Faculty and Researchers | Faculty and Researchers' Publications

2019

Strong scaling for numerical weather prediction at petascale with the atmospheric model NUMA

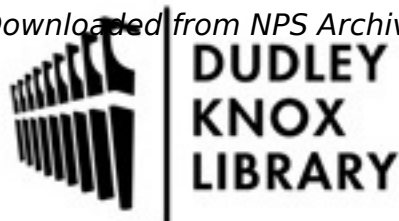Müller, Andreas; Kopera, Michal A.; Marras, Simone; Wilcox, Lucas C.; Issac, Tobin; Giraldo, Francis X.

SAGE

Müller, Andreas, et al. "Strong scaling for numerical weather prediction at petascale with the atmospheric model NUMA." The International Journal of High Performance Computing Applications 33.2 (2019): 411-426.

http://hdl.handle.net/10945/61977

Research Paper

Strong scaling for numerical weather prediction at petascale with the atmospheric model NUMA

**Andreas Müller[1,2], Michal A Kopera[1], Simone Marras[3],
Lucas C Wilcox[1], Tobin Isaac[4] and Francis X Giraldo[1]**

Abstract

Numerical weather prediction (NWP) has proven to be computationally challenging due to its inherent multiscale nature. Currently, the highest resolution global NWP models use a horizontal resolution of 9 km. At this resolution, many important processes in the atmosphere are not resolved. Needless to say, this introduces errors. In order to increase the resolution of NWP models, highly scalable atmospheric models are needed. The non-hydrostatic unified model of the atmosphere (NUMA), developed by the authors at the Naval Postgraduate School, was designed to achieve this purpose. NUMA is used by the Naval Research Laboratory, Monterey as the engine inside its next generation weather prediction system NEPTUNE. NUMA solves the fully compressible Navier–Stokes equations by means of high-order Galerkin methods (both spectral element as well as discontinuous Galerkin methods can be used). NUMA is capable of running middle and upper atmosphere simulations since it does not make use of the shallow-atmosphere approximation. This article presents the performance analysis and optimization of the spectral element version of NUMA. The performance at different optimization stages is analyzed using a theoretical performance model as well as measurements via hardware counters. Machine-independent optimization is compared to machine-specific optimization using Blue Gene (BG)/Q vector intrinsics. The best portable version of the main computations was found to be about two times slower than the best non-portable version. By using vector intrinsics, the main computations reach 1.2 PFlops on the entire IBM Blue Gene supercomputer Mira (12% of the theoretical peak performance). The article also presents scalability studies for two idealized test cases that are relevant for NWP applications. The atmospheric model NUMA delivers an excellent strong scaling efficiency of 99% on the entire supercomputer Mira using a mesh with 1.8 billion grid points. This allows running a global forecast of a baroclinic wave test case at a 3-km uniform horizontal resolution and double precision within the time frame required for operational weather prediction.

1. Introduction

Numerical weather prediction (NWP) has always been considered one of the important computationally intensive uses of supercomputers. Nevertheless, there is a big gap between the size of the available supercomputers and the amount of computing power that is used for operational weather prediction. State-of-the-art operational deterministic weather forecasts typically use about 1000 processors (Bauer et al., 2015) with a global resolution of 9 km, whereas the biggest available supercomputers offer more than 1 million processors allowing more than 10^{15} floating point operations in 1 s (petascale). One of the reasons for this discrepancy is that many weather models do not scale to this large number of

[1] Department of Applied Mathematics, Naval Postgraduate School, Monterey, CA, USA
[2] European Center for Medium-Range Weather Forecasts, Reading, UK
[3] Department of Mechanical and Industrial Engineering, New Jersey Institute of Technology, Newark, NJ, USA
[4] Georgia Institute of Technology, School of Computational Science and Engineering, Atlanta, GA, USA

Corresponding author:
Andreas Müller, Department of Applied Mathematics, Naval Postgraduate School, Monterey, CA 93943, USA.
Email: amueller@anmr.de

processors and therefore are not able to make good use of these big machines. The National Oceanic and Atmospheric Administration (NOAA) has initiated the High-Impact Weather Prediction Project (HIWPP) with the goal to reach NWP at a resolution of 3 km by the year 2020 (Schneider, 2014). Being able to improve the resolution by almost one order of magnitude will allow resolving some of the atmospheric processes explicitly that are currently only described by heuristic approximations (parameterizations). For this reason, it is expected that such a significant improvement in the resolution of weather prediction models will reduce the error and improve the accuracy of weather forecast significantly.

In this article, we show that the non-hydrostatic unified model of the atmosphere (NUMA) (Giraldo and Restelli, 2008; Giraldo et al., 2013; Kelly and Giraldo, 2012) is capable of simulating a global baroclinic wave test case within the time frame required for operational weather prediction at 3 km resolution using a uniform global mesh with 31 layers in the vertical direction. We achieve this performance with double precision and without making use of the commonly used shallow atmosphere approximation; in fact, NUMA (within the NEPTUNE modeling system) was the only model studied by NOAA in the HIWPP study that did not use the shallow atmosphere approximation (Whitaker, 2015). Using the deep atmosphere equations instead allows our simulations to include middle and upper atmospheric processes which are important for long-term (seasonal) weather and climate predictions. Furthermore, our code does not assume any special alignment of its mesh with the horizontal and vertical direction which allows the simulation of arbitrary steep terrain. It was possible to reach the desired resolution thanks to a careful optimization of the code and an excellent strong scaling efficiency of 99% on the entire 3.14 million threads of the supercomputer Mira using a mesh with 1.8 billion grid points. To the best of our knowledge, this article not only presents the first atmospheric model that is capable of reaching the envisioned resolution within operational requirements but also presents the first published strong scalability study up to petascale of fully compressible three-dimensional (3-D) global simulations.

Related work

There has been a lot of work on code optimization and strong scaling efficiency studies before. All weather prediction models have a long history of code optimizations (Wedi et al., 2015). However, not many atmospheric models have shown the capability of making efficient use of entire supercomputers at petascale. Yang et al. (2016) present strong scaling efficiency of 67% at 2 km resolution on the entire 10 M cores of the Sunway TaihuLight supercomputer with a sustained performance of 7.95 PFlops. Johnsen et al. (2013) present strong scaling efficiency of about 65% at almost 300 TFlops sustained performance on the Cray machine Blue Waters for a hurricane simulation using 4 billion grid points. Wyszogrodzki et al. (2012) present strong scaling up to 10^5 cores on the Hopper II system including full parameterizations for moisture using up to 84 million grid points. Strong scaling for the Community Atmosphere Spectral Element (CAM-SE) model using a spectral element method similar to the one utilized in NUMA is presented by Dennis et al. (2012). CAM-SE is targeted at climate prediction. Dennis et al. report strong scaling up to 172,800 cores on the Cray system JaguarPF using 81 million grid points. Wedi et al. (2015) report scaling with the Integrated Forecasting System of the European Center for Medium-Range Weather Forecasts (ECMWF) on 200 k CPUs on Titan and Michalakes et al. (2015) report scaling of the atmospheric models MPAS, FV3, and NEPTUNE up to about 100 k cores on the Cray supercomputer Edison. Another example of strong scaling studies at much smaller scale is given by Nair et al. (2009) and an example of work on shallow water equations is given by Xue et al. (2014).

Outside the atmospheric community, there are a large number of publications on code optimization. It is beyond the scope of this article to give a complete overview of all publications on code optimization. Some outstanding examples for the Blue Gene (BG) architecture are given by Rossinelli et al. (2013) and Rudi et al. (2015). An important example for incompressible Navier–Stokes equations is given by Tufo and Fischer (1999).

Our article is organized as follows. The numerical methods are introduced in Section 2. Section 3 presents the two test cases considered for the studies of this article, Section 4 describes the mesh generation with the p4est library, and Section 5 gives some important technical details about the supercomputer Mira that we use for this work. A theoretical performance model is presented in Section 6. The code optimizations are presented in Section 7 and scalability results are shown in Section 8.

2. Numerical methods

NUMA solves the compressible Navier–Stokes equations which can be written as (see e.g. Müller et al., 2013)

$$\frac{\partial \boldsymbol{q}}{\partial t} + \boldsymbol{\nabla} \cdot \mathbf{F}(\boldsymbol{q}) = \boldsymbol{S}(\boldsymbol{q}) \qquad (1)$$

where t is the time, $\boldsymbol{q} \equiv \boldsymbol{q}(t,x,y,z) = (\rho, \rho\boldsymbol{u}^T, \Theta)^T$ is a time-dependent vector field containing the so-called prognostic variables (air density ρ, 3-D wind speed, $\boldsymbol{u}$, and potential temperature, θ), and x, y, and z are the coordinates in the three space dimensions. The nonlinear operator $\mathbf{F}$ denotes the flux tensor and $\boldsymbol{S}$ is a source function.

