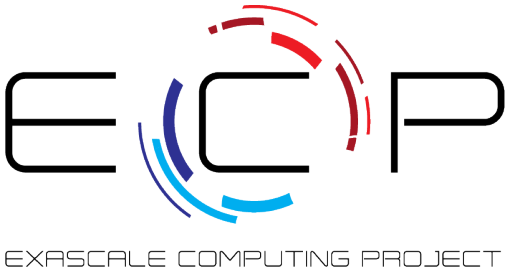
SAND2017-10549R




# ECP Milestone Report

# Deploy Nalu/Kokkos algorithmic infrastructure with performance benchmarking

## WBS 1.2.1.07, Milestone ECP FY17 Q4

Stefan Domino, Shreyas Ananthan, Robert Knaus, and Alan Williams

29 September 2017

# ECP Milestone Report
# Deploy Nalu/Kokkos algorithmic infrastructure with performance benchmarking
# WBS 1.2.1.07, Milestone ECP FY17 Q4

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

29 September 2017

# ECP Milestone Report
# Deploy Nalu/Kokkos algorithmic infrastructure with performance benchmarking
# WBS 1.2.1.07, Milestone ECP FY17 Q4

## Approvals

**Submitted by**:

29 Sept 2017

Stefan P. Domino (SNL)
ECP FY17 Q4

Date

**Approval**:

Douglas B. Kothe, Oak Ridge National Laboratory
Director, Exascale Computing Project

Date

# Revision Log

| Version | Creation Date | Description | Approval Date |
|---|---|---|---|
| 1.0 | 29 September 2017 | Original | |

## EXECUTIVE SUMMARY

The former Nalu interior heterogeneous algorithm design, which was originally designed to manage matrix assembly operations over all elemental topology types, has been modified to operate over homogeneous collections of mesh entities. This newly templated kernel design allows for removal of workset variable resize operations that were formerly required at each loop over a Sierra ToolKit (STK) bucket (nominally, 512 entities in size). Extensive usage of the Standard Template Library (STL) `std::vector` has been removed in favor of intrinsic Kokkos memory views. In this milestone effort, the transition to Kokkos as the underlying infrastructure to support performance and portability on many-core architectures has been deployed for key matrix algorithmic kernels. A unit-test driven design effort has developed a homogeneous entity algorithm that employs a team-based thread parallelism construct. The STK Single Instruction Multiple Data (SIMD) infrastructure is used to interleave data for improved vectorization. The collective algorithm design, which allows for concurrent threading and SIMD management, has been deployed for the core low-Mach element-based algorithm. Several tests to ascertain SIMD performance on Intel KNL and Haswell architectures have been carried out. The performance test matrix includes evaluation of both low- and higher-order methods. The higher-order low-Mach methodology builds on polynomial promotion of the core low-order control volume finite element method (CVFEM). Performance testing of the Kokkos-view/SIMD design indicates low-order matrix assembly kernel speed-up ranging between two and four times depending on mesh loading and node count. Better speedups are observed for higher-order meshes (currently only P=2 has been tested) especially on KNL. The increased workload per element on higher-order meshes benefits from the wide SIMD width on KNL machines. Combining multiple threads with SIMD on KNL achieves a 4.6x speedup over the baseline, with assembly timings faster than that observed on Haswell architecture. The computational workload of higher-order meshes, therefore, seems ideally suited for the many-core architecture and justifies further exploration of higher-order on NGP platforms. A Trilinos/Tpetra-based multi-threaded GMRES preconditioned by symmetric Gauss Seidel (SGS) represents the core solver infrastructure for the low-Mach advection/diffusion implicit solves. The threaded solver stack has been tested on small problems on NREL's Peregrine system using the newly developed and deployed Kokkos-view/SIMD kernels. Efforts are underway to deploy the Tpetra-based solver stack on NERSC Cori system to benchmark its performance at scale on KNL machines.[1]

[1]Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and 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 programatic review and approval process (tracking number 692834). As such, the technical report is not suitable for unlimited release.

# TABLE OF CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

# 1. INTRODUCTION

The Nalu code base, which has been and continues to be actively developed under github[2], is characterized as a low-Mach turbulent flow code that supports both low-order topological element types, e.g., hexahedral, tetrahedral, pyramids, and wedges, and arbitrary higher-order hexahedral promotion.

For a heterogeneous collection of elements that may be associated with a particular wind energy application (for example, the FY17/Q3 ExaWind milestone employed a low-order hybrid mesh), interior mesh entities, i.e., nodes, edges, faces, and elements, are iterated to provide a variety of algorithmic calculations. In the low-Mach implementation, matrix assembly and solver time dominate the overall solution time as compared to, Courant and Reynolds number post-processing, turbulence averaging, etc. As a first step towards deploying a performing, portable algorithmic design, the milestone effort has concentrated on interior matrix assembly for the core low-Mach equation set. A well performing, portable abstraction layer between the code's internal assembly kernels and the underlying architecture, e.g., threading or GPU, is desired. For this project, the Kokkos infrastructure [8] is extensively used as the performance/portability layer.

In this introduction section, a high-level overview of the Sierra Toolkit abstractions is provided. The milestone description is provided in section 2. In order to provide context to the underlying design points for the core assembly kernels, an overview of the numerical method is provided in section 3. Details on code implementation amd the Kokkos-based Nalu kernel design points are provided in section 4. Results and a path forward for future efforts are provided in section 5 and 6, respectively.

## 1.1 Core Sierra Toolkit Abstractions

In order to gain understanding of the work deployed within this milestone effort, a brief overview of the Nalu/Sierra Toolkit (STK) [7] interface is useful. Consider a typical mesh that consists of nodes, sides of elements and elements. Such a mesh, when using the Exodus [10] standard, will likely be represented by a collection of "element blocks", "sidesets" and, possibly, "nodesets". The definition of the mesh (generated by the user through commercial meshing packages such as Pointwise or ICEM-CFD) will provide the required spatial definitions of the volume physics and the required boundary conditions.

An element block is a homogeneous collection of elements of the same underlying topology, e.g., `HEXAHEDRAL-8`. A sideset is a set of exposed element faces on which a boundary condition is to be applied. Finally, a nodeset is a collection of nodes. In general, a particular discretization choice may require `stk::mesh::Entity` types of element, face (or side), edge and node. Variables, such as coordinates, velocity, pressure, etc., registered on the mesh are of type: `stk::mesh::field`. Gathering operations are defined by iterating a mesh entity, such as an element, extracting the nodal data via a `stk::mesh::field_data()` call, and copying this data to a local data structure that can be used elsewhere in the algorithmic structure.

Therefore, the high-level set of requirements on a mesh infrastructure is to support a parallel interface to iterate parts of the mesh and to assemble a quantity of interest to a nodal or elemental field. Consider a case in which a hybrid mesh consisting of a hexahedral, tetrahedral and pyramid topology has been created. As already noted, this mesh would be represented as a set of homogeneous parts of interior elements: `HEXAHEDRAL-8`, `TETRAHEDRAL-4` and `PYRAMID-5` and a set of exposed surfaces on which boundary conditions are applied.

