UC Berkeley
UC Berkeley Previously Published Works

Title
Optimization of parallel particle-to-grid interpolation on leading multicore platforms

Permalink
https://escholarship.org/uc/item/6z40z53j

Journal
IEEE Transactions on Parallel and Distributed Systems, 23(10)

ISSN
1045-9219

Authors
Madduri, K
Su, J
Williams, S
et al.

Publication Date
2012-09-07

DOI
10.1109/TPDS.2012.28

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Optimization of Parallel Particle-to-Grid Interpolation on Leading Multicore Platforms

Kamesh Madduri, Jimmy Su, Samuel Williams,
Leonid Oliker, Stéphane Ethier, Katherine Yelick

CRD/NERSC, Lawrence Berkeley National Laboratory Berkeley

Princeton Plasma Physics Laboratory, Princeton

Electrical Engineering and Computer Science Department,
University of California, Berkeley

October 8, 2013

Disclaimer

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor The Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or The Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or The Regents of the University of California.

Abstract

We are now in the multicore revolution which is witnessing a rapid evolution of architectural designs due to power constraints and correspondingly limited microprocessor clock speeds. Understanding how to efficiently utilize these systems in the context of demanding numerical algorithms is an urgent challenge to meet the ever growing computational needs of high-end computing. In this work, we examine multicore parallel optimization of the particle-to-grid interpolation step in particle-mesh methods, an inherently complex optimization problem due to its low computation intensity, irregular data accesses, and potential fine-grained data hazards. Our evaluated kernels are derived from two important numerical computations: a biological simulation of the heart using the Immersed Boundary method, and a Gyrokinetic Particle-in-Cell based application for studying fusion plasma microturbulence. We develop several novel synchronization and grid decomposition schemes, as well as low-level optimization techniques to maximize performance on three modern multicore platforms: Intel's Xeon X5550 (Nehalem), AMD's Opteron 2356 (Barcelona), and Sun's UltraSparc T2+ (Niagara). Results show that our optimizations lead to significant performance improvements, achieving up to a 5.6× speedup compared to the reference parallel implementation. Our work also provides valuable insight into the design of future auto-tuning frameworks for particle-to-grid interpolation on next-generation systems.

1 Introduction

Particle methods are used in a wide range of numerical applications, including those from fusion physics, astrophysics, computational chemistry, and fluid mechanics. A key computational challenge in these simulations is the associated all-to-all particle interaction calculations. Luckily, in many of these domains, long range forces may be approximated, smoothed, or ignored, allowing scientists to replace the computationally expensive $O(N^2)$ methods with efficient $O(N)$ or $O(N \log N)$ approaches. Unfortunately, adoption of these approximations, while reducing the computational complexity of the problem, introduces potential data hazards in a parallel environment – thus dramatically increasing the performance optimization challenges on memory-constrained multi- and manycore architectures.

In this work, we explore performance optimization of the $O(N)$ particle-mesh methods in the context of two diverse applications, a biological simulation of the heart using the Immersed Boundary (IB) method, and a Gyrokinetic Particle-in-Cell (PIC) based method for studying plasma microturbulence. To facilitate our numerous optimization investigations, we create compact benchmarks for each application's most challenging phase: particle-to-grid interpolation. The particle-to-grid interpolation phase accounts for a substantial fraction – in some cases, up to 50% – of the overall running time in these two applications (see [**?**, 1, 13, 16] for a step-wise execution time breakdown of these applications at various scales of parallelism). Our experimental setup and the problem configurations studied are consistent with the overall distributed-memory parallelization scheme employed in these two applications, i.e., we have extracted the key single-node computational routine in these applications, and the speedup achieved with our new optimizations translates to a proportional overall application performance improvement [22].

Constrained by the desire for memory-efficient solutions and the need to avoid potential race conditions, we explore a number of partial replication and data-synchronization strategies in this work. While grid replication with ghost regions is a common optimization strategy to reduce communication volume in distributed memory implementations, our work focuses on efficient shared memory replication strategies. Further, we study fine-grained synchronization schemes that complement the replication strategies. To gauge the broad value of our techniques, we evaluate performance on three modern multicore systems: Intel's Xeon X5550 (Nehalem), AMD's Opteron 2356 (Barcelona), and Sun's UltraSparc T2+ (Niagara). Finally, to ensure the highest quality results, we employ a number of low-level optimizations designed to maximize performance on these Non-Uniform Memory Access (NUMA) architectures.

Our work makes the following new contributions:

- To the best of our knowledge, our work is the first to identify and generalize the particle-to-grid interpolation computation across the IB and gyrokinetic PIC methods, and present applicable novel multicore-tuned algorithms. Our proposed schemes are sufficiently generic to be applicable to a broad spectrum of particle-mesh based applications.

- We explore the challenges of architecture-targeted optimization and memory-usage management in the context of complex data synchronization and irregular memory access patterns, and demonstrate that our methodology can achieve significant performance benefits on a variety of multicore platforms.

- We observe that the optimal implementation of the particle-to-grid interpolation kernel depends both on application details and parallel platform characteristics. With a unique application-specific combination of optimizations, we achieve up to a 4× and 5.6× performance improvement for the heart simulation and gyrokinetic PIC particle-to-grid phases, respectively.

2 Particle-Mesh Computations

Full long-range particle-particle force interaction simulations carry an unacceptably high computational complexity — $O(N^2)$ in the number of particles. Numerous techniques have therefore been developed to mitigate this force computation time. Particle-mesh computations are one such widely-used class of methods which find applications in astrophysics [3, 17], plasma physics [5], computational chemistry, fluid mechanics, and biology. As the name suggests, the basic strategy adopted in particle-mesh methods is to utilize an auxiliary grid to approximate the density of particles, as it varies in space and time. The key steps in a particle-mesh calculation include calculation of the charge density by assigning "charge" to the mesh points (referred to as the "particle-to-grid interpolation" step), solving the field potential equation (for instance, by solving Poisson's equation to calculate electrostatic or gravitational potential) on the mesh, calculation of force field based on the mesh potential, and then acceleration of particles based on the computed grid forces ("grid-to-particle interpolation" step). All the subroutines involving particles in a particle-mesh method have a computational complexity proportional to the total number of particles. Furthermore, this approach has the advantage of removing close encounter collision effects, thus allowing computation of the collective effects in a system with a smaller number of simulated particles.

Conceptually, the particle-to-grid interpolation step is analogous to generating a 3D histogram. However, unlike simple histograms in which items have integer coordinates and increments, particles located using floating-point coordinates update a bounding box of grid points with a fraction of their charge (interpolation). Due to this indirection (a scatter-add), randomly localized particles often suffer from poor cache reuse. The challenges of this deposition step become far more significant when one contemplates the RISC (Reduced Instruction Set Computing) nature of modern processors, as floating-point increment is typically not an atomic operation. This can prohibit or impede both intra- and inter-thread parallelization of the particle array as multiple particles may attempt to update the same grid points simultaneously.

We now present an overview of the two driver applications of our study, and compare their salient characteristics in the context of particle-to-grid interpolation.

2.1 Heart Code via Immersed Boundary Method

The immersed boundary (IB) method [27,30] is a general technique for modeling elastic boundaries immersed within a viscous, incompressible fluid. This methodology has been applied to several biological and engineering systems, including the simulation of blood flow in the heart [30], sound waves in the cochlea [4], the swimming of eels, sperm and bacteria, and platelet aggregation during blood clotting [26]. These simulations have the potential to improve our basic understanding of biological systems and aid in the development of surgical treatments and prosthetic devices. While it is generally not categorized a particle-mesh method, we note that IB shares many similarities with that class of computations. In this article, we study performance optimizations for a heart blood flow simulation code using the immersed boundary method. We will refer to this application in short as "Heart Code". Figure 1(a) presents a graphic from our heart simulation code: the heart tissue is visualized as gray elastic fiber and exerts a force on the (red blood) fluid in which it is immersed. An important and expensive step in the Heart Code is the "spread force" calculation in which heart fiber forces are spread to the neighboring fluid cells. This is reminiscent of the particle-to-grid interpolation step in particle-mesh computations.

(a) (b)

Figure 1: Advanced volume visualization of the Heart Code and GTC. (a) Heart tissue is modeled as elastic fiber (shown in gray) and exerts a force on the fluid it is immersed in (blood, depicted in red). (b) Electrostatic potential field created by the plasma particles in a GTC simulation.