In the following subsections, we illustrate the main steps of the numerical solution of these equations using a spectral element method. In the last subsection of this section, we describe two different numerical possibilities to organize the data of our simulation. The two methods are identified as continuous Galerkin (CG) *storage* and discontinuous Galerkin (DG) *storage* (Abdi and Giraldo, 2016).
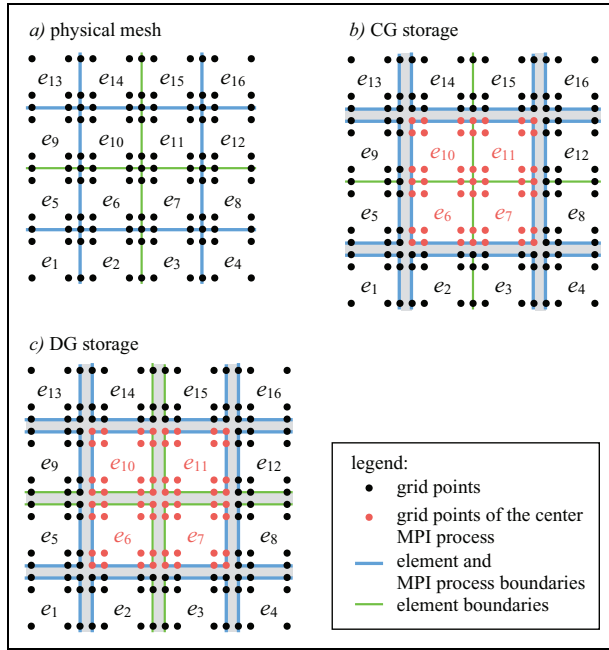
Figure I. Illustration of a sample 2-D cross section containing 16 elements $e_1, \ldots, e_{16}$ of our mesh (a) and two possible approaches to store the data: CG storage (b) and DG storage (c). For illustration purposes, we assume that elements e_6, e_7, e_{10}, and e_{11} are computed in the same MPI process, while the other elements are computed on different MPI processes. The elements and grid points of the MPI process in the center of the figure are highlighted in red. The process boundaries are shown by the blue lines, while element boundaries that are not process boundaries are shown by green lines. The square shape of the elements is used to keep this illustration simple. The elements can have arbitrarily curved faces. 2-D: two-dimensional; CG: continuous Galerkin; DG: discontinuous Galerkin.

2.1. Spatial discretization

In order to discretize equation (1), we introduce a mesh of elements. An example for a two-dimensional (2-D) cross section of our mesh is illustrated in Figure 1(a). Inside each element, we approximate the solution $\boldsymbol{q}$ in each dimension by polynomials of order p. We indicate with $\boldsymbol{q}_N$ the approximation of $\boldsymbol{q}$. To define these polynomials, we introduce a mesh of $p + 1$ grid points inside each element e and in each direction. We use Gauss–Lobatto points for the numerical integration and denote the coordinates of these grid points by (x_i, y_j, z_k). We restrict the rest of this section to the special case of $p = 3$ because this case is most efficient for vectorization on the supercomputer Mira (see Section 7). We define our polynomials in a reference element over the interval $[-1, 1]$ in each direction, with coordinates (ξ, η, ζ) and grid points (ξ_i, η_j, ζ_k). We denote the Jacobian determinant of the coordinate transformation between the reference element and physical element e at grid point (x_i, y_j, z_k) with $J^e_{i,j,k}$. The numerical solution $\boldsymbol{q}_N$ inside element e is given by

$$\boldsymbol{q}_N(t, \xi, \eta, \zeta) = \sum_{l,m,n=1}^{4} \boldsymbol{q}_{l,m,n}(t)\psi_n(\zeta)\psi_m(\eta)\psi_l(\xi) \quad (2)$$

with $\boldsymbol{q}_{l,m,n}(t) = \boldsymbol{q}(t, \xi_l, \eta_m, \zeta_n)$ and the one-dimensional (1-D) Lagrange basis polynomials ψ_i are given by

$$\psi_i(\xi) = \prod_{\substack{m=1,\ldots,p+1 \\ m \neq i}} \frac{\xi - \xi_m}{\xi_i - \xi_m}, \quad \forall i = 1, \ldots, p+1 \quad (3)$$

where Ω_e is the domain of element e.

The goal is now to insert equation (2) into equation (1) and solve it for the values of $\boldsymbol{q}_N$ at the grid points $\boldsymbol{q}_{i,j,k}$. In this article, we use a spectral element method. From now on, the spectral element method will be often referred to with the acronym CG. We multiply equation (1) by $\psi_i(\xi)\psi_j(\eta)\psi_k(\zeta)$ (including $J^e_{i,j,k}$) and integrate over the entire domain Ω. By using Gauss–Lobatto quadrature with quadrature weights w_i, we obtain the following equation:

$$M_{i,j,k}\frac{d\boldsymbol{q}_{i,j,k}}{dt} = -\sum_e J^e_{i,j,k}w_{i,j,k}(\boldsymbol{\nabla}\cdot\mathbf{F}_N - \boldsymbol{S}_N) \quad (4)$$

where $w_{l,m,n} = w_l w_m w_n$, $\mathbf{F}_N = \mathbf{F}(\boldsymbol{q}_N)$, $\boldsymbol{S}_N = \boldsymbol{S}(\boldsymbol{q}_N)$, and $M_{i,j,k} = \sum_e w_{i,j,k}J^e_{i,j,k}$ are the entries of the diagonal mass matrix.

The spatial derivatives in the divergence of the flux tensor $\boldsymbol{\nabla}\cdot\mathbf{F}_N$ are given with equation (2) by

$$
\begin{aligned}
\left.\frac{\partial \boldsymbol{q}_N}{\partial x}\right|_{i,j,k} &= \sum_{m=1}^{4}\boldsymbol{q}_{m,j,k}\left.\frac{d\psi_m(\xi)}{d\xi}\right|_{\xi_i}\left.\frac{\partial\xi}{\partial x}\right|_{i,j,k} \\
&+ \sum_{m=1}^{4}\boldsymbol{q}_{i,m,k}\left.\frac{d\psi_m(\eta)}{d\eta}\right|_{\eta_i}\left.\frac{\partial\xi}{\partial x}\right|_{i,j,k} \\
&+ \sum_{m=1}^{4}\boldsymbol{q}_{i,j,m}\left.\frac{d\psi_m(\zeta)}{d\zeta}\right|_{\zeta_i}\left.\frac{\partial\zeta}{\partial x}\right|_{i,j,k} \quad (5)
\end{aligned}
$$

The products of the values of $\boldsymbol{q}_N$ at the grid points and the derivatives of our basis functions are essentially 4×4 matrix–matrix multiplications. All of the derivatives in equation (5) are computed once at the beginning of the simulation.

The basis functions ψ_i vanish outside of element e. For this reason, the sum over all elements in equation (4) reduces to a single element for the interior grid points. For the grid points along the edges, we need to sum over all neighboring elements weighted by the volume of the elements (this summation is called direct stiffness summation).

2.2. Time discretization

In order to keep communication between different processors simple, we use explicit time integration in the horizontal direction. If the vertical resolution is of the same order of magnitude of the horizontal resolution, we use a fully explicit Runge–Kutta scheme with five stages and third order. In each of those five stages, we need to evaluate the right-hand side of equation (4) and communicate the values of the grid points along the process boundaries (blue lines in Figure 1(a)).

If the vertical resolution is much finer than the horizontal resolution, we organize our mesh in such a way that all

the vertical columns of our elements are always computed in the same MPI process. This allows us to make implicit corrections along the vertical columns after an explicit step of a leap-frog scheme. We call this approach 1D-implicit–explicit (IMEX) (Giraldo et al., 2013). The time step is restricted in this case by the sound waves in the explicitly treated horizontal direction. The time step has to be small enough to make sure that horizontal sound waves cannot travel further than the shortest distance between the nodal points in Figure 1(a).

2.3. Filter

Spectral element methods require stabilization (Marras et al., 2015a). NUMA allows for the use of different stabilization schemes that range from subgrid-scale models (Marras et al., 2015b) to low-pass filters (Boyd–Vandeven). In this article, we use a Boyd–Vandeven filter. The main idea of this filter is to perform a spectral transformation of the nodal values $q_{i,j,k}$ and to dampen the highest order modes. From a computational point of view, this results in multiplying all the values $q_{i,j,k}$ of the element e with a filter matrix. Each time the filter is applied, the new filtered values need to be communicated between neighboring MPI processes. In the future, we will move to Laplacian-based stabilization methods which do not require an additional communication step (see Giraldo et al., 2013).

2.4. CG and DG storage

Each MPI process needs to own a copy of values at the grid points along the process boundaries. This is illustrated in Figure 1(b) by drawing a gray gap between the different processes. There is only one copy in memory for interior grid points even if they are located on a boundary between different elements (green lines in Figure 1(b)). We call this approach CG storage because it requires the solution to be continuous and works only for CG methods.

Another possibility to organize the data is to always store the values along element boundaries for each neighboring element separately (Figure 1(c)). We call this approach DG storage because it allows the use of DG methods. We compare the performance of these two approaches in a simple performance model in Section 6. For more details about the efficient implementation of CG and DG methods, we refer to Abdi and Giraldo (2016).

3. Test cases

Two test cases are considered in this article. One test case is the baroclinic wave instability problem on the sphere by Jablonowski and Williamson (2006). This problem is classically used to test the dynamical core of global circulation models (also in HIWPP, cf. Schneider, 2014). It is initialized by a zonal band of high wind speed in the midlatitudes (jet stream). A Gaussian perturbation of the zonal wind is added. This perturbation leads to wave-like meridional perturbations of the jet stream (Figure 2). After some time, the flow pattern looks similar to the polar front jet stream of the real atmosphere. The spherical mesh that is used in this article is based on a cubed-sphere mesh. The implementation of NUMA recognizes the mesh only through the curved elements generated by p4est in Cartesian coordinates. NUMA could easily handle any other spherical mesh without any impact on the performance measurements in this article.

The other test case is a 3-D rising thermal bubble in a box of 1000 m in each direction. This test case is initialized with a temperature perturbation in a neutrally stratified atmosphere. The precise definition and analysis of the full simulation is reported by Kelly and Giraldo (2012).

