

Lattice Boltzmann Hybrid Auto-Tuning on High-End Computational Platforms

Samuel Williams, Jonathan Carter,
Leonid Oliker, John Shalf, Katherine Yelick

Lawrence Berkeley National Laboratory (LBNL)

National Energy Research Scientific Computing Center (NERSC)

SWWilliams@lbl.gov



Outline

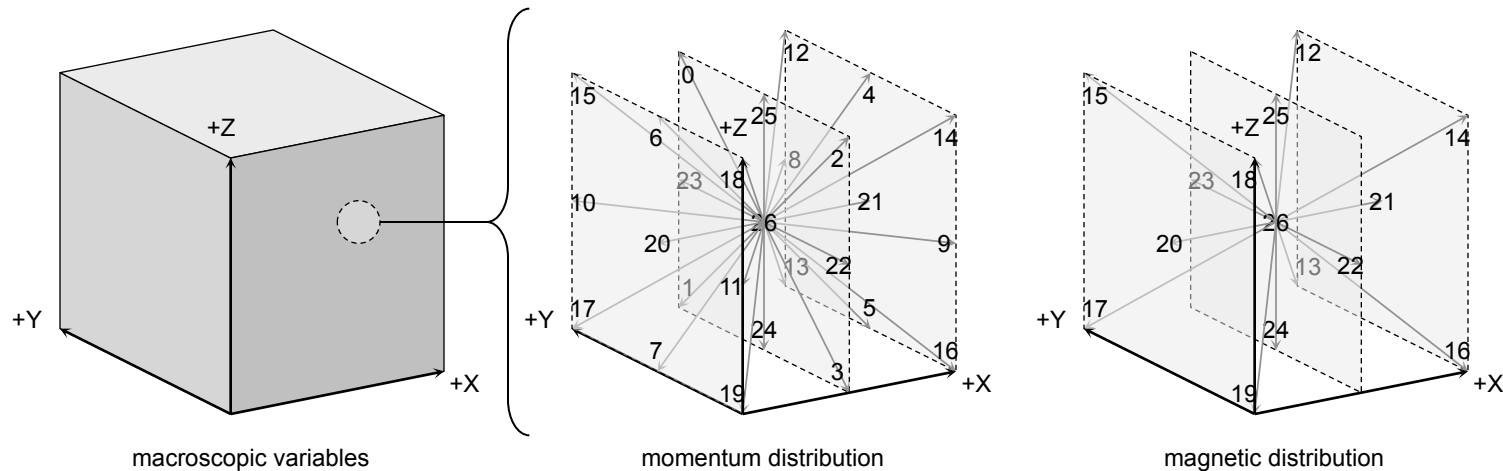
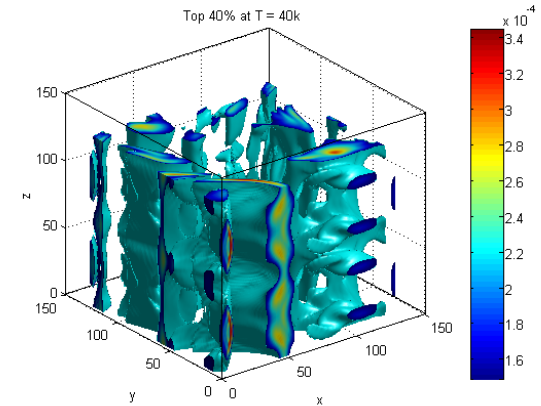
F U T U R E T E C H N O L O G I E S G R O U P

1. LBMHD
2. Auto-tuning LMBHD on Multicore SMPs
3. Hybrid MPI-Pthreads implementations
4. Distributed, Hybrid LBMHD Auto-tuning
5. pthread Results
6. OpenMP results
7. Summary



LBMHD

- ❖ Lattice Boltzmann Magnetohydrodynamics (CFD+Maxwell's Equations)
- ❖ Plasma turbulence simulation via Lattice Boltzmann Method for simulating astrophysical phenomena and fusion devices
- ❖ Three macroscopic quantities:
 - Density
 - Momentum (vector)
 - Magnetic Field (vector)
- ❖ Two distributions:
 - momentum distribution (27 scalar components)
 - magnetic distribution (15 Cartesian vector components)



- ❖ Code Structure
 - time evolution through a series of *collision()* and *stream()* functions
- ❖ When parallelized, *stream()* should constitute 10% of the runtime.
- ❖ *collision()*'s Arithmetic Intensity:
 - Must read 73 doubles, and update 79 doubles per lattice update (1216 bytes)
 - Requires about 1300 floating point operations per lattice update
 - **Just over 1.0 flops/byte (ideal architecture)**
 - Suggests LBMHD is **memory-bound** on the XT4.
- ❖ 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
- ❖ periodic boundary conditions



Auto-tuning LBMHD on Multicore SMPs

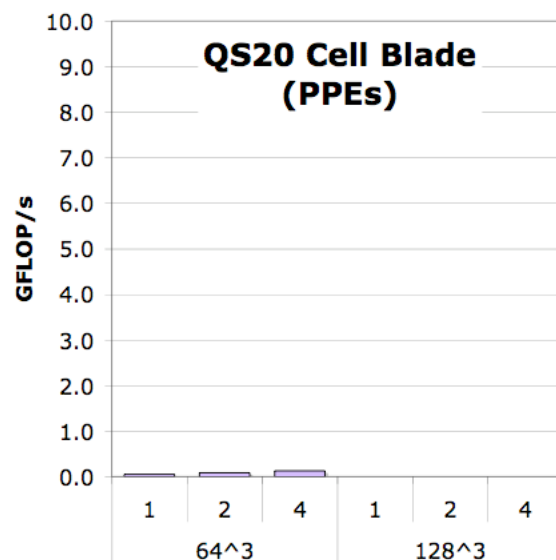
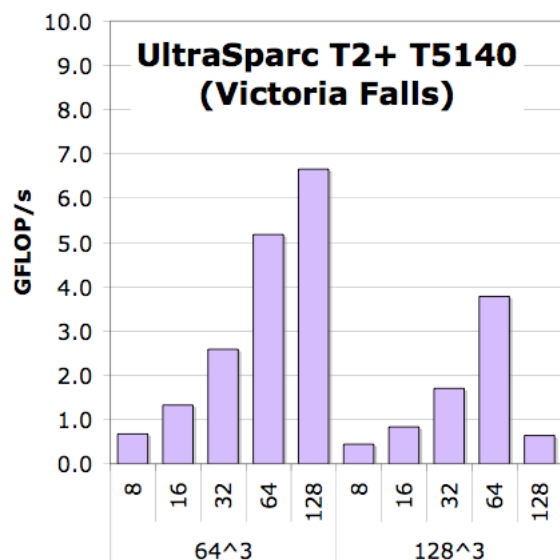
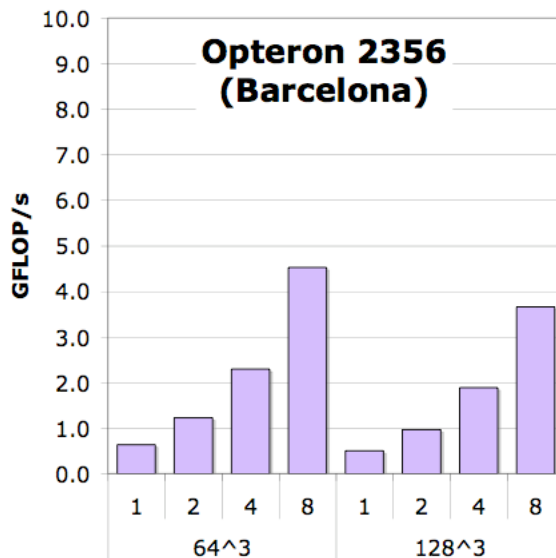
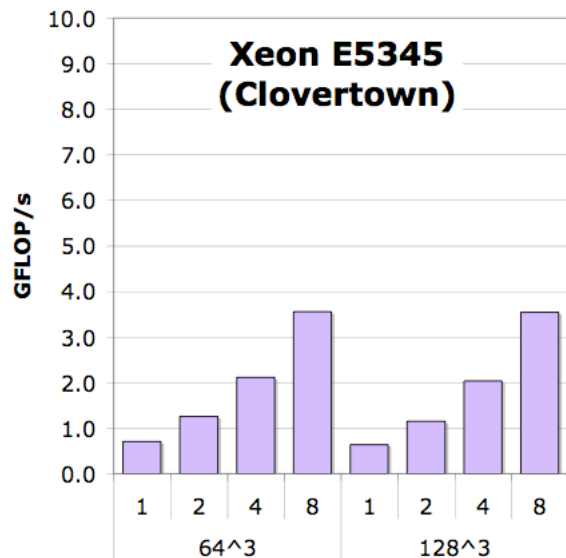
Samuel Williams, Jonathan Carter, Leonid Oliker, John Shalf, Katherine Yelick,
"Lattice Boltzmann Simulation Optimization on Leading Multicore Platforms",
International Parallel & Distributed Processing Symposium (IPDPS), 2008.



