only face adjacencies is used in the current study. Two types of graph edges are defined as follow. The edges connecting graph nodes on different processors cross processor-boundaries, which are referred to as inter-edges (i.e., inter-processor edges). On the other hand, the ones connecting graph nodes on the same processor do not cross processor-boundaries, which are referred to as intra-edges (i.e., intra-processor edges). Figure 3.14 is one example showing a mesh and the corresponding graph, in two dimensional for clarity. The mesh faces in 2-D (regions in 3-D) are defined as the graph nodes and the graph edges are defined for each of the graph nodes that share an edge (face in 3-D). The mesh is partitioned into 3 parts and distributed on 3 processors (see Figure 3.14(a)). The

corresponding graph is build up for this mesh, which is shown in Figure 3.14(b). The red solid lines demonstrate the intra-edges and the green dashed ones represent the inter-edges.

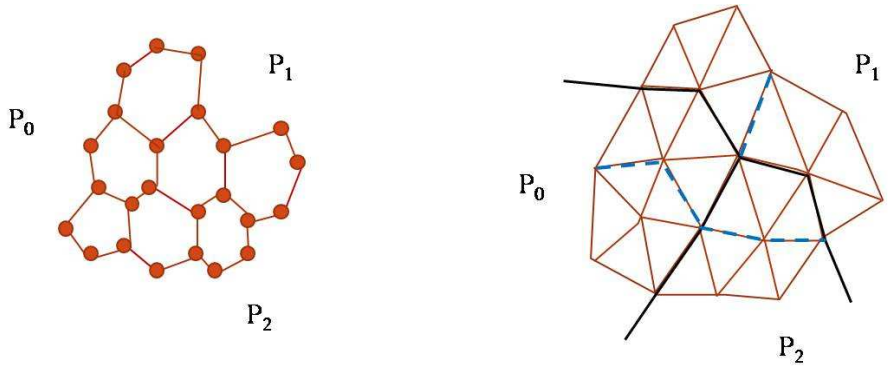


(a) (b)

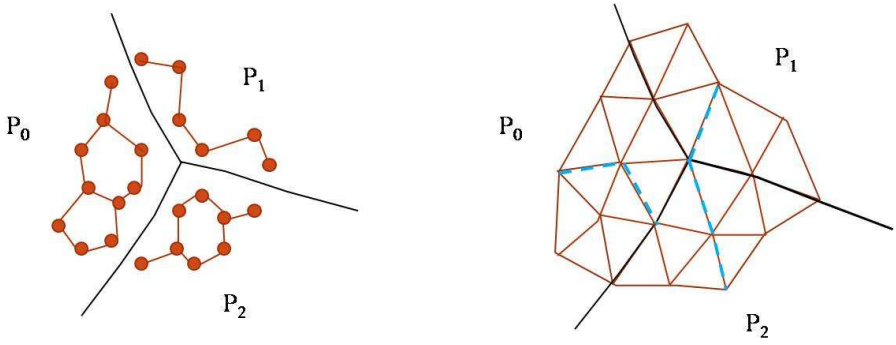
Figure 3.14: A distributed mesh and its corresponding part graph. (a) The mesh is partitioned into 3 parts and distributed on 3 processors (i.e., one part per processor). (b) Corresponding part graph. The circles represent the graph nodes. The red solid edges demonstrate the intra-processor graph edges and the green dashed ones demonstrate the inter-processor graph edges.

Two repartitioning categories are defined based on graph edges provided to graph/hypergraph based partitioner: *global partitioning* which considers both intra-processor and inter-processor edges, and *local partitioning* which considers only intra-edges. The definition is slightly different with the ones in [87], but they are essentially equivalent. The global partitioning considers all the graph edges and provides a balanced partition with the total minimum inter-part communication. On the other hand, the local partitioning considers only the on-processor (intra) graph edges, without knowing the existence of other graph nodes and graph edges on other processors. In this case, the repartition is carried out independently on each processor, as a serial process on each of them. The distributed mesh in Figure 3.14 is redistributed to six parts and two parts per processor, using these two categories of partitioning algorithm respectively. The resulting partitions are shown in Figure 3.15(a) and Figure 3.15(b). The black solid lines represent the inter-processor part

boundaries and the cyan dashed lines demonstrate the intra-processor part boundaries. It clearly shows that, in the global partitioning, the redistribution is based on the global graph information (left of Figure 3.15(a)). The inter-processor part boundary is changing in the redistribution (i.e., mesh entities are migrated between parts on same and different processors). Conversely, in the local partitioning, the redistribution is based on the intra-edges of the graph without knowing the existence of the other processors and the inter-processor part boundaries stay fixed (i.e., there is no migration of mesh entities between processors).



(a) The graph (left) and the partition (right) given by global partitioning



(b) The graph (left) and the partition (right) given by local partitioning

Figure 3.15: The graph and the partition given by (a) global partitioning and (b) local partitioning.

Local partitioning can be accomplished more efficiently without considering about the information on other processors [87], especially for a partition with a large number of graph nodes or a large number of parts. However, as local partitioning

is repeated, the quality of the partition will be decreased due to the compounding effects, since the balance of a partition uses only limited load information, thus limiting the quality of the partition relative to global partitioning. An imbalance tolerance has to be specified (e.g., 1.03) in the partitioning. If the number of mesh regions on each part over the average number of mesh regions across the partition is smaller than the imbalance tolerance, the balance of a partition is satisfied. The imbalance ratio of a partition is then defined as the maximum number of mesh regions per part over the average number across the partition. For example, a mesh with 8 million regions is distributed on 1024 parts, the average number of regions is $8M/1024 = 7813$. If for example, processor (or part) 799 has 8016 regions, which is more regions than all the other 1023 parts, then the imbalance ratio of this partition equals $8016/7813 = 1.026$ (*i.e.* 2.6%). In the local partitioning case, considering this same example with the $8M$ region mesh, the initial partition has 1024 parts distributes on 1024 processors (one part processor), with 2.6% imbalance ratio. This partition is redistributed to $16,384$ parts (16 parts per processor) with 1.03 given as the imbalance tolerance. Since this 1 to 16 redistribution (local partitioning) on each processor considers the average number of regions across the parts on this processor (e.g., 8016 as the base on processor 799) instead of the average across all the parts of the whole partition (i.e., 7813), the imbalance ratio could easily go up to $1.026 \times 1.03 = 1.05678$ or about 5.7% (e.g., the worst part on the processor with the maximum initial imbalance ratio, i.e., processor 799). In the applications, the global and local partitioning are used for different purposes, either separately or combined in a proper way. This issue and its impact on scaling must be carefully considered.

3.5.1.2 Multiple parts per processor in *phParAdapt*

Based on the structure of *phParAdapt* which is discussed in Section 3.2, the design of multiple parts per processor in preprocessing is introduced in this section. There are two main tasks in the preprocessing mode in *phParAdapt* as presented earlier. One is to set up the communication trace (*ilwork*) between parts for peer-to-peer communication in the FEA (see Section 2.4). The other is to setup the mesh

connectivity, initial conditions, boundary conditions, etc., in the form used in the FEA program based on the mesh database, the geometric model, and the attributes. To accomplish the task, only on-part information is needed without communication to other parts. The extension to allow multiple parts in this phase is straight forward. The original program (allowing one part on each processor) is wrapped with an extra loop over the number of parts on each processor (see Algorithm 3). The data structures (e.g., arrays) are extended with one more dimension to handle

Algorithm 3 Design of multiple parts per processor in preprocessing to setup the connectivity, initial conditions, boundary conditions, etc.

for $(ipart = 0; ipart < nparts; ipart++)$ **do**
  CC: ON PART COMPUTATION
  e.g., $ien(npart, nel, nen)$, $x(npart, nshg, nsd)$, $y(npart, nshg, nflow)$, $BC(npart, nshg, numbc)$, and etc.
end for

the number of parts on each processor. For example, the connectivity array (for an element number and a local node number of this element, it gives the global node number on this part), $ien(nel, nen)$ is extended to $ien(nparts, nel, nen)$, where $npart$ is the number of parts on each processor, nel is the number of elements on each part, and nen is the number of local nodes in each element. Note that it is not necessary for $npart$ to be same on different processors. On each processor, the multiple parts is processed one after another (sequentially).

The extension to allow multiple parts in the other main component of preprocessing, i.e., inter-part communication setup phase, is more complicated. The communication in pre-solver is improved from a mode based on each to every other processor to one between neighbors only (details in Section 3.2). However, a simple wrap of loop over the number of parts on each processors over this part of program will cause a dead lock in the communication. Figure 3.16 demonstrates one of such an example. Processor P_0 contains three parts and P_1 contains two parts. The red dashed line demonstrates the communication between neighbors. For example, P_0 is not able to process *part* 1 or *part* 2 without finishing *part* 0 which needs the communication with *part* 1 on P_1, however, P_1 is not able to process *part* 1 without finishing *part* 0 which requires the communication with *part* 1 and *part* 2 on

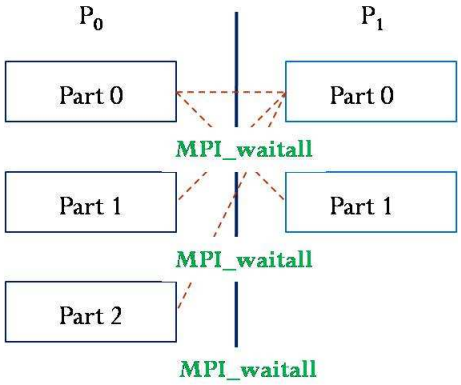


Figure 3.16: An example of dead lock in the communication

P_0. To resolve this problem, this part of preprocessing is decomposed to several pieces; pure on part calculation (without any inter-part communication), inter-part communication, and on part calculation again, and so on, which is shown in Algorithm 4. Non-blocking send and receive is used in the communication part to set up this peer-to-peer communication for the finite element solver (see Section 2.4). It is worth mentioning that, the *tag* in the MPI_Isend equals the local part ID of the receiver, and *dest* is the processor ID (rank) of the receiver. A pair of *dest* and *tag* defines a unique receiver. Likewise for MPI_Irecv. $MPI_Waitall$ stands outside the whole loop to avoid dead locks.

3.5.2 Examples/Applications

One fundamental requirement of scalable finite element solvers is the balance of the work load on each part. The global and local partitioning algorithms will be utilized in this section to provide balanced partitions for finite element solver scaling studies on an abdominal aortic aneurysm (AAA) model on Intrepid (IBM BG/P system) at ALCF, ANL and Kraken (Cray XT5 system) at NICS. The geometric model and the mesh are demonstrated in Figure 3.6. In the first example, a mesh with 133 million elements is considered. The range of parts of the partitions goes from 1,024 to 65,536. A global partitioning is used in the first test. In the second study, a relatively large mesh with 1.07 billion elements on the AAA model is partitioned up to 163,840 parts, which uses all 40 racks of Intrepid. A local partitioning is used in this test to obtain the partitions, complementary to the global

Algorithm 4 Design of multiple parts per processor in preprocessing to setup the inter-part communication for finite element solvers.

```
for (ipart = 0; ipart < nparts; ipart++) do
  on part calculation (no communication with other parts)
end for

CC: NON-BLOCKING SEND AND RECEIVE
for (ipart = 0; ipart < nparts; ipart++) do
  for loop over the number of sends on this part do
    MPI_Isend(*buf,count,datatype,dest,tag,comm,*requst)
  end for
  for loop over the number of receives on this part do
    MPI_Irecv(*buf,count,datatype,source,tag,comm,*requst)
  end for
end for
MPI_Waitall()

for (ipart = 0; ipart < nparts; ipart++) do
  on part calculation (no communication with other parts)
end for

.....
```

partitioning in the first example.

3.5.2.1 AAA model with $O(10^8)$ elements

In the first example, a mesh with 133 million elements on a patient-specific abdominal aortic aneurysm (AAA) model is considered (as shown in Figure 3.6). The mesh is obtained through adaptive refinement of an anisotropic mesh obtained from previous adaptation cycles. In this case, scaling studies are performed on partitions containing from 1,024 parts to 65,536 parts, and thus, using 1,024 to 65,536 processing cores, respectively, on Intrepid. To create all the mesh partitions, a global partitioning strategy is applied and we use the graph partitioner, ParMetis [41], in this example. The partitioning starts with a well balanced mesh (2.7% region imbalance) distributed on 512 parts. The repartitioning is carried out on 512 opteron processors at NICS Kraken-XT4. Table 3.1 lists the average number of regions and vertices of each partition along with imbalance ratio. Due to the "global" partitioning, the imbalance ratio of the regions (the part-objects) satisfies the imbalance tolerance globally (the 4th column of Table 3.1), which is set at 1.03.

Num of parts	Ave num Rgns	Ave num vtxs	Rgn. imbal.	Vtx. imbal.
1,024	130,833	24,413	1.023	1.053
2,048	65,417	12,555	1.023	1.061
4096	32,708	6,498	1.019	1.088
8,192	16,354	3,392	1.019	1.098
16,384	8,177	1,781	1.020	1.130
32,768	4,088	948	1.023	1.158
65,536	2,044	509	1.024	1.203

Table 3.1: The average number of regions and vertices of a 133 million element mesh of an AAA model, along with the imbalance ratios.

After repartitioning, each processor contains multiple parts. For example, each processor has 16 parts in the 8,192 part partition. And the number of regions on each part of this partition is demonstrated in Figure 3.17. There are lightly loaded parts (with a small number of mesh regions) in this partition, however, they have little impact on scaling. The heavily loaded parts of this partition that determine the scaling, satisfy the imbalance tolerance.

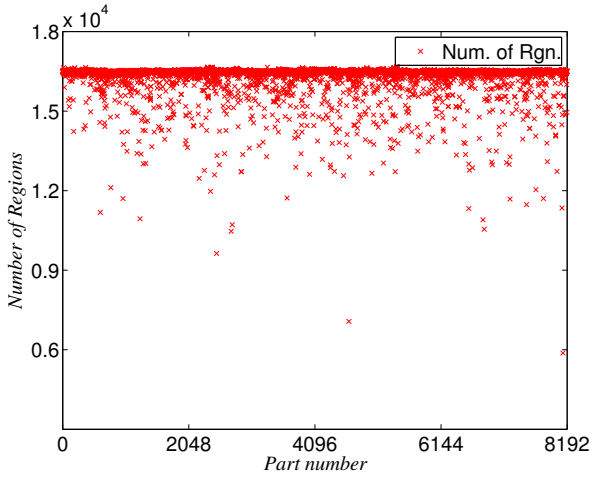


Figure 3.17: Number of regions on each part of a 133 million element mesh of an AAA model for a partition with 8,192 parts.

The scaling study of the finite element solver is performed on Intrepid using all the 4 cores on each compute node, and allowing one part on each processing-core. A Womersley profile [98] is applied at the inlet and rigid vessel walls with no-slip boundary conditions are considered. At outlets a lumped parameter bound-

ary condition [91] is applied. The study involves 20 time steps and the time usage for different partitions is measured and the scaling factor is computed (i.e., $s_factor = t_{base} \times np_{base}/(t_i \times np_i)$), see Table 3.2. A near-perfect strong scaling

$133M$ elements		Eqn. form.		Eqn. soln.		Total	
num. of core	avg. elem./core	time	s-factor	time	s-factor	time	s-factor
1,024 (base)	130,833	191.98	1.0	253.72	1.0	445.7	1.0
2,048	65,417	95.69	1.003	127.35	0.996	223.0	0.999
4,096	32,708	47.67	1.007	63.73	0.995	118.2	1.000
8,192	15,354	23.89	1.004	33.15	0.957	57.04	0.977
16,384	8,177	11.90	1.008	16.89	0.939	28.80	0.967
32,768	4,088	6.02	0.996	8.87	0.894	14.89	0.935
65,536	2,044	3.04	0.987	5.26	0.754	8.30	0.839

Table 3.2: Strong scaling results on an AAA model with 133M elements on Intrepid:IBM BG/P (execution time is measured in seconds).

factor of the finite element solver is observed as we increase the number of processing cores from 1,024 up to 32,768, see the last column in Table 3.2. The compression in analysis time (strong scaling) when increasing the processing cores is required for time critical simulations. The time usage of the two main components of the finite element analysis, equation formation and equation solution, is measured and listed in the table. A near-perfect scaling is observed for system formation all the way up to 65,536 parts, while a degradation of equation solution is observed above 32,768 processing cores. This is due to the increased nodal imbalance for a fixed-size problem distributed on a large number of parts (a 20.3% nodal imbalance is observed on the 65,536 part partition). Algorithms based on iterative load balancing operations between neighboring parts have been developed in this thesis to improve the partition and in turn parallel performance, see details in Chapter 4.

3.5.2.2 AAA model with $O(10^9)$ elements

As a second case, we consider a relatively large mesh with O(10^9) elements for an AAA model (as shown in Figure 3.6). Again, the mesh is obtained through adaptive refinement of an anisotropic mesh obtained from previous adaptation cycles. In this case, scaling studies are also performed on the near-petascale system, Intrepid

(IBM BG/P system) at ALCF, ANL. The number of cores covered range from 4,096 to 163,840, i.e., from 1 rack to all 40 racks of BG/P covering the full system-scale. To be able to create all the mesh partitions, we used hypergraph partitioner in this case, i.e., PHG [16], and a local partitioning strategy is applied. The average number of mesh regions and vertices for each partition along with imbalance ratios are given in Table 3.3. The local partitioning starts with a well balanced (2.49% region

Num of parts	Ave num Rgns	Ave num vtxs	Rgn. imbal.	Vtx. imbal.
4,096 (base)	261,667	48,054	1.0549	1.0736
8,192	130,834	24,732	1.0549	1.0841
16,384	65,417	12,822	1.0549	1.0990
32,768	32,708	6,707	1.0549	1.1210
65,536	16,354	3,550	1.0549	1.1719
98,304	10,903	2,465	1.0549	1.2117
131,072	8,177	1,908	1.0549	1.1782
163,840	6542	1,570	1.0549	1.3571
4k to 160k	6542	1,561	1.0554	1.1950

Table 3.3: The average number of regions and vertices of a 1.07 billion element mesh of an AAA model, along with the imbalance ratios.

imbalance) mesh on 2048 parts, which is obtained by globally partitioning using ParMetis. Each part of the 2048 parts is split to multiple parts, for example, from 1 to 80 parts on each processor when the mesh is redistributed from 2048 parts to 163,840 parts. As mentioned in the previous section, the local partitioning is more efficient, especially for the case with large number of mesh entities and large number of parts, however, the compounding effect makes the imbalance ratio to increase. We see this compounding effect easily from the 4th column in Table 3.3. A 1.03 imbalance tolerance is set up in the local partitioning and an imbalance ratio of 1.0549 is obtained in the final partitions. The local partition is performed on 2,048 opteron processing cores at NICS Kraken (Cray XT5). The vertex imbalance of the 163,840 part partition goes up to 35.71%, which is detrimental to the scaling of equation solution phase of the finite element analysis tool, which will be discussed and resolved in the next chapter. If we take a partition with 4,096 parts which is obtained by the global partitioning procedures and then apply local partitioning on it (each of the 4,096 part split to 40 parts), a slightly better partition is obtained

(from the standpoint of vertex imbalance). The vertex imbalance is 19.5% with 1,561 as average number of vertices per part and the region imbalance is 5.54%, which is expected due to the compounding effect. This partitioning ($4k$ *to* $160k$) is performed on 4,096 Kraken (Cray XT5) processing cores. The information of this partition is listed in the last row of Table 3.3 and it is the one used in the scaling study. Figure 3.18 demonstrates the number of regions on each part of this partition with 163,840 parts.

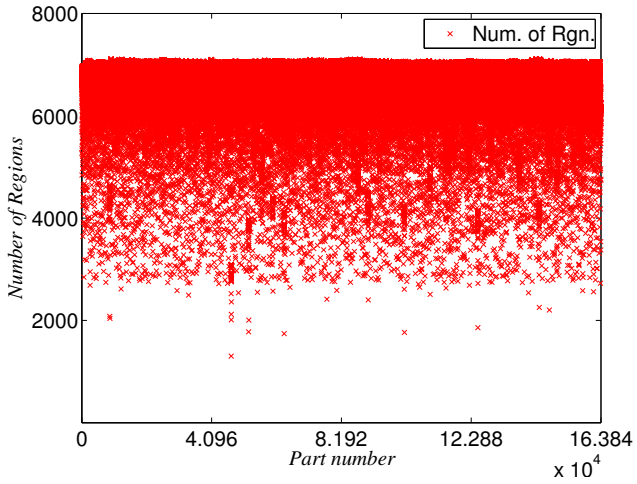


Figure 3.18: Number of regions on each part of a 1.07 billion element mesh of an AAA model for a partition with 163,840 parts.

The same problem set up (boundary conditions) is used as the previous section in the flow simulation and again, 20 time steps are solved in the scaling study. Here, we measure the execution time in terms of equation formation and equation solution so as to understand the effect of partitioning on scaling, specifically due to the imbalance in mesh vertices. Figure 3.19 and Table 3.4 provide the scaling factor and execution time (including for both work components). We observe a near-perfect strong scaling (95% or above) for equation-formation stage all the way to the full system-scale. On the other hand, strong scaling for equation-solution stage drops below 90% at 65,536 cores and onwards. The strong scaling for both work components together is at 90% or above till 65,536 cores and drops to 82% while using 163,840 cores or all the 40 racks of BG/P in Intrepid. Therefore, with 20 time steps in this case, each step takes only 1.25 seconds for an implicit solve (over

163,840 cores) on an adapted mesh with $1B$ elements (where, B denotes billion). This demonstrates the ability of the current procedures to achieve strong scalability on prospective supercomputers with hundreds of thousands of cores.

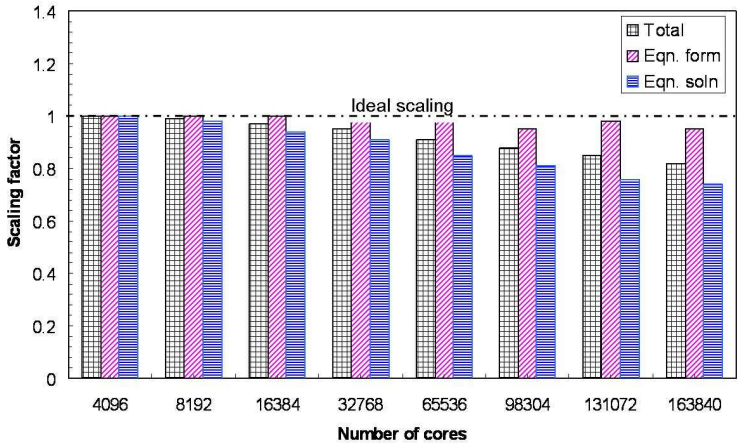


Figure 3.19: Strong scaling performance for an AAA model with O(10^9) elements up to 163,840 cores of Intrepid, IBM BG/P system. In this case, execution time is measured in terms of equation formation, equation solution and the total.

