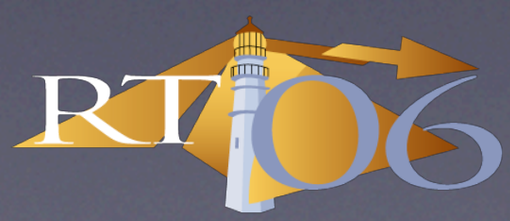
An Evaluation of Parallel Grid Construction for Ray Tracing Dynamic Scenes

Thiago Ize, Ingo Wald, Chelsea Robertson, Steven G. Parker
SCI Institute, University of Utah

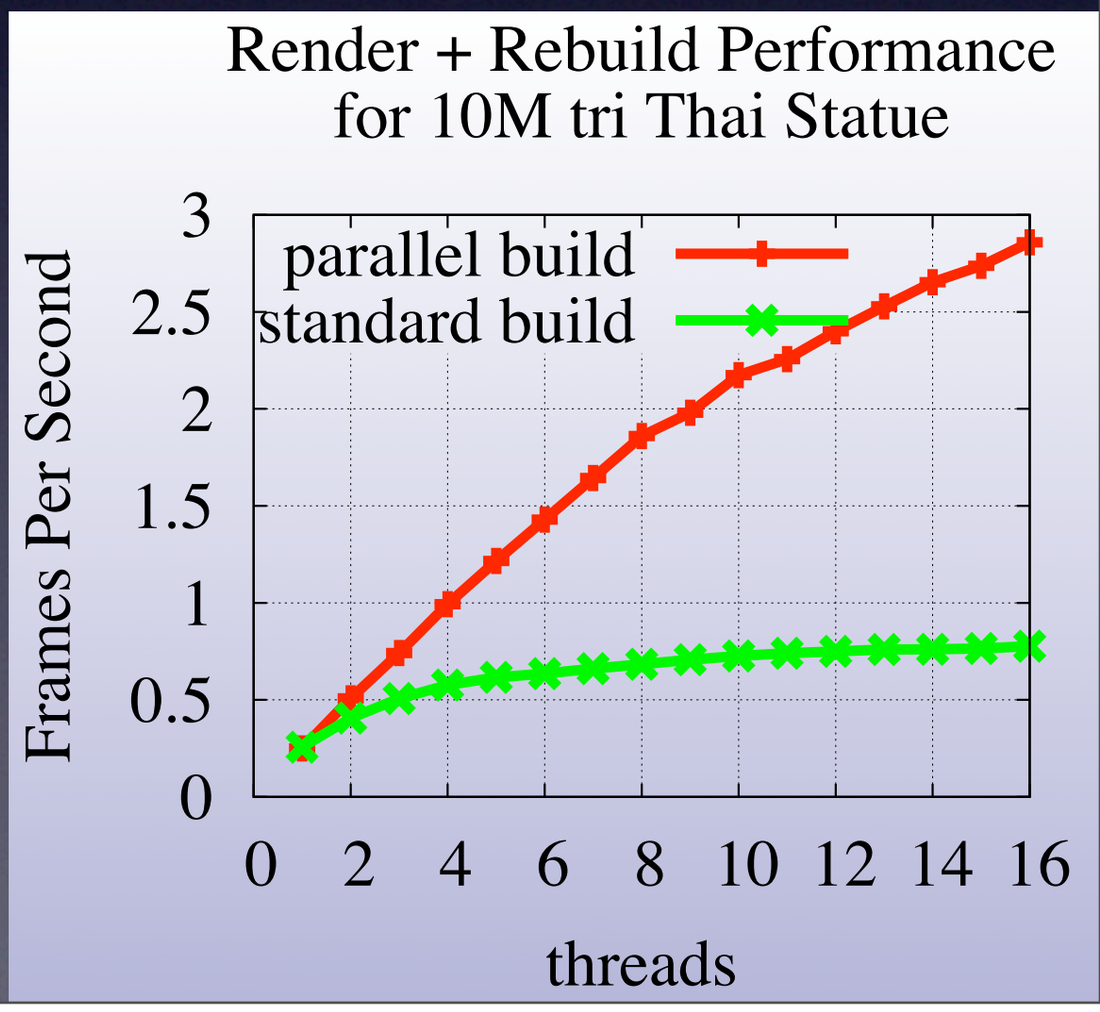
THE UNIVERSITY OF UTAH



SCI INSTITUTE

Motivation

- Coherent grid traversal is fast
- O(n) grid rebuild time is fast. But...
  - "Large" scenes take too long to build
  - Parallelize build
- Multi-core trend



CGT is competitive with other acceleration structures for traversal and can be rebuilt MUCH faster.
Note: the issue with large models exists for any acceleration structure with any type of update method. Only difference is what the definition of "large" is.

Future ray tracing machine: parallel computer

- Multi-core trend: future consumer computers will have many cores (on one or more processor chips)
- Ray tracing hardware, like GPUs, will likely have many cores
- Need to parallelize the build for these machines

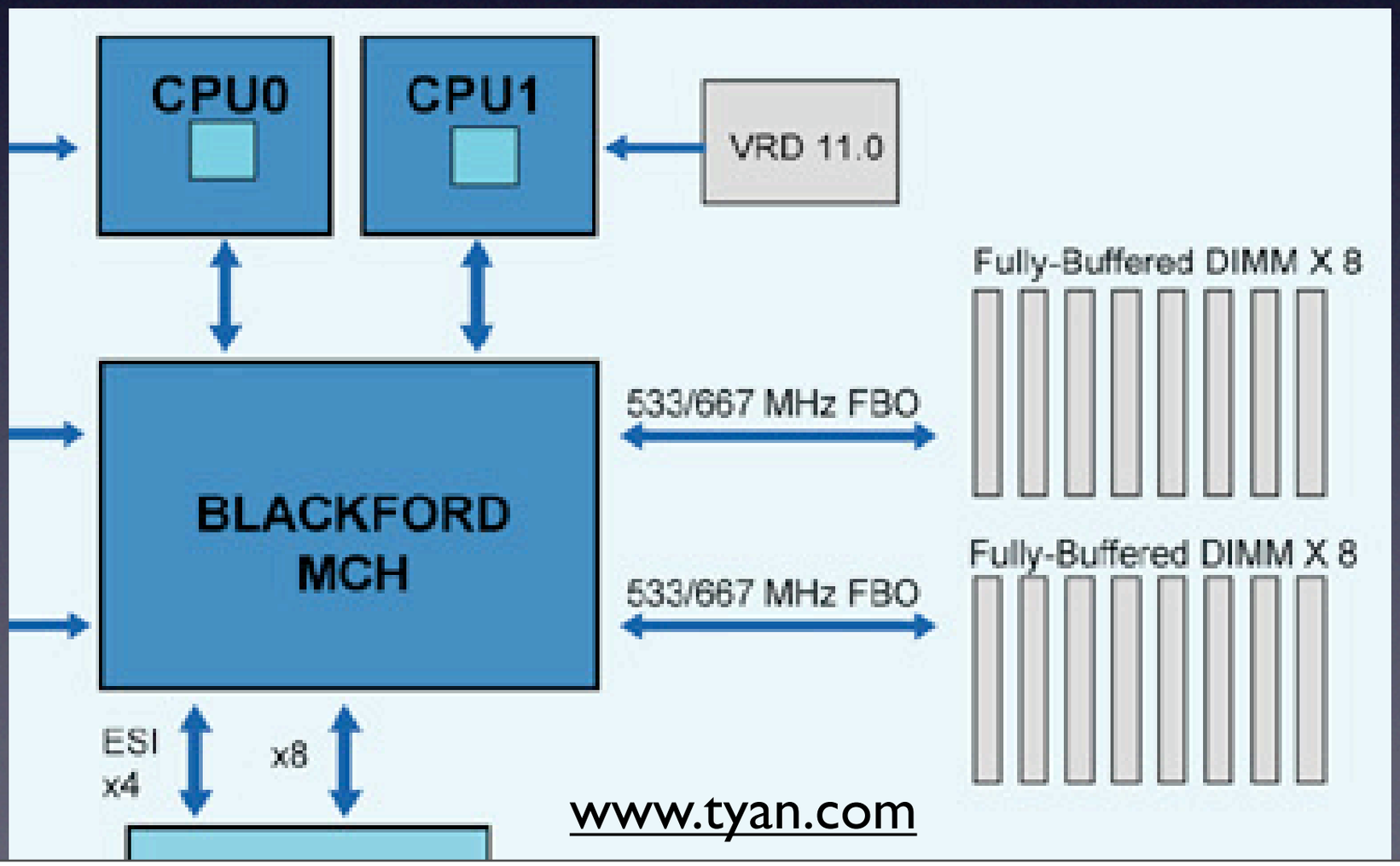
Parallel computers

- 3 main types:
  - SMP: Symmetric Multi-Processing
  - ccNUMA: cache coherent Nonuniform Memory Access
  - Distributed memory: computer clusters

ccNUMA we will usually refer as just NUMA

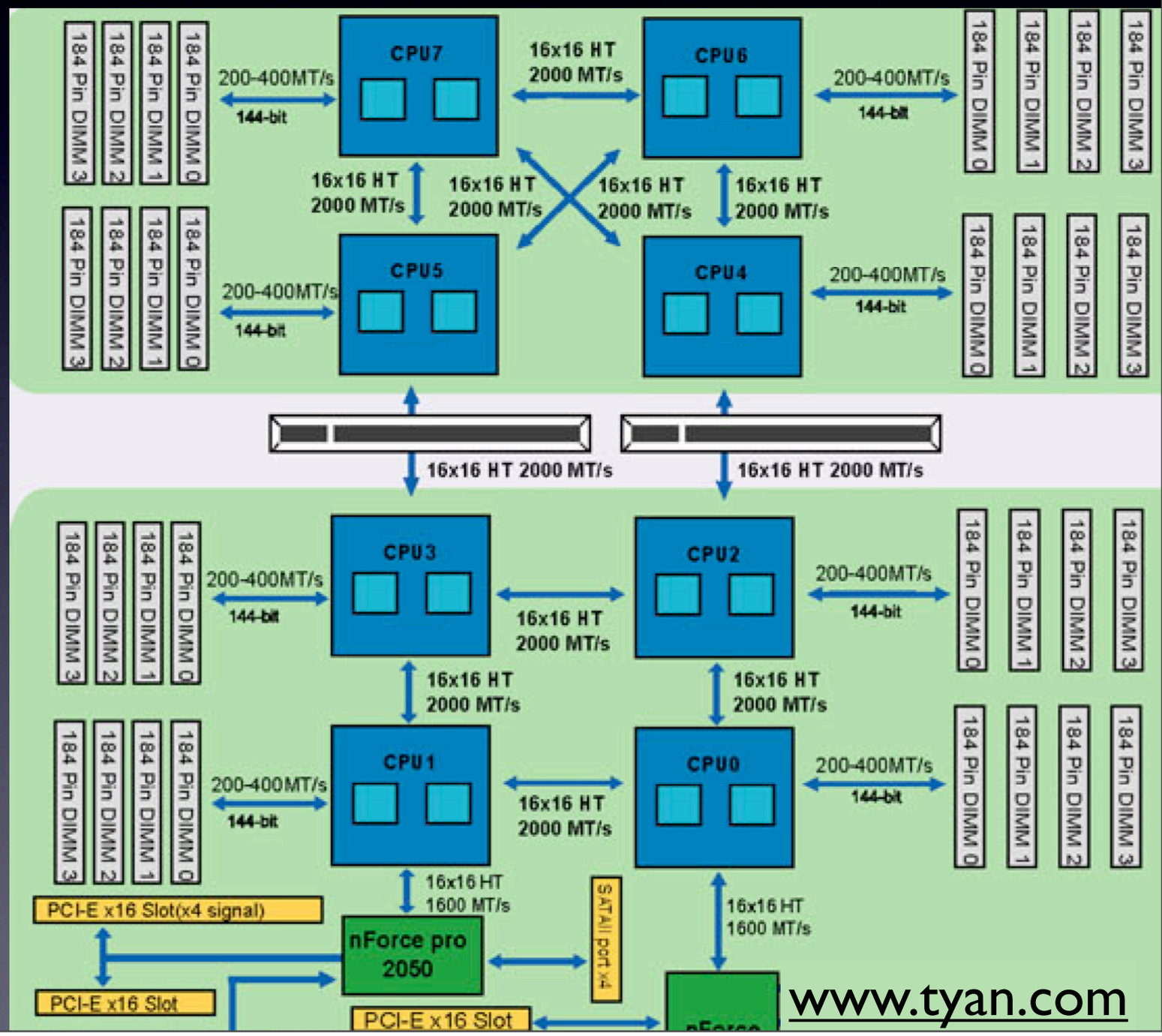
Parallel computers: SMP

- All processors share a memory controller and memory bus
- Does not scale: memory bottleneck