LBMHD Performance

(reference implementation)

F U T U R E T E C H N O L O G I E S G R O U P



- ❖ Generally, scalability looks good
- ❖ Scalability is good
- ❖ but is performance good?

Reference+NUMA



Lattice-Aware Padding

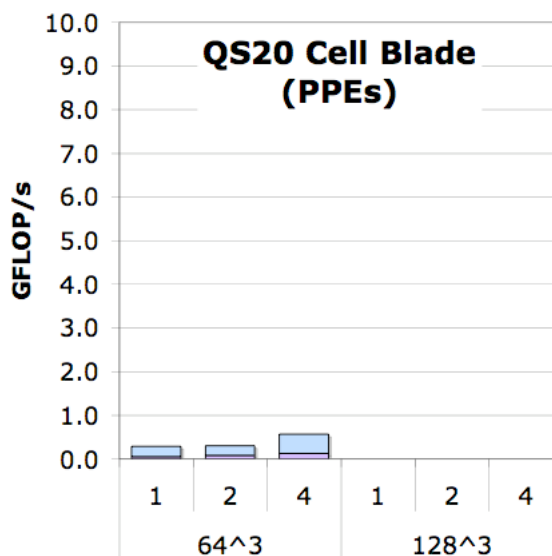
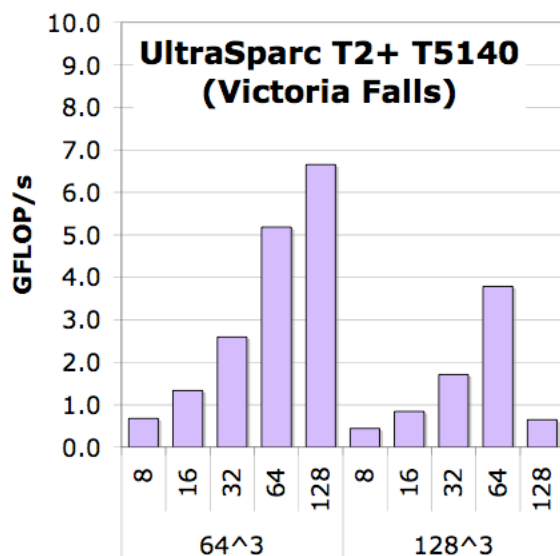
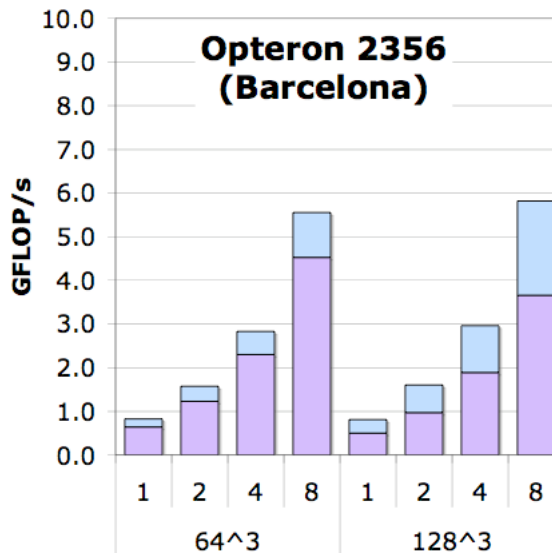
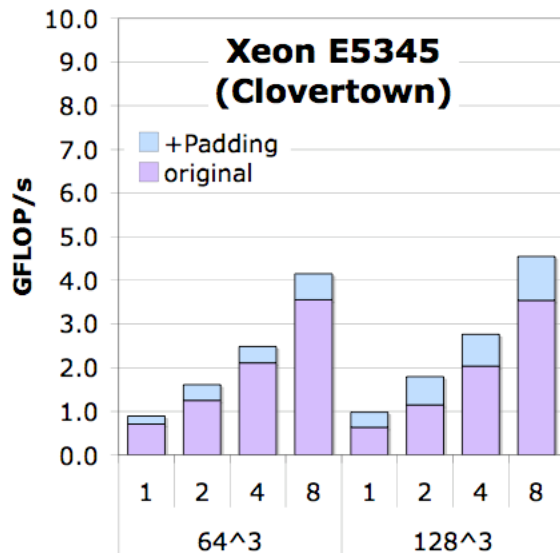
F U T U R E T E C H N O L O G I E S G R O U P

- ❖ For a given lattice update, the requisite velocities can be mapped to a relatively narrow range of cache sets (lines).
- ❖ As one streams through the grid, one cannot fully exploit the capacity of the cache as conflict misses evict entire lines.
- ❖ In an structure-of-arrays format, pad each component such that when referenced with the relevant offsets ($\pm x, \pm y, \pm z$) they are uniformly distributed throughout the sets of the cache
- ❖ Maximizes cache utilization and minimizes conflict misses.

LBMHD Performance

(lattice-aware array padding)

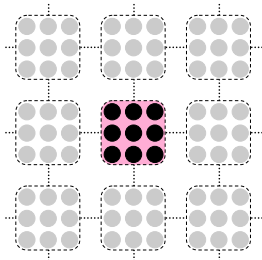
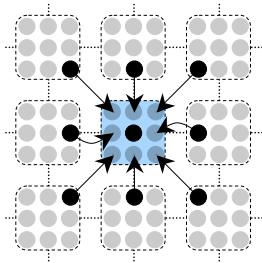
F U T U R E T E C H N O L O G I E S G R O U P



- ❖ LBMHD touches >150 arrays.
- ❖ Most caches have limited associativity
- ❖ Conflict misses are likely
- ❖ Apply **heuristic** to pad arrays

+Padding
Reference+NUMA

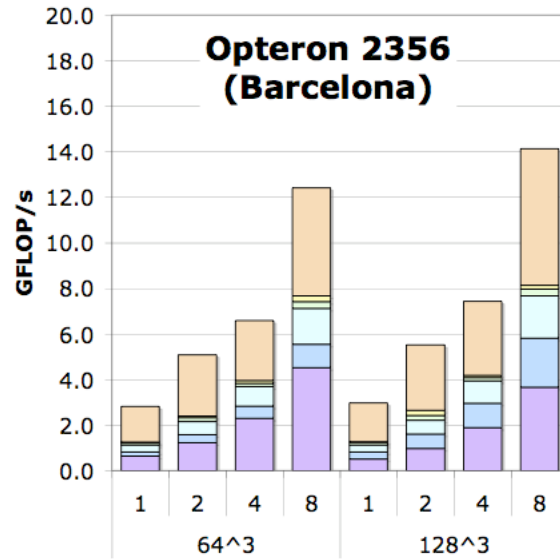
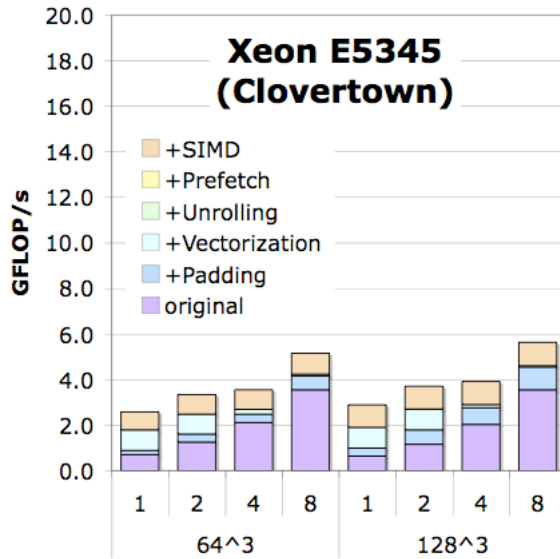
- ❖ Two phases with a lattice method's collision() operator:
 - reconstruction of macroscopic variables
 - updating discretized velocities
- ❖ Normally this is done one point at a time.
- ❖ Change to do a vector's worth at a time (loop interchange + tuning)