Both test cases are important for NWP applications. Operational weather prediction needs to cover the global circulation on the entire Earth like in the baroclinic wave test case. In order to use a higher resolution, for specific localized features of the atmosphere like hurricanes, one needs to run the simulation in limited area mode like in the 3-D rising thermal bubble test case.

4. Mesh generation and load balancing

The data structures and algorithms for parallel mesh generation, partitioning, and load balancing used in our simulations were provided by the p4est library. The p4est library has been used for efficient and scalable parallel adaptive mesh refinement for 2-D advection on the sphere (Burstedde et al., 2014), in other applications such as mantle convection and seismic wave propagation (Burstedde et al., 2010), and as a backend for the deal.II finite element library (Bangerth et al., 2015). Our present article is the first time that p4est is used for full 3-D atmospheric simulations. This article does not make use of the adaptive mesh refinement but it uses the p4est library to generate the grid for the simulations. This approach makes it possible to easily add adaptive mesh refinement in future work.

The p4est library represents 2-D and 3-D domains via a two-level structure, with a macro mesh and a micro mesh. The macro mesh is a conformal quadrilateral or hexahedral mesh, which is encoded as an unstructured mesh that is reproduced on each MPI process. Each element in the macro mesh is then treated as the root of a partitioned quadtree or octree, which recursively refines the macro element isotropically to create a micro mesh. The tree structure is represented in memory as a list of the leaves of the tree, ordered by the Morton curve (also known as the z-curve). This ordering induces a space filling curve that visits the centers of the leaves; while this curve is not a continuous space filling curve, it has many of their nice properties. One important property is that partitioning a domain by dividing the Morton curve into continuous segments creates subdomains that are fairly compact, with low surface-to-volume ratios (Hungershöfer and Wierum, 2002). This means that partitioning by this method keeps
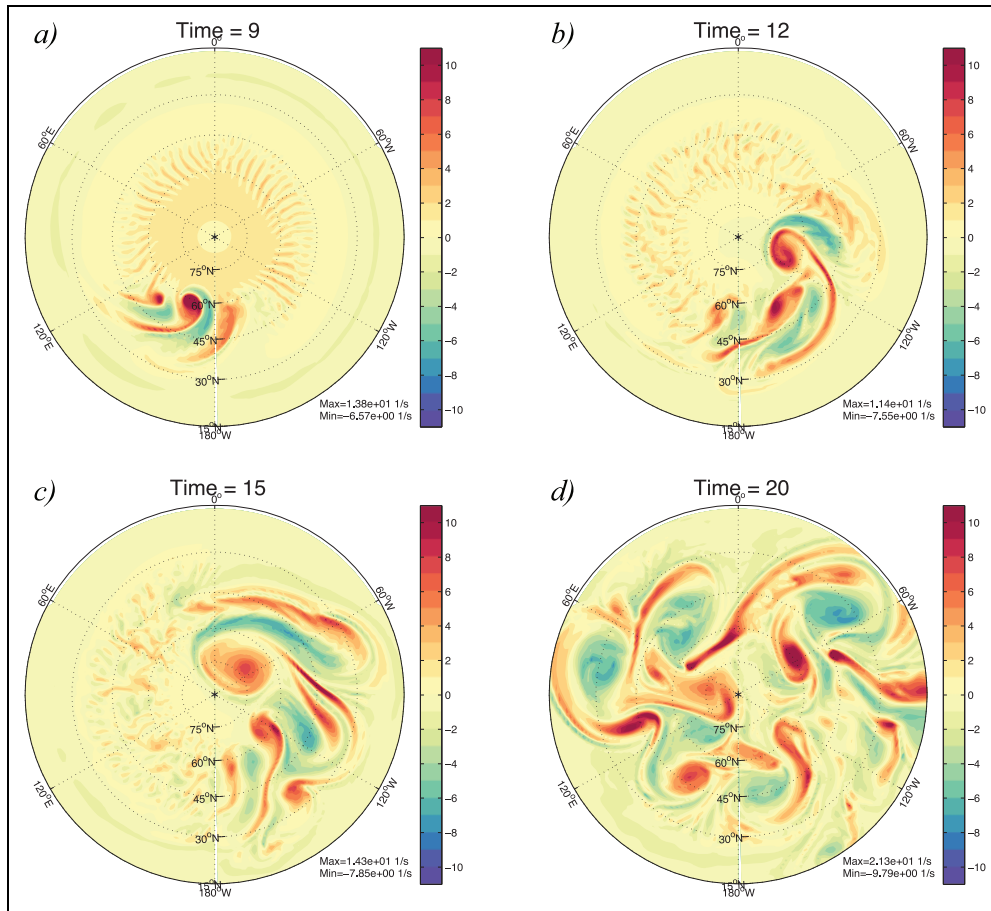
Figure 2. Baroclinic instability simulation at 9, 12, 15, and 20 days using a horizontal resolution of 50 km and a vertical resolution of 1 km. Shown is a top-view looking down onto the northern hemisphere. Plotted is the vertical component of the vorticity $\omega = \nabla \times u$ in s^{-1}. The simulation uses a dynamic subgrid-scale eddy viscosity model for stabilization purposes. We refer to Marras et al. (2015a, 2015b) for more details about this stabilization method and to Jablonowski and Williamson (2006) for more details about the test case. (a) Time = 9. (b) Time = 12. (c) Time = 15. (d) Time = 20.

the intra-process communication during simulations low. A full description of p4est's forest-of-quadtree and forest-of-octree data structures and algorithms can be found in the study by Burstedde et al. (2011).

When used in its raw form, the neighborhood information of an element in the micro mesh (i.e. which elements are adjacent) takes $\log(N_i)$ time to calculate, where N_i is the number of micro mesh elements in the ith partition. To avoid incurring this cost during each time step, the adjacency information for all elements in the ith partition can be converted into a lookup table, much like an unstructured mesh. An efficient approach to creating this information, which can also be used to enumerate the nodes for high-order CG finite elements, is described by Isaac et al. (2015a).

The numerical methods in our simulations involve 3-D computations, but the vertical direction is treated differently from the other two: its grid resolution requirements are different, and achieving efficiency in the IMEX time evolution scheme (and other calculations that are performed only in the vertical direction) requires that vertical columns of elements and degrees of freedom be contiguous

in memory. A forest-of-octrees approach would be ill-suited for these constraints. First, octree refinement is isotropic: the aspect ratio of a macro element is inherited by all of the micro elements created by refinement. This means that the relationship between horizontal and vertical resolution would have to be respected at the macro mesh level, increasing the macro mesh's complexity. Second, the 3-D Morton curve does not respect the need to keep vertical columns contiguous: elements in a column would be separated in memory and, without care, would even be placed in separate partitions.

For these reasons, we want to use a forest-of-quadtrees approach to generate and partition vertical columns but to handle the elements within each column using a different approach. An extension to the p4est library, which was first used in the context of ice sheet modeling (Isaac et al., 2015b), provides the necessary data structures and interface. This extension is a set of "p6est" data types and functions (so named because it uses aspects of the 2-D "p4est" interface and the 3-D "p8est" interface). Essentially, it treats each vertical column as a list, from the bottom to top, of the elements created by recursive
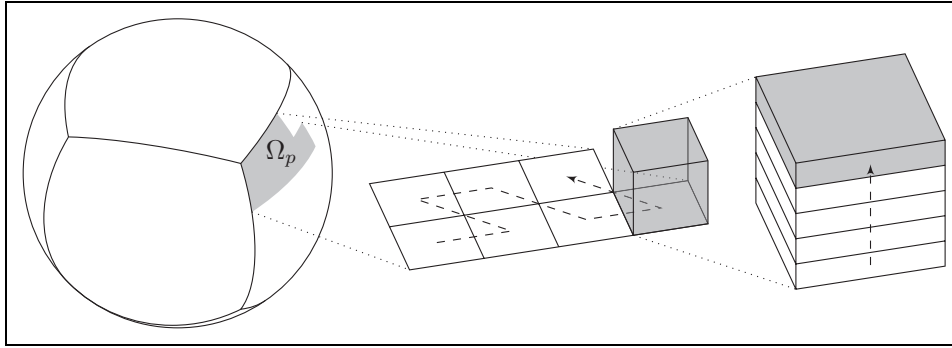
Figure 3. An illustration of meshing with the p6est extension of the p4est library. A macro mesh represents the cubed-sphere domain (left), division of the Morton curve creates the partitions for each MPI process, the columns in the *i*th partition Ω_i are ordered by a 2-D Morton curve (middle), and each vertical column is stored as a list of "layers" from the bottom to the top (right). 2-D: two-dimensional.

bisection of the full column and uses the existing 2-D p4est routines to manage the partitioning of columns and inter-column interactions. A more complete description of this approach can be found in the study by Isaac (2015: Ch. 2). The p6est mesh format is illustrated in Figure 3.

As the elements within a column are defined by recursive bisection, p6est was designed for meshes with 2^k elements per column for some k. Because NUMA must work with meshes which do not have this property, the p6est format was extended for this work to support an arbitrary number of elements per column. The p4est library allows us to generate large meshes very efficiently. The runtime spent on generating the mesh is less than 20 s even in the case of 43 billion grid points and 3 million hardware threads.

It should be noted that, although mesh adaptivity is not used in this work, the p6est data structures support bimodal local mesh adaptivity: elements may be independently refined in the vertical direction, and each column can be independently refined into four smaller columns.

