Gyrokinetic Toroidal Simulations on Leading Multi- and Manycore HPC Systems

Kamesh Madduri[1], Khaled Z. Ibrahim[1], Samuel Williams[1], Eun-Jin Im[3], Stephane Ethier[2], John Shalf[1], Leonid Oliker[1]

[1] *NERSC/CRD, Lawrence Berkeley National Laboratory, Berkeley, USA*
[2] *Princeton Plasma Physics Laboratory, Princeton, USA*
[3] *School of Computer Science, Kookmin University, Seoul 136-702, Korea*
{ *KMadduri, KZIbrahim, SWWilliams, LOliker, JShalf* }*@lbl.gov, eunjin.im@gmail.com, ethier@pppl.gov*

ABSTRACT

The gyrokinetic Particle-in-Cell (PIC) method is a critical computational tool enabling petascale fusion simulation research. In this work, we present novel multi- and manycore-centric optimizations to enhance performance of GTC, a PIC-based production code for studying plasma microturbulence in tokamak devices. Our optimizations encompass all six GTC sub-routines and include multi-level particle and grid decompositions designed to improve multi-node parallel scaling, particle binning for improved load balance, GPU acceleration of key subroutines, and memory-centric optimizations to improve single-node scaling and reduce memory utilization. The new hybrid MPI-OpenMP and MPI-OpenMP-CUDA GTC versions achieve up to a 2× speedup over the production Fortran code on four parallel systems — clusters based on the AMD Magny-Cours, Intel Nehalem-EP, IBM BlueGene/P, and NVIDIA Fermi architectures. Finally, strong scaling experiments provide insight into parallel scalability, memory utilization, and programmability trade-offs for large-scale gyrokinetic PIC simulations, while attaining a 1.6× speedup on 49,152 XE6 cores.

Keywords

Particle-in-Cell, multicore optimization, hybrid programming

1. INTRODUCTION

The HPC community is grappling with some of the most radical changes in computer architecture in decades due to the multicore revolution. Chip-power limitations are driving a rapid evolution of node architecture designs with complex trade-offs between performance, parallelism, power, and productivity. To quantify the strengths and weaknesses of emerging architectural solutions, it is imperative that we investigate the optimization of full-scale scientific applications.

This work explores optimizations across a variety of architectural designs for a production fusion simulation application called Gyrokinetic Toroidal Code (GTC). Fusion, the power source of the stars, has been the focus of active research since the early 1950s. While progress has been impressive — especially for magnetic confinement devices called tokamaks — the design of a practical power plant remains an outstanding challenge. A key topic of current interest is microturbulence, which is believed to be responsible for the experimentally-observed leakage of energy and particles out of the hot plasma core. GTC's objective is to simulate global influence of microturbulence on particle and energy confinement. GTC utilizes the gyrokinetic Particle-in-Cell (PIC) formalism [19] for modeling the plasma interactions, and is orders-of-magnitude faster than full force simulations. However, achieving high parallel and architectural efficiency is extremely challenging due (in part) to potential fine-grained data hazards, irregular data accesses, and low computational intensity of the GTC subroutines.

This is the first study that examines tuning the full GTC production code for the multi- and manycore era, leveraging both homogeneous and heterogeneous computing resources — as opposed to previous investigations that focused on a subset of GTC's computational kernels in isolation [21, 22]. Another key contribution is the exploration of novel optimizations in a multi-node environment, including comparisons between flat MPI and hybrid MPI/OpenMP programming models. To validate the impact of our work, we explore four parallel platforms: the IBM BlueGene/P, Cray's Magny-Cours-based XE6, an Intel Nehalem InfiniBand Cluster, as well as a manycore cluster based on NVIDIA Fermi GPUs. Our optimization schemes include a broad range of techniques including particle and grid decompositions designed to improve locality and multi-node scalability, particle binning for improved load balance, GPU acceleration of key subroutines, and memory-centric optimizations to improve single-node performance and memory utilization.

Overall results demonstrate that our approach outperforms the highly-tuned reference implementation by up to 1.6× using 49,152 cores of the Magny-Cours "Hopper" system, while reducing memory requirements by 6.5 TB (a 10% saving). Additionally, our detailed analysis provides insight into the trade-offs of parallel scalability, memory utilization, and programmability on emerging computing platforms for complex numerical simulations.

2. GTC OVERVIEW AND RELATED WORK

The six-dimensional Vlasov-Maxwell system of equations [20] govern the behavior of particles in a plasma. The "gyrokinetic formulation" [19] removes high-frequency particle motion effects that are not important to turbulent transport, reducing the kinetic equation to a five-dimensional space. In this scheme, the helical motion of a charged particle in a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SC11, November 12-18, 2011, Seattle, Washington, USA
Copyright 2011 ACM 978-1-4503-0771-0/11/11 ...$10.00.

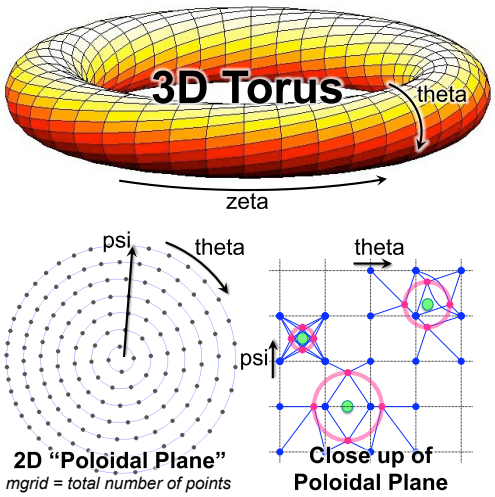

Figure 1: An illustration of GTC's 3D toroidal grid and the "four-point gyrokinetic averaging" scheme employed in the charge deposition and push steps.

magnetic field is approximated with a charged ring that is subject to electric and magnetic fields. The Particle-in-Cell (PIC) method is a popular approach to solve the resulting system of equations, and GTC [11, 12, 20] is an efficient and highly-parallel implementation of this method.

The system geometry simulated in GTC is a torus with an externally-imposed magnetic field (see Figure 1) characteristic of fusion tokamak devices. The self-consistent electrostatic field generated by the particles at each time step is calculated by using a grid discretization, depositing the charge of each particle on the grid, and solving Poisson's equation in Eulerian coordinates. The field value is then used to advance particles in time. GTC utilizes a highly specialized grid that follows the magnetic field lines as they twist around the torus. This permits use of a smaller number of toroidal planes in comparison to a regular grid that does not follow field lines, while retaining the same accuracy.

We next discuss GTC's primary data structures, parallelization methodology, and computational kernels, in order to gain insight into performance tuning opportunities.

2.1 GTC Data Structures and Parallelization

GTC principally operates using two data representations: those necessary for performing grid-related work, and those that encapsulate particle coordinates in the toroidal space. Three coordinates (shown in Figure 1) describe particle position in the torus: ζ (*zeta*, the position in the toroidal direction), ψ (*psi*, the radial position within a poloidal plane), and θ (*theta*, the position in the poloidal direction within a toroidal slice). The corresponding grid dimensions are *mzeta*, *mpsi*, and *mthetamax*. The average number of particles per grid cell is given by the parameter *micell*.

The charge density grid, representing the distribution of charge throughout the torus, and the electrostatic field grid, representing the variation in field strength (a Cartesian vector), are two key grid-related data structures in GTC. Note that the field grid, a vector quantity, requires three times the storage of the charge density grid.

The highly-tuned and parallel implementation of GTC, which we will refer to as the reference version, uses multiple levels of parallel decomposition. A one-dimensional domain decomposition in the toroidal direction is employed to partition grid-related work among multiple processors in a distributed-memory system. GTC typically uses 64 to 128-way partitioning along the torus. In addition, particles within each of these toroidal domains can be distributed among multiple MPI processes for parallelizing particle-related work. A third level of parallelism is intra-node shared memory partitioning of both particle and grid-related work, and this is accomplished using OpenMP pragmas in GTC. The two-level particle and grid decomposition entails inter-node communication, as it is necessary to shift particles from one domain to another (due to the toroidal parallelization), and also obtain the globally-consistent states of grid data structures (due to the particle decomposition).

