2015 IEEE International Conference on Cluster Computing

Developing MiniApps on Modern Platforms Using Multiple Programming Models

O. E. B. Messer

Oak Ridge National Laboratory
Oak Ridge, TN 37831
&
Department of Physics & Astronomy
University of Tennessee
Knoxville, TN 37996

E. D'Azevedo, J. Hill, W. Joubert,
S. Laosooksathit, A. Tharrington

Oak Ridge National Laboratory
Oak Ridge, TN 37831

***Abstract*—We have developed a set of reduced, proxy applications (MiniApps) based on large-scale application codes supported at the Oak Ridge Leadership Computing Facility (OLCF). The MiniApps are designed to encapsulate the details of the most important (i.e. the most time-consuming and/or unique) facets of the applications that run in production mode on the OLCF. In each case, we have produced or plan to produce individual versions of the MiniApps using different specific programming models (e.g., OpenACC, CUDA, OpenMP). We describe some of our initial observations regarding these different implementations along with estimates of how closely the MiniApps track the actual performance characteristics (in particular, the overall scalability) of the large-scale applications from which they are derived.**

I. Introduction

There is a fundamental tension between the size and complexity of modern, large-scale scientific application codes and the design and fielding of new computational platforms on which these codes are run. The ultimate utility of current petascale and future exascale platforms will be judged based on the scientific productivity of their users. This means that modifying and improving the algorithms and implementations used in these large-scale codes to increase efficiency on these new platforms is of primary importance. It is equally clear that a thorough understanding of the requirements and limitations of current and planned software features is absolutely essential in system design. However, such optimization work is often arduous because of the sheer size of the code bases, the complexity of the codes themselves, and, in many cases, their associated build systems. This often means that accessible benchmarks are not available at any given instant in the software and hardware development processes – for vendors to optimize system design points against, for computer science and applied mathematics researchers to use as laboratories to test new algorithmic and implementation ideas, and even for application developers themselves to use as effective test mechanisms to investigate the impacts of new hardware and programming model developments. For these reasons and others, the use of reduced applications that share many of the performance and implementation features of large, fully-featured code bases ("MiniApps") has gained considerable traction in recent years, especially in the context of exascale planning exercises. The central conceit of MiniApps relies on the realization that many large scientific codes contain a small, countable number of "hot spots" that dominate their performance characteristics and that other parts of these large codes contain recurring tropes, where, though the ultimate aim might be different for each routine, the performance characteristics are quite similar [9]. The idea is to encapsulate these behaviors in a simpler, reduced version of the application, mimicking the important characteristics, but obviating the need for the user to be a developer/expert on the code itself.

An ever-expanding group of MiniApps has been developed in recent years. The Mantevo suite [9] was perhaps the first and largest set of MiniApps. Mantevo includes MiniApps designed to mimic the performance of finite-element codes (MiniFE), molecular dynamics codes (MiniMD), electrical circuit design codes (MiniXYCE), and others. It is important to note that the particular choice of MiniApps in the Mantevo suite is a direct result of the interests and expertise of the application community at Sandia National Laboratories, where the suite was developed. Other MiniApps have been produced at Los Alamos National Laboratory, e.g. CGPOP [16] and MCMini [11], and at Argonne National Laboratory (see, e.g. the MiniApps list at the CESAR website, https://cesar.mcs.anl.gov/content/software). We have undertaken a project to instantiate a similar collection of computational artifacts, particular to the user community of the Oak Ridge Leadership Computing Facility (OLCF), that can be changed, revised, or even retired as architectures and software systems evolve.

978-1-4673-6598-7/15 $31.00 © 2015 IEEE
DOI 10.1109/CLUSTER.2015.130

It is interesting to note that MiniApps are often described via exclusion, i.e. by enumerating the things that are not MiniApps. Barrett et. al [4] characterize MiniApps by noting that they are not to be confused with compact applications – wherein a particular implementation of physics is captured in isolation – nor are they so-called skeleton applications, designed to reproduce a particular pattern of inter-process communication (a goal so narrow that the computation performed between communication epochs is sometimes "faked"). Heroux et al. [8] expand this list of what MiniApps are not to include Scalable Synthetic Compact Applications (SSCA), which were produced through the DARPA High Productivity Computing Systems (HPCS) program to evaluate the productivity of emerging HPC systems. The NNSA Exascale Applications Working Group noted [15] that despite these attempts to constrain the definition of MiniApps, their role in cooperative R&D efforts (e.g. in so-called co-design efforts, where hardware and software designers engage directly with application programmers to improve the design of both architectures and algorithms for future systems) requires them to be several things all at once. They must be expansive enough to accurately model full application behavior, but small enough to be manageable and, in some sense, parseable, by researchers ranging from applied mathematicians to compiler writers. We consider a primary aim of the research started under this project to be an attempt to delineate what particular attributes maximize the usefulness of MiniApps for these multifarious purposes.

