

Towards extreme-scale simulations with next-generation Trilinos: a low Mach fluid application case study

Paul Lin, Matthew Bettencourt, Stefan Domino, Travis Fisher, Mark Hoemmen,
Jonathan Hu, Eric Phipps, Andrey Prokopenko, Sivasankaran Rajamanickam, Christopher Siefert,
Eric Cyr, Stephen Kennon
Sandia National Laboratories
Albuquerque, NM USA
ptlin@sandia.gov

Abstract—Trilinos is an object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems. While the original version of Trilinos was designed for highly scalable solutions for large problems, the need for increasingly higher fidelity simulations has pushed the problem sizes beyond what could have been envisioned two decades ago. When problem sizes exceed a billion elements even highly scalable applications and solver stacks require a complete revision. The next-generation Trilinos employs C++ templates in order to solve arbitrarily large problems and enable extreme-scale simulations. We present a case study that involves integration of Trilinos with an engineering application (Sierra low Mach module/Nalu), involving the simulation of low Mach fluid flow for problems of size up to nine billion elements. Through the use of improved algorithms and better software engineering practices, we demonstrate good weak scaling for the matrix assembly and solve for the engineering application for up to a nine billion element fluid flow large eddy simulation (LES) problem on unstructured meshes with a 27 billion row matrix on 131,072 cores of a Cray XE6 platform.

Keywords—Solver library, Trilinos, Vertical integration, Extreme-scale simulations

I. INTRODUCTION

Trilinos is an object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems [1]. While the original version of Trilinos was designed for highly scalable solutions for large problems, the need for increasingly higher fidelity simulations has pushed problem sizes beyond

what one could have envisioned two decades ago. Therefore, there is a need for a revision of Trilinos that both supports arbitrarily large problem sizes and provides a path forward for good performance on future architectures.

Many challenging scientific and engineering problems at Sandia National Laboratories (SNL) require ever increasing fidelity for computational simulations. One example is the accurate simulation of fire environment, for example, large scale hydrocarbon pool fires that occur in accident scenarios. Figure 1a depicts an example hydrocarbon JP-8 pool fire experiment and Figure 1b depicts a numerical simulation.

SNL uses the SIERRA/Fuego [2] engineering application, built on top of the SIERRA Framework [3], to simulate the fire environment. SIERRA/Fuego is a low-Mach number, turbulent reacting flow code. It uses Trilinos' solvers through the Finite Element Interface (FEI). The fire environment scenarios have required increasingly higher fidelity, but the choice of 32-bit ordinals for the entire stack from the applications to solvers (SIERRA Framework, FEI, and Trilinos) limited simulations to less than about two billion entities (e.g., nodes, elements, edges, matrix rows). This severely limits the fidelity of current and especially future simulations, and motivated the development of a new application code.

This paper describes the next-generation Trilinos and presents a case study of 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 paper focuses on next-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. Please refer to [4] for Kokkos’ design and path forward.

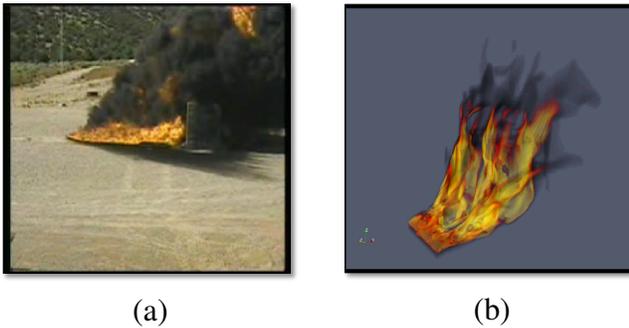


Figure 1. Hydrocarbon JP-8 pool fire (a) experiment and (b) numerical simulation

Our primary contributions in this paper are:

- Significant revision of Trilinos’ new solver stack, focused on performance and scalability.
- Demonstrated the capability and scalability of next-generation Trilinos with a new application SIERRA/Nalu for *9 billion nodes on 131,072 cores*: the largest such simulation on unstructured meshes (in terms of node count) to our knowledge.
- Case study describing issues and their solutions faced in integrating a large solver stack with an application, and tracking performance improvements over time.

