SAND2011-8645C

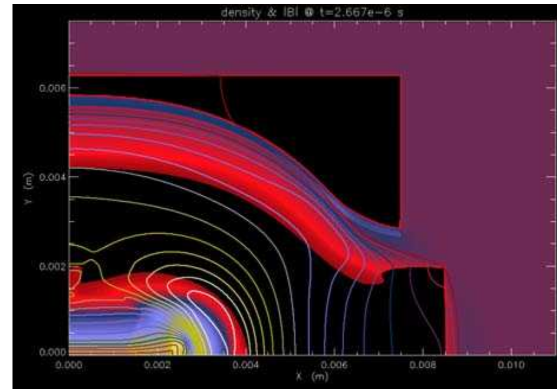
Results of Software Threading Experiments in ASC Codes

November 16, 2011
Sue Kelly for
Richard Drake, Alex Lindblad, and W. Roshan Quadros

Abstract: The number of CPU cores per processor in computer systems continues to increase and the MPI-everywhere approach will likely not be sustainable. One approach that might make more efficient use of the processor resources is to incorporate threads into the programming model. The ASC codes are large and complex. A massive re-write or re-factoring to use threads is daunting. Several code teams at Sandia were tasked to introduce threading into some kernel or subset of their ASC code. The purpose of which was to better understand how to program threads and the impact of threads on their code. This talk will provide a summary of the approaches and results for three of the experiments.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

CODE 1: Add (Some) Threading into ALEGRA



- **ALEGRA:**
  - **Operator-split coupled physics (explicit hydro & implicit magnetics)**
  - **Mostly "old school" C++ and some FORTRAN; many third party libraries**
- **<u>Goal</u>: add threading to material loop and examine performance**
- **<u>Objectives</u>:**
  - **Gain knowledge about adding threading to ALEGRA**
  - **Bring out performance issues when using threads**
  - **Compare performance on a few machines**

Sandia National Laboratories

Thread an Element Loop with OpenMP

Serial:

```
for ( itr = elements.begin();
      itr != elements.end();
      ++itr )
{
  Element* el = *itr;
  // bunch of code…
  Update_Material_State( el );
}
```

OpenMP:

```
int thread_idx;
#pragma omp parallel for private(thread_idx)
for ( thread_idx = 0;
      thread_idx < num_threads;
      ++thread_idx )
{
  itr     = thread_chunk[thread_idx];
  itr_end = thread_chunk[thread_idx+1];
  for ( ; itr != itr_end; ++itr )
  {
    Element* el = *itr;
    // bunch of code…
    Update_Material_State( el );
  }
}
```

- Not shown:
  - Thread chunk decomposition
- Issues:
  - Private variables
  - Accumulation variables
  - Data race conditions (!)

Thread an Element Loop with ThreadPool*

Serial:

```
for ( itr = elements.begin();
      itr != elements.end();
      ++itr )
{
  Element* el = *itr;
  // bunch of code…
  Update_Material_State( el );
}
```

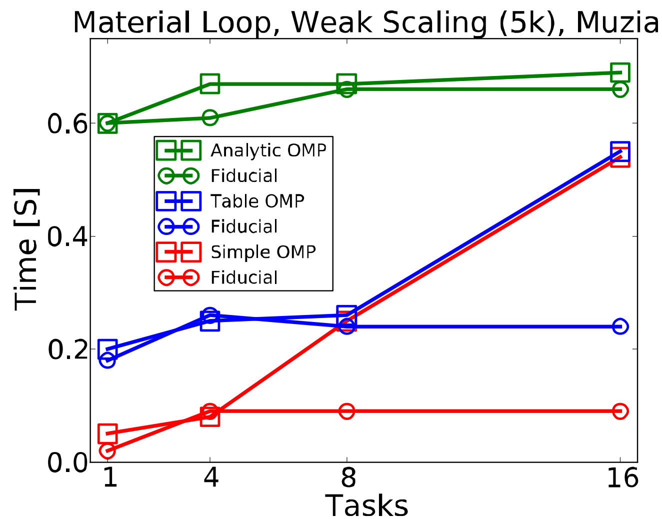
- Same: thread chunking, race condition issues, accumulation variables
- New: Transferring local variables to threads requires additional coding and restructuring

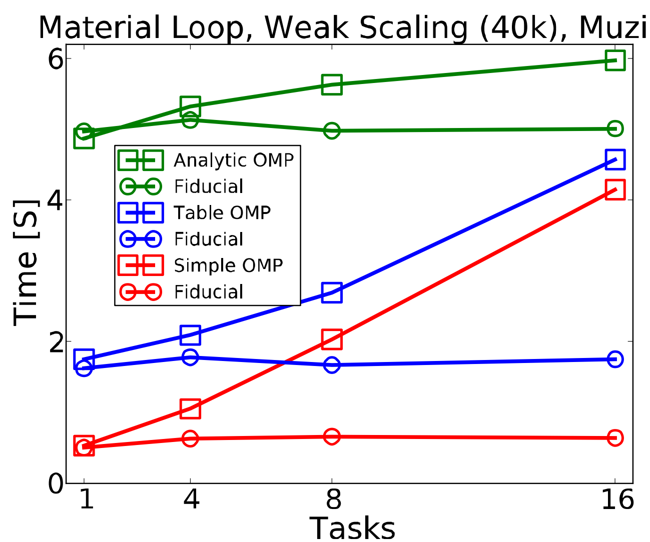
*ThreadPool is a pthreads-based library within the Trilinos library