In a typical algorithm, a developer might want to *select* a set of parts of the mesh, e.g., `block-1`, and iterate these mesh entities to provide useful work towards the simulation goal. In the following example, it is desired to obtain a selector that contains all of the parts of interest to a physics algorithm that are locally owned and active.

```
// define the selector; locally owned, the parts I have served up and active
stk::mesh::Selector s_locally_owned_union = metaData_.locally_owned_part()
  & stk::mesh::selectUnion(partVec_)
  & !(realm_.get_inactive_selector());
```

**Listing 1:** Basic selector usage in STK.

[2] https://github.com/NaluCFD

Given this above selector, a developer can now extract a set of parts, above held in the `partVec_` data structure, that conforms to the above selector rule, e.g., owned and active. The internal STK abstraction that manages such iterations is a `stk::mesh::Bucket`,

```
// given the defined selector, extract the buckets of type ''element''
stk::mesh::BucketVector const& elem_buckets
  = bulkData_.get_buckets( stk::topology::ELEMENT_RANK, s_locally_owned_union );

// loop over the vector of buckets
for ( const stk::mesh::Bucket* bptr : elem_buckets) {
  const stk::mesh::Bucket & bucket = bptr;

  // extract master element (homogeneous over buckets)
  MasterElement *meSCS = realm_.get_surface_master_element(bucket.topology());

  for ( stk::mesh::Entity elem : bucket ) {
    // operate on this elem
    // etc...
  }
}
```

**Listing 2:** Basic bucket looping STK.

In the above example, all elements that are locally owned and active will be provided within an inner bucket loop. Once the element is in hand, data can be gathered while matrix contributions can be scattered to the full system. Note that the same design pattern is used to iterate nodes, edges and exposed surfaces. As such, the above set of pseudo-code represents a pattern within any application code that interfaces STK. In the above example (and in the pre-ECP Nalu code project design), `partVec_` is a vector of parts that hold the three `HEXAHEDRAL-8`, `TETRAHEDRAL-4` and `PYRAMID-5` topologies. Therefore, the iteration over desired elements is heterogeneous in nature. The heterogeneous nature of the collection of parts requires the extraction of integration rule for each bucket provided to the developer. As will be seen in future sections, removal of this design point is desired such that resizing is removed.

## 2. MILESTONE DESCRIPTION & REQUIREMENTS

### 2.1 Milestone Description:

The base algorithmic design in Nalu allows for a single instantiation of a single algorithm, e.g., matrix assembly, to operate on many parts of the mesh that can be heterogeneous in nature, e.g., hex, tet, pyramid, wedge. This design aspect requires re-sizing of workset arrays at the execute STK bucket level. Such resize operations have been shown to negatively affect performance. The intent of this milestone is to drive algorithmic changes that remove the need to resize workset arrays at the execution phase and, rather, elevate this step at algorithm construction. Therefore, algorithmic construction will occur for each unique topology thereby removing the need to resize at execution. Kokkos views will be integrated into the algorithmic design along with porting the Kokkos/Nalu `apply_coefficient()` interface from the CTROTT github location to the master Nalu code base.

### 2.2 Milestone Execution Plan:

1. Algorithm infrastructure modification to operate on a homogeneous topology set; step 1 is to elevate std::vector resize operations to the algorithm constructor.
2. Transition of matrix assemble algorithms to use Kokkos-based matrix `apply_coefficient()`.
3. Usage of Kokkos views in solver-based algorithms.

### 2.3 Milestone Completion Criteria:

A report will be created documenting evaluated and executed algorithm design, and simulation timings along with any identified bottlenecks. Code created will be posted on the public repository along with relevant documentation.

## 3. NUMERICAL OVERVIEW

The core ExaWind numerical methodology falls within the class of vertex-centered finite volume schemes with specific emphasis on the control volume finite element method (CVFEM) [9]. In this section, a brief overview of the underlying discretization is provided. Motivation for possible higher-order methods is provided based on recent detailed method of manufactured solution (MMS) findings. For more information on the core algorithm, the reader is referred to the Nalu theory document, [6], or recent low- and higher-order descriptions [5]. The suite of numerical methods under consideration can be found in [4].

### 3.1 Dual Volume Definition

The control volume finite element method simply defines a dual mesh constructed within each element. A weak variational statement is written and a piece-wise constant test function is applied. Figure 1 provides an overview of both low-order and higher-order node and dual volume rules. Integration points are defined at two locations: for flux calculations, the method uses subcontrol surfaces; for source and time contributions, the subcontrol volume center of each dual volume is used. To recover a low-order edge-based vertex-centered (EBVC) method, dual volume subcontrol area vectors are assembled to the edges of the low-order elements while dual nodal volumes are assembled to the nodes of the elements. Both the edge- and element-based schemes can be considered a Petrov-Galerkin method.

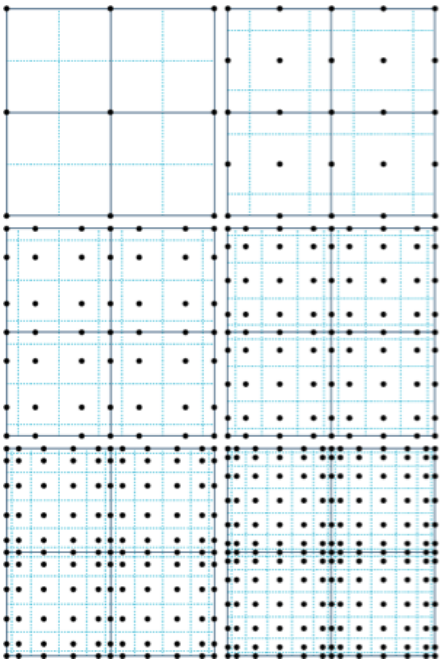


**Figure 1:** Polynomial promotion for a canonical CVFEM quad element patch from $P$=1 (upper left) to $P$=6 (lower right).

Recent code verification studies in support of the FY18/Q1 sliding mesh effort have benchmarked accuracy between the edge- and element-based scheme. In Figure 2, a non-conformal interface is recovered via a mixed CVFEM/DG scheme (shown are the temperature shadings for the MMS). The details of this low-Mach fluids algorithm will be decribed in future project milestones. As such, the inclusion of this use-case within this

report is to motivate higher-order methods development by highlighting a MMS study that exercises the suite of discretizations under investigation within the ExaWind project.

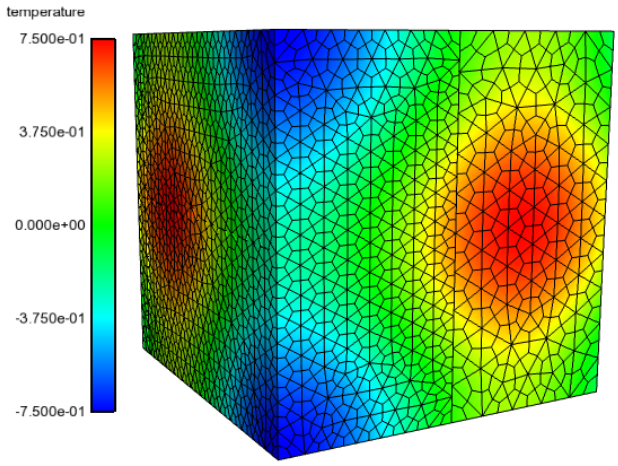



**Figure 2:** Temperature manufactured solution field shadings for the mixed CVFEM/DG MMS study.

In Figure 3, $L_2$ error norms are provided for both the P=1 edge- and element-based scheme and the P=2 element-based scheme. An interesting, yet common, finding in such verification studies is the dramatic error reduction that the higher-order method notes even at modest polynomial promotions. Specifically, in this study, the most refined low-order simulation does not reproduce an error as low as the coarsest P=2 simulation. Assuming that the convergence trends hold, the low-order solution would require approximately three uniform refinements in order to replicate the P=2 error on the coarsest mesh. The error norm reduction benefit found in higher-order schemes has motivated the ExaWind project to pursue this class of discretization approaches. For details regarding the usage of the higher-order methods in a recent LES study, the reader is referred to [5].

From the computer science perspective, it hypothesized that increased local work to the element can possibly be more efficient on next generation platforms. As such, the project is aggressively prioritizing CVFEM (with possible mixed order of underlying polynomial basis) over the edge-based scheme. In this milestone report, all algorithmic work and performance benchmarking will be in the context of low- and higher-order CVFEM.