We also discuss best practices for future applications that will employ the new Trilinos, e.g., direct matrix assembly into Trilinos data structures.

For perspective on how the size of the simulations enabled by the new Nalu code compare 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 Sim-

ulation (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. [5] have shown 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 [6] 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 Hypr algebraic multigrid preconditioner. Their simulations have reached 256k cores on meshes with about 6 billion structured hexahedral elements [7]. Based on our survey of leaders in the fully implicit low Mach field, for unstructured meshes we believe that our 9 billion node simulation on 131,072 cores rivals any other in the field.

We first summarize Trilinos and Nalu in Sections II and III, then discuss their integration in Section IV. Section V presents results showing the new capability provided by the next-generation Trilinos, and Section VI concludes and outlines future work.

II. TRILINOS

A. Trilinos Overview

Trilinos [1] is an open-source software project to develop and implement robust algorithms and enabling technologies for solving large-scale mathematical problems from scientific and engineering applications. It focuses on the core requirement of these applications: solving large linear and nonlinear systems of equations, eigensystems and other related problems, possibly in parallel. Supported capabilities include distributed-memory parallel linear algebra, partial differential equation discretizations, parallel partitioning for load balance, incomplete factorizations and relaxations, multilevel preconditioners (such as

algebraic multigrid), solvers for linear systems with successive or simultaneous multiple right-hand sides, block iterative eigensolvers, nonlinear methods including continuation, time integrators, large-scale nonlinear optimization, and automatic differentiation.

Trilinos is based and hosted at Sandia National Laboratories, but includes significant external partnerships and contributions. Most Trilinos packages have a modified BSD license; a few have the GNU Lesser General Public License. The library grew out of a group of established algorithms efforts at Sandia. It was motivated by a recognition that a modest degree of coordination among these efforts could have a large positive impact on the quality and usability of the software we produce, and therefore enhance the research, development and integration of new algorithms and enabling technologies into applications. Because of the tools and infrastructure that Trilinos provides, the degree of effort required to develop new algorithms and enabling technologies has been substantially reduced because our common base provides an excellent starting point. Furthermore, many applications are standardizing on Trilinos’ interfaces. As a result, these applications have access to all Trilinos solver components without any unnecessary interface modifications.

B. Trilinos packages used by Nalu

This paper mentions two different Trilinos solver stacks: the “current one,” and the “new” or “next-generation” stack. The new stack is a complete rewrite of the old stack, in order to support the following new capabilities:

- Solve problems with more entities than can fit in a 32-bit integer (> 2 billion), without *requiring* use of 64-bit integers (higher memory bandwidth and storage cost) throughout
- Allow use of data types other than double-precision floating-point, e.g., complex or extended precision
- Enable and simplify the ongoing addition of “hybrid” MPI + shared-memory parallelism for many programming models (e.g., OpenMP and CUDA)

Functionality	Current	New
Distributed linear algebra	Epetra [1]	Tpetra [8]
Iterative linear solvers	AztecOO [9]	Belos [10]
Incomplete factorizations	AztecOO, Ifpack	Ifpack2
Algebraic multigrid	ML [11]	MueLu [12]
Partition & load balance	Zoltan [13]	Zoltan2[14]
Direct solvers interface	Amesos	Amesos2 [10]

Table I
COMPARISON OF SIX CURRENT VS. NEW TRILINOS SOLVER
STACK PACKAGES

The new stack uses C++ templates to implement these new capabilities. Table I summarizes both stacks. This paper demonstrates the new stack’s scalability in e.g., Figure 3, especially when they are all integrated together and used in Nalu. Each one of these packages had to undergo significant changes to demonstrate scalability at this scale. The dependency diagram in Figure II-B shows the complex dependencies in these packages.

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’ [10] implementation of the GMRES iteration [15]. Solves for all equations other than the pressure Poisson equation (PPE) were preconditioned using Ifpack2’s symmetric Gauss-Seidel. For the PPE, Nalu used MueLu’s [12] 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 [10]. To improve efficiency of the multigrid preconditioner, a rebalance was performed on the coarser levels of the matrix using Zoltan2’s multijagged algorithm [14]. MueLu explicitly forms coarser-level matrices, and a key computation kernel in this process is Tpetra’s [8] sparse matrix-matrix multiply.

