CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–23
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

In-memory integration of existing software components for parallel adaptive unstructured mesh workflows

Cameron W. Smith[1*], Brian Granzow[1], Gerrett Diamond[1], Daniel Ibanez[2], Onkar Sahni[1], Kenneth E. Jansen[3], Mark S. Shephard[1]

[1] *SCOREC Rensselaer Polytechnic Institute 110 8th St., Troy NY 12180*
[2] *Sandia National Laboratories P.O. Box 5800, Albuquerque, NM 87185-1321*
[3] *University of Colorado Boulder, 1111 Engineering Dr., Boulder, CO 80309*

SUMMARY

Reliable mesh-based simulations are needed to solve complex engineering problems. Mesh adaptivity can increase reliability by reducing discretization errors, but requires multiple software components to exchange information. Often, components exchange information by reading and writing a common file format. This file-based approach becomes a problem on massively parallel computers where filesystem bandwidth is a critical performance bottleneck. Our approach using data streams and component interfaces avoids the filesystem bottleneck. In this paper we present these techniques, and their use for coupling mesh adaptivity to the PHASTA computational fluid dynamics solver, the Albany multi-physics framework, and the Omega3P linear accelerator frequency analysis applications. Performance results are reported on up to 16,384 cores of an Intel Knights Landing-based system.

Received . . .

KEY WORDS: in-memory; parallel; unstructured mesh; workflow; mesh adaptation

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 [1] 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, $1/19th$ of L1 cache [2]. One level further up the hierarchy is the parallel filesystem[†]. 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 I lists the per node peak main memory and filesystem bandwidth across five generations of Argonne National Laboratory leadership class systems: Blue Gene/L [5, 6], Intrepid Blue Gene/P [7, 8], Mira Blue Gene/Q [1, 9], Theta [10, 11], and 2018's Aurora [12]. 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.

*Correspondence to: Email: smithc11@rpi.edu
[†]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 [3, 4].

Table I. 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)

This paper presents a set of in-memory component coupling techniques that avoid filesystem use. We demonstrate these techniques for three different unstructured mesh-based adaptive analysis 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.

Section 3 provides a definition of the information transfer modes and reviews methods to couple workflow components using them. The core interfaces supporting adaptive unstructured mesh workflows are described in Section 3.1, and examples are given for their use in bulk and atomic information transfers. Section 3.2 details the data stream approach developed to avoid filesystem use. Section 4 examines the use of data streams to couple the PHASTA massively parallel computational fluid dynamics analysis package with mesh adaptation. Section 5 describes the use of interfaces to couple the Trilinos based Albany multi-physics framework to a set of mesh services. Section 6 discusses the coupling of curved mesh adaptation and load balancing components with the linear accelerator frequency analysis package Omega3P. In each application example, discussions are broken down into sub-sections covering the PUMI integration, the example's workflow, and performance test results. Section 7 closes the paper.

Throughout the paper the following notation is used:

- C/C++/Fortran code is in a `fixed-pitch` font. For example, the function `printf(...)` is declared in `stdio.h`; the ellipsis represent omitted arguments.
- The suffix Ki denotes 2^{10}. So, for example, 16Ki is equal to $16 * 2^{10} = 16384$.

2. CONTRIBUTIONS AND RELATED WORKS

Our work in scientific computing focuses on the interactions of unstructured mesh-based components executing in the same memory space. This paper extends our previous XSEDE16 conference article [13] with more in-depth discussion of the atomic and bulk information passing approaches, an additional application example, and results on up to 16Ki cores of an Intel Knights Landing system; eight times more cores than the XSEDE16 paper and using a newer generation of architecture.

The three applications discussed all use the Parallel Unstructured Mesh Infrastructure (PUMI) to provide mesh adaptation, load balancing, and field services [14]. In each application an existing analysis package is integrated with PUMI using a specific combination of bulk and atomic information passing methods. The method selected, and the point in the workflow for transferring the information, is a key focus of our work. For interactions with the computational fluid dynamics code PHASTA, a bulk data-stream approach re-uses an existing file protocol to minimize code changes. As shown in the appendix, the code changes to implement this approach are minimal as the same APIs can be used for reading and writing. In PHASTA we implemented an additional abstraction layer to provide clean initialization and finalization APIs, ensure backward compatibility, and support future I/O methods. In total, this abstraction layer was about 500 lines of code with a very low cyclomatic complexity (maximum function score of two from GNU

DOI: 10.1002/cpe

Complexity) [15, 16]. Integration with the Albany multi-physics code uses bulk transfers via PUMI and Albany APIs to pass mesh and field information. In the third application, Omega3P, we use a combination of bulk transfers to pass large collections of mesh and field data structures, and atomic mesh element information transfers for element stiffness matrix formation. In all three cases, efforts focused on selecting points within the adaptive workflow to exchange information and, given our in-depth knowledge of PUMI, determine what analysis software APIs are provided to satisfy the exchanges.

The information passing approaches we used are common to other applications with similar design characteristics. Like PUMI, libMesh [17] and MOAB [18] provide MPI-based unstructured mesh-based services and have been successfully integrated with existing analysis codes using API-based interactions. ParFUM [19] also provides some similar unstructured mesh-based services, but given the requirement to work within the Charm++ [20] runtime-based system, has limited support for integrating with existing software packages. The Helios [21] software framework supports the in-memory integration of multiple existing software packages, including those providing unstructured mesh services, through a common API for bulk and atomic transfers to support rotorcraft aerodynamics simulations. For some domains or developers the cost of implementing the standard API is well worth the access to the ecosystem of tools, and their users, the framework provides.

Outside the domain of scientific computing, performance sensitive enterprise applications use atomic interactions with in-memory databases that far out perform those that reside on disk [22]. Distributed web-based applications often perform bulk transfers between memory spaces across a network via optimized serialization protocols [23, 24]. Similarly, using ADIOS DataSpaces, applications (typically for scientific computing) can perform bulk transfers between different executables [25, 26, 27, 28], on the same node or otherwise, without relying on slow filesystem interactions.

3. 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 [29]. 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 [30, 31, 32]. 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 [33]. In Sections 4 through 6 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 6 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 avoid 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, ADIOS DataSpaces provides a mechanism for the in-memory coupling of multiple executables [25, 26, 27, 28]. Likewise, Rasquin et al. [34] demonstrated in-situ visualization with

DOI: 10.1002/cpe

PHASTA and ParaView using GLEAN [35]. These inter-executable interaction methods can support the necessary workflow interactions, but typically at the cost of decreased performance, especially when information is passed between different system nodes.

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 [36]. 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 [21].

Components that support a common file format and use one file per process (e.g., POSIX C `stdio.h` [37] 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 [24], or Cap'n Proto [23], 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.