It is important to point out that the imbalance in mesh vertices goes up from 7% in the partition with 4,096 parts to 17% in the case of 65,536 parts; and is as high as 19.5% in the partition with 163,840 parts. Furthermore, note that the average number of mesh vertices per part (or core) relatively increases when a fixed-size problem is spread into more and more parts, for example, 48,000 is the average number of mesh vertices per part in the partition with 4,096 parts and is 1,561 on average in the case of 163,840 parts; which is about 30% higher relatively ($1,200$ mesh vertices will be ideal based on 48,000 when going from 4,096 parts to 163,840 parts). This relative growth in the average number of mesh vertices per part on higher partitions leads to a situation where computational work is not properly scaled as compared to the base. Incremental techniques to improve the partitions in order to obtain better balance for both mesh regions and vertices are developed and presented in the next chapter.

The same strong scaling study has been performed on Kraken at NICS using up to 65,536 processing cores. The results are demonstrated in Table 3.5. Consistent

Intrepid: 1.07B element		Eqn. form.		Eqn. soln.		Total	
num. of core	avg. elem./core	time	s-factor	time	s-factor	time	s-factor
4,096 (base)	261,600	388.68	1	456.24	1	844.92	1
8,192	103,800	194.48	1.00	233.86	0.98	428.34	0.99
16,384	65,400	97.48	1.00	120.90	0.94	218.38	0.97
32,768	32,700	49.01	0.99	62.33	0.91	111.34	0.95
65,536	16,350	24.59	0.99	33.70	0.85	58.29	0.91
98,304	10,900	16.42	0.99	23.56	0.81	39.98	0.88
131,072	8,175	12.37	0.98	18.65	0.76	31.02	0.85
163,840	6,540	10.27	0.95	15.34	0.74	25.61	0.82

Table 3.4: Strong scaling results on an AAA model with O(10^9) elements up to 163,840 cores on Intrepid:IBM BG/P (execution time is measured in seconds).

with the observation on Intrepid, the system formation scales all the way up to 65,536 processing cores, however, a degradation of equation solution is observed much earlier due to the architecture differences.

Kraken: 1.07B element		Eqn. form.		Eqn. soln.		Total	
num. of core	avg. elem./core	time	s-factor	time	s-factor	time	s-factor
4,096 (base)	261,600	65.05	1	225.19	1	290.24	1
8,192	103,800	32.80	0.992	110.00	1.014	143.80	1.009
16,384	65,400	16.65	0.977	64.65	0.871	80.79	0.900
32,768	32,700	8.15	0.998	39.42	0.714	47.57	0.763
65,536	16,350	4.06	1.002	22.44	0.627	26.50	0.685

Table 3.5: Strong scaling results on an AAA model with O(10^9) elements up to 65,536 cores on Kraken: Cray XT5.

3.5.3 Discussion and summary

Allowing only one part on each processor is a limitation in petascale simulations, which constrains the whole cycle of adaptive parallel numerical method to a fixed number of processors. This issue is resolved by extending *phParAdapt* to support multiple parts on each processor. A partition redistribution is required to change the number of parts of a partition. Based on the graph information given to the partitioner, the strategy used could be a global partitioning or a local partitioning. Both of them (combined or separately) are used in the applications on the scaling study on an AAA model, up to the whole machine scale (163,840 processing

cores) of Intrepid (IBM BG/P) at ALCF, ANL and also nearly up to full system scale (65,536 processing cores) of Kraken (Cray XT5) at NICS.

This chapter focused on the issues in the mesh adaptation and preprocessing stages when dealing with large-scale problem cases. The bottlenecks towards petascale computation are identified and resolved, including compatible open source software, developed a predictive load balance algorithm to balance the memory usage in the mesh adaptation, and allowing multiple parts per processor in *phParAdapt*. With these obstacles resolved, a mesh with 17.2 million elements is adapted successfully to a 1.06 billion mesh on a straight pipe model using 16,384 processing core at CCNI (IBM BG/L system). A scaling test on a mesh with 1.07 billion elements on an AAA model is performed on Intrepid using up to 163,840 processing cores (the full system) as well as on Kraken using up to 65,536 processing cores (nearly the full system).

CHAPTER 4
PARTITION IMPROVEMENT ALGORITHMS

The ability to scale implicit finite element and other implicit mesh-based computations on massively parallel computers requires ensuring effective balance of work load for both the system formulation and the system solution. This chapter presents Multiple Compute-Object based Partitioning Improvement (MCOPI) algorithms for the incremental redistribution of mesh entities provided by a single compute-object balanced partition. Example computations executed on an IBM Blue Gene/P system demonstrate the usefulness of the procedures, particularly for time critical calculations where individual cores may be lightly loaded. We have recently developed two distinct MCOPI algorithms that have been added to pre-solver to improve scalability. They are referred to as Local Iterative Inter-Part Boundary Modification (LIIPBMod) and Heavy Part Splitting (HPS).

Graph based partitioners [39, 40, 41, 58, 61, 71, 70, 93] are well known to produce a partitioning of the mesh into parts that are well balanced, in terms of the defined part object type, while requiring minimal inter-part communications. In the application of these methods the graph nodes and edges used by the graph-based partitioning algorithm need to be defined. In the case of a 3-D flow analysis, the defined graph nodes (objects to be partitioned into parts) are the mesh regions (which correspond to the finite elements). Graph edges are often defined between each of the graph nodes for mesh regions that share a mesh face. This selection does an excellent job of balancing the number of regions and therefore the workload for the construction of the part level finite element system. However, the workload balance for the iterative solution of the resulting system is proportional to the number of degrees of freedom or mesh vertices in case of linear finite elements (e.g., in matrix-vector products and vector norms). Since mesh vertices are not the objects in the original partitioning, the balance may not be optimal, particularly when the numbers of mesh entities per part is relatively low (e.g., a few thousand). Scaling in this lightly loaded part scenario is of interest because it improves performance

for time critical solution processes where large numbers of processors are used to solve the problem within the short time window; what we refer to as solution time compression. The multi-constraint graph partitioning [41, 71] assigns weight vectors to graph nodes and then partition the mesh in such a way that each of the weights is balanced. Multi-objective graph partitioning [70] assigns weight vectors to graph edge and simultaneously minimize several different cost functions (sums of edge weight). Neither of these balance mesh regions and mesh vertices simultaneously. Multi-phase/multi-physics graph partitioning [41] simultaneously balance the computations performed at each phase (e.g., particle-in-mesh), which is similar to the simultaneous balance of mesh regions and vertices. The added complexity of finite element computation is that the mesh vertices are duplicated on part boundaries. Multi-phase/multi-physics graph partitioning cannot a priori account for the copies on the part boundaries and the partitioned mesh does not result in a balance of on-part vertices and may even become invalid for finite elements. Two multiple compute-object based partitioning improvement algorithms are developed to reduce the vertex imbalance thereby improving the overall balance of two types of mesh entities as required by a scalable implicit solve.

Section 4.1 introduces two recently developed MCOPI algorithms: 1) LIIPB-Mod iteratively performs minor adjustments to the mesh partition by moving a small number of mesh regions to neighboring parts. This is done in a manner that reduces vertex imbalance without creating large changes in the region balance, thus yielding a partition that produces better scalability of the entire finite element solution process. 2) HPS, which at first performs graph-based partitioning to distribute the mesh among certain number of parts (say, 99% of total), and then splits (1% of) parts that are heavily loaded with vertices so that roughly half of the mesh entities can be migrated from the heavy parts to the (1% of) empty parts, thereby eliminating the peak vertex imbalance while maintaining a good element balance. Example results showing the effectiveness of performing these processes are given in Section 4.2.

4.1 Algorithm of MCOPI

The amount of work to iteratively solve the linear algebraic system is proportional to the number of degrees of freedom. In the context of the current work, linear finite elements are used, where the mesh vertices are the holders of the degrees of freedom which need to be balanced on each part of the partition. Graph based partitioners produce well balanced partitions in terms of mesh regions without explicitly considering the balance of the number of mesh vertices, the resulting partition may have substantial vertex imbalance. The mesh vertex imbalance can be as high as 35.7% (see Section 3.5.2.2) when the number of mesh entities per part is only a few thousand. This translates to matrix vector products and vector norms (the two key work components of an iterative solve) with 35.7% more work which dramatically decreases the scalability of the analysis code. Two algorithms to reduce the vertex imbalance are given in the next two sub-sections. Refer Section 3.1 for the notations used in the algorithm description.

4.1.1 LIIPBMod

LIIPBMod locally migrates small numbers of mesh regions (or part-objects) from parts with relatively more vertices to neighboring parts which are relatively lightly loaded. On the heavily loaded part, the mesh vertices on the part boundary are traversed and the ones bounding a small number of elements are identified. If the neighboring part is lightly loaded, the whole "cavity" (all the adjacent elements of the picked vertex) is migrated to the neighboring part. By this minor inter-part boundary adjustment, the vertex imbalance is improved while only modestly perturbing the good element balance. This procedure may need to be repeated for several iterations to achieve the desired improvement to the vertex balance. A similar idea has been used in [48] to iteratively reduce the edge cut in the graph based partitioning by locally moving sets of graph nodes (*helpful sets*) between parts. Algorithm 5 presents the pseudo code of LIIPBMod. Figure 4.1 explains the concept using three, distinct 2-D examples for clarity. The procedure has been implemented for 3-D meshes. In each case, P_0 is the part which has extra vertices that need to move to the neighboring part and P_1 is the neighbor capable of accepting more mesh

entities (only partial meshes are shown on each part). The vertex M_1^0 has a small number of adjacent elements on P_0, which is 1, 2 and 3 respectively in Sub-figures (I)-(III) of Figure 4.1(a). In each case, all the triangles $(M_i^3,\ i = 1, 2, 3)$ adjacent to vertex M_1^0 are moved from P_0 to P_1, hence the number of vertices on P_0 is decreased by one while raising the element count on P_1 by 1, 2 and 3, respectively, in cases I, II and III. Partitions after LIIPBMod algorithm are shown in Figure 4.1(b). Moving mesh vertices bounding a small number of adjacent elements not only reduces the vertex imbalance but also usually decreases the total number of vertices on part boundaries which in turn decreases the total inter part communication during the iterative solve. Once the list of mesh entities that need to be migrated and their destination parts are determined, the algorithm gives the information to FMDB which handles the mesh migration in a six step procedure described below, see [72, 78] for the details of the procedure and related terminologies.

Step 1: Collect a set of entities to be updated by migration based on $POtoMove$. Each part maintains the separate set $entsToUpdt$ with different contents.

Step 2: Determine the residence part IDs $\mathscr{P}$ of entities in $entsToUpdt$ according to the residence part equation.

Step 3: Refresh the partition classification and collect entities to remove.

Step 4: Exchange entities from dimension 0 to 3, create entities on the destination part and update the remote copies of the entities created on the destination part.

Step 5: Remove unnecessary entities collected in Step 3 which will be no longer used.

Step 6: Update the ownership of the partition model entities.

The whole process is done in parallel with the data distributed among the parts. The vertices on inter-part boundaries with only one remote copy are considered to avoid the possibility to create complex or tortuous part boundaries. Figure

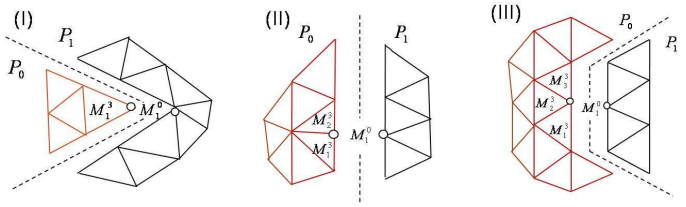
Algorithm 5 Pseudo code for LIIPBMod

```
1: tol_vtx and maxiter are application defined parameters
2: N_0max = maximum N_0 and N_0avg = average N_0 of the partition
3: N_0accept = N_0avg * tol_vtx
4: niter = 0
5: while N_0max > N_0accept && niter < maxiter do
6:    niter++
7:    on a heavily loaded part based on mesh vertices, nMarkedNodes = 0
      nNodesNeedToMove = N_0 - N_0accept
      VIter = {M{M^0}} on part boundary
8:    while M_i^0 = VIter → next && nMarkedNodes < nNodesNeedToMove do
9:       if |{M_i^0{M^3}}| is small (e.g., <= 5)&& M_i^0 has only one remote copy then
10:         R = part ID of the neighbor
11:         if R is not heavily loaded then
12:            for (j = 0; j < |{M_i^0{M^3}}|; j++) do
13:               Insert(M_i^0{M^3}_j, R) to map POtoMove
14:               nMarkednodes++
15:            end for
16:         end if
17:      end if
18:   end while
19:   if POtoMove is not empty then
20:      call Migration(M, POtoMove)
21:      update N_i information on each part, update N_0max
22:   else
23:      break
24:   end if
25: end while
```

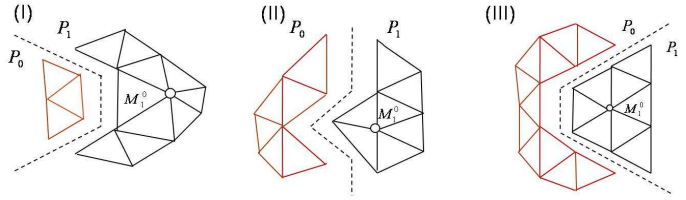
4.2 is one such example. P_1 has elements to give up and M_1^0 is on part boundary which has three remote copies, but has a low number of bounding elements ($|\{M_i^0\{M^3\}\}| = 1$). If P_3 is the only neighbor capable of receiving mesh entities. Migration of M_1^3 from P_1 to P_3 makes P_3 part boundary tortuous (see Figure 4.2(b)). The migration is avoided in such a situation.

4.1.2 HPS

Our studies of the mesh partitions given by a graph-based partitioner show that the percentage of heavily loaded parts (e.g., ones with more than 10% imbalance) is

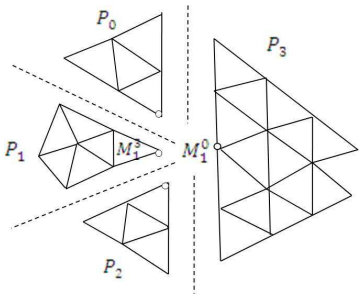


(a) Partition before LIIPBMod algorithm

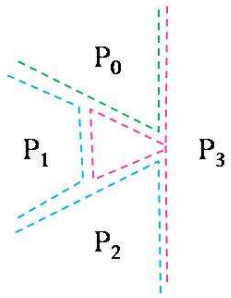


(b) Partition after LIIPBMod algorithm

Figure 4.1: Partition before and after LIIPBMod, dashed lines are inter-part boundaries.



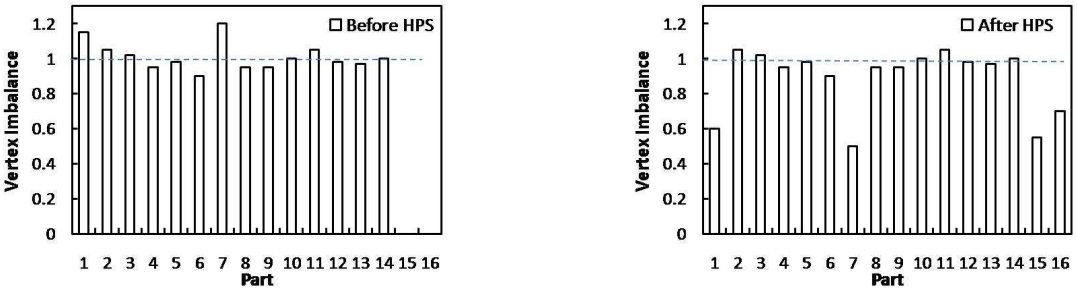
(a) Partition before migration



(b) Inter-part boundary after migration

Figure 4.2: An example of the migration of mesh entities that causes bad partition boundary.

usually less than 1%. The idea of HPS is to break the process into two steps. If the desired number of parts is $numP$, then in the first step the mesh is distributed to, say, $99\% \times numP$ parts by using a graph-based partitioner, leaving the remaining $1\% \times numP$ parts empty. Then, the $1\% \times numP$ parts with the highest vertex load are split into two parts (i.e., roughly half of the mesh entities are migrated from them to one of the empty parts). The splitting makes the heavily loaded parts become lightly loaded. Since the peak of the imbalance determines the scalability, HPS knocks down the peak and hence, improves the performance. Figure 4.3 illustrates this algorithm using a 16 part example. The vertex imbalance on part 1 and part 7 is very high whereas part 15 and 16 are empty (see Figure 4.3(a)). HPS algorithm migrates mesh entities from part 1 to 15 and from 7 to 16 (see Figure 4.3(b)). The original heavily loaded parts (1 and 7) become lightly loaded and the overall load balance is improved. If desired, this partition can be further improved by applying LIIPBMod. Indeed HPS is most useful when LIIPBMod has stagnated, for example, when the most heavily loaded parts are surrounded by other heavily loaded parts and in these scenarios, introducing a very lightly loaded part into this neighborhood of heavily loaded parts allows several neighbors to further improve.



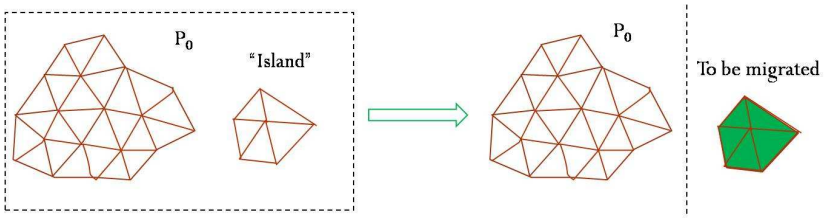
(a) Vertex imbalance ratio before HPS

(b) Vertex imbalance ratio after HPS

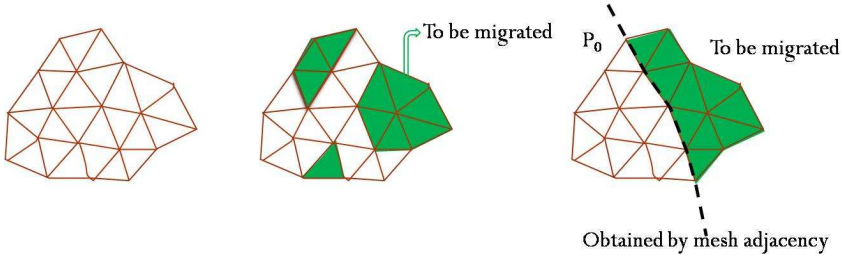
Figure 4.3: Algorithm of HPS

It is worth mentioning that the mesh entities to be migrated from heavily loaded parts to the empty parts in HPS algorithm need to be carefully selected to avoid "broken" inter-part boundaries. Figure 4.4 explains two general rules followed by HPS algorithm. In Figure 4.4(a), the heavily loaded parts have disjoint geome-

tries (it is a common case in the graph based partitioning), one piece ("island") will be migrated to the empty part. Algorithm 6 gives a simple procedure to detect disconnected parts. In Figure 4.4(b), there are no disjoint portions on the heavily loaded parts, then roughly half of mesh entities are migrated to the empty parts, however they are not selected randomly (the middle one in Figure 4.4(b)) since this migration creates bad inter-part boundaries, which is not preferred. Instead mesh adjacency information is used to pick up the mesh entities to be migrated in HPS algorithm (right one in Figure 4.4(b)).



(a) Mesh entities to be migrated if disjoint geometries exist in the heavily loaded parts



(b) Mesh entities to be migrated determined by mesh adjacencies

Figure 4.4: Rules to select mesh entities to be migrated from heavily loaded parts to empty parts.

4.2 Results and discussions

In this section, two geometric models are used to study the partition improvement by applying the MCOPI algorithms. In the first example, the graph partitioner ParMetis is used on a bifurcation pipe model, shown in Figure 4.5. The other application is on an abdominal aorta aneurysm (AAA) model (see Figure 3.6)

Algorithm 6 Procedure to detect disconnected parts

take M_1^0 from the set $\{M\{M^0\}\}$, $numVtx = 0$
push M_1^0 back to the list $vList$ and tag it to be in the $vList$
while $vList$ is not empty **do**
  pop out the first vertex M_k^0 from $vList$
  $numVtx++$
  for $(i = 0; i < |\{M_k^0\{M^1\}\}|, i++)$ **do**
    M_j^0 is the other vertex of mesh edge $M_k^0\{M^1\}_i$
    if (M_j^0 is not yet labeled as in $vList$) **then**
      push M_j^0 back to $vList$
    end if
  end for
end while
if $(numVtx == N_0)$ **then**
  no disconnected parts
else
  disconnected parts exist
end if

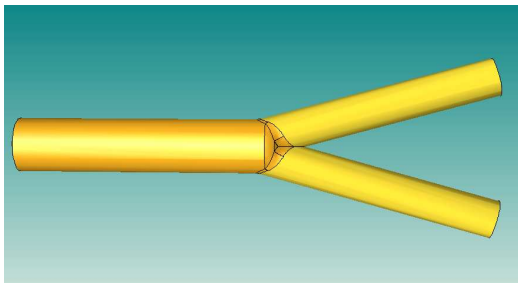


Figure 4.5: Geometry model of a bifurcation pipe.

and Zoltan PHG is utilized on this case to obtain different partitions. MCOPI is applied to the partitions produced by single compute-object partitions (ParMetis and PHG) on these two geometric models to demonstrate the vertex balance improvement. In Section 4.2.1, the region and vertex imbalance ratios by utilization of graph/hypergraph partition on the bifurcation pipe and the AAA model are presented. In both cases, we observe that the vertex imbalance increases when a fixed-size problem is distributed to a large number of parts (to allow compression in time to solution). Section 4.2.2 studies the vertex imbalance improvement by applying the two MCOPI algorithms, LIIPBMod and HPS. Section 4.2.3 presents

the acceleration of the finite element analysis (FEA) with the improved partition to demonstrate the effectiveness of the algorithms.

4.2.1 Graph/hypergraph-based partitions

To create partitions, the mesh database (FMDB) is interfaced with Zoltan library [22], which provides a suite of graph/hypergraph partitioning schemes such as ones available in ParMetis [39, 41], PHG [16], PT-SCOTCH [58], etc. Without loss of generality, ParMetis and PHG partition are utilized in this section on the bifurcation pipe model and the AAA model respectively to study the load balance when a fixed-size problem is partitioned to different numbers of parts.