	Charge	**Push**	**Shift**
Flops	$180mi$	$450mi$	$12mi$
Particles (bytes read)	$40mi$	$208mi$	$8mi + 100mi_s$
Particles (bytes written)	$140mi$	$72mi$	$100mi_s$
Arithmetic intensity (flops/byte)	$\leq$0.56	$\leq$1.28	N/A
Grid data (bytes read)	$\leq$320	$\leq$768	0
Grid data (bytes written)	$\leq$256	0	0

Table 1: Characteristics of memory-intensive loops in particle-processing GTC kernels (charge deposition, push, and shift) for a single time-step in terms of particles (mi) or particles shifted (mi_s).

2.2 GTC Code Structure

The gyrokinetic PIC method involves five main operations for each time step (the corresponding GTC routines are indicated in italics): charge deposition from particles onto the grid using the four-point gyro-averaging scheme (*charge*), solving the gyrokinetic Poisson equation on the grid (*poisson*), computing the electric field on the grid (*field*), using the field vector and other derivatives to advance particles (*push*), and smoothing the charge density and potential vectors (*smooth*). In addition, distributed memory parallelization necessitates a sixth step, *shift*, to move particles between processes or toroidal domains. In this paper, we assume a collision-less system and that the only charged particles are ions (adiabatic electrons). Simulations with kinetic electrons require additional and analogous charge, push, and shift steps for electrons.

The parallel runtime and efficiency of GTC depends on several factors, including the simulation settings (the number of particles per grid cell *micell*, the discretized grid dimensions, etc.), the levels of toroidal (*ntoroidal*) and particle decomposition (*npartdom*) employed, as well as the architectural features of the parallel system, described in previous GTC performance studies on parallel systems [1, 12, 27].

Our work builds on the well-optimized "reference" implementation of GTC, and further improves upon it to address the challenges of HPC systems built from multi- and manycore processors. Thus, understanding and optimizing the following key routines on multi- and manycore architectures is imperative for achieving high performance on today's petascale machines.

Charge deposition: The charge deposition phase of GTC's PIC primarily operates on particle data, but involves the complex particle-grid interpolation step. Particles, represented by a four-point approximation for a charged ring, de-

posit charge onto the grid. This requires streaming through a large array of particles and updating locations in memory corresponding to the bounding boxes (eight grid points each) of the four points on the particle charge ring (see Figure 1). Each particle update may thus scatter increments to as many as 32 unique grid memory locations. In a shared memory environment, these increments must be either guarded with a synchronization mechanism to avoid read-after-write data hazards or redirected to private copies of the charge grid.

As seen in Table 1, for each particle, the reference implementation of this step reads 40 bytes of particle data, performs 180 floating point operations (flops), and updates 140 bytes of particle data — resulting in a low arithmetic intensity of 0.56 flops per byte (assuming perfect grid locality and no cache bypass).

The locality challenges associated with this kernel are dependent on the variation in spatial positions of particles within the grid. If grid locality is not achieved, then the impact of (grid) capacity cache misses may overwhelm and dominate the performance of this kernel; thus, our optimizations focus on maximizing locality and minimizing the impact of synchronization.

In a distributed-memory parallelization, particles within a toroidal segment may be partitioned among multiple processes by setting *npartdom* greater than 1. The effect is that each process maintains a private copy of the charge density grid onto which it deposits charge from the particle it owns. To merge these copies together, GTC performs an "Allreduce" computation, resulting in each process having a copy of the poloidal grid with the cumulative charge densities.

Poisson/Field/Smooth: The three solve steps constitute purely grid-related work, where the floating-point operations and memory references scale with the number of poloidal grid points (*mgrid*). Simulations typically employ high particle densities and so the time spent in grid-related work is typically substantially lower than the particle processing time (charge, push, and shift). Note that multi-node parallelism in these steps is limited to the extent of 1D domain decomposition (the *ntoroidal* setting). When *npartdom* is set to a value greater than 1, each process begins with the global copy of the charge/potential/field grid and executes the Poisson/field/smooth routine independently. The reference GTC version thus trades redundant computation for reduced communication and synchronization.

Push: In this phase, the electric field values at the location of the particles is "gathered" and used for advancing them. This step also requires streaming through the particle arrays and reading irregular grid locations in memory (the electric field values) corresponding to the four bounding boxes of the four points on the ring. This can involve reading data from up to 16 unique memory locations. Additionally, as seen in Table 1, this kernel reads at least 208 bytes of particle data, performs 450 flops, and updates 72 bytes for every iteration of the particle loop. As this routine only performs a gather operation in a shared memory environment, it is devoid of data hazards. Moreover, since the arithmetic intensity of this kernel is higher than charge deposition's, it is somewhat less complex to optimize.

Shift: GTC's shift phase scans through the particle array looking for particles that have moved from the local domain (in the $\pm zeta$ directions). The selected particles are buffered, marking the resultant "holes" in the particle array where a particle has effectively been removed, and sending these buffers to the neighboring domains. Particles are moved only one step in *zeta* at once (thus performing nearest-neighbor communication among processes along the toroidal direction), and so the shift phase iterates up to $mzetamax/2$ times. In the reference implementation, holes are filled sequentially as particles are received. However, such a sequential implementation can be inefficient in highly threaded environments (particularly GPUs). We discuss remediation strategies in Sections 4 and 5. Even when *npartdom* is greater than one, processes still only communicate with their two spatially neighboring processes, thereby simplifying communication.

2.3 Related Work

PIC is a representative method from the larger class of *particle-mesh* methods. In addition to plasma physics (e.g., GTC's gyrokinetic PIC), particle-mesh methods find applications in astrophysics [3, 14], computational chemistry, fluid mechanics, and biology. There are also several popular frameworks from diverse areas that express PIC computations such as VPIC [5], OSIRIS [13], UPIC [9], VORPAL [26], and QuickPIC [15].

Prior work on performance tuning of PIC computations has mostly focused on application-specific domain (mesh) decomposition and MPI-based parallelization. The ordering of particles impacts the performance of several PIC steps, including charge deposition and particle push. Bowers [4] and Marin et al. [23] look at efficient particle sorting, as well as the performance impact of sorting on execution time. A closely-related macro-scale parallelization issue is particle load-balancing [7], and OhHelp [25] is a library for dynamic rebalancing of particles in large parallel PIC simulations. Koniges et al. [18] report performance improvements by overlapping computation with inter-processor communication for gyrokinetic PIC codes. The performance of the reference GTC MPI implementation has been previously well-studied on several large-scale parallel systems [1,12,27]. Prior research also examines expressing PIC computations via different programming models [2,6].

There has also been recent work on new multicore algorithms and optimizations for different PIC steps. Stanchev et al. [28] investigate GPU-centric optimization of particle-to-grid interpolation in PIC simulations with rectilinear meshes. Decyk et al. [10] present new parameterized GPU-specific data structures for a 2D electrostatic PIC code extracted from the UPIC framework. In our prior work on multicore optimizations for GTC, we introduced various grid decomposition and synchronization strategies [22] that lead to a significant reduction in the overall memory footprint in comparison to the prior MPI-based GTC implementation, for the charge and push routines. This paper extends our prior work [21, 22] by developing an integrated CPU- and GPU-optimized version that accelerates a fully-parallel production GTC simulation at scale.