Parallel computers: ccNUMA

- Partition system into nodes
- Each node has its own memory subsystem (node is SMP)
- Scales as long as most memory operations reside in the thread's node



Opterons have integrated memory controller. Each CPU is its own node.
The diagram is of the computer we use for our experiments.

note: image from tyan, but I modified it to make it dual core.

Parallel computers: distributed memory

- Computer clusters
- Memory latency across cluster nodes is slow and parallel grid build is memory intensive
- Not likely to appear on consumer level computers
- We do not target this architecture

Photo Credit: NASA Ames Research Center/Tom Trower

Future ray tracing machine

- Target machine is likely a combination of NUMA and SMP architectures
- Chip(s) might contain several nodes (NUMA)
- Nodes each contain several cores (SMP)
- Ideal algorithm takes NUMA and SMP requirements into account

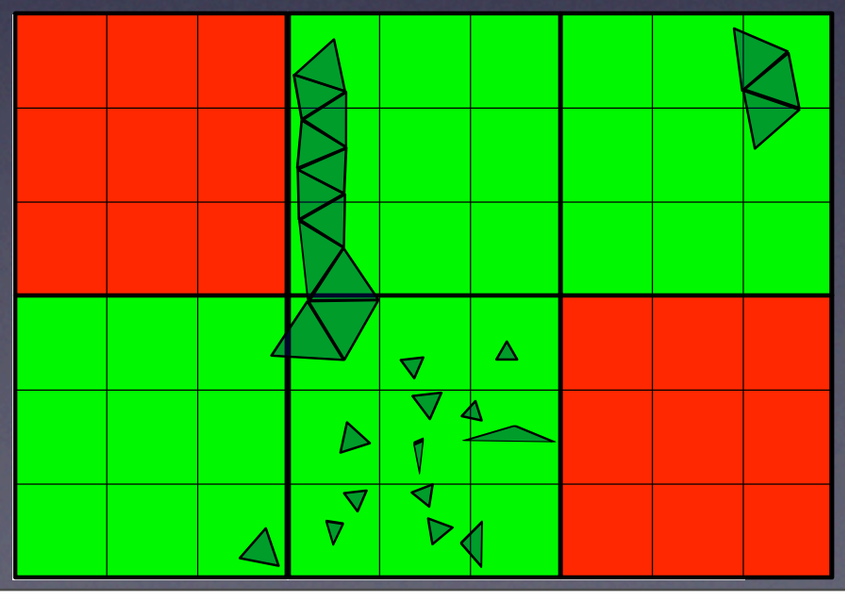
Serial grid rebuild

- Clear the cells of previous triangles
- Insert triangles into grid cells
- Build macro cells

In order to parallelize the grid rebuild, we first have to understand the serial build we use
note: this is the same as the CGT siggraph paper

Serial grid rebuild: clearing

- Naive: Loop through all cells
- Problem: Most cells are already empty causing wasted effort
- Solution: Loop through all macro cells, clearing only cells that reside in a full macro cell



green cells we had to check. Red cells we skipped

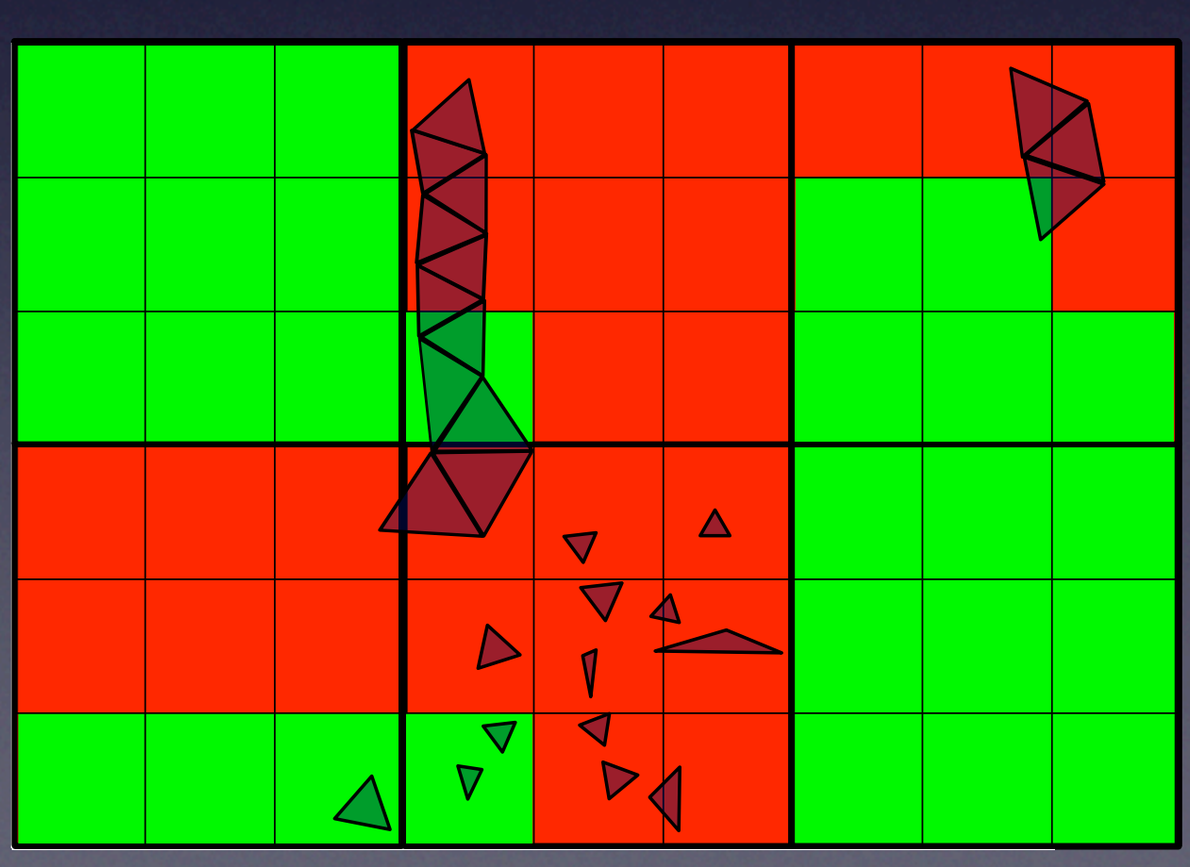
Serial grid rebuild: triangle insertion

- Insert triangle into all cells it overlaps
- Use triangle bounding box for overlap test
  - Results in much faster build
- Traversal stays roughly the same due to mailboxes

For conference (282k):
triangle references: 1.24M -> 0.83M
rendering time: 339ms -> 330ms
serial rebuild time: 82ms -> 245ms (163ms increase)

Serial grid rebuild: macro cell build

- Same idea as clearing
- For each macro cell, look through all grid cells until a nonempty grid cell is found



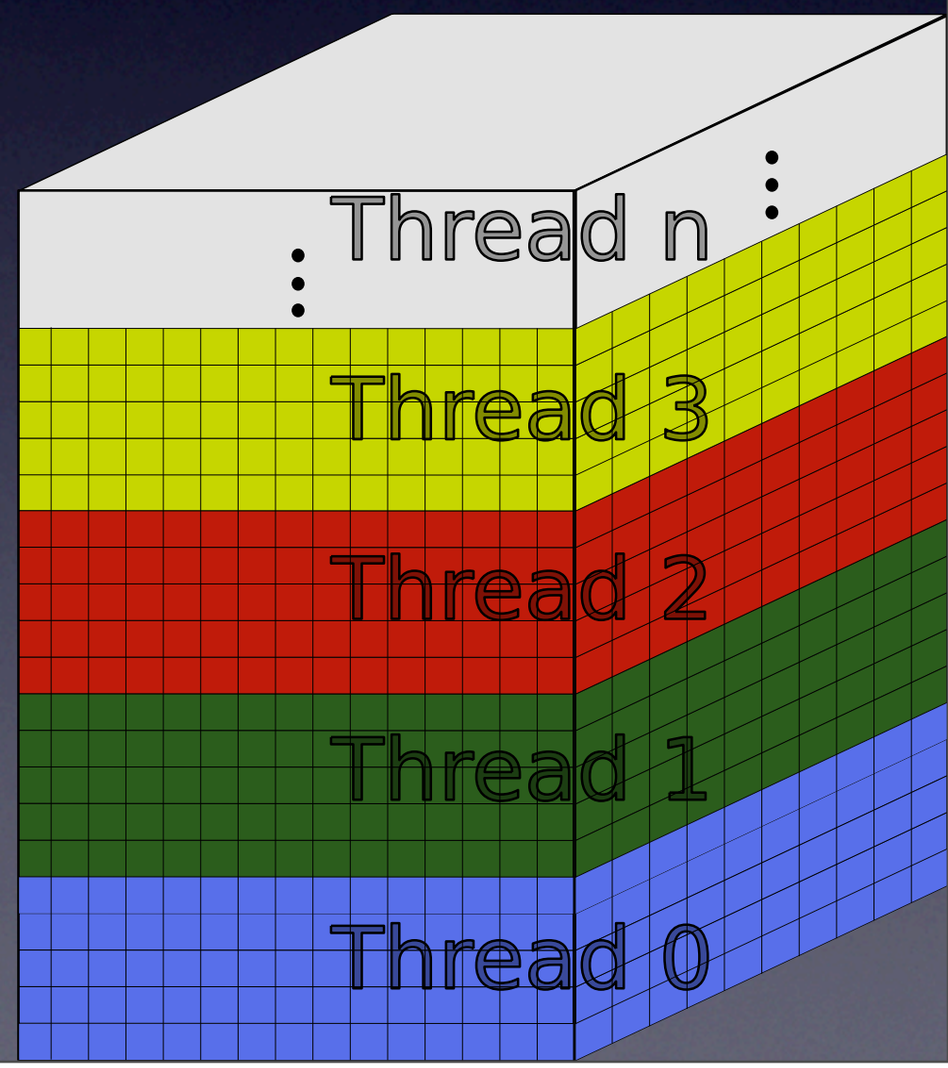
green cells we had to check. Red cells we skipped