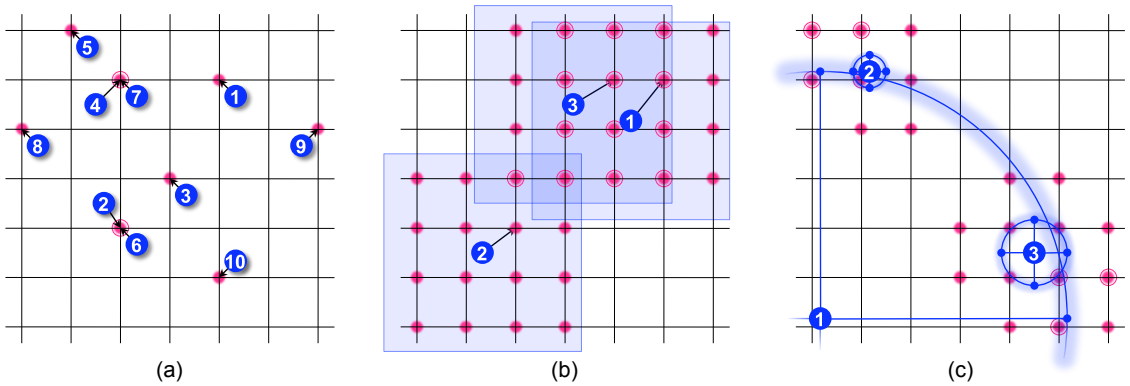

Figure 2: High-level visualization of particle-to-mesh interpolations. (a) 2D Histogram, (b) Smoothed approximation to the 2D Dirac Delta function used in the Heart Code, (c) Charge deposition in GTC (Gyrokinetic PIC simulation).

2.2 Plasma Simulation via Gyrokinetic PIC

Our second application driver for optimizing particle-to-grid interpolation is the Gyrokinetic Toroidal Code (GTC) [12, 21]. GTC was developed to study the global influence of microturbulence on particle and energy confinement, critical elements in understanding practical fusion power plant designs. It is a three-dimensional, fully self-consistent PIC code which solves the kinetic equation in a toroidal geometry. The charge update scheme in GTC differs from traditional PIC codes. In the classic PIC method, a particle is followed directly and the charge is distributed to its nearest neighboring grid points. However, in the gyrokinetic PIC approach used in GTC, the fast circular motion of the charged particle around the magnetic field lines is averaged out and replaced by a charged ring. To study low frequency instabilities that affect energy transport, GTC uses four points on the charged ring [20] to describe the non-local influence of the particle orbit. In this way, the full influence of the fast, circular trajectory is preserved without having to resolve it. However, this scheme inhibits straightforward shared-memory parallelism even further, since the updated positions need to be computed for each particle and at each time step. Furthermore, multiple particles may concurrently attempt to deposit their charge onto the same grid point. Figure 1(b) shows the 3D visualization of electrostatic potential in global, self-consistent GTC simulation of plasma microturbulence in a magnetic fusion device.

2.3 Cross-Cutting Overview

To illuminate the potential challenges to efficient single- and multicore parallelization, it is useful to visualize the differences (and implications) between various particle-mesh methodologies. To that end, we will briefly discuss three examples: a 2D histogram based on floating-point coordinates, the spread force phase of the Heart Code, and GTC's charge deposition phase, shown in Figure 2.

Figure 2(a) visualizes the grid updates performed in a 2D histogram of data points (particles) with floating-point coordinates. Such computations form the core of many PIC applications. Observe that during traversal of the 10 particles (numbered blue circles), there is no sequential or spatial pattern in which grid points (red circles) will be updated. Moreover, it is possible that two particles will update the same grid point, as seen with particles 4 and 7. On a shared memory parallel machine, the update by particle 4 can be concurrent with the update by particle 7. Therefore, it is essential to exploit a synchronization mechanism ensuring that parallel particle updates are ordered to avoid race conditions. Luckily, as data items (particles) are highly localized point-like objects, it is relatively easy to sort or bin particles and partition the grid across accordingly. This obviates the complexity of managing data races.

Figure 2(b) visualizes a 2D particle-to-grid interpolation scheme evocative of the Heart Code. Here particles are interpolated using an approximation to the Dirac Delta Function — effectively, they are now all equal-sized two-dimensional objects (blue patches). As the patch is interpolated onto the grid, 16 points must be updated (the 3D Heart Code updates 64 points in a 4×4×4 cube). Thus, the task of partitioning the grid (and thereby the particles) becomes more complex compared to the histogram example. For example, although particle 2 is bounded by a different box than either 1 or 3, the interpolation of the corresponding patch can create a race condition for grid point updates. Nevertheless, this problem may be efficiently parallelized by augmenting grid partitioning strategies with appropriate synchronization techniques.

Finally, Figure 2(c), visualizes a 2D particle-to-grid interpolation representative of the 4-point averaging technique used in gyrokinetic simulations like GTC. Here particles are neither points nor patches, but variable-sized circles. As a result, it can be extremely challenging to partition the grid or for that matter, even determine statically if a data race will occur. In this paper, we present a number of techniques designed to rectify these challenges and use compact kernel benchmarks similar to Figures 2(b) and (c) to evaluate them on leading multicore processors.

3 Related Work

In this section, we discuss previous work in optimization and large-scale parallelization of IB and PIC methods.

IB Optimization Efforts: Early parallel implementation studies of IB (Peskin and McQueen's heart code [31], Givelberg et al. cochlea construction [15]) were conducted on large shared memory machines. To motivate large-scale distributed memory parallelization of IB, Yau [33], Givelberg, and Yelick [16] observed that high-resolution computational grids are required to reduce numerical error in simulations and to incorporate finer details of the system into the model. They studied parallel performance and scalability of their Titanium language-based implementation up to 128 processors. The key challenges in a distributed memory implementation include capturing interactions between the fluid and immersed boundaries (i.e., the "spread force" step that we investigate in this paper), and load-balancing the computation as the immersed boundary structure and position changes over time. To the best of our knowledge, there has been no prior work on multicore tuning and optimizations for speeding up the immersed boundary method.

PIC Optimization Efforts: VPIC [7], OSIRIS [14], UPIC [10], and QuickPIC [18] are popular

frameworks from diverse application areas to express PIC computations. The ordering of particles impacts the performance of several PIC steps, including particle-grid interpolation. Prior work by Bowers [6] and Marin et al. [25] looks 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 [9], and OhHelp [28] is a library for dynamic rebalancing of particles in large parallel PIC simulations. Koniges et al. [19] report performance improvements by overlapping computation with inter-processor communication for gyrokinetic PIC codes. GTC performance has been well-studied on several large-scale parallel systems [1, 13, 29]. Prior work also examines expressing PIC computations via different programming models [2, 8].

Stanchev et al. [32] investigate GPU-centric optimization of the particle-to-grid interpolation step in PIC simulations with rectilinear meshes. Decyk et al. [11] discuss porting a 2D electrostatic code extracted from the UPIC framework to GPUs. In our prior work on multicore optimizations for GTC's charge deposition kernel, we introduce various grid decomposition and synchronization strategies [24] that lead to a significant reduction in the overall memory footprint, in comparison to the prior MPI-based GTC implementation. We utilize these strategies to develop a novel multicore-optimized GTC implementation, and demonstrate up to a 1.6× performance improvement over the prior GTC MPI-based code, on several leading multicore and manycore platforms [23].

4 Experimental Setup

In this section, we describe the evaluated architectures, the extracted kernels and specific problem instances, as well as the common benchmarking methodology.

4.1 Computational Platforms

This study explores the benefits of our threaded kernel across a variety of leading multicore platforms, summarized in Table 1. To mitigate the impact of long latency snoopy coherency protocols, we limit ourselves to dual-socket symmetric multiprocessors (SMPs). In the near future, as the number of cores per socket continues to scale, we expect on-chip coherency to remain manageable and bounded. However, as the number of sockets scale, the snoopy protocol can become a performance impediment. Thus we expect dual-socket multicore SMPs to be reasonable proxies for future multicore systems.

AMD Opteron 2356 (Barcelona): The Opteron 2356 (Barcelona) is AMD's quad-core processor offering. Each Opteron core runs at 2.3 GHz, has a 64 KB L1 cache, and a 512 KB L2 victim cache. In addition, each chip instantiates a 2MB L3 quasi-victim cache that is shared among all four cores. Each Opteron socket includes two DDR2-667 memory controllers providing up to 10.66 GB/s of raw DRAM bandwidth. Sockets are connected via a cache-coherent HT link creating a coherency and NUMA network for this 2-socket (8 core) machine.

Intel Xeon X5550 (Nehalem): The Nehalem processor is a recent enhancement to the Intel "Core" architecture, and represents a dramatic departure from Intel's previous multiprocessor designs. It abandons the front-side bus (FSB) in favor of on-chip memory controllers. The resultant QuickPath Interconnect (QPI) inter-chip network is similar to AMD's HyperTransport (HT), and it provides access to remote memory controllers and I/O devices, while also maintaining cache coherency. Nehalem is novel in two other aspects: support for two-way simultaneous multithreading (SMT) and TurboMode. The latter allows one core to operate faster than the nominal clock rate under certain workloads. On our machine, TurboMode is disabled due to its inconsistent timing behavior. The system used in this study is a dual-socket, quad-core 2.66 GHz Xeon X5550 with a total of 16 hardware thread contexts. Each core has a private 32 KB L1 and a 256 KB L2 cache,

Table 1: Overview of evaluated multicore SMPs.†1 per socket.