3. EXPERIMENTAL SETUP

3.1 Evaluated Platforms

Cray XE6 "Hopper": Hopper is a Cray XE6 massively parallel processing (MPP) system, built from dual-socket 12-core "Magny-Cours" Opteron compute nodes. In reality,

Core Arch	IBM PPC450d	AMD Opteron	Intel Nehalem	NVIDIA Fermi
Type	dual-issue in-order SIMD	sprscalar out-order SIMD	sprscalar out-order SIMD	dual warp in-order SIMT
Clock (GHz)	0.85	2.1	2.4	1.15
DP GFlop/s	3.4	8.4	9.6	73.6[1]
$/core (KB)	32	64+512	32+256B	48
Memory-Parallelism	HW Prefetch	HW Prefetch	HW Prefetch	Multi-threading
Node Arch	**Blue-Gene/P**	**Opteron 6172**	**Xeon X5530**	**Tesla C2050**
Cores/chip	4	6	4[2]	14[1]
Chips/node	1	4	2	1
Last $/chip	8 MB	6 MB	8 MB	768 KB
STREAM[6]	8.3 GB/s	49.4 GB/s	35.2 GB/s	78.2 GB/s
DP GFlop/s	13.6	201.6	76.8	515
Memory	2 GB	32 GB	24 GB	3 GB
Power	31W[3]	455W[3]	298W[4]	390W[4]
System Arch	**Blue-Gene/P "Intrepid"**	**XE6 "Hopper"**	**Dirac Testbed "Intel Cluster"**	**Dirac Testbed "Fermi Cluster"**
Affinity	N/A	`aprun`	`KMP_AFFINITY`[5]	
Compiler	XL/C	GNU C	Intel C + nvcc	
Interconnect	custom 3D Torus	Gemini 3D Torus	InfiniBand Fat Tree	

Table 2: Overview of Evaluated Supercomputing Platforms. [1]*Each shared multiprocessor (SM) is one "core".* [2]*Two threads per core. Power based on Top500 [29] data*[3] *and empirical measurements*[4]*. The GPU itself is 214W.* [5]*Affinity only used for full threaded experiments.* [6]*STREAM copy.*

each socket (multichip module) has two dual hex-core chips, and so a node can be viewed as a four-chip compute configuration with strong NUMA properties. Each Opteron chip instantiates six super-scalar, out-of-order cores capable on completing one (dual-slot) SIMD add and one SIMD multiply per cycle. 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 6MB L3 cache and dual DDR3-1333 memory controllers capable of providing an average STREAM [24] bandwidth of 12GB/s per chip. Each pair of compute nodes shares one Gemini network chip, which collectively form a 3D torus.

IBM BlueGene/P "Intrepid": Intrepid is a BlueGene/P MPP optimized for energy-efficient supercomputing. Instead of using superscalar and out-of-order execution, the BlueGene Compute chip is built from four power-efficient, PowerPC 450d, dual-issue, in-order, embedded cores. Like the Opterons, they implement two-way SIMD instructions. However, unlike the Opterons, they use fused-multiply add. Each core includes a 32 KB L1, and each socket includes a 8 MB L2 cache. However, unlike the Opterons, the L1 is write-through and the L1 miss penalty is a severe 50 cycles. Thus, locality in the L1 is essential. The memory is low-power DDR2-425 and provides about 8.3 GB/s. Like the Cray machine, Intrepid compute nodes are arrayed into a 3D torus. Although BlueGene's per-core performance is less than half that of the Opterons, the average system power *per core* is also less than half. As such, despite being three years old, BlueGene/P continues to represent a competitive, power-efficient design point.

Intel InfiniBand Cluster "Intel Cluster": The Dirac cluster at NERSC is a small GPU testbed cluster, and is built from 50 dual-socket, quad-core 2.4 GHz Xeon X5530 compute nodes. Like the Hopper's Opteron cores, each Nehalem core is a super-scalar, out-of-order core capable of executing one SIMD add and one SIMD multiply per cycle, but has a higher-associativity private 32 KB L1 and 256 KB L2 cache. Moreover, each core is multithreaded. The four cores on a chip share a 8 MB L3 cache and three DDR3-1333 memory controllers capable of providing up to 17.6 GB/s per chip. Locally, nodes are connected via a QDR InfiniBand fat tree network. By convention, when running CPU-only code on the Dirac cluster, we will refer to it as the "Intel Cluster".

GPU Accelerated Cluster "Fermi Cluster": As mentioned, Dirac is a testbed for GPU technology. Currently, one Tesla C2050 (Fermi) GPU is installed on each Dirac node. Each C2050 includes 448 scalar "CUDA cores" running at 1.15 GHz and grouped into fourteen SIMT-based streaming multiprocessors (SM). This provides a peak double-precision floating-point capability of 515 GFlop/s. Each SM includes a 64 KB SRAM that can be partitioned between cache and "shared" memory in a 3:1 ratio. Although the GPU has a raw pin bandwidth of 144 GB/s to its on-board 3 GB of GDDR5 DRAM, the measured STREAM TRIAD bandwidth using the SHOC benchmark [8] is 87 GB/s, and the measured bandwidth with ECC enabled is only 78.2 GB/s. ECC is enabled in all our experiments. Data may be transferred to and from GPU memory via a PCIe Gen2 link which is measured to be 5.3 GB/s using SHOC on our GPU accelerated testbed cluster. Thus, although the GPU has more than 6× the floating-point capability, it has only 2× the memory bandwidth, but only $\frac{1}{8}^{th}$ the memory capacity of the host. Although the small memory capacity motivates shuffling of data to the host, the low PCIe bandwidth makes locality on the GPU board imperative. In the end, simulation size is curtailed by the small GPU memory. For our experiments, we use the term "Fermi Cluster" to refer to heterogeneous CPU/GPU simulations, and differentiate Dirac results using the GPUs from CPU-only Dirac results.

3.2 Programming Models

Our study explores three popular parallel programming paradigms: Flat MPI, MPI/OpenMP, and MPI/OpenMP/-CUDA. MPI has been the de facto programming model for distributed memory applications, while the hybrid MPI/-OpenMP models are emerging as a productive means of exploiting shared memory via threads. The hybrid nature of the GPU-accelerated Dirac cluster necessitates the use of a MPI/OpenMP/CUDA hybrid programming model. For all conducted experiments, we utilize every CPU core on a given compute node. Therefore, the number of threads per MPI process is inversely related to the number of MPI processes per node, so as to ensure full utilization of cores. Furthermore, we restrict our hybrid experiments to one MPI process per core (flat MPI), one process per chip (NUMA node), and one process per compute node. In the case of the Hopper XE6, these configurations represent 24 processes/node, 4 processes of 6 threads/node, and 1 process of 24 threads/node. GPU accelerated code requires there be only one controlling process on the host side per GPU. The Dirac cluster contains a single GPU per node, so only a single process invokes the GPU accelerated code while one or more OpenMP threads are used for host-side computations.

3.3 Simulation Configurations

The most important parameters describing a GTC simu-

Grid Size	B	C	D
mzeta	1	1	1
mpsi	192	384	768
mthetamax	1408	2816	5632
mgrid (grid points per plane)	151161	602695	2406883
chargei grid (MB)†	2.31	9.20	36.72
evector grid (MB)†	6.92	27.59	110.18
Total Particles (*micell*=20)	3.02M	12.1M	48.1M
Total Particles (*micell*=96)	14.5M	57.9M	231M

Table 3: The GTC experimental settings. $mzeta = 1$ **implies each process operates on one poloidal plane. †minimum per process.**

lation are the dimensions of the discretized toroidal grid, the total number of particles, and the average particle density as measured in the ratio of particles to grid points (labeled *micell*). In all our initial experiments, we set *mzetamax* to 16 and *mzeta* to 1 (i.e., 16 poloidal planes with each process owning one). Moreover, to allow each configuration to constitute the same physical simulation (regardless of the programming model), parallelism is oriented around the node. Each node (irrespective of processes or threads) therefore owns one poloidal plane and all the particles therein. Thus, all scaling experiments in this paper are *strong scaling*.