Parallelizing the build

- Scale to many threads
- Keep memory bandwidth between nodes low
- Must not impact traversal performance

Parallel clear and macro cell build

- Have each thread handle a continuous section of macro cells
- Doesn't load balance well
- Overhead for better load balancing is too high



It's quite possible for certain scenes better load balancing would end up better

Parallel triangle insertion

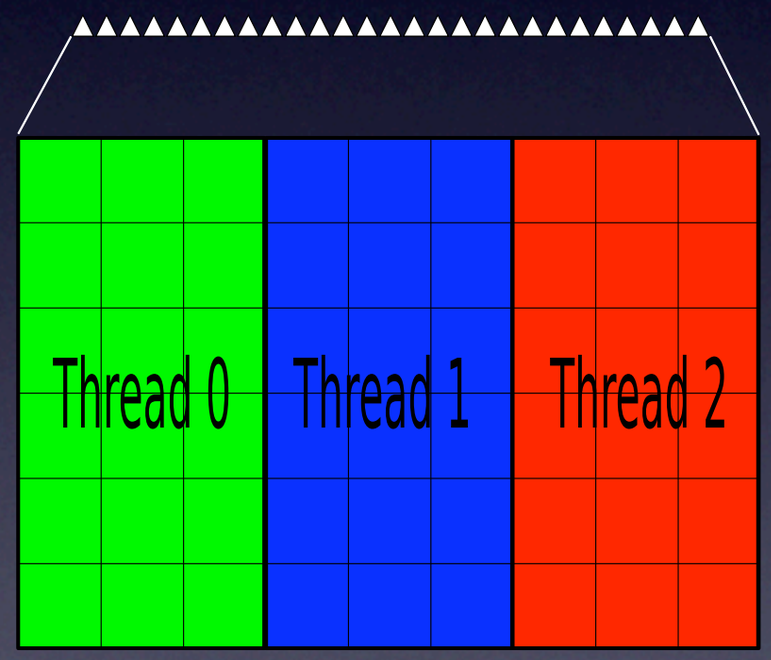
- Analogous to rasterization: instead of pixels (2D grid) use cells (3D grid)
- Extend ideas from existing parallel raster-based rendering algorithms
- Parallel rendering algorithms classified into 3 categories:
  - sort-first
  - sort-last
  - sort-middle

Sort-first: Each thread is given a subset of the grid which it must insert triangles into.
Sort-last: Each thread is given a subset of triangles to insert into the grid.
Sort-middle: Each thread is given a subset of triangles and a subset of the grid. It then routes its triangles to the thread handling that part of the grid.

Sort-first

- Benefits: does not require synchronization on cells or triangles
- Issues:
  - Each thread must look at ALL triangles
  - Build remains O(number of triangles) despite parallelization

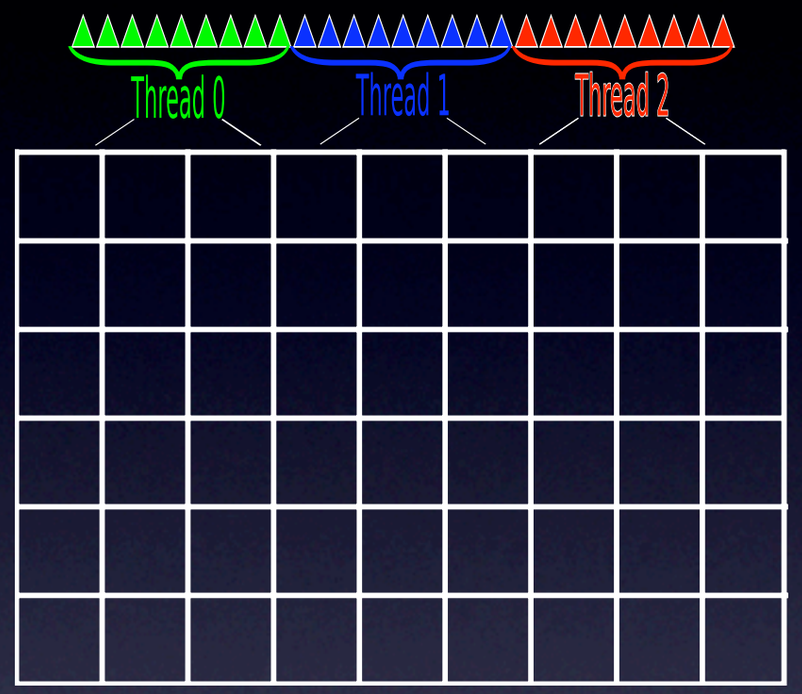
  ➡We do not use this



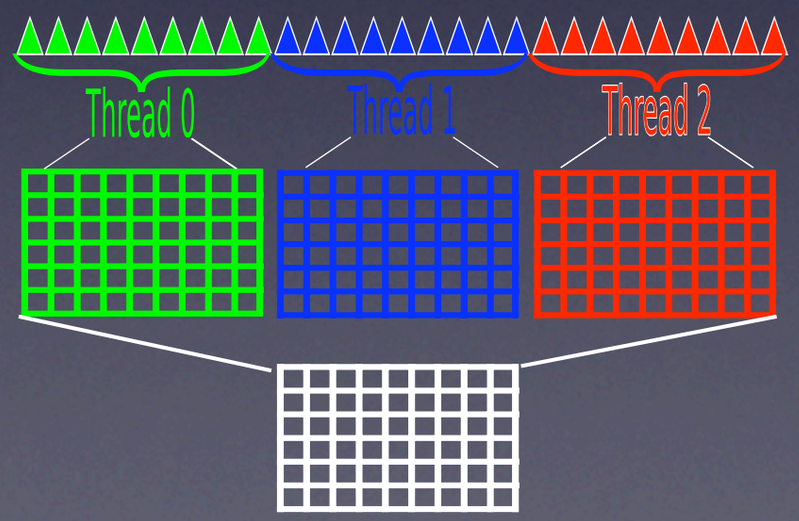
Sort-last

Two approaches:

1. Threads share a single global grid
   - Requires expensive synchronization on cell updates



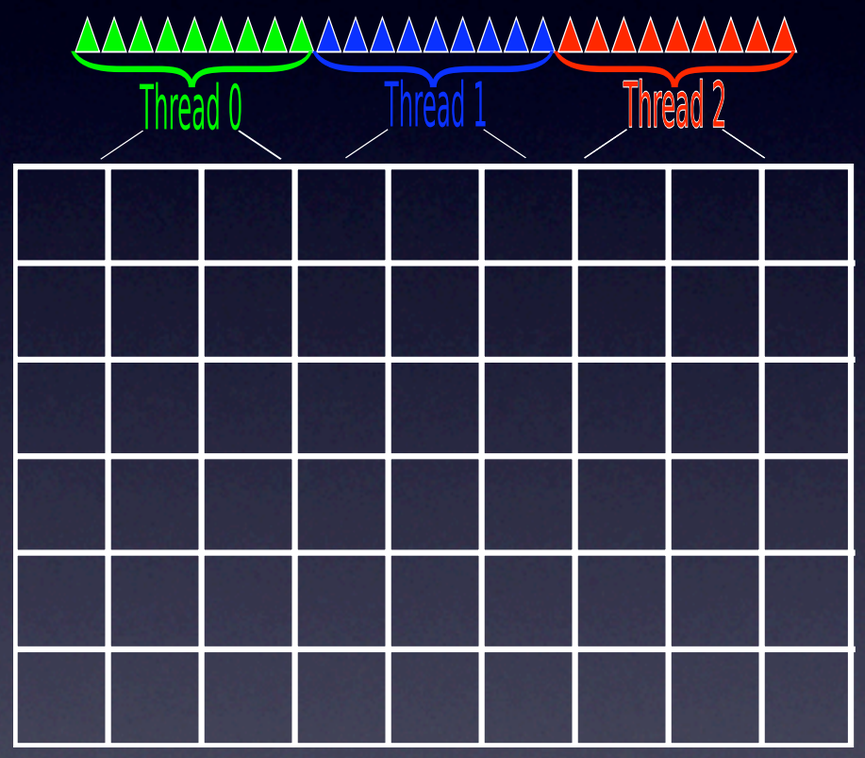
2. Each thread has a copy of its own grid
   - Combine grids later during build
   - Keep grids separate and traverse them all



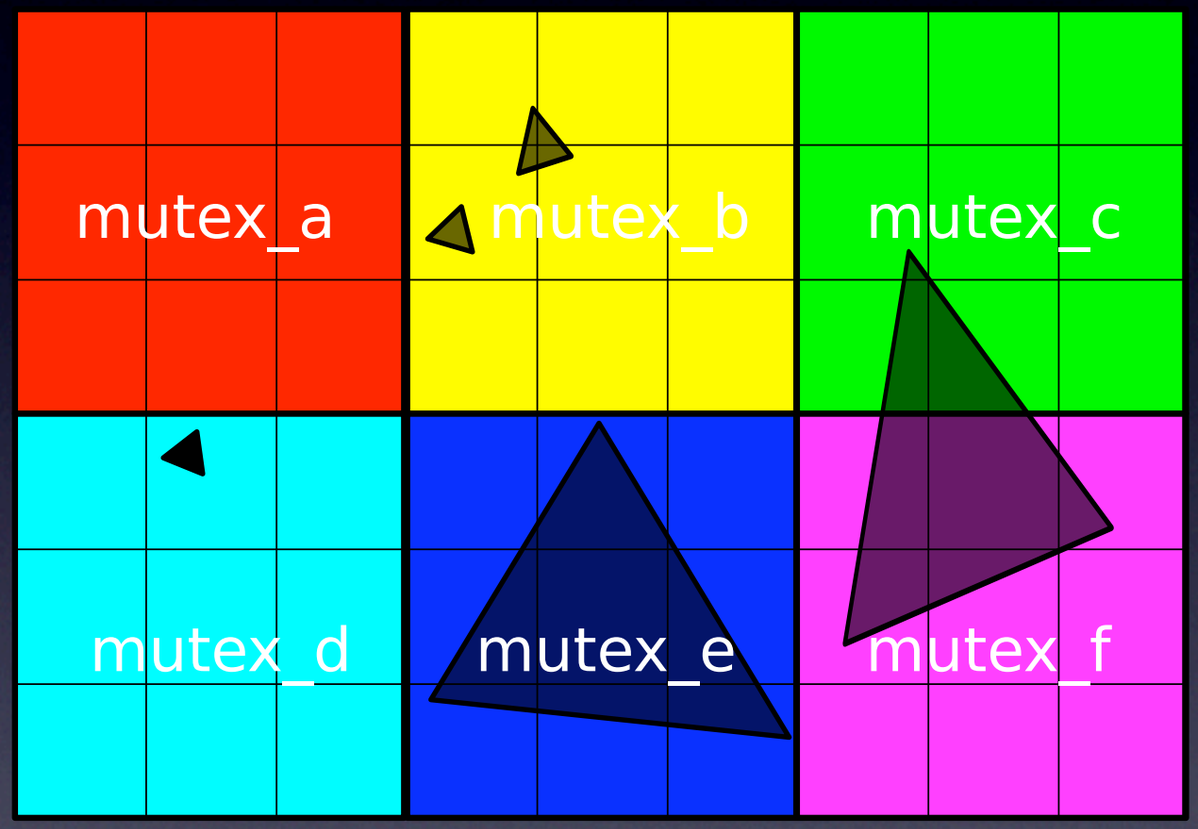
Sort-last: global grid

Threads share a single global grid

- Single mutex for the entire grid
  - Degenerates into serial build
  - Mutex overhead from resolving conflicts actually causes build to get many times slower
- Mutex for every cell
  - Almost no contention - “fast” mutex
  - Fast mutex overhead still exists though
  - Memory overhead



