Performance Characterization for Fusion Co-design Applications

Praveen Narayanan, Alice Koniges, Leonid Oliker,
Robert Preissl, Samuel Williams, Nicholas J. Wright,
Lawrence Berkeley National Laboratory
Maxim Umansky, Xueqiao Xu, *Lawrence Livermore National Laboratory*
Stephane Ethier, Weixing Wang, *Princeton Plasma Physics Laboratory*
Jeff Candy, *General Atomics*
John R. Cary, *Tech-X*

ABSTRACT: Magnetic fusion is a long-term solution for producing electrical power for the world, and the large thermonuclear international device (ITER) being constructed will produce net energy and a path to fusion energy provided the computer modeling is accurate. To effectively address the requirements of the high-end fusion simulation community, application developers, algorithm designers, and hardware architects must have reliable simulation data gathered at scale for scientifically valid configurations. This paper presents detailed benchmarking results for a set of magnetic fusion applications with a wide variety of underlying mathematical models including both particle-in-cell and Eulerian codes using both implicit and explicit numerical solvers. Our evaluation on a petascale Cray XE6 platform focuses on profiling these simulations at scale identifying critical performance characteristics, including scalability, memory/network bandwidth limitations, and communication overhead. Overall results are a key in improving fusion code design, and are a critical first step towards exascale hardware-software co-design — a process that tightly couples applications, algorithms, implementation, and computer architecture.

KEYWORDS: Application benchmarking, profiling, performance characterization, magnetic fusion

I. Introduction

Fusion energy research is a complex, international endeavor, with the next magnetic-fusion plasma confinement device (ITER [1]) to cost in excess of $10B. Prior to the construction of any such large device, there is a need to understand performance as it relates to device parameters in order to arrive at an optimum for demonstrating the next step to fusion energy. Consequently, no experimental campaign will be approved without extensive computations in advance. Thus, the high-fidelity modeling to come from exascale computing will provide major guidance for ITER and beyond. The dominant challenge in fusion modeling is to accurately simulate the wide range of temporal and spatial scales that are coupled in an experimental device. To address this challenge, the fusion community has developed sets of equations to address these scales, which have in

turn led to the development of numerous independent computational applications covering different physics, scales, and regions. Such a computational component view will be crucial for extracting science from computations, as the massive range of physical scales cannot be modeled by brute force even at the exascale.

Fusion computation is enormously difficult because of the wide range of scales in fusion devices. In fusion plasmas the fastest physical time-scale is set by electron gyro-motion, which, at the planned 6 T toroidal field, has a period of $6 \times 10^{-12} s$, while the discharge is expected to last more than $10^3 s$, resulting in a need to compute of order 10^{15} fundamental periods. Length scales also have enormous variation, the plasma shielding length being of order micrometers in the edge, while the plasma is of order 5 m across. Thus, the spatial scales span roughly 6-7 orders of magnitude. The product of these scale variations (cubed for spatial) is 10^{33} – an overwhelmingly large number. Direct simulation of all scales using the most fundamental equations is not possible on existing or any foreseeable computational platforms. Consequently, the fusion community has, over decades, developed a suite of models for tokamaks (and other confinement devices).

This work presents the first study to evaluate and analyze the behavior of several key fusion application models at scale. Specifically we examine four simulation codes on the Cray XE6 Hopper, a leading petascale supercomputing system: GTS [2], [3] which solves the gyrokinetic equation by following the guiding center orbits in flux coordinates, with the associated Poisson-solve, discretized according to an integral method; GYRO [4], [5], [6], [7], a Eulerian gyrokinetic code used to compute turbulent and collisional transport coefficients; BOUT++ [8], a 3D finite-difference code for tokamak edge plasma turbulence that is flexible in terms of the specific moment equations that are implemented; and the VORPAL [9] computational framework that computes the dynamics of plasmas, accelerators, and electromagnetic structures with wide usage across multiple plasma physics and electromagnetics application areas.

Detailed performance results are presented in terms of performance, scalability, communication behavior, and memory bandwidth sensitivity. The performance analysis of these codes is critical for application scientists, algorithm developers, and computer architects to understand and improve the behavior of next-generation fusion codes, while providing an important first step toward an integrated hardware/software co-design of optimized fusion simulations.

II. Experimental Methodology

A. Application Suite

For this work, we examine the performance and scalability of four key apps for fusion simulation: GTS, GYRO, BOUT++, and VORPAL. Each application plays a different role in fusion simulation.

B. Cray XE6 "Hopper"

Hopper is the newest Cray XE6 built from dual-socket, 12-core "Magny-cours" Opteron compute nodes. In reality, each socket (multichip module) has two dual hex-core chips, making each compute node effectively a four-chip compute node with strong NUMA properties. Each Opteron chip instantiates six superscalar, out-of-order cores capable of completing one (dual-slot) SIMD add and one SIMD multiply per cycle. Without proper SIMD code generation, instruction-level parallelism, or high arithmetic intensity, it will be difficult to attain 10% of this throughput. Additionally each core has private 64 KB L1 and 512 KB L2 caches. L2 latency is small and easily hidden via out-of-order execution. The six cores on a chip share a 6 MB L3 cache and dual DDR3-1333 memory controllers capable of providing an average STREAM [10] bandwidth of about 2.0 GB/s per core. Each pair of compute nodes (8 chips) shares one Gemini network chip. Like the XT series, the Gemini chips of the XE6 form a 3D torus. The MPI pingpong latency is typically 1.6 μs and, when using 4 or more MPI tasks per node, the MPI bandwidth is 9.0 GB/s.

C. Programming Model

In this paper, we nominally use the flat MPI programming model. In the case of Hopper, this implies one MPI process on each core (24 processes per node). We exploit the hybrid MPI+OpenMP programming model on GTS as it is the only application that supports it.

