International Journal of
HIGH PERFORMANCE
COMPUTING APPLICATIONS

Research paper

The International Journal of High Performance Computing Applications
1–21
© The Author(s) 2018
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1094342017749957
journals.sagepub.com/home/hpc
SAGE

Toward performance portability of the Albany finite element analysis code using the Kokkos library

Irina Demeshko[1], Jerry Watkins[2], Irina K Tezaur[2], Oksana Guba[3], William F Spotz[3], Andrew G Salinger[3], Roger P Pawlowski[3] and Michael A Heroux[3]

Abstract

Performance portability on heterogeneous high-performance computing (HPC) systems is a major challenge faced today by code developers: parallel code needs to be executed correctly as well as with high performance on machines with different architectures, operating systems, and software libraries. The finite element method (FEM) is a popular and flexible method for discretizing partial differential equations arising in a wide variety of scientific, engineering, and industrial applications that require HPC. This article presents some preliminary results pertaining to our development of a performance portable implementation of the FEM-based Albany code. Performance portability is achieved using the Kokkos library. We present performance results for the Aeras global atmosphere dynamical core module in Albany. Numerical experiments show that our single code implementation gives reasonable performance across three multicore/many-core architectures: NVIDIA General Processing Units (GPU's), Intel Xeon Phis, and multicore CPUs.

Keywords

Performance portability, many-core programming, finite element code, climate simulations, Kokkos library

1. Introduction

Porting large, complex scientific and engineering application codes to multicore/many-core architectures has become a very complicated task due to a variety of programming models, application programming interfaces (APIs), and performance requirements. A major challenge in utilizing heterogeneous resources is the diversity of devices on different machines, which provide widely varying performance characteristics. A program or algorithm optimized for one architecture may not run as well on the next generation of processors or on a device from a different vendor. A program or algorithm optimized for GPU execution is often very different from one optimized for CPU execution. The relative performance of CPUs and GPUs also varies between machines. On one machine, a specific portion of a computation may run best on the CPU, while on another machine it may run best on the GPU. In some cases, it is best to balance the workload between the CPU and the GPU; in other cases, it may be best to execute an algorithm on a device where it runs more slowly but closer to where its output is needed, in order to avoid expensive data transfer operations. It is not uncommon for large application codes to have several different implementations, with each one optimized for a different architecture. This software development approach leads to a code maintainability issue: every new change to the code needs to be implemented in all versions of the code.

Performance portability of a single code base, therefore, has become a critical issue: parallel code needs to execute correctly and remain performant on machines with different architectures, operating systems (OSs), and software libraries. This article presents a new approach for achieving performance portability in a parallel, multiphysics finite element–based code known as Albany (Salinger et al., 2016) using the Kokkos library (Edwards et al., 2014).

1.1. Related work

There are several tools and programming environments that provide options for portability. For example, OpenCL

[1] Los Alamos National Laboratory, Los Alamos, NM, USA
[2] Sandia National Laboratories, Livermore, CA, USA
[3] Sandia National Laboratories, Albuquerque, NM, USA

Corresponding author:
Irina Demeshko, Los Alamos National Laboratory, Los Alamos, NM 87545, USA.
Email: irina@lanl.gov

(Munshi, 2009) offers portability across GPU hardware, OS software, and multicore processors (OpenCL 1.2, OpenMP 4.0); however, it does not offer performance portability (refer to the works of Weber et al. (2010)). Similarly, (OpenMP, 2013) has been a mainstay in hybrid or hierarchical parallelism. It can provide some degree of performance portability, but designing parallelism is a nontrivial effort. OPENERS 2.0 (OpenACC, 2013) is a directive-based programming standard, which can generate OpenCL and CUDA code. Note that OpenCL, OpenMP, and OPENERS have insufficient C++ support. In particular, OpenCL does not allow C++ constructs in the kernels, and OPENERS and OpenMP do not work well in class environments.

Kokkos (Edwards et al., 2014) and RAJA (Hornung and Keasler, 2014) use C++ metaprogramming to provide performance-portable programming models. NVIDIA is taking a similar approach in the HEMI (Harris, 2015) library. OCCA (Medina et al., 2014) is another C++ library that extracts kernels to multiple threading languages (OpenMP, OpenCL, and CUDA). Unlike Kokkos and RAJA, OCCA does not change data layout for different architectures to maximize performance. Tiling abstractions such as Tiling as a Durable Abstraction (Unat et al., 2013) provide a way of incorporating hierarchical parallelism. Another approach for providing performance portability is taken by task-based programming models such as Charm++ (Kale et al., 2008) and Parallex (Kaiser et al., 2009). These models are independent of the number of processors. They also automate resource management and support concurrent composition.

Performance portability of a finite element code has been undertaken by several projects in recent years, including the FEniCS v1.5 (Alnæs et al., 2015) and Firedrake (Rathgeber, Ham, Mitchell et al., 2017) projects. The approach taken in these codes is based on using a domain-specific language (DSL), which enables the application programmer to express his or her mathematical model in a high-level mathematical language and provides the compiler writer with the freedom to make optimal implementation choices. While FEniCS is based on the unified form language (Alnæs et al., 2014) abstractions that work only for the finite element operations, Firedrake presents a simple public API PyOP2 (Rathgeber et al., 2012) that relaxes this limitation. PyOP2 is a Python implementation of the unstructured mesh computation framework OP2 (Mudalige et al., 2012), which applies numerical kernels in parallel over an unstructured mesh. Kernels and parallel loop invocations are "just-in-time" compiled at runtime and cached. Subsequent parallel loops using the same kernel do not incur additional compilation overhead. PyOP2 delivers performance portability across a range of hardware platforms: OpenMP, OpenCL and Message Passing Interface (MPI) on multicore CPUs, and CUDA or OpenCL on General-purpose computing on graphics processing units (GPGPU). Data layout, mesh partitioning, traversal, parallel scheduling, and efficient execution of parallel loops are automatically managed.

HiFlow3 (Heuveline, 2010) is another project that addresses architecture portability for finite element codes. HiFlow3 is a parallel finite element software package that aims at the solution of large-scale problems obtained by means of the finite element method (FEM) for partial differential equations (PDEs). The HiFlow3 project aims at creating a hardware-aware cross-platform portable linear algebra back end.

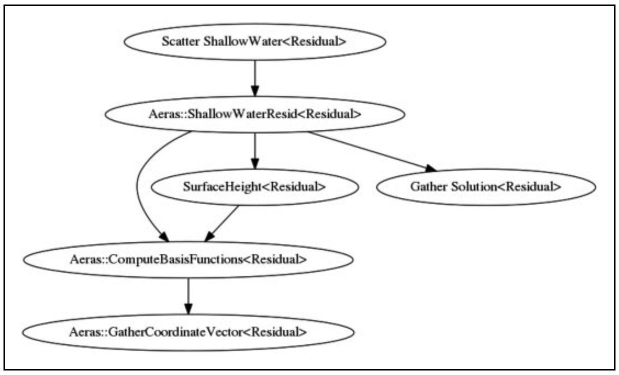

Scatter ShallowWater<Residual>
Aeras::ShallowWaterResid<Residual>
SurfaceHeight<Residual>
Gather Solution<Residual>
Aeras::ComputeBasisFunctions<Residual>
Aeras::GatherCoordinateVector<Residual>

Figure 1. Automatically generated Phalanx graph of the residual evaluator for the Aeras shallow water physics set in Albany.

For the Albany multiphysics finite element code, the subject of the present work, we utilize the Kokkos library for achieving performance portability. Unlike HiFlow3, our focus within the Albany project has been on providing performance portability of the finite element assembly (FEA); performance portability of the linear algebra in Albany will be the subject of future work. Albany and Kokkos are described in more detail below.

1.2. The Albany multiphysics code

Described in more detail in the study by Salinger et al. (2016), Albany is a C++, object-oriented, parallel, unstructured-grid, implicit finite element code for solving PDEs from various fields of science and engineering. The code has a component-based design that makes pervasive use of libraries from the Trilinos project (Heroux et al., 2003) and enables the rapid integration of new physics and capabilities.

At the highest level, the FEM assembly in Albany is based on the Trilinos Phalanx (Pawlowski et al., 2012) package. Phalanx, specifically designed to solve general PDEs, decomposes a complex problem into a number of simpler pieces (*evaluators*) with managed dependencies. Each *evaluator* encodes the variables it depends upon, the variables it evaluates, and the code to actually evaluate the desired term. Phalanx then assembles all of the evaluators for a given problem into a directed acyclic graph. Figure 1 shows a dependency graph automatically created by Phalanx that represents the full PDE residual evaluation for a given set of mesh cells stored in a data structure called the field manager. Albany employs the STK package (Edwards et al., 2010) of Trilinos for mesh database structures and mesh I/O, and the Intrepid2 package (Bochev et al., 2012) of Trilinos for finite element shape function library with general integration kernels. A variety of Trilinos solvers are available for solving the discretized finite element systems assembled within Albany, including Newton-based nonlinear solvers, preconditioned iterative linear solvers, and time integration schemes. Since the focus of this work is performance portability of the FEA in Albany, we do not go into detail about the solvers in Albany.

Albany hosts several science and engineering applications, including the Aeras atmospheric dynamical core (Spotz, William, Smith et al., 2015) used in this study, in addition to the finite elements for land ice experiments (Tezaur et al., 2015) ice sheet solver, the Laboratory for Computational Mechanics (Sun et al., 2013) research code, and the quantum computer aided design (Gao et al., 2013) simulator. Albany is also home to several algorithmic projects that contribute to the code's overall infrastructure and capabilities, for example, embedded uncertainty quantification, adaptive mesh refinement, and model reduction. For each of these projects, it is critical to have one code that is able to obtain highly efficient performance on a variety of architectures, including future architectures.

1.3. Kokkos library and programming model

Kokkos is a library-based programming model, which has been developed at Sandia National Laboratories to provide scientific and engineering codes with user-accessible, many-core, performance portability capabilities. For more information on Kokkos, the reader is referred to the work of Edwards et al. (2014). The key premises on which Kokkos is based are reviewed here.

Performance portability is the primary objective of Kokkos: it is designed to maximize the amount of user code that can be compiled for diverse devices and obtain the same (or nearly the same) performance as a variant of the code that is written specifically for that device. Moreover, Kokkos is expected to provide performance portability not only to currently available advanced architectures but also to *future* generations of advanced architectures. When a new architecture comes out, the Kokkos library will be extended for that architecture by Kokkos developers. Hence, developers of application codes such as Albany that rely on Kokkos are shielded from having to rewrite their code for the new architecture.

There are six primary abstractions in Kokkos:

1. Execution spaces: where to execute,
2. Execution pattern: what kind of operation is performed (parallel_for, parallel_reduce, parallel_scan, and task),
3. Execution policy: how to execute the pattern (Range and Team),
4. Memory spaces: where to store data,
5. Memory layout: mapping logical indices to physical storage, and
6. Memory access traits: how to access the data (atomic and random access).

The Kokkos API has a thin back end implementation layer that maps portable user code to a number of device-specific programming models: CUDA (NVIDIA), OpenMP (OpenMP), and Pthreads (Lewis and Berg, 1998). Kokkos separates the *Memory Space* (where data reside) from the *execution space* (where functions execute, such as NVIDIA GPUs, Intel Xeon Phis, etc.). The integration of these abstractions enables user code to satisfy multiple architecture-specific memory access pattern performance constraints, all without requiring modifications to the application source code.

We describe three of the primary abstractions in Kokkos in some detail below: the Kokkos multidimensional array, the Kokkos parallel pattern, and the Kokkos execution policy.

1.3.1. Kokkos multidimensional array. The data access pattern of a data parallel computational kernel can have a significant impact on its performance: different architectures require different memory access patterns for optimal performance. For example, computations on an NVIDIA GPU must use a coalesced global memory access pattern, while the best performance is achieved on an Intel Xeon Phi if the memory access is continuous. The main advantage of Kokkos is that the Kokkos multidimensional array (Kokkos:: View) has a polymorphic data layout that can be changed to have an optimal access pattern on a specific *execution space*. By default, Kokkos chooses column major layout for GPUs and row major layout for CPUs.

For more details on the Kokkos multidimensional array, please refer to the work of Edwards et al. (2014).

1.3.2. Kokkos parallel pattern. In data parallel computations, multidimensional arrays are partitioned among the threads of a many-core device, and each thread applies one or more computational kernels to its designated subset of these arrays. Kokkos currently implements data parallel execution with parallel_for, parallel_reduce, and parallel_scan operations.

A parallel_for is trivially parallel in that the computational kernel's work is fully disjoint. In a parallel_reduce, each application of the computational kernel generates data that must be reduced among all work items. A parallel_scan is for taking a view and creating a running sum of values to replace the values of the view. Expressing an application in terms of these patterns allows the underlying implementation or compiler to make reasonable choices about valid transformations. The composition of parallel work dispatch and polymorphic array layout capabilities also enables performance portable implementations of parallel algorithms.

A data parallel computational kernel is currently implemented as a functor or C++ lambda. A functor is a C++ class that encapsulates one or more callback functions, shared parameters, and references to data upon which the callback function operates. The C++11 standard lambda feature significantly improves the syntax and usability of the functor pattern, but is not used in the current work since support for lambdas is not available on some platforms we target.

The Kokkos parallel pattern syntax includes the *Execution Policy* and the user function as input data:
parallel_pattern(Policy<Space>,
UserFunction)

An *Execution Policy*, together with an *Execution Pattern*, determines how a function is executed. An example of the Kokkos parallel_for pattern is shown in Section 2.

1.3.3. Kokkos execution policies. The simplest form of execution policies in Kokkos is Range policies. They are used to execute an operation once for each element in a range. There are no prescriptions of order of execution or concurrency, which means that it is not legal to synchronize different iterations.

Team policies are used to implement hierarchical parallelism. For that purpose, Kokkos groups threads into *teams*. A *thread team* is a collection of one or more parallel "threads" of execution. All threads in a team are guaranteed to run concurrently.