Related to this primary aim is the question of whether MiniApps can be effectively divorced from the underlying physics being modeled by a particular application code (see, e.g., the discussion of this point by [3]). We posit that this notion is not a useful operating assumption based on our extensive past experience with transitioning large application codes across many orders of magnitude in scalability and performance. Indeed, we note that the realization that HPCCG (one of the original components of the Mantevo MiniApp suite [9]) could only, "provide a stronger tie to applications of interest," by realizing that, "the context in which the linear system is formed needed [to be](sic) strengthened"[3] suggests a stronger tie between performance and "intent" is often necessary to effectively understand full application performance. Nevertheless, the *additional* application of objective techniques to measure the degree to which a MiniApp models the progenitor application are likely useful as well. An example of this type of analysis is provided by the Byfl tool of [14].

We test several of these assumptions and ideas via our suite of MiniApps by comparing performance measurements made by varying MiniApp inputs and particular code implementations. We also attempt to understand the level of fidelity to the underlying physics required in order to accurately model full-application performance. We begin with a short discussion on formulating MiniApps from existing codes. In the following sections, we provide short descriptions of the first set of OLCF MiniApps, along with a set of initial scalability and performance results. Though many possible measures of performance are possible, we concentrate here on comparing the distributed-memory scalability of the MiniApps to their progenitor apps as a first measure of congruence. Such a comparison is probably a necessary but not sufficient condition to establish the quality of the MiniApps in reproducing the performance characteristics of the production applications.

II. MiniApps distillation and evaluation

The formulation of a new MiniApp can likely be described by a rather generic series of steps, but is dependent on the purpose and provenance of the MiniApp. We are concerned here with MiniApps that are directly derived from existing codes, as opposed to MiniApps that are generated *ab initio* to mimic a whole class of codes or algorithms. Therefore, some of the steps we outline below may not be applicable in every circumstance.

The first step we suggest is a complete performance profile of the progenitor code. Of primary importance in this analysis is determining which parts of the original code are responsible for determining the performance characteristics of the runtime. The most common set of questions in this analysis includes:

- Which code routines/functions perform the bulk of the floating-point operations (FLOPs)?
- How many of these FLOP-rich routines exist? Is the profile 'flat' or concentrated in only a handful of routines?
- Are the FLOP-rich routines tightly coupled to one another – either logically or temporally?
- Do these same routines also dominate the total runtime, or do other routines (e.g., I/O or distributed-memory communication routines) dominate the total time-to-solution?

It then becomes important to decide which performance behaviors will be modeled with the MiniApp, as it may prove difficult or not as useful to model several of them in the same code. A straightforward example of such a separation of concerns could be the formulation of a I/O skeleton app **and** a FLOP-centric MiniApp from the same production code. The I/O portion of the original code might be under constant development and could change quite drastically on short development time scales, while the underlying algorithmic implementations could be far slower to change.

This possibility leads us to consider another step in our MiniApp generation recipe: Given that the MiniApp developers are also most often among the progenitor application developers (Heroux et al. [9] suggest that this is, in fact, the preferred state), it is advantageous for the creation of the MiniApp to become part of the standard development cycle of the production application. Automating the generation (and re-generation) of associated MiniApps is an excellent way to make sure that the performance questions asked of the MiniApp at any epoch provide an accurate picture of the production application's performance at that point. In the absence of an automated way to perform the distillation, some effort needs to be expended for a manual re-distillation whenever major rewrites occur in the production code.

Finally, the build system and run system for the resultant MiniApps must be kept as simple as possible. Because the MiniApps need to be immediately accessible to a varied population of potential users, almost anything beyond a simple `makefile` and the specification of a handful of command-line arguments should be eschewed. Importantly, if more than

this level of complexity seems to be required, it is possible that the resulting MiniApp itself is too complex to be human-parseable, reducing its usefulness.

