Preparing an Incompressible-Flow Fluid Dynamics Code for Exascale-Class Wind Energy Simulations

Preprint

Paul Mullowney,[1] Ruipeng Li,[2] Stephen Thomas,[1] Shreyas Ananthan,[1] Ashesh Sharma,[1] Jon S. Rood,[1] Alan B. Williams,[3] and Michael A. Sprague[1]

1 National Renewable Energy Laboratory
2 Lawrence Livermore National Laboratory
3 Sandia National Laboratories

Presented at the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC21)
Nov 14–19, 2021

Contract No. DE-AC36-08GO28308

Conference Paper
NREL/CP-2C00-79645
September 2021

Preparing an Incompressible-Flow Fluid Dynamics Code for Exascale-Class Wind Energy Simulations

Preprint

Paul Mullowney,[1] Ruipeng Li,[2] Stephen Thomas,[1] Shreyas Ananthan,[1] Ashesh Sharma,[1] Jon S. Rood,[1] Alan B. Williams,[3] and Michael A. Sprague[1]

1 National Renewable Energy Laboratory
2 Lawrence Livermore National Laboratory
3 Sandia National Laboratories

Suggested Citation
Mullowney, Paul, Ruipeng Li, Stephen Thomas, Shreyas Ananthan, Ashesh Sharma, Jon S. Rood, Alan B. Williams, and Michael A. Sprague. 2021. *Preparing an Incompressible-Flow Fluid Dynamics Code for Exascale-Class Wind Energy Simulations: Preprint*. Golden, CO: National Renewable Energy Laboratory. NREL/CP-2C00-79645. https://www.nrel.gov/docs/fy21osti/79645.pdf.

Conference Paper
NREL/CP-2C00-79645
September 2021

Preparing an Incompressible-Flow Fluid Dynamics Code for Exascale-Class Wind Energy Simulations

Paul Mullowney*
Paul.Mullowney@nrel.gov
National Renewable Energy Lab
Golden, Colorado, USA

Ruipeng Li
li50@llnl.gov
Lawrence Livermore National Lab
Livermore, California, USA

Stephen Thomas
Stephen.Thomas@nrel.gov
National Renewable Energy Lab
Golden, Colorado, USA

Shreyas Ananthan
Shreyas.Ananthan@nrel.gov
National Renewable Energy Lab
Golden, Colorado, USA

Ashesh Sharma
Ashesh.Sharma@nrel.gov
National Renewable Energy Lab
Golden, Colorado, USA

Jon S. Rood
Jon.Rood@nrel.gov
National Renewable Energy Lab
Golden, Colorado, USA

Alan B. Williams
william@sandia.gov
Sandia National Laboratories
Albuquerque, New Mexico, USA

Michael A. Sprague
Michael.A.Sprague@nrel.gov
National Renewable Energy Lab
Golden, Colorado, USA

ABSTRACT

The U.S. Department of Energy has identified exascale-class wind farm simulation as critical to wind energy scientific discovery. A primary objective of the ExaWind project is to build high-performance, predictive computational fluid dynamics (CFD) tools that satisfy these modeling needs. GPU accelerators will serve as the computational thoroughbreds of next-generation, exascale-class supercomputers. Here, we report on our efforts in preparing the ExaWind unstructured mesh solver, Nalu-Wind, for exascale-class machines. For computing at this scale, a simple port of the incompressible-flow algorithms to GPUs is insufficient. To achieve high performance, one needs novel algorithms that are application aware, memory efficient, and optimized for the latest-generation GPU devices. The result of our efforts are unstructured-mesh simulations of wind turbines that can effectively leverage thousands of GPUs. In particular, we demonstrate a first-of-its-kind, incompressible-flow simulation using Algebraic Multigrid solvers that strong scales to more than 4000 GPUs on the Summit supercomputer.

KEYWORDS

CFD, Overset, Computational Linear Algebra, GPU Computing, Algebraic Multigrid

ACM Reference Format:
Paul Mullowney, Ruipeng Li, Stephen Thomas, Shreyas Ananthan, Ashesh Sharma, Jon S. Rood, Alan B. Williams, and Michael A. Sprague. 2021. Preparing an Incompressible-Flow Fluid Dynamics Code for Exascale-Class Wind Energy Simulations. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3458817.3476185

1 INTRODUCTION

High-fidelity modeling of wind farms requires high-performance computational fluid dynamics (CFD) tools. Predictive simulations (i.e., those that capture dynamics spanning from micron-scale blade boundary layers to the tens of kilometers of wind farms) are a computational grand challenge requiring exascale-class high-performance computing (HPC) [1, 2]. Predictive simulations must not only be able to capture the geometry of the turbine blades and thin boundary layers; they must do so in the context of turbine rotation and deflection in a turbulent atmospheric environment. Ultimately such tools must handle even more challenging cases such as offshore operation, which includes multiphase fluid dynamics and large platform motions [3]. The development of reliable, high-fidelity, high-performing software will be an invaluable tool for scientific discovery for wind farm design, operation, and optimization.

As part of the U.S. Department of Energy Exascale Computing Project (ECP) [2, 4], the ExaWind project aims to simulate the whole wind farm on next-generation exascale-class computers, which in the United States will be accelerated by graphics processing units (GPUs). The primary physics codes in the open-source ExaWind simulation environment [5] are Nalu-Wind, AMR-Wind, and OpenFAST. Nalu-Wind and AMR-Wind are finite-volume-based CFD codes for the incompressible-flow Navier-Stokes equations, whereas OpenFAST is a wind turbine structural dynamics solver. Nalu-Wind is an unstructured-grid solver that resolves the complex geometry of wind turbine blades and the thin blade boundary layers. AMR-Wind is a block-structured-grid solver with adaptive mesh refinement (AMR) capabilities that captures the background turbulent atmospheric flow and turbine wakes. Nalu-Wind and AMR-Wind models are coupled through overset meshes handled by the Topology Independent Overset Grid Assembler (TIOGA), which is an open-source automated overset mesh assembly library

*All authors contributed equally to this research.

[6, 7]. The CFD models consist of the mass-continuity, Poisson-type equation for pressure and Helmholtz-type equations for transport of momentum and other scalars (e.g., those for turbulence models). For Nalu-Wind, simulation times are dominated by linear-system setup and solution for the continuity and momentum equations. AMR-Wind is built on the AMReX software stack [8] and its geometric multigrid is the primary solver, although it has the option of using the *hypre* library [9] as a solver at the coarsest AMR level. When *hypre* is not used as the bottom solver, linear-system setup and solve are quite fast in comparison to unstructured mesh solvers. When *hypre* is used, AMR-Wind computational throughput and scaling is dictated by the linear system solver performance, just as with Nalu-Wind.

Blade-resolved simulations of wind turbines lead to unstructured grids with challenging features. In particular, one often finds mesh cells with high aspect ratio or mesh cells that are vastly different in size. This leads to poorly conditioned linear systems, especially for the pressure-Poisson equation, which can only be solved efficiently with sophisticated algorithms such as Algebraic Multigrid (AMG) [10]. In contrast, the momentum and turbulent scalar transport equations can be rapidly solved with nonsymmetric Krylov methods such as GMRES [11]. However, even these algorithms still require redesigned preconditioners to find the solutions on highly parallel architectures.

Efficient scaling of unstructured, blade-resolved simulations on large GPU-based machines is a challenging endeavor. In what follows, we will describe our efforts to build a GPU-accelerated CFD package capable of performing blade resolved-simulations for petascale-class simulations. We will show, in detail, the strong scaling characteristics for low- and high-resolution blade-resolved models, and we include a breakdown of the most time consuming model components. Lastly, we provide an important cross machine comparison between Summit at Oak Ridge National Laboratory (ORNL) [12] and Eagle at the National Renewable Energy Laboratory (NREL). These results show the critical influence played by processor architecture and Message Passing Interface (MPI) implementation on the strong scaling performance of an unstructured mesh solver.

Specifically, we report on our efforts to accelerate the Nalu-Wind application when running it primarily on GPUs and using the *hypre* solvers. We describe our techniques for addressing each of the critical algorithmic components, including linear-system assembly, AMG setup, and the design of effective GPU preconditioners/smoothers. Some aspects of these critical components are implemented and executed in the Nalu-Wind software stack including the mesh motion and physics algorithms as well as the graph computation and local linear system assembly. Other algorithms, such as the global-linear-system assembly, AMG setup, and solve are implemented in *hypre*.