III. SIERRA/NALU APPLICATION CODE

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 [16] usage

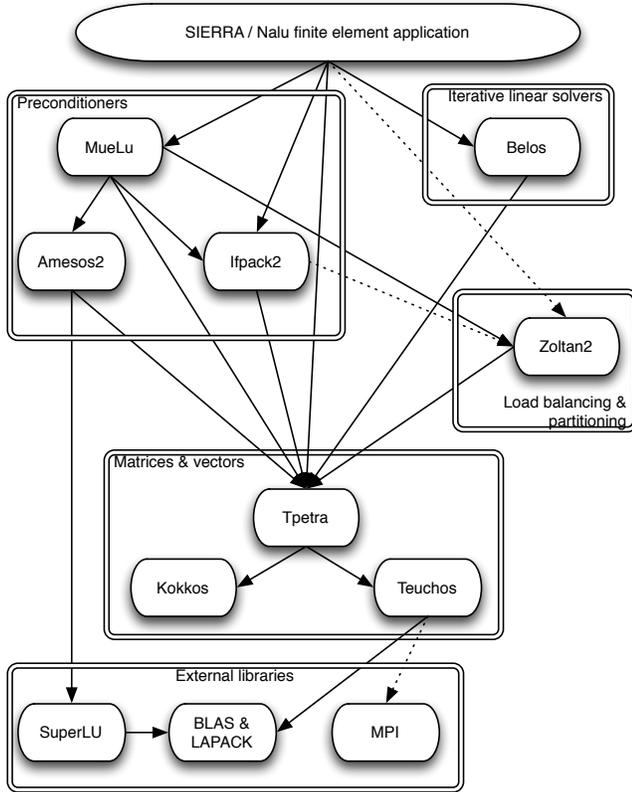


Figure 2. Software dependencies between SIERRA / Nalu, the Trilinos packages it uses, and externally provided libraries. Arrows show dependencies pointing at the target, that is, the “client” used by the other (source) package. Solid lines indicate currently required dependencies, and dotted lines indicate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

along with direct parallel matrix assembly to the Trilinos Epetra and Tpetra data structures.

As turbulent flows involve a very large range of spatial and temporal scales, simulation that resolves all the scales is currently impractical and therefore modeling approaches are required. Nalu supports a variety of turbulence models, however, all are classified under the class of modeling known as Large Eddy Simulation (LES). LES involves a spatial-filtering of the equations. Physical features larger than the mesh element scale will be resolved, while smaller features will be modeled. The spatial-filtered (Favre-filtered) equations are employed for this work. Detailed descriptions of the governing equations are provided in the Nalu

theory manual [17].

Nalu supports both control volume finite element (CVFEM) and edge-based vertex centered (EBVC) discretizations in the context of an approximate pressure projection algorithm (equal order interpolation using residual-based pressure stabilization), and both are employed in this study. Both discretizations are finite volume formulations and both solve for the primitive variables and are considered vertex-based schemes.

IV. 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.

A. Scalability Challenges from Integration

1) Matrix Assembly and Data Structures:

Much of the integration effort revolved around the assembly of finite elements into Trilinos sparse matrices and vectors. This is further complicated when the application, like Trilinos, uses a distributed-memory parallel programming environment. Mesh decomposition, global unknown numbering, and “ghosting” are just a few 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 an unstructured mesh.

The following assumptions drive the assembly process. First, the application knows how to map each unknown to a global identifier. Second, the application has decomposed the computational domain over MPI processes by elements. Processes are assigned both owned and shared nodes / unknowns. Third, assembling the linear system requires contributions from neighboring processes for unknowns near interprocess partition boundaries. Finally, since local unknowns are num-

bered continuously $[0, n)$ and global unknowns are sparse in $[0, n \times N_{proc})$, conversion from local to global unknown numbering (array dereference) is faster than conversion from global to local unknown numbering (hash table look-up).

Our approach led to the solvers owning the sparse matrix and dense vector data structures corresponding to the linear systems to solve, and to the application owning the assembly process and its corresponding mesh and element data structures.