Multidimensional Range policies (MDRange) are implemented for parallel execution over tightly nested loops. With MDRange policies, Kokkos fuses the loops together to increase the number of parallel threads. This policy is recommended for nested loops which are highly parallelizable among multiple dimensions or for nested loops which are unable to fully saturate available resources within a single dimension.

1.4. Main contributions

The main goal of this work is to create an architecture-portable version of Albany's FEA using the Kokkos library. The preliminary results presented herein demonstrate the feasibility of developing a performance portable code by using a single programming model (not a DSL). The specific contributions of this article are as follows:

- the development of an architecture-aware FEM code with performance portable FEA that runs on different architectures by merely switching the Kokkos execution space configure time parameter,
- a demonstration that the Kokkos Albany code executes on a variety of different architectures including NVIDIA P100 GPUs and Intel Knights Landing (KNL) Xeon Phis,
- a demonstration of speedups over traditional MPI-only simulations without implicit data management or architecture-dependent code optimizations for the Aeras atmospheric dynamical core in Albany[1], and
- an overview of the performance of the MPI + GPU implementation for Aeras and a demonstration of improved scaling through the use of high-order discretizations.

Although this article focuses on the specific case of the Albany code, we emphasize that the refactoring approach we describe is general: it can be employed to create performance-portable implementations of other scientific application codes.

2. Albany-to-Kokkos refactoring

The basic algorithm of refactoring a C++ code to use Kokkos is as follows:

Step 1: Replace data array allocations with Kokkos multidimensional arrays (Kokkos:: View).
Step 2: Replace for loops with Kokkos:: parallel_for and the contents of the for loops with functors (or C++ lambdas).

An optional *step 3* is to optimize the code to attain the best possible performance on the architectures of interest, for example, optimize memory transfers between the host and the device and optimize algorithms for threading.

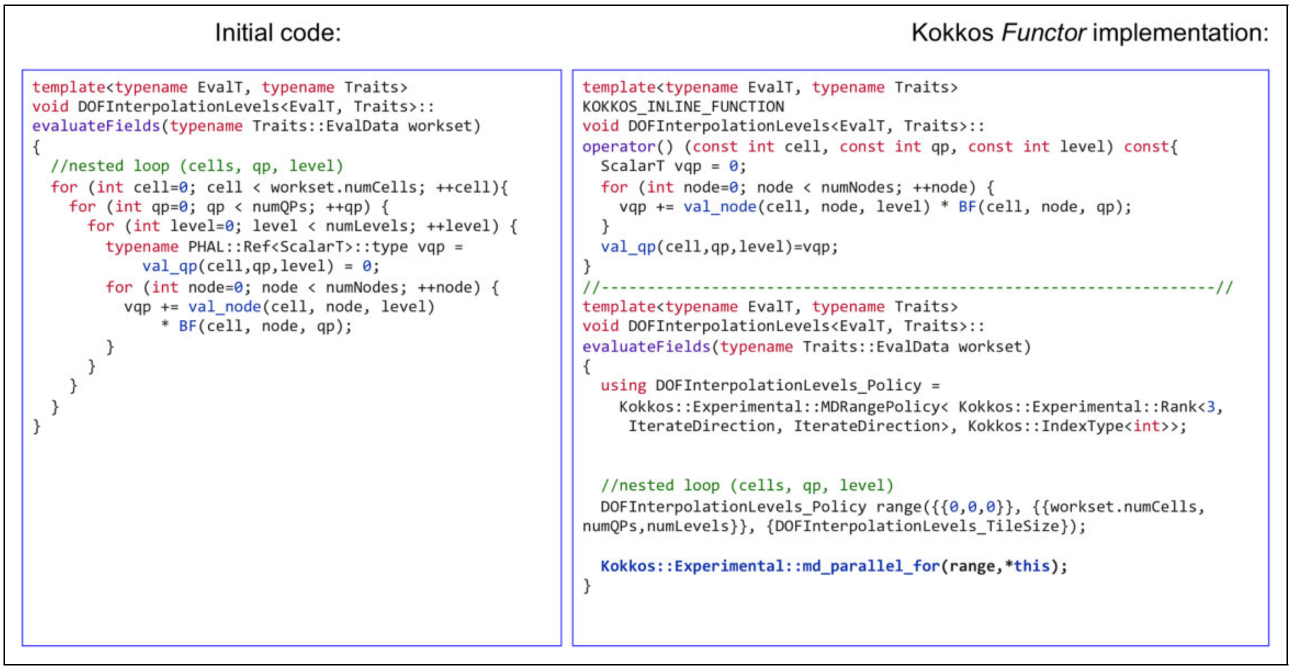
We describe each of the two primary refactor steps in some detail below. As stated earlier, although our presentation is in the context of the Albany framework, the approach is general and can be used to create performance-portable versions of other finite element codes.

2.1. Step 1: Replacing data arrays with Kokkos:: View

The first step in refactoring a C++ code to use Kokkos is to replace temporary array allocations within the code with Kokkos multidimensional arrays (Kokkos:: View). These specialized views provide optimized memory access inside the Kokkos kernels. Moreover, they provide automatic access to unified virtual memory (UVM) for the CUDA execution space, which became available in CUDA 6.0. The mechanism for using UVM in Kokkos is to set the default memory space to CudaUVM. The idea behind UVM is to use page fault memory in virtual memory systems to detect when a piece of memory is being accessed on the GPU, and move the page to the device. The page is then moved back to the CPU when the CPU accesses it. The advantage of UVM is that it abstracts memory management away from the programmer: with unified memory, programmers can access any resource or address within the legal address space, regardless of which pool the address actually resides in, and operate on its contents without first explicitly copying the memory over. Although UVM implicit memory copies do have a performance penalty, UVM enables the porting of codes such as Albany to multicore architectures with minimal effort. Implicit UVM data management handled under the hood by Kokkos:: View can be replaced with manual data management at a later point in time, once some initial level of performance portability is attained.

In a generic FEM code, *step 1* will require at a minimum the replacement of std containers with analogous containers from the Kokkos programming model (e.g. std:: vector replaced with Kokkos:: View). In the specific case of Albany, in addition to making this change, we took advantage of some recent Kokkos-based refactors of several Trilinos packages, most notably Intrepid2 and Phalanx.

By relying on Phalanx to manage not only dependencies between *Albany evaluators* but also some memory allocation and memory access through the usage of PHX:: MDField, the Albany code inherited automatically the benefits of Phalanx in which every Kokkos:: View was wrapped in a PHX:: MDField.



```
template<typename EvalT, typename Traits>
void DOFInterpolationLevels<EvalT, Traits>::
evaluateFields(typename Traits::EvalData workset)
{
  //nested loop (cells, qp, level)
  for (int cell=0; cell < workset.numCells; ++cell){
    for (int qp=0; qp < numQPs; ++qp) {
      for (int level=0; level < numLevels; ++level) {
        typename PHAL::Ref<ScalarT>::type vqp =
            val_qp(cell,qp,level) = 0;
        for (int node=0; node < numNodes; ++node) {
          vqp += val_node(cell, node, level)
              * BF(cell, node, qp);
        }
      }
    }
  }
}
```

```
template<typename EvalT, typename Traits>
KOKKOS_INLINE_FUNCTION
void DOFInterpolationLevels<EvalT, Traits>::
operator() (const int cell, const int qp, const int level) const{
  ScalarT vqp = 0;
  for (int node=0; node < numNodes; ++node) {
    vqp += val_node(cell, node, level) * BF(cell, node, qp);
  }
  val_qp(cell,qp,level)=vqp;
}
//---------------------------------------------------------------//
template<typename EvalT, typename Traits>
void DOFInterpolationLevels<EvalT, Traits>::
evaluateFields(typename Traits::EvalData workset)
{
  using DOFInterpolationLevels_Policy =
    Kokkos::Experimental::MDRangePolicy< Kokkos::Experimental::Rank<3,
     IterateDirection, IterateDirection>, Kokkos::IndexType<int>>;


  //nested loop (cells, qp, level)
  DOFInterpolationLevels_Policy range({{0,0,0}}, {{workset.numCells,
numQPs,numLevels}}, {DOFInterpolationLevels_TileSize});

  Kokkos::Experimental::md_parallel_for(range,*this);
}
```

Figure 2. Example illustrating *Albany-Evaluator-to-Kokkos-Functor* refactoring. This includes (1) replacing the outer loop with a md_parallel_for and (2) moving the inner kernel to an operator() function. The template parameters appearing in this figure are discussed in Section 2.

2.2. Step 2: Replacing for loops with Kokkos:: parallel_for

After replacing each array allocation with a Kokkos:: View, the next step in the refactoring process is to replace the existing high computation for loops with Kokkos:: parallel_for and functors.

In the context of an FEM code, a significant amount of speedup can be achieved by parallelizing each operation over cells. Existing structs and classes can also be used as functors as long as tags are used to differentiate between multiple operator() functions as described by Edwards et al. (2014).

In the case of Albany, evaluators are used as functors to launch Kokkos kernels. *Albany-Evaluator-to-Kokkos-Functor* refactoring includes

- replacing the outer loops with a call to parallel_for or md_parallel_for and
- moving the inner kernel to the C++ method operator().

An example of this implementation for an interpolation operation in Aeras using an MDRange Policy is presented in Figure 2. Most of the Albany evaluators had a nested loop structure similar to the one shown in this figure and were hence straightforward to port to Kokkos. Several Albany evaluators having a more complicated code structure had to be refactored to remove data dependencies inside nested loops.

To transform an Albany evaluator to Kokkos functor, three steps are required: (1) the addition of a C++ operator() method to the body of the evaluator class, (2) the movement of internal calculations comprising the original nested loop inside the operator(), and (3) the addition of a call to md_parallel_for at the original loop location (evaluateField method). Albany divides the total number of elements into a *workset* of elements and then executes each workset in a cycle. For our performance evaluation, we use one workset and therefore workset.numCells corresponds to the total number of cells per MPI rank. We also tell the Kokkos MDRange Policy to fuse the numCells, numQPs, and numLevels loops together to exploit as much parallelism as possible (see Figure 2). Albany evaluators are templated on the ScalarT type. For residual computations, this template parameter is a simplet double, whereas for Jacobian computations, it is a Sacado Automatic Differentiation (Heroux et al., 2003; Phipps and Pawlowski, 2012) type. The Sacado Automatic Differentiation library has already been ported to Kokkos. Note that when C++11 lambda functionality is sufficiently supported on all architectures targeted by Albany, this strategy can be superseded by a syntactically simpler lambda strategy.

The first three steps of the FEA in Albany are as follows: (1) gathering data to local data structures, (2) computing intermediate field variables, and (3) performing element-level numerical integrations, for example, evaluating local stiffness matrices and body force terms. MDRange policies are most useful for tightly nested for loops or simple operations which are embarrassingly parallel, for example, computing intermediate field variables. In the case of interpolation or numerical integration, it is recommended to utilize Team policies to exploit multiple levels of shared-memory parallelism. We emphasize that this is not necessary to achieve reasonable performance. For Aeras, we chose to use Kokkos MDRange policies to increase the computational intensity of our kernels when increasing the levels and the order of the basis polynomials. Team policies for Albany will be explored in more detail in future projects.

The last step is the assembly of local data into the global matrix or vector. Albany utilizes the Tpetra Trilinos library for matrix and vector data structures. Since this package has already been ported to Kokkos, it is not necessary to write loops to fill global matrices and vectors on the Albany side.

The matrix/vector fill operation is not thread safe since a single equation gets contributions from multiple elements. In order to avoid thread collision in the assembly step, Kokkos atomic operations have been used: with Kokkos::atomic_fetch_add, accesses to the same datum are serialized. Our performance study shows that using atomic operations slows down the assembly kernels by not more than 5% of the kernel execution time.

3. Aeras global atmosphere model

The performance of our Kokkos refactor of Albany will be demonstrated in the context of the Aeras global atmosphere model, which is implemented within the Albany code base. To keep this article self-contained, we briefly describe this model and its discretization in the present section.

The Aeras atmosphere solver (Spotz et al., 2015) in Albany is a dynamical core that solves the mass, momentum, and thermodynamic equations of the atmosphere, as well as equations for the advection of tracers. Aeras is a collection of models, described in more detail below, of different complexities intended to isolate and study particular aspects of discretizations in atmospheric modeling. The two-dimensional (2-D) horizontal model is referred to as the *shallow water* model and the 2-D vertical model is referred to as the *X–Z hydrostatic* model. These two models can be combined together to form the *three-dimensional (3-D) hydrostatic model*, which serves as an atmospheric dynamical core in global circulation models. Here, we consider two of these three models: the 2-D shallow water model and the 3-D hydrostatic model.

3.1. The spherical shallow water equations

Here, we consider the shallow water equations stabilized using hyperviscosity

$$\begin{aligned}\frac{\partial \mathbf{u}}{\partial t} &= -(\zeta + f)\hat{\mathbf{k}} \times \mathbf{u} - \nabla\left(\frac{1}{2}\mathbf{u}^2 + gH\right) - \tau\nabla^4\mathbf{u} \\ \frac{\partial h}{\partial t} &= -\nabla\cdot(h\mathbf{u}) - \tau\nabla^4 h\end{aligned} \tag{1}$$

In equation (1), $\mathbf{u} = (u, v)$ is the horizontal velocity vector, $\hat{\mathbf{k}}$ is the outward radial unit vector, $\zeta = \hat{\mathbf{k}}\cdot\nabla\times\mathbf{u}$ is vorticity, f is a Coriolis term, g is gravity, and $H = h + h_s$, with H denoting the fluid-surface height, h denoting the fluid thickness, and h_s denoting the bottom surface elevation. The ∇ operator in equation (1) denotes a 2-D, horizontal operator, excluding the radial component. The parameter $\tau > 0$ denotes the hyperviscosity coefficient and ∇^4 denotes the biharmonic operator. Hyperviscosity is an effective technique for stabilizing the shallow water and hydrostatic equations (Guba et al., 2014). The hyperviscosity parameter has the form $\tau = c_0(\Delta x)^s$, where Δx is the grid spacing, $c_0 > 0$ is a constant, and s is a scaling parameter.