Core Architecture	**AMD Barcelona**	**Intel Nehalem**	**Sun Niagara2**
Type	superscalar out-of-order	superscalar out-of-order	multithreaded dual-issue
Clock (GHz)	2.30	2.66	1.16
DP GFlop/s	9.2	10.7	1.16
L1 Data Cache	64 KB	32 KB	8 KB
private L2 cache	512 KB	256 KB	—
System	**Opteron 2356 (Barcelona)**	**Xeon X5550 (Gainestown)**	**UltraSparc T5140 (Niagara)**
sockets	2	2	2
cores (threads) per socket	4 (4)	4 (8)	8 (64)
Memory Parallelism	HW prefetch	HW prefetch	Multithreading
Shared last-level caches	2×2 MB†	2×8 MB†	2×4 MB†
DRAM Capacity	16 GB	12 GB	32 GB
DRAM Pin Bandwidth (GB/s)	21.33	51.2	42.66(read) 21.33(write)
DP GFlop/s	73.6	85.3	18.7

and each socket instantiates a shared 8 MB L3 cache. Additionally, each socket integrates three DDR3 memory controllers operating at 1066 MHz, providing up to 25.6 GB/s of DRAM bandwidth to each socket. In comparison to the Barcelona system used in this paper, Nehalem has a similar floating-point peak rate but a significantly higher memory bandwidth and cache capacity.

Sun UltraSparc T2+ (Niagara): The Sun "UltraSparc T2 Plus", a dual-socket × 8-core Niagara2-based SMP, presents an interesting departure from mainstream multicore design. Rather than superscalar out-of-order execution, each of the 16 strictly in-order cores supports two groups of four hardware thread contexts (Chip MultiThreading or CMT) — providing a total of 64 simultaneous hardware threads per socket. Each core may issue up to one instruction per thread group assuming there is no resource conflict. The CMT approach is designed to tolerate instruction, cache, and DRAM latency through fine-grained multithreading. Niagara has no hardware prefetching, and software prefetching only places data in the L2 cache. Multithreading may hide instruction and cache latency, but may not fully hide DRAM latency. Our machine is clocked at 1.16 GHz, does not implement SIMD (single instruction, multiple data), but has an aggregate 64 GB/s of DRAM bandwidth in the usual 2:1 read:write ratio associated with FBDIMM. As such, it has significantly more memory bandwidth than either Barcelona or Nehalem, but has less than a quarter the peak flop rate. With 128 hardware thread contexts, we will show that Niagara poses a number of parallelization challenges in our of strong-scaling regime that we do not encounter in either the Nehalem or Barcelona systems.

4.2 Compact Benchmarks and Problem Instances

For our examined applications, we extract the key performance bottlenecks — particle-to-grid interpolation — and encapsulate them into standalone benchmarks. We also define a space of important problem instances.

4.2.1 Heart Code Kernel

The Heart Code application is a shared memory threaded code based on an earlier distributed-memory version [16]. To create a standalone Heart Code benchmark, we extracted its most challenging kernel, the spread force calculation, which is essentially a particle-to-grid interpolation. Nominally, heart fibers are clustered together into the shape of a heart. However, for simplicity, we represent the heart by a sphere of fibers (particles) centered in a cubical fluid grid in our benchmark. Note that all heart fibers are small, equally sized patches that update a 4^3 cube of fluid cells.

Overall, in order to perform particle-to-grid interpolation for one particle (heart fiber), the kernel must perform approximately 312 floating-point operations calculated via the Performance Application Programming Interface (PAPI). The kernel must read 48 bytes of data from the array-of-structures particle array (roughly 50% spatial locality), and with good grid locality, the algorithm can attain an arithmetic intensity of nearly 6.5 flops per byte. At this level, the code is likely compute-bound.

The three input parameters in the Heart Code used to describe the test simulation are the dimensions of the cubical grid (*i.e.* resolution in X, Y, and Z), the radius of the heart, and the number of heart fibers (particles), outlined in Table 2. We fix the dimensions of the fluid grid to be 256^3, vary the average density of particles (number of particles / cubical grid points) from 0.19 to 4.77, and the radius of the heart (from 25 to 100); the latter is labeled as *small, medium, and large*. Particles (heart fibers) are randomly distributed on the surface of the heart sphere such that each octant has the same number of particles, and the density on the surface of the sphere is sufficiently high for the underlying physical simulation. These parameters represent perturbations to commonly-used configurations.

Varying the number of particles affects the particle density (temporal locality) on the sphere surface (but not the cache working set size). We can also impact the number of grid points touched as well as the cache working set size by varying the size of the sphere. When particles are sorted in Z (the slowest varying dimension of the fluid grid), we may constrain the cache working set size to a few XY planes. The working set is therefore expected to be directly proportional to the sphere's radius, while the number of grid points touched should be proportional to the square of the sphere's radius (up to 4 MB). As the minimum cache quanta is a 64 Byte block, we expect working sets to start at 23 KB for a radius equal to 24 and increase linearly with the sphere's radius size. Such estimates are critical to understanding when progressively lager problems will fall out of successive cache levels.

4.2.2 GTC Kernel

The reference GTC version is an MPI code in which (with sufficient parallelism) each process owns one poloidal plane and the particles between its plane and the next in the toroidal direction (long way around the torus). In order to perform the various interpolations, each process must also possess a copy of the next toroidal plane; thus, particles exist in a confined 3D space while the grids are effectively 2D. As with the Heart Code, parallelization of the charge deposition (particle-to-grid interpolation) is the most challenging step. To that end, we preserve the MPI-style decomposition but extract only the charge deposition kernel. For the threaded implementation, all MPI calls are removed in favor of shared memory accesses. Similar to the Heart Code, we evaluate the particle-to-grid interpolation phase, and as such, the benchmark only loops over this one kernel.

The GTC benchmark streams through all particles, calculates the 16 grid addresses to update for each poloidal plane (approximately 115 flops plus one square root), and interpolates charge on up to 32 grid points (2 planes, 16 per plane, each with 2 flops). This amounts to a total of about

Table 2: Summary of Heart Code and GTC problem instances.

Heart Spread Force	**Small (S)**	**Medium (M)**	**Large(L)**
Grid Dimensions	256^3	256^3	256^3
Heart Radius	25	50	100
total grid points	16,777,216	16,777,216	16,777,216
grid points touched	31,500	125,700	502,500
particles per (cubical) point	*— total number of particles —*		
0.19	3.2M	3.2M	3.2M
0.95	16M	16M	16M
4.77	80M	80M	80M

GTC charge deposition	**Small (S)**	**Medium (M)**	**Large(L)**
mzeta	1(+ghost)	1(+ghost)	1(+ghost)
mpsi	192	384	768
mthetamax	1408	2816	5632
mgrid	151,161	602,695	2,406,883
total grid points	302,322	1,205,390	4,813,766
total grid points touched	302,322	1,205,390	4,813,766
micell (#particles/mgrid)	*— total number of particles —*		
2	0.3M	1.2M	4.8M
10	1.5M	6.0M	24.0M
50	7.6M	30.0M	120.0M

180 floating-point operations per particle. Thus, with perfect grid locality, the 40 bytes of data per particle that must be read, and the 128 bytes of data writes, limit the computation to a memory-bound state. If less grid locality is attained, the arithmetic intensity, and thus the performance, will diminish.

There are several input parameters in GTC to describe the test simulation, summarized in Table 2. The ones most relevant to the charge deposition kernel are the size of the discretized toroidal grid, the total number of particles, and the Larmor radius distribution of the particles for the four-point gyrokinetic averaging. Often one replaces the number of particles with the average particle density as measured in the ratio of particles to grid points (labeled as *micell*). Three coordinates describe the position of a particle within the discretized 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 dimensions of a toroidal grid segment are: *mzeta*, *mpsi*, and *mthetamax*. In this paper, we explore three different grid problem sizes, labeled *small, medium, and large*, and experiment with average particle densities of 2, 10, and 50 particles per grid point. Particles are uniformly randomly distributed throughput the grid (unlike the Heart Code where particles are constrained to a spherical subset of the grid). Note that these settings are similar to the ones used in prior experimental studies and GTC production runs [1, 13].

The Larmor radius of a particle ρ is dependent on the value of the particle's magnetic moment and the local value of the magnetic field B. The magnetic moment of the particle, which does not vary during the simulation, is given by $B \cdot \rho^2$. As B changes with the position of the particle, ρ also varies. For all three evaluated GTC problem sizes, the maximum Larmor radius (a function of several other GTC parameters) turns out to be roughly $mpsi/16$. Further, the radii values are initially chosen from a uniform random distribution. In Section 5.2 and 6, we discuss how this

choice affects the parallelization strategies we employ.

Finally, note that varying the number of particles changes the particle density (temporal locality) throughout the grid, but not the cache working set size. Varying the size of the grid changes the number of grid points touched as well as the cache working set size, since the latter is proportional to the Larmor radius (which is in turn proportional to ψ). Generally, we expect the cache working set size to be up to 12% of the grid size.

4.2.3 Performance Expectations

As these parameters are varied, GTC and the Heart Code will exhibit dramatically different changes in chance working set size, memory capacity, and arithmetic intensity. Given that GTC is moderately memory-bound, since it streams through particles, we expect GTC performance to be dependent on the architectural bandwidth to compute balance. Conversely, as the Heart Code is moderately compute-bound, its performance (on tiny problems) is expected to be tied to the peak computational rate of the platform (without SIMD). Additionally, GTC's cache working set size is extremely large and increases directly proportionally with the grid size. We therefore expect GTC performance to drop quickly with increasing grid size due to cache capacity limits. However, the Heart Code's working set is proportional to the square root of the number of grid points touched (those on the surface of the heart sphere) and the cube root of the total grid points in the fluid grid; thus it is expected to be less sensitive to cache effects with growing radii.