5. Blue Gene/Q Mira

The simulations presented in this article were performed on the supercomputer Mira of the Argonne National Laboratory. Mira is an IBM BG/Q system offering 49,152 computational nodes. Each of these nodes has 16 cores resulting in a total number of 786,432 cores. Each core has a quad floating point unit. This permits running up to four MPI processes or up to four OpenMP (OMP) threads on each core. The maximum total number of hardware threads is therefore 3,145,728.

Important for the performance of our code is the memory architecture of Mira. Each computational node has 16 GB of random access memory (RAM). The processor receives its data from RAM through two levels of cache. Each core has its own level 1 (L1) cache of 16 KB, while the L2 cache of Mira is shared among all 16 processors of the computational node and has a size of 32 MB.

In addition to these two levels of cache, each core has an L1 cache prefetch (L1P) of 4 KB (Chung et al., 2012). Whenever the data needed for the simulation cannot be

found in the L1 cache (L1 cache miss), the computer checks if this data are available in L1P. This is always done for an entire cache line of 128 bytes (16 double precision floating point numbers). If the requested cache line is not in L1P, it goes on and looks in L2 cache and eventually in the RAM for this data. The prefetcher of the BG/Q keeps track of previous cache misses. The stream prefetcher of the BG/Q establishes a stream if consecutive cache lines are requested, that is, L1P loads the next cache lines even when no cache miss occurs. If this consecutive data are actually needed by the code, this can save a lot of runtime. However, if the data are not used consecutively by the code, the prefetcher will read a lot of data from L2 cache without ever using it. This can produce a huge number of unnecessary cache misses.

Important for our optimizations is also the vector unit. The registers of Mira have a length of 256 bits offering space for four double precision floating point numbers. This allows performing up to eight double precision fused multiply-add floating point operations per core within the same clock cycle. Each core can perform one floating point instruction and one integer instruction per clock cycle. Load and store instructions take each one cycle of the integer unit.

6. Performance model

Making a theoretical performance model allows us to estimate the expected performance and to compare different numerical methods without fully optimizing them in our code. We created the performance model by counting all floating point operations as well as memory read and write accesses throughout our entire code. The optimal runtime is then computed by using the roofline model for the arithmetic intensity of the counted values. The results of this theoretical model are presented in Tables 1 to 3. We do this for three possible implementations: our CG version avoids having multiple copies of data in memory wherever possible (as in Figure 1(b)). The only variables that still require multiple values at the same physical locations are the metric terms in equation (5). The CG/DG version uses the same approach like the CG version for dynamically

Table 1. Theoretical amount of floating point operations and memory traffic for the rising thermal bubble test case at polynomial degree $p = 3$.[a]

	CG	CG/DG	DG
GFlops per node	3007.00	3007.00	4023.19
Read traffic in GB	2129.42	2537.28	3489.66
Write traffic in GB	661.83	688.34	1168.69
Arithmetic intensity in flops/bytes	1.08	0.93	0.86
Optimal runtime in seconds	97.94	113.18	163.45
Percentage of theoretical peak of processor	14.99	12.97	12.02

CG: continuous Galerkin; DG: discontinuous Galerkin; BG: Blue Gene.
[a]The effect of random memory access for CG is not taken into account in these results. All the numbers given in this table are summed over the entire simulation on 768 BG/Q nodes. The simulations use an effective resolution of 1.30 m in *x*- and *y*-direction and an effective resolution of 0.89 m in *z*-direction which leads to a total number of 7.4×10^8 grid points. This corresponds to 60 k grid points per BG/Q core which is the workload we aim at using on the entire machine Mira. All simulations use a Courant number of 0.7 in the vertical direction and are run for a model time of 1 s. This corresponds for $p = 3$ to 690 time steps. The columns "CG" and "DG" give the results for CG and DG storage, respectively. The column "CG/DG" uses DG storage for data that does not change throughout the simulation and CG storage for the rest. The number of floating point operations is adjusted according to real measurements to take the effects of compiler optimization into account. The optimal runtime is based on the peak memory bandwidth of 28.5 GB/s according to the STREAM benchmark (Morozov et al., 2013). This runtime leads to the given percentage of the theoretical peak performance of the processor. This demonstrates that all of our simulations are expected to be memory bound.

Table 2. Like Table 1 but including an estimate for the effect of random memory access due to CG storage if the previously computed elements are no longer available in cache.

	CG	CG/DG	DG
GFlops per node	3007.00	3007.00	4023.19
Read traffic in GB	3483.05	3138.46	3682.77
Write traffic in GB	853.44	879.95	1360.30
Arithmetic intensity in flops/bytes	0.69	0.75	0.80
Optimal runtime in seconds	152.16	141.00	176.95
Percentage of theoretical peak of processor	9.65	10.41	11.10

CG: continuous Galerkin; DG: discontinuous Galerkin.

Table 3. Theoretical amount of floating point operations and memory traffic for the baroclinic instability test case.[a]

	CG	CG/DG	DG
GFlops per node	61.96	61.96	83.00
Read traffic in GB	58.55	62.18	94.22
Write traffic in GB	22.48	22.48	36.69
Arithmetic intensity in flops/bytes	0.76	0.73	0.63
Optimal runtime in seconds	2.84	2.97	4.59
Percentage of theoretical peak of processor	10.64	10.18	8.82

CG: continuous Galerkin; DG: discontinuous Galerkin.
[a]These results are calculated for 972 BG/Q nodes. The simulations use an effective horizontal resolution of 21.3 km and an effective vertical resolution of 1.0 km which leads to a total number of 4.4×10^7 grid points. This corresponds to 2821 grid points per BG/Q core which is the workload we aim at using on the entire machine Mira. All simulations use a Courant number of 0.4 in the horizontal direction and 6.4 in the vertical direction and are run for a model time of 4 h. This corresponds for $p = 3$ to 947 time steps.

The optimal runtime in Table 1 is the runtime that we expect if we manage to reach 100% of the theoretical peak memory bandwidth as given by the STREAM benchmark in the study by Morozov et al. (2013). The percentage of the theoretical peak performance of the processor is also given in Table 1 for this optimal runtime. Our CG-only version gives us the best performance in all of these results.

CG storage has a big disadvantage that we have not included in Table 1: we have to perform the computations of equation (5) on a per element basis while data stored in CG storage are not arranged on a per element basis. This leads to noncontiguous memory access that appears to be random. Having random memory access will not affect the results in Table 1 if most of the data is already in L2 cache. Therefore, the results of Table 1 will still be correct if the number of elements per compute core is small. Our goal for the rising thermal bubble problem is to use a very high grid resolution which produces much more data than we can fit into L2 cache. In this case, we expect to get a much better estimate for the performance of our code by assuming that none of the data is already in L2 cache and by counting the full cache lines in those cases where CG storage requires us to access only a small portion of that cache line. The results including this estimate for the effect of random memory access are shown in Table 2. Including the effect of random memory access does not change the number of floating point operations but it leads to a significant increase in memory traffic. This makes DG storage now the version with the highest arithmetic intensity and therefore the best percentage of the theoretical peak performance of the processes. The overall runtime of the entire simulation is still slower for DG storage due to the increased number of floating point operations. The winner in terms of overall runtime is our CG/DG version.

All of the results presented so far have been obtained for $p = 3$. Our performance model allows us to compare the optimal performance of different polynomial orders p for

changing variables but allows additional copies of the data at element boundaries (Figure 1(c)) for data related to the reference atmosphere that does not change throughout the simulation. The DG version allows multiple copies of the data along element interfaces (Figure 1(c)) for all variables.

Table 1 shows the expected number of floating point operations and total memory read and write traffic throughout the entire simulation for the rising thermal bubble test case. The additional copies of data along the element interfaces in DG storage lead to a significant increase in the number of floating point operations as well as memory traffic and to a significant reduction of arithmetic intensity. Our results for arithmetic intensity show that all of our cases are memory bound (see roofline plots in Figure 5).
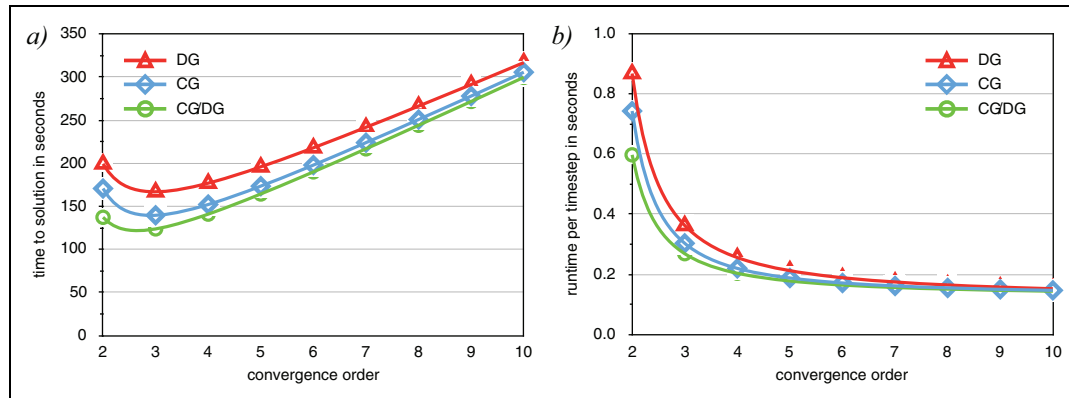
Figure 4. Time to solution (a) and runtime per time step (b) as a function of convergence order $p + 1$ for the CG and DG storage versions and setup shown in Table 2 for the rising thermal bubble test case. All polynomial orders use the same effective resolution (average distance between points) of 1.3 m in the horizontal direction and 0.89 m in the vertical direction and the same Courant number. CG: continuous Galerkin; DG: discontinuous Galerkin.

the three different versions in Table 2. The results are shown in Figure 4. The runtime per time step increases with decreasing order due to the additional copies of data for DG storage (Figure 4(b)). This effect occurs also in our CG version because even in this version we still need to store metric terms in DG storage. The results look very different when the actual time to solution for the entire simulation is considered (Figure 4(a)). The numerical methods considered in this article require reducing the time step with increasing order to keep the explicitly resolved part of the simulation stable (see Section 2.2). This makes low polynomial order much faster than high order. As shown in Figure 4(a), we get the most efficient time to solution for convergence order 3 which corresponds to polynomial order $p = 2$. We still choose $p = 3$ for the remainder of this article because we expect $p = 3$ to be easier to vectorize on the BG/Q architecture.