4.2.1.1 ParMetis on a bifurcation pipe model

In the first example a graph based partitioner, ParMetis, is used though Zoltan's interface to (re-)partition the mesh. This case considers a uniform mesh with roughly 42 million tetrahedral elements on a bifurcation pipe model. Table 4.1 provides the average number of mesh regions and vertices for different partitions along with the imbalance ratios of regions and vertices. A wide range of parts (with an associated wide range of average number of elements per part) is considered. Specifically, the average number of elements varies from as high as about one hundred thousand (required for large simulations) on a partition with 128 parts to as few as one thousand (useful for time critical runs) on a partition with 32,768 parts. When a fixed-size problem is spread to more and more parts, the mesh regions are still well balanced (within 3% region imbalance) indicating that Zoltan has accomplished its objective. However, the vertex imbalance ratio increases which adversely affects the scalability of the flow solver (equation solution phase). When the mesh is partitioned to 16,384 parts, the average number of mesh regions per part is a bit more than two thousand and the vertex imbalance goes up to 14.5%. The imbalance ratio rises to 16.4% when the mesh is distributed to 32,768 parts. The specific part with the maximum number of vertices has 16.4% more work to do than the average in the equation solution phase, which slows down all the other processors. It is also worth noting that the average number of vertices per part does not decrease at the same rate as the regions when a fixed-size problem is spread into more and

more parts due to an increase in mesh vertices shared at inter-part boundaries. The tests are performed on Kraken (Cray XT5) at NICS, using 512 processing cores taking the advantage of FMDB's ability of handling multiple parts on each processor as discussed in Section 3.5. This is a fixed-size problem, where using one part per processor is possible. Without multiple parts per processor, one would need 32,768 processing cores for the case with 32,768 parts, which is not necessary (from a memory perspective) for a problem at this size and not efficient either.

num of parts	avg elem/core	elem imb.	avg vtxs/core	vtx imb.
512	82,371	1.023	16,681	1.0496
1,024	41,186	1.024	8,570	1.052
2,048	20,593	1.021	4,437	1.053
4,096	10,296	1.019	2,315	1.090
8,192	5,148	1.019	1,219	1.084
16,384	2,574	1.026	650	1.145
32,768	1,287	1.024	348	1.164

Table 4.1: Average number of regions and vertices on each part along with the region imbalance and vertex imbalance ratios of different partitions on a bifurcation pipe model with 42M elements.

4.2.1.2 Zoltan PHG on an AAA model

In this section, Zoltan PHG is used as the partitioner on an AAA model (see Figure 3.6) with a 1.07 billion non-uniform, anisotropic, tetrahedral element mesh created by applying several cycles of mesh adaptation. The mesh is partitioned to different number of parts, range from 4,096 up to 163,840. See Table 3.3 in Section 3.5.2.2 for the average number of regions and vertices on each part along with the region and vertex imbalance ratio. The regions are seen to be balanced well (within 6%) on partitions with up to 163,840 part, since regions are defined as graph nodes, which are partition objects. However, the vertex imbalance is getting worse when the mesh is distributed to more and more parts, which is consistent with the trend on the bifurcation pipe model utilizing ParMetis partition in the previous section. A 21.17% vertex imbalance is observed on a 98,304 part partition, which indicates 21.17% more work to do on the heavily loaded part (with the largest number of

vertices) in the equation solution phase of the FEA. Again, due to the increased number of mesh vertices at inter-part boundaries, the average number of vertices per part does not decrease at the same rate as the regions when it is spread into more and more parts, which is consistent with the bifurcation pipe case as well. In this test case, we have applied *local partitioning* (see Section 3.5.1.1). The partition starts with a well balanced (2.49% region imbalance) mesh on 2,048 parts and each part splits equally to obtain the different partitions (e.g., each part splits to 64 parts to achieve the 131,072 part partition). The vertex imbalance rises to 35.72% on the 163,840 part partition where each of the 2048 parts split to 80 parts. The quality of the partition is better if the hierarchic partition starts with a well balanced 4,096 part partition and each part split to 40 parts, which gives 163,840 parts in total. The imbalance information is given in the last row of Table 3.3 and this is the partition used in the scaling study in Section 3.5.2.2.

4.2.2 Balance improvement by MCOPI algorithms

In this section, we present the effectiveness of the MCOPI algorithms by applying them on the partitions produced by graph/hypergraph based partitioners.

4.2.2.1 LIIPBMod

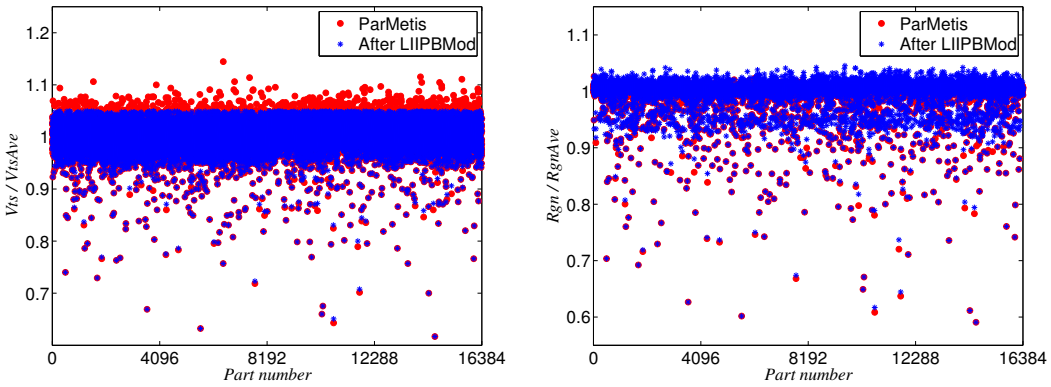
The first MCOPI algorithm, LIIPBMod discussed in Section 4.1.1 locally migrates a small number of mesh entities from heavily loaded parts to relatively lightly loaded parts to reduce the vertex imbalance without perturbing much of the region balance. In the first test, the LIIPBMod algorithm is applied to the partitions produced by ParMetis on the bifurcation pipe model with a 42 million element mesh. Table 4.2 compares the region and vertex imbalance ratios for each partition before and after using the LIIPBMod algorithm along with the time usage of ParMetis partition and LIIPBMod algorithm. The number of iterations LIIPBMod needed to improve the vertex imbalance for each partition is included in the table as well. In this study, the tol_vtx is set up to 1.05, and $maxiter$ is set up to 10 (see Algorithm 5), both of which are parameters set up by the application. In a FEA case where the equation solution dominants, the scalability of the FEA is more sensitive to the vertex balance, while the region balance is less important, it is optimal to set up

$42M$ elem.	elem imbal.		vtx. imbal.		Time usage (sec.)		Iter.
# parts	ParMetis	LIIPBMod	ParMetis	LIIPBMod	ParMetis	LIIPBMod	
512	1.0226	-	1.0496	-	21.0	-	-
1,024	1.0240	1.0245	1.0516	1.0488	24.6	0.6	1
2,048	1.0208	1.0223	1.0527	1.0496	26.3	0.3	1
4,096	1.0218	1.0335	1.0904	1.0493	26.1	0.7	2
8,192	1.0186	1.0336	1.0840	1.0487	26.6	1.2	3
16,384	1.0264	1.0454	1.1446	1.0492	28.8	2.2	3
32,768	1.0241	1.0653	1.1642	1.0578	37.4	18.5	9

Table 4.2: The comparison of region and vertex imbalance ratios before and after using LIIPBMod algorithm on a bifurcation pipe model with a $42M$ element mesh along with the time usage of ParMetis partitioner and LIIPBMod algorithm.

a tight tolerance (small number) for vertex imbalance. On the other hand, in the opposite case where the equation formation dominates, the scalability of the FEA is more sensitive to the region balance and a loose tolerance should be set up for vertex imbalance in the LIIPBMod algorithm.

The 512 part partition has 4.96% vertex imbalance, which is smaller than tol_vtx set up in the test. LIIPBMod algorithm does not change the partition. In the partitions with 1,024 to 16,384 parts the vertex imbalance exceeds 5% (14.46% in the 16,384 part partition) and LIIPBMod is able to reduce the vertex imbalance to 5% with only couple of iterations of LIIPBMod (last column of Table 4.2) without significant perturbation to the region balance. For example, LIIPBMod reduces the vertex imbalance from 14.46% to 4.92% (9.54% improvement) on the 16,384 part partition, while increasing region imbalance slightly from 2.64% to 4.54% (1.9% penalty). Figure 4.6 demonstrates the normalized number of vertices and regions on each part of the 16,384 part partition before and after the LIIPBMod algorithm. The time usage of LIIPBMod is much smaller compared to the time usage of ParMetis (column 6 and 7 of Table 4.2), which itself is negligible compared to the FEA (in the order of hours for the problem at this size). In the 32,768 part partition, LIIPBMod takes more time (19 seconds), which is still smaller compared to graph partition, since more iterations are performed in the LIIPBMod procedure. With the modest impact on region imbalance (1.024% to 1.065%), the algorithm reduces the vertex imbalance by more than 10% (from 16.4% to 5.8%).



(a) Normalized number of vertices on each part before and after LIIPBMod

(b) Normalized number of regions on each part before and after LIIPBMod

Figure 4.6: Comparison of normalized number of regions and vertices on each part of a 16,384 part mesh before and after LIIPBMod on a bifurcation pipe model.

The time usage of graph/hypergraph partition and LIIPBMod algorithm represented in this work includes the time usage of the mesh entity migration (the six steps discussed in Section 4.1.1). These tests are performed on Kraken (Cray XT5) at NICS using 512 processing cores.

In the second test, LIIPBMod is applied on the partitions produced by Zoltan PHG partition on an AAA model with a 1.07 billion element, anisotropic mesh. The same parameters are used as the previous test on the bifurcation pipe model (i.e., $tol_vtx = 1.05$ and $maxiter = 10$). The tests are performed on Kraken at NICS, using 2048 processing cores (4096 cores are used for case of "4k to 160k"). Table 4.3 compares the region and vertex imbalance ratio before and after LIIPBMod. It also contains the time usage of Zoltan PHG and LIIPBMod, again this includes the time usage of mesh entity migration. LIIPBmod reduces the vertex imbalance for all these 7 cases (from 4,096 to 131,072 parts) as well as the "4k to 160k" partition down to 5% (5th column in Table 4.3) with modest impact on region balance and it reduces the vertex imbalance dramatically (up to 16.2%) for the partitions with a high number of parts, which is important for time critical simulations. Figure 4.7 demonstrates the normalized vertices on each part of a 163,840 part partition ("4k to 160k" case) on the AAA model. The partition with 163,840 parts obtained

1.07B elem.	elem imbal.		vtx. imbal.		Time usage (sec.)		Num Iter.
# parts	PHG	LIIPBMod	PHG	LIIPBMod	PHG	LIIPBMod	LIIPBMod
4,096	1.0549	1.0583	1.0736	1.0499	281.1	4.7	3
8,192	1.0549	1.0616	1.0841	1.0499	307.3	9.1	2
16,384	1.0549	1.0704	1.0990	1.0499	325.6	9.9	4
32,768	1.0549	1.0831	1.1210	1.0499	343.6	11.4	5
65,536	1.0549	1.0972	1.1719	1.0497	363.2	17.6	4
98,304	1.0549	1.1070	1.2117	1.0499	376.8	25.7	4
131,072	1.0549	1.1092	1.1782	1.0497	398.4	37.3	4
163,840	1.0549	1.1189	1.3572	1.1884	419.4	97.1	10
4k to 160k	1.0554	1.1182	1.1952	1.0497	138.1	28.4	4

Table 4.3: The comparison of region and vertex imbalance ratios before and after using LIIPBMod algorithm on a 1.07B element mesh on an AAA model along with time usage of partition and the number of iterations.

by splitting each of the 2,048 parts to 80 parts, still has 18.84% vertex imbalance after 10 iterations of LIIPBMod procedures, since there are no candidate vertices (ones with a small number of adjacent regions) to be migrated from the heavily loaded parts to their neighbors or the heavily loaded parts are surrounded by heavily loaded neighbors. Another alternative, HPS, is needed in this situation which can be combined with LIIPBMod algorithm to improve the partition. Time usage of LIIPBMod (7th column of Table 4.3) is small compared to that of hypergraph based partition, and is negligible compared to the FEA.

Furthermore, by carefully selecting the mesh entities to be migrated from heavily loaded parts to their neighbors, LIIPBMod reduces the total number of mesh vertices of a partition (summation of mesh vertices on each part) by decreasing the number of shared mesh entities on inter-part boundaries. For example, the total number of vertices of the partition with 131,072 parts is 250.1 million, while this number is reduced to 245.7 million after LIIPBMod (1.7% fewer). In the "4k to 160k" partition, LIIPBMod reduces the total number of vertices from 255.7 million to 250.1 million, which is 2.2% fewer. This will reduce the inter-part communication during the equation solution.

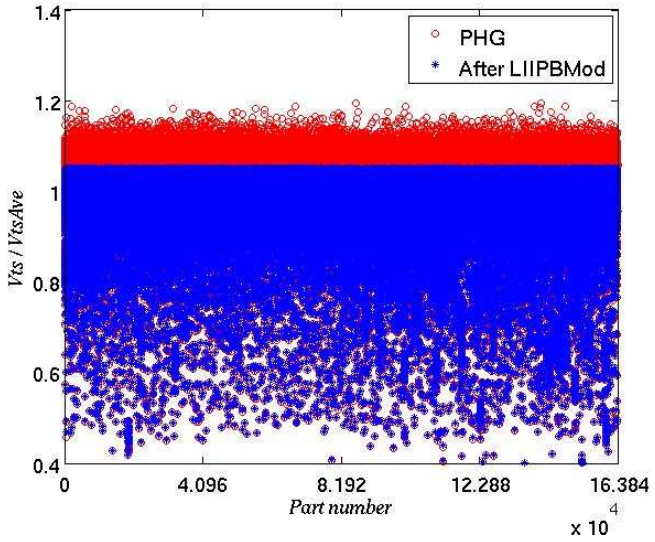
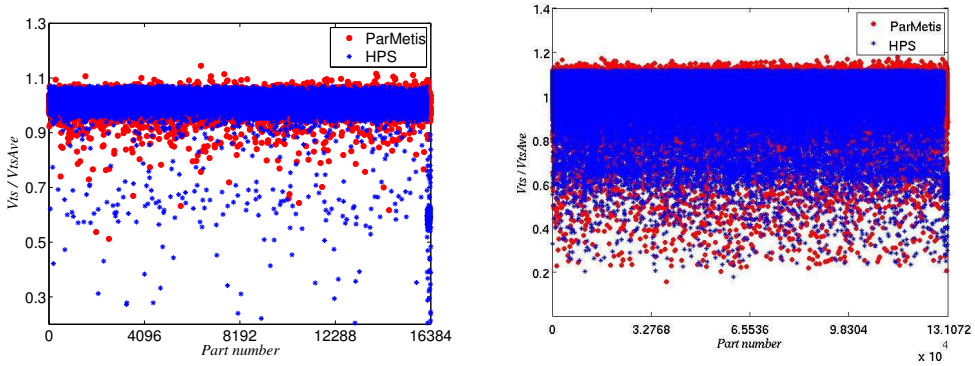


Figure 4.7: Comparison of normalized number of vertices on each part of a 163,840 part partition before and after LIIPBMod on an AAA model.

4.2.2.2 HPS

The second MCOPI algorithm, HPS discussed in Section 4.1.2 splits heavily loaded parts to improve the load balance. In the first test, HPS is applied to the partition produced by graph-based partitioner on a bifurcation pipe model. When the 42 million element mesh is partitioned to 16,384 parts by ParMetis, the region imbalance is 2.6% with 14.5% vertex imbalance. Among the 16,384 parts, only 8 parts have more than 10% vertex imbalance while 98 parts have more than 7% vertex imbalance. Since the number of parts which are heavily loaded is small (i.e., $98/16384 < 1\%$), HPS first distributes the mesh to 16,224 (about $99\% \times 16,384$) parts and leave 160 (about $1\% \times 16,384$) parts empty. In the second step, the 160 most heavily loaded parts are split and the selected mesh entities are migrated from each of them to a corresponding empty part. Figure 4.8(a) compares the normalized number of vertices on each part of the 16,384 part partition by using a graph-based partitioner to the partition produced by HPS. The partition given by HPS has a 2.7% region imbalance with 7.1% vertex imbalance, while the partition given by ParMetis has 2.6% region imbalance with 14.5% vertex imbalance.

In the second test case, HPS is applied to the partition produced by Zoltan



(a) Partition with 16,384 parts on the bifurcation pipe model

(b) Partition with 131,072 parts on the AAA model

Figure 4.8: Comparison of normalized number of vertices on each part of the partition given by graph-based partitioner and HPS on the bifurcation pipe model and the AAA model

PHG on an AAA model. When the 1.07 billion element mesh is distributed on 131,072 parts, the region imbalance is 5.49% with 17.82% vertex imbalance. Among the 131,072 parts, there are 1,190 parts have more than 11% vertex imbalance, with 2,352 parts have more than 10% vertex imbalance. The number of parts with high vertex imbalance (more than 11%) is less than 1.0% (i.e., $1,190/131,072 < 1.0\%$). The mesh is first distributed in to 129,792 parts and leave the other 1,280 parts empty. In the second step, the 1,280 most heavily loaded parts are split and the mesh entities are migrated from them to the corresponding empty parts. Figure 4.8(b) compares the normalized number of vertices on each part on the 131,072 part partition produced by Zoltan PHG and HPS. HPS gives 11.8% vertex imbalance with 6.9% region imbalance. If we reduce the vertex imbalance more aggressively (required in the equation solution dominated applications), more empty parts are reserved in the first step, for example, $2 \times 1,280 = 2,560$ empty parts. However, HPS algorithm is most useful when LIIPBMod has stagnated. Future work will focus on combining LIIPBMod and HPS in an optimal way to improve the partition.

4.2.3 Performance improvement of the finite element analysis

In this section, we present the performance improvement of the finite element analysis (specifically, PHASTA) with the reduced vertex imbalance of the partitions.

4.2.3.1 LIIPBMod on an AAA model

We consider the FEA on an AAA model with a 1.07 billion element mesh as the first example. The scaling study is presented in Section 3.5.2.2. We observe a near-perfect strong scaling (95% or above) for equation formation stage all the way to the full system-scale. On the other hand, strong scaling for equation solution stage drops below 90% at 65,536 cores and onwards. The strong scaling for both work components together is at 90% or above till 65,536 cores and drops to 82% while using 163,840 cores or all the 40 racks of BG/P in Intrepid. The imbalance in mesh nodes goes up from 7% in the partition with 4,096 parts to 17% in the case of 65,536 parts; and is as high as 19.5% in the partition with 163,840 parts. The MCOPI techniques are designed to improve the partitions in order to obtain better balance for both mesh elements and vertices. The LIIPBMod algorithm reduces the vertex imbalance for each of the eight partitions to within 5% with modest impact on region balance (see Table 4.3). The FEA is accelerated due to the improved overall balance. Table 4.4 and Figure 4.9 provide the execution time and the updated scaling factors.

1.07B element		Eqn. form.		Eqn. soln.		Total	
# of core	avg elem/core	time	s_factor	time	s_factor	time	s_factor
4,096 (base)	261,600	388.90	1	455.48	1	844.38	1
8,192	103,800	194.62	1.00	233.27	0.98	427.89	0.99
16,384	65,400	98.31	1.00	118.74	0.94	217.05	0.97
32,768	32,700	49.27	0.99	60.59	0.94	109.87	0.96
65,536	16,350	25.48	0.95	33.17	0.86	58.65	0.91
98,304	10,900	17.12	0.95	21.94	0.87	58.29	0.90
131,072	8,175	12.79	0.95	16.89	0.84	29.68	0.89
4k to 160k	6,540	10.49	0.93	13.65	0.84	24.12	0.88

Table 4.4: Strong scaling results on an AAA model with O(10^9) elements up to 163,840 cores on Intrepid:IBM BG/P with LIIPBMod algorithm.

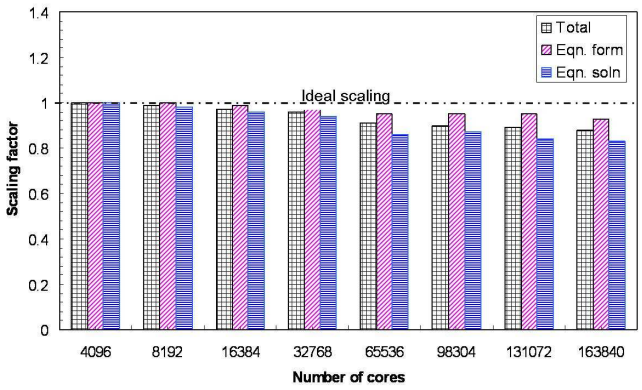


Figure 4.9: Strong scaling performance for an AAA model with O(10^9) regions up to 163,840 cores of Intrepid, IBM system with LIIPBMod.

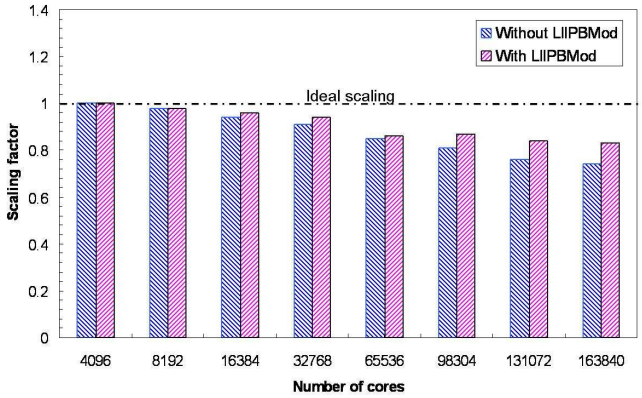
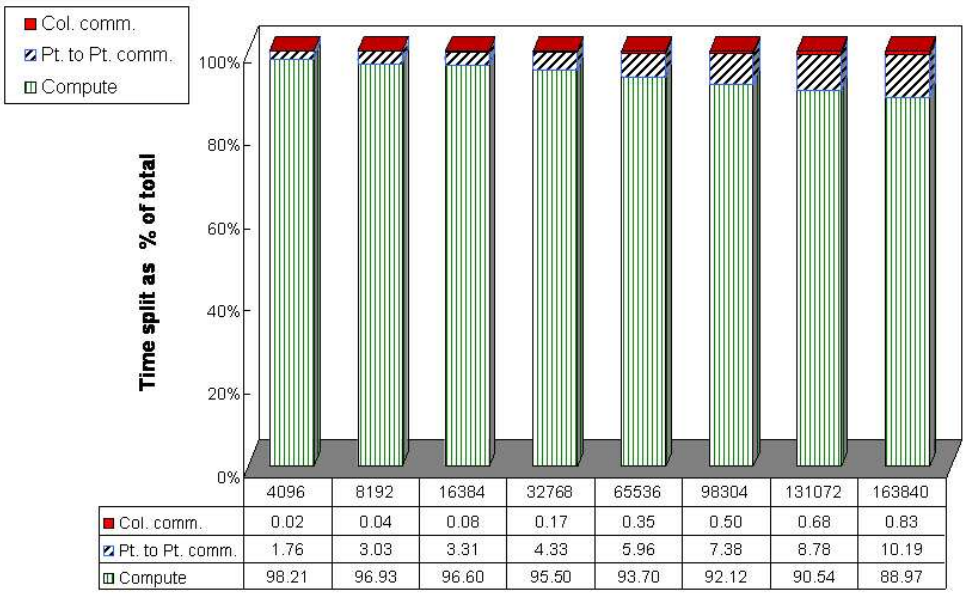


Figure 4.10: Comparison of scaling factors of equation solution with and without LIIPBMod for an AAA model with O(10^9) regions up to 163,840 cores of Intrepid, IBM system.

