Extracting Ultra-Scale Lattice Boltzmann Performance via Hierarchical and Distributed Auto-Tuning

Samuel Williams, Leonid Oliker
Jonathan Carter, John Shalf

Lawrence Berkeley National Laboratory

SWWilliams@lbl.gov

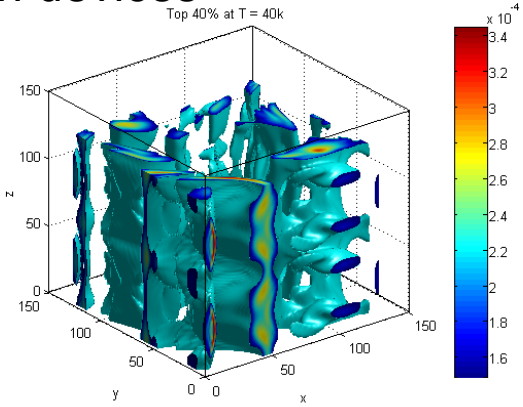
Introduction

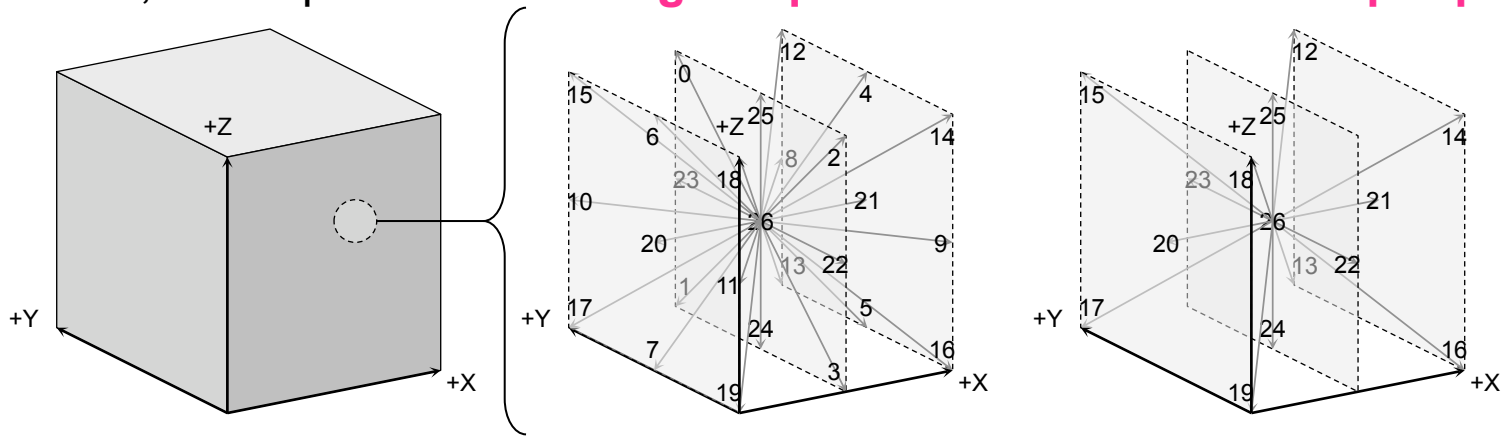
- Management of **data locality and data movement** is essential in attaining high performance.
- However, the subtleties of multicore architectures, deep memory hierarchies, networks, and the randomness of batch job scheduling makes attaining high-performance on large-scale distributed memory applications is increasingly challenging.
- In this talk, we examine techniques to mitigate these challenges and attain high performance on a plasma physics simulation.

LBMHD

LBMHD

- Lattice Boltzmann Magnetohydrodynamics (CFD+Maxwell's Equations)
- Plasma turbulence simulation via Lattice Boltzmann Method (~structured grid) for simulating astrophysical phenomena and fusion devices
- Three macroscopic quantities:
  - Density
  - Momentum (vector)
  - Magnetic Field (vector)
- LBM requires two velocity distributions:
  - momentum distribution (27 scalar components)
  - magnetic distribution (15 Cartesian vector components)
- Overall, that represents a **storage requirement of 158 doubles per point**.





macroscopic variables | momentum distribution | magnetic distribution

LBMHD

- Code Structure:
  - we base our code on the 2005 Gordon Bell nominated implementation.
  - time evolution through a series of *collision( )* and *stream( )* functions
  - *collision( )* evolves the state of the simulation
  - *stream( )* exchanges ghost zones among processes.
  - When parallelized, *stream( )* should constitute ~10% of the runtime.

- Notes on *collision( )* :
  - For each lattice update, we must read 73 doubles, perform about 1300 flops, and update 79 doubles (1216 bytes of data movement)
  - **Just over 1.0 flop per byte (ideal architecture)**
  - Suggests LBMHD is **memory-bound** on most architectures.
  - Structure-of-arrays layout (component's are separated) ensures that cache **capacity requirements are independent of problem size**
  - However, TLB capacity requirement increases to >150 entries (similarly, HW prefetchers must deal with 150 streams)

LBMHD Sequential Challenges

- stencil-like code is sensitive to bandwidth and data movement. (**maximize bandwidth, minimize superfluous data movement**)
- To attain good bandwidth on processors with HW prefetching, we must express a high-degree of **sequential locality**.

  i.e. access all addresses `f[i]…f[i+k]` *for a moderately large k.*
- Code demands maintaining a cache working set of **>80 doubles** per lattice update (*simpler stencils require a working set of 3 doubles*)
- Working set scales linearly with sequential locality.
- For every element of sequential access, we must maintain a working set of **>640 bytes !!!** (thus, to express one page of sequential locality, we must maintain a working set of over **2.5MB per thread**)
- **Clearly, we must trade sequential locality for temporal locality**

LBMHD Parallel Challenges

- Each process maintains a 1 element (79 doubles) deep ghost zone around its subdomain.
- On each time step, a subset of these values (24 doubles) must be communicated for each face of the subdomain.