## 4. IMPLEMENTATION IN NALU

### 4.1 Nalu algorithm and kernel design

To perform linear system assembly, Nalu algorithms iterate over parts of the mesh and for each mesh element, compute dense matrices and vectors to be contributed (scattered) into the sparse linear system. Mesh elements are contained in "parts" such that elements within a part are homogeneous with respect to topology. Parts are further subdivided into "buckets" which represent intersections of potentially overlapping parts. Buckets are the actual containers that are iterated by Nalu's algorithms.

Consider a hybrid mesh simulation in which we have a variety of topologies. For simplicity, assume that we have the topologies quad4, quad9 and tri3, as shown in Figure 4, contained within the incoming Exodus mesh.

A core question is how to manage algorithm contributions in this hybrid mesh use case? In the previous approach used in Nalu, there is one algorithm that holds all parts. Each algorithm type (INTERIOR, WALL_BC, INFLOW_BC) is iterated by `EquationSystem::solve_and_update()`. Also, scratch arrays must be resized at the bucket level due to this heterogeneous design.

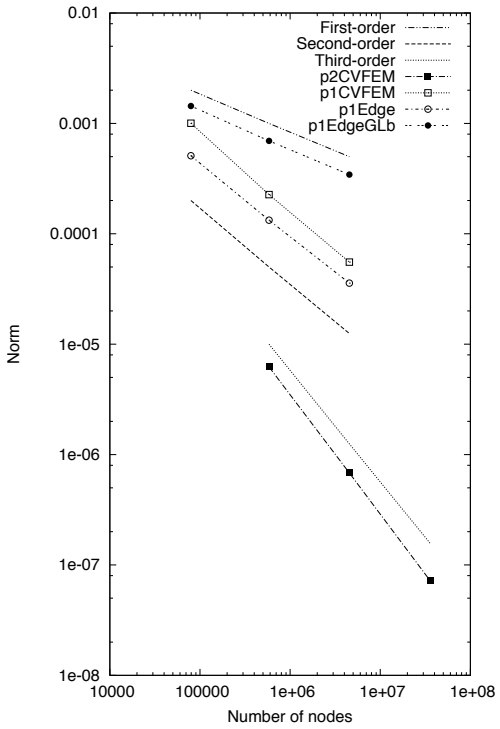



**Figure 3:** Steady Laplace low- and high- order $L_2$ error norms.

As part of this milestone effort, the fundamental design point associated with resizing scratch arrays within the execution stage of the algorithm was removed. In addition to improved code design, performance improvements were hypothesized. The new design is to create one algorithm per unique topology, which allows for allocating scratch arrays of the correct size at algorithm construction, with no need to resize during mesh entity iteration. Furthermore, the algorithms are templated on traits that provide compile-time values for variables such as number of nodes per element, number of integration points, etc. For the above example of four blocks in a mesh, there would be three algorithm instances: MomentumElemSolver_Q4, MomentumElemSolver_Q9, MomentumElemSolver_T3. MomentumElemSolver_Q4 would have two parts, all others would only have a single mesh part.

In the original algorithmic design, unique algorithms for each of the independent variables are created. For a typical wind energy application, momentum, continuity and turbulent algorithms are created. In each of these specialized algorithms, the interface to STK is duplicated while the core algorithmic contribution is coded within the bucket loop. A primary point that the team identified, in addition to interfacing Kokkos, was a clean algorithmic design that would minimize duplicated STK/Kokkos interface code and alleviate this technical understanding to a domain specialist. The `nalu::Kernel`, which will be described in the next section, represents the design structure that would provide this generality and separation between the NGP interface and the desired physics to be coded.

### 4.2 Computational Kernels and data sharing

For each mesh element visited during algorithm execution, a number of computational kernels are called to compute contributions for the linear system. A `nalu::Kernel` is simply defined by an algorithmic contribution that provides a particular matrix contribution. For example, an advection and diffusion kernel manages flux-based contributions. A time integration kernel provides the implicit contribution. In one particular physics simulation, as in an atmospheric boundary layer simulation, a momentum buoyancy source term

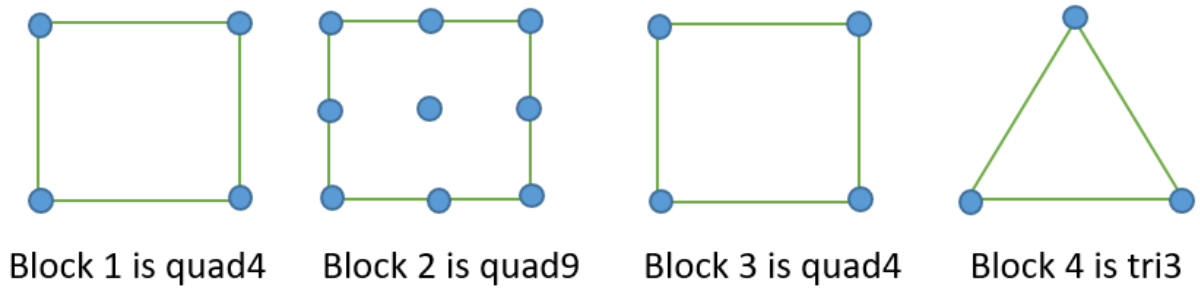



**Figure 4:** Heterogeneous topologies example.

might be required, while in another, an advanced stabilization option may be activated. However, the code design point is to reduce runtime logic checks in favor of a polymorphic design.[3] Finally, as noted in the previous section, it is desired that low-level Kokkos-based calls are contained outside the kernel, thereby providing an abstraction layer for the developer to code outside a threaded/SIMD interface. The pseudo-code in listing 3 outlines the general approach.

```
Algorithm::execute()
{
  for(elem : elements) {
    //gather coords, compute gradients, etc
    for(kernel : computeKernels) {
      kernel->execute(elem, prereq_data, lhs, rhs);
    }
    apply_coeff(lhs, rhs, ...);
  }
}
```

**Listing 3:** Basic Algorithm/Kernel loop structure.

The contribution to `prereq_data`, which is used by all kernels, varies from kernel to kernel. However, the superset of all prerequisites over all created kernels can include data overlap. As such, a required design for kernel design is to eliminate duplicate field gathers and geometric calculations. Creating a common set of field gathers and geometric calculations over all created kernels removes redundant work. In essence, the kernels are no longer autonomous when gathering required field data. However, each kernel is designed to provide a specific algorithmic contribution. Furthermore, if the loop over elements is parallel (multi-threaded), the `prereq_data` must be unique per thread (each thread is processing a different element.)

In the new `nalu::Kernel` design, classes are templated on the algorithmic trait (herein to imply topological integration rule). Listing 4 outlines a typical `nalu::Kernel` constructor while Listing 5 outlines a typical Kokkos view usage.

```
template<typename AlgTraits>
MomentumNSOElemKernel<AlgTraits>::MomentumNSOElemKernel(
  ElemDataRequests& dataPreReqs)
{
  // define master element rule for this kernel
  MasterElement *meSCS
    = sierra::nalu::MasterElementRepo::get_surface_master_element(AlgTraits::topo_);

  // add ME rule
  dataPreReqs.add_cvfem_surface_me(meSCS);
```

[3] It is anticipated that transitioning to the GPU will require a slight interface change to allow for polymorphic-based kernel design

```
  // add fields to gather
  dataPreReqs.add_coordinates_field(*coordinates_, AlgTraits::nDim_, CURRENT_COORDINATES);
  dataPreReqs.add_gathered_nodal_field(*velocityNp1_, AlgTraits::nDim_);

  // add ME calls
  dataPreReqs.add_master_element_call(SCS_GIJ, CURRENT_COORDINATES);
}
```