Sort-last: global grid mutex pool

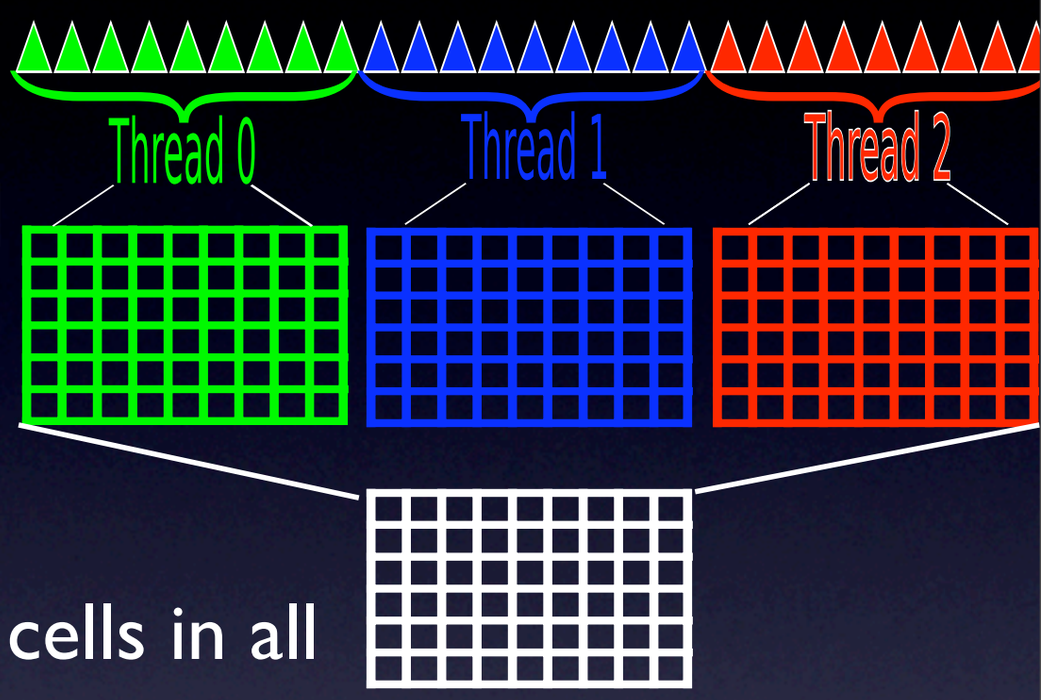


Threads share a single global grid

- Much less memory used
- Contention still low
- Mutex per macro cell
- Fewer mutex locks: if triangle overlaps several cells, lock mutex at start and hold it until done inserting to cells or until in another macro cell

could do many things, such as interleaving mutex across cells.

Sort-last: separate grids



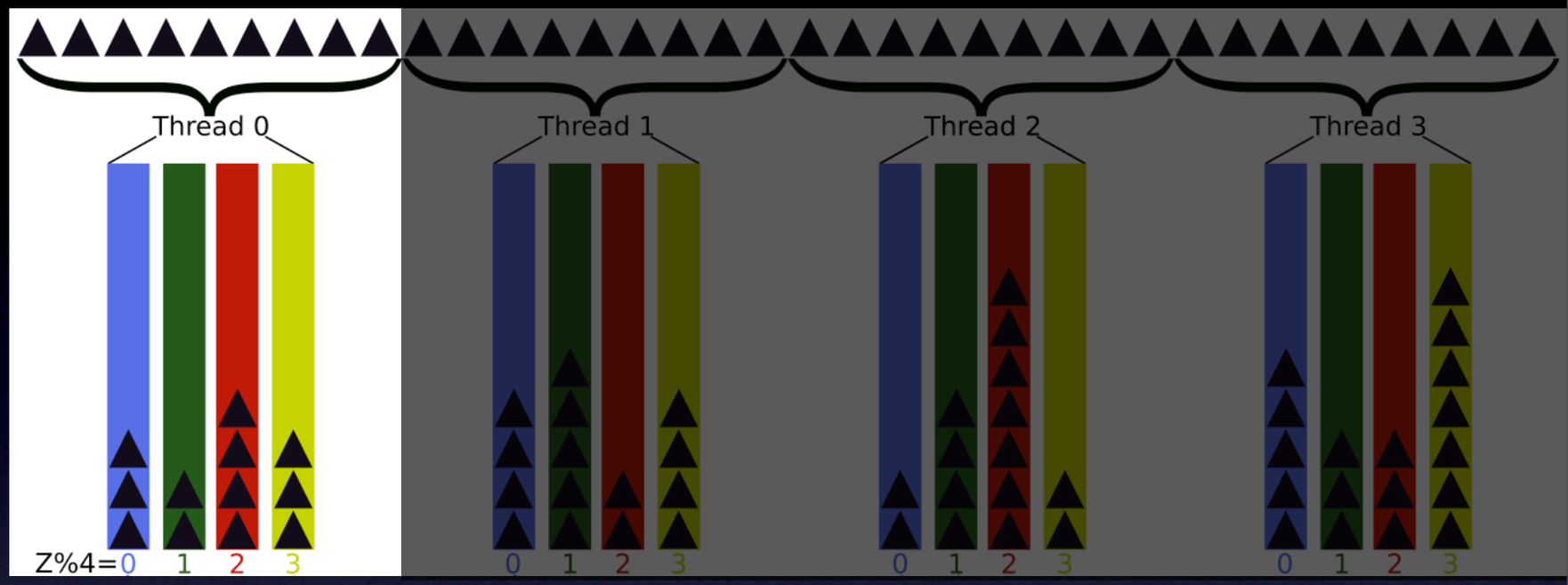
Each thread has a copy of its own grid

- Merge grids after triangle insertion
  - Can do without synchronization
  - High memory bandwidth — look at all cells in all grids. Copy triangles into main grid
  - Must clear all grids — clearing no longer scales
- Don’t merge and instead check grids during traversal
  - Hurts traversal performance
  - Must perform grid clear and macro cell build for all grids — no longer scalable

Sort-middle

- First step same as in sort-last
- Threads get a subset of triangles

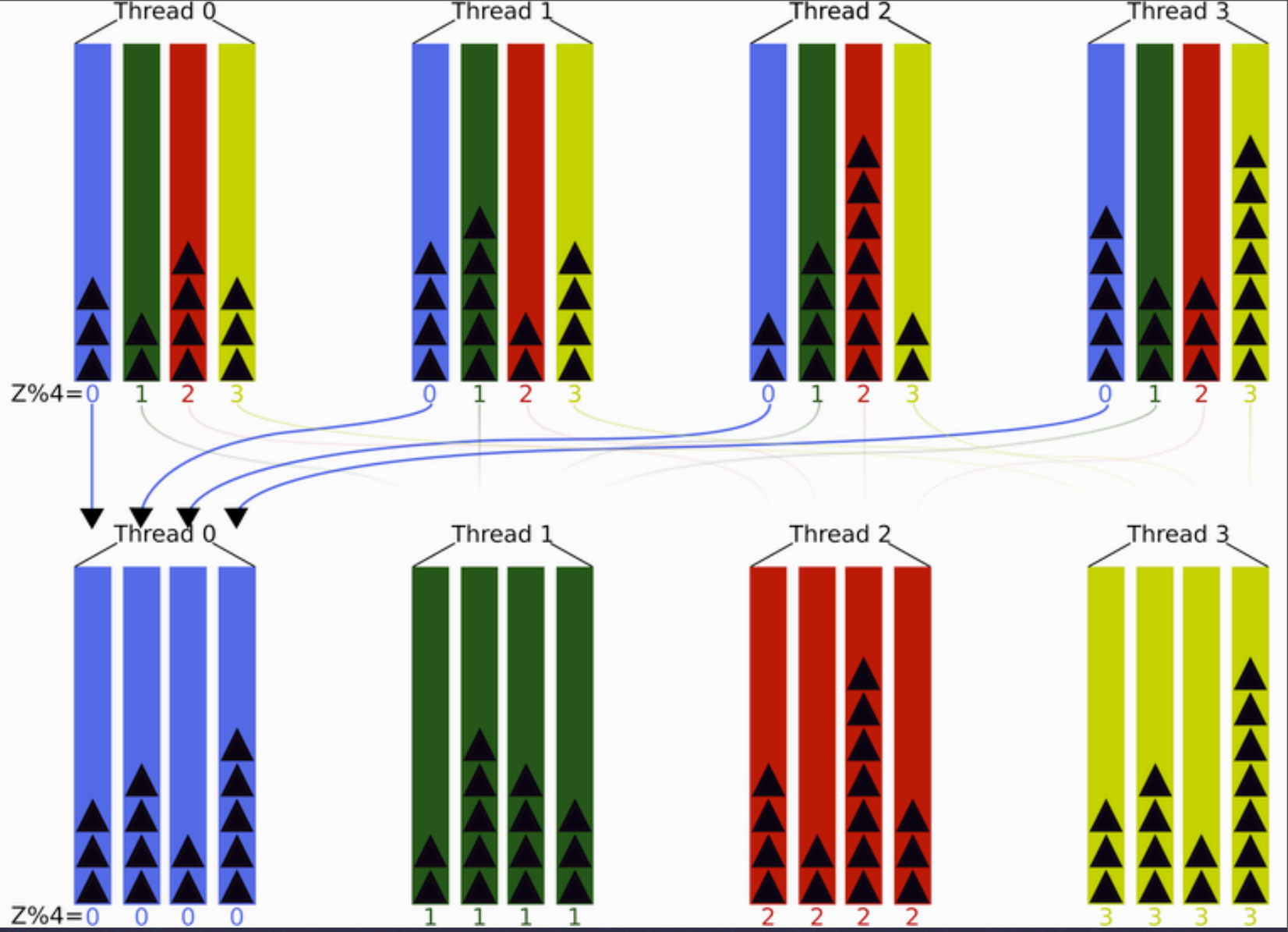
Sort-middle



- Thread performs a coarse bucket sort on its triangles
- We use (z % num_threads) where z is major index of overlapped cells
- Triangle might exist in multiple buckets

Could also do ordering on macro cells, but we did this
This load balances well as long as triangle sizes are randomly distributed across triangle list.
— Big triangles overlap more cells