- For small problems per process (e.g. 24x32x48), each process must communicate **at least 6% of its data**.
- Given network bandwidth *per core* is typically far less than DRAM bandwidth *per core*, this poor surface:volume ratio cam impede performance.
- Moreover, network bandwidth can vary depending on the locations of the communicating pairs.

- **We must explore approaches to parallelization and decomposition that mitigate these effects**

Supercomputing Platforms

Supercomputing Platforms

- We examine performance on three machines: the Cray XT4 (Franklin), the Cray XE6 (Hopper), and the IBM BGP (Intrepid)

	Intrepid	Franklin	Hopper
chips per node	1	1	4
cores per chip	4	4	6
Interconnect	Custom 3D Torus	SeaStar2 3D Torus	Gemini 3D Torus
Gflop/s per core	3.4	9.2	8.4
DRAM GB/s per core	2.07	2.1	2.05
Private cache per core	32KB	64+512KB	64+512KB
LLC cache per core	2MB	512KB	1MB
Node power per core[1]	7.7W	30W	19W

[1]Based on top500

Supercomputing Platforms

- We examine performance on three machines: the Cray XT4 (Franklin), the Cray XE6 (Hopper), and the IBM BGP (Intrepid)

	Intrepid	Franklin	Hopper
chips per node	1	1	4
cores per chip	4	4	6
Interconnect	Custom 3D Torus	SeaStar2 3D Torus	Gemini 3D Torus
Gflop/s per core	3.4	9.2	8.4
DRAM GB/s per core	2.07	2.1	2.05
Private cache per core	32KB	64+512KB	64+512KB
LLC cache per core	2MB	512KB	1MB
Node power per core[1]	7.7W	30W	19W

NUMA

[1]Based on top500

Supercomputing Platforms

- We examine performance on three machines: the Cray XT4 (Franklin), the Cray XE6 (Hopper), and the IBM BGP (Intrepid)

	Intrepid	Franklin	Hopper
chips per node	1	1	4
cores per chip	4	4	6
Interconnect	Custom 3D Torus	SeaStar2 3D Torus	Gemini 3D Torus
Gflop/s per core	3.4	9.2	8.4
DRAM GB/s per core	2.07	2.1	2.05
Private cache per core	32KB	64+512KB	64+512KB
LLC cache per core	2MB	512KB	1MB
Node power per core[1]	7.7W	30W	19W

Similar Bandwidth

[1]Based on top500

Supercomputing Platforms

- We examine performance on three machines: the Cray XT4 (Franklin), the Cray XE6 (Hopper), and the IBM BGP (Intrepid)

	Intrepid	Franklin	Hopper
chips per node	1	1	4
cores per chip	4	4	6
Interconnect	Custom 3D Torus	SeaStar2 3D Torus	Gemini 3D Torus
Gflop/s per core	3.4	9.2	8.4
DRAM GB/s per core	2.07	2.1	2.05
Private cache per core	32KB	64+512KB	64+512KB
LLC cache per core	2MB	512KB	1MB
Node power per core[1]	7.7W	30W	19W

High flop:byte ratio = bandwidth-bound (performance ≈ bandwidth)

[1]Based on top500

Supercomputing Platforms

- We examine performance on three machines: the Cray XT4 (Franklin), the Cray XE6 (Hopper), and the IBM BGP (Intrepid)

	Intrepid	Franklin	Hopper
chips per node	1	1	4
cores per chip	4	4	6
Interconnect	Custom 3D Torus	SeaStar2 3D Torus	Gemini 3D Torus
Gflop/s per core	3.4	9.2	8.4
DRAM GB/s per core	2.07	2.1	2.05
Private cache per core	32KB		
LLC cache per core	2MB	512KB	1MB
Node power per core[1]	7.7W	30W	19W

relatively low flop:byte ratio
= in-core optimization

[1]Based on top500

Supercomputing Platforms

- We examine performance on three machines: the Cray XT4 (Franklin), the Cray XE6 (Hopper), and the IBM BGP (Intrepid)

	Intrepid	Franklin	Hopper
chips per node	1	1	4
cores per chip	4	4	6
Interconnect	Custom 3D Torus	SeaStar2 3D Torus	Gemini 3D Torus
Gflop/s per core	3.4	9.2	8.4
DRAM GB/s per core	2.07	2.1	2.05
Private cache per core	32KB	64+512KB	64+512KB
LLC cache per c			1MB
Node power per core	7.7W	30W	19W

Large private caches

= easy to exploit sequential locality

[1]Based on top500

Supercomputing Platforms

- We examine performance on three machines: the Cray XT4 (Franklin), the Cray XE6 (Hopper), and the IBM BGP (Intrepid)

	Intrepid	Franklin	Hopper
chips per node	1	1	4
cores per chip	4	4	6
Interconnect	Custom 3D Torus	SeaStar2 3D Torus	Gemini 3D Torus
Gflop/s per core	3.4	9.2	8.4
DRAM GB/s per core	2.07	2.1	2.05
Private cache per core	32KB	64+512KB	64+512KB
LLC cache per core	2MB	2MB	5MB
Node power per core[1]	7.7W	30W	70W

[1]Based on top500

Can only express 0.4KB of sequential locality without doubling traffic to LLC

Supercomputing Platforms

- We examine performance on three machines: the Cray XT4 (Franklin), the Cray XE6 (Hopper), and the IBM BGP (Intrepid)

	Intrepid	Franklin	Hopper
chips per node	1	1	4
cores per chip	4	4	6
Interconnect	Custom 3D Torus	SeaStar2 3D Torus	Gemini 3D Torus
Gflop/s per core	3.4	9.2	8.4
DRAM GB/s per core	2.07	2.1	2.05
Private cache per core	32KB	64+512KB	64+512KB
LLC cache per core	2MB	2MB	5MB
Node power per core[1]	7.7W	30W	12W

Low power cores = good energy efficiency if we can get performance

[1]Based on top500

Experimental Methodology

Experimental Methodology

- **Memory Usage**: We will explore three different (per node) problem sizes: 96^3, 144^3, and 240^3. These correspond to using (at least) 1GB, 4GB, and 16GB of DRAM.