Trilinos’ matrix storage classes let users store off-process contributions into a matrix. These classes then can assemble the off-process contributions into a consistent view with a single call to a function that “globally assembles.” Users do not have to specify or even know about the required interprocess communication. However, this simplicity and generality 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.

Fortunately, Trilinos itself lets users precompute and reuse a communication pattern from its source and target data distributions. Our custom global assembly exploits this feature, by splitting the matrix in two. One matrix contains only the owned rows on each process, and the other matrix only includes rows corresponding to off-process contributions. Once we construct the graphs for these two matrices, we created a communication pattern which combines the off-process contributions into the locally owned matrix on the remote process(es). We then reused it for each assembly. This method reduces communication for assembling off-process contributions, while exploiting existing Trilinos features. While this process relies on Trilinos functionality to achieve the goal of efficient assembly, this procedure of splitting the

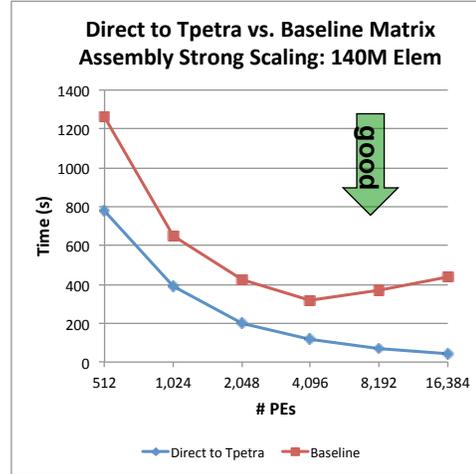


Figure 3. Comparison of matrix assembly: baseline through FEI compared with direct to Tpetra with additional performance optimizations. Latter is faster and scales considerably better: *over a factor of ten reduction in assembly time at 16k cores.*

matrix into local and remote components and communicating the remote portion into neighboring process(es) local matrix contributions is general and would apply if one is building fully assembled matrix representations.

Assembly of these two sub-matrices involves three sets of identifiers: the global unknown identifier, the process-local identifier, and the row in each of the matrices. As mentioned above, using global identifiers is slower (by about 10 times) than local identifiers, due to global to local conversion. Although determining the local identifier is fast, it does not directly correspond to a row in the local or remote matrices. Therefore, we created a mapping between local identifier to row in the matrix, 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 3 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.

2) *Tracking performance over time*: Although the goal for the new application code is to enable higher fidelity simulations, we had to do so without sacrificing the production code’s performance with smaller problems. To achieve this goal, it was essential to track performance comparisons between the new and current solver stack (Table I).

To compare the solver stacks and track their performance over time we used one instance of the test case described in the results section (Section V), a 140 million element problem on 2048 processors of the Cielo Cray XE6 sited at Los Alamos National Laboratory (LANL). The system consists of 8944 compute nodes (dual-socket 2.4GHz 8-core AMD Magny-Cours, total 142,000 cores) with Cray Gemini 3D torus interconnect.

Figure 4 tracks performance over time for the new Tpetra-based Nalu code, compared with the baseline Epetra-based code that employed the SIERRA framework and FEI. The latter will be referred to as “baseline.” Vertical axis plots the time ratio (ratio < 1 means the new code is faster than baseline). “Execute” time is the wall time (sans I/O time) of the simulation. “Matrix assemble” time is the time to fill the matrices for the linear solve. “Matrix solve” time includes the preconditioner setup and linear system solve time or preconditioned iteration time. The “matrix solve” reported here sums the time for the five equations (momentum equations, PPE and mixture fraction equation). We solved the momentum and mixture fraction equations with symmetric Gauss-Seidel preconditioned GMRES, and solved the PPE with MueLu preconditioned GMRES. Multi-grid preconditioner setup is performed once and reused for all time steps.