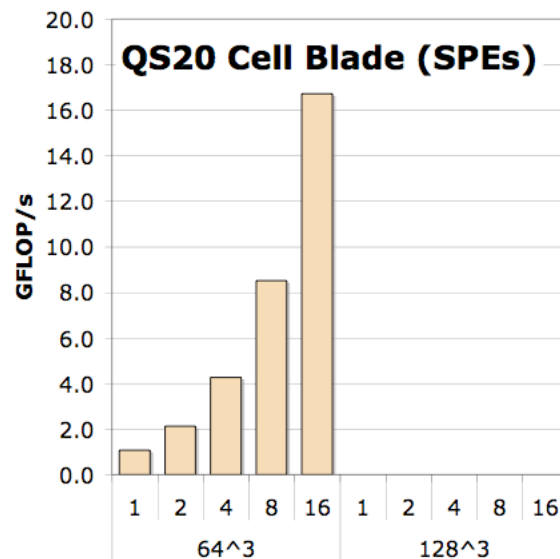
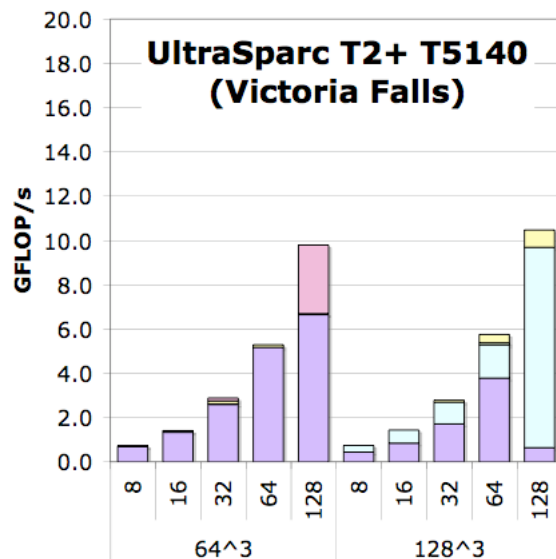
LBMHD Performance

(architecture specific optimizations)

FUTURE TECHNOLOGIES GROUP



- ❖ Add unrolling and reordering of inner loop
- ❖ Additionally, it exploits SIMD where the compiler doesn't
- ❖ Include a SPE/Local Store optimized version



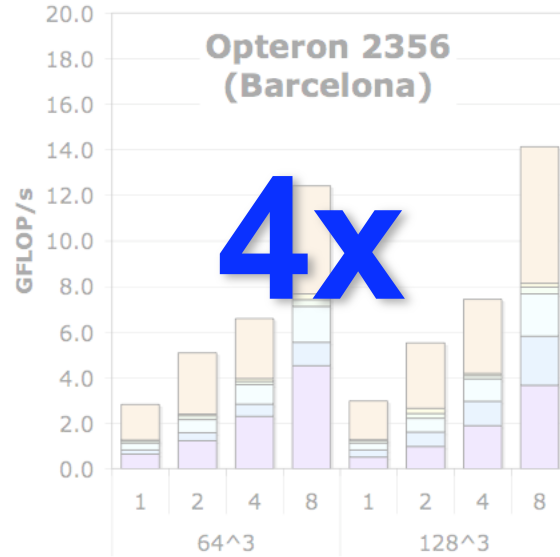
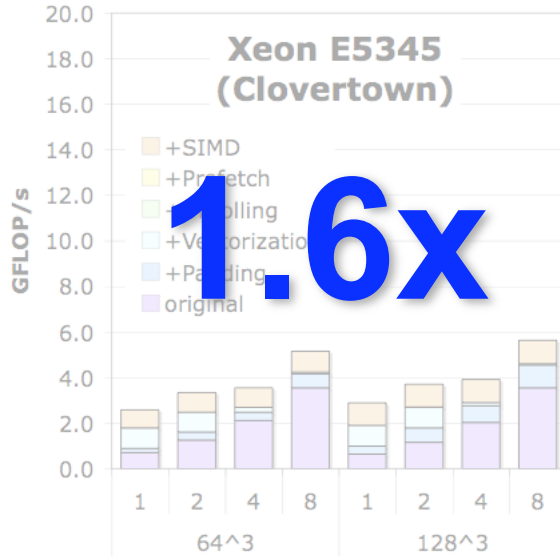
- +small pages
- +Explicit SIMDization
- +SW Prefetching
- +Unrolling
- +Vectorization
- +Padding
- Reference+NUMA



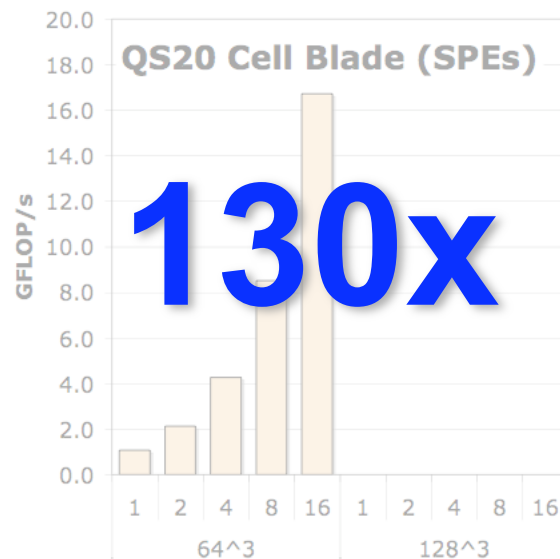
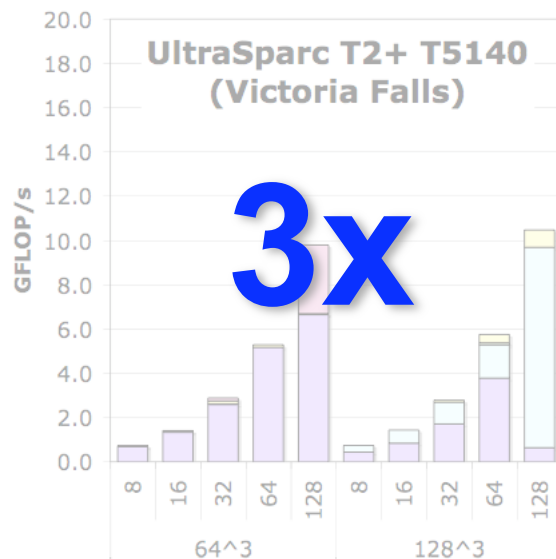
LBMHD Performance

(architecture specific optimizations)

FUTURE TECHNOLOGIES GROUP



- ❖ Add unrolling and reordering of inner loop
- ❖ Additionally, it exploits SIMD where the compiler doesn't
- ❖ Include a SPE/Local Store optimized version



- +small pages
- +Explicit SIMDization
- +SW Prefetching
- +Unrolling
- +Vectorization
- +Padding
- Reference+NUMA

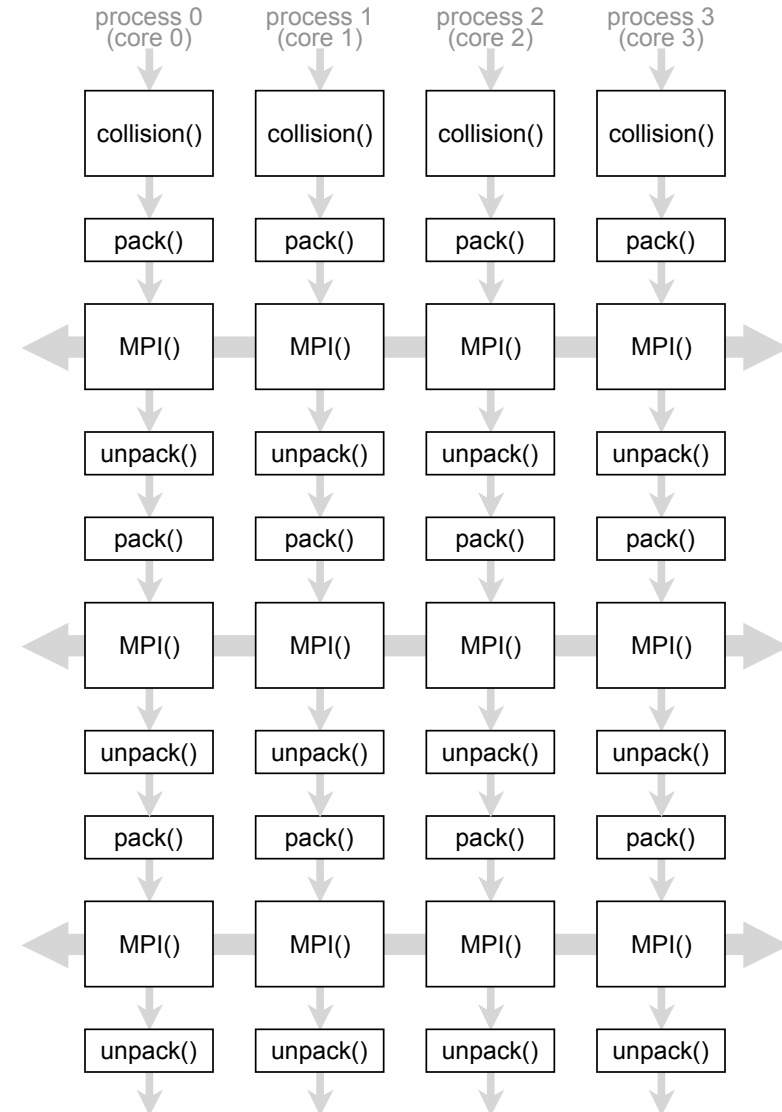
- ❖ Ignored MPP (distributed) world
- ❖ Kept problem size fixed and cubical
- ❖ When run with only 1 process per SMP, maximizing threads per process always looked best

