

## TOWARDS EXTREME-SCALE SIMULATIONS FOR LOW MACH FLUIDS WITH SECOND-GENERATION TRILINOS

PAUL LIN, MATTHEW BETTENCOURT, STEFAN DOMINO, TRAVIS FISHER, MARK HOEMMEN, JONATHAN HU, ERIC PHIPPS, ANDREY PROKOPENKO, SIVASANKARAN RAJAMANICKAM, CHRISTOPHER SIEFERT, STEPHEN KENNON

*Sandia National Laboratories, Post Office Box 5800 Mail Stop 1320  
Albuquerque, NM 87185-1320, USA*

Received July 2014

Revised September 2014

Communicated by Guest Editors

### ABSTRACT

Trilinos is an object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems. While Trilinos was originally designed for scalable solutions of large problems, the fidelity needed by many simulations is significantly greater than what one could have envisioned two decades ago. When problem sizes exceed a billion elements even scalable applications and solver stacks require a complete revision. The second-generation Trilinos employs C++ templates in order to solve arbitrarily large problems. We present a case study of the integration of Trilinos with a low Mach fluids engineering application (SIERRA low Mach module/Nalu). Through the use of improved algorithms and better software engineering practices, we demonstrate good weak scaling for up to a nine billion element large eddy simulation (LES) problem on unstructured meshes with a 27 billion row matrix on 524,288 cores of an IBM Blue Gene/Q platform.

*Keywords:* Solver library, Trilinos, Vertical integration, Extreme-scale simulations

### 1. Introduction

Trilinos is an object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems [1]. While Trilinos was originally designed for scalable solutions of large problems, the fidelity needed by many simulations is significantly greater than what one could have envisioned two decades ago. This requires a revision of Trilinos that supports arbitrarily large problem sizes and provides a path forward for achieving high performance on future architectures.

Computational simulations for many challenging scientific and engineering problems at Sandia National Laboratories (SNL) requires ever increasing fidelity. An example is the accurate simulation of fire environments that occur in accident scenarios., e.g. hydrocarbon pool fires. Figure 1 depicts an example hydrocarbon JP-8 pool fire experiment and a numerical simulation. The SIERRA/Fuego [2] engineer-

2 *Parallel Processing Letters*

ing application, built on top of the SIERRA Framework [3], is currently employed to simulate the fire environment. SIERRA/Fuego is a low Mach number, turbulent reacting flow code. It employs Trilinos' solvers through the Finite Element Interface (FEI). The choice of 32-bit ordinals for the entire stack from the application to solvers (SIERRA Framework, FEI, and Trilinos) has limited simulations to fewer than about two billion entities (e.g., nodes, elements, edges, matrix rows). This is a severe limitation on the fidelity of current and especially future simulations, and drove the development of a new application code.

This paper describes the second-generation Trilinos and its integration into a new engineering application (SIERRA low Mach module/Nalu; henceforth referred to as "Nalu") that does not employ the SIERRA Framework and FEI. This work focuses on the second-generation Trilinos' ability to enable high fidelity simulations that were not previously possible with SIERRA/Fuego and demonstrate scalability for very large problems. We limit this study to inter-node scalability. The new revision of Trilinos also addresses on-node scaling on both multicore nodes and accelerators through the Kokkos package (see [4] for Kokkos' design and path forward). This paper is an extension of previous work [5], and includes results at higher core counts and presents further improvements. Some of the software engineering challenges, e.g. the importance of special cases to improve performance, were discussed in the previous work and will not be discussed here.



Fig. 1. Hydrocarbon JP-8 pool fire (a) experiment and (b) numerical simulation

Our primary contributions in this work are:

- Significant revision of the second-generation Trilinos solver stack, focused on performance and scalability.
- Demonstrated the capability and scalability of the second-generation Trilinos with a new application SIERRA/Nalu for *9 billion nodes on 524,288 cores*.
- Case study describing issues and their solutions faced in integrating a large solver stack with an application, as well as best practices.

For perspective on the size of the simulations enabled by the new Nalu code compared with other applications, we briefly describe the few other comparable large scale simulations in the literature. Problem scale of interest depends on the modeling technique, e.g., Direct Numerical Simulation (DNS) or Large Eddy Simulation (LES), the choice of structured or unstructured meshes (latter more challenging) and explicit or implicit coupling strategies (latter more challenging). On structured meshes, Hawkes et al. [6] have demonstrated LES simulations of up to seven billion elements on 120k cores. On unstructured meshes, implicit LES of reacting flow simulations with 2.6 billion tetrahedral elements [7] and recently 20 billion tetrahedral elements (about 3.5 billion nodes) were done by CORIA-CNRS (University of Rouen and the Institute of Applied Sciences). Intermediate approaches are led by the Uintah/Arches code base effort of the University of Utah (C-SAFE ASCI Alliance center) where structured orthogonal explicit momentum is coupled with a linear pressure Poisson equation (PPE) solve using the Hypra algebraic multigrid preconditioner. Their simulations have reached 256k cores on meshes with about 6 billion structured hexahedral elements [8]. Although the low Mach number simulations performed in this study will reach 9 billion node meshes on 524,288 cores, the data presented will only reflect a small physics simulation time. Timings will be focused on matrix assembly and linear solve times. A full verification and validation study will be the focus of future work.

We summarize Trilinos and Nalu in Sections 2 and 3, then discuss the integration in Section 4. Section 5 presents results showing the new capability provided by the second-generation Trilinos, and Section 6 concludes and outlines future work.

## 2. Trilinos

### 2.1. *Trilinos Overview*

Trilinos [1] is an open-source software project to develop algorithms and enabling technologies for solving large-scale mathematical problems from scientific and engineering applications. Supported capabilities include distributed-memory parallel linear algebra, iterative solvers for large sparse linear systems and eigenproblems, nonlinear solvers and optimization, continuation and bifurcation analysis, time integrators, partial differential equation discretizations, parallel partitioning for load balance, incomplete factorizations and relaxations, multilevel preconditioners (such as algebraic multigrid), automatic differentiation, and embedded uncertainty quantification. Trilinos is based at Sandia National Laboratories, but includes significant external contributions. Most Trilinos packages have a modified BSD license; a few have the GNU Lesser General Public License. Trilinos' common build and test infrastructure substantially reduces the effort required to deploy new algorithms. Furthermore, many applications are standardizing on Trilinos' interfaces, which gives them access to all Trilinos solver components.

## 2.2. Trilinos Packages Used by Nalu

This paper mentions two “generations” of Trilinos solver stacks. The “second-generation” stack is a complete rewrite of the first stack, in order to support the following new capabilities:

- Solve problems with more entities than can be indexed by a standard 32-bit integer ( $> 2$  billion)
- Allow use of data types other than double-precision floating-point, e.g., complex or extended precision, without hard-to-maintain duplicate implementations
- Enable and simplify the ongoing addition of MPI + shared-memory parallelism for many different programming models (e.g., OpenMP, CUDA)

Table 1 summarizes both stacks. This paper demonstrates the second-generation stack’s scalability in e.g., Figure 4, especially when integrated in Nalu. Each one of these packages had to undergo significant changes to demonstrate scalability at this scale. The dependency diagram in Figure 2 shows the complex dependencies between these packages.