D. Performance tools

In order to measure performance characteristics, it is necessary to instrument the code with an approriate performance tool to give pertiment information (such as HWC, MPI statistics, the role played by synchronization calls, cache usage, etc.). In our performance experiments, in addition to recording the runtimes we also instrument them using the Integrated Performance Monitoring (IPM) framework [11], [12], or CrayPAT [13] for the four different applications profiled.

IPM provides an easy to use, low overhead mechanism for obtaining information about the MPI performance characteristics of an application. It uses the profiling interface of MPI (PMPI) to obtain information about the time taken and type of MPI calls, the size of the messages sent and the message destination. Previous measurements have shown that the overhead of using IPM is significantly less that 1%, which makes us confident we are not perturbing the applications by instrumenting them. Instrumenting the code with IPM is a relatively simple matter of adding a linker line for the IPM library. Three of the four codes profiled (GTS, GYRO, VORPAL) used IPM to obtain performance measurements.

CrayPAT is a high-level performance analysis tool for profiling codes in Cray architectures. These tools provide a simple interface to hardware counter and other instrumentation data. After loading the appropriate CrayPAT modules (available on NERSC machines via the perftools module, which loads associated modules and declares environment variables), users may rebuild their code, instrument the executable with the pat_build command, and receive reports from their application runs. CrayPAT captures performance data without the need for source code modifications. Event tracing was used to gather data during these experiments. Tracing records events at function entry and exit points, rather than interrupting execution periodically to capture events. Various function groups may be traced (e.g. MPI, heap, PGAS) for performance. CrayPAT leverages hardware-level performance counters, by means of the Performance API (PAPI) library, which can give such performance information as floating point instructions and cache usage.

For the BOUT++ application we used the CrayPAT profiling infrastructure to obtain performance information. Although CrayPAT overhead can be significant if a large number of function groups are monitored, in the current runs only MPI performance was monitored, which resulted in negligible overhead ($< 1\%$). CrayPAT also has some support for PGAS languages (so far, implemented only in GTS). However, it was seen that the overhead associated with monitoring them was considerable, and we have therefore postponed PGAS performance analysis to the future.

Both IPM and CrayPAT provide options to instrument code (invasively) so as to focus attention to particular sections, where necessary. During the runs, we ensured that the initialization region was discarded from the performance measurements. This is most important in the GTS code where it was seen that initialization overhead is disproportionately large for the PETSc library which the code links to. However, for the other codes, the initialization was not seen to consume much time.

E. DRAM Bandwidth Sensitivity

The memory wall is a well-known impediment to application-performance. Nominally, it comes in two forms: latency-limited and bandwidth-limited. The former is an artifact of the latency to DRAM (measured in cycles) far outstripping the ability for a core to inject sufficient memory-level parallelism to satisfy Little's Law. For many codes, this has been mitigated by hardware stream prefetchers, multiple cores, and good programming practices. The latter, memory bandwidth, is a more fundamental impediment arising from the fact that cores can process data faster than the memory subsystem can supply it. Recent history and technology trends indicate that this imbalance is only going to get worse. To that end, it is imperative we understand how existing applications fare when confronted with reduced memory bandwidth.

To proxy this phenomenon, we increase the number of cores contending for the finite DRAM bandwidth. As Hopper is built from 6-core chips, by increasing the number of active cores per chip from 1 to 6, we reduce the per-core bandwidth from about 12GB/s to 2GB/s. By using `aprun`'s default behavior, we fill all cores within a compute node's first NUMA node with processes before moving to a compute node's second (, third, or fourth) NUMA node. Thus, by confining the number of processes per node to between 1 and 6 for flat MPI

codes, we can scale the number of processes per chip and thereby test the application's sensitivity to reduced bandwidth. Similarly, by confining the number of processes per node to 1 and varying `OMP_NUM_THREADS` from 1 to 6, we can realize the same effect on hybrid applications.

In general, if performance remains constant as we reduce the bandwidth per core, then we may conclude that the application is relatively insensitive to memory bandwidth. Conversely, if run time increases linearly with reduced bandwidth, we may conclude the application's performance is dominated by memory bandwidth [14]. In practice applications are built from multiple kernels, some will be memory-intensive, others will be compute-intensive. When plotted, time should be a linear function with an offset. The relationship between offset and slope is indicative of the room for additional cores or performance optimization. If there is little, it is imperative that application designers develop and integrate new methods lest future performance be limited by the lethargic trends in memory bandwidth. In these experiments, concurrency is kept low to ensure MPI communication does not skew our analysis.

III. GTS

GTS solves the gyrokinetic equations by following the guiding centers of particle orbits using the particle-in-cell (PIC) method. In PIC codes, rather than directly modeling the pairwise interactions between particles, particles interpolate charge onto a grid, one solves Poisson's equation to determine the electrostatic potential, and uses the potential to accelerate and move particles. To account for gyrating particles, GTS models particles with a charged ring and interpolates this ring onto the charge grid using a 4-point averaging scheme. The Poisson solve is implemented using PETSc. GTS has 3 levels of parallelism: a one-dimensional domain decomposition in the toroidal direction, a particle decomposition within each poloidal domain, and finally loop-level multi-threading. The domain and particle distributions are implemented with MPI, while the loop-level multi-threading is implemented using OpenMP directives.