3.1. Component Interfaces

The components in adaptive simulations that provide geometric model, mesh, and field information [38, 14, 39], and the relations between them, are essential to error estimation, adaptation [40, 41, 17], and load balancing [42] services. For example, transferring field tensors during mesh adaptation requires the field-to-mesh relation [43]. Likewise, the mesh-to-model relation (classification [14]) and geometric model shape information enable mesh modifications (e.g., vertex re-positioning) that are consistent with the actual geometric domain [38]. Similarly, classification supports the transformation of the input field tensors onto the mesh to define the boundary conditions, material parameters and initial conditions [44]. Because of this strong dependency, we provide these components and services together as the open-source Parallel Unstructured Mesh Infrastructure (PUMI) [14, 45]. 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 [46]
- APF_MDS - partitioned mesh representation using MDS
- Field - describes the variation of tensor quantities over the domain
- ParMA - multi-criteria dynamic load balancing [42]
- 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

DOI: 10.1002/cpe

improved finite element solution using a patch-level Zienkiewicz-Zhu [47] 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 [48, 49].

3.2. 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` [37] or C++ `iostream` String Stream APIs [50] 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.

4. PHASTA

PHASTA solves complex fluid flow problems [51, 52, 53, 54, 55] on up to 768Ki cores with 3Mi ($3 * 2^{20}$) MPI processes [56] using a stabilized finite element method [57] 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 [45, **?**], 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.

4.1. Integration with PUMI

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 [58] and several of Miller's Smart Library techniques [30]. 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.

4.2. Adaptive Dam-break Example

Fig. 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 [52] . The dense fluid (water) is initially held against the left wall (not pictured) in a square column created by a fictitious constraint

DOI: 10.1002/cpe

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 [52]. 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 1 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 3.2. Next, the chef preprocessor is called on Line 6. The preprocessor first executes adjacency-based mesh entity reordering (l.18) [59] 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.19-20). Preprocessing concludes with the writing of this data to files/streams (l.21).

Line 8 of Algorithm 1 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.34), then executes an analyst-specified number of time steps (l.35), and computes the distance-based mesh size field (l.36). 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.23). To prevent memory exhaustion during mesh refinement procedures, ParMETIS part k-way graph re-partitioning (via Zoltan) is called using weights that approximate the change in mesh element count on each part (l.25, 28). After adaptation, chef executes preprocessing as previously described (l.14).

4.3. Data Stream Performance Testing

We measured the performance of PHASTA-chef [60] POSIX file and data stream information exchange in a workflow supporting the adaptive analysis of a two-phase, incompressible dam-break flow, as shown in Fig. 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 [10, 11]. The two Theta filesystems used by POSIX file tests, GPFS [61] and Lustre [62], 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 [11].

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 [42], 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. 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.

DOI: 10.1002/cpe

Algorithm 1 Two-phase PHASTA-chef Adaptive Loop

```
1:  procedure ADAPTIVELOOP(in max_steps)
2:      pumi_mesh ← load the partitioned PUMI mesh from disk
3:      geom ← load the geometric model from disk
4:      chef_probdef ← 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 ← 0
8:      while step_number < max_steps do
9:          PHASTA(N,file_handle)
10:         step_number ← 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:     end while
16: end procedure
17: procedure PREPROCESSOR(in pumi_mesh, in geom, out chef_probdef, in/out file_handle)
18:     reorder the mesh entities holding degrees-of-freedom
19:     phasta_mesh ← create PHASTA mesh data structures
20:     phasta_fields ← create PHASTA field data structures
21:     write phasta_mesh and phasta_fields to file_handle
22: end procedure
23: procedure MESHADAPT (in/out pumi_mesh, in size_field, in max_iterations)
24:     w ← per element field estimating the change in element volume based on size_field
25:     predictively balance the mesh elements for element weight w
26:     for iteration ← 0 to max_iterations do
27:         coarsen the mesh
28:         re-balance the mesh elements for element weight w
29:         refine the mesh
30:     end for
31:     re-balance the mesh elements
32: end procedure
33: procedure PHASTA(in N, in/out file_handle)
34:     read phasta_mesh, phasta_fields data from file_handle
35:     run the flow solver for N steps
36:     size_field ← isotropic size field based on distance to phasic interface
37:     write the mesh size_field and phasta_fields to file_handle
38: end procedure
```

The minimum, maximum, and average number of bytes read and written per process in a cycle by chef and PHASTA is plotted in Fig. 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.

The time spent by chef transferring data to and from PHASTA is reported in Fig. 3 and Fig. 4. Note, the PHASTA times for these transfers are nearly identical and not reported here. Fig. 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. 4a and Fig. 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

DOI: 10.1002/cpe

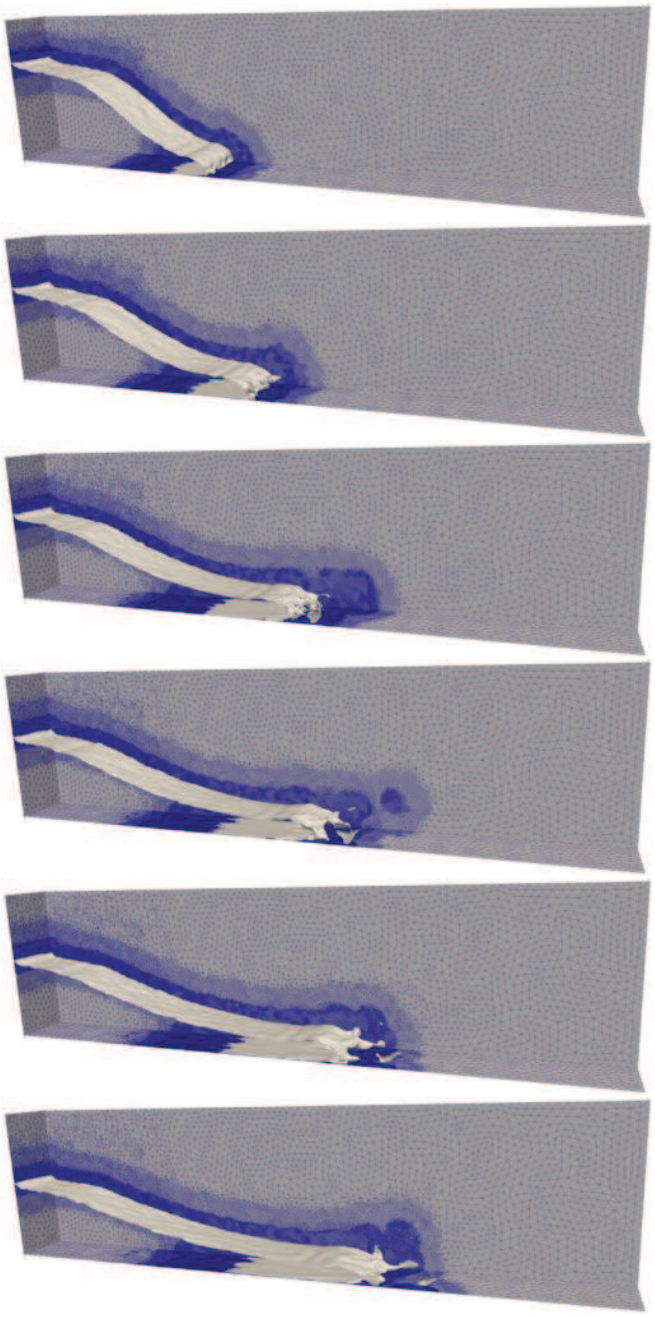


Figure 1. Evolution of an adaptive dam-break case ran on 2048 processes of the ALCF Theta system using two-phase, incompressible PHASTA coupled to PUMI unstructured mesh adaptation with data streams. Each image (top to bottom) represents an advancement in physical time by 1/100 of a second. The air-water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

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. 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.