Table 1. Comparison of first- vs. second-generation Trilinos solver stack packages used by Nalu.

Functionality	Current	New
Distributed linear algebra	Epetra [1]	Tpetra [9]
Iterative linear solvers	AztecOO [10]	Belos [11]
Incomplete factorizations	AztecOO, Ifpack	Ifpack2
Algebraic multigrid	ML [12]	MueLu [13]
Partition & load balance	Zoltan [14]	Zoltan2[15]
Direct solvers interface	Amesos	Amesos2 [11]

We briefly mention how Nalu uses each package here and refer the reader to their respective references for other capabilities supported in these packages. For this study, all linear solves in Nalu used Belos’ [11] implementation of GMRES [16] or TFQMR [17]. Solves for all equations other than the pressure Poisson equation (PPE) were preconditioned using Ifpack2’s symmetric Gauss-Seidel preconditioner. For the PPE, Nalu used MueLu’s [13] algebraic multigrid preconditioner with Ifpack2’s Chebyshev smoother on all levels except for the coarsest, which used the SuperLU direct solver through the Amesos2 interface [11]. To improve efficiency of the multigrid preconditioner, MueLu rebalances its coarser levels using Zoltan2’s multijagged algorithm [18]. MueLu explicitly forms coarser-level matrices, for which a key computational kernel is Tpetra’s [9] sparse matrix-matrix multiply.

Zoltan2’s [15] multijagged algorithm [18] is a recent addition to Trilinos’ partitioning capabilities that is aimed at extreme-scale simulations. The original spatial partitioning algorithm in Zoltan [14] uses parallel recursive coordinate bisection (RCB). Parallel RCB migrates the coordinates after each bisection. At extreme scales, with the number of parts in the hundreds of thousands and the number of

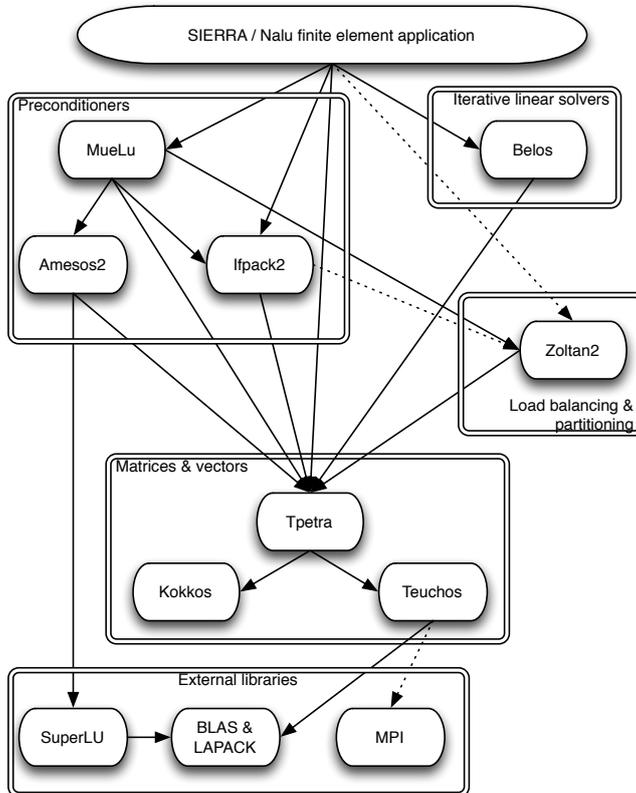


Fig. 2. Software dependencies between SIERRA / Nalu, the Trilinos packages it uses, and externally provided libraries. Arrows show dependencies pointing at the target, that is, the “client” used by the other (source) package. Solid lines indicate currently required dependencies, and dotted lines indicate optional or likely future dependencies. To save space, we omit most dependencies on the utilities package Teuchos.

coordinates in the billions, both bisection and coordinate migration during partitioning are the partitioner’s scalability bottlenecks. Multijagged, a multisection-based partitioner, reduces the number of levels of recursion and is therefore much more scalable for very large numbers of parts. It also minimizes data migration by choosing to migrate only when it will help the partitioner’s scalability. Deveci et al. [18] describe the algorithm in detail and present performance of multijagged with RCB on various data sets that show multijagged to be a scalable alternative to RCB.

### 3. Sierra/Nalu Application Code

SIERRA low Mach module/Nalu is a generalized unstructured, massively parallel, low Mach number variable density turbulent flow application code. This code base began as an effort to prototype Sierra Toolkit [19] usage along with direct parallel matrix assembly to the Trilinos Epetra and Tpetra data structures [1].

6 *Parallel Processing Letters*

This generalized unstructured code base supports both elemental (control volume finite element) and edge (edge-based vertex centered) discretizations in the context of an approximate pressure projection algorithm (equal order interpolation using residual-based pressure stabilization). The generalized unstructured algorithm is second order accurate in space and time. Low dissipation advection schemes are supported in addition to second and third order upwind variants where the use of projected nodal gradients is noted. A variety of turbulence models are supported, however, all are classified under the class of modeling known as Large Eddy Simulation (LES). The chosen coupling approach (pressure projection, operator split) results in a set of fully implicit sparse matrix systems. Linear solves are supported by the Trilinos Tpetra interface.

Detailed descriptions of the governing equations are provided in the Nalu theory manual [20], but we provide a very brief summary here. The Favre-filtered equation set (shown in integral form) that are used in this study are as follows.

The continuity equation is given by

$$\int \frac{\partial \bar{\rho}}{\partial t} dV + \int \bar{\rho} \tilde{u}_i n_i dS = 0,$$

where  $\bar{\rho}$ ,  $\tilde{u}_i$ ,  $V$ ,  $S$  and  $n_i$  denote mean density, Favre-averaged velocity components, volume, surface and normal vector to the surface, respectively.

The momentum equation is given by

$$\int \frac{\partial \bar{\rho} \tilde{u}_i}{\partial t} dV + \int \bar{\rho} \tilde{u}_i \tilde{u}_j n_j dS + \int \bar{p} n_i dS = \int \bar{\tau}_{ij} n_j dS + \int \tau_{u_i u_j} n_j dS + \int (\bar{\rho} - \rho_o) g_i dV,$$

where  $\bar{p}$ ,  $\bar{\tau}_{ij}$ ,  $\rho_o$ ,  $g_i$  denote mean pressure, mean viscous shear stress tensor, ambient density, and gravity vector and the turbulent stress  $\tau_{u_i u_j}$  is defined as

$$\tau_{u_i u_j} \equiv -\bar{\rho}(\widetilde{u_i u_j} - \tilde{u}_i \tilde{u}_j).$$

The conserved mixture fraction (used to compute state space quantities, e.g., density, viscosity, etc.) is

$$\int \frac{\partial \bar{\rho} \tilde{Z}}{\partial t} dV + \int \bar{\rho} \tilde{u}_j \tilde{Z} n_j dS = - \int \tau_{Z u_j} n_j dS + \int \bar{\rho} D \frac{\partial \tilde{Z}}{\partial x_j} n_j dS.$$

