IMPROVING SCALABILITY OF PARALLEL UNSTRUCTURED MESH-BASED ADAPTIVE WORKFLOWS

By

Cameron Walter Smith

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: COMPUTER SCIENCE

Approved by the
Examining Committee:

Mark S. Shephard, Thesis Adviser

Max O. Bloomfield, Member

Christopher D. Carrothers, Member

Barbara Cutler, Member

Onkar Sahni, Member

Rensselaer Polytechnic Institute
Troy, New York

April 2017
(For Graduation May 2017)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

My seven year mission to seek out new methods for load balancing and building unstructured mesh workflows was a team effort. First and foremost, my family and my fiance Alysia provided the much-needed love and moral support to get through long hours, travel, and times of high stress. My colleagues Dan Ibanez, Brian Granzow, Seegyoung Seol, Gerrett Diamond, Dan Zaide, Ben Matthews, Michel Rasquin and many others collectively provided excellent technical support and devoted many hours to help advance my work. When my colleagues and I couldn't work out a bug or conceptual issue professors Ken Jansen, Onkar Sahni, Chris Carrothers, and Scientific Computation Research Center (SCOREC) research staff member Max Bloomfield provided precious chunks of their time to help.

Development, testing, and large scale runs were made possible by the staff and system administrators at the Texas Advanced Computing Center, Argonne National Laboratory, the SCOREC (Ben Matthews, Tim McMullan and Tim Wickberg), and the Computational Center for Innovations (Lindsay Todd, Dan Weeks, and Derek Fox) who kept the software and hardware systems running and up to date.

Thank you to my committee for mentoring me through the Ph.D. program and reviewing my work.

Last, but not least, thanks to my adviser and boss Mark Shephard for leading my professional development as a full-time staff member and my research as a part-time graduate student. Mark's guidance for navigating complex research topics and, at times, even more complex institutional bureaucracies, was critical to my success.

ABSTRACT

High performance parallel adaptive simulations operating on leadership class systems are constructed from multiple pieces of software developed over many years. As increasingly complex systems are deployed new methods must be created to extract performance and scalability. This thesis addresses two key scalability limitations for unstructured mesh-based simulations.

Attaining simulation performance at ever higher concurrency levels requires increasing the performance of transformations within each procedure, as well as the transfer of data between procedures.

Controlling the transformations requires distributing the work evenly across the processors while executing efficient data transfers requires local operations that avoid shared or contended resources. This thesis addresses these requirements through multi-criteria load balancing procedures and in-memory data transfer techniques.

Partition improvement methods defined in this work enable improved application strong scaling on over one million processors through careful control of the balancing requirements. Applied to a computational fluid dynamics simulation running on 524,288 processes with 1.2 billion elements these methods reduce the time of the dominant computational step by up to 28% versus the best existing methods.

The scalable data transfer requirement is addressed through an in-memory functional coupling that avoids the high cost of fileystem access. The methods developed are applied to adaptive simulations in which the time required for information exchange is reduced by over an order of magnitude versus file-based couplings. Three additional simulations for industrial applications are then provided that highlight an in-memory coupling and the automation of key simulation processes.

CHAPTER 1
INTRODUCTION AND CONTRIBUTIONS

Unstructured mesh methods, like finite elements [1] or finite volumes [2], support the effective analysis of complex physical behaviors modeled by partial differential equations over general three-dimensional domains. The most reliable and efficient methods apply adaptive procedures with *a posteriori* error estimators that indicate where and how the mesh is to be modified. Although adaptive meshes can have two to three orders of magnitude fewer elements than a more uniform mesh for the same level of accuracy, there are many complex simulations where the meshes required are so large that they can only be solved on massively parallel systems.

The parallel simulations of interest are defined by a series of procedures which we refer to as a workflow. Fig. 1.1 provides a high-level overview of this sequence. Starting at the top-left the problem definition is specified on the computational domain, typically a CAD model. Next, automated mesh generation procedures driven by the problem definition and optional mesh controls produce a spatial discretization of the computational domain, a mesh. With the mesh, the associated CAD model, and the problem definition, an analysis is executed until some criterion is met that indicates adaptation is required. In a workflow using mesh adaptation procedures this criterion is most effective when it is based on the discretization error [1]. Following the analysis, the adaptation procedure is executed to reduce the error by locally refining and coarsening the mesh [3], [4]. During these mesh modification operations local transfer procedures [5] are executed to maintain an accurate distribution of physical quantities of interest (e.g., the velocity of a fluid or the displacement of a solid). Once an adapted mesh has been created, the analysis procedure is executed again. This solve-adapt cycle is repeated until a stopping criterion is met, such as a pre-defined number of time/load steps. After the last cycle completes, post-processing tools are executed to extract spatial and temporal

Portions of this chapter have been submitted to: C. W. Smith, B. Granzow, G. Diamond, D. A. Ibanez, O. Sahni, K. E. Jansen *et al.*, “In-memory integration of existing software components for parallel adaptive unstructured mesh workflows,” submitted for publication.

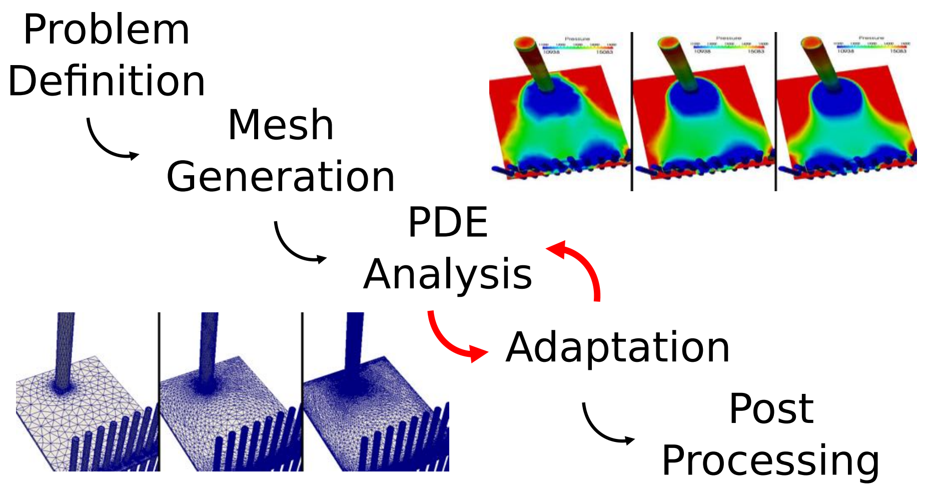


Fig. 1.1. The series of steps in an adaptive unstructured mesh-based workflow and (bottom) a sequence of adapted meshes and (top) the corresponding solution fields for an adaptive manifold flow simulation [6].

characteristics of the physical quantities of interest.

The time spent running a parallel workflow is dominated by the repeated execution of PDE analysis and adaptation procedures. Efficiency of the analysis is maintained by redistributing the mesh after adaptation procedures have non-uniformly refined and coarsened it. Given an existing mesh distribution across the processes running on a parallel system, a partition, dynamic load balancing methods determine which mesh elements should be moved between processes to reduce the imbalance and communication costs. For part counts numbering in the tens of thousands, multi-level methods operating on the dual graph of the mesh are sufficient [7], but beyond this concurrency level these methods often fail due to memory usage that increases significantly with process count [8]. The goal of **Par**titioning using **M**esh **A**djacencies (ParMA) is to perform efficient, multi-criteria, dynamic load balancing of unstructured meshes by directly using the existing mesh adjacency information. Results will demonstrate the ability of ParMA to dynamically re-balance meshes for multiple criteria with billions of mesh regions on over one million processors. Thus, ParMA addresses a key limitation to workflow scalability by enabling efficient transformation of data within the computationally dominant analysis procedures.

Within the solve-adapt cycle are transfers of mesh and field data between analysis and adaptation procedures (depicted by the pair of bold arrows in Fig. 1.1). In massively parallel simulations requiring frequent adaptation, the reading and writing of files between the execution of analysis and adaptation procedures introduces significant computational overheads. To avoid shared and contentious parallel filesystems, our in-memory coupling approaches using data streams and APIs provide scalable and efficient transfers of data by using node-local memory. Thus, in-memory transfer approaches address a second key limitation to parallel workflow scalability.

1.1 Contributions

The work of this thesis develops load balancing and in-memory workflow construction methods for conformal 2D and 3D unstructured meshes composed of quadrilaterals, triangles, tetrahedra, prisms, hexahedra, and pyramids. A conformal mesh is one in which the intersection of two elements is a lower order mesh entity shared by both (i.e., a face for two regions, an edge for two faces, and a vertex for two edges).

1.1.1 Dynamic Partition Improvement

ParMA provides dynamic load balancing methods that compliment existing graph and geometric partitioners to create mesh partitions with billions of elements on millions of processors that are tailored to the needs of a given application. Partition improvement results at various per part element counts on over one million processors are discussed.

Zhou's 2010 work [9] defines the LIIPBMod algorithm for reducing vertex imbalance and the number of vertices on part boundaries while indirectly trying to limit the increase of element imbalance. In 2012, Zhou [10] executes a strong scaling study of a massively parallel computational fluid dynamics application using partitions created with (hyper)graph partitioners and LIIPBMod. Our work, ParMA, defines new algorithms for balancing all entity dimensions (vertices, edges, faces and regions), with weights, while reducing the number of vertices on the part boundaries, the number of disconnected components, and the average number of neighboring parts. ParMA developments were guided by Zhou's work for vertex balancing.

Our work, relative to Zhou's, demonstrates multi-entity balancing on up to 3.5 times more parts, 1Mi, with up to two times smaller parts, 1100 elements. Like Zhou, we focus on balancing tetrahedral meshes. We also support balancing mixed and other monotopological meshes (e.g., all quadrilaterals or all hexahedra).

ParMA's implementation relies on the PUMI parallel unstructured mesh infrastructure [11], and inter-process communication algorithms detailed by Ibanez et al. [12]. We refer readers to these papers for details on the element migration procedure and neighborhood communications for information exchange.

In this work, ParMA, combined with graph and geometric partitioning methods provided by Zoltan [13], satisfies the requirements for dynamic load balancing described in Section 2.2 to over one million parts on meshes with over 12 billion tetrahedral elements. Partition quality requirements 1 and 2 are satisfied by partitioning the mesh with a graph or geometric partitioner and then running ParMA to reduce the imbalance of mesh entity dimensions critical to the application. For example, ParMA is applied to balance the entities used as degree of freedom holders in finite element method procedures. The incremental partition change requirement (3) is implicitly satisfied by the definition of ParMA's diffusion procedure and recursive coordinate bisection. Graph-based methods provided by Zoltan's API also have execution modes that minimize data movement. Requirement 4 is implicitly satisfied as applications in the workflow are driven from the partitioning of the mesh that ParMA produces. Performance requirements 5 and 6 are satisfied by combining ParMA with a partitioner that scales to the required concurrency level. Lastly, requirement 7 is satisfied through Zoltan's API to interact with the mesh data structure and ParMA's direct use of mesh modification and query APIs.

1.1.2 In-memory Component Coupling

Critical to the construction of parallel workflows is the ability to couple existing pieces of software. We define bulk and atomic level couplings implemented using

API- and data stream-based approaches. These in-memory couplings are applied to a monolithic, mixed C/C++ and FORTRAN, computational fluid dynamics (CFD) analysis code, a C++, hp-adaptive, finite element framework for linear accelerator frequency analysis, and a C++ multi-physics framework. Scaling results up to 16,384 processes are provided for the coupling of the massively parallel PHASTA CFD analysis code with mesh adaptation, and the memory overhead for the linear accelerator framework when coupled with mesh adaptation is studied.

1.1.3 Parallel Workflows for Industrial Applications

In addition to the three in-memory workflows discussed above, we also construct parallel workflows for three industrial applications. The first workflow demonstrates an adaptive, multi-phase flow simulation using an in-memory coupling to a closed-source, serial procedure provided by the industrial partner. The second and third workflows focus on reducing the time engineers and analysts have to spend setting up problems and running jobs by applying automation and abstraction techniques.

1.2 Thesis Organization

The thesis is organized into the following chapters.

- Chapter 2 details the ParMA load balancing software and its support for extreme scale workflows.
- Chapter 3 discusses the construction of parallel, unstructured mesh-based, in-memory workflows and the components that define them.
- Chapter 4 describes the in-memory coupling for three unstructured mesh-based workflows and examines their performance relative to file-based approaches on massively parallel systems.
- Chapter 5 discusses three industrial workflows: one using in-memory coupling techniques, and two others using automation technologies.
- Chapter 6 summarizes the work and discusses some possible future efforts.

1.3 Terminology and Notation

2D,3D	two- and three-dimensional.
petascale,exascale	a computer system capable of executing 10^{15} and 10^{18} floating-point operations per second, respectively.
CFD	computational fluid dynamics.
CAE	computer-aided engineering.
CAD	computer-aided design/drafting.
ISV	independent software vendor; typically a for-profit organization with a closed-source product.
I/O	input and output.
API	application program interface.
CPU	central processing unit; typically a socketed device on the motherboard with multiple, independent, out-of-order processing units.
(CPU) core	a processing unit within a CPU.
GPU	graphical processing unit; typically a bus-attached device with thousands of group-synchronized, simple, in-order processing units.
$O(1)$	an operation that executes in constant time.
workflow	the sequence of steps to set up and execute a simulation.
(mesh) part	a set of mesh elements and their closure assigned to a given process.
(mesh) partition	the set of parts forming a distributed mesh.
Ki	suffix to denote 2^{10}. So, for example, 16Ki is equal to $16 * 2^{10} = 16384$.
$l.N$	denotes the Nth line of an Algorithm or Listing.
`code`	C/C++/Fortran is written with a fixed-pitch font. For example, the function `printf(...)` is declared in `stdio.h`; the ellipsis represent omitted arguments.

The following notation describes the topological entities of geometric models, meshes, and partitions, the associations between the topological entities, and the distribution of mesh entities in a partitioned mesh [11].

V	the model, $V \in \{G,\ P,\ M\}$, where G is the geometric model, P is the partition model, and M is the mesh.
$\{V^d\}$	the set of dimension d entities in model V.
V_i^d	the i^{th} entity of dimension d in model V. $d = 0$ for vertex, $d = 1$ for edge, $d = 2$ for face, and $d = 3$ for region.
$\{M_i^d\{M^q\}\}$	a set of mesh entities of dimension q that are adjacent to M_i^d. For instance, $\{M_3^1\{M^3\}\}$ is a set of mesh regions adjacent to mesh edge M_3^1.
$M_i^d \sqsubset G_j^q$	the geometric classification indicating the unique association of mesh entity M_i^d with geometric model entity G_j^q, $d \leq q$.
$M_i^d \sqsubset P_j^q$	the partition classification indicating the unique association of mesh entity M_i^d with partition model entity P_j^q, $d \leq q$.
$w_p(M^d)$	the weight of mesh entities M^d on part p.
I_p^d	the weighted imbalance of dimension d mesh entities on part p; $w_p(M_i^d)/avg(w_{q=0..N-1}(M_i^d))$ where N is the total number of parts in the mesh.
I^d	the maximum imbalance of entity dimension d; $max(I_{p=0..N-1}^d)$.

CHAPTER 2
IMPROVING UNSTRUCTURED MESH PARTITIONS FOR MULTIPLE CRITERIA USING MESH ADJACENCIES

2.1 Introduction

Parallel simulation-based engineering workflows using unstructured meshes require adaptive methods to ensure reliability and efficiency [14]. Starting with a problem specification on a geometric model [15], [16], an effective workflow automatically executes parallel mesh generation [17], analysis, and analysis-based mesh [18], [19] and/or model [20] adaptation. The analyze-adapt cycle is repeated until a desired level of solution accuracy is reached. Between each step in the cycle is an opportunity to improve scalability and efficiency through dynamic partitioning.

Current dynamic load balancing methods do not effectively reduce imbalances to the levels needed by applications capable of strong scaling to the full size of leadership class petascale systems. This chapter presents a scalable approach that quickly reaches the required imbalance levels for multiple criteria by pairing Partitioning using Mesh Adjacencies (ParMA) with current partitioning methods.

A review of (hyper)graph, recursive sectioning geometric, and diffusive partitioning methods is located in Section 2.2. Section 2.3 introduces the dynamic partitioning problem and how our approach meets the requirements to solve it effectively. Section 2.4 reviews the partitioned mesh representation in the Parallel Unstructured Mesh Infrastructure (PUMI) [11], [21]. Sections 2.5 through 2.7 detail partition improvement procedures to support application specific partition requirements. Section 2.8 begins with a comparison of ParMA and its predecessor, LIIPBMod. Next, we present a ParMA feature comparison test and multi-criteria partitioning results on meshes with over 12 billion elements running on over one

Portions of this chapter have been submitted to: C. W. Smith, M. Rasquin, D. Ibanez, K. E. Jansen, and M. S. Shephard, “Improving unstructured mesh partitions for multiple criteria using mesh adjacencies,” submitted for publication.

million cores. Section 2.8 closes with a discussion of scaling improvement in a CFD analysis running on over a half-million cores.

2.2 Unstructured Mesh Partitioning

The dynamic partitioning problem seeks to quickly improve the load balance and reduce communication costs of an existing partition that is reasonably distributed; such as those generated by (hyper)graph and geometric partitioning tools. Hendrickson and Devine [22] define the requirements of dynamic partitioning as: (1) balance the computational work, (2) reduce the inter-processor communication costs, (3) modify the partition incrementally, (4) output the new communication pattern, (5) execute on parallel systems quickly, (6) consume small amounts of memory, and (7) provide an easy to use functional interface. For unstructured meshes these requirements are mostly satisfied by multi-level (hyper)graph and recursive sectioning methods [23]. Multi-level (hyper)graph methods are limited in scalability; memory requirements limit their effective usage on more than several thousand processors [8]. Recursive sectioning methods are limited in quality; they have lower memory and time requirements at the expense of increased inter-part surface area. Additionally, these methods can only balance one dimension of mesh entity. This approach can result in a less-than optimal balance of the other entity dimensions as process counts increase. The balance of the other dimensions can be improved, but not fixed, with carefully defined weights in the multi-constraint partitioning options provided by Zoltan's recursive coordinate bisection implementation and by the multi-level (hyper)graph methods [24]–[26]. Below, we review the graph, geometric sectioning, and diffusive partitioning approaches in more detail.

2.2.1 (Hyper)Graph Partitioning

Graph-based partitioning methods define an assignment of weighted graph nodes to k parts such that each part has the same total weight and the inter-part communication costs are minimized. A graph is constructed from an unstructured mesh by selecting one dimension of mesh entity (i.e., vertices, edges, faces, or regions) to define graph nodes, and one mesh adjacency between the selected entity dimension to define graph edges. At a higher level, the goal of this selection is to represent a work unit with the graph node and an information dependency between two work units by a graph edge. 3D element-based finite element and finite volume codes typically select mesh regions for graph nodes and mesh faces shared by elements for graph edges. This selection results in the unique assignment of mesh regions to parts, which enables efficient local execution of element-level computations [1].

Parallel, multi-level, graph-based partitioning methods produce high quality partitions with tens of thousands of parts in a fraction of the time needed by most analysis procedures [7], [26]–[28]. One approach to generalize these methods to represent more complex information dependencies uses hypergraphs. A hypergraph is defined as a set of weighted nodes and hyperedges. Hyperedges differ from graph edges in that they represent dependencies between multiple graph nodes and, in doing so, have the ability to better model the communication costs of an application [29], [30]. As with graph-based partitioning, the goal of hypergraph partitioning is to balance the node weight across the k parts while minimizing a hyperedge-based objective function. Boman and Devine propose constructing the hypergraph from an unstructured mesh by creating one hypergraph node for each mesh region (in 3D), as is done in the graph-based construction, and a hyperedge connecting the mesh regions bounded by each mesh vertex. This richer representation improves the modeling of communication costs, but results in algorithms that are more compute and memory intensive relative to graph-based methods.

2.2.2 Geometric Partitioning

Geometric methods represent information via spatial coordinates, and relations via distance; the closer two pieces of information are the stronger their relation. The exclusive use of coordinate information significantly reduces the memory requirements of these methods relative to (hyper)graph methods that rely on topological relations [8]. Along with the lower memory cost, the spatial sorting procedures used by geometric methods are also computationally cheaper than the topological traversals needed by graph methods. The lower computational and memory usage costs come at the expense of significant increases in inter-part communications [31].

For applications that require frequent balancing though, the resulting communication overheads may be offset by the time saved computing the partition [8].

Geometric recursive sectioning methods can quickly compute well-balanced partitions for a single entity dimension [32]–[36]. Recursive coordinate (RCB) [37] and inertial bisection (RIB) [35], [36], [38] methods recursively cut the parent domain; RCB along a coordinate axis and RIB perpendicular to the parent domain's principal direction. Multi-sectioning techniques [33], [34] can be considered extensions of the recursive coordinate bisection methods as they define cuts along coordinate axis, but do so with multiple parallel cut planes at each recursion.

Partitioning methods using space-filling curves (SFC) produce partitions of similar quality to RCB and RIB. For 3D unstructured meshes Hilbert [39] and Morton curves have been used effectively by the Zoltan [40] and SPartA [8] packages, respectively. Given the simplicity of SFC partitioning methods (encoding, sorting, then splitting) a high degree of on-node and inter-node concurrency is possible. For example, a constant time Hilbert curve encoding procedure (spatial coordinates to curve position) [39] and its subsequent sorting has been demonstrated on shared-memory devices using a data-parallel implementation [5] and a two-collective splitting approach is used by SParTA. As an added benefit, sorting provides a cache efficient layout of the mesh entities for subsequent mesh-based operations that benefit from topological locality [41], [42].

2.2.3 Diffusive Partitioning

Diffusive partitioning methods efficiently improve an existing partition by transferring load between neighboring parts. Load transfer can be coordinated globally or locally. Global load transfer selects elements to minimize either the total weight of transferred elements, or the maximum weight transferred in to or out from a part [43]–[49]. Alternatively, local load transfer iteratively moves elements from heavily loaded to less loaded parts [47], [50]–[52]. This approach can have significantly lower overall computational costs if the total amount of transferred load is controlled. Control is typically exerted through greedy heuristics. These heuristics first determine the amount of load to transfer between neighboring parts, and then select elements to satisfy the transfer requirement. Fiduccia [53] and Kernighan [54] proposed selecting elements based on the subsequent part quality improvement. For partitioning complex graphs with up to one trillion edges [55], [56] these heuristics have proven successful as part of a label propagation based approach. Likewise, a greedy improvement heuristic is applied to reduce the communication cost of parallel sparse matrix-vector multiplication [57]. In Zhou's work on unstructured meshes a similar heuristic is shown to be highly scalable given a distributed mesh representation [9], [10].

2.3 Dynamic Partitioning Requirements

The dynamic partitioning problem seeks to quickly improve the load balance and reduce communication costs of an existing partition that is reasonably distributed; such as those generated by (hyper)graph and geometric partitioning tools. Hendrickson and Devine [22] define the requirements of dynamic partitioning as: (1) balance the computational work, (2) reduce the inter-processor communication costs, (3) modify the partition incrementally, (4) output the new communication pattern, (5) execute on parallel systems quickly, (6) consume small amounts of memory, and (7) provide an easy to use functional interface. For unstructured meshes these requirements are mostly satisfied by multi-level (hyper)graph and recursive sectioning methods [23]. Multi-level (hyper)graph methods are limited in scalability; memory requirements limit their effective usage on more than several thousand processors [8]. Recursive sectioning methods are limited in quality; they have lower memory and time requirements at the expense of increased inter-part surface area. Additionally, these methods can only balance one dimension of mesh entity. This approach can result in a less-than optimal balance of the other entity dimensions as process counts increase. The balance of the other dimensions can be improved, but not fixed, with carefully defined weights in the multi-constraint partitioning options provided by Zoltan's recursive coordinate bisection implementation and by the multi-level (hyper)graph methods [24]–[26]. Below, we review the graph, geometric sectioning, and diffusive partitioning approaches in more detail.

2.4 Partitioned Mesh Representation

PUMI provides the $O(1)$ queries of intra- and inter- part mesh topology information needed by ParMA via a complete and distributed mesh representation [11], [21]. The distributed mesh is the union of mesh parts. A mesh part is defined as a collection of mesh faces M^2 in 2D, and regions M^3 in 3D, assigned to a processing resource, typically a core or hardware thread. Mesh entities are denoted as M_i^d, where d specifies the dimension and i specifies the id or index. At the shared boundary of two or more parts mesh entities are copied (as shown for mesh vertex M_0^0 and edge M_0^1 in Fig. 2.1) and locally tracked on each part through a remote copy object. Distributed mesh operations involving a mesh entity on the part boundary are coordinated through an ownership protocol; depicted by the discs and bold segments in Fig. 2.1.

Two parts with common boundary mesh entities are neighbors. Sets of mesh entities sharing common neighboring parts form a partition model entity [58]. Like mesh entities, we denote the ith partition model entity of dimension d as P_i^d. A mesh entity is classified on the partition model entity of equal or greater dimension which bounds it. For example, in Fig. 2.1 mesh vertex M_0^0 is classified on the partition model vertex P_0^0, mesh edge M_0^1 is classified on the partition model edge P_1^1, and mesh face M_0^2 is classified on the partition model face P_2^2. These classifications are respectively noted as $M_0^0 \sqsubset P_0^0$, $M_1^1 \sqsubset P_1^1$, and $M_0^2 \sqsubset P_2^2$. Information is exchanged by neighboring parts, typically for synchronizing data associated with part boundary entities, through non-blocking, collective, neighborhood communications provided by PCU [12], [59]. Using these communications, PUMI also provides procedures to efficiently move mesh elements between processors; referred to as migration.

2.5 Partition Improvement

ParMA reduces the peak imbalance of multiple entity dimensions by iteratively migrating some mesh elements from heavily loaded parts to neighboring parts with less load. The entity dimensions to balance are defined by an application specified priority list. For example, if element>vertex is specified then the algorithm prioritizes improvements to element balance over vertex balance. The greater-than

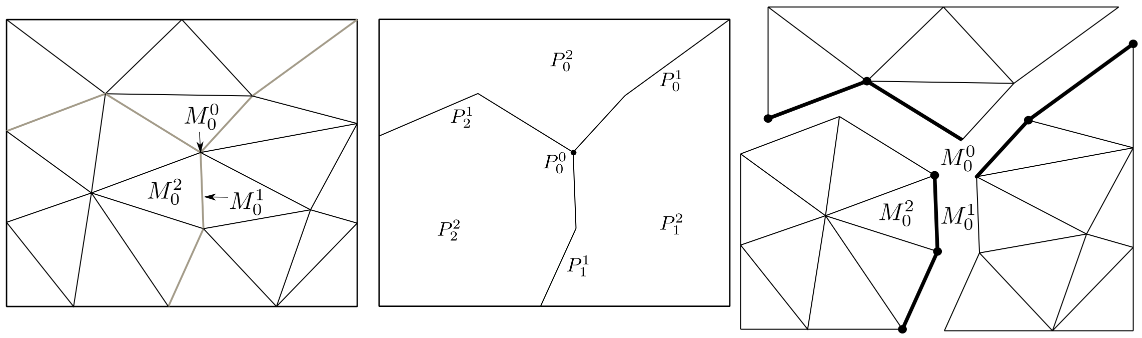


Fig. 2.1. (left) Example of a mesh, (middle) its partition model, and (right) its ownership. Discs and bold segments denote entity ownership.

relation indicates that element balance improvements are allowed to degrade the vertex balance, but vertex balance improvements cannot degrade the element balance. The balance of unlisted entity dimensions (edges and faces in this example) are not considered and may be degraded. If vertex=element is specified, then the algorithm considers the balance of mesh elements and vertices equally important. In this case, the lower-dimension entities are processed first as improvements to their balance tends to improve the balance of the entities they bound (higher-dimension entities). The target imbalance for each listed entity dimension is specified by the application as $tgtImb^d$ where $d \leq d_{max}$ (the maximum dimension entity in a mesh). Applications which perform work on entities regardless of their ownership define the imbalance of a part, I_p^d, as the weight of mesh entities of dimension d existing on part p divided by the average weight of dimension d entities per part. The weight of a mesh entity is set to one when it is not specified by the application. The maximum imbalance of dimension d entities across all parts is noted as I^d.

The ParMA iterative diffusion procedure is summarized in Algorithm 1. This process is repeated for each specified entity dimension in order of descending priority, as described above. For simplicity, the pseudo code is written with only a single entity dimension, d, being passed to the supporting procedures. In practice though, we have the list of higher priority entity dimensions to avoid disturbing the imbalance of the higher priority entities during the balancing of the current, lower priority, entity dimension. Iterations are stopped on line 9 if the target imbalance ($tgtImb^d$) is reached, or they are stopped on line 10 if no migration opportunities remain (discussed in Section 2.7.1), or if a maximum number of iterations is reached. Each diffusive iteration has four steps [50]. First, on line 2, neighboring parts exchange local information (e.g., the weight of mesh entities) using PCU. Next, each part determines how much load needs to be migrated and where it needs to go on line 3, targetting, and then marks elements for migration on line 4, selection. Before migration is executed, on line 5, each part determines if too much weight is being sent to it, and, as necessary, cancels a portion of the incoming element migrations. The cancellation process is detailed in subsection 2.7.0.2. The final step, migration, moves the marked elements to their defined destinations using PUMI.

Algorithm 1 ParMA Load Balancing

```
1: procedure RunStep( (in/out) $mesh$, (in) $d$)
2:     ComputeAndExchangeWeights( (in) $d$, (out) $weight$, (out) $neighborWeights$)
3:     Targeting( (in) $mesh$, (in) $weight$, (in) $neighborWeights$, (out) $targetWeights$)
4:     Selection( (in) $mesh$, (in) $d$, (in) $targetWeights$, (out) $migrationPlan$)
5:     Cancellation( (in) $mesh$, (in) $neighbors$, (in/out) $migrationPlan$)
6:     Migration( (in/out) $mesh$, (in) $migrationPlan$)
7: procedure Balance( (in/out) $mesh$, (in) $dimensions$)
8:     for all $d \in dimensions$ do
9:         while imbalance of $d >$ tolerance do RunStep( (in/out) $mesh$, (in) $d$)
10:            if Balancing Stagnates then
11:                break
```

The targeting and entity selection steps are detailed in the following sections.

2.6 Targeting

ParMA defines the load transfer requirements for balancing a given entity dimension based on the relative weight of the entities in neighboring parts. Parts with an entity imbalance, I_p^d, greater than the specified imbalance, $tgtImb^d$, are defined as heavily loaded parts. A lightly loaded part is defined based on the partition improvement requirements. If the application requires vertex=edge>element then migration to decrease element imbalance should not increase the imbalance of vertices or edges. Thus, during element improvement a part is a 'lightly loaded' target to receive elements if it has fewer vertices, edges and elements than the heavy part.

The amount of load, l^d_{pq}, migrated from a heavily loaded part p to a neighboring part q during improvement of mesh entities of dimension d is defined as

$$l^d_{pq} = \alpha * sf * \left(\sum_i w(M^d_i \in p) - \sum_i w(M^d_i \in q) \right), \tag{2.1}$$

where $w(M^d_i)$ is the application specified weight associated with a given entity i, α is a diffusion rate limiting constant $\in (0, 1]$ [51], and, in 3D, sf is the ratio of mesh faces shared by parts p and q to the total number of faces classified on partition boundaries of p. The surface area bias sf helps define load transfer requirements that can be satisfied in a single iteration by selecting elements for migration that are classified on the part boundary. A large transfer across a small boundary will not only take several iterations to satisfy, it will also lead to a large increase in the number of entities classified on the part boundary as each iteration will 'tunnel' into the part. The entity selection process is detailed in Section 2.7.

We tested the effect of α on run time and imbalance to guide the choice of a conservative default value. The test mesh of the automotive part shown in Fig. 2.2 has 2048 parts and an initial vertex imbalance of 46%. Table 2.1 and Fig. 2.3 respectively show the run time and vertex imbalance as α is varied from 0.2 to 1.0. The target vertex imbalance was set to 5%. With the exception of the $\alpha = 1.0$ case, all the cases reached an imbalance of 6% or 7% before stagnation detection stopped the vertex balancer (Section 2.7.1). Setting α to 0.6 yields the fewest iterations and the shortest run time. Increasing α from this value causes too many elements to be migrated in each iteration, which results in imbalance oscillations that increase the run time. Similarly, lower values of α increase the run time by migrating too few elements in each iteration. Given these observations, α is conservatively set to 0.5 for the remaining tests in this work. Note, this setting of α may be tuned for a specific case to improve performance.

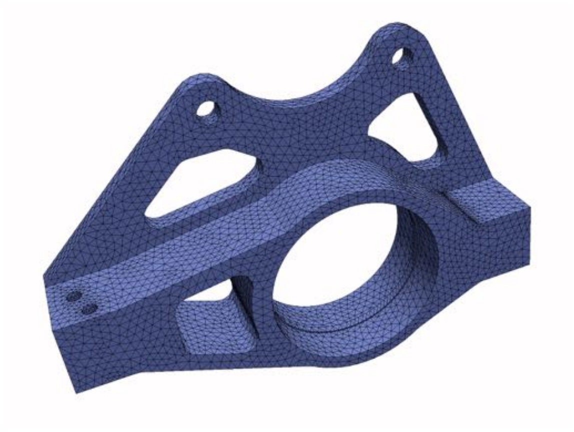


Fig. 2.2. Coarse mesh of the 2014 RPI Formula Hybrid suspension upright.