In order to demonstrate the viability of our optimizations across a wide variety of potential simulations, we explore three different grid problem sizes, labeled *B, C, D*, and explore two particle densities: 20 and 96 particles per grid point. Therefore, a "B20" problem — often used in throughout this paper as it fits into all memory configurations — uses the class B grid size with an average particles per grid point count of 20 (assuming one process per node). Grid size C corresponds to the JET tokamak, the largest device currently in operation [17], and *D* to the forthcoming International Thermonuclear Experimental Reactor (ITER): a multi-billion dollar large-scale device intended to prove the viability of fusion as an energy source [16].

Table 3 lists these settings, which are similar to ones used in our prior experimental study [22], as well as GTC production runs [12]. Note, the grid memory requirements will scale with the number of processes per node. For the three examined GTC problem configurations, the maximum Larmor radius (a function of several GTC parameters) corresponds to roughly *mpsi*/16. The initial Larmor radii follow a uniform random distribution.

Figure 2 visualizes the confluence of programming model and simulation configurations on Hopper. As *mzetamax* and *mzeta* are 16 and 1 respectively, the torus is partitioned into 16 segments of 1 poloidal plane. To assign one of these segments to each 24-core Hopper node, we explore three approaches to parallelism within a node: one process per core, one process per chip, and one process per node. This is achieved by varying *npartdom*. Particles within this toroidal segment are simply partitioned among processes on a node.

Thus, in a D96 simulation with *npartdom*=24, the 231M particles on a node would be partitioned among the processes, causing two negative effects. First, the per-process particle density is only four particles per grid point, and second, the charge and field grids must be replicated on a process by process basis. As a result, compared to the 24 thread hybrid implementation, the flat MPI implementation incurs a 24-way replication of charge grid data. Moreover, all processes on a node must reduce their copies of the charge grid into one via an MPI_Allreduce, and solve Poisson's equation

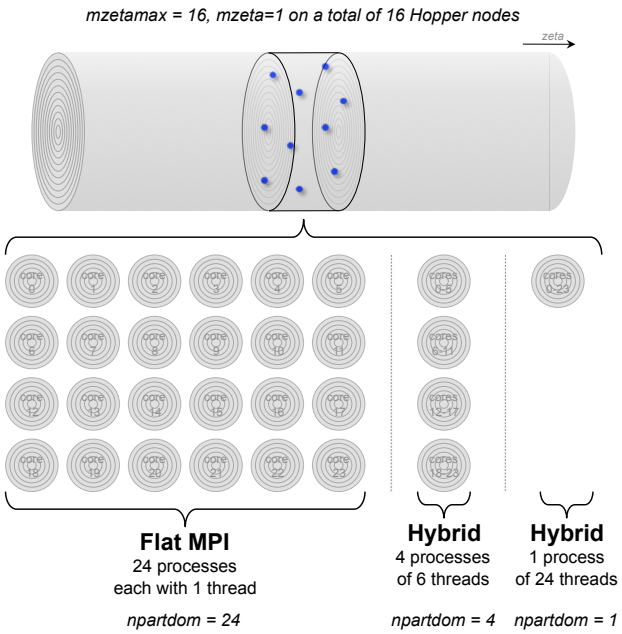

Figure 2: An illustration of the approaches used to exploit parallelism on Hopper, and the node-centric view of the replication of charge and field grids. Particles are partitioned, but never replicated.

redundantly so that each process has a copy of the resultant field grid. Moving to a hybrid implementation reduces the scale of the reduction, as well as the degree of redundancy in the solve. Conversely, the particle shift phase is done on a process-by-process basis. Shift is simplest in the flat MPI implementation and becomes progressively more complex and less concurrent in the hybrid version.

4. CPU OPTIMIZATIONS

The reference production GTC code was developed in Fortran 90, and uses the MPI and OpenMP libraries for parallelism. Our study has rewritten the entire application in C for several reasons. First, our new implementation provides a common low-level optimization substrate and simplifies GPU exploitation. We can also leverage optimized C/OpenMP multithreaded implementations from prior work, developed for the particle-mesh interpolation performed in charge and push [22]. We now discuss the details of the code restructuring as well as CPU specific optimizations.

Code Restructuring: The new C version permits exploration of several data layout alternatives, both for the particle and grid data. As charge and push typically dominate execution time, it is important to select the particle data layout that would maximize the performance of both kernels. We therefore chose the structure-of-arrays (SOA) layout for particle data, which necessitated moving away from the reference GTC representation (array-of-structures). The SOA approach maximizes spatial locality on streaming accesses and thus improves sustained bandwidth. This optimization is employed on both CPUs and GPUs.

Additional low-level restructuring changes include flattening 2D and 3D grid arrays to 1D, zero-indexed arrays in C, pre-allocation of memory buffers that are utilized for temporary storage in every time step, aligned memory allocation

of particle and grid arrays to facilitate SIMD intrinsics and aligned memory references, NUMA-aware memory allocation relying on the first-touch policy, and C implementations of Fortran 90 intrinsics such as `modulo` and `cshift`.

Our work optimizes all six GTC phases on the CPU. Across all platforms, and on all kernels, we use OpenMP for thread-level parallelism. The key parallelization strategy and new contributions are discussed below.

Charge: Our previous single-node work [21, 22] explored optimizing GTC's charge deposition via static replication of grid segments grid, coupled with synchronization via locks and atomics, and using POSIX threads for shared memory parallelism. In addition to the original copy of the charge grid, we create a number of static replicas. Each replica can constitute either a full poloidal plane grid, or a small partition of it extended with a ghost zone in *psi* to account for the potentially-large radius of gyrating particle. Threads may quickly access their own replica without synchronization, or slowly access the globally shared version with a synchronization primitive. Thus, the size (and extra memory usage) of the replica may be traded for increased performance.

Our optimized version leverages OpenMP to parallelize the particle loop, and using our previous results as guidance, selects the best replication and synchronization for each underlying platform. Note that as the BlueGene/P does not support 64-bit atomic operations, we employ the full poloidal grid replication strategy when running hybrid configurations (MPI/OpenMP). In case of the x86 systems, we have an additional cache-bypass optimization. As seen in Table 1 charge deposition involves substantial write traffic to particle data. Given that these unit stride writes are not accessed again until the push phase, we implement SSE intrinsics for streaming (cache-bypass) stores — boosting the computational intensity of the interpolation loop to 1.0.

Poisson/Field/Smooth: The grid-related routines are primarily comprised of unit-stride accesses to grid data structures such as the charge density, potential, and the electric field. The main optimization for these routines is NUMA-aware allocation of both temporary and persistent grid data structures, in order to maximize read bandwidth for grid accesses. Additionally, we fuse OpenMP loops wherever possible to minimize parallel thread fork-join overhead.

Push: Similar to charge, we leverage parts of prior work to implement an optimized, threaded implementation of push. The key optimizations performed in this study is loop fusion to improve the arithmetic intensity and NUMA-aware allocation. We also improve load balancing via OpenMP's guided loop scheduling scheme. This increases performance for larger cache working sets seen by particles near the outer edge, with a reduced number of loop iterations assigned to those threads. Load balancing is thus far more efficient than what is possible with a flat MPI reference approach.

Shift: Recall that shift is an iterative process in which particles move one step at a time around the torus. We optimize the shift phase by threading the MPI buffer-packing and unpacking, as well as via the use of additional buffer space at the end of the particle arrays. In practice each thread enumerates a list of particles that must be moved, buffer space then is allocated, threads fill in disjoint partitions of the buffer, and the send is initiated. When particles are received, they may be filled into the "holes" emptied by departing particles. However, performing hole-filling every step may unnecessarily increase overhead. Therefore, particle arrays are allocated with extra elements at the end of arrays, and incoming particles may be copied in bulk into this segment. If the buffer is exhausted, holes are filled with particles starting from the end of the particle arrays. The hole filling frequency can thus be a runtime parameter, and we tune it based on the expected particle movement rate.