In the above equation,  $\tilde{Z}$  and  $D$  denote Favre-averaged mixture fraction and an effective molecular mass diffusivity, where sub-filter correlations have been neglected in the molecular diffusive flux vector. The turbulent diffusive flux vector is

$$\tau_{Z u_j} \equiv \bar{\rho} (\widetilde{Z u_j} - \tilde{Z} \tilde{u}_j).$$

This sub-filter correlation is modeled by the gradient transport approximation

$$\tau_{Z u_j} \approx -\bar{\rho} D_t \frac{\partial \tilde{Z}}{\partial x_j}.$$

Finally, the one equation turbulent model used for LES closure is

$$\int \frac{\partial \bar{\rho} k^{\text{sgs}}}{\partial t} dV + \int \bar{\rho} k^{\text{sgs}} \tilde{u}_j n_j dS = \int \frac{\mu_t}{\sigma_k} \frac{\partial k^{\text{sgs}}}{\partial x_j} n_j dS + \int (P_k^{\text{sgs}} - D_k^{\text{sgs}}) dV.$$

Above,  $k^{\text{sgs}}$ ,  $\mu_t$ ,  $P_k^{\text{sgs}}$  and  $D_k^{\text{sgs}}$  denote subgrid turbulent kinetic energy, subgrid turbulent eddy viscosity, production of subgrid turbulent kinetic energy, and dissipation of turbulent kinetic energy, respectively.

Nalu supports two discretizations: control volume finite element (CVFEM) and edge-based vertex centered (EBVC). Both are finite volume formulations and considered to be vertex-based schemes. Figure 3 shows how the control volumes (the mesh dual) are constructed about the nodes. For the CVFEM approach, a linear basis is defined. This linear basis is used to interpolate within the element. Gradients of the basis functions are used for diffusion terms. When using CVFEM, the canonical 9-point (2D quadrilateral meshes) and 27-point (3D hexahedral meshes) stencils are recovered. For the EBVC scheme, the dual mesh is used to construct area vectors at edge mid-points and nodal volumes based on the subcontrol volumes. When using EBVC, the canonical five-point (2D quadrilateral meshes) and 7-point (3D hexahedral meshes) stencils are recovered.

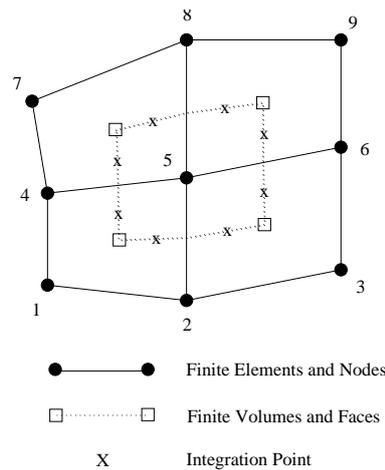


Fig. 3. A control volume centered about a finite-element node.

For the time dependent simulations performed for this work, the BDF2 time integrator [21] was used. For each time step, two nonlinear steps (Picard steps) were performed. A Krylov subspace method iterative solver is employed (e.g. GMRES), preconditioned by symmetric Gauss-Seidel except for the PPE which is preconditioned by an algebraic multigrid preconditioner.

#### 4. Integrating Trilinos in Nalu

Integrating a large application code base with an equally large solver stack brought up several challenges on both sides. We describe some of these and how we overcame

them in this section. They are grouped into two subsections, based on whether the challenges related more to parallel scalability and performance, or to software engineering. However, we found that most cut across both software engineering and performance issues.

#### **4.1. Scalability Challenges from Integration**

##### *4.1.1. Matrix assembly and data structures*

Much of the integration effort revolved around the assembly of discrete finite element matrices and vectors into Trilinos distributed memory data structures. The application, like Trilinos, uses a distributed-memory parallel programming environment. Mesh decomposition, global numbering of unknowns, and “ghosting” are just some of the difficulties with mapping an application to a solver library. We briefly outline our approach below, which we believe applies to a broad range of applications involving discretization of partial differential equations on unstructured meshes.

We construct the assembly process using the following assumptions. First, unknowns are located at nodes. The application knows how to map each node to a unique global identifier or “index.” The unknowns at each node have a simple implicit mapping to a global identifier corresponding to a row in the linear system. Second, the application decomposes the computational domain over MPI processes by elements. Multiple processes may share nodes at process boundaries, but each node has a unique owning process. We call the set of nodes on each process which that process owns its *owned* nodes, and the set of nodes shared by that process, but not owned by it, its *remote* nodes (see Figure 4a). Similarly, each matrix has corresponding owned and remote rows and columns. Third, assembling the linear system requires contributions from neighboring processes for all shared nodes. Finally, nodes and unknowns on each process are numbered consecutively in  $[0, n)$  with a local index. Global indices in  $[0, N)$  are sparse on each process, so conversion from local to global numberings (which uses an array look-up) is faster than conversion from global to local (which uses a hash table look-up). In our approach, Trilinos owns the sparse matrix and dense vector data structures corresponding to the linear systems to solve, and the application owns the assembly process and its corresponding mesh and element data structures.

Trilinos’ matrix storage classes have the option to let users store off-process contributions directly into a matrix. The matrix can then assemble off-process contributions into a consistent view with a single function call that does not require specification of the required interprocess communication. However, this simplicity comes at a price: the function assumes that the communication pattern may change from call to call. *Computing* this pattern itself requires communication, which is more expensive than *executing* the pattern. However, in our application, the mesh changes either rarely or not at all. This means that the communication pattern rarely or never changes. Thus, it was more efficient to implement a custom global assembly approach that reuses a precomputed communication pattern. Trilinos has

more specialized interfaces that let users precompute and reuse a communication pattern from its source and target data distributions.

This approach's communication pattern follows a "push" model. As noted above, shared nodes require contributions from neighboring processes. Only the contributions for each finite element owned on a process are known to the nodes on that process. Thus, a shared node that is owned on a given process cannot construct its full dependency graph without receiving the dependency information from other processes. The communication pattern needed to communicate the dependencies is the same as what is needed to assemble the matrix entries.

Our approach relies on two matrices on each process: one for owned rows and one for remote rows. To construct these matrices, first we construct the dependency graphs. The graph for the remote matrix is entirely locally defined, where the columns of the matrix correspond to the contributions of the local finite elements to the remote nodes. The owned matrix is initially constructed using local data, where the columns are the contributions of the local finite elements to the owned nodes. The rows of the remote graph are communicated to the owning processes. The received dependency data is added to the owned graph, completing the required dependency information. Owned and remote matrices are created from the owned and remote graphs, respectively. Similarly, owned and remote linear system residual vectors are constructed. During assembly, these matrices and vectors are filled on each process, and the remote entries are communicated to the owning processes using the same communication pattern as the graph. This process relies on Trilinos functionality to achieve the goal of efficient assembly, but this procedure of splitting the matrix into local and remote components and communicating the remote portion to owning process matrix contributions is general and would apply if one is building fully assembled matrix representations.