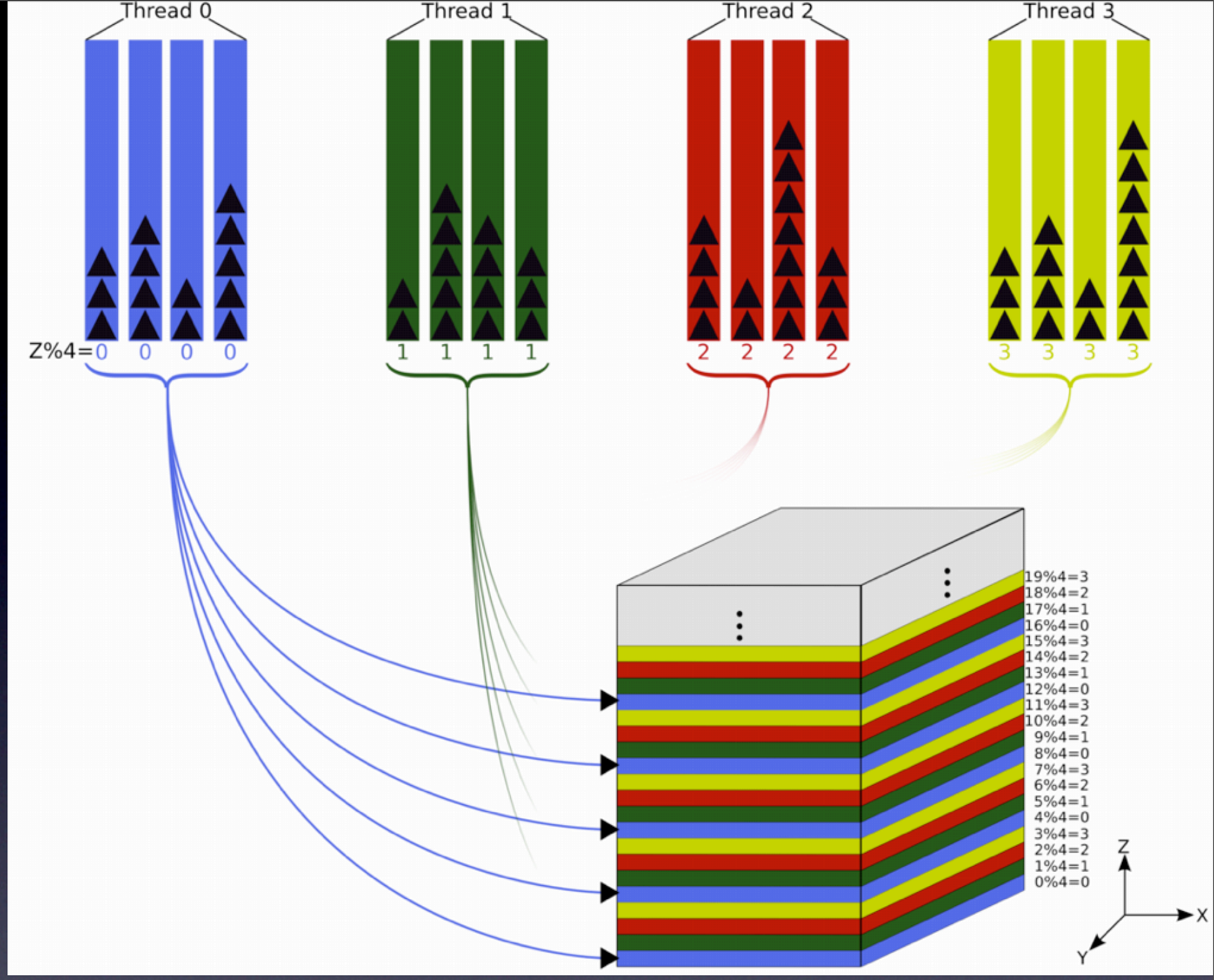
Sort-middle



- After bucket sort, each thread gets a group of similar buckets

This is why number of buckets == number of threads

Sort-middle



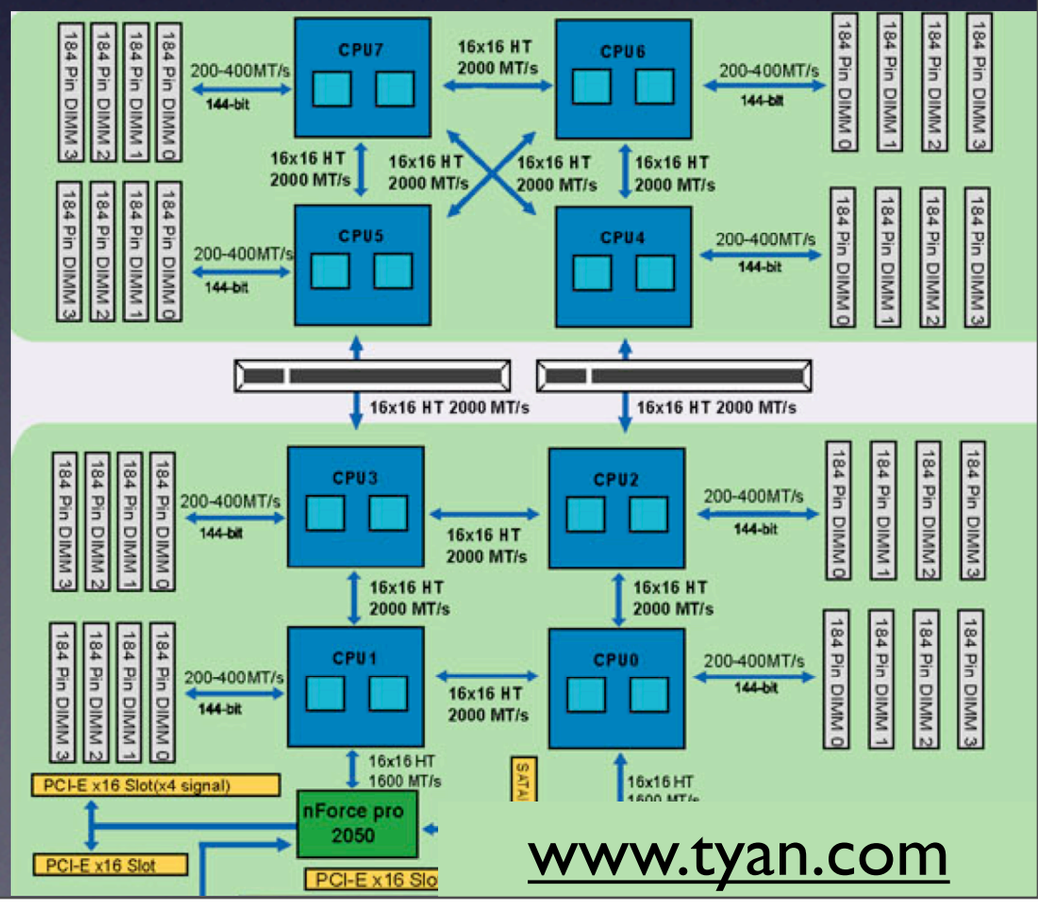
- Each thread inserts into specific locations of grid
- No write conflicts between threads

What to measure

- Different build methods
- Scalability
- NUMA and SMP performance
- Build times (scalable *and* fast)

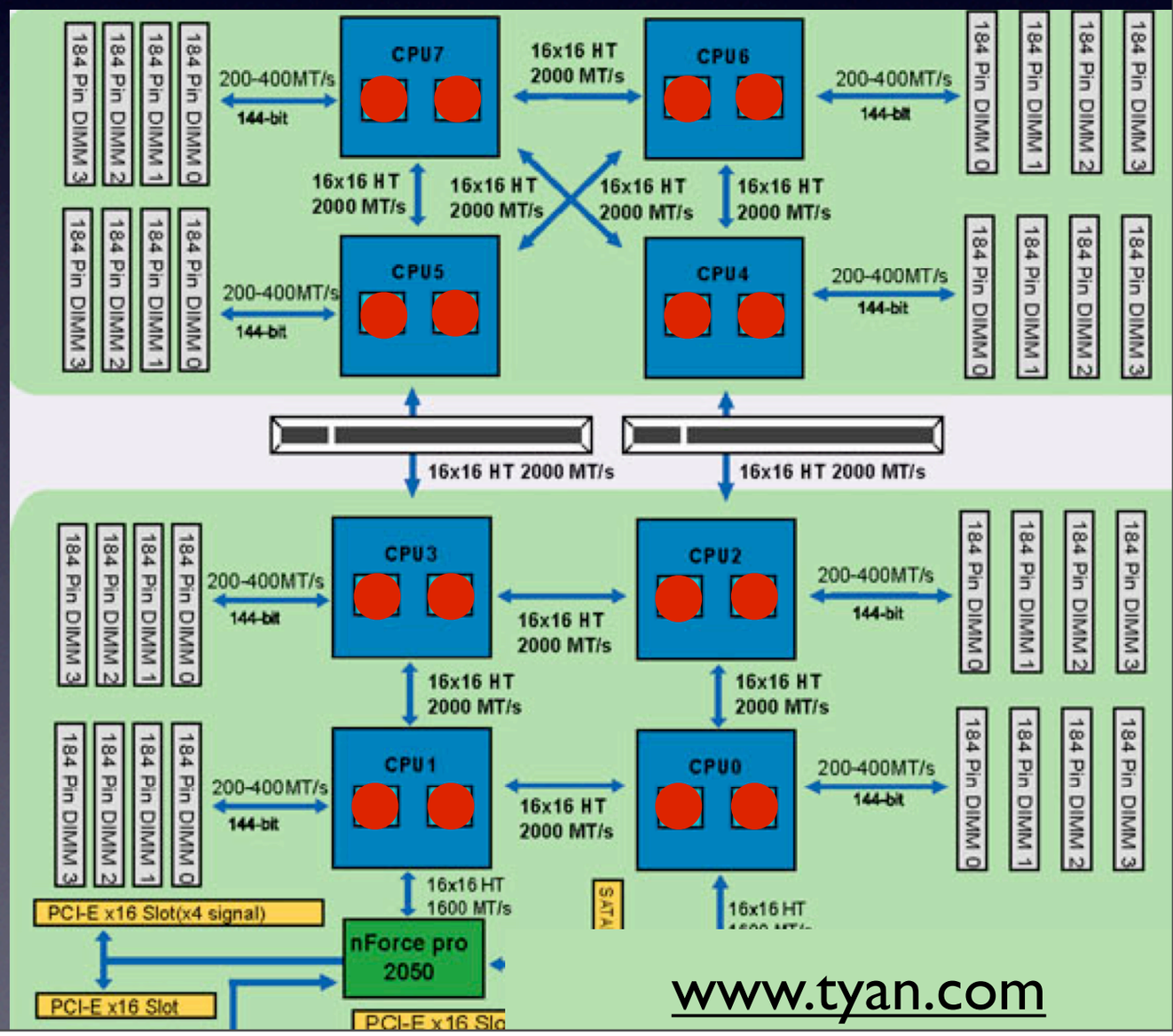
Results

- 8 2.4GHz dual-core Opteron 880 processors
- 8 nodes, each node has 2 cores
- 6.4GB/s local memory bandwidth per node
- Each node has 8GB local memory



Thread and memory allocation

- Spread out threads
- Allocate memory only on nodes that will use that memory
- Interleave memory allocations across nodes using that memory