DOI: 10.1002/cpe

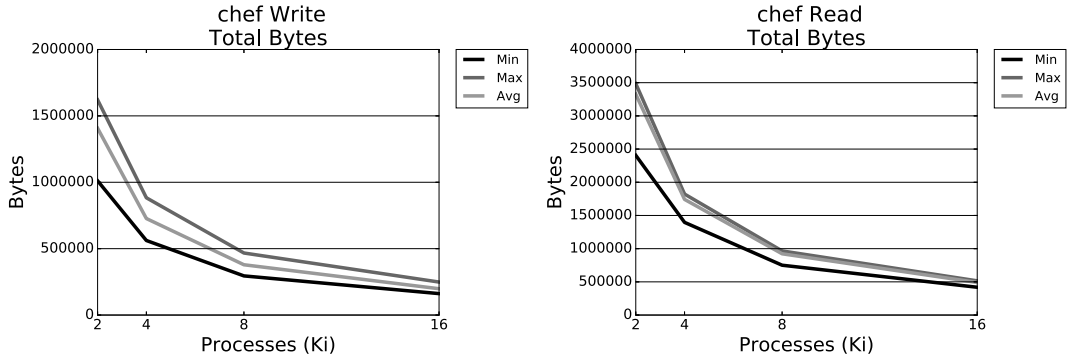


Figure 2. chef total bytes read and written per process. PHASTA reads(writes) the same number of bytes that chef wrote(reads).

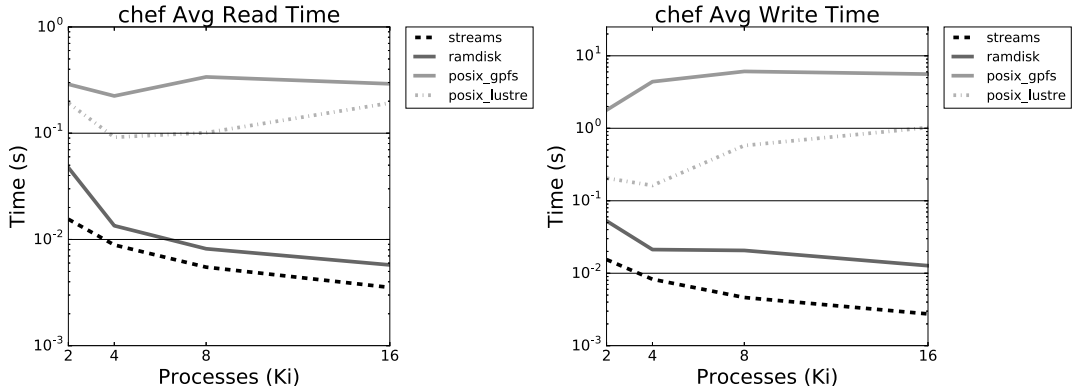
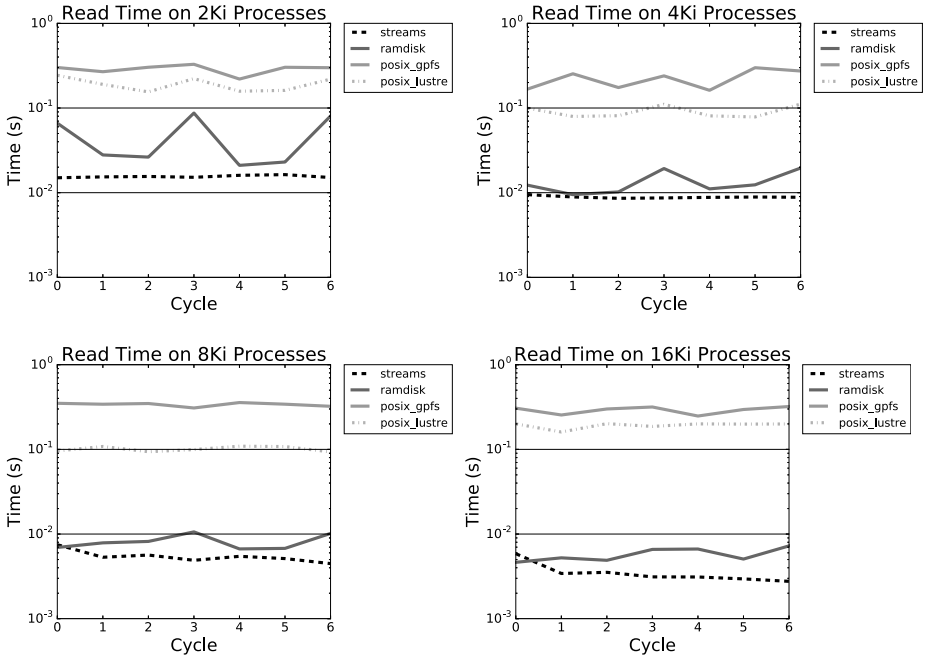


Figure 3. chef streams, ramdisk, and POSIX average read and write time on ALCF Theta. Lower is better.

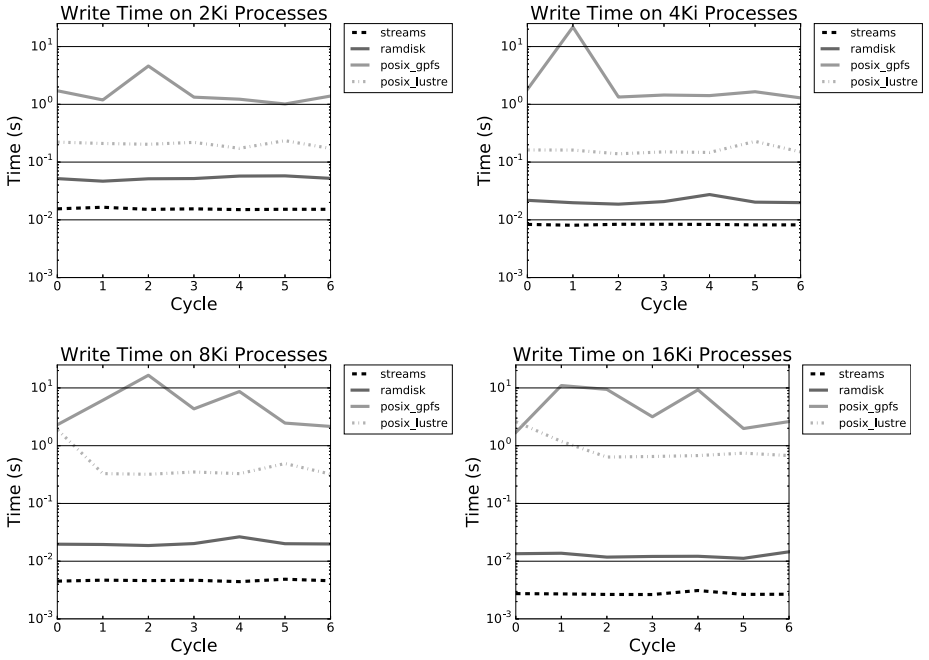
Furthermore, serial testing on one Theta node indicates that using preallocated buffers with `open_memstream` can further improve streaming write performance by over two times. The performance penalty of dynamic buffer expansion for the non-preallocated writes can clearly be seen in Fig. 5 by the large 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`) [63], but we have not tested these APIs on Theta.