This study of the Exawind software stack performance differs from other studies of scalable CFD packages, which can be used to simulate wind turbines, in several, key aspects. The Fun3D CFD simulator can handle both compressible and incompressible flows, but they have only reported on highly scaleable GPU solvers in the compressible regime where the global pressure-Poisson solve is not required [13]. In contrast to Exawind, Fun3D uses high order elements in order to increase the flops per memory access ratio and thereby achieve excellent strong and weak scaling. The OpenFOAM package uses the Nvidia AMGx library for its GPU accelerated linear solves [14]. AMGx has previously demonstrated excellent weak scaling on up to 512 Kepler class GPUs [15], although those simulations consisted of structured grid models whose matrices arise from a seven-point stencil. While their scaling studies addressed elliptic solves, the matrices were well conditioned and thus straightforward to handle by AMG methods. This contrasts with our study, which focuses on poorly conditioned matrices that lack structure. More specifically, we focus on the strong scaling behavior down to 10^5 DoFs per GPU whereas [15] addressed problems whose minimum size was an order of magnitude larger. Ellipsys3D [16–18] was recently validated against Nalu-Wind for turbine blade modelling [19]. This code uses a block structured formulation in curvilinear coordinates to solve the underlying physics equations and multigrid methods for solving the global pressure-Poisson equation. Although this codes achieves near perfect strong scaling with MPI, this software is not, to our knowledge, GPU-accelerated.

This paper is organized as follows. In §2, we give a brief overview of our meshing and domain decomposition strategy and our approach to handling mesh motion. This is followed by descriptions of linear-system assembly algorithms in §3. In §4, the *hypre* AMG setup (§4.1) and smoother algorithms (§4.2) are covered. We give detailed performance results in §5. It should be emphasized that our efforts could not be characterized as a simple GPU port. New algorithm development, designed and optimized for highly parallel architectures, as well as run-time parameter tuning, were necessary steps to achieve the performance reported in this paper.

2 EXAWIND MESHING STRATEGY

In the present work, we use the Nalu-Wind solver to model the flow past a wind turbine using overset-mesh methodology. The computational model is composed of multiple independent meshes for different flow regimes; for example, the rotor is modeled using a body-fitted mesh that is embedded in a wake-capturing unstructured mesh. The Nalu-Wind meshes are, in general, moving with the turbine through rotor rotation. Meshes are coupled through the overset method, for which connectivity must be continually updated as the meshes move. As an example, Figure 1 shows representative unstructured-grid overset meshes for a turbine blade. As described in [20] for Nalu-Wind-only models, the linear systems (associated with the discrete governing equations) are created independently and the solution of the global coupled linear system is accomplished with an additive Schwarz algorithm; in other words, the solution to the equivalent global linear systems is approximated through outer iterations that couple (potentially many) smaller linear systems. While there is an acceptable accuracy reduction, the approach provides enormous benefits. For example, we avoid the need to reduce challenges in weak scaling to the huge problem sizes envisioned (especially for the global pressure-Poisson equation), we simplify the mesh-creation process, we remove the need to re-initialize matrices for each mesh as they move, and we enable our AMR-Wind/Nalu-Wind multi-solver approach. We refer the reader to [20] for details regarding the algorithm, verification, and validation of the approach in the context of Nalu-Wind-only meshes.

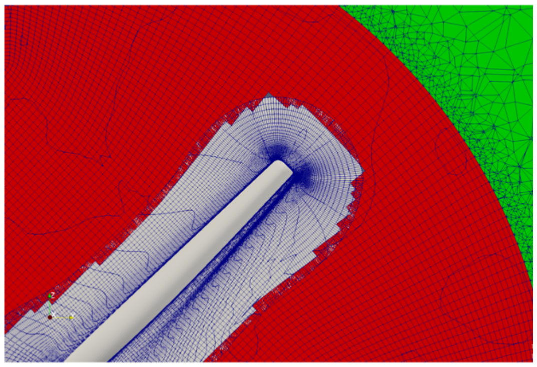

Figure 1: Example of the overset-meshing approach used in ExaWind. Shown are three overlapping Nalu-Wind meshes used to simulate the NREL Phase VI turbine [20].

In this paper, we focus on the performance of Nalu-Wind, for which the underlying mesh data structures are implemented with the Sierra Toolkit (STK) libraries [21] that are distributed with the Trilinos Project [22]. The STK mesh module is used to store the computational mesh and fields, and it provides data access and traversal on GPU platforms. Trilinos provides access to ParMETIS partitioning [23], for mesh rebalancing, through the Zoltan2 software package [24]. The Kokkos library [25] is used heavily, both within the STK mesh library and directly in Nalu-Wind, to provide execution constructs (looping mechanisms as well as data structures) that allow performance portability across traditional CPUs as well as GPU platforms. These data structures and programming constructs allow one to easily write device-portable algorithms for evaluating the key physics algorithms on the mesh. These quantities serve as the input to the local linear-system assembly algorithms which also run on device via Kokkos.

3 NALU-WIND/*hypre* LINEAR SYSTEM ASSEMBLY

Once the terms appearing in the governing equations for incompressible-flow surrounding the wind turbine are evaluated on the mesh, sparse linear systems are built for the solvers embedded in a nonlinear Picard iteration. The construction of the linear system for each governing equation has three stages:

Stage 1: Nalu-Wind graph computation
Stage 2: Nalu-Wind local assembly
Stage 3: *hypre* global assembly

Alternative local assembly strategies based on matrix-matrix multiplication patterns have shown good performance, on the order of 5× faster, when compared with CPU-only algorithms [26]. These approaches are attractive in that they could potentially combine the first two stages into a single algorithm. For our application though, this would impose a significant redesign of the Nalu-Wind application and thus we cannot directly compare these two approaches. Moreover, our computational results show that other algorithmic components including AMG setup and solve, are still, despite being GPU-accelerated, a more dominant computational cost than assembly. Thus, our attention remains focused on making improvements in those areas.

3.1 Nalu-Wind Graph Computation

The graph-computation stage computes the exact sparsity pattern of a linear system for each governing equation. All of the different components of the mesh elements including, edges, faces, and nodes are traversed sequentially. Contributions to each degree of freedom (DoF) are stored. Boundary-condition nodes, including periodic, Dirichlet, and overset DoFs are accounted for precisely. Coordinate (COO) matrices, which includes the row and column indices, are computed for both the owned and shared DoFs. These matrices are sorted in row-major format. Several auxiliary data structures are also constructed that enable matrix element location determination in the next stage. Most of this computation runs on the CPU although at the very end, key data structures are either moved or computed on the GPU device for use in the next stage. The graph computation is sequential, but it could be made more amenable to parallel computation by using parallel scan algorithms to compute the amount of memory needed as well as the indexing data structures needed for computing the graph in parallel threads. This would require multiple traversals through the mesh.

3.2 Nalu-Wind Local Assembly

Once the components of the governing equations are evaluated on the STK mesh, the Nalu-Wind assembly phase can use the graph to fill the matrix and RHS elements in a data-parallel manner. Elements of a particular type, such as an edge or face, are grouped together and iterated over in a parallel fashion. In this formulation, neighboring elements can contribute to the same matrix element value. Because of the massively parallel nature of the underlying implementation, it is possible that the update of these values occurs simultaneously from different threads. To overcome this, we use device atomic operations. While this precludes guaranteed bitwise reproducibility from run to run, the accuracy of the governing-equation discretization is unaffected. Moreover, a reproducible algorithm was implemented although it required significantly more memory and a global sorting algorithm–both of which yield significant performance degradation. For the Nalu-Wind application, both speed of execution and device-memory consumption are primary optimization targets that have been deemed more important than reproducibility. One could perform compensated summation [27] to minimize the effect of the potential discrepancies, but this has not yet been implemented. The output of this stage are the matrix values for the COO matrices, both owned and shared, as well as the right-hand vector entries, both owned and shared. The underlying kernels have been written using the Kokkos API.

Significant efforts were made to optimize this step. This includes the use of (1) both linear and binary search algorithms to determine the matrix write locations as well as (2) read-only, texture memory to query the auxiliary data structures. These auxiliary data structures help determine the write location quickly.

3.3 *hypre* Global Assembly

On a distributed-memory computer, *hypre* distributes the matrix and the vectors in a 1D block-row fashion among the MPI processes.

Thus, in the Nalu-Wind application code, we assemble two COO matrices on each rank as described above. The first matrix contains rows for DoFs owned by the calling MPI rank. The second matrix contains contributions to rows for DoFs owned by other MPI ranks. Viewed from the perspective of the receiver rather than the donor, the received contributions from other MPI ranks either are added to existing row elements or are inserted as new entries. Each COO matrix is sorted in row-major format and contains no duplicates. Given this structure of the assembled matrices, a straightforward algorithm can be written that combines these per rank matrices into a globally consistent linear system.

The procedure shown in Algorithm 1 can be efficiently implemented on massively parallel architectures such as GPUs. Our initial version has been implemented in *hypre* using the CUDA Toolkit and Thrust library. Other GPU architectures can be supported provided implementations exist for the *stable_sort_by_key* and *reduce_by_key* algorithms. Moreover, Algorithm 1 can be implemented to minimize memory usage, which can be critical to the performance, provided the following assumptions are met on $A_{\text{own}} = \{i_{\text{own}}, j_{\text{own}}, a_{\text{own}}, nnz_{\text{own}}\}$ and $A_{\text{send}} = \{i_{\text{send}}, j_{\text{send}}, a_{\text{send}}, nnz_{\text{send}}\}$:

- $nnz_{\text{local}} = nnz_{\text{own}} + \max(nnz_{\text{send}}, nnz_{\text{recv}})$,
- i_{own} and i_{send} are stacked in a contiguous buffer of size nnz_{local},
- j_{own} and j_{send} are stacked in a contiguous buffer of size nnz_{local}, and
- a_{own} and a_{send} are stacked in a contiguous buffer of size nnz_{local}.