3.2. The 3-D hydrostatic equations

The 3-D hydrostatic model is a common set of governing equations for the atmosphere in climate models. Here we adopt the hydrostatic model proposed by Taylor (2012). To this end, as for the shallow water equations, the ∇ operator will denote a 2-D, horizontal operator. Vertical derivatives will be denoted by $\partial/\partial\eta$, where η denotes the vertical coordinate system,[2] and $\dot{\eta}$ is defined in equation (6). Similarly, $\mathbf{u} = (u, v)$ denotes the 2-D horizontal velocity.

The prognostic momentum, continuity, and thermodynamic equations comprising the 3-D hydrostatic model are as follows

$$\begin{aligned}&\frac{\partial \mathbf{u}}{\partial t} + (\zeta + f)\hat{\mathbf{k}}\times\mathbf{u} + \nabla\left(\frac{1}{2}\mathbf{u}^2 + \phi\right) + \dot{\eta}\frac{\partial \mathbf{u}}{\partial \eta} + \frac{RT_v}{p}\nabla p \\ &\quad + \tau\nabla^4\mathbf{u} = 0\end{aligned} \tag{2}$$

$$\frac{\partial}{\partial t}\left(\frac{\partial p}{\partial \eta}\right) + \nabla\cdot\left(\mathbf{u}\frac{\partial p}{\partial \eta}\right) + \frac{\partial}{\partial \eta}\left(\dot{\eta}\frac{\partial p}{\partial \eta}\right) + \tau\nabla^4\left(\frac{\partial p}{\partial \eta}\right) = 0 \tag{3}$$

$$\frac{\partial T}{\partial t} + \mathbf{u}\cdot\nabla T + \dot{\eta}\frac{\partial T}{\partial \eta} - \frac{RT_v}{c_p p}\omega + \tau\nabla^4 T = 0 \tag{4}$$

Here, $\zeta = \nabla\times\mathbf{u}$ is the vorticity, f is the Coriolis parameter, and $\hat{\mathbf{k}}$ is the unit vector in the vertical direction. T and p denote the temperature and pressure, respectively. $\partial p/\partial\eta$ serves as a density-like quantity and $\omega = \frac{Dp}{Dt} = \frac{\partial p}{\partial t} + \mathbf{u}\cdot\nabla p + \dot{\eta}\frac{\partial p}{\partial \eta}$.

Three quantities are obtained from diagnostic equations, starting with ϕ, the geopotential, given by the following

$$\phi = \phi_s + \int_{\eta}^{\eta_s}\frac{RT_v}{p}\frac{\partial p}{\partial \eta}d\eta' \tag{5}$$

where ϕ_s is the surface geopotential, η_s is the value of the η-coordinate at the surface, and R is the universal gas constant. Quantity $\dot{\eta}$ is obtained from the following

$$\dot{\eta}\frac{\partial p}{\partial \eta} = -\frac{\partial p}{\partial t} - \int_0^{\eta}\nabla\cdot\left(\frac{\partial p}{\partial \eta'}\mathbf{u}\right)d\eta' \tag{6}$$

and the virtual temperature T_v is given by

$$RT_v = (c_p - qc_v)T \tag{7}$$

where c_p and c_v are the specific heats at constant pressure and volume, respectively, and q is specific humidity. As with the shallow water equations, equations (2) to (4) are stabilized via hyperviscosity of the form $\tau\nabla^4$.

In most atmosphere dynamical cores, the dynamical equations (2) to (4) are coupled to a set of so-called physics equations for various tracers that are advected by the atmosphere, for example, moisture, cloud physics, and so on. The Aeras module in Albany has the capability to specify tracers, but we omit discussion of the tracer equations here, as the test case considered contains no physics. A more detailed discussion of the 3-D hydrostatic equations with physics can be found in the study by Spotz et al. (2015).

3.3. Albany implementation of Aeras

It is known (Evans et al., 2010) that low-order (linear and quadratic) finite elements based on Lagrange basis functions are suboptimal for the atmospheric model equations presented above. In contrast, the spectral element method (SEM) has proven to be accurate and efficient for atmospheric modeling, not to mention a variety of other geophysical applications (Baer et al., 2006; Taylor et al., 2007). This method employs higher order basis functions and Gauss–Lobatto–Legendre node points that also serve as quadrature points within each element to evaluate the integrals arising in the variational formulation of the governing PDEs. A nice feature of the SEM is it leads to a diagonal mass matrix, which is trivial to invert. When the SEM is combined with explicit time stepping, it is possible to obtain spectral accuracy while retaining both parallel efficiency and the geometric flexibility of unstructured meshes (Dennis et al., 2012; Taylor et al., 2007).

The Albany code was originally designed to support low-order finite elements with linear (bilinear in 2-D and trilinear in 3-D) and quadratic (biquadratic in 2-D and triquadratic in 3-D) Lagrange basis functions, and was extended to allow support for spectral elements of arbitrary orders for the Aeras project. In this implementation, a linear mesh is read in using the Trilinos package STK (Edwards et al., 2010) and enriched with additional nodes. A Gauss–Lobatto quadrature rule is used in place of standard Gauss quadrature, the default numerical integration rule in Albany. Both quadrature rules are available through the Trilinos package Intrepid2 (Bochev et al., 2012). As Albany was originally an implicit solver, it was necessary to modify some routines to take advantage of the diagonal mass matrix obtained with spectral elements and Gauss–Lobatto quadrature.

The shallow water equations and the 3-D hydrostatic equations are solved on the surface of the sphere, discretized with quadrilaterals. Formally, these elements are shells, that is, elements that can be parameterized with two dimensions and lie on a 2-D manifold existing within a 3-D space. Another modification to Albany required for the Aeras project was the addition of support for shell elements. For the numerical examples given here, a cubed-sphere grid is used to discretize the horizontal directions on the sphere. Figure 3 shows an example of this type of mesh which is referred to as “ne30” because it has 30 elements along each edge of the cube. Throughout this article, we also use $p\,x$ as an abbreviation for the element order where x is the degree of the polynomial basis.

For the 3-D hydrostatic model, it is necessary to create a discretization of the vertical coordinate η. Following the works of Taylor (2012), we discretize the vertical transport operators using central-difference approximations to the differential operators (Figure 4). Vertical transport using finite differences is supported by column data structures, which were added to Albany for the Aeras project. At a given surface element node, the entire vertical column is referenced to that particular node. Since the entire column for each surface node is contained on processor, finite differencing is straightforward and requires no MPI data communication. Column data structures were implemented by redefining element level “gather” and “scatter” routines. Gather routines collect all data necessary to compute element integration kernels from a global vector and provide it in an element centric pattern. Scatter routines distribute the resulting data into a global residual vector. With these specialized gather and scatter routines, Albany evaluators are used to discretize the PDEs comprising the 3-D hydrostatic model. Integrals in the vertical direction are conveniently cast as summations on columns.

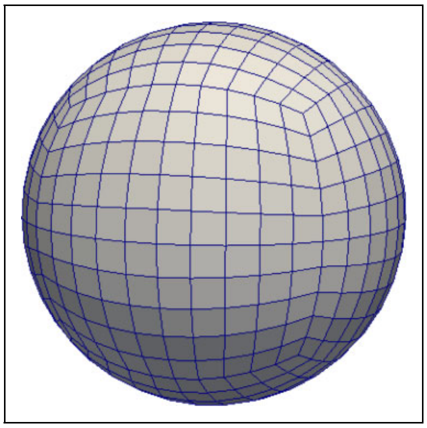


Figure 3. An example of a cubed-sphere grid for Aeras shallow water and 3-D hydrostatic simulations. This is the ne10 grid where “ne10” refers to the number of elements along each edge of the cube. 3-D: three-dimensional.

For time-stepping, Albany employs the Trilinos package Rythmos. Rythmos is a transient integrator for ordinary differential equations and differential algebraic equations (Ober et al., 2013). It has support for explicit, implicit, one-step, and multistep algorithms. For Aeras, we utilize the explicit Runge–Kutta schemes available through Rythmos to take advantage of the diagonal structure of the mass matrix.

Due to the diagonal mass matrix, Aeras does not require any linear solves and performance portability of the FEA is sufficient to achieve almost full portability of this module in Albany. Indeed, the FEA comprises almost 99% of the total CPU time for a typical Aeras problem.

4. Evaluation results

4.1. The evaluation environment

We evaluated the performance of Aeras on two different computer architectures located at Sandia National Laboratories, known as Ride and Bowman. A summary of the

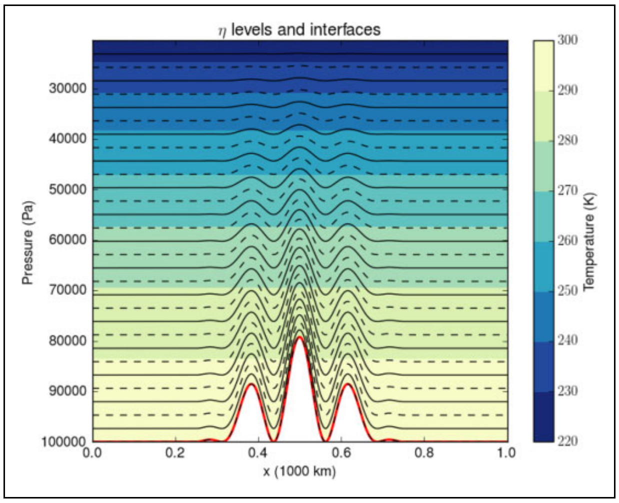


Figure 4. Illustration of the hybrid vertical coordinates used by Aeras. Here 15 model levels (solid black lines) and 16 interfaces (dashed lines) define the vertical coordinate system.

Table 1. Evaluation environments.

Name	Ride (Garrison partition)	Bowman (knl-delta partition)
Nodes	12	10
CPU	IBM dual Power8 (8-core)	Intel Xeon Phi 7250 (KNL)
Coprocessor	NVIDIA Tesla quad P100	None
Memory/node	512 GB	96-GB DDR4; 16-GB MC-DRAM
Interconnect	Mellanox ConnectX-4 IB	Intel Omni-Path HFI 100
OS	RedHat 7.3	RedHat 7.1
Compiler	gcc 5.4.0	Intel 17.0.098
MPI	openmpi 1.10.4	openmpi 1.10.4
Nvcc compiler version	8.0.44	None

OS: operating system; KNL: Knights Landing; MPI: message passing interface.

hardware, software, and compiler information for the clusters and Aeras build configurations can be found in Table 1.

The Garrison partition on the IBM Power8 GPU cluster called Ride is equipped with 12 nodes, each with an IBM dual Power8 CPU and an NVIDIA Tesla quad P100 GPU. Each node has two sockets with eight cores and two GPUs connected to each socket. Each physical core is also split into eight hardware threads.

The knl-delta partition on the Intel KNL cluster called Bowman has 10 nodes, with each node containing a single KNL. Each KNL is configured to quadrant mode where the chip is divided into four virtual quadrants but is a single NUMA domain. The MCDRAM memory is configured to cache mode where it serves as a last-level cache. The KNL contains 68 physical cores each with 4 hardware threads.

Albany/Aeras was compiled on both clusters considered using three Kokkos execution spaces: Kokkos:: Serial, Kokkos:: OpenMP, and Kokkos:: Cuda. All three spaces utilize MPI to distribute work among CPU cores. We note that the Albany MPI implementation does not use any MPI-shared memory support. For the CUDA implementation, CUDA UVM is used to avoid manual memory transfers and CUDA-Aware MPI or direct GPU to GPU communication has yet to be implemented, forcing expensive memory transfers between host and device. Kokkos:: Serial is used to run cases where only MPI is used while Kokkos:: OpenMP and Kokkos:: Cuda are used to run cases where OpenMP threads or GPUs are used for fine grain parallelism, respectively. In our studies, MPI ranks are mapped to cores and OpenMP threads are mapped to hardware threads. We found these to be the optimal configurations when utilizing MPI and OpenMP for these particular problems in Aeras. Mapping MPI ranks to hardware threads reduces performance due to scheduling issues, as multiple ranks compete for the resources on a single core. Mapping OpenMP threads to cores reduces performance since those cores could have been utilized for coarser grain parallelism (i.e. MPI). For MPI + GPU, each rank is assigned a single core and GPU; therefore, there is only a maximum of four cores being utilized per node.

We use a variety of different configurations when utilizing the different types of architectures so it is important to define a general convention to summarize how many MPI ranks, OpenMP threads, and GPUs are being used. In this article, r(MPI + $xARCH$) is used, where r is the number of MPI ranks and x is the number of OpenMP threads or GPUs. $ARCH$ is the architecture used for the additional level of parallelism (e.g. OMP for OpenMP).

For details on the versions of Trilinos, Kokkos, Albany, and the required third-party libraries (TPLs) employed in the numerical studies presented in this article, the reader is referred to Section 6.

4.2. Test cases

We evaluate our Kokkos implementation of Aeras on some common test cases for the shallow water equations (1) and 3-D hydrostatic equations (2) to (4), described succinctly here. For a detailed discussion of the verification of the Aeras dynamical core on these test cases, the reader is referred to the study by Spotz et al. (2015).

4.2.1. Test Case 5 for the shallow water equations. The first test case considered is for the shallow water equations and involves a zonal flow over an isolated mountain. This test case is referred to herein as “Test Case 5” or simply “TC5” (Williamson et al., 1992). To give the reader an idea of the character of the test case solution, we include Figure 5, which depicts the longitudinal velocity at several times.