The impact of I/O on overall workflow performance was measured on 16Ki processes of the ALCF Mira BlueGene/Q. In the test, the dam-break case was executed for three and a half cycles: four PHASTA solves (ten time steps each, with field updates enabled) that are interleaved with three mesh adaptations. Using the data stream method reduces the total execution time by 12%, approximately eight minutes, versus the POSIX file based method on a GPFS filesystem. While it may be tempting to focus on the impact of I/O on the overall workflow execution time, we caution readers that this measure 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 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.

DOI: 10.1002/cpe



(a) chef read times.



(b) chef write times.

Figure 4. Time for chef to read and write using streams, ramdisk, and POSIX on ALCF Theta. Lower is better.

5. ALBANY

Albany [64, 65] 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

DOI: 10.1002/cpe

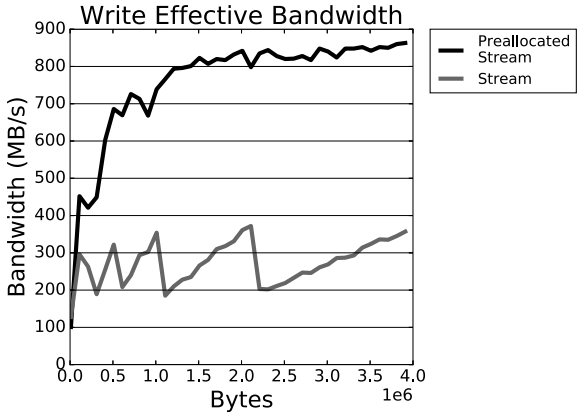


Figure 5. Streaming write performance with and without preallocation on a single node of ALCF Theta in the cache-quad configuration. Higher is better. The code used for these tests is described in the Appendix.

components and heavily leverages packages from the Trilinos project [66]. 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.

5.1. Integration with PUMI

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 2 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).

5.2. Adaptive Solderball Simulation

We ran an in-memory adaptive simulation of a solderball array subject to thermal creep [67]. Fig. 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 3 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

DOI: 10.1002/cpe

Algorithm 2 Construction of Neumann Boundary Condition Structure

1: // store the mapping of geometric model faces to `side_sets`
2: `invMap` $\leftarrow$ mapping of G_j^2 to `side_sets`
3: `size_set_list` $\leftarrow \emptyset$
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` $\leftarrow$ local id of M_j^3
13: `side_id` $\leftarrow$ local face index of M_i^2
14: `side_struct` $\leftarrow$ {`elm_LID`,`side_id`,M_j^3}
15: insert `side_struct` into `side_set_list`
16: **end if**
17: **end for**

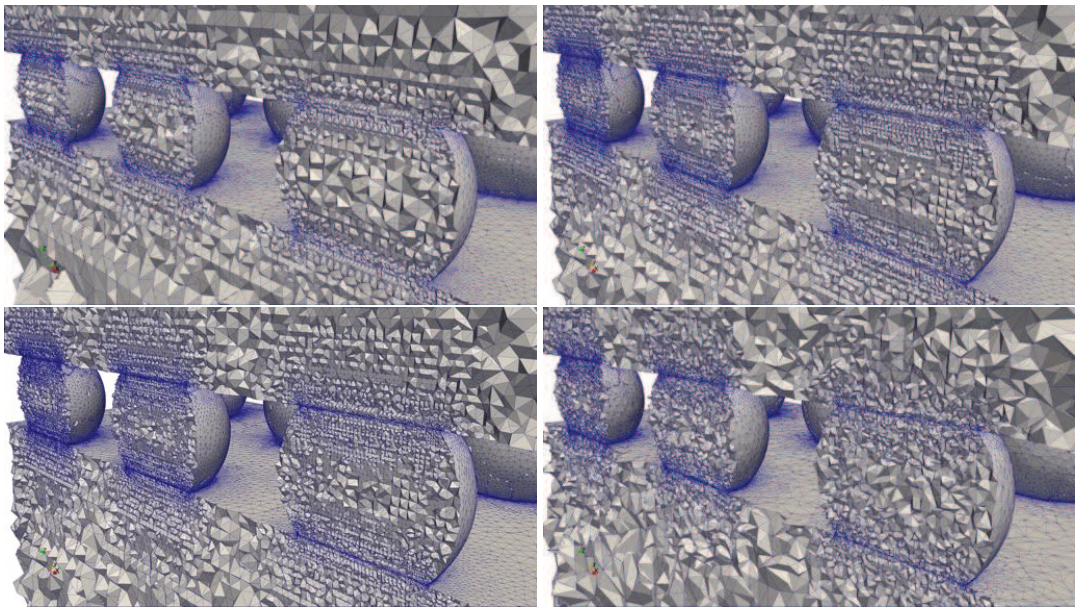


Figure 6. Four adaptation cycles (top to bottom, left to right) of 3x3 solderball mesh. The mesh is refined near the high stress gradients at the interface between the solderballs and the upper and lower slabs.

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` [14] (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 [68, 69] are called to determine the value of state variables at repositioned or newly created integration points. After mesh coarsening, Zoltan's interface [70] 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

DOI: 10.1002/cpe

transformation of PUMI unstructured mesh information (l.14-15) and APF field information (l.16) into Albany analysis data structures.

Algorithm 3 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)
17:     end while
18: end procedure
```

5.3. Performance Testing of In-memory Transfers

Fig. 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 (as shown by the large error bar in Fig. 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. 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.

6. OMEGA3P

Omega3P is a C++ component within ACE3P for frequency analysis of linear accelerator cavities [72]. 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.

DOI: 10.1002/cpe

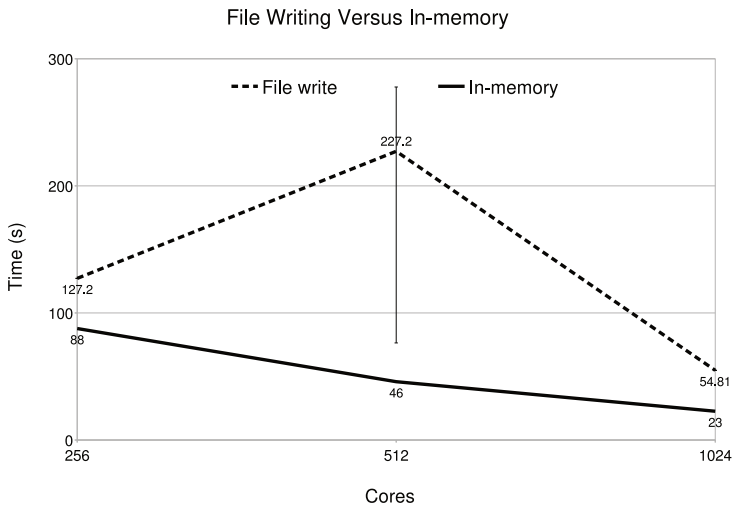


Figure 7. Average per cycle file writing and in-memory transfer times. Minimum and maximum bars are only shown for the 512 core file writing data point where they are 6% more, or less, than the mean, respectively. Lower is better.

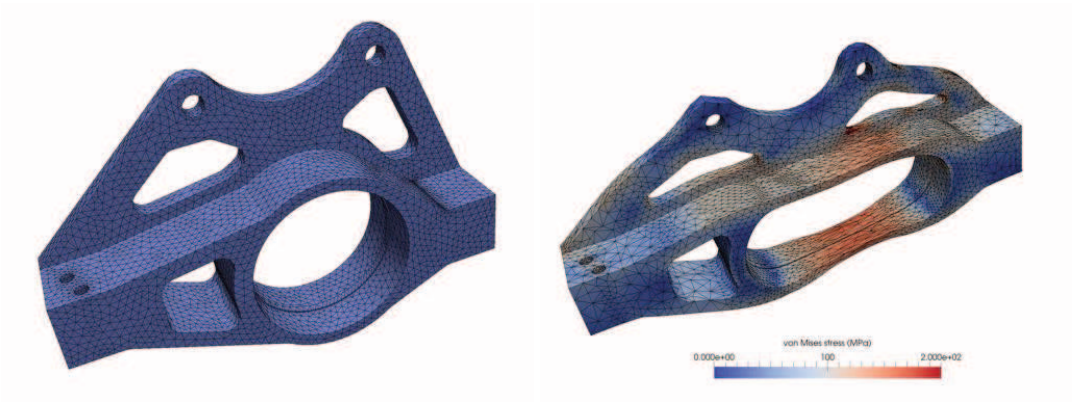


Figure 8. Large deformation of the RPI Formula Hybrid suspension upright [71].

6.1. Integration with PUMI

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 [73, 74, 75].

6.2. Adaptive Linear Accelerator Cavity Analysis

The mesh management and computational steps in the adaptive Omega3P-PUMI workflow are listed in Algorithm 4. Fig. 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

DOI: 10.1002/cpe

for quadratic Nedelec shape functions [76]) (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 5 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.

Algorithm 4 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
14:     end while
15: end procedure
```

