2015-11-05

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

Müller, Andreas

American Mathematical Society

arXiv:1511.01561v1 [cs.DC] 5 Nov 2015
http://hdl.handle.net/10945/47546

Strong Scaling for Numerical Weather Prediction at Petascale with the Atmospheric Model NUMA

Andreas Müller*, Michal A. Kopera*, Simone Marras†, Lucas C. Wilcox*, Tobin Isaac‡ and Francis X. Giraldo*
*Department of Applied Mathematics
Naval Postgraduate School, Monterey, CA USA
Email: amueller@anmr.de, makopera@nps.edu, lwilcox@nps.edu, fxgirald@nps.edu
†Department of Geophysics
Stanford University, Stanford, CA USA
Email: smarras@stanford.edu
‡Computing Institute
University of Chicago, IL, USA
Email: tisaac@ices.utexas.edu

***Abstract*—Numerical weather prediction (NWP) has proven to be computationally challenging due to its inherent multiscale nature. Currently, the highest resolution NWP models use a horizontal resolution of approximately 15 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). Mesh generation is done using the p4est library. NUMA is capable of running middle and upper atmosphere simulations since it does not make use of the shallow-atmosphere approximation.

This paper presents the performance analysis and optimization of the spectral element version of NUMA. The performance at different optimization stages is analyzed using hardware counters with the help of the Hardware Performance Monitor Toolkit as well as the PAPI library. Machine independent optimization is compared to machine specific optimization using BG/Q vector intrinsics. By using vector intrinsics the main computations reach 1.2 PFlops on the entire machine Mira. The paper 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 us to run a global forecast of a baroclinic wave test case at 3 km uniform horizontal resolution and double precision within the time frame required for operational weather prediction.

I. 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 forecast typically use about 1000 processors [1] with a global resolution of approximately 15 km, whereas the biggest available supercomputers offer more than one million processors allowing more than 10^{15} floating point operations in one second (petascale). One of the reasons for this discrepancy is that many weather models do not scale to this large number of 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 numerical weather prediction at 3 km resolution by the year 2020 [2]. 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 forecasts significantly.

In this paper, we show that the Non-hydrostatic Unified Model of the Atmosphere, NUMA [3], [4], [5], is capable of simulating a global baroclinic wave test case within the timeframe 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. This 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 our knowledge, this paper 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 3D global simulations.

Related Work: Johnson et al. (2013) [6] 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) [7] 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 atmospheric model CAM-SE using a spectral element method similar to the one utilized in NUMA is presented by Dennis et al. (2012) [8]. 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. Other publications do either not solve the fully compressible Navier-Stokes equations [9], [10] or show strong scaling only at much smaller scale [11]. None of these publications is targeted at enabling numerical weather prediction at petascale.

Our paper is organized as follows: the numerical methods are introduced in Section II. Section III presents the two test cases considered for the studies of this paper and Section IV describes the mesh generation with the p4est library. The code optimizations are presented in Section V and scalability results are shown in Section VI.

II. Numerical Methods

NUMA solves the compressible Navier-Stokes equations which can be written as (see e.g.[12])

$$\frac{\partial \boldsymbol{q}}{\partial t} + \boldsymbol{\nabla} \cdot \mathbf{F}\left(\boldsymbol{q}\right) = \boldsymbol{S}\left(\boldsymbol{q}\right), \tag{1}$$

where t is the time, $\boldsymbol{q} \equiv \boldsymbol{q}(t,x,y,z) = \left(\rho, \rho\,\boldsymbol{u}^{\mathrm{T}}, \Theta\right)^{\mathrm{T}}$ is a time dependent vector field containing the so called prognostic variables (air density ρ, 3D wind speed, $\boldsymbol{u}$, and potential temperature, θ) and x, y, 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 *CG storage* and *DG storage*.

A. Spatial Discretization

In order to discretize eq. (1) we introduce a mesh of elements. An example for a 2D cross section of our mesh is illustrated in Fig. 1a. Inside each element we approximate the solution $\boldsymbol{q}$ in each dimension by polynomials of order p. We indicate with $\boldsymbol{q}_N$ the approximate of $\boldsymbol{q}$. To define these polynomials we introduce a mesh of $p+1$ grid points inside each element e and in each direction. To simplify numerical integration we use Lobatto points. We 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 (see Section V). 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 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

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

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

$$\psi_i(\xi) = \begin{cases} \prod_{m \neq i} \frac{\xi - \xi_m}{\xi_i - \xi_m} & ,\ \xi \in [-1,1] \\ 0 & ,\ \xi \notin [-1,1], \end{cases} \tag{3}$$

where Ω_e is the domain of element e.