A. Simulation Parameters

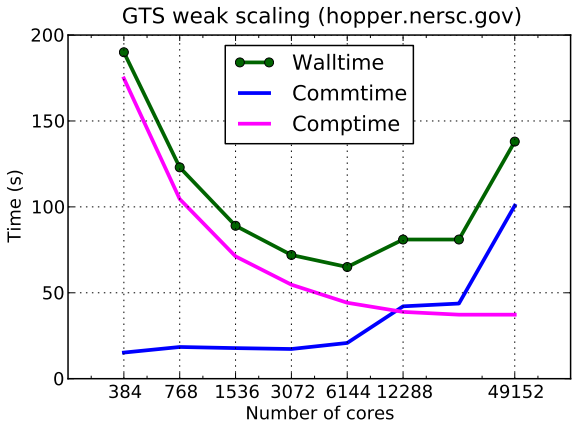
In GTS, weak scaling experiments were conducted on the NERSC Hopper machine by keeping the number of particles per process constant while maintaining a constant overall grid resolution for all concurrencies. Thus the time spent operating on particles is expected to be roughly constant, while the overhead of reducing each copy of the poloidal plane is expected to scale with the number of processes. The Poisson solve's overhead is expected to decrease with increased concurrency as the size of the grid remains constant.

As each Hopper compute node is effectively 4 NUMA nodes each of 6 cores, we limit OpenMP parallelism to 6-way to avoid NUMA issues. Experiments scale up to 8,192 processes (each using 6 threads) for a total of 49,152 cores.

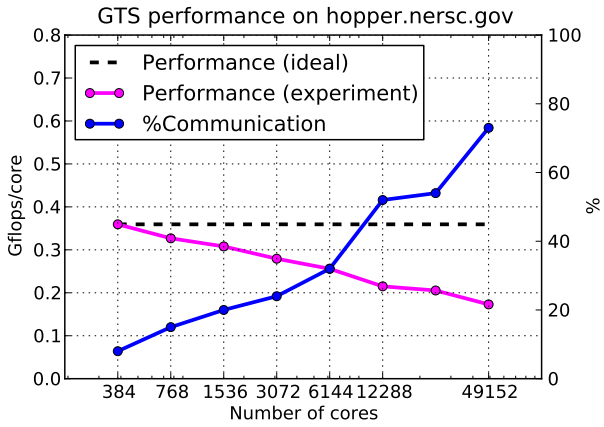
B. Scalability and Analysis

Performance for the GTS weak scaling experiments is shown in Figures 1(a) and 1(b). Figure 1(a) presents the time (measured in seconds) spent in computation and communication running GTS with increasing levels of concurrency. Figure 1(b) reveals the GTS runtime behavior for a similar range of concurrencies (x-axis) as depicted in Figure 1(a), but highlights the GTS overall performance ("Performance (experiment)") in flop/s (normalized to a per core basis, vertical values correspond to the left y-axis). Figure 1(b) also gives the ratio of communication time ("%Communication") to the GTS wall-clock time (vertical values correspond to the right y-axis). Results show the increasing communication overhead in GTS with higher levels of concurrency. In addition, we can see that the runtime first decreases at higher scale, but then begins to rise at 6,144 processing cores.

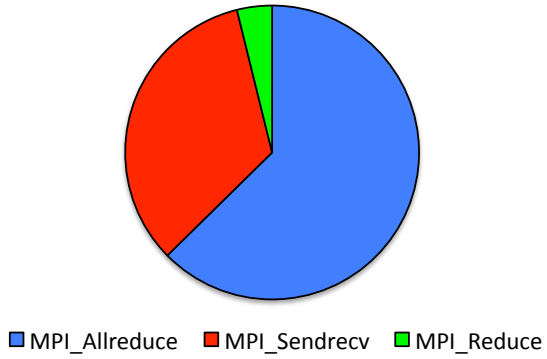
GTS consists of both particle based (PIC) kernels and a grid-based Poisson kernel. However, weak-scaling is only applied in the context of the PIC kernels, while the grid-based Poisson solver exhibits strong-scaling behavior that reaches a performance plateau at 6,144 processors. Beyond a concurrency of 6,144, the influence of the Poisson step becomes mostly insignificant, and subsequent scaling is dominated by the PIC kernels. Unfortunately, the 1D grid decomposition strategy necessitates a "replicate-and-reduce" approach to PIC. Thus, the reduction of per-process copies of poloidal planes into one plane results in a increase in communication time. These two scaling trends (Poisson solve and reductions) result in the decreasing computation trend and the increasing communication



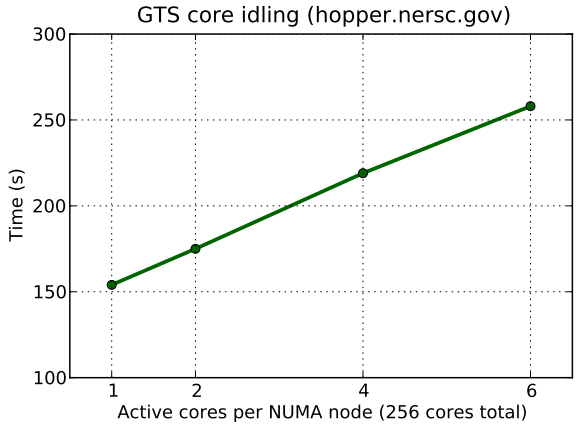
(a) Scalability



(b) Performance per core and communication overhead



(c) MPI Time Breakdown



(d) Bandwidth Sensitivity

Fig. 1. GTS Performance Characteristics on the Cray XE6. Figures 1(a), 1(b) show *weak* scaling performance degradation at higher concurrency from increased communication overhead. Figure 1(a) shows overall walltime, time spent in communication and time spent in computation. Figure 1(b) presents the flops performance (Y-axis on left) and the communication as percentage of runtime (Y-axis on right). Figure 1(c) shows the role of collectives in the breakdown of MPI functions at high concurrency (49,152). Figure 1(d) shows degradation of performance due to increased DRAM bandwidth contention.

trend shown in Figure 1(a). The sum of these two scaling trends results in a minimum at around 6K cores. To verify this, we inspect the time spent in the reduction collective (`MPI_Allreduce`) and show that at extreme scales, it is the dominant communication routine (Figure 1(c)). The net result is that communication time eventually reaches more than 70% of the run time. This depresses the average flop rate considerably.