4.3 Common Benchmarking Setup

Reference Implementations: As the original GTC version is a Fortran application whose parallelism is expressed via MPI, it is included as a reference baseline. The MPI implementation statically partitions the particles within a toroidal segment among the participating MPI tasks. Each task maintains a private copy of the two bounding poloidal planes, and may thus deposit charge independently. An MPI reduction is performed to reduce all P copies of the grid to one. We explore a range of degrees of parallelism when running the MPI implementation on the SMPs. When benchmarking GTC, we perform successive iterations of local charge depositions and MPI reductions. The Heart Code is inherently a threaded application with no MPI equivalent; thus no distributed-memory baseline is presented.

PThreads Versions: To evaluate performance of the threaded implementations using the PThreads intrinsics, we employ a SPMD (single process, multiple data) inspired threading model in which communication of particles or updates to grid values is handled through loads and stores to shared memory rather than message passing (via any conduit). Typically, P threads are created at startup, and the particles are statically partitioned, allow them to initialize their data, and then benchmark repeated charge depositions. Additionally, loops are parallelized over particle updates, and the Pthreads loop partitioning is static and uniform.

Timing and GFlop/s: High-precision cycle counters are used on the evaluated platform to measure total time for ten iterations of charge deposition. We manually counted the number of floating point operations (flops) within the function (where multiply, add, or divide is considered one flop), and express average GFlops/s as the ratio of total flops to the average execution time over ten iterations.

Build Software Environment: The Pthreads codes are built on the three machines using the GNU C compiler and aggressive architecture-specific optimization flags. We use GNU Fortran90 compiler with identical optimization flags to compile the GTC Fortran/MPI code, and the MPICH2 library (version 1.0.8) with a fast shared memory communication device (`ch3:shm` on the Nehalem

and Niagara systems, and `ch3:nemesis` on the Barcelona system).

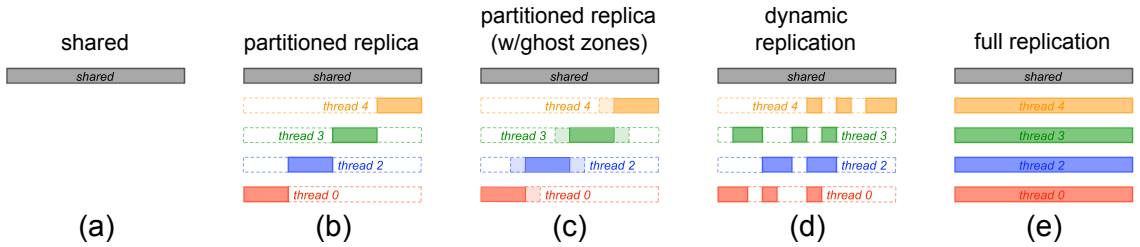

Figure 3: The five grid decomposition/replication strategies we utilized in the Pthreads implementations. Note, color denotes thread-private data. As the replication in (d) is particle-distribution dependent, we only show an arbitrary replication pattern.

5 Particle-to-Grid Optimization

In this section we discuss the various optimization strategies, focusing on synchronization and replication techniques, as well as low-level tuning details.

5.1 Synchronization Optimizations

In traditional MPI implementations of PIC methods, such as the original GTC version, the density grid is entirely replicated among processes, allowing autonomous updates of local copies without potential conflicts. MPI communication then proceeds to reduce all these grid copies into one, while redistributing the data for the solve phase. Unfortunately, such approaches are wasteful with respect to increasingly power-hungry DRAM. Luckily, modern multicore SMPs allow us to trade multiple processes per SMP for multiple threads per process; thus eliminating redundant grid copies for one copy in shared memory. However, the particle-to-grid interpolation includes a scatter-increment operation, creating the potential for data hazards where multiple processes may attempt to simultaneously increment the same memory location. In this section, we discuss our solutions to address this fine-grained data synchronization challenge.

One obvious solution to this problem is to implement a lock for each element requiring synchronized access, referred to here as fine-grained locks. This approach requires one lock for each grid point (a 64-bit double-precision number). Clearly, fine-grained locks involve both a substantial computational and memory overhead, as a lock must be acquired and released for every grid update. However, the cost of these updates can be amortized by assigning one lock to multiple grid elements. As there is a substantial 3D spatial correlation between grid updates in both Heart Code and GTC, we define two additional locking strategies for each that vary the number of grid points locked at a time. These are referred to as medium and coarse-grained locks. The details of how neighboring grid cells are grouped for a given lock is application-dependent and is discussed in the following subsections.

Our final solution for the synchronization problem is to emulate atomic floating-point increment via the atomic compare-and-swap instruction. On the x86 machines we wrote hand-crafted assembly routine to accomplish this, while on the SPARC architecture we utilize the CAS (compare-and-swap) intrinsic.

5.1.1 Heart Code Synchronization Notes

Although all four synchronization approaches were explored in the Heart Code, we found atomics consistently delivered the best performance by a significant margin. Therefore, we only present the Heart Code atomics data. Since each particle updates 64 grid points, the number of atomic updates is given by $64mi$ (where mi is the total number of particles). The ratio of the number of updates to the number of grid points touched (a measure of contention at each grid point) varies from 407 to $81,270$ and is based on the grid size and the effective particle density.

5.1.2 GTC Synchronization Notes

In GTC, fine-grained locking places one lock variable around every grid point. The number of fine-grained synchronous updates (i.e., lock, update, unlock) is $32mi$, and the ratio of the number of updates to the number of updated locations is $32micell$ (between 64 and 1600 for the GTC problem configurations studied in this paper).

We can utilize application-specific knowledge to reduce locking overhead. Note that in our shared memory GTC benchmark, the number of poloidal planes (*mzeta*) is always one. As such, particles may move only in the constrained 3D space between the two poloidal planes (ζ and $\zeta+1$). As the interpolation step will always update the same point (same ψ and θ) in both planes (different ζ), we choose to coalesce the locks for the points at (ψ, θ, ζ) and $(\psi, \theta, \zeta+1)$ into a single lock that controls access to both grid points. We call this strategy medium-grained locking. We reduce the number of lock-based updates to $16mi$ in this approach, but the ratio of number of updates to number of updated locations remains the same as the fine-grained locking approach.

In the four-point averaging method used in GTC, each charged ring (particle) is represented by four points. Two of the nearest grid neighbors of each one of these points lie on the same flux surface of constant ψ. Thus we define a *coarse-grained lock* as one that places all grid points at constant ψ (but any θ or ζ) behind a single lock. Doing so allows as many as four points per lock/unlock process. As different locking strategies are explored, from fine-grained locks to coarse-grained locks, we progressively amortize the overhead of locking by updating one, two, or four grid points; conversely, the available parallelism is reduced from *mgrid* to *mpsi*.

5.2 Partial Replication Optimizations

Although exploration of various synchronization strategies allows some freedom to amortize the overhead associated with updates to the shared grid, we explore additional amortization via grid replication. This approach makes subsets of the replicas thread-private, where one given thread may update certain grid points in the replica. Doing so allows the elimination of synchronization, in exchange for global reduction overhead at the conclusions of the updates. Additionally, there is potential for locality benefit as shared data reduces the volume of thrashing between the cores within a socket or between the sockets of an SMP. The contention estimates in the previous section assume a uniform random distribution of particles, and these partial replication schemes further reduce the synchronization overhead by exploiting particle ordering.

To explore this approach, we define five geometrically-based grid replication strategies. Their stylized visualizations are shown in Figure 3 in which a high-dimensional grid has been reduced to a 1D box. The solid portions of the box represent the subsets of the grid "owned" by that thread. Reading the figure from left to right, shows a reduction in locking overhead but an increase in memory usage combined with more expensive reductions. In the rest of this subsection, we discuss the five approaches and detail how we map these generic strategies to the requirements of the Heart Code and GTC.

Shared Grid: In the baseline shared grid approach, there is one copy of the grid shared by all threads. Selecting the appropriate synchronization approach must therefore be handled carefully. Note that if each thread's particles are spatially binned, then each thread's cache will be mapped to portion of this array, creating a degree of locality. However there can be thrashing at these boundaries, as multiple threads may contend for the data. Since there is just one copy of the grid, there is no need for a reduction.

Partitioned Replica: When particles are spatially binned, we can create one replica of the grid that is statically and geometrically partitioned among threads (based on the distribution of particles among threads). Doing so allows the elimination of virtually all locks in exchange for updates to the replica (based on a simple inspection of a particle's coordinate). There are a few cases at the boundaries of these partitions where a thread will need to update a grid location outside its partition. As it may not access another thread's replica (due to the lack of reciprocal synchronization), the thread must instead update the shared copy via an appropriate synchronization mechanism. Once all P threads have completed their updates and pass through a barrier, all the data in the P partial replicas is added to the shared grid. Although this method doubles the grid memory capacity requirements and uses a simple reduction, it eliminates a substantial fraction of the synchronization operations for moderate thread parallelism (the savings are application-specific). At extreme thread parallelism, the fact that particles are not point masses results in large numbers of accesses to the shared grid.