**Listing 4:** Attributes of a kernel; part A the constructor.

```
template<typename AlgTraits>
void
MomentumNSOElemKernel<AlgTraits>::execute(
  SharedMemView<DoubleType**>& lhs,
  SharedMemView<DoubleType *>& rhs,
  ScratchViews<DoubleType>& scratchViews)
{
  SharedMemView<DoubleType**>& v_uNp1
    = scratchViews.get_scratch_view_2D(*velocityNp1_);
  SharedMemView<DoubleType***>& v_gijUpper
    = scratchViews.get_me_views(CURRENT_COORDINATES).gijUpper;

  for ( int ip = 0; ip < AlgTraits::numScsIp_; ++ip ) {

    // determine scs values of interest
    for ( int ic = 0; ic < AlgTraits::nodesPerElement_; ++ic ) {

    // assemble each component
    for ( int k = 0; k < AlgTraits::nDim_; ++k ) {

      // determine scs values of interest
      for ( int ic = 0; ic < AlgTraits::nodesPerElement_; ++ic ) {

        // save off velocityUnp1 for component k
        const DoubleType& ukNp1 = v_uNp1(ic,k);

        // denominator for nu as well as terms for "upwind" nu
        for ( int i = 0; i < AlgTraits::nDim_; ++i ) {
          for ( int j = 0; j < AlgTraits::nDim_; ++j ) {
            gUpperMagGradQ += constant*v_gijUpper(ip,i,j);
          }
        }
     }
  }
}
```

**Listing 5:** Attributes of a kernel; part B the body.

The usage of Kokkos multi-dimentional arrays significantly reduces possible developer error in hand-rolled indexing, see Listing 6.

```
      // before pointer offset
      const double *p_gUpper = &ws_gUpper_[nDim_*nDim_*ip];

      // determine scs values of interest
      for ( int ic = 0; ic < AlgTraits::nodesPerElement_; ++ic ) {

        // before/after velocity component k
```

```
const double ukNp1 = ws_uNp1(ic*nDim+k);
const DoubleType& ukNp1 = v_uNp1(ic,k);

for ( int i = 0; i < AlgTraits::nDim_; ++i ) {
  for ( int j = 0; j < AlgTraits::nDim_; ++j ) {
    // before/after complex tensor product
    gUpperMagGradQ +=
      constant*p_gUpper[i*nDim_+j]*ws_Gju_[row_ws_Gju+k*nDim_+j]*axi;
    gUpperMagGradQ +=
      constant*v_gijUpper(ip,i,j)*v_Gju(ic,k,j)*axi;
  }
```

**Listing 6:** Before and after MD-array usage.

Thread parallelism was implemented using the Kokkos library. The Kokkos interface uses the `SharedMemView` mechanism for creating thread-local scratch arrays for use by the computational kernels. Since the prereq data for kernels potentially consists of quite a few arrays (e.g., coordinates, scs-area-vec, scv-volume, dndx, etc), these arrays are grouped into a `ScratchViews` structure. The representative algorithm pseudo-code is shown in listing 7:

```
Algorithm::execute()
{
  int bytes_per_thread = //scratch bytes per element
  auto team_exec = get_team_policy(... bytes_per_thread, ...);
  Kokkos::parallel_for(team_exec, buckets, [&](team)
  {
    ScratchViews scratchViews(topo, meSCS, dataNeeded ...);
    Kokkos::parallel_for(team, bucket.size()) {
      fill_pre_req_data(dataNeeded, elem, ..., scratchViews);
      for(kernel : computeKernels) {
         kernel->execute(elem, lhs, rhs, scratchViews);
      }
    }
    apply_coeff(lhs, rhs, ...);
  }
}
```

**Listing 7:** Thread-parallel Algorithm structure.

In general, the lifespan of the `ScratchViews` struct is significant. It must be created inside the outer `Kokkos::parallel_for` loop. Moreover, the struct contains views into a single block of allocated memory that Kokkos creates just once resulting in improved performance.

### 4.3 Integration of Tpetra Assembly Threading via Kokkos

Linear system assembly includes creating and initializing a sparse graph, and managing local and shared/ghosting contributions to coefficient matrix and vector objects. Thread-safe and thread-scalable assembly, which was obtained by using Kokkos data structures, to accumulate graph-edges (connections in the sparsity pattern) has been achieved. This work was initially prototyped by Christian Trott and company in a separate repository as part of a SNL/ASC effort to develop STK/Kokkos/Application design patterns for algorithms. This code base was incorporated within the main repository in support of the present milestone. The primary Kokkos structure used for the accumulation of sparsity pattern connections is `Kokkos::UnorderedMap`. It allows for thread-safe insertion, which is not available in standard library (STL) containers. The implementation also uses Kokkos structures for summing into coefficient values. The Trilinos Tpetra/Kokkos libraries will ultimately support assembly on GPU devices. It should be noted that several aspects of graph initialization are still not thread-safe, including the insertion of indices into `Tpetra::CrsGraph` via the `insertGlobalIndices` method. Thus graph initialization in Nalu is still not benefiting from threaded execution. This will be completed during follow-on work.

### 4.4 SIMD Vectorization

Explicit SIMD instructions allow Nalu to take advantage of vector processing units even when compiler auto-vectorization fails. Compiler auto-vectorization currently is not up to the task of vectorizing realistic computational loops. The STK libraries (which are distributed with the Trilinos [2] libraries) include a "`stk_simd`" module which handles the task of detecting which set of vector instructions is available on a given platform. Possible vector instruction-sets include SSE (vector-width 2), AVX (vector-width 4), and AVX512 (vector-width 8). These vector instruction sets are documented Intel Intrinsics [1]. The `stk_simd` module also abstracts the calls to specific vector instructions to provide portability. As an example, the `stk_simd` function `stk::simd::sqrt(..)` will automatically call through to `_mm256_sqrt_pd(..)` if AVX instructions are available, or `_mm512_sqrt_pd(..)` if AVX512 instructions are available, or simply `std::sqrt(..)` if no vector instructions are available.

The `stk_simd` module provides a data type `stk::simd::Double` which is an array of doubles of the correct length for the vector instruction set that is enabled. That length is provided by the constant `stk::simd::ndoubles`. `stk_simd` also provides basic math operators and initialization operators so that usage code is impacted as little as possible. The syntax for basic usage is shown in listing 8.

```
stk::simd::Double x = 1.0;
stk::simd::Double y = 2.0;
stk::simd::Double z = stk::math::sqrt(x*y);
```

**Listing 8:** Basic SIMD Usage.

The general approach that has been used for deployment of SIMD in Nalu follows the approach used by the Sierra/Thermal-Fluids ASC FY17 Level two milestone team, and is described below.

A `typedef DoubleType`, which can be set to `stk::simd::Double` when appropriate, has been introduced within the code base. This allows computational kernels to be coded in terms of DoubleType, and to then be largely un-impacted by SIMD vs non-SIMD execution, apart from the need to convert certain intrinsic operations from a standard-library version such as, `std::sqrt`, to the `stk::math` version. The only explicit handling we have to do is at the Algorithm level where we interleave element data such that when a computational kernel is executed, it operates on data where neighboring entries in each `DoubleType` correspond to different elements. In other words, the kernel is operating on `stk::simd::ndoubles` elements at the same time.