4.2.2. Baroclinic instability test case for 3-D hydrostatic equations. For the 3-D hydrostatic equations, we consider a well-known 3-D dynamical core test case for the sphere: the perturbed baroclinic instability test, originally from the works of Jablonowski and Williamson (2006) and included as test case 4-1 in the study by Ullrich et al. (2012). The initial conditions correspond to a laminar polar front jet with a small perturbation superimposed to trigger the wave's development. The wave begins to develop in earnest after day 4 and breaks into a closed midlatitude cyclone by approximately day 9. As for the shallow water test case, we include a figure depicting the test case solution to give the reader an idea of the nature of the baroclinic instability problem. Figure 6 shows the developing baroclinic wave at day 9, as computed by Aeras, on the ne16 grid ($\approx 5^\circ$ horizontal resolution) with 15 vertical levels (top row) and 30 vertical levels (second row). Results from the ne30 grid with 15 levels and 30 levels are shown in rows three and four, respectively. The first column shows the surface pressure and the second column shows the temperature along the model level closest to the 850-hPa level (level 12 in the 15 level simulations and level 25 in the 30-level simulations).

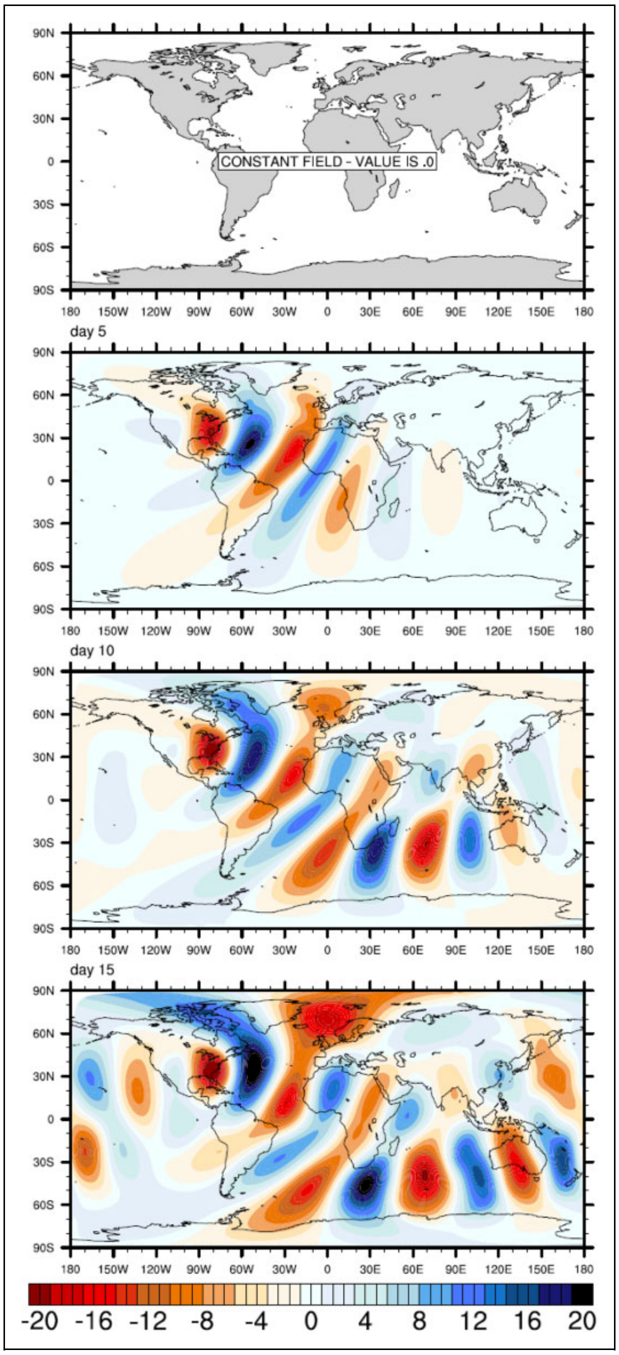
4.3. Aeras performance results

In this section, the computational performance of Aeras is summarized using three Kokkos execution spaces: Kokkos:: Serial, Kokkos:: OpenMP, and Kokkos:: Cuda. It is emphasized that the same finite element code base was used for each case. All simulations were run on the Ride and Bowman clusters, with the configurations described in Section 4.1. The shallow water TC5 and the 3-D hydrostatic baroclinic instability test cases described in the section entitled *Aeras verification* were used for the performance studies with a few modifications described below.

Five uniform cubed-sphere meshes (Figure 3) of varying resolutions were considered. The properties of these meshes are summarized in Table 2. High-order quadrilateral shell elements of degree 3 through 6 were employed on each mesh for the shallow water test case while only a shell element of degree 3 with 10 levels was used for the 3-D hydrostatic test case. For the 3-D hydrostatic case, we found that the GPU would run out of memory for cases with higher order elements.

An explicit four-stage, third-order Runge–Kutta time-stepping scheme is used for all simulations. We focus our performance analysis on the computation of the residual since the residual computation takes the majority of the time within the simulation. The solution is advanced for 100 time steps in order to average the performance over 400 residual computations. For the shallow water test case, hyperviscosity is turned off and the time step is fixed to $\Delta t = 1 \times 10^{-5}$ s for all cases to avoid instabilities. The hyperviscosity coefficients and time steps used for the 3-D hydrostatic simulations and are known to be stable are shown in Table 3.

The performance results are generated by creating two timers over two sections of the code. First, the total computation of the global residual is considered which includes (1) a gathering of global information across MPI ranks, (2) a gathering of information from a global structure to an element local structure, (3) the computation of a residual in each element, (4) a scattering of information from a local structure to a global structure, and (5) a scattering of global information across MPI ranks. For the remainder of this article, this timer will be referred to as "GlobalResidual." A second timer is used to evaluate the computational performance of the Kokkos implementations and excludes the copies and communication used during MPI gather/scatter operations. This timer is referred to as "evaluateFields." The wall-clock times given are generated by taking an average over the 400 calls to each timer.



90N
60N
30N
0
30S
60S
90S
CONSTANT FIELD - VALUE IS .0
180 150W 120W 90W 60W 30W 0 30E 60E 90E 120E 150E 180
day 5
day 10
day 15
-20 -16 -12 -8 -4 0 4 8 12 16 20

Figure 5. Longitudinal velocities in meters per second for Aeras shallow water TC5 at the following times, from top to bottom: Initial time, day 5, day 10, and day 15. TC5: Test Case 5.

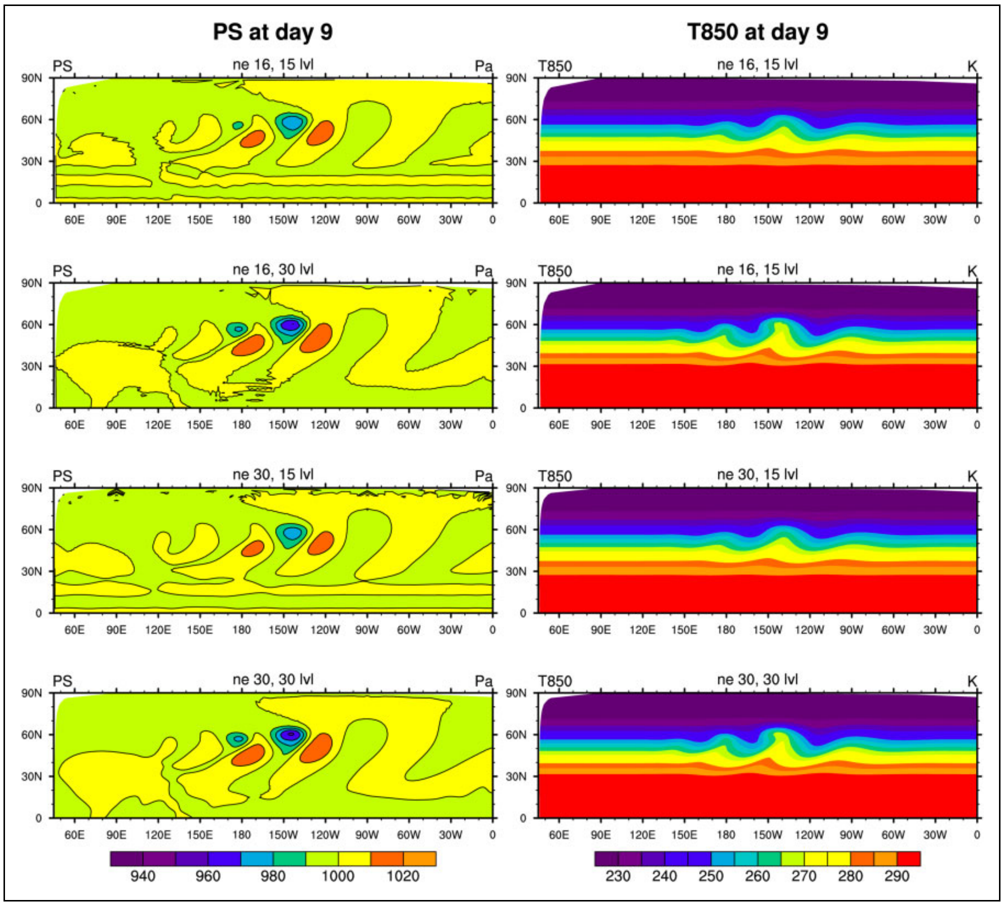


Figure 6. Perturbed baroclinic instability. Aeras solutions for surface pressure (1st column) and temperature at ≈ 850 hPa (2nd column) on the ne16 grid (rows 1 and 2) and the ne30 grid (rows 3 and 4). Computations that used 15 vertical levels are shown in rows 1 and 3; computations using 30 vertical levels are shown in rows 2 and 4.

Table 2. Cubed-sphere mesh resolutions considered for Aeras performance results.

Name	Δt	τ
ne30	1.0°	5400
ne60	0.5°	21,600
ne120	0.25°	86,400
ne240	0.125°	345,600
ne480	0.062°5	1,382,400

Table 3. Fixed time steps Δt and hyperviscosity coefficients τ used for 3-D hydrostatic performance results.

Name	Δt	τ
ne30	30	5.00×10^{15}
ne60	10	1.09×10^{14}
ne120	5	1.18×10^{13}

3-D: three-dimensional.

4.3.1. Architecture comparison. The performance of the OpenMP and CUDA execution spaces are evaluated by comparing 16(MPI + 8OMP) and 4(MPI + GPU) to 16MPI simulations on a single node on Ride and 68(MPI + 4OMP) to 68MPI on a single KNL on Bowman. As explained in Section 4.1, these were the optimal configurations for MPI, MPI + OpenMP, and MPI + GPU for Aeras. Figure 7 summarizes the most notable trends by showing wall-clock times for the 3-D hydrostatic case on the ne30 mesh. A more comprehensive table is given in the appendix. The results show that a speedup is achieved by the OpenMP and CUDA Kokkos execution spaces over the Serial execution space even when including the communication costs in GlobalResidual. The OpenMP implementation is able to achieve a slight improvement over the Serial implementation on the same hardware, while the speedup from the CUDA implementation is primarily from the four GPUs on the node. In this case, the wall-clock time is dominated by communication costs from host to device. One would expect better performance if all computations in the code were performed on the GPU, but all operations in the software stack do not yet utilize Kokkos. It is also important to note that the CUDA implementation is only utilizing 4 CPU cores out of the 16 CPU cores on the node so 12 CPU cores are not being utilized. It may be possible to assign multiple MPI ranks to a single GPU to utilize all CPU cores, but this was not tested.

Table 1A also shows that the results for the shallow water case are very similar to those for the 3-D hydrostatic test case. There is a slight increase in speedup when

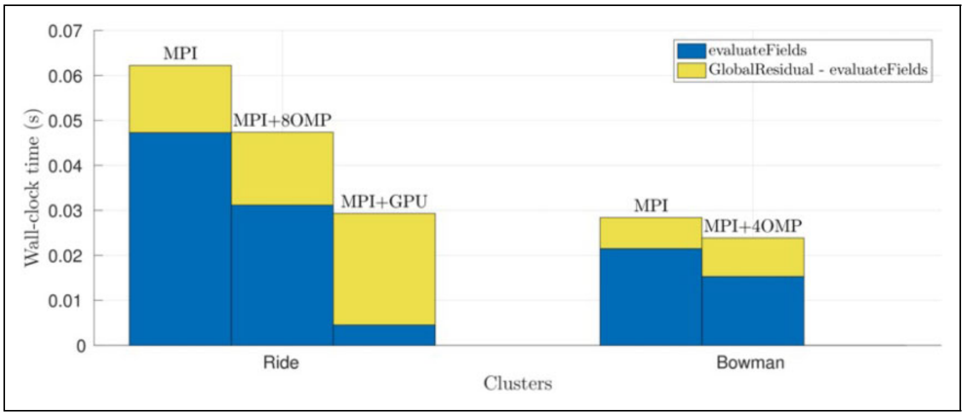


Figure 7. A comparison of wall-clock times (averaged over 400 evaluations) for Kokkos Serial, OpenMP, and CUDA execution spaces on a single node of the Ride and Bowman clusters (3-D hydrostatic test case, ne30 mesh). The sections in blue represent the residual computation along with the assembly of global and local finite element structures, while the sections in yellow are primarily dominated by global assembly and communication costs. Notably, the computational kernels perform better on the CUDA implementation, but performance is hindered by the communication costs from host to device. 3-D: three-dimensional.

increasing the element order as the computational cost becomes more significant, helping to mask the communication costs. This is shown in more detail in Section 4.3.2.

The performance of the P100 GPU and KNL Xeon Phi is also compared in Figures 8 and 9 for the same test case described above, showing strong and weak scaling on up to four devices. The results show that the P100 obtains faster results on larger worksets on a single device but struggles to strong and weak scale because of communication costs. In fact, when communication costs are not accounted for, as in the evaluateFields timer, the P100 tends to perform well. In the case of the shallow water test case (results given in Figures 1A to 1C), the KNL outperforms the P100 even for the largest problem because of communication costs.