Here, nnz_{recv} is the total number of COO entries received from all other ranks on a given MPI rank. This value is easily computed using *MPI_Allreduce* API calls after the graph-computation step completes (§3.1).

Algorithm 1 Global Matrix Assembly algorithm

```
   Input $A_{\text{own}}, A_{\text{send}}, nnz_{\text{own}}, nnz_{\text{send}}, nnz_{\text{recv}}$
   Output $A_{\text{diag}}, nnz_{\text{diag}}, A_{\text{offd}}, nnz_{\text{offd}}$
1: procedure GLOBAL MATRIX ASSEMBLE
2:     MPI_Send $A_{\text{send}}$ to appropriate ranks
3:     MPI_Recv $A_{\text{recv}}$ from appropriate ranks
4:     Stack all received buffers : $A_{\text{all}} = [A_{\text{own}}, A_{\text{recv}}]$
5:     stable_sort_by_key: $A_{\text{all}} = \{i_{\text{all}}, j_{\text{all}}, a_{\text{all}}\}$
6:     reduce_by_key: $\{i_{\text{nnz}}, j_{\text{nnz}}, a_{\text{nnz}}\} \leftarrow \{i_{\text{all}}, j_{\text{all}}, a_{\text{all}}\}$
7:     Split $\{i_{\text{nnz}}, j_{\text{nnz}}, a_{\text{nnz}}\}$ into $\{i_{\text{diag}}, j_{\text{diag}}, a_{\text{diag}}\}$ and $\{i_{\text{offd}}, j_{\text{offd}}, a_{\text{offd}}\}$
8: end procedure
```

The set $A_{\text{recv}} = \{i_{\text{recv}}, j_{\text{recv}}, a_{\text{recv}}, nnz_{\text{recv}}\}$ will likely include duplicate i, j entries and will not be sorted because nearby mesh elements that are owned by different MPI ranks can contribute to the same matrix elements. Once all receive elements are collected, they are copied onto the contiguous buffers described above immediately after the owned portion. This is where the pre-computation of the nnz_{recv} is essential as it allows one to allocate enough space at the outset, thus avoiding expensive device allocation and memcpy calls in the middle of the algorithm. The *reduce_by_key* algorithm finds all nearby elements in the data structure that have the same i,j pair values and sums them together. The last line splits the matrix into diagonal and off-diagonal blocks, an efficient decomposition for performing a Sparse Matrix Vector Multiplies (SpMV), in parallel. SpMVs are the primary workhorse of Krylov and AMG algorithms.

A comparable procedure exists for multi-GPU vector assembly based on similar assumptions. Given the data structures $RHS_{\text{own}} = \{i_{\text{own}}, r_{\text{own}}\}$ and $RHS_{\text{send}} = \{i_{\text{send}}, r_{\text{send}}\}$ assembled on an MPI rank, we assume:

- $n_{\text{local}} = n_{\text{own}} + \max(n_{\text{send}}, n_{\text{recv}})$,
- i_{own} and i_{send} are stacked in a contiguous buffer of size n_{local}
- r_{own} and r_{send} are stacked in a contiguous buffer of size n_{local}
- n_{own} is exactly equal to the number of DoFs owned by a particular rank.

n_{recv} is the total number of RHS entries received from all other ranks on a given rank and can be pre-computed after the graph compute step. RHS_{recv} will likely include duplicates and will not be sorted. The global vector assembly is given in Algorithm 2,

Algorithm 2 Global Vector Assembly algorithm

```
   Input $RHS_{\text{own}}, RHS_{\text{send}}, n_{\text{own}}, n_{\text{send}}, n_{\text{recv}}$
   Output $RHS$
1: procedure GLOBAL VECTOR ASSEMBLE
2:     MPI_Send $RHS_{\text{send}}$ to appropriate ranks
3:     MPI_Recv $RHS_{\text{recv}}$ from appropriate ranks
4:     stable_sort_by_key: $RHS_{\text{recv}} = \{i_{\text{recv}}, r_{\text{recv}}\}$
5:     reduce_by_key: $\{i_{\text{new}}, RHS_{\text{new}}\} \leftarrow \{i_{\text{recv}}, RHS_{\text{recv}}\}$
6:     $RHS \leftarrow RHS_{\text{own}}$
7:     $RHS[i_{\text{new}}]$ += $RHS_{\text{new}}[i_{\text{new}}]$
8: end procedure
```

A key distinction between the vector (Algorithm 2) and matrix (Algorithm 1) assemblies is that in the vector assembly, the sort and reduce steps are only performed over the values received from other ranks. This is because RHS_{own} is sorted and the length of these data structures is exactly equal to the number of rows owned by this rank. After the copy (Step 6 in Algorithm 2), the reduced received values are added to the correct elements in a parallel manner. One could simply apply the stable sort and reduce by key over the entire stacked data structure; however, sorts are typically expensive algorithms. Because $n_{\text{recv}} \ll n_{\text{own}}$, applying the sort and reduce steps over a much smaller data structure has shown nontrivial performance advantages.

We have experimented with a similar approach in the global matrix assembly. Since $nnz_{\text{recv}} \ll nnz_{\text{own}}$, one might expect significant performance benefits. In contrast to the vector case where the final steps include a copy and a scatter-add, the matrix algorithm transforms into the addition of two sparse matrices that can be completed using the cuSPARSE library for NVIDIA architectures. Numerical experiments have shown little performance benefit over Algorithm 1. This is because in all likelihood, sparse matrix addition is implemented using sorting algorithms. One benefit of this approach is the memory usage–profiling has indicated a smaller memory footprint than the full sorting approach.

The description above refers to algorithms developed in a branch of the *hypre* source code. From the application perspective, the assembled COO matrices are injected into *hypre* API methods to build the global matrix.

HYPRE_IJMatrixSetValues2,
HYPRE_IJVectorSetValues2,
HYPRE_IJMatrixAddToValues2, and
HYPRE_IJVectorAddToValues2.

First we apply the "SetValues2" routines to set the matrix/RHS of the owned rows on the calling MPI rank. To add the off-rank matrix/RHS elements, we then call "AddToValues2" routines using the shared matrix/RHS elements as input. Then, we can all the assembly routines

HYPRE_IJMatrixAssemble, and
HYPRE_IJVectorAssemble,

which encapsulate Algorithms 1 and 2 entirely. The advantage of this implementation is that it completes the assembly in six *hypre* API calls. It then leverages the internal-messaging structure of *hypre* to properly build the full matrix and RHS. Details on *hypre*'s unstructured interface and IJ object assembly on GPUs can be found in [28].

4 *hypre* AMG SOLVER

In this section, we provide an overview of the AMG method exploited in this work using *hypre*'s BoomerAMG on GPUs. As discussed in [29], this is a particularly powerful method for solving these challenging systems.

4.1 AMG Setup

AMG methods [10, 30, 31] are widely used and efficient scalable solvers/preconditioners for large-scale linear systems arising from partial differential equations (PDEs) due to their optimal complexity, numerical scalability and good parallelism [32]. In the setup phase of AMG methods, a multilevel hierarchy that consists of linear systems with exponentially decreasing sizes on coarser levels is built. A strength-of-connection (SoC) matrix S, is typically first computed to indicate directions of algebraic smoothness used in coarsening algorithms. The construction of S can be performed efficiently on GPUs, because each row of S can be computed independently by selecting entries in the corresponding row of A with a prescribed threshold value θ. BoomerAMG currently only provides the parallel maximal independent set (PMIS) coarsening [33] on GPUs, which is a modified from Luby's algorithm [34] for finding maximal independent sets using random numbers. The process of selecting coarse points in this algorithm is massively parallel, which makes it appropriate for GPUs. The random numbers used in PMIS are generated by the cuRAND library.