NUMA API

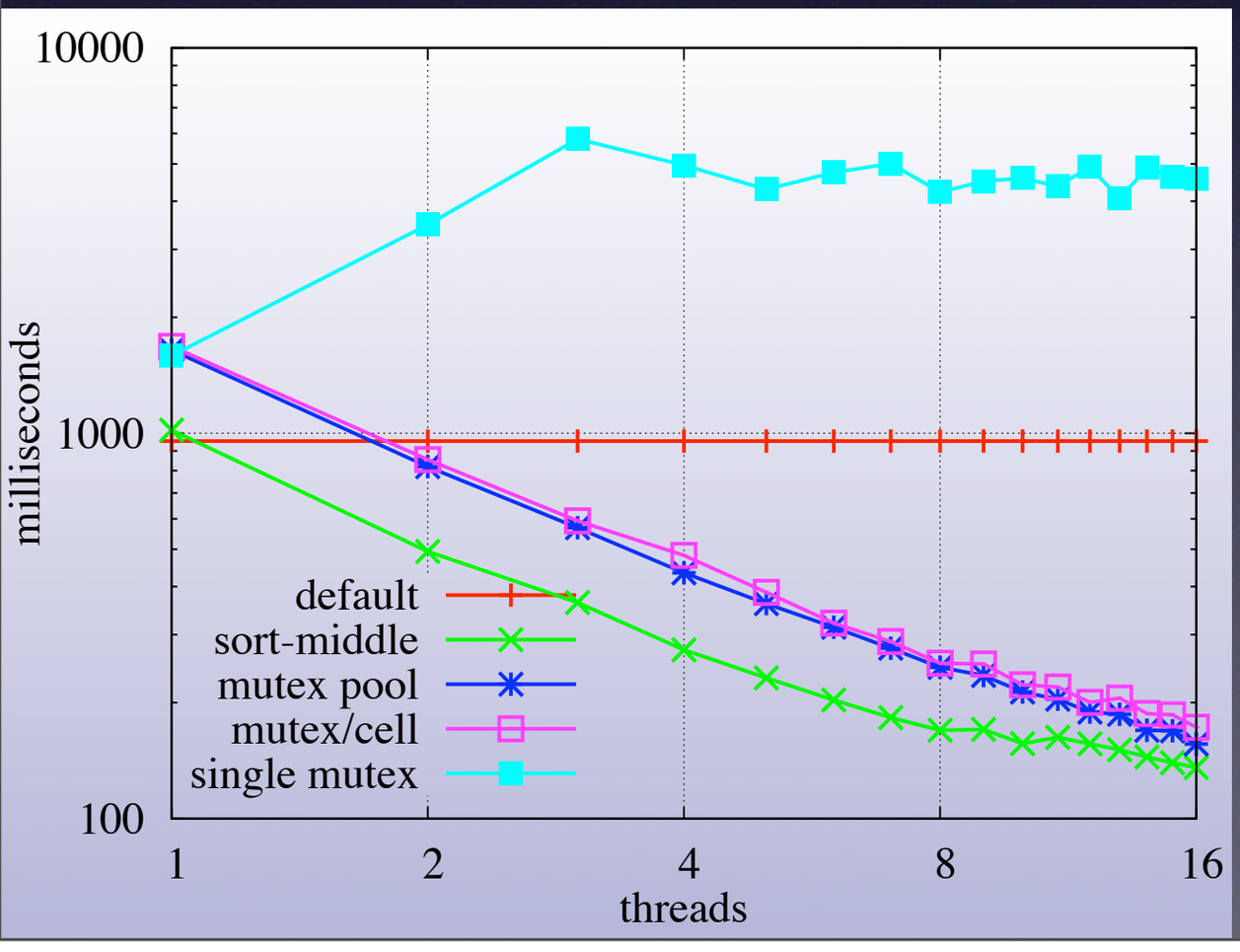
- OS won't always do the best thing
- Use to control memory assignments to nodes
- Download newer version that works...

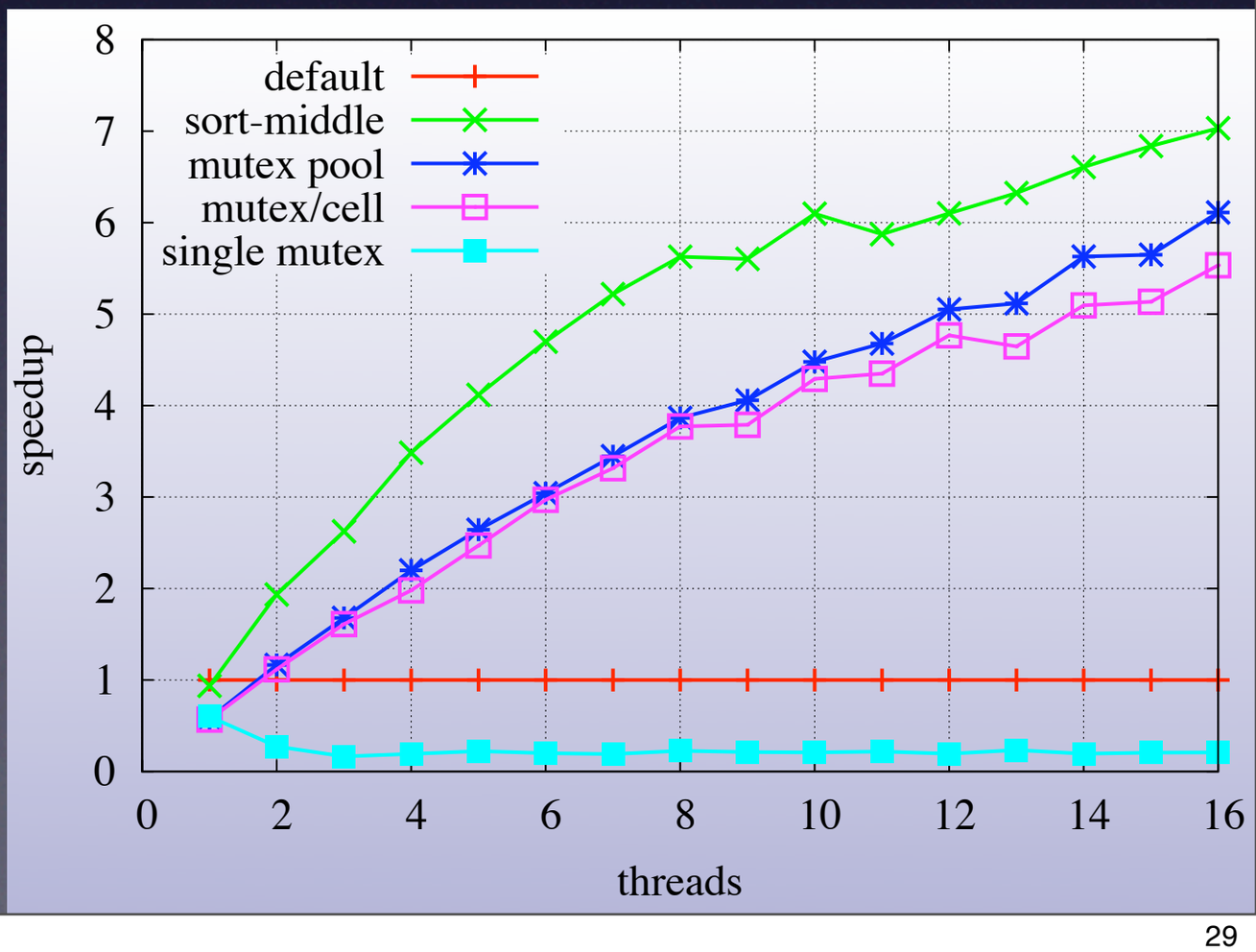
http://ftp.suse.com/pub/people/ak/numa/

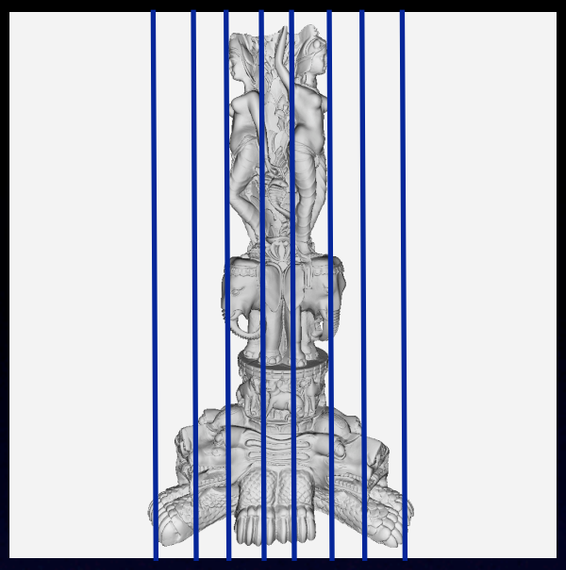
- Use `numactl --hardware | grep free` to make sure all nodes really have free memory

file cache often uses up memory that won't be released until all the other nodes have no more free memory.