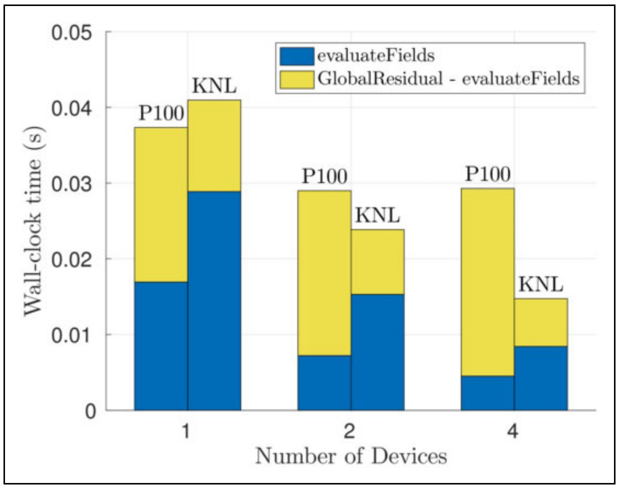


Figure 8. MPI strong scalability in terms of wall-clock time (averaged over 400 evaluations) on up to four P100 GPUs or KNL CPUs referred to as devices in the figure (3-D hydrostatic test case, ne30 mesh). The sections in blue represent the residual computation along with the assembly of global and local finite element structures, while the sections in yellow are primarily dominated by global assembly and communication costs. Notably, the computational kernels perform better on the CUDA implementation, but performance is hindered by the communication costs from host to device, becoming worse when strong scaling. 3-D: three-dimensional; KNL: Knights Landing; GPU: general processing unit; MPI: message passing interface.

4.3.2. MPI strong scalability. An MPI strong scalability study is performed on all executions spaces on up to 16 CPUs (128 cores), 4 KNLs (272 cores), and 16 GPUs to study the performance of the code as the workload per device becomes smaller and the communication overhead becomes more significant. The study is performed on both the Ride and Bowman clusters for the shallow water test case on the ne120 mesh and the 3-D hydrostatic test case on the ne30 mesh. The results are given in Tables 1C to 1F.

From the results, several observations can be made. First, the results show that the MPI implementation scales quite well in all cases. Conversely, the MPI + OpenMP and MPI + GPU strong scaling efficiency tends to drop when including the communication costs in GlobalResidual. An example of the MPI strong scalability for all architectures on the Ride cluster for the shallow water test case discretized using a p6 spectral element is shown in Figure 10. In this case, 8MPI or 8(MPI + 8OMP) are used per CPU and 1(MPI + GPU) is used per GPU.

There is also a drop in performance in evaluateFields when solving the 3-D hydrostatic equations. This may be due to a smaller workload per device with increased parallelization. The issue becomes more apparent when observing the strong scaling results of the GPU implementation. This is shown in Figure 11 for all architectures on the Ride cluster.

It is also important to note that, as expected, increasing the element order improves the MPI strong scalability for all architectures. In particular, the OpenMP and GPU implementations greatly benefit from the higher orders because the added computation works to mask the communication costs in GlobalResidual. The strong scaling of the GPU implementation is also improved in evaluateFields because the MD_Range policies allow for more opportunity for

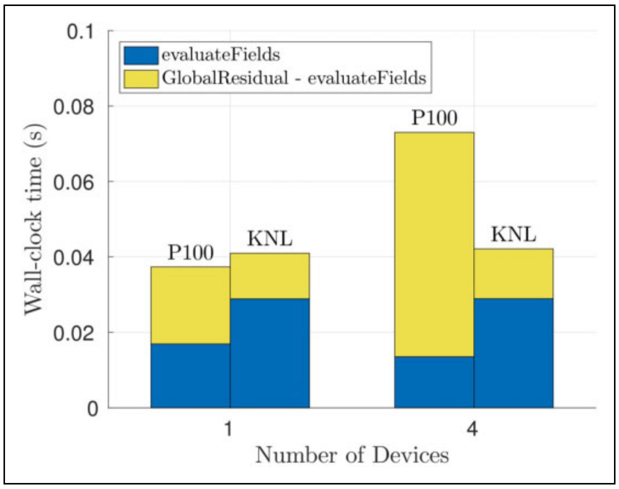


Figure 9. MPI weak scalability in terms of wall-clock time (averaged over 400 evaluations) for the P100 and KNL architectures where the ne30 mesh is used on one device and the ne60 mesh is split among four devices (3-D hydrostatic test case). The sections in blue represent the residual computation along with the assembly of global and local finite element structures, while the sections in yellow are primarily dominated by global assembly and communication costs. Notably, the computational kernels perform better on the CUDA implementation but performance is hindered by the communication costs from host to device, becoming worse when weak scaling. 3-D: three-dimensional; KNL: Knights Landing; MPI: message passing interface.

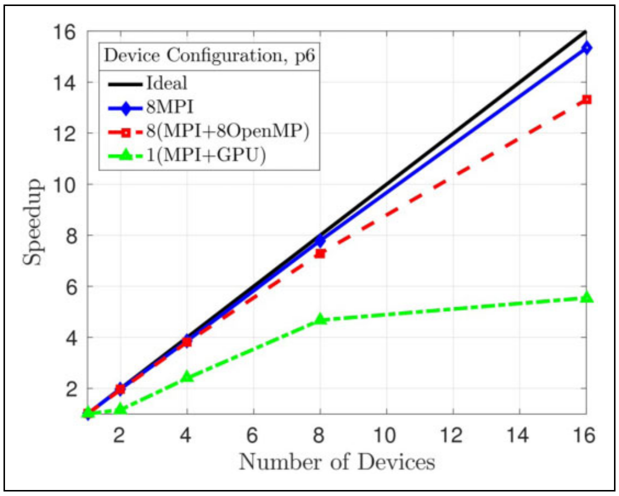


Figure 10. MPI strong scalability study on up to 16 CPU/GPUs on Ride where speedup is computed over a single device (CPU/GPU) and the GlobalResidual timer is used (shallow water test case, ne120 mesh). This case uses basis polynomials of degree 6 (p6). The strong scaling performance of MPI + GPU is hindered by the communication costs from host to device. GPU: general processing unit; MPI: message passing interface.

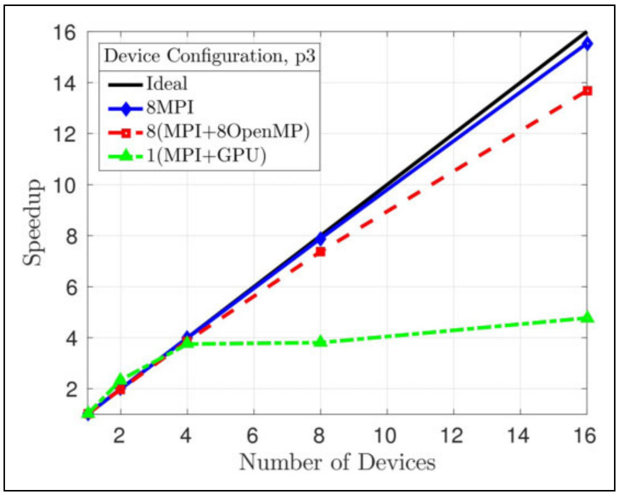


Figure 11. MPI strong scalability study on up to 16 CPU/GPUs on Ride where speedup is computed over a single device (CPU/GPU) and the evaluateFields timer is used (3-D hydrostatic test case, ne30 mesh). This case uses basis polynomials of degree 3 (p3). The strong scaling performance of MPI + GPU drops as significantly as the workload becomes smaller. 3-D: three-dimensional. GPU: general processing unit; MPI: message passing interface.

parallelization inside the kernels. This is discussed in more detail in Section 4.3.4. The reader can observe the benefit of using higher orders to obtain better strong scalability for MPI + OpenMP by examining Figure 12.

4.3.3. MPI weak scalability. An MPI weak scalability study is performed on up to 16 CPUs (128 cores), 4 KNLs (272 cores), and 32 GPUs for the various execution spaces in order to assess the performance of the code when the workload per device remains constant, as additional resources are used to solve a larger problem. As before, the study is performed on both the Ride and Bowman clusters for the shallow water and 3-D hydrostatic test problems. On the Ride cluster, weak scaling is studied by increasing the number of available sockets, where one socket contains one CPU and two GPUs. A series of three meshes are used for each case: the ne120, ne240, and ne480 meshes for the shallow water test, and the ne30, ne60, and ne120 meshes for the 3-D hydrostatic test. Detailed results can be found in Tables 1G to 1J.

The results show near perfect weak scaling on all architectures except the GPU. Figures 13 and 14 show examples of MPI weak scalability for all architectures on the Ride cluster for the shallow water test discretized using a p6 spectral element, and the 3-D hydrostatic test discretized using a p3 spectral element, respectively. In this case, 8MPI, 8(MPI + 8OMP), and 2(MPI + GPU) are used per socket. Despite the poor MPI + GPU weak scalability, the GPU is able to obtain a substantial gain in performance for the shallow water test case. In the 3-D hydrostatic test case, the GPU loses much of its performance when scaling to 32 GPUs. The poor weak scalability of MPI + GPU is explained by the large variation among MPI ranks as explained in Section 4.3.4.

Another interesting observation is that the weak scalability of MPI + GPU tends to improve with increasing element order. Figure 15 shows an example of weak scalability for MPI + GPU on the Ride cluster for the shallow water test case as the element order is increased. The results show that the efficiency, defined in the appendix, drops significantly for lower orders, but the GPU is better able

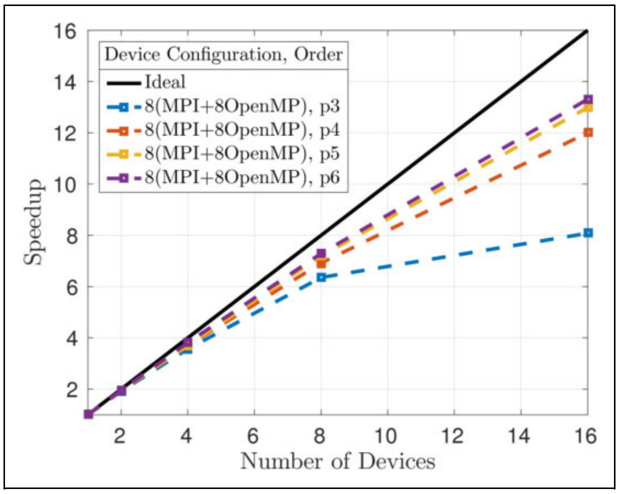


Figure 12. MPI + OpenMP strong scalability study on up to 16 CPU/GPUs on Ride where speedup is computed over a single device (CPU/GPU) and the GlobalResidual timer is used (shallow water test case, ne120 mesh). This case uses basis polynomials of degree 3 through 6 (p3 to p6). The strong scaling performance improves as the element order is increased because the added computation works to mask the communication costs. GPU: general processing unit; MPI: message passing interface.

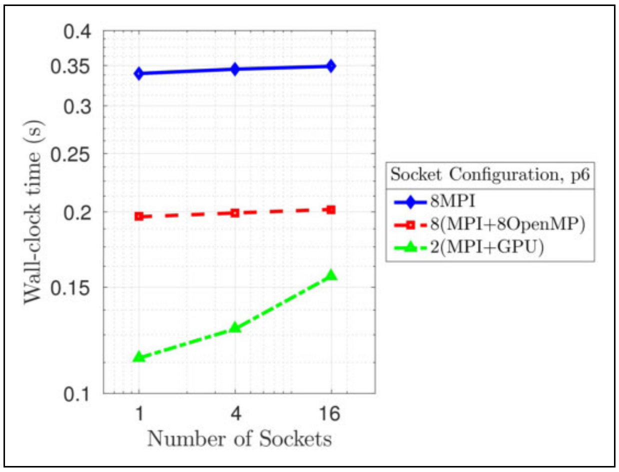


Figure 13. MPI weak scalability study in terms of wall-clock time (averaged over 400 evaluations) on up to 16 Sockets (16 CPUs and 32 GPUs) on Ride where the GlobalResidual timer is used (shallow water test case). This case uses basis polynomials of degree 6 (p6). Though the simulation speeds are faster, the weak scaling performance of MPI + GPU is hindered by the communication costs from host to device. GPU: general processing unit; MPI: message passing interface.

to maintain weak scaling for the p6 spectral element, which has more computational work available.

4.3.4. GPU profiling. In this section, we provide a brief overview of the performance of the MPI + GPU implementation by using NVIDIA's profiling tool called nvprof (Nvp, 2014). We analyze three of the most costly kernels within the evaluateFields timer of the code for the shallow water equations called "GatherSolution," "ShallowWaterResid," and "ScatterSolution." These kernels, as their names suggest, are responsible for the gathering of information from a global structure to an element local structure, the computation of the residual in each element, and the scattering of information from a local structure to a global structure, respectively. They are not responsible for any MPI communication. The percentage of wall-clock time for each kernel with respect to evaluateFields is shown in Figure 16 for the shallow water test case with element order 3 on the ne120 mesh on a single GPU.

Four performance properties are reported in Figure 17 for each kernel: compute utilization, bandwidth utilization, memory load efficiency, and occupancy. Compute and bandwidth utilizations estimate the percentage of maximum computational throughput and device memory bandwidth for the GPU that is utilized. Global memory load efficiency, defined as the number of bytes requested divided by the number of bytes actually transferred, gives an estimate of how well the kernel is utilizing the available bandwidth. Finally, occupancy estimates the percentage of maximum number of warps on the GPU that are active.

