# Resource-Efficient, Hierarchical Auto-Tuning of a Hybrid Lattice Boltzmann Computation on the Cray XT4

Samuel Williams*, Jonathan Carter*, Leonid Oliker* John Shalf*, Katherine Yelick*†
*CRD/NERSC, Lawrence Berkeley National Laboratory
†CS Division, University of California at Berkeley

**ABSTRACT:** We apply auto-tuning to a hybrid MPI-pthreads lattice Boltzmann computation running on the Cray XT4 at National Energy Research Scientific Computing Center (NERSC). Previous work showed that multicore-specific auto-tuning can improve the performance of lattice Boltzmann magnetohydrodynamics (LBMHD) by a factor of 4× when running on dual- and quad-core Opteron dual-socket SMPs. We extend these studies to the distributed memory arena via a hybrid MPI/pthreads implementation. In addition to conventional auto-tuning at the local SMP node, we tune at the message-passing level to determine the optimal aspect ratio as well as the correct balance between MPI tasks and threads per MPI task. Our study presents a detailed performance analysis when moving along an isocurve of constant hardware usage: fixed total memory, total cores, and total nodes. Overall, our work points to approaches for improving intra- and inter-node efficiency on large-scale multicore systems for demanding scientific applications.

**KEYWORDS:** Lattice Boltzmann, Hybrid, MPI, Multicore, Auto-tuning

## 1 Introduction

Today's massively parallel processing machines are all built from shared-memory multicore processors. However, two major challenges arise in optimizing application performance on them: optimizing node-local computation, and optimizing communication between nodes. In this paper, we apply the well-established auto-tuning technique as a first step in optimizing single-node performance. However, rather than accepting for fact that either a cubical domain decomposition is optimal or that the auto-tuned implementation for a cubical subdomain is optimal for a non-cubical subdomain, we explore the performance benefits of co-tuning both the domain decomposition and the local computation in a resource- and time-efficient manner. Finally, perhaps one of the most controversial topics in programming today is examined — that of hybrid MPI. Rather than assigning one core to each MPI process, one uses multiple cores per MPI process via threading.

In this paper we quantify the application-level performance benefits of auto-tuning local computation, domain decomposition, and the division of computational resources among MPI processes. To that end, as an application, we selected Lattice Boltzmann Magnetohydrodynamics (LBMHD) — a structured grid based algorithm that simulates homogeneous isotropic turbulence in magnetohydrodynamics. Results show that for moderate sized problems on the Cray XT4, although the bulk of the performance gains come from single-thread optimization, an additional 17% performance boost is achieved through tuning the domain decomposition and balance between threads per process and processes per node.

The rest of the paper is organized as follows. Section 2 discusses background and previous auto-tuning efforts of our target application: LBMHD. Section 3 outlines the architecture, programming model, tools, and methodology used throughout the rest of the paper. Sections 4 and 5 introduce our approach to hybrid, distributed auto-tuning, present the XT4 performance results, and provide performance analysis. Finally, Section 6 provides some high-level insights resulting from this work, as well as some possible future directions.

## 2 LBMHD

In this section, we commence with a discussion of the history and computational physics of lattice Boltzmann methods. We will then proceed to the computa-

tional characteristics and previous auto-tuning efforts. By no means is this discussion extremely detailed or comprehensive. It is included to provide the fundamental knowledge and context.

## 2.1 Background

Lattice Boltzmann methods (LBM) have emerged from the field of statistical mechanics as an alternative to other numerical simulation techniques in the areas of fluid dynamics, complex fluid flows, interface dynamics, and quantum fluids [10]. The basic idea is to develop a simplified kinetic model that incorporates the essential physics and reproduces correct macroscopic averaged properties. In the field of computational fluid dynamics LBM have grown in popularity due to their flexibility in handling irregular boundary conditions and straightforward inclusion of mesoscale effects such as porous media, or multiphase and reactive flows. More recently LBM have been successfully applied to the field of magnetohydrodynamics [4, 7].

The LBM equations break down into two separate terms, each operating on a set of distribution functions; a linear free-streaming operator and a local non-linear collision operator. Implicit in the method is a discretization of velocities and space onto a lattice, where a set of mesoscopic quantities (density, momenta, etc.) and distribution functions are associated with each lattice site. The most common current form of LBM makes use of a Bhatnagar-Gross-Krook [1] (BGK) inspired collision operator — a simplified form of the exact operator that casts the effects of collisions as a relaxation to an equilibrium distribution function on a single timescale. For the fluid dynamics case, in discretized form, we write:

$$f_a(\mathbf{x} + \mathbf{c}_a\Delta t, t + \Delta t) = f_a(\mathbf{x}, t) - 1/\tau \left(f_a(\mathbf{x}, t) - f_a^{eq}(\mathbf{x}, t)\right) \quad (1)$$

where $f_a(\mathbf{x}, t)$ denotes the fraction of particles at time step $t$ moving with velocity $\mathbf{c}_a$, and $\tau$ the relaxation time which is related to the fluid viscosity. $f^{eq}$ is the local equilibrium distribution function, constructed from the macroscopic variables, the form of which is chosen to impose conservation of mass and momentum, and impose isotropy. The velocities $\mathbf{c}_a$ arise from the basic structure of the lattice and are chosen in concert with the form of the collision operator and equilibrium distribution function. A typical discretization in 3D is the D3Q27 model [15] which uses 27 distinct velocities (including zero velocity) is shown in Figure 1.