In other words, the local representation of the owned matrix,  $O$ , which has  $n$  rows and  $m$  total nonzero columns on a given process,  $p$ , is defined as

$$O_p = \left( \begin{array}{ccc|ccc} a_{0,0} & \cdots & a_{0,n-1} & a_{0,n} & \cdots & a_{0,n+r-1} & a_{0,n+r} & \cdots & a_{0,m-1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & \cdots & a_{n-1,n-1} & a_{n-1,n} & \cdots & a_{n-1,n+r-1} & a_{n-1,n+r} & \cdots & a_{n-1,m-1} \end{array} \right),$$

where  $r$  is the number of remote rows and the subscripts on the entries,  $a$ , indicate the local row and column ids. The first submatrix is all locally owned data, the second is all data that is available on process, and the third submatrix is filled in with remote communication. The remote matrix,  $R$ , on each process is defined as

$$R_p = \left( \begin{array}{ccc|ccc} a_{n,0} & \cdots & a_{n,n-1} & a_{n,n} & \cdots & a_{n,n+r-1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n+r-1,0} & \cdots & a_{n+r-1,n-1} & a_{n+r-1,n} & \cdots & a_{n+r-1,n+r-1} \end{array} \right).$$

Note that only data available on the process is added to the remote matrix.

Assembly of these two sub-matrices involves three sets of identifiers: the global index of the unknown, its process-local index, and the row in each of the matrices. As

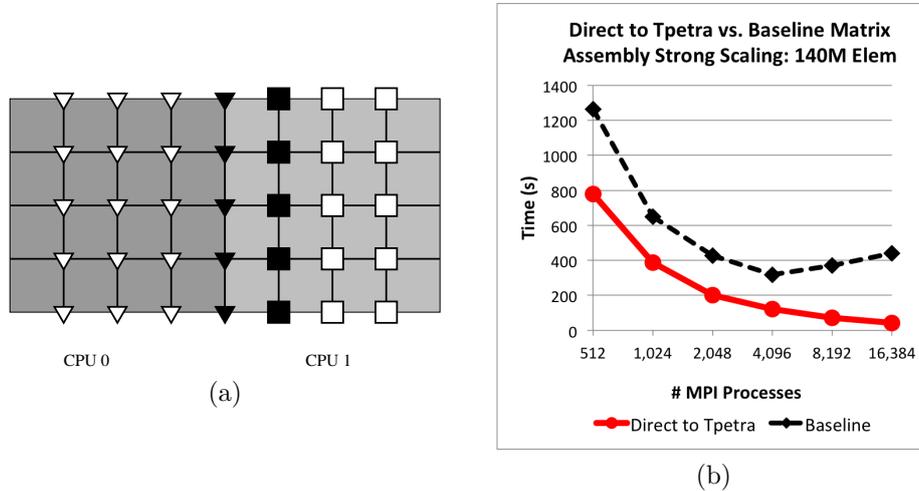


Fig. 4. (a) Illustration of node ownership, triangle nodes are all owned by CPU rank 0, square nodes owned by CPU rank 1. Solid black triangle nodes are remote nodes of CPU rank 1. Solid black square nodes are contributors for owned nodes of CPU 0 but not actually remote nodes of CPU 0. (b) Comparison of matrix assembly: baseline compared with direct to Tpetra with additional performance optimizations. Latter is faster and scales considerably better: *over a factor of ten reduction in assembly time at 16k cores.*

mentioned above, using global indices is substantially slower than local indices, due to global to local conversion. Although determining the local index is fast, it does not directly correspond to a row in the local or remote matrices. Therefore, we created a mapping between each local identifier and each row in the matrix. The mapping is made fast by constructing a local ordering where the first sequence of values corresponds to the locally owned unknowns, and the remaining values correspond to unknown contributions on remote processes. This ordering allows one to assemble the matrices using only local indices. This approach adds a level of indirection to the assembly process, and thus a level of complexity. However, the combination of these two processes, split local and remote matrices and local indexing, greatly improves performance in the assembly process. Figure 4b presents a comparison of the matrix assembly for the baseline approach that uses the SIERRA framework and FEI, versus the direct assembly to Tpetra approach with the two performance optimizations described above. The latter approach is consistently faster and scales significantly better, providing *over a factor of ten reduction in assembly time at 16k cores.* Ideally this approach can be generalized and added to the Trilinos libraries.

#### 4.1.2. Multigrid scaling

The MueLu multigrid library utilizes a variant of an algebraic multigrid method called *smoothed aggregation* [22]. This algorithm constructs coarser levels by transforming a group of fine level nodes (an *aggregate*) into a single coarse level node.

It does this in an uncoupled fashion, where every parallel MPI process owns a subdomain, and aggregates are prohibited from crossing subdomain boundaries. In the following, a transfer operator from a coarse to a fine level is called a *prolongator*, while the transfer operator from a fine to a coarse level is called a *restrictor*.

Simulations of real problems at large scales allowed us to identify scaling bottlenecks which did not manifest at smaller scales, and to optimize several kernels. We provide a few examples as follows.

**Linear algebra kernels** Smoothed aggregation algebraic multigrid constructs the prolongator in several steps. First, it forms a filtered matrix  $A_f$  from the original matrix  $A$ , by identifying strongly connected degrees of freedom. Next, it forms a tentative prolongator  $P_{tent}$  via aggregation based on the filtered matrix. Finally, it smooths  $P_{tent}$  via a (damped) Jacobi iteration to form the final prolongator  $P$ ,

$$P = \left( I - \omega_f D_f^{-1} A_f \right) P_{tent} , \quad (1)$$

where  $D_f$  is the diagonal of  $A_f$ , and  $\omega_f = \frac{4}{3\rho(D_f^{-1}A_f)}$  uses an estimate of the spectral radius of  $D_f^{-1}A_f$ . We describe three optimizations to improve scalability: optimizing the construction of  $A_f$ , using a cheaper approximation of  $\omega_f$ , and optimizing the Jacobi kernel.

We originally built the matrix  $A_f$  (more specifically, its graph) by constructing a new matrix containing only unfiltered entries. This approach was shown to scale poorly at large processor counts. We replaced it by zeroing out filtered entries in a copy of the original matrix  $A$ , and modifying the sparse matrix-matrix multiplication to ignore zero entries. This approach avoided multiple rounds of communication associated with a new matrix construction. This improvement is called “improved construction of filtered matrix” in Section 4.1.3.

Our original approach to calculate  $\omega_f$  required an estimate for the largest eigenvalue of  $A_f$ , which was expensive as it required multiple rounds of matrix-vector multiplications. We replaced  $\omega_f$  with an  $\omega$  value based on the original matrix  $A$ ,  $\omega = \frac{4}{3\rho(D^{-1}A)}$ , where  $D$  is the diagonal of  $A$ .  $\rho(D^{-1}A)$  had previously been computed during the setup for the Chebyshev smoother, so reusing this value avoided additional matrix-vector products. This improvement is called “eigenvalue reuse” in Section 4.1.3.