- **Programming Models**: We explore the flat MPI, MPI+OpenMP, and MPI+Pthreads programming models. OpenMP/Pthread comparisons are used to quantify the performance impact from increased productivity.

- **Scale**: We auto-tune at relatively small scale (64 nodes = 1536 cores on Hopper), then evaluate at scale (2K nodes = 49152 cores on Hopper)

- Note, in every configuration, we always use every core on a node. That is, as we increase the number of threads per process, we proportionally reduce the number of processes per node.

Performance Optimization

Sequential Optimizations

- **Virtual Vectors**: Treat the cache hierarchy as a virtual vector register file with a parameterized vector lenght. ***This parameter allows one to trade sequential locality for temporal locality.***

- **Unrolling/Reordering/SIMDization**: restructure the operations on virtual vectors to facilitate code generation. Use SIMD intrinsics to map operations on virtual vectors directly to SIMD instructions. ***Large parameter space for balancing RF locality and L1 BW***

- **SW Prefetching**: Memory access pattern is complicated by short stanzas. Use SW prefetching intrinsics to hide memory latency. ***Tuning prefetch distance trades temporal/sequential locality***

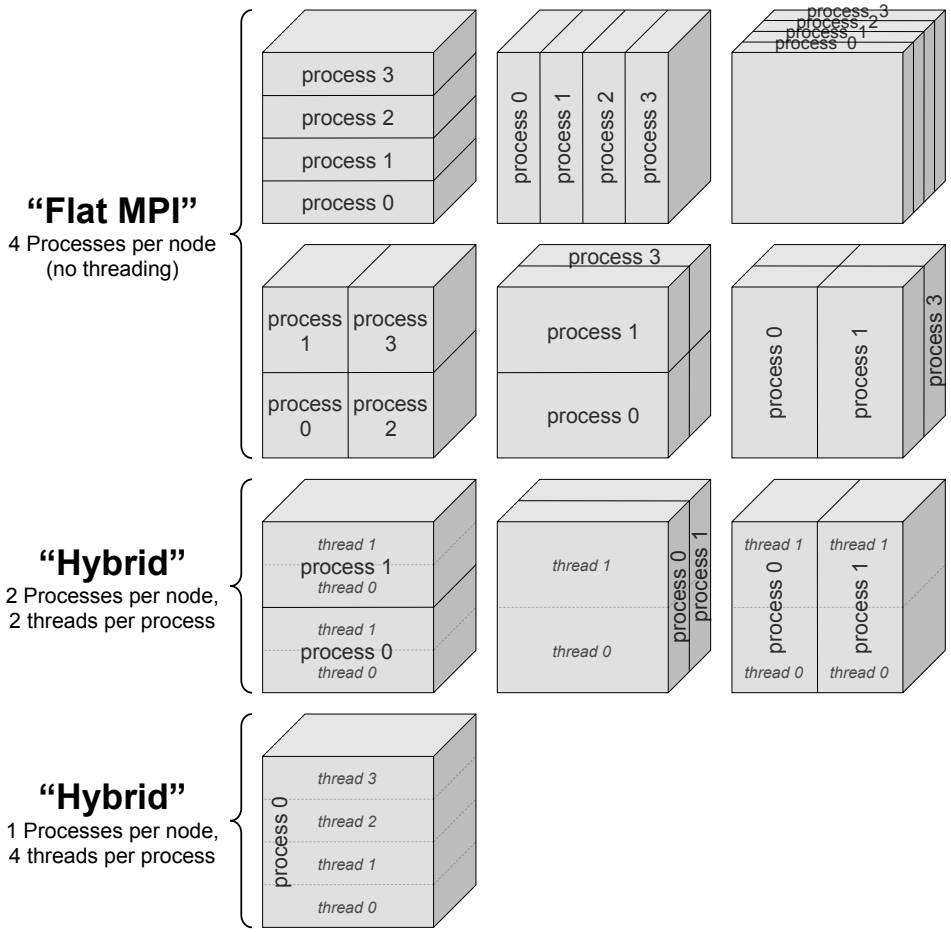
- **BGP-specific**: restructure code to cope with L1 quirks (half bandwidth, write-through) and insert XL/C-specific pragmas (aligned and disjoint)

Optimizations for Parallelism

- **Affinity**: we use `aprun` options coupled with first touch initialization to optimize for NUMA on Hopper. Similar technology was applied for Intel clusters.

- **MPI/Thread Decomposition**: We explore all possible decompositions of the cubical per node grid among processes and threads. Threading is only in the z-dimension.