III. AORSA MiniApp

AORSA (All Orders Spectral Algorithm) models the response of the plasma radio frequency (RF) waves in a fusion tokamak device [10]. The plasma state is described by a distribution function. For RF applications, the fast-wave time scale leads to effective approximation of the electric field, magnetic field, and distribution function as a time-averaged equilibrium part and a rapidly oscillating time harmonic part. The time harmonic terms satisfy the generalized Helmholtz equation, relating the frequency of wave, the plasma current induced by the wave fields, and the plasma conductivity kernel. Fourier modes are used as basis functions to represent the electric field. Collocation on an $M \times M$ rectangular grid is used to construct a complex dense linear system of size N = $3 \times M \times M$. A reduced linear system can be constructed by transforming the linear system (using the fast Fourier transform) into the real physical space and considering only collocation points within the plasma region – typically about 30% of grid points are in the vacuum region. Each processor constructs complete block rows of the matrix to facilitate a two-dimensional fast Fourier transform to real space. The variables in the vacuum region are eliminated, the reduced rows are transformed back to a Fourier representation, and the data reshuffled into a two-dimensional block cyclic distribution that is compatible with ScaLAPACK. Finally, the dense linear system is solved by LU factorization; performance analysis tells us this is the most computationally intensive part of the calculation. AORSA is written using Fortran90 and MPI and, by default, makes calls to ScaLAPACK `PZGETRF/PZGETRS` for the LU factorization and the solution of the large dense complex system of linear equations. Because the dominant computational kernel is the dense LU factorization, an appropriate MiniApp for AORSA is the LU factorization of a dense matrix. With the advent of accelerators and co-processors in the HPC architecture spectrum and the relative limited memory available in each compared to traditional CPU architectures, additional care must be taken in memory management. Specific to AORSA, the amount of memory required to store the large, dense linear system far exceeds the available accelerator (GPU or MIC) memory, even on large distributed supercomputers such as Titan at the OLCF. Thus, out-of-core factorization methods, originally developed when CPU architectures were relatively memory-scarce [5], [7], must be evaluated. In this work, we adapt the "left-looking" out-of-core algorithm for LU factorization described in [5] with a minor change that seeks to minimize the data transfer between the host and the device memory. Central to this algorithm is the necessity for an in-core parallel LU factorization method for a smaller sub matrix of the large linear system that operates primarily on the device with minimal communication between devices. For details of the out-of-core algorithm, see [5].

Figure 1 illustrates the effectiveness of the AORSA MiniApp in mimicking the scalability of the full AORSA application. Titan, the OLCF flagship supercomputer that employs a heterogeneous CPU/GPU architecture, was used for these simulations. Titan is a Cray XK7 composed of 200 cabinets. Titan is a hybrid architecture where each individual compute node uses both a conventional 16-core AMD 6274 Opteron CPU connected to 32GB of 1600 MhZ DDR3 SDRAM and an NVIDIA K20x (Kepler) GPU with 6 GB of GDDR5 memory. Titan, with 18,688 of these hybrid compute nodes, has a theoretical peak computational performance of more than 27 PFLOPS. Each of the compute nodes is interconnected with Cray's high-performance, 3D-torus Gemini network. As expected, overall scalability of AORSA and its MiniApp are similar indicating that the LU factorization is the dominant computational kernel. There are still potential opportunities for performance optimization and tuning, such as individual tuning of matrix block sizes for ScaLAPACK on both CPU and GPU devices, exploiting asynchronous data transfer operations, and using look-ahead computation of the next panel to reduce the time spent on the critical path for LU factorization. However, the current gigaflop performance of the GPUs is approximately 20 times the performance of a single CPU core; thus, pursuing efficient new algorithms implemented on GPUs offers a potential advantage in both cost and power efficiency.

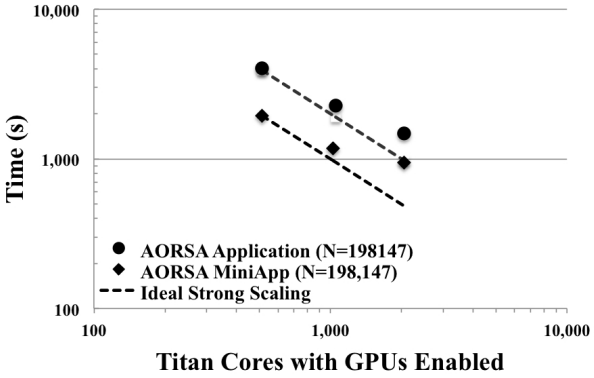

Fig. 1. A comparison of the strong scaling for the full AORSA code and the AORSA MiniApp using Titan. The flop-heavy portions of the algorithm are computed on the GPU while the remainder of the algorithm uses the CPUs. The dashed lines indicate the ideal strong scaling behavior.

Figure 2 illustrates our initial efforts to study the performance of the AORSA MiniApp on Intel Xeon Phi. These results were obtained using Beacon, a 48-node cluster operated by the National Institute of Computational Science (NICS) at the University of Tennessee. Each node is equipped with two 8-core Intel Xeon E5-2670 processors with 256 GB of memory and four Intel Xeon Phi coprocessors (5110P) with 8 GB of memory. The results reported are for LU factorizations of matrix size N = 90,000 using a single Xeon Phi. While comparing the scalability to the full application is warranted, currently the small size of Beacon limits the realistic simulations that can be accomplished. However, in the single node case, automatic offloading – when the system determines that using the co-processor is advantageous – only activates for larger ScaLAPACK block sizes. Automatic offloading is currently available for a limited set of Intel Math Kernel Library (MKL) functions. It is, therefore, not surprising that the the automatic offloading activates only at larger block sizes when the flops are concentrated relative to communication and data transfer. Significant performance optimizations remain before the Xeon Phi can become as performant as the current GPU implementation.