Hybrid MPI+Pthreads Implementation

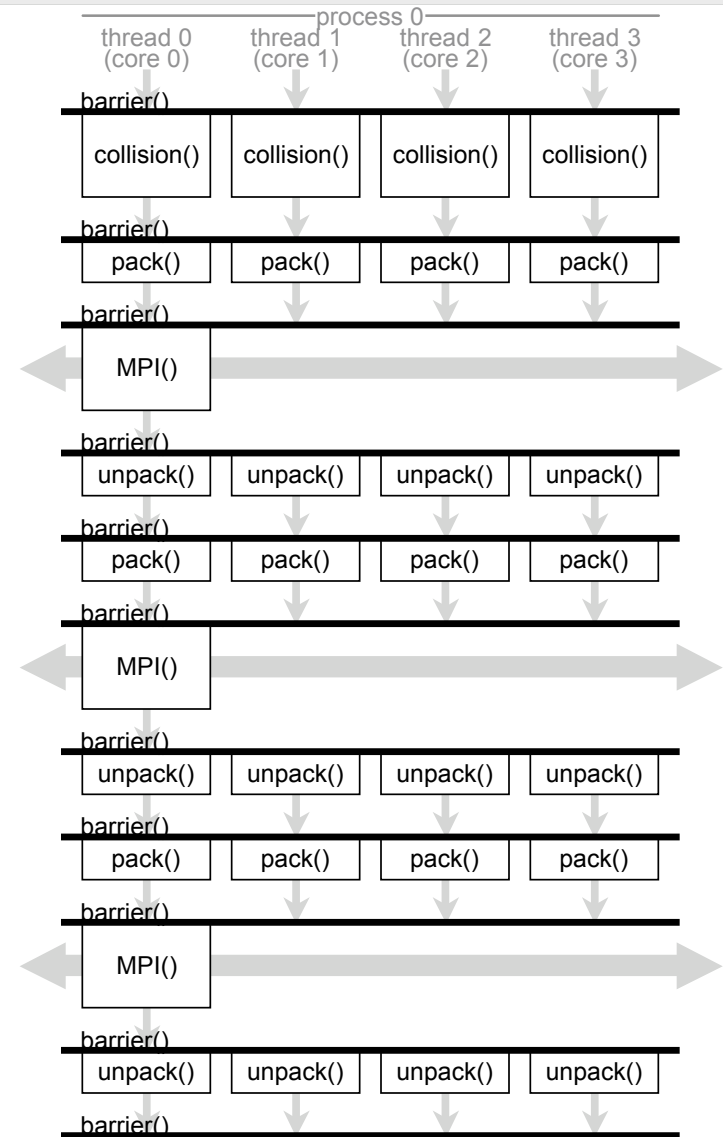
Flat MPI

F U T U R E T E C H N O L O G I E S G R O U P

- ❖ In the flat MPI world, there is one process per core, and only one thread per process
- ❖ All communication is through MPI



- ❖ As multicore processors already provide cache coherency for free, we can exploit it to **reduce MPI overhead and traffic**.
- ❖ We examine using pthreads and OpenMP for threading (other possibilities exist)
- ❖ For correctness in pthreads, we are required to include an intra-process (thread) barrier between function calls for correctness.
(we wrote our own)
- ❖ Implicitly, OpenMP will barrier via the #pragma
- ❖ We can choose any balance between processes/ node and threads/process
(we explored powers of 2)
- ❖ Initially, we did not assume a thread-safe MPI implementation (many versions return MPI_THREAD_SERIALIZED). **As such, only thread 0 performs MPI calls**



Distributed, Hybrid Auto-tuning



The Distributed Auto-tuning Problem

F U T U R E T E C H N O L O G I E S G R O U P

- ❖ We believe that even for relatively large problems, auto-tuning only the local computation (e.g. IPDPS'08) will deliver sub-optimal MPI performance.
- ❖ Want to explore MPI/Hybrid decomposition as well
- ❖ We have a combinatoric explosion in the search space coupled with a large problem size (number of nodes)

at each concurrency:

for all aspect ratios

for all process/thread balances

for all thread grids

for all data structures

for all coding styles (reference, vectorized, vectorized+SIMDized)

for all prefetching

for all vector lengths

for all code unrollings/reorderings

benchmark

- ❖ We employ a resource-efficient 3-stage greedy algorithm that successively prunes the search space:

for all data structures

for all coding styles (reference, vectorized, vectorized+SIMDized)

for all prefetching

for all vector lengths

for all code unrollings/reorderings

benchmark

1. Prune variant space

at limited concurrency (single node):

for all aspect ratios

for all probe sizes and ranges

for all thread grids

benchmark

2. Prune parameter space

at full concurrency:

for all process/thread counts

benchmark

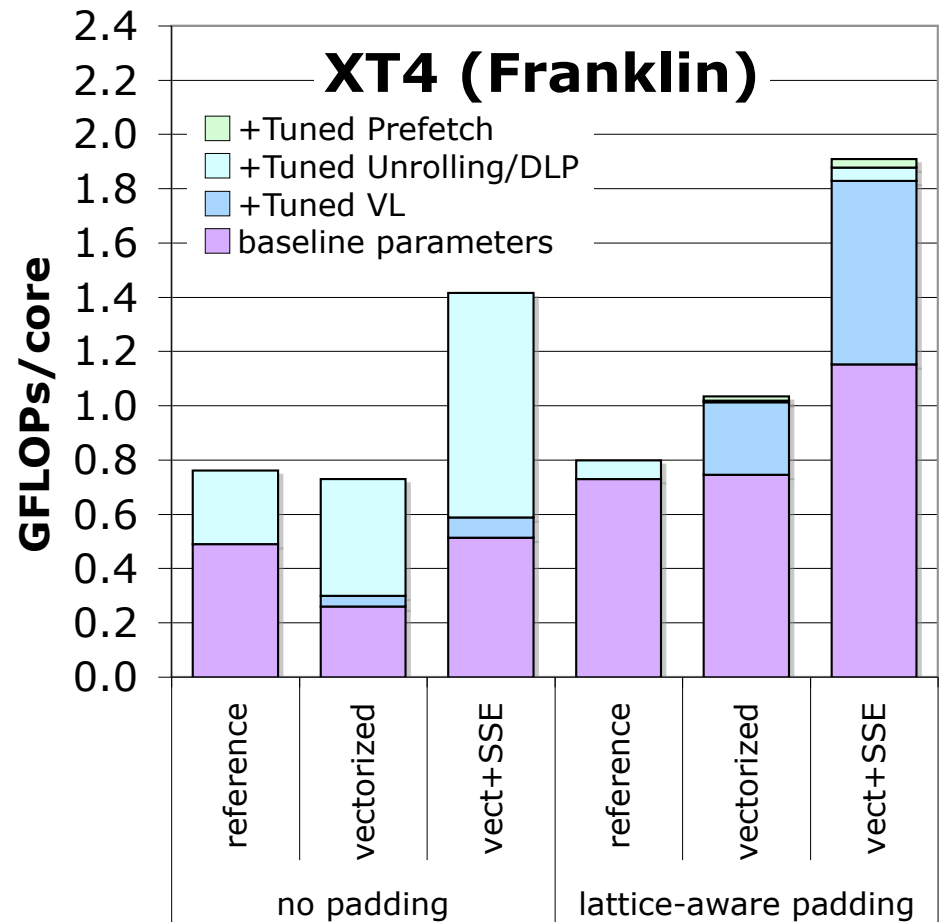
3. Production

Stage 1

F U T U R E T E C H N O L O G I E S G R O U P

- ❖ In stage 1, we prune the code generation space.
- ❖ We ran this as a 128^3 problem with 4 threads.