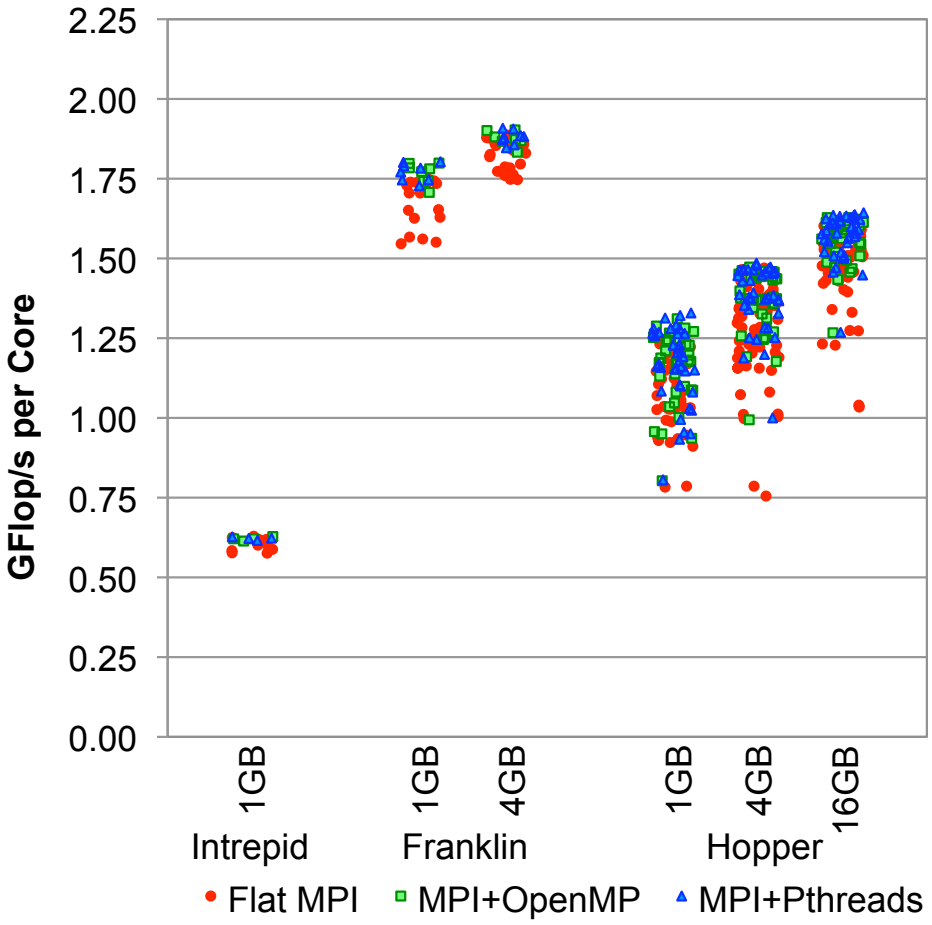
- **Optimization of stream( )**: we explored sending individual velocities and entire faces. We also thread buffer packing. MPI decomposition also affects time spent accessing various faces.



Hierarchical Auto-tuning

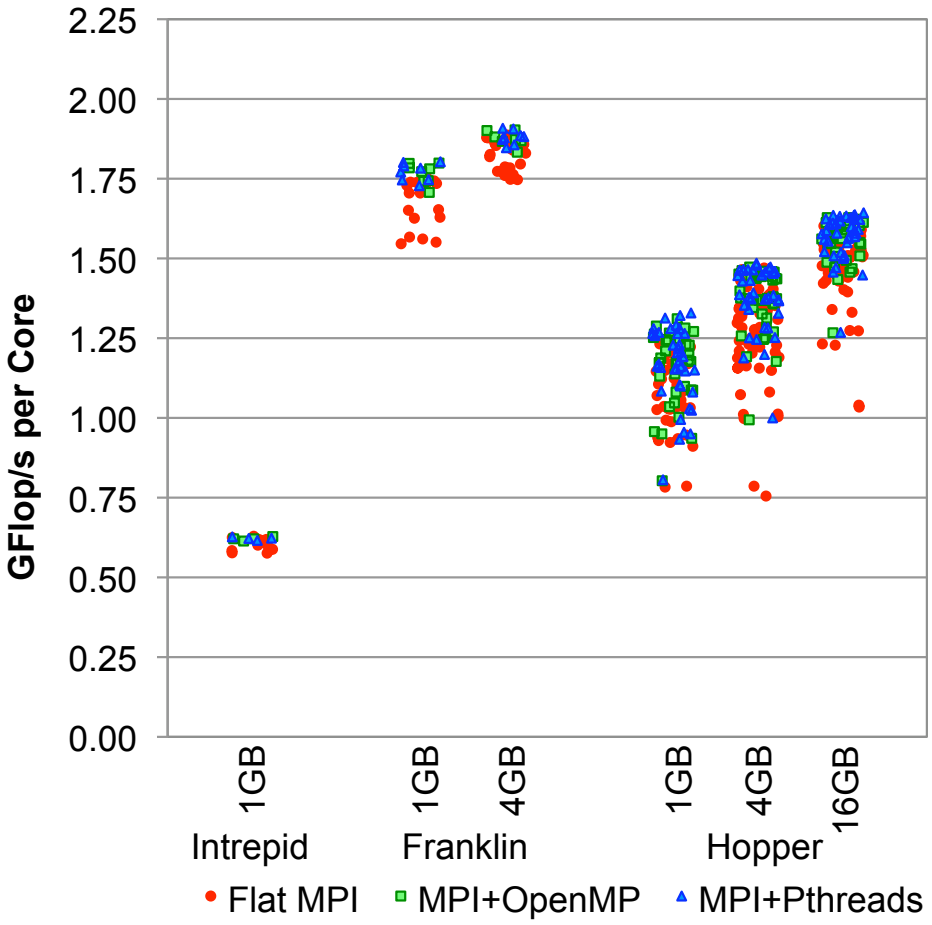
Hierarchical Auto-tuning

- Prior work (IPDPS'08) examined auto-tuning LBMHD on a single node (ignoring time spent in MPI)
- To avoid a combinatorial explosion, we auto-tune the parallel (distributed) application in a two-stage fashion using 64 nodes.
  - Stage 1: explore sequential optimizations on the 4GB problem.
  - Stage 2: explore MPI and thread decomposition for each problem size.
- Once the optimal parameters are determined, we can run problems at scale.