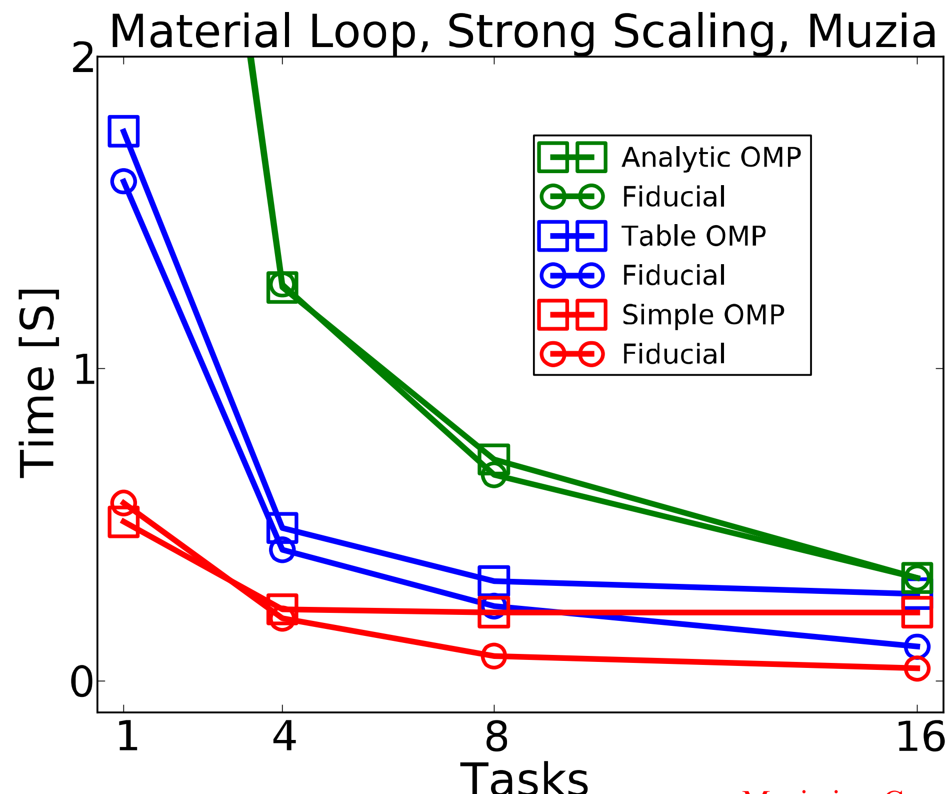
ThreadPool:

```
struct Args {
  vector<THashList::iterator> * chunks;
  UnsDynamics * reg;
  State COMMIT_STATE;
  … // other arguments
};

void work_kernel( TPI_Work* work ) {
  Args* args = (Args*) work->info;
  itr     = args->chunks[work->rank];
  itr_end = args->chunks[work->rank+1];
  for ( ; itr != itr_end; ++itr )
  {
    Element* el = *itr;
    // bunch of code…
    Update_Material_State( el );
  }
}

{
  Args args( chunks, reg, COMMIT_STATE, … );
  TPI_Run( work_kernel, &args );
}
```

Timings for Threaded Material Update Loop







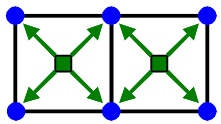
Muzia is a Cray XE6 with dual socket 8-core AMD Magny-cour CPUs

Material Model Types:

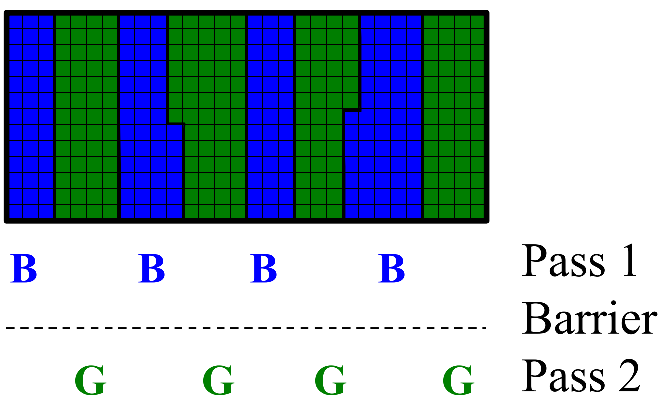
- Analytic: Lots of computation
- Table: Tabular lookup and interpolation
- Simple: Very little computation

Sandia National Laboratories

Threading an Element Assembly Loop (A Scatter)



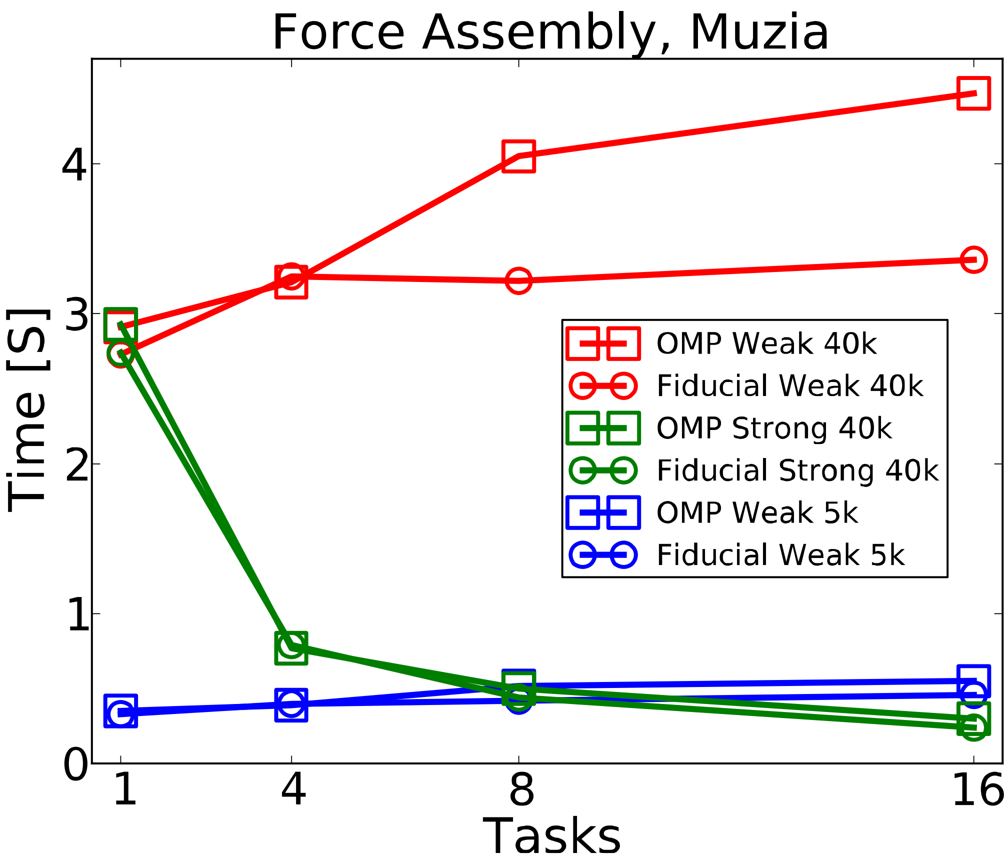
Decompose into twice the number of threads*