As a simple illustration, consider a pseudo-code function, shown in listing 9, which computes a right-hand-side contribution as a function of density.

```
void execute(const DoubleType* densityAtElemNodes, DoubleType* rhs)
{
    for(i : nodesPerElement) {
        rhs[i] = f(densityAtElemNodes[i]);
    }
}
```

**Listing 9:** Function using DoubleType.

When `DoubleType` is `stk::simd::Double`, and `stk::simd::ndoubles` is 4, the function computes 4 element rhs contributions from 4 elements-worth of nodal-density values. Listing 10 shows pseudo-code for the data handling that is performed at the algorithm level to interleave and de-interleave the data on which the kernels operate.

```
int numSimdElems = stk::simd::ndoubles;
ScratchViews<double> prereqData[numSimdElems];
ScratchViews<DoubleType> simdPrereqData;
for(i : numSimdElems) {
  gather_coords_and_other_fields(elem[i], prereqData[i]);
}
```

```
copy_and_interleave(prereqData, simdPrereqData);

master_elem_operations(simdPrereqData);

SharedMemView<DoubleType> simdLhs, simdRhs;
for(kernel : computeKernels) {
  kernel->execute(simdPrereqData, simdLhs, simdRhs);
}

SharedMemView<double> lhs, rhs;
for(i : numSimdElems) {
  extract_vector_lane(simdLhs, i, lhs);
  extract_vector_lane(simdRhs, i, rhs);
  apply_coeff(..., rhs, lhs);
}
```

**Listing 10:** SIMD interleave/de-interleave in Nalu.

The specific data interleave procedure from Listing 9 is shown in Figure 5.

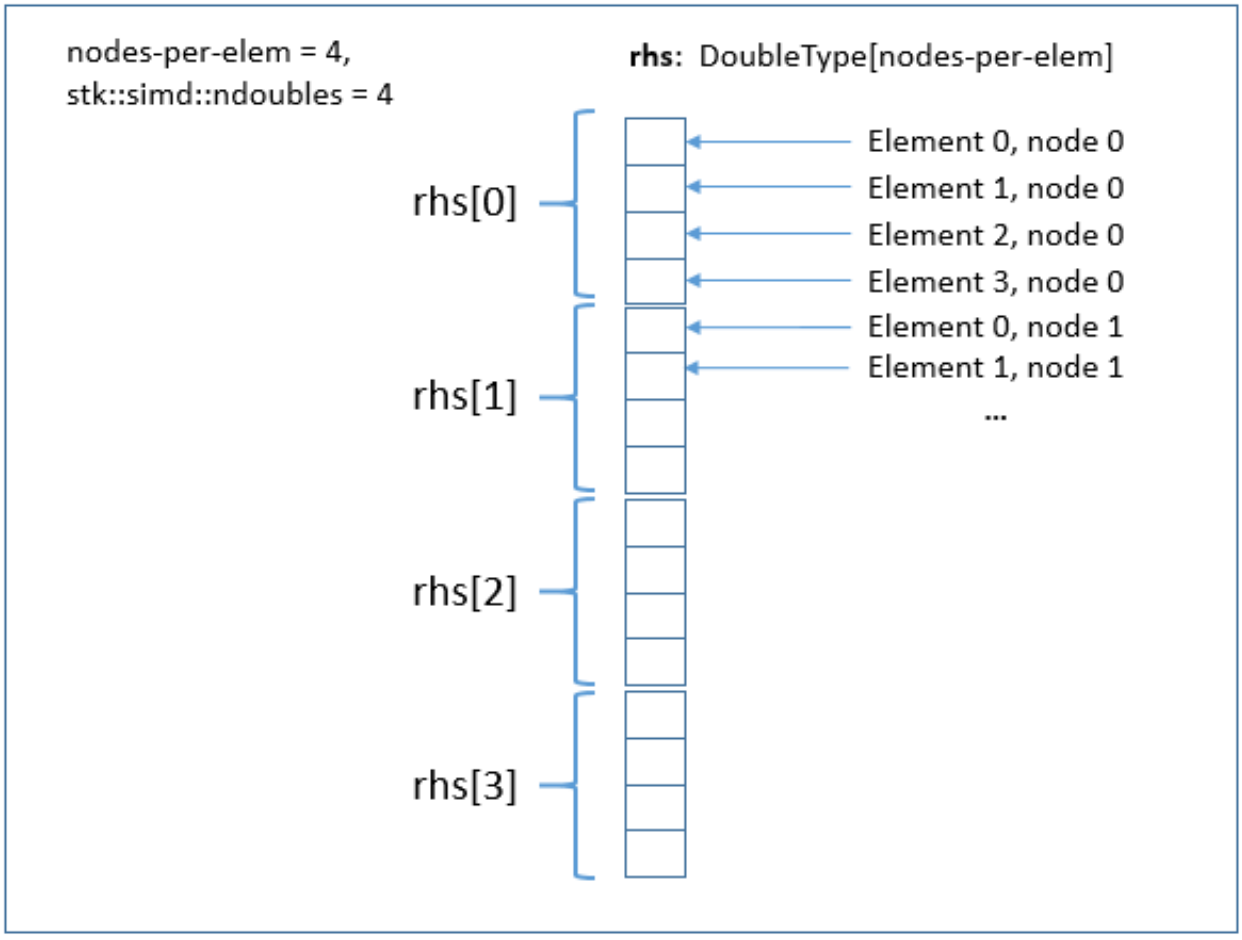



**Figure 5:** Interleaved array illustration.

To assess the performance of this approach, a unit-test was constructed that runs the algorithm from Listing 10 on a relatively simple hex-8 mesh. The test creates and runs two different element-source kernels, MomentumNSOElem and MomentumAdvDiff. The team was able to run this test for a mesh size of 125000

elements with a single MPI rank, with a single thread, on a KNL node where the vector-width is 8 (AVX512 instructions). This was run with and without the above SIMD enhancements and profiled both with the Intel Vtune [3] performance analysis tool. Figure 6 shows the times for the no-SIMD version of the test, as reported by vtune.

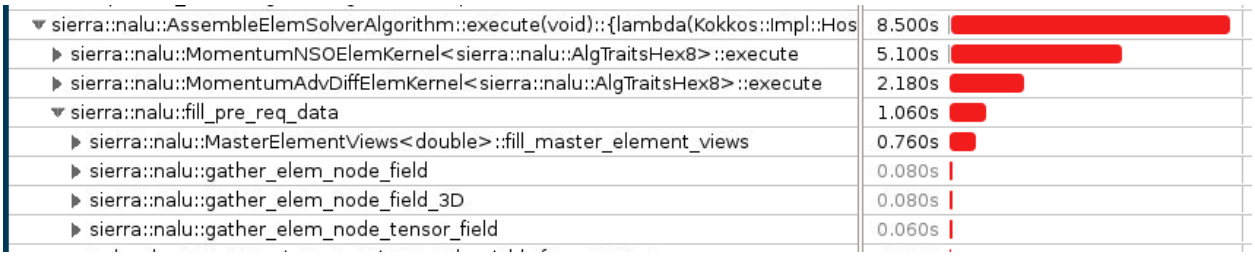



**Figure 6:** Two Momentum Kernels unit-test, no SIMD, 125000 Hex-8 elements, KNL architecture.

Figure 7 shows the times for the SIMD version of the test. Some aspects of the reported times deserve clarification. The gather_coords_and_other_fields function in the pseudo-code (listing 10) is represented by `sierra::nalu::fill_pre_req_data` in the vtune output. Also, the master_elem_operations function in the pseudo-code listing is represented by `sierra::nalu::fill_master_element_views`.

Comparing the vtune outputs for the two runs (with and without SIMD), it is noted that the kernels themselves show speedups of 5.5x for the nonlinear stabilization operator element kernel, and 3.3x for the advection/diffusion kernel. Since the vector-width is eight on this platform, ideal speedups closer to 8x would have been realized. However, obtaining ideal speedups appears to be dependent on ideal memory access streaming, with no cache misses, etc. The speedups we observe are consistent with those reported by other projects evaluating and deploying the SIMD infrastructure.

The overhead of the SIMD approach is represented in the `copy_and_interleave` and `extract_vector_lane` functions. Initially those were occupying about 17% of the overall algorithm time, however, using the vtune output, the team was able to pursue low-level code optimizations in the copying and interleaving code to lower this overhead (current level of roughly 6%).