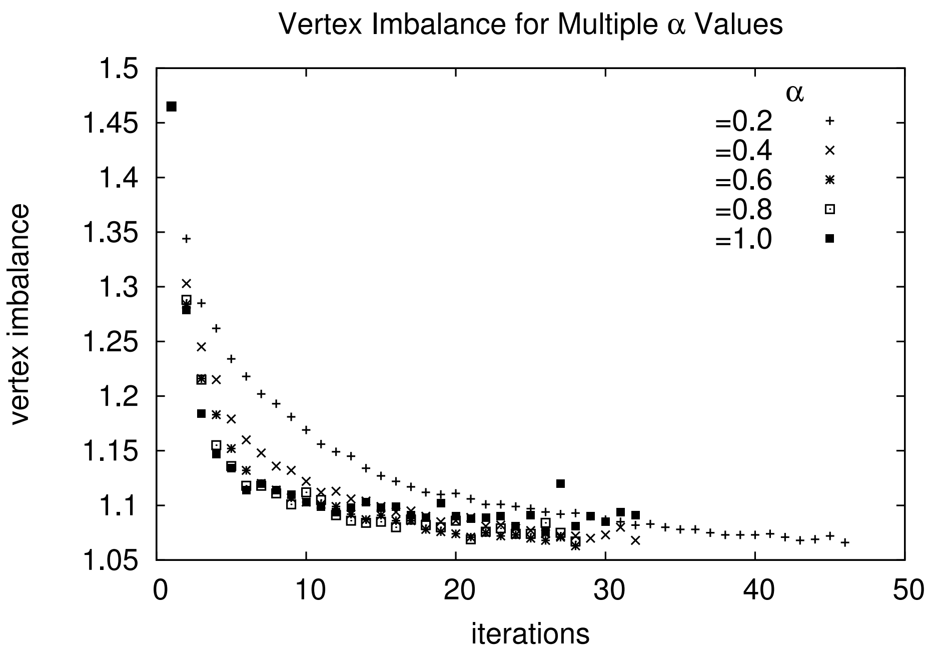


Fig. 2.3. Effects of α on the number of iterations, and vertex imbalance. The initial mesh has 2048 parts and a 46% vertex imbalance.

Table 2.1. Diffusion iterations and run time for various α settings.

α	iterations	time (s)	I^0
0.2	45	19.3	1.07
0.4	31	14.0	1.07
0.6	27	12.5	1.06
0.8	27	13.5	1.07
1.0	31	16.6	1.09

Compared to Zhou's LIIPBMod, ParMA's use of Equation 2.1 enables finer grained migrations. In LIIPBMod, a part is a target for migration if (1) the difference between the vertex imbalance of the source part and the target part is greater than 2% or (2) the vertex imbalance is less than 4.5%. Note, LIIPBMod does not support weights associated with mesh vertices.

2.7 Entity Selection

Entity selection's primary objective is to reduce the imbalance of a given entity dimension. While selecting mesh elements for migration it is important to maintain inter-part boundaries with low surface area as an increase in the number of mesh entities classified on boundaries increases application communications, and in some cases, also the computational load [60]. Thus, entity selection's secondary objective is to reduce the number of mesh entities classified on partition model entities of dimension $d < d_{max}$.

Entity selection satisfies the objectives with part-level and entity-level heuristics. In Section 2.7.0.1 we describe how the part-level heuristic defines a vertex traversal order for evaluating the entity-level heuristic. Next, in Section 2.7.0.2, we describe how the entity-level heuristic evaluates the topology of a cavity; the set of elements adjacent to a given vertex. Combined, these two procedures reduce both the surface-to-volume ratio of the parts and their entity imbalance. Pseudo code for the selection procedure, as called in Algorithm 1, is given in Algorithm 2 and described in the following sections.

Algorithm 2 ParMA Selection

```
1: procedure SELECTION( (in) $mesh$, (in) $d$)
2:     if $dist$ not set then
3:         IDENTIFYDISCONNECTEDCOMPONENTS((in) $mesh$, (out) $comps$)
4:         SETVERTEXCOMPONENTIDS((in) $mesh$, (in) $comps$, (out) $ids$)
5:         FINDTOPOLOGICALCENTERS((in) $mesh$, (in) $comps$, (out) $centers$)
6:         COMPUTECOREDISTANCE((in) $mesh$, (in) $ids$, (in) $centers$, (out) $dist$)
7:         OFFSETCOREDISTANCE((in) $mesh$, (out) $dist$)
8:     else
9:         UPDATEDISTANCE((in) $mesh$, (in/out) $dist$)
10:    for $cavSize \in \{2, 4, 6, 8, 10, 12\}$ do
11:        CREATETRAVERSALQUEUE((in) $mesh$, (in) $dist$, (out) $q$)
12:        for all $v \in q$ do
13:            if SHOULDMIGRATECAVITY((in) $mesh$, (in) $v$, (in) $cavSize$) then
14:                Add cavity of $v$ to $migrationPlan$
```

2.7.0.1 Part-level Core Distance Heuristic

The number of mesh entities classified on partition model entities is reduced by migrating elements that are furthest from the topological center of the part, referred to as ”the core”. To find these elements we traverse the part boundary vertices in order of their distance from the core. We define this distance as the shortest edge-based path between a vertex and the part’s core. Thus, as diffusive iterations are executed, elements bounding vertices far from the core are migrated and the maximum distance of the part is reduced [61], [62]. This approach satisfies the second entity selection objective by forming parts with lower surface to volume ratios and reduced communications. In Algorithm 2 the core is found on line 5 and the distance is computed on line 6.

To understand the distance computation procedures, we must first account for parts produced by the graph and geometric partitioning that have multiple connected components. We define a (connected) component of a part as the set of elements in which there exists a path via M^{d-1} adjacencies (faces in 3D) between any two elements. Given this complexity, we first identify the components (lines 3 and 4 of Algorithm 2), compute the distance in each component (lines 5 and 6 of Algorithm 2) , and then offset the component distances to ensure a strictly increasing ordering for the traversal of boundary vertices (line 7 of Algorithm 2); we want the traversal to process the entire boundary of one component before moving on to the next one. The remainder of this subsection defines these procedures.

Connected components are identified via a breadth-first M^{d-1} adjacency-based traversal [63] starting at the first mesh element in the part (based on iterator ordering). As elements are visited, they are marked with the component id. When there are no more unmarked M^{d-1} adjacent elements to visit, the component id is incremented and the traversal is restarted with an unmarked element in another component. This process is repeated until all elements in the part are marked with a component id.

By traversing M^{d-1} mesh adjacencies between elements we have identified components with the strongest topological connectivity. But, to compute the core distance at mesh vertices, we first need to uniquely assign vertices to components. For vertices bounded by elements with the same component id the assignment is obvious. The problem comes with vertices at the common boundaries between components formed by lower dimension topological mesh adjacencies (i.e., an edge or vertex adjacency). To resolve this assignment issue, we set the vertex id to the lowest bounding component id. Now that vertices have component ids, we can find the vertices at the topological center of each component.

We find the central vertices in a component via a breadth-first traversal starting from all the boundary vertices of a component. When there are no more vertices to visit the traversal ends. From the set of vertices with the largest traversal depth, the first (based on vertex iterator ordering) is chosen as the component's core. The left half of Fig. 2.4 shows the vertices marked with their traversal depth. Note that selecting a different vertex with a depth of three could reduce the maximum distance to any boundary vertex, thus representing a more central vertex, and result in a small improvement to the subsequent boundary traversal. From the central vertices Dijkstra's algorithm [64] is run to compute the core distance to all other vertices in the component. The core distance at each vertex is shown in the right half of Fig. 2.4. A more complex example of distancing is shown in Fig. 2.5.

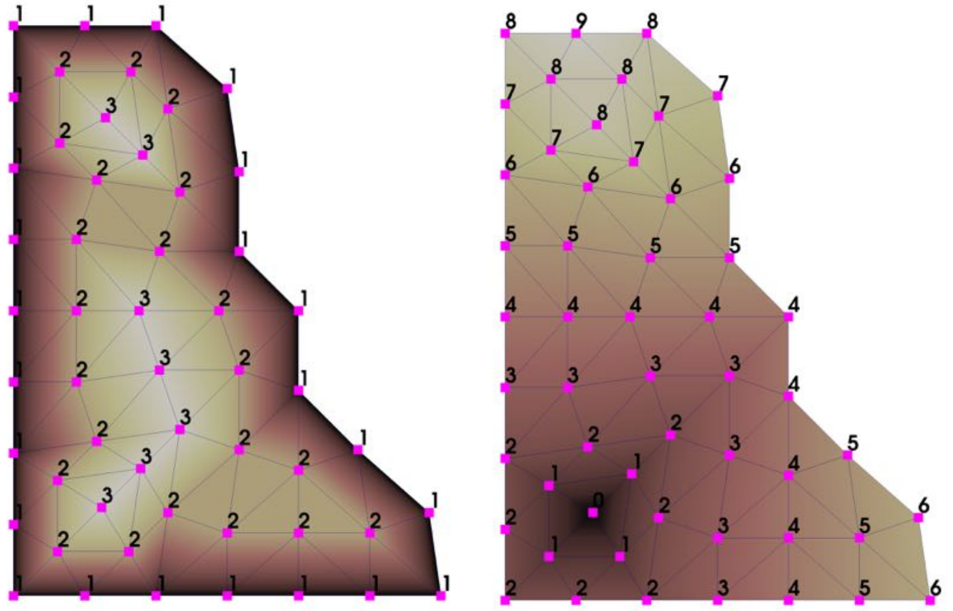
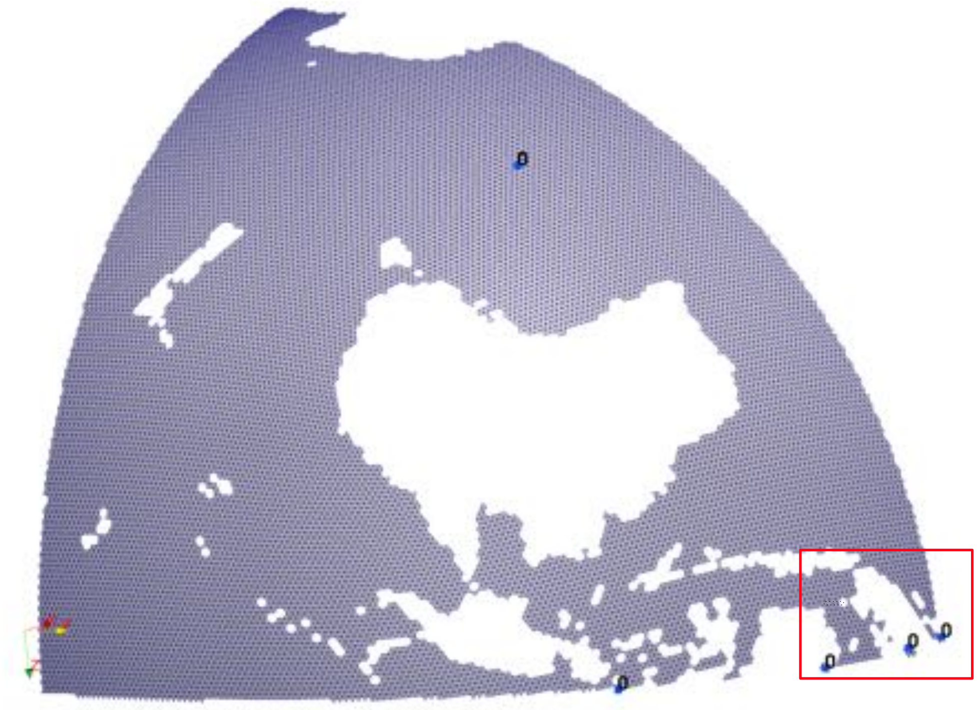
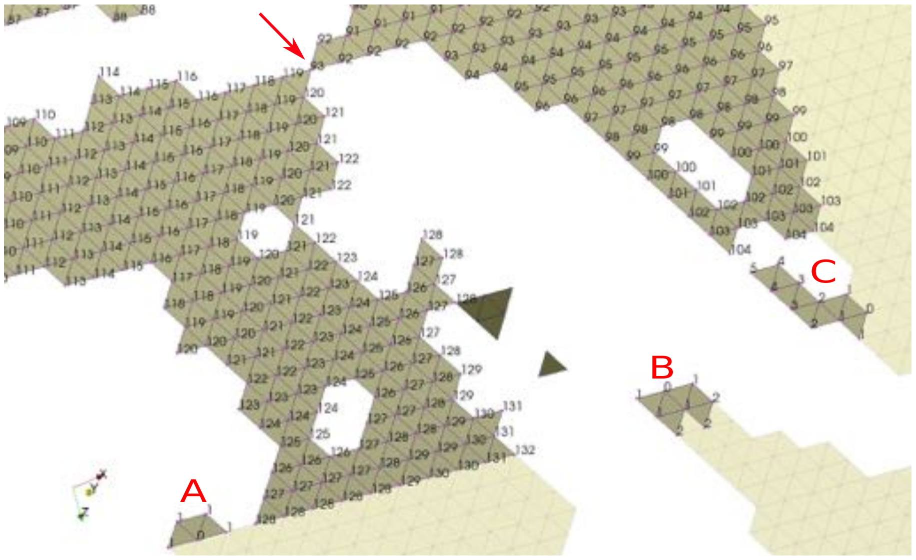


Fig. 2.4. (left) The distance from each vertex to the boundary and (right) the distance from the core vertex (marked with a zero near the bottom left corner).



(a) Component core vertices marked with a zero.



(b) An edge-disconnected junction (arrow) and three disconnected components (A, B, C).

Fig. 2.5. Components in one of four parts of the MPAS 60km [65] ocean mesh. Dark shaded elements are isolated (no M^{d-1} adjacency path to elements on the part boundary) and light shaded elements are on a different part.

Now that all components have vertices with distance, we must offset the distances so that our element selection procedure can traverse all the boundary vertices of a component before moving to another component. In Algorithm 2 the offset is computed on line 7. Fig. 2.6 depicts the distance of the disconnected components before and after the offset is applied. Algorithm 3 computes the component vertex distance offsets. The procedure begins by sorting the components in order of descending depth; forming the list c. Next, on line 2, the deepest component, r_0, has its offset set to zero. Lines 3 through 4 then compute the offset of the i^{th} component, r_i, by summing the previous component's offset and maximum distance, $r_{i-1} + max(R(M_j^0 \in c(i-1)))$, (where $R(M_j^0)$ is the distance of a vertex), plus an upper bound on a component's distance increase, $maxDistIncrease$. This upper bound enables fast distance updates by including a buffer into the offset that allows the parts to grow during diffusion iterations without overlapping. As each diffusion iteration can only add one layer of elements to a component, the maximum growth in distance for a component is bounded by the number of iterations. So, $maxDistIncrease$ is set to the maximum number of diffusive iterations. The final step on line 5 loops over the components in ascending order of their depth and applies the offset to their vertices. This component traversal order, combined with the conditional checking that the current distance value is less than the offset, prevents the distance of vertices on the boundary of two components being offset multiple times.

Algorithm 3 Vertex Distance Offset

```
1: c ← sortDescending(components)
2: $r_0 \leftarrow 0$ //component zero's offset
3: for $i \leftarrow 1, numComponents$ do
4:     $r_i \leftarrow r_{i-1} + max(R(M_j^0 \in c(i-1))) + 1 + maxDistIncrease$
5: for $i \leftarrow numComponents, 1$ do
6:     for $M_j^0 \in c(i)$ do
7:         if $R(M_j^0) < r_i$ then
8:             $R(M_j^0) \leftarrow R(M_j^0) + r_i$;
```

Within a component, detection of non-manifold [66] portions of the boundary is critical to ensure that the core distance accurately records the shortest M^{d-1}

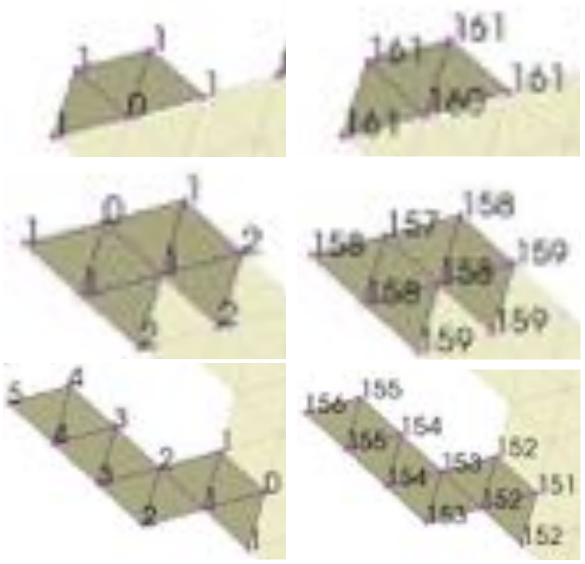


Fig. 2.6. Core distance of disconnected components A, B, and C from Fig. 2.5 (left) before, and (right) after, the offset is applied.

adjacent path from the core to each vertex. For example, consider the 2D non-manifold vertex junction indicated by the arrow in Fig. 2.5b. Here the paths from the core vertex marked in the upper portion of Fig. 2.5a to either side of the junction will have significantly different lengths due to the large holes in the mesh formed by land masses. Detection of a non-manifold junction at a given boundary vertex, s, is through the breadth-first traversal of s's cavity vertices (i.e., the vertices bounding elements in the cavity), rooted at the distance-1 parent of s. Vertices in the cavity are reachable via M^{d-1} adjacencies if the traversal can visit them without passing through s. For example, consider vertex s in the cavity depicted in Fig. 2.7 to have the lowest distance in the priority queue of vertices being processed by Dijkstra's algorithm. The detection traversal starts at vertex p, the parent of s, by enqueuing vertices f and h. s is also edge-adjacent to p, by definition, but it is skipped as paths through it are not considered. The traversal continues by dequeuing a vertex and enqueuing its edge-adjacent vertices that have not been previously visited and are not s. Fig. 2.7 depicts the depth of each edge in the traversal tree with hash marks. If there existed another element that was adjacent to s that was also adjacent to e and d, or h and b, then the edge (e, d) or (h, b) would provide an edge-adjacent path from p to b, c and d and the junction would be identified as manifold.

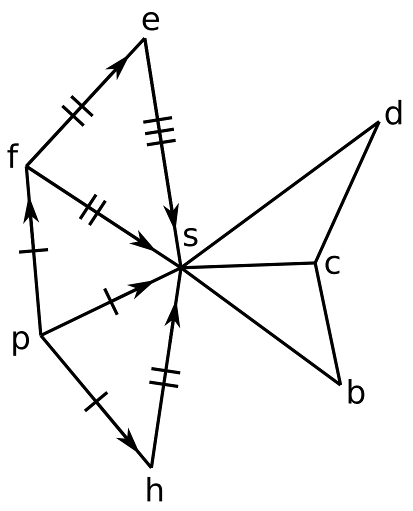


Fig. 2.7. Determining if s is a non-manifold component boundary by creating a cavity of elements bounded by s, and then trying to walk from its parent vertex p to b, c, or d without going through s. The hash marks indicate the depth of each edge visited in the walk.

Compared to LIIPBMod, our part-level heuristic supports improvement of lower quality partitions by directly accounting for connected components, and non-manifold junctions within components.

In LIIPBMod, the boundary vertices are iterated over based on the order they appear in the underlying data structure without consideration for the part topology.

2.7.0.2 Entity-level Cavity Heuristics

In the previous subsection we described how the part-level heuristic defines a vertex traversal order for evaluating entity-level heuristics. In this subsection, we define those entity-level heuristics and how they select elements for migration to reduce the entity imbalance. We start by describing size-based cavity selection. Next, we describe and demonstrate how multiple boundary traversals with increasing cavity size limits benefit partition improvement. In Algorithm 2 these steps are listed on lines 10 through 13. Lastly, we detail cancellation; a critical mechanism for multi-criteria load balancing.

Our entity-level, gain-like heuristic [53], [54] is based on Zhou's cavity-based approach [9], [10], but is more flexible. Like LIIPBMod, we check the number of elements in the cavity (the set of elements adjacent to a vertex on a part boundary), but we also check the adjacencies within the cavity, and the on- and off-part adjacencies external to the cavity. With this additional information we can migrate cavities that are bounded by vertices classified on partition model vertices, edges, and faces. LIIPBMod's heuristic avoided multi-part junctions; any cavity whose bounding vertex is classified on a partition model vertex or edge was not eligible for migration. In addition to more flexible migration, our heuristics improve the selection quality with (1) multiple boundary traversals with increasing cavity size in a single iteration, and (2) support for migrations to be canceled by the receiver.

The primary check for selection is based on the number of elements in a cavity. If a cavity is small, then migrating it will decrease the number of entities in the source part and classified on partition model entities. Conversely, migrating a face-connected cavity (i.e., between any two elements in the cavity there exists a path via face adjacencies) with several elements can result in an increase in the number of mesh entities classified on partition model entities. However, migrating small cavities with a few disconnected elements can yield significant entity reductions. Note, LIIPBMod uses a fixed cavity size of five elements.

To illustrate the effect of size and connectivity on entity reductions consider the cavities depicted in Fig. 2.8 and the reductions listed in Table 2.2. Fig. 2.8 (a-c) and (d-f) respectively depict face-connected and face-disconnected cavities. Here, the vertices bounding the cavities are marked with a disc. Vertices classified on the partition model face P_j^2 bounded by parts P_0^3 and P_1^3 are marked with a circle or disc, and in (c) a vertex classified on a partition model region, $M_i^0 \sqsubset P_0^3$, is marked with a square. In this example, all elements are migrated from P_0^3 to P_1^3. After migration the faces bounded by the circled vertices are now on the part boundary between P_0^3 to P_1^3, and in cavities (a-b) and (d-f) there is one less vertex classified on P_0^3. Migration of cavity (c) does not change the number of entities classified on the part boundary since there is an entity added to the part boundary for each one migrated.

Ideally, we would like to select the combination of cavities for migration that results in the greatest imbalance reductions. Solving this problem exactly would be expensive, so instead, we iterate over the part boundary multiple times in order of

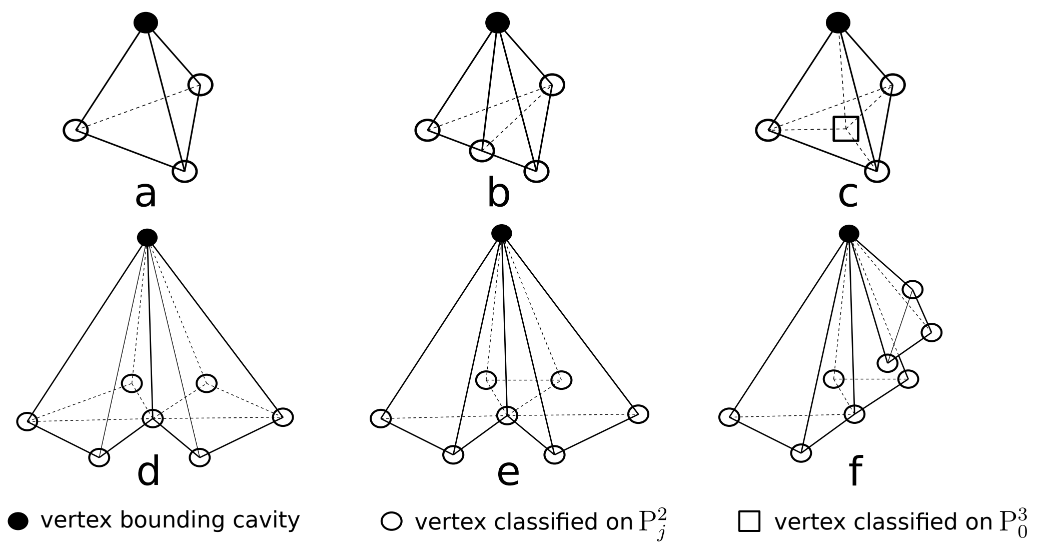

a
b
c
d
e
f
● vertex bounding cavity
○ vertex classified on P_j^2
□ vertex classified on P_0^3

Fig. 2.8. Vertex bounded cavities being migrated from part 0 to 1.

Table 2.2. Reduction in the number of mesh entities classified on P_j^2 for cavities (a-f) depicted in Fig. 2.8.

	Cavity					
Entity Dimension	a	b	c	d	e	f
Vertex	1	1	0	1	1	1
Edge	3	2	0	5	7	8
Face	2	2	0	4	6	6

descending vertex distance while relaxing (increasing) the cavity size selection limit before executing the PUMI element migration procedure. Thus, the first traversal of the boundary will select only cavities with one or two elements, followed by cavities with less than four elements in the second traversal (the first traversal may have created new one or two element candidates), and so on. The traversal stops at a cavity size limit of 12; roughly half of the average number of elements adjacent to a vertex in a tetrahedral mesh [67].

We tested the effectiveness of selection with an increasing size limit versus a static size limit by balancing a small test mesh. For both approaches the cavity size limit is set to 12. The test mesh of the suspension upright has 228 thousand elements and is partitioned to 2048 parts using RIB. The RIB partition has a perfect element balance and a 53% vertex imbalance. Our runs with vertex balancing ParMA targets a 5% vertex imbalance. Balancing with the increasing cavity size limit requires 2.0 seconds on 2048 Blue Gene/Q cores. At the end of the run, the target vertex imbalance is reached, the element imbalance is 9%, and the average number of vertices per part is reduced by 3.4%. On the same number of cores, the fixed cavity size run takes 3.4 seconds to reach the target vertex imbalance and has a 15% element imbalance, and a slight (0.07%) increase in the average number of vertices per part.

Once a cavity is selected, it needs to be assigned to a neighboring part for migration. The assignment and subsequent migration should result in a reduction of the number of mesh entities classified on the part boundary. In a 3D mesh we assign the cavity to the part that shares the most mesh edges with it. Counting shared edges avoids counting vertices (the lowest dimension shared entity) that are not adjacent to a higher dimension shared entity (an edge or a face) while providing more information than the counting of shared faces (the highest dimension shared entity, in 3D). Fig. 2.9 depicts a two element cavity with entities classified on both partition model faces and edges. Specifically, the cavity has two faces shared with part one (dark shaded), two faces with part two (unshaded), and an additional classification of edge F on the partition model edge shared with part two (dashed line in bold). Counting shared edges correctly identifies part two as the destination; it has six cavity edges versus part one only having five. The 'Sum' row of Table 2.3 lists the total cavity edge count on each part when the cavity elements are owned by part zero, the initial owner, and parts one and two, the two possible target parts. For this example, migrating the cavity to part two reduces the total number of shared edges from 20 to 19; if part one were selected the total number of shared edges would increase by one. If multiple parts are tied for the most shared edges then the first part with remaining capacity is selected as the destination.

As the part boundary is traversed and the cavity heuristic selects elements for migration, the weight of the selected entities is tracked to prevent migrating too much weight to the target parts. Tracking is based on the simple rule, rooted in the unique assignment of elements to parts, that an entity will not exist on the part if all the elements it bounds are marked for migration. Thus, the weight tracking mechanism checks for this condition, and if satisfied, adds the entities weight to the running total for the given destination part.

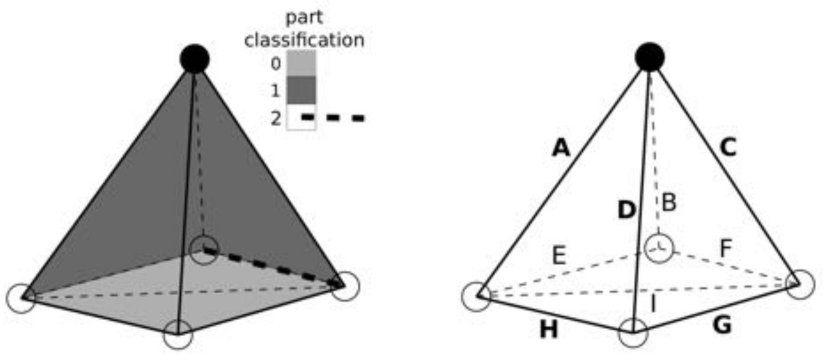


Fig. 2.9. Counting mesh entity partition model classification to select either part one or part two for the migration of part zero cavity elements. The cavity bounding vertex is marked with a disc. (left) Part classification; faces in the foreground classified on part 2 are not shaded. (right) Mesh edge labels. For clarity, edges in the foreground have bold labels.

During the balancing of lower priority entity dimensions (e.g., elements during vertex > element balancing) the imbalance of higher priority entity dimensions is preserved by canceling the migration of some elements [47]. First, the sending parts determine how much weight associated with higher priority entities is migrated to the target parts. These weights are then sent to the respective targets using PCU's neighborhood communication procedures [12]. The target part then iterates over the incoming migration requests in descending order of the migration weight, accepts the request if capacity remains, reduces the remaining capacity accordingly, and sends the accepted weight to the sender. The sending part then traverses the list of migration elements in the order they were selected (i.e., descending distance from the parts topological core), and keeps elements in the list until the peer's higher priority entity weight capacity is exceeded. A summary of the interaction between the part-level and entity-level heuristics is given in Section 2.7.2.

2.7.1 Stagnation Avoidance

A stagnation [9] avoidance procedure stops execution of diffusion when the imbalance or part shape has not improved over several iterations. Specifically, a second order accurate backward finite difference [68] approximates the rate of change

Table 2.3. Existence of Fig. 2.9 cavity edges on parts. The column groups list the edge existence prior to migration of the cavity (Owner=0), and after migration to part N (Owner=N). An entry is '1' if the edge exists on the part. The last row lists the total number of cavity edges on each part.

		Cavity Owner								
		0			1			2		
	Part	0	1	2	0	1	2	0	1	2
Edge	A	1	1	1	0	1	1	0	1	1
	B	1	1	0	0	1	0	0	1	1
	C	1	1	1	0	1	1	0	1	1
	D	1	0	1	0	1	1	0	0	1
	E	1	1	0	1	1	0	1	1	1
	F	1	1	1	1	1	1	1	1	1
	G	1	0	1	1	1	1	1	0	1
	H	1	0	1	1	1	1	1	0	1
	I	1	0	0	1	1	0	1	0	1
	Sum	9	5	6	6	9	6	5	5	9

of the imbalance, *imb*, and the average number of boundary mesh vertices per part, *sides*. Diffusion is stopped if the rate of change in *imb* is less than one percent of the target imbalance and the change in *sides* is less than one-hundredth of the initial *sides*.

2.7.2 Time Complexity

The part-level heuristic requires first executing an $O(|M^d| + |M^{d-1}|)$ element-based, breadth-first traversal to identify disconnected components. Next, the vertex component ids are set, $O(|M^0|)$, and boundary vertices are inserted into STL sets, $O(|M^0|log|M^0|)$. The component vertices are then traversed in breadth-first order via edge adjacencies to locate the topological center, $O(|M^0|+|M^1|)$. For simplicity, our implementation uses an STL set to maintain the vertices at each tree-depth. This choice adds $O(|M^0|log|M^0|)$ to the cost and could be avoided with a list-based traversal. Next, Dijkstra's algorithm is run to compute vertex distances, $O(|M^1| + |M^0|log|M^0|)$. As the vertices are visited, adjacent elements are accessed for non-manifold topology detection; a cost increase of $O(|M^d|)$. Lastly, the vertex distances are offset, $O(|M^0|)$. The overall complexity of the part-level heuristic for a 3D tetrahedral mesh is $O(|M^1| + |M^2| + |M^3| + |M^0|log|M^0|)$. But, for each entity dimension being balanced these procedures only need to be executed once.

In subsequent iterations we can execute a lower cost distance update on just the boundary vertices (line 9 of Algorithm 2).

In each iteration the entity-level heuristic first requires building a STL map-based distance queue of vertices to traverse, $O(|M^0|log|M^0|)$. The vertices in the queue are then traversed, $O(|M^0|)$, and cavities constructed by adjacent element queries, $O(|M^d|)$. Lastly, the cavity edges are queried for determining the destination part, $O(|M^1|)$. Thus, the entity-level heuristic's complexity is $O(|M^0|log|M^0|+|M^1|+|M^d|)$.

A detailed analysis of convergence and overall time complexity of general diffusive load balancing procedures can be found in the work of Subramanian [50] and Berenbrink [69].

2.8 Results

ParMA support for balancing 2D and 3D unstructured meshes with complex topological features are demonstrated in the following subsections. First, we compare ParMA against its predecessor LIIPBMod on up to 256Ki ($256{\cdot}2^{10}$) parts. We then test the effect of ParMA's entity selection features described in Section 2.7 on partition quality and imbalance. Then, we present ParMA's ability to improve partitions created with graph and geometric partitioning methods on up to 1Mi (2^{20}) parts. Lastly, we discuss the effect of partition improvement on the scalability of PHASTA computational fluid dynamics up to 512Ki parts.