Table 3 shows the results of our theoretical performance model for the baroclinic instability test case. As described in the introduction, our final goal is to run this test case at 3 km resolution on the entire supercomputer Mira. This setup gives us only 20 elements per hardware thread which makes it possible to fit most of the data in each time step into L2 cache. For this reason, we do not estimate the effect of random memory access in this case. Our CG and CG/DG versions are again significantly faster than our DG version. We will use the CG/DG version for our optimizations in the next section because this version gives us a significant advantage for the rising thermal bubble test case in Table 2.

7. Code optimizations

The goal of this section is to present the optimization steps of our CG storage version of NUMA for $p = 3$. To take the advantage of the reduced amount of data compared to DG storage, we aim at computing as much work as possible on a per grid point basis and try to avoid making computations on a per element basis. The main structure of our code is illustrated in Code example 1. Computations that need to be

computed element-wise are highlighted in blue. Communication is highlighted in red.

We tried to optimize all parts of our code. We found *create_rhs* (the computation of the right-hand side in equation (4), see also Code example 1) to be the only function that contains enough floating point operations to allow significant optimizations.

Tables 5 to 8 show performance measurements for different versions of *create_rhs*. The different versions are explained in Table 4. For the rest of this section, we simply refer to the different versions in these tables. Versions MPI1 to OMP are measured by using the Hardware Performance Monitoring Toolkit. Version T gives theoretical expectations from our performance model as presented in the previous section. As in Table 2, we include here our estimate for random memory access for the rising thermal bubble test case. Version O is another theoretical result that is obtained by making some further optimizations. These optimizations consist of avoiding some unnecessary memory access and minimizing memory access by computing metric terms in equation (5) in each stage of our time integration method. This leads to a significant increase in the number of floating point operations but more importantly it allows a significant reduction in memory traffic. We will try these optimizations in our future work.

All of the measurements in Tables 5 to 8 are taken over the entire simulation. The column "% peak" gives the percentage of the theoretical peak performance of the processor. The column "% max." shows how close this part of the code is to the maximum performance according to the roofline model for the given arithmetic intensity. The column "vect." shows how many percent of all floating point operations are vectorized. The column "FMA" gives the percentage of fused multiply-add operations among all floating point operations. We have not estimated the optimal number of fused multiply-add operations in our performance model. For this reason, we leave the column "fma" empty for the versions T and O. The column "mix"

Code example 1. Pseudocode of the main structure of our code NUMA. The part of the code highlighted in blue needs to be computed element-wise. The rest (black lines) can be computed for each grid point separately. MPI communication is highlighted in red.

```
 1   create the mesh
 2   initialize data q at time t=0
 3   while time < final time
 4     begin loop over time integration stages
 5       compute right hand side of Eq. (4) for current stage (create_rhs)
 6       communicate
 7       DSS and multiply with inverse mass matrix
 8       update q
 9     end
10     compute IMEX corrections for each vertical column of grid points
11     compute filter
12     communicate
13     DSS and multiply with inverse mass matrix
14     increment time
15   end
```

Table 4. Description of the different optimization stages shown in Tables 5 to 8.[a]

Version	Description
MPI1	$p = 3$, one MPI process per core
MPI4	Like version MPI1, four MPI processes per core
PORT	Like version MPI4, optimized for compiler vectorization
BG/Q	Like version MPI4, rewritten using vector intrinsics
OMP	Like version BG/Q, four OMP threads per core
T	Theoretical expectation for version OMP
O	Like T, further optimized (see text for details)

BG: Blue Gene; OMP: OpenMP.
[a]Version PORT is the best portable version that we could achieve. Version BG/Q is the best version that we could get by using BG/Q vector instructions without OMP and version OMP is the best result with OMP. The versions T and O are theoretical results based on counting memory access by hand and estimating the runtime by using the memory bandwidth according to the STREAM benchmark results by Morozov et al. (2013).

Table 5. Performance measurements for *create_rhs* (the computation of the right-hand side in equation (4), see also Code example 1) with the Hardware Performance Monitor Toolkit for the rising thermal bubble test case and the different versions of our code as described in Table 4.[a]

				Flops		
	Runtime (s)	% peak	% max.	vect. (%)	FMA (%)	Mix (%)
MPI1	802.2	1.4	8.8	14.4	80.0	18.0
MPI4	351.4	3.2	25.6	14.4	80.0	18.0
PORT	180.4	6.6	48.8	73.9	81.3	21.2
BG/Q	88.1	13.9	81.6	98.6	75.7	28.9
OMP	96.7	12.6	71.3	98.6	75.7	28.1
T	86.2	14.2	100.0	100.0		50.0
O	41.5	36.7	100.0	100.0		50.0

BG: Blue Gene; FMA: fused multiply-add.
[a]We use the setup as described in Table 1. The column "runtime" shows the overall time spent in *create_rhs* (over all 690 time steps (3450 executions of *create_rhs*)). All measurements were obtained by taking the average values over four arbitrary nodes. We refer to the main text for the explanation of the other columns.

Table 6. Performance measurements for the rising thermal bubble test case like in Table 5 but for the entire time loop.

				Flops		
				vect.	FMA	Mix
	Runtime (s)	% peak	% max.	(%)	(%)	(%)
MPI1	972.7	1.4	14.8	13.6	79.1	17.9
MPI4	454.3	3.0	34.9	13.6	79.1	17.1
PORT	283.6	5.1	56.2	63.1	80.2	18.4
BG/Q	191.7	7.7	82.5	84.0	75.6	20.5
OMP	198.5	7.4	72.7	83.5	75.0	21.6
T	141.0	10.4	100.0	100.0		50.0
O	80.8	21.9	100.0	100.0		50.0

BG: Blue Gene; FMA: fused multiply-add.

Table 7. Performance measurements for *create_rhs* like in Table 5 but for the baroclinic instability test case for the setup explained in the caption of Table 3.

				Flops		
	Runtime (s)	% peak	% max.	vect. (%)	FMA (%)	Mix (%)
MPI1	9.5	1.3	5.9	14.4	80.0	17.9
MPI4	4.0	3.0	24.1	14.4	80.0	17.8
PORT	2.1	6.2	51.3	73.9	81.3	20.7
BG/Q	1.0	13.3	78.1	98.6	75.7	28.1
OMP	1.1	11.7	68.4	98.6	75.7	27.1
T	0.7	18.9	100.0	100.0		50.0
O	0.5	34.3	100.0	100.0		50.0

BG: Blue Gene; FMA: fused multiply-add.

shows the percentage of floating point instructions among all instructions.

We made simulations with a fairly high polynomial order of $p = 6$. These simulations confirmed that order $p = 3$ gives us significantly better time to solution (version MPI1). We have not seen a significant impact on the

Table 8. Performance measurements for the entire time loop like in Table 6 but for the baroclinic instability test case.[a]

				Flops		
	Runtime (s)	% peak	% max.	vect. (%)	FMA (%)	Mix (%)
MPI1	19.3	1.5	10.2	12.6	77.3	20.4
MPI4	9.4	3.4	37.8	12.6	77.3	19.5
PORT	7.5	4.4	49.4	38.0	77.9	20.7
BG/Q	6.4	5.2	55.6	49.1	75.6	21.7
OMP	5.8	5.2	42.1	48.4	74.4	22.1
T	3.0	10.2	100.0	100.0		50.0
O	2.2	15.6	100.0	100.0		50.0

BG: Blue Gene; FMA: fused multiply-add.

[a]Even though only *create_rhs* is changed between the different simulations in this table, the runtime saved for the entire time loop is larger than the time saved in *create_rhs*. The reason for this behavior is that the reduced runtime of *create_rhs* leads to an improved synchronization between different MPI processes which reduces the time spent in MPI communication.

accuracy of our test cases. We use $p = 3$ for all measurements shown in the rest of this article because this order is very well suited for vectorization on Mira (four double precision floating point numbers fit into one register). Another significant speedup was obtained by running four MPI processes per core (version MPI4).

We computed the derivatives of equation (5) in versions MPI1 and MPI4 for each of the five variables of q separately. We tried merging all these derivatives into one matrix for each direction and element. This version was significantly faster but not as fast as version PORT below. We also used the BLAS function dgemm but without any improvement on the performance.

The rest of our optimizations can be categorized into three main topics which we discuss in the following subsections: compiler optimizations, BG/Q vector intrinsics, and OMP. At the end of this section, we give a short description of possible next steps for further optimization.

7.1. Compiler optimization

To improve the performance while retaining portability, we worked first on enabling better optimization through the compiler. We spent some time on finding the best level of compiler optimization for each function of our code. We found a few functions for which level 3 optimization gave us wrong results. This is not surprising because level 3 compiler optimization is not IEEE compliant.

Many of our operations in *create_rhs* looked initially like Code example 2. The operations were computed for each grid point of the element separately which makes it impossible for the compiler to vectorize the code. This explains the very low fraction of vectorized operations in versions MPI1 and MPI4 (column "vect." in Tables 5 to 8).