The horizontal axis plots days since March 12. On March 13, only the matrix assembly was faster

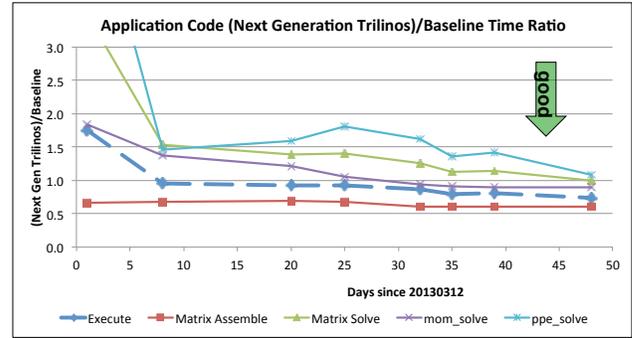


Figure 4. Tracking performance over time, comparing new Tpetra-based Nalu with baseline code (SIERRA/Framework, FEI, and Epetra). Vertical axis: ratio of the wall times of baseline code to new Tpetra-based Nalu code. Ratio < 1 means new code is faster. By the end of the time period, execute and solve times are 26% and 1% faster respectively than baseline.

than the baseline (as a result of the improvements described previously); the matrix solve time was slower. By March 21, the “execute” time was faster for the new vs. old codes. Day #20 (April 1) presents the first comparisons between the new code and the baseline with all six of the required Trilinos next-generation solver packages integrated into the new code. By the end (April 29 code base), execute time for the new code was 26% faster than the baseline, and the “matrix solve time” (sum of solve time for all five equations including preconditioner setup time) was 1% faster than the baseline. Note that sometimes the performance regressed. This demonstrates the value of carefully tracking performance.

3) *Effect of remapping on multigrid*: The MueLu algebraic multigrid library uses smoothed aggregation. This aggregates locally (i.e., aggregates cannot cross process boundaries) to group nodes on finer levels into a single node on the next coarser level. As the number of nodes per process is reduced on the coarser levels, the coarser level matrices are restricted to a subset of MPI processes, and their data are reassigned through rebalancing. Zoltan2 determines the new process assignment for the data, and MueLu performs the data migration.

Initial comparisons showed a performance difference between MueLu with Zoltan and MueLu

with Zoltan2. We determined that a “remap” step, implemented in Zoltan but not in Zoltan2, was the main cause. “Remap” partitions the rebalanced matrix in order to maximize overlap with the original matrix partitions. This minimizes the amount of data to communicate when applying the preconditioner. The MueLu and Zoltan2 teams consequently implemented a remapping procedure like Zoltan’s. Remap’s effect was not large enough to be included in the original design, but its impact showed at scale. For the 140 million element mesh test case on 2048 cores, discussed in Section IV-A2, the preconditioned iteration time (linear solve plus preconditioner apply, but not setup) for Zoltan2 without vs. with remap is 181 resp. 143 seconds. Lack of remap increases solve time by 27%. This illustrates the value of exercising codes at large scales on real problems with natural load imbalance.

B. Software Engineering Challenges

1) *Special cases for performance*: Solver developers tend first to write general code that implements the full promised set of features. Later, applications help the developers identify frequently used special cases that need a speed-up. Nalu helped Trilinos identify many such special cases. We found that addressing them added very little code maintenance overhead. Also, Nalu exercises a common set of use cases for linear solvers and preconditioners, so speeding up common special cases helped other applications.

For example, constructing Jacobi and symmetric Gauss-Seidel smoothers from a sparse matrix was taking too long. We fixed this by optimizing repeated extraction of the diagonal of a sparse matrix whose values change, but whose structure does not change.

We also found, to our surprise, that the Intel C++ compiler (e.g. Intel 11) refused to optimize inside a recursive function with template parameters. This slowed down communication setup. The function in question sorts one array and applies the resulting permutation to other arrays. We fixed this by writing a nonrecursive sort. Another example involved Trilinos’ calls to the

BLAS’ dense matrix-matrix multiply (DGEMM) for dense matrix-vector or dot products, when the corresponding specialized BLAS operations (DGEMV resp. DDOT) were actually faster. Although DGEMM worked, it was slower than more specific functions. This is probably because vendors tune DGEMM for when all matrix inputs have multiple columns, to optimize important benchmarks like LINPACK.

