Memory-Efficient Optimization of Gyrokinetic Particle-to-Grid Interpolation for Multicore Processors

Kamesh Madduri, **Samuel Williams**, Stephane Ethier, Leonid Oliker, John Shalf, Erich Strohmaier, Katherine Yelick

Lawrence Berkeley National Laboratory (LBNL)
National Energy Research Scientific Computing Center (NERSC)
Princeton Plasma Physics Laboratory (PPPL)

SWWilliams@lbl.gov

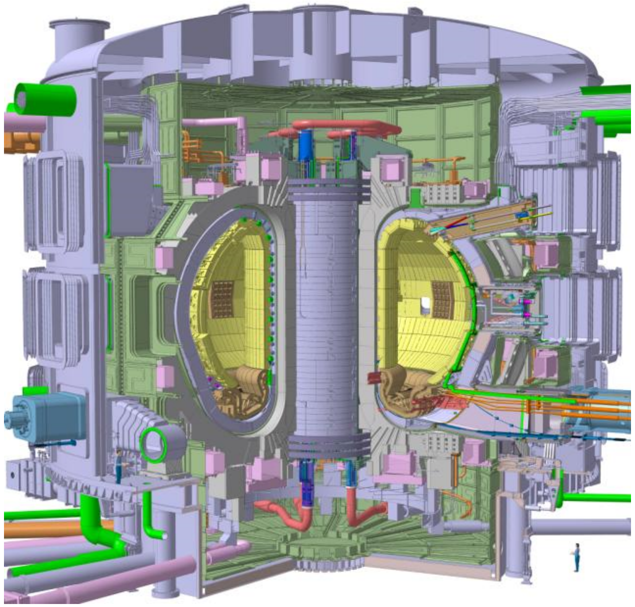
Outline

1. Gyrokinetic Toroidal Code
2. Challenges for efficient PIC simulations
3. Solutions for multicore
4. Results
5. Summary and Discussion

Gyrokinetic Toroidal Code

Gyrokinetic Toroidal Simulations

- Simulate the particle-particle interactions of a charged plasma in a Tokamak fusion reactor
- With millions of particles per processor, the naïve N^2 method is totally intractable.
- Solution is to use a particle-in-cell (PIC) method



Particle-in-Cell Methods

- Particle-in-cell (or particle-mesh) methods simulate particle-particle interactions in O(N) time by examining the field rather than individual forces.

- Typically involves iterating on four steps:
  - Scatter Charge: determine ρ
  - Poisson Solve: $\nabla^2\varphi \sim \rho$
  - Gather: $\nabla\varphi$ accelerates particles
  - Push:

- This requires creation of two auxiliary meshes (arrays):
  - the spatial distribution of charge density
  - the spatial distribution of electromagnetic potential

- In the sequential world, the sizes of the particle arrays are an order of magnitude larger than the grids

Challenges:

- Technology
- PIC
- GTC
- Memory-level parallelism
- Locality

Technology

- In the past DRAM capacity per core grew exponentially.
- In the future, **DRAM costs will dominate** the cost & power of extreme scale machines
- As such, DRAM per socket will remain constant or grow slower than cores

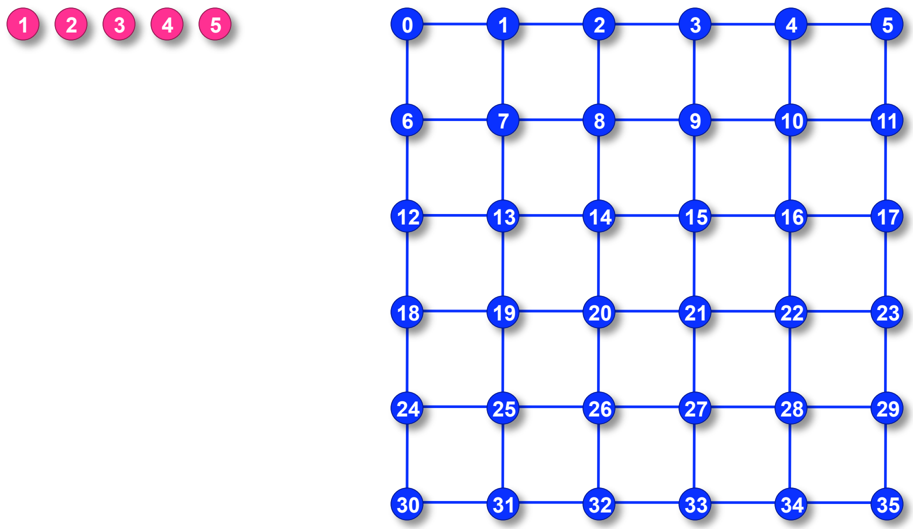
- Applications must be re-optimized for a fixed DRAM budget

  = sustained Flop/s per byte of DRAM capacity

- Algorithms/optimizations whose DRAM capacity requirements scale linearly with the number of cores are unacceptable