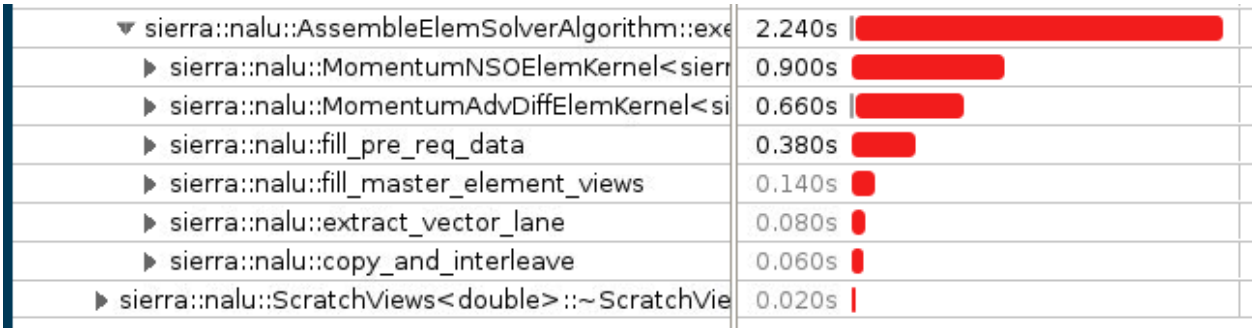



**Figure 7:** Two Momentum Kernels unit-test, 125000 Hex-8 elements, KNL architecture.

## 5. PERFORMANCE RESULTS

In this section, performance results are provided for the Nalu code base using a variety of platforms including Haswell, KNL, and classic Sandy Bridge architectures.

### 5.1 Consolidated Design using Kokkos-based Computational Kernels

The performance improvements for the matrix assembly timings with the new Kokkos and SIMD enabled computations were benchmarked on the NERSC Cori[4] machine. The code was benchmarked on both the Haswell and KNL partitions on the machine for the open-jet test case with the following meshes: 1) coarse low-order ($P = 1$) mesh (17.5M `HEXAHEDRAL-8` elements); 2) fine low-order ($P = 1$) mesh (140M `HEXAHEDRAL-8` elements); and 3) a high-order ($P = 2$) mesh obtained using element promotion of the coarse low-order mesh once (17.5M `HEXAHEDRAL-27` elements). The non-consolidated kernel design from the original Nalu source code was chosen as the baseline for performance comparisons. The performance was then benchmarked successively for various algorithmic design changes introduced in the code:

**Hybrid** The default production run setup used in Nalu before ExaWind project. This configuration is termed *hybrid* because it mixes element contributions and nodal source contributions while assembling the terms in the Navier-Stokes equation. The hybrid method eliminates the necessity to perform sub-control volume calculations that is necessary for the lumped-mass element source terms that will be used in a pure element-based algorithm implementation. While the matrix assembly timings of the *hybrid* implementation is not a correct baseline for evaluating the gains from Kokkos-based threading and SIMD vectorization, it does however, provide an estimate of the actual gains that a casual Nalu user might observe when moving from the current release to the latest version used in this project. Furthermore, a hybrid configuration does not provide design order of accuracy for high-order meshes. Therefore, this configuration is only run for the low-order meshes. A transition strategy to support nodal-based assembly is captured in section 6.

**Baseline B** Lumped mass element based source terms for consistent assembly timing comparison with the consolidated algorithm design explored in this study.

**Configuration C1** Consolidated Kernel design using `ScratchViews` and element data registration (Sec. 4.2). This configuration uses all the terms used in the Baseline.

**Configuration C2** Consolidated Kernels with SIMD datatype (i.e., `DoubleType` is set to `stk::simd::Double`) – see Sec. 4.4.

**Configuration C3** Consolidated Kernels with SIMD on multiple OpenMP threads.

**Configuration C4** Improvements to *SIMD interleave* code and matrix `sumInto` logic based on performance analysis using VTune from last step.

The matrix assembly times were compared for the four equation systems: momentum, continuity, turbulence kinetic energy (TKE), and mixture fraction. Tables 2 and 3 show the matrix assembly timings for the four equation systems on Cori's Haswell and KNL partitions respectively. Table 1 shows the MPI and OpenMP settings that were used for the various runs. To account for the differences in the clock speeds and machine architectures between Haswell and KNL, the simulations were performed on twice the number of MPI ranks on KNL as compared to Haswell. On most runs, the comparison is done on the basis of number of nodes instead of the same MPI ranks.

[4] http://www.nersc.gov/users/computational-systems/cori/configuration/

**Table 1:** Table showing the details of parallel execution (MPI ranks and OpenMP threads) for the various code configuration results shown in this section.

| | Haswell | | | | KNL | | | |
|---|---|---|---|---|---|---|---|---|
| Code Configuration | Num. Nodes | MPI ranks | Ranks per node | OMP Threads | Nodes Nodes | MPI ranks | Ranks per node | OMP Threads |
| **Coarse $P = 1$ mesh (17.5M HEX-8 elements)** | | | | | | | | |
| Hybrid | 5 | 160 | 32 | 1 | 5 | 320 | 64 | 1 |
| Baseline | 5 | 160 | 32 | 1 | 5 | 320 | 64 | 1 |
| C1 | 5 | 160 | 32 | 1 | 5 | 320 | 64 | 1 |
| C2 | 5 | 160 | 32 | 1 | 5 | 320 | 64 | 1 |
| C3 | 10 | 160 | 16 | 2 | 5 | 320 | 64 | 2 |
| C4 | 10 | 160 | 16 | 2 | 5 | 320 | 64 | 2 |
| **Fine $P = 1$ mesh (140M HEX-8 elements)** | | | | | | | | |
| Hybrid | 40 | 1280 | 32 | 1 | 40 | 2560 | 64 | 1 |
| Baseline | 40 | 1280 | 32 | 1 | 40 | 2560 | 64 | 1 |
| C1 | 40 | 1280 | 32 | 1 | 40 | 2560 | 64 | 1 |
| C2 | 40 | 1280 | 32 | 1 | 40 | 2560 | 64 | 1 |
| C3 | 80 | 1280 | 16 | 2 | 40 | 2560 | 64 | 2 |
| C4 | 80 | 1280 | 16 | 2 | 40 | 2560 | 64 | 2 |
| **Coarse $P = 2$ mesh (17.5M HEX-27 elements)** | | | | | | | | |
| Baseline | 40 | 1280 | 32 | 1 | 40 | 2560 | 64 | 1 |
| C1 | 40 | 1280 | 32 | 1 | 40 | 2560 | 64 | 1 |
| C2 | 40 | 1280 | 32 | 1 | 40 | 2560 | 64 | 1 |
| C3 | 80 | 1280 | 16 | 2 | 80 | 2560 | 32 | 2 |
| C4 | 80 | 1280 | 16 | 2 | 80 | 2560 | 32 | 2 |

**Table 2:** Matrix assembly timing comparisons on Haswell partition of NERSC Cori system. Code configurations: H – Hybrid element/node algorithm; B – Baseline non-consolidated algorithm; C1 – consolidated kernel algorithms; C2 – C1 with SIMD datatypes; C3 – C2 with OpenMP threading; C4 – C3 with `sumInto(...)` and SIMD interleave optimizations.