Prolongator smoothing (Equation 1) traditionally takes three steps: first, a matrix-matrix multiplication  $B = A_f P_{tent}$ , then a diagonal scaling  $C = \omega D_f^{-1} B$ , and finally, a matrix-matrix addition:  $P = C + P_{tent}$ . This maximizes software reuse of existing computational kernels, but requires multiple passes over the data and rounds of communication. We implemented an improved approach, which uses a fused kernel for the entire prolongator smoothing operation (Equation 1). We begin by assuming that all matrices are distributed by row and the row distributions of  $A_f$  and  $P_{tent}$  are identical to the row distribution desired for  $P$ . We also assume that all entries on the diagonal of  $A_f$  are non-zero (or else  $D_f^{-1}$  would be unde-

finer). These assumptions constrain the sparsity pattern (ignoring zeros introduced by numerical cancellation) of  $B = A_f P_{tent}$  to contain the sparsity pattern of  $P_{tent}$ . Thus, the parallel communication structure of  $P$  is identical to that of  $B$  (assuming that zeros introduced by numerical cancellation are not dropped), and we can use that communication structure for  $P$ . The fused prolongator smoothing kernel can duplicate the structure of the matrix-matrix multiplication kernel, where the only modifications occur in the serial component.

**Rebalancing kernels** Each consecutive coarsening during aggregation reduces the number of nodes per subdomain. Once this number gets too small, communication starts to dominate computation. At this point, we use either Zoltan’s or Zoltan2’s geometric partitioning to rebalance the problem to a smaller number of processors. This reduces the amount of communication.

Initial comparisons showed a performance degradation between MueLu with Zoltan and MueLu with Zoltan2. The main cause was a specific algorithm, called *remapping*, present in Zoltan but absent in Zoltan2. Remapping reduces the amount of migrated data by maximizing the overlap of the original matrix with the rebalanced one. It improves the communication pattern, which reduces the preconditioner apply time. The effect of remapping for small problems was not large enough to be included in the original design, but its impact was significant at scale. The MueLu and Zoltan2 teams consequently implemented a remapping procedure similar to Zoltan’s remapping. For the 140 million element test case on 2048 cores, discussed in Section 4.1.3, the PPE solve time (not including setup) for MueLu/Zoltan2 without vs. with remapping is 181 resp. 143 seconds. Lack of remapping increased solve time by 27%; effects are even more pronounced at scale.

Despite significant improvements, data migration remained an expensive operation, particularly during setup. The original approach was to migrate three matrices: coarse matrix, prolongator and restrictor. We improved performance by replacing migration of the prolongator and restrictor during the setup phase by a significantly cheaper migration during the solve phase, which required us to migrate only vectors. This improvement is called “implicit matrix rebalancing” in Section 4.1.3.

#### 4.1.3. *Tracking performance over time*

Although the goal for the new code is to enable higher fidelity simulations, we had to do so without sacrificing the production code’s performance for smaller problems. To achieve this goal, it was essential to track performance comparisons between the first- and second-generation solver stacks (Table 1). We used one instance of the test case described in the results section (Section 5), a 140 million element problem run on 2048 processors of the Cielo Cray XE6 sited at Los Alamos National Laboratory. The system consists of 8944 compute nodes (dual-socket 2.4GHz 8-core AMD Magny-Cours, total 142,000 cores) with a Cray Gemini 3D torus interconnect.

Figure 5 tracks performance over time for the new Tpetra-based Nalu code com-

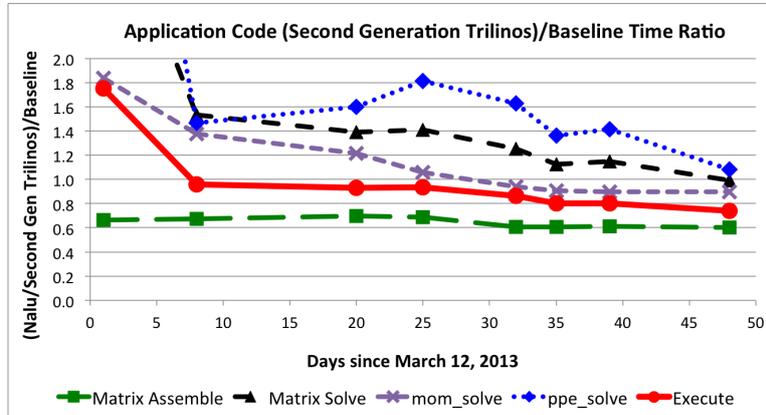


Fig. 5. Tracking performance over time, comparing new Tpetra-based Nalu with baseline. Vertical axis: ratio of the times of new Tpetra-based Nalu code to baseline code. Ratio  $< 1$  means new code is faster. By the end, execute and solve times are 26% and 1% faster respectively than baseline.

pared with the baseline Epetra-based code that employed the SIERRA framework and FEI, for the edge-based scheme (ratio between new and baseline was greater for edge-based than element-based method). Vertical axis plots the time ratio (ratio  $< 1$  means the new code is faster than the baseline). “Execute” time is the wall time (sans I/O time) of the simulation. “Matrix assemble” time is the time to construct the matrices. The “matrix solve” reported here sums the time for the five equations (momentum equations, PPE, and mixture fraction). We solved the momentum and mixture fraction equations with symmetric Gauss-Seidel preconditioned GMRES, and solved the PPE with multigrid-preconditioned GMRES. We set up the multigrid preconditioner once in MueLu and reuse it for all time steps.

The horizontal axis plots days since March 12, 2013. On March 13, only the matrix assembly was faster than the baseline (as a result of the improvements described in Section 4.1.1); the solve time was slower. By March 21, the “execute” time was faster for the new vs. baseline codes. By the end (April 29 code base), execute time for the new code was 26% faster than the baseline, and the “matrix solve time” was 1% faster than the baseline. Note that sometimes the performance regressed. This demonstrates the value of carefully tracking performance.

For the study presented in Figure 5, multigrid preconditioner setup was performed only once and reused for the entire simulation. Other test cases (e.g. adaptive mesh) may require the multigrid preconditioner setup to be performed every nonlinear step, so we provide a comparison of the multigrid setup and solve times between MueLu and ML. ML has previously demonstrated good scaling for solve time for over 100,000 cores in a different application code [23]. Figure 6 tracks performance over time (days since Nov 24, 2013) for multigrid setup and solve comparing MueLu with baseline ML for both edge-based and element-based methods. For the rest of this paper, we differentiate between multigrid “setup” and “solve,”