Figure 4.10 compares the scaling factors of equation solution part of the FEA before and after using LIIPBMod algorithm. The LIIPBMod algorithm improves the vertex imbalance, hence it accelerates the equation solution phase of the FEA (compared column 5 of Table 3.4 and Table 4.4). For example, on the 131,072 part partition, the vertex imbalance is reduced by 12.85%, the equation solution time is reduced from 18.65 second to 16.89 second (10.4%), while the scaling factor of equation solution is increased from 0.76 to 0.84. Due to the increased region imbalance, the equation formation is slowed down by 3.3%. However, the overall quality of the

partition is improved, therefore, the total time usage of the FEA is reduced by 4.5%. In many applications, the equation solution is more dominated than this test case, thus even greater time reductions can be expected. It is worth mentioning that, the FEA takes about 30 seconds to run 20 steps on 131,072 processing cores, which means 1.5 second per step. It usually takes thousands of time steps for one cardiac cycle (e.g., 5000 steps), which means $1.5 \times 5,000/3,600 = 2.1$ hour on 131,072 cores. Several cardiac cycles give converged solution, for example 5 cycles will need $2.1 \times 5 \times 131,072 = 1,376,256$ cpu hours. Due to the reduced vertex imbalance, we save 4.5% time in total, which is about 62 thousand cpu hours for the whole simulation. The LIIPBMod algorithm takes only 37.3 seconds on 2,048 processing cores for this 131,072 part partition, which is $37.3 \times 2048/3,600 = 21.1$ cpu hours. Compared to 62 thousand cpus hours, the time spend on LIIPBMod algorithm is negligible ($21.1/62,000 = 0.03\%$).

With LIIPBMod, the performance of the FEA is dramatically improved, especially of the equation solution phase. However, even the vertex imbalance is reduced to a reasonable level, we still observe a degradation of scaling factor at high core counts due to a high communication over computation ratio. The communication time starts to become substantial and thus detrimental to strong scaling. Sahni *et al.* [69] studied the communication time, both point-to-point and collective, of the case on an AAA model with $1B$ elements. It can be clearly seen graphically in Figure 4.11 that at high core counts the communication time starts to become substantial. This is expected in a strong scaling study because at high core counts the computational work on each processor-core becomes progressively lighter. These point-to-point communications (time usage is significantly higher than collective) occur in equation formation but occur at a much large number of times in the equation solution stage (to complete vectors ($\mathbf{p}$) during sparse matrix-vector products, see Algorithm 1). This explains why the equation formation stage achieves near-linear strong scaling while the equation solution is modestly impacted at large core counts no matter how balance the work load each processor has.



	4096	8192	16384	32768	65536	98304	131072	163840
Col. comm.	0.02	0.04	0.08	0.17	0.35	0.50	0.68	0.83
Pt. to Pt. comm.	1.76	3.03	3.31	4.33	5.96	7.38	8.78	10.19
Compute	98.21	96.93	96.60	95.50	93.70	92.12	90.54	88.97

Figure 4.11: Analysis of parallel performance in terms of computation and communication time for an AAA model with O(10^9) elements up to 163,840 cores of Intrepid, IBM BG/P system (communication time is composed of point-to-point and collective ones) [69].

In this chapter we have developed and tested two, multiple compute-object based partitioning improvement algorithms, LIIPBMod and HPS, both of which improve vertex balance. Compared to LIIPBMod, HSP has a better control of the region imbalance. Since the mesh is partitioned to 99% of the total parts, each part has 1% more mesh entities (since 1% of parts get nothing) and the splitting step does not increase the region imbalance. The HPS algorithm can substantially improve the vertex balance. With more, but still modest, impact on element balance, LIIPBMod makes better improvement to the vertex balance. It reduces the vertex imbalance ratio up to 16% (e.g., from 21.17% to 4.99%). Furthermore, with careful selection of part boundary entities to migrate from the heavily vertex loaded parts to their neighboring parts, LIIPBMod usually reduces the total number of vertices on part boundaries, which also decreases the total inter-part communications. Finally, the two can be combined in a complementary way to improve the partition more efficiently.

CHAPTER 5
DATA REORDERING ALGORITHM

A critical aspect for utilizing petascale resources is the scalability of the underlying numerical methods, but any performance improvements in on-processor calculations are desirable, particularly in situations with tight constraints on time to solution. This chapter presents a data reordering algorithm, which reorders both mesh vertices and regions to increase cache coherency during two primary stages of FEA, i.e., equation formation and equation solution. The effective use of the memory hierarchy on processing cores in turn leads to improved flop rate (or floating point operations per second).

In calculations based on unstructured meshes, the irregular mesh structures and connections make the data storage order a strong contributor to the effective use of the memory hierarchy. Since the storage order of the data is defined by the labeling of the entities in the mesh, it is desirable to have the mesh entities labeled such that those that interact, or have sequential access during calculations have labels that are as close to each other as possible. The historic method to address this issue is to employ the methods used in "bandwidth" and "wavefront" minimization such as RCM [20], GPS [30, 31, 33] and others. In previous work on edge-based unstructured mesh computations [19, 45, 46], the nodes are reordered using a bandwidth-minimization technique (RCM, recursive bisection, etc.) for optimal cache performance before subsequently renumbering the edges according to the nodes on each edge [45] or applying nodal reordering according to maximum connectivity to reduce indirect memory access [19, 46] specifically for edge-based methods. Oliker et al. [53, 54, 55] compared effects of data reordering strategies, RCM and self-avoiding walk (SAW) developed in [32] for two-dimensional unstructured meshes. In their tests on smaller cache architectures, SAW is about twice as fast as RCM, while there is little difference in parallel performance between RCM and SAW on larger cache architectures. It is worth noting that SAW is not a technique designed specifically for vertex reordering and incurs a higher construction

cost [55]. Another alternative approach is the permutation method of rows (and columns) of a matrix which is presented in [60] to increase cache reuse in sparse-matrix and dense-vector product, which is the kernel of equation solution of our finite element analysis. However this algorithm does not help the other primary stage of our finite element analysis, element formation and the overhead of the algorithm is generally high. Data reordering algorithms for general unstructured meshes are described in [17]. In the first step they apply existing classic approaches (such as bandwidth reduction, the greedy method, etc.,) to either vertices, edges, faces or regions. In the second step, they consistently renumber the other entities used in the solver. Mappings, such as a list of connected vertex pairs needs to be constructed to use the existing classic approaches in the first step, which can be expensive (see Section 5.1 for detailed discussions).

The examples considered in this work are based on linear C^0 finite elements. Here, the only mesh entities labeling required are the mesh regions and mesh vertices. The mesh vertices are the holders of the degrees of freedom that play the primary role in the equation solution stage. Thus, it is desirable to label vertices with coupled degrees of freedom such that they are close to each other. On the other hand, during element integration (equation formation) the ordering of both mesh regions and vertices is important since the process traverses the regions and accesses information related to the vertices of those regions. Note that in parallel finite element analysis, the ordering of mesh entities is required individually on parts making it easy to define. It is worth mentioning that the automatic unstructured mesh generators typically do not concern themselves with an ordering of entities since they typically do not know the details of the analysis algorithms. Furthermore, even if ordering was performed on the initial mesh, it would be disrupted by mesh adaptation procedures in their attempt to incrementally add and delete entities to achieve optimal resolution meshes.

By taking advantage of the fact that a mesh topology is maintained during the pre-processing of the finite element analysis input, an algorithm that effectively reorders the mesh regions and vertices based on topological adjacency has been implemented. See Section3.1 for the brief introduction to the topology concept.

5.1 Algorithm of data reordering

As discussed in references [17, 19, 45, 46, 53, 54, 55, 60], improving data locality in unstructured mesh calculations can have a substantial influence on the computational performance of the numerical operations performed. One can maximize the data locality by accounting for the locality of the entities in the mesh, and the non-zero entries in the resulting finite element matrices. Since the ordering of the operations and information in the finite element matrices are derived from the labeling given to the mesh entities, one wants to define a labeling where mesh entities considered to be nearby in space and related with each other have labels that are close in number.

The relationships between mesh entities dictate where terms are placed in sparse matrices and the order in which they are accessed during numerical operations. In the context of finite element analysis such information is easily identified based on the mesh adjacencies. For example, in the case of linear finite elements, the degrees of freedom are associated with vertices. The non-zero entities in the stiffness matrix for the equation associated with a given degree of freedom or a vertex are due to all of the vertices that are associated with elements that bound (sharing) the given vertex (i.e., $\{\{M_i^0\{M^3\}\}\{M^0\}\}$). Figure 5.1 shows an example, where a vertex, 29, has five elements around it and all the other vertices that bound

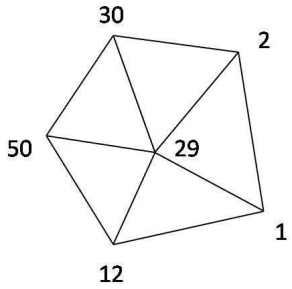


Figure 5.1: All the vertices that bound the five elements around a given vertex 29.

these five elements are 1, 2, 12, 30, and 50. The entities in the stiffness matrix due to vertex 29, are shown in the matrix A. Matrix-vector products with this matrix will enjoy sequential access of the matrix entries (since sparse storage is naturally in row order) but the second half of each product pair will require entries from the

product vector associated with these non-zeros as shown in the vector P. Note that the labels of mesh entities determine the entries in the vector that will be accessed one after another and thus, when the mesh entities labeled without consideration of order, the entries accessed subsequently may be placed far away in the vector.

$$A = \begin{pmatrix} & & & & & A_{1,29} & & \\ & & & & & A_{2,29} & & \\ & & & & & \vdots & & \\ & & & & & A_{12,29} & & \\ & & & & & \vdots & & \\ A_{29,1} & A_{29,2} & \dots & A_{29,12} & \dots & A_{29,29} & A_{29,30} & \dots A_{29,50} \\ & & & & & A_{30,29} & & \\ & & & & & \vdots & & \\ & & & & & A_{50,29} & & \end{pmatrix} \qquad P = \begin{pmatrix} P_1 \\ P_2 \\ \vdots \\ P_{12} \\ \vdots \\ P_{29} \\ P_{30} \\ \vdots \\ P_{50} \end{pmatrix}$$

In the case where one maintains a complete set of mesh adjacencies, the determination of the vertices that are related to a given vertex can be found via constant time operations (i.e., O(1) operations). The first step determines the mesh entities defining the finite elements around a given vertex while the second step collects the set of vertices that bound those mesh entities. In the example shown in Figure 5.2, it takes constant time to determine all the elements that M_1^0 bounds, which are elements $M_1^0\{M^2\}_1$, $M_1^0\{M^2\}_2$, $M_1^0\{M^2\}_3$, $M_1^0\{M^2\}_4$ and $M_1^0\{M^2\}_5$. For each of these five elements another constant time operation is required to determine all the bounding vertices, hence, constant time operations are used for collecting the set of vertices (combining M_2^0, M_3^0, M_4^0, M_5^0, and M_6^0,) related to a given vertex M_1^0. For other numerical operations, and/or other equation discretization methods, different relationships between mesh entities are important. In each of those cases, the underlying discretization method determines such relationships which can be easily defined in terms of appropriate mesh adjacencies.

Classic approaches to define an effective labeling of the desired mesh entities rely on graph-based procedures [20, 30, 31], where the nodes in the graph are associated to the mesh entities of interest and graph edges are defined between graph

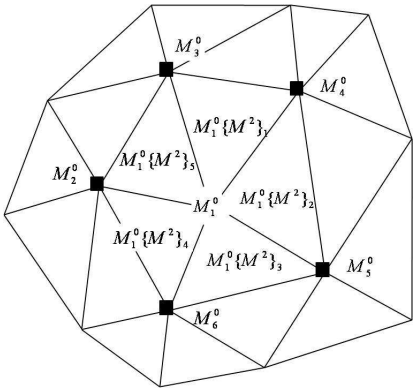


Figure 5.2: Constant time operations are used to determine the adjacent vertices for a given vertex.

nodes that are related to each other. Considering again the linear finite element example, there is a graph node for each mesh vertex whereas graph edges are defined between nodes for which there will be a non-zero term in the resulting stiffness matrix. The key idea in graph-based labeling procedures [20, 30, 31] is that once a graph node is assigned a label, all the unlabeled graph nodes connected to it by edges are labeled as soon as possible. The primary differences in the various algorithms are the criteria used to decide the node to be labeled first and the following graph traversing mechanisms used in assigning labels to the unlabeled connected nodes. Typically one of the most time consuming, and most data intensive, aspects in the execution of these procedures include the definition of the graph from a given mesh. In the case where the starting data is the list of vertices for each element, this process requires traversing the elements, where a graph node is defined for each vertex of an element (if one does not exist already) and also graph edges are defined between graph nodes associated with the vertices of the element.

An alternative, but equivalent, approach to the classic graph-based labeling has been defined and is applied in the present work. This approach takes advantage of the fact that maintaining the adjacencies of a set of mesh entities in a complete representation is functionally equivalent to having the graph structure for any desired mesh entity relationship. Equivalence is defined here in that once those mesh entities that will represent the graph nodes are known, the process of determining the graph edges between them due to mesh entity relationships can be performed in

a number of O(1) operations per graph node [11, 75]. Note that depending on the specific form of the complete mesh representation used, the number of operations can range from one to a few hundred, but each is of O(1) since it is not a function of the total number of entities in the mesh [12, 72]. The mesh entity labeling algorithm presented here employs this equivalence to label the desired mesh entities based on a single traversal through selected mesh adjacencies which is equivalent to a specific graph traversal such as those used in the more classic procedures. In fact, by carefully defining the traversal process and the labeling order, these methods have been found to define orderings that, on average, produce slightly better matrix profiles for finite element meshes of any polynomial order when compared to Reverse Cuthill McKee and GPS [11, 75].

5.1.1 Linear finite element data access

In the current work based on linear finite elements, the mesh entities to be labeled are the mesh regions ($\{M\{M^3\}\}$) that represent the finite elements and the mesh vertices ($\{M\{M^0\}\}$) since they are the mesh entities that hold the degrees of freedom in the matrix equation system. The relabeling of both the regions and vertices is important in the system formation phase which has four major steps, 1) localization of the data from global to element level, 2) computation of the residual (right-hand side) vector and tangent (left-hand side) matrix at the element level, 3) globalization of the residual vector from the element to the global level, and 4) filling of entries in a sparse-matrix. It is also important in the basic kernel of sparse-matrix-vector products during equation solution stage based on iterative schemes. The pseudo code of these primary steps are demonstrated in Algorithm 7 and the notation used in the pseudo code are defined in Table 5.1 (note that *ylocal* and *yglobal* describe the solution variable which is taken to be a scalar at each vertex in the pseudo code but in many interesting problems it is a vector which inflates ranks of all arrays involving solution or residual by one and tangent matrices by two (e.g., $LHS(nnz_tot)$ becomes $LHS(nnz_tot, nvar, nvar)$ while $Resglobal(N_3)$ becomes $Resglobal(N_3, nvar)$). In the process of localization and globalization (see Pseudo-codes 1 and 3 in Algorithm 7), the mesh regions are traversed, each vertex of

Algorithm 7 Pseudo code of localization, element level integrals, globalization, sparse-matrix filling and AP product

1. Pseudo code of Localization

for $(e = 1; e <= N_3; e++)$ **do**
  for $(j_{nen} = 1; j_{nen} <= nen; j_{nen}++)$ **do**
    $gdof = ien(e, j_{nen})$
    $ylocal(e, j_{nen}) = yglobal(gdof))$
    $Xlocal(e, j_{nen}, 1 : nsd) = Xglobal(gdof, 1 : nsd))$
  end for
end for

2. Pseudo code of Element Level Integrals

for $(e = 1; e <= N_3; e++)$ **do**
  compute $Reslocal(e, 1 : nen)$ **and** $xKeke(e, 1 : nen, 1 : nen)$ **based on** $Xlocal(e, 1 : nen, 1 : nsd)$ **and** $ylocal(e, 1 : nen)$
end for

3. Pseudo code of Globalization

$Resglobal(1 : N_0) = 0$
for $(e = 1; e <= N_3; e++)$ **do**
  for $(j_{nen} = 1; j_{nen} <= nen; j_{nen}++)$ **do**
    $gdof = ien(e, j_{nen})$
    $Resglobal(gdof) \mathrel{+}= Reslocal(e, j_{nen})$
  end for
end for

4. Pseudo code of Sparse-Matrix Filling

$LHS(1 : nnz_tot, 1 : nen, 1 : nen) = 0$
for $(e = 1; e <= N_3; e++)$ **do**
  for $(j_{nen} = 1; j_{nen} <= nen; j_{nen}++)$ **do**
    $gdof = ien(e, j_{nen})$
    $c = col(gdof),\ n = col(gdof + 1) - 1$
    for $(k_{nen} = 1; k_{nen} <= nen; k_{nen}++)$ **do**
      $b = ien(e, k_{nen})$
      nzg**=search for the match of** b **in array** row **confined to** $istart = c$ **and** $istop = n$
      $LHS(nzg) \mathrel{+}= xKebe(e, j_{nen}, k_{nen})$
    end for
  end for
end for

5. Pseudo code of Sparse AP product

$q(1 : N_0) = 0$
for $(i = 1; i <= N_0; i++)$ **do**
  for $(k = col(i); k <= col(i + 1) - 1; k++)$ **do**
    $j = row(k)$
    $q(i) \mathrel{+}= LHS(k) * P(j)$
  end for
end for

nen :	number of vertices per region
N_0 :	number of vertices in the mesh
N_3 :	number of regions in the mesh
nsd:	number of space dimensions
$gdof$:	global degree of freedom
e:	element number
j_{nen} & k_{nen}:	local node number
nnz_tot:	total number of non-zeros in stiffness matrix
$ien(N_3, nen)$:	connectivity array (e and j_{nen} yield $gdof$)
$ylocal(N_3, nen)$:	local variable array
$yglobal(N_0)$:	global variable array
$Xlocal((N_3, nen, nsd)$:	local coordinate
$Xglobal(N_0, nsd)$:	global coordinate
$Reslocal(N_3, nen)$:	local residual array
$Resglobal(N_0)$:	global residual array
$xKebe(N_3, nen, nen)$:	local stiffness matrix
$LHS(nnz_tot)$:	global sparse-matrix
$col(N_0 + 1)$ & $row(nnz_tot)$:	auxiliary arrays

Table 5.1: Definitions of symbols in the pseudo code.

the region is mapped to a global degree of freedom ($gdof$) (unique "on part" global numbering which range from 1 to the number of dof on a given part) which is in the same ordering with the mesh vertices. Relabeling both the regions and vertices increases the cache hits since the vertices of adjacent regions will have global degrees of freedom that are close. Then, for a given vertex $gdof$, if $yglobal(gdof)$ and $Xglobal(gdof)$ (in localization phase), or $Resglobal(gdof)$ (in globalization phase), are in the cache, the vertices in the adjacent regions which are processed next are likely to be in the cache resulting in cache hit. In the process of the sparse-matrix filling, a mapping from the vertices of each region to the global degrees of freedom is created as well. The position of this global degree of freedom in the sparse-matrix is then searched. Therefore vertex and region relabeling improves the cache performance in the process of sparse-matrix filling. However, relabeling has no effect on the element-level integral (see Pseudo-code 2 in Algorithm 7), in which the memory hierarchy is already used efficiently due to the localization of the data. Previous work [17] reports high performance improvement (over a factor of two) in the equation formation phase, since without a localization procedure, region and vertex ordering has more influence on the computation of the stiffness matrix and

residual vector.

In the solution phase of the finite element analysis, only the vertex relabeling is important. The sparse matrix-vector product (see Pseudo-code 5 in Algorithm 7) work is proportional to the number of global degree of freedom. The sparse storage structure naturally stores the LHS entries in the order they will be used (sequential by row) but proper vertex order ensures that the items which follow each other in the product vector p (list of j) are close in memory despite the indirect addressing of the sparse storage format. The last key kernel in the iterative solve is the vector dot-product but this is not discussed here since it naturally enjoys sequential access for any ordering.

5.1.2 Adjacency-based reordering algorithm

Algorithm 8 for element and nodal labeling given below considers degrees of freedom associated with mesh vertices, edges, faces and regions, and with the assumption that mesh regions are finite elements. For clarity, Figure 5.3 uses a 2D example to explain the procedures on setting mesh entity labels, but the current algorithm is developed and used for 3D unstructured meshes. In the 2D context, mesh faces are the elements, which need to be labeled. The left picture of Figure 5.3 illustrates the starting configuration of a 16 node, 5 face mesh case, and the right one shows the new labels of the mesh nodes and elements. Table 5.2 demonstrates the status of the queue and the nodes labeled on each step of the relabeling process. It is worth pointing that most work in Algorithm 8 deals with vertex reordering and it reorders the other entities naturally with small portion of effort.