| | Matrix assembly time (s) | | | | | | Speedup ratio | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Equation | Hybrid | Baseline | Conf. C1 | Conf. C2 | Conf. C3 | Conf. C4 | B:C1 | C1:C2 | B:C2 | B:C3 | B:C4 | H:B | H:C4 |
| **Coarse $P=1$ mesh (17.5M HEX-8 elements)** | | | | | | | | | | | | | |
| Momentum | 233.9 | 264.5 | 278.0 | 219.2 | 154.7 | 115.9 | 0.95 | 1.27 | 1.21 | 1.71 | 2.28 | 0.88 | 2.02 |
| Continuity | 60.9 | 60.9 | 61.2 | 50.8 | 36.8 | 27.2 | 0.99 | 1.21 | 1.20 | 1.66 | 2.24 | 1.00 | 2.24 |
| TKE | 70.1 | 93.2 | 94.3 | 70.1 | 52.2 | 41.9 | 0.99 | 1.34 | 1.33 | 1.78 | 2.23 | 0.75 | 1.67 |
| Mix. Frac. | 68.8 | 91.7 | 86.6 | 62.5 | 45.9 | 36.1 | 1.06 | 1.38 | 1.47 | 2.00 | 2.54 | 0.75 | 1.91 |
| **Fine $P=1$ mesh (140M HEX-8 elements)** | | | | | | | | | | | | | |
| Momentum | 246.9 | 277.8 | 290.1 | 228.8 | 161.1 | 118.2 | 0.96 | 1.27 | 1.21 | 1.72 | 2.35 | 0.89 | 2.09 |
| Continuity | 64.4 | 64.4 | 65.1 | 54.4 | 38.5 | 28.9 | 0.99 | 1.20 | 1.18 | 1.67 | 2.23 | 1.00 | 2.23 |
| TKE | 73.9 | 97.7 | 99.2 | 73.9 | 55.6 | 44.9 | 0.98 | 1.34 | 1.32 | 1.76 | 2.17 | 0.76 | 1.64 |
| Mix. Frac. | 72.5 | 96.1 | 90.9 | 66.1 | 48.2 | 38.3 | 1.06 | 1.38 | 1.45 | 1.99 | 2.51 | 0.76 | 1.89 |
| **Coarse $P=2$ mesh (17.5M HEX-27 elements)** | | | | | | | | | | | | | |
| Momentum | – | 967.3 | 937.6 | 697.7 | 470.3 | 392.2 | 1.03 | 1.34 | 1.39 | 2.06 | 2.47 | – | – |
| Continuity | – | 249.4 | 201.1 | 172.0 | 118.9 | 95.5 | 1.24 | 1.17 | 1.45 | 2.10 | 2.61 | – | – |
| TKE | – | 254.5 | 277.1 | 257.9 | 157.6 | 132.7 | 0.92 | 1.07 | 0.99 | 1.61 | 1.92 | – | – |
| Mix. Frac. | – | 251.7 | 261.4 | 243.0 | 147.4 | 123.6 | 0.96 | 1.08 | 1.04 | 1.71 | 2.04 | – | – |

**Table 3:** Matrix assembly timing comparisons on KNL partition of NERSC Cori system. Code configurations: H – Hybrid element/node algorithm; B – Baseline non-consolidated algorithm; C1 – consolidated kernel algorithms; C2 – C1 with SIMD datatypes; C3 – C2 with OpenMP threading; C4 – C3 with `sumInto(...)` and SIMD interleave optimizations.

| | Matrix assembly time (s) | | | | | | Speedup ratio | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Equation | Hybrid | Baseline | Conf. C1 | Conf. C2 | Conf. C3 | Conf. C4 | B:C1 | C1:C2 | B:C2 | B:C3 | B:C4 | H:B | H:C4 |
| **Coarse $P=1$ mesh (17.5M HEX-8 elements)** | | | | | | | | | | | | | |
| Momentum | 329.6 | 398.0 | 473.0 | 352.0 | 250.7 | 203.1 | 0.84 | 1.34 | 1.13 | 1.59 | 1.96 | 0.83 | 1.62 |
| Continuity | 91.7 | 91.7 | 98.1 | 73.3 | 54.2 | 41.1 | 0.94 | 1.34 | 1.25 | 1.69 | 2.23 | 1.00 | 2.23 |
| TKE | 117.6 | 156.5 | 170.7 | 101.5 | 81.4 | 64.6 | 0.92 | 1.68 | 1.54 | 1.92 | 2.42 | 0.75 | 1.82 |
| Mix. Frac. | 115.2 | 154.0 | 152.2 | 87.3 | 67.0 | 53.0 | 1.01 | 1.74 | 1.76 | 2.30 | 2.90 | 0.75 | 2.17 |
| **Fine $P=1$ mesh (140M HEX-8 elements)** | | | | | | | | | | | | | |
| Momentum | 343.0 | 419.4 | 493.0 | 374.9 | 266.2 | 220.9 | 0.85 | 1.32 | 1.12 | 1.58 | 1.90 | 0.82 | 1.55 |
| Continuity | 97.1 | 97.1 | 105.4 | 80.2 | 61.0 | 46.9 | 0.92 | 1.31 | 1.21 | 1.59 | 2.07 | 1.00 | 2.07 |
| TKE | 123.8 | 165.7 | 184.6 | 114.9 | 94.3 | 76.8 | 0.90 | 1.61 | 1.44 | 1.76 | 2.16 | 0.75 | 1.61 |
| Mix. Frac. | 121.6 | 163.3 | 162.6 | 96.7 | 76.4 | 61.5 | 1.00 | 1.68 | 1.69 | 2.14 | 2.66 | 0.74 | 1.98 |
| **Coarse $P=2$ mesh (17.5M HEX-27 elements)** | | | | | | | | | | | | | |
| Momentum | – | 1746.8 | 1642.3 | 894.0 | 456.2 | 379.4 | 1.06 | 1.84 | 1.95 | 3.83 | 4.60 | – | – |
| Continuity | – | 370.8 | 308.5 | 228.9 | 114.6 | 94.8 | 1.20 | 1.35 | 1.62 | 3.23 | 3.91 | – | – |
| TKE | – | 449.6 | 496.8 | 290.1 | 146.9 | 126.1 | 0.90 | 1.71 | 1.55 | 3.06 | 3.57 | – | – |
| Mix. Frac. | – | 446.0 | 461.0 | 276.9 | 140.2 | 119.7 | 0.97 | 1.67 | 1.61 | 3.18 | 3.73 | – | – |

1. The consolidated algorithm design without any NGP enhancements (e.g., SIMD and/or OpenMP threading) tend to be slower than both the hybrid and baseline non-consolidated assembly for the low-order simulations on Cori. The disparity in performance between consolidated and hybrid simulations can be attributed to the use of only element-based algorithms in the consolidated design as compared to nodal source and time contributions terms employed in the baseline simulations. Although the nodal approach involves a separate matrix assemble (along with a separate gather), it avoids a dual nodal volume calculation. However, for the simulation comparison between baseline and consolidated algorithm, the same connectivity graph is used. In fact, the baseline includes extra field data gathers for coordinates and the independent variables. This must be further explored in future milestone efforts. The Haswell performance disparity seems more in line with what the team has seen on local institutional resources while the KNL data seems noteworthy.

2. In contrast to low-order meshes, the high-order $P = 2$ mesh matrix assembly performance timings are on par with the non-consolidated approach. Here the cost of element data gather is amoritized by the extra work performed by computational kernels on each element.

3. The introduction of SIMD vectorization recovers the loss in performance from consolidated design on both machines for all equations except the momentum equation on low-order meshes. However, on high-order mesh the momentum equation assembly times show almost a 2x speedup on KNL. This can be attributed to the wide vector width (= 8) available on KNL cores. At this point, it appears that the cost of interleaving and de-interleaving data for multiple elements is sufficiently amoritized by processing 8 elements per instruction, as compared to only 4 on Haswell.

4. Activating OpenMP threading[5] in conjunction with SIMD vectorization further increases the gains observed on both machines, especially for the high-order $P = 2$ mesh where a factor of 3x speedup is observed on KNL machines.

While the performance gains on low-order meshes have been quite modest, the situation is quite different for the higher-order mesh case, not just in terms of speedup factors but also the viability of a higher-order pathway in the ExaWind project. With the baseline non-consolidated approach, the higher order approach suffered approximately a $4 - 5\times$ runtime penalty on matrix assembly when compared to the fine low-order mesh. In contrast, the introduction of SIMD and OpenMP threading reduce this time difference to a factor of 2 on KNL. Furthermore, the runtimes on KNL are slightly faster with the NGP code compared to Haswell. Of course, further performance studies are necessary to understand the tradeoffs associated with $P > 2$, and the point of diminishing returns with respect to the number of threads per MPI rank. However, the higher-order pathway on next-generation platforms certainly appears encouraging.