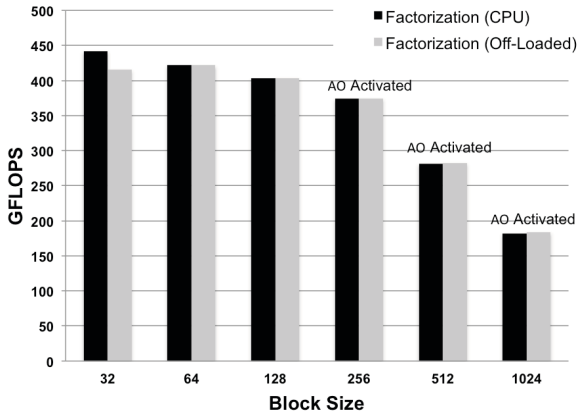

Fig. 2. Comparison of the AORSA MiniApp performance when using just the CPU or when portions of the computation are automatically offloaded (AO), as determined by the system, to the Xeon Phi. Note that the automatic offloading is only activated at larger block sizes, as indicated by "AO Activated". For smaller block sizes, the system determines that AO is not advantageous, and the CPU performs all of the computations.

IV. ZIZ: A CHIMERA MINIAPP

We have recently developed a MiniApp designed to serve as a proxy for the CHIMERA code that we have dubbed Ziz[1]. CHIMERA [12], [13] is a multi-dimensional, multi-physics code designed to study core-collapse supernovae. The code is made up of three essentially independent parts: hydrodynamics, nuclear burning, and a neutrino transport solver combined within an operator-split approach. The hydrodynamics is directionally split, and the ray-by-ray transport and the thermonuclear kinetics solve occur after the radial sweep occurs, when all the necessary data for those modules is local to a processor. The directionally split hydrodynamics is implemented via MPI, while OpenMP is used to parallelize the local-to-each-MPI-rank work performed in the nuclear kinetics solver and the neutrino transport. All of the constituent parts of CHIMERA are written in FORTRAN 90. The combination of directionally split hydrodynamics and operator-split local physics provides the context for the communication and computation patterns found in CHIMERA. The neutrino transport in CHIMERA is performed only in the radial direction (as this step is the most computationally intensive, and would preclude realistic runtimes if treated in full generality). The result is a subcommunicator-local sparse linear solve at each timestep. The nuclear composition in regions that are not in nuclear statistical equilibrium is evolved via a completely local dense linear solve that is sub-cycled within each hydrodynamic timestep for every cell in the domain. Ziz will eventually be able to capture the behaviors of the directionally split hydrodynamics (i.e. the restricted data transposes), the operator-split local physics, and the dense and sparse linear solves used in the kinetics and transport solves. The resulting MiniApp should be useful as a proxy for a whole class of multi-physics codes. Individual modules for nuclear burning, sparse linear system solution, and hydrodynamics (specifically, flux reconstruction at finite-volume interfaces) will be produced to allow modular CPU-GPU MiniApps to be constructed. In an initial foray, we have used the directionally split hydro version of Ziz to quantify a handful of architectural impacts on Cray XK7 and XC30 platforms and have compared these impacts to results from a new Infiniband-based cluster at the OLCF.

[1]A ziz is a giant griffin-like bird in Hebrew mythology, often portrayed as something somewhat akin to a Greek chimera.

A. Experiments with Ziz

We have performed scaling studies for Ziz on three different platforms at OLCF: Titan, Eos, and Rhea. The architecture of Titan was described previously in Section III.

Eos is a 744-node Cray XC30 cluster. The system has two external login nodes. The compute nodes are organized in blades. Each blade contains four nodes connected to a single Aries NIC. Every node has 64 GB of DDR3 SDRAM and two Intel Xeon E5-2670 CPUs with 8 physical cores each. Intels Hyper-threading (HT) technology allows each physical core to work as two logical cores so each node can function as if it has 32 cores. Each of the two logical cores can store a program state, but they share most of their execution resources. In total, the Eos compute partition contains 11,904 traditional processor cores (23,808 logical cores with Intel Hyper-Threading enabled), and 47.6 TB of memory.

Rhea is a 196-node commodity-type Linux cluster. Each of Rhea's nodes contain two 8-core 2.0 GHz Intel Xeon E5-2650 processors with Hyper-Threading and 64GB of main memory. The nodes are connected via Mellanox 4X FDR Infiniband MT27500 network controllers.