Radial binning: For efficient parallel execution of charge and push, and for reducing the cache working set size of grid accesses, we make the important assumption that particles are approximately radially-binned. This is however not likely to be satisfied as the simulation progresses, as particles may move in the radial direction as well as across toroidal domains (causing holes in the particle array). Thus, we implement a new multithreaded radial binning routine, as an extension to the shift routine. Radial binning frequency is be a parameter that can be set dynamically, and is currently based on the number of toroidal domains and the expected hole creation frequency.

On-the-fly auxiliary array computations: GTC requires twelve double precision values per particle ($96mi$ bytes) to maintain the coordinates and state of each particle. This also represents the data volume per particle, exchanged in the shift routine. In addition, seven auxiliary arrays of size mi words each ($44mi$ bytes total) are maintained just to facilitate exchange of data from the disjoint charge and push phases of the simulation. The auxiliary information, accounting for particle data write traffic in charge and read traffic in push, can be avoided if we redundantly perform approximately 120 floating point operations per particle in the push phase. While we implemented this optimization, it did not lead to an overall improvement in performance for the problem configurations and parallel systems tested in this paper. This may however be an important optimization for future memory- and bandwidth-constrained systems.

5. GPU ACCELERATION

To investigate the potential of manycore acceleration in a cluster environment, we employ the Fermi GPUs in the Dirac system. A judiciously selected subset of the computational phases were ported to the GPU. As particle arrays can constitute the bulk of memory usage, our implementation strives to keep particles on the GPU to maximize performance. In practice, this constrains the size of possible problem configurations. We leverage our previous research into tuning the charge and push phases for GPUs [21]; however, significant additional effort was required to implement and optimize the GPU shift functionality.

Our implementation strategy requires that most auxiliary arrays are kept in the GPU memory to communicate data between GTC's computational phases. For instance, the indices of the electrostatic field that are computed during the charge phase are stored for use by the push phase. To facilitate correct CPU/GPU integration, some previously developed charge and push optimizations [21], such as radial binning and full sorting of particle updates, are no longer applicable or have yet to be evaluated. Additionally certain memory-intensive optimizations, such as keeping additional particle buffer space for shifted particles, were not suitable with the limited GPU memory capacity.

Charge: Similar to the OpenMP version, CUDA parallelizes the iterations of particle arrays among thread blocks

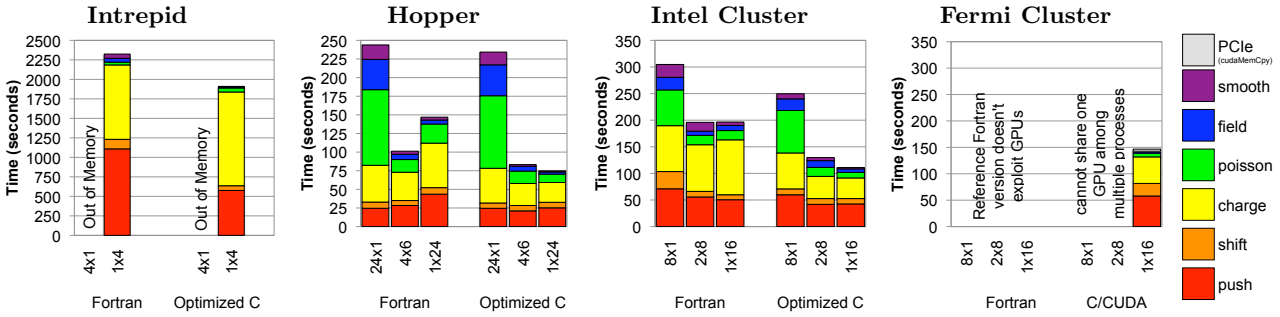

Figure 3: Breakdown of runtime spent in each phase for the B20 problem as a function of architecture, optimization, and threading. Note, minor axis is "processes×threads", or equivalently, "`npartdom`×`OMP_NUM_THREADS`". Additionally, observe each is plotted on a different timescale. PCIe time has been removed from each kernel's time and tabulated in a separate bar.

with special attention paid to attain memory coalescing. The thread blocks then perform the requisite scatter-add operations to a charge grid resident in the GPU's device memory. However, unlike the CPU implementations of charge deposition, there is no grid replication on the GPU. Rather, the GPU version relies solely on fast double-precision atomic increment (implemented via a compare-and-swap operation). Unfortunately, the gather/scatter nature of this kernel makes memory coalescing of grid data all but impossible. Nevertheless, we alleviate this performance impediment via a transpose within shared memory. Thus, computation is organized so that CUDA threads access successive particles when reading, but is reorganized so that threads work together on one charge deposition (of up to 32 neighboring grid points) when writing. Given the limited use of shared memory, we configure the GPU to favor the L1 cache. Note that the GPU charge is responsible for both initializing the charge grid and transferring it back to host memory.

We also highlight that the charge phase is also partly executed on the CPU, thus exploring architectural heterogeneity. Specifically, the communication involved in reduction of the charge density across domains and subsequent boundary condition updates are managed by the CPU. The solve routines utilize the charge density grid and produce the electrostatic field vector used in the push phase.

Poisson/Field/Smooth: The three solve-related phases are performed on the CPU as they mandate communicating the computed values with other nodes via MPI. We predict that migrating these computations to the GPU will likely not substantially improve Fermi Cluster performance, as their fraction of the runtime is small (as detailed in Section 6).

Push: For the push phase, we leverage our extensive GPU optimizations [21]. Unfortunately, the transpose operation that facilitated charge showed no benefit here. Our analysis indicates that a simple decomposition of one particle per CUDA thread and 64 threads per block performed well. Additionally, conditional statements were replaced with redundant computation to avoid divergent code, and some variables were judiciously placed in CUDA's "constant" memory. We also observe that the per-SM SRAM should be configured to prefer the L1 cache (i.e., 16 KB shared + 48 KB L1) for the larger grid size. Finally, prior to execution, push must transfer the field grid produced by the CPU solver from host memory to the GPU's device memory.

Shift: The shift phase is far more challenging to implement on the GPU. First, the GPU must enumerate a list of all particles currently residing in device memory whose toroidal coordinate is now outside of the GPU's domain and pack them into special buffers. Although a sequential implementation of such an operation is straightforward, a GPU implementation must express massive parallelism. To that end, each thread block maintains small shared buffers that are filled as it traverses its subset of the particle array. Particles are sorted into three buffers for left shift, right shift and keep buffer. Whenever the local buffer is exhausted, the thread block atomically reserve a space in a pre-allocated global buffer and copy data from the local buffer to the global one. Packed particles are also flushed over the original particle array, thus clustering particle holes towards the end of each thread block assignment. The use of atomics causes partial shuffling of the particle that fortunately does not affect the correctness of the implementation.

The structure of arrays representation that benefited the charge and push routines proved problematic here. Reducing the number of memory transfers, from 24 to 2 in our case, required using the array of structures organization. Consequently data is transposed while flush to the global buffer. This organization also helps in facilitating the messaging mechanism. The global buffer is copied back to the host where the normal iterative shift algorithm is executed. Upon completion, the host then transfers a list of incoming particles to the GPU, where unpacking involves filing clustered holes in the particle arrays and transposing the data back to the GPU data structure of arrays layout. Maximizing parallelism and attaining good memory coalescing, not to mention correctness in the presence of "Heisenbugs", made porting this kernel extremely challenging yet mandatory, to avoid expensive transfer of particle arrays to the CPU.

6. RESULTS AND ANALYSIS

In this section, we present the performance of GTC using 16 nodes on all platforms, where the underlying problem (simulation) remains constant. Our optimizations explore the balance between threads and processes via the *npartdom* (particle decomposition) configuration parameter. Additionally we present large-scale simulations to understand strong-scaling behavior at high concurrency.