Conceptually, a LBM simulation proceeds by a sequence of *collision()* and *stream()* steps, reflecting the structure of the master equation. The *collision()* step involves data local only to that spatial point; the mesoscopic variables at each point are calculated from the

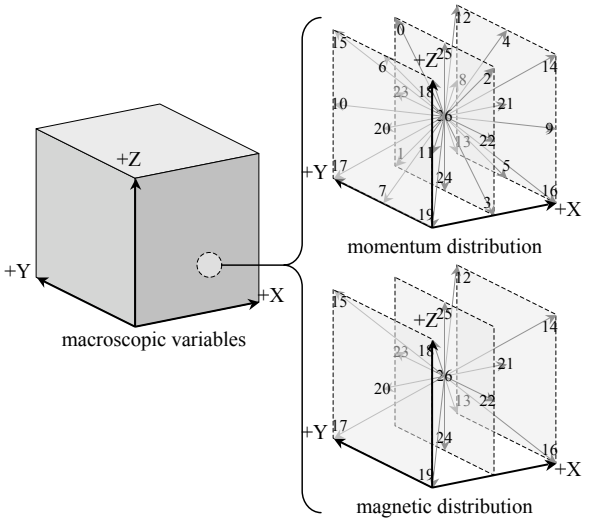



**Figure 1. Data structures for LBMHD. For each point in space, in addition to the macroscopic quantities of density, momentum, and magnetic field, two lattice distributions are maintained.**

distribution functions and from them the equilibrium distribution formed through a complex algebraic expression originally derived from appropriate conservation laws. Finally the distribution functions are updated according to Equation 1. This is followed by the *stream()* step that evolves the distribution functions along the appropriate lattice vectors. In practice, most implementations follow Wellein and co-workers [12] who incorporated the data movement of the *stream()* step directly into the *collision()* step. In this formulation, either the newly calculated particle distribution function can be scattered to the correct neighbor as soon as it is calculated, or equivalently, data can be gathered from adjacent cells to calculate the updated value for the current cell. Using this method, data movement is reduced and the collision step looks much more like a stencil kernel—in that data are accessed from multiple nearby cells. In a parallel distributed memory version of LB, a stream step is still required to refresh the ghost cells or enforce boundary conditions on the outer lattice faces.

LBMHD [6] was developed to study homogeneous isotropic turbulence in magnetohydrodynamics (MHD), the interaction of an electrically conducting fluid with a magnetic field, for a simple system of periodic boundary conditions. MHD turbulence plays an important role in many branches of physics [2]: from astrophysical phenomena to plasma instabilities in magnetic fusion devices. The kernel of LBMHD is similar to that of the fluid flow LBM except that the regular distribution

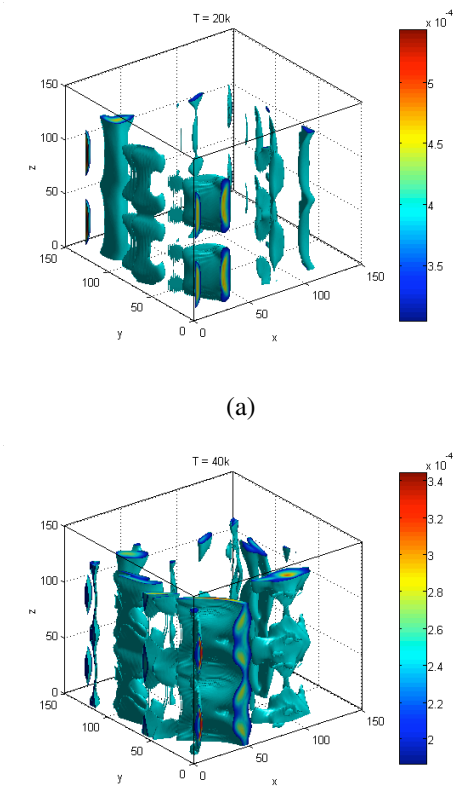
functions are augmented by magnetic field distribution functions, and the macroscopic quantities augmented by the magnetic field. In the case of these functions, the number of phase space velocities needed to recover information on the magnetic field is reduced from 27 to 15. The differing components for particle and magnetic field are shown in Figure 1. Figure 2 is reproduced from one of the largest 3-dimensional LBMHD simulations conducted to date [3], aiming to understand better the turbulent decay mechanisms starting from a Taylor-Green vortex — a problem of relevance to astrophysical dynamos. Here we show the development of turbulent structures in the z-direction as the initially linear vorticity tubes deform.

The original Fortran implementation of the code was parallelized using MPI, partitioning the whole lattice onto a 3-dimensional processor grid. This achieved high sustained performance on vector architectures, but a relatively low percentage of peak performance on superscalar platforms [8]. The application was rewritten, for our previous study [14], around two lattice data structures, representing the state of the system, the various distribution functions and macroscopic quantities, at time $t$ and at time $t + 1$. At each time step one lattice is updated from the values contained in the other. The algorithm alternates between these each data structures as time is advanced. The lattice data structure is a collection of arrays of pointers to double precision arrays that contain a grid of values.

In the original MPI version of the LBMHD code, the *stream()* function updates the ghost-zones surrounding the lattice domain held by each task. Rather than explicitly exchanging ghost-zone data with the 26 nearest neighboring subdomains, we use the shift algorithm, which performs the exchange in three steps involving only six neighbors. The shift algorithm makes use of the fact that after the first exchange between processors along one cartesian coordinate, the ghost cells along the border of the other two directions can be partially populated. The exchange in the next coordinate direction includes this data, further populating the ghost cells, and so on. Palmer and Nieplocha provide a recent summary [9] of the shift algorithm and compare the tradeoffs with explicitly exchanging data with all neighbors using different parallel programming models.

## 2.2 Performance Characteristics

LBMHD's *collision()* operator performs about 1300 floating-point operations per lattice update. In doing so, it must read 73 values (momentum velocities, magnetic field velocities, and macroscopic density) and write 79 values (new momentum velocities, magnetic field velocities, as well as the new macroscopic density, momentum, and magnetic field). On a write-allocate architecture, the writes generate twice the memory traffic as reads. As such, LBMHD's unoptimized arithmetic intensity (FLOP:DRAM byte ratio) is about 0.70. For an architecture like the Opteron, with a FLOP:DRAM byte ratio of about 3, it is clear LBMHD will be heavily memory bound. However, initial experiments showed this was not the bottleneck. Although the structure-of-arrays format made LBMHD cache and vector friendly, it is a very TLB unfriendly code — touching 150 arrays on an architecture with a 16 entry L1 TLB will generate a great number of TLB capacity misses.




