SAND2020-5244R

ECP-U-2020-xxx

Compare linear-system solver and preconditioner stacks with emphasis on GPU performance and propose phase-2 NGP solver development pathway

WBS 2.2.2.01, Milestone ECP-Q2-FY20

Jonathan Hu
Luc Berger-Vergiat
Stephen Thomas
Kasia Swirydowicz
Ichitaro Yamazaki
Paul Mullowney
Shreyas Ananthan
Sivasankaran Rajamanickam
Jay Sitaraman
Michael A. Sprague

May 14, 2020

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative. A portion of this research was supported by the U.S. Department of Energy Wind Energy Technologies Office.

This work was authored in part by the National Renewable Energy Laboratory, operated by Alliance for Sustainable Energy, LLC, for the U.S. Department of Energy (DOE) under Contract No. DE-AC36-08GO28308.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This report followed the Sandia National Laboratories formal review and approval process (SAND2020-XXXX R). As such, the technical report is suitable for unlimited release.

A portion of the research was performed using computational resources sponsored by the Department of Energy's Office of Energy Efficiency and Renewable Energy and located at the National Renewable Energy Laboratory. A portion of the research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

ECP-U-2020-xxx

ECP Milestone Report
Compare linear-system solver and preconditioner stacks with emphasis on GPU performance and propose phase-2 NGP solver development pathway
WBS 2.2.2.01, Milestone ECP-Q2-FY20

Office of Advanced Scientific Computing Research
Office of Science
US Department of Energy

Office of Advanced Simulation and Computing
National Nuclear Security Administration
US Department of Energy

May 14, 2020

ECP Milestone Report
Compare linear-system solver and preconditioner stacks with emphasis on GPU performance and propose phase-2 NGP solver development pathway
WBS 2.2.2.01, Milestone ECP-Q2-FY20

APPROVALS

Submitted by:

Michael Sprague

Michael A. Sprague ECP-Q2-FY20	7 May 2020 Date

Approval:

Thomas Evans ORNL	Date

REVISION LOG

Version	Creation Date	Description	Approval Date
1.0	2020-05-14	Original pre-release; draft sent to Tom Evans	
1.1	2020-05-15	Some corrections applied and document approved for public release	

EXECUTIVE SUMMARY

The goal of the ExaWind project is to enable predictive simulations of wind farms comprised of many megawatt-scale turbines situated in complex terrain. Predictive simulations will require computational fluid dynamics (CFD) simulations for which the mesh resolves the geometry of the turbines and captures the rotation and large deflections of blades. Whereas such simulations for a single turbine are arguably petascale class, multi-turbine wind farm simulations will require exascale-class resources.

The primary physics codes in the ExaWind project are Nalu-Wind, which is an unstructured-grid solver for the acoustically incompressible Navier-Stokes equations, and OpenFAST, which is a whole-turbine simulation code. The Nalu-Wind model consists of the mass-continuity Poisson-type equation for pressure and a momentum equation for the velocity. For such modeling approaches, simulation times are dominated by linear-system setup and solution for the continuity and momentum systems. For the ExaWind challenge problem, the moving meshes greatly affect overall solver costs as reinitialization of matrices and recomputation of preconditioners is required at every time step.

In this report we evaluated GPU-performance baselines for the linear solvers in the Trilinos and *hypre* solver stacks using two representative Nalu-Wind simulations: an atmospheric boundary layer precursor simulation on a structured mesh, and a fixed-wing simulation using unstructured overset meshes. Both strong-scaling and weak-scaling experiments were conducted on the OLCF supercomputer Summit and similar proxy clusters. We focused on the performance of multi-threaded Gauss-Seidel and two-stage Gauss-Seidel that are extensions of classical Gauss-Seidel; of one-reduce GMRES, a communication-reducing variant of the Krylov GMRES; and algebraic multigrid methods that incorporate the afore-mentioned methods. The team has established that AMG methods are capable of solving linear systems arising from the fixed-wing overset meshes on CPU, a critical intermediate result for ExaWind FY20 Q3 and Q4 milestones. For the fixed-wing strong-scaling study (model with 3M grid-points), the team identified that Nalu-Wind simulations with the new Trilinos and *hypre* solvers scale to modest GPU counts, maintaining above 70% efficiency up to 6 GPUs. However, there still remain significant bottlenecks to performance: matrix assembly (*hypre*), AMG setup (*hypre* and Trilinos) In the weak-scaling experiments (going from 0.4M to 211M gridpoints), it's shown that the solver apply phases are faster on GPUs, but that Nalu-Wind simulation times grow, primarily due to the multigrid-setup process.

Finally, based on the report outcomes, we propose a linear solver path-forward for the remainder of the ExaWind project. Near term, the NREL team will continue their work on GPU-based linear-system assembly. They will also investigate how the use of alternatives to the NVIDIA UVM (unified virtual memory) paradigm affects performance. Longer term, the NREL team will evaluate algorithmic performance on other types of accelerators and merge their improvements back to the main *hypre* repository branch. Near term, the Trilinos team will address performance bottlenecks identified in this milestone, such as implementing a GPU-based segregated momentum solve and reusing matrix graphs across linear-system assembly phases. Longer term, the Trilinos team will do detailed analysis and optimization of multigrid setup.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The ultimate goal of the ExaWind project is to enable scientific discovery through predictive simulations of wind farms comprised of many megawatt-scale turbines situated in complex terrain. Predictive simulations will require computational fluid dynamics (CFD) simulations for which the mesh resolves the geometry of the turbines (blade resolved) and captures the rotation and large deflections of blades. Whereas such simulations for a single turbine are arguably petascale class, multi-turbine wind farm simulations will require exascale-class resources [15] accelerated with graphics processing units (GPUs).

The primary solvers in the open-source ExaWind software stack [14] are Nalu-Wind[1], AMR-Wind[2], and OpenFAST[3]. Nalu-Wind is an unstructured-grid high-fidelity computational fluid dynamics (CFD) flow solver for wind turbine and wind farm applications. It solves the acoustically incompressible Navier-Stokes equations and is equipped with both Reynolds-Averaged Navier-Stokes (RANS) and Large Eddy Simulation (LES) models for turbulence closure. Additional transport equations, e.g., enthalpy, turbulent kinetic energy, etc., are available depending upon the problem. The Nalu-Wind code is closely tied to the Trilinos libraries, leveraging greatly the Sierra Toolkit (STK), the Kokkos abstraction layer for parallel-performance portability, and the linear-system solver libraries (MueLu, Belos, Tpetra). Through ExaWind, Nalu-Wind is also linked to the *hypre*[4] linear-system solver stack and to the Topology Independent Overset Grid Assembler[5] (TIOGA) for overset meshes. A detailed description of the Nalu-Wind numerical discretization scheme and the solution procedure has been given in the ECP ExaWind FY19-Q2 report with an overview given in [14]. AMR-Wind is a wind-focused structured-grid, acoustically incompressible CFD solver built on AMReX[6], a software framework for block-structured adaptive mesh refinement (AMR), and is a new addition to the ExaWind software stack. OpenFAST is a whole-turbine simulation code developed at the National Renewable Energy Laboratory (NREL) that includes models for nonlinear deflections of blades, the control system, and the tower.

The fluid-modeling pathway for ExaWind is for the near-turbine fluid to be modeled with an unstructured grid in Nalu-Wind, which enables body-conforming meshes well suited for resolving boundary layers. The far-field fluid can be modeled with either unstructured meshes in Nalu-Wind or structured meshes in AMR-Wind. In either case, the meshes are coupled through the overset method via TIOGA. Nalu-Wind and AMR-Wind rely on pressure projection to maintain continuity, which entails solving a Poisson-type equation at least once every time step, as well as solving Helmholtz-type momentum equations for velocity and other physical and model quantities. Nalu-Wind is equipped to solve these equations with the linear-system solvers and preconditioners from the *hypre* and/or Trilinos packages. AMR-Wind solves its systems with a multi-level multigrid solver. For the simulations where Nalu-Wind is coupled to AMR-Wind, global linear systems are solved in a decomposed manner where the preferred solver of each package is employed and coupling is accomplished via additive Schwarz. Performance analysis and optimization of this coupling approach is the subject of current [13] and future work.

This milestone is focused on the linear solvers and preconditioners used by Nalu-Wind, Trilinos and *hypre*, with particular emphasis on performance on GPU-accelerated systems. The objectives are to

1. understand and document the performance of the *hypre* and Trilinos solvers in their current form on a GPU-accelerated system (after many ECP-supported improvements including those in Exawind),
2. document a plan for maximizing the amount of the software stack that can be executed on the GPUs and optimizing performance.

Our testbed for this milestone is the Summit system at the Oak Ridge Leadership Computing Facility (OLCF). We have conducted a comprehensive suite of performance analyses that include weak and strong scaling studies.

[1] https://github.com/exawind/nalu-wind
[2] https://github.com/exawind/amr-wind
[3] https://github.com/openfast/openfast
[4] https://github.com/hypre-space/hypre
[5] https://github.com/jsitaraman/tioga
[6] https://github.com/AMReX-Codes/amrex

2. MILESTONE DESCRIPTION

In this section, we provide the approved milestone description and execution plan followed by a brief description of how the milestone was completed. Details regarding completion are included in the following sections.

2.1 DESCRIPTION

For incompressible-flow computational fluid dynamics simulations, the vast majority of simulation time is spent setting up and solving the underlying linear systems. These costs are exacerbated in blade-resolved turbine simulations, for which moving meshes require that linear systems and preconditioners be re-established at every time step. With the importance of the linear-system solver stack, the ExaWind project has supported evaluation and development of two solver-stack pathways: Trilinos (Belos/Tpetra/MueLu) and *hypre* . In this milestone we will report on ExaWind-supported advances under these two software stacks with particular focus on strong-scaling performance on CPU and GPU configurations for both the momentum and pressure-Poisson systems. The test problems will include wind-relevant problems established in previous milestones. Based on results, we will describe recommended paths forward for accomplishing predictive simulations on an exascale class system in the second phase of the ExaWind project.

2.2 EXECUTION PLAN

1. Optimize solver stacks for the wind energy problem with emphasis on next-generation platforms and improving the strong scaling limit on GPUs over current GPU capabilities.
2. Compare performance of the two solver stacks on wind relevant problems including performance on a next-generation platform such as GPU.
3. Provide a plan for linear-solver software stack development for post FY20 with a clear pathway to support predictive wind farm simulation on an exascale system.

Completion Criteria: Technical report describing the milestone accomplishment and a highlight slide summarizing those accomplishments.

2.3 OVERVIEW OF MILESTONE COMPLETION

The following is a concise description of how each of the items in Section 2.2 was satisfied for milestone completion.

1. Linear-system solvers:
   (a) A new two-stage Gauss-Seidel smoother was implemented in *hypre* and Trilinos, specifically targeted at achieving optimal GPU performance and improved solver scaling.
   (b) Trilinos: the entire setup of the multigrid solver has been re-implemented using kokkos for portability to GPU (with the exception of sparse direct solvers). Multiple improvements to the multi-threaded Gauss-Seidel algorithm have been made to improve numerical convergence (clustering algorithm) and performance (parallel triangular solve). The solve phase was on the GPUs before this work. Thus, the entire multigrid code is now GPU-ready, with the possible exception of coarse sparse direct solvers.
   (c) *hypre* : The solve parts of C-AMG V-cycle was implemented on GPU, including the non-square matrix-vector products for prolongation and restriction with asynchronous MPI. A reproducible sparse matrix assembly was implemented for the GPU but has not been interfaced to Nalu-Wind at the time of writing. The assembly procedure also conforms to the CASC-LLNL IJMatrixAddToValues API. The setup of C-AMG has to be GPU-enabled in the future.
2. Linear-system solver performance was examined with simulations performed on the OLCF Summit supercomputer for an atmospheric boundary layer (ABL) simulation that is a key component in any wind turbine simulation and a fixed-wing simulation, which was used as a proxy for a wind turbine blade. Strong and weak scaling studies were performed and documented. See §7 for details.

3. A development path forward for both Trilinos and *hypre* is provided in §8. Extensive studies and results from Summit simulations have revealed that the pressure continuity solvers in Trilinos and *hypre* are scaling sub-linearly and the time per iteration is growing at higher numbers of MPI ranks or equivalently for larger numbers of total pressure degrees of freedom. To improve throughput and reduce momentum solver time further, short-recurrence MINRES and block Krylov methods should be explored and implemented. The overall solver scalability for our target wind-turbine simulations in 2021-2023 should be further addressed. Additive Schwarz algorithms currently implemented in Nalu-Wind have the potential to mitigate these effects, but their strong-scaling properties for our proposed large-scale simulations have yet to be determined and would be a focus of our path forward work. In addition, the MPI communication on Summit may be affected by CPU-GPU memory transfers and synchronization. The CPU-GPU memory model (UVM versus mirror) and performance portability for Aurora and Frontier should be further explored in the *hypre* stack. Certainly a priority is to understand and address the sub-linear scaling of the pressure continuity equation solvers.

3. DESCRIPTION OF SIMULATIONS

As described in the Introduction, this milestone is focused on the Nalu-Wind CFD solver and its underlying linear solvers and preconditioners in *hypre* and Trilinos. Nalu-Wind equations are integrated in time using an implicit, second-order accurate Backward-Difference Formula (BDF2) time-stepping scheme. The resulting set of equations for a given time step are discretized in space, resulting in a large system of coupled nonlinear equations. This system of equations is decoupled, linearized and solved using a pressure projection scheme with a Picard fixed-point iteration. At the core of the Picard iterative process each of the physical equations mentioned in the previous paragraph results in a set of linear equations over the mesh points. Efficient GPU computation of solutions for these linear equations is the focus of this report and will be assessed using the two simulations presented below.

In order to evaluate the performance of the linear solvers employed within Nalu-Wind, we consider two specific wind-energy-relevant simulations: (a) large-eddy simulations (LES) of an atmospheric boundary layer (ABL) *precursor*, and (b) unsteady RANS simulation of the McAlister-Takahashi wind-tunnel experiment of a NACA0015 fixed wing. Detailed descriptions of the ABL precursor and the McAlister wing simulations, as well as the rationale for choosing these as candidate problems for this milestone, are presented below.

3.1 ABL PRECURSOR SIMULATIONS

LES simulations of wind farms require specification of time-varying, turbulent inflow and temperature profiles at the various boundaries of the computational domain. A common process for generating these boundary conditions are the so called ABL precursor simulations. Precursor simulations generally use a structured, Cartesian grid with cells of uniform resolution that encloses the desired volume of interest. Periodic boundary conditions are imposed on the four sides (x and y faces) and wall functions are imposed on the lower surface. The top boundary is modeled as follows: a symmetry (zero-gradient) boundary condition is applied to velocity field and a constant temperature gradient is enforced to simulate conditions above the capping inversion layer. The flow field is initialized with sinusoidal perturbations in the velocity field to trigger turbulence generation. The one equation k-SGS turbulence model is applied for LES closure. For ABL neutral boundary layer simulations, i.e., no net surface heat flux into or from the flow at the ground, the simulations are typically run for about 20,000 seconds before reaching equilibrium.