6.1 Optimization and Threading

To highlight the benefits of our optimizations as well as the trade-offs of using multiple threads per process, we examine

Kernel	BGP Intrepid	XE6 Hopper	Intel Cluster	Fermi Cluster
push	1.93×	1.12×	1.31×	0.95×
shift	2.03×	0.89×	1.04×	0.40×
charge	0.79×	1.43×	2.26×	1.73×
poisson	0.70×	1.53×	1.67×	2.52×
field	3.52×	2.43×	2.54×	3.10×
smooth	9.09×	2.76×	3.17×	13.8×
overall speedup	**1.22×**	**1.35×**	**1.77×**	**1.34×**

Table 4: Speedup of B20 problem by kernel (best threaded C vs. best threaded Fortran). Note, baseline for the Fermi Cluster is the best Fortran implementation on the Intel Cluster.

the B20 problem in detail. Figure 3 shows the time spent in each phase for 200 time steps of the simulation, as a function of machine, optimization (major axis), and threading (minor axis). The three threading configurations per platform include the flat MP (one MPI process per core), MPI/-OpenMP (one MPI process per chip), and MPI/OpenMP (one MPI process per compute node) — for both the reference Fortran GTC as well as our optimized C version. As Intrepid has only one chip per node, both hybrid implementations represent identical configurations. Moreover, as all simulations use every core on a node, changes in threads per process are realized by decreasing *npartdom*.

Results demonstrate a dramatic increase in performance when threading is used. This benefit primarily arises from retasking cores from performing redundant solves (Poisson/-field/smooth) to collaborating on one solve per node. Interestingly, it appears that the Fortran version on Hopper does not effectively exploit the NUMA nature of the compute nodes. As such, when using 24 threads per process on Hopper, there is a marked degradation in Fortran performance in push, charge, and Poisson. Conversely, our optimized implementation delivers moderate improvements in charge (obviates the intra-node reduction) and solve (trading redundancy for collaboration), but a slight decrease in push performance. These effects are present, but less dramatic on the Intel cluster. Finally, the benefit of threading on Intrepid is more basic. This problem size per node cannot be run using the flat MPI programming model as it runs out of memory. This highlights the importance of moving to a threaded model, allowing us to exploit shared memory and avoid redundant poloidal replicas in the solver routines. We also note that on all platforms, our optimized implementation attains best performance with one process per node.

Figure 3 also relates the relative contribution to runtime for each GTC computational component for a given architecture and threading configuration. Results show that the threading and optimization choice can have a profound impact on the relative time in each routine. Generally, as the number of threads increase, there is an improvement in the solver routines (due to reduced redundancy) as well as the shift routine (particles need not be shifted if they remain on the node). However, in the reference Fortran implementation, threading has a minimal impact on charge or push performance. As such, the fraction of time spent in charge and push can become dramatic, exceeding 33% of the best runtime on the x86 machines and constituting a combined 93% of the runtime on Intrepid. Future work will address the performance of these phases though a variety of auto-tuning techniques (unrolling, explicit SIMDization, etc.).

Results also show that, contrary to conventional wisdom, PCIe overhead does not substantially impede GPU performance. Rather the challenges of data locality and fine-grained synchronization in the push and particularly the charge routines result in slower GPU execution times compared with the Intel CPU Cluster. Unfortunately, the extra GPU memory bandwidth cannot make up for its inability to form a large working set in its relatively small L2 cache. Migrating the solver computations to the GPU would have a negligible impact on performance as it constitutes a small fraction of the runtime.

Table 4 details the B20 performance benefits of our optimized approach compared with the fastest reference implementation on a kernel-by-kernel basis. For our x86 optimizations, the complex charge routine delivers more than a 2× speedup due to our locality-aware, low-synchronization particle-grid interpolation approach. Unfortunately, the lack of 64-bit atomic operations prevents us from leveraging those techniques to improve Intrepid charge performance (an inefficient replicate/reduce approach [22] is realized through OpenMP). The benefits observed in the three grid-related routines is even more dramatic, partially due to the attained peak performance with one process per node (thus all the on-node parallelism is tasked for one solve). Interestingly, BlueGene/P speedups vary wildly — a testament to the differences between XLF and XLC compilers, as well as the lack of ISA-specific BlueGene optimizations in our current implementation. Our shift routine also includes time for radial binning, which is performed every four time steps for this configuration. Hence we do not achieve a substantial speedup over the Fortran version on the x86 platforms.

Finally, Table 4 shows that the GPU implementation benefited from speedups on the solver phases (as particles are not flushing the grid points from the caches. The good speedup on charge (relative to reference Fortran) coupled with the lack of any improvement in push performance — due to GPU's difficulties in addressing the irregular data patterns — resulted in the overall optimized C/CUDA implementation achieving only a 34% improvement relative to the reference Fortran/OpenMP code.

6.2 Performance and Energy Comparison

Figure 4(left) summarizes our optimization impact (compared with the fastest reference implementation), as well as the relative per-node performance across all architectures (normalized to the BlueGene/P reference version). Results show the significant speedup of a Hopper node compared with the commodity Intel cluster, GPU-accelerated Fermi Cluster, and low-power BlueGene/P node, attaining a 1.5×, 2.0×, and 25.5× performance advantage (respectively). On a per-core level, the Nehalem achieves the highest speed, exceeding the performance of the PPC450 and Opteron by factors of 8.6× and 2.0×. Overall, through our optimizations, we attain significant application-level speedups of 1.22×, 1.35×, 1.77×, and 1.34× on Intrepid, Hopper, the Intel Cluster, and the Fermi Cluster respectively, compared to the fastest (previously optimized) Fortran version.

We now turn to Figure 4(right) that shows the energy efficiency trade-offs (derived from empirical measurements and the November 2010 Top500 [29] data, see Table 2) between our evaluated architectures — an increasingly critical metric in supercomputing design. Although all machines delivered similar energy efficiencies before optimization, our x86-

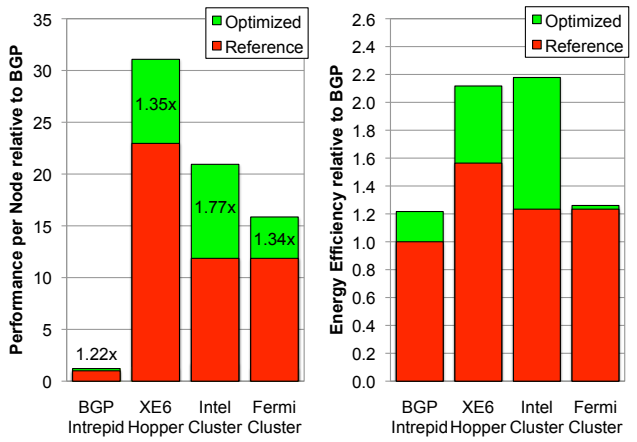

Figure 4: Performance and energy efficiency (normalized BGP = 1.0) before and after optimization for B20. Baseline for the GPU is fastest Fortran version on the Intel Cluster. Speedups through optimization are labeled. Power based on Top500 [29] and empirical measurements (see Table 2).

specific optimizations resulted in the Intel cluster and Hopper attaining an efficiency advantage of approximately 1.8× and 1.7× compared with BlueGene/P and the Fermi cluster. Nonetheless, there is clear value in the BlueGene methodology, as it has taken commodity processors three years to surpass it in terms of energy efficiency. It is also important to highlight that the Fermi cluster delivered lower energy efficiency than the Intel cluster due to its higher power and lower performance on these demanding irregular computations.