An examination of MPI communication for the highest concurrency experiment running at 49,152 processing cores (Figure 1(c)) reveals the most time consuming MPI functions - MPI_Allreduce, MPI_Sendrecv and MPI_Reduce. MPI functions with lower impact on overall communication time ($<$ 5%) are exempt from this analysis. The above mentioned reduction of per-process copies of poloidal planes requires communication intense MPI reduction operations at each GTS iteration step. Besides an increasing memory footprint due to the "replicate-and-reduce" approach, the increasing communication overhead of grid reduction operations makes a 2D domain decomposition necessary in GTS towards scaling to higher concurrencies. The Sendrecv communication originates from shifting particles between adjacent toroidal domains due to the one-dimensional domain decomposition in the toroidal direction. Advanced communication techniques such as one-sided messaging (Partitioned Global Address Space languages or the MPI-2 extensions) or non-blocking MPI send and receive functions might reduce the costs of this phase in GTS.

C. DRAM Bandwidth Sensitivity

The sensitivity to DRAM memory bandwidth contention is tested by running the codes for a given concurrency (the number of MPI processes is constant in this experiments) but by spreading these MPI processes over different nodes. Using this approach, the contention for memory bandwidth in a given node can be varied. Where applicable (only GTS has OpenMP implemented) OpenMP threading is turned off, so that only pure MPI is used. The number of cores used per node are thus varied from 1 to 6. A notable aspect here is that one must *pack* the MPI processes within an individual NUMA node containing 6 cores, thereby ensuring that the experiment adequately measures bandwidth contention. In that respect, it would be inaccurate to distribute the MPI processes in different nodes since they would then not compete for bandwidth. And hence, although a hopper node consists of 24 processing cores, the most straight forward configuration for a memory bandwidth contention experiment is the foregoing. Distributing MPI processes in different NUMA nodes might bring in other aspects into play, such as the inter-node communication, which are not particularly germane to the task of measuring sensitivity to bandwidth. Memory bandwidth contention experiments in GTS indicate that the code is affected by bandwidth contention, as seen from figure1(d). The runtime increases (or equivalently, the flops would decrease) as as more cores per NUMA node are used, implying increased bandwidth contention by the MPI processes in the NUMA node. We see run time increase by about 66% when all 6 cores in the NUMA node contend for bandwidth. Moreover, as contention is increased we see the application spends progressively more of its run time in bandwidth-intensive routines (from about 14% up to 50%). Such a strong corelation to bandwidth will impede its ability to exploit more cores or optimization.

IV. GYRO

GYRO [4], [5], [6], [7] is an Eulerian gyrokinetic code used to model turbulence and collisional transport using either an explicit or semi-implicit time disrectization and a 5D phase-space (2D grid, spectral, 2D velocity) discretization using a mixture of finite-difference, finite-element, spectral, and pseudo-spectral methods. Dense and sparse linear solvers are hand-coded using LAPACK and UMFPACK/MUMPS, respectively. Depending on scale, either an hand-coded convolution or FFTW is used to evaluate the nonlinear Poisson bracket.

A. Simulation Parameters

We conducted weak scaling experiments ranging from 128 to 8,192 processes. This is accomplished through the job-manager TGYRO, which can execute multiple instances of GYRO and run them together. Because of a nonlinear convolution operation in toroidal harmonics, the total work performed in this scan is a weakly nonlinear function of problem size.

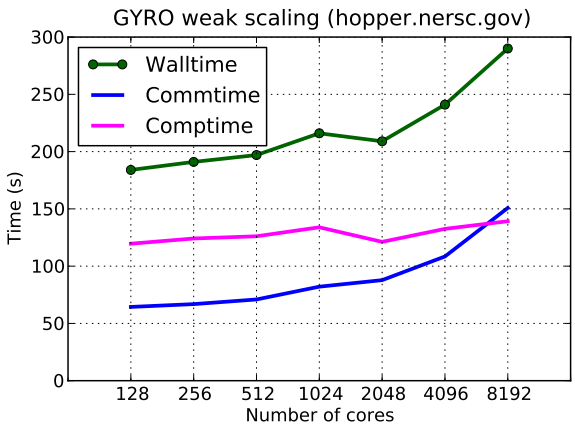
B. Scalability and Analysis

Performance figures for the GYRO weak scaling experiments are presented in Figures 2(a) and 2(b). We observe a marked degradation in overall performance beyond 2K processes (less than 60% parallel efficiency at 8K cores). We observe that although the computation time remained roughly constant across this range, the communication time grew quickly and ultimately begins to impede scalability. Application-level performance degradation is observed when this increase becomes a critical bottleneck (over 50% of the runtime). Despite a slight increase due to the convolutions, the computation time remained relatively flat.

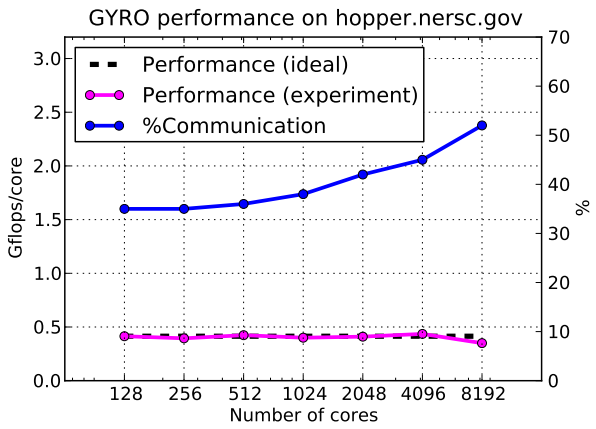
The breakdown of MPI calls in Figure 2(c) at highest concurrency of 8,192 processes indicates that MPI collective communication, especially MPI_Allreduce and MPI_Alltoall, constitutes a major overhead. The large number of MPI_Alltoall calls are likely to be latency limited due to their small message sizes (2KB).