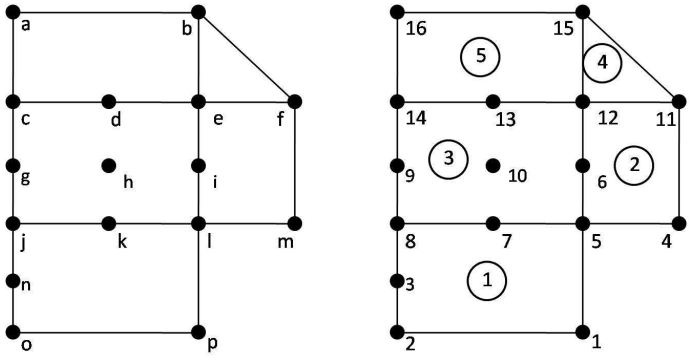


Figure 5.3: A 2D example of nodal and element labeling

Algorithm 8 Pseudo code for nodal and element labeling

getstart(), take M_1^0 from the set $\{M\{M^0\}\}$, $labelnode = N_0$; $labelelement = N_3$
push $M_1^0\{D\}_0$ back to the list *nodeList* and tag it to be in the *nodeList*
while *nodeList* is not empty **do**
  pop out the first node $M_k^d\{D\}_p$ from *nodeList*
  $M_k^d\{D\}_p \rightarrow setlabel(labelnode--)$
  if M_k^d is not a vertex **then**
    continue
  else
    for $(i = 0; i < |\{M_k^0\{M^1\}\}|, i++)$ **do**
      take edge $M_i^1 = M_k^0\{M^1\}_i$
      for $(j = 0; j < |\{M_i^1\{M^3\}\}|, j++)$ **do**
        take element $M_j^3 = M_i^1\{M^3\}_j$
        if M_j^3 is not labeled **then**
          $M_j^3 \rightarrow setlabel(labelelement--)$
        end if
        for $(p = 0; p < |\{M_j^3\{D\}\}|; p++)$ **do**
          if $M_j^3\{D\}_p$ is not labeled and $M_j^3\{D\}_p$ is not tagged to be in *nodeList* **then**
            push $M_j^3\{D\}_p$ back to *nodeList* and tag it to be in the *nodeList*
          end if
        end for
      end for
      for $(j = 0; j < |\{M_i^1\{M^2\}\}|, j++)$ **do**
        take face $M_j^2 = \{M_i^1\{M^2\}\}_j$
        for $(p = 0; p < |\{M_j^2\{D\}\}|; p++))$ **do**
          if $M_j^2\{D\}_p$ is not labeled and $M_j^2\{D\}_p$ is not tagged to be in *nodeList* **then**
            push $M_j^2\{D\}_p$ back to *nodeList* and tag it to be in the *nodeList*
          end if
        end for
      end for
      take the other vertex $M_l^0 = \{M_i^1\{M^0\}\}(M_k^0)$
      if $M_l^0\{D\}_0$ is not labeled and $M_l^0\{D\}_0$ is not tagged to be in *nodeList* **then**
        for $(p = 0; p < |\{M_i^1\{D\}\}|; p++)$ **do**
          if $M_i^1\{D\}_p$ is not labeled and $M_i^1\{D\}_p$ is not tagged to be in *nodeList* **then**
            push $M_i^1\{D\}_p$ back to *nodeList* and tag it to be in the *nodeList*
          end if
        end for
        push $M_l^0\{D\}_0$ back to *nodeList* and tag it to be in the list
      else
        for $(p = 0; p < |\{M_i^1\{D\}\}|; p++)$ **do**
          if $M_i^1\{D\}_p$ is not labeled and $M_i^1\{D\}_p$ is not tagged to be in *nodeList* **then**
            $M_i^1\{D\}_p \rightarrow setlabel(labelnode--)$
          end if
        end for
      end if
    end for
  end if
end while

Queue	In process	Node label assigned
a	-	start of process-none assigned yet
b,c	a	16
c,e,f	b	15
e,f,h,g,j	c	14,13 (d done since e already in queue)
f,h,g,j,i,l	e	12
h,g,j,i,l,m	f	11
g,j,i,l,m	h	10
j,i,l,m	g	9
i,l,m,n,o	j	8,7 (k done since l already in queue)
l,m,n,o	i	6
m,n,o,p	l	5
n,o,p	m	4
o,p	n	3
p	o	2
-	p	1

Table 5.2: The status of the queue and the nodes labeled on each pass through

5.2 Results and discussions

In this sub-section, two problem cases are used to study the performance improvement of the FEA solver (specifically PHASTA) by applying the new data reordering algorithm presented in Section 5.1. The first example involves a bifurcation pipe model which is shown in Figure 4.5. The other example includes non-uniform, anisotropic, tetrahedral meshes obtained by applying mesh adaptation [68] on an abdominal aorta aneurysm (AAA) model (see Figure 3.6). These two geometries and meshing approaches are used to study different aspects of the data ordering. In the first study, the relative impact of the current data reordering algorithm on each phase of FEA solver, including system formation (localization, globalization, element level integrals, and sparse-matrix filling) and system solution is presented in Section 5.2.1. In Section 5.2.2, an 8.8 million region mesh on the bifurcation pipe model is partitioned into different numbers of parts (from 128 parts up to 8192 parts), which allows a study of the effect of varying average work load per part in different partitions. These first two tests were performed on IBM Blue Gene/L at

Rensselaer's Computational Center for Nanotechnology Innovations (CCNI). A similar study on an AAA model with a 133.6 million region mesh is presented in Section 5.2.3. For this study, the computations are performed on various supercomputers, including IBM Blue Gene (BG/L and BG/P), Cray XT (XT3 and XT5), and Sun Constellation Cluster, to study the data reordering effect on various architectures. In Section 5.2.4, a hardware performance tool, CrayPat on Cray XT5 is used to predict the penalty due to cache misses, which confirms that the acceleration of FEA solver is due to the improved cache performance. In this test, the effect of data reordering algorithm is studied on cases with different load per part by fixing the number of processor cores and increasing the mesh size (i.e., the load per part changes from one case to another but in each case the load is balanced among parts). Note that one part per processor-core is considered in these examples (i.e., the number of parts equals the number of processor cores).

5.2.1 Relative effect of data reordering

In this section, the effect of data reordering algorithm on each primary phase of FEA solver is studied on an 8.8 million region mesh of the bifurcation pipe model (see Figure 4.5). A steady parabolic boundary condition is applied at the inlet of the bifurcation pipe model. The computation are performed for 100 time steps with 3 non-linear iterations per step using 128 CCNI BG/L cores. Table 5.3 provides the execution times for each primary phase of the finite element analysis, system formation (where localization and globalization, sparse-matrix filling, element level integrals are the main components) and system solution. Furthermore, these times are compared between three strategies: without any reordering, reordering of vertices only and reordering of both vertices and regions. The values in the parentheses are the normalized time usage of each phase of FEA solver (normalized by total time usage of the analysis without any data reordering, which is 5422.8 seconds). It is clearly shown in the table that reordering, of both the nodes and elements, improves FEA solver performance the most among these strategies, which takes 81.7% of the original analysis time (the one without using data reordering), while reordering of only nodes account for most of the improvement taking 83.6% of the original analysis

time. Reordering both vertices and regions simultaneously improves the time usage of localization and globalization (from 1.1% to 0.5%), while (as expected) reordering of vertices only does not improve performance of the localization and globalization steps. A similar observation can be made for the step of sparse-matrix filling. On the other hand, the data reordering algorithm has negligible effect on element-level integrals due to their use of localized data structures as mentioned in the previous section. Relabeling of vertices only improves the performance of the equation solution phase from 71.8% to 55.4%, but including region relabeling has negligible effect which is expected and also consistent with the discussion in Section 5.1. The relative improvement in each phase is obtained by comparing values in column 3 (reordering of vertices only) and 4 (reordering of vertices and elements) with column 2 (without data reordering). For example, relabeling of both vertices and elements improves localization and globalization by a factor of over two (61.5/27.9 = 2.2); a big percentage gain in these steps but only a small fraction in terms of the total time. Henceforth, overall performance improvements based on the total execution time of the analysis will be presented.

Time in seconds (in %)	Without data reordering	Reordering of vertices only	Reordering of vertices & elements
Localization & globalization	61.5 (1.1%)	67.2 (1.2%)	27.9 (0.5%)
Sparse matrix filling	163.8 (3.0%)	162.8 (3.1%)	132.2 (2.4%)
Element level integral	997.7 (18.4%)	998.6 (18.4%)	994.3 (18.3%)
Total of system formation	1520.4 (28.0%)	1524.1 (28.1%)	1429.9 (26.3%)
System solution	3894.9 (71.8%)	3005.7 (55.4%)	2992.8 (55.2%)
Total of analysis	5422.8 (100%)	4537.3 (83.6%)	4430.3 (81.7%)

Table 5.3: Data reordering effect on different phases of implicit FEA.

5.2.2 Effect of processor load (strong scaling)

In these runs, the bifurcation pipe model is considered but now the mesh is partitioned into different numbers of parts. This allows us to study the effect of data reordering with different mesh sizes or load per processor core. To create different mesh partitions, the mesh database (FMDB) is interfaced with Zoltan library [22] where the graph based partitioning package ParMETIS [41] is used. Table 5.4 lists

the average number of mesh regions and vertices per part for different partitions of an 8.8 million element uniformly refined tetrahedral mesh. A wide range of average number of elements per part is used, where it varies from as high as (about) hundred thousand (required for big simulations) on a partition with 128 parts to as few as several thousand (useful for time critical runs) on a partition with 8192 parts. It is worth noting that the average number of vertices per part does not decrease at the same rate as the regions when a fixed-size problem is spread into more and more parts. For example, 14020 is the average number of mesh vertices per part in the partition with 128 parts and it is 306 in the 8192 part partition; which is about 39.7% higher relatively (219 mesh vertices will be ideal based on the average of 14020 when going from 128 parts to 8192 parts). This occurs due to an increase in mesh vertices shared at inter-part boundaries. These tests are performed on IBM Blue Gene/L at the CCNI.

Number of parts	128	256	512	1024	2048	4096	8192
Avg. num. of rgns.	68856	34463	17231	8616	4308	2154	1077
Avg. num. of vtx.	14020	7262	3780	1984	1050	560	306

Table 5.4: Average number of regions and vertices per part of an 8.8 million region mesh of a bifurcation pipe model.

The performance improvement ratio obtained due to data reordering for each partition is shown in Figure 5.4. The time usage of FEA solver with and without using data reordering algorithm along with improvement is listed in Table 5.5. The algorithm has greater effect on heavily loaded processor cores, which can be easily observed in Figure 5.4. This happens because in the run with heavily loaded parts (i.e., run on a lower core count) relatively more time is spent accessing the memory compared to the light loaded computation (higher core count). Also the larger the data a given part must hold, the smaller the percentage of total data will be resident in cache, so higher cache misses are more likely when the data is randomly ordered and accessed. The data reordering algorithm attempts to produce a more uniform rate of accessing data from memory for all mesh sizes. This algorithm improves the FEA performance by 18.0% for the run on 128 parts, while it improves 9.3% on the 4096 part run and 2.3% on the 8192 part run. Though the improvement diminishes

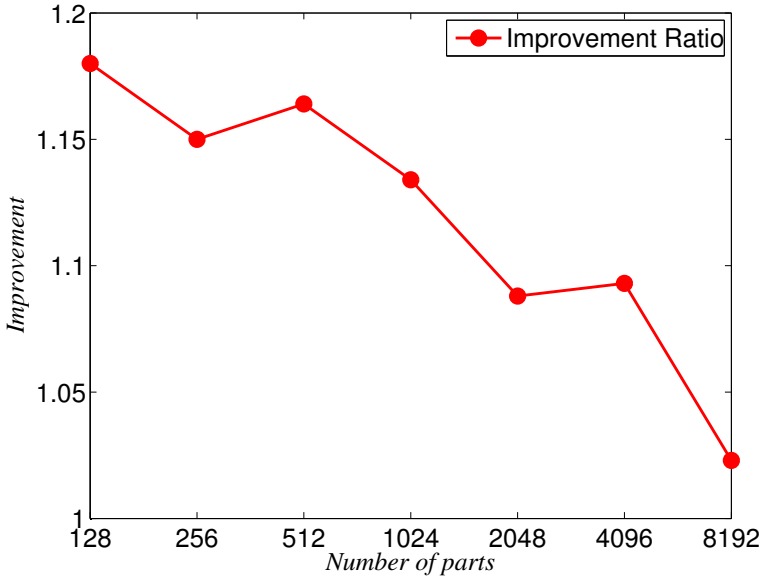


Figure 5.4: FEA solver performance improvement due to current data reordering algorithm on a bifurcation pipe model on CCNI's BG/L.

as the load is reduced, data reordering reduced the computational time for all the test cases.

Number of parts	128	256	512	1024	2048	4096	8192
Without reordering (sec.)	5422.8	2643.1	1316.3	679.0	351.7	201.3	136.5
With reordering (sec.)	4430.3	2246.6	1100.0	587.8	320.7	182.6	133.3
Improvement	18.0%	15.0%	16.4%	13.4%	8.8%	9.3%	2.3%

Table 5.5: Time usage of FEA solver on a bifurcation pipe model on CCNI's BG/L along with performance improvement.

5.2.3 Effect of different computer architectures

Another example considers a 133.6 million element non-uniform, anisotropic, tetrahedral mesh created by applying mesh adaptation [68] on an abdominal aorta aneurysm (AAA) model (see Figure 3.6). A Womersley profile [98] is applied at the inlet and rigid vessel walls with no-slip boundary condition are considered. At outlets a lumped parameter boundary condition [91] is applied. The computation is run for 20 steps and the time usage for different partitions on different supercomputers with and without using data reordering algorithm is measured, see Table 5.6. The

performance improvement ratio obtained by applying the current data reordering algorithm is shown in Figure 5.5.

Number of parts		1024	2048	4096	8192
CCNI-BGL IBM BG/L	Without (sec.)	492.96	246.50	123.88	61.89
	With (sec.)	376.68	191.70	98.88	50.94
	Improvement	23.6%	22.2%	20.2%	17.7%
Intrepid, ANL IBM BG/P	Without (sec.)	445.70	223.04	111.40	57.04
	With (sec.)	348.08	177.88	90.89	46.91
	Improvement	21.9%	20.2%	18.4%	17.7%
Kraken, NICS Cray XT5	Without (sec.)	133.27	59.66	31.23	/
	With (sec.)	118.99	56.12	30.74	/
	Improvement	10.7%	5.9%	1.6%	/
Ranger, TACC Sun Cluster	Without (sec.)	194.22	91.33	48.42	/
	With (sec.)	166.33	84.19	47.55	/
	Improvement	14.4%	7.8%	1.8%	/
Bigben, PSC Cray XT3	Without (sec.)	155.53	64.51	30.03	/
	With (sec.)	132.31	57.65	28.92	/
	Improvement	14.9%	10.6%	3.7%	/

Table 5.6: Time usage of FEA solver on an AAA model with and without using data reordering algorithm on different supercomputers along with performance improvement.

This result is consistent with the one shown in Section 5.2.2 in that the data reordering algorithm has more benefit in cases with heavily loaded processor cores (i.e, in runs on lower number of processors). It also suggests that the amount of performance gain that can be achieved by applying data reordering algorithm is dependent on the system architecture as well. In Table 5.7 we summarize the supercomputer systems considered in this study.

The data reordering algorithm has a more significant effect on the Blue Gene (up to 23.6% on BG/L and up to 21.9% on BG/P) due to a much smaller L2 cache, which is a prefetch buffer against L3. The L2 cache of Blue Gene (BG/L and BG/P) is only 2K bytes in size compared with 1 MB in the case of BigBen and 512KB in the case of Ranger and Kraken. The consequence of this dramatic difference in L2 cache size is that 4096 Ranger/Kraken processing cores have $4K * 0.5MB = 2GB$ L2 cache in total, but same number of processing cores on Blue Gene have only $4K * 2KB = 8MB$ L2 cache which is 1/256 of the other systems. This cache

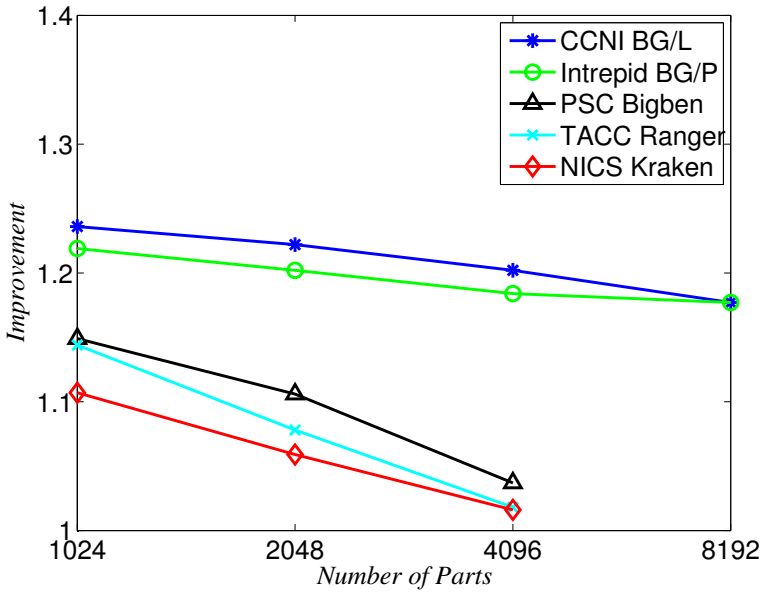


Figure 5.5: FEA solver performance improvement using current data reordering algorithm on an AAA model on various supercomputers including IBM Blue Gene (CCNI-BGL and Intrepid-BGP), Cray XT (Bigben-XT3 and Kraken-XT5) and Sun Constellation Cluster (Ranger).

size difference also explains why the performance improvement diminishes when reaching 4096 processing cores on Ranger, BigBen and Kraken, but still has over 20% performance improvement on Blue Gene. Because the L2 cache on the Blue Gene is prefetch driven, the reordering algorithm has a greater sustained positive impact on cache performance since it orders data items in such a way as to make it obvious to the prefetcher which data items to go after next.

5.2.4 Cache performance modeling

On the Cray XT5 (NICS Kraken), the hardware performance tool CrayPat is used to predict the penalty due to cache misses. In these runs, the effect of data reordering on the cases with different load per part is studied by fixing the number of processor cores and increasing the mesh size. Note this study is complementary to the strong scaling study (fixed-size problem distributed into different number of processor cores) discussed earlier. Hardware counters provide low-overhead access to a wealth of detailed performance information related to CPU's functional units,

System name	CCNI-BGL@RPI	Intrepid@ALCF	Bigben@PSC	Ranger@TACC	Kraken@NICS
System type	IBM BG/L	IBM BG/P	Cray XT3	Sun Const.	Cray XT5
Proc.type	PPC 700MHz	PPC 850MHz	AMD 2.6GHz	AMD 2.3GHz	AMD 2.3GHz
SPN	1	1	1	4	2
CPCN	2	4	2	16	8
CPS	2	4	2	4	4
L1 size	32KB/core	32KB/core	64KB/core	64K/core	64KB/core
L2 size	2KB/core	2KB/core	1MB/core (P)	512KB/core	512KB/core
L3 size	4MB/socket (S)	8MB/socket (S)	–	2MB/socket (S)	2MB/socket (S)

Table 5.7: Summary of systems considered in this study. SPN refers to sockets per node, CPCN to cores per compute node and CPS to cores per socket. The tests reported in this paper use all the cores on each node. S refers to shared cache on a socket among its cores while P refers to private cache on cores of a socket.

caches, main memory, etc. The hardware performance, specifically cache performance, is studied in this section. In this study four meshes of different sizes are used, where 8.8 million element mesh of a bifurcation pipe model (see Figure 4.5) is adapted to obtain three additional meshes of different sizes that have 22 million, 42 million and 92 million elements, respectively. Computation based on each mesh involves 5 time steps on 128 cores (16 nodes). Table 5.8 compares the time usage of FEA solver with and without using data reordering algorithm.

Mesh size (million)	8.8	22	42	92
Without (sec.)	69.80	196.72	444.08	1091.62
With (sec.)	64.34	153.71	307.41	654.46
Difference (sec.)	5.46	43.01	136.27	437.15

Table 5.8: Time usage of FEA solver on different meshes with and without using data reordering algorithm on NICS Kraken-XT5.

The NICS Kraken-XT5 system consists of compute nodes with 2.3GHz quad-core AMD Opteron processors (Barcelona). It has 64KB L1 cache per core for data (and same for instructions) with a 3 cycle latency, a 512KB L2 cache per core with a 12 cycle latency [92], and 2MB L3 cache shared among cores on a socket (or quad-core processor) with 38 cycle latency [6]. Note that the 38 cycle latency is a reported average. It turns out the L3 cache may have some additional latency for other reasons: (i) cache access between the four cores is doled out in a round-robin fashion, and (ii) FIFO buffers that sit in front of this cache in order to deal

with cores running at what may be vastly different clock speeds. Additionally, the speed of the Barcelona's L3 cache varies between 1.8 GHz to 2.0 GHz depending on processor model [7].

The total penalty difference due to changes in cache misses could be computed by a simplified model as:

$$(\Delta L1_DCM * L1_Latency + \Delta L2_DCM * L2_Latency + \Delta L3_CM * L3_Latency)/Clock_speed, \tag{5.1}$$

where $\Delta L1_DCM$ is the change in L1 data cache misses, $\Delta L2_DCM$ is the change in L2 data cache misses, and $\Delta L3_CM$ is the change in L3 cache misses. The CrayPat report of cache misses is listed in Table 5.9 and the penalty differences due to cache misses are computed for the cases with and without using the data reordering algorithm based on Eqn. 5.1. Note, L1_DCA in the table refers to L1 data cache access. The penalty differences due to cache misses (last column of Table 5.9) are close to the differences measured in the execution time of the analysis with and without using data reordering algorithm shown in the last row of Table 5.8. For the case with 92 million region mesh, the prediction of time difference due to cache miss penalty (464.13 sec.) is higher than the total difference observed in execution time (437.15 sec.) with and without data reordering. This is because the AMD Barcelona processor has an innovative prefetching algorithm that attempts to prefetch data in the L1 data cache and DRAM levels [92]. Due to such a data prefetching capability, some of the cache misses are hidden since the instruction pipeline will not stall on prefetched data that misses at any level in the cache. Thus, the time to service a cache miss is not incurred as a real execution time penalty since instruction processing continues during the processing of the cache miss. But at small mesh sizes, the amount of main memory used is much less and so there is not as much an opportunity to prefetch within a single iteration.

Additionally, Table 5.9 shows that the most dramatic improvement in memory hierarchy efficiency occurs in the L2 data cache. Here, it is observed that for the 42 million region mesh case, L2 data cache misses are 311% higher without reordering than with reordering (e.g. 5.3 billion without reordering vs. 1.7 billion with re-

ordering) and for the 92 million region mesh case this trend increases to 366% more cache misses when reordering is not used (e.g., 14.3 billion without reordering vs. 3.9 billion with reordering).

Mesh (mil.)	Without Data Reordering ($X10^9$)				With Data Reordering($X10^9$)				Penalty Diff (sec.)
	L1_DCA	L1_DCM	L2_DCM	L3_CM	L1_DCA	L1_DCM	L2_DCM	L3_CM	
8.8	32.9	3.1	0.5	1.8	32.9	2.5	0.3	1.6	5.13
22	80.4	9.2	1.9	5.32	80.6	6.2	0.9	3.4	40.85
42	152.7	18.9	5.3	13.5	153.1	11.9	1.7	7.9	120.43
92	332.0	41.1	14.3	39.2	333.4	25.8	3.9	15.6	464.13

Table 5.9: Comparison of cache access and cache misses for different meshes of a bifurcation model on NICS Kraken-XT5 with and without using data reordering.

As mentioned, these tests studied the data reordering effect on different load per core cases by fixing the number of processor cores and increasing the mesh size. The data in Table 5.9 is normalized by the lightest loaded case (8.8 million region mesh case), i.e., all the values are divided by the first row in Table 5.9. The normalized cache access and cache misses information is provided in Table 5.10.

Mesh	Without Data Reordering ($X10^9$)				With Data Reordering($X10^9$)			
	L1_DCA	L1_DCM	L2_DCM	L3_CM	L1_DCA	L1_DCM	L2_DCM	L3_CM
1.0(1.0)	1.0(1.0)	1.0(1.0)	1.0(1.0)	1.0(1.0)	1.0(1.0)	1.0(1.0)	1.0(1.0)	1.0(1.0)
2.5(1.0)	2.4(0.96)	3.0(1.2)	3.8(1.5)	3.0(1.2)	2.4(0.96)	2.5(1.0)	3.0(1.2)	2.1(0.84)
4.8(1.0)	4.6(0.96)	6.1(1.3)	10.6(2.2)	7.5(1.6)	4.7(0.98)	4.8(1.0)	5.7(1.2)	4.9(1.0)
10.5(1.0)	10.1(0.96)	13.4(1.3)	28.6(2.7)	21.8(2.0)	10.1(0.96)	10.3(0.98)	13.0(1.2)	9.75(0.93)

Table 5.10: Comparison of normalized cache access and cache misses for different meshes of a bifurcation model on NICS Kraken-XT5 with and without using data reordering algorithm. Case of 8.8 million region mesh is taken as the base to normalize, i.e., normalized by the first row in Table 5.9. The information is normalized again (in the parentheses) by the relative work load which is in the first column (of a given row) of this table.