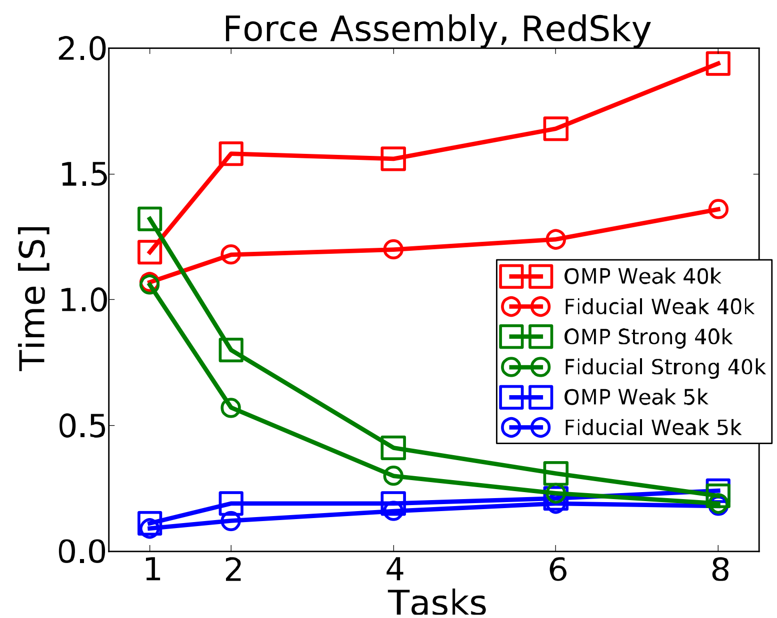



```
for ( int phase = 0; phase < 2; ++phase )
{
  int thread_idx;
#pragma omp parallel for private(thread_idx)
  for ( thread_idx = 0;
        thread_idx < num_threads;
        ++thread_idx )
  {
    int part = 2*thread_idx + phase;
    itr     = thread_chunk[part];
    itr_end = thread_chunk[part+1];
    for ( ; itr != itr_end; ++itr )
    {
      Element* el = *itr;
      Node** nds = el->Nodes();
      // read/write node data, read elem data
    }
  }
}
```

* Used Zoltan with 1D RCB

Force Assembly Loop Performance



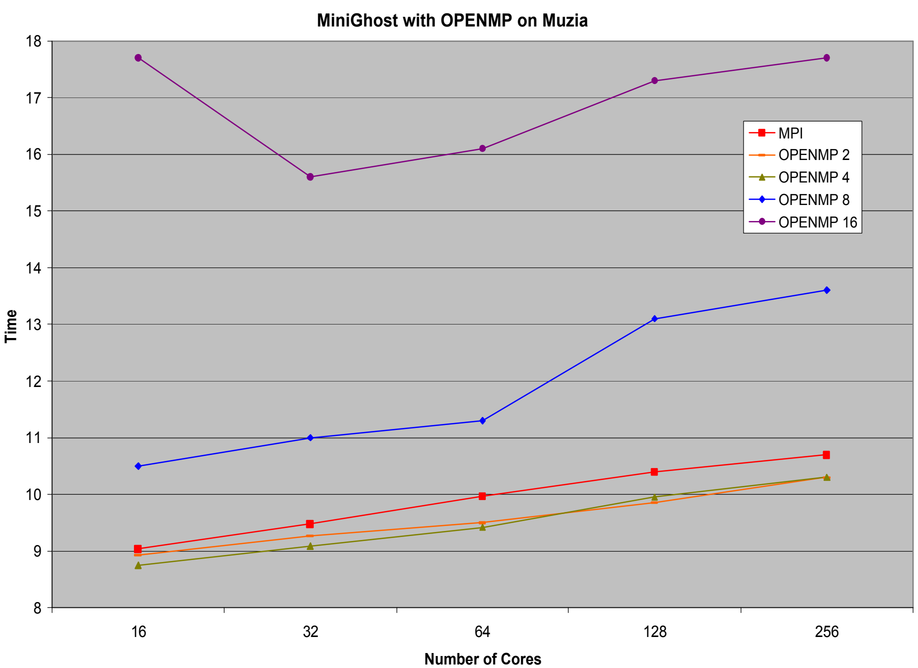


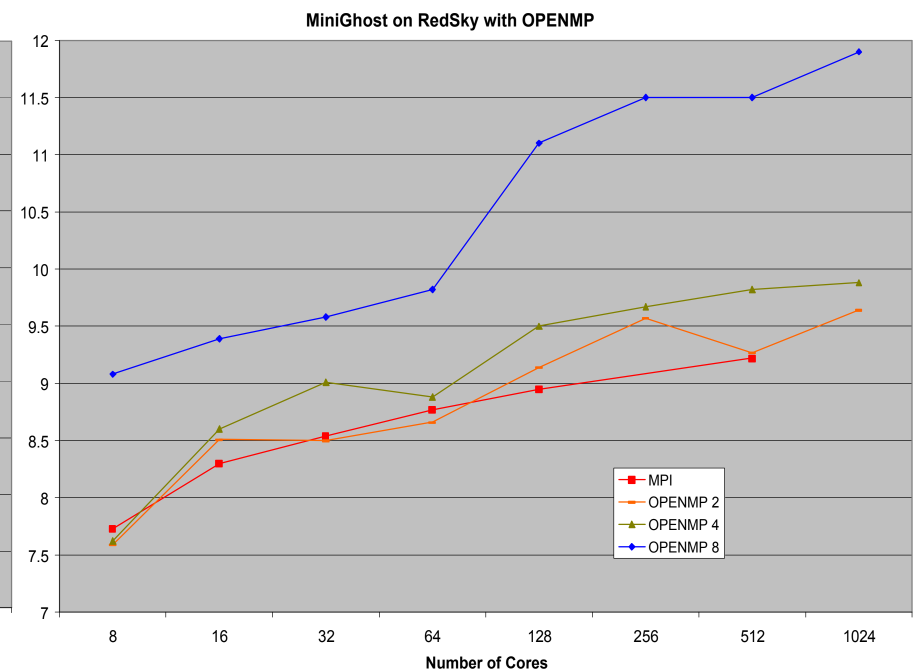
RedSky is a Sun cluster with dual socket/quad core, Intel Nehalem CPUS

Sandia National Laboratories

MPI & Threading Combinations
(Barrett & Vaughan)

- Used the "miniapp" called MiniGhost
- Stencil based PDE solver
- Weak scaling with different number of threads per MPI rank





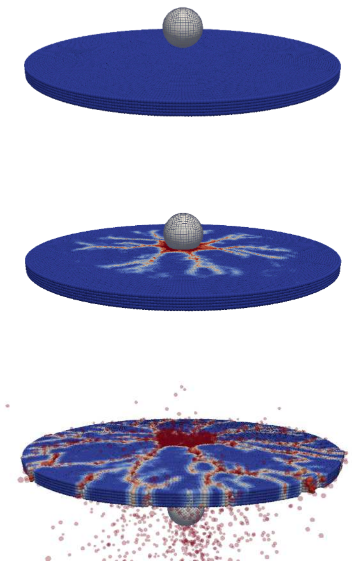
Sandia National Laboratories

What Next?