As shown in Figure 17, the GPU utilization of the GatherSolution and ScatterResidual kernels is extremely low and the operations are latency bound. The occupancy of these

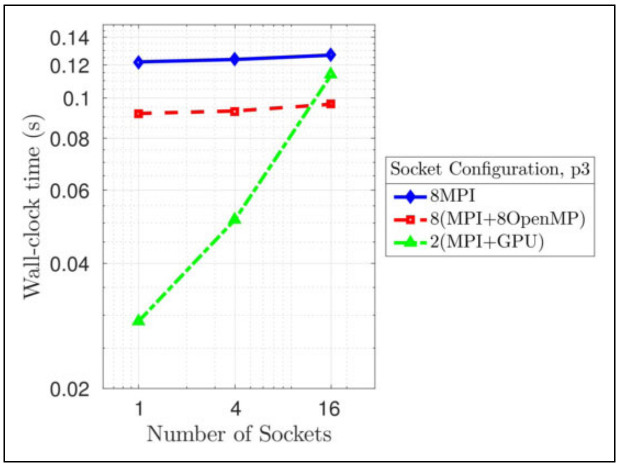


Figure 14. MPI weak scalability study in terms of wall-clock time (averaged over 400 evaluations) on up to 16 Sockets (16 CPUs and 32 GPUs) on Ride where the GlobalResidual timer is used (3-D hydrostatic test case). This case uses basis polynomials of degree 3 (p3). The weak scaling performance of MPI + GPU slows down the simulation enough to perform worse than the MPI + OpenMP implementation. This is mostly caused by the communication costs from host to device. 3-D: three-dimensional. GPU: general processing unit; MPI: message passing interface.

kernels is relatively high, so the warps on the GPU are being sufficiently utilized. It is also apparent that the GatherSolution kernel has low global memory load efficiency.

We found that an MD_Range policy reduced the performance of the GatherSolution kernel but slightly improved the ScatterResidual kernel for this case.

The ShallowWaterResid kernel seems to have a higher global memory load efficiency but a lower occupancy, but still suffers from uncoalesced memory. Figure 17 shows that

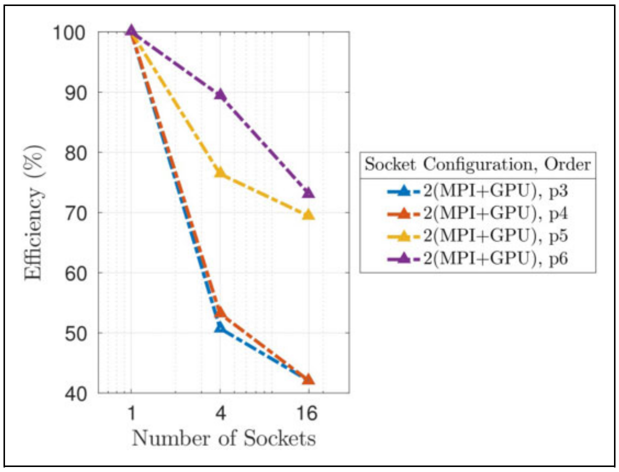


Figure 15. MPI + GPU weak scalability in terms of efficiency on up to 16 Sockets (16 CPUs and 32 GPUs) on Ride where the GlobalResidual timer is used (shallow water test case). This case uses basis polynomials of degree 3 through 6 (p3 to p6). The weak scaling performance improves as the element order is increased because the added computation works to mask the communication costs. GPU: general processing unit; MPI: message passing interface.

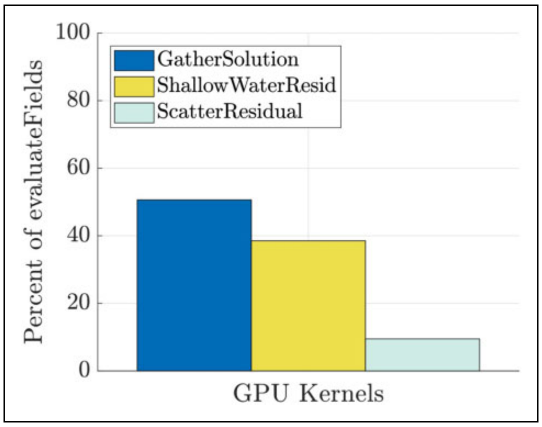


Figure 16. A comparison of wall-clock time percentages of evaluateFields for three kernels on a single GPU (shallow water test case, ne120). This case uses basis polynomials of degree 3 (p3). A majority of the time (roughly 60%) is spent assembling global and local finite element structures (i.e. the GatherSolution and ScatterResidual kernels). GPU: general processing unit.

it is bandwidth bound, as 75% of the memory bandwidth is being utilized. The MD_Range policy used in the ShallowWaterResid kernel helps to produce better strong scaling when the element order is high, as shown in Figure 18.

Finally, putting all these kernels together in an MPI framework causes some of the MPI ranks to lag, producing a large variation in the wall-clock times for MPI + GPU strong scaling and weak scaling. Figure 19 shows an example of this variation for MPI + GPU strong scalability on the shallow water test discretized using a p3 spectral element. This large variation is not caused by issues with load balancing, as all test cases are properly load balanced. For example, for the shallow water test case, ne120 mesh on 32 GPUs, the distribution of elements among 32 GPUs is exactly 2700 elements per GPU.

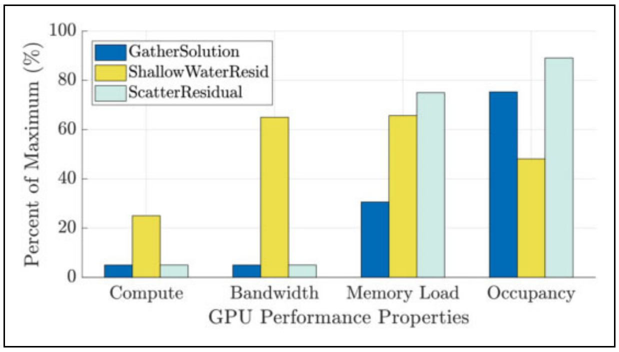


Figure 17. A comparison of GPU performance properties for three kernels on a single GPU (shallow water test case, ne120). This case uses basis polynomials of degree 3 (p3). One of the most noticeable characteristics is that the GPU is heavily underutilized in the GatherSolution and ScatterResidual kernels. GPU: general processing unit.

5. Concluding remarks

In this article, we have described a way to achieve a single, performance portable implementation of a large finite element multiphysics code known as Albany using the Kokkos library. Our numerical experiments on the Aeras dynamical core implemented within the Albany multiphysics finite element code show that (1) a single code can execute correctly in several evaluation environments (MPI, OpenMP, and CUDAUVM) and (2) reasonable performance is achieved across several different architectures, including NVIDIA P100 GPUs and Intel KNL Xeon Phis, without manual data management or architecture-dependent code optimizations. In our analysis, we have been able to demonstrate speedups over traditional MPI-only simulations by utilizing GPUs and hardware threads via OpenMP for fine grain parallelism. We also provide a detailed discussion into the limitations of the MPI + GPU scalability and show that results can be improved by utilizing Kokkos MDRange policies and increasing the element order to increase the computational intensity of the problem and mask communication costs.

The refactoring of the Albany code is an ongoing process and future work includes a collaboration with Trilinos developers toward creating a performance-portable suite of linear solvers accessible by the numerous Albany physics discretized via implicit methods. We also plan to implement manual CPU-GPU data management, CUDA-Aware MPI and improve memory access patterns to further boost scalability and performance for the GPU implementation in Albany in future work. Other ongoing work includes integrating Kokkos into the High-Order Method Modeling Environment (Dennis et al., 2012), a production atmosphere dynamical core, as part of the Climate Model Development and Verification project funded by the U.S. Department of Energy's Biological and Environmental Research program.

6. Results reproducibility

In this section, we provide some details pertaining to the compilation of Trilinos, Kokkos, and Albany, so that the interested reader can replicate our results.

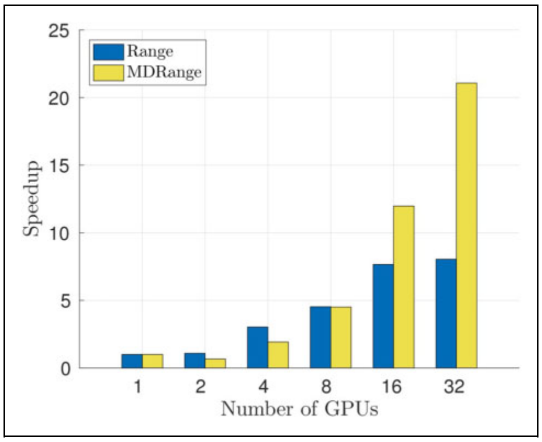


Figure 18. MPI + GPU strong scalability for the Range and MD_Range policies on up to 32 GPUs where speedup is computed over a single GPU and the evaluateFields timer is used (shallow water test case, ne120 mesh). This case uses basis polynomials of degree 6 (p6). The MD_Range policies scale better at high element orders because there is more computational work available to be parallelized. GPU: general processing unit; MPI: message passing interface.

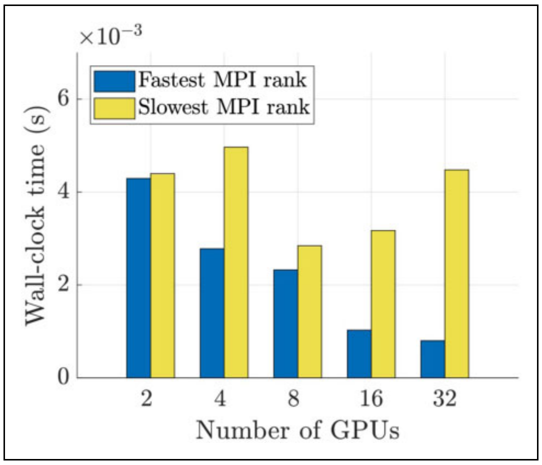


Figure 19. A comparison of wall-clock times (averaged over 400 evaluations) between the fastest MPI rank and the slowest MPI rank for an MPI + GPU strong scalability study on up to 32 GPUs where the evaluateFields timer is used (shallow water test case, ne120 mesh). This case uses basis polynomials of degree 3 (p3). The variation among MPI ranks in evaluateFields leads to poor performance when scaling. GPU: general processing unit; MPI: message passing interface.

The results presented in this article were generated using the following branches and git commit hashes of Trilinos, Kokkos, and Albany:

- Trilinos[3]: master branch, git commit hash 4a8b7ce13614558b9839e029f3b502eb3bdddca6.
- Kokkos[4]: master branch, git commit hash aed197c676e1edff4861d6b73ec8f8b1d359cacd.
- Albany[5]: aeras-mdrange tag of the master branch, git commit hashe 067178efef32301ada5ce1b99b4ae7a5409c7cb.

These builds have been tested with the following TPLs: cmake-3.5.2 and 3.6.2, gcc-5.4.0, intel-17.0.098, openmpi-1.10.4, boost-1.55.0 and 1.60.0, openblas-0.2.19, netcdf-4.4.1, pnetcdf-1.6.1, and 1.7.0, hdf5 -1.8.17. For detailed information about compiling Trilinos, Albany, and the required TPLs, the reader is referred to the following Albany wiki pages: https://github.com/gahansen/Albany/wiki/building-albany-and-supporting-tools and https://github.com/gahansen/Albany/wiki/Information-about-Aeras-MDRange. The cmake configure scripts for building Trilinos and Albany can be found in the Albany/doc directory and are linked in the latter wiki page. Detailed information of the architectures and compilers employed in our numerical studies can be found in Table 1.

Acknowledgments

The authors would like to thank Christian Trott, Carter Edwards, and Dan Sunderland for their help with Kokkos; Adam Delora, Mauro Perego, and Kyungjoo Kim for their work on Intrepid and Intrepid2; and Eric Phipps, Eric Cyr, and Andrew Bradley for their help with Trilinos and Albany. Sandia National Laboratories is a multiprogram 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 (DOE's) National Nuclear Security Administration under contract DE-NA0003525.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by Sandia National Laboratories Directed Research and Development (LDRD) program for the development of the Aeras code. The Albany to Kokkos refactoring work leveraged Frameworks, Algorithms and Scalable Technologies for Mathematics (FASTMath) funding from the Advanced Scientific Computing (ASC) program within DOE's NNSA. Some support for the performance studies described herein was provided through the Climate Model Development and Verification (CMDV) project funded by the DOE's Biological and Environmental Research (BER) program.

Notes

1. Note that the Albany MPI implementation does not use any MPI-shared memory support.
2. For an explicit definition of the vertical coordinate η, the reader is referred to the works of Taylor, (2012).
3. Trilinos is available on github at https://github.com/trilinos/Trilinos.
4. Kokkos is available on github at https://github.com/kokkos.
5. Albany is available on github at https://github.com/gahansen/Albany.

References

Alnæs MS, Blechta J, Hake J, et al (2015) The FEniCS project version 1.5. *Archive of Numerical Software* 3(100): 9–23.

Alnæs MS, Logg A, Ølgaard KB, et al. (2014) Unified form language: a domain-specific language for weak formulations of partial differential equations. *ACM Transactions on Mathematical Software (TOMS)* 40(2): 9. Available at: https://doi.org/10.1145/2566630.

Baer F, Wang H, Tribbia JJ, et al. (2006) Climate modeling with spectral elements. *Monthly Weather Review* 134: 3610–3624.

Bochev P, Edwards HC, Kirby RC, et al. (2012) Solving pdes with intrepid. *Scientific Programming* 200(2): 151–180.

Dennis JM, Edwards J, Evans KJ, et al. (2012) CAM-SE: a scalable spectral element dynamical core for the community atmosphere model. *Journal of High Performance Computing Applications* 260(1): 74–89.

Edwards HC, Trott CR and Sunderland D (2014) Kokkos: enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing* 740(12): 3202–3216.

Edwards HC, Williams AB, Sjaardema GD, et al. (2010) *SIERRA Toolkit Computational Mesh Conceptual Model: Technical Report*. Albuquerque, New Mexico 87185 and Livermore, California 94550: Sandia National Labs SAND Report.

Evans KJ, Taylor MA and Drake JB (2010) Accuracy analysis of a spectral element atmospheric model using a fully implicit solution framework. *Monthly Weather Review* 138: 3333–3341.