Hierarchical Auto-tuning

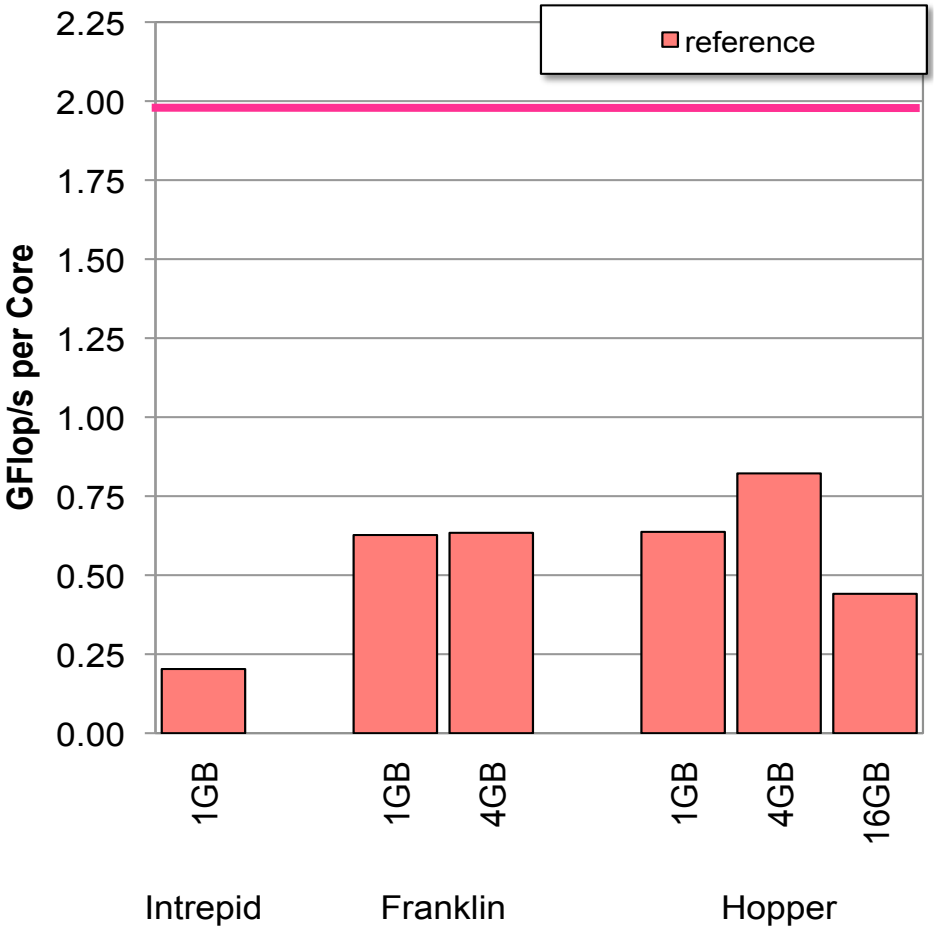
- Best VL was 256 elements on the Opterons (fitting in the L2 was more important than page stanzas)
- Best VL was 128 elements on the BGP (clearly fitting in the L1 did not provide sufficient sequential locality)
- The Cray machines appear to be much more sensitive to MPI and thread decomposition (**threading provided a 30% performance boost over flat MPI**)
- Although OpenMP is NUMA-aware, it did not scale perfectly across multiple sockets (pthreads did).