PIC Challenges

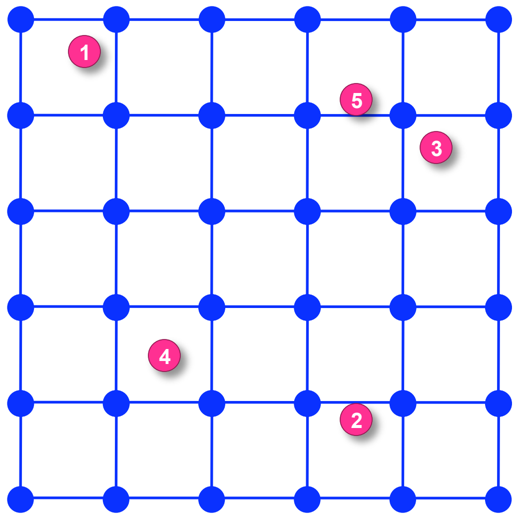
- Nominally, **push** is embarrassingly parallel, and the technologies for **solving PDEs** on structured grids are well developed.
- Unfortunately efficient HW/SW support for **gather**/**scatter** operations is still a developing area of research
  (single thread/multicore/multinode)



Although particles and grid points appear linearly in memory,

PIC Challenges

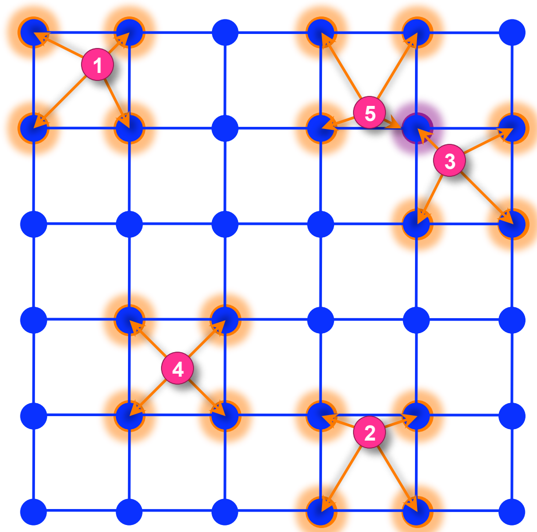
- Nominally, **push** is embarrassingly parallel, and the technologies for **solving PDEs** on structured grids are well developed.
- Unfortunately efficient HW/SW support for **gather**/**scatter** operations is still a developing area of research (single thread/multicore/multinode)



When the particles' spatial coordinates are mapped to the grid, there is no correlation

PIC Challenges

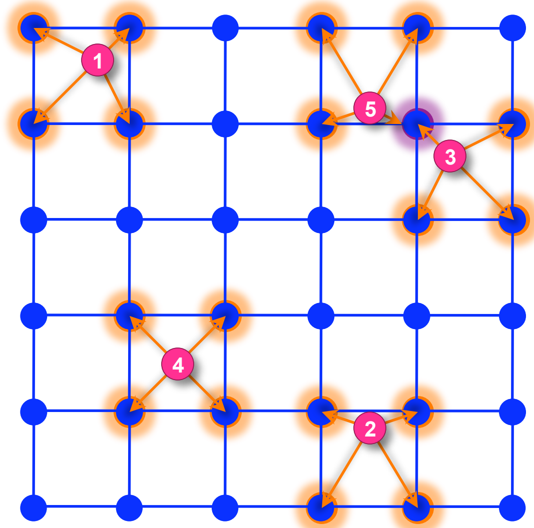
- Nominally, **push** is embarrassingly parallel, and the technologies for **solving PDEs** on structured grids are well developed.
- Unfortunately efficient HW/SW support for **gather**/**scatter** operations is still a developing area of research (single thread/multicore/multinode)



Thus particles will update random locations in the grid, or conversely, grid points are updated by random particles

PIC Challenges

- Nominally, **push** is embarrassingly parallel, and the technologies for **solving PDEs** on structured grids are well developed.
- Unfortunately efficient HW/SW support for **gather**/**scatter** operations is still a developing area of research (single thread/multicore/multinode)

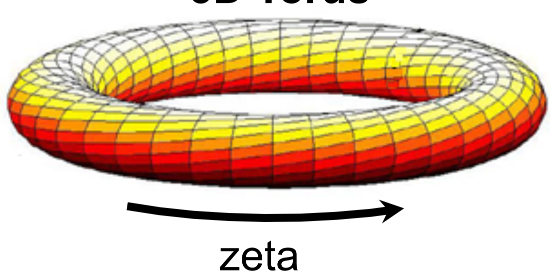


Moreover, the load-store nature of modern microprocessors demands the operations be serialized (load-increment-store)