From mesh-generation and simulation perspective this is a simple problem but has several key algorithmic ingredients fundamental to modeling a wind farm and, therefore, presents an ideal setup to explore strong and weak scaling behavior of the linear solvers. For this milestone, the neutral ABL precursor simulation is performed for a $5\text{km} \times 5\text{km} \times 1\text{km}$ domain at four different grid resolutions – see Table 1. The 20 m resolution mesh (≈ 3.26 million pressure DOFs) was chosen as a candidate for strong scaling studies.

3.2 MCALISTER WING SIMULATIONS

Unsteady RANS simulation of a fixed-wing, with a NACA0015 cross section, operating in uniform inflow was chosen as the second problem to evaluate the performance of linear solvers. Unlike the ABL precursor

Δx [m]	grid points	Δt [s]	max CFL	time steps
40	412,776	2.4	0.51	10
20	3,268,805	1.2	0.51	10
10	26,248,250	0.6	0.51	10
5	210,546,279	0.3	0.51	10

Table 1: Details of the computational meshes used for weak scaling studies of the neutral ABL precursor simulations.

problem, resolving the high-Reynolds number boundary layer over the wing surface requires resolutions of $O(10^{-5})$ normal to the surface resulting in cell aspect ratios of $O(40,000)$. These high aspect ratios, coupled with the loss of diagonal dominance in the momentum system, present a significant challenge for the iterative solvers. Overset meshes were employed to generate body-fitted meshes for the wing and the wind tunnel geometry, this presents additional complications for the linear solver in form of overset constraint rows.

The simulations were performed for a wing at 12° angle of attack, a 1 m chord length, denoted c, 3.3 aspect ratio, i.e., $s = 3.3c$, and a square wing tip. The inflow velocity is $u_\infty = 46$ m/s, the density is $\rho_\infty = 1.225$ kg/m^3, and the dynamic viscosity is $\mu = 3.756 \times 10^{-5}$ kg/(m s), leading to a Reynolds number, $Re = 1.5 \times 10^6$. The computational mesh was generated with commercial mesh generation software, point-wise, and models the half-wing as well as the wind tunnel walls. Wall normal resolutions were chosen to adequately represent the boundary layers on both the wing and tunnel walls. The $k-\omega$ SST RANS turbulence model was employed for the simulations. Due to the complexity of mesh generation, only one mesh with approximately 3 million grid points was generated. Only strong scaling studies were performed for the McAlister wing problem.

3.3 MOMENTUM SOLUTION PROCEDURE FOR TRILINOS AND HYPRE STACKS

Currently one significant difference in how these equations are solved in the *hypre* and Trilinos solver stacks is segregation of the momentum equations. The segregated solver makes the hypothesis that the block diagonal terms in the momentum equation are dominant and close to each other which allows the solver to use a single diagonal block to solve for each component of the velocity field as seen in equation (1). This approach is currently not implemented in the Trilinos solver stack for GPUs (an algorithm exists for CPUs) which means that the momentum equation solved by the Trilinos stack is three times larger than that solved by the *hypre* stack.

$$\begin{bmatrix} M_{xx} & M_{xy} & M_{xz} \\ M_{yx} & M_{yy} & M_{yz} \\ M_{zx} & M_{zy} & M_{zz} \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} f_x \\ f_y \\ f_z \end{bmatrix} \approx M_{xx} \begin{bmatrix} v_x & v_y & v_z \end{bmatrix} = \begin{bmatrix} f_x & f_y & f_z \end{bmatrix} \tag{1}$$

4. OVERSET MESHES

ExaWind simulation environment makes extensive use of overset-mesh methodology to support modeling arbitrary motion and deformations such as blade rotation, nacelle yaw, as well as blade bending and twisting due to structural dynamics. Overset-meshes employ body-fitted, near-body meshes embedded within a background mesh. Flow information is exchanged between the overlapping meshes using field interpolation from the donor cell to the receptor node on the receiver mesh. Overset domain connectivity is the process that determines where the flow equations are solved and how information is exchanged between the overlapping grids when they overlap in a given point in space. The process tags each node in the collection of computational meshes as *field*, *fringe*, or *hole* nodes. Field nodes are nodes on the mesh where the flow equations are solved, fringe nodes are receptors that receive data from a *donor cell* located in another mesh that solves the flow equations at that point in space, and hole nodes are nodes that either lie within a solid body (and, therefore, cannot have a valid solution) or are entirely bounded by fringe nodes (and, therefore, need not be solved for at a given time-step). Within the ExaWind simulation environment, domain connectivity is performed using the third-party library TIOGA.

4.1 CONSTRAINT ROWS AND IMPLICATIONS FOR AMG PRECONDITIONERS

From a linear system perspective, this results in introduction of a set of constraint equations within the system, i.e., the rows corresponding to the fringe nodes are modified such that instead of the terms originating from the discrete operators of the terms in the PDE, it contains the interpolation stencil for the solution at the fringe node from the donor cell. These constraints can be written as

$$\alpha_s \left[\Delta\phi_i^m - \sum_{k=1}^{N^e} w_k^e \Delta\phi_k^n \right] = 0 \,, \qquad \sum_{k=1}^{N^e} w_k^e = 1 \,, \tag{2}$$

where $\Delta\phi_i^m$ is the solution update to the fringe node on mesh m for a field ϕ, $\Delta\phi_k^n$ is the solution update (determined by the linear system) at nodes ($k = \{1, \dots, N^e\}$) of the donor element e on mesh n, and w_k^e is the interpolation weight determined by the shape functions of the donor element, and α_s is a row-scaling coefficient whose value is determined based on a procedure described next. The solution is fully coupled and does not require a solution-exchange step after the linear solve. However, introduction of constraint rows within an elliptic system poses a challenge to the efficient operation of algebraic multigrid preconditioners and often results in poor convergence during the iterative solution process. As mentioned in FY19-Q2 report, the team is considering potential mitigation approaches. First, decoupling the linear system using an alternating Schwarz approach. This is the focus of the FY20-Q3 milestone. The second option is a constraint-elimination step for which the constraint equations are recast as a restriction operator that is used to generate a new linear system consisting of only the field degrees-of-freedom.

While alternate approaches were being considered, ad hoc testing revealed that the convergence of the linear systems improved if the scaling parameter, α_s, was chosen such that the diagonal value of the constraint row was close to its neighbors. To determine a suitable value for α_s, we consider the discretized system of equations solved at each Picard nonlinear iteration step (see [14]):

$$A_p \Delta\hat{u}_p^{k+1} + A_{nb}\Delta\hat{u}_{nb}^{k+1} = r^k - \nabla p^k \,, \qquad \text{Momentum} \tag{3}$$

$$-\tau L \Delta p^{k+1} = -\nabla \cdot \left(\rho \hat{u}^{k+1}\right) + \tau L p^k - D\tau G p^k \,, \qquad \text{Continuity} \tag{4}$$

$$u^{k+1} = \hat{u}^{k+1} - \left(\frac{\tau}{\rho}\right) \nabla \left(\Delta p^{k+1}\right) \,, \qquad \text{Projection} \tag{5}$$

$$\text{where} \qquad A_p = -\sum A_{nb} + \frac{\gamma_1}{\Delta t}\rho\Delta V_p; \qquad \tau = \mathrm{diag}(A)^{-1}; \qquad \Delta\phi^{k+1} = \phi^{k+1} - \phi^k \,.$$

A_p represents the diagonal term of the A matrix, A_{nb} are the coefficients of the off-diagonal columns, and r^k is the residual of the momentum equation system. The last two terms on the right-hand side (RHS) of Eq. 4 play the role of Rhie-Chow interpolation. For transport equations (e.g., momentum, enthalpy, turbulent kinetic energy), the leading diagonal term is of the order $O(\gamma_1 \Delta V_p/\Delta t)$. Therefore, we choose $\alpha_s = \gamma_1 \Delta V_p/\Delta t$ for the transport equations. On the other hand, for elliptic pressure Poisson equation, the diagonal term is of the form $\Delta V_p/\Delta x^2$. Therefore, for the pressure continuity system, we approximate $\alpha_s = \sqrt[3]{\Delta V_p}$.

It must be emphasized that the current choice of α_s is arbitrary and its implications on the performance of AMG preconditioners and convergence of the iterative solution process have not been thoroughly examined. The scaling parameter is described here for completeness, as it is being currently used in production runs using the ExaWind simulation environment. Further study of the implications of the choice of α_s on the performance of AMG preconditioners and the convergence characteristics of the iterative solvers is warranted, and is anticipated to be a focus of future milestones.

4.2 MESHES, DYNAMIC RANGE AND CONDITION NUMBER ESTIMATION

The matrices for the momentum and continuity equations in Nalu-Wind are large and quite sparse with on the order of twenty non-zeros per row. The dimension of these matrices is $\mathcal{O}(10)$ billion for the problems of interest. The convergence of iterative solvers is determined to a large extent by the condition number of the matrices. The condition number is also a measure of the sensitivity of the problem $Ax = b$ to perturbations of the form $(A + \Delta A)x = (b + \Delta b)$ that can occur from floating point errors arising in matrix assembly.

The condition number is formally defined as the ratio of the largest to smallest singular values of the matrix A as given by $\kappa(A) = \sigma_{\max}/\sigma_{\min}$. The singular values of a large sparse matrix are difficult and expensive to compute in practice. Typically, a sparse eigenvalue package such as ARPACK could be employed. However, an alternative estimate would be desirable in the design and evaluation of solver algorithms. One possible surrogate is based on the dynamic range of the simulation as determined by the ratio of the largest to smallest mesh grid cell volumes.

$$\kappa(A) \approx \Delta V_{\max}/\Delta V_{\min}$$

When $\kappa(A) > 1e+16$, or the inverse of the machine round-off error ε, then the problem may not be tractable. Equilibration and re-ordering can mitigate these effects but at additional costs. For the momentum matrix, diagonal dominance is assumed and results when upwind discretisations are employed. When this is not the case, ILUTP type preconditioners may be needed and these are difficult to implement on the GPU, although recent work by Anzt and Chow (2020) may result in ILU algorithms amenable to GPU implementation. These will not be described here but are possibly of interest in our ongoing efforts.

5. LINEAR SOLVER ALGORITHMS

In this section, we discuss the key linear solvers that are used in this report. We begin with a brief background on the two AMG methods that Nalu-Wind uses. We then discuss linear solver algorithms employed by Nalu-Wind on GPU architectures: a one-step communication GMRES Krylov algorithm; a two-stage Gauss-Seidel smoother with an inexact iterative lower triangular solve; and a multi-threaded Gauss-Seidel relaxation smoother.

5.1 ALGEBRAIC MULTIGRID

Algebraic multigrid (AMG) methods are effective scalable solvers that are well suited for high-performance computer architectures [18, 9, 10]. When employed as a stand-alone solver or as a preconditioner for a Krylov iteration such as GMRES [12], AMG can in theory solve a linear system with n unknowns in $\mathcal{O}(n)$ operations.

An AMG method accelerates the solution of a linear system

$$Ax = b \tag{6}$$

through error reduction by using a sequence of coarser matrices called a *hierarchy*. We will refer to the sequence of matrices as A_k, where $k = 0 \ldots m$, and A_0 is the matrix from (6). Each A_k has dimensions $m_k \times m_k$ where $m_k > m_{k+1}$ for $k < m$. For the purposes of this paper, we will assume that

$$A_k = P_k^T A_{k-1} P_k \,, \tag{7}$$

for $k > 0$, where P_k is a rectangular matrix with dimensions $m_{k-1} \times m_k$. P_k is referred to as a *prolongation matrix* or *prolongator*. R_k is the *restriction matrix* and $R_k = P_k^T$ in the Galerkin formulation of AMG. Associated with each A_k, $k < m$, is a solver called a *smoother*, which is usually an inexpensive iterative method, e.g., Gauss-Seidel, polynomial, or incomplete factorization. The *coarse solver* associated with A_m is often a direct solver, although it may be an iterative method if A_m is singular.

The setup phase of AMG is nontrivial for several reasons. Each prolongator P_k is developed algebraically from A_{k-1} (hence the name of the method). Once the transfer matrices are determined, the coarse-matrix representations are recursively computed from A through sparse matrix-matrix multiplication.

In the AMG solve phase, a few steps of a smoother are applied to the finest-level linear system with a zero initial guess. This is referred to as *pre-smoothing*. A residual is calculated and restricted to the next coarser level, for which it becomes the right-hand side for the next-coarser linear system. This process repeats recursively until the coarsest level is reached. The coarsest-level system is usually solved with a direct method. The solution of the coarsest-system solve is then interpolated to the next finer level, where it becomes a correction for that level's previous approximate solution. A few steps of *post-smoothing* are applied after this correction. This process is repeated until the finest level is reached. This describes a V-cycle, the simplest complete AMG cycle. See Algorithm 1 for the complete description. AMG methods achieve optimality (constant work per degree of freedom in A_0) through complementary error reductions by the smoother and solution corrections propagated from coarser levels.

The two main AMG methods employed by Nalu-Wind are Ruge-Stüben AMG (also called *classical* AMG or C-AMG) and smoothed aggregation AMG (SA-AMG). As mentioned previously, the main focus of both

Algorithm 1 AMG single-cycle algorithm ($\nu = 1$ yields V-cycle) for solving $Ax = b$ for $m + 1$ level hierarchy.

```
//Solve $Ax = b$.
Set $x = 0$.
Set $\nu = 1$ for $V$-cycle.
call Multilevel($A, b, x, 0, \nu$).

function MULTILEVEL($A_k$, $b$, $x$, $k$, $\nu$)
    // Solve $A_k x = b$ ($k$ is current grid level)
    // Pre smoothing step
    $x = S_k^1(A_k, b, x)$
    if ($k \neq m$) then
        // $P_k$ is the prolongator of $A_k$
        // $R_k$ is the restrictor of $A_k$
        $r_{k+1} = R_k(b - A_k x)$
        $A_{k+1} = R_k A_k P_k$
        $v = 0$
        for $i = 1 \ldots \mu$ do
            MULTILEVEL($A_{k+1}$, $r_{k+1}$, $v$, $k+1$, $\nu$)
        end for
        $x = x + P_k v$
        // Post smoothing step
        $x = S_k^2(A_k, b, x)$
    else
        $x = A_k^{-1}\ b$
    end if
end function
```

methods is determining an effective grid transfer operator, which governs how data is interpolated between multigrid levels, and a complementary smoother. In C-AMG, the unknowns (*C-points*) of the coarse grid system are a proper subset of the fine grid (*F-points*) unknowns. There are a number of different algorithms for selecting the C-points, which F-points they interpolate to, and the weights of the interpolation. In SA-AMG, the coarse level unknowns are formed by grouping fine level unknowns together into *aggregates*. As with C-AMG, there are different algorithms for selecting the aggregates. Once the aggregates are formed, interpolation weights are calculated via local ortho-normalization. The aggregation algorithm is performed local to an MPI rank or sub-domain.