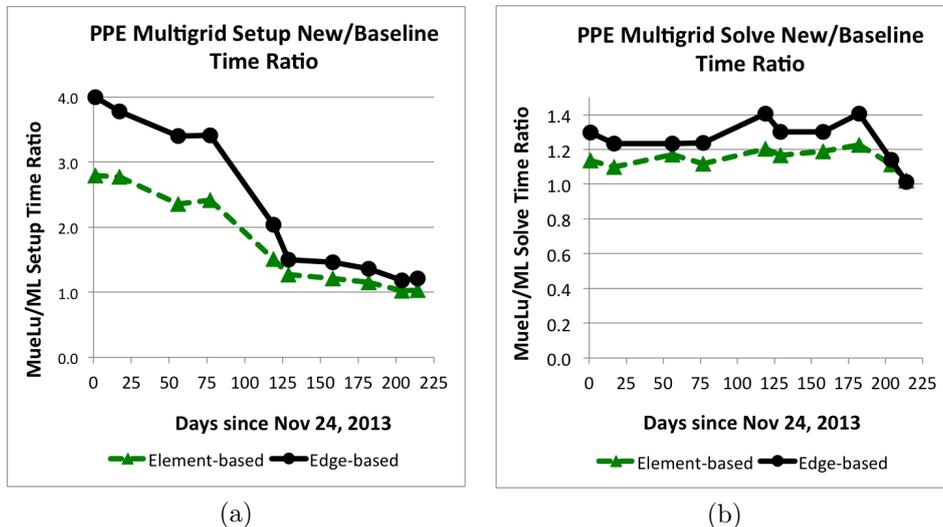


Fig. 6. Performance over time comparing new MueLu with baseline ML for both edge-based and element-based methods. Vertical axis: ratio of MueLu to ML times; ratio  $< 1$  means MueLu is faster. (a) multigrid setup (b) multigrid solve (does not include setup). By the end of the time period, MueLu is competitive with ML, with MueLu setup for edge-based and element-based methods being within 21% and 3% respectively of ML and MueLu solve being within 2% of ML.

where the latter does not include setup time. The vertical axis plots the ratio of the wall times of MueLu to ML (ratio  $< 1$  means MueLu is faster). By the end, MueLu is competitive with ML, with MueLu setup for edge-based and element-based methods being within 21% and 3% respectively of ML and MueLu solve being within 2% of ML. We correlated improvements in time with improvements in algorithms. For example, the substantial improvements to MueLu setup between days 77 and 119 (for edge-based method, MueLu/ML dropped from  $3.4\times$  to  $2.0\times$ ) and between days 119 and 129 (for edge-based method, MueLu/ML dropped from  $2.0\times$  to  $1.5\times$ ) were primarily due to implicit matrix rebalancing, improved construction of filtered matrix, and eigenvalue reuse (first drop), and fused prolongator smoothing operation for Equation 1 (second drop). These improvements were described in detail in Section 4.1.2. The substantial improvements to MueLu solve between days 182 and 204 (for edge-based, MueLu/ML dropped from  $1.41\times$  to  $1.14\times$ ) and between days 204 and 214 (for edge-based, MueLu/ML dropped from  $1.14\times$  to  $1.01\times$ ) were due to a fix to a filtered matrix construction (first drop), and an improved aggregation procedure (second drop). For the case where multigrid setup needs to be performed every nonlinear step, the PPE setup and solve times are roughly the same (for element-based, setup time is roughly 40% of setup plus solve time).

Recall that the purpose of comparing MueLu with ML for the 140 million element test case was to demonstrate that little performance is being lost for small to medium-sized test cases by moving from Epetra/ML to Tpetra/MueLu. For larger problems, MueLu's setup time does not scale as well as ML's. MueLu has a differ-

ent relationship with its computational kernels than ML. ML is self-contained, while MueLu depends more on other Trilinos packages. In particular, a key MueLu setup kernel – sparse matrix-matrix multiply – is implemented in Tpetra, while ML implemented its own. Improving MueLu’s setup time will require improvements to both MueLu and Tpetra. However, Tpetra/MueLu offers substantially new capability compared with ML.

## 4.2. Software Engineering Challenges

### 4.2.1. Challenges and risks of integration

Our success relied on the integration of two independently developed codes, Sierra and Trilinos. This was not a one-time stunt, but a path towards use of the second-generation Trilinos in Sierra’s production runs. Sierra has around  $10^6$  lines of code, and the relevant parts of Trilinos have about  $10^5$  lines of code. Both code bases have over 100 developers constantly making changes. Furthermore, the two teams have entirely different work cultures. All these factors increase the challenge of integrating the two code bases. Historical examples of failure to meet this challenge often result from differences in work culture between contributing teams, or from ignorance of the potential risk [24, 25].

### 4.2.2. Continuous integration

Our key to success was continuous integration between solvers and the application. This began by nightly tests of Trilinos in Sierra, which was considerably more challenging than one might expect. Just ensuring that Sierra had the latest version of Trilinos took much of a full-time Trilinos developer’s effort. Beyond this first step, continuous performance tests helped us identify and correct performance problems in Trilinos that only showed up in a real application. Trilinos already has many “drivers” that exercise scalability and performance, including a Poisson driver for MueLu. The drivers did help us identify and fix performance problems. However, since they did not fully represent the linear systems generated by the application code, the drivers would sometimes predict larger performance improvements than Sierra could actually achieve, and did not expose the importance of load balancing. Our most important metric of success is performance comparisons between the new and baseline application codes, as shown in Section 4.1.3. These comparisons continue to this day, and still drive performance improvements. In our experience, the key to real impact is a combination of continuous integration of the code bases, unit tests, scaling tests with drivers, and continuously tracking performance with real applications.

## 5. Results

The new Trilinos is intended to enable very large scale simulations; we demonstrate this capability in Nalu. Previously we presented scaling results to 131,072 cores

on a Cray XE6 architecture [5]. Here, we present scaling studies to 524,288 cores on an IBM Blue Gene/Q platform (Lawrence Livermore National Laboratories Sequoia). The test case (Figure 7a) is a mixture fraction-based turbulent open jet with Reynolds number 6600 [26]. The jet emanates from a pedestal from the bottom of the cylindrical computational domain, which is discretized with unstructured hexahedral elements. Nalu solves six equations: PPE, coupled momentum system for three component directions, mixture fraction, and subgrid-scale kinetic energy. The majority of the total run time of Nalu can be split into two main portions: the matrix assembly phase and the linear solve phase. As they are substantially different, scaling of each phase will be treated separately. MueLu preconditioner setup for the PPE is performed once and reused for the entire simulation.

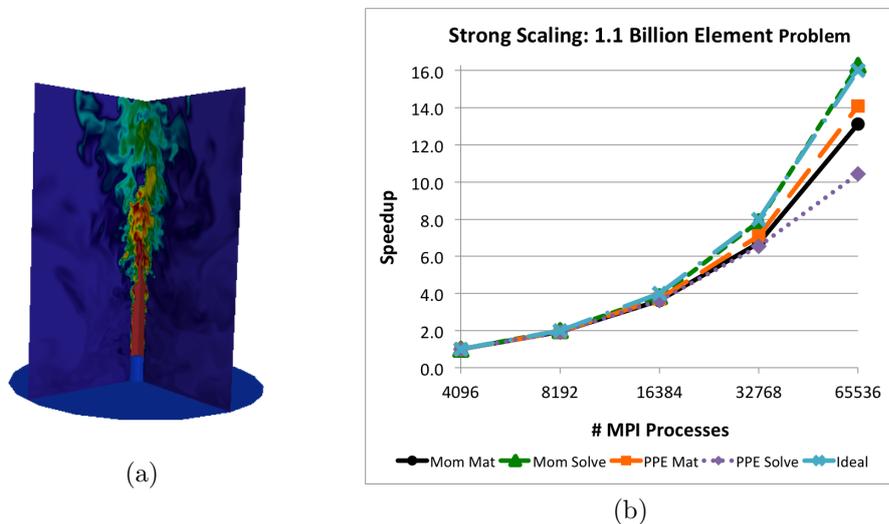


Fig. 7. (a) Test case is a mixture fraction-based turbulent open jet ( $Re=6600$ ). Jet emanates from pedestal at bottom of domain. (b) Strong scaling for element-based method for a 1.12 billion element mesh. Obtained ideal scaling for momentum solve time while assemble time scales well.