The goal is now to insert eq. (2) into eq. (1) and solve it for the values of $\boldsymbol{q}_N$ at the grid points $\boldsymbol{q}_{i,j,k}$. In this paper we use a spectral element method. From now on, the spectral element method will be often referred to with the acronym CG, from Continuous Galerkin. We multiply eq. (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:

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

where $w_{l,m,n} = w_l\, w_m\, w_n$, $\mathbf{F}_N = \mathbf{F}\left(\boldsymbol{q}_N\right)$, $\boldsymbol{S}_N = \boldsymbol{S}\left(\boldsymbol{q}_N\right)$ 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 eq. (2) by:

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

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 eq. (5) are computed once at the beginning of the simulation.

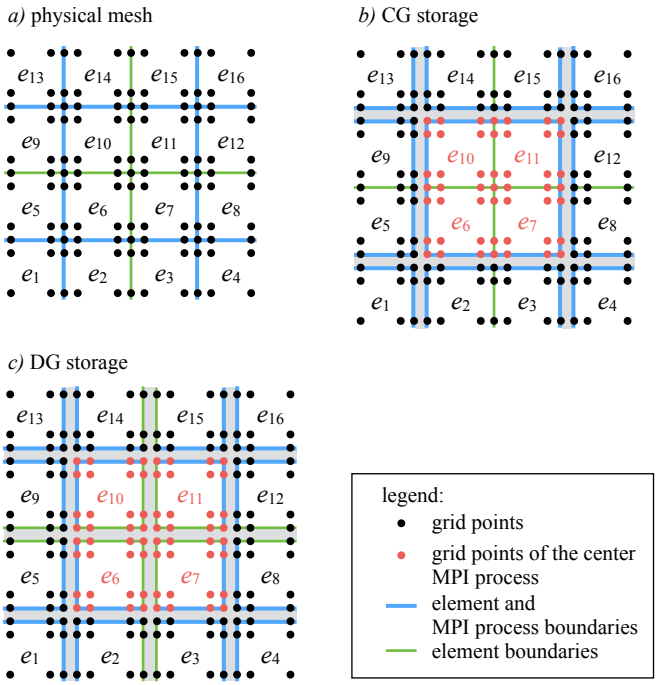

Figure 1: Illustration of a sample 2D cross section containing 16 elements $e_1, \ldots, e_{16}$ of our mesh (a) and two possible approaches to store the data: CG storage (b) and DG storage (c). For illustration purposes we assume that elements e_6, e_7, e_{10} and e_{11} are computed in the same MPI 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.

The basis functions ψ_i vanish outside of element e. For this reason the sum over all elements in eq. (4) reduces to a single element for 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 DSS which stands for Direct Stiffness Summation).

B. 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 eq. (4) and communicate the values of the grid points along the process boundary (blue lines in Fig. 1a).

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-IMEX (IMplicit-EXplicit) [5].

C. Filter

Spectral element methods require stabilization [13]. NUMA allows for the use of different stabilization schemes that range from subgrid-scale models [14] to low-pass filters (Boyd-Vandeven). In this paper we use a Boyd-Vandeven filter. The main idea of this filter is to perform a spectral transformation of the nodal values $\boldsymbol{q}_{i,j,k}$ and to dampen the highest order modes. From a computational point of view this results in multiplying all the values $\boldsymbol{q}_{i,j,k}$ 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 [5].

D. 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 Fig. 1b 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 Fig. 1b). We call this approach CG storage because it requires the solution to be continuous and works only for Continous Galerkin methods.

Another possibility to organize the data is to always store the values along element boundaries for each neighboring element separately (Fig. 1c). We call this approach DG storage because it allows the use of Discontinous Galerkin methods. This approach has significant advantages even for Continous Galerkin methods: it allows the access of data in our computations in a continuous way whereas CG storage produces a random memory access. This feature makes DG storage much better suited for stream prefetching (see Section V). The main disadvantage of DG storage is that up to 2.4 times more data (64 vs. 27 grid points per element for $p = 3$ in 3D) needs to be stored and communicated between different MPI processes.

Two versions of NUMA exist: one version allows only CG storage whereas the other version allows both CG and DG storage. We restrict our work in this paper to the version of NUMA that allows only CG storage because this version was most advanced when we started the work presented in this paper. Therefore we try to take advantage of the reduced amount of data whenever possible. The optimization and

scalability with DG storage will be one of our next goals in our future research.

III. Test Cases

Two test cases are considered in this paper. One test case is the baroclinic wave instability problem by Jablonowski and Williamson [15]. This problem is classically used to test the dynamical core of global circulation models (GCM, spherical geometry). It is initialized by a zonal band of high wind speed in the mid-latitudes (jet stream). A Gaussian perturbation of the zonal wind is added. This perturbation leads to wave like meridional perturbations of the jet stream. After some time the flow pattern looks similar to the polar front jet stream of the real atmosphere. The NUMA results of the full simulation (25 days of simulated time) are reported in [13].