Interpolation operators in AMG transfer residual errors between adjacent levels. Many interpolation schemes are available in BoomerAMG on CPUs. The so-called direct interpolation [31] is straightforward to port to GPUs because the interpolatory set of a fine point i is just a subset of the neighbors of i, so that the interpolation weights can be determined solely by the i-th equation. A bootstrap AMG (BAMG) [35] variant of direct interpolation is generally found to be better than the original formula. The weights w_{ij} are computed by solving local optimization problem

$$\min \left\| a_{ii} w_i^{\mathsf{T}} + a_{i,C_i^s} \right\|_2 \quad \text{s.t. } w_i^{\mathsf{T}} f_{C_i^s} = f_i, \tag{1}$$

where w_i is the vector that contains w_{ij}, C_i^s denotes strong C-neighbors of i and f is a target vector that needs to be interpolated exactly. For elliptic problems where the near null space is spanned by constant vectors (i.e., $f = \vec{1}$), closed-form solution of (1) is given by

$$w_{ij} = -\frac{a_{ij} + \beta_i / n_{C_i^s}}{a_{ii} + \sum_{k \in N_i^w} a_{ik}}, \quad \beta_i = \sum_{k \in \{F_i \cup C_i^w\}} a_{ik}\,, \tag{2}$$

where $n_{C_i^s}$ denotes the number of points in C_i^s, C_i^w the weak C-neighbors of i, F_i the F-neighbors, and N_i^w the weak neighbors. A known issue of PMIS coarsening is that it can result in F-points without C-neighbors [36]. In these situations, distance-one interpolation algorithms often work well, but interpolation operators that can reach C-points at a larger range, such as the extended interpolation [36], can generally yield much better convergence. However, implementing extended interpolations is much more complicated mainly because the sparsity pattern of the interpolation operator cannot be determined a priori, which would require dynamically combining C-points in a distance-2 neighborhood, and furthermore, efficient implementation can be even more difficult on GPUs. With minor modifications to the original form, it turns out that the extended interpolation operator can be rewritten in standard sparse matrix computations such as matrix-matrix (M-M) multiplications and diagonal scalings with certain FF- and FC-submatrices. For instance, the so-called "MM-ext" interpolation takes the form

$$W = -\left[(D_{FF} + D_\gamma)^{-1} (A_{FF}^s + D_\beta) \right] \left[D_\beta^{-1} A_{FC}^s \right]$$

with $D_\beta = \text{diag}(A_{FC}^s \vec{1}_C)$ and $D_\gamma = \text{diag}(A_{FF}^w \vec{1}_F + A_{FC}^w \vec{1}_C)$, where A is assumed to be decomposed into $A = D + A^s + A^w$, the diagonal, the strong part and the weak part respectively, and A_{FF}^w, A_{FC}^w, A_{FF}^s and A_{FC}^s are the corresponding submatrices of A^w and A^s. This formulation allows simple and efficient implementations that can utilize available optimized sparse kernels on GPUs. Similar approaches that are referred to as "MM-ext+i" are modified from the original extended+i algorithm [36]. "MM-ext+e" are also available in BoomerAMG. See [37] for details on the class of M-M based interpolation operators.

Aggressive coarsening can reduce the grid and operator complexities of the AMG hierarchy, where a second coarsening is applied to the C-points obtained from the first coarsening to produce a smaller set of final C-points. In this work, we use the A-1 aggressive coarsening strategy described in [31]. The second PMIS coarsening is performed with the CC block of $S^{(A)} = S^2 + S$ that has nonzero entry $S_{ij}^{(A)}$ if i is connected to j with at least a path of length less than or equal to two. Aggressive coarsening is usually used with two-stage interpolation [38] which computes a second-stage interpolation matrix P_2 and combined with the first-stage P_1 as $P = P_1 P_2$. The aforementioned MM-based interpolation is also available for the second stage.

Finally, Galerkin triple-matrix products are used to build coarse-level operators. This computation is performed using parallel primitives from Thrust and routines from cuSPARSE or *hypre*'s own sparse kernels. We refer to [28] for the details we omit here on the algorithms used in *hypre* for computing distributed sparse M-M multiplications on GPUs.

4.2 Two-Stage Gauss Seidel Smoother

The Nalu-Wind time integrator employs the one-reduce GMRES linear solver for the momentum and pressure-Poisson governing equations as described in [39]. The momentum solver is preconditioned with a two-stage Gauss-Seidel relaxation scheme as described below. The pressure-Poisson preconditioner is based on an AMG algorithm using aggressive PMIS coarsening at the first two levels combined with the matrix-based approach for the second-stage interpolation and it applies a two-stage Gauss-Seidel relaxation as the smoother within an AMG V-cycle [40].

To solve a linear system $A\mathbf{x} = \mathbf{b}$, the Gauss-Seidel (GS) iteration is based on the matrix splitting $A = L + D + U$, where L and U are the strictly lower and upper triangular parts of the matrix A, respectively. Then, the traditional GS updates the solution based on the following recurrence,

$$\mathbf{x}_{k+1} := \mathbf{x}_k + M^{-1}\mathbf{r}_k, \quad k = 0, 1, 2, \ldots \tag{3}$$

where $\mathbf{r}_k = \mathbf{b} - A\mathbf{x}_k$, $A = M - N$, and $M = L + D$, $N = -U$ or $M = U + D$, $N = -L$ for the forward or backward sweeps, respectively. In the following, we use $\mathbf{a}_j$ to denote the j−th column of a matrix A, while $a_{i,j}$ or x_i is the (i, j)−th element of A or the i−th element of a vector $\mathbf{x}$, respectively. To avoid explicitly forming the matrix inverse M^{-1} in (3), a sparse-triangular solve is used to apply M^{-1} to the current residual vector $\mathbf{r}_k$.

To improve the solver scalability, *hypre* implements a hybrid variant of Gauss-Seidel [41], where the neighboring processes first exchange the elements of the solution vector on the boundary, but then each process independently applies the local relaxation. Furthermore, in *hypre*, each process may apply multiple local GS sweeps for each round of the neighborhood communication. With this approach, each local relaxation updates only the local part of the vector $\mathbf{x}_{k+1}$ (during the local relaxation, the non-local solution elements on the boundary are not kept consistent among the neighboring processes). This hybrid algorithm is shown to be effective for many problems [41].

$$\widehat{\mathbf{x}}_{k+1} := \widehat{\mathbf{x}}_k + \widehat{M}^{-1}(\mathbf{b} - A\widehat{\mathbf{x}}_k), \quad k = 0, 1, 2, \ldots \tag{4}$$

where $\widehat{M}^{-1}$ represents the approximate triangular system solution, i.e., $\widehat{M}^{-1} \approx M^{-1}$. In our approach, a Jacobi-Richardson (or Jacobi) inner iteration is employed. In particular, if $\mathbf{g}_k^{(j)}$ denotes the approximate solution from the j-th inner iteration at the k-th outer GS iteration, then the initial solution is chosen to be the diagonally scaled residual vector,

$$\mathbf{g}_k^{(0)} = D^{-1}\mathbf{r}_k, \tag{5}$$

and the $(j + 1)$−st JR iteration computes the approximate solution by the recurrence

$$\begin{aligned} \mathbf{g}_k^{(j+1)} &:= \mathbf{g}_k^{(j)} + D^{-1}(\mathbf{r}_k - (L + D)\mathbf{g}_k^{(j)}) && (6) \\ &= D^{-1}(\mathbf{r}_k - L\mathbf{g}_k^{(j)}). && (7) \end{aligned}$$

When zero inner sweeps are performed, the two-stage GS recurrence becomes

$$\widehat{\mathbf{x}}_{k+1} := \widehat{\mathbf{x}}_k + \mathbf{g}_k^{(0)} = \widehat{\mathbf{x}}_k + D^{-1}(\mathbf{b} - A\widehat{\mathbf{x}}_k),$$

and this special case corresponds to Jacobi-Richardson for the global system, or local system on each process. When s inner iterations are performed, it follows that

$$\begin{aligned} \widehat{\mathbf{x}}_{k+1} &:= \widehat{\mathbf{x}}_k + \mathbf{g}_k^{(s)} = \widehat{\mathbf{x}}_k + \sum_{j=0}^{s}(-D^{-1}L)^j D^{-1}\widehat{\mathbf{r}}_k \\ &\approx \widehat{\mathbf{x}}_k + (I + D^{-1}L)^{-1}D^{-1}\widehat{\mathbf{r}}_k = \widehat{\mathbf{x}}_k + M^{-1}\widehat{\mathbf{r}}_k, \end{aligned}$$

where M^{-1} is approximated by the degree-s Neumann expansion. Note that $D^{-1}L$ is strictly lower triangular so that the Neumann series converges in a finite number of steps.

The two-stage GS recurrence (4) can be also written as

$$\begin{aligned} \widehat{\mathbf{x}}_{k+1} &:= \widehat{\mathbf{x}}_k + \widehat{M}^{-1}(\mathbf{b} - (M - N)\widehat{\mathbf{x}}_k) && (8) \\ &= (I - \widehat{M}^{-1}M)\widehat{\mathbf{x}}_k + \widehat{M}^{-1}(\mathbf{b} + N\widehat{\mathbf{x}}_k). && (9) \end{aligned}$$

In the classical one-stage recurrence (3), the preconditioner matrix is taken as $\widehat{M}^{-1} = M^{-1}$, and only the second term remains in the recurrence (9), leading to the following "compact" form,

$$\mathbf{x}_{k+1} := M^{-1}(\mathbf{b} + N\mathbf{x}_k). \tag{10}$$

Hence, the recurrences (3) and (10) are mathematically equivalent, while the recurrence (10) has a lower computation cost.