- Large risk for production app
  - Threading
    - How far will tiling get us?
    - Developer programming mechanism?
    - Top level threading vs loop threading?
    - Threading performance?
    - Thread correctness checking tools essential !!
  - Accelerators
    - Need to rewrite (almost) all algorithm implementations
    - Explicit memory movement needed?
- However, progress can be made
  - Data layout
  - Thread safety
  - Serial algorithm improvements
  - Sustainable performance tests

Code 2: Threading Options & Performance in Sierra/Solid Mechanics

- Thread the linear peridynamics material model internal force algorithm
  – Message Passing Interface (MPI)
  – Threading Building Blocks (TBB)
  – Thread Pool (TPI)
  – OpenMP (OMP)
- Strong scaling on a per node basis
- Weak scaling on a per node basis
- Weak scaling on larger, multi-node, problems
- Qualitative evaluation of code complexity
  – Implementation
  – Maintainability



Brittle fracture simulation

Peridynamics is available in Sierra/SolidMechanics for the modeling of material failure

Sandia National Laboratories

Threading Implementations

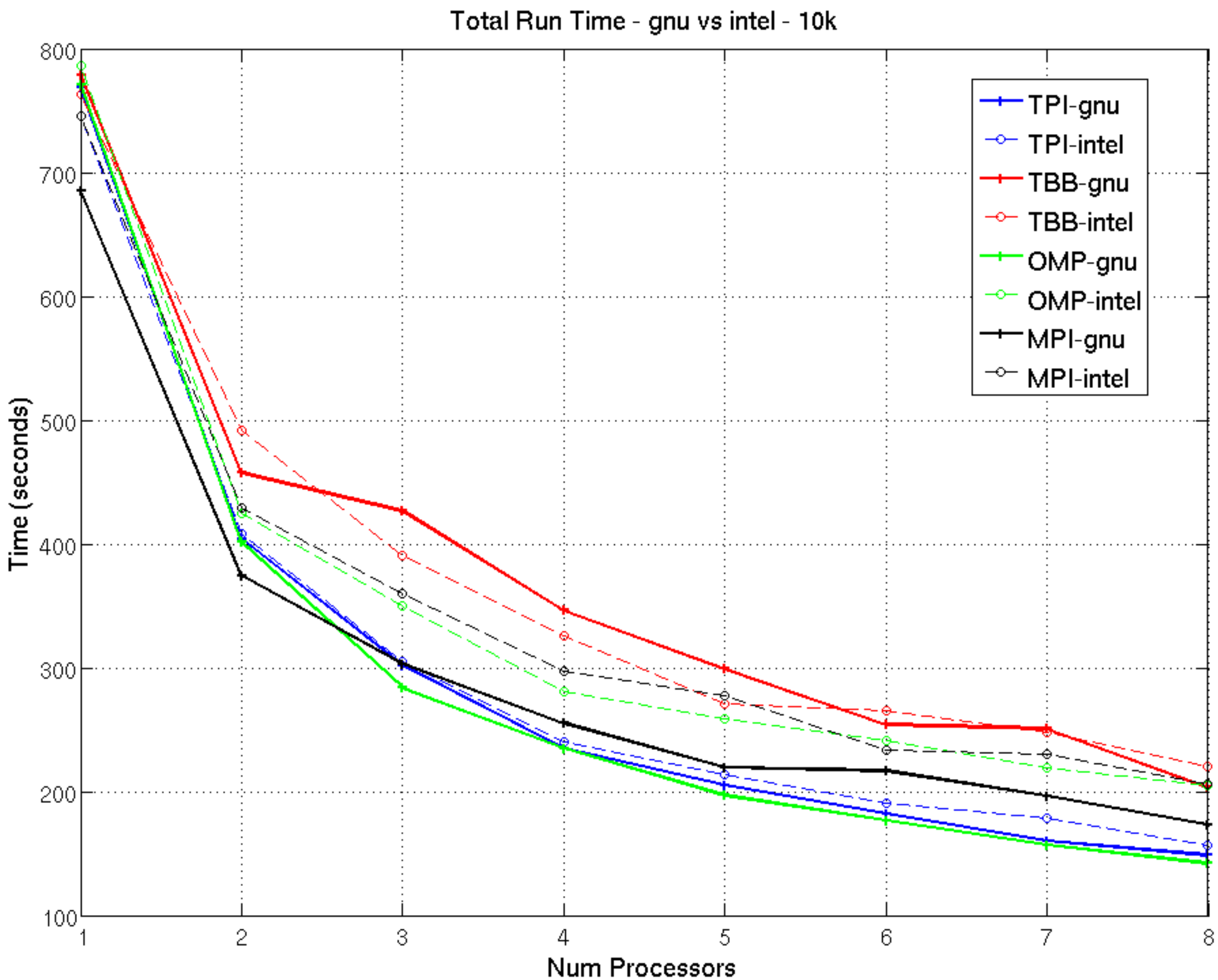
- Three phase
  - Compute dilatation
  - Compute element force, store to element, cache for neighbor (reduction)
  - Write neighbor forces
- Increase in memory footprint, copy of neighbor forces

- OpenMP
  - Easy to augment code - 5 #pragma statements
  - Manual vector reduction across threads
- Threading Building Blocks
  - 2 new classes, 2 methods each – dilatation and force calculation
  - Built in vector reduction operator
- ThreadPool (a pthreads-based library within Trilinos)
  - Manual initialize, join, computation of workspan - 1 struct, 5 new methods
  - Built in vector reduction operator