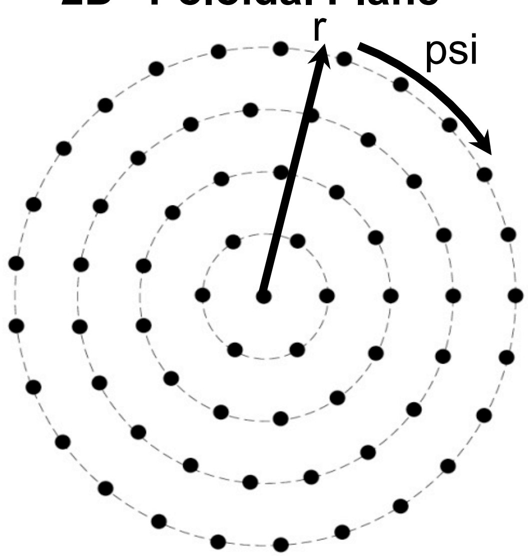
Sequential GTC Challenges

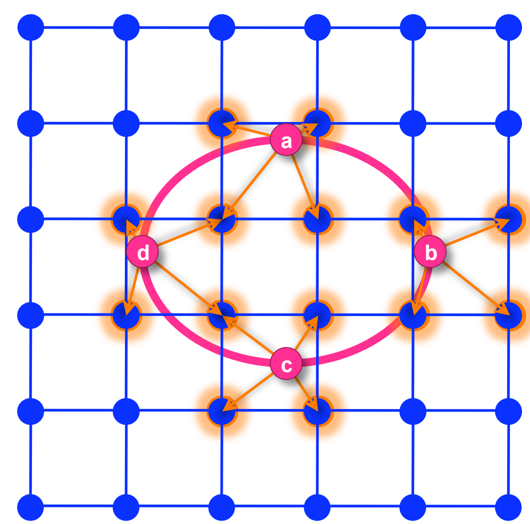
- As if this weren't enough, GTC further complicates matters as
  - the grid is a 3D torus
  - points in psi are spatially uniform
  - particles are non-circular rings (approximated by 4 points), and
- Luckily rings only exist in a poloidal plane, but the radius of the ring can grow to ~6% of the poloidal radius.

3D Torus



2D "Poloidal Plane"

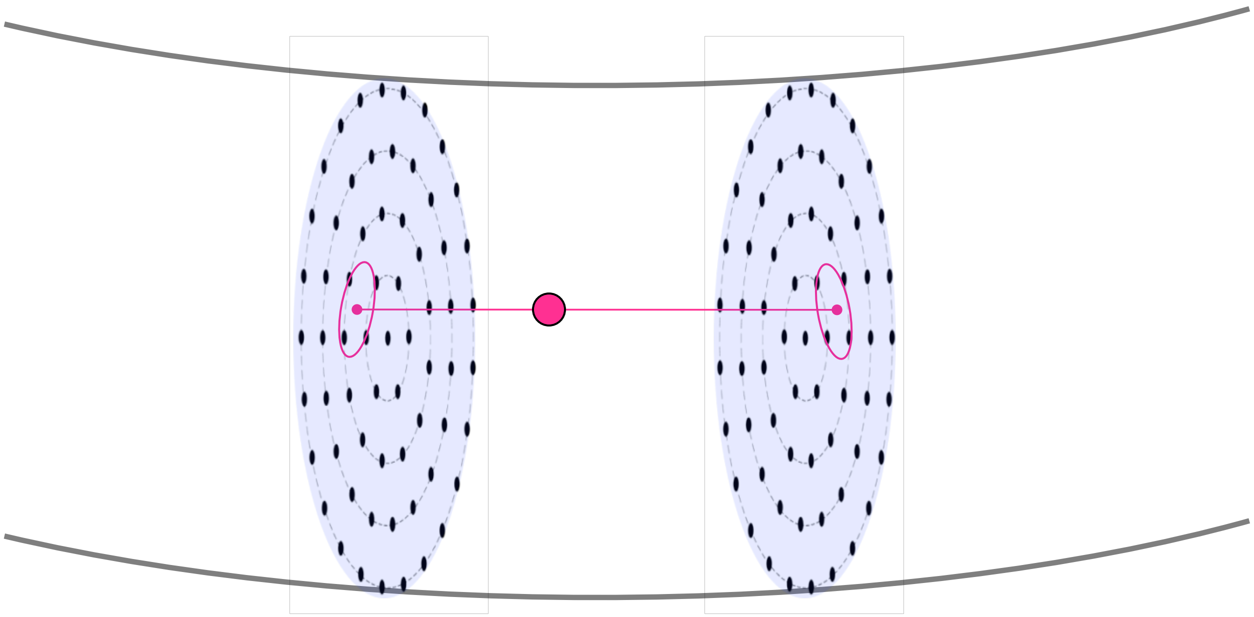




mgrid = total number of points

3D Issues

- Remember, GTC is a 3D code
- As such, particles are sandwiched between two poloidal planes
- and scatter their charge to as many 16 points in each plane



Multicore GTC Challenges

(memory-level parallelism)

- Although out-order processors reorder instructions to exploit instruction-level parallelism, they resolve the data dependencies in hardware.
- If the sequence $\texttt{load}_1$, $\texttt{add}_1$, $\texttt{store}_1$, $\texttt{load}_2$, $\texttt{add}_2$, $\texttt{store}_2$ runs on one core, hardware can reorder it into :

  $\texttt{load}_1$, $\texttt{load}_2$, $\texttt{add}_1$, $\texttt{add}_2$, $\texttt{store}_1$, $\texttt{store}_2$ assuming addresses 1 and 2 are different.

- However, if sequences 1 and 2 run on different cores,

  this benefit is lost and the **programmer must manage the data dependency in software.**

Multicore GTC Challenges

(Data locality)

- Multicore SMPs have complex memory hierarchies.
- Although the caches are coherent, data migration between caches is slow and should be avoided.
- Moreover each core has a limited cache size. If random access working set exceeds the size, performance will be diminished.