2) *Continuous integration of solvers and applications*: Both Sierra and Trilinos are large code bases with over 100 developers constantly making changes relevant to performance. Given these constant changes, we found continuous integration of the solver library with its application key to success, since it kept both code bases building without errors and passing tests. Nevertheless, continuous integration is considerably more challenging than one might expect. The two code bases have different repositories, developers, build systems, and even software engineering philosophies. It was difficult even to ensure that the application had the latest version of Trilinos.

Another aspect of continuous integration is the testing framework. In our experience, unit testing and scaling tests with mini-driver codes are necessary, but not sufficient to improve application codes. Although the most important metric is performance comparisons between the new and baseline application code, as shown in Section IV-A2, this work also employed unit testing and driver codes that could test a subset of the Trilinos packages. Examples of the latter included a Poisson driver that the MueLu team used to test correctness and performance. We had mini-drivers to compare performance of the Belos vs. Aztec linear solvers. The use of the driver codes was critical to identify and fix performance problems as they appeared. However, since they did not fully represent the complexity of the application code, there were cases where the driver codes would predict larger performance improvements than actually achievable with the application code. While one could argue that better mini-drivers would have solved that problem, in our experience, the key to real impact is a combination of continu-

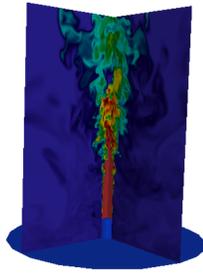


Figure 5. Mixture fraction-based turbulent open jet ($Re=6600$) test case used for the simulations. Jet emanates from pedestal at bottom of domain.

ous integration of the repositories, unit-testing and scaling tests with mini-drivers, and continuously tracking performance with the real applications themselves.

V. RESULTS

We now demonstrate Nalu’s ability to perform very large scale simulations using new Trilinos. The test case (Figure 5) is a mixture fraction-based turbulent open jet with Reynolds number 6600 [18]. The jet emanates from a pedestal from the bottom of the cylindrical computational domain, which we discretize with unstructured hexahedral elements. Nalu solves six equations: PPE, coupled momentum system for three component directions, mixture fraction, and subgrid-scale kinetic energy. Simulations ran on the Cielo Cray XE6 platform.

Figures 6 resp. 7 show weak scaling for assembly and solve for the coupled momentum system resp. PPE. The vertical axis is time per nonlinear step and the horizontal axis is the MPI process count. “Assemble” is the local matrix assembly time, “assemble+load_complete” adds global assembly time, and “solve” is the matrix solve time per nonlinear step. PPE results exclude preconditioner setup and plot only the multigrid-preconditioned GMRES time. Muelu’s multigrid setup time currently scales significantly worse than ML. MueLu has a different relationship to 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

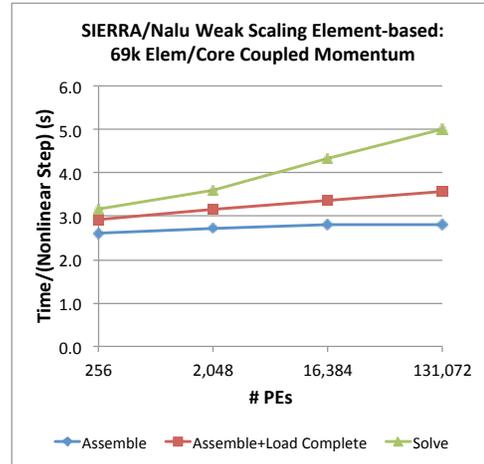


Figure 6. Nalu weak scaling for coupled momentum. Assemble time scales well. If plotted time per Krylov iteration, time would have been almost flat; here the increase in solve time is due to increase in iteration count. Very good scaling as increase of problem size is 512x and largest problem has 27 billion row matrix.