Partitioned Replica (with ghost zones): To further eliminate synchronizations at the boundaries, a ghost region can be added to each thread's portion of the replica. The ghost region consists of the expected spatial overlap resulting from the grid points updated by a particle. This virtually eliminates all synchronization traffic, but requires an overhead (ghost zones) proportional to the number of threads. As such, with sufficient thread parallelism, the ghost zones dominate the memory capacity.

Dynamic Replication: When particle distributions are extremely non-uniform, the inflexibility of the static partition grid methods results in large reductions of zeros; such an approach is inefficient and should be avoided. To that end, we also propose a dynamic replication strategy in which on the first step, threads determine those points they will update and allocate 3D blocks from a pool of memory as replicas. Threads may subsequently update their private replica, and once done, must reduce these sparsely populated replicas with the shared grid. Such an approach completely eliminates the need for synchronized accesses to the shared grid, but requires complex indirection and a reduction at phase completion.

Full Replication: As a final comparison point, we include a full replication strategy. Here, every thread replicates the entire grid and thus mimics the MPI algorithm, exchanging message passing for loads and stores to shared memory. This approach eliminates all synchronization (thus our fifth synchronization strategy: "none") but requires copious DRAM memory and large reductions. Such strategies are acceptable at extremely high particle densities (when the particles per grid point is greater than the number of threads) or when the overhead of any synchronization is prohibitively expensive.

5.2.1 Heart Code Replication Details

The Heart Code requires particle updates that are spatially tightly grouped. Thus, after spatial decomposition, the grid elements touched by a given thread only slightly overlap with those by the next thread. Since the benefits of using partitioned replica with ghost zones would be insignificant, we only explore four grid replication strategies for the Heart Code: none (just the one shared grid), partitioned replica, dynamic replication, and full replication. The mapping of these techniques to

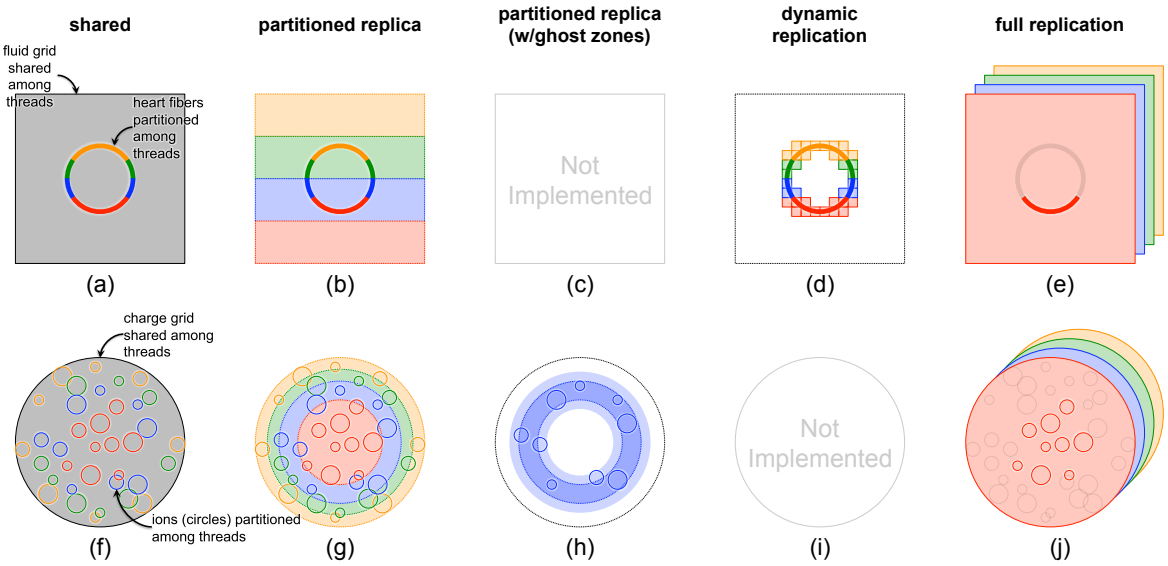

Figure 4: Mapping of the generic grid replication strategies of Figure 3 to the specific grid geometries of the Heart(a-e) and GTC(f-j) codes. For simplicity we show only a Y–Z cross-section for the Heart Code and the ψ–θ cross-section for GTC.

the specific geometry of the Heart Code's fluid grid is visualized in Figure 4 in which we show a Y–Z cross-section of the cubical grid. Note that due to the typical distribution of particles in the Heart Code (a spherical shell), it is likely that the full replication strategy will be wasteful. Nevertheless, we explore these results in order to quantify the impact.

The partitioned grid strategy partitions the cubical grid in the Z (slowest varying) dimension into fixed-sized slabs (one per thread). These P slabs are directly mapped to the P partitions of this replication approach. Remember, the sphere's radius is not directly tied to the dimension of the grid. Thus, a uniform slab-wise partitioning of the grid does not guarantee a uniform partitioning of the heart. As such, the ratio of updates to the replica (loads and stores) to updates to the shared grid (atomics) is highly variable. Some threads may update their replica almost exclusively, while others may only update the shared grid. The facets of this approach are visualized in Figure 4(b) via the red and blue threads.

Finally, in the dynamic replication strategy, replicas are rectahedral subdivisions of the original full grid. The dimensions of the replicas are restricted such that the full grid dimensions are an integer multiple of them. Although within a given simulation the replica size is fixed, we tune the code by exploring all legal replica grid dimensions and report the best results. As shown in Figure 4(d) the dynamic replicas are created only for the region in which heart fibers exist. This is clearly an efficient choice given the non-uniform distribution of heart fibers.

5.2.2 GTC Replication Details

For GTC, we implement all decomposition strategies except dynamic replication, as shown in Figure 3(a), (b), (c), and (e). The dynamic strategy is designed to achieve performance when the particle distribution is extremely nonuniform (contrary to GTC's characteristics). However, unlike the Heart Code, the particles in GTC are large circles whose average Larmor radius is roughly $mpsi/16$ and independent of their ψ coordinate. Thus, the four-point averaging scheme results in four widely dispersed clusters ($2 \times 2 \times 2$) of grid updates, making the partitioned grid with ghost zones an ideal map for GTC.

When mapping the stylistic replication strategies shown in Figure 3 to the complex toroidal geometry of GTC, we choose to partition in the radial ψ dimension. The result is visualized in Figure 4 (showing only the ψ–θ poloidal plane) with particles represented by variable sized charge rings. As the particle distribution is roughly uniform in the partitioned replica approach, we choose a decomposition that assigns each thread approximately the same number of points. When adding ghost zones the same bounds are used for ψ, but here the partitions are radially extended in both directions by $mpsi/16$ to account for the large Larmor radius. This creates substantial overlap and replication when using many threads. Nonetheless, the resultant ghost zones encompass the 4-point charge rings, as seen in Figure 4(h).

Note that while the partial replica schemes, combined with a spatial ordering of particles, significantly reduce the synchronization requirements, the average Larmor radius limits the efficacy of these schemes at high thread concurrencies. For thread-level parallelism below 16, there are virtually no shared grid updates in the partitioned replica with ghost zones scheme, as the ghost zones are large enough to encompass the particle updates at the partition boundaries. However, at 128-way thread concurrency, each radial partition would be approximately $mpsi/128$ rings wide. Thus, as high as 50% of the updates would go to the shared grid, as only sixteen of the 32 updates would be constrained to a local partition.

The one-dimensional radial partitioning of the grid also causes some load imbalance in parallel updates to the shared grid. The number of updates performed by each thread is the same, but the range of memory addresses touched by threads, and thus the total cost of performing the updates, is variable. The threads that are assigned the inner-most and outer-most radial partitions also lack ghost regions in one direction, leading to an imbalance in the number of shared grid updates. Further, in order to keep the grid reductions simple and limited to just three in all, the size of a ghost zone cannot be larger than the size of its adjacent radial partitions. Thus, at 128-way thread concurrency, each radial partition and the two ghost regions adjacent to the partition would be approximately $mpsi/128$ rings wide.

5.3 Additional Performance Tuning

In addition to the principal optimizations to manage data hazards, we explored a number of additional tuning strategies to maximize performance.

Sorting: Given our straightforward parallelization of the loop over particles it is important to devise a means of maximizing locality — to avoid a threads cache working set potentially as large as the entire grid size. We therefore perform a rudimentary spatial binning of particles, which is far less complex than a full sort since particles are only binned by one of their coordinates. As both codes use a 1D parallelization grid replication strategy, we simply bin particles along that same dimension. Thus for the Heart Code, we perform a spatial sort in the Z dimension, whereas in GTC we sort in radially in ψ.

Affinity: To achieve repeatable performance, thread pinning is used on all three architectures, allowing thread binding to an execution unit during the program runs. For the Pthreads implementations, we enumerate hardware thread contexts so that when increasing the number of threads, we exploit multithreading within a core, then multicore on chip, and finally multiple sockets on the SMP. The Linux or Solaris affinity routines accomplish this via inspection of `/proc/cpuinfo`.