(a)

(b)

**Figure 2. Vorticity tubes deforming near the onset of turbulence in LBMHD simulation (a) after 20K iterations, and (b) after 40K iterations.**

## 2.3 Previous Auto-tuning Efforts

Auto-tuning, or automated tuning, is premised on the belief that if one could enumerate the potentially useful optimizations, and devise an automated method of generating and benchmarking them, one could find the

best performing implementation for all architectures [5, 11, 13]. We previously implemented an LBMHD auto-tuner focused exclusively on single-node performance that delivered excellent speedups across a wide range of dual-socket multicore SMPs [14]. Although the XT5 would be well represented by the Opteron 2356 in that study, the XT4 used in this study is a somewhat different architecture. Most notably, there are only four cores, and they share a uniform memory access interface to main memory. In our previous study, we implemented an auto-tuner that in addition to applying small optimizations to a C reference implementation, also produced both vectorized and vectorized using SIMD intrinsics variants. We are very judicious in differentiating the terms vectorization and SIMDization. Vectorization is a code transformation that creates a vector (array) of temporaries in memory, and performs one (or a few) operation (not one statement) at a time per element. The subtle benefit is one is essentially blocking for the TLB as one limits the number of terms in the original statement evaluated at a time. SIMDization is a code generation technique that maps data-parallel operations to SIMD instructions through the use of software intrinsics. The benefit of vectorization (TLB blocking) is that it allows one to trade larger cache working sets for improved TLB locality. The auto-tuner would then search for the optimal "vector length" — essentially the working set it would attempt to keep in cache, and the number of consecutive elements it would access per TLB page. All experiments were performed on cubical domains with varying degrees of thread-level concurrency. As only one process was run, it should come as no surprise that maximizing the number of threads per process was ideal. Although vectorization provided a substantial performance boost, the auto-tuner's use of SSE, notably cache bypass instructions like *movntpd* which cuts memory traffic by 33% through the elimination of write allocation cache line fills provided an even greater increase. On this memory-bound kernel, by using this optimization, the auto-tuner improved arithmetic intensity, and thus performance, by 50%.

The rest of this paper examines the applicability of that auto-tuner to the distributed implementation of LBMHD and postulates that single-thread auto-tuning is insufficient and auto-tuning should be extended to both the domain decomposition and the balance of threading and multiple processes per compute node.

# 3 Experimental Setup

In this section we discuss the hardware, programming model, compilers, and experimental methodology used throughout this work.

## 3.1 XT4 (Franklin)

In this work, we use the Cray XT4 (named Franklin) at the National Energy Scientific Computing (NERSC) Center. Each compute node comprises one 2.3GHz quad-core Opteron (Budapest) processor, a SeaStar2 router, and 8GB of DDR2-800 memory. This provides the node with a peak performance, DRAM bandwidth, and per-link network bandwidth of 36.8GFlop/s, 12.8GB/s, and 6.4GB/s respectively. The XT4 is composed of 9,532 distributed memory compute nodes arranged in a 3D torus.

Each Opteron processor includes four cores, with private 64K L1 and 512K L2 caches, and a shared 2MB L3 cache. Each core is a superscalar out-of-order processor capable of decoding three x86 instructions and executing 6 micro-ops per cycle. The Opteron's high per-core peak floating-point performance arises from three major factors: 4-cycle pipelining, 2-way double-precision SIMD, and parallel adder and multiplier functional units. Codes lacking balance between multiplies and adds, like LBMHD, will make poor use of the multiplier datapath.

As LBMHD's optimized FLOP:DRAM Byte ratio is only about 1.0, peak performance is ultimately limited by the 9.7GBs/s sustainable bandwidth to about 9.5 GFlop/s per socket or 2.4 GFlop/s per core — ignoring time spent in *stream()*. Thus, if one already achieves peak bandwidth, the only potential optimization is to minimize memory traffic. This includes avoiding cache capacity and conflict misses as well as eliminating write allocations (cache bypass).

## 3.2 Hybrid MPI

In a flat MPI implementation of an application, one MPI process is assigned to each core of the multicore SMP nodes of a massively-parallel computer. Although one can trivially port applications from one generation of multicore chips to the next, achieving good performance can be elusive. To that end, one can embrace an alternate approach — a hybrid implementation. By hybrid, we mean two different programming models are included in the application. There are many advantages of using a hybrid implementation on computers built from multicore SMPs. For example, one can significantly reduce the number of system calls (from one per core to one per node). Second, one can eliminate superfluous memory copies that can be handled through the existing cache-coherency mechanisms. Third, many applications use identical read-only structures on each process. Using a hybrid implementation requires fewer copies per node (a dramatic reduction in memory capacity requirements), and can improve cacheability as all threads re-

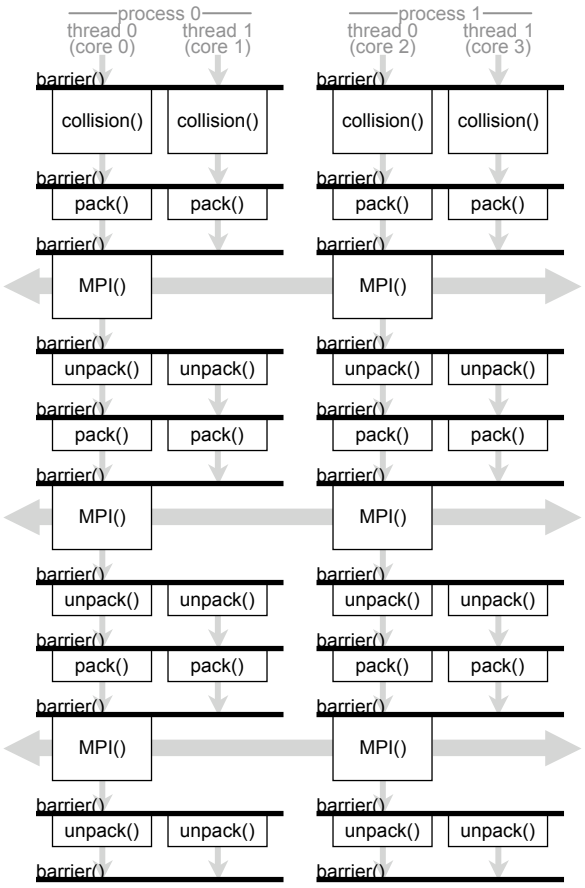



**Figure 3. Key loop in a hybrid LBMHD implementation running with two processes each of two threads. Boxes represent function calls, arrows the flow of data, and barrier() is a intra-process (thread) barrier.**