Summary of Scaling Results

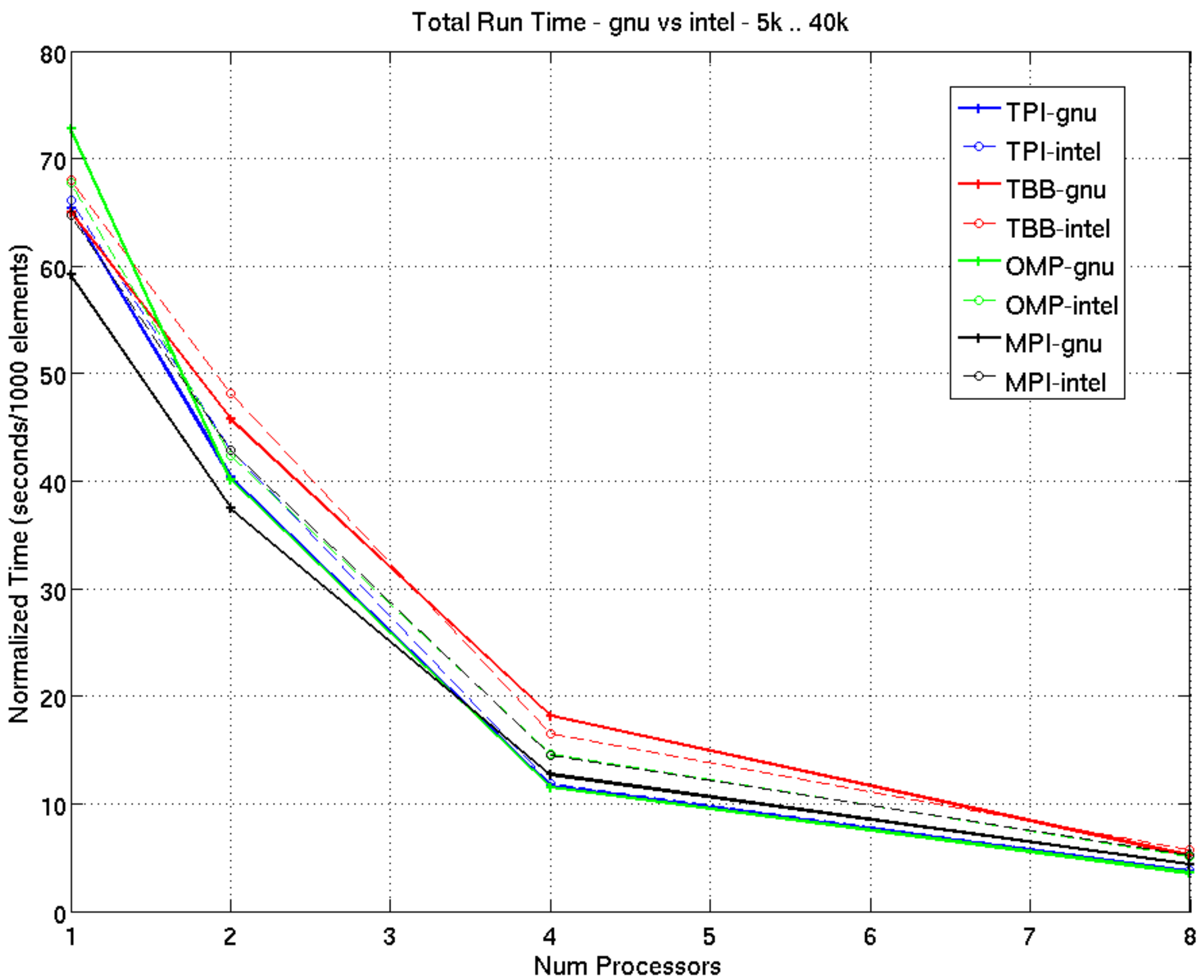
- Simple rotating plate test problem
- Threaded models can offer up to 25% speedup over strictly mpi runs on a single node
- For problems that do not contain contact
  - Typical solid mechanics problems with contact spend 15-20% of time in internal force algorithm
- Problem size dependent – the larger the better
- Compiler dependent, gcc 4.4.4 and intel 11.1

Per Node Strong Scaling



Sandia National Laboratories

Per Node Weak Scaling



Conclusions

- Significant difference in level of effort to implement various threading models (Kokkos might alleviate this)
- Increased difficulties debugging threaded code
- Threading offers a significant decrease in communication costs on a per node basis
- Had to choose one, OpenMP
  - Standard, widespread
  - Similar performance as TPI
- Performance gains on a per node basis can be significant (problem and compiler dependent)

Unanswered Questions

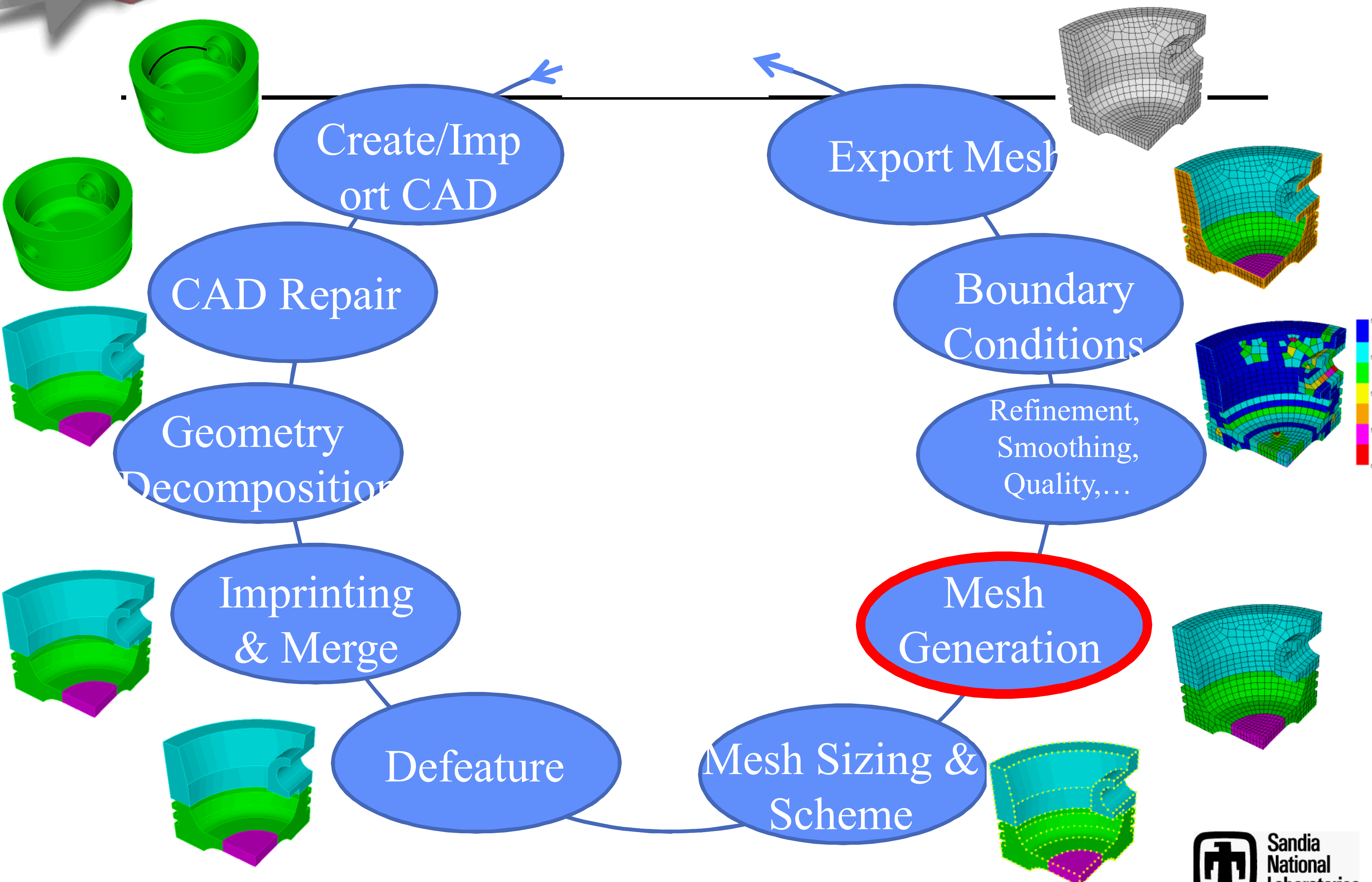
- How well do these models scale to higher core counts?
- How easy would it be to add threading to more complex internal force routines?
- We need more scaling data and a determination of where Sierra/SM would benefit the most from threading.
- How does threading other, more performance critical, areas of the code (e.g. contact) impact performance?
- What data structure changes are required to use gpus instead of multi-cores cpus?