NUMA: As all machines in our study are dual-socket SMPs, data is carefully initialized to avoid undesired NUMA effects. To that end, after threads are pinned, they allocate and initialize their relevant data structures including their partitions of the grid replicas and their subsets of the particle arrays. Failure to correctly implement this can dramatically impact performance.

Data Layout: The particle position, velocity, and other auxiliary data (six double-precision val-

Table 3: Optimizations implemented by kernel.

		Heart Code					GTC				
		synchronization					synchronization				
		coarse	medium	fine	atomics	none	coarse	medium	fine	atomics	none
Replication	None (Shared)	✓	✓	✓	✓		✓	✓	✓	✓	
	Partitioned	✓	✓	✓	✓		✓	✓	✓	✓	
	Partitioned+Ghost	–	–	–	–		✓	✓	✓	✓	
	Dynamic	✓	✓	✓	✓		–	–	–	–	
	Full					✓					✓

		Heart Code	GTC
Miscellaneous	Particle Sorting	✓	✓
	Thread Affinity	✓	✓
	NUMA Allocation	✓	✓
	Data Layout	✓	✓
	Loop Fusion	–	✓
	SIMD	–	x86-only

ues per particle in total) are stored in an array-of-structures representation in the original GTC Fortran/MPI code. Since our kernel requires only five of the six values in the charge deposition phase, we use a structure of arrays representation in the C version. This ensures maximal spatial locality and facilitates SIMDization. We also flatten all multi-dimensional grid arrays and align them to cache line boundaries during initialization. In the Heart Code, particles are stored in an array-of-structures representation. Multi-dimensional grid arrays are also flattened.

Loop Fusion: The charge deposition phase in the MPI/Fortran version of GTC comprises two distinct loops, one for inspecting the particles and computing the grid locations to increment charge densities at, and the other for performing the actual charge increments. Since both these loops iterate over the particle arrays, we fuse them to reduce the memory read traffic. Grid location address calculation is relatively simpler in the Heart Code, and the floating point operation count is dominated by the force increments. The spread force operations are done in a single loop.

SIMD: For GTC on the Barcelona and Nehalem systems, we use SSE (Streaming SIMD Extensions) intrinsics to SIMDize charge density increments at grid points that are consecutively laid out in ζ. The impact of this optimization varies with problem size, and we see the most benefit for smaller grid sizes. The SPARC ISA does not support double-precision SIMD instructions, and SIMDization has not yet been implemented for the Heart Code, and will be the subject of future work.

Fine Partitioning: In the partitioned grid approaches for GTC, we attempt to decompose the grid into contiguous annuli such that each region contains approximately the same number of grid points. However, note that the number of grid points per annulus varies, and annuli have more grid points as we move radially outward. For parallel runs where the number of threads may be of the same order as the number of radial flux surfaces (this is the case for some grid sizes on Niagara), this annulus-granularity partitioning leads to a load imbalance. Hence we relax this constraint and perform a two-dimensional partitioning of the grid, where non-overlapping regions of a single annulus may be assigned to different threads in order to balance the number of grid points per thread. We refer to this optimization as "fine partitioning".

Tuning: In the Heart Code, we search over several possible block sizes in the dynamic replication approach, and tune and report results for the best configuration. We do not perform any additional

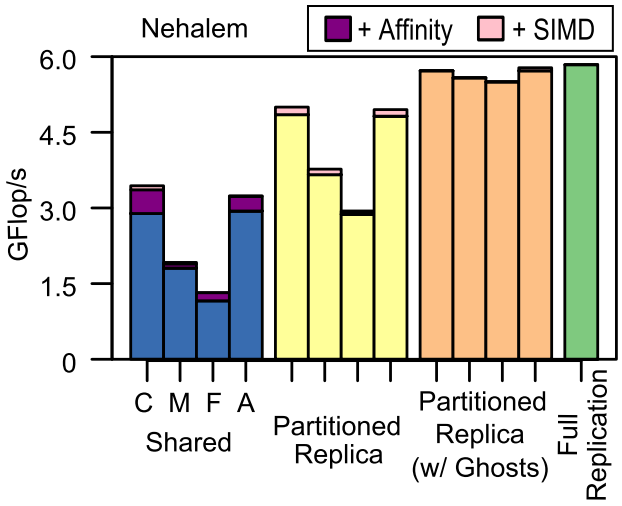

Figure 5: GTC parallel performance on Nehalem relative to varying replication and synchronization strategies using **(C)**oarse, **(M)**edium, **(F)**ine, and **(A)**tomics, for the M10 problem size. Results also show the impact of affinity and SIMD optimizations.

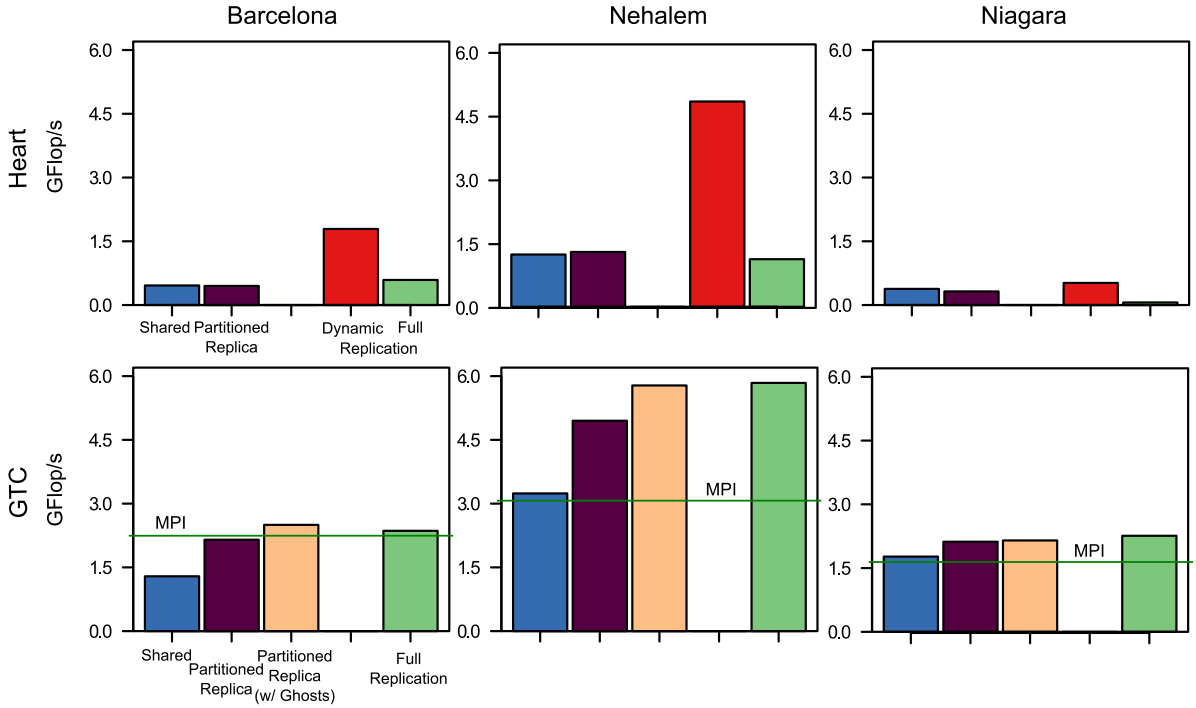

Figure 6: Performance relative to various grid replication strategies for the medium grid size and medium particle density problem.

low-level tuning in GTC.

Optimization Summary: Table 3 presents the optimizations employed by each kernel. Of the 25 possible choices, only the relevant combinations of synchronization and data replication are chosen. Additionally, SIMD is only utilized on architectures that support it (Barcelona and Nehalem).

6 Performance Results and Analysis

We now present Heart Code and GTC kernel performance. To provide analysis via extraction of salient performance bottlenecks, we vary problem size, particle density, approaches to data synchronization and data replication, as well as thread-level parallelism.

6.1 Synchronization and Replication Analysis

We begin by presenting the trade-off between synchronization (locks and atomics) and data replication in the context of GTC. Figure 5 shows performance as a function of synchronization strategy — **(C)**oarse, **(M)**edium, **(F)**ine, **(A)**tomics — for each replication strategy on Nehalem; recall that full replication does not require synchronization (Table 3). Note that experiments conducted to evaluate the trade-offs between locks and atomics on the Heart Code (not shown) determined that atomics always delivered significantly better performance.

Across all grid replication strategies, performance improves as the locking overhead is amortized (Fine to Coarse) with an additional benefit (Fine vs. Atomics) seen when amortizing Pthread's fine-grained locking overhead by direct use of atomics. Observe that at full parallelism on Nehalem (16-way threading), contention for locks does not impede performance. Another key result is that, using our partitioned replica with ghost zones replication in conjunction with atomic updates, we achieve performance on par with the naïve full replication strategy. Although the code is more complex than simpler full replication, our approach not only eliminates a global reduction, but dramatically reduces the grid memory requirements from $O(P \cdot mgrid)$ to $O(mgrid)$ (where P is the number of threads of execution). In GTC, a vast amount of computation is required to determine grid array indices and interpolate charge; thus the overhead of atomic updates is effectively amortized. As this ratio is unique to GTC, it is appropriate for all codes to explore the best synchronization and replication strategy.