fer to the same memory locations. Finally, some applications scale poorly with an increasing number of MPI processes — an unfortunate match with the current landscape of stalled cpu clock frequencies and an exponentially increasing number of cores over time. Fixing the number of MPI processes, and scaling the number of cores can potentially eliminate impediments to poor scalability.

For global parallelism and communication, we require a message passing or PGAS language; we elected to evolve the existing MPI implementation of LBMHD. For local communication and parallelism, we use a shared memory, single-program, multiple-data (SPMD) model based on POSIX threads (pthreads) as the cache-coherent capability of the Opteron quad-core processors obviates the needs for message passing or PGAS languages, and we perceive pthreads as providing increased (closer to metal) control of hardware resources.

The modifications of the existing flat-MPI implementation of LBMHD to a hybrid MPI-Pthreads implementations were minor. First, as thread creation can be expensive, we avoid the fork-join model by creating threads at the beginning of the application and then using barriers and predication to fan out work — essentially a threaded version of SPMD. The key time loop in LBMHD iterates on two functions: *collision()*, which performs local computation, and *stream()*, which, in a three-phase communication scheme [9], packs the surfaces of the local grid, performs MPI communication, and unpacks to the grid for communication in the $Z$, $Y$, and $X$ dimensions. Initial experimentation showed the need to parallelize *stream()* over threads to avoid the problems of Amdahl's Law. In addition, we decided to split stream into six functions (pack/unpack $\times$ 3 dimensions) and 3 blocks of MPI calls. The XT4 supports a high performance MPI library which can provide, in MPI terminology, `MPI_THREAD_FUNNELED` or `MPI_THREAD_SERIALIZED` support. in these two models, only the main thread will make MPI calls, or any threads may make an MPI calls but the application writer must guarantee that only one thread does so at a time are the respective requirements of application writers. The capability to have any thread make an MPI call at any time is currently supported via a lower performing library. Because of this, in our implementation only thread 0 within each process performs MPI communication. When coupled with parallelized *collision()* and *stream()*, it is clear the hybrid implementation requires many barriers to ensure correct behavior — locally bulk synchronous.

Consider the program flow shown in Figure 3. We see the execution of one time step using a hybrid implementation running with two MPI processes each of two threads. Clearly, we see four major phases: *collision()*, MPI communication in $Z$, MPI communication in $Y$, and MPI communication in $X$. Communication of values produced by one thread's execution of *collision()* can be passed to the other thread in the same process through the cache-coherency mechanisms or via thread 0's MPI call to communicate with the threads of the other process. Depending on problem dimensions, processor architecture, network performance, and local optimization, the time spent in each phase can vary dramatically.

### 3.3 Compilers and Tools

In this work, we only used the gcc compiler as it faired better with the intrinsic-laden SSE code. We attempted to use the affinity routines to pin threads, but the performance benefit was negligible. Perhaps in the fu-

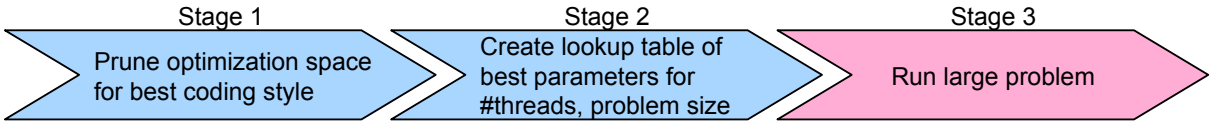



**Figure 4. Stages in the greedy hybrid distributed auto-tuning approach.**

ture, on NUMA architectures like the XT5, affinity will become critical.

### 3.4 Problem Size

In all our experiments, we mandate a per-core memory foot print of 256K ($64^3$) grid points. This is about 300MB per core. In addition, although we may change the balance between processes per node and threads per process, we always use four cores per node. Thus, depending on the hybrid implementation, the MPI process memory footprint ranges from 300MB–1.2GB, but the minimum per node memory footprint is always about 1.2GB. For the large-scale runs, we only explored concurrencies with 128 nodes (512 cores). In the future, we will experiment with both larger and smaller per-core problem sizes as well as different numbers of cores.

### 3.5 Calculating Performance

LBMHD performs about 1300 floating-point operations, including one divide, per lattice update. For the distributed runs in this work, we calculate average performance, counting the floating-point divide as one FLOP, by taking the total number of floating-point operations and dividing by the overall execution time as measured on process zero. However, local-only tuning, as discussed below, only measured the best (instead of average) performance for any of the 10 iterations.

## 4 Hybrid Auto-Tuning

In this section, we discuss our time- and resource-efficient approach to auto-tuning distributed applications. To that end, we employ a greedy 3-stage auto-tuning algorithm that attempts to discover the optimal solutions to the local-optimization problem before integrating with a distributed tuner or ultimately, a large-scale run.