To improve vectorization, we changed our code in such a way that the operations are performed for the entire element at once (Code example 3). Our measurements for version PORT show that this simple modification leads to a significant improvement of the vectorization.

7.2. BG/Q vector intrinsics

To make even better use of the vector unit, we rewrote our function *create_rhs* by using BG/Q vector intrinsics (Code example 4). This gave us another significant speedup (version BG/Q). This brings *create_rhs* to an excellent level of about 80% of the maximum attainable performance according to the roofline model (column "% max." in Tables 5 and 7 and Figure 5).

Measurements showed that about 13% of all loads hit L2 cache and about 7% hit L1P. These fairly large numbers seem to be due to the random memory access that CG storage produces. Optimal would be if the prefetcher could bring all data into L1 cache before it is needed. We tried different prefetching strategies and handwritten prefetching but could not improve the performance compared to the default strategy. According to our performance model, we still expect CG storage to be significantly faster compared to DG storage even though CG storage makes prefetching very difficult.

7.3. OpenMP

OMP allows a reduction in the number of MPI processes. This leads for CG storage to a reduced amount of work for some parts of the code (namely the black text in Code example 1). However, we need to be very careful to avoid race conditions. In *create_rhs*, race conditions can occur in the summation over all the elements in equation (4). Using OMP atomic statements made our code too slow. The best solution that we could find was to reorder the elements inside each MPI process in such a way that different OMP threads can never compute neighboring elements at the same time. To ensure this, we need to synchronize all threads by using an OMP barrier after each element computation. These barriers slow down *create_rhs* by less than 10% (version OMP). Nevertheless, we obtain in the case of the baroclinic instability a noticeable improvement on the runtime of the entire simulation due to the reduced amount of work for the IMEX corrections in the vertical direction. We obtained the best performance by using 4 OMP threads per MPI process (2, 8, 16, and 64 OMP threads per MPI process were slower).

Comparing our final version OMP with our theoretical results (version T) shows reasonably good agreement (Tables 6 and 8). The number of floating point operations in our theoretical performance model has been tuned to agree with our overall measurements of version OMP in order to take compiler optimizations into account.

7.4. Next steps

The roofline plots in Figure 5 show that our optimizations have given us a massive improvement in the number of

Code example 2. Fortran code similar to a function from the non-optimized initial version of NUMA (used in versions MPI1 and MPI4 in Tables 5 to 8).

```fortran
1  real :: rho, rho_x, rho_y, rho_z, u, v, w, rhs
2  do e=1,num_elem ! loop through all elements
3     do i=1,num_points_e ! loop through all points of the element e
4        ... ! compute derivatives rho_x, rho_y, rho_z
5        rhs = u*rho_x + v*rho_y + w*rho_z + ...
6     end do !i
7  end do !e
```

Code example 3. Like Code example 2 rewritten for improved compiler vectorization (used in version PORT).

```fortran
1  real, dimension(num_points_e) :: rho, rho_x, rho_y, rho_z, u, v, w, rhs
2  do e=1,num_elem ! loop through all elements
3     ... ! compute derivatives like rho_x, rho_y, rho_z
4     rhs = u*rho_x + v*rho_y + w*rho_z + ...
5  end do !e
```

Code example 4. Like Code example 2 rewritten with vector intrinsics (used in versions BG/Q and OMP).

```fortran
1  real, dimension(4,4,4) :: rho, rho_x, rho_y, &
2     rho_z, u, v, w, u_x, v_y, w_z, rhs
3  !IBM* align(32, rho, rho_x, rho_y, rho_z, u, v, w, u_x, v_y, w_z, rhs)
4  ! declare variables representing registers: (each contains four double
        precision floating point numbers)
5  vector(real(8)) vct_rho, vct_rhox, vct_rhoy, vct_rhoz
6  vector(real(8)) vct_u, vct_v, vct_w, vct_rhs
7  if (iand(loc(rho), z'1F') .ne. 0) stop 'rho is not aligned'
8  ... ! check alignment of other variables
9  do e=1,num_elem ! loop through all elements
10    do k=1,4 ! loop over points in z-direction
11       do j=1,4 ! loop over points in y-direction
12          ... ! compute derivatives rho_x, ...
13          ! load always four floating point numbers:
14          vct_u = vec_ld(0, u(1,j,k))
15          vct_v = vec_ld(0, v(1,j,k))
16          vct_w = vec_ld(0, w(1,j,k))
17          vct_rhox = vec_ld(0, rho_x(1,j,k))
18          vct_rhoy = vec_ld(0, rho_y(1,j,k))
19          vct_rhoz = vec_ld(0, rho_z(1,j,k))
20          ! rhs = u*rho_x
21          vct_rhs = vec_mul(vct_u,vct_rhox)
22          ! rhs = rhs + v*rho_y
23          vct_rhs = vec_madd(vct_v,vct_rhoy,vct_rhs)
24          ! rhs = rhs + w*rho_z
25          vct_rhs = vec_madd(vct_w,vct_rhoz,vct_rhs)
26          ! write result from register into cache:
27          call vec_st(vct_rhs, 0, rhs(1,j,k))
28          ...
29       end do !j
30    end do !k
31 end do !e
```

gigaflops per second per node which has brought us very close to the maximum attainable performance at the given arithmetic intensity. Our measurements show also that our final version OMP achieves an excellent level of vectorization (98.6% of all floating point operations are vectorized). The main remaining bottleneck of our optimized code is the low arithmetic intensity and the low instruction mix (Tables 5 to 8). As mentioned in Section 5, each load and store instruction takes one cycle of the integer unit. This explains the low instruction mix and demonstrates that our code could perform a lot more floating point operations while still achieving the same performance. In order to improve performance further, we need to increase arithmetic intensity by reducing the overall amount of memory
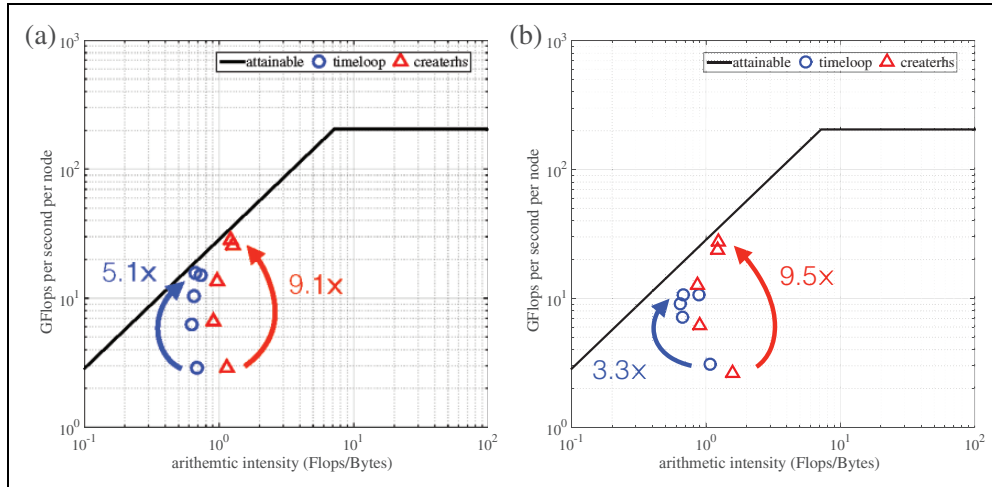
Figure 5. Roofline plot for the different versions shown in Tables 5 and 6 for (a) the rising thermal bubble test case and (b) for the baroclinic instability test case. For the identification of the different versions, we refer to the data in the tables. The arrows illustrate the performance improvement between the slowest and fastest version.

traffic. We have found some unnecessary memory access which we will remove in our future work and we have found that we could reduce memory traffic by recomputing the metric terms in equation (5) in each stage of our time integration method. These optimizations have been included in our performance model and are shown in version O in Tables 5 to 8. According to our performance model, we should be able to achieve another factor 2 of speedup for *create_rhs*. This should bring us to about 35% of the theoretical peak performance of the processor. Recomputing the metric terms in each time integration stage should also improve the very low percentage of floating point instructions among all instructions (column "mix" in Tables 5 to 8). We will try these optimizations in our future work.

8. Strong scaling results

We present in this section strong scaling results up to the entire machine Mira for the baroclinic wave test case (Figures 6, 7, and 9) and the rising thermal bubble test case (Figures 8 and 10). All these results use version OMP from Section 7.

The runtime of the entire simulation for the baroclinic wave test case is shown in Figure 6. The dynamics of a 1-day forecast needs to be finished within less than about 4.5 min runtime (more than 320 model days per wall clock day). We reach this goal on the entire machine Mira for our 3.0 km uniform horizontal resolution simulation of the baroclinic wave test case which takes 4.15 min runtime per 1 day forecast (346.6 model days per wall clock day).