Finally, we observe that the benefits of affinity and SIMDization are relatively small on the GTC kernel–architecture combination. Given that the baseline approach includes NUMA-aware allocation optimization, this ensures that the grid and particle arrays are evenly distributed across memory banks. The relatively small benefit with affinity pinning seems to indicate that the OS default thread pinning scheme for this case may be similar to our thread pinning strategy. SIMD is of little benefit as the code requires the streaming through a few million particles, and is memory-bound (for reasonable cache locality of the grid points).

6.2 Replication Parallel Performance Analysis

For each kernel, we specialize the five general grid replication strategies and apply them to our two codes. The Heart Code does not include the *replica with ghost* approach as particles are spatially confined, while GTC does not include *dynamic replication* as the particle distribution is uniform. Figure 6 presents results on the three evaluated architectures for the medium-sized grid with (on average) moderate particle density (approximately one per grid point for Heart, and ten per grid point for GTC). For each strategy, we present parallel performance using the best synchronization method as well as all relevant optimizations.

On the Heart Code, we observe shared and partitioned replica deliver comparable performance. This may seem surprising, as the partitioned replica should eliminate most atomic accesses. However, given the bulk synchronous nature of the replica algorithm, performance is limited to the slowest thread (as all threads have equal numbers of particles, the one with the most updates of the shared grid). For any moderate degree of parallelism, at least one thread's updates are entirely to the shared grid. Thus, it is no faster than the naïve shared implementation. The full replication

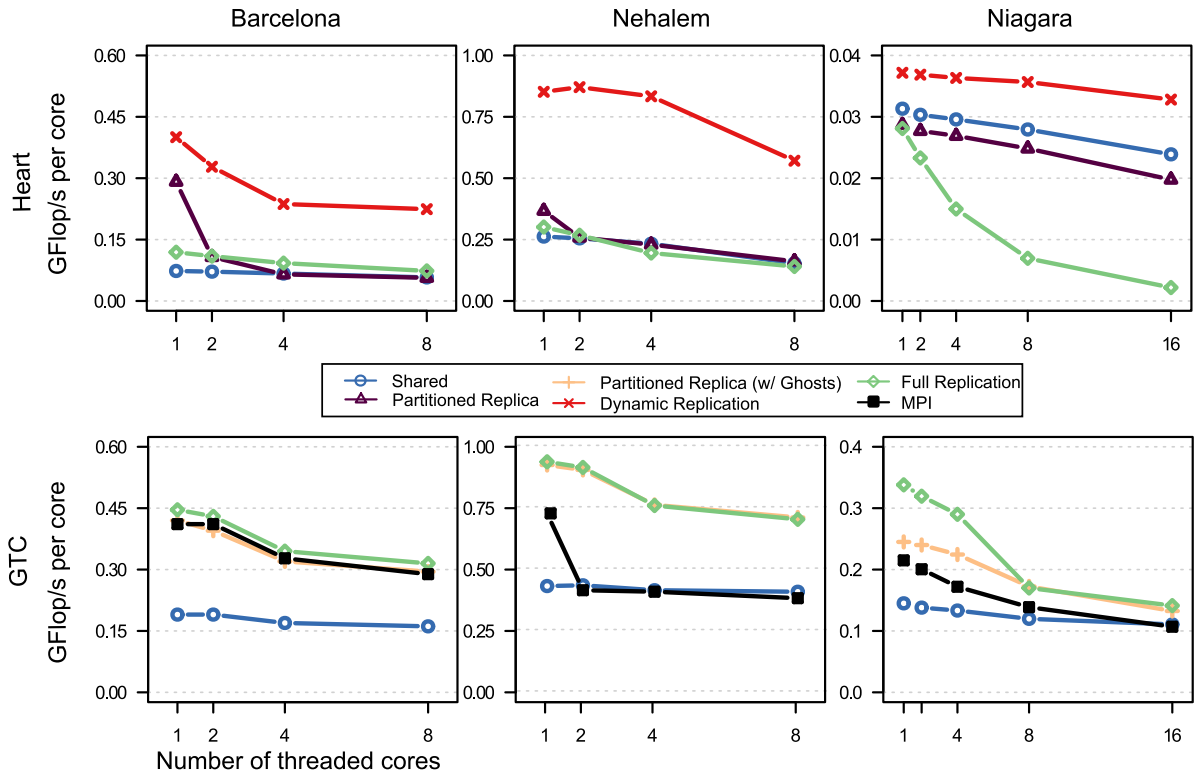

Figure 7: Parallel scaling as a function of grid replication strategy for the medium grid size and medium particle density problem. Note that the Niagara graphs are on different scales.

approach wastes (linearly with threads) flops, bandwidth, and memory capacity in order to avoid all synchronization. This results in comparable performance on low concurrency x86 processors, but very low performance on highly multithreaded processors such as Niagara, where the wasted flops and bandwidth from the reduction dominate. Clearly, the efficient dynamic replication provides the best of both worlds by eliminating most synchronization without wasting flops, bandwidth, or memory capacity (for nonuniform distributions).

On GTC, observe that strategies which provide some replication attain nearly twice the performance of the reference shared grid approach (on all three platforms). This demonstrates that replacing synchronized updates (locks or atomics) with regular loads and stores has a substantial benefit for the typical case. Additionally, we observe that there is no substantive performance benefit of using the memory-hungry full replication strategy. The green line represents the performance of the MPI implementation (algorithmically closest to the full replication strategy) of the same problem configuration. On Barcelona, observe that the performance of both the Pthreads and MPI schemes are quite similar. On Nehalem, no improvement was achieved when spawning 16 processes, indicating that the MPI implementation was unable to utilize the 2-way SMT support. Similarly, we were unable to spawn more than 64 processes on the Niagara system, and this may explain the performance gap between the shared memory full replication version and the MPI implementation. Further, our memory efficient versions outperform the MPI code, while only requiring a tiny fraction of the memory capacity. This difference is most profound on Niagara, where the memory for shared grids is roughly only 1% compared with MPI's distributed and replicated grids.

Overall results show that atomics are slow but useful in avoiding superfluous replication. Thus, the grid replication strategy must be judiciously chosen based on particle distribution and platform. Note also that machine performance is largely correlated with its bandwidth.

6.3 Replication Scalability Analysis

To fully understand the performance limitations of our grid replication strategies and the underlying architectures, we evaluate performance as a function of thread concurrency, as shown in Figure 7. Note that threads are enumerated so that we exploit SMT before multicore, and multicore before multisocket (until all cores are fully threaded). Thus, in Figure 7, the first data point fully exploits SMT on one core while the final data point is the only configuration to use the second socket.

For the Heart Code, observe that parallel efficiency of the partitioned replica approach drops rapidly on x86 machines. We believe this is an artifact of the aforementioned imbalance of atomics among threads. For shared and full replication, we expect the moderate increase in computation to be small for these particle densities and concurrencies. On Niagara, all partial replica approaches scale well as the relatively weak cores are likely heavily compute-bound, thus hiding the atomics overhead. However, since full replication requires a massive increase in computation and bandwidth for the 128-way reduction, parallel efficiency quickly drops. On all three architectures, as thread parallelism increases, the superiority of the dynamic replication strategy is manifested.

Examining GTC, the shared grid approaches deliver flat performance confirming that contention does not adversely affect performance at these concurrencies. As synchronization overhead is amortized by exploiting the partial or full replication strategies, a substantial boost in performance can be seen (although some performance is lost when scaling to higher thread parallelism). This is most evident on Niagara where the partitioned grid replica with ghost zones strategy can be challenging to load balance and results in a substantial time reduction due to relatively large ghost zones. The full replication presents an extreme example where the benefit of no synchronization is initially evident, but the cost of global reductions impedes parallel efficiency.

The MPI version of GTC uses the full replication strategy and scales reasonably well on most machines. Our MPI process pinning scheme is similar to the Pthreads versions, i.e., we first saturate a core, then a socket, and then run on multiple sockets. No additional performance improvement is attained between eight and sixteen processes on Nehalem, and initializing more than 64 was not possible on Niagara. Hence the parallel scaling results for Nehalem and Niagara correspond to one and four processes per core respectively. On Barcelona and Niagara, MPI parallel scaling tracks the full replication and partitioned replica strategies. However, on Nehalem, there is a sudden drop in MPI performance after which it tracks the shared grid approach. Regardless of performance, the memory requirements of such an algorithm are increasingly prohibitive.

On all architectures and implementations, results show good multicore and multisocket scalability, demonstrating that our approach effectively contains the respective working sets to each processor's cache and efficiently exploits the NUMA nature of these machines. Although the shared approach — which entirely requires atomic operations — scales well in virtually every case (except the multisocket Nehalem region), its performance is far below the potential when using data-aware selection of the appropriate grid replication strategy.

6.4 Perturbations in Problem Configuration

Finally, to understand the resilience of our implementations to perturbations in grid size (working set) and particle density (temporal locality) on the studied platforms, we examine a variety of problem parameters, shown in Figure 8. Although we only include the memory-efficient (no full replication) approaches, there are some problems that are larger than the available SMP memory capacity (unlabeled white boxes). A typical problem size is represented in the center of each heatmap.