### 4.1 The Case Against Scaling Conventional Auto-tuning up to Distributed Applications

Our previous LBMHD auto-tuning efforts focused exclusively on single-node, threaded performance [14]. To that end, we had one problem size and always used as many threads as there was hardware support for. Despite our efforts, the combinatoric optimization-parameter space still was significant. First, in terms of data-structure, even if we restrict ourselves to only the structure-of-arrays form, we may still explore array padding. For threading, we overlay a 2D *thread grid* on top of the domain and only explore threading in powers of two in the $Y$ and $Z$ dimensions. This concept requires the search of an additional dimension of size $1 + log(threads)$, where "threads" is the number of hardware thread contexts per node. Note, the thread grid concept does not change data structures, only loop bounds. When it comes to code generation, our auto-tuner produced three basic flavors: reference, vectorized, and explicitly (via intrinsics) SIMDized vectorized implementations. For the vectorized variants, we must attempt to discover the vector length (VL) that delivers optimal performance. VL is essentially a parameter that allows us to trade decreased TLB pressure for increased cache pressure — clearly each architecture will have a unique balance between these. To prune the exploration of vector length, we only examined vector lengths in multiples of a cache line up to 128 elements, then a telescoping power-of-two exploration to 1K (19 possibilities). In addition, we explored only power of two unrollings and reorderings by up to a cache line of both the reference and vectorized variants (another dimension of 10 values). Finally, we explored only three prefetch distances for the vectorized forms. All-in-all, after pruning, this results in about 5500 different implementations that we must search through to find the best. As each trial required 4-10 seconds, we need more than 6 CPU hours to explore the entire space.

When one moves to a distributed implementation of LBMHD, several new optimization dimensions appear. First, in a hybrid implementation, one must determine the appropriate balance between threads and processes. Luckily, this only increases the search space by roughly

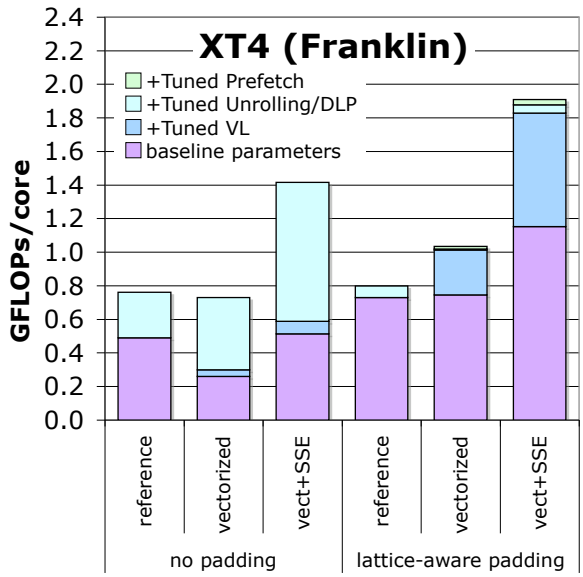



**Figure 5. Performance (in isolation) as a function of coding style. Note, we employ a 4×1 thread grid on $128^3$ domain.**

$log(threads)/2$. Even when coupled with the mapping of threads within a process, in the context of applying Moore's law to multicore, collectively these terms ensure the search space does not suffer a combinatoric explosion. The more challenging component is the problem decomposition among processes. Alternately, one can recast this as the search for the optimal per-process aspect ratio (assuming fixed volume). Combined, these dimensions increase the search space by perhaps a factor of 100×. Obviously naively implementing a distributed auto-tuner running across N nodes to search for the optimal implementation for a N-process application is not a cost effective approach as one requires perhaps $1000 \times N$ CPU hours for tuning alone.

## 4.2 Efficient Distributed Application Auto-tuning

We propose a three-stage approach to tuning distributed, weakly-scaled applications. The aggregate overhead for auto-tuning is independent of the number of processes, but will scale with both the thread-level parallelism afforded by multicore processors as well as with per-node memory capacity. The latter is true because large memory capacities produce many more per-process aspect ratios that we must benchmark. Figure 4 shows the three stages in our greedy distributed auto-tuner.

### 4.2.1 Stage 1: Local Tuning

Our first stage attempts to prune the code optimization space to the variants that consistently deliver superior performance. To that end, we only explore a $128^3$ problem size and one thread-level concurrency: a $4 \times 1$ thread grid (4 threads in the $Z$ dimension).

Figure 5 presents the performance results (as measured in GFLOPs/core) for the three basic coding styles both with and without lattice-aware padding (an algorithm not a search) as different optimization parameters are tuned for. Note, this exploration is exhaustive, not greedy. That is, we exhaustively searched all combinations, but for clarity, we present the data to show the benefits for increasingly complex auto-tuners. That is, one could implement a subset of the capability of our auto-tuner, but will suffer a commensurate loss in performance. Clearly, the combination of lattice-aware padding, vectorization, and SSE delivers superior performance. The SSE benefit came from two components: the use of cache-bypass instructions like *movntpd*, and code quality superior to that produced by gcc. As it turned out, prefetching by a cache line was also best. One shouldn't be disheartened by the low fraction of peak flop/s (25%) as LBMHD is memory-bound. As such, the optimized implementation achieves an average, useful bandwidth in excess of 8GB/s on a machine with a sustained DRAM bandwidth of 9.7GB/s. In successive tuning stages, we only run the vectorized SSE code variant with lattice-aware padding and 64 byte prefetching, but will explore other optimization dimensions and will research the vector length, unrolling and reordering.

### 4.2.2 Stage 2: Threads/Aspect Ratio

In the second stage, we create a database of optimal vector lengths, unrollings, and reorderings indexed by the number of MPI processes per node, the thread grid topology (Figure 6), and the per-process aspect ratio (Figure 7). In this stage we mandate 256K grid points per core, and use the results from stage 1 as a starting point. We also prune the search space by enforcing a minimum dimension of 32 and maximum dimension of 512. Nevertheless, more than 100 configurations are possible.

Recall the *stream()* operation packs and unpacks the six surfaces of the domain. Although this is extremely efficient for some surfaces (XY) as all accesses are unit stride, it can be extremely inefficient for others (YZ) because the memory access pattern could be stride-128. Tuning across different aspect ratios captures this potential performance variation, but does not include the effect of the change in surface area, and hence message size, on MPI time.