Sandia National Laboratories

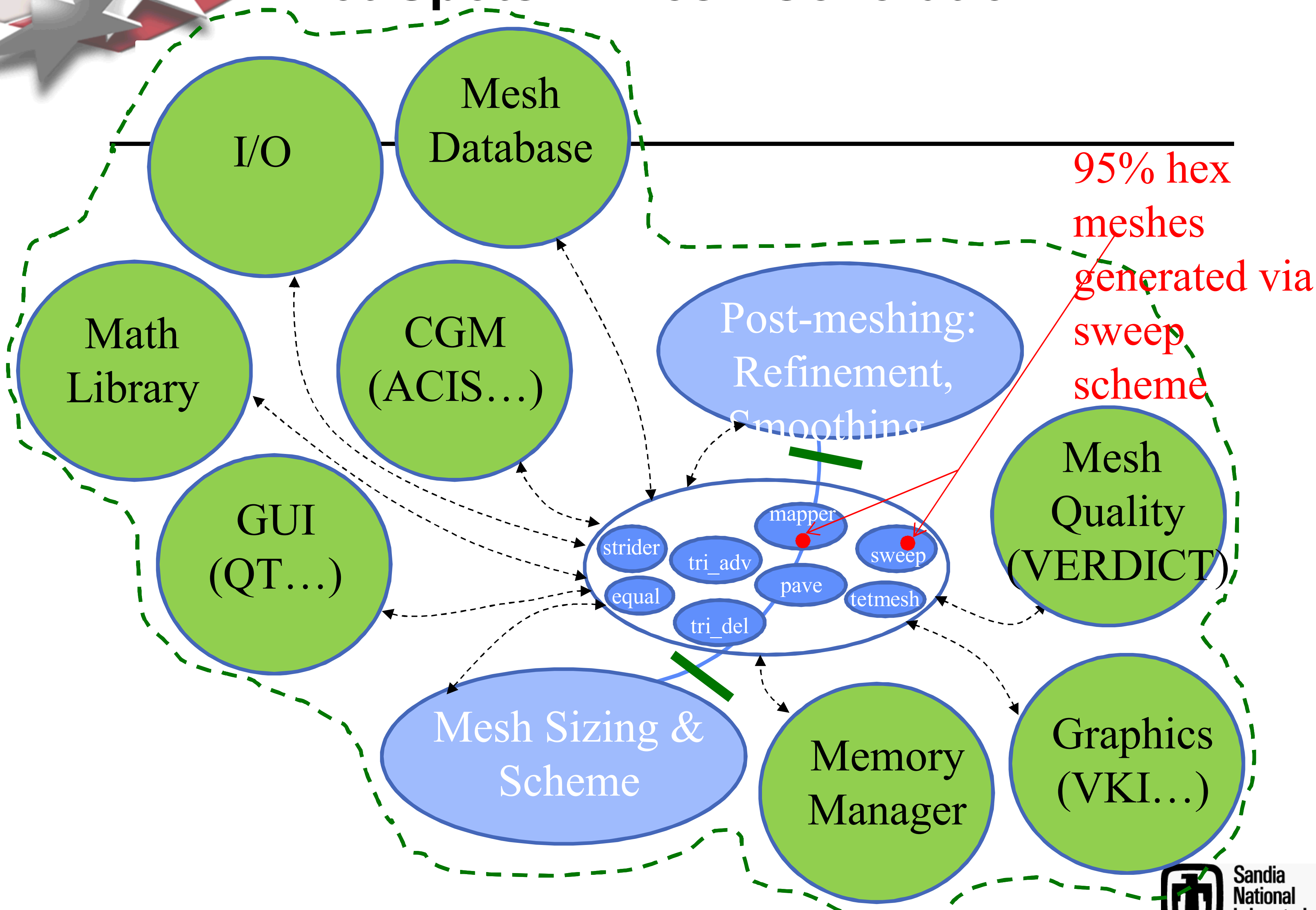
Code 3: Adding Threads in CUBIT for Shared Memory Parallel Mesh Generation

- OpenMP at Local Hot-Spots of Surface & Volume Meshing
- ThreadPool for Global Parallelism of Surface Meshing