2.8.1 LIIPBMod Comparison

We compare the performance of ParMA
vertex>element improvement against LIIPBMod on a 64Ki, 128Ki, and 256Ki partition of a 941 million element tetrahedral abdominal aortic aneurysm (AAA) mesh. This mesh was generated by successively refining the initial coarse mesh shown in Fig. 2.10. Three test partitions of the mesh were created by running local ParMETIS (one instance per process [9]) part k-way on a 16Ki base partition created with global ParMETIS part k-way. Our partition improvement test then executed ParMA and LIIPBMod on the three partitions using the Mira Blue Gene/Q system at the Ar-

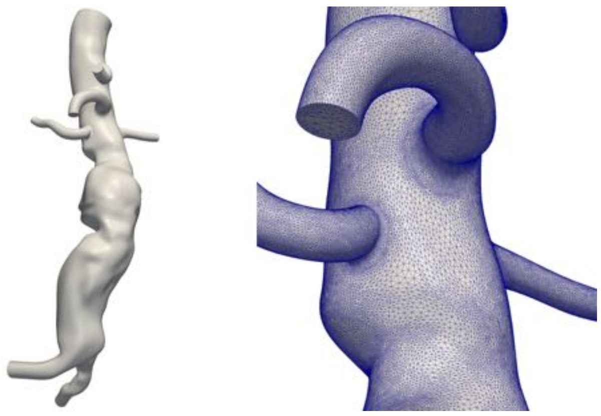


Fig. 2.10. Abdominal aortic aneurysm (AAA) geometric model and close-up view of a coarse mesh.

gonne Leadership Computing Facility (ALCF).

Fig. 2.11 depicts the change in vertex and element imbalance resulting from ParMA and LIIPBMod. In these tests LIIPBMod targets a 5% vertex imbalance and ParMA targets 5% vertex and element imbalance. Note that LIIPBMod does not explicitly target reducing the element imbalance; it simply tries not to harm it significantly while balancing vertices. LIIPBMod balancing stagnates at around 10% for the vertex imbalance and, at 256Ki, increases the element imbalance by two percentage points. At all three partition sizes ParMA meets the vertex and element imbalance target of 5% and executes 75% faster than LIIPBMod. For these partitions ParMA and LIIPBMod have an insignificant effect (less than one percent) on the total number of vertices. The ParMA features that support fast balancing are discussed in Sections 2.5, 2.6, and 2.7. Next, we discuss the performance cost and partition quality improvements of these features.

2.8.2 Feature Tests

We tested ParMA vertex>element improvement on a 497,058 triangular-element MPAS North America 15km-to-75km graded ocean mesh partitioned to 1Ki parts. The initial partition generated with local ParMETIS part k-way has a vertex and element imbalance of 36% and 17%, respectively, and on average, 280 vertices per part.

Configuration 1 of Table 2.4 serves as the baseline for feature inclusion. It

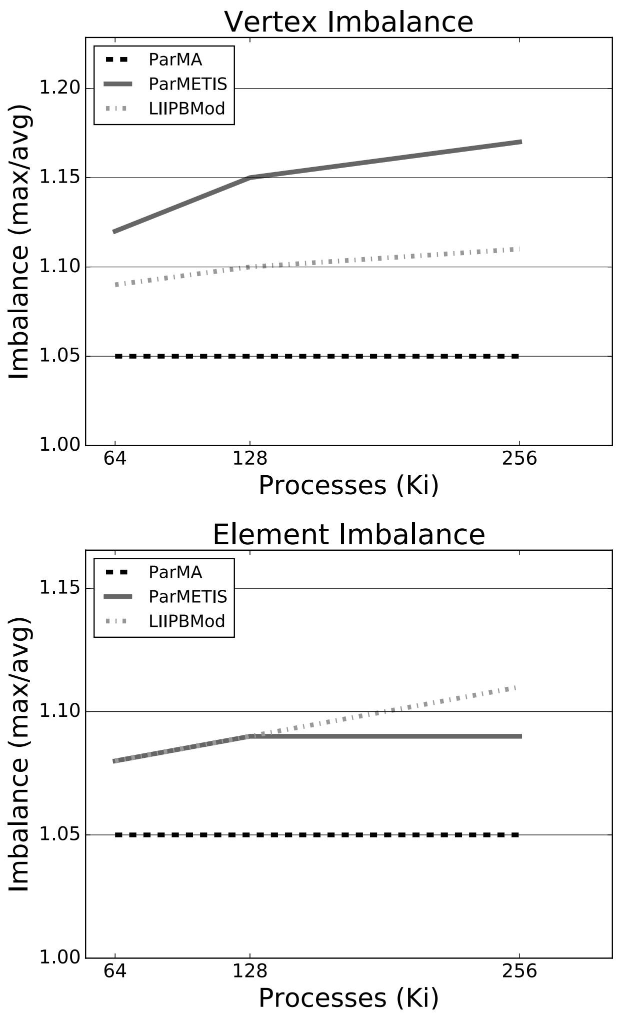


Fig. 2.11. Evolution of the (top) vertex and (bottom) element imbalance with ParMA, LIIPBMod, and ParMETIS in the 941 million element AAA mesh.

uses iterator-based part boundary vertex traversal (disabled graph distance), disables detection of non-manifold part junctions, has a fixed cavity size for selection, and when balancing elements, does not cancel selections to help preserve vertex imbalance. Configurations 2, 3, 4, and 5 successively add the features listed in Table 2.4.

For each configuration Fig. 2.12a and Fig. 2.12b depict the change in partition quality, relative to the initial partition, after ParMA balancing. ParMA's target imbalance was set to 5% for vertices and elements. Partition quality is measured in three ways: (1) the average number of neighbors per part, counted via

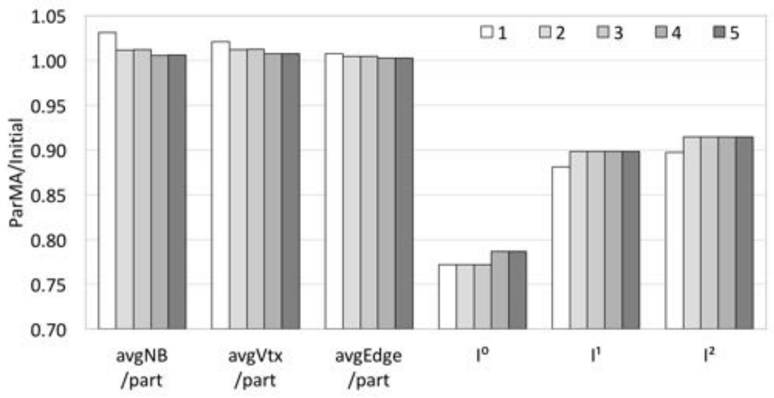
Table 2.4. ParMA test configurations.

Configuration	Enabled Features
1	None
2	1 + core distance traversal
3	2 + non-manifold feature detection
4	3 + increasing cavity size selection
5	4 + selection cancellation

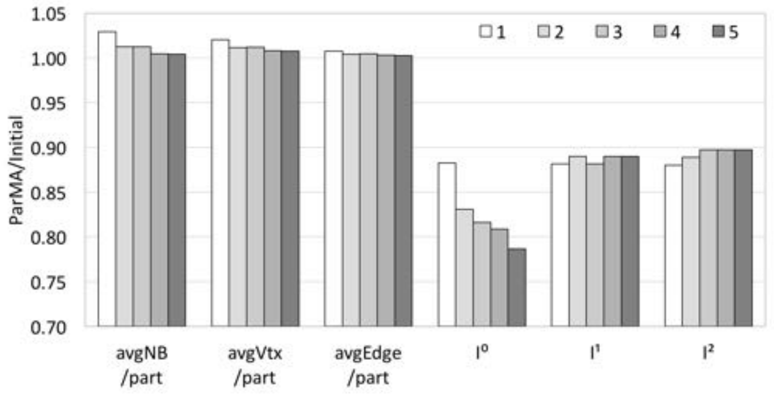
shared vertices, ‘avgNB/part’, (2) the average number of vertices and edges per part, ‘avgVtx/part’ and ‘avgEdge/part’, and (3) the entity imbalance, I^d. For each of these measures a value of one indicates no change from the initial partition, while a value greater (lower) than one indicates an increase (decrease) in the measure relative to the initial partition.

Fig. 2.12a depicts the improvement in quality after vertex balancing. The average number of neighbors, vertices, and edges per part increases by one percent or less with all features enabled. Relative to the over 20% decrease in vertex imbalance, these increases are negligible.

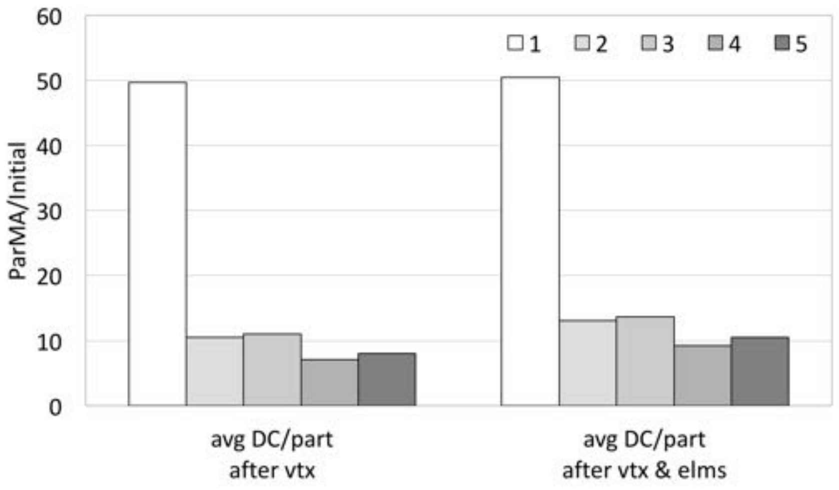
Vertex>element balancing, Fig. 2.12b, further improves the partition quality as features are enabled. After Configuration 1 (element balancing with no features enabled), the element imbalance is reduced from 5% to 3% at the cost of a vertex imbalance increase from 5% to 20%. Enabling core distance traversal, Configuration 2, reduces the average number of disconnected components per part. Fig. 2.12c shows that the disconnected component count, relative to the initial partition, increases by 50x in Configuration 1 while Configuration 2 only has a 10x increase. The large reduction in disconnected components reduces the number of vertices on the part boundaries. This reduction in turn helps limit the vertex balance increase to 13% after element balancing. The features of Configuration 4 further reduce the number of boundary vertices (as indicated by the reductions in average neighbors, edges, and vertices per part) and results in a 10% vertex imbalance. With all features enabled, Configuration 5, a final vertex imbalance of 7% is reached while maintaining the 5% element imbalance and further improving the other quality measures. This Configuration runs in 72% of the time of Configuration 1; 3.07 seconds versus 4.25 seconds. The faster run time is mainly the result of vertex balancing times reducing from



(a) Partition quality after vertex balancing.



(b) Partition quality after vertex $>$ element balancing.



(c) Disconnected components.

Fig. 2.12. Partition quality of a 1,024 part MPAS North America 15km to 75km graded ocean mesh using the ParMA configurations listed in Table 2.4.

4.09 seconds to 2.82 seconds, and only a slight increase in the element balancing times from 0.16 seconds to 0.25 seconds.

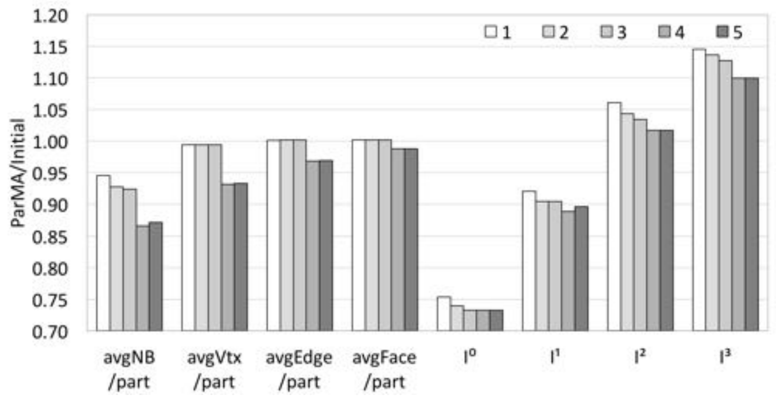
We also ran the feature test on the 3D 2.3 million element RPI Formula Hybrid suspension upright mesh, the geometric model depicted in Fig. 2.2. The test mesh has 2,048 parts and a 46% vertex and 10% element initial imbalance. ParMA's target imbalance was set to 5% for both vertex and vertex>element balancing. Fig. 2.13 shows the results of the tests. In Configuration 1 balancing the mesh vertices to 10% increases the element imbalance to 26%. The subsequent element balancing reduces the element imbalance to 5% in 17.8 seconds, but increases the vertex imbalance to 29%. As features are enabled the partition quality and imbalances improve at the cost of increased run time. Running with all features enabled (Configuration 5) requires 37.6 seconds (two times longer than Configuration 1), and reaches an element imbalance of 5% and a vertex imbalance of 9%.

A critical difference of the 3D upright tests to the 2D MPAS tests is the large reduction in disconnected parts and the related decrease in the average neighbors and entities per part. Compared to the initial partition, Configuration 5 of the upright test reduces the average neighbors, vertices, and disconnected components per part by 17%, 8%, and 93% respectively, and 3%, 2%, and 27% versus Configuration 1. This difference is mostly due to the change from 2D to 3D and the increased connectedness of the geometric model that enables more migration opportunities; the MPAS mesh has multiple geometric surfaces which only share one or two vertices with other surfaces.

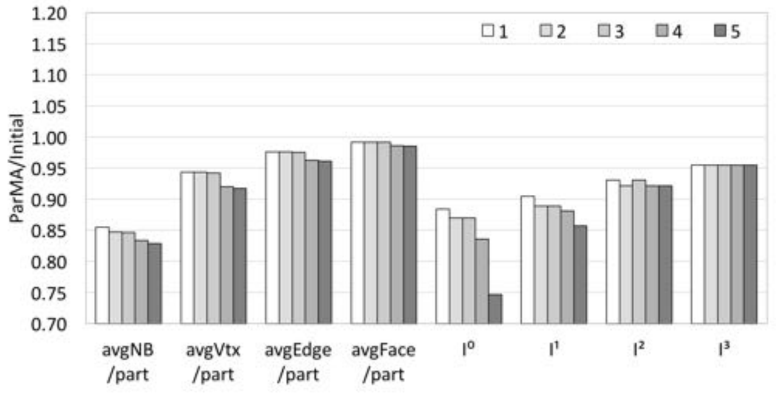
The feature tests were run using one part per core on the Blue Gene/Q at the Rensselaer Center for Computational Innovations. Tests with all features enabled and additional balancing criteria are described next.

2.8.3 Multi-Criteria Improvement

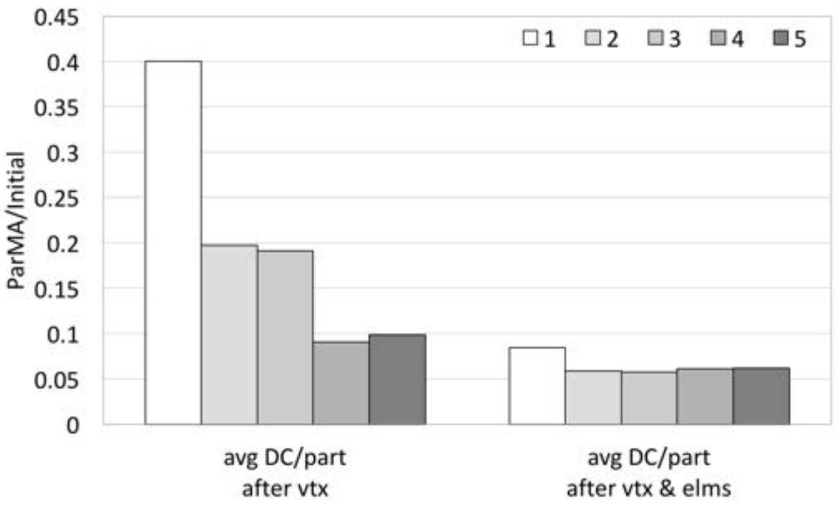
Analysis codes which have work associated with multiple entity dimensions and have a non-uniform distribution of that work require multi-criteria balancing. Codes with this requirement include finite elements with non-uniform p, particle-in-cell [70], contact/impact [71], atomistic-to-continuum [72], and other multi-model



(a) Partition quality after vertex balancing.



(b) Partition quality after vertex > element balancing.



(c) Disconnected components.

Fig. 2.13. Partition quality of a 2,048 part RPI Formula Hybrid suspension upright mesh using the ParMA configurations listed in Table 2.4. Lower is better.

or multi-physics techniques [73]. ParMA satisfies this requirement by balancing the entity dimensions defined in a priority-sorted list. For each entity in the mesh the application also optionally provides weights specifying the associated computational load. To test this ability we ran ParMA vertex=edge>element balancing on a 2.3 million element, 2,048 part mesh of the suspension upright. The test emulates a non-uniform work distribution associated with edges by setting entity weights. On part zero edge weight is set to two; all other parts have entity weights of one.

Two initial partitions were used in testing; one is the result of mesh adaptation (listed as ‘adapt’), and another is generated with RIB. The partitions’ average entity counts and imbalances are listed in Table 2.5. The adapt partition, relative to the RIB partition, has a ten point higher element imbalance, and on average, four more neighbors and two more disconnected components per part. Given the lower initial quality, ParMA improvement on the adapt partition requires about 350% more time to run (27.4 seconds versus 7.7 seconds), and has final entity imbalances (noted in the ‘elements’ row) a few points higher than the final ParMA imbalances of the RIB partition. Note that, even with the run time increase, the time spent in ParMA is insignificant relative to the time spent executing a typical finite element analysis on a partition of this size.

Despite an initial weighted-edge imbalance of over 90% in both partitions, ParMA reduces the entity imbalances to less than 9% while also reducing the average per part entity weights by up to 5%. Critical to this result is ParMA’s ability to diffuse away edge weight from the heavily imbalanced part zero while not overloading other parts. Diffusion reduces the number of mesh edges in part zero from 1674 to 901 in the adapt partition, and from 1665 to 889 in the RIB partition.

2.8.4 Partitioning to Over One Million Parts

ParMA quickly reduces large imbalances and improves part shape of a 1.6 billion element suspension upright mesh partitioned from 128Ki to 1Mi (2^{20}) parts (approximately 1500 elements/part). The initial 128Ki partition has less than 7% imbalance for all entity dimensions. We ran the tests on the Mira Blue Gene/Q located at the ALCF. One hardware thread was used per part.

Table 2.5. vertex=edge>element partition improvement on a 2.3 million element, 2048 part, mesh of the RPI Formula Hybrid suspension upright of Fig. 2.2.

stage	avg/part			I^0	I^1	I^2	I^3	time (s)
	vtx	edge	face					
adapt	**357.749**	**1741.012**	**2497.981**	**1.46**	**1.92**	**1.15**	**1.10**	
vertices	334.0	1687.7	2469.3	1.08	1.92	1.16	1.19	10.27
edges	330.5	1679.0	2464.3	1.09	1.06	1.09	1.13	6.88
elements	**328.829**	**1674.661**	**2461.637**	**1.09**	**1.06**	**1.07**	**1.08**	10.26
RIB	**350.457**	**1737.694**	**2503.369**	**1.53**	**1.92**	**1.10**	**1.00**	
vertices	337.8	1705.0	2483.4	1.04	1.95	1.07	1.10	3.01
edges	333.2	1692.8	2475.8	1.06	1.05	1.04	1.07	3.86
elements	**331.387**	**1687.526**	**2472.306**	**1.07**	**1.05**	**1.04**	**1.04**	0.85

- Partitioning with global RIB completes in 103 seconds and results in a 209% vertex imbalance and a perfect element imbalance. ParMA runs on 1Mi processors in 20 seconds and reduces the vertex imbalance to 6%, only increases the element imbalance to 4%, and reduces the average number of vertices per part by 5.5%.

- Local partitioning with ParMetis (one serial instance of ParMETIS for each initial part) completes in 9.0 seconds and results in a 63% vertex imbalance and a 12% element imbalance. ParMA runs in parallel on 1Mi processors in 9.4 seconds and reduces the vertex imbalance to 5%, the element imbalance to 4%, and reduces the average number of vertices per part by 2%.

Partitioning a 12.9 billion element mesh from 128Ki ($<$ 7% imbalance) to 1Mi parts (approximately 12 thousand elements/part) using serial instances of ParMETIS completes in 60 seconds and results in a 35% vertex imbalance and an 11% element imbalance. Running ParMA in parallel on 1Mi processors takes 36 seconds to reduce the vertex and element imbalances to 5% and reduce the average number of vertices per part by 0.6%.

Table 2.6 lists the number of elements, the initial and target part counts, and the initial entity imbalances, I^{0-3} for vertices, edges, faces and regions, respectively, for three partitions. Table 2.7 lists the results of ParMA runs on those partitions. Note, the column ‘dec. (%)’ lists the percentage decrease in the average vertices per part after ParMA relative to the partitioning stage, ‘Split’.

Table 2.6. Initial meshes for upright tests. The name of each is mesh is describing the number of elements in the target part.

name	elements	parts	target parts	elms per tgt. part	I^0	I^1	I^2	I^3
small	1.6×10^9	2^{17}	2^{20}	1541.7	1.06	1.06	1.06	1.07
medium	12.9×10^9	2^{17}	2^{20}	12 333.8	1.05	1.06	1.07	1.07
large	12.9×10^9	2^{17}	2^{19}	24 667.6	1.05	1.06	1.07	1.07

Table 2.7. X+ParMA vertex > element upright test results.

scope	density	method	stage	avg vtx	dec. (%)	I^0	I^1	I^2	I^3	tot (s)
local	small	rib	Split	455.1		1.34	1.18	1.13	1.13	10.67
			ParMA	427.6	6.42	1.07	1.06	1.05	1.05	8.94
		pmetis	Split	427.0		1.63	1.32	1.13	1.12	8.99
			ParMA	418.8	1.97	1.05	1.05	1.04	1.04	9.48
	medium	rib	Split	2825.5		1.31	1.14	1.08	1.07	54.32
			ParMA	2752.0	2.67	1.06	1.05	1.04	1.05	48.81
		pmetis	Split	2687.7		1.35	1.14	1.11	1.11	59.81
			ParMA	2671.3	0.61	1.05	1.05	1.04	1.05	36.15
	large	rib	Split	5273.9		1.16	1.13	1.12	1.13	42.69
			ParMA	5122.9	2.95	1.05	1.04	1.04	1.05	52.87
		pmetis	Split	5132.4		1.21	1.09	1.10	1.10	37.02
			ParMA	5102.2	0.59	1.04	1.04	1.04	1.04	41.55
global	small	rib	Split	470.1		**3.09**	2.07	1.45	1.00	103.14
			ParMA	445.4	5.54	**1.06**	1.04	1.03	1.04	20.23
	large	rib	Split	5367.3		2.49	1.70	1.29	1.00	96.79
			ParMA	5228.8	2.65	1.05	1.02	1.03	1.04	379.84

2.8.5 CFD Scaling Improvement

As an example of ParMA's ability to improve simulations of very complex geometric models at extreme scale, consider the geometry shown in Fig. 2.14. The left side of the figure depicts the surface of the vertical tail and rudder while the right side provides a detailed view of a complex geometric junction. At this junction we show a close-up view of a clip-plane cutting through the very small gap between the vertical stabilizer and the rudder where many parts are contained. In this region several of the parts are "cutoff" from the surrounding geometry and have a limited number of neighbors to diffuse through for partition improvement.

The partitions of the 1.2 billion element tetrahedral mesh for this study were obtained through a series of steps. First, mesh adaptation was executed on a 4Ki part mesh using an error-based size field [18], [19]. To balance and partition this mesh, global ParMETIS part k-way [25] was executed to create an 8Ki part mesh.

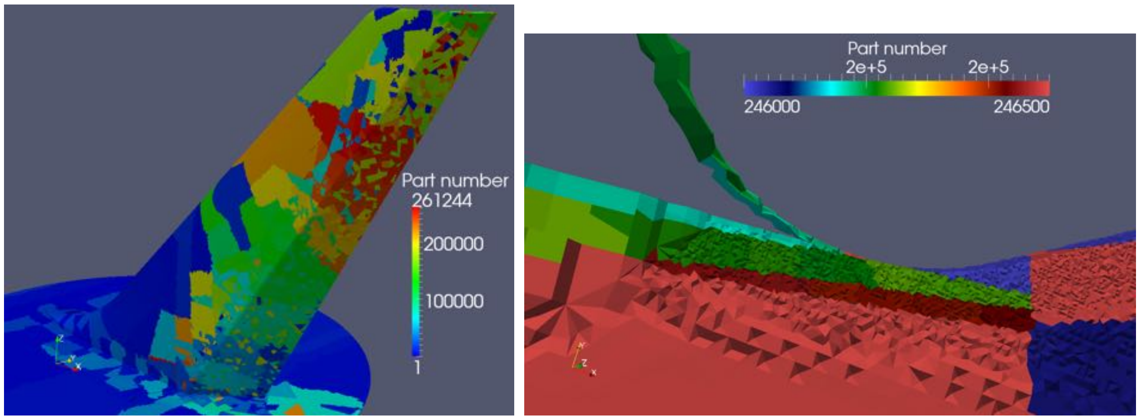


Fig. 2.14. (left) Full view of the vertical stabilizer and rudder and (right) a slice at their junction colored by part number illustrating the complex geometry and small features of the fluid mesh.

Starting from this 8Ki part mesh, with a 7% vertex imbalance and 1% element imbalance, ParMETIS part k-way was applied locally to each part to create partitions of the mesh in powers of two from 64Ki parts to 512Ki parts. These partitions were then balanced using ParMA vertex>element to create a second set of partitions.

The flow in this case is solved by PHASTA. PHASTA is a stabilized finite element analysis code [74] using an implicit solver. The code is written in FORTRAN and is parallelized with MPI. PHASTA's computational work is dominated by equation formation and equation solution. Both types of work are executed on the same partition of mesh elements [75]. An ideal partition will have balanced elements for equation formation work, and balanced vertices, the degree-of-freedom holder, for equation solution work. Furthermore, the partition will have parts with a low surface-to-volume ratio to limit the cost of neighborhood communications that exchange information on boundary vertices [76].

As shown in Fig. 2.15, through three part-count doublings, ParMA is able to improve the vertex imbalance with only insignificant increases in element imbalance. For example, in the largest partition, 512Ki parts, ParMA reduces the vertex imbalance from 54% to 6%, and only increases the element imbalance from 1.8% to 3%. As expected, the 1.2 percentage point increase in element imbalance has no effect on the nearly perfect scaling of equation formation (scaling factor, defined as $(time(base) \cdot procs(base))/(time(test) \cdot procs(test))$, of 0.96 maintained). Criti-

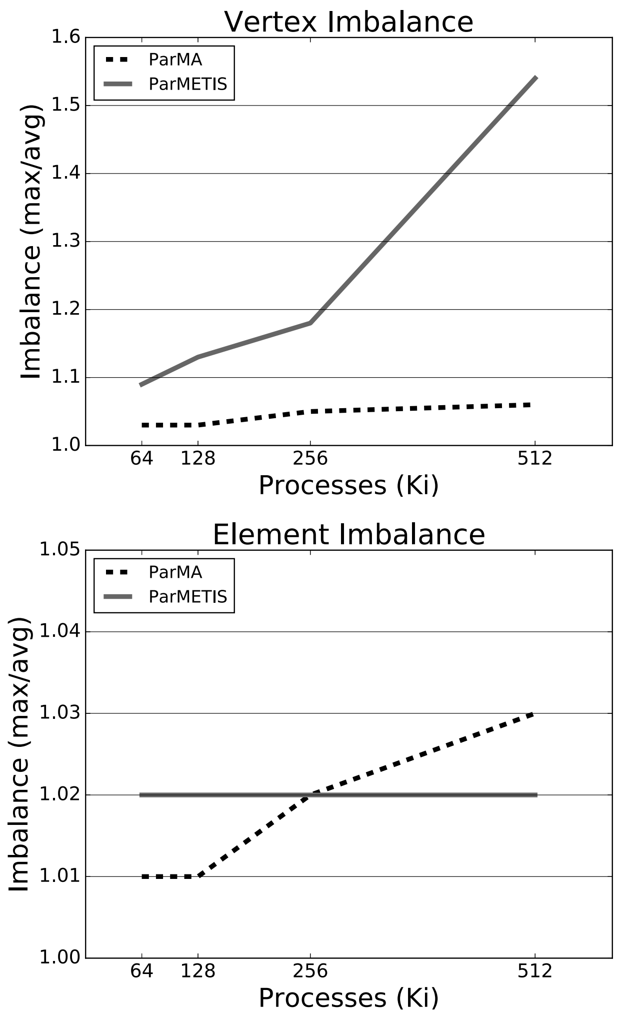


Fig. 2.15. Evolution of the (top) vertex and (bottom) element imbalance with and without ParMA.

cally though, ParMA improves the linear algebra work performance by 28% over the ParMETIS partition, and improves scaling from 0.82 to 1.14, as shown in Fig. 2.16. As sparse linear algebra is memory bandwidth limited [42], a super-linear scaling is observed as the working data size is reduced and cache utilization is increased. Similar, but less dramatic, performance and scaling gains are observed in the 256Ki part case. In the smaller 64Ki and 128Ki partitions the performance difference is negligible. All PHASTA runs were performed on Mira using one process per core. This configuration, although not optimal for achieving peak floating point performance on the Blue Gene/Q, was selected to avoid unfortunate process to core mappings

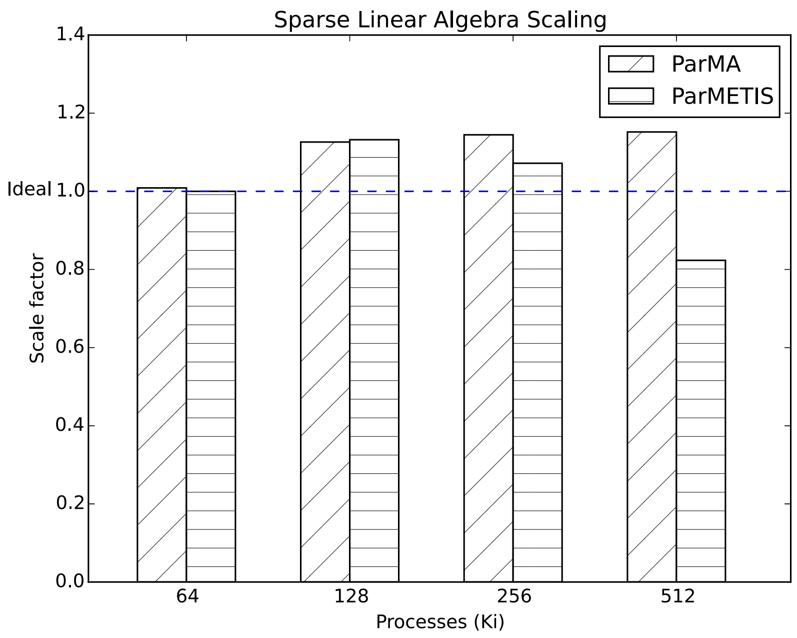


Fig. 2.16. Improvement of PHASTA sparse linear algebra scaling with ParMA. The PHASTA performance on the 64Ki ParMETIS partition is used as a baseline for all runs. Higher is better.

that could assign two heavily loaded processes to the same core, and thus confound the interpretation of performance results.

2.9 Summary

ParMA enables scalable data transformations within components by providing fast, multi-criteria, dynamic load balancing procedures that execute efficiently on over one million cores of leadership-class parallel systems. These procedures rely on part-level (graph distance) and entity-level (cavity size, surface area) heuristics to define the traversal order over the part boundary, which elements to select for migration, and the destination part that should receive the elements. In addition to the selection heuristics are mechanisms to reject migration requests if they harm the imbalance of higher priority mesh entities (cancellation) and to gracefully stop ParMA if beneficial changes can no longer be made (stagnation avoidance). The net result of these efforts is a load balancing method that outperforms the previous versions of the approach (LIIPBMod) in quality and run time, and, versus a leading graph-based method, improves performance of a massively parallel CFD code at 512Ki processors by 28%.

CHAPTER 3
CONSTRUCTING PARALLEL ADAPTIVE WORKFLOWS

3.1 Introduction

The ability to apply algorithmic or mathematical advances to a particular simulation depends on the coupling of those procedures with existing simulation-based engineering tools. Solving the most advanced examples of these simulations for real-world problems of interest requires more memory than is available on a single workstation or server. Thus, parallel workflows that operate effectively on distributed memory parallel systems are needed.