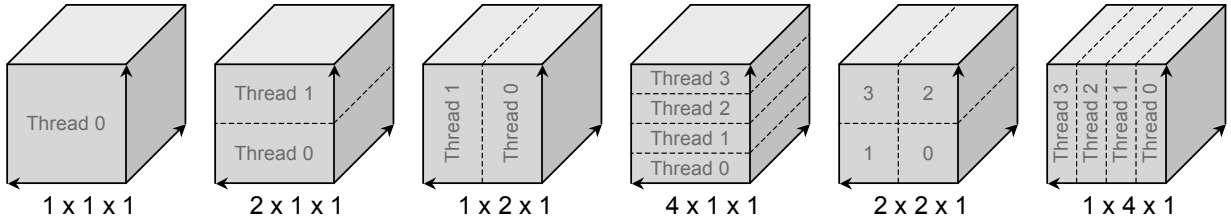



**Figure 6. Thread Grid (Threads in $Z \times Y \times X$) applied to each MPI process. Note, we do not explore threading within a process in the $X$ dimension. Also, the thread grid concept does not change data structure, only loop bounds.**

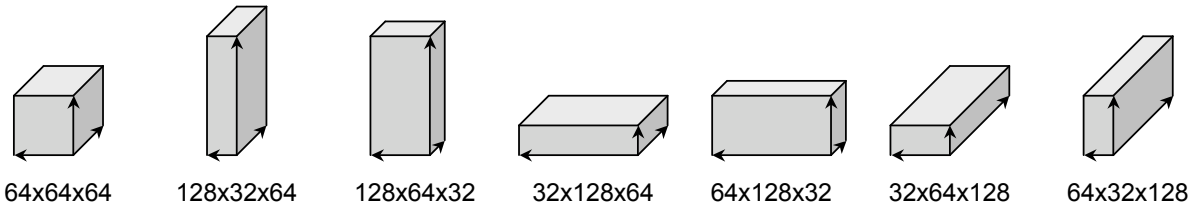



**Figure 7. Exploration of alternate per-process aspect ratios. Note: only shown are a subset of the 1 thread/process cases. We explore subdomains as large as 512 in any one dimension. The process grid concept is the cartesian structure that decomposes a $512^3$ domain into one of these subdomains.**

As LBMHD is a data-parallel SPMD application, any hybrid tuning must be performed cognizant that the threads of all MPI processes on the same node will be executing the same code at roughly the same program counter. Thus, tuning for any subset of the production number of threads per node is totally inappropriate as this would favor memory-inefficient code variants that will under perform when the node is fully utilized. To that end, we evolved the existing single-node threaded auto-tuner into a distributed hybrid auto-tuner. The critical addition was an *MPI_barrier()* before the timing trials for each optimization configuration. This ensures all threads across all processes remain synchronized as they traverse the optimization space.

Figure 8 shows the performance variation across different aspect ratios, and thread grid topologies for the three possible process×thread combinations on the XT4. Observe, there is nearly a 30% variation in performance as aspect ratio changes. Typically, aspect ratios that favor large $X$ (unit stride) dimensions (small $YZ$ planes) deliver the best performance. In the absence of MPI send's and recv's (using the sum of *collision()* and *stream()* time) the 4 threads per process hybrid implementation consistently delivers better performance than the versions with 1 or 2 threads per processes.

#### 4.2.3 Stage 3: Run Full Problem

The final stage of the auto-tuner performs some moderately-sized distributed runs. One should be mindful that in our hybrid implementation, only thread 0 performs MPI communication. As such, as we increase the per-process thread-level parallelism, we reduce the volume of communication by exchanging costly messages for free cache-coherency, but simultaneously reduce the potential MPI bandwidth. Thus, we take the best performing aspect ratios from stage two for the three different thread-process balances based not only on best overall performance, but also based on the aspect ratios that yielded the fastest *collision()* or *stream()* times. We can then run these 9 configurations at moderate concurrencies using the best known parameters. Note, given the aspect ratio, and overall problem size and MPI concurrency, the MPI cartesian decomposition is uniquely specified.

Table 1 presents a subset of configurations used for this moderate-concurrency benchmarks. Recall the problem size and core-concurrency is fixed at $512^3$ and

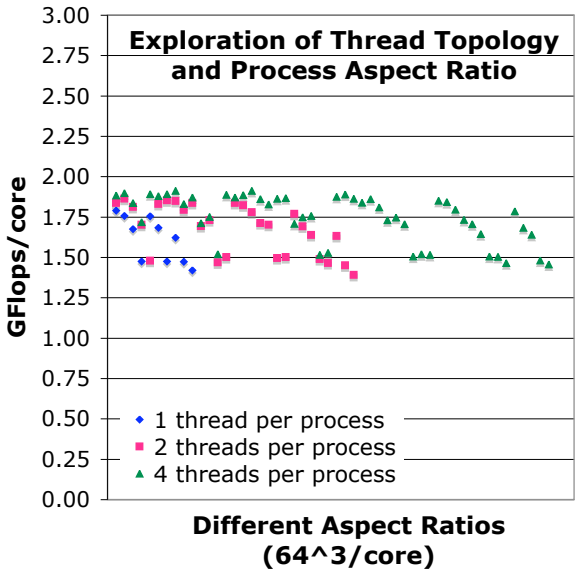



**Figure 8. Performance across the variety of thread topologies, and process aspect ratios.**

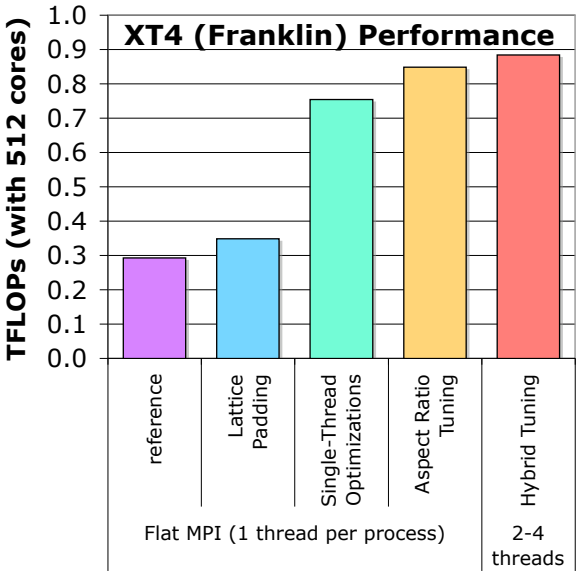



**Figure 9. Performance on the XT4 (Franklin) with incrementally more complex auto-tuning. All cases use 512 cores to simulate a $512^3$ problem.**

512 respectively. Thus, a small number of processes ($p$) in a dimension of the process grid denotes a large dimension in the processes' domain ($512/p$).

