Auto-tuning a Lattice Boltzmann Computation for a Hybrid MPI-Pthreads Environment

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

1. LBMHD
2. Previous work: Auto-tuning LMBHD on Multicore SMPs
3. Hybrid MPI-Pthreads implementations
4. Distributed, Hybrid LBMHD Auto-tuning
5. Results
6. Summary
LBMHD
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)
LBMHD

- 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
Previous Work: Auto-tuning LBMHD on Multicore SMPs


Best Paper, Application Track
Generally, scalability looks good
- **Scalability is good**
- **but is performance good?**
Lattice-Aware Padding

- 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)

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

- 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)

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

- +SIMD
- +Prefetch
- +Unrolling
- +Vectorization
- +Padding
- original

<table>
<thead>
<tr>
<th>Xeon E5345 (Clovertown)</th>
<th>Opteron 2356 (Barcelona)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFLOP/s</td>
<td>GFLOP/s</td>
</tr>
<tr>
<td>20.0</td>
<td>20.0</td>
</tr>
<tr>
<td>18.0</td>
<td>18.0</td>
</tr>
<tr>
<td>16.0</td>
<td>16.0</td>
</tr>
<tr>
<td>14.0</td>
<td>14.0</td>
</tr>
<tr>
<td>12.0</td>
<td>12.0</td>
</tr>
<tr>
<td>10.0</td>
<td>10.0</td>
</tr>
<tr>
<td>8.0</td>
<td>8.0</td>
</tr>
<tr>
<td>6.0</td>
<td>6.0</td>
</tr>
<tr>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UltraSparc T2+ T5140 (Victoria Falls)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFLOP/s</td>
</tr>
<tr>
<td>20.0</td>
</tr>
<tr>
<td>18.0</td>
</tr>
<tr>
<td>16.0</td>
</tr>
<tr>
<td>14.0</td>
</tr>
<tr>
<td>12.0</td>
</tr>
<tr>
<td>10.0</td>
</tr>
<tr>
<td>8.0</td>
</tr>
<tr>
<td>6.0</td>
</tr>
<tr>
<td>4.0</td>
</tr>
<tr>
<td>2.0</td>
</tr>
<tr>
<td>0.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>QS20 Cell Blade (SPEs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFLOP/s</td>
</tr>
<tr>
<td>20.0</td>
</tr>
<tr>
<td>18.0</td>
</tr>
<tr>
<td>16.0</td>
</tr>
<tr>
<td>14.0</td>
</tr>
<tr>
<td>12.0</td>
</tr>
<tr>
<td>10.0</td>
</tr>
<tr>
<td>8.0</td>
</tr>
<tr>
<td>6.0</td>
</tr>
<tr>
<td>4.0</td>
</tr>
<tr>
<td>2.0</td>
</tr>
<tr>
<td>0.0</td>
</tr>
</tbody>
</table>

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

collision() only
FUTURE TECHNOLOGIES GROUP

LBMHD Performance
(architecture specific optimizations)

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

1.6x

4x

3x

130x

*collision() only
Limitations

- 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 Implementations
Flat MPI

- 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 use pthreads for threading (other possibilities exist)

For correctness, we are required to include an intra-process (thread) barrier between function calls for correctness. (we wrote our own)

We can choose any balance between processes/node and threads/process (we explored powers of 2)

We did not assume a thread-safe MPI implementation. As such, only thread 0 performs MPI calls.
Distributed, Hybrid Auto-tuning
The Distributed Auto-tuning Problem

- 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
Our Approach

- 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 thread grids
            - benchmark
  - at limited concurrency (single node):
    - for all aspect ratios
      - for all process/thread balances
        - benchmark
  - at full concurrency:
    - for all process/thread balances
      - benchmark

1. Prune variant space
2. Prune parameter space
3. Production
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.
Hybrid Auto-tuning requires we mimic a 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.
We create a database of optimal VL/unrolling/DLP parameters for each thread/process balance, thread grid, and aspect ratio configuration.
Stage 3

- 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
  - $\text{collision}(\ )$ time
  - local $\text{stream}(\ )$ time
Results
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.
When examining the execution time breakdown, we see how the auto-tuner consistently favored faster `collision()` times.

In the hybrid world we see a trade off between MPI time and `stream()` time.

As threads/process increase, we get less bandwidth, but less traffic.
Summary & Discussion
Summary

- Multicore cognizant auto-tuning dramatically improves (2.5x) even 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, it is critical all components be thread-parallelized.
An auto-tuned library routine is inappropriate for this code
(common optimizations applied to a unique kernel)

Unfortunately, an auto-tuning compiler is also insufficient
(ultimate performance is heavily dependent on lattice-aware padding)

Need a third option that allows for:
- application of standard optimizations to unique kernels
- subsumes full control over data allocation cognizant of cache topology and memory access pattern.

An area of current and future work
Acknowledgements

- 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
Memory access patterns for Stencils

- Laplacian, Divergence, and Gradient
- Different reuse, Different #'s of read/write arrays
- Simple example reading from 9 arrays and writing to 9 arrays
- Actual LBMHD reads 73, writes 79 arrays
True Arithmetic Intensity (AI) ~ 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

- 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
  - \( \frac{36.8}{12.8} \approx 2.9 \text{ flops per byte} \)

- 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