An effective preconditioner for the momentum equation is given by a compact form of the two-stage symmetric Gauss-Seidel (SGS2) relaxation that consists of the following steps for each outer iteration

$$\begin{aligned} \mathbf{r}_k &:= b - L\mathbf{g}_k^{(j)} && (11) \\ \mathbf{g}_k^{(j+1/2)} &:= D^{-1}\ (\ \mathbf{r}_k - U\ \mathbf{g}_k^{(j)}\ ) && (12) \\ \mathbf{r}_k &:= b - U\mathbf{g}_k && (13) \\ \mathbf{g}_k^{(j+1)} &:= D^{-1}\ (\ \mathbf{r}_k - L\ \mathbf{g}_k^{(j+1/2)}\ ) && (14) \end{aligned}$$

Two outer and two inner inner iterations often leads to rapid convergence in less than five preconditioned GMRES iterations.

5 SIMULATION RESULTS

The performance of our implementation is measured through a series of blade-resolved simulations of NREL 5-MW wind turbines on the Summit supercomputer. The NREL 5-MW turbine [42] is a notional reference turbine with a 126 meter rotor that is appropriate for offshore wind studies. The simulations here use the model described in [5], but with rigid blades, and they include low- and high-resolution models of a single-turbine as well as a low-resolution model of two turbines in sequence (see table 1). An example of a flow field is shown in Figure 2. These models use inflow and outflow boundary conditions in the directions normal to the blade rotation and symmetry boundary conditions in other directions. For each simulation, we perform 50 time steps from a cold start with four Picard iterations per time step. The cold start implies that the simulation will undergo an *initial transient phase* from a non-physical initial solution guess before settling into a quasi-steady solution state. This initial transient phase is more challenging for the linear-system solvers and will require more GMRES iterations per equation system. However, our simulations indicate the overhead is less than 20%. For these studies we use all the available GPU or CPU resources per Summit node; 42 Power9 CPU cores and 6 NVIDIA V100 GPUs for the CPU and GPU studies, respectively.

The data for the strong-scaling plots are computed as follows. For each turbine simulation, a log file is generated that yields two sets

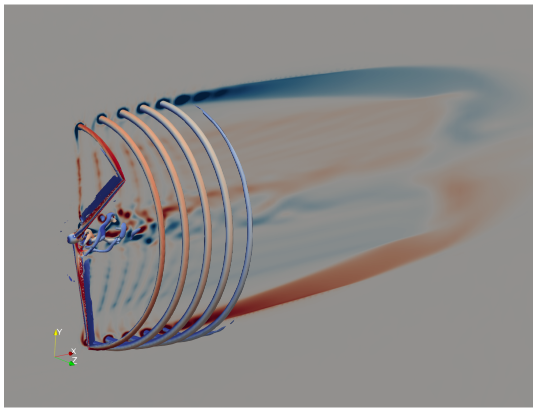

Figure 2: Flowfield (isosurfaces of Q-criterion colored by vorticity magnitude and a plane with vorticity-magnitude isocontours) for the NREL 5-MW rotor with rigid blades operating in uniform inflow of 8 m/s. [5]

Table 1: NREL 5-MW turbine mesh sizes.

NREL5MW Mesh	1 Turbine	2 Turbines	1 Turbine Refined
Mesh Nodes	23,022,027	44,233,109	634,469,604

of data. For each time step, the log file outputs the time spent doing nonlinear iterations (NLI) (i.e., GPU-accelerated physics and math algorithms). This gives 50 distinct data points that have variation. In the strong-scaling plots 3, 8, and 9, each point and its associated error bar is computed by the mean and standard deviation of these 50 data points. At the end of the simulation, the application outputs the total time spent in each equation system doing local assembly and physics algorithms in Nalu-Wind, and global assembly, preconditioner setup, and solve in *hypre*. Each of these data points are scaled by 50 to get the per equation breakdowns shown in Figures 6 and 7.

5.1 Low-Resolution, Single-Turbine Performance

The strong-scaling performance of the low resolution single-turbine mesh is shown in Figure 3. We also show results from an earlier, baseline GPU version of the implementation. The baseline results contain the fast GPU implementations of AMG setup (§4.1) and the two-stage Gauss-Seidel smoother (§4.2). The results also contain a more general GPU implementation of linear system assembly than what is described in §3. Without those implementations, the baseline would be substantially slower than even the CPU results. One could characterize the gain provided by those algorithms as first-order optimizations as they resulted in an order of magnitude performance gain. The improvements from the baseline, shown in Figure 3, are second-order optimizations in that they add 30%-40% improvement in performance, but not an order of magnitude.

Across the board, the current GPU implementation is substantially faster than the baseline. The acceleration is particularly evident for 3-4 Summit nodes where the baseline implementation slows down exactly in the regime where we expect accelerators to

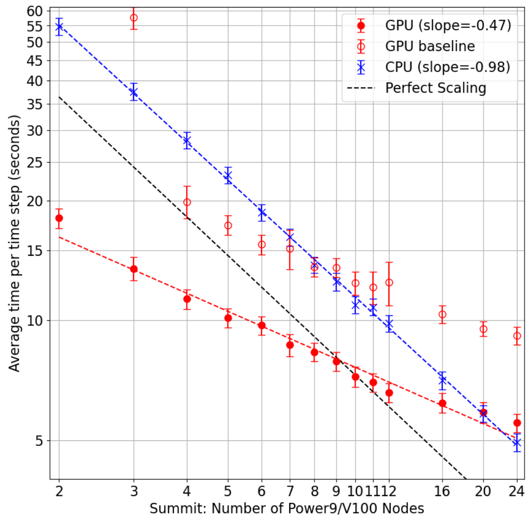

Figure 3: Average nonlinear-iteration time per time step (NLI) for the NREL 5-MW turbine mesh (table 1).

perform well (i.e., many mesh nodes per GPU). The performance can be tied directly to the version of Nalu-Wind and *hypre* assembly algorithms. *hypre* has a more general version of matrix assembly. This generality comes with a price; it requires more device memory, more data motion, and more complex algorithms to implement fully. We refer the reader to [28] for details. Our implementation, described in §3.3, is tuned to our particular use case although the underlying implementation exists in a branch of the *hypre* source code.

The optimized assembly algorithm accounts for a substantial fraction of the performance gain as measured from the baseline results, though not all. A rough empirical estimate of the gain attributed to this optimization is 50%. Small but nontrivial gains were attained through the use of read-only texture memory and linear-search algorithms in the Nalu-Wind local assembly (see §3.2). In addition, the inclusion of a second inner iteration (equations 5-7 with $j == 1$) in the two-stage Gauss-Seidel algorithm has proven effective at reducing the number of GMRES iterations by roughly 2× for the momentum and scalar transport equations. Lastly, parameter tuning of the BoomerAMG preconditioner, which is used in the pressure-Poisson equation, has also yielded modest, but nontrivial gains particularly in the setup phase. These latter optimizations account for 25% of the gain over the baseline, which is estimated empirically.

Our original implementation used an RCB (recursive bisection) algorithm for domain decomposition. We have observed for meshes relevant to wind-turbine simulations that RCB can lead to imbalanced and/or skewed subdomains. This is illustrated in Figure 4. In particular, the small, disconnected red and light blue slivers are of great concern as they will lead to inefficient messaging patterns during program execution.

To mitigate this problem, we introduced ParMETIS-based mesh rebalancing as part of our simulation workflow. To measure the

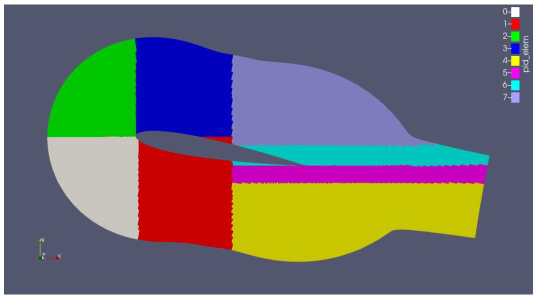

Figure 4: Domain decomposition of an airfoil from a recursive bisection algorithm.