## 5 Results

Figure 9 shows the performance of the progressively auto-tuned and optimized implementations of LBMHD running on the 512 cores of the XT4. The reference implementation uses the reference code style without explicit unrolling or reordering using a 1×1 thread grid and a 8×8×8 processor grid ($64^3$ per process). We observe that even lattice-aware array padding can improve application performance by about 15% — an impressive yet disturbing outcome for such a conceptually-simple optimization. Single-thread optimizations originally presented in [14] can improve the flat MPI application performance by more than a factor of two. The final flat MPI bar shows the best result of using the aspect ratio that promised either the fastest local overall time or the fastest local *stream()* time. The former delivered 1% better performance than the latter. Overall, tuning the aspect ratio for even this relatively large per-core problem size yielded 12% better performance. The final bar marks the performance gain one can achieve by also tuning the number of threads per process. For each of these two possibilities (2 or 4) we benched the three locally optimal configurations to find the best global implementation. We observe two interesting results. First, the configuration yielding the fastest local overall and *stream()* performance were identical and yielded the best overall performance. Second, using two processes of two threads produced about 2% better overall performance when compared with one process of four threads. This is indicative that one may need some thread-level parallelism across MPI calls. Overall, the hybrid implementation improved performance by 4% over the tuned flat MPI implementation, and 3× faster than the original reference implementation. In the future, as we explore smaller per-core problem sizes, we may see a substantially different balance between threads and processes.

| | Processes ×Threads | Process Grid | Thread Grid | Tuning: VL | unroll | DLP |
|---|---|---|---|---|---|---|
| Reference | 512×1 | 8×8×8 | 1×1 | N/A | 1 | 1 |
| Single-thread optimization | 512×1 | 8×8×8 | 1×1 | 512 | 2 | 2 |
| Aspect ratio tuning | 512×1 | 16×16×2 | 1×1 | 256 | 8 | 4 |
| Hybrid tuning | 256×2 | 16×8×2 | 2×1 | 256 | 4 | 4 |
| | 128×4 | 8×8×2 | 4×1 | 256 | 4 | 4 |

**Table 1. Example of best parameters for a variety of configurations for best overall (sans MPI) performance.**

Figure 10 breaks down the execution time for the

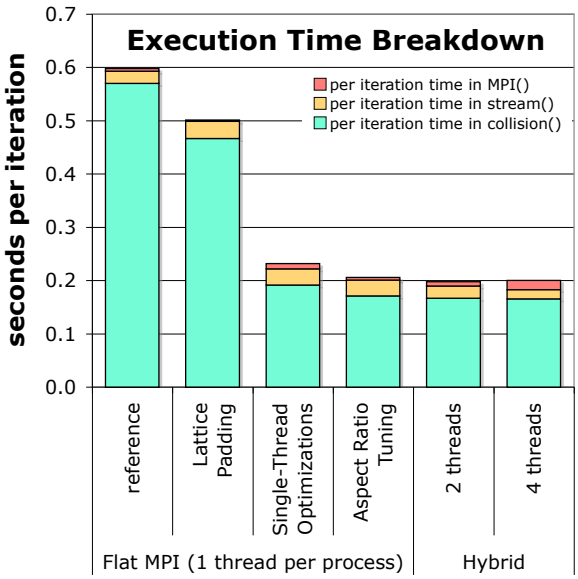



**Figure 10. Breakdown of execution time on Franklin with incrementally more complex auto-tuning always using 512 cores to simulate a $512^3$ problem.**

fastest iteration as a function of optimization. Note, although our timing methodology easily calculated the total time in *stream()* and MPI, it prevented an accurate and consistent means of determining the exact split between time in *stream()* and time in MPI. We observe that single-thread optimizations dramatically improved *collision()* time, and selection of the appropriate domain decomposition slightly further improved *collision()* time. Yet, time in *stream()* changed little. As one would expect, for a fixed volume, the minimal surface area is achieved with a cubical domain. However, the bandwidth to access certain surfaces may be suboptimal. As we change the aspect ratio, we invariably increase the surface area. This can be marginally offset by improved bandwidth. As we move to a hybrid implementation with increasing thread concurrency, we see a progressively reduced time in *stream()* — 0.30, 0.23, and 0.17 seconds, but see progressively more time in MPI — 0.005, 0.008, 0.017 seconds. As such, as there is little performance variation in the communication-less *collision()* function as a function of threads, we see an best performance when *stream()* + MPI time is minimized.

## 6 Conclusions

In this work we examined the propositions that flat MPI implementations deliver superior performance, and that only exploiting local auto-tuning is sufficient to achieve globally-optimal performance. The data shows that neither of these assumptions hold. Although single-thread optimizations, albeit cognizant of concurrent MPI processes on the same multicore chip, do provide the bulk of the potential performance advantages for moderately large problems, an additional 17% performance enhancement can be had through tuning both the MPI process domain decomposition and the balance between threads and processes on multicore architectures. Moreover, we achieve these performance gains using only a few CPU hours of auto-tuning time. This overhead is negligible when one considers that LBMHD may be run on thousands of processors for tens of thousands of time steps.