5.2 ONE-REDUCE GMRES SOLVERS

Solving large sparse linear systems of the form $Ax = b$ is a basic and fundamental component of DOE physics based modeling and simulation software. Within the Trilinos (SNL) and *hypre* (LLNL) solver stacks a critical step is Gram-Schmidt orthogonalization algorithms, where synchronization in the form of global reductions is a considerable barrier to scalability and performance — especially strong scaling and sustained performance on accelerated architectures such as GPUs. Strong scaling challenges arise in many applications where the need to simulate long time horizons favors relatively fewer degrees of freedom per computational unit.

Recent work by Świrydowicz and Thomas to reduce the synchronization requirements for Krylov iterative solvers, such as the generalized minimum residual method (GMRES) [12], has resulted in *low-sync one-reduce Gram-Schmidt orthogonalization algorithms*, where the 100-year old [8] Gram-Schmidt orthogonalization algorithms applied to the Arnoldi-QR factorization have been reformulated and now only require one global MPI reduction (global communication and synchronization) per matrix column or iteration, compared to a quadratically increasing number of inner-products. Results for comparing standard GMRES and the algorithm are described in more detail in [5].

5.3 TWO-STAGE GAUSS-SEIDEL PRECONDITIONER AND SMOOTHER

Given a large sparse linear system $Ax = b$, the traditional form of a Gauss-Seidel iteration is based on the matrix splitting $A = D + L + U$, where D is diagonal, L is strictly lower and U is strictly upper triangular. The standard Gauss-Seidel iteration can be written as

$$x_{k+1} = x_k + (D + L)^{-1} \, (\, b - Ax_k \,) \tag{8}$$

where $x_0 = 0$ is the initial solution iterate. To avoid computing an inverse, a sparse triangular solver is usually employed.

The lower triangular solver algorithm is a recurrence and lacks sufficient fine-grained parallelism to achieve a high execution rate on SIMT type architectures such as the NVIDIA Volta GPU. An alternative approach is to replace the lower triangular solve in the above iteration with an inner Jacobi-Richardson type iteration of the form

$$y_{j+1} = y_j + D^{-1} \, [\, r_k - (\, D + L \,)\, y_j \,] \tag{9}$$

where $r_k = b - Ax_k$ and $y_0 = 0$. The application of Jacobi iterations to solve large sparse triangular linear systems for ILU preconditioners, rather than direct methods based on forward and backward recurrence relations, was recently proposed by Chow et. al. [2].

The preconditioner for a Krylov subspace iteration such as GMRES then takes the form

$$x_{k+1} = x_k + y_{t+1} \tag{10}$$

where t is the number of inner Jacobi-Richardson iterations performed. For one iteration of the Jacobi iteration, consider substitution of the expression for y_k from equation (9) into (10) in order to obtain

$$\begin{aligned} x_{k+1} &= x_k + y_k + D^{-1} \, [\, r_k - (\, D + L \,)\, y_k \,] && (11) \\ &= x_k + D^{-1} r_k + D^{-1} \, [\, r_k - (\, D + L \,)\, D^{-1} r_k \,] && (12) \end{aligned}$$

The above expression represents a two-stage Gauss-Seidel iteration. More generally, two-stage nested iteration has been studied in a series of papers by Frommer and Szyld, [7], [16], [3], [4].

In practice, we find that one inner Jacobi iteration is sufficient to reproduce the convergence rate of the standard GMRES-SGS (symmetric Gauss-Seidel) solver. A clear advantage of our approach is that the sparse lower triangular solver has been replaced with several sparse matrix-vector multiplications. These kernels are roughly 20× faster than the lower triangular solver on a GPU.

Numerical results using Nalu-Wind will be presented in §7 that show the performance of two-stage (symmetric) Gauss-Seidel both as a smoother in C-AMG and as a preconditioner to a Krylov method.

5.4 MULTI-THREADED SPARSE TRIANGULAR SOLVE

Sparse triangular solution is a major bottleneck in iterative and direct solvers for large sparse linear systems. The traditional parallel algorithm is based on level-set scheduling of the elimination tree, followed by substitutions determined by these data dependencies. Independent computations proceed within each level of the tree. However, each level in the tree must be completed before moving on to the next, Saad (2003). This allows for limited parallel computations within a given level of the tree and results in a serial bottleneck. Kelly and Rajamanickam [6], have implemented multi-threaded triangular solvers for the GPU using Kokkos and based on multi-coloring and clustering. The former is known to increase the iteration count for GMRES solvers in certain problems, whereas clustering is a robust algorithm which maintains convergence rates.

Numerical results using Nalu-Wind will be presented in §7 that show the algorithmic scalability of a Multithreaded Gauss-Seidel as a preconditioner to a Krylov method.

6. PERFORMANCE PORTABILITY

Each of the past three announcements for DOE systems have included a very different node level architecture. Aurora will use Intel's Xe GPUs, Frontier and El Capitan will use AMD GPUs, whereas Perlmutter will use NVIDIA GPUs. This makes portability an important aspect of our plans for the upcoming milestones.

There are three different alternatives for performance portability - (1) A directive-based approach such as OpenMP directives (2) use of a portable programming model such as Kokkos or RAJA. (3) Implement the code in the native GPU programming model. There are distinct advantages and disadvantages with each option. While a directive-based approach relies on a standard, the time lag in the OpenMP standard and its current design based on CPU threading makes it difficult to rely on for our immediate milestones. Using libraries such as Kokkos or RAJA bring in an external dependency that is not standardized, but it allows us to focus on solver algorithms while Kokkos and Kokkos Kernels developers work with vendors for portable options. Direct implementation in the native programming model provides the flexibility of optimizing for each architecture at the risk of code duplication and the effort to maintain the code base on all architectures. In the solver effort, we have chosen two different paths. The Trilinos stack is implemented with Kokkos as the programming model. The *hypre* stack uses the native CUDA programming model. This results in interesting choices in data structures and implementation. For example, Trilinos relies on the standard CRS matrix data structure for all the architectures. *hypre* uses custom data structures to split the on-rank diagonal and off-rank portions. The implications of such data structure conversions as the matrix changes every time-step is yet to be understood. The Trilinos CRS data structure could remain the same as long as the non-zero structure of the matrix remains the same. The data structure has to be created again otherwise. *hypre* data structures need to be ordered and sorted in order to use standard CSR functions in libraries such as cuSparse. These changes might impact the cost of assembly and that will factor more into the overall performance as the non-zero structure changes.

6.1 TRILINOS IMPLEMENTATION DETAILS

The Trilinos solvers and matrix assembly described in this report rely on the Kokkos ecosystem, which consists of the Kokkos Core programming model, Kokkos Kernels sparse linear algebra library, and Kokkos Tools. The Kokkos team works with all the hardware architecture vendors - AMD, ARM, IBM, Intel, and NVIDIA - before the hardware is even fully designed. Kokkos provides a back-end for each of the accelerator architecture in its native programming model. The Kokkos team is also developing a OpenMP target back-end. This portability allows the Kokkos based assembly and solver path to run on exascale hardware as soon as they are released. The Exawind team also has Kokkos Kernels developers in it (Rajamanickam, Yamazaki) who ensure the kernels needed for the assembly and solvers are well optimized on new architectures. For example, the Kokkos team just release a HIP back-end in collaboration with the AMD developers. A SYCL backend is being developed for GPUs on Aurora. Once the back-ends are developed, Kokkos Kernels implementations of the smoothers described here will be exercised in performance tests. When the Kokkos ecosystem is integrated into Trilinos, the continuity solver can be immediately exercised on the corresponding architecture. We will use Kokkos Tools to fine tune the performance on each of the architectures. This will allow us to focus on future architectures for algorithm development and fine tuning.

For the memory model, Trilinos and *hypre* from the CASC–LLNL team rely on UVM. Alternative memory models are being considered for future exascale architectures.

6.2 HYPRE IMPLEMENTATION DETAILS

The matrix-vector multiplies (SpMv) for the one-reduce GMRES in *hypre* have been custom written for performance and data re-use. In this report, we focus on the specific components of the AMG V-cycle presented earlier in Algorithm 1. In particular, the entire V-cycle now executes on the GPU with prolongation and restriction non-square matrix vector multiplies at each level of the V-cycle computed on the GPU. Indeed, two separate matrix-vector multiplies are implemented, the second being for a transposed matrix optionally stored by *hypre* for added performance. The V-cycle P/R SpMv matvecs are asynchronous and implemented with `MPI_Irecv`, `MPI_Isend` and `MPI_WaitAll`, with different MPI tags per each level.

The momentum solver preconditioner and the AMG smoothers are based on the two-stage Gauss-Seidel algorithms described previously. These also rely on SpMv matrix-vector products. In *hypre* the matvecs are merged into a single CUDA kernel launch in order to reduce latency and minimize the memory footprint. By far one of the largest costs in the solver GPU implementation is associated with the GPU memory CUDA alloc and free across solver iterations, in particular for the smoother. Every effort has been made to further reduce the number of temporary vectors in GMRES solvers and these are merged with *hypre* matrix data structures. Currently, there are no plans to use a memory pool or equivalent to alleviate these overheads. However, this could be considered in our future plans.

In previous milestone reports, we have focused on the strong scaling characteristics of the GMRES Krylov iterative solvers and in particular achieving flat execution times for the Gram-Schmidt orthogonalization kernels on CPU and GPU. For the *hypre* solver stack we have adopted a mirror-memory model which maintains both CPU and GPU CUDA pointers. With some duplication and movement of data as needed by the solver algorithms. This contrasts with the CASC–LLNL team who currently employ the shared unified virtual memory (UVM) model that relies on migration of memory pages from the main memory to the GPU global memory, rather than direct programmer management of the data.

Memory pre-fetching from UVM to the GPU device memory may result in a speed-up and higher memory bandwidth utilization, however, this approach has not been employed in our current *hypre* code development and performance studies to date. For our path forward with CASC-LLNL, such a comparison is certainly merited. The current version of *hypre* master v2.18.0 as of mid-March has prefetch commands in key areas like the SpMV.

7. NUMERICAL RESULTS

In this section, we present numerical results for the simulations described in §3. The experiments in this section use the Trilinos and *hypre* solver stack described in §5. All experiments were run on the Summit supercomputer at Oak Ridge National Laboratory. Summit has 4608 compute nodes, each with two IBM Power9 CPUs and six NVIDIA Volta V100 GPUs [11]. Each Power9 CPU has 22 cores, and there are 512 GB of DDR4 memory available to the CPUs.

In the following discussion, strong- and weak-scaling results are presented. In *strong scaling*, the global problem size is fixed, and timing results are presented over a range of MPI process counts. In *weak scaling*, the problem size is held constant on each MPI rank, and timing results over a range of MPI ranks are presented. Unless otherwise noted, all results were generated from restart files and run for an additional ten time steps.

The codes used to obtain the results presented in the rest of this report are publicly available online, and the precise location and versions are reported in Table 2.

Name	Location	Branch	SHA1
Nalu-Wind	https://github.com/exawind/nalu-wind/	master	a3fd9cc
Trilinos	https://github.com/trilinos/Trilinos/	master	ac87b08
hypre	https://github.com/Exawind/hypre	Feb2020Branch	c486551
LLNL *hypre*	https://github.com/hypre-space/hypre	v2.18.2	fbe2530

Table 2: Location and versions of the code used to perform the simulations reported in this section. These allow for full reproducibility of the results presented.

7.1 METHODOLOGY

7.1.1 hypre and Trilinos Solver Settings

For the ABL and McAlister simulations on GPU described herein, *hypre* is configured with the host to GPU CUDA mirror-memory model, together with the one-reduce GMRES Krylov solver for both momentum and continuity. The momentum preconditioner is the two-stage Gauss-Seidel iteration on the GPU with merged sparse matrix-vector products (SpMv) in a single kernel launch per smoother call. The pressure preconditioner is the AMG V–cycle with matrix-vector multiplies for the P/R prolongation and restriction operators on GPU. One sweep of the two-stage Gauss-Seidel is employed as the C-AMG smoother and also as the momentum preconditioner. HMIS coarsening was found to be more effective for the ABL problem and leads to lower times.

In order to perform IBM Power9 CPU comparison runs, the current *hypre* release from CASC-LLNL is employed. The standard GMRES algorithm is applied and the C-AMG smoother and momentum preconditioner are two sweeps of the symmetric Gauss-Seidel algorithm. PMIS coarsening is employed in the McAlister strong scaling runs. Otherwise the choice of CPU and GPU *hypre* algorithms remain the same.

For the experiments in this section, unless otherwise specified, the MueLu AMG solvers used for the continuity solve employ a degree-2 Chebyshev smoothing and a serial direct solver. The linear solvers for other physics are GMRES preconditioned by standard Gauss-Seidel, multi-threaded Gauss-Seidel, or two-stage Gauss-Seidel. Where appropriate, relevant solver details are given with individual results.

7.1.2 Comparing CPU to GPU performance

A single Summit node has 42 total CPU cores (two 21-core CPUs) and 6 GPUs. When doing performance comparisons, the Trilinos experiments compare performance of all a node's GPU devices to all of the node's CPU cores. When this isn't feasible, e.g., a problem is not large enough to occupy all GPUs or cores, we fall back to using the comparison ratio of a single GPU to approximately 7 CPU cores.

For the McAlister CPU strong scaling comparison runs reported herein, a single Summit node was employed for the *hypre* stack runs up to twenty MPI ranks. These runs also only employed the symmetric Gauss-Seidel smoother for continuity C-AMG and the momentum preconditioner. In the case of the 20 MPI rank runs with CPU assembly + GPU solvers, the simulations employed two and four GPU's per node in order to determine the effects of MPI communication across multiple nodes on the model scaling. Here, the two-stage Gauss-Seidel smoother for the GPU was applied.

7.2 SINGLE NODE PERFORMANCE OF TWO-STAGE GAUSS-SEIDEL

Before evaluating performance of the entire solver stack, we first consider performance of just single-level iterative preconditioners that will be used in all physics except continuity. This performance study compares the single GPU execution time of the Trilinos-Belos GMRES solver when using two-stage Gauss-Seidel described in §5.3 against the triangular-solver recurrence implementations of sequential SGS and MTSGS. The execution time for one Trilinos-Belos GMRES solve applied to the ABL precursor 40 m mesh momentum matrix is displayed in Figure 1. This plot illustrates the rather dramatic decrease in GPU compute time for the two-stage smoother compared to the sequential and multi-color order triangular solve symmetric Gauss–Seidel algorithms.

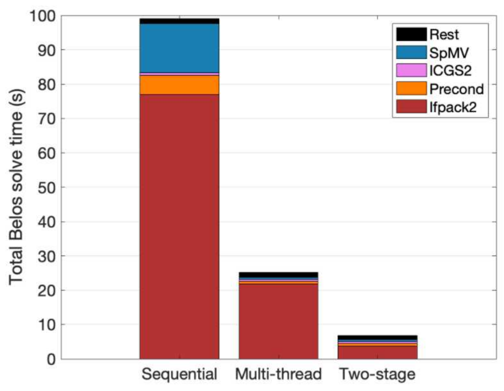


Figure 1: Total Trilinos GMRES solve time (GPU) for ten time steps of Nalu-Wind ABL 40 m simulation. Three preconditioners are compared: sequential Gauss-Seidel, multi-threaded Gauss-Seidel, and two-stage Gauss-Seidel. SGS was used both for the preconditioner (one sweep) and for the multigrid smoother (three sweeps). Two-stage SGS used just one Jacobi sweep for the inner iteration. Of the three preconditioners, two-stage SGS is the fastest and thus preferred.