The other test case is a 3D 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 in [4].

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 3D rising thermal bubble test case.

IV. 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 2D advection on the sphere [16], in other applications such as mantle convection and seismic wave propagation [17], and as a backend for the deal.II finite element library [18]. Our present paper is the first time that p4est is used for full 3D atmospheric simulations.

The p4est library represents two- and three-dimensional 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 its 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 [19]. This means that partitioning by this method keeps 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 [20].

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 continuous Galerkin finite elements, is described in [21].

The numerical methods in our simulations involve three-dimensional computations, but the radial 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 radial direction) requires that radial 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 radial resolution would have to be respected at the macro mesh level, increasing the macro mesh's complexity. Second, the three-dimensional Morton curve does not respect the need to keep radial 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 radial 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 [22], 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 two-dimensional "p4est" interface and the three-dimensional "p8est" interface). Essentially, it treats each radial column as a list, from the bottom to top, of the elements created by recursive bisection of the full column, and uses the existing two-dimensional p4est routines to manage the partitioning of columns and intercolumn interactions. A more complete description of this approach can be found in [23, Chapter 2]. The p6est mesh format is illustrated in Figure 2.

The p6est functions include a method for enumerating high-order continuous Galerkin finite elements, and pro-

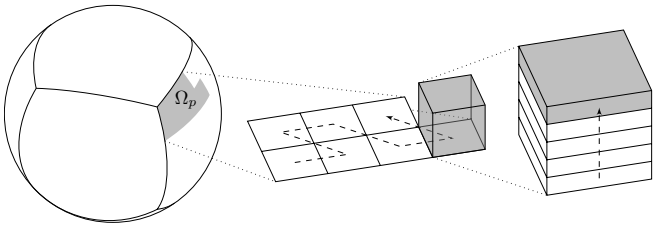

Figure 2: An illustration of meshing with the p6est extension of the p4est library. A macro mesh represents the cubed-sphere domain (left); division of the Morton curve creates the partitions for each MPI process; the columns in the ith partition Ω_i are ordered by a 2D Morton curve (middle); each radial column is stored as a list of "layers" from the bottom to the top (right).

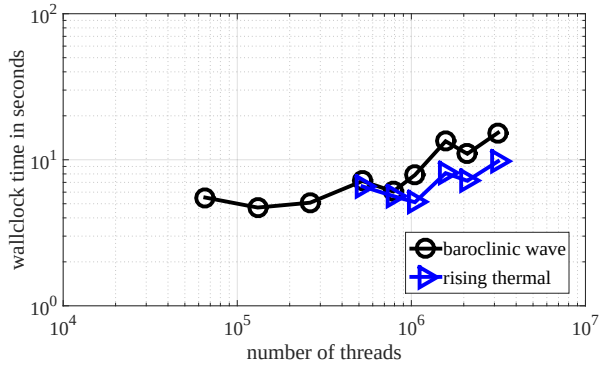

Figure 3: Wallclock time in seconds of the mesh generation with the p4est library up to the entire machine Mira for the simulations in Fig. 4 and 6.

vides point-to-point communicators for distributed vectors for these discretizations, using the same data structures as the underlying p4est library.[1] In p6est, however, the order of nodes is changed so that radial columns of degrees of freedom are numbered contiguously.

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 runtime of the mesh generation is shown in Fig. 3. Even for 43 billion grid points (blue results) it takes always less than 20 seconds runtime.

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 radial direction, and each column can be independently refined into four smaller columns.

[1]I.e., `p6est_lnodes_new()`, which enumerates Gauss-Lobatto nodes for continuous Galerkin finite element calculations, returns a `p8est_lnodes_t` object, the same type returned by `p8est_lnodes_new()`, the equivalent function for a forest of octrees.

V. Code Optimizations

The goal of this section is to optimize our CG storage version of NUMA. To take 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.

```
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 (createrhs)
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
```

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

We tried to optimize all parts of our code. We found createrhs to be the only function that contains enough floating point operations to allow significant optimizations. We expect that we need to merge the different parts of our code into one loop over all elements to optimize the entire code. Since this requires us to move to DG storage we focus in this paper on optimizing createrhs for CG storage and leave the optimization of the other parts for the DG storage version of NUMA.

Table I and II show performance measurements for different versions of createrhs. For the rest of this section we simply refer to the different versions in these tables.