Algorithm 5 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 5 Jacobian Calculation for Matrix Assembly

```
1:  // loop over DistMesh elements
2:  for all M_i^3 ∈ {M^3} do
3:      pumiElementPtr ← 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 ← getBaryCentricCoords(integration point)
8:          apf::Matrix3x3 J
9:          apf::getJacobian(pumiElementPtr, xi, J)
10:         // complete element matrix computation
11:     end for
12:     // insert element matrix contributions into stiffness matrix
13: end for
```


DOI: 10.1002/cpe

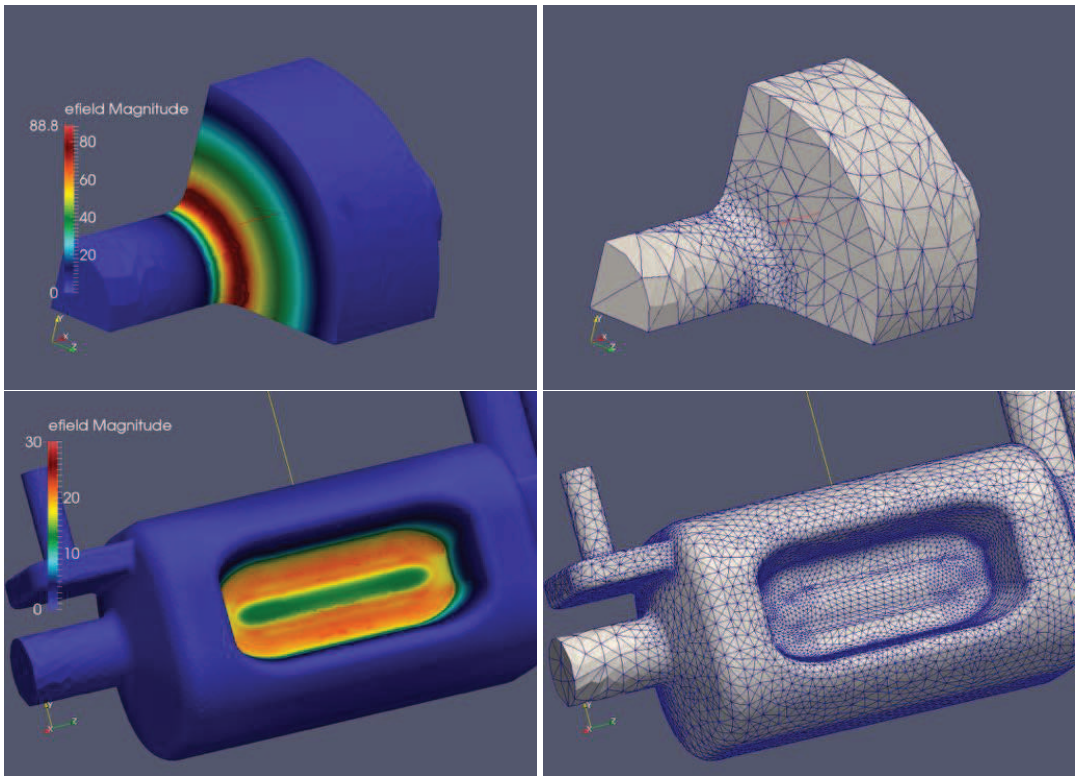


Figure 9. The first eigenmode electric field (left column) and adapted meshes (right column) for the `pillbox` (top row) and `cav17` (bottom row) test cases.

6.3. *Memory Usage Overhead*

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. Fig. 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. 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 2.4%, 0.8% and 1.3%, 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.

7. CLOSING REMARKS

As we move towards the exascale computers being considered [77, 78], it is clear that one of the only effective means to construct parallel adaptive simulations is by using in-memory interfaces that avoid filesystem interactions. In the strong scaling regime that these systems target, the problem size per core is decreasing and thus freeing up additional memory to utilize in-memory techniques that have a memory overhead. It is precisely in this regime that the in-memory coupling techniques become critical as overheads from IO can degrade scaling if not properly controlled. Determining the inflection point at which additional memory becomes available is dependant on the application and the problem being studied.