Gao X, Nielsen E, Muller R, et al. (2013) Quantum computer aided design simulation and optimization of semiconductor quantum dots. *Journal of Applied Physics* 114: 1–19.

Guba O, Taylor MA, Ullrich PA, et al. (2014) The spectral element method on variable resolution grids: evaluating grid sensitivity and resolution-aware numerical viscosity. *Geoscientific Model Development* 7: 4081–4117.

Harris M (2015) Developing portable CUDA C/C++ code with Hemi. Available at: http://devblogs.nvidia.com/parallelforall/developing-portable-cuda-cc-code-hemi/ (accessed 11 January 2018).

Heroux MA, Bartlett R, Howle V, et al. (2003) *An Overview of Trilinos*: *Technical Report SAND2003-2927*. Albuquerque, NM: Sandia National Laboratories.

Heuveline V (2010) Hiflow3: a flexible and hardware-aware parallel finite element package. In: *Proceedings of the 9th workshop on parallel/high-performance object-oriented scientific computing*. Available at: http://journals.ub.uni-heidelberg.de/index.php/emcl-pp/article/view/11675 (accesed January 12, 2018).

Hornung RD and Keasler JA (2014) *The RAJA Portability Layer: Overview and Status: Technical Report*. Livermore, CA: Lawrence Livermore National Laboratory (LLNL).

Jablonowski C and Williamson DL (2006) A baroclinic instability test case for atmospheric dynamical cores. *Quarterly Journal of the Royal Meteorological Society* 132: 2943–2975. DOI: 10.1256/qj.06.n.

Kaiser H, Brodowicz M and Sterling T (2009) Parallex an advanced parallel execution model for scaling-impaired applications. In: *Proceedings of the 2009 international conference on parallel processing workshops*, pp. 394–401. doi:10.1109/ICPPW.2009.14.

Kale LV, Bohm E, Mendes CL, et al. (2008) Programming petascale applications with Charm++ and AMPI. In: *Petascale Computing: Algorithms and Applications*, pp. 421–441.

Lewis B and Berg DJ (1998) *Multithreaded Programming With Pthreads*. Upper Saddle River, NJ: Prentice-Hall, Inc.

Medina DD, St-Cyr A and Warburton T (2014) OCCA: a unified approach to multi-threading languages. *SIAM Journal on Scientific Computing (SISC)*. Avaialable at: http://arxiv.org/abs/1403.0968.

Mudalige GR, Giles MB, Reguly I, et al. (2012) Op2: an active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In: *Innovative Parallel Computing (InPar)*, pp. 1–12. San Jose, CA.

Munshi A (2009) The opencl specification. In Khronos (Ed.) *Hot Chips 21 Symposium (HCS)*, 2009, pp. 1–314. Stanford, CA: IEEE.

Nvp (2014) Nvidia visual profiler (nvprof) users' guide. Available: http://docs.nvidia.com/cuda/profiler-users-guide (accessed 11 January 2018).

Ober CC, Barlett RA, ToCoffey DS, et al. (2013) *Rythmos: solution and analysis package for differential-algebraic and ordinary-differential equations. Technical report*. Albuquerque, New Mexico 87185 and Livermore, California 94550: Sandia National Labs SAND Report.

OpenACC (2013) *The OpenACC Application Programming Interface: Technical Report*. OpenACC-Standard.org Avaialable at: https://www.openacc.org (accessed 11 January 2018).

OpenMP (2013) *OpenMP Application Program Interface*: *Technical Report*. OpenMP Architecture Review Board.

Pawlowski RP, Phipps ET and Salinger AG (2012) Automating embedded analysis capabilities and managing software complexity in multiphysics simulation. Part I: template-based generic programming. *Science Program* 200(2): 197–219. ISSN: 1058-9244. DOI: 10.1155/2012/202071.

Phipps E and Pawlowski R (2012) Efficient expression templates for operator overloading-based automatic differentiation. In: *Proceedings of the 6th international conference on automatic differentiation*, July 2012. Avaialable at: https://doi.org/10.1007/978-3-642-30023-3_28.

Rathgeber F, Ham DA, Mitchell L, et al. (2017) Firedrake: automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software (TOMS)* 43(3): 24.

Rathgeber F, Markall GR, Mitchell L, et al. (2012) Pyop2: a high-level framework for performance-portable simulations on unstructured meshes. In: *2012 SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC)*, pp. 1116–1123. IEEE.

Salinger A, Bartlett R, Bradley A, et al. (2016) Albany: using agile components to develop a flexible, generic multiphysics analysis code. *International Journal for Multiscale Computational Engineering* 14(4): 415–438.

Spotz WF, Smith TM, Demeshko IP, et al. (2015) Aeras: a next-generation global atmosphere model. *Procedia Computer Science* 51: 2097–2106.

Sun W, Ostien J and Salinger A (2013) A stabilized assumed deformation gradient finite element formulation for strongly coupled poromechanical simulations at finite strain. *International Journal for Numerical and Analytical Methods* 37: 2755–2788.

Taylor MA (2012) Conservation of mass and energy for the moist atmospheric primitive equations on unstrucutred grids. In: Lauritzen PH, Jablonowski C, Taylor MA and Nair RD (eds) *Numerical Techniques for Global Atmospheric Models. Lecture Notes in Computational Science and Engineering*, 80. Berlin, Heidelberg: Springer.

Taylor MA, Edwards J, Thomas S, et al. (2007) A mass and energy conserving spectral element atmospheric dynamical core on the cubed-sphere grid. *Journal of Physics: Conference Series*, 78: 012074.

Tezaur I, Perego M, Salinger A, et al. (2015) Albany/FELIX: a parallel, scalable and robust finite element higher-order stokes ice sheet solver built for advanced analysis. *Geoscientific Model Development* 8: 1–24.

Ullrich PA, Jablonowski C, Kent J, et al. (2012) *Dynamical Core Model Intercomparison Project (DCMIP) Test Case Document: Technical Report*. National Center for Atmospheric Research. Available at: https://earthsystemcog.org/projects/dcmip-2012/test_cases (accessed 11 January 2018).

Unat D, Chan C, Zhang W, et al. (2013) Tiling as a durable abstraction for parallelism and data locality. In: *WOLFHPC: Workshop on domain-specific languages and high-level frameworks for HPC*.

Weber R, Gothandaraman A, Hinde RJ, et al. (2010) Comparing hardware accelerators in scientific applications: a case study. *IEEE Transactions on Parallel and Distributed Systems* 99: 58–68.

Williamson DL, Drake JB, Hack JJ, et al. (1992) A standard test set for numerical approximations of the shallow water equations in spherical geometry. *Journal of Computational Physics* 102: 211–224.

Author biographies

Irina Demeshko is a computational scientist in the codesign team at LANL (CCS-7). Her research work is focused around new HPC technologies in application to large-scale scientific simulation codes. In particular, she is interested in adapting simulation codes to the state-of-the-art HPC architecture by using task-based programming models like Legion. She got her PhD in computation science from the Tokyo Institute of Technology in 2013. Her PhD thesis work was focused on porting NICAM climate/weather simulation code from RIKEN to CPU-GPU heterogeneous architecture. Prior to Los Alamos, she spent 2.5 years at Sandia National Laboratory where she worked on porting Albany code to Kokkos.

Jerry Watkins is a graduate technical intern in the Extreme Scale Data Science and Analytics Department at Sandia National Laboratories in Livermore, CA, USA. He is also a PhD candidate in the Aeronautics and Astronautics Department at Stanford University with a particular interest in the development of high-order numerical methods for GPU architectures. He graduated with highest honors from the University of California, Davis, with a Bachelors degree in mechanical and aerospace engineering and obtained a Masters degree in aeronautics and astronautics from Stanford University in 2014. He has received multiple awards including a Departmental Citation Award from UC Davis, the Rose M. Chappelear Memorial Stanford University fellowship, and the National Science Foundation Graduate Research fellowship.

Irina K Tezaur is a principal member of technical staff (PMTS) in the Extreme Scales Data Science and Analytics Department at Sandia National Laboratories in Livermore, CA, USA. Prior to joining Sandia, she received a PhD in computational and mathematical engineering (CME) from Stanford University under the guidance of Professor Charbel Farhat. She also holds Bachelors and Masters degrees in mathematics from the University of Pennsylvania. In 2008, She was the recipient Duke University's Robert J. Melosh Medal for the best student paper in finite element analysis. In 2007, she was awarded two fellowships, which funded my graduate studies: the National Defense Science and Engineering Graduate (NDSEG) fellowship and the National Physical Science Consortium (NPSC) fellowship.

Oksana Guba is a computational scientist at the Center for Computing Research at Sandia National Labs. Her current research is focused on atmospheric modeling, with emphasis on numerical methods for PDEs and high performance computing.

William F Spotz, Ph.D., is a computational scientist in the Center for Computing Research at Sandia National Laboratories in Albuquerque, New Mexico. He earned his Ph.D. in 1995 in Aerospace Engineering at the University of Texas at Austin focusing on computational fluid dynamics, and conducted his postdoctoral fellowship with the Advanced Study Program at the National Center for Atmospheric Research in Boulder, Colorado. His research at Sandia has spanned a wide diversity of topics within computational science, and he was principal investigator of the project that developed the Aeras application. He also served for two years as a program manager in the Department of Energy Office of Science.

Andrew G Salinger is a computational scientist at the Center for Computing Research at Sandia National Labs. His current research focus is on developing improved software and algorithms for the E3SM Earth system modelling project, drawing in part on years of experience contributing to the Trilinos and Albany open source software projects.

Roger P Pawlowski is a Principal Member of Technical Staff in the Multiphysics Applications Department at Sandia National Laboratories. His research interests include numerical algorithm development, high performance computing, novel computing architectures, and large-scale PDE solution technology. He contributes to HPC physics applications in the areas of computational fluid dynamics, magnetohydrodynamics and plasma physics. He leads the Nonlinear Analysis area for Trilinos, an effort to develop algorithms and enabling technologies within an object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems. Trilinos contributions include the NOX nonlinear solvers library, the Phalanx multiphysics library and the Panzer finite element assembly library. Dr. Pawlowski is also a co-lead for the Drekar application code, used for large-scale multiphysics PDE research in plasma physics.

Michael A Heroux is a Senior Scientist at Sandia National Laboratories, Director of SW Technologies for the US DOE Exascale Computing Project (ECP) and Scientist in Residence at St. John's University, MN. His research interests include all aspects of scalable scientific and engineering software for new and emerging parallel computing architectures.

Appendix 1

The data collected from the performance studies described in Section 4.3 are summarized in the following sections in order to give a more detailed overview of the performance of Aeras.

1.1. Architecture comparison

Tables 1A and 1B report the speedup of the OpenMP and CUDA execution spaces over the Serial execution space for the shallow water and 3-D hydrostatic cases on the ne120 and ne30 meshes, respectively, for the Ride and Bowman clusters. The tables are generated by reporting the wall-clock time in place of the speedup for the Kokkos Serial execution space in order to give the reader the ability to compute the wall-clock time for all simulations. The speedup is used to determine the overall improvement achieved though the use of next generation architectures on Ride and Bowman is given by

$$\text{Speedup} = \frac{t_{\text{Serial}}}{t_{\text{Exe}}} \tag{1A}$$

where t_{Exe} is the wall-clock time given by the timers GlobalResidual and evaluateFields using execution space "Exe" for additional level of parallelism. Here, valid options for "Exe" are "Serial," "OpenMP," and "CUDA."

Figures 1A to 1D compare the P100 GPU and KNL Xeon Phi for the shallow water test case on the ne120 mesh, separating the results from the GlobalResidual and evaluateFields timers. MPI strong and weak scalability in terms of wall-clock time is shown for up to four devices for each case.

1.2. MPI Strong Scalability

Tables 1C to 1F report speedup over a single CPU or GPU when performing an MPI strong scalability study for the shallow water and 3-D hydrostatic cases on the ne120 and ne30 meshes, respectively, for the Ride and Bowman clusters. The tables are generated by reporting the wall-clock time in place of the speedup for a single device in order to give the reader the ability to compute the wall-clock time for all simulations. The speedup is used to determine how well the code utilizes additional resources to achieve a faster simulation of the same problem. In this case, the speedup is given by

$$\text{Speedup} = \frac{t_1}{t_d} \tag{1B}$$

where t_d is the wall-clock time given by the timers GlobalResidual and evaluateFields when running the simulation using d devices.

1.3. MPI weak scalability

Tables 1G to 1J report weak scaling efficiency with respect to a single socket (one CPU or two GPUs) for the Ride cluster and single KNL for the Bowman

Table 1A. Kokkos OpenMP and CUDA speedup over Kokkos Serial (MPI only) on a single node of the Ride cluster (ne120 mesh).[a]

		GlobalResidual			evaluateFields		
Kokkos execution space		Serial	OpenMP	CUDA	Serial	OpenMP	CUDA
Node configuration		16MPI	16(MPI + 8OMP)	4(MPI + GPU)	16MPI	16(MPI + 8OMP)	4(MPI + GPU)
Shallow water	p3	(0.0343s)	1.64	2.14	(0.0323s)	1.83	8.29
	p4	(0.0622s)	1.64	2.56	(0.0585s)	1.77	8.23
	p5	(0.106s)	1.66	2.69	(0.100s)	1.76	8.57
	p6	(0.171s)	1.70	3.08	(0.163s)	1.79	7.07
3-D Hydrostatic	p3	(0.0622s)	1.31	2.12	(0.0473s)	1.52	10.5

3-D: three-dimensional; GPU: general processing unit.
[a]Wall-clock time is given in place of speedup for the Kokkos Serial execution space.