As the mesh size increases (from 8.8 million to 22 million, 42 million, and 92 million), the computational load per part increases to 2.5, 4.8, and 10.5 times, respectively. The L1 data cache access (L1_DCA) increases at the same ratio, which is expected. It also worth noting that, for a fixed-size problem on the same number of processor cores, L1_DCA is expected to be same (comparing columns 2 and 6 in Table 5.9). The cache misses on each of the L1, L2, and L3 levels increase as

the problem size increases as well but not in a proportional way due to fundamental differences in cache performance created by reordering the data. Specifically, comparing columns 3 and 7 we see that L1 cache misses increase at a faster rate than the mesh size growth when the data is not reordered. To see the difference in the growth rates more clearly we have rescaled each row of Table 5.10 by the first column and placed the results in parentheses (e.g., 1.3 indicates a 30% relative increase in cache misses for the largest mesh). The effect is even more dramatic on the L2 cache (170%) and L3 cache (100%) misses.

With data reordering, the increase in cache misses is close to the rate of the problem size increasing. The normalized L1_DCM by the relative problem size are 1.0, 1.0, 1.0, and 0.98, respectively with data reordering algorithm (column 7 of Table 5.10 in parentheses). This confirms the observation in Section 5.2.2 that the cases with heavily loaded cores spend disproportionally more time accessing the memory compared to the light loaded processors. L2 and L3 cache results are less uniform but still show substantially more stable results as compared to cases without reordering. The data reordering algorithm improves the cache performance in general and is reasonably successful at producing the same rate of data access from memory for all mesh sizes.

With this understanding of the cache performance, we have the opportunity to revisit the data from Table 5.8. Table 5.11 presents that data with rows 2 and 3 rescaled to give the time spent per element in the 92 million element case relative to the time spend per element in the each case/column (e.g., $s_e = (t_{92}/92M)/(t_i/n_i)$ where t_i is the time spent on the ith mesh (column) and n_i is the number of elements in the ith mesh). Without data reordering we appear to have super-linear speedup

Mesh size (million)	8.8	22	42	92
Without (s_e)	1.496	1.327	1.222	1.0
With (s_e)	0.973	1.018	0.972	1.0
Improvement	7.8%	21.9%	30.7%	40.0%

Table 5.11: Scaling factors (s_e) of FEA solver on different meshes with and without using data reordering algorithm on NICS Kraken-XT5.

as we reduce the mesh size (better cache utilization due to smaller range of data). With data ordering we have a more scalable cache performance (much closer to 1). While we have given up the passive super-linear performance, the data ordering algorithm introduced here has replaced it with something that is uniformly better. Table 5.11 also shows the relative improvement in the fourth row (Table 5.8 row 4 divided by row 2). Improvement is highest in the most heavily loaded cases (e.g., nearly 40% in the 92 million element case).

It is worth noting that the time spent reordering the mesh in all of the above studies is negligible compared to the time spent in preprocessing the mesh (preparing boundary conditions, initial conditions and partitioning). This preprocessing time in turn is negligible compared to solution times for simulations of practical interest. For example, the total time usage of preprocessing of the case on an AAA model with 133.6 million elements takes 190 seconds using 512 processing cores on Intrepid, where data reordering takes 7 seconds. To solve the same problem for 5000 steps, it takes about 50 hours of wall clock time on 512 Intrepid cores.

In this chapter, we have presented our work including implementation and testing of a data reordering algorithm in a FEA flow solver that improves the flop rate by relabeling mesh regions and mesh vertices. Mesh entities are reordered such that the ones visited, one after another, are topologically close to each other. This algorithm improves the usage of the memory hierarchy. Two examples are considered, one is a bifurcation pipe model and the other is an abdominal aortic aneurysm model. The following strategies are emphasized to demonstrate the usefulness of current data reordering algorithm:

- The first test was on an 8.8 million region mesh of a bifurcation pipe model where the relative impact of the data reordering algorithm on each phase of FEA solver was studied. Results showed that the reordering of both the vertices and regions simultaneously is important in the equation formation phase, while reordering of vertices improves performance in the equation solution stage.

- In the second study, an 8.8 million region mesh on a bifurcation pipe model was partitioned into different number of parts, which yields different average work load per part. This study indicates that the improvement due to the data reordering algorithm decays as the parts become lightly loaded.
- A similar study was performed on a much larger mesh (133.6 million regions) on an AAA model considering different architectures including IBM Blue Gene (BG/L and BG/P), Cray XT (XT3 and XT5) and Sun Constellation Cluster. It additionally suggests that the amount of performance gain that can be achieved by applying current data reordering algorithm is dependent on the system architecture as well.
- In the last test on Cray XT5, the data reordering effect was studied with different working load per part by fixing the number of processor cores and increasing the mesh size. The hardware performance tool, CrayPat, was used to measure the cache misses. This data was combined with an estimate of the miss penalty, to model the acceleration achieved by the FEA solver with data reordering. It closely accounts for observed decrease in the overall execution time.
- In all cases the time required to reorder the data (execution of Algorithm 8) is a small fraction of the cpu-time saving realized in the finite element analysis with reordered data.

CHAPTER 6
APPLICATION TO LARGE CARDIOVASCULAR SYSTEM

In this chapter, we demonstrate the capability of our high performance software by applying them to a numerically intensive application, which involves pulsatile incompressible blood flow in a "whole" body model. Trauma is now the third leading cause of death and the number one killer of people younger than 45 years of age. Vascular trauma comprises 3% of all civilian trauma injuries [18]. The physiological response to blood loss during vascular trauma makes use of various mechanisms in order to keep the overall systemic pressure relatively constant. These mechanisms include increasing the cardiac output, systemic vasoconstriction, and manipulating venous return, among others [64]. Because of the systemic nature of these responses, accurate simulation of the impact of vascular trauma on the body requires the consideration of larger portions of the vasculature than have previously been considered [26, 47, 59, 62, 83, 91]. A patient-specific 3D model, composed of 78 arteries extending across most of the body was constructed to meaningfully represent many of the large systemic arteries. We start with a 760MB CT dataset comprising 1479 slices at a 512x512 resolution, which covers the entire body from the neck to the feet. The dataset is loaded into SimVascular [79] and paths are created for the major arteries that span multiple subsections. Vessels are segmented using SimVascular's 2-D segmentation tool and 78 vessels are created in the 3-D model. Figure 6.1 shows an iso-surface image of the CT dataset and the solid model projected over the iso-surface. An initial unstructured mesh with 7 million elements is created and the solution-based mesh adaptivity is carried out to adapt the mesh iteratively to a higher resolution [68]. Transient incompressible Navier-Stokes equations assuming Newtonian fluid are considered as governing equations for blood flow simulation.

Figure 6.1: The CT dataset (left) and the model projected over the iso-surfaces (right)

6.1 Blood flow simulation in a 3-D "whole" body model

In this section, we demonstrate the blood flow simulation results on a 3-D "whole" body model using PHASTA with the capabilities of the adaptive boundary layer meshing techniques developed in [67, 68]. The layout of a patient specific 3D model composed of 78 arteries extending across most of the body is illustrated in Figure 6.2 (left), along with a magnified view marked for the main arteries (right). The height of the model is approximately $150cm$ with roughly a diameter of $3cm$ at the inlet (which is non-circular) and a diameter of $2-3mm$ at small arteries. The inflow velocity is assumed to be a Womersley profile [98], with Womersley number $\alpha = 5.6$. The volumetric flow rate at the inlet is shown in the inset within Figure 6.2. The waveform has a time period $t_p = 1.1s$; and a peak and trough of roughly $237cc/s$ and $16cc/s$, respectively. The inset also shows two instants of the peak in systole that is the heart contraction phase and the early diastole which is the dilation phase. The viscosity and the density are assumed to be $\mu = 0.04dynes\ s/cm^2$ and $\rho = 1.06g/cm^3$, respectively. No-slip conditions are assumed on the vessel walls and the RCR boundary conditions are prescribed at the outlets [91] (where an electric circuit analog of a proximal resistance R_p in series with a parallel arrangement of a capacitance C and a distal resistance R_d is applied to relate downstream pressure

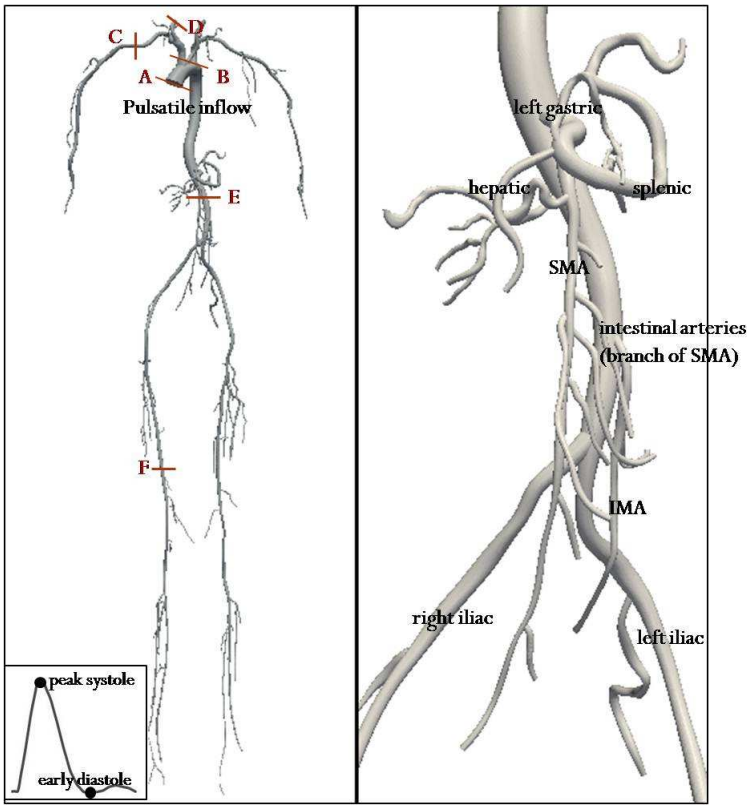


Figure 6.2: Whole body model composed of 78 arteries along with labels for sections (left) and labels of main arteries in the magnified view (right). The inset shows the volumetric flow rate at inlet (the two dots correspond to instants of peak systole and early diastole).

with the flow rate). As this case is representative of physiologic conditions we present the physiological parameters of pressure and volumetric flow rate as well as local quantities like velocity and wall shear stress.

Four cardiac cycles are simulated to obtain a periodic flow and thus, results for the last cycle are presented. Each cycle was divided into 1100 time steps with equal step size (1 ms). The computations are started using a fairly coarse pre-defined boundary layer mesh, with two layers of constant thickness and isotropic interior volume mesh, consisting of approximating $1.86M$ nodes and $7.1M$ elements. Four iterations of the flow computations and mesh modifications are performed to obtain an adapted boundary layer mesh with roughly $9.0M$ nodes and $42.8M$ elements. A single mesh modification step is applied based on the solution averaged over a

whole cardiac cycle. The average speed profile obtained over the last cardiac cycle is used to compute the error indicators. Figure 6.3 presents an overall view of the initial boundary layer meshes along with several magnified views for different arteries. Due to the high ratio of the model's total height over arteries diameters ($150cm/3mm = 500$), the use of a global uniform mesh size will create an initial mesh with too many elements while attempting to obtain an acceptable resolution of the small blood vessels. Therefore, the local mesh size is varied for different blood vessels when generating the initial mesh, which is in between 0.03cm and 0.1cm. A constant height, 0.004cm, is used for each of the two layers in the boundary layer for all the blood vessels. Even though, the initial mesh is still relatively uniform and isotropic in the interior volume, this 7.1M element mesh does not have high resolution on small blood vessels (e.g., Tibial artery in Figure 6.3 shows only tens of triangles on the cross surface).

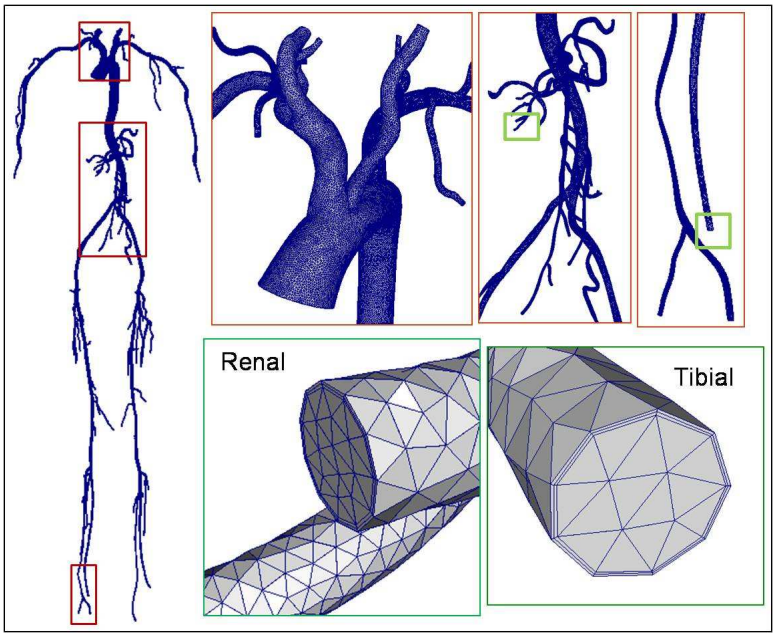


Figure 6.3: The initial boundary layer mesh on the "whole" body model with 7.1M elements.

The solution based, anisotropic, boundary layer mesh adaptation is used to improve the mesh quality. Four iterations of flow computation and mesh modification give an adapted mesh with 42M elements. Figure 6.4 presents magnified views of selected arteries. The effect of flow features can be clearly seen on the adapted

meshes. Elongated elements aligned in the flow direction are created in the regions where one component is varying sharply relative to the other ones such as near the vessel walls due to boundary layers. Isotropic elements are created where flow variation in each direction is relatively similar, for example, in branching regions. Figure

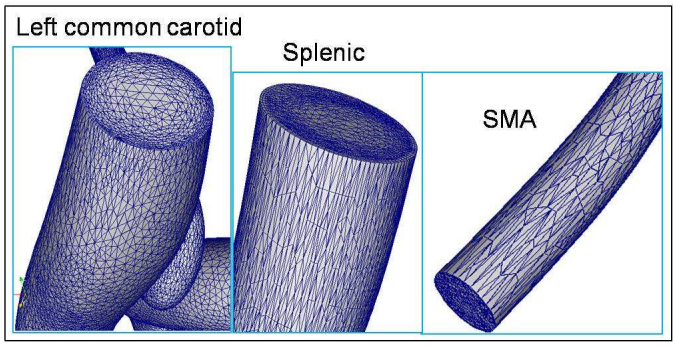


Figure 6.4: The adapted mesh (4th adaptation) on the "whole" body model with 42.8M elements.

6.5 compares the initial mesh with the adapted mesh on selected arteries to demonstrate the high quality of the adapted mesh even on small blood vessels (e.g., radial artery). The minimum local size in the adapted mesh is 0.004cm while the maximum size is 0.8cm. The adaptation increases the mesh size by a factor of 6 (42M/7M), but increase the local resolution by a factor of 25 (0.1cm/0.004cm) in some regions. To achieve the same level of resolution, uniform refinement will increase the mesh by $25 \times 25 \times 25 = 15.6$ *thousand* times, due to the three directions. This is one of the advantages of the anisotropic adaptation. More over, it can be seen in the case of adapted boundary layer mesh that layered and graded elements are maintained near the walls. The transition of element at the interface between the layered part and the interior tetrahedral part of the mesh became smooth in the adapted mesh. This avoids mesh based artifacts in the solution due to a rapid change in the element size. Note that, the newly created nodes are projected ("snapped") onto the solid model surface to improve the geometric approximation.

Results obtained on the adapted mesh are presented as the distribution of pressure, volumetric flow rate, velocity and wall shear stress. Figure 6.6 and Figure 6.7 display the surface distribution of pressure and wall shear stress respectively over the vessel walls at three instants (peak systole, early diastole and end diastole)

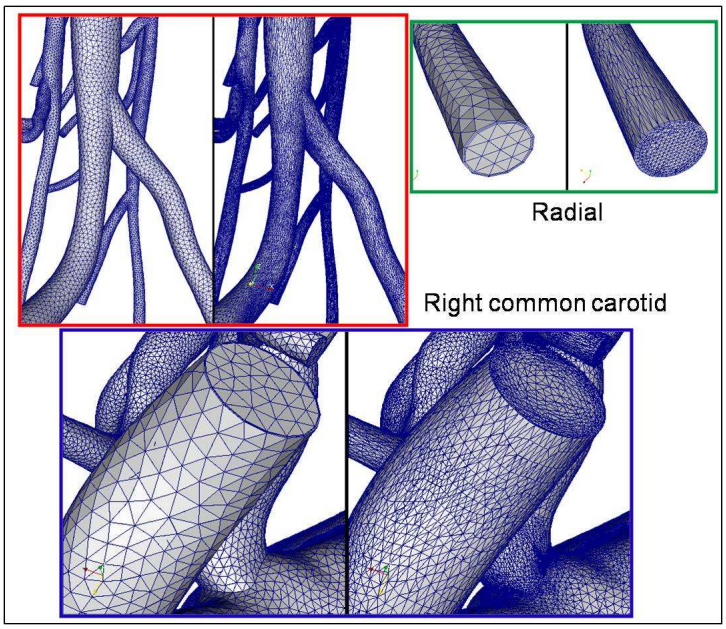


Figure 6.5: The comparison of initial and adapted meshes (4th adaptation).

of a cardiac cycle. At peak systole where the aortic inflow is maximum, the peak pressure over the model reaches 120mmHg (first one in Figure 6.6). Then soon after the aortic valve is closed (early diastole), aortic inflow is getting very low and the pressure is decreasing. On the top part of the model, the pressure drops below 100mmHg and the highest pressure of the model at this instant distributes over the legs due to the phase lag (middle one in Figure 6.6). The pressure continues to decrease until the end of the cycle (end diastole). At this point, the pressure reaches the minimum value over the whole cycle, which is around 60mmHg (right one in Figure 6.6). The trend of wall shear stress is similar. At peak systole, the wall shear stress is higher than the other instants and decreases afterward. The wall shear stress at end diastole is very low, due to the small inflow rate, especially in the aorta. The wall shear stress of small blood vessels is not as low, even at diastole, because of the small diameters of the arteries. Figure 6.8 shows the wall shear stress of the aorta in a magnified view with the same scale ($0 - 20 dynes/cm^2$). Some regions of aorta experience low shear stress over a long period during a cycle. Some research suggests that such regions face the high risk to disease making it important to provide this predictive ability. In the magnified view, we see a smoother distribution of wall

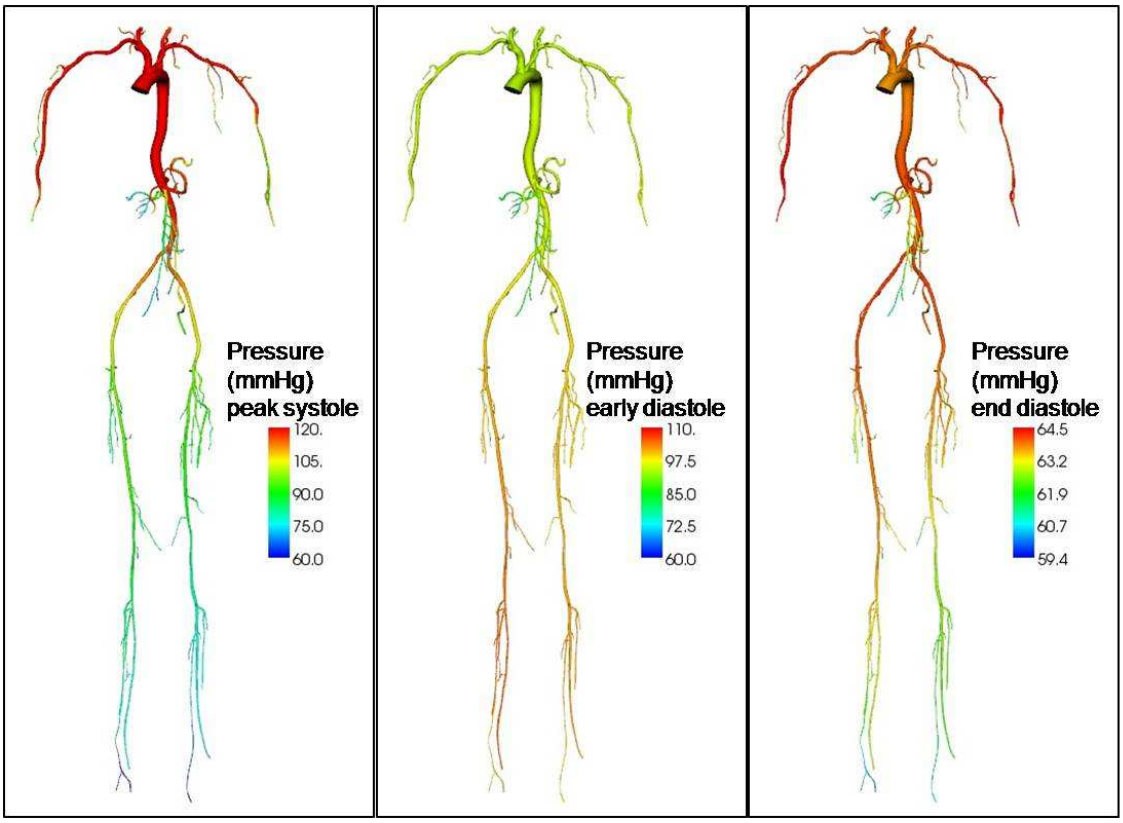


Figure 6.6: Surface distribution of the pressure over vessel walls obtained on the adapted mesh at peak systole, early diastole and end diastole. Note $1mmHg = 133.3Pa = 133.3N/m^2$**.**

shear stress on the model than many other works [28, 80, 89, 91], this is due to the fact that the adaptive boundary layer mesh is applied in this simulation [68].

In Figure 6.9 velocity magnitude on six selected cut-planes (see Figure 6.2) are shown at four instants of a cardiac cycle. At the inflow decreasing phase, flow features get complex on some cut-planes due to the decelerating flow in a complex geometry. Small scale flow features are developed and the adapted mesh demonstrates its ability to capture those features.

Figure 6.10 demonstrates temporal variation of the pressure and the flow rate at various outlets. We observed net back flow at early diastole at the outlets of left and right common carotid aorta. A phase lag of pressure relative to flow is observed, which is reasonable. We captured the realistic pressure and flow wave form by setting up the proper capacitance in RCR boundary conditions.