C. DRAM Bandwidth Sensitivity

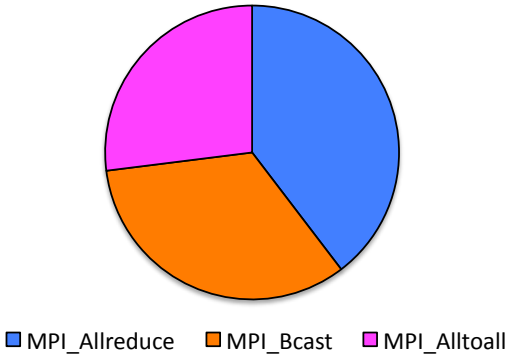
Figure 2(d) shows that as the number of processes per NUMA node increases, contention for DRAM bandwidth results in an increase in runtime. A linear extrapolation suggests that with 1 process per NUMA node, bandwidth plays a relatively small (less than 8%) part in performance. However, as the number of processes per NUMA node grows to 6, we see more than a 60% increase in runtime. At that point, perhaps half the runtime is due to the a lack of scaling on a bandwidth-instensive component of the



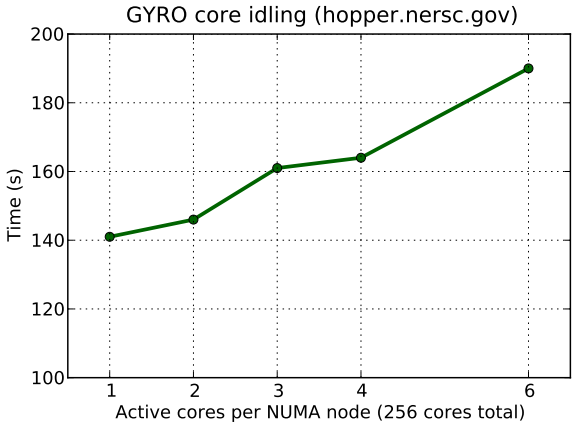
(a) Scalability



(b) Performance per core and communication overhead



(c) MPI Time Breakdown



(d) Bandwidth Sensitivity

Fig. 2. GYRO Performance Characteristics. Figures 2(a), 2(b) show *weak* scaling performance degradation at higher concurrency from increased communication overhead. Figure 2(a) shows overall walltime, time spent in communication and time spent in computation. Figure 2(b) presents the flops performance (Y-axis on left) and the communication as percentage of runtime (Y-axis on right). Figure 2(c) shows the role of collectives in the breakdown of MPI functions at the highest concurrency run (8,192).. Figure 2(d) shows degradation of performance due to increased bandwidth contention.

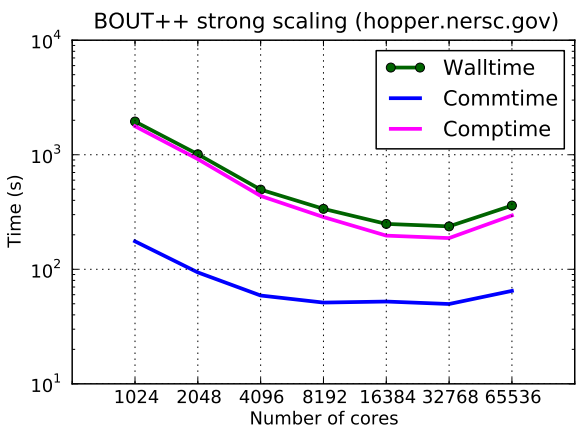
application. Thus, with 6× the active cores, we see only 3.5× the aggregate performance per NUMA node.

V. BOUT++ RESULTS

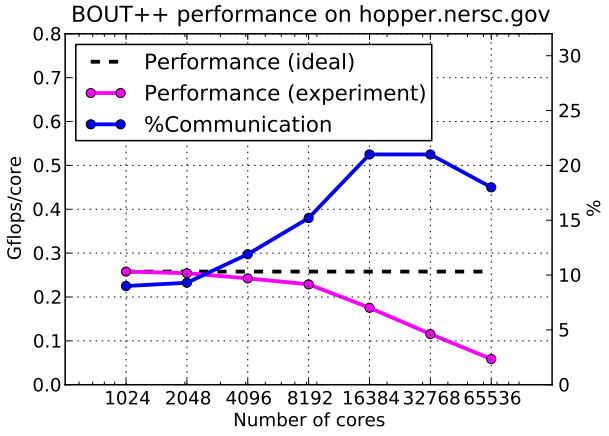
BOUT++ [8] is a 3D finite-difference structured grid code used to model collisional edge plasmas in a toroidal/poloidal geometry. Time evolution is primarily through the implicit Newton Krylov method. A range of finite difference schemes are used, including 2^{nd} and 4^{th} order central difference, 2^{nd} and 4^{th} order upwinding and 3^{rd} order WENO. Several different algorithms are implemented for Laplacian inversion of vorticity to get potential, such as a tridiagonal solver (Thomas algorithm), a band-solver (allowing 4th order differencing), and the Parallel Diagonal Dominant (PDD) algorithm. The code uses a 2D parallelization in the x, y directions. There is no parallelization in the z direction. This is transformed into the tokamak coordinate framework ψ, θ, ζ by means of a "ballooning" transformation [15], [16].