Table 1B. Kokkos OpenMP speedup over Kokkos Serial (MPI only) on a single node (or single KNL) of the Bowman cluster (ne30 mesh).[a]

		GlobalResidual		evaluateFields	
Kokkos execution space		Serial	OpenMP	Serial	OpenMP
Node configuration		68MPI	68(MPI + 4OMP)	68MPI	68(MPI + 4OMP)
Shallow water	p3	(0.0141s)	1.09	(0.0120s)	1.46
	p4	(0.0234s)	1.09	(0.0208s)	1.29
	p5	(0.0377s)	1.12	(0.0342s)	1.31
	p6	(0.0568s)	1.09	(0.0528s)	1.23
3-D Hydrostatic	p3	(0.0284s)	1.19	(0.0215s)	1.40

3-D: three-dimensional; KNL: Knights Landing.
[a]Wall-clock time (averaged over 400 evaluations) is given in place of speedup for the Kokkos Serial execution space.

Table 1C. MPI strong scalability on the Ride cluster (shallow water test case, ne120 mesh).[a]

		GlobalResidual					evaluateFields				
Elements per device		86400	43200	21600	10800	5400	86400	43200	21600	10800	5400
# of CPUs or GPUs		1	2	4	8	16	1	2	4	8	16
8MPI	p3	(0.0677s)	1.98	3.86	7.49	14.4	(0.0644s)	2.0	3.99	7.98	16.0
	p4	(0.123s)	1.97	3.88	7.72	14.9	(0.117s)	2.0	3.98	7.99	15.9
	p5	(0.208s)	1.97	3.91	7.75	15.0	(0.199s)	1.98	3.99	7.97	15.9
	p6	(0.339s)	1.99	3.86	7.79	15.3	(0.326s)	2.0	4.0	7.99	16.0
8(MPI + OMP8)	p3	(0.0401s)	1.92	3.57	6.36	8.08	(0.0350s)	1.99	4.01	8.0	15.6
	p4	(0.0734s)	1.93	3.7	6.92	12.0	(0.0657s)	1.99	3.97	8.02	16.1
	p5	(0.124s)	1.95	3.7	7.2	13.0	(0.113s)	1.98	3.98	7.95	16.0
	p6	(0.196s)	1.95	3.82	7.29	13.3	(0.181s)	1.98	3.97	7.93	15.8
1(MPI + GPU)	p3	(0.0281s)	1.5	1.76	2.17	1.14	(0.00793s)	1.82	2.04	3.06	4.65
	p4	(0.0494s)	1.62	2.03	1.61	1.69	(0.0160s)	2.04	2.25	3.51	6.19
	p5	(0.0829s)	1.15	2.1	3.27	3.03	(0.0276s)	0.72	2.35	6.5	10.5
	p6	(0.133s)	1.16	2.4	4.67	5.54	(0.0443s)	0.665	1.92	4.5	12.0

[a]GPU: general processing unit.
Wall-clock time (averaged over 400 evaluations) is given in place of speedup for a single device while speedup over a single device is reported for all other simulations.

Table 1D. MPI strong scalability on the Ride cluster (3-D hydrostatic test case, ne30 mesh).[a]

	GlobalResidual					evaluateFields				
Elements per device	5400	2700	1350	675	337	5400	2700	1350	675	337
# of CPUs or GPUs	1	2	4	8	16	1	2	4	8	16
8MPI	(0.122s)	1.96	3.85	7.32	13.8	(0.0944s)	2.0	3.99	7.89	15.5
8(MPI + 8OMP)	(0.0917s)	1.94	3.57	6.4	10.9	(0.0618s)	1.98	3.88	7.37	13.7
1(MPI + GPU)	(0.0373s)	1.29	1.27	1.41	2.09	(0.0169s)	2.35	3.75	3.8	4.77

3-D: three-dimensional; GPU: general processing unit.
[a]Wall-clock time (averaged over 400 evaluations) is given in place of speedup for a single device while speedup over a single device is reported for all other simulations.

cluster for the shallow water and 3-D hydrostatic cases. The tables are generated by reporting the wall-clock time in place of the efficiency for resources on a single socket or KNL in order to give the reader the ability to compute the wall-clock time for all simulations. The efficiency is used to determine how well the code is able to maintain the same wall-clock time when simulating larger problems with a proportionally equal larger amount of resources. The weak scaling efficiency is given by

$$\text{Weakscalingefficiency} = (\frac{t_1}{t_d})100\% \tag{1C}$$

where t_d is the wall-clock time given by the timers GlobalResidual and evaluateFields when d sockets or KNLs were used for the simulation.

Table 1E. MPI strong scalability on the Bowman cluster (shallow water test case, ne120 mesh).[a]

		GlobalResidual			evaluateFields		
Elements per Device		86400	43200	21600	86400	43200	21600
# of KNLs		1	2	4	1	2	4
68MPI	p3	(0.0262s)	1.86	3.40	(0.0237s)	1.97	3.84
	p4	(0.0447s)	1.91	3.57	(0.0414s)	1.99	3.91
	p5	(0.0727s)	1.93	3.67	(0.0682s)	1.99	3.94
	p6	(0.111s)	1.96	3.76	(0.105s)	1.99	3.95
68(MPI + 4OMP)	p3	(0.0211s)	1.64	2.5	(0.0157s)	1.91	3.59
	p4	(0.0387s)	1.80	3.04	(0.0317s)	1.96	3.8
	p5	(0.0595s)	1.78	3.21	(0.0511s)	1.96	3.85
	p6	(0.0951s)	1.83	3.32	(0.0844s)	1.97	3.89

KNL: Knights Landing.
[a]Wall-clock time (averaged over 400 evaluations) is given in place of speedup for a single device while speedup over a single device is reported for all other simulations.

Table 1F. MPI strong scalability on the Bowman cluster (3-D hydrostatic test case, ne30 mesh).[a]

	GlobalResidual			evaluateFields		
Elements per Device	5400	2700	1350	5400	2700	1350
# of KNLs	1	2	4	1	2	4
68MPI	(0.0529s)	1.86	3.40	(0.0422s)	1.96	3.80
68(MPI+4OMP)	(0.0410s)	1.72	2.78	(0.0289s)	1.89	3.43

3-D: three-dimensional; KNL: Knights Landing.
[a]Wall-clock time (averaged over 400 evaluations) is given in place of speedup for a single device while speedup over a single device is reported for all other simulations.

Table 1G. MPI weak scalability on the Ride cluster (shallow water test case).[a]

		GlobalResidual			evaluateFields		
Mesh		ne120	ne240	ne480	ne120	ne240	ne480
Sockets		1	4	16	1	4	16
8MPI	p3	(0.0677s)	96.7	96.8	(0.0644s)	100.0	100.0
	p4	(0.123s)	97.5	96.4	(0.117s)	100.0	100.0
	p5	(0.208s)	97.8	97.2	(0.199s)	100.0	100.0
	p6	(0.339s)	98.3	97.3	(0.326s)	100.0	100.0
8(MPI + 8OMP)	p3	(0.0401s)	98.8	96.2	(0.0350s)	101.0	101.0
	p4	(0.0734s)	98.7	96.1	(0.0657s)	100.0	100.0
	p5	(0.124s)	98.6	97.0	(0.113s)	100.0	100.0
	p6	(0.196s)	98.5	97.2	(0.181s)	99.8	100.0
2(MPI + GPU)	p3	(0.0187s)	50.7	42.0	(0.00435s)	52.6	51.0
	p4	(0.0304s)	53.2	42.0	(0.00783s)	41.0	64.2
	p5	(0.0723s)	76.4	69.4	(0.0383s)	106.0	114.0
	p6	(0.114s)	89.4	73.1	(0.0665s)	99.7	115.0

GPU: general processing unit.
[a]Wall-clock time (averaged over 400 evaluations) is given in place of efficiency for a single socket while efficiency with respect to resources on a single socket is reported for all other simulations.

Table 1H. MPI weak scalability on the Ride cluster (3-D hydrostatic test case).[a]

	GlobalResidual			evaluateFields		
Mesh	ne30	ne60	ne120	ne30	ne60	ne120
Sockets	1	4	16	1	4	16
8MPI	(0.122s)	98.7	96.3	(0.0944s)	100.0	100.0
8(MPI + 8OMP)	(0.0917s)	98.6	95.0	(0.0618s)	101.0	101.0
2(MPI + GPU)	(0.0290s)	57.0	25.5	(0.00722s)	84.3	86.8

3-D: three-dimensional; GPU: general processing unit.
[a]Wall-clock time (averaged over 400 evaluations) is given in place of efficiency for a single socket while efficiency with respect to resources on a single socket is reported for all other simulations.

Table 1I. MPI weak scalability on the Bowman cluster (shallow water test case).[a]

		GlobalResidual		evaluateFields	
Mesh		ne120	ne240	ne120	ne240
KNLs		1	4	1	4
68MPI	p3	(0.0262s)	99.1	(0.0237s)	100.0
	p4	(0.0447s)	98.9	(0.0414s)	100.0
	p5	(0.0727s)	99.6	(0.0682s)	100.0
	p6	(0.111s)	99.6	(0.105s)	100.0
68(MPI + 4OMP)	p3	(0.0211s)	90.5	(0.0157s)	99.5
	p4	(0.0387s)	98.7	(0.0317s)	100.0
	p5	(0.0595s)	98.3	(0.0511s)	100.0
	p6	(0.0951s)	95.8	(0.0844s)	99.5

KNL: Knights Landing.
[a]Wall-clock time (averaged over 400 evaluations) is given in place of efficiency for a single KNL while efficiency with respect to a single KNL is reported for all other simulations.

Table 1J. MPI weak scalability on the Bowman cluster (3-D hydrostatic test case).[a]

	GlobalResidual		evaluateFields	
Mesh	ne30	ne60	ne30	ne60
KNLs	1	4	1	4
68MPI	(0.0529s)	98.7	(0.0422s)	100.0
68(MPI + 4OMP)	(0.0410s)	97.2	(0.0289s)	99.9

3-D: three-dimensional; KNL: Knights Landing.
[a]Wall-clock time (averaged over 400 evaluations) is given in place of efficiency for a single KNL while efficiency with respect to a single KNL is reported for all other simulations.

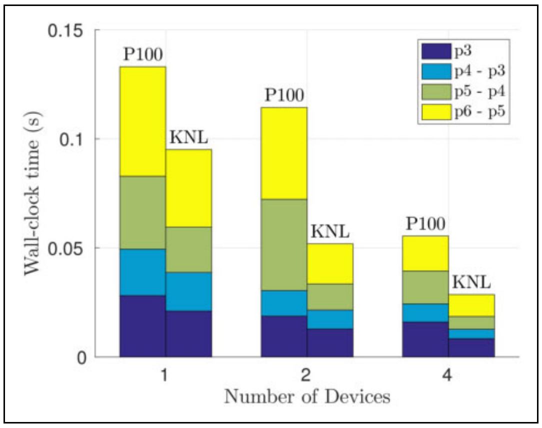


Figure 1A. MPI strong scalability in terms of wall-clock time (averaged over 400 evaluations) for the P100 and KNL architectures on up to four devices using the GlobalResidual timer (shallow water test case, ne120 mesh). KNL: Knights Landing.

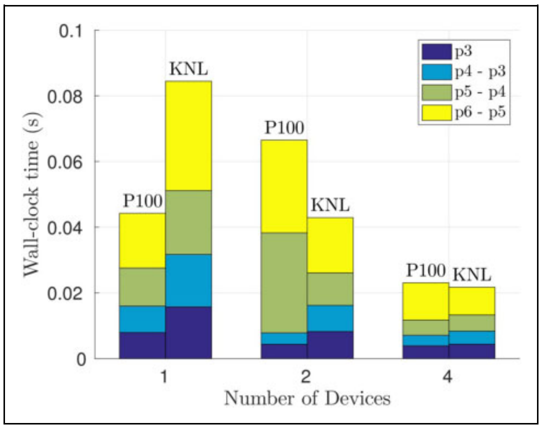


Figure 1B. MPI strong scalability in terms of wall-clock time (averaged over 400 evaluations) for the P100 and KNL architectures on up to four devices using the evaluateFields timer (shallow water test case, ne120 mesh). KNL: Knights Landing.

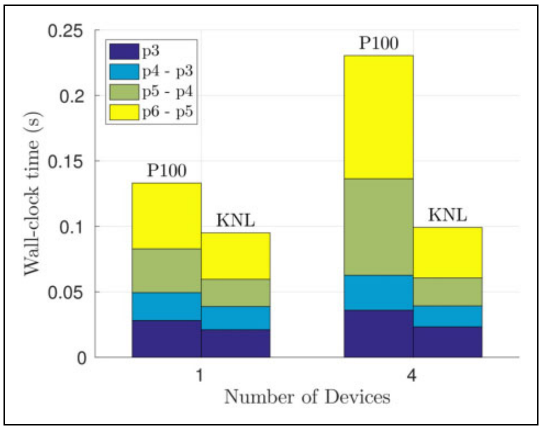


Figure 1C. MPI weak scalability in terms of wall-clock time (averaged over 400 evaluations) for the P100 and KNL architectures on up to four devices using the GlobalResidual timer (shallow water test case). KNL: Knights Landing.

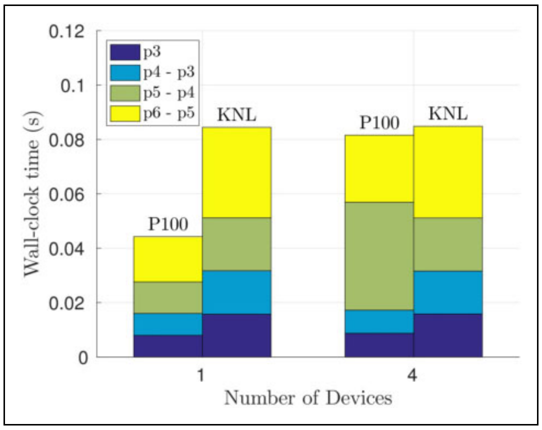


Figure 1D. MPI weak scalability in terms of wall-clock time (averaged over 400 evaluations) for the P100 and KNL architectures on up to four devices using the evaluateFields timer (shallow water test case). KNL: Knights Landing.