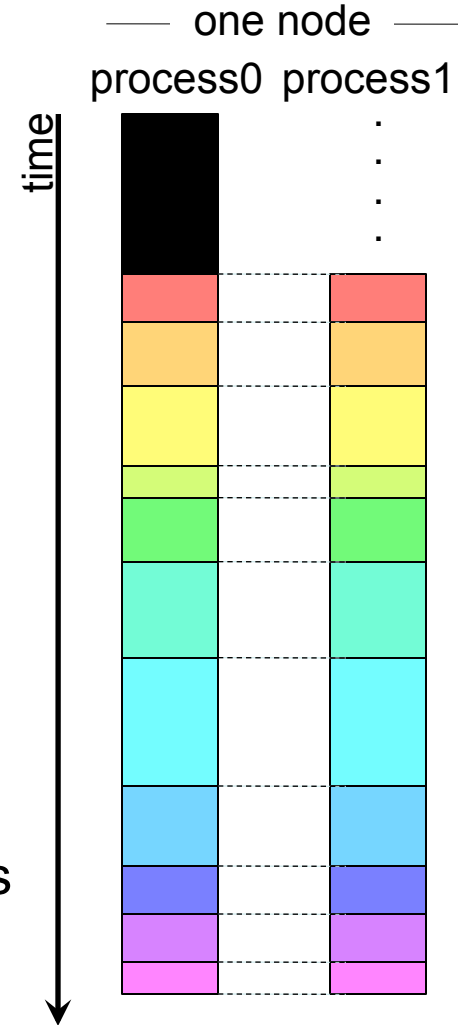
- ❖ As VL, unrolling, and reordering may be problem dependent, we only prune:
 - padding
 - coding style
 - prefetch distance
- ❖ We observe that vectorization with SIMDization, and a prefetch distance of 64 Bytes worked best



Stage 2

F U T U R E T E C H N O L O G I E S G R O U P

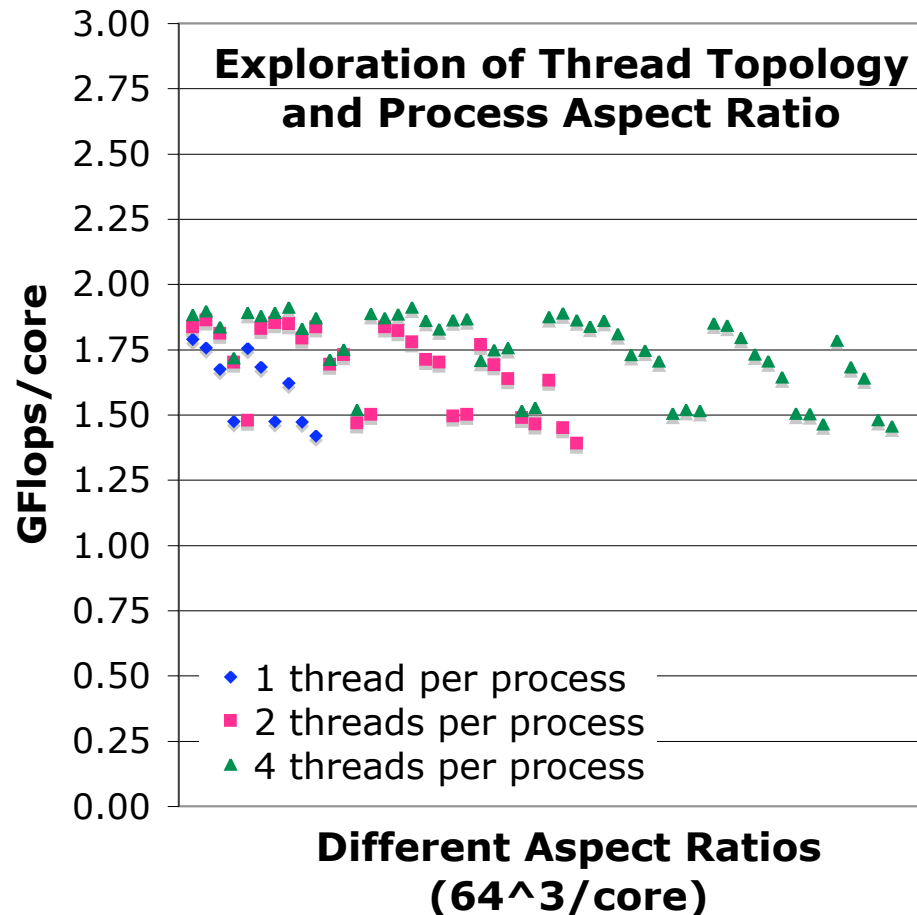
- ❖ Hybrid Auto-tuning requires we mimic the SPMD environment
- ❖ Suppose we wish to explore this color-coded optimization space.
- ❖ In the serial world (or fully threaded nodes), the tuning is easily run
- ❖ However, in the MPI or hybrid world a problem arises as processes are not guaranteed to be synchronized.
- ❖ As such, one process may execute some optimizations faster than others simply due to fortuitous scheduling with another processes' trials
- ❖ Solution: add an `MPI_barrier()` around each trial (a configuration with 100's of iterations)



Stage 2 (continued)

F U T U R E T E C H N O L O G I E S G R O U P

- ❖ We create a database of optimal VL/unrolling/DLP parameters for each thread/process balance, thread grid, and aspect ratio configuration



Stage 3

F U T U R E T E C H N O L O G I E S G R O U P

- ❖ Given the data base from Stage 2,
- ❖ we run few large problem using the best known parameters/thread grid for different thread/process balances.
- ❖ We select the parameters based on minimizing
 - overall local time
 - *collision()* time
 - local *stream()* time



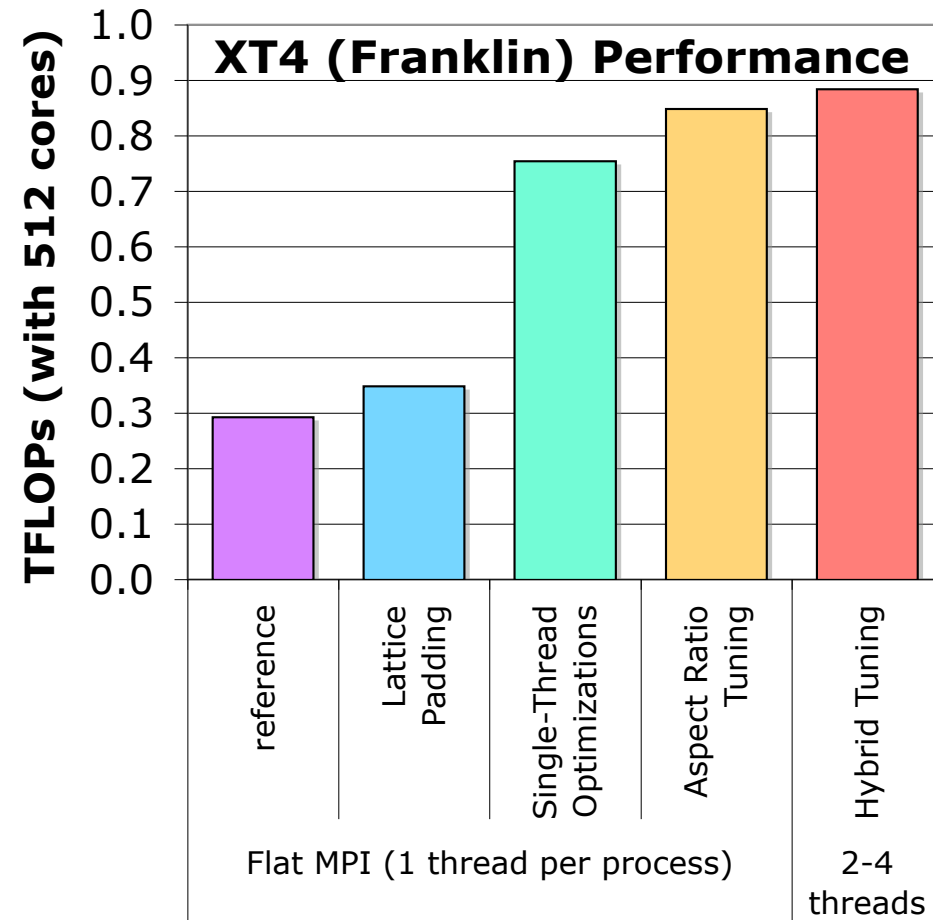
Results

XT4 Results

(512³ problem on 512 cores)

F U T U R E T E C H N O L O G I E S G R O U P

- ❖ Finally, we present the best data for progressively more aggressive auto-tuning efforts
- ❖ Note each of the last 3 bars may have unique MPI decompositions as well as VL/unroll/DLP
- ❖ Observe that for this large problem, **auto-tuning flat MPI delivered significant boosts (2.5x)**
- ❖ However, expanding auto-tuning to include the domain decomposition and balance between threads and processes provided an extra 17%
- ❖ 2 processes with 2 threads was best





What about OpenMP ?