A. Simulation Parameters

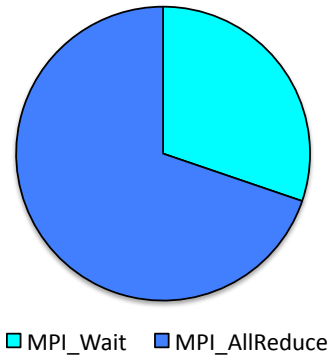
In BOUT++, strong scaling experiments runs are carried out by keeping the total number of grid points in the radial, poloidal and toroidal directions constant. Unlike other applications discussed in this paper, BOUT++ is evaluated in the strong scaling regime as that is how the developers typically use it. The experiments are conducted on up to 65,536 processor cores. Since the domain decomposition is in the toroidal and poloidal directions (2D), the size of each MPI subdomain becomes smaller with



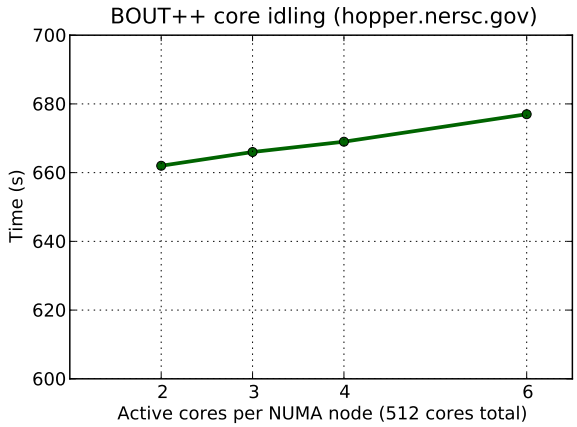
(a) Scalability



(b) Performance per core and communication overhead



(c) MPI Time Breakdown



(d) Memory Bandwidth Sensitivity

Fig. 3. BOUT++ Performance Characteristics on the Cray XE6. Figures 3(a), 3(b) show *strong* scaling performance degradation at higher concurrency. Figure 3(a) shows overall walltime, time spent in communication and time spent in computation. Figure 3(b) presents the flops performance (Y-axis on left) and the communication as percentage of runtime (Y-axis on right). MPI communication time and percentage do not grow at high concurrency demonstrating that performance degradation at high concurrency is related to the increasing time spent in computation. Figure 3(c) shows the role of MPI collectives at high concurrency (65,536). Figure 3(d) illustrates that BOUT++ is relatively insensitive to bandwidth contention.

increased concurrency, resulting in more *boxes* with a fewer number of grid points per MPI box. This results in an increase of the surface to volume ratio (at the highest concurrency of 65,536 there are only two grid points in the poloidal direction). Clearly it may be inefficient to run at this regime, but nonetheless provides an interesting insight into performance trends at high concurrency.

B. Scalability and Analysis

Performance for the strong scaling experiments are presented in Figures 3(a) and 3(b). Note, Figure 3(a) is plotted on a log-log scale. We observe good scaling behavior to about 8,192 cores and slight performance degradation at higher levels of concurrency. More detailed analysis shows that BOUT++ performs more calculations than expected for a perfectly strong-scaled experiment. This will be a subject of future investigations.

Examining BOUT++ communication overhead in Figures 3(a) and 3(b) shows that computation time scales well to about 16K cores, while communication time begins to plateau at around 4K cores. In fact, the fraction of time spent in communication rapidly increases beyond 2K cores. Interestingly, it saturates at about 20% of the runtime at 16K cores and beyond. As the parallelization scheme reaches its limits, we believe the communication overhead is likely an artifact of the asymptotic limits to the surface to volume ratio. 3D decompositions or threaded implimentations may mitigate the impact of communication. Future work will explore

optimization opportunities within the computational and communication components.

C. DRAM Bandwidth Sensitivity

Figure 3(d) shows the run time as we increase the number of active cores per chip, or in other words, increase contention for the limited per-chip DRAM bandwidth. Although we observe a linear relationship, a breakdown into bandwidth- and non-bandwidth-intensive components suggests the application initially spends 99% of its time in the non-bandwidth-intensive component. As the number of active cores increases, performance per chip increases quickly, and this faction drops to perhaps 96%. Clearly, DRAM bandwidth plays a small role in BOUT++ performance.

VI. VORPAL RESULTS

The VORPAL computational framework [9] models the dynamics of plasmas, accelerators, and electromagnetic structures including RF heating [17] in fusion plasmas. Like GTS, VORPAL uses the particle-in-cell (PIC) method to track individual particles. However, unlike GTS, VORPAL does not use the 4-point charge ring averaging scheme. VORPAL uses a 3D cartesian domain decomposition using MPI, such that one may construct the domain as a collection of rectangular slabs in three dimensions. Each domain is described by two slabs, one which is referred to as the physical region, which contains all the local cells of each MPI process. The other domain, which is referred to as the extended region, is the physical region plus one layer of guard cells in each direction. The cells which are communicated between neighboring processor is simply the slab that is the intersection of the sending processor's physical region with the receiving processors extended region.

A. Simulation Parameters

In VORPAL, weak scaling experiments were run on the NERSC Hopper machine. The physics being done in these experiments is the wave propagation in vacuum. For this particular case, it allows to compute the resonant modes of a rectangular cavity, but when combined with embedded boundaries, we can also compute the modes and frequencies of metallic and dielectric structures to high precision, up to parts in 10^5 when combined with Richardson extrapolation.

We conducted weak scaling experiments, with a fixed domain size of 40^3 cells per MPI-subdomain. There are no particles, only fields, and those fields are updated using the Yee scheme. The update is explicit with communication overlapped by independent computation. There is no global communication. The only communication is of surface field values (10088 cells and 30264 field values per field), and this compares with the volume computation of 64000 cells or 192000 field values per field. Thus, the ratio of communication to computation remains constant as we scale the number of processing cores. The underlying experiments were conducted on up to 65,536 cores on Hopper.