Originally we expected that the performance of our code would improve with increasing polynomial order ($p = 6$, version A). We found however that order $p = 3$ gives significantly better time to solution (version B) without significant impact on the accuracy of our test cases. We use $p = 3$ for the rest of this paper because this order is very well suited for vectorization on Mira (four double precision floating point numbers fit into one register). Another significant

	runtime		flops		instructions		loads that hit			traffic per node
description	**timeloop**	**createrhs**	**qpx**	**fma**	**mix**	**issue**	**L1P**	**L2**	**DDR**	**DDR**
A: $p = 6$, 1 MPI process per core	1099.8 s	855.5 s	31.1 %	85.1 %	23.0 %	39.5 %	3.7 %	1.8 %	0.2 %	3.2 GB/s
B: $p = 3$, 1 MPI process per core	972.7 s	803.5 s	14.4 %	80.0 %	18.0 %	36.2 %	1.4 %	3.5 %	0.2 %	2.4 GB/s
C: like B, 4 MPI processes per core	454.3 s	352.2 s	14.4 %	80.0 %	18.0 %	67.7 %	2.6 %	6.8 %	0.5 %	8.5 GB/s
D: like C, all derivatives of e in one matrix	293.0 s	189.5 s	33.0 %	79.4 %	26.5 %	61.1 %	1.8 %	8.0 %	0.7 %	13.9 GB/s
E: like D, use BLAS function dgemm	447.8 s	336.7 s	51.0 %	85.8 %	16.9 %	53.6 %	2.2 %	11.0 %	0.6 %	8.4 GB/s
F: like C, optimized for compiler vectorization	283.6 s	181.1 s	73.9 %	81.3 %	21.2 %	49.3 %	4.5 %	13.8 %	0.7 %	15.3 GB/s
G: like D, some vector intrinsics	205.0 s	103.6 s	86.4 %	75.7 %	27.0 %	50.6 %	7.5 %	6.4 %	0.9 %	22.4 GB/s
H: like C, rewritten using vector intrinsics	191.7 s	88.8 s	98.6 %	75.7 %	28.9 %	39.7 %	7.2 %	12.2 %	1.1 %	23.6 GB/s
I: like H, 4 OpenMP threads per core	198.5 s	96.6 s	98.6 %	75.7 %	28.1 %	39.9 %	7.6 %	13.6 %	1.0 %	21.8 GB/s
optimal	0 s	0 s	100 %	100 %	50 %	100 %	0 %	0 %	0 %	28.5 GB/s [24]

Table I: Performance measurements for createrhs (the computation of the right hand side in eq.(4), see also code example 1) with the Hardware Performance Monitor Toolkit for the rising thermal bubble test case on 768 nodes of the BG/Q computer Vesta at the Argonne National Laboratory. The simulations with p=3 use 256 x 256 x 384 elements (6.8×10^8 grid points). This corresponds to 2048 elements (55k grid points) per BG/Q core which is the workload we aim at using on the entire machine Mira. For p=6 we use half the number of elements in each direction. All simulations use a CFL number of 0.7 in the vertical direction and are run for a model time of 1 second. This corresponds for p=3 to 690 timesteps (3450 executions of createrhs) and for p=6 to 1123 timesteps (5615 executions of createrhs). The last row indicates what kind of values we try to achieve for each of the columns. The column "qpx" 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. The column "mix" shows the percentage of floating point instructions among all instructions. The column "issue" shows how close this part of the code is to the maximum issue rate of one integer/load/store instruction per cycle per core. The three columns "L1P", "L2" and "DDR" show how many of the loads hit the L1P buffer, the L2 cache and the DDR memory respectively. Finally the last column gives the total bandwidth between L2 cache and DDR memory per node.

	runtime		flops		instructions		loads that hit			traffic per node
description	**timeloop**	**createrhs**	**qpx**	**fma**	**mix**	**issue**	**L1P**	**L2**	**DDR**	**DDR**
A: $p = 6$, 1 MPI process per core	28.0 s	11.8 s	31.1 %	85.1 %	22.7 %	31.0 %	3.1 %	3.9 %	0.4 %	1.5 GB/s
B: $p = 3$, 1 MPI process per core	19.3 s	9.4 s	14.4 %	80.0 %	17.9 %	33.2 %	1.0 %	4.1 %	0.3 %	1.5 GB/s
C: like B, 4 MPI processes per core	9.4 s	4.0 s	14.4 %	80.0 %	17.8 %	63.3 %	2.3 %	8.4 %	0.5 %	7.2 GB/s
D: like C, all derivatives of e in one matrix	7.6 s	2.3 s	33.0 %	79.4 %	26.0 %	56.8 %	1.5 %	9.5 %	0.8 %	10.3 GB/s
E: like D, use BLAS function dgemm	9.4 s	3.9 s	51.0 %	85.8 %	16.5 %	52.5 %	1.7 %	11.9 %	0.7 %	7.0 GB/s
F: like C, optimized for compiler vectorization	7.5 s	2.1 s	73.9 %	81.3 %	20.7 %	47.1 %	4.3 %	13.2 %	0.9 %	13.4 GB/s
G: like D, some vector intrinsics	6.6 s	1.2 s	86.4 %	75.7 %	26.3 %	51.7 %	6.7 %	8.1 %	0.9 %	18.2 GB/s
H: like C, rewritten using vector intrinsics	6.4 s	1.0 s	98.6 %	75.7 %	28.1 %	41.8 %	6.0 %	12.8 %	1.3 %	19.7 GB/s
I: like H, 4 OpenMP threads per core	5.8 s	1.1 s	98.6 %	75.7 %	27.1 %	40.2 %	6.3 %	16.7 %	0.6 %	17.8 GB/s
optimal	0 s	0 s	100 %	100 %	50 %	100 %	0 %	0 %	0 %	28.5 GB/s [24]