### 5.2 Code Timing Comparison Between Master and pre-ExaWind Version 1.1

In this brief section, a high level view of overall application code improvements made to Nalu since the beginning of the ExaWind effort is provided. The performance test case is a 150 million element Vestas V27 (225 kw) wind turbine using a hybrid mesh (hexahedral, tetrahedral, pyramid, and mesh topologies). The case is run on the Sandia institutional cluster, Sky Bridge (Cray-based 2.6 GHz Sandy Bridge with infiniband interconnect). As has been described, the full master element topology transition is required for proper Kokkos-based kernel performance testing. As such, this benchmark between the master code base (09/2017) and version 1.1 (10/2016) is performed using the previously deployed algorithm implementation in Nalu. Therefore, the timings capture various improvements made to the Nalu::TpetraLinearSystem class over the year. Highlights of code changes agnostic to the new kernel design include transitioning to direct Kokkos calls in linear system assembly code and a new `sumInto()` implementation.

The V27 simulation timings, for purposes of this milestone, are restricted to matrix assembly only. However, in general, overall simulation improvements are clear for this case involving parallel searches and the DG/CVFEM non-conformal interface. In general, matrix assembly performance is now nearly 1.8 times faster for momentum and scalar assembly. The recent re-write of the `sumInto()` method alone attributed a 1.2 times speed-up. This data suggests that as we improve the underlying kernel design and Tpetra linear system

[5]The current study used two OpenMP threads per MPI rank.

interfaces, the application code base is continually improving in peformance. Linear solver performance regressed a modest 3-5% over the same development time frame. Investigation into this slight regression is ongoing.

# 6. CONCLUSION

In this milestone effort, significant gains have been made in the underlying element-based matrix assemble procedure. Deployment of Kokkos in addition to a SIMD interface within a sustainable algorithmic kernel-based design has been deployed in the open-source NaluCFD/Nalu master code base. The newly deployed code is tested within an integrated and unit nightly test suite. Matrix assembly speed-up rates have been demonstrated to be 2-6 times faster than the previously deployed Nalu algorithms. The newly deployed algorithmic structure allows for utilization of the threaded Tpetra-based solver stack. The threaded solver stack (GMRES, preconditioned by SGS) has been demonstrated on Cori for limited physics.

The underlying `nalu::Kernel` design has allowed for rapid development of new algorithmic contributions and seems to balance complexity of implementation and raw code performance. The foundational design for kernels will allow for an efficient abstraction layer between the physics to be implemented and the underlying Kokkos-based NGP team-based parallelism strategy. Moreover, unit testing for all new kernels can be easily peformed within the unit test `nalu::unit::Helper` construct. For simulations that require hybrid meshes, future design optimizations may be required to provide balanced work over all algorithms created.

## 6.1 Path Forward

A set of paths to explore in future ExaWind milestone efforts are as follows:

1. The general team consensus is that the consolidated algorithm, without SIMD and threading, should be as fast if not faster than the previous algorithmic design based on the fact that there is shared work between the activated kernels. Formally, the non-consolidated implementation is performing more work as the advection and diffusion and time contribution duplicates the gathering of data, specifically, velocity and coordinates. Moreover, for the case of the momentum assembly, the former algorithmic structure involves a gradient reconstruction algorithm approach for higher-order upwinding. Detailed profiling is required to determine the reason for slow down. Machine variability and possibly Kokkos multidimensional array overhead should be addressed.

2. Thus far, the full SIMD/Threaded Kokkos-based kernels have been deployed to the full set of interior topologies. However, in order to fully gain performance in the master element calculations, e.g., area vector, gradient operator, etc., the former FORTRAN-based implementation must be converted to Kokkos views. This procedure, which was accomplished for the hexahedral topology, allows for efficient SIMD-based evaluations. Low-order three-dimensional topological gaps include the following: tetrahedral, wedge, and pyramid. For the higher-order path forward, which includes polynomial orders greater than two, hexahedral elements will be converted. This conversion will be accompanied by high quality unit testing to verify correctness of the new implementation.

3. For low-order simulations, a hybrid-based assembly may be beneficial for underlying performance. Specifically, time and source terms can be nodally lumped thereby removing the need to apply these contributions at the element looping level. This is, no doubt, a trade-off between additional elemental contributions and adding an additional nodal assembly procedure. Work to understand this trade space in the context of a kernel structure that is nodally-based is required.

4. The team is only at the beginning of understanding and testing solver performance for advection/diffusion systems using the GMRES/SGS Tpetra solver stack. More work to capture solver performance is required.

5. Understanding current gaps in he Tpetra-based solver interface, with specific emphasis on the pressure Poisson solve, will begin.

6. Comparing Nalu v1.1 to the master code base has demomstrated steady and deliberate speed-ups in application-owned operations. In fact, the underlying non-kernel implementations are noting performance speed-up due to the advances in matrix assembly through general `Nalu::TpetraLinearSystem` improvements. However, solver performance (using the same solver settings) is slowly regressing (3-5%). The team will begin routine performance testing of the solver stack at an affordable testing scale to monitor solver performance as the project proceeds.

7. Thus far, only interior matrix contributions have been deployed using the Kokkos-based SIMD kernels. The team will deploy this design structure for flux boundary conditions and, in time, non-matrix assembly procedures including projected nodal gradients, Courant number calculations, and others. However, a performance-based approach will be used to define performance bottlenecks.

8. Although the team is concentrating on low- and higher-order element-based schemes, the team feels that evaluating edge-based schemes in the Kokkos-based SIMD kernel design will be low-cost. As such, deploying a generalized algorithmic kernel to work for nodes, edges, faces and elements is planned.

9. Drive thread-safe graph initialization and insertion.

10. Although a significant set of algorithms have been converted to kernels, the full set of elemental source terms must be transitioned. Temporal, advection/diffusion, stabilization and select source terms have already been converted. Depending on the path forward defined for nodal-based source/time terms, this effort should be small.

## REFERENCES


[1] Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide, 2016. [Online; accessed 20 Sept. 2017].

[2] The Trilinos Project. https://trilinos.org, 2016. [Online; accessed 9 February 2016].

[3] Intel Vtune Amplifier. https://software.intel.com/en-us/vtune-amplifier-help-introduction, 2017. [Online; accessed 20 Sept. 2017].

[4] S. Domino. Github: NaluCFD/Nalu. https://github.com/NaluCFD/Nalu. [Online; accessed 21 Sept. 2017].

[5] S. Domino. A comparison between low-order and higher-order low-Mach discretization approaches. In Parviz Moin and Javier Urzay, editors, *Studying Turbulence Using Numerical Simulation Databases - XV*, pages 387–396. Stanford Center for Turbulence Research, 2014.

[6] Stefan Domino. Sierra low Mach module: Nalu theory manual – version 1.0. Technical Report SAND2015-3107W, Sandia National Laboratories, 2015.

[7] H. C. Edwards, A. Williams, G. D. Sjaardema, D. G. Baur, and W. K Cochran. Sierra toolkit computational mesh model. Technical Report SAND2010-1192, Sandia National Laboratories, 2010.

[8] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[9] G. Schneider and M. Raw. A skewed, positive influence coefficient upwinding procedure for control-volume-based finite-element convection-diffusion computation. *Num. Heat Trans.*, 9:1–26, 1986.

[10] Larry A. Schoof and Victor R. Yarberry. ExodusII: A finite element data model. Technical Report SAND92-2137, Sandia National Laboratories, 1994.