Conversion to OpenMP

F U T U R E T E C H N O L O G I E S G R O U P

- ❖ Converting the auto-tuned pthreads implementation to OpenMP seems relatively straightforward (#pragma omp parallel for)
- ❖ We modified code to be single source that supports:
 - **Flat MPI**
 - **MPI+pthread**s
 - **MPI+OpenMP**
- ❖ However, it is imperative (especially on NUMA SMPs) to correctly utilize the available affinity mechanisms:
 - on XT, aprun has options to handle this
 - on linux clusters (like NERSC's Carver), user must manage it:

```
#ifdef _OPENMP
    #pragma omp parallel
    {Affinity_Bind_Thread( MyFirstHWThread+omp_get_thread_num());}
#else
    Affinity_Bind_Thread( MyFirstHWThread+Thread_Rank);
#endif
```
 - use both to be safe
- ❖ Failure to miss these or other key pragmas can cut performance in half (or 90% in one particularly bad bug)

Optimization of Stream()

F U T U R E T E C H N O L O G I E S G R O U P

- ❖ In addition, we further optimized the stream() routine along 3 axes:
 1. messages could be blocked (24 velocities/direction/phase/process) or aggregated (1/direction/phase/process)
 2. packing could be sequential (thread 0 does all the work) or thread parallel (using pthreads/openMP)
 3. MPI calls could be serialized (thread 0 does all the work) or parallel (MPI_THREAD_MULTIPLE)

Optimization of Stream()

F U T U R E T E C H N O L O G I E S G R O U P

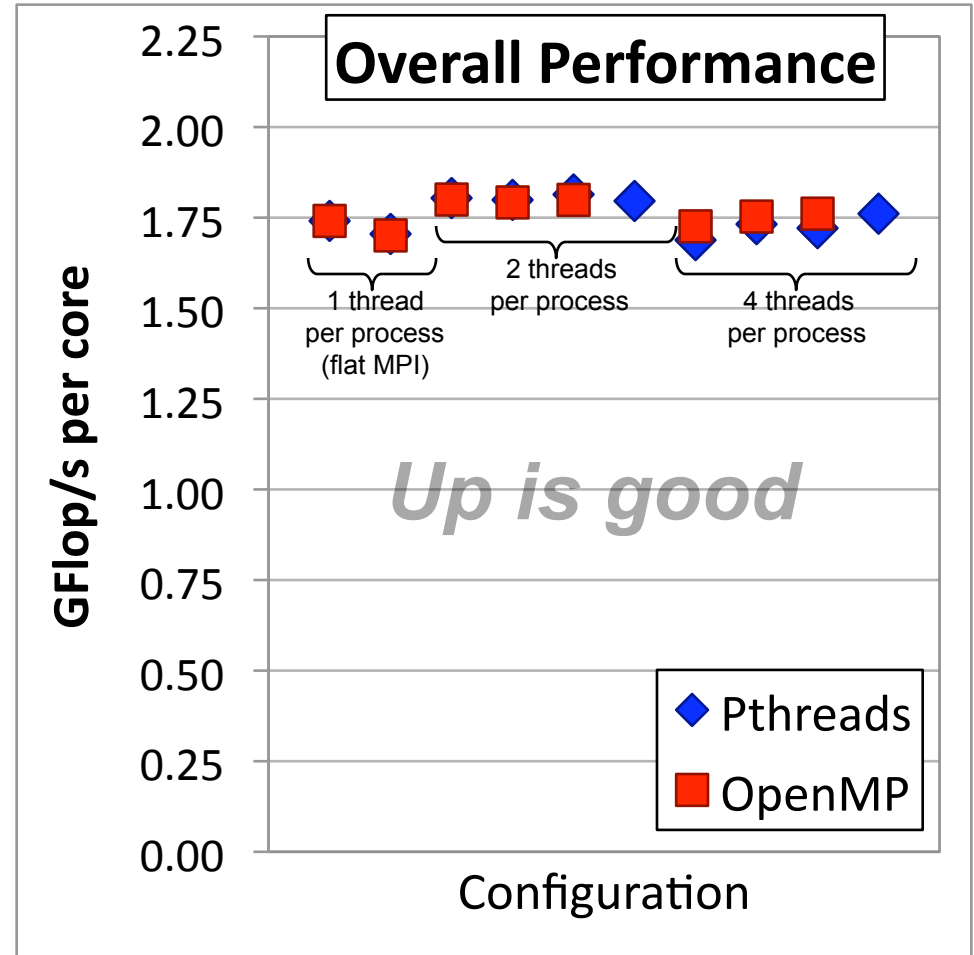
- ❖ In addition, we further optimized the stream() routine along 3 axes:
 1. messages could be blocked (24 velocities/direction/phase/process) or aggregated (1/direction/phase/process)
 2. packing could be sequential (thread 0 does all the work) or thread parallel (using pthreads/openMP)
 3. MPI calls could be serialized (thread 0 does all the work) or parallel (MPI_THREAD_MULTIPLE)
- ❖ Of these eight combinations, we implemented 4:
 - aggregate, sequential packing, serialized MPI
 - blocked, sequential packing, serialized MPI
 - aggregate, parallel packing, serialized MPI (**simplest openMP code**)
 - blocked, parallel packing, parallel MPI (**simplest pthread code**)

Optimization of Stream()

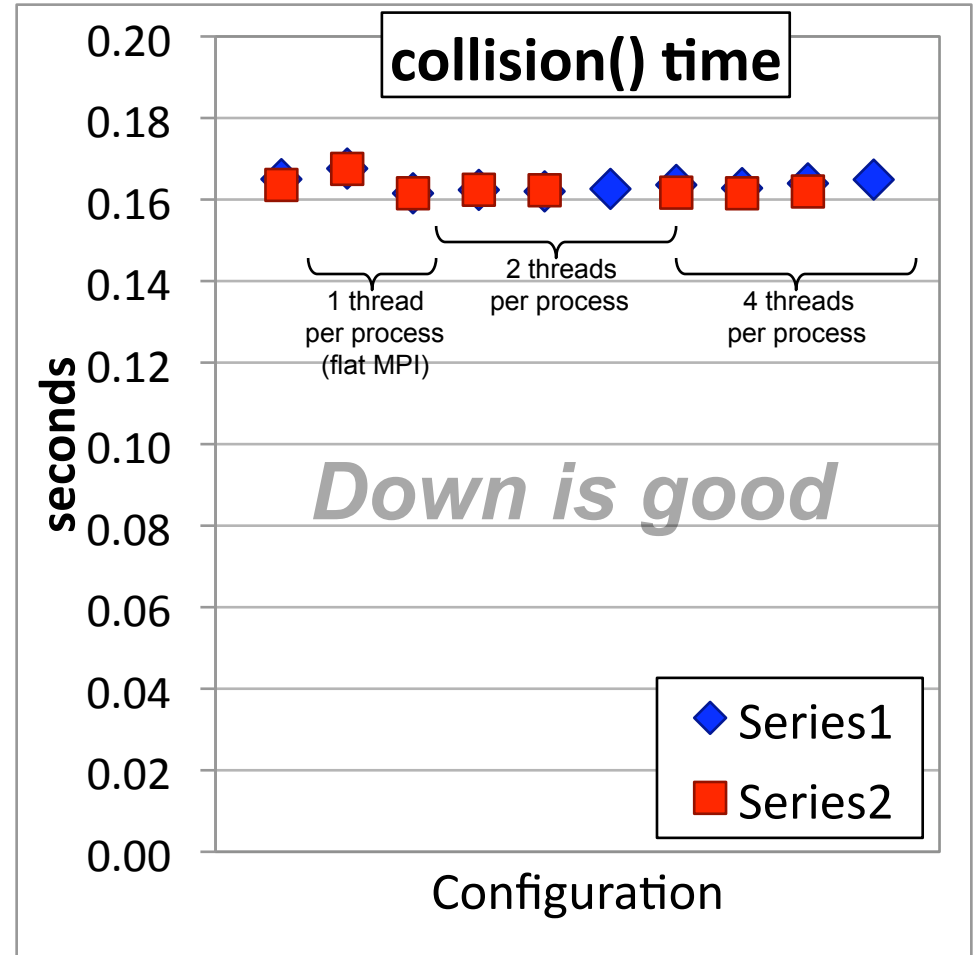
F U T U R E T E C H N O L O G I E S G R O U P