Results at Scale

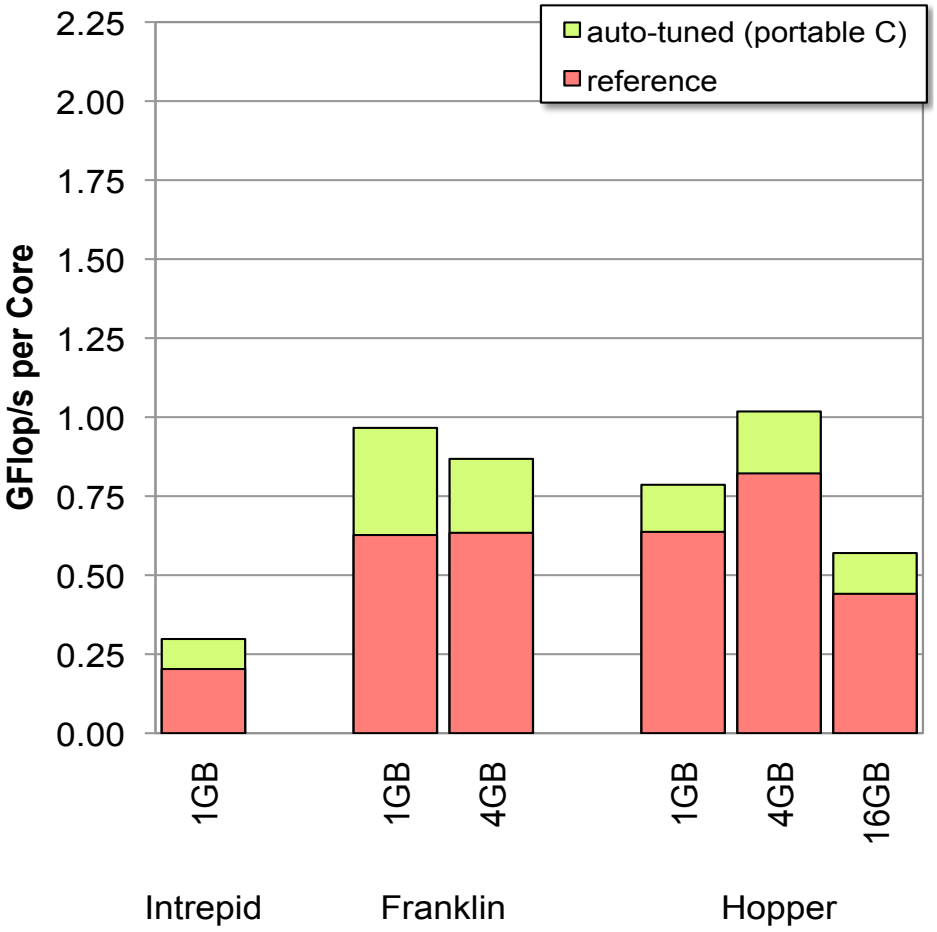
Performance at Scale

- Using the best configurations from our hierarchical auto-tuning, we evaluate ***performance per core*** using 2K nodes on each platform for each problem size.
- Note, due to limited memory per node, not all machines can run all problem configurations.
- **Assuming we are bandwidth-limited, all machines should attain approximately 2GFlop/s. (Roofline limit)**
- Clearly, **reference flat MPI** (which obviates NUMA) performance is far below our expectations.



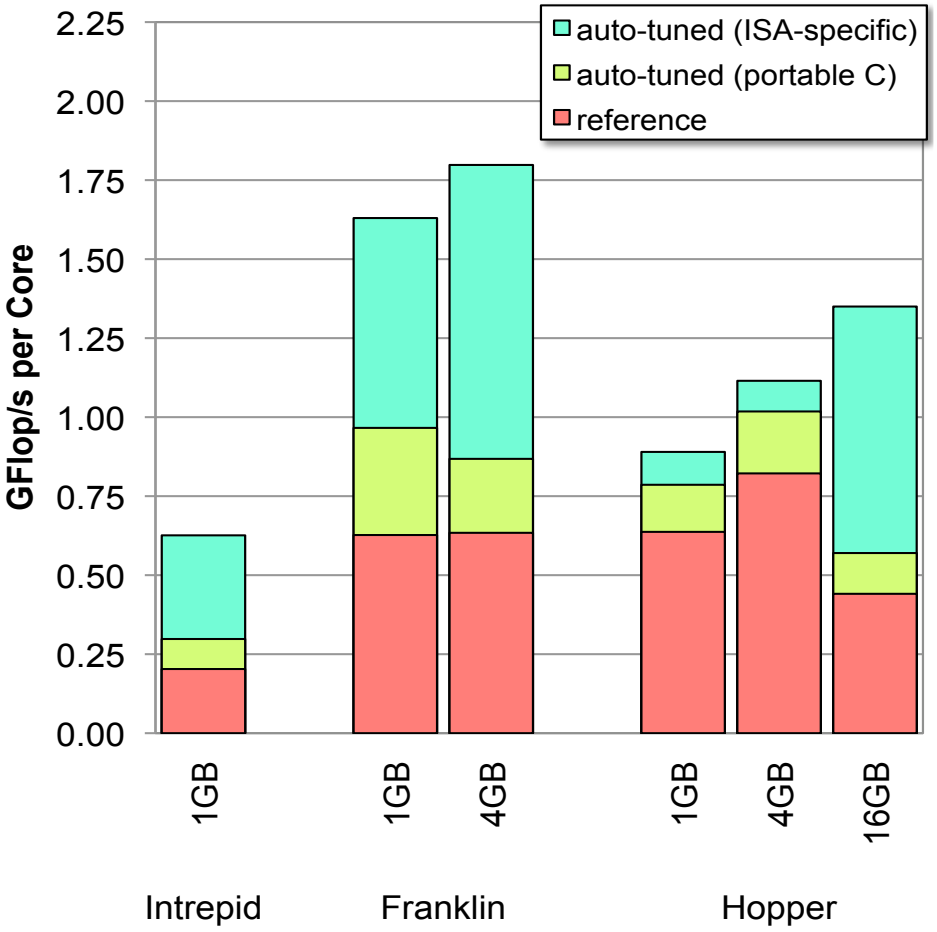
Performance at Scale

- Auto-tuning the sequential implementation in a portable C fashion shows only moderate boosts to performance.
- Clearly, we are well below the performance bound.



Performance at Scale

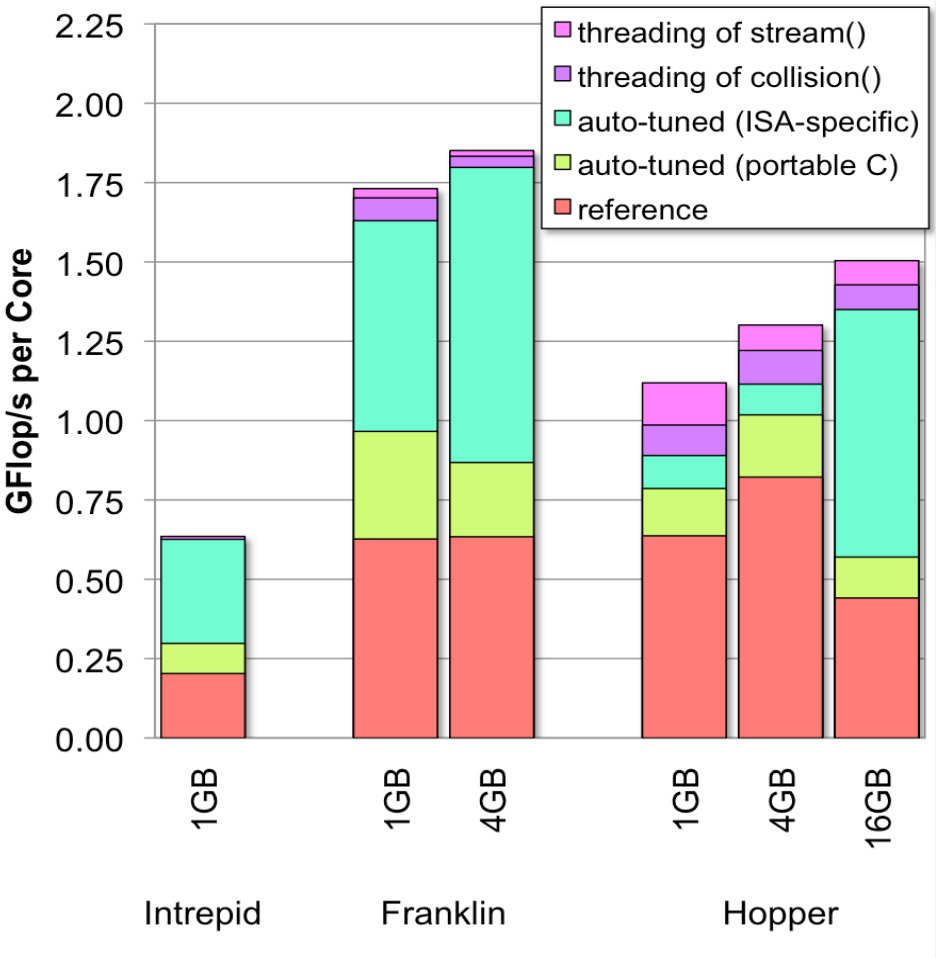
- However, auto-tuning the sequential implementation with ISA-specific optimizations shows a **huge boost on all architectures**.
- Some of this benefit comes from SIMDization/prefetching, but on Franklin, a large benefit comes from cache bypass instructions.