3.2 The Current State-of-the-Art

Advances in hardware and algorithms have provided many orders of magnitude improvement in the ability to perform large-scale simulations. Combined, these advances enable parallel simulations to operate efficiently on the largest petascale computers (e.g., unstructured mesh CFD software that scales to over 768,000 cores [76]). As plans to move to exascale computing are carried out [78] though, it is clear that the inability to effectively increase CPU clock rates requires all truly large-scale computations be performed on massively parallel computers. These future massively parallel computers will be more heterogeneous and therefore more complex to program. On the positive side, progress on the development of next generation massively parallel computers is leading to systems that are much more cost effective to purchase, power, and maintain. This means an increased ability to cost effectively employ the most computationally intense simulations in engineering

Portions of this chapter previously appeared in: M. S. Shephard, C. Smith, and J. E. Kolb, "Bringing hpc to engineering innovation," *Comput. in Sci. & Eng.*, vol. 15, no. 1, pp. 16–25, Feb. 2013

Portions of this chapter previously appeared in: C. W. Smith, S. Tran, O. Sahni, F. Behafarid, M. S. Shephard, and R. Singh, "Enabling HPC simulation workflows for complex industrial flow problems," in *Proc. XSEDE Conf.: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, 2015, pp. 41:1–41:7

design processes, assuming the required software tools and methods of applying the software are available.

The national laboratories, particularly the U.S. Department of Energy (DOE) with programs like SciDAC's FASTMath and the Exascale Computing Project, are actively developing new generations of software that can effectively operate on massively parallel computers. These developments include simulation tools for DOE applications and software that aid in the development of large-scale simulation tools. Three examples of different classes of tools that help support the development of parallel simulations are: Trilinos [79] is an infrastructure of over 50 composable packages that can be used to construct large-scale, multi-physics simulations. The Portable, Extensible Toolkit for Scientific Computation (PETSc) [80] is known primarily for its set of linear, and non-linear, algebraic system equation solvers that have been integrated into numerous simulation codes. Zoltan [13] is a parallel load balancing service that interacts with application data to determine how to distribute it for the most effective parallel execution. In addition, there are many parallel analysis procedures developed that execute specific simulations produced by DOE and Department of Defence laboratories. These primarily open source software packages are beginning to receive increased attention by industry and, to some extent, independent software vendors (ISVs). Although the open source nature of such software is attractive, the majority of these software packages are developed and supported by small teams that are most often focused on the advancement of a specific science application. These packages typically include specialized features designed for use by domain experts, are complex to integrate into simulation workflows, and lack adequate support systems for broad use. Some packages though, have been made more generally usable and are supported by more substantial developer and user teams. However, the ability of those teams to continue to provide long-term support through government R&D budgets is a complex issue and not guaranteed.

The maturation of computer-aided design (CAD) and computer-aided engineering (CAE) technologies have made engineering simulations a cornerstone in the design and manufacture of products ranging from aerospace vehicles, to consumer goods, to medical devices. The key CAD/CAE tools being used in these processes include geometric design, analysis model generation, engineering analysis and visualization. Over the past several years an increasing number of these engineering analysis packages execute in parallel. In some cases, parallelism has focused on taking advantage of higher core count shared memory workstations and do not address distributed memory methods as needed for inter-node parallelism on massively parallel computers. Those that have addressed the distributed parallelism needs typically had to develop new versions of the code to gain any reasonable level of scalability. However, these new codes, at least initially, have a limited set of functionalities as compared to the existing more fully featured codes that have been under development for many years.

Some CAE oriented ISVs have begun to develop new generations of software that employ data structures and algorithms that allow them to operate and, for the computationally intensive portions, scale on massively parallel computers. In addition, some of this software is designed with sufficient modularity to support interactions through easy to use interfaces such that users can combine procedures from multiple sources to meet their simulation needs.

3.3 Approach to Eliminating Technical Impediments

The execution of parallel simulations typically requires a workflow that couples multiple simulation procedures. For example, a single physics simulation requires linking geometric modeling systems with mesh generators and a mesh-based solver. For multiple reasons, ranging from a single package's best practices and validation processes, to the interactions of multiple organizations or companies doing different steps in the process, these capabilities are typically not provided within a single software system and thus the effective coupling of multiple software packages is critical. The need to couple packages becomes more acute in multi-physics and multi-scale problems where procedures based on different models using different physical parameters must be coupled across scales [72], [81]. The ability to support these workflows is made even more complex when specific computationally intensive simulation steps must be executed on a massively parallel computer. One approach to couple software packages for the effective construction of simulation workflows is through the use of functional interfaces.

The most effective way to design simulation workflows are based on geometric models of the computational domain. Accordingly, CAE analysis tools have begun to move from mesh-based problem definitions to more geometry-based definitions. These simulations are typically executed using a graphical user interface that can accept CAD information. Many of these interfaces are oriented toward a specific vendor's set of CAE analysis tools while some others are oriented to interact through APIs supporting general interactions with geometry [82] and simulation attribute information. In those projects where the workflow developed employed software from several sources, we found using a generalized interface that can interact with geometry, attributes, and analysis procedures to be quite effective [11], [83], [84]. Given the problem definition, most analysis procedures require the domain be decomposed into a mesh of simple shapes. The greatest flexibility is provided when the parallel analysis procedures can accept an unstructured mesh; a decomposition of the domain into elements of various topological type (e.g., tetrahedrons, hexahedrons, prisms, and pyramids in 3D), order (i.e., straight sided or curved), quality, and volume. Mesh-based finite element [1] and finite volume [2] analysis methods are available for many classes of problems. The use of these methods allows both the application of fully automatic mesh generation and adaptive mesh control.

Workflow development for various academic and industrial applications has provided valuable lessons on the design of interoperable interfaces. One key lesson is that there are some common high-level interfaces that can be defined for coupling many of the simulation procedures. These interfaces are primarily focused on methods to load inputs into the data structures of analysis procedures, and subsequently extract simulation data from those structures. For procedures using a geometry-based problem definition and a solution field interface, this approach allows the fast integration of multiple meshing, analysis and visualization procedures, as well as supporting the ability to quickly replace any of their implementations. The typical initial implementation of these interface methods tend to pass information between major procedures using files. Although the most straightforward implementation, file I/O (serial or parallel) is a major bottleneck in the execution of large-scale parallel simulations. When the method used to execute the coupling of procedures is through APIs, it is conceptually straightforward to bypass the use of files and transfer information directly between data structures. However, the effective implementation, given existing software packages, is reasonably complex.

3.4 Component Interactions

The design and implementation of procedures within existing software packages directly affects how they interact with other workflow procedures. Procedures provided by a given software package are often grouped by functionality into a component; a reusable unit of composition with a set of interfaces to query and modify encapsulated state information [85]. The most interoperable, reusable, and extensible components are those with APIs, minimal dependencies, minimal exposure of symbols (e.g., through use of the unnamed namespace in C++ or the `static` prefix in C), scoped interfaces (e.g., via C++ `namespaces` or function name prefixes), and no resource leaks [86]–[88]. Conversely, many legacy components (e.g., analysis codes) may simply have file or command line interfaces (i.e., they do not provide libraries with APIs) and have little control of the symbols and memory they use. The xSDK project formalizes these levels of interoperability and, from that, defines basic requirements of packages for inclusion in its ecosystem [89]. In Sections 4.2 through 4.4 we discuss the design of three different analysis components and the impact of each design on coupling with an unstructured mesh adaptation component.

In-memory component interactions are supported by bulk or atomic information transfers. A bulk transfer provides a large set of information following some provided format. For unstructured meshes this transfer could be an array of node-to-element connectivities passed from a mesh adaptation or generation procedure to an analysis code. Conversely, an atomic transfer provides a single, or highly localized, piece of information. Continuing the connectivity example, an atomic transfer would be the nodes bounding a single element. In Section 4.4 we provide another atomic example that computes Jacobians of mesh elements classified on curved geometric model entities.

Our approach for high performing and scalable component interactions avoids filesystem I/O by implementing bulk and atomic transfers with component APIs or data streams. Thus, component interactions in this work are within a single executable typically built from multiple libraries. Alternatively, the ADIOS tools provide a mechanism for the in-memory coupling of multiple executables [90], [91]. Likewise, Rasquin et al. [92] demonstrated in-situ visualization with PHASTA and ParaView using GLEAN [93].

The type of interaction chosen to couple a pair of components depends on their implementations. Components with APIs that encapsulate creation, deletion, and access to underlying data structures support in-memory interactions at different levels of granularity. At the finest level a developer may implement all atomic mesh entity query functions such that components can share the same mesh structure; trading increased development costs for lower memory usage. An excellent example of mesh data sharing is the use of octree structures in the development of parallel adaptive analyses [94]. At a coarser level, a developer may simply create another mesh representation (a bulk transfer) through use of interfaces encapsulating mesh construction; trading higher memory usage for lower development costs. Although this method will allow for in-memory integration, it can suffer from the same disadvantages as the former approach in that a significant amount of time and effort will be required for code modification and verification. A generalization of this coarser level approach defines common sets of interfaces through which all components interact. For example, in the rotorcraft aerodynamics community the HELIOS platform provides a set of analysis, meshing, adaptation, and load balancing components via the Python-based Software Integration Framework [95].

Components that support a common file format and use one file per process (e.g., POSIX C `stdio.h` [96] or C++ `iostream`) can use our data stream approach with minimal software changes. Here, the bulk transfer is taken to nearly its highest level; the exchange of process-level data sets. This approach is also a logical choice for legacy analysis codes that do not provide APIs to access or create their input and output data structures.

Using a serialization framework like Google's FlatBuffers [97], or Cap'n Proto [98], also supports bulk data exchanges through use of their APIs and data layout specification mechanism. Furthermore, some of these frameworks provide a 'zero copy' mode that avoids encode and decode overheads; the serialized information can be directly accessed after transfer. Like the HELIOS approach, this approach is an interesting option for components that will be integrated with many others.

Details for implementing the bulk and atomic transfers are given in the following sub-sections.

3.4.1 Component Interfaces

The components in adaptive simulations that provide geometric model, mesh, and field information [11], [82], [99], and the relations between them, are essential to error estimation, adaptation [100]–[102], and load balancing [103] services. For example, transferring field tensors during mesh adaptation requires the field-to-mesh relation [104]. Likewise, the mesh-to-model relation (classification [11]) and geometric model shape information enable mesh modifications (e.g., vertex re-positioning) that are consistent with the actual geometric domain [82]. Similarly, classification supports the transformation of the input field tensors onto the mesh to define the boundary conditions, material parameters and initial conditions [15]. Because of this strong dependency, we provide these components and services together as the open-source Parallel Unstructured Mesh Infrastructure (PUMI) [11], [105]. PUMI's unstructured mesh components include:

- PCU - neighborhood-based non-blocking collective communication routines
- GMI - geometric modeling queries supporting discrete models and, optionally, Parasolid, ACIS, and Simmetrix GeomSim models using the Simmetrix SimModSuite library
- MDS - array-based modifiable mesh data structure [106]
- APF_MDS - partitioned mesh representation using MDS
- Field - describes the variation of tensor quantities over the domain
- ParMA - multi-criteria dynamic load balancing [103]

- MeshAdapt - parallel, size field driven local refinement and coarsening.

A good example of PUMI advanced component interface usage is the Superconvergent Patch Recovery (SPR) error estimator. The SPR routines estimate solution error by constructing an improved finite element solution using a patch-level Zienkiewicz-Zhu [107] least squares data fit. SPR provides two methods which use the patch-recovery routines. The first method recovers discontinuous solution gradients over a patch of elements and approximates an improved solution by fitting a continuous solution over the elemental patch. The second method provides an improved solution in much the same way as the first, but operates directly on integration point information obtained by the finite element analysis. For both methods, the improved and primal solutions are then used to create a mesh size field that is passed to MeshAdapt to guide mesh modification operations [4], [108].

3.4.2 Data Streams

Components can pass information and avoid expensive filesystem operations through the use of buffer-based data streams. This approach is best suited for components already using POSIX C `stdio.h` [96] or C++ `iostream` String Stream APIs [109] as few code changes are required. The key changes entail passing buffer pointers during the opening and closing of the stream, and adding control logic to enable stream use.

In a component using the POSIX APIs, a data stream buffer is opened with either the `fmemopen` or `open_memstream` functions from `stdio.h`. `open_memstream` only supports write operations and automatically grows as needed. `fmemopen` supports reading and writing, but uses a fixed size, user specified, buffer. Once the buffer is created, file read and write operations are performed through POSIX APIs accepting the `FILE` pointer returned by the buffer opening functions; i.e., `fread`, `fwrite`, `fscanf`, `fprintf`, etc. After all read or write operations are complete, a call to fclose will automatically dellocate the buffer created with `fopen`. A buffer created with `open_memstream` requires the user to deallocate it.

Example uses of the POSIX C and C++ `iostream` APIs are located in the Appendix.

3.5 Components for 3D Unstructured Mesh-based Workflows

Automated, parallel, adaptive, simulation workflows require the interactions of multiple software components. Fig. 3.1 shows the set of components needed for reliable unstructured mesh simulations with a focus on the role of the parallel mesh infrastructure. To effectively support the integration with multiple parallel analysis components, as well as alternative meshing and visualization technologies, the parallel mesh structures and services interact through APIs [82], [99]. These APIs are designed specifically for an in-memory passing of information that is needed between different simulation components in going from the problem specification to the simulation results. At the highest level, simulation information is specified in terms of attributes on a description of the computational domain, typically a CAD model, and parameters defining the physics model. Boundary-based geometric model and mesh representations, building on the abstraction of topological entities and their adjacencies, are ideally suited for the specification of, and maintaining of, the relationship between domain descriptions [67], [110], [111]. Problem definition attributes are transformed into mathematical fields specified over geometric sub-domains, through the relationships between the geometric model and mesh, while physics model parameters are transformed for selection of partial differential equations and discretization methods. Combined, this information is used to determine the desired output fields. In the case of a mesh-based simulation, the domain information is discretized into a mesh that is adapted during the simulation. The mesh-based analysis discretizes the mathematical model and solves the resulting algebraic systems to determine vectors of unknowns that correspond to distribution coefficients for solution fields discretized over the mesh.

In the following sub-sections we review the functionality of the core unstructured mesh workflow components depicted in Fig. 3.1 and available implementations. A review of available partitioning tools is located in Section 2.2.

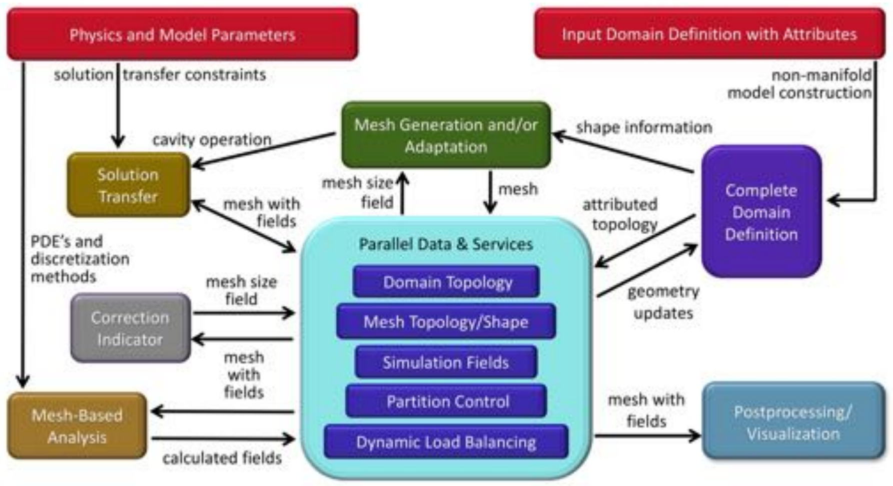


Fig. 3.1. Components for parallel, adaptive, mesh-based simulation workflows [11].

3.5.1 Geometric Model Definition and Interrogation

Non-manifold boundary representations provide an effective representation of the computational domain that can be coupled with mesh generation/adaptation and analysis attributes for the simulation workflow [82]. Attribute information specified on topological entities of the geometric model need to subsequently be related to the mesh discretizing the domain. This critical association of a mesh entity to exactly one geometric model entity of equal or higher order is known as classification. Fig. 3.2 depicts a two-dimensional geometric model with four vertices, G^0_{1-4}, four edges, G^1_{1-4}, and one face, G^2_1, and a mesh classified against it [11]. The mesh faces are classified on the single geometric model face; noted as $M^2_i \sqsubset G^2_1$. Mesh edges M^1_i adjacent to a single mesh face are classified on the appropriate geometric model edges G^1_{1-4}. All other mesh edges are classified on the geometric model face; a higher order classification. The mesh vertices at the corners of Fig. 3.2 are respectively classified against the geometric model vertices G^0_{1-4}. Likewise, mesh vertices along the model edges G^1_{1-4} are classified on those edges. The remaining unclassified interior mesh vertices are classified on the geometric model face.

Some mesh data structures and geometric modeling interfaces also support a reverse classification query. Given a geometric model entity, this query will return the set of mesh entities of equal order classified on it. If provided, this query

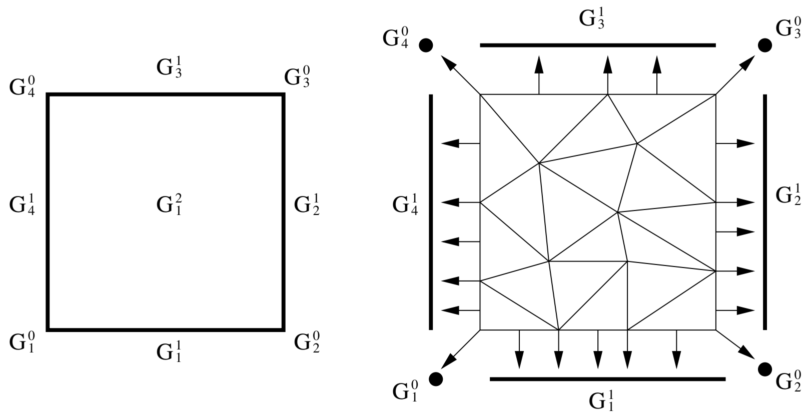


Fig. 3.2. (left) A simple geometric model and (right) a mesh classified on it [11]. Arrows show the classification of boundary mesh vertices and edges to geometric model entities. The interior mesh vertices, edges, and faces are classified on the geometric model face G_1^2.

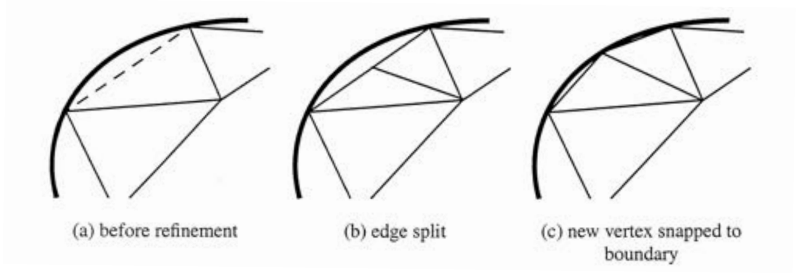


Fig. 3.3. Mesh adaptation's use of a positional geometric model query to place a new mesh vertex on a geometric model edge [113].

can improve the efficiency of transferring attribute information associated with geometric model entities to mesh entities; the most common example being boundary conditions.

In addition to topological information supporting classification, geometric modeling interfaces can also provide geometric shape information to answer positional queries and/or provide geometric shape parameters. These queries enable operations of mesh adaptation and analysis components to properly account for curved domains. In mesh adaptation these queries are critical for positioning existing or new vertices (or control points for higher order elements [112]) such that the mesh can accurately tessellate the volume of the computational domain. Fig. 3.3 depicts an example where a new mesh vertex is positioned on the geometric model edge via a parametric location query [113] or closest point projection.

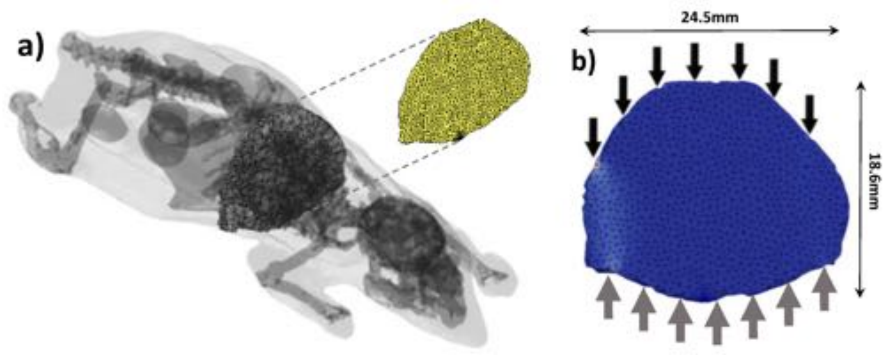


Fig. 3.4. The mesh of a 2D slice through the segmented Digimouse created using Simmetrix GeomSim. The location of light emitters and detectors are shown with black and gray arrows, respectively [119].

Outside of simulation workflows where engineers use CAD systems to design devices are workflows where the computational domain information is provided in a discrete form. The most common forms of discrete domain information are triangulated surfaces, voxel level forms, such as image data, and point clouds. The source of this data can range from volumetric medical imaging (CT, PET, etc.) of living tissues, to surface point clouds generated by satellite or LIDAR imaging of geographical features. Tools are available to convert this data to a topological representation of the computational domain which can provide the fundamental classification queries required by simulation workflows [114]. However, there is limited functionality to recreate geometry information for precisely satisfying positional queries on curved domain boundaries. To fill this gap, surface reconstruction techniques are available [115], [116]. Examples of voxel- and point cloud-based geometric model generation are depicted in Fig. 3.4 and Fig. 3.5. The resulting Digimouse [117] model supported adaptive mesh optimization for a mesh-based Monte Carlo simulation of light propagation through tissues with heterogeneous material properties [118], [119]. The latter example of the Landers fault system supported petascale high order dynamic rupture earthquake simulations that were an SC '14 Gordon Bell Finalist [120].

A robust, closed-source component that supports workflow interactions with both discrete and parametric geometric models is GeomSim [122] from Simmetrix. Its functionality is provided through functional interfaces to several CAD systems

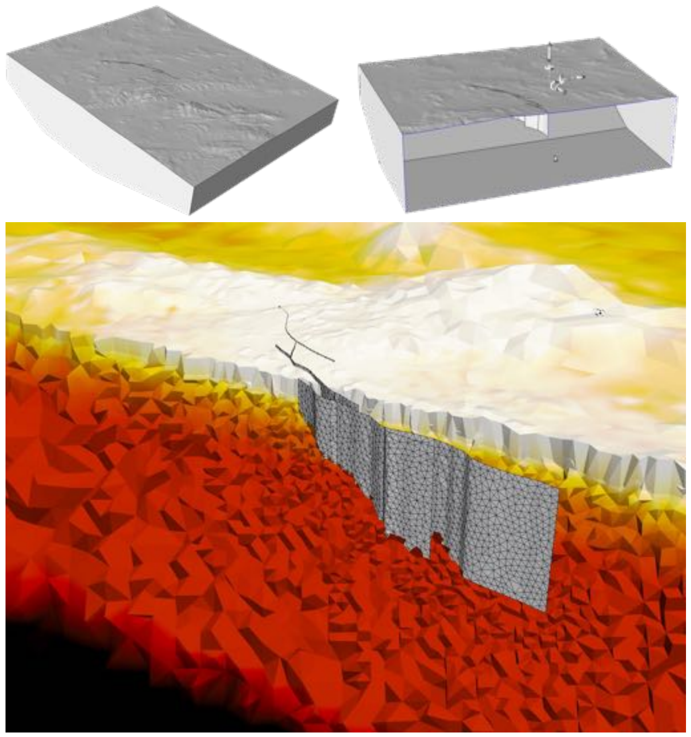


Fig. 3.5. (top) Geometric model and (bottom) mesh of the Landers fault system for SeisSol dynamic rupture earthquake simulations [120], [121].

(NX, Pro/E, SpaceClaim, and SolidWorks) and modeling kernels (Parasolid, ACIS, and Granite). Built on those interfaces, it has capabilities to correct poorly defined geometry, perform boolean operations on multiple CAD models (both discrete and parametric), automatically remove small features, when desired, and create complementary domains.

Applications which require an entirely open-source workflow can use components built upon the open-source (LGPL) Open CASCADE [123] modeling kernel. Unfortunately, its usability is hampered by a lack of documentation and a code base with over two million lines. To address the usability issues, the Common Geometry Module [83], [124], [125] (BSD-3 derivative) and EGADS [126] (LGPL) systems provide a simplified interface over Open CASCADE. Along with OpenCSM, a library for specifying sequences of geometric construction operations provided by EGADS, the Engineering Sketch Pad provides a web-enabled system for construction of parametric geometry used for automated design optimization [84].

3.5.2 Mesh Generation

To avoid the mesh generation bottleneck in simulation workflows a mesh generator must be (1) fully automated, (2) include methods for creating desired mesh types and gradations (including anisotropic meshes), and (3) be driven from a geometric model-based specification of the mesh control information. Given a valid, properly toleranced geometric model, an automated mesh generator is one that executes to completion without user interaction; including the case where no size controls are set [127]. By definition, there can be no requirement for the user to operate on individual mesh entities to 'repair' the mesh. Such a robust mesh generator must rely on geometric model kernels for accurate answers to topological and positional queries.

The Simmetrix MeshSim and distributed memory Parallel MeshSim components satisfy the mesh generation requirements. Parallel MeshSim can generate a one billion element mesh on a CAD model in about six minutes on 224 Intel Xeon processors. Meshes with up to 13 billion elements have been generated on 2048 processors. In this thesis the majority of the meshes generated were done so with the Simmetrix tools.

Another closed-source mesh generator that meets the requirements is from Pointwise [128], [129]. Unlike, MeshSim though, they only support shared memory parallelism. Thus, the size of the initial mesh they generate is limited by the amount of memory available on the system. There are also several serial open-source mesh generators that partially meet the requirements: gmsh (GPL) [130], NETGEN (LGPL) [131], and GRUMMP (BSD derivative) [132], [133].

3.5.3 Mesh Adaptation

As transient simulations evolve, the areas of the computational domain with large discretization error will also change. Procedures to automatically change the spatial discretization, the mesh, to control these errors are required. To operate effectively mesh adaptation procedures must run in parallel, apply local modification operations towards satisfaction of the applications requests, conform to the definition of the computational domain, and provide application 'hooks' to support the transfer of fields and other data associated with the mesh as it is modified.

Applications specify how they need the mesh to change through a three-dimensional metric tensor at each mesh vertex [3], [101], [134]. The tensor defines how edges incident to each vertex should be rotated and scaled to reduce the discretization error. For example, in the proximity of a shock the gradients normal to the front will be much higher than those along the front. As such, an application can define the tensor using derivatives of the quantity of interest to capture this anisotropy. In many applications though, it is sufficient to define isotropic scaling. For these cases the size field can be expressed as a scalar at each mesh vertex.

PUMI's parallel mesh adaptation component (MeshAdapt) meets the requirements by supporting refinement, coarsening, and nodal repositioning [11]. With each local mesh modification operation applied (split, collapse, swap, etc.) fields on the mesh are transferred. Field transfers are supported through interpolation and projection operations in the topological proximity of the modified mesh entities [5]. For fields with higher continuity or conservation requirements Omega_h [135], [136] (BSD-2-Clause) provides the necessary transfer mechanisms along with the majority of the PUMI MeshAdapt functionality [5], [136].

The Simmetrix SimModSuite Parallel MeshSimAdapt [122] satisfies the requirements while also providing more advanced support for meshes with stacks of semi-structured elements (e.g., often found in meshes used for solving flow problems with boundary layers). MeshSimAdapt procedures have executed on meshes as large as 92 billion elements on 786,432 cores [76].

Like MeshAdapt and MeshSimAdapt, the LGPL licensed MAdLib implements advanced mesh motion and local mesh modification operators, but only on tetrahedral meshes in serial [134]. Another serial open-source adaptation code is GRUMMP. It provides mesh quality improvement via edge/face swaps, and vertex re-positioning, but coarsening procedures are focused on supporting multi-grid methods.

3.5.4 Mesh Data Structures

Components that answer questions about mesh topology and permit association of field data are referred to as mesh data structures. A distributed memory parallel mesh data structure must also support the queries to determine the links between entities that are classified on part boundaries. Parallel data structures typically also support some level of ghosting; the copying of one or more layers of mesh entities across the part boundaries for read-only access [137].

Mesh data structures that support mesh adaptation must also support topological modifications and the migration of mesh entities between processes. Such modifiable mesh data structures are far more complex as efficient storage becomes a challenge. For example, a mesh entity of a given order (vertex, edge, face, region) has a fixed number of downward adjacencies depending on its topological type (triangle, quadrilateral, tetrahedron, prism, hexahedron, etc.). But, the upward adjacencies are only bounded by the quality the mesh; meshes with poor quality tend to have high-degree vertices (characterized by their one-to-many vertex-to-edge fan out).

Currently, the most memory efficient parallel mesh data structures that support adaptation are those using structures-of-arrays (SoA) [11]. While SoA implementations are not as intuitive as their object-based brethren, the complexity can be hidden under an object-based interface that significantly increases ease of use. A more recent benefit of SoA implementations is their data-parallel friendliness. On devices such as GPUs with thousands of small compute units, or CPUs with wide vector units, memory accesses are most efficient when they can be combined together into a fewer larger transfers (coalescing) [138], [139]. Since SoA implementations allocate large contiguous arrays where each entry represents a mesh entity or some scalar data associated with an entity, then an algorithm that traverses the mesh (dependency/collision issues aside) will implicitly have unit-stride coalesced accesses. That is in stark contrast to the irregular memory access pattern when traversing an object-based implementation where each object is individually allocated (to reduce the dynamic resizing complexities). An array-of-structs (AoS) implementation could avoid the irregular access pattern by pre-allocating the structs, but still suffer from non unit-stride access penalties [139]. As mesh data structure algorithms are memory bandwidth bound, the run time advantage of SoA implementations over object-based can be significant.

APF_MDS [11] (BSD-3-Clause) and Omega_h both provide parallel modifiable mesh data structures implemented using SoA. APF_MDS provides distributed memory parallelism via MPI and supports PUMI's MeshAdapt implementation. Omega_h supports hybrid super computers through MPI for inter-process communications and Kokkos [140] for intra-process shared memory communication and computation. MOAB (LGPL) is an alternative array-based distributed memory parallel mesh data structure that supports modification [141].

3.6 Summary

Parallel workflows for unstructured mesh-based, adaptive simulations are most effectively constructed from existing software components. Ideally, these components provide APIs that support bulk (sets of data over a significant portion of the domain on a given process) or atomic (sets of data over small sub-domains on a given process) queries and modification procedures to encapsulated state information. When the existing software component does not provide these APIs, or provides a very limited set of them, but supports POSIX file-based interactions, then the bulk transfer of data into and out of the component is well-supported by data streams. Components supporting these bulk and atomic transfers are easily coupled with other unstructured mesh-based components when their operations are based on a mesh classified against a geometric and partition model. PUMI uses this approach for its APIs that query and modify the mesh, its distribution across processes, and its relation to the geometric model. PUMI's unstructured mesh components include [11]:

- PCU - neighborhood-based non-blocking collective communication routines
- GMI - geometric modeling queries supporting discrete models and, optionally, Parasolid, ACIS, and Simmetrix GeomSim models using the Simmetrix SimModSuite library
- MDS - array-based modifiable mesh data structure
- APF_MDS - partitioned mesh representation using MDS

- Field - describes the variation of tensor quantities over the domain
- ParMA - multi-criteria dynamic load balancing
- MeshAdapt - parallel, size field driven local refinement and coarsening.

CHAPTER 4
IN-MEMORY INTEGRATION OF EXISTING SOFTWARE COMPONENTS FOR PARALLEL ADAPTIVE UNSTRUCTURED MESH WORKFLOWS

4.1 Introduction

Simulations on massively parallel systems are most effective when data movement is minimized. Data movement costs increase with the depth of the memory hierarchy; a design trade-off for increased capacity. For example, the lowest level on-node storage in the IBM Blue Gene/Q A2 processor [143] is the per core 16KiB L1 cache (excluding registers) and has a peak bandwidth of 819 GiB/s. The highest level on-node storage, 16GiB of DDR3 main memory, provides a million times more capacity but at a greatly reduced bandwidth of 43GiB/s, 1/19*th* of L1 cache [144]. One level further up the hierarchy is the parallel filesystem[1]. At this level, the bandwidth and capacity relationship are again less favorable and further compromised by the fact that the filesystem is a shared resource. Table 4.1 lists the per node peak main memory and filesystem bandwidth across five generations of Argonne National Laboratory leadership class systems: Blue Gene/L [147], [148], Intrepid Blue Gene/P [149], [150], Mira Blue Gene/Q [143], [151], Theta [152], [153], and 2018's Aurora [154]. Based on these peak values the bandwidth gap between main memory and the filesystem is at least three orders of magnitude. Software must leverage the cache and main memory bandwidth performance advantage during as many workflow operations as possible to maximize performance.

This chapter presents the use of in-memory component coupling techniques that avoid filesystem use for three different unstructured mesh-based, parallel, adap-

Portions of this chapter have been submitted to: C. W. Smith, B. Granzow, G. Diamond, D. A. Ibanez, O. Sahni, K. E. Jansen *et al.*, "In-memory integration of existing software components for parallel adaptive unstructured mesh workflows," submitted for publication

[1]For the sake of simplicity we assume that the main memory of other nodes is not available. But, there are checkpoint-restart methods that use local and remote memory for increased performance [145], [146].