All of these smoothers are available in Trilinos release (see Table 2) for both CPU and GPUs. Nalu-Wind with Trilinos may now employ any of these algorithms in any of the physics linear solvers, and further strong scaling results comparing these are presented below in the sequel.

hypre release has standard Gauss-Seidel. The NREL branch described in Table 2 has a GPU implementation of the two-stage smoother, and a CPU version is being developed.

7.3 STRONG SCALING

In this section, we examine the Nalu-Wind strong-scaling behavior using *hypre* and Trilinos. We focus on two meshes, the ABL 20 m simulation with 3.3 million pressure DOFs and the McAlister mesh, also with roughly 3.2 million pressure DOFs.

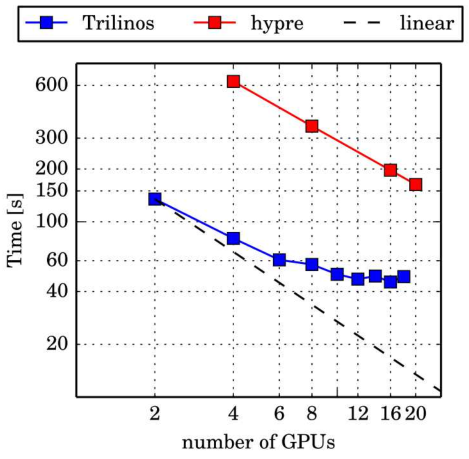


Figure 2: Total simulation time, ABL 20 m strong scaling study, Trilinos and *hypre*. Large time offset for *hypre* due to sparse matrix assembly on the CPU. Solvers on GPU.

7.3.1 hypre ABL Strong Scaling

Total simulation plot time for Nalu-Wind using the *hypre* solver stack is given in Figure 2. Several key aspects of the *hypre* solver stack should be noted. From the results presented in our weak scaling studies, based on the 20 m mesh, it was observed that the *hypre* one-reduce GMRES with C-AMG preconditioner requires a larger number of iterations to solve the pressure continuity problem, when compared to SA-AMG in Trilinos. In particular, we observe that the absolute wall-clock solve time for *hypre* , using a given number of GPUs, is larger than the Trilinos-Belos GMRES with SA-AMG preconditioner. However, the cost per iteration of the *hypre* one-reduce GMRES solvers is lower than the Trilinos-Belos MueLu solver. These results are plotted in Figure 3. Furthermore, the *hypre* continuity and the momentum solvers exhibit near perfect linear strong scaling. The *hypre* solvers continue to scale well beyond sixteen (16) GPUs.

We also note that the C-AMG set-up cost is not scaling and the RAP products are not yet on GPU in our implementation. However, these are now available in the UVM v2.18.0 from CASC-LLNL. Although the set-up has in general not scaled well even on parallel CPUs.

7.3.2 Trilinos ABL Strong Scaling

We now consider the strong-scaling behavior of Nalu-Wind using Trilinos solvers on the same ABL 20 m mesh. The solver for the continuity system is GMRES preconditioned with SA-AMG and a Chebyshev smoother. Similar to the two-level Gauss-Seidel approach, the Chebyshev smoother is based on a matrix-vector product kernel, which is fast and parallelizes well on the GPU. The three remaining equations (Momentum, TKE and Enthalpy) are solved via GMRES solver preconditioned by two-stage Gauss-Seidel.

Figure 2 plots the total Nalu-Wind runtime as a function of number of GPUs. The black dotted line is ideal strong scaling. We observe that strong-scaling begins to fall off at 8 GPUs. Figure 4 illustrates the strong scaling characteristics of just the Trilinos linear solve for the different physics equations. Two-stage

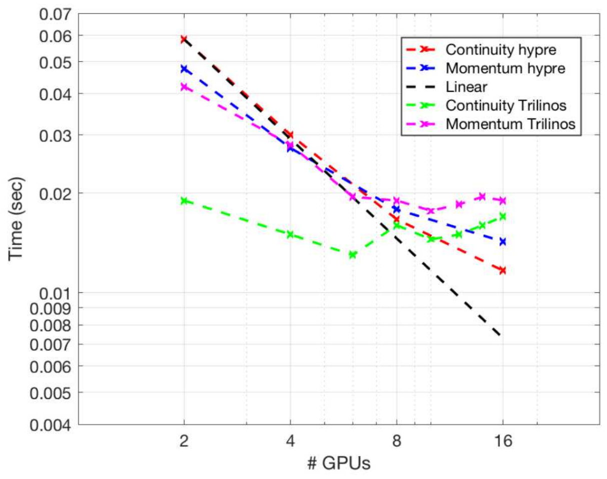


Figure 3: Time per solver iteration. ABL 20 m strong scaling. *hypre* and Trilinos.

Gauss-Seidel exhibits good strong scaling for both the preconditioner setup and apply phases, up to six (6) GPUs, and is close to linear at setup time. However, with the lighter workload, the GMRES symmetric Gauss-Seidel solver is less scalable (GPU time increases) for enthalpy and turbulent kinetic energy (TKE), and clearly exhibits a dependency on the number of compute nodes employed for computation. This is observed as the number of GPUs increases from 6 to 8 and from 12 to 14. The sub-linear scaling is less apparent for the momentum equations, but is still observable at the 12 to 14 GPU transition. The GMRES SA-AMG solver employed to solve the continuity system does not scale at setup and behaves similarly to the GMRES symmetric Gauss-Seidel momentum solver during the apply phase after six (6) GPUs.

7.3.3 hypre McAlister Strong Scaling

For the set of strong scaling GPU runs, the *hypre* stack is configured with the one-reduce GMRES solver and two-stage Gauss-Seidel momentum preconditioner and pressure continuity C-AMG smoother. The Nalu-Wind model was run on up to twenty (20) GPUs. Each MPI rank is associated with a single GPU. The assembly step is still performed on the CPU and the resulting matrices are copied to the GPU for the GMRES solvers.

The timing results are plotted in Figures 5a and 5b, respectively. The former represents the relative work time per iteration, or cost per unit of work given by $(time/(timesteps \times Picard \times iterations))$, whereas the latter is the absolute total run time. Timer details are summarized in Table 6. We observe that the assembly time on CPU decreases linearly with MPI ranks as expected.

For these runs, to determine if linear strong scaling has been achieved, the above times are normalized by the number of time steps, nonlinear Picard iterations per time step and the number of GMRES iterations in order to obtain the GPU time per solver iteration. These times are given separately for the continuity in Table 7 and momentum Table 8 equations and also plotted in Figure 5a. We observe that the momentum times exhibit almost perfect linear scaling, however, the continuity times decrease at a lower sub-linear rate. We attribute this to the workload and MPI communication overhead associated with the AMG V-cycle, although this clearly merits further investigation in our path forward.

7.3.4 Trilinos McAlister Strong Scaling

In this section we analyze the algorithmic strong scalability of the Trilinos solver stack for the McAlister simulation described in §3. At the time of writing this report, not all Nalu-Wind boundary conditions have been converted to run on GPUs. Hence, we primarily consider solver scalability in terms of the GMRES solver iteration counts. The goal is to assess the suitability of various Trilinos solver choices for the McAlister simulation, prior to being able to run the simulation on GPUs. Additionally, we are unable to assess GPU

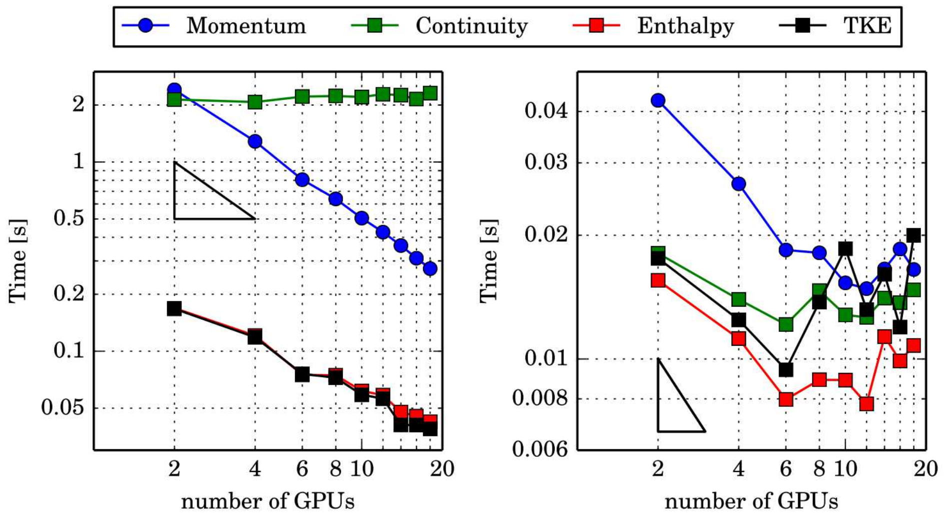


Figure 4: Trilinos solver setup and apply times, ABL 20 m strong scaling. Left panel displays cost of a single preconditioner setup per solve for a time step and right panel displays the cost of a single linear solver iteration per time step.

performance, choices made for the CPU may not be performant on the GPU. These tests were run on a single node of a small Sandia testbed with identical hardware (IBM Power9/NVIDIA V100) to that of Summit, with the exception that each node has only four GPUs.

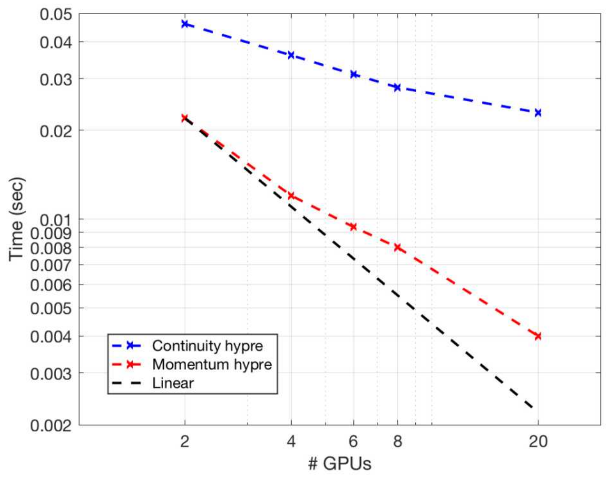
The continuity solve is GMRES preconditioned by either unsmoothed aggregation algebraic multigrid (PA-AMG) or smoothed aggregation algebraic multigrid (SA-AMG). Both AMG preconditioners use second degree Chebyshev polynomial smoothers and a coarse grid direct solve. Coarsening uses a classic drop threshold of 0.02. For the momentum, turbulent kinetic energy (TKE), and specific dissipation rate (SDR) transport equations, the solver is GMRES preconditioned with one of three different preconditioners: symmetric Gauss-Seidel (SGS), multithreaded symmetric Gauss-Seidel (MTSGS), and two-stage Gauss-Seidel (SGS2). The latter two algorithms are discussed in §5.3-5.4. The simulations were run on one MPI rank and even numbers of MPI ranks up to 20. In each run, the same preconditioner (SGS, MTSGS, or SGS2) was used for momentum, TKE, and SDR.

Overall simulation scaling times on CPU are given in Figure 6. The momentum, TKE, and SDR linear solves are GMRES/SGS2. The continuity solve is GMRES with either PA-AMG (blue line) or SA-AMG (green line). The momentum, TKE, and SDR linear solvers all demonstrate algorithmic scalability. As the processor count increases, there is flat or negligible growth in the iteration count. The continuity GMRES PA-AMG solver exhibits very mild variation in iterations. The continuity GMRES SA-AMG iterations have a pronounced spike for 10 and 12 MPI ranks, as a result of the solver reaching the maximum iteration limit. This accounts for the "hump" in the green curve in Figure 6. Finally, SA-AMG has modestly better iteration counts than PA-AMG. However, SA-AMG has an operator complexity of 1.94–2.10, whereas PA-AMG has a complexity of 1.27-1.28. Operator complexity is a measure of the flops required in a single AMG V-cycle, and values closer to one are generally faster (all other factors being equal). This accounts for the steeper slope in the PA-AMG (blue) plot.

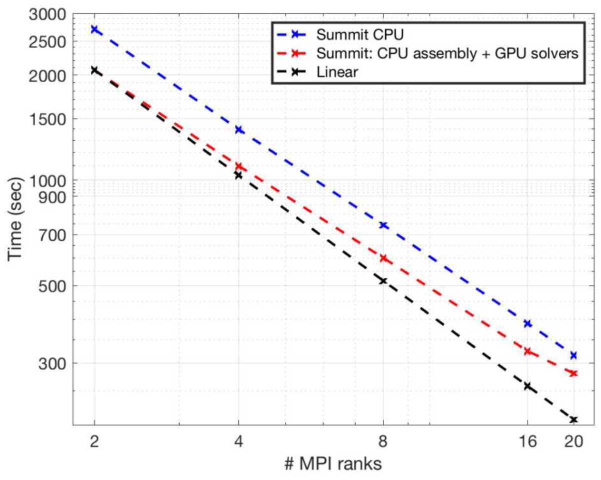
Detailed results can be found in §10 in Table 9. For each physics solver, there are three main columns labeled "SGS", "MTSGS", and "SGS2". These reflect the preconditioners used for momentum, TKE, and SDR. For continuity, there are two columns corresponding to whether PA-AMG or SA-AMG was used.

7.4 WEAK SCALING

Both the Trilinos and *hypre* solver stack teams have performed weak scaling studies, based on the ABL precursor simulation described in § 3.1. The mesh resolutions are 5 m, 10 m, 20 m and 40 m. Therefore, for



(a) Time per solver iteration.



(b) Total run time.

Figure 5: Time per solver iteration and total run time, McAlister strong scaling study (assembly on CPU and solvers on GPU), *hypre*.

each mesh the number of MPI ranks must increase by a factor of 8× in order to determine if the execution time remains flat or grows slightly while maintaining a constant problem size per MPI rank. We have allowed both the Trilinos and *hypre* solver stack teams some discretion in the selection of the number of MPI ranks to achieve the optimal performance, as the two solver stacks may differ in the optimal thread occupancy and sustained bandwidth utilization of the devices.

The total run time for Nalu-Wind coupled to the Trilinos and *hypre* solver stacks is plotted in Figure 7. The *hypre* assembly is not performed on GPU and leads to a large time offset. The deviation from linear scaling occurs earlier for Trilinos compared to *hypre* when the single node boundary of six (6) GPUs in surpassed on Summit.