B. Scalability and Analysis

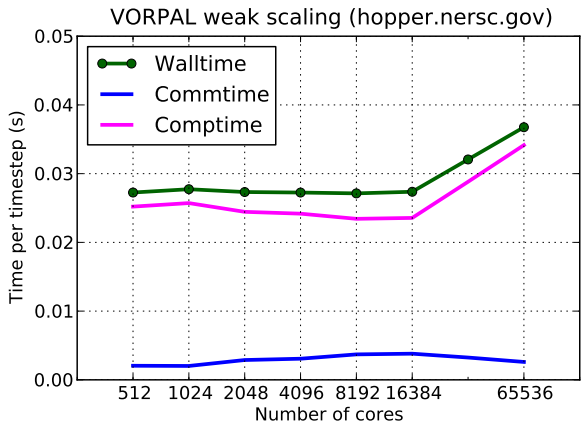
Performance figures for the weak scaling experiments are presented in Figures 4(a) (wall clock time per time step) and 4(b). We observe linear scaling on up to 16K cores, then a marked decrease in compute performance. The result depresses parallel efficiency to about 71%.

The communication overhead, as seen from Figures 4(a) remains roughly constant with increasing concurrency and represents a small fraction of the overall time (less than 13%). The decrease in the time spent in communication is primarily due to an increase in computational overhead. This increase in runtime at the two highest levels of concurrencies seems anomalous and will be the subject of future investigations. Nevertheless, there is a small decrease in communication time at extreme scales.

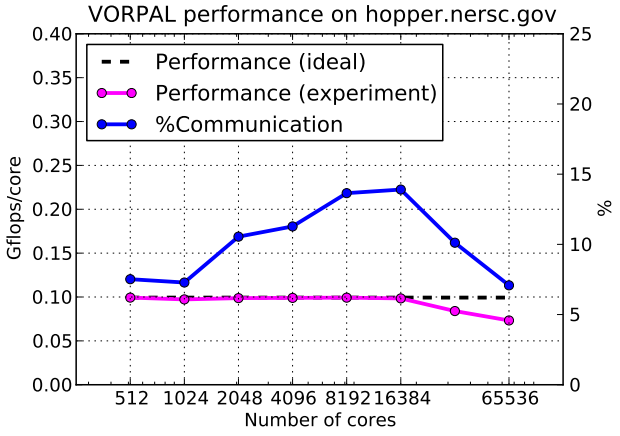
The MPI overhead breakdown chart in Figure 4(c) shows that communication time is dominated by the MPI_Waitany and MPI_Wait routines. This indicates a heavy use of non-blocking send and receive operations with time being tabulated in the waits.

C. DRAM Bandwidth Sensitivity

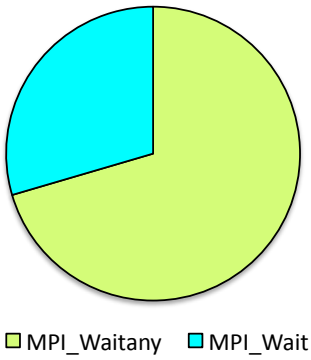
Figure 4(d) visualizes the impact of DRAM bandwidth contention on performance. We observe a linear increase in time with respect to concurrency. When studying the application breakdown into compute-intensive and bandwidth-intensive components, we observe that for 6 MPI processes per NUMA node, roughly two-thirds of the run time is



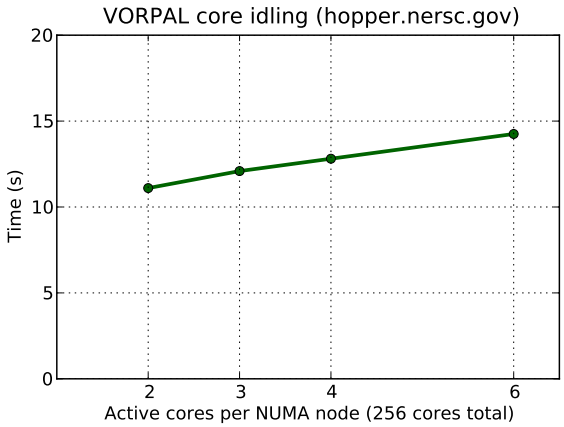
(a) Scalability



(b) Performance per core and communication overhead



(c) MPI time breakdown.



(d) Memory Bandwidth Sensitivity

Fig. 4. VORPAL Performance Characteristics. Figures 4(a), 4(b) show *weak* scaling performance degradation at higher concurrency. Figure 4(a) shows overall walltime, time spent in communication and time spent in computation. Figure 4(b) presents the flops performance (Y-axis on left) and the communication as percentage of runtime (Y-axis on right). MPI communication time and percentage do not grow at high concurrency demonstrating that performance degradation at the two highest concurrencies (32,768, 65,536) is owing to increase in time spent in computation. Figure 4(c) shows the role of collectives in the breakdown of MPI functions at high concurrency (65,536). Figure 2(d) shows degradation of performance due to increased bandwidth contention.

spent in compute-intensive components. Nevertheless, we observe a 50% increase in runtime coupled with a 300% increase in work, which results in a 2× increase in aggregate performance per NUMA node.

VII. Conclusions

In this paper, we examined four key fusion applications at high levels of concurrencies on the NERSC Hopper Cray XE6 machine. We observe that each code has its own fundamental impediments to scalability arising from various communication bottlenecks. For example, GTS's multinode scalability is impeded by the charge grid reductions (a collective operation) and the effectively strong scaled grid component. Such bottlenecks can be remedied by higher particle densities or by moving to a 2D or 3D spatial decomposition. Similarly, the collective impediments on GYRO might be mitigated by migrating to a hybrid programming model. Conversely, VORPAL shows good scalability up to 16K processes, but scaling seems to be limited by increasing computation beyond there. Studying BOUT++ reveals similar challenges with performance degradation beyond 32K processes for both communication and computation. Even if the future holds that weak scaling is only applied at the node-level and multicore is applied in a strong scaling regime, it is imperative we explore remedies for these scalability impediments as future exascale machines will likely have 100K-1M nodes (processes).