DOI: 10.1002/cpe

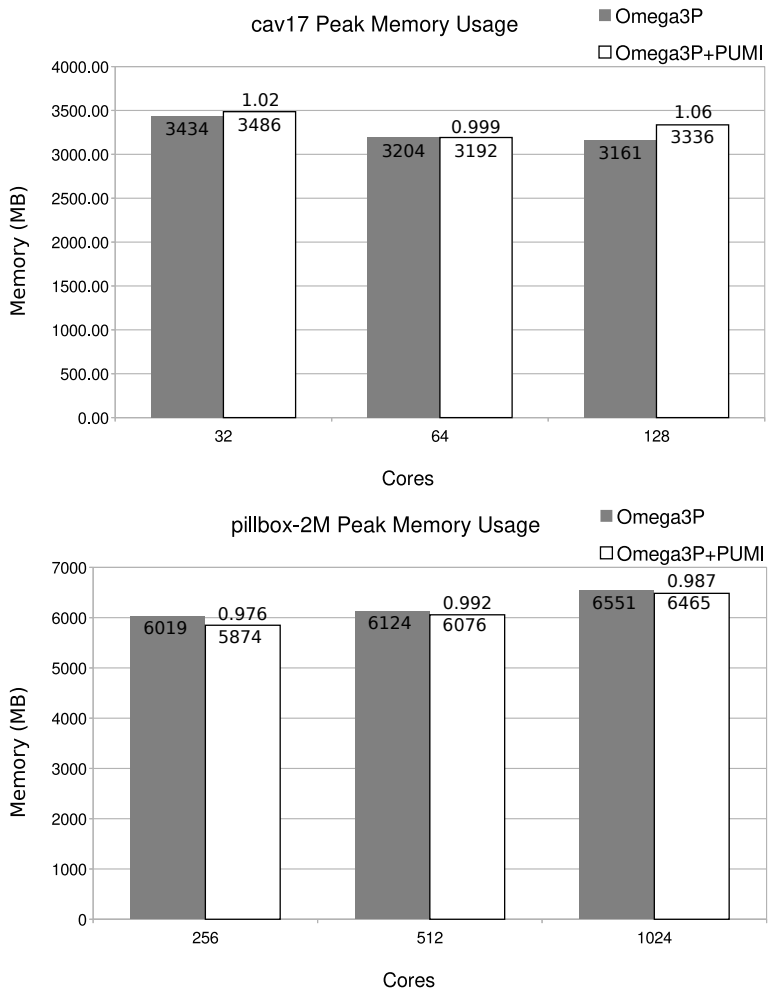


Figure 10. Peak per node memory usage for two Omega3P and Omega3P+PUMI test cases: (top) cav17 and (bottom) pillbox-2M. The numbers inside the bars list the peak memory usage in MB. The numbers above the Omega3P+PUMI bars list the ratio of the peak memory used by Omega3P+PUMI relative to the peak memory used by Omega3P.

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 (for data streams we simply replace existing I/O calls and write a simple abstraction layer), and demonstrated up to two orders of magnitude performance advantage of information transfer within adaptive simulations with negligible increases in overall memory usage. For a specific PHASTA adaptive workflow the I/O performance increase reduced the overall execution time by 12%. 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 [79] and LLNL's MFEM [80].

ACKNOWLEDGEMENTS

Thanks to Ben Matthews at the National Center For Atmospheric Research in Boulder, Colorado, for motivating the PHASTA streaming work and discussion of performance testing.

DOI: 10.1002/cpe

This research was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under awards DE-SC00066117 (FASTMath SciDAC Institute) and DE-SC0014609 (Community Project for Accelerator Science and Simulation-3).

An award of computer time was provided by the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program and a separate award of computer time by the Theta Early Science program. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

1. Haring R, Ohmacht M, Fox T, Gschwind M, Satterfield D, Sugavanam K, Coteus P, Heidelberger P, Blumrich M, Wisniewski R, *et al.*. The IBM Blue Gene/Q compute chip. *IEEE Micro* Mar 2012; **32**(2):48–60, doi: 10.1109/MM.2011.108.
2. Lo YJ, Williams S, Van Straalen B, Ligocki TJ, Cordery MJ, Wright NJ, Hall MW, Oliker L. Roofline model toolkit: A practical tool for architectural and program analysis. *5th Int. Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Comput. Syst.*, 2014; 129–148.
3. Besta M, Hoefler T. Fault tolerance for remote memory access programming models. *Proc. 23rd Int. Symp. High-performance Parallel and Distributed Comput.*, 2014; 37–48, doi:10.1145/2600212.2600224.
4. Islam TZ, Mohror K, Bagchi S, Moody A, de Supinski BR, Eigenmann R. MCREngine: A scalable checkpointing system using data-aware aggregation and compression. *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*, IEEE, 2012; 1–11, doi:10.1109/SC.2012.77.
5. Yu H, Sahoo RK, Howson C, Almasi G, Castanos JG, Gupta M, Moreira JE, Parker JJ, Engelsiepen TE, Ross RB, *et al.*. High performance file I/O for the Blue Gene/L supercomputer. *The 12th Int. Symp. High-Performance Comput. Architecture*, 2006; 187–196, doi:10.1109/HPCA.2006.1598125.
6. Adiga NR, Almasi G, Almasi GS, Aridor Y, Barik R, Beece D, Bellofatto R, Bhanot G, Bickford R, Blumrich M, *et al.*. An overview of the BlueGene/L supercomputer. *Proc. 2002 ACM/IEEE Conf. Supercomputing*, 2002; 60–60, doi:10.1109/SC.2002.10017.
7. Lang S, Carns P, Latham R, Ross R, Harms K, Allcock W. I/O performance challenges at leadership scale. *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*, IEEE, 2009; 40:1–40:12.
8. Alam S, Barrett R, Bast M, Fahey MR, Kuehn J, McCurdy C, Rogers J, Roth P, Sankaran R, Vetter JS, *et al.*. Early evaluation of IBM BlueGene/P. *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*, IEEE, 2008; 23:1–23:12.
9. Bui H, Finkel H, Vishwanath V, Habib S, Heitmann K, Leigh J, Papka M, Harms K. Scalable parallel I/O on a Blue Gene/Q supercomputer using compression, topology-aware data aggregation, and subfiling. *Parallel, Distributed and Network-Based Process. (PDP), 22nd Euromicro Int. Conf.*, 2014; 107–111.
10. Sodani A, Gramunt R, Corbal J, Kim HS, Vinod K, Chinthamani S, Hutsell S, Agarwal R, Liu YC. Knights Landing: Second-generation Intel Xeon Phi product. *IEEE Micro* Mar 2016; **36**(2):34–46, doi:10.1109/MM.2016.25.
11. Jeffers J, Reinders J, Sodani A. *Intel Xeon Phi Processor High Performance Programming*. Knights Landing edn., Morgan Kaufmann Inc.: Boston, MA, USA, 2016.
12. Aurora fact sheet 2015. URL `http://www.intel.com/newsroom/assets/Intel\_Aurora\_factsheet.pdf`.
13. Smith CW, Chitale K, Ibanez DA, Orecchio B, Seol ES, Sahni O, Jansen KE, Shephard MS. In-memory integration of existing software components for parallel adaptive unstructured mesh workflows. *XSEDE16*, 2016; 1–6, doi: 10.1145/2949550.2949650.
14. Ibanez DA, Seol ES, Smith CW, Shephard MS. PUMI: Parallel unstructured mesh infrastructure. *ACM Trans. Math. Softw.* May 2016; **42**(3):17:1–17:28, doi:10.1145/2814935.
15. McCabe TJ. A complexity measure. *IEEE Transactions on Software Engineering* Dec 1976; **SE-2**(4):308–320, doi:10.1109/TSE.1976.233837.
16. Gnu complexity website 2011. URL `\url{https://www.gnu.org/software/complexity/}`.
17. Kirk BS, Peterson JW, Stogner RH, Carey GF. libMesh : a C++ library for parallel adaptive mesh refinement/coarsening simulations. *Eng. with Comput.* Dec 2006; **22**(3):237–254, doi:10.1007/s00366-006-0049-3.
18. Tautges TJ, Meyers R, Merkley K, Stimpson C, Ernst C. MOAB: A mesh-oriented database. *Technical Report SAND2004-1592*, Sandia Nat. Labs, Albuquerque, NM, USA 2004.
19. Lawlor OS, Chakravorty S, Wilmarth TL, Choudhury N, Dooley I, Zheng G, Kalé LV. ParFUM: a parallel framework for unstructured meshes for scalable dynamic physics applications. *Eng. with Comput.* Sep 2006; **22**(3):215–235, doi:10.1007/s00366-006-0039-5.
20. Kale LV, Krishnan S. Charm++: Parallel Programming with Message-Driven Objects. *Parallel Programming using C++*, Wilson GV, Lu P (eds.). MIT Press, 1996; 175–213.
21. Sankaran V, Sitaraman J, Wissink A, Datta A, Jayaraman B, Potsdam M, Mavriplis D, Yang Z, O'Brien D, Saberi H, *et al.*. Application of the Helios computational platform to rotorcraft flowfields. *48th AIAA Aerospace Sciences Meeting*, vol. 1230, 2010; 1–28, doi:http://dx.doi.org/10.2514/6.2010-1230.
22. Larson PA, Lomet D ((eds.)). *Special Issue on Main-Memory Database Systems*, vol. 36. IEEE, 2013.
23. Varda K, Renshaw D. Cap'n proto 2016. URL `https://capnproto.org/`.
24. van Oortmerssen W. Flatbuffers 2016. URL `http://google.github.io/flatbuffers/index.html`.
25. Bennett JC, Abbasi H, Bremer PT, Grout R, Gyulassy A, Jin T, Klasky S, Kolla H, Parashar M, Pascucci V, *et al.*. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*, IEEE, 2012; 1–9.
26. Zhang F, Docan C, Parashar M, Klasky S, Podhorszki N, Abbasi H. Enabling in-situ execution of coupled scientific workflow on multi-core platform. *Proc. 26th Int. Parallel & Distributed Process. Symp.*, 2012; 1352–1363.