7.4.1 *hypre* ABL Weak Scaling

The weak scaling study timings for Nalu-Wind and *hypre*–BoomerAMG are reported in Table 10 for the 40 m mesh on four (4) MPI ranks. These Nalu-Wind execution times indicate that the time per time step is currently close to 10 seconds for the ABL precursor simulation. To determine if the time per computational unit of work remains relatively constant, we have determined the total number of momentum and continuity solver iterations for the above simulations. Then the average time per GMRES solver iteration was computed is displayed in Tables 11 and 12.

A second set of weak scaling timings for Nalu and *hypre*–BoomerAMG are reported in Table 13 for the 40 m mesh starting from two (2) MPI ranks and increasing up to 1024 ranks. The average time per GMRES iteration and time per time step versus number of MPI ranks is plotted in Figure 8 for the ABL precursor sequence of meshes, to illustrate the weak scaling characteristics of the Nalu-Wind model for this problem. The average time per iteration remains fairly flat, indicating good weak scaling of the GMRES-AMG implementation. However, the time per time step grows slightly as the number of solver iterations gradually increases with the number of MPI ranks employed in the simulations.

In order to further reduce the *hypre* solver stack times for the ABL precursor simulation, the *hypre*-BoomerAMG PMIS coarsening algorithm was replaced with HMIS. All of the ABL simulations in this report with *hypre* employ HMIS coarsening. The number of pressure continuity iterations is reduced to 15 with the solve times dropping as a result. The number of C-AMG V-cycle levels is also set to six (6) in these runs. We note that the *hypre* stack continuity solve time at 512 GPUs is 24 seconds, and is less than the Trilinos time of 27 seconds.

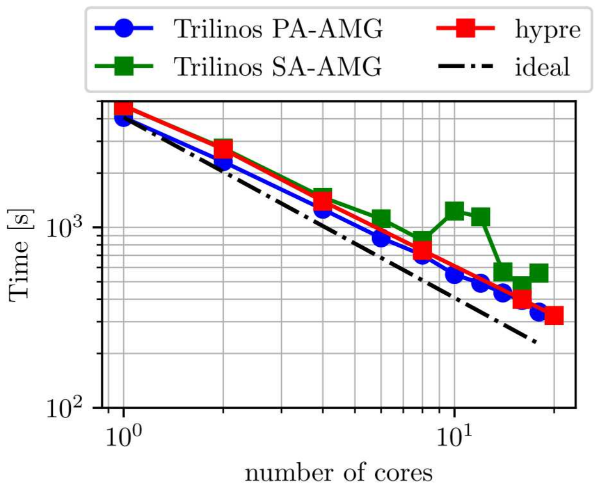


Figure 6: Total wall clock time, McAlister CPU strong scaling study. The red line corresponds to *hypre* release v2.18.2 run with defaults. The blue line corresponds to runs using Trilinos PA-AMG for continuity, the green line Trilinos SA-AMG. All other physics use two-stage Gauss-Seidel (SGS2).

7.4.2 Trilinos ABL Weak Scaling

The solver employed for the continuity system is GMRES preconditioned with SA-AMG and a Chebyshev smoother. Similar to the two-level Gauss-Seidel approach, the Chebyshev smoother is based on a matrix-vector product kernel, which is fast and parallelizes well on the GPU. The same approach is employed to solve the three remaining equations (Momentum, TKE and Enthalpy), and consists of a GMRES solver preconditioned by the two-stage Gauss-Seidel iteration presented in section 5.3. Based on the strong scaling study performed in section 7.3, a single GPU achieves good performance and load balance for the coarsest 40 m mesh resolution.

It can be observed in Table 15, that the GMRES solver, with two-stage Gauss-Seidel preconditioner, provides a mathematically scalable approach to solve the transport equations. The maximum number of iterations required to solve these linear systems is low and remains constant across all mesh resolutions tested. Algorithmically, however, the solver is not scaling as well as would be desired with the solve time roughly doubling from mesh to mesh. The GMRES and SA-AMG solver employed to solve the continuity system is exhibiting similar scaling properties to the GMRES and two-stage Gauss-Seidel. Further investigation of the kernels used in the solvers will be needed to assess what causes the time growth within the solver. The assembly, however, is both fast and scalable.

7.5 CPU VERSUS GPU

For comparison with the Trilinos based Nalu-Wind simulations and the *hypre* GPU runs described above, the stock *hypre* branch was coupled to Nalu-Wind for IBM Power9 CPU based simulation runs. In these *hypre* runs, the symmetric Gauss-Seidel (SGS) smoother was specified as the momentum preconditioner and pressure C-AMG smoother. The standard GMRES iterative solver is applied with higher orthogonalization costs, although these were not specifically measured. For the low number of MPI ranks these additional costs are not expected to be a significant factor. A breakdown of the Nalu-Wind *hypre* stack run times is given in Table 16. A plot of the total run times comparing the runs above with CPU assembly and solvers runs on the GPU with the CPU only runs was given in Figure 5.

Although the total simulation time for the Nalu-Wind *hypre* configuration is faster on 20 CPU cores than 4 GPUs (in part due to matrix assembly being performed on CPUs), the GPU pressure solve phase alone is 38% faster than the IBM Power9 CPU pressure solve phase on Summit.

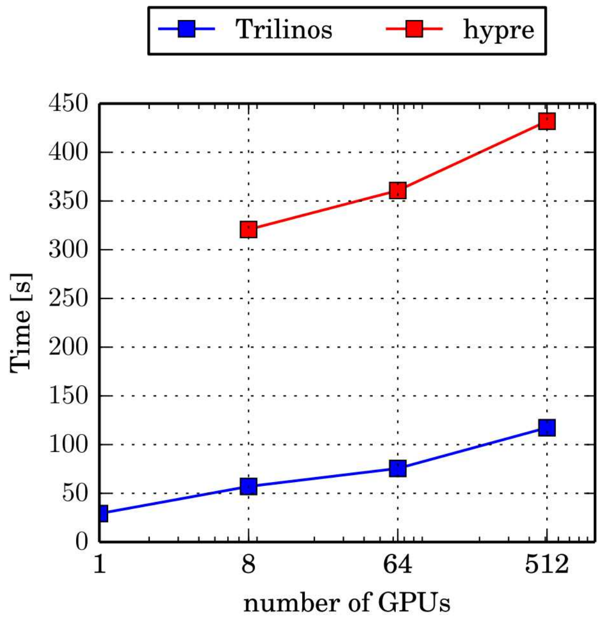


Figure 7: Total wall clock time, ABL weak scaling study, Trilinos and *hypre*. Large time offset for *hypre* due to sparse matrix assembly on the CPU. Solvers on GPU.

7.5.1 Trilinos Comparison on ABL Precursor

In order to assess the performance of the re-factored GPU implementation, the Trilinos solver stack was employed to perform weak scaling studies using the ABL precursor simulations on both CPUs and GPUs. To enable a straightforward comparison based on Summit hardware, it was decided to compare 7 CPUs per GPU for these runs because a Summit node contains 6 GPUs and 42 CPUs. In section 7.4.2: 1, 8, 64 and 512 GPUs or 7, 56, 448 and 3584 CPUs were employed for 40 m, 20 m, 10 m and 5 m mesh resolutions, respectively. Figure 9 displays both the time spent in the linear solvers (left panel) and the efficiency of the linear solvers (right panel) with respect to the number of degrees of freedom in the meshes (vs nodes). To avoid a complex analysis of the 40 m resolution which runs on a single GPU, the efficiency plot employs the 20 m mesh as a reference point, hence the efficiency measured is 1.0 for the 20 m mesh.

The more expensive linear solvers are associated with the momentum and continuity equations. It is observed that their GPU implementations result in substantial gains for both solvers. The momentum solve is approximately 2.0 times faster on GPUs, but the continuity solver is increasingly less efficient on larger meshes (higher number of nodes), achieving 4.0× speed-up on the 10 m resolution mesh and 1.5× speed-up on the 5 m resolution mesh.

8. PATH FORWARD

8.1 *HYPRE* PATH FORWARD

In order to meet our performance objectives, several critical components of the *hypre* linear-solver software stack will be addressed in Q3. These include but are not limited to sparse matrix assembly on device and a comparison between the UVM and mirror memory approach for *hypre* solver execution. The results of this comparison task will be shared with the *hypre* team and a decision on the memory model will be finalized and implemented in the master branch of *hypre* . All algorithmic developments such as the communication optimal GMRES and the GPU-accelerated smoothers will be brought forward per our discussions with the *hypre* team. Furthermore, our performance studies reveal some surprising results when running small

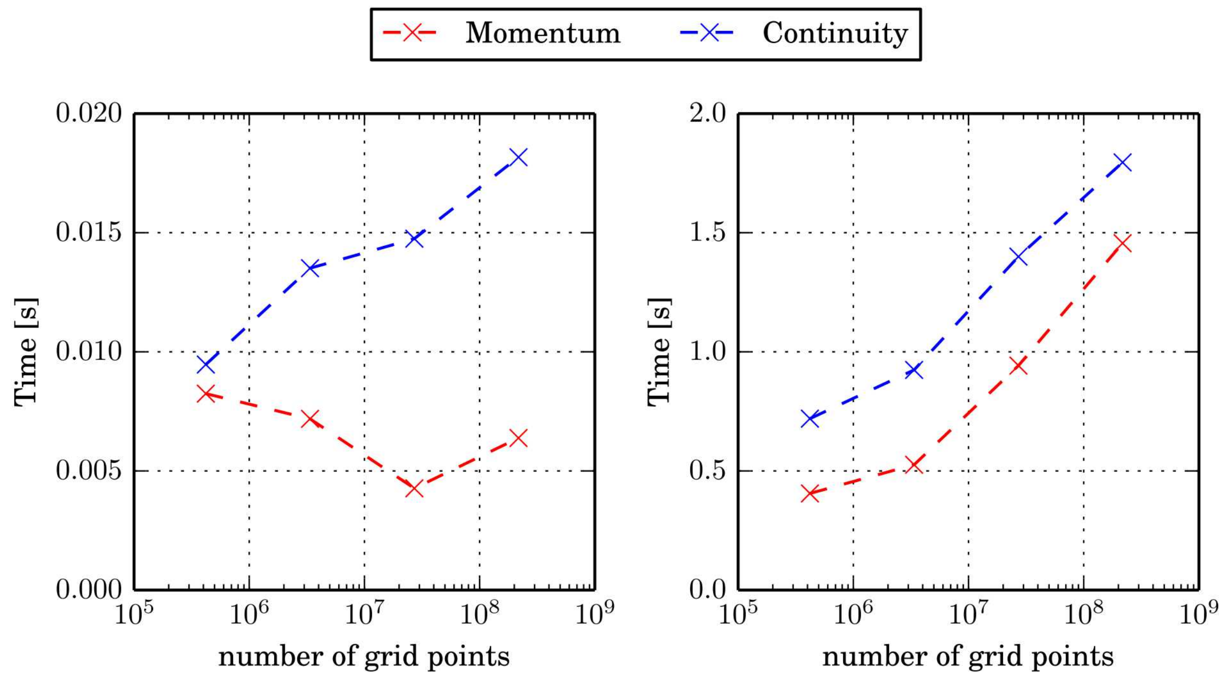


Figure 8: Weak-scaling study 40 m, 20 m, 10 m, and 5 m ABL meshes on GPU. *hypre*. Left: Time per iteration. Right: Time per time step.

problems on a low number of GPUs. In particular, the performance of the continuity solver for the 40 m mesh for a small number of GPUs is unexpected (see Tables 13 and 15 for the *hypre* and Trilinos results). One potential explanation is that the parameter choices for the *hypre* solver lead to suboptimal communication patterns on these small problems. A fully implemented and unified *hypre* solver stack including assembly and an optimal memory design will likely shed light on this issue. Moreover, improvements to fundamental performance-limiting algorithms such as the AMG set-up of the V-cycle and RAP products, will be explored in conjunction with the *hypre* team. As time permits, we will also investigate tools to move the *hypre* GPU solver stack code beyond NVIDIA-based architectures. This includes potentially porting more algorithmic components to Kokkos and/or using HIP to execute CUDA-centric developments on AMD architectures.

8.1.1 **hypre** *GPU linear-system assembly*

We have developed algorithms that allow us to assemble linear systems on the GPU for use in the *hypre* package (though not ready for deployment for this milestone). Ultimately, this will allow us to use GPU-accelerated versions of Trilinos and *hypre* in the same Nalu-Wind simulation where each back end is targeted to solve particular equations in the model. This implementation differs in some respects from the corresponding assembly methods used for Trilinos linear systems. In particular it was necessary to conform to existing APIs for both Nalu-Wind and *hypre* . This requirement forced us to write a novel algorithm that obeyed these API constraints and that can eliminate errors from non-reproducibility while providing significant acceleration over the existing CPU implementation.

We can show that our implementation assembles the matrix correctly for all equation systems in the ABL precursor simulation. The current implementation shows $7 - 10\times$ acceleration over a single CPU core when running on a single GPU. Moreover, we have a prototype multi-GPU implementation working for up to 16 GPUs. The performance results are promising. We are confident that additional performance gains can be achieved as the implementation is optimized.

The algorithm is structured with an initialization phase, and then 3 main computational stages. The initialization phase is run only when the mesh is (re)initialized. For stationary -mesh simulations, this occurs only once. The computational stages occur with every linear system assembly, i.e., every time the physics

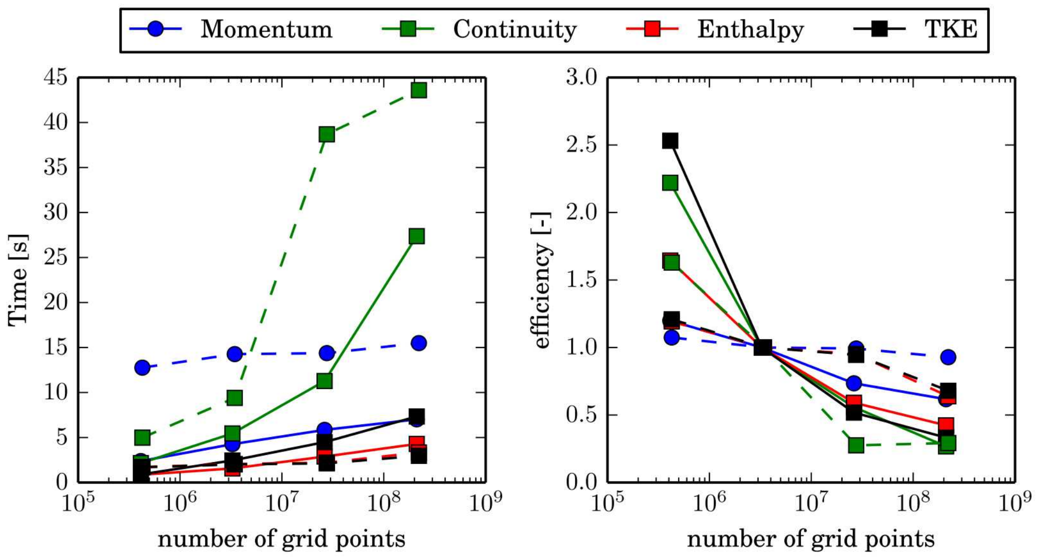


Figure 9: Comparison of the Trilinos solver stack on CPU and GPU. The left panel displays time spent in the linear solver while the right panel is the solver efficiency for the 20 m mesh as point of reference. Solid lines indicate GPU baseline simulations while dashed lines indicate CPU simulations.

equations require updating. The 3 stages are:

Stage 1: Device Coefficient Application

Stage 2: Assemble Compressed Sparse Row (CSR) matrix and RHS Vectors

Stage 3: Assemble *hypre* Linear System

Stage 1: Device Coefficient Application

In the first stage, device coefficient application, our algorithm walks over the unstructured grid entities, i.e., edges, faces, nodes, and populates a large COO (coordinate list) data structure. In this stage we are required to conform to the Nalu-Wind API, which uses the Kokkos programming model, to fill the coordinate list in device memory. This code runs on the GPU or CPU depending on which back end processor is targeted during compilation. It receives as input, data from other physics equations updates, as well previous versions of the current equation being solved. These data reside in the memory of the target device and they are assembled/populated into the target memory data structure, the coordinate list. During the initialization phase, the size of the coordinate list can be estimated to ensure sufficient space is allocated for this stage.