As future manycore architectures will likely see a 10-100× increase in on-chip parallelism, we must investigate the viability for these codes to effectively exploit such parallelism. History has shown that memory bandwidth is and will be a major impediment to effective exploitation of multicore. To that end, we constructed an experiment that allowed us to explore each application's sensitivity to reduced memory bandwidth. We observe that GTS is particularly sensitive to memory bandwidth (time increased quickly with increasing contention) while BOUT++ showed very little sensitivity (time was nearly constant). From these data points, we can extrapolate the factor by which we can increase performance without increasing bandwidth. We believe that performance cannot be enhanced (via either optimization, wide SIMD, or many more cores) by more than a factor 2×, 3×, 20×, and 3× on GTS, GYRO, BOUT++, and VORPAL respectively without substantial increases in memory bandwidth or fundamental changes to data structure or algorithm. Moreover, such performance gains are only attainable with a vast increase in memory capacity per FPU ratio — an unlikely scenario given exascale trends. This, in conjunction with the the meager increases expected in memory bandwidth necessitate co-design of applications (algorithms, parallelization, optimization, and architecture) to ensure we can extract larger performance enhancements.

ACKNOWLEDGMENTS

All authors from Lawrence Berkeley National Laboratory were supported by the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231. Jeff Candy is supported by GA DOE contract number DE-FG03-95ER54309. All authors from Lawrence Livermore National National Laboratory were supported by the DOE contract number DE-AC52-07NA27344. We acknowledge the help of the VORPAL team, D. Alexander, K. Amyx, T. Austin, G. I. Bell, D. L. Bruhwiler, R. S. Busby, J. Carlsson, J. R. Cary, E. Cormier-Michel, Y. Choi, B. M. Cowan, D. A. Dimitrov, M. Durant, A. Hakim, B. Jamroz, D. P. Karipides, M. Koch, A. Likhanskii, M. C. Lin, J. Loverich, S. Mahalingam, P. Messmer, P. J. Mullowney, C. Nieter, K. Paul, I. Pogorelov, V. Ranjbar, C. Roark, B. T. Schwartz, S. W. Sides, D. N. Smithe, A. Sobol, P. H. Stoltz, S. A. Veitzer, D. J. Wade-Stein, G. R. Werner, N. Xiang, C. D. Zhou. The authors also wish to thank Harvey Wasserman and David Skinner for meaningful discussions.

REFERENCES

[1] "ITER Organization. ITER: The Way to New Energy," http://www.iter.org.

[2] W. X. Wang, P. H. Diamond, T. S. Hahm, S. Ethier, G. Rewoldt, and W. M. Tang, "Nonlinear flow generation by electrostatic turbulence in tokamaks," *Physics of Plasmas*, vol. 17, no. 7, pp. 072 511–+, Jul. 2010.

[3] W. X. Wang, Z. Lin, W. M. Tang, W. W. Lee, S. Ethier, J. L. V. Lewandowski, G. Rewoldt, T. S. Hahm, and J. Manickam, "Gyro-kinetic simulation of global turbulent transport properties in tokamak experiments," *Physics of Plasmas*, vol. 13, no. 9, p. 092505, 2006.

[4] J. Candy and R. Waltz, "An Eulerian gyrokinetic-Maxwell solver," *J. Comput. Phys.*, vol. 186, p. 545, 2003.

[5] ——, "Anomalous transport in the DIII-D tokamak matched by supercomputer simulation," *Phys. Rev. Lett.*, vol. 91, pp. 045 001–1, 2003.

[6] ——, "Velocity-space resolution, entropy production and upwind dissipation in Eulerian gyrokinetic simulations," *Phys. Plasmas*, vol. 13, p. 032310, 2006.

[7] J. Candy, R. Waltz, S. Parker, and Y. Chen, "Relevance of the parallel nonlinearity in gyrokinetic simulations of tokamak plasmas," *Phys. Plasmas*, vol. 13, p. 074501, 2006.

[8] B. Dudson, M. Umansky, X. Xu, P. Snyder, and H. Wilson, "BOUT++: a framework for parallel plasma fluid simulations," *Comput. Phys. Commun.*, vol. 180, p. 1467, 2009.

[9] C. Nieter and J. Cary, "VORPAL: a versatile plasma simulation code," *Journal of Computational Physics*, vol. 196, no. 2, pp. 448–473, 2004.

[10] "STREAM: Sustainable Memory Bandwidth in High Performance Computers," http://www.cs.virginia.edu/stream.

[11] D. Skinner, "Integrated Performance Monitoring: A portable profiling infrastructure for parallel applications," in *Proc. ISC2005: International Supercomputing Conference*, vol. to appear, Heidelberg, Germany, 2005.

[12] N. J. Wright, W. Pfeiffer, and A. Snavely, "Characterizing parallel scaling of scientific applications using IPM," in *The 10th LCI International Conference on High-Performance Clustered Computing*, March 10-12, 2009.

[13] "Cray Performance Analysis Tools Release Overview and Installation Guide," http://docs.cray.com/books/S-2474-52/S-2474-52.pdf.

[14] John Levesque and Jeff Larkin and Martyn Foster and Joe Glenski and Garry Geissler and Stephen Whalen and Brian Waldecker and Jonathan Carter and David Skinner and Helen He and Harvey Wasserman and John Shalf and Hongzhang Shan and Erich Strohmaier, "Understanding and mitigating multicore performance issues on the amd opteron architecture," LBNL Report 62500, March 2007.

[15] X. Q. Xu, M. V. Umansky, B. Dudson, and P. B. Snyder, "Boundary plasma turbulence simulations for tokamaks," *Communications in Computational Physics*, vol. 4, pp. 949–979, 2008.

[16] M. V. Umansky, X. Q. Xu, B. Dudson, and P. B. Snyder, "Status and verification of edge plasma turbulence code bout," *Computer Physics Communications*, vol. 180, pp. 887–903, 2009.

[17] N. Xiang and J. Cary, "Second-Harmonic Generation of Electron-Bernstein Waves in an Inhomogeneous Plasma," *Physical review letters*, vol. 100, no. 8, p. 85002, 2008.