Table II: Performance measurements like in Table I but for the baroclinic wave test case at a horizontal resolution of 21.3 km and a vertical resolution of 1 km. All simulations use the same effective resolution, same final model time of 4 hours and same CFL number of 0.4 in the horizontal direction and 6.4 in the vertical direction. For p=3 this corresponds to 144 elements per cubed sphere edge, 10 elements in the vertical and 947 timesteps. For p=6 we use 72 elements per cubed sphere edge, 5 elements in the vertical and 1582 timesteps. All simulations use 972 BG/Q nodes on Vesta. This setup gives us the same workload per node like in Section VI on the entire machine Mira.[2]

speedup was obtained by running four MPI processes per core (version C).

We computed the derivatives of eq. (5) in versions A, B and C for each of the five variables of $\boldsymbol{q}$ separately. By merging all these derivatives into one matrix for each direction and element we achieved another significant speedup (version D). We also tried to use the BLAS function dgemm (version E) without any improvement of the performance.

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

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 OpenMP. At the end of this section we give a short description of possible next steps for further optimization.

A. 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 createrhs 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 A, B, C and D (column qpx in Table I and II). 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 F show that this simple step leads to a significant improvement of the vectorization.

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

Code example 2: Fortran code similar to a function from the non-optimized initial version of NUMA (used in versions A, B, C, D and E in Table I and II).

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

Code example 3: Like code example 2 rewritten for improved compiler vectorization (used in version F)

B. BG/Q Vector Intrinsics

To make even better use of the vector unit we rewrote our function createrhs by using BG/Q vector intrinsics (code example 4). We first kept the computation of the derivatives unchanged (version G). This gave us another significant speedup. Using vector intrinsics for the entire function createrhs gave us another minor speedup (version H).

C. OpenMP

OpenMP allows reducing 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 createrhs race conditions can occur in the summation over all the elements in eq. (4). Using OpenMP 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 OpenMP threads can never compute neighboring elements at the same time.

```
real, dimension(4,4,4) :: rho, rho_x, rho_y, &
   rho_z, u, v, w, u_x, v_y, w_z, rhs
!IBM* align(32, rho, rho_x, rho_y, rho_z, u, v, w,
     u_x, v_y, w_z, rhs)
! declare variables representing registers: (each
     contains four double precision floating point
     numbers)
vector(real(8)) vct_rho, vct_rhox, vct_rhoy, vct_rhoz
vector(real(8)) vct_u, vct_v, vct_w, vct_rhs
if (iand(loc(rho), z'1F') .ne. 0) stop 'rho is not
     aligned'
... ! check alignment of other variables
do e=1,num_elem ! loop through all elements
   do k=1,4 ! loop over points in z-direction
      do j=1,4 ! loop over points in y-direction
         ... ! compute derivatives rho_x, ...
         ! load always four floating point numbers:
         vct_u = vec_ld(0, u(1,j,k))
         vct_v = vec_ld(0, v(1,j,k))
         vct_w = vec_ld(0, w(1,j,k))
         vct_rhox = vec_ld(0, rho_x(1,j,k))
         vct_rhoy = vec_ld(0, rho_y(1,j,k))
         vct_rhoz = vec_ld(0, rho_z(1,j,k))
         ! rhs = u*rho_x
         vct_rhs = vec_mul(vct_u,vct_rhox)
         ! rhs = rhs + v*rho_y
         vct_rhs = vec_madd(vct_v,vct_rhoy,vct_rhs)
         ! rhs = rhs + w*rho_z
         vct_rhs = vec_madd(vct_w,vct_rhoz,vct_rhs)
         ! write result from register into cache:
         call vec_st(vct_rhs, 0, rhs(1,j,k))
         ...
      end do !j
   end do !k
end do !e
```

Code example 4: Like code example 2 rewritten with vector intrinsics (used in versions G, H and I)

To ensure this we need to synchronize all threads by using an OpenMP barrier after each element computation. These barriers slow down createrhs by less than 10% (version I). Nevertheless we obtain in the case of the baroclinic instability a noticeable improvement of the runtime of the entire timeloop due to the reduced amount of work for the IMEX corrections in the vertical direction. We obtained the best performance by using 4 OpenMP threads per MPI process (2, 8, 16 and 64 OpenMP threads per MPI process were slower).