- **Given the random access nature (scatter/gather) of GTC, how do we partition the problem to mitigate these limitations?**

Multicore Solutions

Focus: Charge Deposition

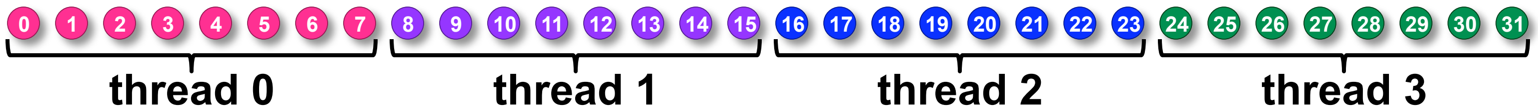
- The charge deposition phase is the most complex as it requires solving the **data dependency challenges** in addition to the **data locality challenges** found in the gather phase

- As such, this talk will focus on optimizing charge deposition (**scatter**) phase for shared memory (**threaded**) multicore environments

- In the MPI version of GTC, the torus is first partitioned in zeta (around the torus) into "**poloidal planes**" (1 per process)
- Unfortunately the physics limits this decomposition to about 64-256 processes.
- Currently, additional processes work collaboratively on each poloidal plane reducing together at the end of scatter.
- We explore threading rather than MPI parallelization of each plane

Managing Data Locality
(Particle Decomposition)

- Throughout this work we use a simple 1D decomposition of the particle array:



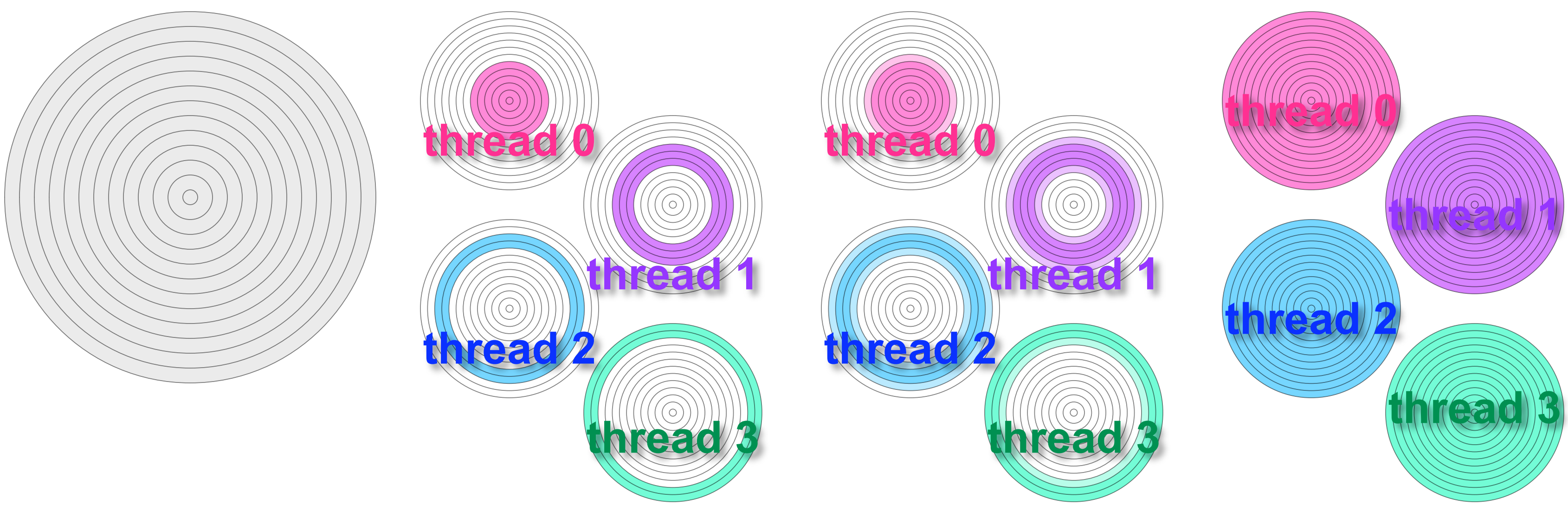
- Particles are initially sorted by their radial coordinate

Managing Data Locality

(Grid Decomposition)

- We explored four different strategies for managing data locality and total memory usage.
- In all cases there is a shared grid.
- It may be augmented with (private) per-thread copies
- **update thread's copy of grid if possible, else update shared grid.**

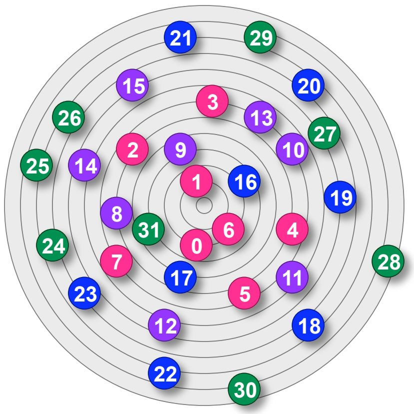
shared grid	partitioned grid	partitioned grid (w/ghosts)	replicated grids



no replication	*O(1) replicas*	*O(1) replicas*	*O(P) replicas*
full overlap	*no overlap*	*overlap by rmax/16*	*no overlap*