Stage 2: Device Linear System Assembly

In the second stage, the coordinate list data are then reordered and reduced into matrix/RHS elements via device algorithms. The algorithms developed for this stage were written in the CUDA programming language. The primary workhorse of this stage is the stable sort_by_key algorithm. We use stable sort_by_key to reorder the coordinates lists into bins that can then be reduced to matrix/RHS elements. The stable sort_by_key algorithm can be utilized in a manner that allows us to compute matrix/RHS elements by summing from smallest (in magnitude) elements to the biggest via compensated summation. This approach minimizes round-off error. A faster version of the sort, where only row/column coordinates are considered, is also available. Once the sort is completed, several custom kernels scan the sorted data structure and assemble the matrix/RHS elements. At the end of this stage, the assembled matrix/RHS is split into an owned component

	Momentum	Continuity	Enthalpy	Turbulent Kinetic Energy
CPU Assembly	38.2s	22.8s	27.0s	36.5s
GPU Assembly	5.24s	2.41s	2.84s	2.70s
Acceleration	7.3×	9.5×	9.5×	13.5×

Table 3: Single CPU vs Single GPU assembly on the NREL Eagle HPC System. Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz vs NVIDIA Tesla V100. ABL precursor simulation 40 m mesh. *hypre* solvers

that has rows on this MPI rank and a shared component, which has rows needed on other MPI ranks. This splitting is also executed on device.

Stage 3: Hypre Linear System Assembly

In the final stage, the assembled CSR matrix, built in stage 2, is then used to build the *hypre* linear system. We use the *hypre* API methods

HYPRE_IJMatrixSetValues

HYPRE_IJMatrixAddToValues

HYPRE_IJVectorSetValues

HYPRE_IJVectorAddToValues

to build the matrix/RHS in 2 steps. First we apply HYPRE_IJMatrixSetValues, HYPRE_IJVectorSetValues to set the matrix/RHS of the owned rows on the calling MPI rank. In order to set the off-rank matrix/RHS elements, we then call HYPRE_IJMatrixAddToValues, HYPRE_IJVectorAddToValues using the shared matrix/RHS elements as input. The beauty of this implementation is that it completes the assembly in 4 *hypre* API calls. It then leverages the internal-messaging structure of *hypre* to properly build the full matrix/RHS.

In the development branch of Nalu-Wind, where this is being implemented, we have not yet integrated a GPU accelerated version of *hypre* . Therefore, we copy the matrix/RHS built in stage 2 back to the host. Then we call host versions of the *hypre* APIs shown above. It is important to note that because we are using a non-GPU *hypre* branch in this development, the solves also run on the CPU. In the subsequent performance studies, we report only assembly times rather than per time step. We expect substantial speed gains to be achieved once this stage is moved to a pure device implementation in the assembly and solve. Current timings show that the CPU versions of the *hypre* APIs consume upwards of 30% of the total assembly time.

Performance

For production Nalu-Wind simulations, we wish to enable the use of both Trilinos and *hypre* solves concurrently (e.g., Trilinos for momentum and *hypre* for continuity). However, in order to test the performance of this initial implementation, we ran the ABL precursor simulation where all linear systems were solved via the *hypre* packages. The results for 1 CPU Assembly vs 1 GPU Assembly are shown in Table 3. The ABL precursor simulation was run for 10 time steps, four (4) Picard iterations per time step. The time reported in Table 3 is the total time spent in assembly for all 40 assemblies (including initialization). The initial results show a good initial acceleration versus 1 CPU core.

Fine-grained performance analysis of the GPU assembly for 1 device (Table 4) indicates the most likely places to achieve performance gains. While the bulk the of the time is spent in stage 2, and the majority of this time is spent sorting the COO lists, significant fractions are spent copying the data back to the host and then applying the CPU *hypre* APIs for matrix assembly. GPU accelerated versions of the *hypre* APIs listed above could provide immediate benefits. These ratios are consistent for 2 and 4 GPU simulations thus suggesting that performance gains will extend to large multi-device simulations.

In Table 5, we show the performance of the *hypre* GPU assembly for the Continuity Equation on the 20 m mesh. We compare the results of 1 GPU to 8 CPU cores, 2 GPUs to 16 CPU cores, and 4 GPUs to 32 CPU

	Momentum	Continuity	Enthalpy	Turbulent Kinetic Energy
Stage 1	40	17	24	26
Stage 2	38	47	44	43
Stage 3	22	36	32	31

Table 4: Percentage of GPU Assembly time spent in each stage for the ABL precursor simulation 40 m mesh.

CPU Cores/GPUs	8/1	16/2	32/4	#/8	#/16
CPU Assembly	45.2s	22.9s	11.4s		
GPU Assembly	41.1s	11.2s	6.12s	3.38s	1.88s

Table 5: *hypre* GPU and CPU assembly total assembly time on the ABL precursor simulation 20 m mesh for the continuity equation.

cores. The CPU results are pulled from Tables 14, 13 and 10 where we add the assembly and load times together. We also show the assembly time for 8 and 16 GPUs though we do not have a corresponding CPU comparison. Overall, the results are encouraging though the results for 1 device are curious. In this case, we suspect that the device memory is nearly full from all the systems being solved on one device. Device kernels that use a large number of registers can "spill" into L2 caches and global memory. This prevents core algorithms from running at full occupancy. However, the results for 2 and 4 GPUs, which are 2x faster than 16 and 32 cores respectively, are highly encouraging and we expect continued performances gains as more devices are leveraged.

Next Steps

We have successfully run our implementation on the 20 m ABL precursor simulation on up to 16 GPUs on the NREL Eagle HPC System. For this simulation, only the continuity equation was assembled and solved (via CPU) in *hypre*. The remaining physics equations were assembled/solved via Trilinos on the GPU. The results for total assembly time show reveal that 1 GPU is roughly equivalent to 16 CPU cores (for 2 or more GPUs). At this point, the bottleneck in the simulation for this development branch is now the CPU solve. Integrating the GPU accelerated *hypre* will lead to a major gain in the overall simulation times.

The most important next step is to show that our implementation runs successfully on more than 16 GPUs. This is a straightforward task that will implemented in the immediate future. In addition, performance profiling via the NVVP (NVIDIA Visual Profiling) tool reveals key areas where memory can be saved and speed gains can be attained in the device kernels in Stage 2. Finally, we need to resolve any issues that may arise when running our implementation at exascale. Once this work is completed, we will merge this development branch into the Nalu-Wind master branch for production runs. Simultaneously, we will integrate a GPU build of *hypre* and perform the final assemblies/solves on the GPU. We will then be able to test whether or not we can meet our performance objectives on large simulations.

The code development outline above will also enable us to do some important numerical studies. Our efforts to implement the sparse matrix assembly algorithms for the Hypre stack on GPU have revealed interesting numerical sensitivity. The PIs (Mullowney) have created a solution concordance tool that can track the agreement between CPU and GPU solutions. In addition, this tool can find differences in the assembled matrices and right-hand sides. For example, we have compared atomic summations versus sorted COO and reductions for the matrix assembly and found sensitivity to perturbations in the right-hand side vector that led to large relative residual errors ($> 1e-5$) in the GMRES solutions. With a fully integrated GPU-based software stack, our team will be in a position to verify that the level of error is reduced below the GMRES relative residual solver tolerances. The sensitivity is related to the condition number of the matrices as described earlier in Section 5.

8.1.2 Memory Model Comparison: UVM vs Mirror

The *hypre* GPU scaling studies were executed on a branch of *hypre* that uses a mirror model for the GPU memory. That is, the GPU data structures for the matrix, RHS, solutions, and other key quantities in the iteration have a corresponding version in host memory. In contrast, the current version of *hypre* master branch utilizes the unified virtual memory (UVM) model. Both the host and device can access data in the UVM memory space. Specific prefetch commands can be issued to suggest to the CUDA runtime to move the data to specific devices prior to execution. There are strengths and weaknesses to each approach. The UVM programming model lends itself to simpler code that is easier to manage. The mirror model allows the programmer to explicitly dictate when and where data moves between host and device memory spaces for initialization and inter-GPU communication.

The mirror memory approach relies on the theory that explicit control leads to better performance. Proponents of UVM suggest that the performance gains are minimal and do not warrant the additional code complexity. We intend to test these two paradigms by performing strong and weak scaling studies using a branch of *hypre* that supports both of these memory models. Work is already well underway to merge these two paradigms into a single branch off of the *hypre* master branch. Once this is completed, we will be in a position to perform these studies. The results of this analysis will be presented to the *hypre* team and a decision on the forward path will be discussed. We expect to complete this work in FY20 Q4.

8.1.3 New algorithms

The *hypre* C-AMG set-up of the V-cycle and RAP products to construct coarse level matrices remained on the CPU. These triple-products are costly and therefore we should adopt the *hypre* version of these for GPU or implement our own, possibly based on the NVIDIA cuSparse library. Further improvements to the AMG smoothers are possible through asynchronous iterations for the two-stage smoother. Clearly, our results suggest that the two-stage preconditioner and smoother are very efficient. More investigation is warranted along with new and more efficient algorithms for triangular solvers.

Because the two-stage preconditioner is a form of polynomial preconditioner, the short recurrence Lanczos type MINRES iteration may be applicable as a replacement for GMRES as the momentum solver. The resulting iterations require fewer floating-point operations and have the potential to further reduce the Nalu-Wind execution times.

8.1.4 Device portability and beyond

For the Trilinos solver stack, the Kokkos abstraction layer also relies on GPU unified-memory for the NVIDIA Volta V100 GPU architecture. Key computational kernels may also be based on cuBLAS and cuSparse library implementations from NVIDIA along with Kokkos-Kernels. Moving forward to Intel (Aurora) and AMD (Frontier) GPUs, Aurora will have unified memory and rely on C++/SYCL compiler language support to generate GPU kernel code and may support the OpenCL standard for GPU kernels. Whereas AMD will support abstraction layers such as Kokkos and Raja, along with the HIP interface for GPU kernels, which is similar to CUDA and includes hipBLAS and hipSparse libraries. In all cases, the CPU to GPU memory transfers must be considered for performance and portability.

8.2 TRILINOS PATH FORWARD

The work performed for this milestone allowed us to test the software stack on a GPU architecture and to establish a performance baseline for the linear solver components. This baseline will help us guide some of the future work centered on the performance of the Trilinos stack on GPUs, while other clear algorithmic improvement will be tackled simultaneously.

8.2.1 Fine grained performance analysis

The first task required after the finalization of this milestone will be to understand why the solvers in the Trilinos stack have limited strong scaling abilities, especially the multigrid solver setup, but also the solver apply phase in general as seen in Figure 4.

Multiple technical issue may be the cause of this behavior:

- slow MPI communication,
- unintended memory movement between device and host,

- under-performant use of temporary allocations in kernels,
- poor occupancy/efficiency of device kernels,
- kernel launch overhead costs.

Differentiating between the multiple causes listed above will likely require the development of multiple performance test case and metrics and to isolate individual components of the code. A side effect of this effort should be an enhanced understanding and testing of existing pathways in our software.

8.2.2 Integration and testing on GPU

In general the work performed on this milestone uncovered difficulties in the integration and testing of various components of the software. New physics kernels developed on GPU to compute Jacobians, forcing terms and boundary conditions have allowed us to complete the ABL simulations on Summit's GPUs but have also required substantial amount of attention from the linear solver team to assess and correct for proper simulation results. A strategy that allows both new physics and kernels to be implemented while avoiding regression on GPU will be key to effectively deliver on future milestones.

8.2.3 Linear system and preconditioner setup optimization

Some additional improvements will be developed to setup the Trilinos linear systems more efficiently. First work is already underway to use a single graph for the scalar equations and a second one for the momentum equation. This will largely reduce the setup cost which becomes more important with moving meshes and more complex geometries that cannot allow us to reuse the linear system and preconditioner from time-step to time-step. Part of this work has already been implemented but still requires significant improvement to become practical for production runs. Second the Trilinos stack will move to a segregated momentum system on GPU as is already the case on CPU. This will again improve the linear system setup time and reduce the overall memory footprint of our simulations.

Finally more exploratory work might take place to allow the multigrid preconditioner to reuse part of its symbolic computation between non-linear iterations and potentially between time-steps.

8.2.4 New architecture performance assessment

As presented in section 6.1 the Trilinos stack relies heavily on Kokkos and Kokkos-kernels to mitigate some of the challenges associated with the rapidly changing hardware environment within the DOE computing infrastructure. However there will still be a need to test and integrate new features exposed through Kokkos (tasks, CUDA graph, streams, etc..) as well as new algorithms in Kokkos-Kernels. Finally some native algorithms are implemented directly in various packages of the Trilinos stack, for instance the aggregation routines in algebraic multigrid. These algorithms, while portable because they are written with Kokkos, will need at least some tuning for performance on upcoming accelerators.

8.2.5 Iterative solver and local kernels improvement

Some work will be performed around the linear solver performance, at least three directions will be assessed before an execution plan can be drafted more precisely:

- potential gains can be achieved substituting MINRES for GMRES,
- some optimization can be performed in the iterative solver for multiple right hand sides used in the segregated momentum linear system,
- work done on mixed precision arithmetic for the orthogonalization process in Belos will be assessed for performance in Nalu-Wind.

Additionally, further testing of the damping parameter and number of inner iterations with the two stage Gauss-Seidel preconditioner will be conducted to improve performance. Further gains with the two-stage Gauss-Seidel preconditioner will be attempted by tuning the communication implementation in the global

SpMV kernels while the new interface to cuSparse in the local SpMV kernel will be assessed for performance. Finally a domain decomposition approach based on the FROSch method recently implemented in Trilinos' Ifpack2 package will be tested and tuned as a preconditioner for the momentum solver.