- ❖ In addition, we further optimized the stream() routine along 3 axes:
 1. messages could be blocked (24 velocities/direction/phase/process) or aggregated (1/direction/phase/process)
 2. packing could be sequential (thread 0 does all the work) or thread parallel (using pthreads/openMP)
 3. MPI calls could be serialized (thread 0 does all the work) or parallel (MPI_THREAD_MULTIPLE)
- ❖ Of these eight combinations, we implemented 4:
 - aggregate, sequential packing, serialized MPI
 - blocked, sequential packing, serialized MPI
 - aggregate, parallel packing, serialized MPI (**simplest openMP code**)
 - blocked, parallel packing, parallel MPI (**simplest pthread code**)
- ❖ Threaded MPI on Franklin requires using
 - threaded MPICH
 - calling using MPI_THREAD_MULTIPLE
 - setting MPICH_MAX_THREAD_SAFETY=multiple

- ❖ When examining overall performance per core (512³ problem with cores), we see choice made relatively little difference
- ❖ MPI+pthreads was slightly faster with 2thread/process
- ❖ MPI+OpenMP was slightly faster with 4 threads/process
- ❖ choice of best stream() optimization is dependent on thread concurrency and threading model



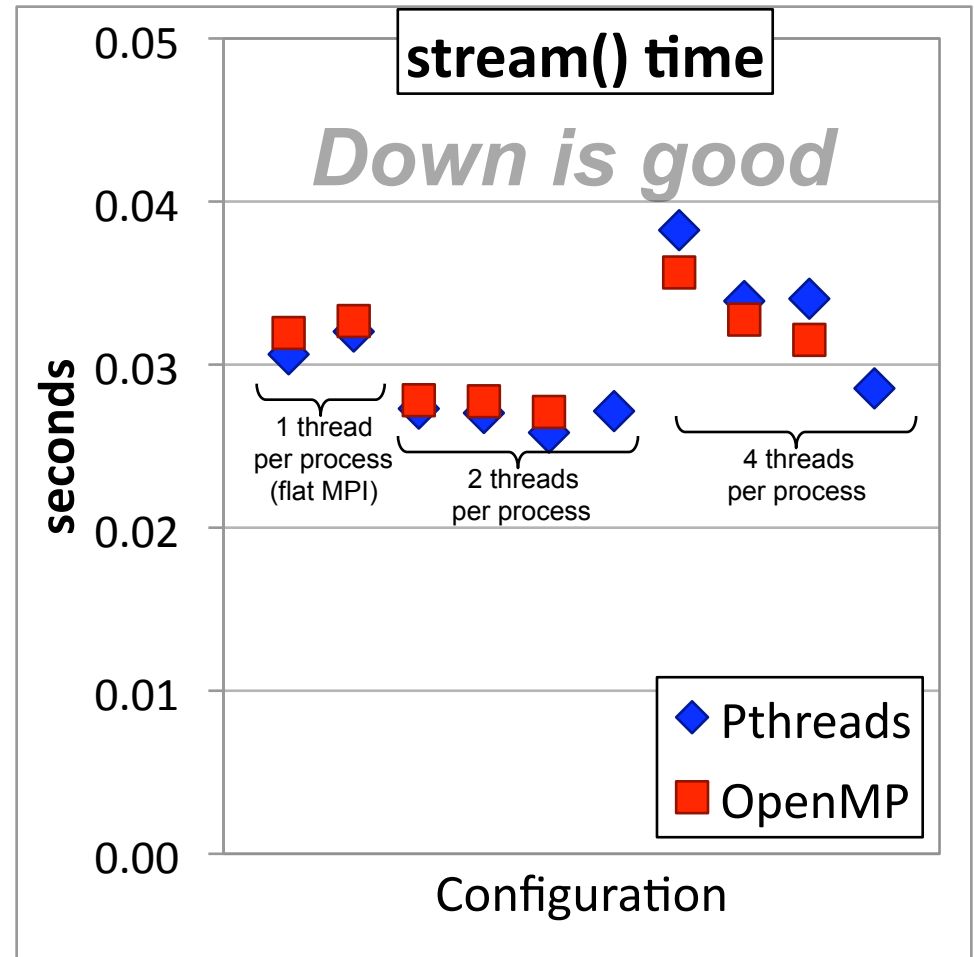
- ❖ When we look at collision time, we see that surprisingly 2 threads per process delivered slightly better performance for pthreads, but 4 threads per process was better for OpenMP.
- ❖ Interestingly, threaded MPI resulted in slower compute time



MPI vs. MPI+Pthreads vs. MPI+OpenMP

F U T U R E T E C H N O L O G I E S G R O U P

- ❖ Variation in stream time was dramatic with 4 threads.
- ❖ Here the blocked implementation was far faster.
- ❖ Interestingly, pthreads was faster for 2 threads, openMP was faster for 4 threads.



Summary & Discussion

- ❖ Multicore cognizant auto-tuning **dramatically improves (2.5x)** flat MPI performance.
- ❖ Tuning the domain decomposition and hybrid implementations yielded almost an **additional 20%** performance boost.
- ❖ Although hybrid MPI promises improved performance through reduced communication, the observed benefit is thus far small.
- ❖ Moreover, the performance difference among hybrid models is small.
- ❖ Initial experiments on the XT5 (Hopper) and the Nehalem cluster (Carver) show similar results (little trickier to get good OpenMP performance on the linux cluster)
- ❖ LBM's probably will not make the case for hybrid programming models (purely concurrent with no need for collaborative behavior)



Acknowledgements

F U T U R E T E C H N O L O G I E S G R O U P

- ❖ Research supported by DOE Office of Science under contract number DE-AC02-05CH11231
- ❖ All XT4 simulations were performed on the XT4 (Franklin) at the National Energy Research Scientific Computing Center (NERSC)
- ❖ George Vahala and his research group provided the original (FORTRAN) version of the LBMHD code.

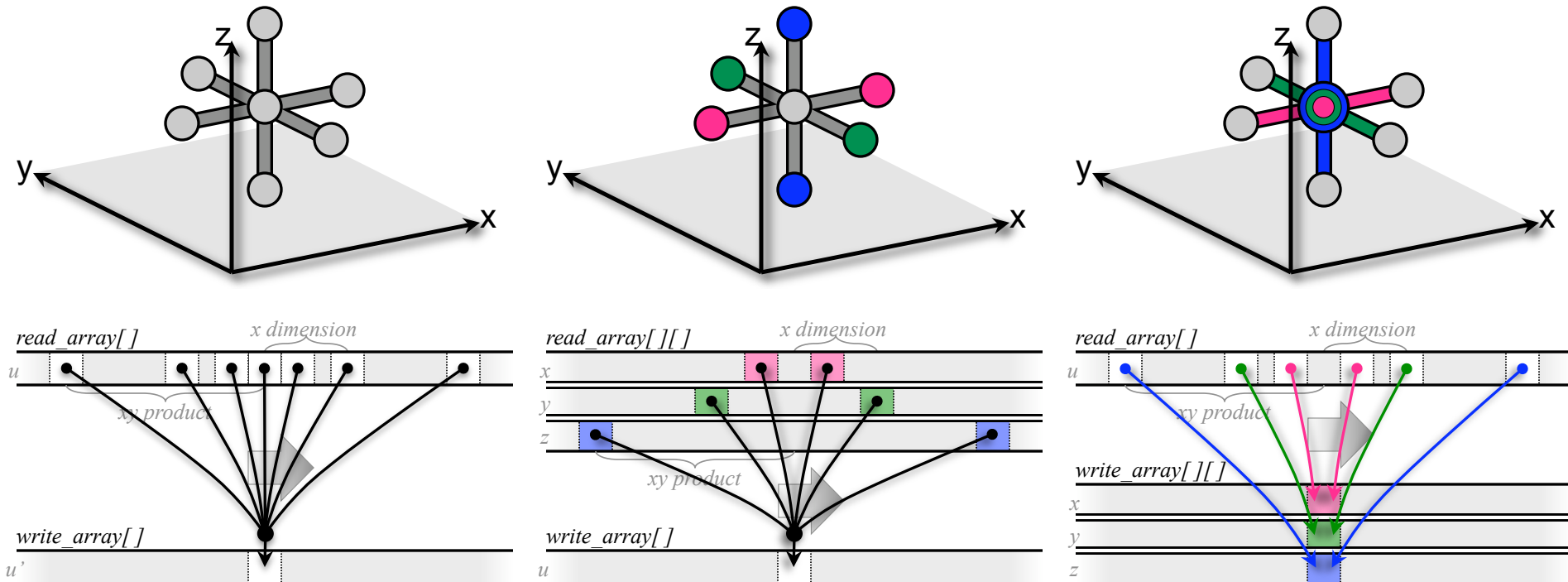


Questions?