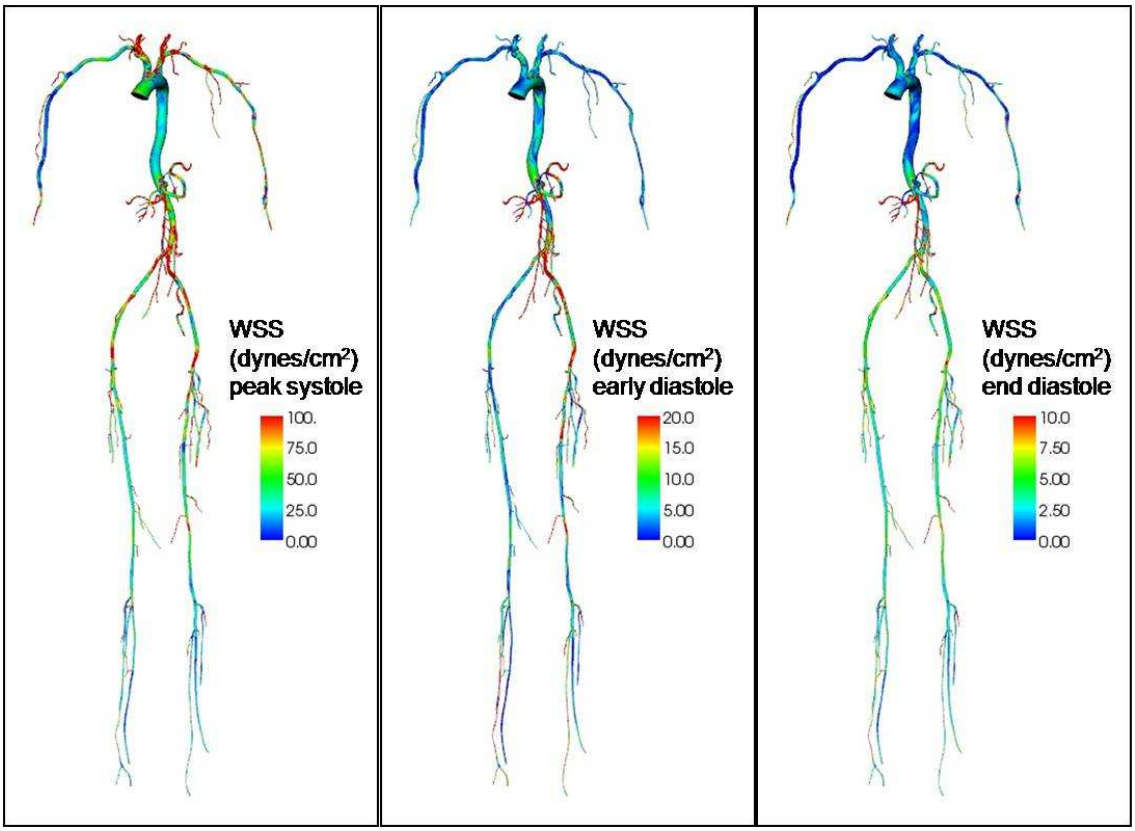


Figure 6.7: Surface distribution of the wall shear stress over vessel walls obtained on the adapted mesh at peak systole, early diastole and the end diastole. Note $1dynes/cm^2 = 0.1Pa = 0.1N/m^2$**.**

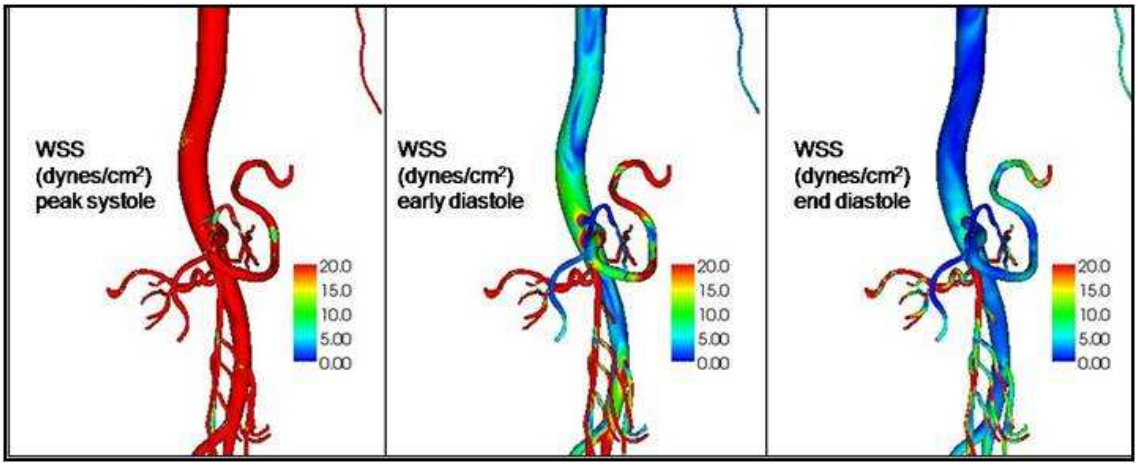


Figure 6.8: Magnified view of surface distribution of the wall shear stress over aorta walls obtained on the adapted mesh at peak systole, early diastole and end diastole. Note $1dynes/cm^2 = 0.1Pa = 0.1N/m^2$**.**

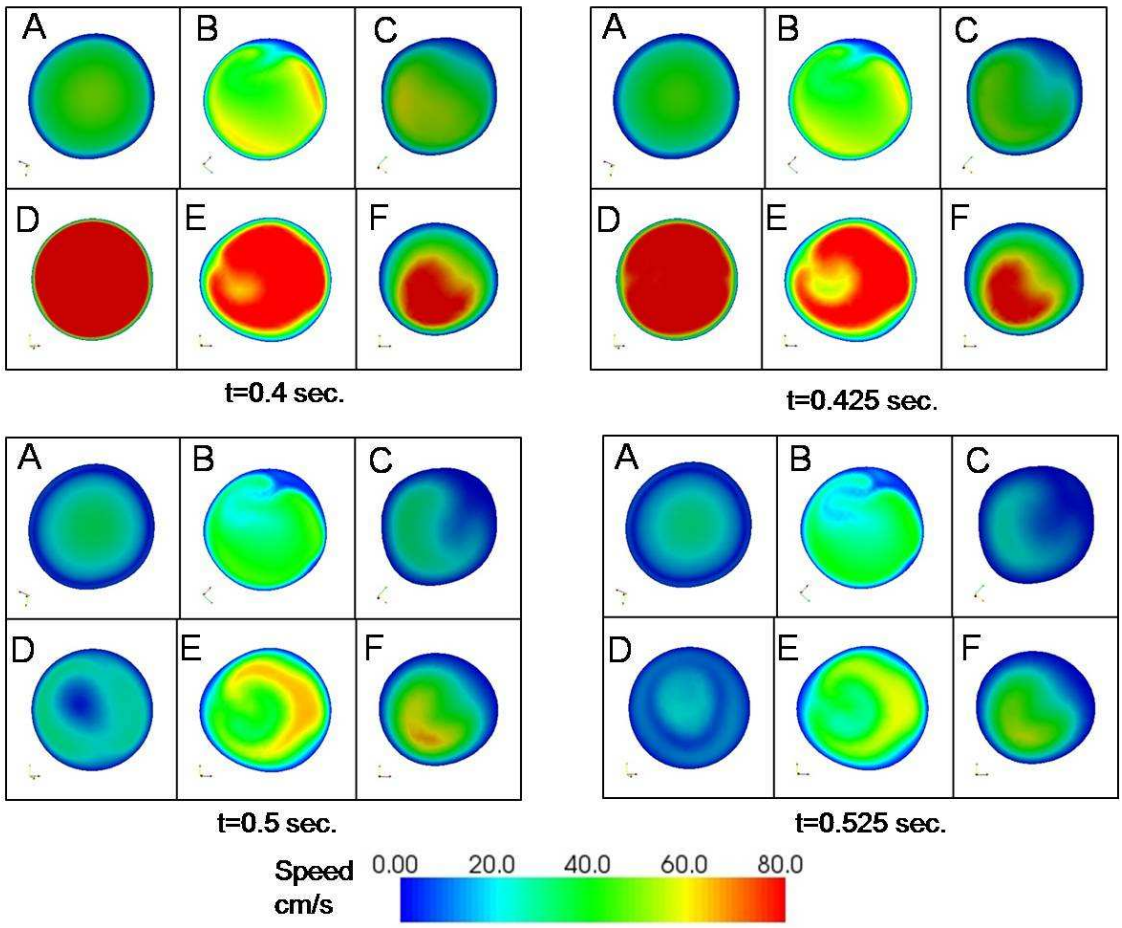


Figure 6.9: Velocity magnitude on selected cut-planes obtained on the adapted mesh at four instants. See Figure 6.2 for the location of the six cut-planes.

Figure 6.11 provides a comparison of the results obtained on the adapted mesh (M4) with the one computed on the initial mesh (M0). The temporal variation of pressure and velocity magnitude at center of brachial artery and descending aorta are presented for both meshes. The difference of results (both pressure and velocity magnitude) is observed on the initial mesh and the adapted mesh. Figure 6.12 demonstrates the comparison of the velocity magnitude along with the contours for cut plane B and E (see Figure 6.2 for the location of the cut planes) on initial mesh and adapted mesh at early diastole. The flow features are complex at those two locations at early diastole due to the complicated geometry. The contours on the initial mesh are not smooth because of the relative coarse mesh and very smooth on

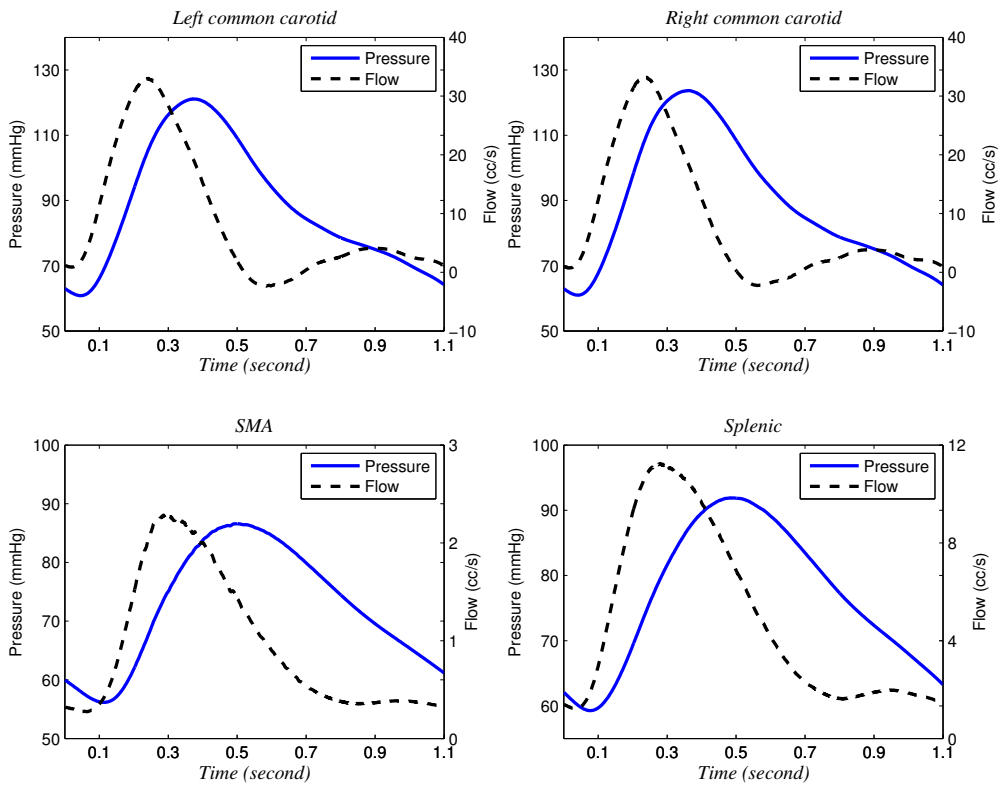


Figure 6.10: Pressure and volumetric flow rate computed on the adapted mesh at various outlets in a "whole" body model using RCR boundary conditions. (Note: $1mmHg = 133.3Pa = 133.3N/m^2$ **and** $1cc/s = 10^{-6}m^3/s$**.)**

the adapted mesh.

Figure 6.13 presents and compares the wall shear stress over vessel walls in a magnified view obtained on the initial (M_0) and the adapted mesh (M_4) at peak systole. Obvious differences are observed between these two meshes. The wall shear stress on the initial mesh already captured the distribution trend, the coarse mesh result is not as smooth as the result from the adapted mesh and is also over predicted (relative to the adapted mesh) in some regions. Clearly the adaptivity provides a substantially better prediction for wall shear stress.

In this chapter, we have demonstrated the simulation on a patient specific 3-D "whole" body model composed of 78 arteries covers a larger portion of vasculature than have been previously modeled. The unstructured initial mesh with predefined

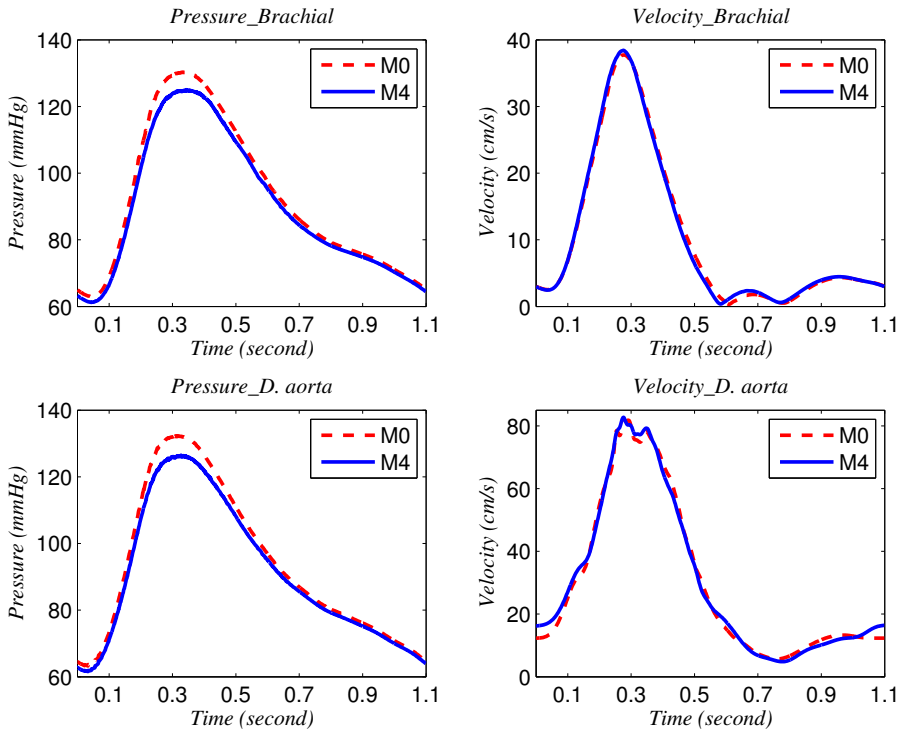


Figure 6.11: Comparison of temporal variation of pressure and velocity magnitude at the center of brachial artery and descending aorta between initial mesh and the adapted mesh.

boundary layers has been adapted based on the solution field to a mesh with about 42M elements. Without increasing the total number of elements too much (6 times), anisotropic mesh adaptation provides much higher local resolution in the adapted mesh (a factor of 25 in certain regions). Flow features, including pressure, velocity magnitude, volumetric flow rate and wall shear stress were presented on the adapted mesh. The small scale flow features in the decreasing phase are captured by the high resolution adapted mesh. The simulation on the final mesh (M4) was performed on Kraken (Cray XT5) at NICS, using 2048 processing cores and takes 90 mins (3,000 cpu hours) for one cardiac cycle. These results indicate that real problems with such a complex geometric model can be finished in a reasonable time frame as long as the flow solver has strong scalability, which is very important for time critical simulations.

Future work on this "whole" body model includes tuning the boundary conditions to each of the 78 arteries to split the flow to each region of body closer to

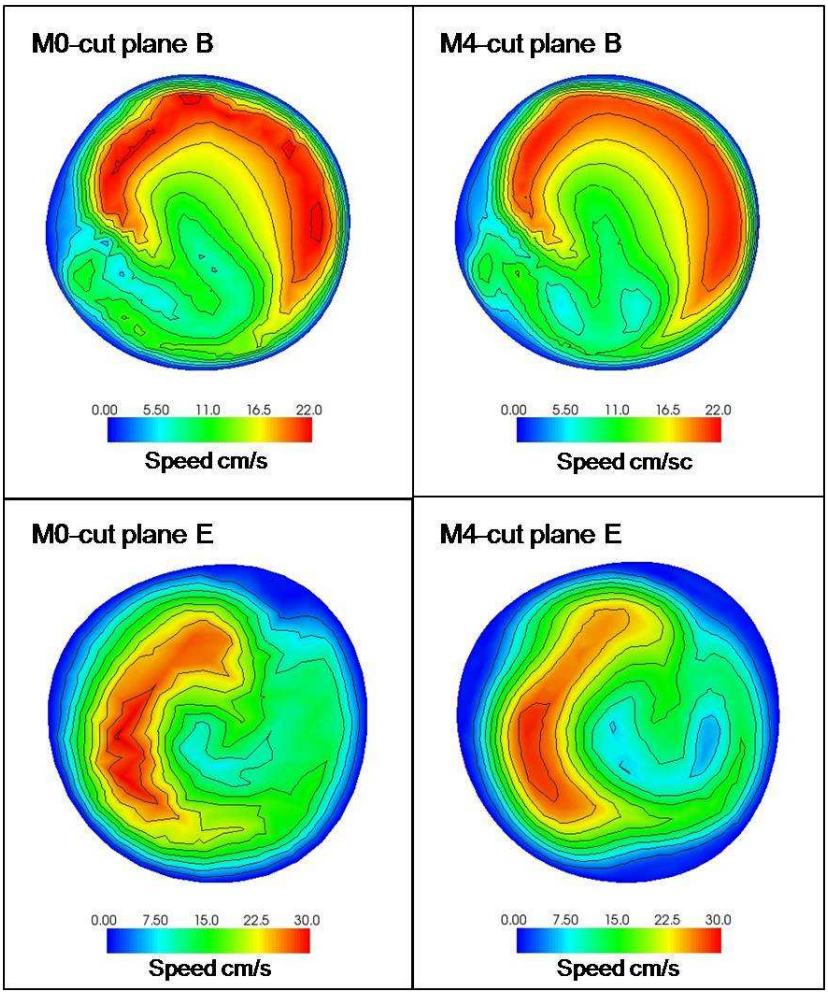
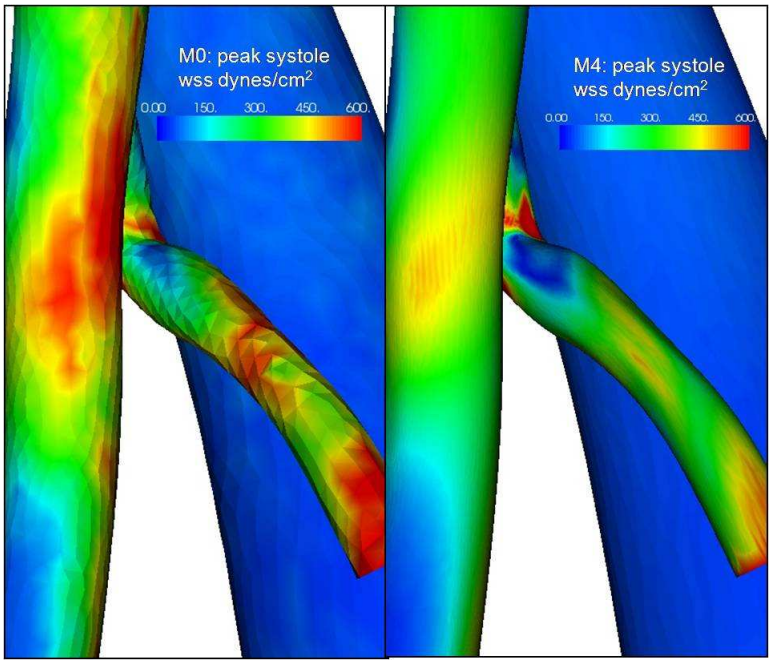
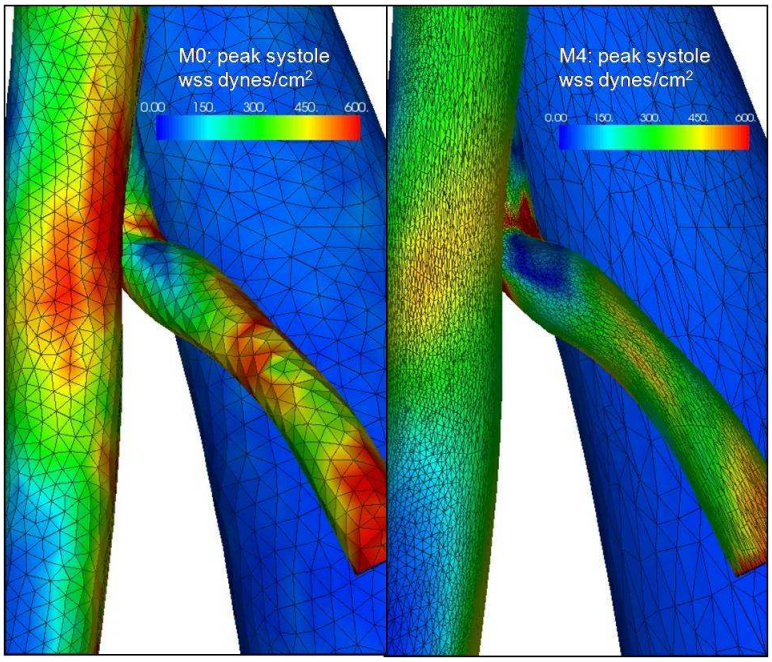


Figure 6.12: Comparison of velocity magnitude along with contours for cut plane B and E on the initial and the adapted meshes at early diastole.

realistic value. Vascular trauma will also be included in the future simulations.



(a) wall shear stress



(b) wall shear stress along with mesh

Figure 6.13: Comparison of wall shear stress on the initial and the adapted meshes at peak systole.

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

In this work we have focused our attention to identify and resolve the bottlenecks facing unstructured implicit finite element methods march towards petascale, and larger simulations. We have demonstrated the capabilities of our methods with strong scalability on the large scale supercomputers and the ability to solve the problems of interest with complex geometry and physics in a reasonable time frame. Excellent strong scaling was presented for various applications, including an abdominal aorta aneurysm model with more than 1 billion elements. The PHASTA flow solver scales all the way up to the 163,840 cores on a near-petascale system; Intrepid (IBM BG/P) at ALCF, ANL, i.e., all 40 racks of BG/P covering the full system scale. In this test case, each step takes only 1.2 seconds for an implicit solve (over 163,840 cores) on an adapted mesh with more than 1 billion elements. This demonstrates the ability of current procedures to achieve strong scalability on prospective supercomputers. In the other application, a "whole" body model composed of 78 arteries represents many of the major arteries in the systemic circulation was constructed. A solution-based, anisotropic boundary layer mesh adaptivity procedure was shown to dramatically improve the local mesh resolution to all the capture of the flow features due to the complex geometry without excessively increasing total number of elements. By distributing the work it takes 90 mins for PHASTA to compute one cardiac cycle on 2048 processing cores on Kraken (Cray XT5) at NICS, which shows the ability of our method to solve real-world problem with complex geometries in a realistic time. In this way, this work has provided a reliable and efficient computational tool that can be used for design and development purpose, for example, surgical-planning.

Based on the software and the methodologies which were designed for either serial or parallel computation with modest number of cores, this work has identified and resolved the obstacles for large scale adaptive unstructured implicit finite element analysis. Additionally, the performance of the finite element analysis is

improved dramatically by the algorithms developed in this work by providing better load balance on each processing core and better use of the hardware hierarchy (e.g., the memory subsystem). The issues addressed in this thesis are summarized as follow:

- Support the compatibility of preprocessing software *phParAdapt* with open sourced libraries, including mesh data base FMDB, mesh adaptivity tool *MeshAdapt*, and *meshmodel*. An open source attribute system is implemented to associate boundary conditions directly with the geometric model. This is one of the key components needed to enable us to take advantage of the latest super computers, since some of the commercial software that provides this capability are not presently supported on all the platforms.