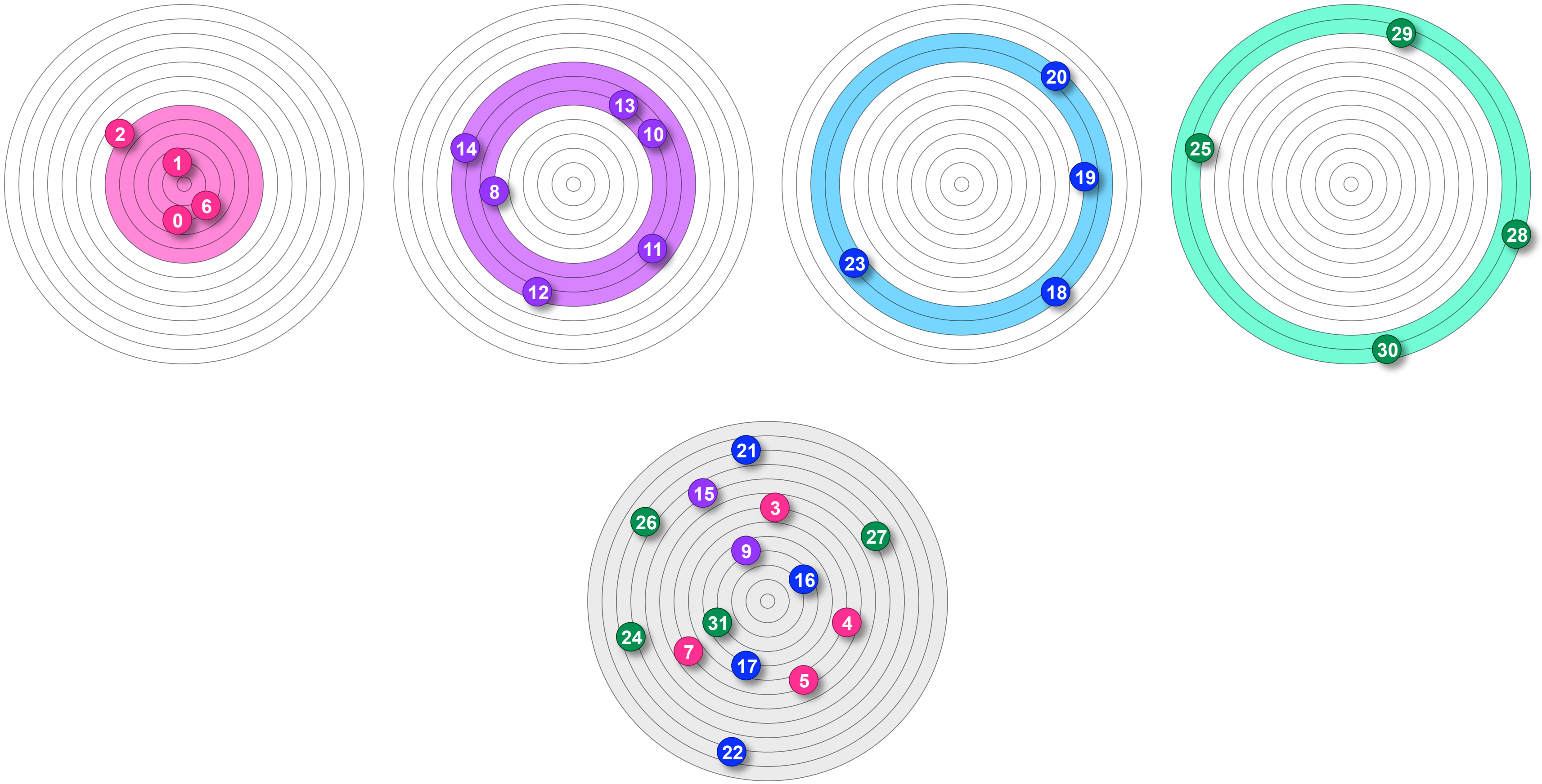
Example #1

- Consider an initial distribution of particles on the shared grid.
- As the grid is a single shared data structure, all updates require some form of synchronization



Example #2

- When using the partitioned grid, we see that some accesses go to the private partitions, but others go to the shared grid (where they will need some form of synchronization)



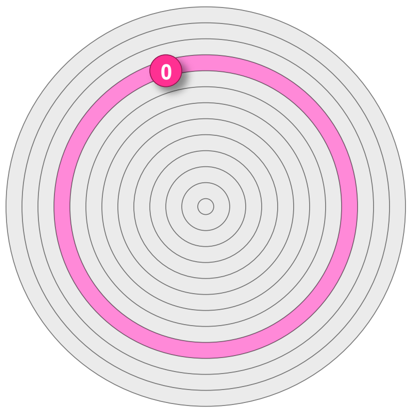
Managing Data Hazards
(Synchronization)

- We explored five different synchronization strategies:
  - **coarse** lock all r & zeta for a given psi (2 rings)
  - **medium** lock all zeta for a given r & psi (2 grid points)
  - **fine** lock one grid point at a time
  - **atomic** 64b FP atomic increment via CAS (required some assembly/intrinsics)
  - **none** one barrier at the end of the scatter phase

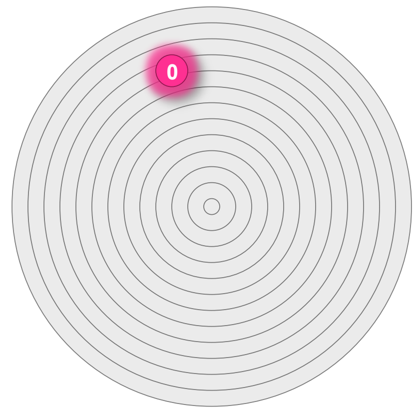
- Remember the coarser the lock
  - the more overhead is amortized
  - the less the available concurrency

Visualizing Locking Granularity

Coarse



Medium / Fine



note, medium locking locks the same point in both sandwiching poloidal planes where fine locks the point in one plane at a time.