6.3 Memory Savings

Computer architecture design is seeing memory capacity per node growing slower than peak performance. Thus, reducing memory usage is a success metric on par with performance. Using the threading paradigm (in either C or Fortran), reduces the processes per node, and thus the total number of field grid copies per node. These savings in memory scale with the number of cores and problem size. For example, the class B problem on Intrepid memory requirements are only reduced by 21 MB, while the D size configuration on Hopper results in significant saving of 2.5 GB. Additionally, in our optimized version, threads share one copy of the grid and partitions a second (unlike the reference Fortran, where each thread creates a redundant charge grid copy), resulting in an additional 800 MB of savings for the class D Hopper problem. Overall, our approach saves 330 MB, 3.3 GB, and about 1 GB for Intrepid, Hopper, and the Intel Cluster respectively, corresponding to 16%, 4%, and 1% of each node's DRAM. These memory saving will have increasingly important impact on forthcoming platforms that are expected to contain higher core counts with reduced memory capacities.

6.4 Perturbations to Problem Configurations

Having examined B20 in detail, we now explore a range of configurations to understand our optimization impact across a wide range of problems. Figure 5 provides insight into how

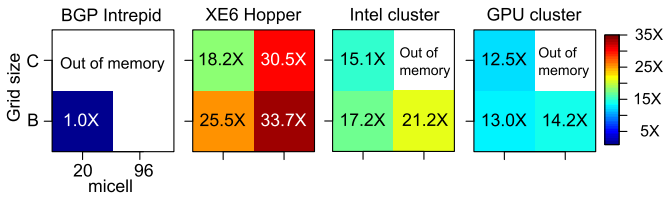

Figure 5: Heatmaps showing per-node performance (particles pushed per second per time step) as a function of system and problem configuration. All numbers are normalized to B20 Intrepid performance (for $mzeta = 16$).

performance varies as a function of problem configuration and architecture. The vertical axis corresponds to the grid size. Whereas the class B grid is relatively small (2.3 MB per component) and could likely fit in cache, the class C grid is larger (9.2 MB per component) and will challenge the 6-8 MB cache subsystems used on each CPU. The horizontal axis of Figure 5, *micell*, denotes the average number of particles per grid point (particle density). Increasing *micell* has two effects that are expected to increase performance. First, because the computational complexity of charge, push, and shift phases vary linearly with *micell*, the time spent in those routines increases relative to the time spent in the solver routines (whose computational complexity depends on *mzetamax*). Additionally, a larger *micell* increases the temporal locality seen by the charge and push phases.

Given that the total memory required for a simulation grows linearly with grid size and particle density, it is not surprising that at 16 nodes all but the B20 problem was too large for Intrepid and the C96 was too large for the Fermi and Intel Clusters. The only way to run larger problems is to use multiple nodes per plane. Broadly speaking, on the x86 machines, increasing the grid size and the corresponding working set results in a performance degradation. Conversely, an increase in particle density (increased locality, decreased solve time), causes a performance improvement.

6.5 Large Scale runs

Ultimately, ultra-scale simulations must focus on large grid sizes with high particle densities. To understand the potential scaling impediments, we examine the ITER-scale D96 problem with *ntoroidal*=64 on the Cray XE6 Hopper. Compared to the B20 problem in the previous subsection, the size of the grid per node is increased by a factor of 16×, while simultaneously the particle density is increased by almost a factor of 5×. Our experiments scale the number of nodes from 256 to 2048 (6144 to 49,152 cores) in a *strong scaling* regime by increasing *npartdom* from 4 to 32. Thus, at 256 nodes, each node uses a field grid of 110 MB to push 58 million particles. As we increase concurrency, the field grid remains constant, but the number of particles per node drops to about 7 million. All experiments leverage our fastest optimized hybrid implementation on Hopper.

Figure 6(left) shows that performance increases by a factor 4.1× (relative to the 6144-core reference point), attaining a 1.6× overall improvement compared with the fastest reference implementation on 49,152 cores. Note, however, that parallel and energy efficiency (red line) achieves only 51%, mostly due to the increasing fraction of time spent in the solver routines, clearly visible in Figure 6(right). In reality,

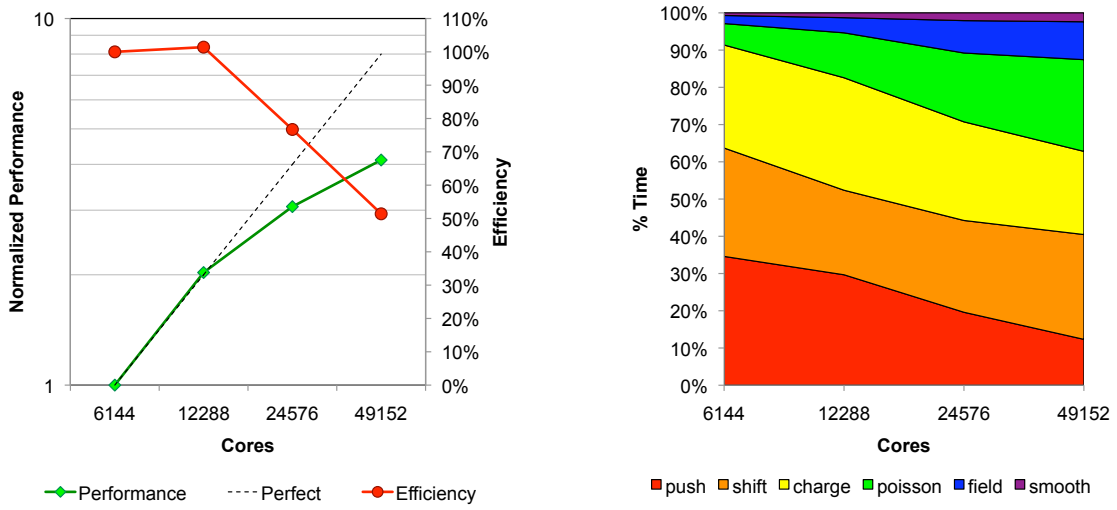

Figure 6: Strong-scaling results on Hopper using the D96 problem. (left) performance and efficiency as a function of concurrency, normalized to 6144 cores. (right) the breakdown of runtime among phases.

the actual solve times remain almost constant, but the time spent in push, shift, and charge decrease by up to 11.5×. Mitigating the solver scaling impediment requires either parallelization of the 2D solvers across *npartdom*, something more advanced implementations of gyrokinetic simulations already implement with PETSc, or increasing the particle density proportionally with *npartdom* via weak scaling simulations. Nonetheless, the unique characteristics of particle distributions within a poloidal plane allowed push to attain super-linear scaling, while shift and charge improved by 4.3 and 5.1× respectively.

7. CONCLUSIONS

In this work, we explore the impact of multicore-specific optimizations on a production gyrokinetic fusion application previously tuned for single core parallel systems. The global capability of the GTC code is unique in that it allows researchers to study the scaling of turbulence with the size of the tokamak and to systematically analyze important global dynamics. Our work explores novel multi- and manycore centric optimization schemes for this complex PIC-based code, including multi-level particle and grid decomposition to alleviate the effects of irregular data accesses and fine-grained hazards, increasing computational intensity through optimizations such as loop fusion, and optimizing multiple diverse numerical kernels in a large-scale application. Results across a diverse set of parallel systems demonstrate that our threading strategy is essential for reducing memory requirements, while simultaneously improving performance by 1.22×, 1.35×, and 1.77×, and 1.34× on BlueGene/P, the Cray XE6, an Intel Cluster, and a Fermi Cluster (respectively). Additionally our methodology extends the potential concurrency of these simulations, allowing the potential ultra-scale experiments of ITER-sized fusion devices.

The B20 problem provided insights into the scalability and performance optimization challenges (CPU and GPU) facing GTC simulations. For example, our memory-efficient optimizations on CPUs improved performance by up to 1.77× while reducing memory usage substantially. Proper use of threading placed simulations in the ideal regime where they are limited by the performance of on-node charge and push calculations. Despite our highly-optimized GPU charge and push routines, we observe that GTC's data locality and fine-grained data synchronization challenges are at odds with the underlying GPU architecture's synchronization granularity and small cache sizes — thus mitigating the GPU's potential. The result is that GPU-accelerated GTC was substantially slower than optimized CPU implementation.