Examining variations in Heart Code problem sizes, observe the substantial performance changes

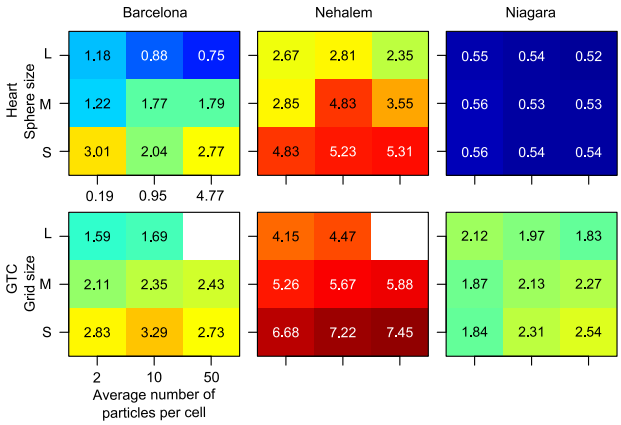

Figure 8: Parallel performance (in GFlop/s) of fastest memory-efficient variant for various problem configurations. Heart Code tunes block sizes for each configuration

with respect to sphere size for Barcelona and Nehalem. As the sphere's radius increases, the size of the working set of dynamic replicas grows. Eventually, the working set is so large that it falls out of the fastest caches, resulting in a performance drop. For higher densities, the increased temporal locality mitigates the impact of L1 or L2 cache misses and thus generally improves performance.

Additionally, observe that Niagara's performance is flat. This is not surprising, as the Heart Code is quite computationally intense and Niagara is a relatively weak (floating-point) processor. As the sphere sizes produce ~ 23 KB working sets, which will not fit in the 8 KB L1, but will all fit in the L2. Thus, there is no performance variability with respect to grid size. A similar issue arises with respect to particle density, where increased temporal locality on a compute-bound processor results in little performance change. As discussed in Section 6.2, the optimal strategy here is always dynamic replication.

Broadly speaking, we see a similar trend on GTC, with Niagara delivering flat performance, while the x86 systems see performance reductions with increasing grid size. Additionally, performance generally improves with increasing particle density. Owing to the lack of update locality inherent in gyrokinetic PIC simulations, the optimal data replication strategy is partitioned grid with ghost zones, while Niagara requires fine partitioning.

Although the Heart Code appears more sensitive to problem size, it actually has a much smaller working set than GTC (sometimes fitting in L1), which grows slowly with increasingly larger spheres. However, unlike local-store based platforms, many lines in set associative caches may go unused. Overall results show that compute-bound processors with small caches will be insensitive to variations in problem sizes, but at the cost of lower performance.

6.5 Cross-Architecture Comparison

Figure 9 presents summary performance after parallelization and optimization. The reference Heart Code computation corresponds to the shared grid version with particle sorting and atomic updates, while GTC's reference parallel version uses a shared grid with fine-grained locking and NUMA-aware memory allocation.

Observe that on most machines, parallelism significantly improves performance. Additionally, optimization (via synchronization and replication) delivers a further boost to parallel performance. Overall, our optimized implementation outperformed the reference serial case by up to 87.5× and 23.6×, and the reference parallel case by up to 4× and 5.6×, for the Heart Code and GTC, respectively. Additionally, Nehalem achieves the best performance as it has relatively superior flops

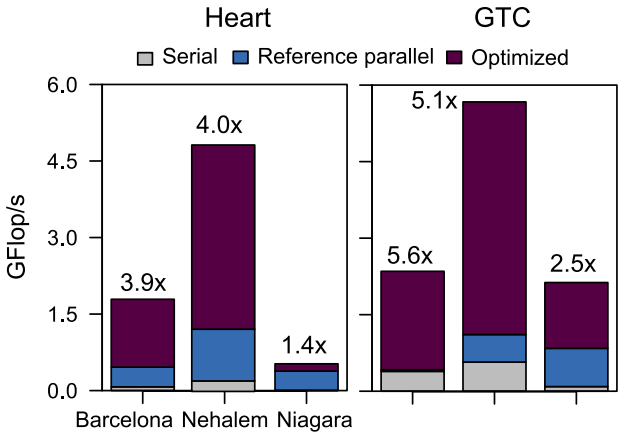

Figure 9: Summary performance impact of the various tuning strategies. Speedup numbers indicate the performance ratio of optimized implementation to the reference parallel code.

capability, bandwidth rate, and cache size. Barcelona delivers better performance than Niagara on the compute-intensive Heart Code, while achieving comparable runtimes on the more bandwidth-intensive GTC.

7 Conclusions

In this work, we explore a number of techniques designed to improve particle-to-grid interpolation – the most challenging phase of particle-in-cell computations — on modern multicore systems. Rather than the accepted flat replication strategy found in MPI implementations, we exploit the trend towards large manycore processors by selecting and optimizing an implementation in which the target grid is shared and with minimized replication. Balancing the resultant data synchronization and data locality challenges are taxing and resulted in 25 parameterizable implementations.

Results show that the optimal implementation and parameterization was unique to a particular application, target architecture, and particle distribution. For example, we found that in the Heart Code, the nonuniform distribution of heart fibers within the fluid grid demanded dynamic replication to minimize accesses to the shared grid and the complexity of the final reduction. Conversely, the spatially uniform distribution of particles coupled with non-localized grid updates arising from the Larmor radius pushed GTC towards the partitioned grid with ghost zones approach. Overall, we were able to reduce memory usage substantially while simultaneously significantly improving performance via our novel optimization schemes.

In future, we will continue to generalize our approach and encapsulate it behind a single interface that hides the complexity of performance optimization and tuning from the application scientist. Additionally, we will extend this concept to the distributed memory PGAS environment. Finally, the optimizations and analyses presented here are a critical step towards designing auto-tuning frameworks, which will be extended to numerous classes of particle methods and architectural platforms.

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. Dr. Ethier was supported by the DOE Office of Fusion Energy Sciences under contract number DE-

AC02-09CH11466. Additional support comes from Microsoft (Award #024263) and Intel (Award #024894) funding, and by matching funding by U.C. Discovery (Award #DIG07-10227). Further support comes from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems. We would like to express our gratitude to Intel and Sun for their hardware donations.

References

[1] M.F. 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.C. Fox. Particle-in-cell simulation codes in High Performance Fortran. In *Proc. ACM/IEEE Conference on Supercomputing (SC'96)*, page 38, November 1996.

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

[4] R.P. Beyer. A computational model of the cochlea using the immersed boundary method. *Journal of Computational Physics*, 98:145–162, 1992.

[5] C.K. Birdsall and A.B. Langdon. *Plasma Physics Via Computer Simulation*. McGraw Hill Higher Education, 1984.

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

[7] K.J. Bowers, B.J. Albright, B. Bergen, L. Yin, K.J. Barker, and D.J. 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, November 2008. IEEE Press.

[8] S. Briguglio, B.D. Martino 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.

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

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

[11] V.K. Decyk, T.V. Singh, and S.A. Friedman. Graphical processing unit-based particle-in-cell simulations. In *Proc. 10th International Computational Accelerator Physics Conference*, Aug–Sep 2009.

[12] S. Ethier, W.M. 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.

[13] S. Ethier, W.M. 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.

[14] R.A. 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, April 2002.

[15] E. Givelberg, J. Bunn, and M. Rajan. Detailed simulation of the cochlea: Recent progress using large shared memory parallel computers. Technical Report CACR-2001-190, California Institute of Technology, 2001.

[16] E. Givelberg and K. Yelick. Distributed Immersed Boundary simulation in Titanium. *SIAM Journal on Scientific Computing*, 28(4):1361 – 1378, 2007.

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

[18] 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.

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

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

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

[22] K. Madduri, K.Z. Ibrahim, S. Williams, E-J. Im, S. Ethier, J. Shalf, and L. Oliker. Gyrokinetic toroidal simulations on leading multi- and manycore hpc systems. In *Proc. ACM/IEEE Conf. on Supercomputing (SC 2011), to appear*, November 2011.

[23] 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`.

[24] 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)*, November 2009.

[25] 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.

[26] D.M. McQueen and C.S. Peskin. A three-dimensional computer model of the human heart for studying cardiac fluid dynamics. *Computer Graphics*, 34:56–60, 2000.

[27] R. Mittal and G. Iaccarino. Immersed boundary methods. *Ann. Rev. Fluid Mech.*, 37:239–261, 2005.

[28] 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.

[29] 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, November 2004. IEEE Computer Society.

[30] C.S. Peskin and D.M. McQueen. A three-dimensional computational method for blood flow in the heart: (I) immersed elastic fibers in a viscous incompressible fluid. *Journal of Computational Physics*, 81:372–405, 1989.

[31] C.S. Peskin and D.M. McQueen. Shared-memory parallel vector implementation of the immersed boundary method. *Journal of Supercomputing*, 11(3):213–236, 1997.

[32] 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.

[33] S.M. Yau. Experiences in using Titanium for simulation of immersed boundary biological systems. Master's thesis, UC Berkeley, 2002.