Table 4.1. Per node main memory and filesystem peak bandwidth over five generations of Argonne National Laboratory systems. The values in parentheses indicate the increase relative to the previous generation system.

	Memory BW (GiB/s)	Filesystem BW (GiB/s)
BG/L	5.6	0.0039
BG/P	14 (2.4x)	0.0014 (0.36x)
BG/Q	43 (3.1x)	0.0049 (3.5x)
Theta	450 (11x)	0.058 (12x)
Aurora	600 (1.3x)	0.020 (0.34x)

tive workflows. These demonstrations highlight the need for in-memory coupling techniques that are compatible with the design and execution of the analysis software involved. Key to this compatibility is supporting two interaction modes: bulk and atomic information transfers.

4.2 PHASTA

PHASTA solves complex fluid flow problems [75], [155]–[158] on up to 768Ki cores with 3Mi ($3*2^{20}$) MPI processes [76] using a stabilized finite element method [74] primarily implemented with FORTRAN77 and FORTRAN90. Support for mesh adaptivity, dynamic load balancing, and reordering has previously been provided by the C++ PUMI-based component, chef, through file I/O. This file-based coupling uses a format and procedures that were originally developed over a decade ago. Our work adds support for PHASTA and chef in-memory bulk data stream transfers. We show performance of this approach with a multi-cycle test using a fixed size mesh and present an adaptive, two-phase dam-break analysis.

The data stream approach for in-memory interactions was the logical choice given the existing POSIX file support, and the lack of PHASTA interfaces to modify FORTRAN data structures. The chef and PHASTA data stream implementation maintains support for POSIX file-based I/O by replacing direct calls to POSIX file open, read and write routines with function pointers.

Our work also adds a few execution control APIs to run PHASTA within an adaptive workflow. The API implementation uses the singleton design pattern [159] and several of Miller's Smart Library techniques [86]. This approach provides backward compatibility for legacy execution modes, such as scripted file-based adaptive loops, with minimal code changes and easily accounts for the heavy reliance on global data common to legacy FORTRAN codes.

Fig. 4.1 depicts the evolution of the adaptive mesh for a dam-break test case ran on ALCF Theta using two-phase incompressible PHASTA-chef with data streams [156] . The dense fluid (water) is initially held against the left wall (not pictured) in a square column created by a fictitious constraint representing a dam. The remainder of the domain (1.25 column heights high and five column heights wide) is air. When the constraint is removed, as if the dam broke, the dense fluid falls and advances to the right [156]. A distance and curvature-based refinement band tracks the air-water interface. Outside of these bands the mesh is graded to a reference coarse size.

Algorithm 4 lists the steps in the two-phase adaptive analysis. Note, the terms 'read' and 'write' are used to describe transfers from and to both streams and files. On Lines 2-4 the PUMI partitioned mesh, geometric model, and problem definition information is loaded. Next, on Line 5 the I/O mode is set to either data streams or POSIX files by initializing the $file\_handle$ as described in Section 3.4.2. Next, the chef preprocessor is called on Line 6. The preprocessor first executes adjacency-based mesh entity reordering (l.16) [42] to improve the efficiency of the assembly and linear algebra solution procedures. Next, it creates the finite element mesh (i.e., nodes and element connectivity), solution field, and structures containing the point-to-point communication protocols and boundary conditions (l.17-18). Preprocessing concludes with the writing of this data to files/streams (l.19).

Line 8 of Algorithm 4 begins the solve-adapt cycle that runs until the requested number of solver time steps is reached. The PHASTA solver first reads its input information from chef via files or streams (l.29), then executes an analyst-specified number of time steps (l.30), and computes the distance-based mesh size field (l.31). The solver then writes the computed mesh size field and solution field to files/streams. Those fields are read on Line 11 and attached to the PUMI mesh. Next, chef drives MeshAdapt with the mesh size field (l.20). To prevent memory exhaustion during mesh refinement procedures, ParMETIS part k-way graph repartitioning (via Zoltan) is called using weights that approximate the change in mesh element count on each part (*l*.22, 25). After adaptation, chef executes preprocessing as previously described (*l*.14).

Algorithm 4 Two-phase PHASTA-chef Adaptive Loop

```
1:  procedure adaptiveLoop(in $max\_steps$)
2:      $pumi\_mesh \leftarrow$ load the partitioned PUMI mesh from disk
3:      $geom \leftarrow$ load the geometric model from disk
4:      $chef\_probdef \leftarrow$ load the chef problem definition from disk
5:      initialize $file\_handle$ for streams or POSIX I/O
6:      preprocessor($pumi\_mesh$,$geom$,$chef\_probdef$,$file\_handle$)
7:      $step\_number \leftarrow 0$
8:      while $step\_number < max\_steps$ do
9:          PHASTA($N$,$file\_handle$)
10:         $step\_number \leftarrow step\_number + N$
11:         read $size\_field$ and $phasta\_fields$ from $file\_handle$ and attach to $pumi\_mesh$
12:         meshadapt($pumi\_mesh$,$size\_field$,$max\_iterations$)
13:         ParMA(vtx>elm,$pumi\_mesh$)
14:         preprocessor($pumi\_mesh$,$geom$,$chef\_probdef$,$file\_handle$)
15: procedure preprocessor(in $pumi\_mesh$, in $geom$, out $chef\_probdef$, in/out $file\_handle$)
16:     reorder the mesh entities holding degrees-of-freedom
17:     $phasta\_mesh \leftarrow$ create PHASTA mesh data structures
18:     $phasta\_fields \leftarrow$ create PHASTA field data structures
19:     write $phasta\_mesh$ and $phasta\_fields$ to $file\_handle$
20: procedure MeshAdapt (in/out $pumi\_mesh$, in $size\_field$, in $max\_iterations$)
21:     $w \leftarrow$ per element field estimating the change in element volume based on $size\_field$
22:     predictively balance the mesh elements for element weight $w$
23:     for $iteration \leftarrow 0$ to $max\_iterations$ do
24:         coarsen the mesh
25:         re-balance the mesh elements for element weight $w$
26:         refine the mesh
27:     re-balance the mesh elements
28: procedure PHASTA(in $N$, in/out $file\_handle$)
29:     read $phasta\_mesh$, $phasta\_fields$ data from $file\_handle$
30:     run the flow solver for $N$ steps
31:     $size\_field \leftarrow$ isotropic size field based on distance to phasic interface
32:     write the mesh $size\_field$ and $phasta\_fields$ to $file\_handle$
```

We measured the performance of PHASTA-chef [160] POSIX file and data stream information exchange in a workflow supporting the adaptive analysis of a

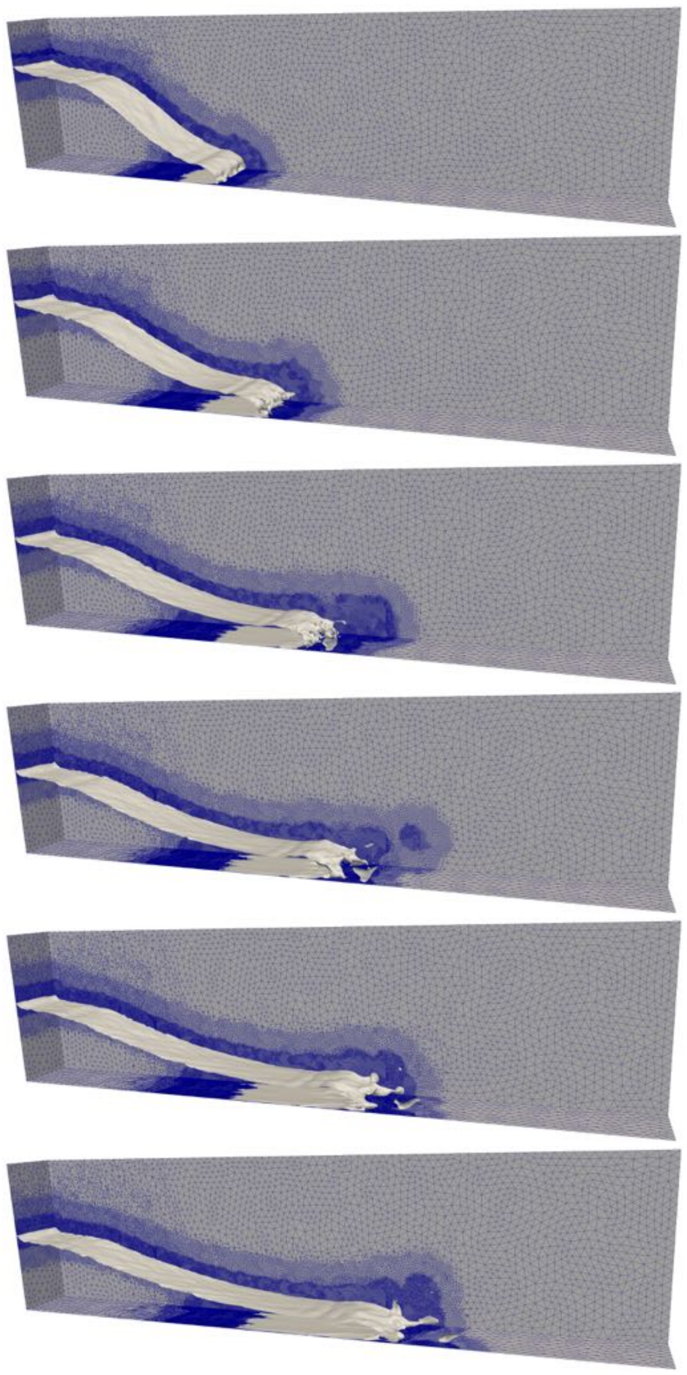
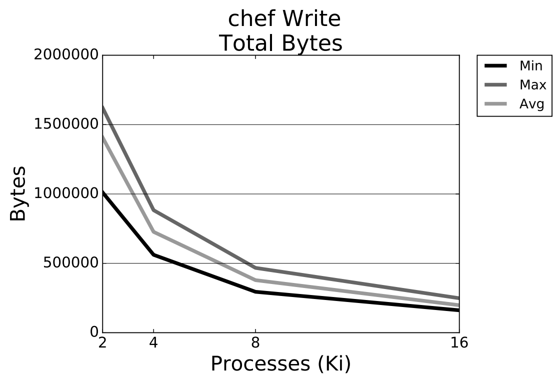


Fig. 4.1. Evolution of an adaptive dam-break case ran on 2048 processes of the ALCF Theta system using two-phase, incompressible PHASTA coupled to PUMI unstructured mesh adaptation with data streams. Each image (top to bottom) represents an advancement in physical time by 1/100 of a second. The air-water phasic interface iso-surface is shown in gray.

two-phase, incompressible dam-break flow, as shown in Fig. 4.1. Workflow tests ran on the Intel Knights Landing Theta Cray XC40 system at the Argonne Leadership Computing Facility (ALCF) using 64 processes per node with a total of 2Ki, 4Ki, 8Ki, and 16Ki processes. All nodes were configured in the 'cache-quad' mode [152], [153]. The two Theta filesystems used by POSIX file tests, GPFS [161] and Lustre [162], were in their default configuration for all runs. Test time is recorded using the low-overhead Read Time-Stamp Counter instruction (`rdtsc()`) provided by the Intel compiler. Unlike some other high resolution timers, `rdtsc()` is not affected by variations to the Knights Landing core frequency [153].

Each test initially loads the same mesh with 2Ki parts and 124 million elements. For the tests running on 4Ki, 8Ki, or 16Ki processes the first step is to partition the mesh using a graph-base partitioner to the target number of processes. Once partitioned, the chef preprocessor is executed. The preprocessor reads the solution field produced by PHASTA, balances the mesh using ParMA [103], and then creates and writes the PHASTA mesh and field data structures. Following the initial preprocessing, the test executes seven solve-then-preprocess cycles. In the adaptive workflow used to study the dam-break flow (shown in Fig. 4.1) the preprocess step is preceded by execution of MeshAdapt. For our information exchange performance tests though, this step is not necessary. Since we are not adapting the mesh, the mesh size does not change during the test. Combining this preprocess-only approach with a limited PHASTA flow solver execution mode we can force the workflow to perform the same work in each cycle.

After preprocessing with chef, the workflow executes the PHASTA solver. PHASTA starts by reading the mesh and field structures produced by chef, and then executes one time step with field updates disabled. With the field updates disabled the time spent in the solver is the same in each cycle. While this configuration does not produce meaningful flow results, it performs sufficient linear system solve work to emulate the data access and movement of multiple complete solution steps. Once the linear system is solved, PHASTA writes the solution field and control passes back to chef to run the preprocessor. After six more solve-then-preprocess cycles, the test is complete.



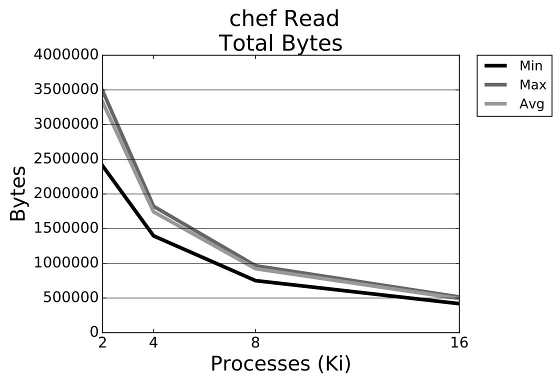


Fig. 4.2. chef total bytes read and written per process. PHASTA reads(writes) the same number of bytes that chef wrote(reads).

The minimum, maximum, and average number of bytes read and written per process in a cycle by chef and PHASTA is plotted in Fig. 4.2. Since we have a fixed mesh, the bytes read/written in each cycle is the same. This extends across the different I/O method tests (streams, POSIX, ramdisk) as the initial partitioning and load balancing called during preprocessing is deterministic. Note, in the tested configuration PHASTA writes additional fields that are not required for input. Due to the lack of these additional fields the chef byte count is smaller for write than read, while the PHASTA byte count is smaller for read than write.

While it may be tempting to report the impact of I/O on the overall workflow execution time, we omit this statistic as it is highly dependent on the application and the time it spends in the flow solver and adaptation procedures. Specifically, as the number of steps of the flow solver executed between each adaptation increases, the fraction of time spent in I/O decreases. If the implicit solve were replaced by an explicit solve, then the solve time may decrease by an order of magnitude. Lastly, the number of entities modified or created during adaptation strongly impacts the fraction of time spent adapting the mesh. Prior to this work, the large time spent reading and writing files drove research towards less frequent adaptation to amortize the I/O time. The dramatic reduction of time in data transfer provides alternatives. For these reasons, we choose to primarily report the performance of the approaches in terms of direct time spent transferring data between components.

The time spent by chef transferring data to and from PHASTA is reported

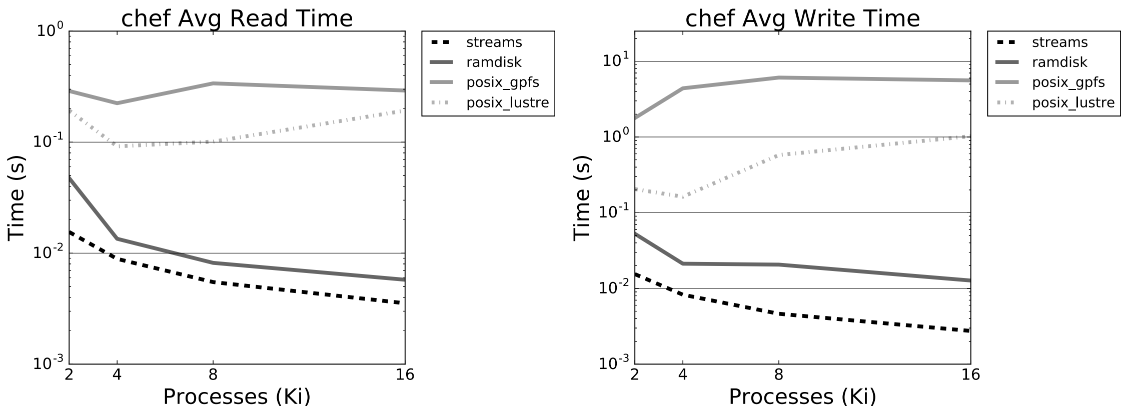
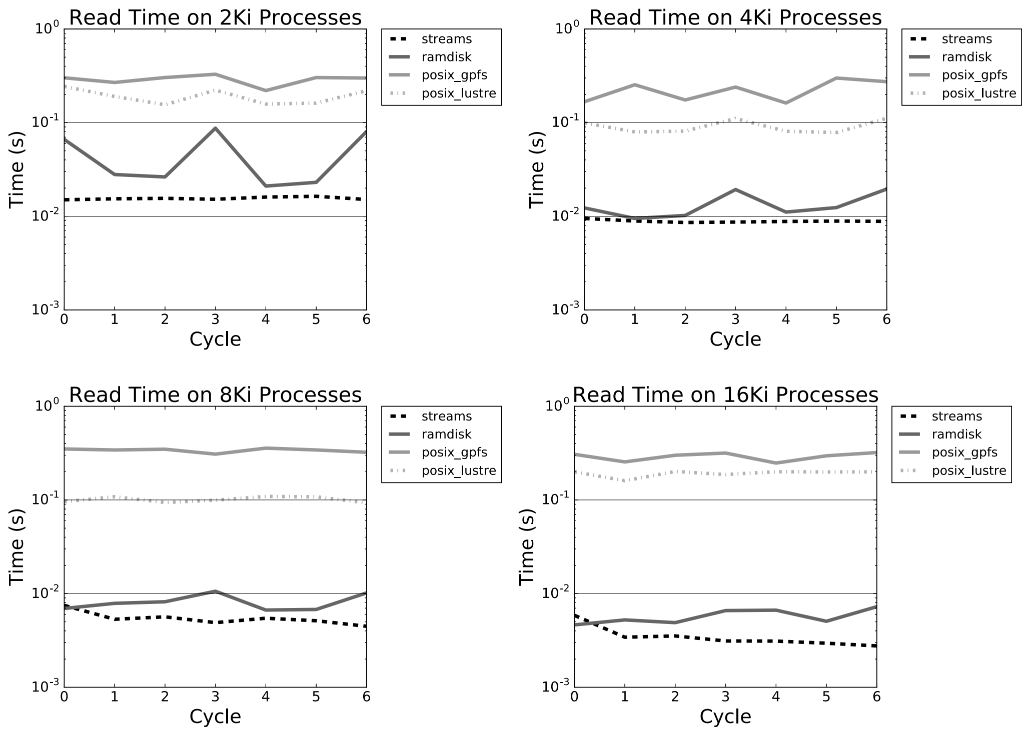


Fig. 4.3. chef streams, ramdisk, and POSIX average read and write time on ALCF Theta. Lower is better.

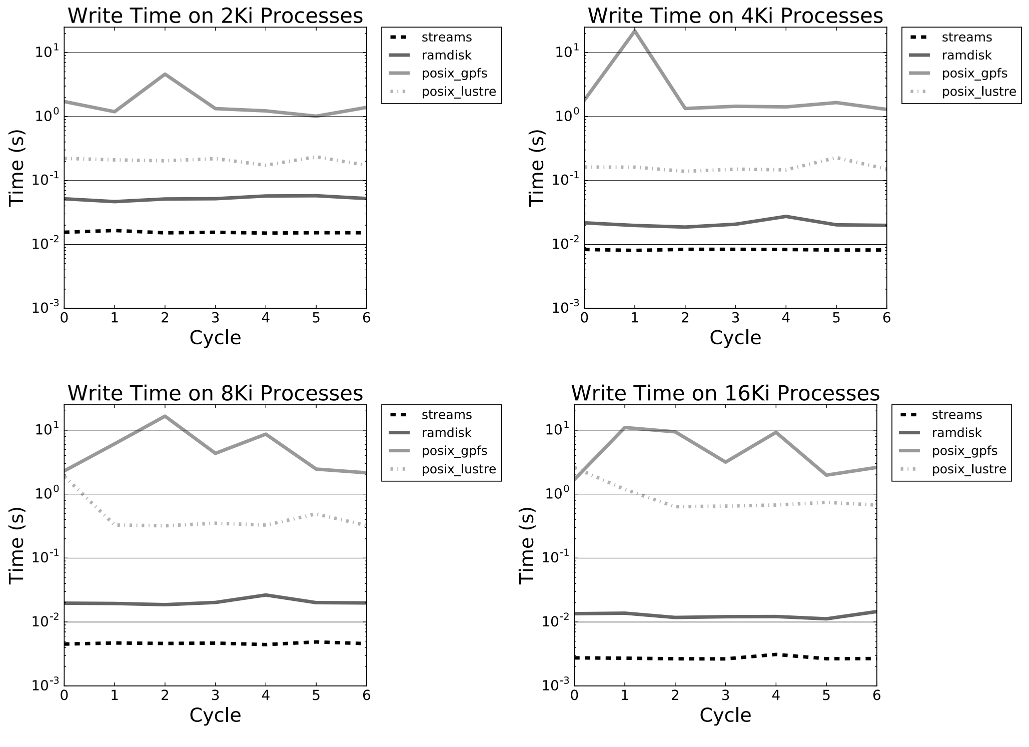
in Fig. 4.3 and Fig. 4.4. Note, the PHASTA times for these transfers are nearly identical and not reported here. Fig. 4.3 depicts the average time spent reading and writing at each process count using data streams, a 96GB ramdisk in main memory (DRAM), and the GPFS and Lustre filesystems. At each process count Fig. 4.4a and Fig. 4.4b depict the time spent reading and writing in each solve-preprocess cycle. The read time is reported for the function responsible for opening the PHASTA file/stream containing solution field data, reading the data, attaching the data to the mesh, and closing the file/stream. Likewise, the write time includes the time to open, write, and close, plus the time to detach the solution data from the mesh.

Across all process counts read and write times are highest when using POSIX files on the GPFS filesystem. The Lustre filesystem performs better, especially for writes, and has significantly lowered variability between cycles. As expected though, Lustre is slower than the ramdisk and streams. Stream writes and reads outperform Lustre by over an order of magnitude at all process counts. At 8Ki and 16Ki the performance gap widens to over two orders of magnitude. Also, note that the stream and ramdisk performance improves with the increase in process count and reduction in bytes transferred per process (see Fig. 4.2), whereas the filesystem performance degrades for Lustre and remains flat for GPFS. Clearly, avoiding operations accessing the shared file system can save a significant amount of time over the course of a parallel adaptive analysis.

Furthermore, serial testing on one Theta node indicates that using preallo-



(a) **chef read times.**



(b) **chef write times.**

Fig. 4.4. Time for chef to read and write using streams, ramdisk, and POSIX on ALCF Theta. Lower is better.

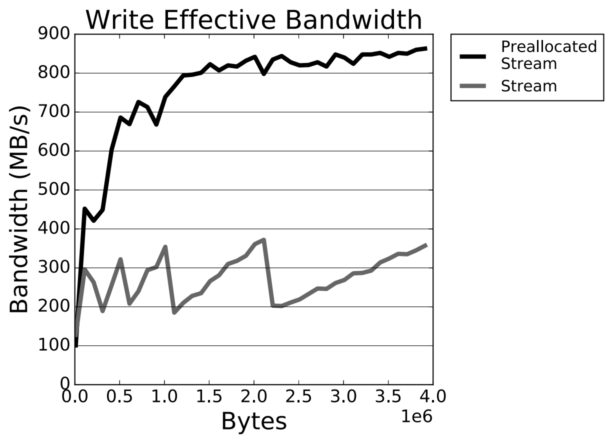


Fig. 4.5. Streaming write performance with and without preallocation on a single node of ALCF Theta in the cache-quad configuration. Higher is better. The code used for these tests is described in the Appendix.

cated buffers with `open_memstream` can further improve streaming write performance by over two times. The performance penalty of dynamic buffer expansion for the non-preallocated writes can clearly be seen in Fig. 4.5 by the large drop in effective (bytes/time(open+write+close)) bandwidth at approximately 0.25MB, 0.5MB, 1MB and 2MB. Likewise, POSIX file performance may be improved through use of the POSIX asynchronous I/O interface (`aio`) [163], but we have not tested these APIs on Theta.

4.3 Albany Solderball

Albany [164], [165] is a parallel, implicit, unstructured mesh, multi-physics, finite element code used for the solution and analysis of partial differential equations. The code is built on over 100 software components and heavily leverages packages from the Trilinos project [79]. Both Albany and Trilinos adopt an 'agile components' approach to software development that emphasizes interoperability. Albany has been used to solve a variety of multi-physics problems including ice sheet modeling and modeling the mechanical response of nuclear reactor fuel. The largest Albany runs have had over a billion degrees of freedom and used over 32Ki cores. Albany's high performance, generality, and component-based design made it an ideal candidate for the construction of an in-memory adaptive workflow using bulk API-based transfers.

The Albany analysis code provides an abstract base class for the mesh discretization. Implementing the class with PUMI's complete topological mesh representation simply required understanding Albany's discretization structures. Like most finite element codes, Albany stores a list of mesh nodes and a node-to-element connectivity map to define mesh elements. Albany's Dirichlet and Neumann boundary conditions though, need additional data structures. The Dirichlet boundary condition data structure is simply an array of constrained mesh nodes. The more complex Neumann boundary condition structure requires lists of mesh elements associated with constrained mesh faces; a classification check followed by a face-to-element upward adjacency query. Algorithm 5 details this process. Here the notation M_j^d (G_j^d) refers to the j^{th} mesh (model) entity of dimension d, $M_j^d \sqsubset G$ returns the geometric model classification of M_j^d, and $\{M_i^d\{M^q\}\}$ is the set of mesh entities of dimension q that are adjacent to M_i^d. The PUMI implementation of Albany's discretization and boundary condition structures allows us to define and solve complex problems without having to create a second complex mesh data structure (e.g., a Trilinos STK mesh).

Algorithm 5 Construction of Neumann Boundary Condition Structure

```
1:  // store the mapping of geometric model faces to side_sets
2:  invMap ← mapping of $G_j^2$ to side_sets
3:  size_set_list ← $\emptyset$
4:  for all $M_i^2 \in \{M^2\}$ do
5:      // get the geometric model classification of the mesh face
6:      $G_j^d \leftarrow M_i^2 \sqsubset G$
7:      if $G_j^d \in$ invMap then
8:          // for simplicity of the example we assume the model is manifold
9:          // upward adjacent element to the mesh face
10:         $M_j^3 \leftarrow \{M_i^2\{M^3\}\}$
11:         // collect additional element and face info
12:         elm_LID ← local id of $M_j^3$
13:         side_id ← local face index of $M_i^2$
14:         side_struct ← {elm_LID,side_id,$M_j^3$}
15:         insert side_struct into side_set_list
```

We ran an in-memory adaptive simulation of a solderball array subject to

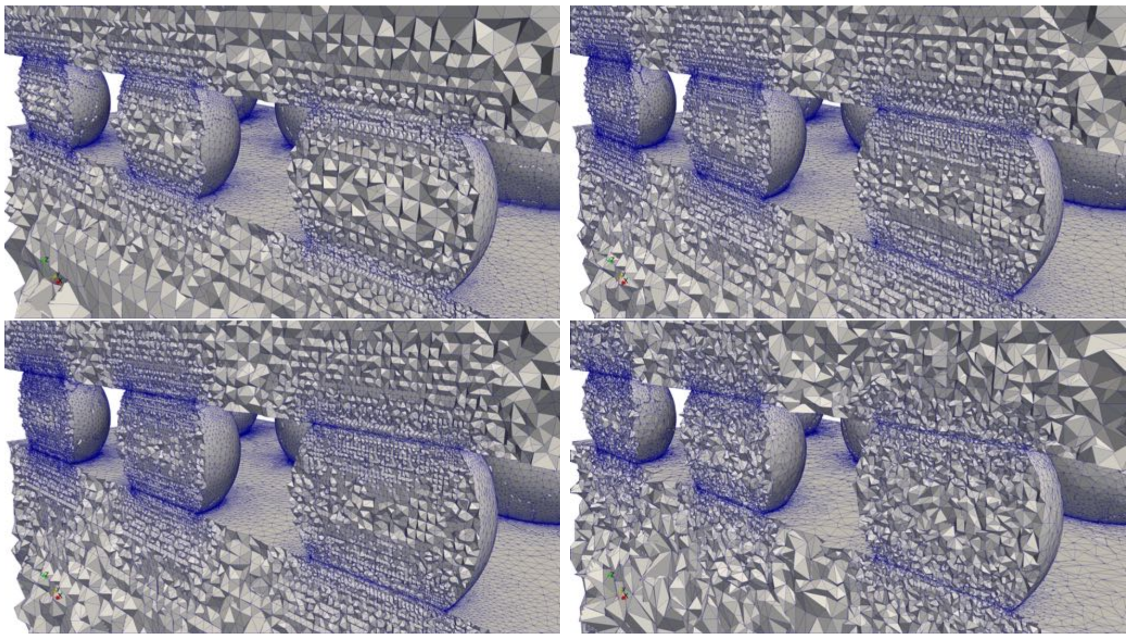


Fig. 4.6. Four adaptation cycles (top to bottom, left to right) of 3x3 solderball mesh. The mesh is refined near the high stress gradients at the interface between the solderballs and the upper and lower slabs.

thermal creep [166]. Fig. 4.6 depicts the results of the parallel adaptive analysis using the in-memory integration of SPR and the PUMI unstructured mesh tools with Albany. The adaptive workflow ran four solve-adapt cycles on 256, 512, and 1024 cores of an IBM Blue Gene/Q using an initial mesh of 8M tetrahedral elements. The adapted meshes contain only negligible differences across the range of core counts.

Algorithm 6 lists the steps of the Albany-PUMI adaptive workflow. The workflow begins by loading the PUMI mesh, geometric model, and XML formatted problem definition (l.2-4). It then creates the node and element mesh connectivity (l.5) and sets of mesh entities with boundary conditions (l.6) for Albany. Next, the workflow enters into the solve-adapt cycle(l.8). Note, throughout the cycle the PUMI mesh is kept in memory. At the top of the cycle one load step is solved (l.9). Following the load step, the solution information (a displacement vector at mesh nodes) and history-dependent state variables at integration points are passed in-memory to an APF `FieldShape` [11] (l.10). SPR then computes mesh-entity level error estimates based on an improved Cauchy stress field (l.11). The estimated error is then transformed into an isotropic mesh size field, which MeshAdapt then uses to drive local mesh modification procedures (l.12). As the mesh modifications (split, collapse, etc...) are applied the `FieldShape` transfer operators [167], [168] are called to determine the value of state variables at repositioned or newly created integration points. After mesh coarsening, Zoltan's interface [13] to ParMETIS is called to predictively balance the mesh and prevent memory exhaustion on parts where heavy refinement occurs. Once adaptation is complete ParMA rebalances the mesh (l.13) to reduce element and vertex imbalances for improved linear system assembly and solve performance. The adaptive cycle concludes with the transformation of PUMI unstructured mesh information (l.14-15) and APF field information (l.16) into Albany analysis data structures.

Algorithm 6 Albany-PUMI Adaptive Loop

```
 1: procedure adaptiveLoop(max_steps)
 2:     pumi_mesh ← load the partitioned PUMI mesh from disk
 3:     geom ← load the geometric model from disk
 4:     probdef ← load the Albany problem definition from disk
 5:     createConnectivity(pumi_mesh)
 6:     createNodeAndSideSets(pumi_mesh,probdef)
 7:     step_number ← 0
 8:     while step_number < max_steps do
 9:         solveLoadStep(step_number++)
10:         getFields(pumi_mesh)
11:         size_field ← SPR(pumi_mesh)
12:         MeshAdapt(pumi_mesh,size_field)
13:         ParMA(vtx>elm,pumi_mesh)
14:         createConnectivity(pumi_mesh)
15:         createNodeAndSideSets(pumi_mesh,probdef)
16:         setFields(pumi_mesh)
```

Fig. 4.7 depicts the factor of two performance advantage of in-memory transfers of fields and mesh data between Albany, PUMI, and SPR versus the writing of the mesh to POSIX files. Based on this data we estimate the performance advantage of the in-memory approach over a file-based loop that both reads and writes files to be about four times higher. Another advantage demonstrated by this data is the low in-memory transfer time imbalance; defined as maximum cycle time divided by the average cycle time. The in-memory approach has less than a 6% imbalance across all core counts while the file writing approach has a 22% imbalance at 512 cores

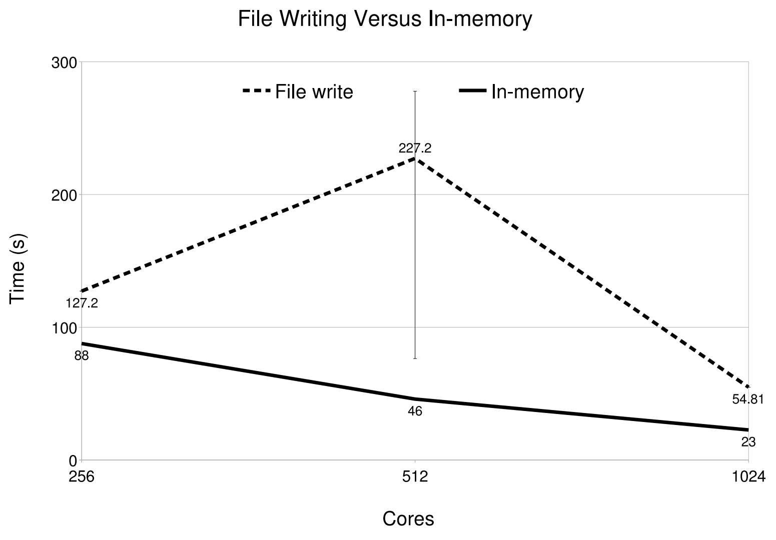


Fig. 4.7. Average per cycle file writing and in-memory transfer times. Minimum and maximum bars are only shown for the 512 core file writing data point where they are 6% more, or less, than the mean, respectively. Lower is better.