Initial experiments with Ziz have centered on investigating the degree to which the scaling performance of CHIMERA is mimicked by the MiniApp and how that scaling behavior changes when the interconnect is changed from Gemini to Aries to Infiniband. A comparison of CHIMERA weak scaling to Ziz "hydro-only" weak scaling on Titan is shown in Figure 3. The "hydro-only" qualification refers to a version of Ziz wherein only the hydrodynamics module is included, i.e. there is no active module for radiation transport or for nuclear kinetics. As expected, the weak scaling behavior of this version of Ziz becomes less efficient much more quickly than does the more physics-laden full CHIMERA runs. Previous profiling of CHIMERA has shown that the hydrodynamics calculation (including the requisite `MPI_AllToAll` communications) represents roughly 33% of the total runtime in CHIMERA for runs at less than O(30K) cores. For those same runs, roughly 45% of the runtime is dedicated to the transport solve, and $\approx$20% is spent in the nuclear burning module, the balance being spread among other routines, including I/O. The transport and nuclear burning solves also significantly increase the payload sizes for the transpose `MPI_AllToAll`s, increasing the number of double-precision REALs per zone from 6 to 180 (160 for the transport and 14 for the nuclear burning). Nevertheless, it is important to understand the hydro-only scaling of the code, to better understand the turnover in the parallel efficiency that occurs above O(30K) cores.

Comparing scaling results on different machines presents a somewhat different picture, however. Shown in Figure 4 are a subset of the same Ziz runs performed on Eos and on Rhea, along with the Titan scaling. The Eos and Rhea results are restricted in number because of the size of each of the machines and the particular modulo arithmetic that must be satisfied for each dimension in a Ziz (or CHIMERA)

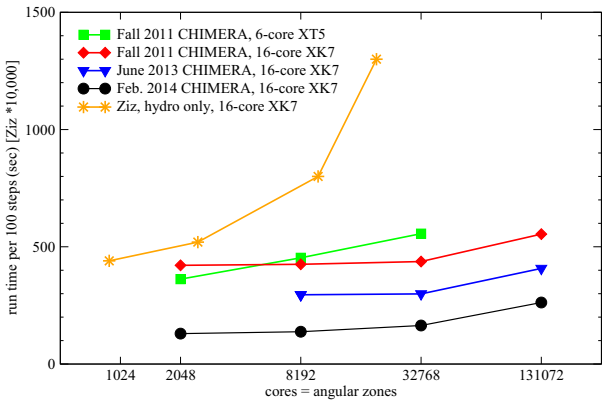

Fig. 3. A comparison of hydro-only Ziz scaling and historical CHIMERA scaling experiments. The ordinate (i.e. time required for 100 time steps) is multiplied by 10000 for the Ziz results: The time required to perform a hydro-only update is dwarfed by the transport and burning solves in the full CHIMERA code.

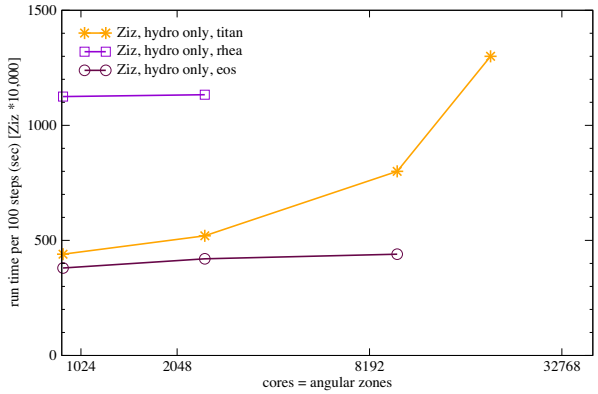

Fig. 4. A comparison of hydro-only Ziz scaling on Titan, Eos, and Rhea. The ordinate (i.e. time required for 100 time steps) is multiplied by 10000 for the Ziz results, just as in Figure 3.

run. Perhaps unsurprisingly, Eos is (a) somewhat faster for a given number of MPI ranks and (b) weakly scales all the way out to 10,000 ranks with excellent efficiency. The runtime on Rhea is significantly longer than either of the Cray platforms, but the weak scaling is not quite as bad as for Titan at low rank counts. It should be noted that the Cray platform results were both obtained with code compiled with the CCE compiler suite (CCE 8.2.2 on Titan and CCE 8.1.9 on Eos), while the Rhea results came from code compiled with Intel ifort (version 13.1.3). Neither the Eos nor the Rhea results allowed HyperThreading. Additional tests on Eos with Intel ifort (version 13.1.3) produced results within 3% of the CCE results presented in Figures 3 and 4.