Performance at Scale

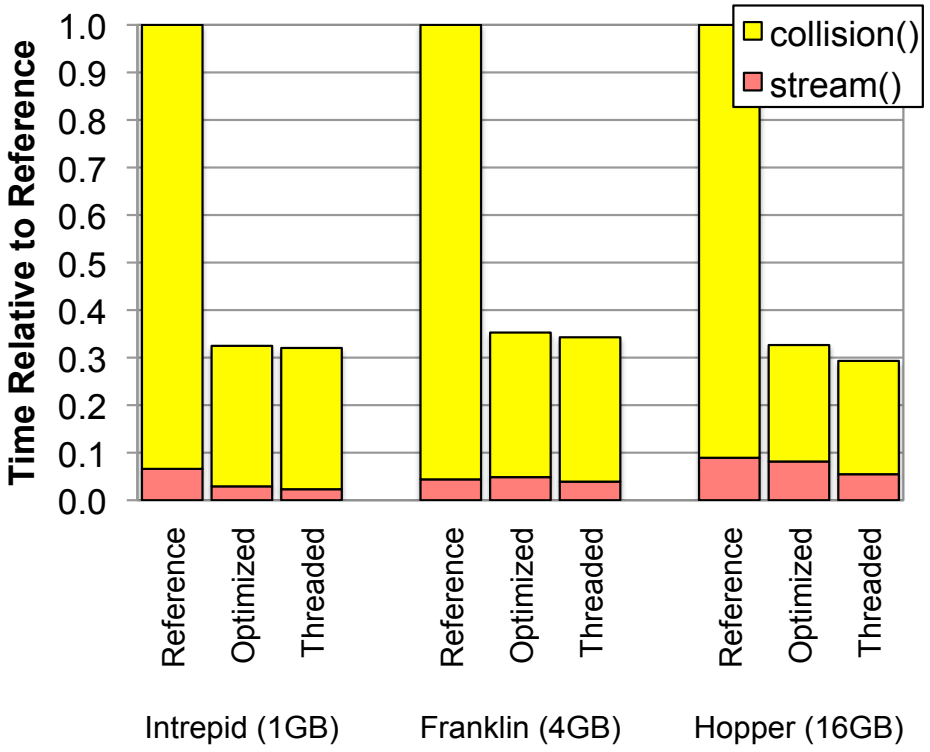
- Threading showed no benefit on Intrepid.
- We observe that performance on Franklin is approaching the 2GFlop/s performance bound.
- In fact, when examining collision( ), we observe that **Franklin attains 94% of its STREAM bandwidth**.

- **Overall, we see a performance boost of 1.6-3.4x over reference.**
- On Hopper, the threading boosts performance by 11-26% depending on problem size.
- **Why does the benefit of threading vary so greatly ?**