(as shown by the large error bar in Fig. 4.7). Since the heaviest parts in our test meshes have at most 5% more elements and 12% more vertices than the average part, and the data transfers are proportional to the number of mesh vertices and elements on each part, then we conclude that the observed imbalance in file-based I/O is attributable to shared filesystem resource contention.

Using the Albany-PUMI workflow we also simulated the tensile loading of the 2014 RPI Formula Hybrid race car suspension upright. Fig. 4.8 depicts the upright in its initial state, and after multiple load steps. Without adaptation the severe stretching of domain would result in invalid elements and the subsequent failure of the analysis.

In the following section we couple PUMI to another modular C++ analysis package. Unlike Albany though, the provided unstructured mesh APIs are less well-defined and require a different approach.

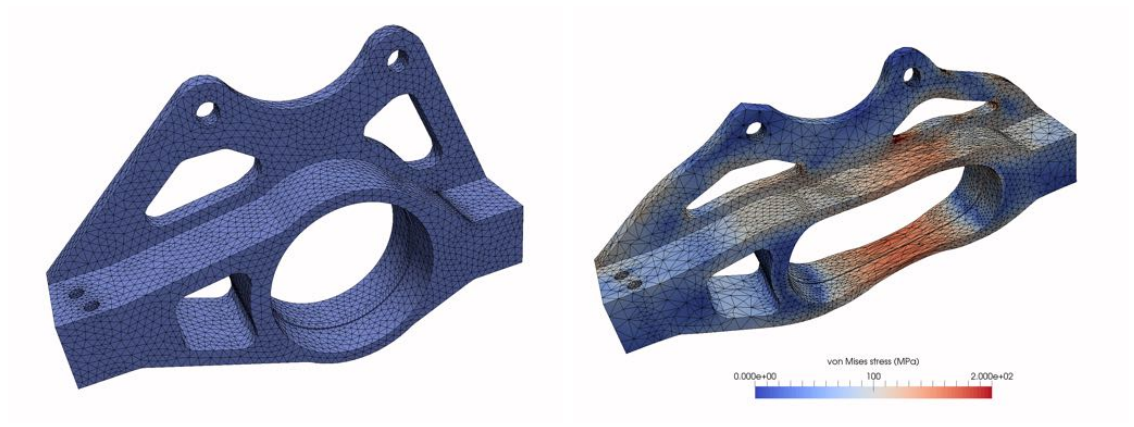


Fig. 4.8. Large deformation of the RPI Formula Hybrid suspension upright [169].

4.4 Omega3P Cavity Frequency Analysis

Omega3P is a C++ component within ACE3P for frequency analysis of linear accelerator cavities [170]. It is built upon multiple components that include distributed mesh functionality (DistMesh), tensor field management, vector and matrix math, and many linear solvers. Our in-memory integration of PUMI with Omega3P leverages these APIs to execute bulk mesh and field transfers, and atomic element Jacobian transfers for element stiffness matrix formation.

In the previous section we discussed a similar in-memory integration for efficient parallel adaptive workflows with Albany. In Omega3P, as with Albany, we again assume a small increase in memory usage from storing both the PUMI mesh and Omega3P DistMesh. This small memory overhead lets us avoid spending time destroying and reloading the PUMI mesh after the adaptation and analysis steps, respectively. Furthermore, having access to the PUMI mesh supports the atomic transfer of exact geometry of curved domains needed for calculation of mesh element Jacobians during element stiffness matrix formation. This capability is critical in Omega3P for maintaining convergence of higher order finite elements when the geometric model has higher order curvature [112], [171], [172].

Algorithm 7 lists the steps needed to compute the exact Jacobian using the APF `getJacobian(...)` API and its underlying basis functions. To set up the Jacobian computation, during the PUMI-to-Distmesh conversion, a pointer to each

PUMI mesh element is stored with the corresponding DistMesh element object as they are being created. As the DistMesh elements are being traversed for element stiffness matrix assembly the PUMI element pointer is retrieved (l.3). With this pointer and the barycentric coordinates of the element (l.7) the 3x3 Jacobian matrix is computed with the call to the APF `FieldShape` `getJacobian` function (l.9).

Algorithm 7 Jacobian Calculation for Matrix Assembly

```
 1: // loop over DistMesh elements
 2: for all $M_i^3 \in \{M^3\}$ do
 3:     $pumiElementPtr \leftarrow getPumiElementPtr(M_i^3)$
 4:     for all integration points do
 5:         // compute element Jacobian using APF's FieldShape class
 6:         // associated with the PUMI mesh element
 7:         $xi \leftarrow$ getBaryCentricCoords(integration point)
 8:         apf::Matrix3x3 $J$
 9:         apf::getJacobian($pumiElementPtr$,$xi$,$J$)
10:         // complete element matrix computation
11:     // insert element matrix contributions into stiffness matrix
```

The mesh management and computational steps in the adaptive Omega3P-PUMI workflow are listed in Algorithm 8. Fig. 4.9 depicts adapted meshes and fields generated using this process. Execution of the workflow begins by loading a distributed PUMI mesh and the geometric model (l.2-3). Next, ParMA balances the owned and ghosted mesh entities holding degrees of freedom (edges and faces for quadratic Nedelec shape functions [173]) (l.5). PUMI APIs are then used to create a DistMesh instance from the balanced PUMI mesh (l.6); a bulk transfer. The time required for this procedure is less than 0.1% of the total workflow execution time. Next, the workflow runs the solve-adapt cycle until the eigensolver has converged (l.7). Note, the atomic Jacobian transfer of Algorithm 7 occurs during the eigensolver execution. Following the solver's execution, the electric field is attached to the PUMI mesh (l.8) via a bulk transfer, the DistMesh is destroyed (l.9), a size field is computed by SPR (l.10), and the mesh is adapted with PUMI (l.11). The cycle ends by balancing the PUMI mesh with ParMA and creating a new DistMesh.

The increase in peak memory usage from storing two copies of the mesh and field data is insignificant relative to the applications overall memory usage.

Algorithm 8 Omega3P-PUMI Adaptive Loop

```
1: procedure ADAPTIVELOOP($max\_steps$)
2:     $pumi\_mesh$ ← load the partitioned PUMI mesh from disk
3:     $geom$ ← load the geometric model from disk
4:     $probdef$ ← load the Omega3P problem definition from disk
5:     PARMAGHOST(edge=face>rgn,$pumi\_mesh$)                      ▷ quadratic Nedelec
6:     $dist\_mesh$ ← CREATEDISTMESH($pumi\_mesh$)                  ▷ bulk
7:     while not ($converged$ ← EIGENSOLVER($dist\_mesh$)) do      ▷ atomic
8:         GETELECTRICFIELD($pumi\_mesh$)                          ▷ bulk
9:         DESTROY($dist\_mesh$)
10:        $size\_field$ ← SPR($pumi\_mesh$)
11:        MESHADAPT($pumi\_mesh$,$size\_field$)
12:        PARMAGHOST(edge=face>rgn,$pumi\_mesh$)                  ▷ quadratic Nedelec
13:        $dist\_mesh$ ← CREATEDISTMESH($pumi\_mesh$)              ▷ bulk
```

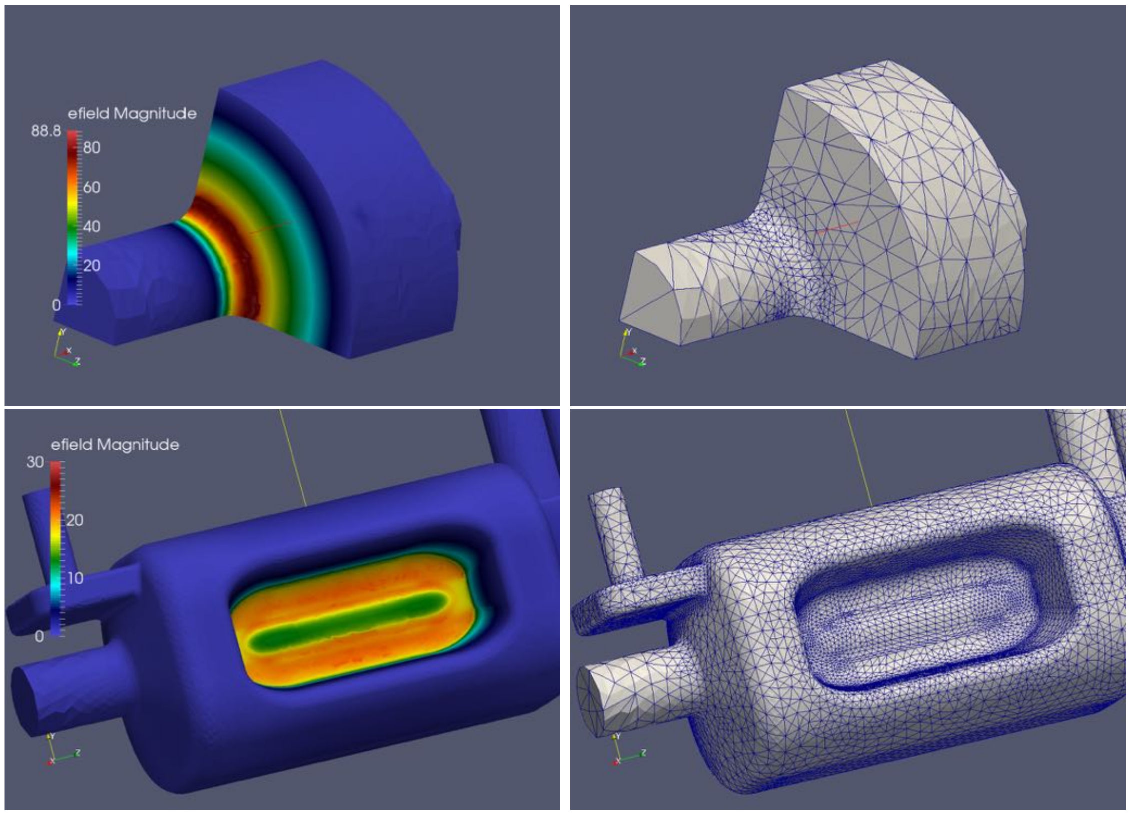


Fig. 4.9. The first eigenmode electric field (left column) and adapted meshes (right column) for the `pillbox` (top row) and `cav17` (bottom row) test cases.

Fig. 4.10 shows the peak per node memory usage over the entire Omega3P execution on the `cav17` and `pillbox-2M` cases for both the original Omega3P code and with the code that executes PUMI mesh conversion and load balancing (labelled as Omega3P+PUMI). In the `cav17` test case (top half of Fig. 4.10), the peak memory when storing the PUMI mesh increases by 2% at 32 cores and by 6% at 128 cores, and decreases slightly at 64 cores (less than 1%). On the other hand, for the pillbox-2M case at 256, 512, and 1024 cores the peak memory is actually reduced by 0.87%, 1.1% and 2.9%, respectively. The small reduction is the result of differences in the mesh loading and balancing processes. Specifically, at 256 processes ParMA balanced the mesh elements (owned and ghosted) in the PUMI workflow to a 14% imbalance while the non-PUMI workflow using ParMETIS has a 38% imbalance. These results show that the in-memory integration has an insignificant memory overhead.

4.5 Summary

In-memory parallel adaptive workflows for three applications have been demonstrated using bulk data streams, bulk APIs, and a combination of bulk and atomic APIs. The in-memory transfer of data was significantly faster than file-based transfers for PHASTA and Albany, while the memory overhead for Omega3P was insignificant. Key to the three couplings was the use of PUMI's component APIs for queries to, and modifications of, the mesh, its partitioning, and its associated fields.

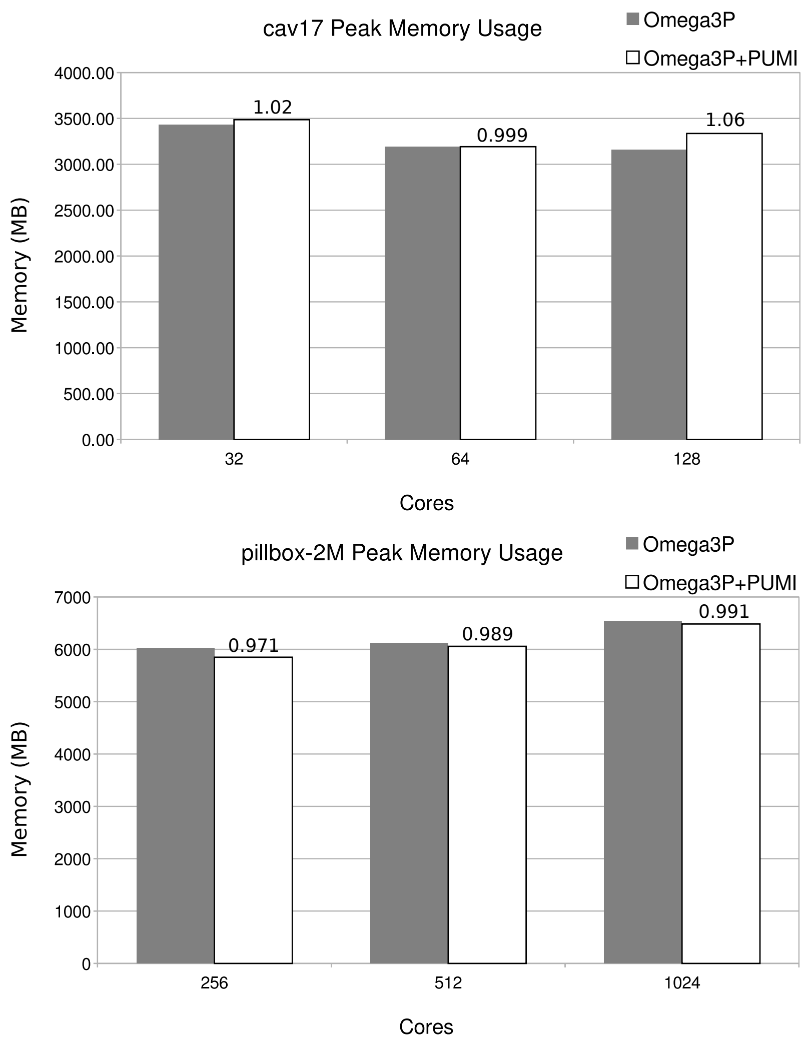


Fig. 4.10. Peak per node memory usage for two Omega3P and Omega3P+PUMI test cases: (top) cav17 and (bottom) pillbox-2M. The numbers above the Omega3P+PUMI bars list the ratio of the peak memory used by Omega3P+PUMI relative to the peak memory used by Omega3P.

CHAPTER 5
WORKFLOWS DEVELOPED FOR THE NEW YORK STATE HIGH PERFORMANCE COMPUTING CONSORTIUM

The High Performance Computing Consortium (HPCNY), supported by the Empire State Development Division of Science, Technology and Innovation (NYSTAR), is a multi-year effort to address the impediments listed in Section 3.3. HPCNY supports computational scientists to work directly with New York State industry to apply massively parallel simulations on supercomputer systems. Computational scientists are based at Rensselaer Polytechnic Institute, University of Buffalo, SUNY Stoneybrook/Brookhaven National Lab, Icahn School of Medicine at Mount Sinai, and Marist College. Critical to the success of the computational scientists is the base institution's faculty with extensive knowledge of high performance computing in a broad range of application areas, existing industrial and software vendor collaborations, and on-site HPC hardware systems programmers.

Industrial partners work with Rensselaer through HPCNY at the level needed to address their computing requirements. At one extreme are industrial partners that have the necessary technical personnel, business case, and software to utilize available HPC hardware. For these interactions an allocation on the Intel Xeon cluster and IBM Blue Gene/Q at the Rensselaer Center for Computational Innovations (CCI) paired with occasional help from systems programmers and computational scientist is sufficient.

The more common cases are industrial partners that identify a computing need, but face most of the impediments described in Section 3.3. For these interactions

Portions of this chapter previously appeared as: M. S. Shephard, C. Smith, and J. E. Kolb, "Bringing hpc to engineering innovation," *Comput. in Sci. & Eng.*, vol. 15, no. 1, pp. 16–25, Feb. 2013

Portions of this chapter previously appeared as: C. W. Smith, S. Tran, O. Sahni, F. Behafarid, M. S. Shephard, and R. Singh, "Enabling HPC simulation workflows for complex industrial flow problems," in *Proc. XSEDE Conf.: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, 2015, pp. 41:1–41:7

the appropriate combination of computational scientists, researchers, and software vendors are brought to bear on the problem. In many cases the initial statement of the problem is of broad scope. Thus, the first task then is to define a specific problem that includes the relevant physical phenomena and geometric complexity. Execution of this first problem demonstrates the computational and analytical performance of the chosen HPC software technologies as well as the ability of those technologies to integrate into simulation workflows accessible to the industrial partners. From a successful first problem demonstration the business case can be stated and supported. For the technical contact at the industrial partner the business case often provides the necessary management level support to continue work with HPCNY computational scientists to generalize the workflows for broader use.

While technical interactions proceed, HPCNY business administrators and computational scientists work with the industrial partners to collect economic impact statements and write press releases summarizing the interactions. Despite their non-technical nature, these materials are critical for the success and growth of the consortium. For NYSTAR they establish the return-on-investment needed to justify continued funding from the state. For other New York State companies they provide examples of prior success that are often enough to motivate a first contact with HPCNY.

Companies typically understand that there is potential for HPC to improve their competitive advantage but face some technical impediments. In the following sections we describe how the technical impediments are addressed for three industrial flow problems. The first problem (Section 5.1) is an example where we address the limited fidelity and performance provided by a commercial CFD software suite by coupling scalable and efficient mesh adaptation with a massively parallel, open-source, CFD analysis component. In the second problem (Section 5.2), simulating the flow of a non-Newtonian fluid through a twin-screw extruder, we apply a similar approach, but also address another critical bottleneck; the manual transfer of information between the engineer and the software. In a production environment, this transfer can be just as limiting to productivity as a poorly performing flow solver, or a file-based transfer between parallel components in a workflow. Thus, for the extruder workflow we automated the process of executing the workflow on remote parallel systems by defining a web-based gateway for job submission and management. In the third example problem (Section 5.3), workflow automation is again the focus. For this workflow we automate the problem set up and execution steps required to run an ensemble of simulations for studying pump design using a closed-source CFD analysis framework.

5.1 Micromechanical Device Analysis

Micromechanical device engineers at a New York State equipment manufacturer study the design of a multi-phase flow system that is driven by a structural boundary condition. 3D simulations use a commercial CFD software suite ran on in-house multi-core workstations. But, engineers typically run reduced fidelity 2D simulations due to the long execution time of 3D simulations, limited computing resources, and finite design periods. In these simulations the fidelity is further reduced as the structure driving the boundary condition is not influenced by the fluid flow. Rensselaer computational scientists defined and demonstrated an end-to-end workflow for guiding device design that uses the PHASTA CFD [74], [158] suite of tools for automated 3D parallel adaptive simulations [10] that account for fluid-structure interactions.

Modifications to the PHASTA parallel fluid dynamics simulation software were required to couple it to the structural mechanics code provided by the device engineers. To support these interactions mechanisms were implemented that interpolate fields between the different time and spatial discretizations . Modifications were also made to the PHASTA error estimator to reduce discretization error and reduce computational costs. At the phasic interface, where there is a change in material properties, the mesh is refined. Away from the interface, the mesh is coarsened to reduce computational costs. Between the refined and coarse zones the mesh smoothly transitions between the two sizes [18], [19]. Fig. 5.1 depicts the mesh over six adaptation cycles. Simulations using this workflow were run on up to 512 processors of a CCI cluster. Increased scalability of this workflow is now possible through the in-memory coupling of the PHASTA analysis component with the PUMI mesh

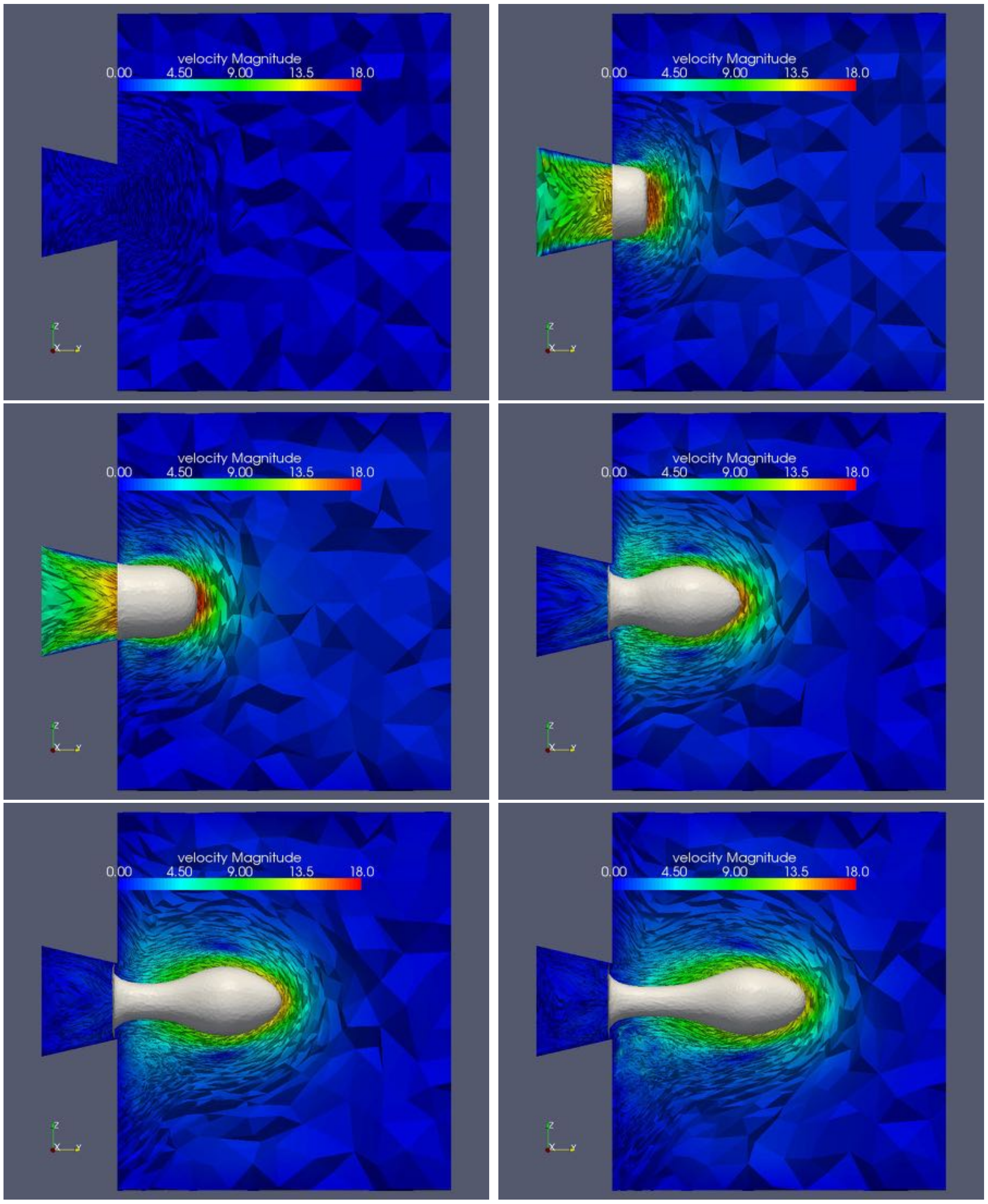


Fig. 5.1. An axial slice of the 3D mesh at six consecutive adaptation cycles in the multi-phase PHASTA simulation (top to bottom, left to right). The gray surface is the phasic interface.

adaptation and load balancing components described in Section 4.2. This scalable workflow enables device engineers to leverage HPC resources to run high fidelity system simulations in hours instead of days, and thus reduce the time required to develop a new device.

5.2 High-fidelity Viscous Flow Simulation

The second industrial flow problem to demonstrate component-based simulation workflows is a non-Newtonian viscous flow in a twin-screw extruder. The material flow of interest exhibits highly nonlinear behavior (e.g., follows generalized non-Newtonian constitutive law) along with nonlinear partial slip at the screw surfaces. Simulations support studying the processing performance of these materials within extruder systems that are designed with thin gaps between adjacent screws (e.g., twin-screw extrusion) and with passage walls.

The PHASTA CFD analysis component was selected for these simulations due to our extensive experience with it, its support for non-Newtonian material models [174], [175], non-linear partial slip boundary conditions, and scalability. PHASTA inputs are created with the chef pre-processing tool using meshes generated with Simmetrix MeshSim [122]. Twin screw extuder meshes produced by MeshSim have automatically generated layered element structures that span thin section gaps and unstructured tetrahedron away from the gaps.

The geometry of the extruder is composed of complex curved surfaces with corners and thin gaps at which critical physics occurs. High aspect-ratio semi-structured boundary layer meshes are constructed over these complex surfaces, including in thin gaps, and appropriately graded into the general unstructured mesh in the remainder of the domain. The mesh, and axial velocity of the flow, is shown in Fig. 5.2 across two threads and an axial section. For this problem with multiple threads of the screw and tighter gaps, meshes can range from 20 to 50 millions elements in order to obtain accurate solution. In-turn, to obtain the solution in a reasonable time frame, less than a few hours, the PHASTA flow analysis requires on the order of 500 cores.

It should be noted that while these research efforts were being pursued com-

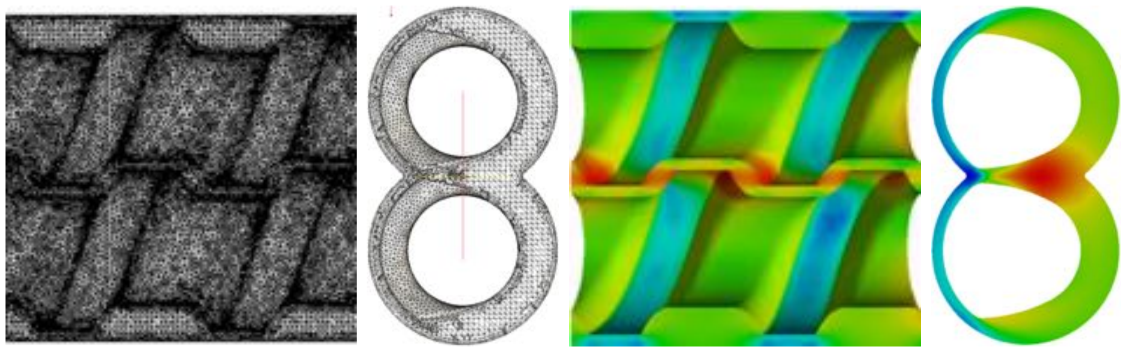


Fig. 5.2. Twin-screw extruder (left) mesh and (right) axial velocity: (sub-left) two threads of the screw and (sub-right) cross-section of the extruder.

putational scientists also worked with the industrial partner's domain experts to support their sub-continuum modeling efforts. These researchers and engineers had experience using the computational tools, but needed assistance installing and optimizing them for the Rensselaer Blue Gene/Q system. Performance tuning identified installation options, problem sizes, and run time environment options that increased simulation efficiency of the *ab initio* packages VASP and GROMACS, and the molecular statics/dynamics package LAMMPS.

Science domain experts are often proficient in the multiple aspects of remotely executing jobs on large parallel systems. Despite this apparent proficiency, they spend significant time overcoming problems that would be trivial for a systems programming expert. Furthermore, the solutions to said problems are often not aligned with best practices. Clearly, training is one path to recover productivity and increase skills in related critical areas (such as reproducibility and data management) but there is another approach. Web-based science gateways let a small team of system and programming experts create workflows for execution on multiple different remote systems without burdening the domain experts with details of each system. For the domain experts, they simply use a web-browser to upload their input data, set simulation control parameters, choose a parallel system to execute on and the level of parallelism, then submit the job. When execution completes the outputs are made available for download.

We have built a science gateway to support PHASTA users [176]. A science

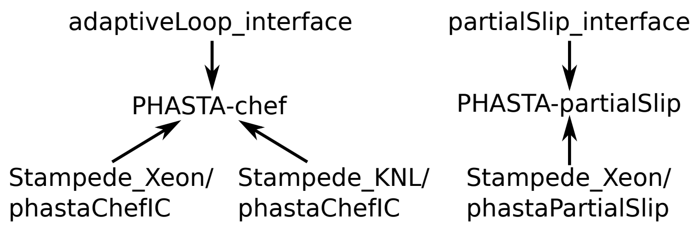

adaptiveLoop_interface
partialSlip_interface
PHASTA-chef
PHASTA-partialSlip
Stampede_Xeon/ phastaChefIC
Stampede_KNL/ phastaChefIC
Stampede_Xeon/ phastaPartialSlip

Fig. 5.3. PGA (top) Application Interfaces, (middle) Modules, and (bottom) Deployments defined for the PHASTA gateway workflows.

gateway is a community-developed set of tools, applications, and data collections that are integrated through a portal or a suite of applications. These gateway technologies support PHASTA workflows on HPC systems while hiding complexities such as data management, job scheduling, and run-time environment setup. The PHASTA science gateway was created using the PHP Reference Gateway for Airavata (PGA) [177]–[179] and is hosted in the XSEDE gateway hosting environment. PGA is a general-purpose gateway framework developed to enable scientific application in a browser environment. It provides user management, application cataloging and experiment management.

The PHASTA partial-slip workflow described in the previous section, and the in-memory adaptive workflow of Section 4.2, were implemented with two PGA Application Modules. Each module acts as a hub to associate workflow inputs and outputs (Application Interfaces) with the execution mechanisms (Deployments). For the partial-slip workflow the Deployment executes the serial pre-processing executable, followed by execution of the PHASTA binary with support for the necessary non-Newtonian fluid model and partial-slip boundary conditions. The in-memory workflow has two Deployments associated with it; one for execution of the PHASTA-chef binary on the TACC Stampede Xeon host processors and another for the Stampede KNL nodes. Fig. 5.3 depicts these associations.

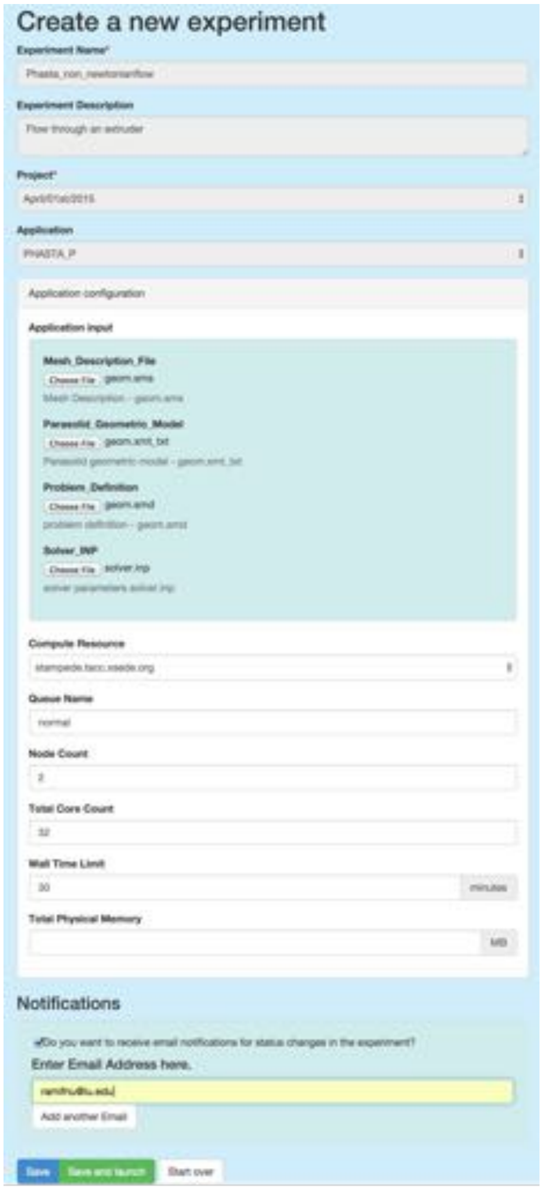
Simulations, called experiments in the gateway, are defined by uploading a set of input files that specify the problem definition, simulation parameters, and required compute resources. The left half of Fig. 5.4 depicts the experiment creation interface. The application inputs include the complete definition of the analysis domain via the geometric model (typically from a CAD software such as SolidWorks, NX, etc.) and the unstructured mesh. Also, included are information associated geometric model entities, such as physical attribute information (e.g., loads, boundary conditions, material properties) and simulation parameters (e.g., initial mesh control, time steps, convergence requirements, solver options, etc.). Lastly, the compute resource inputs specify the HPC system, TACCs Stampede system for this example, the node and core count, and the maximum run time.