D. Next Steps

Our measurements show that our final version I achieves an excellent level of vectorization (98.6% of all floating point operations are vectorized). The main weakness of our code is the very low percentage of floating point instructions among all instructions and the fairly high number of loads that hit L1P buffer and L2 cache. 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. These issues could be avoided by merging the different parts of our code into one loop over all the elements and by rearranging the data

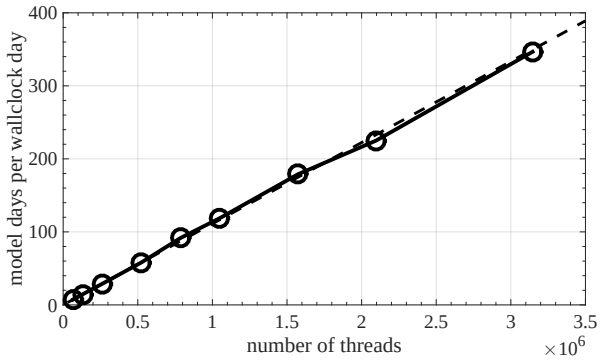

Figure 4: Strong scaling for the baroclinic wave test case with polynomial order $p = 3$ using 1024 elements per cubed sphere edge and 10 elements in the vertical direction (31 degrees of freedom). This corresponds to a global effective resolution of 3.0 km and a total number of about 1.8 billion grid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

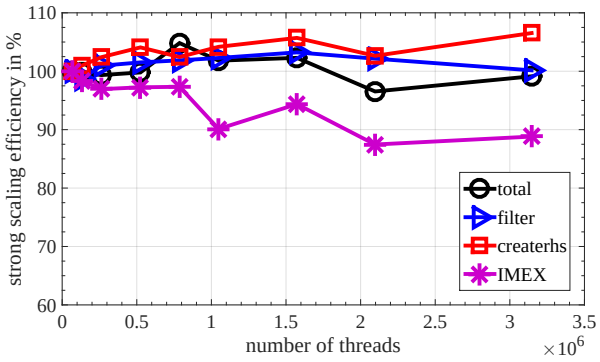

Figure 5: Strong scaling efficiency over base run on 6.5×10^4 threads for the simulations shown in Fig. 4.

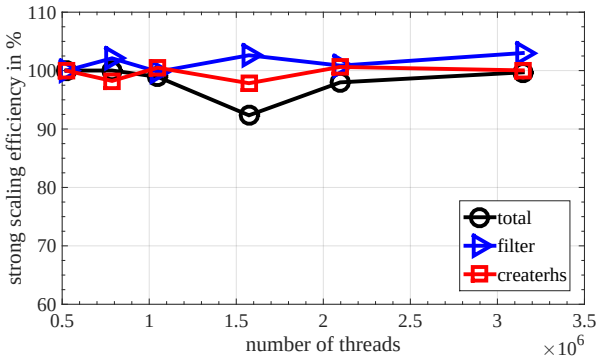

Figure 6: Strong scaling efficiency over base run on 7.7×10^5 threads for the rising thermal bubble test case using $1024 \times 1024 \times 1536$ elements which corresponds to about 43 billion grid points.

in the order in which it is needed. This will be possible in the DG storage version of NUMA.

VI. Strong Scaling Results

We present in this section strong scaling results up to the entire machine Mira for the baroclinic wave test case (Fig. 4, 5 and 7) and the rising thermal bubble test case (Fig. 6 and 8). All these results use version I from Section V.

The runtime of the entire simulation for the baroclinic wave test case is shown in Fig. 4. The dynamics of a one day forecast needs to be finished within less than about 4.5 minutes runtime (more than 320 model days per wallclock 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 minutes runtime per one day forecast (346.6 model days per wallclock day).

The strong scaling efficiency of the simulations in Fig. 4 is shown in Fig. 5 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 createrhs 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 time the time spent in our OpenMP barriers is decreasing. The IMEX part gives us the lowest scaling efficiency. We still need to understand the reason for this behavior.

The lowest scaling efficiency for the entire simulation is obtained for 2.1×10^6 threads. This is due to non-optimal load balancing. The number of elements per thread is always perfectly balanced for all results shown in this paper. 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 Fig. 6. 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.