Figure 7(b) presents a strong scaling study for coupled momentum and PPE for the 1.12 billion element case as processor cores increases from 4k to 64k. Ideal speedup as plotted is a curved line because the vertical axis is plotted on a linear scale rather than a logarithmic scale. In Figure 7(b), the fourth curve is the matrix solve time for PPE (not including the multigrid setup time). For both coupled momentum and PPE, the scaling for the matrix assembly (local plus global assembly) is very good. For coupled momentum, scaling of the matrix solve time is optimal. For PPE, the matrix solve time is very good.

Figure 8 presents weak scaling for matrix assembly and linear system solve for the coupled momentum system for both the edge-based and element-based methods. The vertical axis is parallel efficiency and the horizontal axis is the MPI process

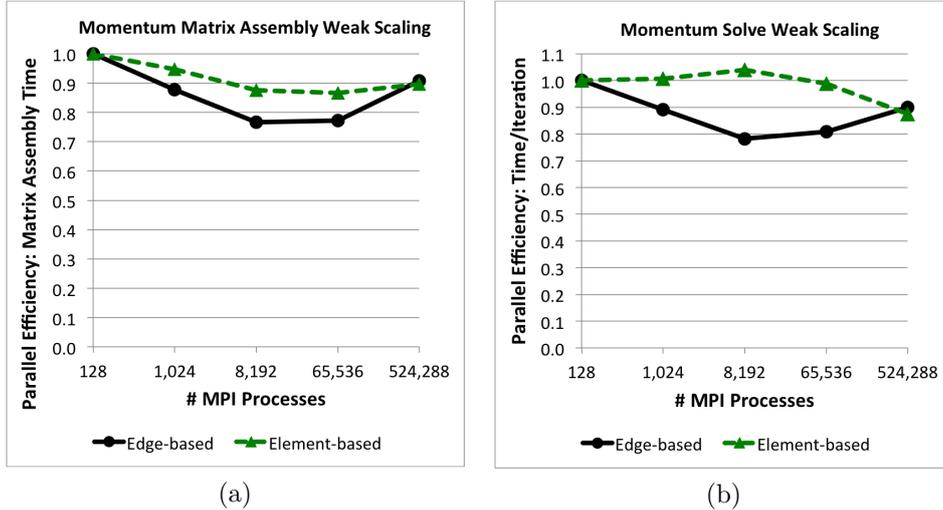


Fig. 8. Weak scaling for coupled momentum: (a) Matrix assembly time. (b) Solve time per Krylov iteration (Symmetric Gauss-Seidel preconditioned TFQMR). Very good scaling as increase in problem size is 4096x and largest problem has 27 billion row matrix.

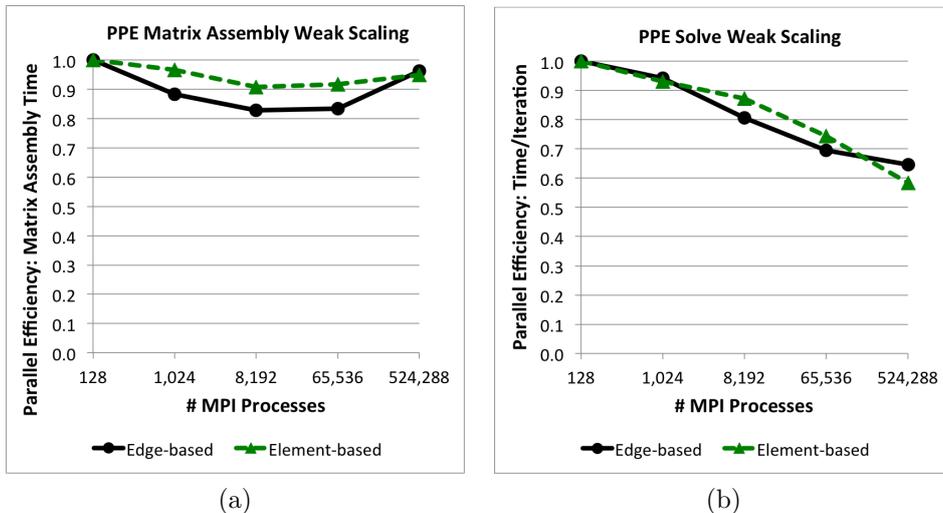


Fig. 9. Weak scaling for PPE: (a) Matrix assembly time. (b) Solve time per Krylov iteration (multigrid preconditioned TFQMR; not including multigrid setup). Very good scaling as increase in problem size is 4096x and largest problem has 9 billion row matrix run on 524,288 cores.

count. Each processor core has 17,000 elements. The largest problem has 9 billion elements, which produces a coupled momentum system matrix with 27 billion rows, on 524,288 MPI processes. The matrix assembly time includes both the time for local assembly as well as global assembly. The linear solve time is the time per TFQMR Krylov iteration. Parallel efficiency is still high even with 524,288 MPI

processes. Both the edge-based and element-based methods scale well. <sup>a</sup>

Figure 9 presents weak scaling for matrix assembly time and solve time for the PPE for both the edge-based and element-based methods. The vertical axis is parallel efficiency and the horizontal axis is the MPI process count. The largest problem has 9 billion elements, which produces a PPE matrix with 9 billion rows, on 524,288 MPI processes. Note that the problem is scaled by a factor of 4096 $\times$ . The linear solve time is the time per TFQMR Krylov iteration and does not include multigrid preconditioner setup time. Parallel efficiency is good even with 524,288 MPI processes. Currently, the preconditioner setup has suboptimal scalability. Implementing scalable algebraic multigrid setup is extremely challenging, because of the creation of the hierarchy of levels and generation of the coarser level matrices with a triple matrix product and the matrix-matrix product for the prolongator smoothing operation (Equation 1). A substantial effort is underway by the MueLu and Tpetra teams to improve setup scaling by comparing MueLu and ML for large scales (ML is limited to 2 billion rows). Efforts focus on optimizing Tpetra's matrix-matrix multiply and communication avoiding approaches.

## 6. Conclusions

We presented the second-generation Trilinos solver stack and described the integration process with the SIERRA low Mach module/Nalu application code. This study highlighted the value of tracking performance during the integration process. We showed good weak scaling for the matrix assembly and solve in Nalu, for up to a 9 billion element fluid flow large eddy simulation (LES) problem on unstructured meshes with a 27 billion row matrix on 524,288 cores of an IBM Blue Gene/Q platform. The main remaining scaling bottleneck is multigrid setup (mainly sparse matrix-matrix multiplication); ongoing work will address this issue.