Once the experiment is defined, clicking the Save and launch button shown in the left half of Fig. 5.4 will execute the PHASTA workflow. The experiment execution request is supported through APIs provided by SciGaP [178]. SciGaP APIs process the user request, create a job scheduler script specific to a compute resource (PBS, SLURM, etc.), and monitor the status of a job, as shown in the right half of Fig. 5.4. SciGaP also supports email notifications triggered by job status changes; an important mechanism for effective interactions with scheduled HPC systems. At the end of an experiment, the SciGaP service moves the outputs to the PGA storage location for users to download.

5.3 Semi-automated Pump Design

Hydraulic engineers at a New York State pump manufacturer study pump design over a range of operating conditions with the goal of developing an optimum pump configuration and geometry. Parallel CFD simulations provide a cost effective way to study the performance. One critical factor for providing competitive advantage is reducing the time needed to set up and run simulations. Since an ensemble of simulations are needed to cover the design space, a vast majority of the engineers' time is spent in simulation set up. Therefore, Rensselaer computational scientists worked with Simmetrix software engineers to define an automated workflow for setting up the ensembles. Setting up a simulation entails association of mesh generation controls and problem definition attributes with the geometric model, mesh generation, and creation of the CFD analysis software inputs.

The workflow depicted in Fig. 5.5 combines customizations to the Simmetrix and ANSYS tools for semi-automated set up and execution of an ensemble of ANSYS CFX [180] parallel simulations. Phase 1 of the workflow uses the Simmetrix Ab-



Create a new experiment

Experiment Name*: Phasta_non_newtonianflow

Experiment Description: Flow through an extruder

Project*: April/01st/2015

Application: PHASTA_P

Application configuration

Application input

Mesh_Description_File: geom.sms
Mesh Description - geom.sms

Parasolid_Geometric_Model: geom.xmt_txt
Parasolid geometric model - geom.xmt_txt

Problem_Definition: geom.smd
problem definition - geom.smd

Solver_INP: solver.inp
solver parameters solver.inp

Compute Resource: stampede.tacc.xsede.org

Queue Name: normal

Node Count: 2

Total Core Count: 32

Wall Time Limit: 30 minutes

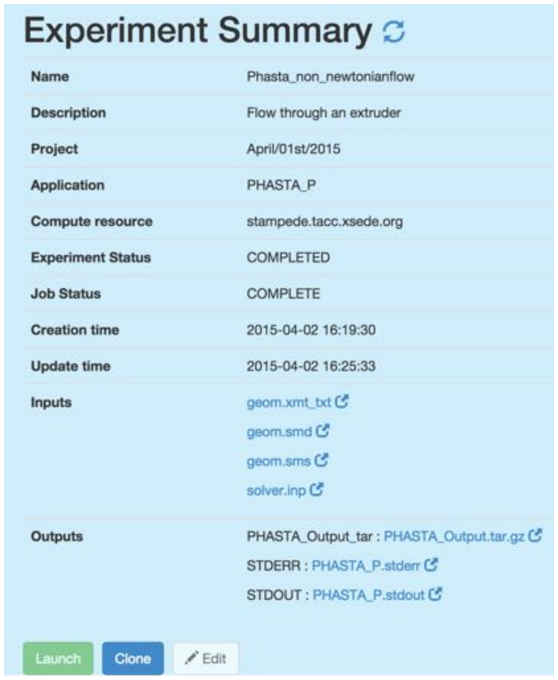
Total Physical Memory: MB

Notifications

✓ Do you want to receive email notifications for status changes in the experiment?

Enter Email Address here.

ramdhu@u.edu



Experiment Summary

Name	Phasta_non_newtonianflow
Description	Flow through an extruder
Project	April/01st/2015
Application	PHASTA_P
Compute resource	stampede.tacc.xsede.org
Experiment Status	COMPLETED
Job Status	COMPLETE
Creation time	2015-04-02 16:19:30
Update time	2015-04-02 16:25:33
Inputs	geom.xmt_txt geom.smd geom.sms solver.inp
Outputs	PHASTA_Output_tar : PHASTA_Output.tar.gz STDERR : PHASTA_P.stderr STDOUT : PHASTA_P.stdout

Fig. 5.4. PHASTA gateway experiment creation and management interface.

stractModel component [122], [181], mesh size control attributes, and a custom set of Attribute Definitions to support user creation of a problem template. The template defines the association of AbstractModel components representing features of the geometric model with the mesh and problem definition attributes. For example, the geometric model features of an impeller includes the blade, inlet, outlet, hub, and shroud surfaces. In the impeller template an inflow boundary condition will be

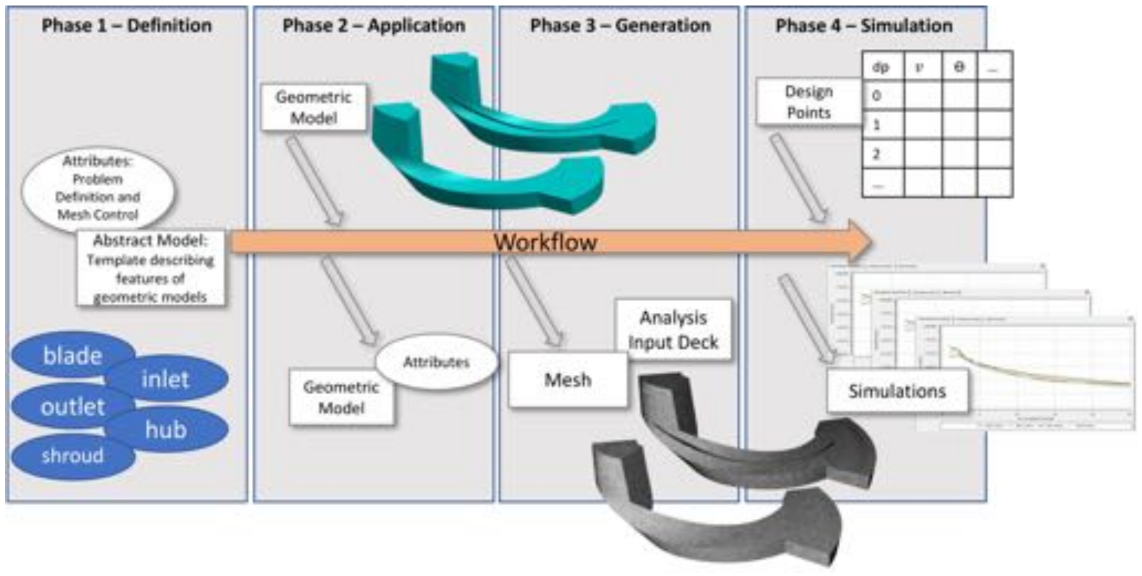
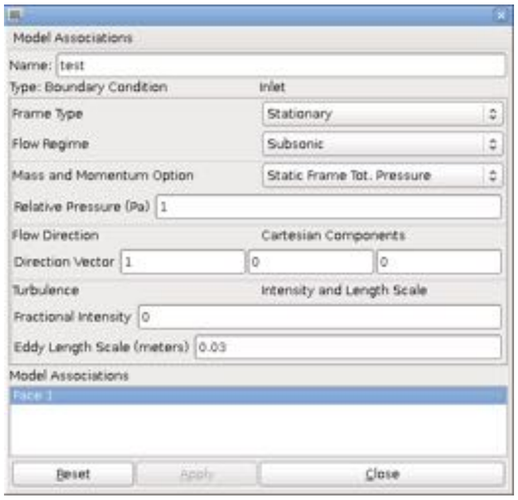


Fig. 5.5. Workflow from abstraction to simulation [122].

associated with the inlet surface, a no-slip condition with the hub, blade and shroud surfaces, and an outflow condition on the outlet surface. Likewise, size gradation, boundary layer, curvature refinement mesh generation controls can be associated with the AbstractModel components.

Custom Attribute Definitions define the sets of information needed to create boundary conditions, initial conditions, material properties, and simulation control parameters to drive ANSYS CFX two-phase simulations. Fig. 5.6 depicts the SimModeler interface for the inlet attribute and Listing 5.1 shows a snippet of the code that defines it. In the graphical interface the geometric model associations are listed at the bottom with required fields listed above. The depicted inlet boundary condition is created by the derived boundary condition type specified on Line 3 of the Attribute Definition code. Its parent type specification on Line 1 defines the possible geometric model entity dimensions (face, edge or vertex `< f e v >`) that the derived inlet type can be associated with. Fields required for specifying the inlet types 'flow direction' are listed on Lines 5 through 14. Using the parent and derived type mechanism, three options are provided to users. The `cartesian` option requires a one dimensional tensor of length three (l.7, `tensor1 3`) while the `cylindrical` flow direction specification requires one double (l.11-13, `double`) for each of the axial, radial and theta coordinate components. Using this syntax and


Model Associations

Name: test

Type: Boundary Condition — Inlet

Frame Type: Stationary

Flow Regime: Subsonic

Mass and Momentum Option: Static Frame Tot. Pressure

Relative Pressure (Pa): 1

Flow Direction: Cartesian Components

Direction Vector: 1 | 0 | 0

Turbulence: Intensity and Length Scale

Fractional Intensity: 0

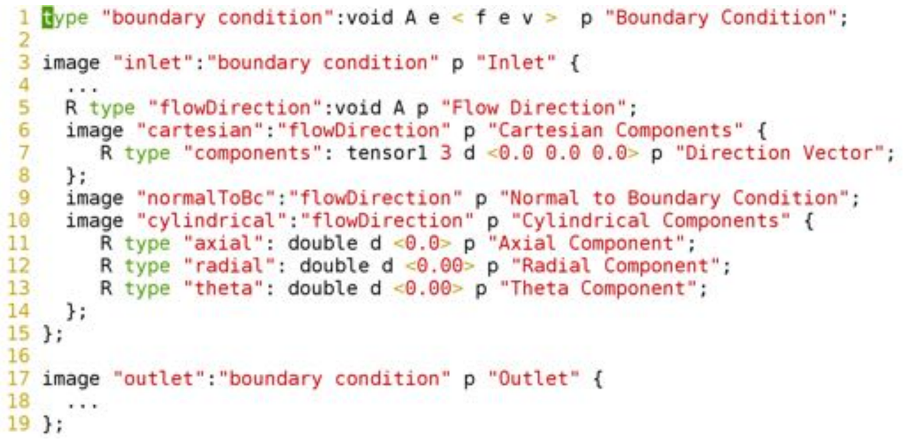
Eddy Length Scale (meters): 0.03

Model Associations

Face 1

Fig. 5.6. Custom attribute definition interface for the inlet boundary condition.

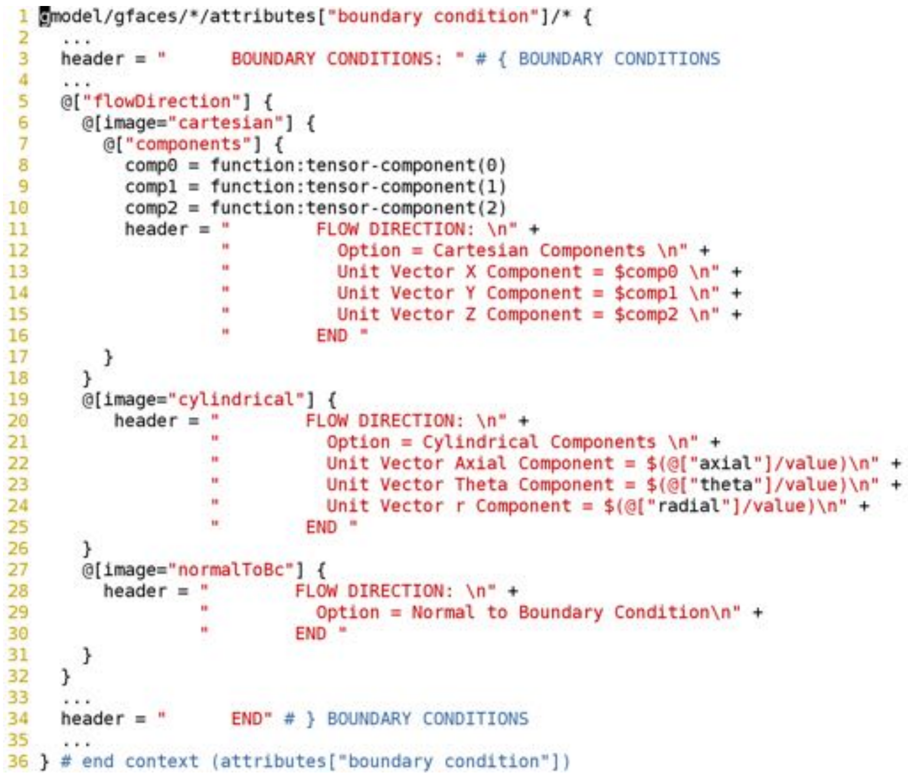
Listing 5.1. A snippet of the Attribute Definition code defining the inlet attribute shown in Fig. 5.6.


```
type "boundary condition":void A e < f e v >  p "Boundary Condition";

image "inlet":"boundary condition" p "Inlet" {
  ...
  R type "flowDirection":void A p "Flow Direction";
  image "cartesian":"flowDirection" p "Cartesian Components" {
     R type "components": tensor1 3 d <0.0 0.0 0.0> p "Direction Vector";
  };
  image "normalToBc":"flowDirection" p "Normal to Boundary Condition";
  image "cylindrical":"flowDirection" p "Cylindrical Components" {
     R type "axial": double d <0.0> p "Axial Component";
     R type "radial": double d <0.00> p "Radial Component";
     R type "theta": double d <0.00> p "Theta Component";
  };
};

image "outlet":"boundary condition" p "Outlet" {
  ...
};
```

similar constructs the full set of problem definition attributes are defined. The attributes and their organization through the type mechanism are defined following a layout that mirrors the CFX interface; the challenges of teaching engineers a new set of tools and processes is difficult enough without complicating it with sweeping interface changes.

Phase 2 of Fig. 5.5 combines tagged geometric model instances (i.e., identification strings associated with geometric model entities) with the templates defined in Phase 1 to create fully attributed geometric models. Each model is then meshed

Listing 5.2. A portion of the Export Pattern code for writing the inlet boundary condition flow direction information to the ANSYS CFX analysis input file.



```
 1 @model/gfaces/*/attributes["boundary condition"]/* {
 2   ...
 3   header = "      BOUNDARY CONDITIONS: " # { BOUNDARY CONDITIONS
 4   ...
 5   @["flowDirection"] {
 6     @[image="cartesian"] {
 7       @["components"] {
 8         comp0 = function:tensor-component(0)
 9         comp1 = function:tensor-component(1)
10         comp2 = function:tensor-component(2)
11         header = "        FLOW DIRECTION: \n" +
12                  "          Option = Cartesian Components \n" +
13                  "          Unit Vector X Component = $comp0 \n" +
14                  "          Unit Vector Y Component = $comp1 \n" +
15                  "          Unit Vector Z Component = $comp2 \n" +
16                  "        END "
17       }
18     }
19     @[image="cylindrical"] {
20        header = "        FLOW DIRECTION: \n" +
21                 "          Option = Cylindrical Components \n" +
22                 "          Unit Vector Axial Component = $(@["axial"]/value)\n" +
23                 "          Unit Vector Theta Component = $(@["theta"]/value)\n" +
24                 "          Unit Vector r Component = $(@["radial"]/value)\n" +
25                 "        END "
26     }
27     @[image="normalToBc"] {
28       header = "        FLOW DIRECTION: \n" +
29                "          Option = Normal to Boundary Condition\n" +
30                "        END "
31     }
32   }
33   ...
34   header = "      END" # } BOUNDARY CONDITIONS
35   ...
36 } # end context (attributes["boundary condition"])
```

in Phase 3 and CFX analysis input decks are created. Each input deck is composed of a mesh, in a FLUENT format, and the CFX analysis definition file (.ccl). The analysis file is created using a custom Simmetrix SimModSuite Export Pattern. The portion of the CFX Export Pattern code for writing the inlet boundary condition's flow direction information to the analysis definition input file is shown in Listing 5.2. On Line 1 is a loop over all the geometric model faces with `boundary condition` attributes associated with them. Lines 6, 19 and 27 respectively process the three derived types for the flow direction. Within the blocks for the `components` and `cylindrical` derived types are formatted strings containing the tensor (l.13-15) and double precision (l.22-24) values. These strings are in turn written to the precious `header` variable which is flushed to the CFX analysis definition file.

In Phase 4 of Fig. 5.5 the user loads the CFX analysis input and Fluent mesh into ANSYS Workbench along with a table of design points. Each design point defines values for selected boundary conditions to drive an ensemble of simulations that covers the design space. Each design point simulation is then executed on a cluster using one of the natively supported job schedulers or our custom integration of the ANSYS Remote Solver Manager with the open-source job scheduler SLURM [182].

Hydraulic engineers using this workflow bypass many tedious and error prone steps. This automation increases their time spent on design and analysis, which results in better products. Using an early version of this process helped bring a heavy-duty pump to market which later won industry product awards.

5.4 Summary

In-memory and semi-automated workflows for three industrial applications were demonstrated. For the in-memory PHASTA application the challenge was to work with the limitations of the industrial partners structural mechanics code while providing a workflow that efficiently simulated the complex multi-phase flow. The second and third applications emphasized the definition of workflow automation to limit the time the analyst or engineer spends setting up problems or running simulations.

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

As we move towards the exascale computers being considered [78], [183], it is clear that one of the few effective means to construct parallel adaptive simulations is by using in-memory interfaces that avoid filesystem interactions. Of course, the cost of refactoring existing large-scale parallel partial differential equation solvers to fully interact with the type of structures and methods used by mesh adaptation components is an extremely expensive and time-consuming process. To address these costs, we presented approaches for in-memory integration of existing solver components with mesh adaptation components, discussed how code changes can be minimized, and demonstrated the performance advantage within adaptive simulations. Demonstrations with the massively parallel computational fluid dynamics, solid mechanics, and electromagnetics adaptive workflows showed orders of magnitude performance improvements in I/O procedures using in-memory coupling instead of files at up to 16Ki processes of the ALCF Theta system. In addition to efforts on developing in-memory approaches with the PHASTA, Albany, and Omega3P solvers, efforts are underway to interface other state-of-the-art solvers including NASA's FUN3D [184] and LLNL's MFEM [185].

Parallel scalability of these workflow components is maintained with new methods to dynamically balance the computational domain. These methods work directly on the unstructured mesh alongside traditional graph and geometric methods to quickly reduce the source of imbalance the consuming workflow component is sensitive to. This approach improves the linear algebra work performance of PHASTA computational fluid dynamics by 28% over a graph-based partition, and improves scaling from 0.82 to 1.14, on 512Ki processes of the ALCF Mira system.

As the high performance computing community prepares for near-exascale systems at the national laboratories, and near-petascale systems become the norm for academia and advanced industry users, we continue to advance parallel unstructured mesh-based simulation technologies for efficiently exploiting the massive amount of computing power on hand and on the horizon. The challenge is clear; adapt to hardware changes and application needs by applying new programming and algorithmic approaches while maintaining support for the existing user base. We address this challenge by focusing a significant portion of our efforts on leveraging the wealth of existing components and the thousands of person-hours invested in them. In the era of distributed memory message passing between many-core nodes, this has been a broadly successful approach and is expected to continue well into the next decade.

6.2 Future Work

6.2.1 Partitioning and Load Balancing

The ParMA load balancing algorithms that worked directly on unstructured meshes are being generalized to support applications that use a different mesh distribution (e.g., node partitions), or simply have information and dependencies between them that can be represented with a graph. EnGPar [186] will implement the ParMA diffusive algorithms and multi-level graph partitioning procedures using data-parallel operations on a graph structure using multiple edge-types to represent different application information dependencies. Efforts are underway to implement the diffusive algorithms and explore the use of the Kokkos [140] programming model for performance portability.

6.2.2 In-memory Component Coupling

Parallel system vendors have started to deploy an additional layer in the memory hierarchy between the parallel filesystem and main memory (DRAM) on each node. The additional layer is implemented by Cray DataWarp devices on the Cori XC40 system at the National Energy Research Scientific Computing Center. In the 2018-2019 Aurora system at the Argonne Leadership Computing Facility the layer will be implemented with Intel SSDs. By design, we expect transfer tests operating on the new layer to be slower than data streams and APIs operating out of DRAM or high bandwidth memory. However, depending on the locality of the SSD or DataWarp devices to compute nodes, the performance could vary significantly and should be evaluated.

6.2.3 Unstructured Mesh-Based Workflows

Application support using the PHASTA-PUMI and Albany-PUMI parallel in-memory workflows continues with a fluid structure interaction problem and an additive manufacturing analysis. In both cases the coupling approaches developed are reused while specific aspects of the finite element and mesh adaptivity components are modified. For PHASTA, developments are focused on implementing the discontinuous Galerkin method to account for interactions across the solid-gas interface. In the Albany additive manufacturing workflow the focus is on mesh adaptation methods to support the evolution of the geometry as layers of material are deposited.

Ongoing interactions with industrial users are focused on two applications with evolving geometry. The first application is simulating flow in a gas filled chamber with a moving displacer. Implementation of this workflow requires combining mesh motion to track the moving geometry and mesh adaptation to maintain element quality. The second application uses MeshAdapt to support simulation of an underground reservoir with evolving geometry.

REFERENCES

[1] T. J. Hughes, *The finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Mineola, NY, USA: Courier Dover Publications, 2012.

[2] T. Sonar, "Chapter 3 - classical finite volume methods," in *Handbook of Numerical Methods for Hyperbolic Problems - Basic and Fundamental Issues*. Braunschweig, Germany: Elsevier, 2016, pp. 55–76.

[3] X. Li, M. S. Shephard, and M. W. Beall, "3D anisotropic mesh adaptation by mesh modification," *Comput. Methods in Appl. Mech. and Eng.*, vol. 194, no. 48-49, pp. 4915–4950, Nov. 2005.

[4] F. Alauzet, X. Li, E. S. Seol, and M. S. Shephard, "Parallel anisotropic 3D mesh adaptation by mesh modification," *Eng. with Comput.*, vol. 21, no. 3, pp. 247–258, May 2006.

[5] D. A. Ibanez, "Conformal mesh adaptation on heterogeneous supercomputers," Ph.D. dissertation, Dept. Comput. Sci., Rensselaer Polytechnic Inst., Troy, NY, 2016.

[6] A. Ovcharenko, K. C. Chitale, O. Sahni, K. E. Jansen, and M. S. Shephard, "Parallel adaptive boundary layer meshing for CFD analysis," in *Proc. 21st Int. Meshing Roundtable*, 2012, pp. 437–455.

[7] G. Karypis and V. Kumar, "Parallel multilevel series k-way partitioning scheme for irregular graphs," *Siam Rev.*, vol. 41, no. 2, pp. 278–300, Jun. 1999.

[8] D. F. Harlacher, H. Klimach, S. Roller, C. Siebert, and F. Wolf, "Dynamic load balancing for unstructured meshes on space-filling curves," in *26th Int. Parallel and Distributed Process. Symp. Workshops PhD Forum*, 2012, pp. 1661–1669.

[9] M. Zhou, O. Sahni, K. Devine, M. Shephard, and K. Jansen, "Controlling unstructured mesh partitions for massively parallel simulations," *SIAM J. Scientific Comput.*, vol. 32, no. 6, pp. 3201–3227, Nov. 2010.

[10] M. Zhou, O. Sahni, T. Xie, M. S. Shephard, and K. E. Jansen, "Unstructured mesh partition improvement for implicit finite element at extreme scale," *The J. Supercomputing*, vol. 59, no. 3, pp. 1218–1228, Dec. 2012.

[11] D. A. Ibanez, E. S. Seol, C. W. Smith, and M. S. Shephard, "PUMI: Parallel unstructured mesh infrastructure," *ACM Trans. Math. Softw.*, vol. 42, no. 3, pp. 17:1–17:28, May 2016.

[12] D. Ibanez, I. Dunn, and M. S. Shephard, “Hybrid MPI-thread parallelization of adaptive mesh operations,” *Parallel Comput.*, vol. 52, no. 1, pp. 133–143, Jan. 2016.

[13] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, “Zoltan data management services for parallel dynamic applications,” *Comput. in Sci. Eng.*, vol. 4, no. 2, pp. 90–96, Mar. 2002.

[14] M. S. Shephard, C. Smith, and J. E. Kolb, “Bringing hpc to engineering innovation,” *Comput. in Sci. & Eng.*, vol. 15, no. 1, pp. 16–25, Feb. 2013.

[15] R. O’Bara, M. Beall, and M. Shephard, “Attribute management system for engineering analysis,” *Eng. with Comput.*, vol. 18, no. 4, pp. 339–351, Nov. 2002.

[16] M. S. Shephard, M. Beall, R. O’Bara, and B. Webster, “Toward simulation-based design,” *Finite Elements in Anal. and Design*, vol. 40, no. 12, pp. 1575–1598, Jul. 2004.

[17] S. Tendulkar, M. Beall, M. S. Shephard, and K. Jansen, “Parallel mesh generation and adaptation for CAD geometries,” in *Proc. NAFEMS World Congr.*, 2011, pp. 1–12.

[18] K. C. Chitale, M. Rasquin, O. Sahni, M. S. Shephard, and K. E. Jansen, “Anisotropic boundary layer adaptivity of multi-element wings,” in *52nd Aerospace Sciences Meeting (SciTech). AIAA Paper*, 2014, vol. 117, pp. 1–14.

[19] O. Sahni, J. Müller, K. E. Jansen, M. S. Shephard, and C. A. Taylor, “Efficient anisotropic adaptive discretization of cardiovascular system,” *Comput. Methods in Appl. Mech. and Eng.*, vol. 195, no. 41-43, pp. 5634–5655, Aug. 2006.

[20] E. Ramm, E. Rank, R. Rannacher, K. Schweizerhof, E. Stein, W. Wendland *et al.*, *Error-controlled Adaptive Finite Elements in Solid Mechanics.* West Sussex, England: John Wiley & Sons, 2003.

[21] E. S. Seol, C. W. Smith, D. A. Ibanez, and M. S. Shephard, “A parallel unstructured mesh infrastructure,” in *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*, 2012, pp. 1124–1132.

[22] B. Hendrickson and K. Devine, “Dynamic load balancing in computational mechanics,” *Comput. Methods in Appl. Mech. and Eng.*, vol. 184, no. 2, pp. 485–500, Apr. 2000.

[23] K. Schloegel, G. Karypis, and V. Kumar, “Graph partitioning for high-performance scientific simulations,” in *Sourcebook of Parallel Computing.* San Francisco, CA, USA: Morgan Kaufmann Inc., 2003, pp. 491–541.

[24] C. Aykanat, B. B. Cambazoglu, and B. Uçar, “Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices,” *J. Parallel and Distributed Comput.*, vol. 68, no. 5, pp. 609–625, May 2008.

[25] G. Karypis and V. Kumar, “Multilevel algorithms for multi-constraint graph partitioning,” in *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*, 1998, pp. 1–13.

[26] K. Schloegel, G. Karypis, and V. Kumar, “Parallel static and dynamic multi-constraint graph partitioning,” *Concurrency and Computation: Practice and Experience*, vol. 14, no. 3, pp. 219–240, Mar. 2002.

[27] U. Çatalyürek, M. Deveci, K. Kaya, and B. Uçar, “UMPa: A multiobjective, multi-level partitioner for communication minimization,” *Contemporary Math.*, vol. 588, no. 1, pp. 53–64, Feb. 2013.

[28] D. LaSalle and G. Karypis, “Multi-threaded graph partitioning,” in *Parallel & Distributed Process. (IPDPS), IEEE 27th Int. Symp.*, 2013, pp. 225–236.

[29] U. V. Çatalyürek and C. Aykanat, “Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication,” *Parallel and Distributed Syst., IEEE Trans.*, vol. 10, no. 7, pp. 673–693, Jul. 1999.

[30] U. V. Çatalyürek, E. G. Boman, K. D. Devine, D. Bozdağ, R. T. Heaphy, and L. A. Riesen, “A repartitioning hypergraph model for dynamic load balancing,” *J. Parallel and Distributed Comput.*, vol. 69, no. 8, pp. 711–724, Aug. 2009.

[31] S. Schamberger and J.-M. Wierum, “Partitioning finite element meshes using space-filling curves,” *Future Generation Comput. Syst.*, vol. 21, no. 5, pp. 759–766, May 2005.

[32] E. G. Boman, K. D. Devine, V. J. Leung, S. Rajamanickam, L. A. Riesen, D. Mehmet *et al.*, “Zoltan2: Next-generation combinatorial toolkit,” Sandia Nat. Labs, Albuquerque, NM, USA, Tech. Rep. SAND2012-9373C, 2012.

[33] M. Deveci, S. Rajamanickam, K. Devine, and U. Çatalyürek, “Multi-jagged: A scalable parallel spatial partitioning algorithm,” *Parallel and Distributed Syst., IEEE Trans.*, vol. 27, no. 3, pp. 803–817, Mar. 2015.

[34] E. Saule, E. Ö. Baş, and Ü. V. Çatalyürek, “Load-balancing spatially located computations using rectangular partitions,” *J. Parallel and Distributed Comput.*, vol. 72, no. 10, pp. 1201–1214, Oct. 2012.

[35] V. Taylor and B. Nour-omid, “A study of the factorization fill-in for a parallel implementation of the finite element method,” *Int. J. Numer. Meth. Engng*, vol. 37, no. 22, pp. 3809–3823, Nov. 1994.

[36] R. D. Williams, “Performance of dynamic load balancing algorithms for unstructured mesh calculations,” *Concurrency: Pract. Exper.*, vol. 3, no. 5, pp. 457–481, Oct. 1991.

[37] M. J. Berger and S. H. Bokhari, “A partitioning strategy for nonuniform problems on multiprocessors,” *Comput., IEEE Trans.*, vol. 100, no. 5, pp. 570–580, May 1987.

[38] H. Simon, “Partitioning of unstructured problems for parallel processing,” *Comput. Syst. in Eng.*, vol. 2, no. 2, pp. 135–148, Feb. 1991.

[39] J. Skilling, “Programming the hilbert curve,” in *23rd Int. Workshop on Bayesian Inference and Maximum Entropy Methods in Sci. and Eng.*, 2004, vol. 707, no. 1, pp. 381–387.

[40] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik *et al.*, “New challenges in dynamic load balancing,” *Appl. Numerical Math.*, vol. 52, no. 2, pp. 133–152, Feb. 2005.

[41] N. E. Gibbs, W. G. Poole, and P. K. Stockmeyer, “An algorithm for reducing the bandwidth and profile of a sparse matrix,” *SIAM J. Numerical Anal.*, vol. 13, no. 2, pp. 236–250, Apr. 1976.

[42] M. Zhou, O. Sahni, M. S. Shephard, C. D. Carothers, and K. E. Jansen, “Adjacency-based data reordering algorithm for acceleration of finite element computations,” *Scientific Programming*, vol. 18, no. 2, pp. 107–123, May 2010.

[43] Y. Hu and R. Blake, “An improved diffusion algorithm for dynamic load balancing,” *Parallel Comput.*, vol. 25, no. 4, pp. 417–444, Apr. 1999.

[44] Y. F. Hu, R. J. Blake, and D. R. Emerson, “An optimal migration algorithm for dynamic load balancing,” *Concurrency: Practice and Experience*, vol. 10, no. 6, pp. 467–483, May 1998.

[45] H. Meyerhenke, B. Monien, and S. Schamberger, “Graph partitioning and disturbed diffusion,” *Parallel Comput.*, vol. 35, no. 10, pp. 544–569, Oct. 2009.

[46] C.-W. Ou and S. Ranka, “Parallel incremental graph partitioning,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 8, pp. 884–896, Aug. 1997.

[47] K. Schloegel, G. Karypis, and V. Kumar, “Multilevel diffusion schemes for repartitioning of adaptive meshes,” *J. Parallel and Distributed Comput.*, vol. 47, no. 2, pp. 109–124, Dec. 1997.

[48] ——, “Wavefront diffusion and LMSR: Algorithms for dynamic repartitioning of adaptive meshes,” *Parallel and Distributed Syst., IEEE Trans.*, vol. 12, no. 5, pp. 451–466, May 2001.

[49] C. Walshaw, M. Cross, and M. Everett, "Dynamic mesh partitioning: A unified optimisation and load-balancing algorithm," Univ. of Greenwich, London, UK, Tech. Rep. 95/IM/06, 1995.

[50] R. Subramanian and I. D. Scherson, "An analysis of diffusive load-balancing," in *Proc. sixth Annu. ACM Symp. Parallel algorithms and architectures*, 1994, pp. 220–225.

[51] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *J. Parallel and Distributed Comput.*, vol. 7, no. 2, pp. 279–301, Oct. 1989.