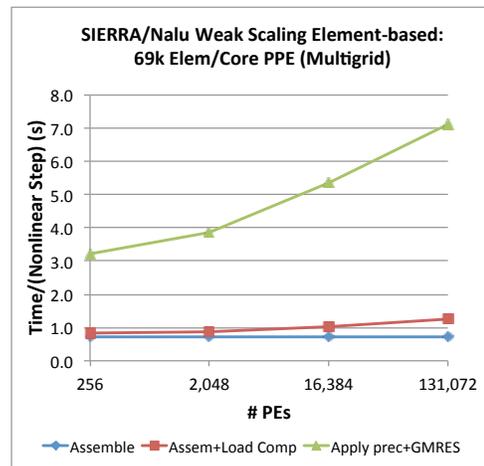


Figure 7. Nalu weak scaling for PPE. Assemble time scales well. Increase in solve time is due to increase in iteration count. Very good scaling as increase of problem size is 512x and largest problem has 9 billion row matrix.

implemented in Tpetra, while ML implemented its own. Improving MueLu’s setup time will require improvements to both MueLu and Tpetra. A large effort is underway by the MueLu team to improve setup performance and scaling by comparing MueLu and ML. Current efforts focus on optimizing Tpetra’s matrix-matrix multiply and reducing unnecessary communication.

The solve time’s increase per nonlinear step in

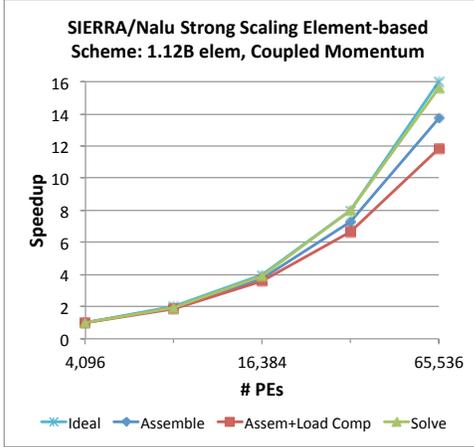


Figure 8. Nalu strong scaling for coupled momentum for 1.12 billion element mesh. Obtained ideal scaling for solve time while assemble time scales very well.

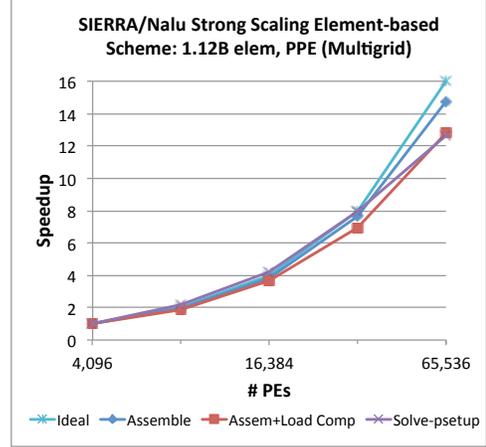


Figure 9. Nalu strong scaling for PPE for 1.12 billion element mesh. Very good scaling considering the size of the mesh and core counts (up to 65,536 cores).

Figure 6 as the problem is scaled up is due to the iteration count increase. If time per iteration were plotted, it would be almost flat (it increases 4% when weak scaling from 256 to 131,072 cores). The analogous increase in Figure 7 has mainly the same cause (iterations increased 50% from 256 to 131,072 cores). GMRES' property of orthogonalizing each new basis vector against all previous basis vectors leads to a superlinear time increase with respect to the iteration count. Note that for both figures, the problem is scaled by a factor of 512x, with the largest matrix for the coupled momentum and PPE being 27 billion rows resp. 9 billion rows on 131,072 processor cores. Given the size of the largest problem, scaling is very good. This scaling study had the additional purpose of demonstrating the capability to provide high fidelity simulations for analysts at SNL. For the simulation run on 131,072 cores, the maximum memory required by any MPI process was a bit more than half the available memory per mpi process, so we actually could have run a problem with almost twice as many elements.

Figures 8 and 9 present parallel speedup for coupled momentum and PPE respectively for the 1.12 billion element case as processor cores are increased from 4k to 64k. Ideal speedup is also plotted, and is a curved line because the vertical

axis is plotted on a linear scale rather than a logarithmic scale. In Figure 9, the fourth curve is the matrix solve time but not including the preconditioner setup time. For both coupled momentum and PPE, the scaling for the matrix assembly is very good. For coupled momentum, scaling of the matrix solve time is optimal, matching the ideal speedup. For PPE, the matrix solve time minus the preconditioner setup time is very good.