Comparison build methods

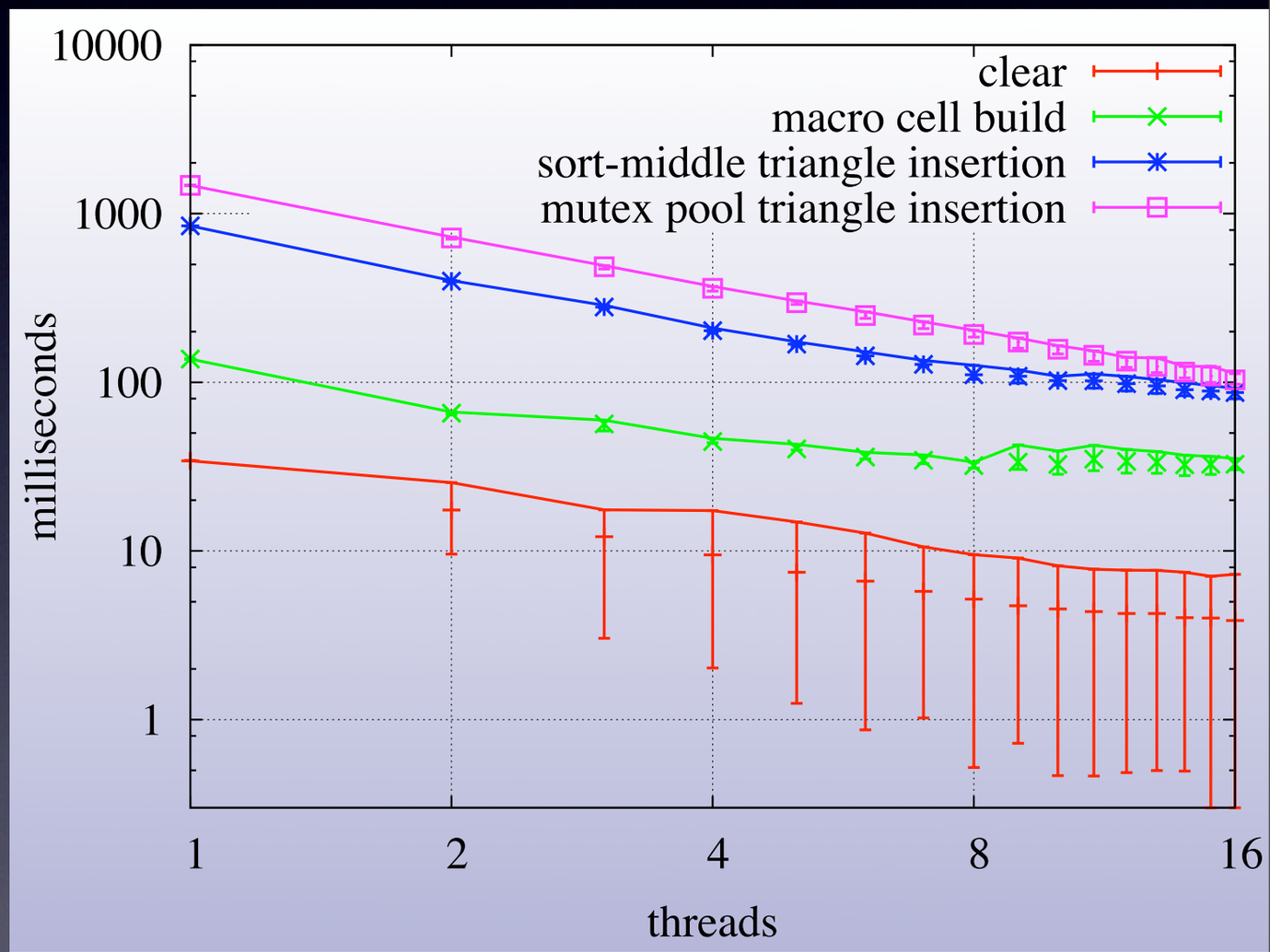






Individual steps

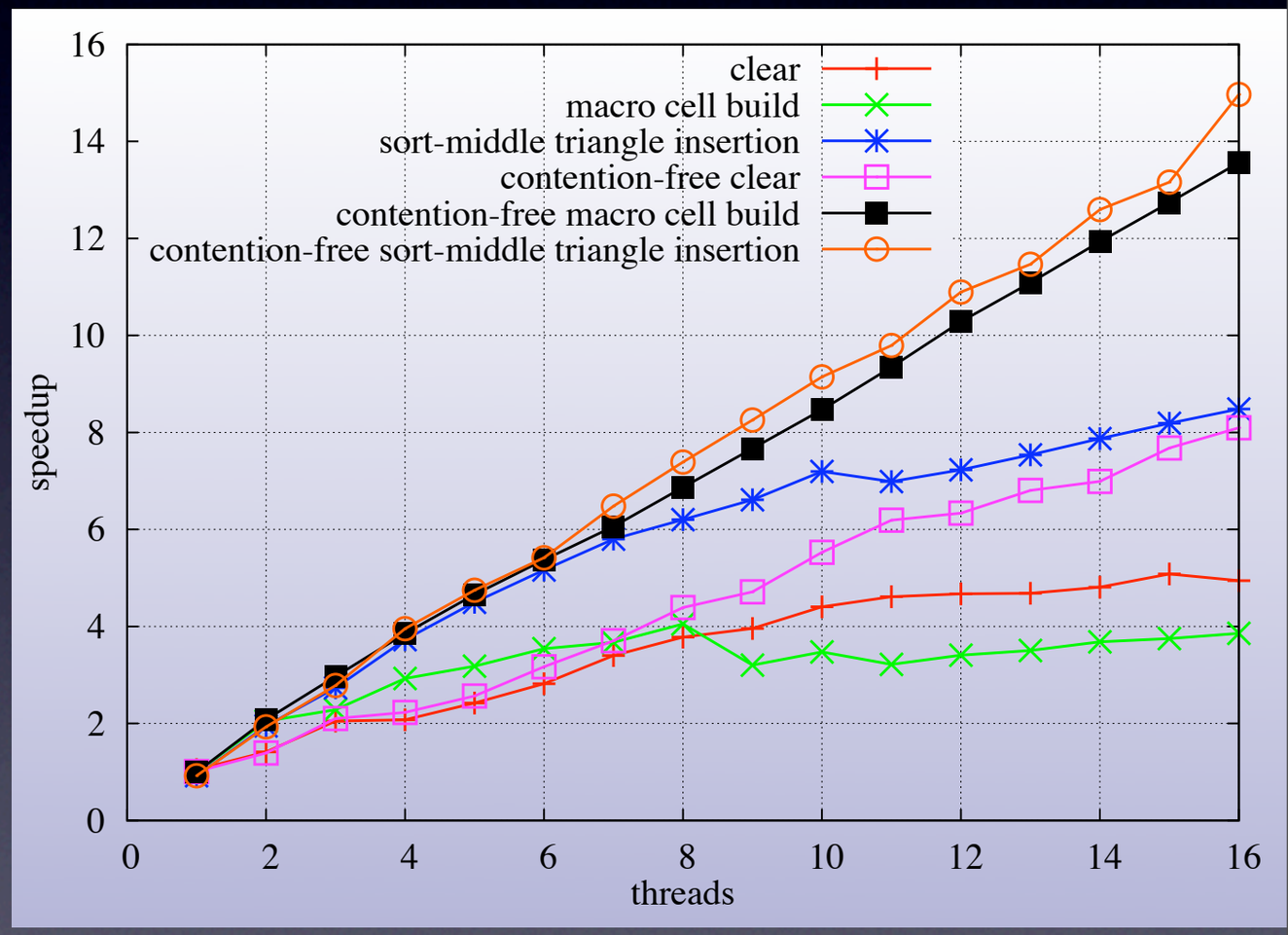
- Poor load balancing for clear step
- Spike at 9 threads for macro cell build due to memory bottleneck



why poor load balancing — triangle distribution

Memory system overhead

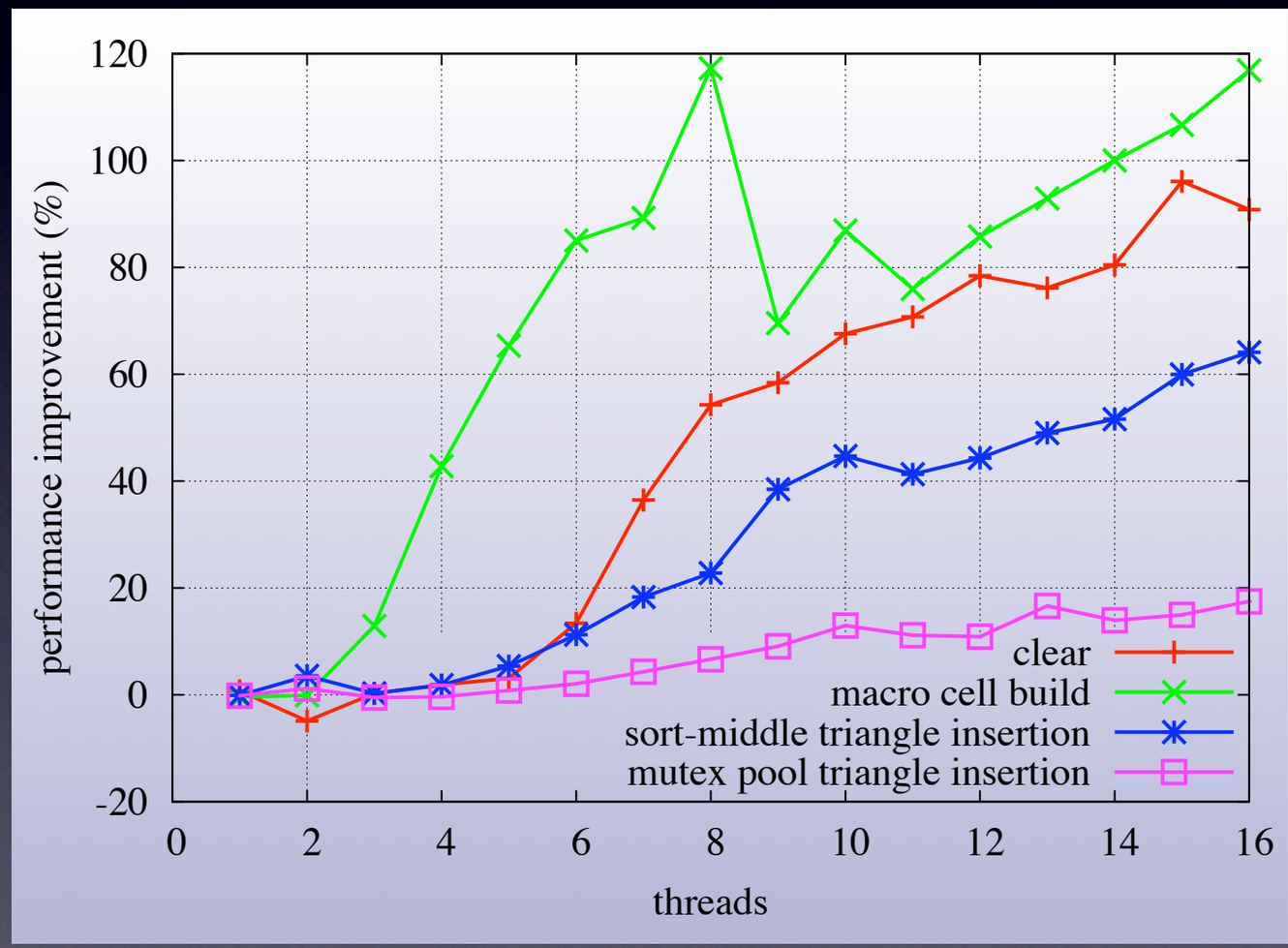
- Remove resource contention
- Memory bottleneck
- load balancing - clear



contention-free build measures all types of contention (false sharing, bottlenecks, etc...)

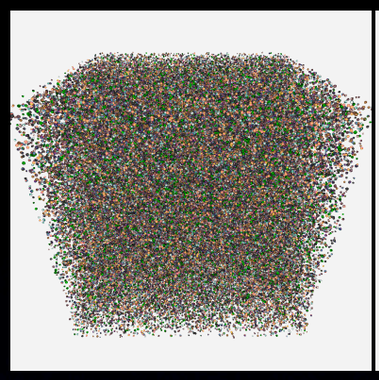
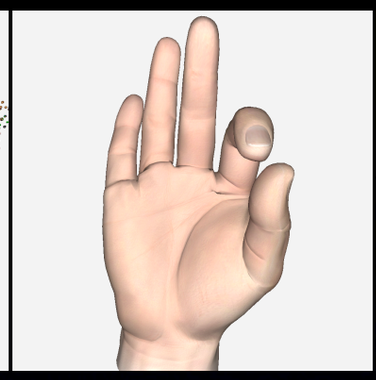
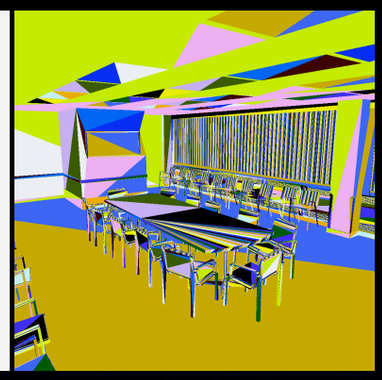
Targeting NUMA

- Allocate only on nodes that will access that memory
- Interleave memory allocations
- Total build time (with sort-middle) improves by 80%
- mutex pool is faster than sort-middle without NUMA optimizations