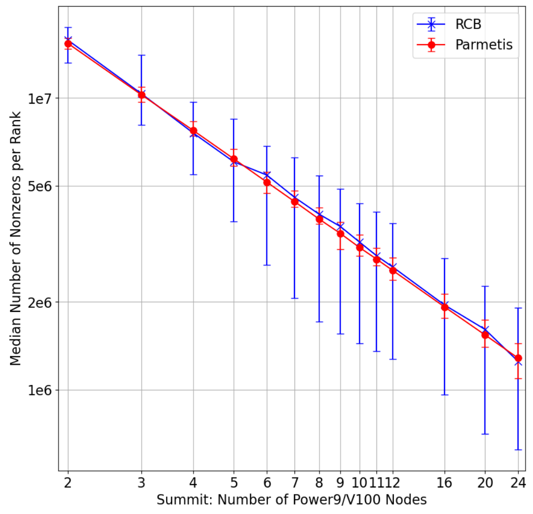

Figure 5: Median NNZ per rank with error bars defined by the minimum and maximum NNZ for RCB and ParMETIS domain decompositions for the lower resolution, single-turbine mesh.

amount of balance in the repartitioned linear systems, we compute the median number of nonzeros per GPU (MPI rank). Figure 5 shows the median nonzeros per GPU for the pressure-Poisson system. The error bars are given by the minimum and maximum number of nonzeros per GPU. A similar plot using the standard deviation of the nonzeros per GPU would show a very tight spread for ParMETIS, but not for RCB. Indeed, the use of ParMETIS reduces the variation in the nonzeros per rank by approximately ten for all node configurations. We surmise that this reduction in spread leads to substantially more efficient communication patterns. Overall, we estimate, empirically, that the use of ParMETIS accounts for the remaining 25% of the performance gain from our baseline implementation. Furthermore, intelligent domain partitioning will benefit any component or algorithm of the program that uses MPI communication including CPU-based algorithms. The blue scaling plot in Figure 3 is also based on a ParMETIS domain decomposition. RCB decompositions shift the blue curve upward by a nontrivial amount.

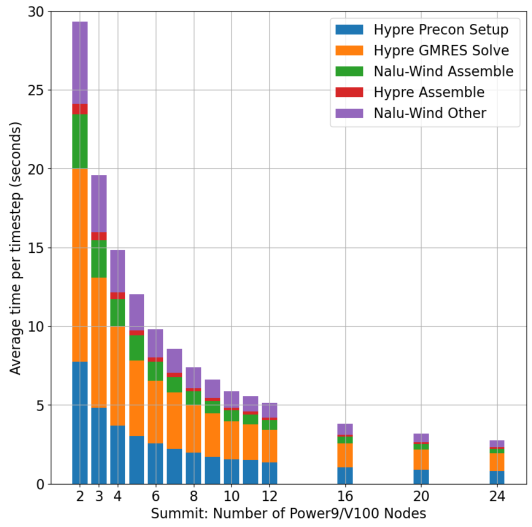

Figure 6: CPU pressure-Poisson equation time breakdown for the low resolution NREL 5-MW turbine mesh (table 1).

The average nonlinear iteration time per time step (NLI) can be further subdivided into the time spent solving each equation system. Figures 6 and 7 show the pressure-Poisson timing breakdowns for the CPU and GPU respectively. We omit the momentum and turbulent scalar transport results as they comprise a much smaller fraction of the NLI than pressure-Poisson. The average time per time step for each equation is given by the peak of these bar charts. They can be summed, when including the momentum and scalar transport terms, to get the mean NLI reported in Figure 3. For each per equation breakdown plot, we use the same y-scale in order to easily compare different equations across different compute resources (i.e., CPU or GPU). The purple sub-bar captures the graph computation (sparsity pattern) and physics algorithms. The green sub-bar captures the local assembly. This particular stage shows a 4× speedup over the CPU implementation which is competitive with the gains observed in [26]. The red, blue, and orange sub-bars capture the global assembly, preconditioner setup, and solve phase, respectively. The time spent in the preconditioner setup and solve phase dominates the pressure-Poisson component on the CPU, but the scaling is quite good. On the GPU, the scaling degrades considerably. As the number of DoFs per GPU decreases, the scalable performance of AMG degrades−this is unsurprising due to the additional communication burden that using accelerators imposes. Though not shown in here, the performance of the preconditioner setup degrades considerably when the cuSPARSE implementation (`cusparseDcsrgemm`, v10.2) of sparse matrix-matrix multiply (SpGEMM) is used. Thus, we use *hypre*'s hash-based SpGEMM implementation, which exhibits superior throughput. There was an anomaly in the GPU performance at eight Summit nodes, the cause of which is unknown.

The cross-over point between CPU and GPU performance, as measured by average time per time step, occurs around 20 Summit nodes (120 NVIDIA V100 GPUs) or roughly 200,000 mesh nodes per GPU. For the pressure-Poisson in isolation, the cross-over point

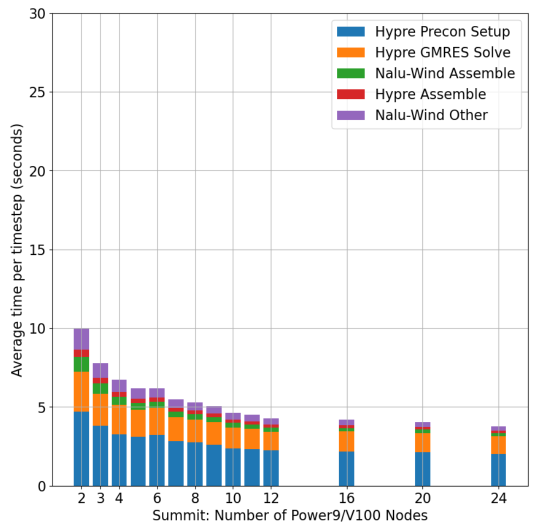

Figure 7: GPU pressure-Poisson equation time breakdown for the low resolution NREL 5-MW turbine mesh (table 1).

occurs around 14 Summit nodes (270,000 mesh nodes/GPU); however, the momentum and turbulent scalar-transport solves show better performance for fewer mesh nodes per device. These latter equations are solved with a GMRES iteration and a simple two-stage Gauss-Seidel preconditioner, and thus they do not have the additional communication burdens of AMG. Although not shown here, a standalone pressure-Poisson solve shows better scaling performance when using fewer than 12 V100s as expected. In this regime, though, the entire application consumes too much device memory and thus slows down considerably. With sufficient device memory, we would expect the GPU-MPI implementation to outperform the MPI CPU-only implementation by even greater margins if the number of mesh nodes per GPU could be increased. It would be interesting to measure the performance on machines with NVIDIA A100 GPUs, as these devices have 80GB of device memory and are 5x the size of the V100 devices on Summit.

5.2 Dual-Turbine and High-Resolution, Single-Turbine Performance

The average nonlinear iteration time per time step for the dual-turbine and refined single-turbine meshes are given in Figures 8 and 9. The dual-turbine mesh shows very similar performance to the lower resolution single-turbine mesh. There may be a bit more variation in the time per time step, as indicated by the larger error bars in the GPU curves in Figure 8 versus Figure 3. For the refined single-turbine mesh, the scaling behavior is consistent with the smaller meshes although there is far greater fluctuation. These simulations employ up to 4,320 GPUs (30,420 CPUs) (i.e., 1/6 of the Summit resources−a nontrivial percentage of the entire machine). To our knowledge, this is one of the first physics applications to run challenging, unstructured-mesh AMG solvers at this scale on GPU architectures.

We use ParMETIS domain decompositions for both the dual-turbine and the highly resolved single-turbine simulation for CPU

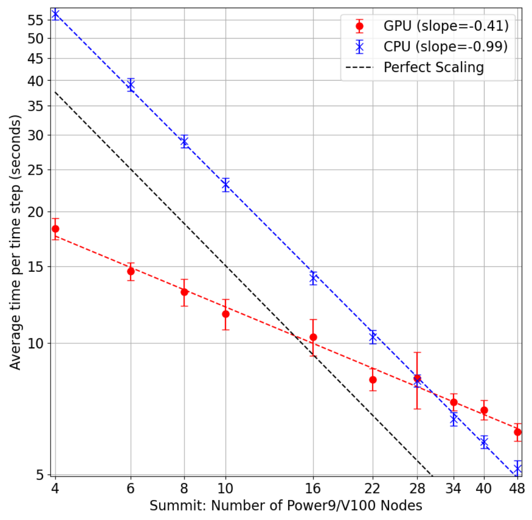

Figure 8: Average nonlinear iteration time per time step for the NREL 5-MW two turbine mesh (table 1).