Locality × Synchronization

- There are 20 combinations of grid decomposition and data synchronization.
- However,
  - 3 won't guarantee correct results (lack of required synchronization)
  - 4 are nonsensical (synchronization when none is required)
- As such, only 15 needed to be implemented

decomposition \ *synchronization*	coarse	medium	fine	atomic	none
shared	✔	✔	✔	✔	incorrect
partitioned	✔	✔	✔	✔	incorrect
partitioned (w/ghosts)	✔	✔	✔	✔	incorrect
replicated	nonsensical	nonsensical	nonsensical	nonsensical	✔

Miscellaneous

- In addition, we implemented a number of sequential optimizations including:
  - Structure-of-arrays data layout
  - explicit SIMDization (via intrinsics)
  - Data alignment
  - loop fusion
  - process pinning

Results

Experimental Setup

- We examined charge deposition performance on three multicore SMPs:
  - Dual-socket, quad-core, hyperthreaded 2.66GHz Intel Nehalem
  - Dual-socket, octal-core, 8-way VMT 1.16GHz Sun Niagara2
  - Dual-socket, quad-core 2.3GHz Barcelona (in SC'09 paper)

Niagara is a proxy for the TLP of tomorrow's manycore machines

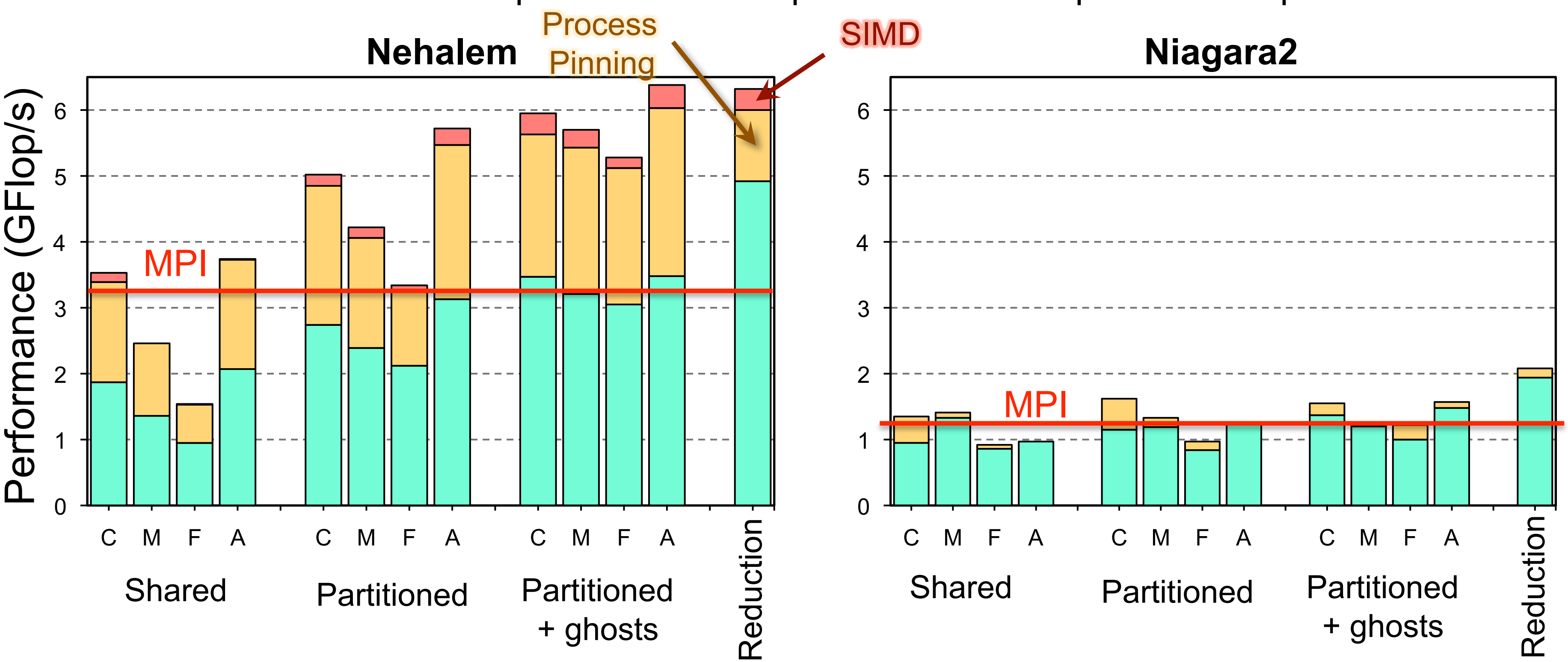
- Problems are based on:
  - grid size (mgrid) 32K, 151K, 600K, 2.4M
  - particles per grid point (micell) 2, 5, 10, 20, 50, 100

- Generally, we examine the performance of the **threaded variant** as a function of optimization or problem size
- Additionally, we compare against the conventional wisdom MPI version.