Common Work Flow in CUBIT



Hot-Spots in Mesh Generation



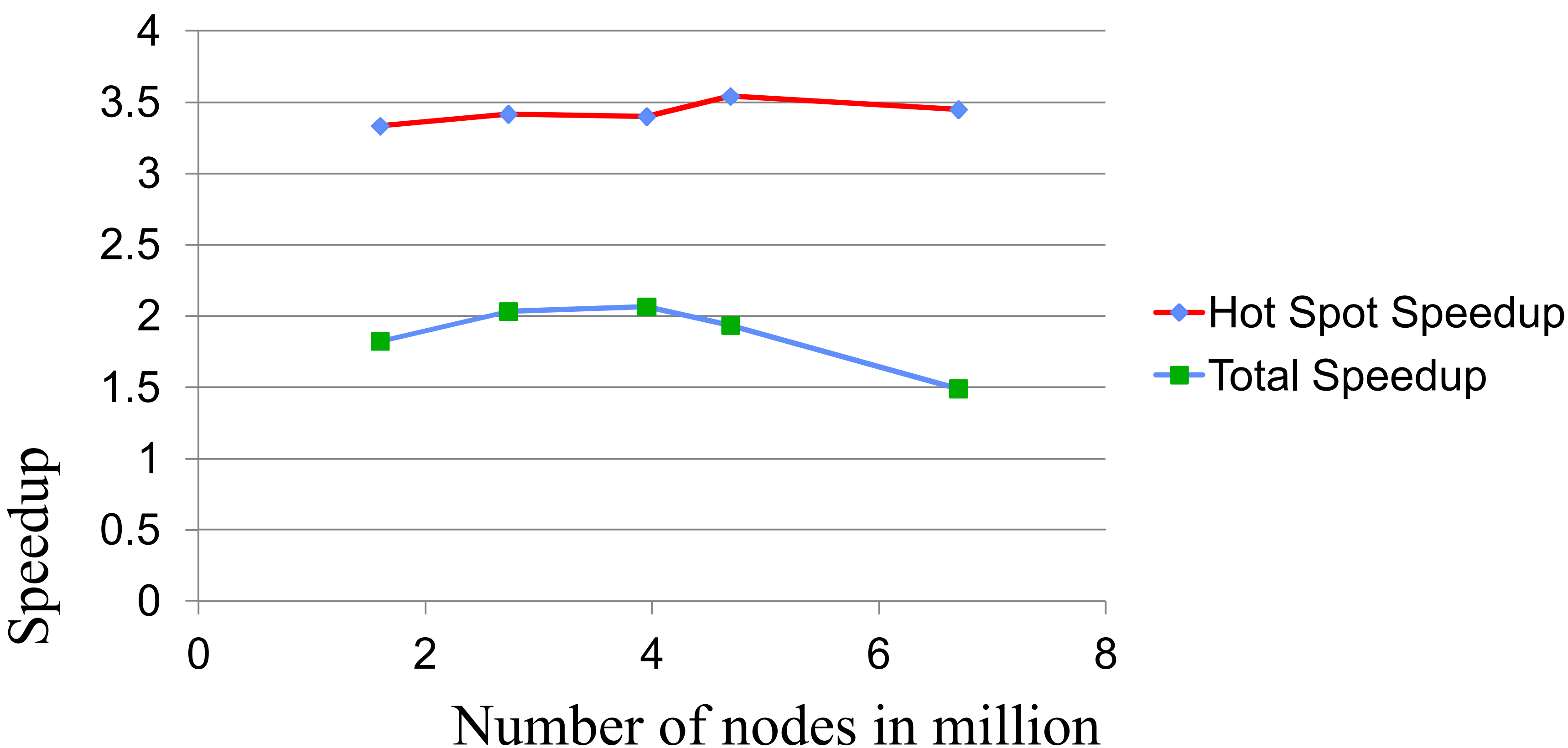
Performance

Nodes in Million	Serial Hot Spot Time	Serial Total Time	Parallel Hot Spot Time	Parallel Total Time
1.6	20	31	6	17
2.73	41	61	12	30
3.95	68	95	20	46
4.69	85	120	24	62
6.7	138	474	40	318

OS: Windows 7 (64bit)

Hardware: Quad-core Intel Xeon CPU X5450 @ 3 GHz, 4 GB RAM

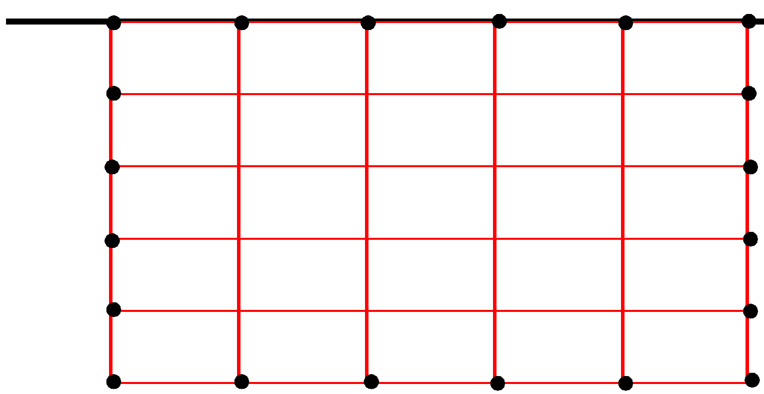
Speedup

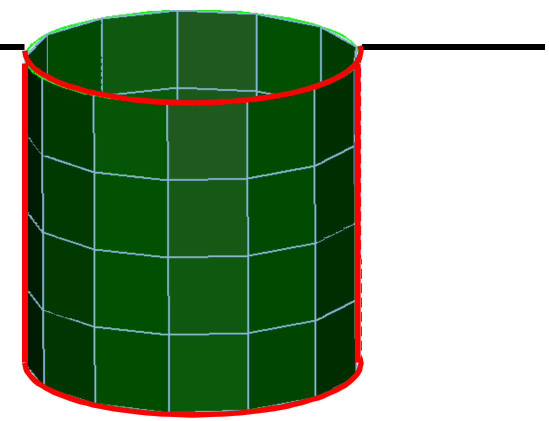


OS: Windows 7 (64bit)

Hardware: Quad-core Intel Xeon CPU X5450 @ 3 GHz, 4 GB RAM

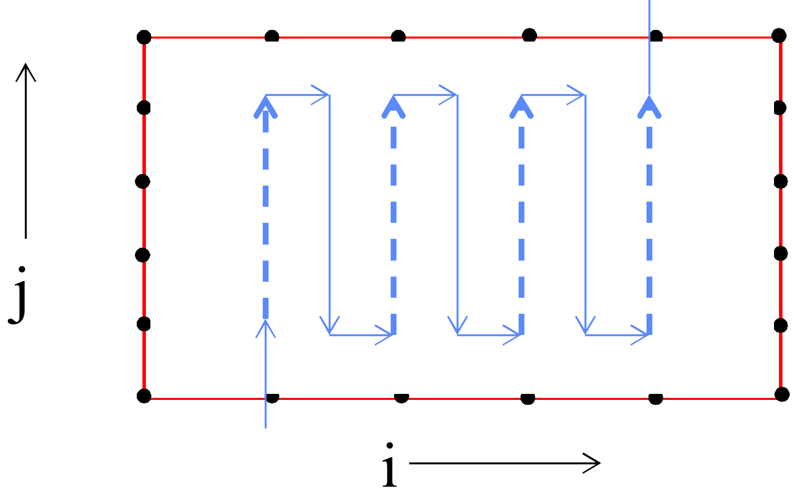
Surface Mapper (Serial & Parallel)