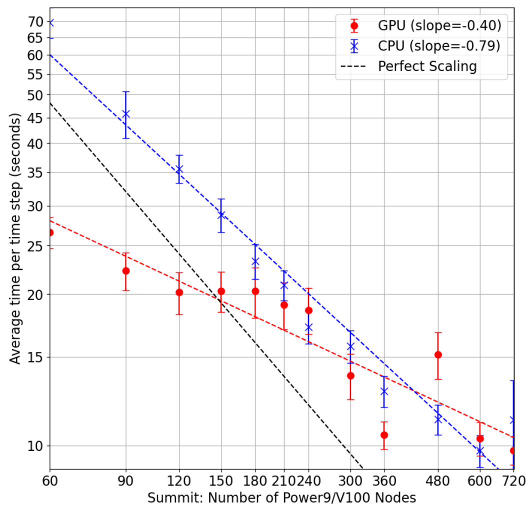

Figure 9: Average nonlinear iteration time per time step for the NREL 5-MW refined turbine mesh (table 1).

and GPU simulations. For the highly resolved case, a plot of the median nnz per rank and the corresponding spread, as measured through the minimum and maximum nnz, is shown in Figure 10. This plot shows a very different pattern from 5. While the use of ParMETIS reduces the maximum, it also reduces the minimum. Thus, the overall spread seems largely unchanged compared to RCB. This may account for some of the variation in the NLI timings for the highly refined single-turbine case. Indeed, there is some empirical evidence to suggest the ParMETIS approach breaks down at large processor counts [43]. It is worth noting that both the CPU and GPU performance for high resolution models show greater variability than low resolution models. Moreover, the CPU shows a significant drop in strong scaling performance compared to the low

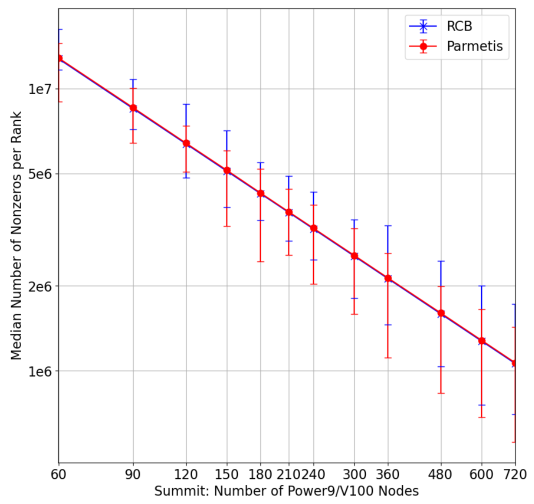

Figure 10: Median NNZ per rank with error bars defined by the minimum and maximum NNZ for RCB and ParMETIS domain decompositions for the refined single-turbine mesh.

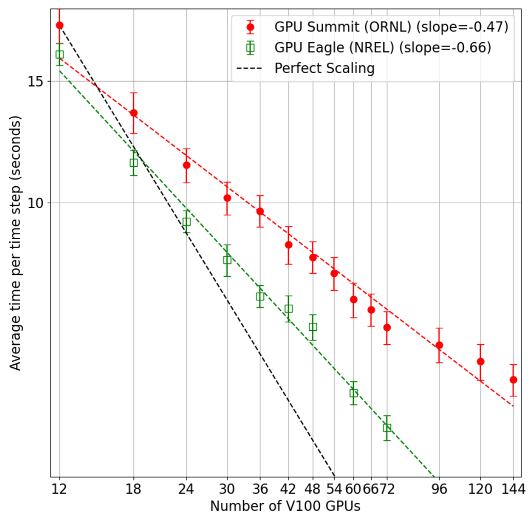

Figure 11: Average nonlinear iteration time per time step for the NREL 5-MW lower-resolution, single-turbine mesh (table 1) for two machines: 1) System 1 is Summit which has 6 V100 GPUs per node with 42 Power 9 cores, and 2) Eagle, which has 2 V100 GPUs per node with 36 x86 cores per node.

resolution model; -.79 slope versus -.98. One could conclude that the performance drop may be due to the specific nature of this particular problem–highly unstructured, imbalanced, with poorly conditioned linear solves. It is likely though that the MPI implementation plays some role and perhaps an important one. In particular, the interplay of GPU communication and MPI is worth considering at a deeper level.

5.3 Hardware, Software, and Architecture Dependence

We also performed strong scaling studies of the lower resolution, single-turbine mesh on the Eagle. This machine has 2 V100 PCIe GPUs per node and 36 x86 CPU cores. Eagle is significantly smaller than Summit and thus we were only able to perform simulations of the small, single-turbine case. The software versions of Nalu-Wind and Hypre are identical. ParMETIS domain decompositions are performed for each simulation. Aside from hardware differences, each system has different MPI implementations and base compiler; Spectrum MPI and GCC 7.4.0 versus HPE MPT and GCC 8.4.0. Moreover, the actual GPUs themselves are somewhat different; V100 SXM2 versus V100 PCIe. The PCIe GPU has a slightly reduced peak double precision performance.

In Figure 11, we compare the strong scaling performance of Summit versus Eagle [44]. The difference in strong scaling performance between these 2 systems is stunning. Indeed, 72 GPUs on Eagle is nearly 40% faster than 144 GPUs on Summit. While the slope of Eagle is still not optimal, it is drastically improved compared with Summit. Moreover, a detailed breakdown shows that the gains are made almost exclusively in the pressure-Poisson AMG setup and solve. Consider the largest simulations for each system of 72 GPUs (Eagle) and 144 GPUs (Summit). The average time per time step spent doing AMG setup is 1.3s versus 2.0s on Summit. The solve phase shows equivalent gains: .8s versus 1.1s. This is a 30%-40% gain with half the GPU resources! This suggests that the hardware-software configuration plays a critical role in the strong scaling performance of an application using implicit solvers on unstructured grids.

However, reaching optimal strong scaling remains a challenge even for an x86-based machine. Considering the nature of row-based decomposition for sparse matrices, it is inevitable that a greater percentage of the matrix entries will move to the off-diagonal block as one increases the number of compute resources (i.e. more GPUs in the strong scaling limit). This incurs a greater messaging overhead in the SpMV kernels and more computation that waits on the completion of that messaging before it can execute. It may be that applications whose linear systems have more entries per row clustered around the main diagonal can effectively hide the additional communication burden that strong scaling imposes. For the Nalu-Wind case, we have on average eight entries per row, which may not be enough to hide messaging costs in the strong scaling limit, even on more optimal architectures such as Eagle.

6 DISCUSSION

In this paper, we have demonstrated the performance of the Nalu-Wind CFD application on petascale-level computations for wind turbine simulations. The simulations were designed to keep the number of mesh nodes per GPU consistent across the three different strong-scaling studies, although it is difficult to be overly precise here. Our objective with these studies was to approximate the weak-scaling performance of our application for real turbine meshes. Given that our largest mesh, with 640 million mesh nodes, ran on 1/6 the total GPU resources on Summit, which has peak double-precision computational throughput of 200 PetaFlops/sec, we estimate that a mesh with approximately four billion nodes would display similar strong scaling characteristics on the entire

Summit machine. Moreover, a mesh with 20-30 billion mesh nodes would require exascale compute resources to handle properly.

Overall, the performance on GPU-accelerated architectures is encouraging, especially given the complexity of this particular physics application. To our knowledge, this is first physics application to run complex solvers such as AMG on unstructured-mesh problems on so many GPU accelerators. However, it is worth emphasizing several critical points about our implementation because other teams with comparable physics applications may face similar challenges.

Though our baseline implementation had good performance, ample evidence suggests that we were not operating in an optimal manner. The most obvious indicator of this was the jump in performance when reducing from four to three Summit nodes. Indeed, implementations before the baseline showed a jump between four and five Summit nodes. In each of these cases, the most common, sub-optimal performance bottleneck was the overuse of device DRAM and excessive data motion. In some cases, large device allocations occurred in between kernel launches. Although this is inevitable to some extent, minimization of these is absolutely critical. With each identification of significant, inefficient memory use, we gained not only the expected performance for fewer Summit nodes (i.e., many mesh nodes per GPU), but we also observed an overall downward shift in the strong scaling curve. Thus the overall application moved closer to peak performance. This is not to say that Nalu-Wind is operating at peak performance, but significant strides have been made.

Many of the memory inefficiencies for our application, existed within Nalu-Wind source code. Others exist or existed within the *hypre* library. We made significant efforts to build a linear-system assembly algorithm with the simplest possible structure in both Nalu-Wind and *hypre*. Achieving this simplicity required detailed knowledge of the *hypre* source code and data-flow patterns. We concede that many other application teams may not have the resources to justify such a deep dive into a complex code such as *hypre*. When high performance is a top-level goal, this level of effort may be needed. Ultimately, our goal is to make our assembly algorithm available in the main branch of *hypre*; however, we have not yet found a way to write the implementation in such a way that another application could easily take advantage of our algorithm.

When we introduced the ParMETIS-based domain decomposition, we were expecting to find strong-scaling slopes that were closer to optimal. The reality is that although the baseline was improved substantially (i.e., the overall curve shifted down), the slope remained largely unchanged. This was true regardless of the compute cluster used (i.e. Summit or Eagle). Figures 3 and 7 show that for 24 Summit nodes, the pressure-Poisson system consumes 60%-70% of a time step. Thus the strong scaling of the entire application is largely determined by the strong scaling of the AMG-preconditioned GMRES solver for pressure-Poisson. As shown in Figure 11, this is both highly hardware architecture and MPI-version dependent. We hypothesize that the larger, refined mesh would perform substantially better if run on an appropriately sized machine with the same hardware and software versions as are available on Eagle. Given the drop in performance for the Summit architecture, this suggests a greater collaboration is needed between application scientists, the system engineers at super-computing facilities, and the vendors who supply the critical libraries. AMG-based solvers are critical tool for many applications and they require hardware and software that can keep up with the data flow and computation demands that these sparse algorithms impose.