The percentage of the theoretical peak performance in terms of floating point operations is shown in Fig. 7 and 8. Not surprisingly we obtain the best performance for our optimized part createrhs. For the baroclinic wave test

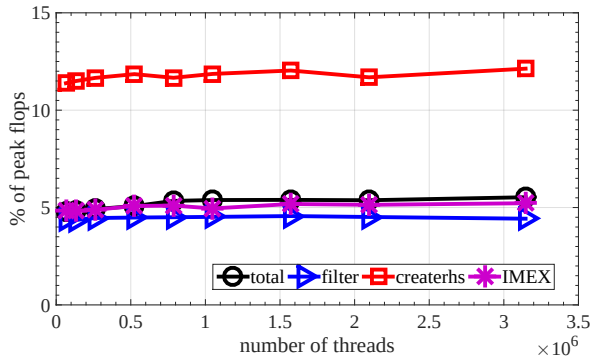

Figure 7: Percentage of theoretical peak performance in terms of floating point operations for the baroclinic wave test case like in Fig. 4.

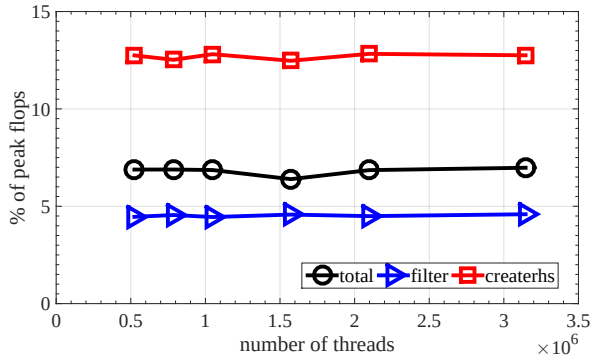

Figure 8: Percentage of theoretical peak performance in terms of floating point operations for the rising thermal bubble test case like in Fig. 6.

createrhs reaches 1.21 PFlops (12.1% of peak) and for the rising 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 bubble test on the entire machine Mira.

VII. Conclusions

In this paper, we present the optimization and performance analysis of the atmospheric model NUMA. We are able 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 our simulation is at 0.70 PFlops for a rising thermal bubble test case using explicit time integration. We have not optimized the entire 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.

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 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 one day forecast at 3.0 km resolution within 4.15 minutes runtime and fulfills the requirements for operational weather prediction (less than 4.5 minutes runtime for the dynamics of a one day forecast).

As explained in the introduction, we expect this massive increase in resolution to be a major step towards more accurate weather forecasts. Nevertheless, the demand to increase the resolution of NWP models does not end at 3 km resolution [1]. The demand for better performance is even more severe when high resolution climate prediction is considered. Climate prediction requires forecast periods of more than one hundred years. To simulate such a long period of time at a resolution of 3 km would still require about one 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 paper shows that we need to merge the different parts in our code and improve our memory access. We are confident that we can achieve this goal by using DG storage. DG storage allows us to access the data in the order in which it is stored and enables much better use of the stream prefetcher. DG storage will also allow us to merge different parts of the code since all computations are performed on an element basis (compare code example 1). For this reason, we expect to improve our sustained performance beyond 1 PFlops by moving to DG storage. 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. Aurora is expected to achieve a peak performance of 180 PFlops (18 times more than Mira). We hope to be able to reach 1 km resolution for global numerical weather prediction once Aurora is available and once our code is fully optimized for that machine.

VIII. Acknowledgements

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. We would like to thank Vitali Morozov at the Argonne National Laboratory for his support in analyzing the performance of our code with the Hardware Performance Monitor Toolkit. Financial support for the work presented in this paper was provided by the Office of Naval Research through program element PE-0602435N, the Air Force Office of

Scientific Research through the Computational Mathematics program, and the National Science Foundation (Division of Mathematical Sciences) through program element 121670. AM, MK, and SM are grateful to the National Research Council of the National Academies.

REFERENCES

[1] P. Bauer, A. Thorpe, and G. Brunet, "The quiet revolution of numerical weather prediction," *Nature*, vol. 525, no. 7567, pp. 47–55, 2015.

[2] "High-Impact Weather Prediction Project," NOAA, Tech. Rep., 2014. [Online]. Available: http://hiwpp.noaa.gov/docs/HIWPP_ProjectPlan_Public.pdf

[3] F. X. Giraldo and M. Restelli, "A study of spectral element and discontinuous Galerkin methods for the Navier-Stokes equations in nonhydrostatic mesoscale atmospheric modeling: Equation sets and test cases," *J. Comput. Phys.*, vol. 227, no. 8, pp. 3849–3877, 2008.

[4] J. F. Kelly and F. X. Giraldo, "Continuous and discontinuous Galerkin methods for a scalable three-dimensional nonhydrostatic atmospheric model: limited-area mode," *J. Comput. Phys.*, vol. 231, no. 24, pp. 7988 – 8008, 2012.