[52] M. H. Willebeek-LeMair and A. P. Reeves, "Strategies for dynamic load balancing on highly parallel computers," *Parallel and Distributed Syst., IEEE Trans.*, vol. 4, no. 9, pp. 979–993, Sep. 1993.

[53] C. Fiduccia and R. Mattheyses, "A linear-time heuristic for improving network partitions," in *19th Design Automation Conf.*, 1982, pp. 175–181.

[54] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell System Tech. J.*, vol. 49, no. 2, pp. 291–307, Apr. 1970.

[55] G. M. Slota, K. Madduri, and S. Rajamanickam, "Complex network partitioning using label propagation," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S620–S645, 2016.

[56] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri, "Partitioning trillion-edge graphs in minutes," in *Int. Parallel & Distributed Process. Symp. (IPDPS)*, 2017.

[57] R. H. Bisseling and W. Meesen, "Communication balancing in parallel sparse matrix-vector multiplication." *ETNA. Electronic Transactions on Numerical Analysis*, vol. 21, pp. 47–65, 2005.

[58] E. S. Seol, "FMDB: flexible distributed mesh database for parallel automated adaptive analysis," Ph.D. dissertation, Dept. Comput. Sci., Rensselaer Polytechnic Inst., Troy, NY, 2005.

[59] A. Ovcharenko, D. Ibanez, F. Delalondre, O. Sahni, K. E. Jansen, C. D. Carothers *et al.*, "Neighborhood communication paradigm to increase scalability in large-scale dynamic scientific applications," *Parallel Comput.*, vol. 38, no. 3, pp. 140–156, Mar. 2012.

[60] 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," *Comput. Methods in Appl. Mech. and Eng.*, vol. 190, no. 3-4, pp. 305–319, Oct. 2000.

[61] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw, “Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM,” *Parallel Comput.*, vol. 26, no. 12, pp. 1555–1581, Nov. 2000.

[62] H. Meyerhenke and S. Schamberger, “Balancing parallel adaptive FEM computations by solving systems of linear equations,” in *Euro-Par Parallel Process.*, 2005, pp. 209–219.

[63] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.* Cambridge, MA, USA: MIT Press, 2009, ch. 22.2.

[64] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction To Algorithms.* Cambridge, MA, USA: MIT Press, 2001.

[65] D. Jacobsen, M. Petersen, T. Ringler, and M. Duda. (2014). *MPAS-Ocean Model User’s Guide. Version 2.0.* [Online]. Available: http://oceans11.lanl.gov/mpas_data/mpas_ocean/users_guide/release_2.0/mpas_ocean_users_guide_2.0.pdf, Accessed on: Mar. 17, 2017.

[66] K. Weiler, “The radial edge structure: a topological representation for non-manifold geometric boundary modeling,” in *Geometric Modeling for CAD Appl. First IFIP WG5.2 Work. Conf.*, 1988, pp. 3–36.

[67] M. W. Beall and M. S. Shephard, “A general topology-based mesh data structure,” *Int. J. Numerical Methods in Eng.*, vol. 40, no. 9, pp. 1573–1596, May 1997.

[68] B. Fornberg, “Generation of finite difference formulas on arbitrarily spaced grids,” *Math. of Computation*, vol. 51, no. 184, pp. 699–706, Oct. 1988.

[69] P. Berenbrink, T. Friedetzky, and Z. Hu, “A new analytical method for parallel, diffusion-type load balancing,” *J. Parallel and Distributed Comput.*, vol. 69, no. 1, pp. 54–61, Jan. 2009.

[70] P. H. Worley, E. D’Azevedo, R. Hager, S.-H. Ku, E. Yoon, and C. Chang, “Balancing particle and mesh computation in a particle-in-cell code,” in *Proc. Cray Users Group Meeting*, 2016, pp. 1–10.

[71] J. Fingberg, A. Basermann, G. Lonsdale, J. Clinckemaillie, J.-M. Gratien, and R. Ducloux, “Dynamic load balancing for parallel structural mechanics simulations with DRAMA,” in *Developments in Engineering Computational Technology.* Edinburgh, UK: Civil-Comp Press, 2000, pp. 199–205.

[72] B. FrantzDale, S. J. Plimpton, and M. S. Shephard, “Software components for parallel multiscale simulation: an example with LAMMPS,” *Eng. with Comput.*, vol. 26, no. 2, pp. 205–211, Dec. 2010.

[73] C. Chevalier, G. Grospellier, F. Ledoux, J. Weill, and F. Arpajon, “Load balancing for mesh based multi-physics simulations in the Arcane framework,” in *Proc. 8th Int. Conf. Eng. Computational Technol.*, 2012, pp. 47–62.

[74] C. H. Whiting and K. E. Jansen, “A stabilized finite element method for the incompressible navier-stokes equations using a hierarchical basis,” *Int. J. Numerical Methods in Fluids*, vol. 35, no. 1, pp. 93–116, Jan. 2001.

[75] 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,” in *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*, 2009, pp. 1–12.

[76] M. Rasquin, C. Smith, K. Chitale, E. S. Seol, B. A. Matthews, J. L. Martin *et al.*, “Scalable implicit flow solver for realistic wing simulations with flow control,” *Comput. in Sci. & Eng.*, vol. 16, no. 6, pp. 13–21, Dec. 2014.

[77] C. W. Smith, S. Tran, O. Sahni, F. Behafarid, M. S. Shephard, and R. Singh, “Enabling HPC simulation workflows for complex industrial flow problems,” in *Proc. XSEDE Conf.: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, 2015, pp. 41:1–41:7.

[78] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre *et al.*, “The international exascale software project roadmap,” *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 1, pp. 3–60, Feb. 2011.

[79] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda *et al.*, “An overview of the Trilinos project,” *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, Sep. 2005.

[80] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman *et al. PETSc.* (2016) [Online]. Available: http://www.mcs.anl.gov/petsc, Accessed on: Mar. 17, 2017.

[81] F. Delalondre, C. W. Smith, and M. S. Shephard, “Collaborative software infrastructure for adaptive multiple model simulation,” *Comput. Methods in Appl. Mech. and Eng.*, vol. 199, no. 21-22, pp. 1352–1370, Apr. 2010.

[82] M. W. Beall, J. Walsh, and M. S. Shephard, “A comparison of techniques for geometry access related to mesh generation,” *Eng. with Comput.*, vol. 20, no. 3, pp. 210–221, Aug. 2004.

[83] T. J. Tautges, “CGM: A geometry interface for mesh generation, analysis and other applications,” *Eng. with Comput.*, vol. 17, no. 3, pp. 299–314, Oct. 2001.

[84] R. Haimes and J. Dannenhoffer, “The Engineering Sketch Pad: A solid-modeling, feature-based, web-enabled system for building parametric geometry,” in *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Expo*, 2013, pp. 1–21.

[85] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. New York, NY, USA: ACM Press and Addison-Wesley, 2002.

[86] M. Miller, J. Reus, R. Matzke, Q. Koziol, and A. Cheng, “Smart libraries: Best SQE practices for libraries with emphasis on scientific computing,” in *Proc. Nucl. Explosives Code Developer’s Conf.*, 2004, pp. 1–30.

[87] J. Brown, M. G. Knepley, and B. F. Smith, “Run-time extensibility and librarization of simulation software,” *Comput. in Sci. & Eng.*, vol. 17, no. 1, pp. 38–45, Feb. 2015.

[88] W. Gropp, “Exploiting existing software in libraries: successes, failures, and reasons why.” in *Object Oriented Methods for Interoperable Scientific and Eng. Comput.: Proc. 1998 SIAM Workshop*, pp. 21–29.

[89] B. Smith and R. Bartlett, “xSDK community package policies,” U.S. Dept. Energy, Office of Sci., Chicago, IL, USA, Tech. Rep. V0.3, 2016.

[90] J. C. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, T. Jin *et al.*, “Combining in-situ and in-transit processing to enable extreme-scale scientific analysis,” in *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*, 2012, pp. 1–9.

[91] F. Zhang, C. Docan, M. Parashar, S. Klasky, N. Podhorszki, and H. Abbasi, “Enabling in-situ execution of coupled scientific workflow on multi-core platform,” in *Proc. 26th Int. Parallel & Distributed Process. Symp.*, 2012, pp. 1352–1363.

[92] M. Rasquin, P. Marion, V. Vishwanath, B. Matthews, M. Hereld, K. Jansen *et al.*, “Electronic poster: Co-visualization of full data and in situ data extracts from unstructured grid CFD at 160k cores,” in *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*, 2011, pp. 103–104.

[93] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka, “Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems,” in *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*, 2011, pp. 19:1–19:11.

[94] C. Burstedde, L. C. Wilcox, and O. Ghattas, “`p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees,” *SIAM J. Scientific Comput.*, vol. 33, no. 3, pp. 1103–1133, May 2011.

[95] V. Sankaran, J. Sitaraman, A. Wissink, A. Datta, B. Jayaraman, M. Potsdam *et al.*, "Application of the Helios computational platform to rotorcraft flowfields," in *48th AIAA Aerospace Sciences Meeting*, 2010, vol. 1230, pp. 1–28.

[96] *International Standard - Information technology Portable Operating System Interface (POSIX)Base Specifications*, ISO/IEC/IEEE-9945:2009(E), 2009.

[97] W. van Oortmerssen. *FlatBuffers*. (2016) [Online]. Available: http://google.github.io/flatbuffers/index.html, Accessed on: Mar. 17, 2017.

[98] K. Varda and D. Renshaw. *Cap'n Proto*. (2016) [Online]. Available: https://capnproto.org/, Accessed on: Mar. 17, 2017.

[99] C. Ollivier-Gooch, L. Diachin, M. S. Shephard, T. Tautges, J. Kraftcheck, V. Leung *et al.*, "An interoperable, data-structure-neutral component for mesh query and manipulation," *Trans. Math. Software*, vol. 37, no. 3, pp. 29:1–29:28, Sep. 2010.

[100] M. A. Park and D. L. Darmofal, "Parallel anisotropic tetrahedral adaptation," in *46th AIAA Aerospace Sciences Meeting and Exhibit*, 2008, pp. 1–19.

[101] A. Loseille, V. Menier, and F. Alauzet, "Parallel generation of large-size adapted meshes," in *Proc. 24th Int. Meshing Roundtable*, 2014, pp. 57–69.

[102] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey, "libMesh : a C++ library for parallel adaptive mesh refinement/coarsening simulations," *Eng. with Comput.*, vol. 22, no. 3, pp. 237–254, Dec. 2006.

[103] C. W. Smith, M. Rasquin, D. Ibanez, K. E. Jansen, and M. S. Shephard, "Improving unstructured mesh partitions for multiple criteria using mesh adjacencies," *SIAM J. Scientific Comput.*, submitted for publication.

[104] P. Farrell and J. Maddison, "Conservative interpolation between volume meshes by local galerkin projection," *Comput. Methods in Appl. Mech. and Eng.*, vol. 200, no. 1-4, pp. 89–100, Jan. 2011.

[105] SCOREC. *PUMI GitHub repository*. (2014) [Online]. Available: https://github.com/SCOREC/core, Accessed on: Oct. 26, 2016.

[106] D. Ibanez and M. S. Shephard, "Modifiable array data structures for mesh topology," *SIAM J. Scientific Comput.*, vol. 39, no. 2, pp. C144–C161, 2017.

[107] O. C. Zienkiewicz and J. Z. Zhu, "The superconvergent patch recovery and a posteriori error estimates. part 1: The recovery technique," *Int. J. Numerical Methods in Eng.*, vol. 33, no. 7, pp. 1331–1364, May 1992.

[108] P. Frey and F. Alauzet, "Anisotropic mesh adaptation for CFD computations," *Comput. Methods in Appl. Mech. and Eng.*, vol. 194, no. 48-49, pp. 5068–5082, Nov. 2005.

[109] A. Langer and K. Kreft, *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference.* Reading, MA, USA: Addison-Wesley Professional, 2000.

[110] R. V. Garimella, "Mesh data structure selection for mesh generation and FEA applications," *Int. J. Numerical Methods in Eng.*, vol. 55, no. 4, pp. 451–478, Jul. 2002.

[111] R. Haimes and C. Crawford, "Unified geometry access for analysis and design," in *Proc. 16th Int. Meshing Roundtable*, 2003, pp. 21–31.

[112] S. Dey, M. S. Shephard, and J. E. Flaherty, "Geometry representation issues associated with p-version finite element computations," *Comput. Methods in Appl. Mech. and Eng.*, vol. 150, no. 1, pp. 39–55, Dec. 1997.

[113] M. W. Beall, "An object-oriented framework for the reliable automated solution of problems in mathematical physics," Ph.D. dissertation, Dept. Aeronautical Eng., Rensselaer Polytechnic Inst., Troy, NY, 1999.

[114] O. Klaas, M. Beall, and M. S. Shephard, "Generation of geometric models and meshes from segmented image data," in *Proc. NAFEMS Americas Conf.*, 2014, pp. 1–4.

[115] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum *et al.*, "Reconstruction and representation of 3D objects with radial basis functions," in *Proc. 28th Annu. Conf. Comput. Graph. and Interactive Techn.*, 2001, pp. 67–76.

[116] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE Int. Conf. Robotics and Automation*, 2011, pp. 1–4.

[117] B. Dogdas, D. Stout, A. F. Chatziioannou, and R. M. Leahy, "Digimouse: a 3D whole body mouse atlas from CT and cryosection data," *Physics in Medicine and Bio.*, vol. 52, no. 3, pp. 577–587, Feb. 2007.

[118] A. Edmans, X. Intes, and C. W. Smith, "Mesh optimization for monte carlo based optical tomography," in *40th Annu. Northeast Bioengineering Conf. (NEBEC)*, 2014, pp. 1–2.

[119] A. Edmans and X. Intes, "Mesh optimization for monte carlo-based optical tomography," *Photonics*, vol. 2, no. 2, pp. 375–391, Apr. 2015.

[120] A. Heinecke, A. Breuer, S. Rettenberger, M. Bader, A.-A. Gabriel, C. Pelties *et al.*, "Petascale high order dynamic rupture earthquake simulations on heterogeneous supercomputers," in *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*, 2014, pp. 3–14.

[121] S. Rettenberger, C. W. Smith, and C. Pelties, "Poster: Optimizing CAD and mesh generation workflow for SeisSol," presented at the Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC), New Orleans, LA, USA, Nov. 16-21, 2014.

[122] *Simulation Modeling Suite 11.0 Documentation*, Simmetrix Inc., Clifton Park, NY, 2017.

[123] Open CASCADE SAS. *Open CASCADE Modeling Kernel.* (1999) [Online]. Available: https://www.opencascade.com/content/latest-release, Accessed on: Mar. 19, 2017.

[124] V. Mahadevan. *SIGMA: Scalable Interfaces for Geometry and Mesh Based Applications.* (2014) [Online]. Available: http://sigma.mcs.anl.gov/, Accessed on: Mar. 19, 2017.

[125] ——. *The Common Geometry Module (CGM).* (2012) [Online]. Available: https://bitbucket.org/fathomteam/cgm, Accessed on: Mar. 19, 2017.

[126] R. Haimes and M. Drela, "On the construction of aircraft conceptual geometry for high-fidelity analysis and design," in *50th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Expo*, 2012, pp. 1–21.

[127] M. S. Shephard and M. K. Georges, "Reliability of automatic 3d mesh generation," *Comput. Methods in Appl. Mech. and Eng.*, vol. 101, no. 1, pp. 443–462, Dec. 1992.

[128] Pointwise. *Mesh Generation Software for CFD.* (1995) [Online]. Available: http://www.pointwise.com/, Accessed on: Mar. 19, 2017.

[129] J. Steinbrenner and J. Abelanet, "Anisotropic tetrahedral meshing based on surface deformation techniques," in *45th AIAA Aerospace Sciences Meeting and Exhibit*, 2007, pp. 6691–6706.

[130] C. Geuzaine and J.-F. Remacle, "Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities," *Int. J. Numerical Methods in Eng.*, vol. 79, no. 11, pp. 1309–1331, May 2009.

[131] J. Schöberl, "NETGEN an advancing front 2D/3D-mesh generator based on abstract rules," *Comput. and Visualization in Sci.*, vol. 1, no. 1, pp. 41–52, Feb. 1997.

[132] S. Gosselin and C. Ollivier-Gooch, “Tetrahedral mesh generation using delaunay refinement with non-standard quality measures,” *Int. J. Numerical Methods in Eng.*, vol. 87, no. 8, pp. 795–820, Feb. 2011.

[133] L. A. Freitag and C. Ollivier-Gooch, “Tetrahedral mesh improvement using swapping and smoothing,” *Int. J. Numerical Methods in Eng.*, vol. 40, no. 21, pp. 3979–4002, Nov. 1997.

[134] G. Compere, J.-F. Remacle, J. Jansson, and J. Hoffman, “A mesh adaptation framework for dealing with large deforming meshes,” *Int. J. Numerical Methods in Eng.*, vol. 82, no. 7, pp. 843–867, May 2010.

[135] D. Ibanez. *Omega_h GitHub repository.* (2016) [Online]. Available: https://github.com/ibaned/omega_h, Accessed on: Oct. 26, 2016.

[136] D. Ibanez and M. Shephard, “Mesh adaptation for moving objects on shared memory hardware,” in *Proc. 25th Int. Meshing Roundtable*, 2016, pp. 1–5.

[137] M. Mubarak, S. Seol, Q. Lu, and M. S. Shephard, “A parallel ghosting algorithm for the Flexible Distributed Mesh Database.” *Scientific Programming*, vol. 21, no. 1, pp. 17–42, 2013.

[138] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with CUDA,” *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008.

[139] K. Rupp, “Strided memory access on CPUs, GPUs, and MIC,” Feb. 14, 2016. [Online]. Available: https://www.karlrupp.net/2016/02/strided-memory-access-on-cpus-gpus-and-mic/, Accessed on: Mar. 24, 2017.

[140] H. C. Edwards and C. R. Trott, “Kokkos: Enabling performance portability across manycore architectures,” in *2013 Extreme Scaling Workshop*, 2013, pp. 18–24.

[141] T. J. Tautges, R. Meyers, K. Merkley, C. Stimpson, and C. Ernst, “MOAB: A mesh-oriented database,” Sandia Nat. Labs, Albuquerque, NM, USA, Tech. Rep. SAND2004-1592, 2004.

[142] C. W. Smith, B. Granzow, G. Diamond, D. A. Ibanez, O. Sahni, K. E. Jansen *et al.*, “In-memory integration of existing software components for parallel adaptive unstructured mesh workflows,” submitted for publication.

[143] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam *et al.*, “The IBM Blue Gene/Q compute chip,” *IEEE Micro*, vol. 32, no. 2, pp. 48–60, Mar. 2012.

[144] Y. J. Lo, S. Williams, B. Van Straalen, T. J. Ligocki, M. J. Cordery, N. J. Wright *et al.*, “Roofline model toolkit: A practical tool for architectural and

program analysis," in *5th Int. Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Comput. Syst.*, 2014, pp. 129–148.

[145] M. Besta and T. Hoefler, "Fault tolerance for remote memory access programming models," in *Proc. 23rd Int. Symp. High-performance Parallel and Distributed Comput.*, 2014, pp. 37–48.

[146] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. R. de Supinski, and R. Eigenmann, "MCREngine: A scalable checkpointing system using data-aware aggregation and compression," in *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*, 2012, pp. 1–11.

[147] H. Yu, R. K. Sahoo, C. Howson, G. Almasi, J. G. Castanos, M. Gupta *et al.*, "High performance file I/O for the Blue Gene/L supercomputer," in *The 12th Int. Symp. High-Performance Comput. Architecture*, 2006, pp. 187–196.

[148] N. R. Adiga, G. Almasi, G. S. Almasi, Y. Aridor, R. Barik, D. Beece *et al.*, "An overview of the BlueGene/L supercomputer," in *Proc. 2002 ACM/IEEE Conf. Supercomputing*, 2002, pp. 60–60.

[149] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O performance challenges at leadership scale," in *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*, 2009, pp. 40:1–40:12.

[150] S. Alam, R. Barrett, M. Bast, M. R. Fahey, J. Kuehn, C. McCurdy *et al.*, "Early evaluation of IBM BlueGene/P," in *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*, 2008, pp. 23:1–23:12.

[151] H. Bui, H. Finkel, V. Vishwanath, S. Habib, K. Heitmann, J. Leigh *et al.*, "Scalable parallel I/O on a Blue Gene/Q supercomputer using compression, topology-aware data aggregation, and subfiling," in *Parallel, Distributed and Network-Based Process. (PDP), 22nd Euromicro Int. Conf.*, 2014, pp. 107–111.

[152] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani *et al.*, "Knights Landing: Second-generation Intel Xeon Phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, Mar. 2016.

[153] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming*, Knights Landing ed. Boston, MA, USA: Morgan Kaufmann Inc., 2016.

[154] Intel Inc., Santa Clara, CA, USA, "Aurora fact sheet," Available: http://www.intel.com/newsroom/assets/Intel_Aurora_factsheet.pdf, Accessed on: Oct. 28, 2016.

[155] A. Y. Galimov, O. Sahni, R. T. L. Jr., M. S. Shephard, D. A. Drew, and K. E. Jansen, "Parallel adaptive simulation of a plunging liquid jet," *Acta Mathematica Scientia*, vol. 30, no. 2, pp. 522–538, Mar. 2010.

[156] J. M. Rodriguez, O. Sahni, R. T. L. Jr., and K. E. Jansen, "A parallel adaptive mesh method for the numerical simulation of multiphase flows," *Comput. & Fluids*, vol. 87, pp. 115–131, Oct. 2013.

[157] M. Zhou, O. Sahni, H. J. Kim, C. A. Figueroa, C. A. Taylor, M. S. Shephard *et al.*, "Cardiovascular flow simulation at extreme scale," *Computational Mech.*, vol. 46, no. 1, pp. 71–82, Dec. 2010.

[158] C. W. Smith and K. E. Jansen. *PHASTA GitHub repository.* (2015) [Online]. Available: https://github.com/PHASTA/phasta, Accessed on: Mar. 18, 2017.

[159] A. Alexandrescu, *Modern C++ Design : Generic Programming and Design Patterns Applied.* Boston, MA, USA: Addison-Wesley, 2001.

[160] C. W. Smith. *PHASTA-chef GitHub repository.* (2016) [Online]. Available: https://github.com/PHASTA/phastaChef, Accessed on: Mar. 18, 2017.

[161] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *FAST '02: Proc. 1st USENIX Conf. File and Storage Technologies*, 2002, pp. 1–15.

[162] F. Wang, S. Oral, G. Shipman, O. Drokin, T. Wang, and I. Huang, "Understanding Lustre internals," Oak Ridge Nat. Lab (ORNL); Center for Computational Sciences, Oak Ridge, TN, USA, Tech. Rep. 951297, 2009.

[163] S. Bhattacharya, S. Pratt, B. Pulavarty, and J. Morgan, "Asynchronous I/O support in Linux 2.5," in *Proc. Linux Symp.*, 2003, pp. 371–386.

[164] A. G. Salinger, R. A. Bartett, Q. Chen, X. Gao, G. Hansen, I. Kalashnikova *et al.*, "Albany: A component-based partial differential equation code built on Trilinos," Sandia Nat. Labs, Albuquerque, NM, USA, Tech. Rep. SAND2013-8430J, 2013.

[165] A. G. Salinger, R. A. Bartlett, A. M. Bradley, Q. Chen, I. P. Demeshko, X. Gao *et al.*, "Albany: Using component-based design to develop a flexible, generic multiphysics analysis code," *Int. J. Multiscale Computational Eng.*, vol. 14, no. 4, pp. 415–438, 2016.

[166] Z. Li, M. O. Bloomfield, and A. A. Oberai, "Simulation of finite-strain inelastic phenomena governed by creep and plasticity," *Computational Mech.*, submitted for publication.

[167] R. Radovitzky and M. Ortiz, “Error estimation and adaptive meshing in strongly nonlinear dynamic problems,” *Comput. Methods in Appl. Mech. and Eng.*, vol. 172, no. 1-4, pp. 203–240, Jul. 1999.

[168] M. Ortiz and J. Q. IV, “Adaptive mesh refinement in strain localization problems,” *Comput. Methods in Appl. Mech. and Eng.*, vol. 90, no. 1-3, pp. 781–804, Feb. 1991.

[169] C. W. Smith, D. Ibanez, B. Granzow, and G. Hansen. (2014). *PAALS Tutorial.* [Online]. Available: https://github.com/gahansen/Albany/wiki/PAALS-Tutorial-2014, Accessed on: Mar. 28, 2017.

[170] K. Ko, A. Candel, L. Ge, A. Kabel, R. Lee, Z. Li *et al.*, “Advances in parallel electromagnetic codes for accelerator science and development,” in *Proc. LINAC2010*, pp. 1028–1032.

[171] X. Luo, M. S. Shephard, J.-F. Remacle, R. M. O’Bara, M. W. Beall, B. Szabó *et al.*, “p-version mesh generation issues,” in *Proc. 11th Int. Meshing Roundtable*, 2002, pp. 343–354.

[172] X.-J. Luo, M. S. Shephard, R. M. O’bara, R. Nastasia, and M. W. Beall, “Automatic p-version mesh generation for curved domains,” *Eng. with Comput.*, vol. 20, no. 3, pp. 273–285, Sep. 2004.

[173] P. Ingelström, “A new set of h (curl)-conforming hierarchical basis functions for tetrahedral meshes,” *Microwave Theory and Techn., IEEE Trans.*, vol. 54, no. 1, pp. 106–114, Jan. 2006.

[174] V. L. Marrero, J. A. Tichy, O. Sahni, and K. E. Jansen, “Numerical study of purely viscous non-newtonian flow in an abdominal aortic aneurysm,” *J. Biomechanical Eng.*, vol. 136, no. 10, pp. 101 001–1–101 001–10, Oct. 2014.

[175] O. Sahni, F. Behafarid, and L. Fovargue, “3D finite element formulation of nonlinear partial-slip condition on curved geometries,” presented at the 67th Annu. Meeting of the APS Division of Fluid Dynamics, San Francisco, CA, USA, Nov. 23-25, 2014.

[176] C. W. Smith. *PHASTA Science Gateway* [Online]. Available: https://phasta.scigap.org, Accessed on: Mar. 22, 2017.

[177] S. Marru, L. Gunathilake, C. Herath, P. Tangchaisin, M. Pierce, C. Mattmann *et al.*, “Apache Airavata: A framework for distributed applications and computational workflows,” in *Proc. ACM Workshop on Gateway Comput. Environments*, 2011, pp. 21–28.

[178] M. Pierce, S. Marru, B. Demeler, R. Singh, and G. Gorbet, “The Apache Airavata application programming interface: Overview and evaluation with

the UltraScan science gateway," in *Proc. 9th Gateway Comput. Environments Workshop*, 2014, pp. 25–29.

[179] M. E. Pierce, S. Marru, L. Gunathilake, D. K. Wijeratne, R. Singh, C. Wimalasena *et al.*, "Apache airavata: design and directions of a science gateway framework," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 16, pp. 4282–4291, Nov. 2015.

[180] ANSYS. *ANSYS CFX.* (1994) [Online]. Available: http://www.ansys.com/Products/Fluids/ANSYS-CFX, Accessed on: Mar. 29, 2017.

[181] Simmetrix. *Abstract: Abstract Simulation Modeling.* (1999) [Online]. Available: http://www.simmetrix.com/products/SimulationModelingSuite/GeomSimAbstract/GeomSimAbstract.html, Accessed on: Mar. 29, 2017.

[182] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Process.: 9th Int. Workshop*, 2003, pp. 44–60.

[183] D. L. Brown, P. Messina, P. Beckman, D. Keyes, J. Vetter, M. Anitescu *et al.*, "Scientific grand challenges," in *Crosscutting Technologies for Comput. at the Exascale*, 2010, pp. 1–116.

[184] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, "Achieving high sustained performance in an unstructured mesh CFD application," in *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*, 1999, pp. 1–13.

[185] LLNL. *MFEM: Modular finite element methods.* (2016) [Online]. Available: http://mfem.org, Accessed on: Mar. 17, 2017.

[186] SCOREC. *EnGPar GitHub repository.* (2016) [Online]. Available: https://github.com/SCOREC/EnGPar, Accessed on: Mar. 30, 2016.

[187] CERN. *Zenodo.* (2013) [Online]. Available: https://zenodo.org, Accessed on: Mar. 17, 2017.

APPENDIX A
IN-MEMORY EXAMPLE CODE

Example code for using the POSIX C APIs and C++ `iostream` for data streaming are shown in Listings A.1 and A.2, respectively. Additional details on their compilation and usage are available in a Zenodo [187] dataset (`http://dx.doi.org/10.5281/zenodo.345749`). The dataset also includes the timed version of the POSIX C example code that was used to generate the bandwidth results shown in Fig. 4.5.

Listing A.1. POSIX C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char** argv) {
  const char *method, *mode;
  int i;
  size_t bytes;
  FILE* f;
  char filename[1024];
  char* buf = NULL;
  size_t len;
  char* data;
  if( argc != 4 ) {
    printf("Usage: %s <stream|posix>"
           "<read|write> <number of bytes>\n",argv[0]);
    return 0;
  }
  method = argv[1];
  mode = argv[2];
  bytes = atoi(argv[3]);

  data = (char*) malloc(bytes*sizeof(char));
  for(i=0;i<bytes;i++) data[i] = 1;
  /**** open stream ****/
  if( !strcmp(method,"stream") && !strcmp(mode,"write") ) {
    f = open_memstream(&buf, &len);
  } if( !strcmp(method,"stream") && !strcmp(mode,"writeprealloc") ) {
    buf = malloc(bytes*sizeof(char));
    f = fmemopen(buf, bytes, "w");
  } else if( !strcmp(method,"stream") && !strcmp(mode,"read") ) {
    f = fmemopen(buf, bytes, "r");
  } else if( !strcmp(method,"stream") && !strcmp(mode,"readprealloc") ) {
    buf = malloc(bytes*sizeof(char));
    f = fmemopen(buf, bytes, "r");
  /**** open posix ****/
  } else if( !strcmp(method,"posix") && !strcmp(mode,"write") ) {
    f = fopen("/tmp/foo.txt", "w");
  } else if( !strcmp(method,"posix") && !strcmp(mode,"read") ) {
    sprintf(filename,"/tmp/%lu.dat",bytes);
    f = fopen(filename, "r");
  }

  /**** read|write ****/
  if( !strcmp(mode,"write") || !strcmp(mode,"writeprealloc") ) {
    fwrite(data,sizeof(char),bytes,f);
  } else if( !strcmp(mode,"read") || !strcmp(mode,"readprealloc") ) {
    fread(data,sizeof(char),bytes,f);
  }
  fclose(f);

  if( !strcmp(method,"stream") &&
      ( !strcmp(mode,"write") ||
        !strcmp(mode,"writeprealloc") ||
        !strcmp(mode,"readprealloc") ) ) {
    free(buf);
  }
  free(data);
  return 0;
}
```

Listing A.2. C++ iostream

```
#include <iostream>
#include <fstream>
#include <sstream>

using namespace std;

ostream* open_writer(const char* name, bool stream);
istream* open_reader(const char* name, ostream* os=NULL);
void write(ostream* fh, int* data, size_t len);
size_t read(istream* fh, int*& data);

ostream* open_writer(const char* name, bool stream) {
  if(stream) {
    (void) name;
    ostringstream* oss = new ostringstream;
    return oss;
  } else {
    ofstream* ofs = new ofstream;
    ofs->open(name,ofstream::binary);
    return ofs;
  }
}

istream* open_reader(const char* name, ostream* os) {
  if(os) {
    (void) name;
    ostringstream* oss = reinterpret_cast<ostringstream*>(os);
    istringstream* iss = new istringstream(oss->str());
    return iss;
  } else {
    ifstream* ifs = new ifstream;
    ifs->open(name,ifstream::binary);
    return ifs;
  }
}

void write(ostream* fh, int* data, size_t len) {
  const char* buf = reinterpret_cast<char*>(data);
  streamsize sz = static_cast<streamsize>(len*sizeof(int));
  fh->write(buf,sz);
}

size_t read(istream* fh, int*& data) {
  fh->seekg(0,fh->end);
  streamsize sz = fh->tellg();
  fh->seekg(0,fh->beg);
  cout<< "read size " << sz << "\n";
  size_t numints = static_cast<size_t>(sz)/sizeof(int);
  cout<< numints << "\n";
  data = new int[numints];
  char* buf = reinterpret_cast<char*>(data);
  fh->read(buf,sz);
  return numints;
}

int main() {
  const char* fname = "foo.txt";
  int outdata[3] = {0,3,13};
  for(int i=0;i<2;i++) {
    bool streaming = i;
    ostream* oh = open_writer(fname,streaming);
    write(oh,outdata,3);
    int* indata = NULL;
    istream* ih = open_reader(fname,oh);
    size_t len = read(ih,indata);
    delete oh;
    delete ih;
    for(size_t j=0; j<len; j++)
      cout << indata[j] << " ";
    cout << "\n";
    delete [] indata;
  }
}
```