ACKNOWLEDGMENTS

The National Renewable Energy Laboratory (NREL) is operated by Alliance for Sustainable Energy, LLC, for the U.S. Department of Energy (DOE) under Contract No. DE-AC36-08GO28308. Lawrence Livermore National Laboratory operates under DOE Contract DE-AC52-07NA27344 (LLNL-PROC-821116). Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the DOE's National Nuclear Security Administration (NNSA) under contract DE-NA0003525. This report followed the Sandia National Laboratories formal review and approval process (SAND2021-4327 C). As such, the technical report is suitable for unlimited release. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the DOE Office of Science (SC) and the NNSA, and was performed using computational resources of the Oak Ridge Leadership Computing Facility, which is a DOE SC User Facility supported under Contract DE-AC05-00OR22725, and computational resources sponsored by the DOE's Office of Energy Efficiency and Renewable Energy and located at NREL. The views expressed in the article do not necessarily represent the views of the DOE or the U.S. Government. The U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this work, or allow others to do so, for U.S. Government purposes.

REFERENCES

[1] M.A. Sprague, S. Boldyrev, P. Fischer, R. Grout, W. Gustafson Jr., and R. Moser. Turbulent flow simulation at the Exascale: Opportunities and challenges workshop. Technical report, 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.

[2] F. Alexander et al. Exascale applications: skin in the game. *Phil. Trans. R. Soc.A*, 378, 2020.

[3] T. T. Tran and D. Kim. A CFD study into the influence of unsteady aerodynamic interference on wind turbine surge motion. *Renewable Energy*, 90:204–228, 2016.

[4] P. Messina. The exascale computing project. *Computing in Science Engineering*, 19(3):63–67, 2017.

[5] M.A. 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.

[6] B. Roget and J. Sitaraman. Robust and efficient overset grid assembly for partitioned unstructured meshes. *Journal of Computational Physics*, 260:1–24, 2014.

[7] M.J. Brazell, J. Sitaraman, and D.J. Mavriplis. An overset mesh approach for 3D mixed element high-order discretizations. *Journal of Computational Physics*, 322:33–51, 2016.

[8] W. Zhang and et al. AMReX: A Framework for Block-Structured Adaptive Mesh Refinement. *Journal of Open Source Software*, 4(37):1370, 2019.

[9] R. D. Falgout and U. Meier-Yang. hypre: A library of high performance preconditioners. In *International Conference on computational science*, pages 632–641, 2002.

[10] J. W. Ruge and K. Stüben. Algebraic multigrid. In *Multigrid methods*, pages 73–130. SIAM, 1987.

[11] Y. Saad and M. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Comput.*, 7:856–869, 1986.

[12] Oak Ridge National Laboratory. ORNL Summit user guide. https://docs.olcf.ornl.gov/systems/summit_user_guide.html.

[13] Andrew C. Kirby and Dimitri J. Mavriplis. Gpu-accelerated discontinuous galerkin methods: 30x speedup on 345 billion unknowns. In *2020 IEEE High*

Performance Extreme Computing Conference (HPEC), pages 1–7, 2020.
[14] Thilina Rathnayake, Sanath Jayasena, and Mahinsasa Narayana. Openfoam on gpus using amgx. In *Proceedings of the 25th High Performance Computing Symposium*, HPC '17, San Diego, CA, USA, 2017. Society for Computer Simulation International.
[15] M. Naumov, M. Arsaev, Patrice Castonguay, J. Cohen, J. Demouth, Joe Eaton, S. Layton, N. Markovskiy, I.Z. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka. Amgx: A library for gpu accelerated algebraic multigrid and preconditioned iterative methods. *SIAM Journal on Scientific Computing*, 37:S602–S626, 01 2015.
[16] Michelsen J.A. Basis3d - a platform for development of multiblock pde solvers. Technical Report AFM 92-05, Technical University of Denmark, 1992.
[17] Michelsen J.A. Block structured multigrid solution of 2d and 3d elliptic pdes. Technical Report AFM 94-06, Technical University of Denmark, 1994.
[18] Paul Van der Laan. Ellipsys3d large eddy simulation data of single wind turbine wakes in neutral atmospheric conditions. Technical report, Technical University of Denmark, 2019.
[19] C. Grinderslev, G. Vijayakumar, S. Ananthan, N. Sørensen, F. Zahle, and M. Sprague. Validation of blade-resolved computational fluid dynamics for a mw-scale turbine rotor in atmospheric flow. *Journal of Physics: Conference Series*, 1618:052049, 09 2020.
[20] A. Sharma, S. Ananthan, J. Sitaraman, S. J. Thomas, and M. A. Sprague. Overset meshes for incompressible flows: On preserving accuracy of underlying discretizations. *Journal of Computational Physics*, 428:109987, 2021.
[21] H. Carter Edwards, A.B. Williams, G.D. Sjaardema, D.G. Baur, and W.K. Cochran. Sierra toolkit computational mesh conceptual model. Technical Report SAND2010-1192, Sandia National Laboratories, 2010.
[22] M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. J Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, et al. An overview of the trilinos project. *ACM Trans. Math. Soft. (TOMS)*, 31:397–423, 2005.
[23] G. Karypis, K. Schloegel, and V. Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library. 01 1997.
[24] The Zoltan2 Project Team. The Zoltan2 Project Website.
[25] H. Carter Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.
[26] Rhaleb Zayer, Markus Steinberger, and Hans-Peter Seidel. Sparse matrix assembly on the gpu through multiplication patterns. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8, 2017.
[27] W. Kahan. Pracniques: Further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40, January 1965.
[28] R. D. Falgout, R. Li, B. Sjogreen, and U. Meier-Yang. Porting *hypre* to heterogeneous computer architectures: Strategies and experiences. *submitted to Parallel Computing*, 2020.
[29] S.J. 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. Comput.*, 41:S196–S219, 2019.
[30] A. Brandt, S. McCormick, and J. W. Ruge. Algebraic multigrid (AMG) for sparse matrix equations. In Evans, editor, *Sparsity and Its Applications*. Cambridge University Press, Cambridge, 1984.
[31] K. Stüben. Algebraic multigrid (AMG): an introduction with applications. In Ulrich Trottenberg and Anton Schuller, editors, *Multigrid*. Academic Press, Inc., USA, 2000.
[32] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. Meier-Yang. *Scaling Hypre's Multigrid Solvers to 100,000 Cores*, pages 261–279. Springer London, London, 2012.
[33] H. De Sterck, U. Meier-Yang, and J. J. Heys. Reducing complexity in parallel algebraic multigrid preconditioners. *SIAM Journal on Matrix Analysis and Applications*, 27(4):1019–1039, 2006.
[34] M Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15:1036–1053, 1986.
[35] T. Manteuffel, S. McCormick, M. Park, and J. Ruge. Operator-based interpolation for bootstrap algebraic multigrid. *Numerical Linear Algebra with Applications*, 17(2-3):519–537, 2010.
[36] H. De Sterck, R. D. Falgout, J. W. Nolting, and U. Meier-Yang. Distance-two interpolation for parallel algebraic multigrid. *Numerical Linear Algebra with Applications*, 15(2-3):115–139, 2008.
[37] R. Li, B. Sjogreen, and U. Meier-Yang. A new class of AMG interpolation operators based on matrix matrix multiplications. *To appear SIAM Journal on Scientific Computing*, 2020.
[38] U. Meier-Yang. On long-range interpolation operators for aggressive coarsening. *Numerical Linear Algebra with Applications*, 17(2-3):453–472, 2010.
[39] K. Świrydowicz, J. Langou, S. Ananthan, U. Meier-Yang, and S.J. Thomas. Low synchronization Gram-Schmidt and GMRES algorithms. *Num. Linear Alg. Appl.*, 28:1–20, 2020.
[40] S. Thomas, I. Yamazaki, L. Berger-Vergiat, J. Hu, B. Kelly, , P. Mullowney, S. Rajamanickam, and K. Świrydowicz. Two-stage Gauss–Seidel preconditioners and smoothers for Krylov solvers on a GPU cluster. *SIAM J. Sci Comput.*, 2021.
[41] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. Meier-Yang. Multigrid smoothers for ultraparallel computing. *SIAM J. Sci. Comput.*, 33:2864–2887, 2011.
[42] J. Jonkman, S. Butterfield, W. Musial, and G. Scott. Definition of a 5-MW reference wind turbine for offshore system development. Technical Report NREL/TP-500-38060, National Renewable Energy Laboratory, 2009.
[43] F. Kong, R. Stogner, D. Gaston, J. Peterson, C. Permann, A. Slaughter, and R. Martineau. A general-purpose hierarchical mesh partitioning method with node balancing strategies for large-scale numerical simulations. In *2018 IEEE/ACM 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 65–72, 11 2018.
[44] National Renewable Energy Laboratory. Eagle computing system. https://www.nrel.gov/hpc/eagle-system.html.