The strong scaling efficiency of the simulations in Figure 6 is shown in Figure 7 for the different parts of the code. The entire code reaches a strong scaling efficiency of 99.1% on the entire machine Mira. The parts *create_rhs* and filter show a scaling efficiency of more than 100%. This is not surprising because the problem fits better into L2 cache with increasing number of threads and at the same
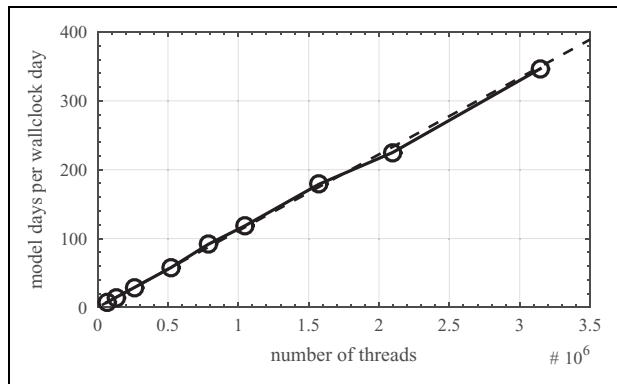

Figure 6. Strong scaling for the baroclinic wave test case with polynomial order $p = 3$ using 1024 elements per cubed-sphere edge and 10 elements in the vertical direction (31 degrees of freedom). This corresponds to a global effective resolution of 3.0 km and a total number of about 1.8 billion grid points. The dashed line shows ideal strong scaling over a base run on 6.5×10^4 threads.
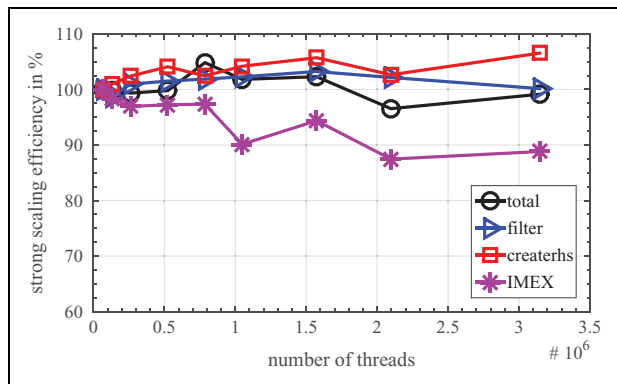

Figure 7. Strong scaling efficiency over base run on 6.5×10^4 threads for the simulations shown in Figure 6 (baroclinic instability).

time the time spent in our OMP barriers is decreasing. The IMEX part gives us the lowest scaling efficiency. We still need to understand the reason for this behavior.
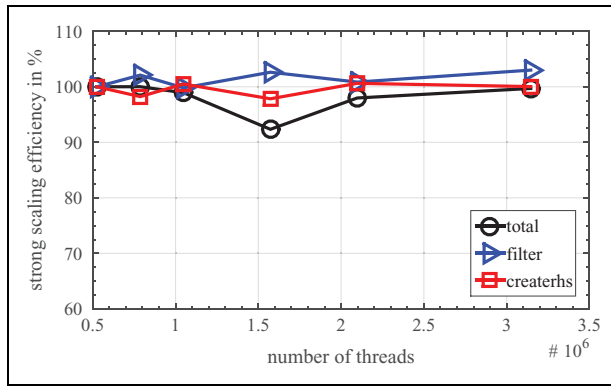
Figure 8. Strong scaling efficiency over base run on 7.7×10^5 threads for the rising thermal bubble test case using $1024 \times 1024 \times 1536$ elements which corresponds to about 43 billion grid points.
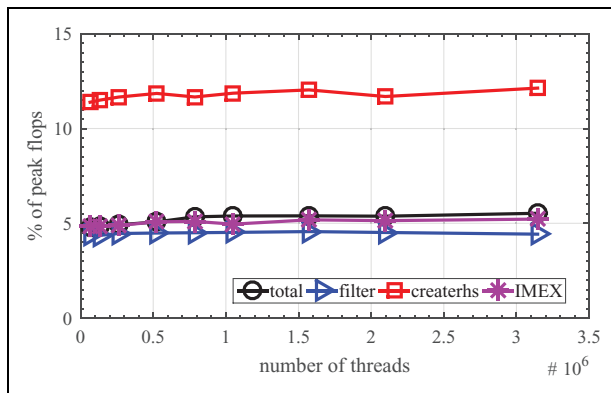

Figure 9. Percentage of theoretical peak performance in terms of floating point operations for the baroclinic wave test case like in Figure 6.
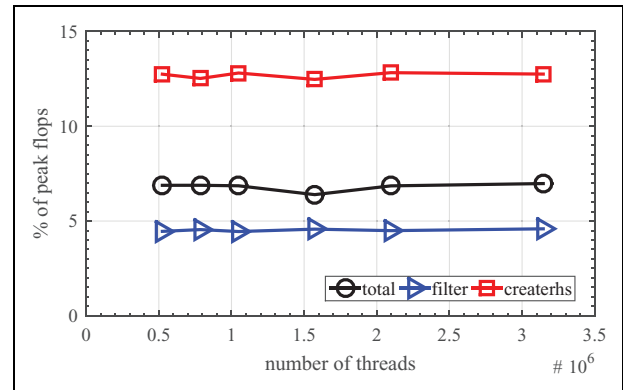

Figure 10. Percentage of theoretical peak performance in terms of floating point operations for the rising thermal bubble test case like in Figure 8.

The percentage of the theoretical peak performance in terms of floating point operations is shown in Figures 9 and 10. Not surprisingly, we obtain the best performance for our optimized part *create_rhs*. For the baroclinic wave test, *create_rhs* reaches 1.21 PFlops (12.1% of peak) and for the rising thermal bubble test, it reaches 1.28 PFlops (12.8% of peak) on the entire machine. The sustained performance of the entire simulation is at 0.55 PFlops for the baroclinic wave test and at 0.70 PFlops for the rising thermal bubble test on the entire machine Mira.

9. Conclusions

In this article, we present the optimization and performance analysis of the atmospheric model NUMA. Our optimizations have improved the performance of the main computational kernel *create_rhs* of our code by a factor of almost 10 and have brought us very close to the maximum attainable performance due to the peak memory bandwidth (Figure 5). These optimizations allow us to perform most of the computations at 1.2 PFlops on the entire supercomputer Mira by using BG/Q vector intrinsics. The sustained performance of the entire simulation is at 0.70 PFlops for a rising thermal bubble test case using explicit time integration. We have not optimized all parts of the code yet. For the baroclinic wave test, the non-optimized computations for the implicit part of the time integration lead to a slightly lower sustained performance of 0.55 PFlops. We expect to improve our performance significantly by optimizing the remaining non-optimized parts of our code. The best portable version of the main computations was found to be about two times slower than the best non-portable version. We used the IBM XL compiler in this work. We expect the comparison between portable and non-portable optimizations to be very different on other architectures due to the availability of different compilers.

We have shown that NUMA achieves a near perfect strong scaling efficiency of 99.7% for the rising thermal bubble test case using 43 billion grid points on the entire 3.14 million threads of Mira. For the baroclinic wave test

The lowest scaling efficiency for the entire simulation is obtained for 2.1×10^6 threads. This is due to nonoptimal load balancing. The number of elements per thread is always perfectly balanced for all results shown in this article. However, the arrangement of these elements can vary which leads for CG storage to variations in the number of grid points. So far we use the very simple mesh partitioning built into the p4est library. We expect to be able to improve this result by using more advanced mesh partitioning algorithms.

The strong scaling efficiency of the rising thermal bubble test case is shown in Figure 8. We achieve 99.7% strong scaling efficiency on the entire machine for this case. We use a much larger total number of grid points of about 43 billion grid points for this case because we plan to use our code for hurricane and cloud simulations at this kind of problem size. We have not optimized the memory usage of our code. The smallest number of threads that can handle this problem is currently 7.7×10^5. We expect to be able to reduce the memory usage of our code significantly. Also we need to understand the reason why the simulation using 1.6×10^6 shows a reduced performance.

case on the sphere, we obtain a strong scaling efficiency of 99.1% using a mesh with 1.8 billion grid points. This allows us to compute a 1-day forecast at 3.0 km resolution within 4.15 min runtime and fulfils the requirements for operational weather prediction (less than 4.5 min runtime for the dynamics of a 1-day forecast).

As explained in the introduction, we expect this massive increase in resolution to be a major step toward more accurate weather forecasts. Nevertheless, the demand to increase the resolution of NWP models does not end at 3 km resolution (Bauer et al., 2015). The demand for better performance is even more severe when high resolution climate prediction is considered. Climate prediction requires forecast periods of more than 100 years. To simulate such a long period of time at a resolution of 3 km would still require about 1 year of runtime on the entire machine Mira when tracers and physics parameterizations are taken into account. For this reason, we need to continue to work on improving the performance of our code and to optimize it for next generation supercomputers.

Our analysis in this article shows that we need to reduce the amount of memory traffic to further improve our performance. It should be possible to achieve this by recomputing metric terms in each stage of our time integration method. For this reason, we expect to improve our sustained performance of the entire simulation beyond 1 PFlops. This should allow us to reach a uniform horizontal resolution close to 2 km within operational requirements. The next goal will be the optimization of our code for the upcoming next generation supercomputer Aurora at the Argonne National Laboratory. We hope to be able to reach 1 km resolution for global NWP once Aurora is available and once our code is fully optimized for that machine.

Authors' note

Michal A Kopera is now affiliated to Department of Earth and Planetary Science, University of California, Santa Cruz, CA, USA.

Acknowledgements

Declaration of Conflicting Interests

Funding

References

Abdi DS and Giraldo FX (2016) Efficient construction of unified continuous and discontinuous Galerkin formulations for the 3D Euler equations. *Journal of Computational Physics* 320: 46–68.

Bangerth W, Heister T, Heltai L, et al. (2015) The deal.II library, version 8.2. *Archive of Numerical Software* 3(100): 1–8.

Bauer P, Thorpe A and Brunet G (2015) The quiet revolution of numerical weather prediction. *Nature* 525(7567): 47–55.