B. Mimicking additional computational intensity

In an attempt to better represent the additional computation and communication load provided by the transport and nuclear kinetics modules of CHIMERA, we added a fixed number of additional floating-point operations to Ziz directly after each call to `sweepx1` and `sweepx2` (the two subroutines that handle the radial sweeps in each half-timestep, including the

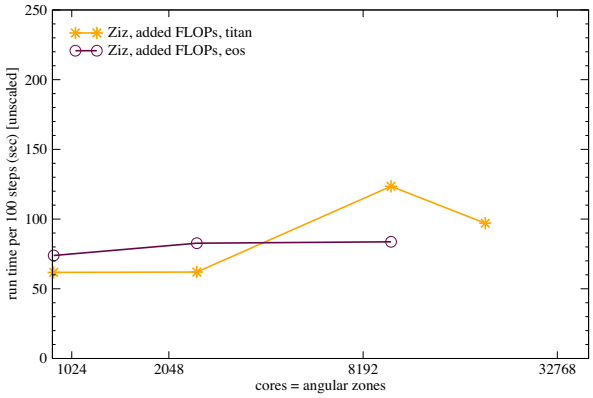

Fig. 5. A comparison of the artificially FLOP-intensive version of Ziz scaling on Titan and Eos. Note that, unlike earlier figures, the ordinate has not been scaled here.

transposes). Assuming that the preponderance of the FLOPs in the transport are found in the LU decomposition of the diagonal blocks and that the kinetics solve is similarly dominated by LU decompositions for each radial zone's network, we added roughly 70 MFLOPs/zone to Ziz at this point in the program flow. We also add a communication payload to each directional transpose that is as large as the neutrino data that would be evolved via the extra FLOPs (i.e., in our fiducial case, an additional 1.3 MB/zone). The corresponding scaling for Titan and Eos is shown in Figure 5. A curious feature emerges: Although the overall weak scalability on Titan and Eos is better with the additional local work, the overall performance of Eos relative to Titan is substantially degraded. In fact, the actual wall time to solution for Eos now *exceeds* that for Titan. Though we have no established cause for this inversion, we do offer one additional observation.

Profiling this version of Ziz with CrayPat allowed us to determine the breakdown of time dedicated to computation and communication versus the hydro-only version. This comparison is shown in Table I. `fakeflops` is the function containing the additional computational work. The stark difference in the profiles of the FLOP-rich versions on Titan and Eos suggests that load imbalance on EOS might be the culprit in the hunt for an explanation of the timing inversion.

C. Initial work on an OpenACC Ziz version

We also have a beta OpenACC version of Ziz under active development. This port is based on a version of MVH3 that was used as the "code lab" for an early 2012 OLCF GPU programming tutorial. Aside from a handful of loop reorderings that proved necessary for the port, little additional coding is required. Because Ziz relies heavily on the use of global variables declared and managed in Fortran modules (as does CHIMERA), care must be taken to properly scope `private` and `copyin` variables for `acc parallel` loops. Aside from these modest code changes, the only additional code present in the OpenACC version of Ziz is the addition of `acc parallel loop` directives around the single loops containing calls to the main PPM function (`ppmlr`) in the each of directional sweep routines (`sweepx1, sweepx2, sweepy`, and `sweepz`). `ppmlr` contains the most computa-

TABLE I. CRAYPAT PROFILING - TOP FUNCTIONS AS % OF TIME

platform	hydro only	added FLOPs
Titan	`MPI_AllToAll` = 38%, `parabola` = 20%	`MPI_AllToAll` = 45%, `fakeflops` = 45%, `parabola` < 1%
Eos	`MPI_AllToAll` = 30%, `parabola` = 14%	`MPI_AllToAll` = 78%, `fakeflops` = 19%

tionally intensive hydro routine (`parabola`), which is already highly vectorizable. The result is a 94% increase in the overall per-rank performance on Titan:

```
OpenAcc Ziz
speed = 887.4 kz/s/pe
Ziz
speed = 457.1 kz/s/pe.
```

The next step for the OpenACC port of Ziz will be to add accelerated versions of the LU decompositions performed in the transport and burning modules. Because these changes will, for the most part, be confined to formation of the individual Jacobians and the use of accelerated libraries, the expected code changes for these additions are expected to be as modest as those performed for the hydro module.

The architectural differences between the Cray XK7 and XC30 platforms include both a different processor and a different network technology. These two intertwined facets of each platform can impact the performance of even a reduced application like Ziz in unexpected ways. Layering complexity in the Ziz MiniApp will allow each of the individual pieces of multi-physics modeled by the code to be measured and analyzed separately. In addition, the pairwise and higher-order interactions between the various modules can also be delineated in this manner.

V. THE SWEEP MINIAPP: A PROXY FOR DENOVO

The sweep MiniApp is designed to be a performance proxy for the Denovo radiation transport code. Denovo solves the linear Boltzmann transport equation using the discrete ordinates method and has uses in nuclear technology applications [6]. It is part of the SCALE nuclear safety analysis code package [1] and is the primary deterministic radiation transport code employed by the Consortium for Advanced Simulation of Light Water Reactors (CASL) headquartered at Oak Ridge National Laboratory. The MiniApp and the the algorithms it is designed to capture are quite similar to the SNAP MiniApp developed at Los Alamos National Laboratory [17].