Our distributed auto-tuning methodology deftly ensured that threads and processes would remain synchronized as they traversed the optimization space — at each point replicating the SPMD behavior one would expect in a tuned parallel application. We assumed that the time required for MPI would be small, and for this problem size that was true. However, in retrospect, we believe that an additional stage should be inserted between the existing stages 2 and 3. This stage would benchmark the MPI time for communicating a variety of buffer sizes for differing numbers of "active" threads per node across a few different concurrency classes (e.g. 128 nodes, 1K nodes, etc...) The MPI communication time garnered in this stage could be added with the overall local time (*collision()*+*stream()*) to determine a better configuration for large runs.

In the future, we do not expect the memory capacity per SMP to grow as quickly as the number of cores in its multicore chip. As such, the relative memory capacity per core will decrease. To that end, future work will inevitably study the performance impacts of smaller per-core problem sizes and means to alleviate them including hybrid implementations, alternate data structures, and multiple time steps per communication phase.

## Acknowledgments

We would like to thank George Vahala and his research group for the original version of the LBMHD code. All authors from LBNL were supported by the ASCR Office in the DOE Office of Science under contract number DE-AC02-05CH11231.

## References

[1] P. Bhatnagar, E. Gross, and M. Krook. A model for collisional processes in gases I: small amplitude processes in charged and neutral one-component systems. *Phys. Rev.*, 94:511, 1954.

[2] D. Biskamp. *Magnetohydrodynamic Turbulence*. Cambridge University Press, 2003.
[3] J. Carter, M. Soe, L. Oliker, Y. Tsuda, G. Vahala, L. Vahala, and A. Macnab. Magnetohydrodynamic turbulence simulations on the Earth Simulator using the lattice Boltzmann method. In *Proc. SC2005: High performance computing, networking, and storage conference*, 2005.
[4] P. Dellar. Lattice kinetic schemes for magnetohydrodynamics. *J. Comput. Phys.*, 79, 2002.
[5] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.
[6] A. Macnab, G. Vahala, L. Vahala, and P. Pavlo. Lattice Boltzmann model for dissipative MHD. In *Proc. 29th EPS Conference on Controlled Fusion and Plasma Physics*, volume 26B, Montreux, Switzerland, June 17-21, 2002.
[7] D. Martinez, S. Chen, and W. Matthaeus. Lattice Boltzmann magnetohydrodynamics. *Phys. Plasmas*, 1, 1994.
[8] L. Oliker, J. Carter, M. Wehner, A. Canning, S. Ethier, et al. Leading computational methods on scalar and vector HEC platforms. In *Proc. SC2005: High performance computing, networking, and storage conference*, page 62, Seattle, WA, 2005.
[9] B. Palmer and J. Nieplocha. Efficient algorithms for ghost cell updates on two classes of MPP architectures. In *Proc. PDCS International Conference on Parallel and Distributed Computing Systems*, pages 192–197, 2002.
[10] S. Succi. The Lattice Boltzmann equation for fluids and beyond. *Oxford Science Publ.*, 2001.
[11] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. of SciDAC 2005, J. of Physics: Conference Series*, page 521530. Institute of Physics Publishing, June 2005.
[12] G. Wellein, T. Zeiser, S. Donath, and G. Hager. On the single processor performance of simple lattice Boltzmann kernels. *Computers and Fluids*, 35(910), 2005.
[13] R. C. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimization of Software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.
[14] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Lattice boltzmann simulation optimization on leading multicore platforms. In *International Parallel & Distributed Processing Symposium*, 2008.
[15] D. Yu, R. Mei, W. Shyy, and L. Luo. Lattice Boltzmann method for 3D flows with curved boundary. *Journal of Comp. Physics*, 161:680–699, 2000.

## About the Authors

**Samuel Williams** is a Computer Scientist in the Future Technologies Group at the Lawrence Berkeley National Laboratory. He received both his Ph.D. and master's in Computer Science from the University of California at Berkeley while performing research at LBL as well as in Berkeley's Parallel Computing Laboratory. He received bachelor's degrees in Electrical Engineering, Applied Mathematics, and Physics from Southern Methodist University. His current research interests include performance and energy analysis and optimization of scientific kernels running on multicore architectures. He previously conducted research into the synthesis of embedded DRAM with vector architectures for multimedia processing. email: *SWWilliams@lbl.gov*

**Jonathan Carter** heads the User Services Group at the National Energy Research Scientific Computing Center (NERSC). He received his Ph.D. in Chemistry from the University of Sheffield in the United Kingdom, and performed postdoctoral work at the University of British Columbia and at IBM Almaden Research Center. His research interests include HPC evaluation, programming models for scientific computing on multicore architectures and large-scale distributed memory systems. email: *JTCarter@lbl.gov*

**Lenny Oliker** is a Computer Scientist in the Future Technologies Group at Lawrence Berkeley National Laboratory. He received bachelor degrees in Computer Engineering and Finance from the University of Pennsylvania, and performed both his doctoral and postdoctoral work at NASA Ames research center. Lenny has co-authored over 60 technical articles, four of which received best paper awards. His research interests include HPC evaluation, multi-core auto-tuning, and power-efficient computing. email: *LOliker@lbl.gov*

**John Shalf** leads the computer architecture group at the National Energy Research Scientific Computing Center (NERSC). His interests are in computer architecture, performance evaluation of emerging systems, programming models, and power efficient computing technology. email: *JShalf@lbl.gov*

**Katherine Yelick** is a Professor of Electrical Engineering and Computer Science at the University of California at Berkeley and Director of the National Energy Research Scientific Computing (NERSC) Center at Lawrence Berkeley National Laboratory. She received her Bachelors, Masters, and Ph.D. degrees from the Massachusetts Institute of Technology. She has led or co-led the Berkeley UPC, Titanium, and Bebop projects and her current research interests include parallel computing, memory hierarchy optimizations, programming languages, and compilers. email: *KAYelick@lbl.gov*