Performance

as a function of grid decomposition and synchronization

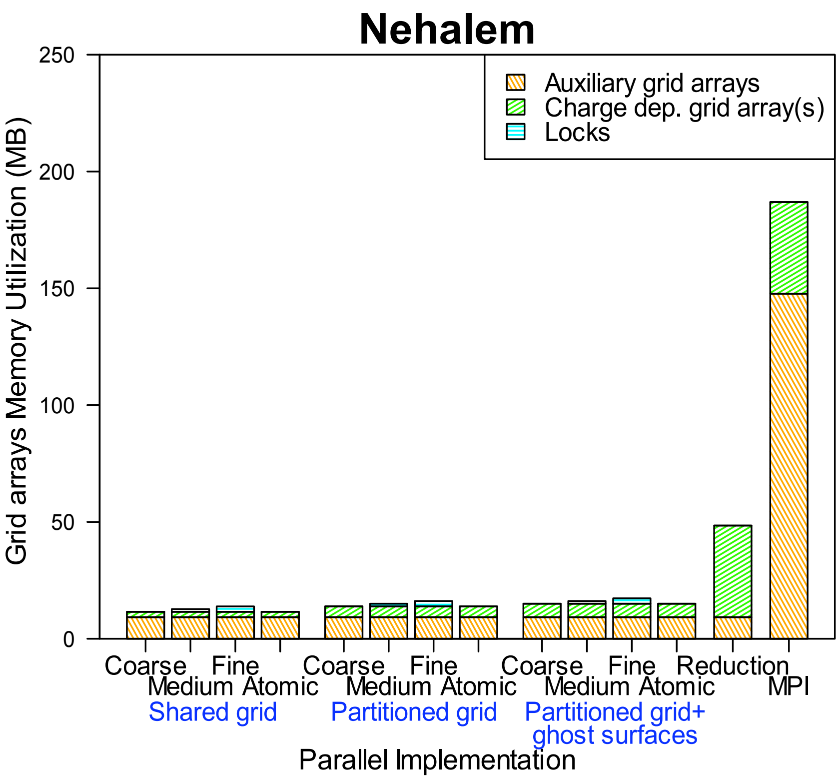
- Consider problem with **150K grid points and 5 particles/point**
  - As locks become increasingly finer, the overhead of pthreads becomes an impediment, but Atomic operations reduce the overhead dramatically
  - Nehalem did very well with the partially overlapping decomposition
  - Performance is much better than MPI
  - Partitioned decomposition attained performance comparable to replication

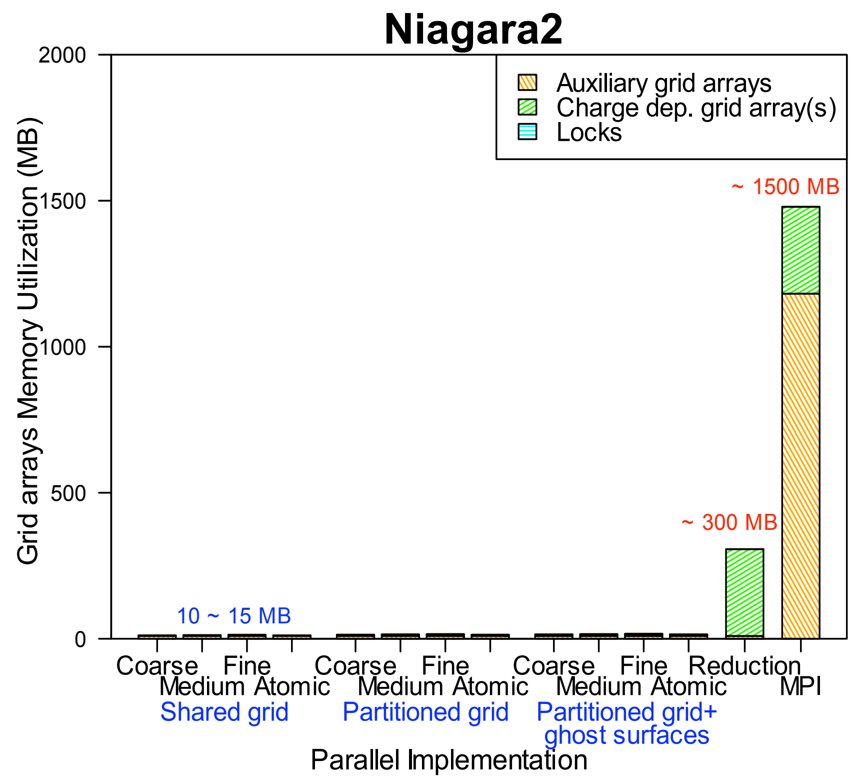


Memory Usage

as a function of grid decomposition and synchronization

- Although the threaded performance was comparable to either the MPI variant or the naïve replication approach, **the memory usage was dramatically improved**
- ~12x on Nehalem, and ~100x on Niagara