8.3 MIXED-PRECISION GMRES AND PRECONDITIONERS

With a view towards even higher sustained GPU performance (NVIDIA Ampere 300 TeraFlops), the use of mixed-precision FP-16 (TensorCore), FP-32 and FP-64 floating point units on the GPU should be considered. The PI's (Thomas, Swirydowicz, Yamazaki) have developed, in collaboration with Erin Carson at Charles University in Prague, a new mixed-precision GMRES solver that performs all inner-products in single precision and yet retains double precision accuracy. The same approach may also apply to Trilinos and *hypre* AMG preconditioners and these are all being explored within the ECP xSDK multi-precision efforts. A publication in IJHPCA detailing these advances is forthcoming from the xSDK team. We plan to explore mixed-precision arithmetic in solvers in the time-frame 2021-23.

8.4 PATH SUMMARY

The future plans for the *hypre* solver stack include the following. First, complete the interface of Nalu-Wind to the new reproducible *hypre* sparse matrix assembly GPU code. Then evaluate the GPU performance of the full model based on this code. Then with the *hypre* LLNL team, integrate these developments into the main *hypre* branch, including the memory model. These are FY20 Q4 activities which will also include an investigation of the growth in solver iterations at higher MPI ranks and testing of the Schwarz algorithms for decoupled solves.

With the new Aurora and Frontier machines becoming available in 2021, we will explore the performance of Nalu-Wind on the Intel Xe and AMD GPUs. In the same FY21 Q4 time frame we will report on the use of mixed precision solvers developed in collaboration with the xSDK team. We will also explore solver core algorithm improvements to improve GPU performance on the new GPU architectures. These may also include new preconditioners, and techniques to improve nonlinear convergence. Finally, we will continue to explore and improve GPU performance with a view towards multi-turbine simulations in FY22 and report on progress for FY22 Q4.

The *hypre* development objectives for FY20-FY22 are described below:

- FY20 Q4: We will complete the interfacing of on-GPU assembly to the Nalu-Wind *hypre* solver stack and test. The linear-system solvers and preconditioner setup will be evaluated in the model and performance documented as compared to the current FY20 Q2 results. We will investigate the solver scaling and iteration count increases with the objective of mitigating iteration count growth. The existing *hypre* code branch will be merged with the main development branch from CASC-LLNL. In particular, the C-AMG set-up with RAP triple matrix products will be included.
- FY21 Q4: We will begin to explore and evaluate the performance of Intel Xe and AMD GPUs for Nalu-Wind. The Schwarz based decoupled solvers will be compared with the current FY20 Q2 solver scaling. Work will continue to improve the existing GPU core solver algorithms. A further goal is to explore mixed precision GMRES with the SNL and LLNL mathematics and xSDK teams. A key question to address is level of effort required to bring high performance, mixed-precision solvers into production Nalu-Wind simulations.
- FY22 Q4: We will further expand mixed precision investigations to preconditioners with xSDK. We will evaluate AMD GPU programming models and performance along with GPU performance portability.

The Trilinos development objectives for FY20-FY22 are described below:

- FY20 Q4: The segregated momentum system will be implemented for the GPU code path in Nalu-Wind. It will subsequently be compared to the results obtained for this milestone. Additional work will be conducted to use a single CrsGraph share by all matrices in the simulation to reduce the amount of communication during the linear system setup. Initial performance investigation will be made to improve the strong scaling of SpMV based preconditioner on GPU.

- FY21 Q4: The work on iterative solvers using MINRES, mixed precision GMRES and multiple right hand sides will be conducted. The multigrid setup performance on accelerators will be investigated especially with respect to strong scaling. The FROSch domain decomposition method will be explored as a potential alternative preconditioner for the momentum solver. Initial tests with of the HIP and SYCL backends from Kokkos and Kokkos-Kernels will be performed to ensure portability of the stack to Frontier and Aurora machines.

- FY22 Q4: Further portability studies will be conducted to assess the need for new algorithmic tuning.

9. CONCLUDING REMARKS AND NEXT STEPS

Nalu-Wind solves the acoustically incompressible Navier-Stoke equations, where mass continuity is maintained by an approximate pressure projection scheme. The governing equations for momentum, pressure, and scalar quantities are discretized in time, where an outer Picard fixed-point iteration is employed to reduce the nonlinear-system residual at each time step. In this milestone report we focused on two representative fluid dynamics problems related to wind energy, namely a atmospheric boundary layer (ABL) precursor simulation, and a fixed-wing simulation. These two problems were employed in both weak and strong scaling studies of Nalu-Wind performance on the ORNL Summit supercomputer. Both the Trilinos and *hypre* solver stacks were evaluated. Considerable new knowledge has been gained concerning the choice of numerical algorithms and the achievable performance on Summit. Our focus was on the NVIDIA Volta V100 GPU, however, comparisons with IBM Power9 based CPU runs were also performed.

The Nalu-wind simulation time is dominated by the time needed to setup and solve the linear equations associated with the linearized governing physics (e.g., momentum and pressure equations) at each time step. Set-up time for *hypre* C-AMG and MueLu SA-AMG have a large component associated with RAP sparse matrix triple products to create the V-cycle hierarchy. A Krylov method, like the Generalized Minimal Residual (GMRES) [12] iteration, is used to solve the linear systems, through either the Trilinos or *hypre* solver software stack. For solving the momentum systems, one typically employs Gauss-Seidel (GS) or symmetric Gauss-Seidel (SGS) as a preconditioner to accelerate the convergence of the Krylov solver. The pressure systems are solved using an algebraic multigrid (AMG) preconditioner, and GS is often applied as a smoother to relax or remove oscillatory components of the solution error (e.g., those associated with the large eigenvalues of the system), which the coarse-grid solver fails to eliminate. A scalable solver would maintain a roughly constant number of iterations to solve the linear systems. Equivalently, the solver time should remain constant as the number of MPI ranks increases while weak scaling to a higher total number of nodes. The absolute continuity solver times, although improved with HMIS coarsening for *hypre* , are slightly larger than Trilinos for low MPI ranks (see 15 and Figure 9). The relative solver times as measured by time per iteration are quite close for both stacks. It can be observed in our ABL weak scaling timing data on Summit that the continuity GPU solve time increases from 2 seconds to 27 seconds for Trilinos, whereas the *hypre* continuity solver time increases more gradually from 10.5 seconds on 8 GPUs to 24 seconds on 512 GPUs (for details see Figure 8 and Figure 9, as well as Table 14 and Table 15). This latter result can be attributed to the increasing number of GMRES iterations for *hypre* . Similarly, an important result of our studies is that both the Trilinos and *hypre* solvers exhibited an increasing number of iterations at higher numbers of MPI ranks on GPUs. This merits further investigation for scaling to the node counts necessary to meet the Nalu-Wind simulation time targets on Exascale class machines.

On a distributed-memory computer, both Trilinos and *hypre* implement a hybrid variant [1] of GS where the boundary information is exchanged to compute the residual vector, but then each MPI process performs a fixed number of the local GS steps (typically one step), independently. This hybrid GS is shown to be effective, and scalable, on many problems (e.g., the iteration counts roughly stay constant with the increasing process count). However, to perform the local GS, each process still requires a local sparse-triangular solve, which is inherently challenging to parallelize on GPU architecture. Although several techniques have been proposed to improve the parallel performance, the local sparse-triangular solve could still be the bottleneck for the parallel scalability of the solver, and in turn, of the simulation. In this report, we examined the performance of an iterative two-stage Gauss-Seidel algorithm as a replacement for the triangular solve. We found that the two-stage algorithm was much faster on the GPU and improves the overall scalability of Nalu-Wind, resulting in lower run-times as well. For the continuity equation solver in Trilinos-MueLU with SA-AMG, a Chebyshev smoother is applied, which relies on matrix-vector products. The two-stage Gauss-Seidel smoother can also be applied and we plan to evaluate the performance of this combination.

For end-to-end simulation times, the Trilinos stack leads to lower times for ABL at lower numbers of GPUs. Although this comparison is based on CPU assembly for the *hypre* stack. The times for McAlister will be obtained once code updates related to boundary conditions are available. Both the Trilinos and *hypre* solvers achieve comparable compute times per solver iteration on the NVIDIA Volta V100 GPU. However, we observed that the *hypre* solvers continue to strong scale to higher numbers of MPI ranks when compared to the Trilinos solvers. This certainly merits further investigation and may be due to several factors such as the MPI libraries on Summit combined with the need for a coarse grid solver for the elliptic pressure problems. We have demonstrated close to ideal linear strong scaling characteristics of the *hypre* stack using a combination of the IBM Power9 CPU for sparse matrix assembly and the GPU for the solver components (momentum, pressure, TKE, enthalpy and specific dissipation rate). Our next step forward is to complete the interface of our reproducible and high-accuracy *hypre* matrix assembly on GPU to the current Nalu-Wind code base. Once completed, we plan to report strong-scaling results on Summit with the entire Nalu-Wind *hypre* stack on GPU for the 2020 Q3 milestone report.

Our weak scaling studies have provided initial estimates of the sustained model integration rates on high processor counts (up to 2048 GPUs on Summit). An integration rate of $40,000$ time steps of the ABL precursor simulation, completed within a two-hour time window, is likely achievable and within reach at the target mesh resolutions of 10m or less. However, it is clear that $\mathcal{O}(100-1000)$ GPUs will be required for such an integration. Most notably, our studies revealed that the GPU to CPU mirror-memory management model combined with CUDA allowed us to achieve very high performance levels on Summit and for the McAlister blade problem, run on up to 128 GPUs, we observe good strong scaling characteristics for momentum and other transport equations, but not for continuity. Because the CASC-LLNL team are currently employing the unified virtual memory (UVM) model, possibly with pre-fetching, these two approaches should be further investigated to determine the best path forward for performance without sacrificing portability on the new Intel (Aurora) and AMD (Frontier) exascale architectures. The strong scaling characteristics of Nalu-Wind and Trilinos, when using Kokkos and the UVM model, should be further investigated.

For future solver development, the segregated momentum solver needs to be implemented for GPU and tested in Trilinos on ABL and McAlister problems. In addition, a block-Krylov method is applicable for this problem and will be implemented and tested. For polynomial-type preconditioners such as the two-stage Gauss-Seidel algorithm, the short-recurrence MINRES Krylov iteration exhibits excellent convergence rates. Further work on MINRES is justified and warranted in order to reduce the cost of the momentum equation solvers and thereby also reduce the end-to-end solution time of Nalu-Wind. The *hypre* C-AMG set-up phase with RAP sparse matrix triple products should be incorporated into our Nalu-Wind code base. For the integration-rate targets required to meet the criteria for wind-turbine simulations consisting of multiple turbines in the 2020–23 time frame, the strong scaling characteristics of the current Nalu-Wind solvers must be improved. We note that the deviation from linear scaling in Figure 5 occurs earlier for the GPU implementation of the momentum and continuity solvers. Clearly, the pressure continuity solver for *hypre* , and to a greater extent Trilinos, are scaling sub-linear. Indeed, the CPU-based simulations reported in our earlier milestone reports and our recent paper, Thomas et al. [17], exhibit far better strong scaling characteristics. The strong scaling of the Trilinos stack should be examined in the near future for the McAlister problems. The proposed Schwarz subdomain approach may reduce the number of nodes per GMRES solve, however, the strong scaling characteristics of this algorithm have yet to be established for Nalu-Wind and requires further investigation by the PI's.

10. APPENDIX

	continuity					momentum					
GPUs	init	assm	load	solve	iters	init	assm	load	solve	iters	
2	0.32	283.05	4.20	59.47	31.85	0.35	423.84	10.86	36.46	8.22	
4	0.18	134.57	12.48	46.63	31.98	0.19	208.06	7.77	20.14	8.22	
6	0.13	87.56	8.15	38.09	31.10	0.13	136.04	8.46	15.49	8.22	
8	0.10	65.03	7.93	35.36	31.93	0.10	100.63	6.17	13.09	8.22	
12	0.08	42.59	6.69	33.39	32.93	0.08	65.28	5.71	9.88	8.22	
16	0.06	32.18	5.78	30.29	31.22	0.07	48.64	4.93	7.75	8.22	
18	0.07	28.55	5.28	32.33	31.02	0.08	42.66	5.06	7.97	8.62	
20	0.05	25.32	5.01	27.79	30.27	0.08	38.47	4.81	7.04	8.22	
24	0.04	21.02	4.22	28.34	32.15	0.05	31.81	3.83	6.25	8.22	
32	0.04	15.88	4.09	28.48	33.59	0.04	23.70	4.92	5.42	8.63	
64	0.03	7.79	2.57	25.60	32.05	0.03	11.59	3.74	5.03	8.63	
128	0.03	3.89	2.07	21.32	34.02	0.02	5.69	2.50	3.97	8.83	

	Turbulent Kinetic Energy					Specific Dissipation Rate					
GPUs	init	assm	load	solve	iters	init	assm	load	solve	iters	Total Time
2	0.32	256.69	10.52	14.05	4.1	0.32	256.25	15.55	15.37	4.1	2064.88
4	0.18	128.42	8.82	7.64	4.2	0.18	126.85	11.80	8.65	4.1	1098.28
6	0.13	82.92	6.82	5.64	–	0.13	83.89	6.61	6.39	–	775.71
8	0.10	60.44	8.17	4.35	4.5	0.10	60.99	9.25	4.97	4.2	613.84
12	0.08	40.08	5.64	3.46	–	0.08	40.48	5.75	4.18	–	431.28
16	0.06	29.76	4.85	2.79	4.7	0.09	30.14	5.01	3.25	4.8	344.22
18	0.06	26.54	3.95	2.77	–	0.06	26.55	4.24	3.29	–	324.04
20	0.05	23.94	3.55	2.30	4.7	0.05	24.10	3.93	2.75	4.3	290.71
24	0.04	19.59	3.29	2.04	–	0.07	19.59	3.52	2.49	–	254.23
32	0.03	14.77	4.40	1.69	–	0.04	14.64	4.67	2.07	–	219.48
64	0.02	6.90	2.50	1.30	–	0.02	6.97	2.52	1.84	–	145.78
128	0.02	3.35	1.99	0.91	–	0.02	3.39	2.01	1.29	–	101.84

Table 6: McAlister strong-scaling execution time breakdown for Nalu-Wind with *hypre* solver stack. Init, assembly, load complete times (on CPU). Solve times and iterations (on GPU), for momentum and continuity equations. Turbulent kinetic energy and enthalpy, assembly on CPU and solve time on (GPU)

Ranks	Av. iter	Min iter	Max iter	Total iter	Total time	Av time per iter
2	31.85	30	36	1306	59.4680	0.0455
4	31.98	31	35	1311	46.6282	0.0356
6	31.10	31	35	1275	38.0866	0.0299
8	31.93	30	37	1309	37.5335	0.0287
12	32.93	31	37	1350	34.1913	0.0283
20	30.27	30	37	1241	28.4876	0.0230

Table 7: McAlister continuity GPU times for *hypre* stack. Time per solver iteration.