27. Docan C, Parashar M, Klasky S. Dataspaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing* Jun 2012; **15**(2):163–181.
28. Sun Q, Jin T, Romanus M, Bui H, Zhang F, Yu H, Kolla H, Klasky S, Chen J, Parashar M. Adaptive data placement for staging-based coupled scientific workflows. *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*, IEEE, 2015; 65:1–65:12.
29. Szyperski C. *Component Software: Beyond Object-Oriented Programming*. Second edn., ACM Press and Addison-Wesley: New York, NY, USA, 2002.
30. Miller M, Reus J, Matzke R, Koziol Q, Cheng A. Smart libraries: Best SQE practices for libraries with emphasis on scientific computing. *Proc. Nucl. Explosives Code Developer's Conf.*, 2004; 1–30.
31. Brown J, Knepley MG, Smith BF. Run-time extensibility and librarization of simulation software. *Comput. in Sci. & Eng.* Feb 2015; **17**(1):38–45, doi:doi.ieeecomputersociety.org/10.1109/MCSE.2014.95.
32. Gropp W. Exploiting existing software in libraries: successes, failures, and reasons why. *Object Oriented Methods for Interoperable Scientific and Eng. Comput.: Proc. 1998 SIAM Workshop*; 21–29.
33. Smith B, Bartlett R. xSDK community package policies. *Technical Report V0.3*, U.S. Dept. Energy, Office of Sci., Chicago, IL, USA 2016, doi:10.6084/m9.figshare.4495136.v2.
34. Rasquin M, Marion P, Vishwanath V, Matthews B, Hereld M, Jansen K, Loy R, Bauer A, Zhou M, Sahni O, *et al.*. Electronic poster: Co-visualization of full data and in situ data extracts from unstructured grid CFD at 160k cores. *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*, IEEE, 2011; 103–104, doi:10.1145/2148600.2148653.
35. Vishwanath V, Hereld M, Morozov V, Papka ME. Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems. *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*, IEEE, 2011; 19:1–19:11, doi:10.1145/2063384.2063409.
36. Burstedde C, Wilcox LC, Ghattas O. `p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM J. Scientific Comput.* May 2011; **33**(3):1103–1133, doi:10.1137/100791634.
37. International standard - information technology portable operating system interface (POSIX)base specifications 2009, doi:10.1109/IEEESTD.2009.5393893.
38. Beall MW, Walsh J, Shephard MS. A comparison of techniques for geometry access related to mesh generation. *Eng. with Comput.* Aug 2004; **20**(3):210–221.
39. Ollivier-Gooch C, Diachin L, Shephard MS, Tautges T, Kraftcheck J, Leung V, Luo XJ, Miller M. An interoperable, data-structure-neutral component for mesh query and manipulation. *Trans. Math. Software* Sep 2010; **37**(3):29:1–29:28.
40. Park MA, Darmofal DL. Parallel anisotropic tetrahedral adaptation. *46th AIAA Aerospace Sciences Meeting and Exhibit*, 2008; 1–19.
41. Loseille A, Menier V, Alauzet F. Parallel generation of large-size adapted meshes. *Proc. 24th Int. Meshing Roundtable*, 2014; 57–69.
42. Smith CW, Rasquin M, Ibanez D, Jansen KE, Shephard MS. Improving unstructured mesh partitions for multiple criteria using mesh adjacencies. *SIAM J. Scientific Comput.* ; Accepted.
43. Farrell P, Maddison J. Conservative interpolation between volume meshes by local galerkin projection. *Comput. Methods in Appl. Mech. and Eng.* Jan 2011; **200**(1-4):89–100, doi:http://dx.doi.org/10.1016/j.cma.2010.07.015.
44. O'Bara R, Beall M, Shephard M. Attribute management system for engineering analysis. *Eng. with Comput.* Nov 2002; **18**(4):339–351.
45. SCOREC. PUMI GitHub repository 2014. URL `https://github.com/SCOREC/core`.
46. Ibanez D, Shephard MS. Modifiable array data structures for mesh topology. *SIAM J. Scientific Comput.* 2017; **39**(2):C144–C161, doi:10.1137/16M1063496.
47. Zienkiewicz OC, Zhu JZ. The superconvergent patch recovery and a posteriori error estimates. part 1: The recovery technique. *Int. J. Numerical Methods in Eng.* May 1992; **33**(7):1331–1364, doi:10.1002/nme.1620330702.
48. Alauzet F, Li X, Seol ES, Shephard MS. Parallel anisotropic 3D mesh adaptation by mesh modification. *Eng. with Comput.* May 2006; **21**(3):247–258, doi:10.1007/s00366-005-0009-3.
49. Frey P, Alauzet F. Anisotropic mesh adaptation for CFD computations. *Comput. Methods in Appl. Mech. and Eng.* Nov 2005; **194**(48-49):5068–5082, doi:http://dx.doi.org/10.1016/j.cma.2004.11.025.
50. Langer A, Kreft K. *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference*. Addison-Wesley Professional: Reading, MA, USA, 2000.
51. Galimov AY, Sahni O, Jr RTL, Shephard MS, Drew DA, Jansen KE. Parallel adaptive simulation of a plunging liquid jet. *Acta Mathematica Scientia* Mar 2010; **30**(2):522–538, doi:10.1016/S0252-9602(10)60060-4.
52. Rodriguez JM, Sahni O, Jr RTL, Jansen KE. A parallel adaptive mesh method for the numerical simulation of multiphase flows. *Comput. & Fluids* Oct 2013; **87**:115–131, doi:10.1016/j.compfluid.2013.04.004.
53. Sahni O, Zhou M, Shephard MS, Jansen KE. Scalable implicit finite element solver for massively parallel processing with demonstration to 160k cores. *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*, IEEE, 2009; 1–12.
54. Zhou M, Sahni O, Kim HJ, Figueroa CA, Taylor CA, Shephard MS, Jansen KE. Cardiovascular flow simulation at extreme scale. *Computational Mech.* Dec 2010; **46**(1):71–82.
55. Smith CW, Jansen KE. PHASTA GitHub repository 2015. URL `https://github.com/PHASTA/phasta`.
56. Rasquin M, Smith C, Chitale K, Seol ES, Matthews BA, Martin JL, Sahni O, Loy RM, Shephard MS, Jansen KE. Scalable implicit flow solver for realistic wing simulations with flow control. *Comput. in Sci. & Eng.* Dec 2014; **16**(6):13–21, doi:http://dx.doi.org/10.1109/MCSE.2014.75.
57. Whiting CH, Jansen KE. A stabilized finite element method for the incompressible navier-stokes equations using a hierarchical basis. *Int. J. Numerical Methods in Fluids* Jan 2001; **35**(1):93–116, doi:10.1002/1097-0363(20010115)35:1⟨93::AID-FLD85⟩3.0.CO;2-G.
58. Alexandrescu A. *Modern C++ Design : Generic Programming and Design Patterns Applied*. Addison-Wesley: Boston, MA, USA, 2001.