[5] F. X. Giraldo, J. F. Kelly, and E. M. Constantinescu, "Implicit-explicit formulations of a three-dimensional nonhydrostatic unified model of the atmosphere (NUMA)," *SIAM J. Sci. Comput.*, vol. 35, no. 5, pp. B1162–B1194, 2013.

[6] P. Johnsen, M. Straka, M. Shapiro, A. Norton, and T. Galarneau, "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*. IEEE, 2013, pp. 1–7.

[7] A. A. Wyszogrodzki, Z. P. Piotrowski, and W. W. Grabowski, "Parallel implementation and scalability of cloud resolving EULAG model," in *Parallel Processing and Applied Mathematics*. Springer, 2012, pp. 252–261.

[8] J. Dennis, J. Edwards, K. J. Evans, O. Guba, P. H. Lauritzen, A. A. Mirin, A. St-Cyr, M. A. Taylor, and P. H. Worley, "CAM-SE: A scalable spectral element dynamical core for the Community Atmosphere Model," *International Journal of High Performance Computing Applications*, p. 1094342011428142, 2011.

[9] H. M. Tufo and P. F. Fischer, "Terascale spectral element algorithms and implementations," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*. ACM, 1999, p. 68.

[10] W. Xue, C. Yang, H. Fu, X. Wang, Y. Xu, L. Gan, Y. Lu, and X. Zhu, "Enabling and scaling a global shallow-water atmospheric model on Tianhe-2," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 745–754.

[11] R. D. Nair, H. W. Choi, and H. M. Tufo, "Computational aspects of a scalable high-order discontinuous Galerkin atmospheric dynamical core," *Comput. Fluids*, vol. 38, no. 2, pp. 309 – 319, 2009.

[12] A. Müller, J. Behrens, F. X. Giraldo, and V. Wirth, "Comparison between adaptive and uniform discontinuous Galerkin simulations in dry 2D bubble experiments," *J. Comput. Phys.*, vol. 235, pp. 371 – 393, 2013.

[13] S. Marras, J. F. Kelly, M. Moragues, A. Müller, M. A. Kopera, M. Vázquez, F. X. Giraldo, G. Houzeaux, and O. Jorba, "A Review of Element-Based Galerkin Methods for Numerical Weather Prediction: Finite Elements, Spectral Elements, and Discontinuous Galerkin," *Archives of Computational Methods in Engineering*, pp. 1–50, 2015.

[14] S. Marras, M. Nazarov, and F. X. Giraldo, "Stabilized high-order Galerkin methods based on a parameter-free dynamic SGS model for LES," *J. Comput. Phys.*, vol. 301, pp. 77–101, 2015.

[15] C. Jablonowski and D. L. Williamson, "A baroclinic instability test case for atmospheric model dynamical cores," *Q. J. R. Meteor. Soc.*, vol. 132, no. 621C, pp. 2943–2975, 2006.

[16] C. Burstedde, D. Calhoun, K. Mandli, and A. R. Terrel, "ForestClaw: Hybrid forest-of-octrees AMR for hyperbolic conservation laws," in *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, M. Bader, A. Bode, and H.-J. Bungartz, Eds. Amsterdam: IOS Press BV, 2014, pp. 253–262.

[17] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, and L. C. Wilcox, "Extreme-scale AMR," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–12.

[18] W. Bangerth, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, B. Turcksin, and T. Young, "The deal.II Library, Version 8.2," *Archive of Numerical Software*, vol. 3, no. 100, 2015.

[19] J. Hungershöfer and J.-M. Wierum, "On the quality of partitions based on space-filling curves," in *Computational Science—ICCS 2002*. Springer, 2002, pp. 36–45.

[20] C. Burstedde, L. C. Wilcox, and O. Ghattas, "p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees," *SIAM Journal on Scientific Computing*, vol. 33, no. 3, pp. 1103–1133, 2011.

[21] T. Isaac, C. Burstedde, L. C. Wilcox, and O. Ghattas, "Recursive algorithms for distributed forests of octrees," *SIAM Journal on Scientific Computing (to appear)*, 2015.

[22] T. Isaac, G. Stadler, and O. Ghattas, "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 (in review)*, 2015.

[23] T. Isaac, "Scalable, Adaptive Methods for Forward and Inverse Problems in Continental-Scale Ice Sheet Modeling," Ph.D. dissertation, The University of Texas at Austin, August 2015.

[24] V. Morozov, K. Kumaran, V. Vishwanath, J. Meng, and M. E. Papka, "Early Experience on the Blue Gene/Q Supercomputing System," *IEEE Xplore*, pp. 1229–1240, 2013.