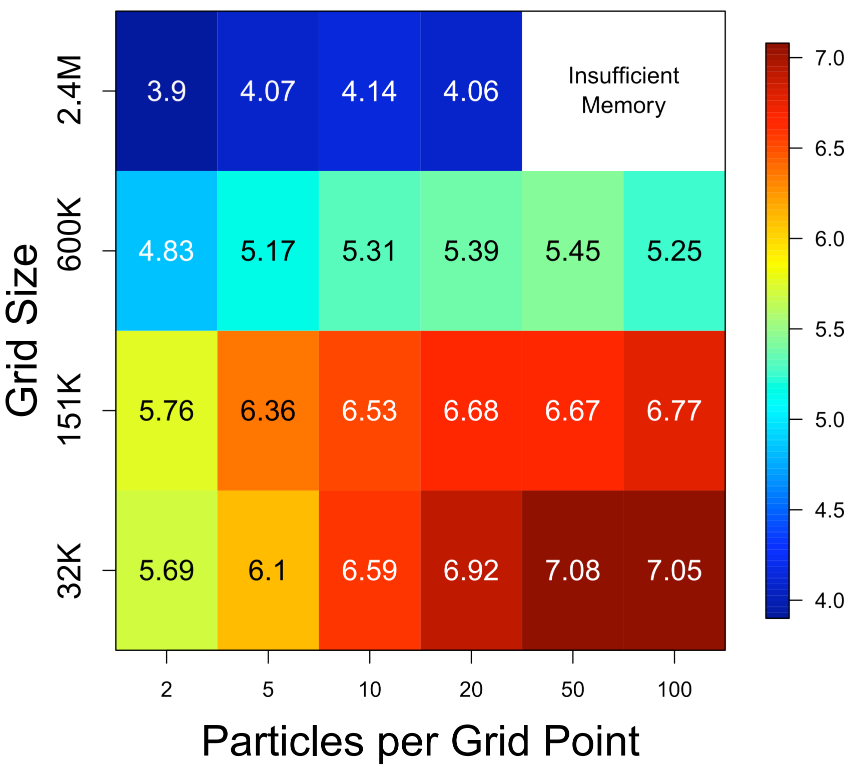


Performance

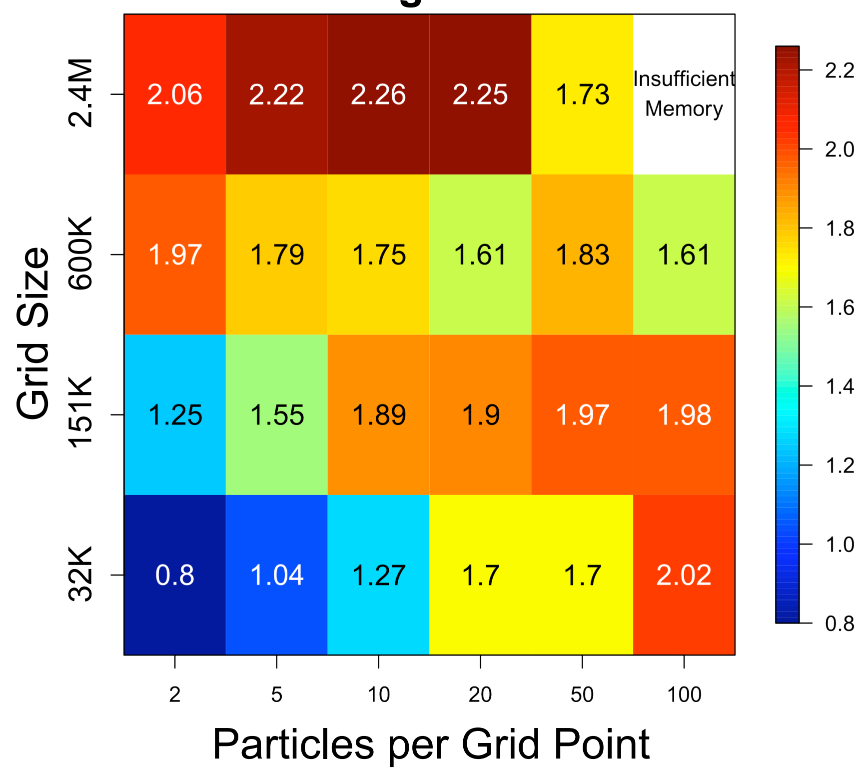
as a function of problem configuration

- For the memory-efficient implementations (i.e. no replication)
- Performance generally increases with increasing density (higher locality)
- Performance generally decreases with increasing grid size (larger working set)
- On Niagara, problems need to be large enough to avoid contention among the 128 threads

Nehalem



Niagara2



Summary & Discussion

Summary

- GTC (and PIC in general) exhibit a number of challenges to locality, parallelism, and synchronization.
  - **Message passing implementations won't deliver the efficiency**
  - **Managing data dependencies is a nightmare for shared memory**

- We've shown that threading the charge deposition kernel can deliver **roughly twice the performance** of the MPI implementation

- Moreover, we've shown that we can be **memory-efficient** (grid partitioning with synchronization) without sacrificing performance.

Further Reading

K. Madduri, S. Williams, S. Ethier, L. Oliker, J. Shalf, E. Strohmaier, K. Yelick, *"Memory-Efficient Optimization of Gyrokinetic Particle-to-Grid Interpolation for Multicore Processors"*, Supercomputing (SC), 2009.

Acknowledgements

- Research supported by DOE Office of Science under contract number DE-AC02-05CH11231
- Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227)

Questions?