Burstedde C, Calhoun D, Mandli K, et al. (2014) ForestClaw: hybrid forest-of-octrees AMR for hyperbolic conservation laws. In: Bader M, Bode A and Bungartz HJ (eds) *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*. Amsterdam: IOS Press BV, pp. 253–262.

Burstedde C, Ghattas O, Gurnis M, et al. (2010) Extreme-scale AMR. In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, Louisiana, USA, 13–19 November 2010, pp. 1–12. Washington, DC, USA: IEEE Computer Society.

Burstedde C, Wilcox LC and Ghattas O (2011) P4est: scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM: SIAM Journal on Scientific Computing* 33(3): 1103–1133.

Chung IH, Kim C, Wen HF, et al. (2012) Application data prefetching on the IBM Blue Gene/Q supercomputer. In: *2012 International conference for high performance computing, networking, storage and analysis (SC)*, Salt Lake City, UT, USA, 10–16 November 2012, pp. 1–8. Los Alamitos, CA, USA: IEEE Computer Society Press.

Dennis JM, Edwards J, Evans KJ, et al. (2012) CAM-SE: a scalable spectral element dynamical core for the community atmosphere model. *International Journal of High Performance Computing Applications* 26(1): 74–89.

Giraldo FX and Restelli M (2008) A study of spectral element and discontinuous Galerkin methods for the Navier-Stokes equations in nonhydrostatic mesoscale atmospheric modeling: equation sets and test cases. *Journal of Computational Physics* 227(8): 3849–3877.

Giraldo FX, Kelly JF and Constantinescu EM (2013) Implicit-explicit formulations of a three-dimensional nonhydrostatic unified model of the atmosphere (NUMA). *SIAM: SIAM Journal on Scientific Computing* 35(5): B1162–B1194.

Hungershöfer J and Wierum JMM (2002) On the quality of partitions based on spacefilling curves. In: Peter M. A. Sloot (ed.), *Computational Science—ICCS 2002*, Amsterdam, The Netherlands, 21–24 April 2002 pp. 36–45. Berlin; Heidelberg; New York; Barcelona; Hong Kong; London; Milan; Paris; Tokyo: Springer.

Isaac T, Burstedde C, Wilcox LC, et al. (2015a) Recursive algorithms for distributed forests of octrees. *SIAM Journal on Scientific Computing* 37(5): C497–C531.

Isaac T, Stadler G and Ghattas O (2015b) Solution of nonlinear Stokes equations discretized by high-order finite elements on nonconforming and anisotropic meshes, with application to ice sheet dynamics. *SIAM Journal on Scientific Computing* 37(6): B804–B833.

Isaac TG (2015) *Scalable, adaptive methods for forward and inverse problems in continental-scale ice sheet modeling.* PhD Thesis, UT Electronic Theses and Dissertations. Available at: http://hdl.handle.net/2152/31372.

Jablonowski C and Williamson DL (2006) A baroclinic instability test case for atmospheric model dynamical cores. *Quarterly Journal of the Royal Meteorological Society* 132(621C): 2943–2975.

Johnsen P, Straka M, Shapiro M, et al. (2013) Petascale WRF simulation of hurricane sandy: deployment of NCSA's cray XE6 blue waters. In: *2013 SC - International conference for high performance computing, networking, storage and analysis*, Denver, Colorado, USA, 17–22 November 2013, pp. 1–7. Washington, DC, USA: IEEE Computer Society.

Kelly JF and Giraldo FX (2012) Continuous and discontinuous Galerkin methods for a scalable three-dimensional nonhydrostatic atmospheric model: limited-area mode. *Journal of Computational Physics* 231(24): 7988–8008.

Marras S, Kelly JF, Moragues M, et al. (2015a) A review of element-based Galerkin methods for numerical weather prediction: finite elements, spectral elements, and discontinuous Galerkin. *Archives of Computational Methods in Engineering* 23(4): 673–722.

Marras S, Nazarov M and Giraldo FX (2015b) Stabilized high-order Galerkin methods based on a parameter-free dynamic SGS model for LES. *Journal of Computational Physics* 301: 77–101.

Michalakes J, Benson R, Black T, et al. (2015) Evaluating performance and scalability of candidate dynamical cores for the next generation global prediction system. Available at: https://www2.cisl.ucar.edu/sites/default/files/Michalakes_Slides.pdf

Morozov V, Kumaran K, Vishwanath V, et al. (2013) Early experience on the Blue Gene/Q supercomputing system. In: *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium*, 20 May 2013, pp. 1229–1240. IEEE.

Müller A, Behrens J, Giraldo FX, et al. (2013) Comparison between adaptive and uniform discontinuous Galerkin simulations in dry 2D bubble experiments. *Journal of Computational Physics* 235(0): 371–393.

Nair RD, Choi HW and Tufo HM (2009) Computational aspects of a scalable high-order discontinuous Galerkin atmospheric dynamical core. *Computers and Fluids* 38(2): 309–319.

Rossinelli D, Hejazialhosseini B, Hadjidoukas P, et al. (2013) 11 PFLOP/s simulations of cloud cavitation collapse. In: *2013 International conference for high performance computing, networking, storage and analysis (SC)*, Denver, Colorado, USA, 17–22 November 2013, pp. 1–13. Washington, DC, USA: IEEE Computer Society.

Rudi J, Malossi ACI, Isaac T, et al. (2015) An extreme-scale implicit solver for complex PDEs: highly heterogeneous flow in earth's mantle. In: *Proceedings of the international conference for high performance computing, networking, storage and analysis*, Austin, TX, USA, 15–20 November 2015, p. 5. New York, NY, USA: ACM.

Schneider T (2014) *High-Impact Weather Prediction Project (HIWPP). Technical report, NOAA.* Available at: http://web.archive.org/web/20150221115209/http://hiwpp.noaa.gov/docs/HIWPP_ProjectPlan_Public.pdf

Tufo HM and Fischer PF (1999) Terascale spectral element algorithms and implementations. In: *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, Portland, OR, USA, 14–19 November 1999, p. 68. New York, NY, USA: ACM.

Wedi NP, Bauer P, Deconinck W, et al. (2015) *The Modelling Infrastructure of the Integrated Forecasting System: Recent Advances and Future Challenges.* Technical Memorandum 760 (2015), ECMWF. pp. 48.

Whitaker J (2015) *HIWPP Non-hydrostatic Dynamical Core Tests: Results from Idealized Test Cases. Technical report, NOAA.* Available at: http://web.archive.org/web/20151003234909/; http://www.nws.noaa.gov/ost/nggps/DycoreTestingFiles/HIWPP_idealized_tests-v8%20revised%2005212015.pdf

Wyszogrodzki AA, Piotrowski ZP and Grabowski WW (2012) Parallel implementation and scalability of cloud resolving EULAG model. In: Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski and JerzyWaśniewski (eds), *Parallel Processing and Applied Mathematics 9th International Conference, PPAM 2011,* Torun, Poland, 11–14 September 2011, pp. 252–261.Berlin, Heidelberg: Springer.

Xue W, Yang C, Fu H, et al. (2014) Enabling and scaling a global shallow-water atmospheric model on Tianhe-2. In: *2014 IEEE 28th international parallel and distributed processing symposium*, Phoenix, AZ, USA, 19–23 May 2014, pp. 745–754. IEEE.

Yang C, Xue W, Fu H, et al. (2016) 10M-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics. In: *SC16: international conference for high performance computing, networking, storage and analysis*, Salt Lake City, UT, USA, 13–18 November 2016, pp. 57–68. IEEE Press.

Author biographies

Andreas Müller received a BS, MSc, and PhD in physics from the University of Mainz, Germany. After 4 years as a National Research Council postdoctoral fellow at the Naval Postgraduate School, he is now a scientist at the European Centre for Medium-Range Weather Forecasts in Reading, UK. His current work focuses on developing energy efficient scalable numerical methods toward weather prediction on exascale supercomputers.

Michal A Kopera received BS and MSc in mechanical engineering from Warsaw University of Technology, and MSc in scientific computing and PhD in engineering from

the University of Warwick. After 5 years as a National Research Council postdoctoral fellow at the Naval Postgraduate School, he is now an assistant researcher in the Department of Earth and Planetary Sciences at the University of California, Santa Cruz.

Simone Marras received a MS in aerospace engineering from Politecnico di Milano and a PhD from the Universitat Politècnica de Catalunya jointly with the Barcelona Supercomputing Center. After 2 years as an NRC research associate at the Naval Postgraduate School and 2 years as a research scientist at Stanford University, he is currently an assistant professor in the Department of Mechanical and Industrial Engineering at the New Jersey Institute of Technology. His research interests include computational fluid dynamics for compressible, large eddy simulation of turbulence, and aeroacoustics.

Lucas C Wilcox received a BS in mathematical and computer sciences from the Colorado School of Mines in 2001. He graduated with a MSc and PhD in applied mathematics from Brown University in 2002 and 2006, respectively. He was an ICES Postdoctoral Fellow at the Institute for Computational Engineering and Science at the University of Texas at Austin, and is currently an associate professor in the Department of Applied Mathematics at the Naval Postgraduate School.

Tobin Isaac is an assistant professor in the School of Computational Science and Engineering at Georgia Tech. He has a BA from Rice University and a PhD in computational science, engineering and mathematics from the University of Texas at Austin. His research topics include numerical methods and data structures for partial differential equations (PDEs) and PDE-constrained inference.

Francis X Giraldo received a BS from Princeton University and a PhD from the University of Virginia. He is currently a professor in the Department of Applied Mathematics at the Naval Postgraduate School. His research interests include the analysis of numerical methods, computational fluid dynamics, and high-performance computing.