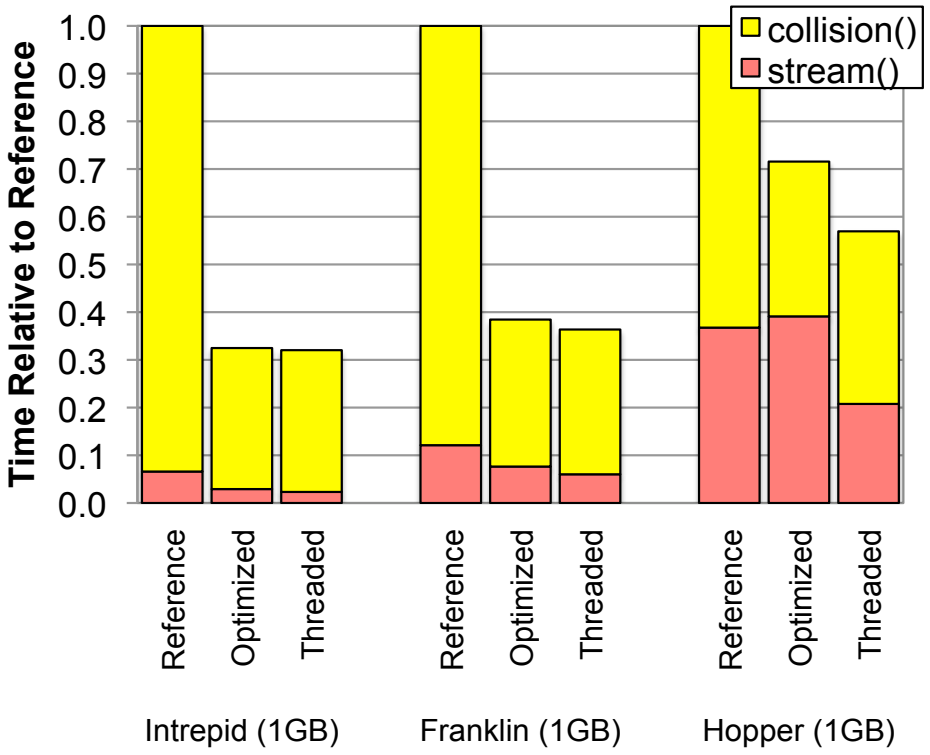
Memory and Communication

- For the largest problem per node, we examine how optimization reduces time spent in collision( ) (computation) and stream( ) (communication).
- Clearly, **sequential auto-tuning can dramatically reduce runtime**, but the time spent in communication remains constant.
- Similarly, **threading reduces the time spent in communication** as it replaces explicit messaging with coherent shared memory accesses.



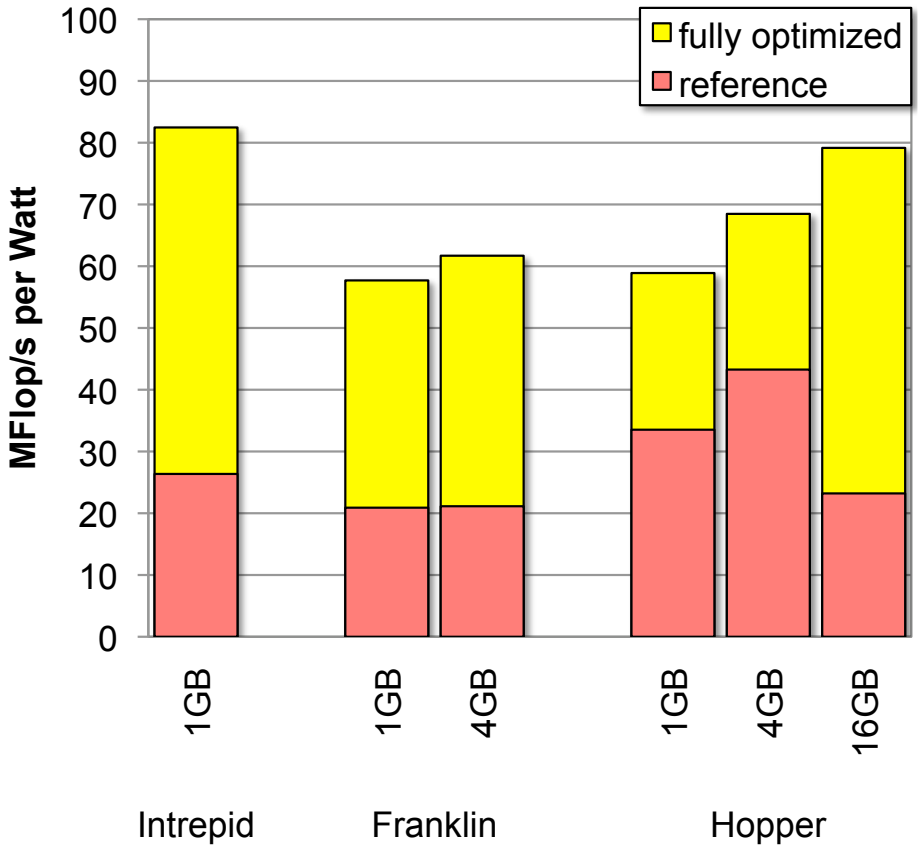
Memory and Communication

- However, when we consider the smallest (1GB) problem, **we see the time spent in stream( ) is significant.** Note, this is an artifact of Hopper's greatly increase ***node*** performance.
- After sequential optimization on Hopper, **more than 55% of the time is spent in communication**.
- This does not bode well for future GPUs or accelerators on this class of computation as they only address computation performance (yellow bars) and lack enough memory to amortize communication (red bars)



Energy Efficiency

- Increasingly, energy is the great equalizer among HPC systems.
- Thus, rather than comparing performance per node or performance per core, we consider energy efficiency (performance per Watt).
- Surprisingly, despite the differences in age and process technology, **all three systems deliver similar energy efficiencies** for this application.
- The energy efficiency of the Opteron-based system only improved by 33% in 2 years.



Conclusions

Conclusions

- Used LBMHD as a testbed to evaluate, analyze, and optimize performance on BGP, XT4, and XE6.

- We employed a two-stage approach to auto-tuning and **delivered a 1.6-3.4x speedup** on a variety of supercomputers/problem sizes.
- We observe a similar improvement in energy efficiency observing that after optimization, all machines delivered similar energy efficiencies.
- We observe that **threading (either OpenMP or pthreads) improves performance over flat MPI** with pthreads making better use of NUMA architectures.
- Unfortunately, performance on small problems is hampered by finite network bandwidth. **Accelerator's sacrifice of capacity to maximize bandwidth exacerbates the problem.**

Future Work

- We used an application-specific solution to maximize performance.

- We will continue to investigate OpenMP issues on NUMA archs.
- Just as advances in in-core performance will outstrip DRAM bandwidth, DRAM bandwidth will outstrip network bandwidth. This will put a great emphasis on **communication hiding** techniques and **on-sided PGAS-like communication.**
- In the future, we can use Active Harmony to generalize the search process and DSL's/source-to-source tools to facilitate code generation.

- Although we showed significant performance gains, in the future, we must ensure these translate to superior (O(N)) algorithms.

Acknowledgements

- All authors from Lawrence Berkeley National Laboratory were supported by the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231.
- This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.
- George Vahala and his research group provided the original (FORTRAN) version of the LBMHD code.

Questions?