The majority of execution time for a Denovo run (typically 80-99%) is generally spent in a 3D-sweep wavefront calculation. The Denovo sweep kernel has been successfully ported to GPUs for the ORNL Titan system [2]. The sweep MiniApp considered here is a reimplementation of the Denovo sweeper to target multiple current and future architectures under a wide range of alternative programming models.

In its current version, the sweep MiniApp is comprised of roughly 6000 lines of source code, compared to Denovo's line count in excess of 200,000 lines. To allow for maximum portability to programming models such as OpenCL, which requires kernels to be implemented in C, the sweep MiniApp is written in the C language using an unobtrusive object-like

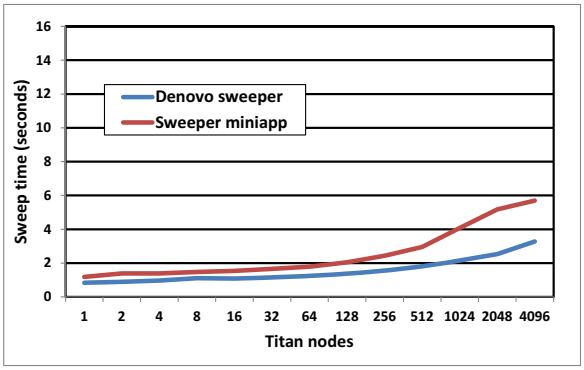

Fig. 6. A comparison of weak scaling for the Denovo sweep kernel and the sweep MiniApp.

programming style. The code currently supports MPI, CUDA, OpenMP and Intel Xeon Phi coprocessor directives; ports to OpenCL, OpenACC, and OpenMP 4 are currently in progress.

As in previous sections, we concentrate first on the distributed memory scalability of the sweep MiniApp versus that of Denovo. Figure 6 shows the comparative performance of the Denovo GPU sweeper and the sweep MiniApp for a series of test cases in a weak scaling regime. In both cases the codes exhibit good weak scaling behavior, with slight efficiency loss at high node counts, which is to be expected due to the nature of the sweep algorithm. The MiniApp runs slightly slower than the Denovo sweeper because the latter code is more mature and more heavily tuned to the GPU architecture of Titan. The excellent agreement in the scaling behavior between the two codes validates the suitability of the sweep MiniApp as a performance proxy for the Denovo sweep kernel.

The convergence of pre-exascale node architectures to a generally uniform configuration of cores, threads, and vectors makes it conceivable to support diverse architectures in a single code base. The sweep MiniApp is written to support many threads per compute node, whether they be CUDA or OpenMP threads. Figure 7 shows a strong scaling study on one Titan processor socket for a test case using OpenMP threading. Near-perfect scaling is attained with low OpenMP overhead.

Figure 8 demonstrates weak scaling performance for hybrid MPI+OpenMP execution on the NICS Beacon system using a single Intel Xeon Phi processor. These runs use four OpenMP threads per core and scale across processor cores using MPI. The code is parallelized using OpenMP and Intel SIMD directives and is run on the coprocessor in native mode. Good weak scaling performance is attained. The fraction of processor floating point peak performance achieved on the Intel Phi is similar to what is obtained on a Titan GPU.

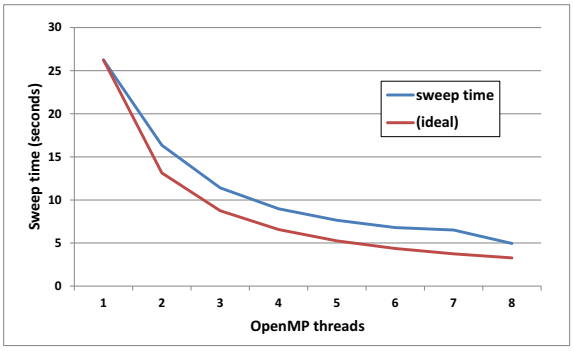

Fig. 7. OpenMP strong scaling performance for the sweep MiniApp.

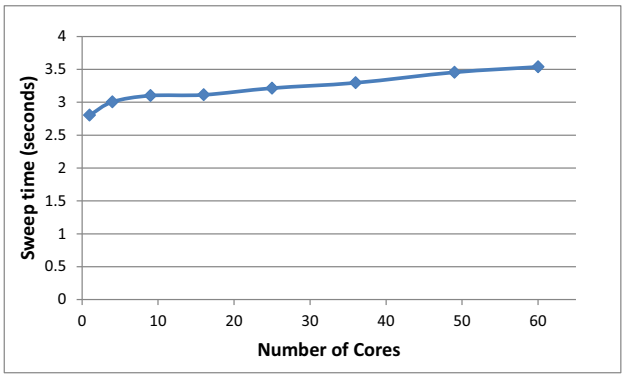

Fig. 8. Weak scaling of the sweep MiniApp on Intel Xeon Phi.