- Extend the capability of *phParAdapt* to accommodate multiple part per processor. Since in the loop of adaptive simulations, the number of elements (and therefore the number of parts) may grow by three orders of magnitude, allowing only one part on each processor is insufficient to meet the requirements of large scale, parallel, adaptive applications

- Design, a predictive load balancing algorithm within the parallel mesh adaptivity to achieve close to an equal number of mesh entities on each part after adapting the mesh. Predictive load balance is required in the situation where adaptivity is heterogeneous, causing a dramatic increase in the number of elements in certain regions (or decrease in the elements in other regions). Without such a capability, the resulting mesh imbalance can be so severe that the processors exhaust available memory.

- Develop a data reordering algorithm to increase the cache coherency thus improving flop rate. This algorithm enables the effective usage of the memory subsystem by increasing the data locality. The per-core performance is improved for both equation formation and equation solution stage of the FEA. We have observed a up to 40% acceleration on Cray XT5 and up to 24% on Blue Gene/L.

- Design and implementation of a multiple compute-object based partition improvement algorithms to reduce the vertex imbalance while maintaining satisfactory element balance. This development enables a significant improvement in the strong scalability for both the system formation and the system solution. It accelerates the equation solution by more than 10% in a test on an AAA model by reducing the vertex imbalance by 12%. Two such algorithms are developed in the current work.

With these issues resolved, our method has demonstrated its strong scalability up to hundreds of thousands of processors and its ability to meet the requirement of various real-world applications. There are still open issues in the numerical methods towards petascale computations, here we present a few of the numerous research areas that need to be addressed:

- Improve point-to-point communications. We plan to allow the partition-graph to explicitly account for network topology (such as 3D torus) and bidirectional network links (when possible). In addition to balancing the work load, the communication load on each processor needs to be balanced as well.

- Apply the methods to various applications with higher resolution (with $O(10^{10})$ elements) and to extend the study to other near-petascale systems such as Ranger at TACC and Kraken at NICS.

- Combine the existing graph/hypergraph partitioners with the multiple compute-object based partition improvement algorithms developed in this work in an optimal way to further improve the partition quality.

- Extend the "whole" body simulation. The boundary conditions need to be further improved to represent practical flow rate more reasonably. Vascular trauma will be included in the future simulations.

- Integrate developments (by other research group members) in parallel boundary layer adaptivity as they become available into the petascale framework described in this thesis to better address large scale viscous boundary layer

flows. Specifically, this includes multiple parts per process which was shown in this thesis to be key to success.

- Though not explicitly discussed in this thesis, the volume of data created by the simulation tools combined with petascale computers can be overwhelming. Two ongoing research developments need to be integrated carefully into this petascale framework. The first is an overhaul of our file I/O. The current paradigm writes one file per part per time step which can not only overwhelm the file system but also users trying to manage the literally millions of files that can be produced. Other research group members have made progress with MPI-IO to address this situation but it is important to integrate that work into this framework and evaluate its performance in the petascale applications.

- Another approach to deal with the volume data created by petascale simulation is *in situ* processing. The idea is to link our framework with a post-processing library to allow specific data analysis/visualization filters to be applied to the data to provide "on the fly" or *in situ* processing and perhaps even rendering of the data. Currently Kitware Inc. (developers of ParaView [8]) have developed libraries that link to PHASTA to continuously evaluate and export a filter chain that has been defined prior to the launch of the simulation. This development, and their planned future developments (which include dynamic redefinition of the filter chain based on continuous rendering of ongoing simulations) should be thoroughly integrated into the framework described in this thesis, evaluated for its impact on parallel scaling, and demonstrated for its potential to more fully realize the vision of a dramatic compression of the complete time-to-solution since data analysis/insight extraction is ultimately critical to the value of a numerical simulation.

REFERENCES

[1] Parametric Technologies Corp. Pro/TOOLKIT reference manual 128 Technology Drive.

[2] Eletronic Data System Corp. Parasolid V6 Functional Discription.

[3] Simmetrix Inc. Simulation Modeling Suite, http://www.simmetrix.com, August 21, 2009.

[4] http://www.opencascade.org, August 21, 2009.

[5] Spatial Technology Inc.http://www.spatial.com., August 21, 2009.

[6] http://www.anandtech.com, August 21st, 2009.

[7] http://techreport.com/articles.x/13176/3, August 21st, 2009.

[8] http://www.paraview.org/, August 21st, 2009.

[9] M. Ainsworth and J.T. Oden. *A Posteriori Error Estimation in Finite Element Analysis*. John Wiley & Sons, New York, 2002.

[10] E. Bänsch. Local refinement in 2 and 3 dimensions. *Impact of Comp. in Sci. and Engng.*, 3:181–191, 1991.

[11] M.W. Beall. *An Object-Oriented Framework for the Reliable Automated Solution of Problems in Mathematical Physics*. PhD thesis, Rensselaer Polytechnic Institute, May 1999.

[12] M.W. Beall and M.S. Shephard. A general topology-based mesh data structure. *Int. J. Numer. Meth. Engng.*, 40(9):1573–1596, 1997.

[13] S. Behara and S. Mittal. Parallel finite element computation of incompressible flows. *Parallel Computing*, 35(4):195–212, 2009.

[14] C. Bekas, A. Curioni, P. Arbenz, C. Flaig, G.H. van Lenthe, R. Müller, and A.J. Wirth. Extreme scalability challenges in micro-finite element simulations of human bone. *International supercomputing conference*, 2008.

[15] R. Biswas, M. Aftosmis, C. Kiris, and B. Shen. Petascale computing: Impact on future NASA missions. *Chapter in: Petascale computing: Algorithms and Applications (Chapman & Hall/CRC Computational Science Series)*, 2008.

[16] E. Boman, K. Devine, L.A. Fisk, R. Heaphy, B. Hendrickson, V. Leung, C. Vaughan, U. Catalyurek, D. Bozdag, and W. Mitchell. Zoltan home page. http://www.cs.sandia.gov/Zoltan, 1999.

[17] D.A. Burgess and M.B. Giles. Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines. *Advances in Engineering Software*, 28:189–201, 1997.

[18] C. Compton and R. Rhee. Peripheral vascular trauma. *Perspectives in Vascular Surgery and Endovascular Therapy*, 17(4):297–307, 2005.

[19] A.L.G.A. Coutinho, M.A.D. Martins, R.M. Sydenstricker, and R.N. Elias. Performance comparison of data-reordering algorithms for sparse matrix-vector multiplication in edge-based unstructured grid computations. *International Journal for Numerical Methods in Engineering*, 66:431–460, 2006.

[20] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. *In Proc. 24th Nat. Conf. ACM*, pages 157–172, 1969.

[21] H.L. de Cougny and M.S. Shephard. Parallel refinement and coarsening of tetrahedral meshes. *Int. J. Numer. Meth. Engng.*, 46:1101–1125, 1999.

[22] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.

[23] S. Ethier, W.M. Tang, R. Walkup, and L. Oliker. Large-scale gyrokinetic particle simulation of microturbulence in magnetically confined fusion plasmas. *IBM Journal of Research and Development*, 52:105–115, January 2008.

[24] J.E. Flaherty, R.M. Loy, C. Özturan, M.S. Shephard, B.K. Szymanski, J.D. Teresco, and L.H. Ziantz. Parallel structure and dynamic load balancing for adaptive finite element computation. *Applied Numerical Mathematics*, 26:241–263, 1996.

[25] J.E. Flaherty, R.M. Loy, M.S. Shephard, B.K. Szymanski, J.D. Teresco, and L.H. Ziantz. Predictive load balancing for parallel adaptive finite element computation. *Proc. of Int. Conf. on Parallel and Distributed Processing Techniques and Applications, PDPTA'97*, 1:460–469, 1997.

[26] L. Formaggia, J.F. Gerbeau, F. Nobile, and A. Quarteroni. On the coupling of 3D and 1D Navier-Stokes equations for flow problems in compliant vessels. *Comput. Methods Appl. Mech. Engrg.*, 191:561–582, 2001.

[27] L.P. Franca and S.L. Frey. Stabilized finite element methods: II. the imcompressible Navior-Stokes equations. *Comp. Meth. Appl. Mech. Engng.*, 99:209–233, 1992.

[28] T. Frauenfelder, M. Lotfey, T. Boehm, and S. Wildermuth. Computational fluid dynamics: hemodynamic changes in abdominal aortic aneurysm after stent-graft implantation. *Cardiovasc. Intervent. Radiol.*, 29(4):613–623, 2006.

[29] O. Ghattas, C. Burstedde, M. Gurnis, G. Stadler, E. Tan, T. Tu, L.C. Wilcox, and S. Zhong. Scalable adaptive mantle convection simulation on petascale supercomputers. *Proceedings of IEEE/ACM SC '08*. Finalist paper for the Gordon Bell Prize 2008.

[30] N.E. Gibbs, W.G. Poole JR., and P.K. Stockmeyer. An algorithm for reducing the bandwidth and profile of sparse matrix. *SIAM J. Numer. Anal.*, 18(2):235–251, April 1976.

[31] N.E. Gibbs, W.G. Poole JR., and P.K. Stockmeyer. A comparison of several bandwidth and profile reduction algorithms. *ACM Trans. Math. Softw.*, 2(4):322–330, Dec. 1976.

[32] G. Heber, R. Biswas, and G.R. Gao. Self-avoiding walks over adaptive unstructured grids. *Concurrency: Practice Experience*, 12:85–109, 2000.

[33] JR. H.L.Crane, N.E. Gibbs, JR. W.G. Poole, and P.K. Stockmeyer. Matrix bandwidth and profile reduction. *ACM Trans. Math. Softw.*, 2(4):375–377, 1976.

[34] T.J.R. Hughes, L. Mazzei, and K.E. Jansen. Large-eddy simulation and the variational multiscale method. *Computing and Visualization in Science*, 3:47–59, 2000.

[35] K.E. Jansen. Unstructured grid large eddy simulation of wall bounded flow. Technical report, NASA Ames / Stanford University, Center for Turbulence Research, 1993.

[36] K.E. Jansen. Unstructured grid large eddy simulation of flow over an airfoil. Technical report, NASA Ames / Stanford University, Center for Turbulence Research, 1994.

[37] K.E. Jansen. A stabilized finite element method for computing turbulence. *Comp. Meth. Appl. Mech. Engng.*, 174:299–317, 1999.

[38] K.E. Jansen, C.H. Whiting, and G.M. Hulbert. A generalized-α method for integrating the filtered Navier-Stokes equations with a stabilized finite element method. *Comp. Meth. Appl. Mech. Engng.*, 190:305–319, 1999.

[39] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–13, 1998.

[40] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. *SIAM review*, 41(2):278–300, 1999.

[41] G. Karypis, K. Schloegel, and V. Kumar. Parmetis - parallel graph partitioning and fill-reducing matrix ordering.

[42] X. Li. *Mesh Modification Procedures for General 3D Non-manifold Domains*. PhD thesis, Rensselaer Polytechnic Institute, July 2003.

[43] X. Li, M.S. Shaphard, and M.W. Beall. Accounting for curved domains in mesh adaptation. *Int. J. Numer. Meth. Engng.*, 58:247–276, 2003.

[44] X. Li, M.S. Shaphard, and M.W. Beall. 3D anisotropic mesh adaptation by mesh modification. *Comp. Meth. Appl. Mech. Engng.*, 194:4915–4950, 2005.

[45] R. Löhner. Renumbering strategies for unstructured-grid solvers operating on shared-memory, cache-based parallel machines. *Computer methods in applied mechanics and engineering*, 163:95–109, 1998.

[46] R. Löhner and M. Galle. Minimization of indirect addressing for edge-based field solvers. *Communications in numerical methods in engineering*, 18(5):335–343, 2002.

[47] A.L. Marsden, I.E. Vignon-Clementel, F.P. Chan, J.A. Feinstein, and C.A. Taylor. Effects of exercise and respiration on hemodynamic efficiency in CFD simulations of the total cavopulmonary connection. *Ann. Biomed. Eng.*, 35(2):250–263, 2007.

[48] B. Monien, R. Preis, and R. Diekmann. Quality matching and local improvement for multilevel graph-partitioning. *Parallel Computing*, 26:1609–1634, 2000.

[49] J. Müeller, O. Sahni, X. Li, K.E. Jansen, M.S. Shephard, and C.A. Taylor. Anisotropic adaptive finite element method for modeling blood flow. *Computer methods in biomechanics and biomedical engineering*, 8(5):295–305, 2005.

[50] R.M. O'Bara, M.W. Beall, and M.S. Shephard. Analysis model visualization and graphical analysis attribute specification system. *Finite Elements in Analysis and Design*, 19:325–348, 1994.

[51] L. Oliker, R. Biswas, and R.C. Strawn. Parallel implementation of an adaptive scheme for 3D unstructured grids on the SP2. *In Proc. 3rd International Workshop on Parallel Algorithm for Irregularly Structured Problems*, 1996.

[52] L. Oliker, A. Canning, J. Carter, C. Iancu, M. Lijewski, S. Kamil, J. Shalf, H. Shan, E. Strohmaier, S. Ethier, and T. Goodale. Scientific adaptation performance on candidate petascale platforms. *International Parallel & Distributed Processing Symposium (IPDPS)*, March 24-30 2007. Winner Best Paper.

[53] L. Oliker, X. Li, P. Husbands, and R. Biswas. Ordering schemes for sparse matrices using modern programming paradigms. *LBNL47803*, 2000.

[54] L. Oliker, X. Li, P. Husbands, and R. Biswas. Parallel conjugate gradient: effects of ordering strategies, programming paradigms, and architectural platforms. *LBNL45828*, 2000.

[55] L. Oliker, X. Li, P. Husbands, and R. Biswas. Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Review*, 44(3):373–393, 2002.

[56] L. Oliker, J. Shalf, J. Carter, A. Canning, S. Kamil, and M. Lijewski. Performance characteristics of potential petascale scientific applications. *Chapter in: Petascale Computing: Algorithms and Applications (Chapman & Hall/Crc Computational Science Series) by David A. Bader*, 2008.

[57] C.C. Pain, A.P. Umpleby, C.R.E. de Oliveira, and A.J.H. Goddard. Tetrahedron mesh optimization and adaptivity for steady-state and transient finite element calculations. *Comput. Methods Appl. Mech. Engrg.*, 190:3771–3796, 2001.

[58] F. Pellegrini. Scotch 5.0 user's guide. Technical report, LaBRI, August 2007.

[59] K. Perktold, R. Peter, and M. Resch. Pulsatile non-newtonian blood flow simulation through a bifurcation with an aneurysm. *Biorheology*, 26(6):1011–1030, 1989.

[60] J.C. Pichel, D.E. Singh, and J. Carretero. Reordering algorithms for increasing locality on multicore processors. *Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 123–130, September 2008.

[61] R. Preis and R. Diekmann. The party partitioning-library, user guide - version 1.1. Technical Report tr-rsfb-96-024, University of Paderborn, September 1996.

[62] A. Quarteroni, S. Ragni, and A. Veneziani. Coupling between lumped and distributed models for blood flow problems. *Comput. Visual. Sci.*, 4(2):111–124, 2001.

[63] J.F. Remacle, O. Klaas, J.E. Flaherty, and M.S. Shephard. Parallel algorithm oriented mesh database. *Engineering with Computers*, 18:274–284, 2002.

[64] N.M. Rich, K.L. Mattox, and A. Hirshberg. *Vascular Trauma*. Elsevier Saundres, second edition, 2004.

[65] Y. Saad and M.H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal of Scientific and Statistical Computing*, 7:856–869, 1986.

[66] O. Sahni. *Automated Adaptive Viscous Flow Simulations*. PhD thesis, Rensselaer Polytechnic Institute, December 2007.

[67] O. Sahni, K.E. Jansen, M.S. Shephard, C.A. Taylor, and M.W. Beall. Adaptive boundary layer meshing for viscous flow simulations. *Engn. with Comp.*, 24(3):267–285, 2008.

[68] O. Sahni, Y. Müller, K.E. Jansen, M.S. Shephard, and C.A. Taylor. Efficient anisotropic adaptive discretization of the cardiovascular system. *Comp. Meth. Appl. Mech. Engng.*, 195:5634–5655, 2006.

[69] O. Sahni, M. Zhou, M.S. Shephard, and K.E. Jansen. Scalable implicit finite element solver for massively parallel processing with demonstration to 160k

cores. *Accepted by Proceedings of IEEE/ACM SC '09*. Finalist paper for the Gordon Bell Prize 2009.

[70] K. Schloegel, G. Karypis, and V. Kumar. A new algorithm for multi-objective graph partitioning. In *Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, pages 322–331, Springer, Berlin, 1999.

[71] K. Schloegel, G. Karypis, and V. Kumar. Parallel static and dynamic multi-constraint graph partitionin. *Computation and Concurrency: Practice and Experience*, 14:219–240, 2002.

[72] E.S. Seol and M.S. Shephard. Efficient distributed mesh data structure for parallel automated adaptive analysis. *Engineering with Computers*, 22(3-4):197–213, 2006.

[73] F. Shakib, T.J.R. Hughes, and Z. Johan. A multi-element group preconditioned GMRES algorithm for nonsymmetric systems arising in finite element analysis. *Comp. Meth. Appl. Mech. Engng.*, 75:415–456, 1989.

[74] Farzin Shakib. http://www.acusim.com, August 21, 2009.

[75] M.S. Shephard and M.W. Beall. unpublished class notes, rensselaer polytechnic institute. 1990.

[76] M.S. Shephard, J.E. Flaherty, K.E. Jansen, X. Li, X.-J. Luo, N. Chevaugeon, J.-F. Remacle, M.W. Beall, and R.M. O'Bara. Adaptive mesh generation for curved domains. *Journal for Applied Numerical Mathematics*, 53(2-3):251–271, 2005.

[77] M.S. Shephard, K.E. Jansen, O. Sahni, and L.A. Diachin. Parallel adaptive simulations on unstructured meshes. *Journal of Physics Conference Series*, 78(doi:10.1088/1742-6596/78/1/012053), 2007.

[78] M.S. Shephard and E.S. Seol. Flexible distributed mesh data structure for parallel adaptive analysis. *Advanced Computational Infrastructures for Parallel and Distributed Adaptive Applications*, pages 1–38, 2007.

[79] SimVascular. Cardiovascular modeling and the simulation application. http://wiki.simtk.org/simvascular, August 21st, 2009.

[80] D.A. Steinman, J.S. Milner, C.J. Norley, S.P. Lownie, and D.W. Holdsworth. Image-based computational simulation of flow dynamics in a giant intracranial aneurysm. *AJNR Am. J. Neuroradiol.*, 24(4):559–566, 2003.

[81] A.G. Sunderland, M. Ashworth, N. Li, C. Moulinec, Y. Fournier, and J. Uribe. Towards petascale computing with parallel CFD codes. Technical report, The UK's World-Class Service for World-Class Research, 2008.

[82] S. Swaminarayan, T.C. Germann, K. Kadau, and G.C. Fossum. 369 tflops/s molecular dynamics simulations on the roadrunner general-purpose heterogeneous supercomputer. *Proceedings of IEEE/ACM SC '08*. Finalist paper for the Gordon Bell Prize 2008.

[83] B.T. Tang, C.P. Cheng, M.T. Draney, N.M. Wilson, P.S. Tsao, R.J. Herfkens, and C.A. Taylor. Abdominal aortic hemodynamics in young healthy adults at rest and during lower limb exercise: quantification using image-based computer modeling. *Am. J. Physiol. Heart Circ. Physiol.*, 291(2):H668–676, 2006.

[84] C.A. Taylor, T.J.R. Hughes, and C.K. Zarins. Finite element modeling of blood flow in artieries. *Comp. Meth. Appl. Mech. Engng.*, 158:155–196, 1998.

[85] A.E. Tejada-Martínez and K.E. Jansen. On the interaction between dynamics model dissipation and numerical dissipation due to streamline upwind/petro-galerkin stabilization. *Comp. Meth. Appl. Mech. Engng.*, 194(9-11):1225–1248, 2005.

[86] A.E. Tejada-Martínez and K.E. Jansen. A parameter-free dynamics subgrid model for large-eddy simulation. 195(23-24):2919–2938, 2006.

[87] J.D. Teresco, K.D. Devine, and J.E. Flaherty. Partitioning and dynamic load balancing for the numerical solution of partial differential equations. In

Numerical Solution of Partial Differential Equations on Parallel Computers. Springer-Verlag, 2005.

[88] T.E. Tezduyar and A. Sameh. Parallel finite element computations in fluid mechanics. *Comput. Methods Appl. Mech. Engrg.*, 195:1872–1884, 2006.

[89] R. Torii, M. Oshima, T. Kobayashi, K. Takagi, and T. E. Tezduyar. Computer modeling of cardiovascular fluid-structure interactions with the deforming-spatial-domain/stabilized space-time formulation. *Comput. Methods Appl. Mech. Engrg.*, 195(13-16):1885 – 1895, 2006. A Tribute to Thomas J.R. Hughes on the Occasion of his 60th Birthday.

[90] R. Verfürth. *A Review of Posteriori Error Estimation and Adaptive Mesh-Refinement Techniques*. Teubner-Wiley, Stuttgart, 1996.

[91] I.E. Vignon-Clementel, C.A. Figueroa, K.E. Jansen, and C.A. Taylor. Outflow boundary conditions for three-dimensional finite element modeling of blood flow and pressure in arteries. *Comput. Methods Appl. Mech. Engrg.*, 195(29-32):3776–3796, 2006.

[92] B. Waldecker. Amd technical briefing. Avail on the web at: http://www.nccs.gov/wp-content/training/scaling_workshop_pdfs/AMD_ORNL_073007.pdf.

[93] C. Walshaw and M. Cross. JOSTLE: Parallel multilevel graph-partitioning software-an overview. *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58, 2007.

[94] J. Wan, S. Kocak, and M.S. Shephard. Automated adaptive 3-D forming simulation process. *Engineering with Computers*, 21:47–75, 2005.

[95] M. Wehner, L. Oliker, and J. Shalf. Towards ultra-high resolution models of climate and weather. *International Journal of High Performance Computing Applications*, 22(2):149–165, 2008.

[96] C.H. Whiting and K.E. Jansen. A stabilized finite element methods for the incompressible navier-stokes equations using hierarchical basis. *International Journal for Numerical Methods in Fluids*, 35:93–116, 2001.

[97] C.H. Whiting, K.E. Jansen, and S. Dey. Hierarchical basis in stabilized finite element methods for compressible flows. *Comp. Meth. Appl. Mech. Engng.*, 192:5167–5185, 2003.

[98] J. Womersley. Method for the calculation of velocity, rate of flow and viscous drag in arteries when the pressure gradient is known. *J. Physiol.*, 127(553-563), 1955.

[99] M. Zhou, O. Sahni, M.S. Shephard, C.D. Carothers, and K.E. Jansen. Data reordering algorithm for acceleration of finite element computations. 2009. submitted.