The second-generation Trilinos solver stack allows arbitrarily large global entities and provides a path forward for future computing architectures. Its integration into Nalu required significant revisions, which fixed correctness, per-process performance, and parallel scalability issues. All resulting new features and fixes in Trilinos will prove useful for a wide variety of other application codes, not just Nalu. Furthermore, best practices we learned for improving Nalu's finite-element assembly performance will apply generally to any finite-element application with implicit solves using Trilinos. We continue to mature the new Trilinos, especially for current and upcoming manycore architectures.

<sup>a</sup>Recall that the element-based method employs a 27-point stencil while the edge-based method employs a 7-point stencil, so the linear system for the element-based method has roughly four times the number of nonzeros per matrix row as the edge-based method. For the case of unstructured meshes of decent quality, the edge-based method can therefore be substantially faster. However, for generalized unstructured meshes of poor quality, the element-based approach has superior robustness and accuracy [20], as well as a more favorable computation/communication ratio.

## Acknowledgment

We thank the following colleagues for their contributions: Ryan Bond, Kevin Copps, Eric Cyr, Mehmet Deveci, Karen Devine, Jeremie Gaidamour, Micheal Glass, Michael Heroux, Robert Hoekstra, Kyran Mish, Brent Perschbacher, Kendall Pierson, Jim Willenbring, and Alan Williams. The Lawrence Livermore National Laboratory (LLNL) Sequoia team lent us invaluable assistance (John Gyllenhaal and Scott Futral deserve special mention). Finally, our thanks to friends from other institutions – Jenny Gable Brown, Anthony Stuckey, Jimmy Su, Jon Kuroda, Bryan Catanzaro, Matt Knepley, and Piotr Luszczek – who contributed historical examples of unsuccessful software integration projects.

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

## References

- [1] M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, A. Williams, and K. Stanley. An overview of the Trilinos project. *ACM Transactions on Mathematical Software*, 31(3):397–423, 2005.
- [2] S. Domino, C. Moen, S. Burns, and G. Evans. SIERRA/Fuego: A multi-mechanics fire environment simulation tool. In *41st Aerospace Sciences Meeting and Exhibit, AIAA 2003-149*, 2003.
- [3] H. C. Edwards and J. Stewart. Sierra: A software environment for developing complex multiphysics applications. In K. Bathe, editor, *1st MIT Conference on Computational Fluid and Solid Mechanics*. Elsevier, 2001.
- [4] H. C. Edwards, D. Sunderland, V. Porter, C. Amsler, and S. Mish. Manycore performance portability: Kokkos multidimensional array library. *Scientific Programming*, 20(2):89–114, 2012.
- [5] P. Lin, M. Bettencourt, S. Domino, T. Fisher, M. Hoemmen, J. Hu, E. Phipps, A. Prokopenko, S. Rajamanickam, C. Siefert, E. Cyr, and S. Kennon. Towards extreme scale simulation with next-generation Trilinos: a low Mach fluid application case study. In *Workshop on Large-Scale Parallel Processing (LSPP) 2014, in conjunction with IPDPS2014*, 2014.
- [6] E. Hawkes, O. Chatakonda, H. Kolla, A. Kerstein, and J. Chen. A petascale direct numerical simulation study of the modelling of flame wrinkling for large-eddy simulations in intense turbulence. *Combustion and Flame*, 159(8):2690–2703, 2012.
- [7] V. Moureau, P. Domingo, and L. Vervisch. From large-eddy simulation to direct numerical simulation of a lean premixed swirl flame: Filtered laminar flame-PDF modeling. *Combustion and Flame*, 158(7):1340–1357, 2011.
- [8] J. Schmidt, M. Berzins, J. Thornock, T. Saad, and J. Sutherland. Large scale parallel solution of incompressible flow problems using Uintah and Hypre. Technical Report UUSCI-2012-002, University of Utah, 2012.
- [9] C. Baker and M. Heroux. Tpetra and the use of generic programming in scientific computing. *Scientific Programming*, 20(2):115–128, 2012.
- [10] R. Tuminaro, M. Heroux, S. Hutchinson, and J. Shadid. Aztec user's guide–version 2.1. Technical Report SAND99-8801J, Sandia National Laboratories, 1999.
- [11] E. Bavier, M. Hoemmen, S. Rajamanickam, and H. Thornquist. Amesos2 and Belos:

- Direct and iterative solvers for large sparse linear systems. *Scientific Programming*, 20(3):241–255, 2012.
- [12] M. Gee, C. Siefert, J. Hu, R. Tuminaro, and M. Sala. ML 5.0 smoothed aggregation user's guide. Technical Report SAND2006-2649, Sandia National Laboratories, 2006.
- [13] J. Gaidamour, J. Hu, C. Siefert, and R. Tuminaro. Design considerations for a flexible multigrid preconditioning library. *Scientific Programming*, 20(3):223–239, 2012.
- [14] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.
- [15] E. Boman, K. Devine, V. Leung, S. Rajamanickam, L. A. Riesen, M. Deveci, and Ü. Çatalyürek. Zoltan2: Next-generation combinatorial toolkit. Technical report, Sandia National Laboratories, 2012.
- [16] Y. Saad and M. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, July 1986.
- [17] R. Freund. A transpose-free quasi-minimal residual algorithm for non-hermitian linear systems. *SIAM Journal on Scientific Computing*, 14(2):470–482, 1993.
- [18] M. Deveci, S. Rajamanickam, K. Devine, and Ü. Çatalyürek. Multi-jagged: A scalable parallel spatial partitioning algorithm. *IEEE Transactions on Parallel and Distributed Systems (In revision)*, 2014.
- [19] H. C. Edwards, A. Williams, G. Sjaardema, D. Baur, and W. Cochran. Toolkit computational mesh conceptual model. Technical Report SAND2010-1192, Sandia National Laboratories, 2010.
- [20] S. Domino. Low Mach Sierra thermal/fluids module Nalu: theory manual. Technical report, Sandia National Laboratories internal document, 2014.
- [21] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, volume 14 of *Springer Series in Computational Mathematics*. Springer, Berlin, 1991.
- [22] P. Vaněk, J. Mandel, and M. Brezina. Algebraic multigrid based on smoothed aggregation for second and fourth order problems. *Computing*, 56:179–196, 1996.
- [23] P Lin. Improving Multigrid Performance for Unstructured Mesh Drift-Diffusion Simulations on 147,000 cores. *International Journal for Numerical Methods in Engineering*, 91:971–989, 2012.
- [24] N. Leveson and C. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.
- [25] I. Perepu and V. Gupta. ERP implementation failure at Hershey Foods Corporation. Technical Report 908-001-1, ICFAI Center for Management Research, 2008.
- [26] A. Abdel-Rahman, W. Chakroun, and S. Al-Fahed. LDA measurements in the turbulent round jet. *Mechanics Research Communications*, 24(3):277–288, 1997.