59. Zhou M, Sahni O, Shephard MS, Carothers CD, Jansen KE. Adjacency-based data reordering algorithm for acceleration of finite element computations. *Scientific Programming* May 2010; **18**(2):107–123.
60. Smith CW. PHASTA-chef GitHub repository 2016. URL `https://github.com/PHASTA/phastaChef`.
61. Schmuck F, Haskin R. GPFS: A shared-disk file system for large computing clusters. *FAST '02: Proc. 1st USENIX Conf. File and Storage Technologies*, 2002; 1–15.
62. Wang F, Oral S, Shipman G, Drokin O, Wang T, Huang I. Understanding Lustre internals. *Technical Report 951297*, Oak Ridge Nat. Lab (ORNL); Center for Computational Sciences, Oak Ridge, TN, USA 2009, doi:10.2172/951297.
63. Bhattacharya S, Pratt S, Pulavarty B, Morgan J. Asynchronous I/O support in Linux 2.5. *Proc. Linux Symp.*, 2003; 371–386.
64. Salinger AG, Bartett RA, Chen Q, Gao X, Hansen G, Kalashnikova I, Mota A, Muller RP, Nielsen E, Ostien J, *et al.*. Albany: A component-based partial differential equation code built on Trilinos. *Technical Report SAND2013-8430J*, Sandia Nat. Labs, Albuquerque, NM, USA 2013.
65. Salinger AG, Bartlett RA, Bradley AM, Chen Q, Demeshko IP, Gao X, Hansen GA, Mota A, Muller RP, Nielsen E, *et al.*. Albany: Using component-based design to develop a flexible, generic multiphysics analysis code. *Int. J. Multiscale Computational Eng.* 2016; **14**(4):415–438.
66. Heroux MA, Bartlett RA, Howle VE, Hoekstra RJ, Hu JJ, Kolda TG, Lehoucq RB, Long KR, Pawlowski RP, Phipps ET, *et al.*. An overview of the Trilinos project. *ACM Trans. Math. Softw.* Sep 2005; **31**(3):397–423, doi: http://doi.acm.org/10.1145/1089014.1089021.
67. Li Z, Bloomfield MO, Oberai AA. Simulation of finite-strain inelastic phenomena governed by creep and plasticity. *Computational Mech.* ; Submitted for publication.
68. Radovitzky R, Ortiz M. Error estimation and adaptive meshing in strongly nonlinear dynamic problems. *Comput. Methods in Appl. Mech. and Eng.* Jul 1999; **172**(1-4):203–240, doi:http://dx.doi.org/10.1016/S0045-7825(98)00230-8.
69. Ortiz M, IV JQ. Adaptive mesh refinement in strain localization problems. *Comput. Methods in Appl. Mech. and Eng.* Feb 1991; **90**(1-3):781–804, doi:http://dx.doi.org/10.1016/0045-7825(91)90184-8.
70. Devine K, Boman E, Heaphy R, Hendrickson B, Vaughan C. Zoltan data management services for parallel dynamic applications. *Comput. in Sci. Eng.* Mar 2002; **4**(2):90–96, doi:10.1109/5992.988653.
71. Smith CW, Ibanez D, Granzow B, Hansen G. Paals tutorial 2014. URL `https://github.com/gahansen/Albany/wiki/PAALS-Tutorial-2014`.
72. Ko K, Candel A, Ge L, Kabel A, Lee R, Li Z, Ng C, Rawat V, Schussman G, Xiao L, *et al.*. Advances in parallel electromagnetic codes for accelerator science and development. *Proc. LINAC2010*; 1028–1032.
73. Dey S, Shephard MS, Flaherty JE. Geometry representation issues associated with p-version finite element computations. *Comput. Methods in Appl. Mech. and Eng.* Dec 1997; **150**(1):39–55, doi:10.1016/S0045-7825(97)00103-5.
74. Luo X, Shephard MS, Remacle JF, O'Bara RM, Beall MW, Szabó B, Actis R. p-version mesh generation issues. *Proc. 11th Int. Meshing Roundtable*, 2002; 343–354.
75. Luo XJ, Shephard MS, O'bara RM, Nastasia R, Beall MW. Automatic p-version mesh generation for curved domains. *Eng. with Comput.* Sep 2004; **20**(3):273–285, doi:10.1007/s00366-004-0295-1.
76. Ingelström P. A new set of h (curl)-conforming hierarchical basis functions for tetrahedral meshes. *Microwave Theory and Techn., IEEE Trans.* Jan 2006; **54**(1):106–114.
77. Brown DL, Messina P, Beckman P, Keyes D, Vetter J, Anitescu M, Bell J, Brightwell R, Chamberlain B, Estep D, *et al.*. Scientific grand challenges. *Crosscutting Technologies for Comput. at the Exascale*, 2010; 1–116.
78. Dongarra J, Beckman P, Moore T, Aerts P, Aloisio G, Andre JC, Barkai D, Berthou JY, Boku T, Braunschweig B, *et al.*. The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.* Feb 2011; **25**(1):3–60, doi:10.1177/1094342010391989.
79. Anderson WK, Gropp WD, Kaushik DK, Keyes DE, Smith BF. Achieving high sustained performance in an unstructured mesh CFD application. *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*, IEEE, 1999; 1–13, doi:10.1145/331532.331600.
80. LLNL. MFEM: Modular finite element methods 2016. URL `http://mfem.org`.
81. CERN. Zenodo 2013. URL `https://zenodo.org`.

8. APPENDIX

Example code for using the POSIX C APIs and C++ `iostream` for data streaming are shown in Listings 1 and 2, respectively. Additional details on their compilation and usage are available in a Zenodo [81] 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. 5.

Listing 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;
}
```


DOI: 10.1002/cpe

Listing 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";
```


DOI: 10.1002/cpe

```
    delete [] indata;
  }
}
```


DOI: 10.1002/cpe