BACKUP SLIDES

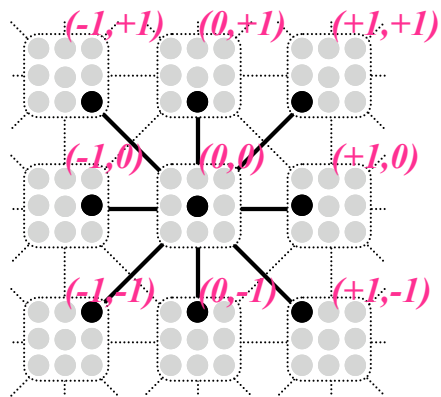
- ❖ Laplacian, Divergence, and Gradient
- ❖ Different reuse, Different #'s of read/write arrays



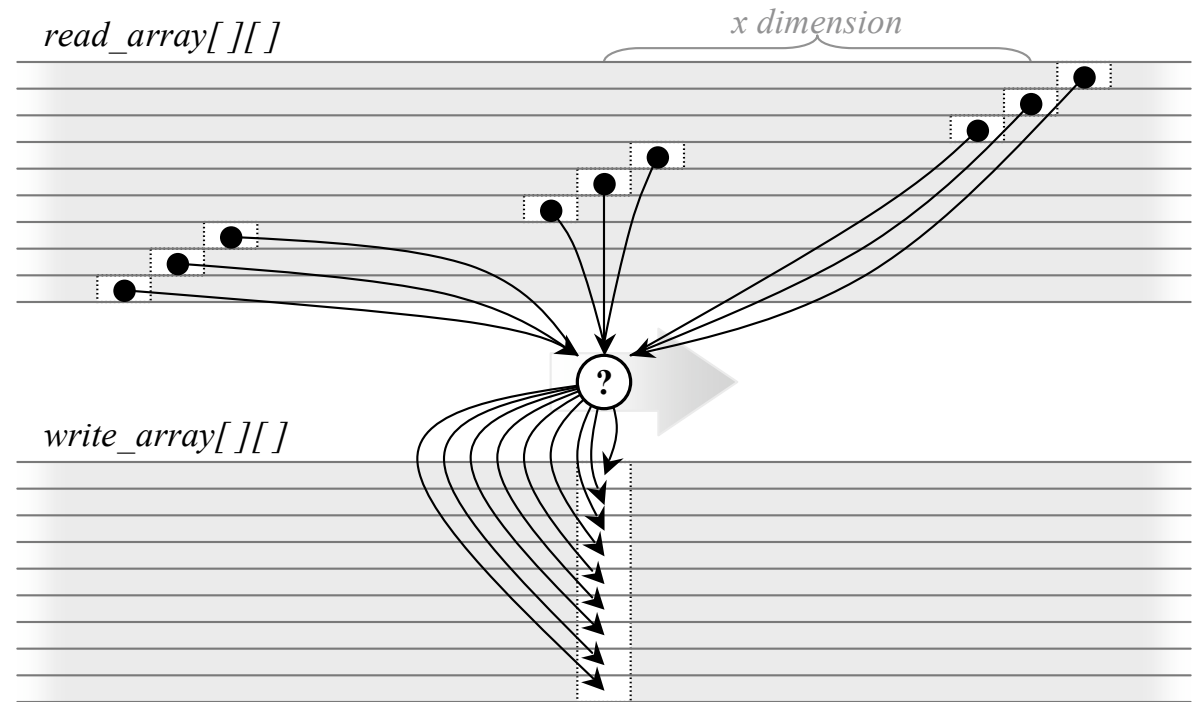
LBMHD Stencil

F U T U R E T E C H N O L O G I E S G R O U P

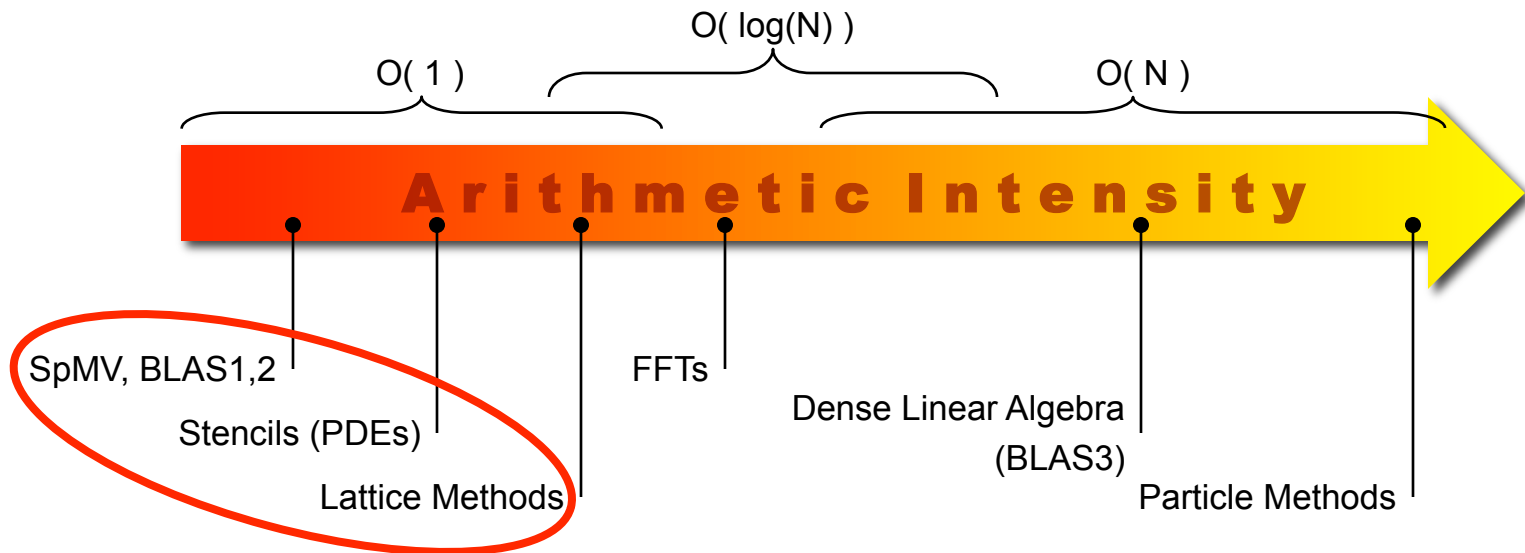
- ❖ Simple example reading from 9 arrays and writing to 9 arrays
- ❖ Actual LBMHD reads 73, writes 79 arrays



(a)



(b)



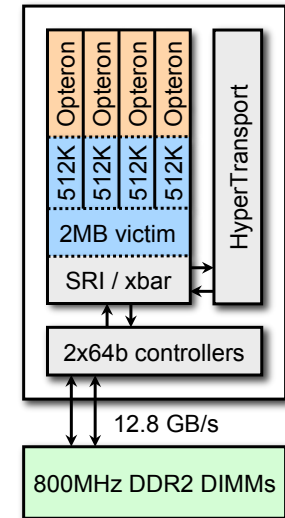
- ❖ **True Arithmetic Intensity (AI) $\sim$ Total Flops / Total DRAM Bytes**
- ❖ Some HPC kernels have an arithmetic intensity that scales with problem size (increased temporal locality), but remains constant on others
- ❖ Arithmetic intensity is ultimately limited by compulsory traffic
- ❖ Arithmetic intensity is diminished by conflict or capacity misses.

Kernel Arithmetic Intensity and Architecture

F U T U R E T E C H N O L O G I E S G R O U P

- ❖ For a given architecture, one may calculate its flop:byte ratio.
- ❖ For a 2.3GHz Quad Core Opteron (like in the XT4),
 - 1 SIMD add + 1 SIMD multiply per cycle per core
 - 12.8GB/s of DRAM bandwidth
 - $= 36.8 / 12.8 \sim \mathbf{2.9 \text{ flops per byte}}$

- ❖ When a kernel's arithmetic intensity is substantially less than the architecture's flop:byte ratio, transferring data will take longer than computing on it
→ **memory-bound**



- ❖ When a kernel's arithmetic intensity is substantially greater than the architecture's flop:byte ratio, computation will take longer than data transfers
→ **compute-bound**