In terms of programming productivity, porting GTC to the GPU-accelerated Dirac cluster proved taxing given the need not only to implement architecture-specific transformations but also to create new, massively-parallel implementations of the shift phase. The lack of mature development and debug support tools for hybrid codes on large-scale systems impeded the productivity of this process.

To evaluate our methodology at scale, we explored strong-scaling D96 experiments on Hopper using up to 49,152 cores, and showed that our optimization scheme achieves a 1.6× speedup over the fastest reference version, while reducing memory requirements by 6.5 TB (a 10% savings). Results show that in the strong scaling regime, the time spent in solver remains roughly constant, while charge/push/shift exhibit parallel scalability. Thus, the redundancy in the solver (performing the same calculation (*npartdom* times) impedes scalability and efficiency, motivating exploration of solver parallelization schemes that will be the subject of future work. Additionally, future work will address the limited DRAM capacity of next-generation system via radial partitioning techniques that lower memory requirements.

Finally, results also highlight that raw performance per node is not necessarily the only relevant metric. We observe that Hopper and the Intel Cluster deliver the highest overall energy efficiency, while surprisingly, the BlueGene and Fermi Cluster delivered lower energy efficiency — in the case of Fermi consuming more power and delivering less performance than the Intel Cluster. As energy and power will ultimately constrain future machine scale, it is imperative that we understand the interactions between program, architecture, and energy efficiency today. Thus it is critical for the computational community to study diverse architectural comparisons on optimized complex applications, as presented here.

8. ACKNOWLEDGMENTS

All authors from Lawrence Berkeley National Laboratory were supported by the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05-CH11231. Dr. Ethier is supported by the DOE contract DE-AC02-09CH11466. Eun-Jin Im was supported by Basic Science Research program through the National Research Foundation of Korea (NRF), funded by the Ministry of Education, Science and Technology under grant number 2011-0005992. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

9. REFERENCES

[1] M. Adams, S. Ethier, and N. Wichmann. Performance of particle in cell methods on highly concurrent computational architectures. *Journal of Physics: Conference Series*, 78:012001 (10pp), 2007.

[2] E. Akarsu, K. Dincer, T. Haupt, and G. Fox. Particle-in-cell simulation codes in High Performance Fortran. In *Proc. ACM/IEEE Conference on Supercomputing (SC'96)*, page 38, Nov. 1996.

[3] E. Bertschinger and J. Gelb. Cosmological N-body simulations. *Computers in Physics*, 5:164 – 175, 1991.

[4] K. Bowers. Accelerating a particle-in-cell simulation using a hybrid counting sort. *Journal of Computational Physics*, 173(2):393–411, 2001.

[5] K. Bowers, B. Albright, B. Bergen, L. Yin, K. Barker, and D. Kerbyson. 0.374 Pflop/s trillion-particle kinetic modeling of laser plasma interaction on Roadrunner. In *Proc. 2008 ACM/IEEE Conf. on Supercomputing*, pages 1–11, Austin, TX, Nov. 2008. IEEE Press.

[6] S. Briguglio, B. M. G. Fogaccia, and G. Vlad. Hierarchical MPI+OpenMP implementation of parallel PIC applications on clusters of Symmetric MultiProcessors. In *Proc. Recent Advances in Parallel Virtual Machine and Message Passing Interface (Euro PVM/MPI)*, pages 180–187, Sep–Oct 1996.

[7] E. Carmona and L. Chandler. On parallel PIC versatility and the structure of parallel PIC approaches. *Concurrency: Practice and Experience*, 9(12):1377–1405, 1998.

[8] A. Danalis, G. Marin, C. McCurdy, J. Meredith, P. Roth, K. Spafford, V. Tipparaju, and J. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proc. 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU '10)*, pages 63–74. ACM, 2010.

[9] V. K. Decyk. UPIC: A framework for massively parallel particle-in-cell codes. *Computer Physics Communications*, 177(1-2):95–97, 2007.

[10] V. K. Decyk and T. V. Singh. Adaptable particle-in-cell algorithms for graphical processing units. *Computer Physics Communications*, 182(3):641–648, 2011.

[11] S. Ethier, W. Tang, and Z. Lin. Gyrokinetic particle-in-cell simulations of plasma microturbulence on advanced computing platforms. *Journal of Physics: Conference Series*, 16:1–15, 2005.

[12] S. Ethier, W. Tang, R. Walkup, and L. Oliker. Large-scale gyrokinetic particle simulation of microturbulence in magnetically confined fusion plasmas. *IBM Journal of Research and Development*, 52(1-2):105–115, 2008.

[13] R. Fonseca et al. OSIRIS: A three-dimensional, fully relativistic particle in cell code for modeling plasma based accelerators. In *Proc. Int'l. Conference on Computational Science (ICCS '02)*, pages 342–351, Apr. 2002.

[14] R. Hockney and J. Eastwood. *Computer simulation using particles*. Taylor & Francis, Inc., Bristol, PA, USA, 1988.

[15] C. Huang et al. QUICKPIC: A highly efficient particle-in-cell code for modeling wakefield acceleration in plasmas. *Journal of Computational Physics*, 217(2):658–679, 2006.

[16] The ITER project. http://www.iter.org/.

[17] JET, the Joint European Torus. http://www.jet.efda.org/jet/, last accessed Apr 2011.

[18] A. Koniges et al. Application acceleration on current and future Cray platforms. In *Proc. Cray User Group Meeting*, May 2009.

[19] W. Lee. Gyrokinetic particle simulation model. *Journal of Computational Physics*, 72(1):243–269, 1987.

[20] Z. Lin, T. Hahm, W. Lee, W. Tang, and R. White. Turbulent transport reduction by zonal flows: Massively parallel simulations. *Science*, 281(5384):1835–1837, 1998.

[21] K. Madduri, E. J. Im, K. Ibrahim, S. Williams, S. Ethier, and L. Oliker. Gyrokinetic particle-in-cell optimization on emerging multi- and manycore platforms. *Parallel Computing*, 2011. in press, http://dx.doi.org/10.1016/j.parco.2011.02.001.

[22] K. Madduri, S. Williams, S. Ethier, L. Oliker, J. Shalf, E. Strohmaier, and K. Yelick. Memory-efficient optimization of gyrokinetic particle-to-grid interpolation for multicore processors. In *Proc. ACM/IEEE Conf. on Supercomputing (SC 2009)*, pages 48:1–48:12, Nov. 2009.

[23] G. Marin, G. Jin, and J. Mellor-Crummey. Managing locality in grand challenge applications: a case study of the gyrokinetic toroidal code. *Journal of Physics: Conference Series*, 125:012087 (6pp), 2008.

[24] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/.

[25] H. Nakashima, Y. Miyake, H. Usui, and Y. Omura. OhHelp: a scalable domain-decomposing dynamic load balancing for particle-in-cell simulations. In *Proc. 23rd International Conference on Supercomputing (ICS '09)*, pages 90–99, June 2009.

[26] C. Nieter and J. Cary. VORPAL: a versatile plasma simulation code. *Journal of Computational Physics*, 196(2):448–473, 2004.

[27] L. Oliker, A. Canning, J. Carter, J. Shalf, and S. Ethier. Scientific computations on modern parallel vector systems. In *Proc. 2004 ACM/IEEE Conf. on Supercomputing*, page 10, Pittsburgh, PA, Nov. 2004. IEEE Computer Society.

[28] G. Stantchev, W. Dorland, and N. Gumerov. Fast

parallel particle-to-grid interpolation for plasma PIC simulations on the GPU. *Journal of Parallel and Distributed Computing*, 68(10):1339–1349, 2008.

[29] Top500 Supercomputer Sites. http://www.top500.org.