Ranks	Av. iter	Min iter	Max iter	Total iter	Total time	Av time per iter
2	8.22	4	12	1644	36.4600	0.0222
4	8.22	4	12	1644	20.1353	0.0122
6	8.22	4	12	1644	15.4857	0.0094
8	8.22	4	12	1644	13.1663	0.0080
12	8.22	4	12	1644	11.2095	0.0068
20	8.22	4	12	1644	7.2457	0.0044

Table 8: McAlister momentum GPU times for *hypre* stack. Time per solver iteration.

REFERENCES

[1] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, *Multigrid smoothers for ultraparallel computing*, SIAM J. Sci. Comput, 33 (2011), pp. 2864–2887.

[2] E. Chow, H. Anzt, J. Scott, and J. Dongarra, *Using Jacobi iterations and blocking for solving sparse triangular systems in incomplete factorization preconditioning*, Journal of Parallel and Distributed Computing, 119 (2018), pp. 219–230.

[3] A. Frommer and D. Szyld, *H-splittings and two-stage iterative methods*, Numerische Mathematik, 63 (1992), pp. 345—356.

[4] ——, *Asynchronous two-stage iterative methods*, Numerische Mathematik, 69 (1994), pp. 141—153.

[5] K. Świrydowicz, J. Langou, S. Ananthan, U. Meier-Yang, and S. Thomas, *Low synchronization Gram-Schmidt and GMRES algorithms*, (2020).

[6] B. Kelly and S. Rajamanickam, *Multicolor block Gauss-Seidel using Kokkos*, SIAM Conference on Parallel Processing in Scientific Computing, 2020.

[7] P. Lanzkron, D. Rose, and D. Szyld, *Convergence of nested iterative methods for linear systems*, Numerische Mathematik, 58 (1991), pp. 685—702.

[8] S. J. Leon, Å. Björck, and W. Gander, *Gram-Schmidt orthogonalization: 100 years and more*, Numerical Linear Algebra with Applications, 20 (2013), pp. 492–532.

[9] P. Lin, M. Bettencourt, S. Domino, T. F. an M. Hoemmen, J. J. Hu, E. Phipps, A. Prokopenko, S. Rajamanickam, C. Siefert, and S. Kennon, *Towards extreme-scale simulations for low Mach fluids with second-generation Trilinos*, Parallel Process. Lett., 24 (2014).

[10] P. Lin, J. Shadid, J. Hu, R. Pawlowski, and E. Cyr, *Performance of fully-coupled algebraic multigrid preconditioners for large-scale VMS resistive MHD*, Journal of Computational and Applied Mathematics, (2018).

MPI	Continuity						Momentum					
ranks	SGS		MTSGS		SGS2		SGS		MTSGS		SGS2	
1	37.8	39.1	37.8	39.0	37.8	38.9	4.9	4.9	7.1	7.1	5.4	5.4
2	39.0	41.0	39.0	40.7	38.9	40.7	5.0	5.0	7.2	7.2	5.4	5.4
4	39.8	39.4	39.8	39.3	39.8	39.5	5.3	5.3	7.0	7.0	5.6	5.6
6	41.9	45.8	41.8	45.6	41.8	45.7	6.1	6.1	7.3	7.3	6.2	6.2
8	44.6	38.7	44.6	39.0	44.6	39.1	5.3	5.3	7.1	7.1	5.7	5.7
10	41.5	119.2	41.5	118.7	41.5	119.0	5.6	5.6	7.1	7.1	5.9	5.9
12	42.2	121.6	42.2	121.6	42.2	121.5	6.0	6.0	7.1	7.1	6.1	6.1
14	42.6	33.3	42.6	32.5	42.6	32.3	5.7	5.7	7.2	7.2	6.0	6.0
16	43.0	36.1	43.0	36.0	43.0	36.1	6.2	6.2	7.3	7.3	6.3	6.3
18	40.8	43.8	40.8	43.8	40.8	43.8	5.8	5.8	7.2	7.2	6.0	6.0
20	41.9	57.8	41.9	40.1	41.9	44.9	5.9	7.5	7.2	7.3	6.2	6.3
MPI	Turbulent Kinetic Energy						Specific Dissipation Rate					
ranks	SGS		MTSGS		SGS2		SGS		MTSGS		SGS2	
1	3.3	3.3	5.7	5.7	4.4	4.4	3.6	3.6	7.4	7.4	5.3	5.3
2	3.4	3.4	5.7	5.7	4.4	4.4	3.6	3.6	7.4	7.4	5.3	5.3
4	3.4	3.4	5.7	5.7	4.4	4.4	3.6	3.6	7.4	7.4	5.3	5.3
6	3.4	3.4	5.7	5.7	4.4	4.4	3.6	3.6	7.4	7.4	5.3	5.3
8	3.5	3.5	5.7	5.7	4.4	4.4	3.6	3.6	7.4	7.4	5.3	5.3
10	3.4	3.4	5.7	5.7	4.4	4.4	3.6	3.6	7.4	7.4	5.3	5.3
12	3.4	3.4	5.7	5.7	4.4	4.4	3.6	3.6	7.4	7.4	5.3	5.3
14	3.6	3.6	5.7	5.7	4.5	4.5	3.9	3.9	7.4	7.4	5.4	5.4
16	3.5	3.5	5.7	5.7	4.4	4.4	3.6	3.6	7.4	7.4	5.3	5.3
18	4.2	4.2	5.7	5.7	4.5	4.5	4.0	4.0	7.4	7.4	5.3	5.3
20	3.7	3.8	5.7	5.7	4.4	4.4	3.9	3.8	7.4	7.3	5.3	5.3

Table 9: McAlister strong-scaling (CPU) GMRES iteration counts using Trilinos preconditioners, by physics phase. The main columns indicate that symmetric Gauss-Seidel (SGS), multithreaded symmetric Gauss-Seidel (MTSGS), or two-stage symmetric Gauss-Seidel (SGS2) was used for physics other than continuity. Each main column has two subcolumns. The left subcolumn corresponds to a run using PA-AMG for the continuity solve, the right subcolumn SA-AMG.

		continuity					momentum					
GPUs	Δx	init	assm	load	solve	iters	init	assm	load	solve	iters	
4	40 m	0.03	10.1	0.56	7.2	18	0.03	21.3	0.76	4.0	4	
32	20 m	0.04	10.2	1.2	9.2	17	0.04	20.1	2.15	5.2	6	
256	10 m	0.04	10.2	1.4	13.1	23	0.06	19.6	2.3	8.9	8	
2048	5 m	0.09	10.3	2.2	17.9	24	0.08	19.8	3.4	14.1	18	
		Turbulent Kinetic Energy					Enthalpy					
GPUs	Δx	init	assm	load	solve	iters	init	assm	load	solve	iters	Total Time
4	40 m	0.44	4.7	0.10	2.9	4.1	0.47	4.2	0.12	2.9	4.1	83
32	20 m	0.48	4.8	0.31	3.7	5.1	0.70	4.2	0.32	4.2	5.7	105
256	10 m	0.54	4.8	0.64	5.0	5.1	1.94	4.2	0.97	5.9	5.9	131
2048	5 m	0.62	4.7	1.1	8.2	8.2	1.59	3.9	1.1	10.5	10.5	188

Table 10: ABL weak-scaling execution time breakdown for Nalu-Wind with *hypre* solver stack. Init, assembly, load complete (on CPU). Solve and iterations (on GPU), for momentum, continuity, TKE and enthalpy.

	Av. iter	Min iter	Max iter	Total iter	Total time	Av time per iter
40 m	18.00	18	18	720	11.6666	0.0162
20 m	20.85	19	24	834	13.0149	0.0156
10 m	25.00	22	29	1000	17.4860	0.0175
05 m	28.90	28	29	1156	31.6392	0.0274

Table 11: ABL continuity equation with *hypre* stack. MPI rank sequence 4, 32, 256, 2048

	Av. iter	Min iter	Max iter	Total iter	Total time	Av time per iter
40 m	7.34	4	9	881	6.4015	0.0073
20 m	8.33	4	13	1000	6.5317	0.0065
10 m	11.01	4	23	1321	11.4466	0.0087
05 m	19.40	18	35	2328	19.1419	0.0082

Table 12: ABL momentum equation with *hypre* stack. MPI rank sequence 4, 32, 256, 2048

		Continuity					Momentum					
GPUs	Δx	init	assm	load	solve	iters	init	assm	load	solve	iters	
2	40 m	0.05	22.1	0.5	8.3	18	0.05	43.1	0.8	7.2	4	
16	20 m	0.06	21.1	1.8	11.0	18	0.06	40.9	2.2	8.0	6	
128	10 m	0.06	20.9	2.2	13.7	17	0.07	40.0	3.2	10.5	8	
1024	5 m	0.11	21.1	4.3	19.6	23	0.12	40.1	4.4	19.1	17	
		Turbulent Kinetic Energy					Enthalpy					
GPUs	Δx	init	assm	load	solve	iters	init	assm	load	solve	iters	Total Time
2	40 m	0.44	9.5	0.08	5.5	4.1	0.55	8.4	0.17	5.6	4.1	150
16	20 m	0.60	9.7	0.37	6.5	5.5	1.2	8.4	0.28	7.1	5.5	174
128	10 m	0.93	9.6	0.73	9.2	7.1	1.7	8.3	0.80	10.6	8.1	201
1024	5 m	0.84	9.3	2.91	13.2	9.4	2.0	7.3	1.52	18.0	12.4	269

Table 13: ABL weak scaling execution time breakdown for Nalu-Wind with *hypre* solver stack. Init, assembly, load complete times (on CPU). Solve time and iterations (on GPU), for momentum, continuity, TKE and enthalpy.

		Continuity					Momentum					
GPUs	Δx	init	assm	load	solve	iters	init	assm	load	solve	iters	
8	20 m	0.05	43.16	2.10	10.03	17.0	0.11	84.16	2.1	12.88	5.0	
64	10 m	0.05	42.25	3.95	14.50	17.0	0.11	81.33	6.5	13.00	8.0	
512	5 m	0.17	42.55	5.52	24.85	22.0	0.14	81.17	10.7	24.72	17.0	
		Turbulent Kinetic Energy					Enthalpy					
GPUs	Δx	init	assm	load	solve	iters	init	assm	load	solve	iters	Total Time
8	20 m	0.94	19.4	0.31	13.0	5.1	1.62	17.0	0.62	13.7	5.4	320
64	10 m	1.21	19.3	17.9	17.9	7.1	2.21	16.5	1.10	20.3	8.1	360
512	5 m	1.43	19.3	2.3	35.6	12	2.34	16.7	3.84	35.6	9.8	400

Table 14: ABL GPU weak-scaling baseline for Nalu-Wind with *hypre* solver stack. C-AMG with HMIS coarsening

		Continuity					Momentum					
GPUs	Δx	init	assm	load	solve	iters	init	assm	load	solve	iters	
1	40 m	0.42	0.15	0.01	2.15	10	0.65	2.34	0.01	2.38	5	
8	20 m	0.53	0.17	0.61	5.45	11	0.80	2.58	1.19	4.26	7	
64	10 m	0.56	0.17	1.13	11.28	12	0.82	2.55	3.32	5.85	7	
512	5 m	0.66	0.18	4.05	27.37	15	0.88	2.51	4.12	7.02	7	
		Turbulent Kinetic Energy					Enthalpy					
GPUs	Δx	init	assm	load	solve	iters	init	assm	load	solve	iters	Total Time
1	40 m	0.41	0.23	0.01	0.90	5	0.42	0.37	0.02	0.86	5	29.20
8	20 m	0.52	0.28	0.84	2.45	5	0.53	0.43	0.79	1.57	6	57.05
64	10 m	0.56	0.29	1.67	4.49	5	0.56	0.43	1.65	2.88	6	75.53
512	5 m	0.68	0.30	5.06	7.34	5	0.61	0.40	3.47	4.29	7	117.32

Table 15: ABL GPU weak-scaling baseline for Nalu-Wind coupled to Trilinos solver stack.

	continuity					momentum					
CPUs	init	assm	load	solve	iters	init	assm	load	solve	iters	
2	0.29	291.2	2.2	530.8	21	0.29	437.2	6.73	143.8	8	
4	0.16	135.4	4.1	271.4	21	0.16	210.7	9.03	78.3	8	
8	0.08	66.8	7.3	148.5	21	0.09	103.7	4.47	41.5	8	
16	0.05	32.9	5.0	79.5	21	0.05	49.2	5.64	23.8	8	
20	0.04	25.9	4.7	63.3	21	0.04	39.9	6.43	20.5	8	
	Turbulent Kinetic Energy					Specific Dissipation Rate					
CPUs	init	assm	load	solve	iters	init	assm	load	solve	iters	Total Time
2	0.29	253.5	17.0	40.3	4.1	0.29	253.4	12.3	39.8	4.1	2699
4	0.16	124.7	10.6	21.3	4.2	0.16	126.2	12.8	20.8	4.1	1396
8	0.09	59.1	7.9	11.7	4.5	0.08	61.2	8.3	11.1	4.2	745
16	0.06	29.3	4.2	6.7	4.7	0.06	30.3	4.4	6.6	4.8	390
20	0.04	23.1	3.6	5.2	4.7	0.04	24.2	3.7	4.9	4.3	316

Table 16: McAlister strong-scaling for Nalu-Wind with *hypre* solver stack. Execution time breakdown (CPU).

[11] Oak Ridge National Laboratory, *ORNL Summit web site*. https://www.olcf.ornl.gov/summit.

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

[13] A. Sharma, S. Ananthan, J. Staraman, S. Thomas, and M. Sprague, *Overset meshes for incompressible flows: On preserving accuracy of underlying discretizations*, Journal of Computational Physics, (2020). Under review.

[14] M. Sprague, S. Ananthan, G. Vijayakumar, and M. Robinson, *Exawind: A multifidelity modeling and simulation environment for wind energy*, Journal of Physics: Conference Series, 1452 (2020). 012071, https://iopscience.iop.org/article/10.1088/1742-6596/1452/1/012071/pdf.

[15] M. Sprague, S. Boldyrev, P. Fischer, R. Grout, W. Gustafson Jr., and R. Moser, *Turbulent flow simulation at the Exascale: Opportunities and challenges workshop*, tech. rep., U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, 2017. Published as Tech. Rep. NREL/TP-2C00-67648 by the National Renewable Energy Laboratory.

[16] D. Szyld and T. Jones, *The two stage and multisplitting methods for the parallel solution of linear systems*, SIAM Journal on Matrix Analysis and Applications, 13 (1992), pp. 671—679.

[17] S. Thomas, S. Ananthan, S. Yellapantula, J. J. Hu, M. Lawson, and M. A. Sprague, *A comparison of classical and aggregation-based algebraic multigrid preconditioners for high-fidelity simulation of wind-turbine incompressible flows*, SIAM J. Sci. Statist. Comput., (2019).

[18] U. M. Yang, *Parallel algebraic multigrid methods — high performance preconditioners*, in Numerical Solution of Partial Differential Equations on Parallel Computers, A. B. an A. Tveito, ed., vol. 51 of Lecture Notes in Computational Science and Engineering, Springer, Berlin, Heidelberg, 2006.