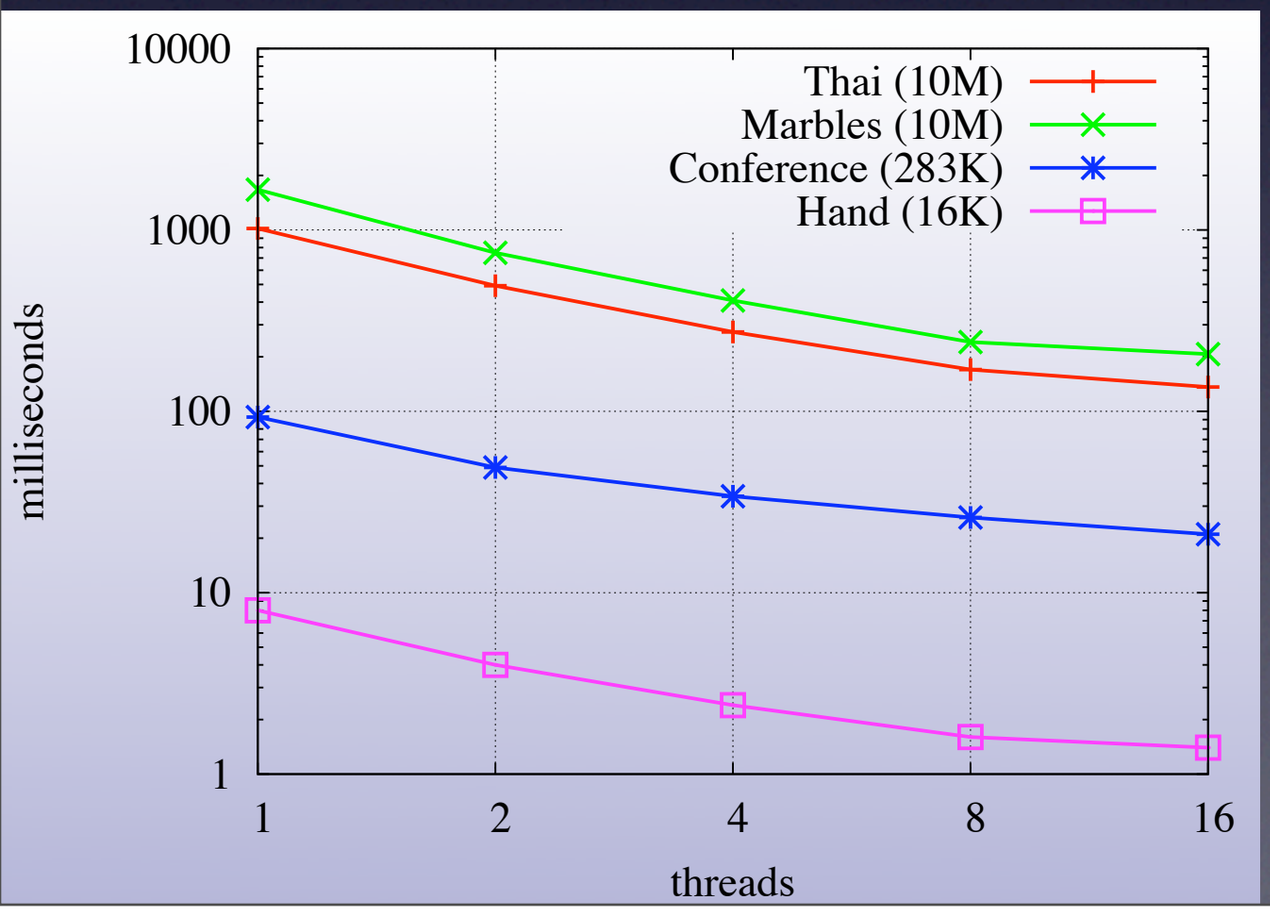


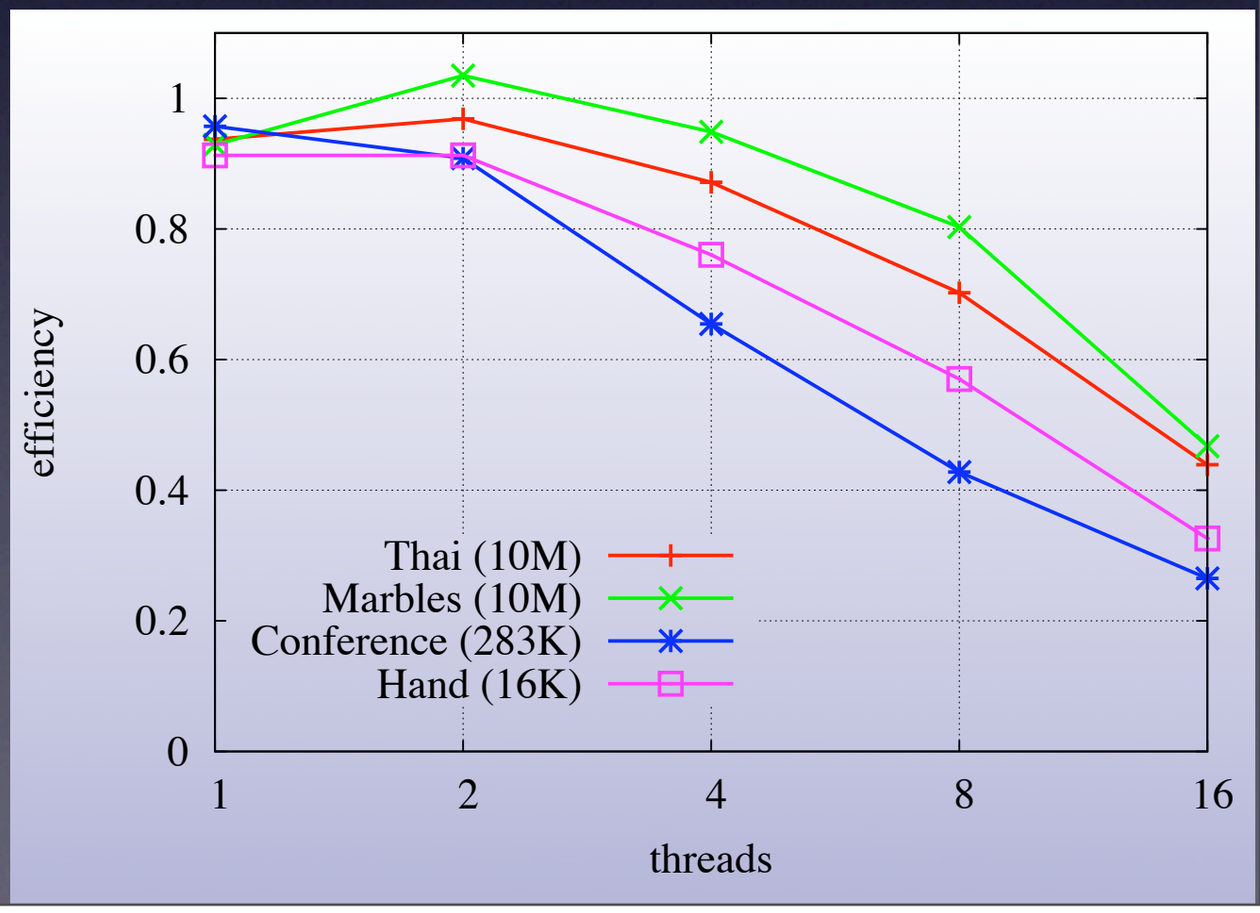
at 16 threads:
sort middle stage takes 92 and 151 ms
mutex pool stage takes 113 and 132 ms

Scene comparisons

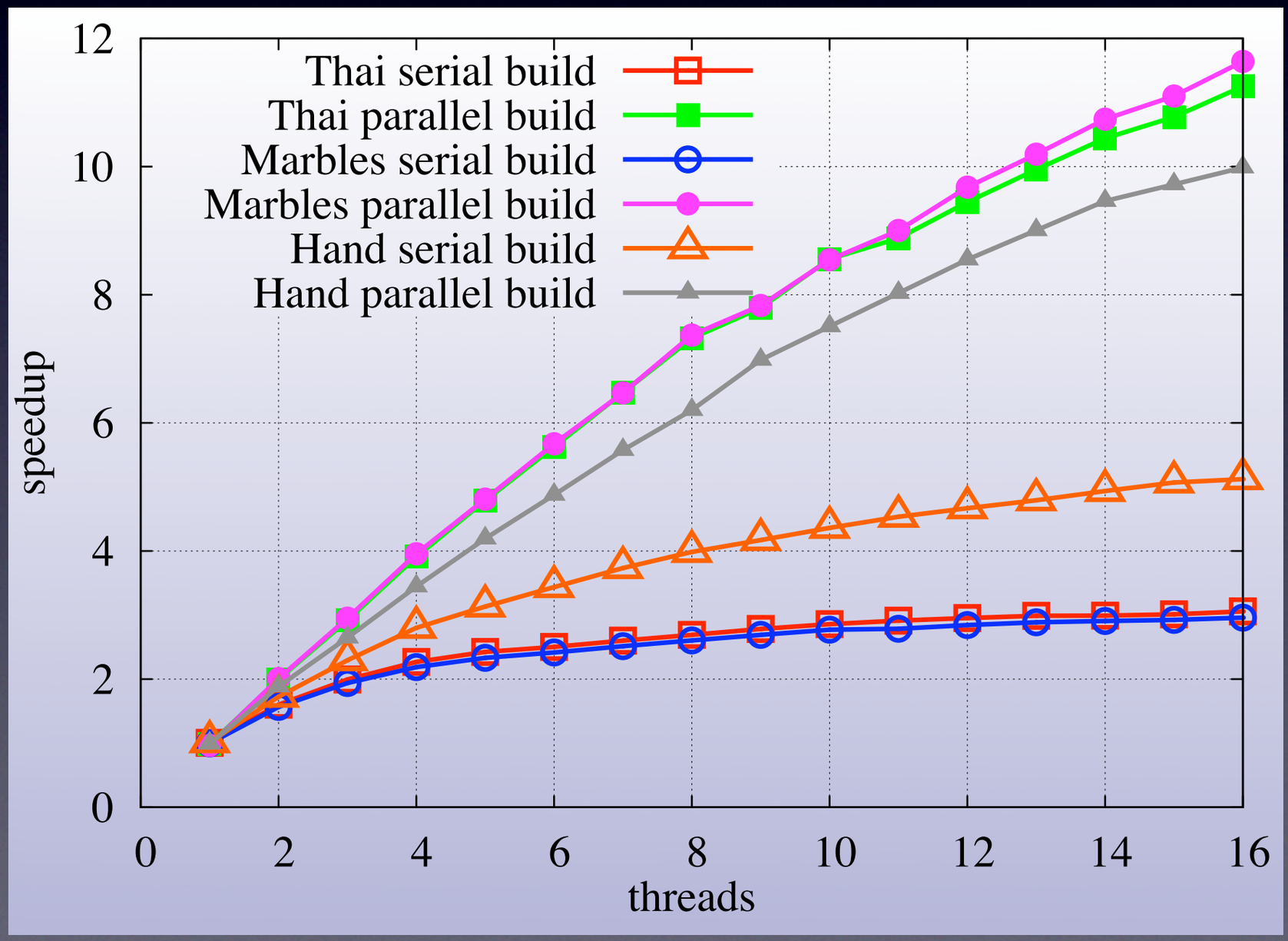
- Efficient even for small triangle counts
- Conference is a complicated scene





Total performance

- rebuild + ray cast



hand only scales to 13x (out of 16 cores) for just rendering...
16 threads:
  Thai: 0.78fps – 2.86fps
  Marbles: 0.50fps – 1.97fps
  Hand: 78fps – 150fps

Discussion

- Load balancing
- Memory bandwidth of future computers
- Cell: has 4 times the memory bandwidth of our test system
- Fair comparisons: type of systems, measure both performance and scaling, scenes
- Rebuild code available at: www.cs.utah.edu/~thiago

Conclusion

- Grid rebuild can scale to many cores
- Can even achieve interactive performance for large scenes with our parallel build
- Sort-middle has little overhead and scales well
- Importance of memory bandwidth
- Optimize code for NUMA machines

Note: generating data will really be the bottleneck