VI. CONCLUSIONS

We presented a case study of integrating a new set of sparse matrix solvers and data structures in Trilinos 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 131,072 cores of a Cray XE6 platform. For fully implicit low Mach field simulations on unstructured meshes we believe that our 9 billion node simulation on 131,072 cores rivals any other simulation in the field. The main remaining scaling bottleneck is multigrid setup; ongoing work will address this issue.

The new Trilinos solver stack allows arbitrarily large global entities and provides a path forward

for future 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 large-scale simulations, 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.

ACKNOWLEDGMENT

The authors would like to thank the following colleagues for their contributions: Anthony Agelastos, Ryan Bond, Kevin Copps, Mehmet Deveci, Karen Devine, Jeremie Gaidamour, Micheal Glass, Michael Heroux, Robert Hoekstra, Stephen Kennon, Kyran Mish, Brent Perschbacher, Kendall Pierson, Jim Willenbring, and Alan Williams.

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 *et al.*, “An overview of the Trilinos project,” *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 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. Edwards and J. Stewart, “Sierra: A software environment for developing complex multiphysics applications,” in *1st MIT Conference on Computational Fluid and Solid Mechanics*, K. Bathe, Ed., 2001.
- [4] H. Edwards, D. Sunderland, V. Porter, C. Amsler, and S. Mish, “Manycore performance portability: Kokkos multidimensional array library,” *Scientific Programming*, vol. 20, no. 2, pp. 89–114, 2012.
- [5] 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*, vol. 159, no. 8, pp. 2690–2703, 2012.
- [6] 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*, vol. 158, no. 7, pp. 1340–1357, 2011.
- [7] J. Schmidt, M. Berzins, J. Thornock, T. Saad, and J. Sutherland, “Large scale parallel solution of incompressible flow problems using Uintah and hypre,” University of Utah, Tech. Rep. UUSCI-2012-002, 2012.
- [8] C. Baker and M. Heroux, “Tpetra and the use of generic programming in scientific computing,” *Scientific Programming*, vol. 20, no. 2, pp. 115–128, 2012.
- [9] R. S. Tuminaro, M. Heroux, S. A. Hutchinson, and J. N. Shadid, “Aztec user’s guide—version 2.1,” Sandia National Laboratories, Tech. Rep. SAND99-8801J, 1999.
- [10] E. Bavier, M. Hoemmen, S. Rajamanickam, and H. Thornquist, “Amesos2 and Belos: Direct and iterative solvers for large sparse linear systems,” *Scientific Programming*, vol. 20, no. 3, pp. 241–255, 2012.
- [11] M. Gee, C. Siefert, J. Hu, R. Tuminaro, and M. Sala, “ML 5.0 smoothed aggregation user’s guide,” Sandia National Laboratories, Tech. Rep. SAND2006-2649, 2006.
- [12] J. Gaidamour, J. Hu, C. Siefert, and R. Tuminaro, “Design considerations for a flexible multigrid preconditioning library,” *Scientific Programming*, vol. 20, no. 3, pp. 223–239, 2012.
- [13] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, “Zoltan data management services for parallel dynamic applications,” *Computing in Science and Engineering*, vol. 4, no. 2, pp. 90–97, 2002.
- [14] M. Deveci, S. Rajamanickam, K. D. Devine, and U. V. Catalyurek, “Multi-jagged: A scalable multi-section based spatial partitioning algorithm,” Sandia National Laboratories, Tech. Rep., 2012.
- [15] Y. Saad and M. Schultz, “GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems,” *SIAM J. Sci. Stat. Comput.*, vol. 7, no. 3, pp. 856–869, Jul. 1986.
- [16] H. Edwards, A. Williams, G. Sjaardema, D. Baur, and W. Cochran, “Toolkit computational mesh conceptual model,” Sandia National Laboratories, Tech. Rep. SAND2010-1192, 2010.
- [17] S. Domino, “Low Mach Sierra Thermal/Fluids Module Nalu: Theory Manual,” Sandia National Laboratories internal document, Tech. Rep., 2013.
- [18] A. Abdel-Rahman, W. Chakroun, and S. Al-Fahed, “LDA measurements in the turbulent round jet,” *Mechanics Research Communications*, vol. 24, no. 3, pp. 277–288, 1997.