VI. Summary and Future Work

Our experience with performance benchmarking and modeling for leadership-class supercomputers has convinced us of the necessity of keeping benchmark codes up-to-date with respect to their progenitor production codes. It is this notion that has motivated the empirical tests of performance behavior between MiniApps and full applications described in this paper. This same philosophy is reflected in our plans to produce and support versions of these (and other) MiniApps in the future. We plan to release public versions of the MiniApps to various communities, including vendors, computer scientists, developers, etc., using all of the programming models presented here. These codes will have to be updated and modified as new programming models come to the fore and older systems mature and change.

VII. Acknowledgments

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This research was sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U. S. Department of Energy.

References

[1] SCALE: A modular code system for performing standardized computer analyses for licensing evaluations. Technical Report ORNL/TM-2005/39, Oak Ridge National Laboratory, Oak Ridge, TN, USA, 2009.

[2] Christopher Baker, Gregory Davidson, Thomas Evans, Steven Hamilton, Joshua Jarrell, and Wayne Joubert. High performance radiation transport simulations: Preparing for TITAN. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC12), Salt Lake City, Utah, November 10-16, 2012*, pages 1–10, 2012.

[3] Richard F. Barrett, Paul S Crozier, Douglas W Doerfler, Simon D. Hammond, Michael A Heroux, Paul T. Lin, Heidi K Thornquist, Timothy G. Trucano, and Courtenay T. Vaughan. Summary of Work for ASC L2 Milestone 4465: Characterize the Role of the Mini-Application in Predicting Key Performance Characteristics of Real Applications. *Sandia National Laboratories, Tech. Rep. SAND2012-4667*, 2012.

[4] Richard F. Barrett, Michael A. Heroux, Paul T. Lin, Courtenay T. Vaughan, and Alan B. Williams. Poster: Mini-applications: Vehicles for co-design. In *Proceedings of the 2011 Companion on High Performance Computing Networking, Storage and Analysis Companion*, SC '11 Companion, pages 1–2, New York, NY, USA, 2011. ACM.

[5] Eduardo D'Azevedo and Jack Dongarra. The design and implementation of the parallel out-of-core ScaLAPACK LU, QR, and cholesky factorization routines. *Concurrency: Pract. Exper.*, 12(15):1481–1493, dec 2000.

[6] Thomas M Evans, Alissa S Stafford, Rachel N Slaybaugh, and Kevin T Clarno. Denovo: A new three-dimensional parallel discrete ordinates code in scale. *Nuclear technology*, 171(2):171–200, 2010.

[7] Brian C. Gunter, Wesley C. Reiley, and Robert A. van de Geijn. Implementation of out-of-core cholesky and qr factorizations with pooclapack, 2000.

[8] Michael A. Heroux. Miniapplications: Vehicles for co-design. Invited talk at the Workshop on Distributed Supercomputing (SOS) 2011, Engelberg, Switzerland , 2011.

[9] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 2009.

[10] E. F. Jaeger, L. A. Berry, E. DAzevedo, D. B. Batchelor, M. D. Carter, K. F. White, and H. Weitzner. Advances in full-wave modeling of radio frequency heated, multidimensional plasmas. *Physics of Plasmas*, 9(5):1873–1881, 2002.

[11] R. Marcus. MCMini: Monte Carlo on GPGPU. *Los Alamos National Laboratory, Tech. Rep. LA-UR-12-23206*, 2012.

[12] O. E. B. Messer, S. W. Bruenn, J. M. Blondin, W. R. Hix, and A. Mezzacappa. Multidimensional, multiphysics simulations of core-collapse supernovae. *Journal of Physics Conference Series*, 125:012010, 2008.

[13] O. E. B. Messer, J. A. Harris, S. T. Parete-Koon, and M. A. Chertkow. Multicore and Accelerator Development for a Leadership-Class Stellar Astrophysics Code. *Lecture Notes in Computer Science*, 7782:92, 2013.

[14] Scott Pakin and Patrick McCormick. Hardware-independent application characterization. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. Institute of Electrical & Electronics Engineers (IEEE), 2013.

[15] R Springmeyer, C Still, M Schulz, J Ahrens, S Hemmert, R Minnich, P McCormick, L Ward, and D Knoll. From Petascale to Exascale: Eight Focus Areas of R&D Challenges for HPC Simulation Environments. *Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-474731*, 2011.

[16] Andrew Stone, John M. Dennis, Michelle Mills Strout, Andrew Stone, John M. Dennis, and Michelle Mills Strout. The CGPOP Miniapp, Version 1.0. *Colorado State University, Technical Report CS-11-103*, 2011.

[17] R. J. Zerr and R. S. Baker. SNAP: SN (discrete ordinates) application proxy: Description. Technical Report LA-UR-13-21070, Los Alamos National Laboratory, Los Alamos, NM, 2013.