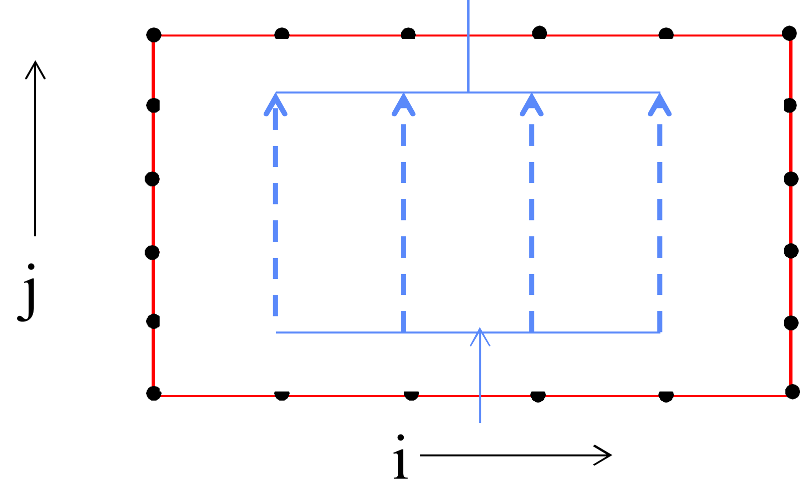


Mapped Mesh in Parametric Space

Mapped Mesh in World Space

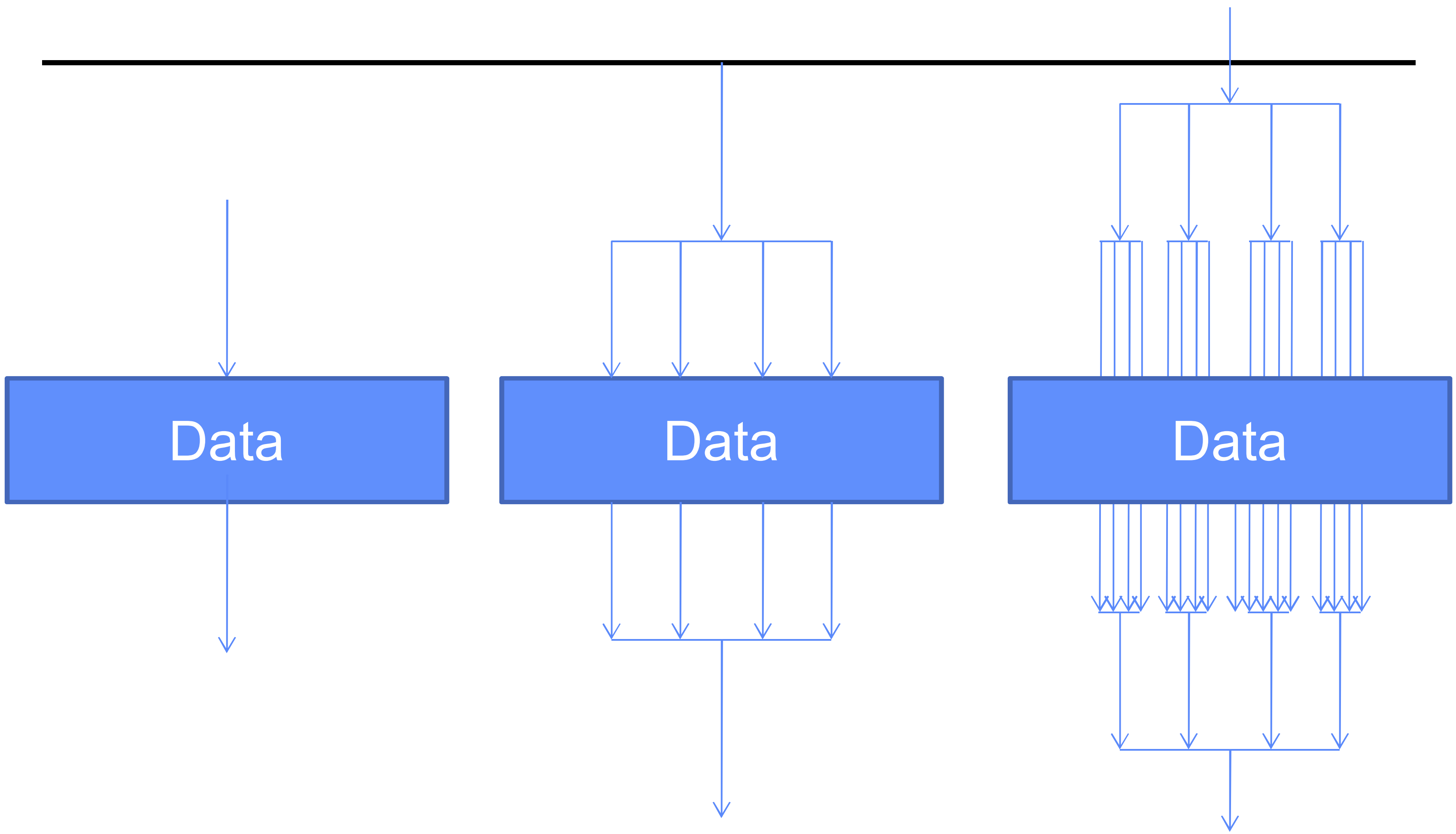


Old Serial Method



New Parallel Method

Types of Programs



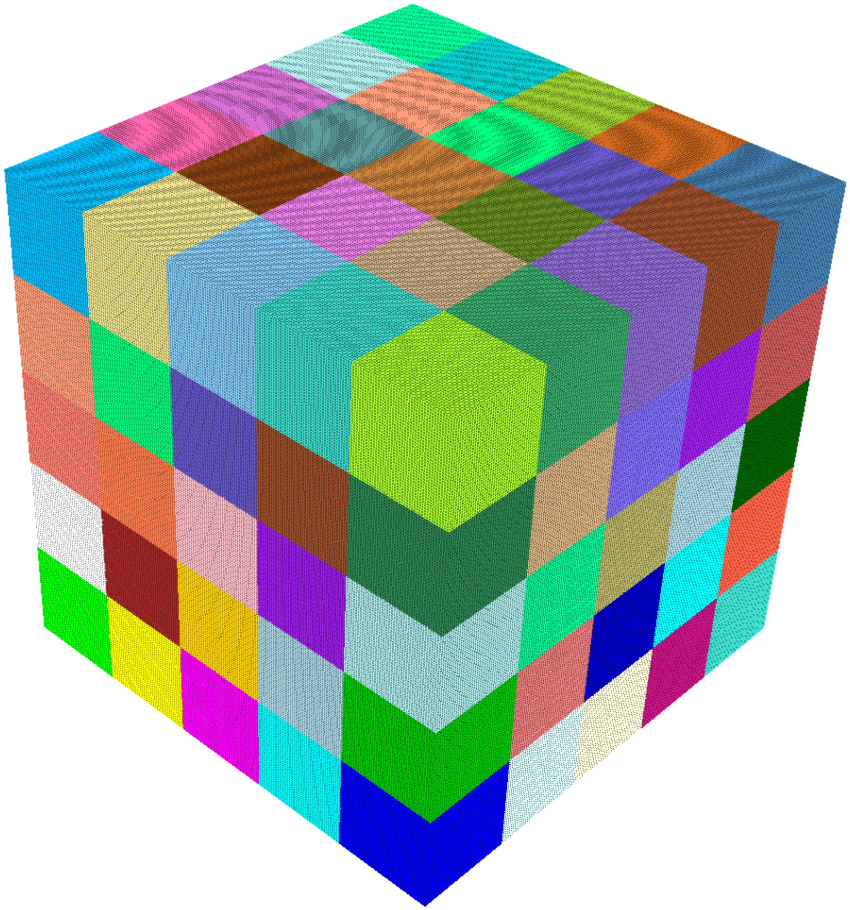
Serial

Parallel

Nested Parallel (parallel in both i & j)

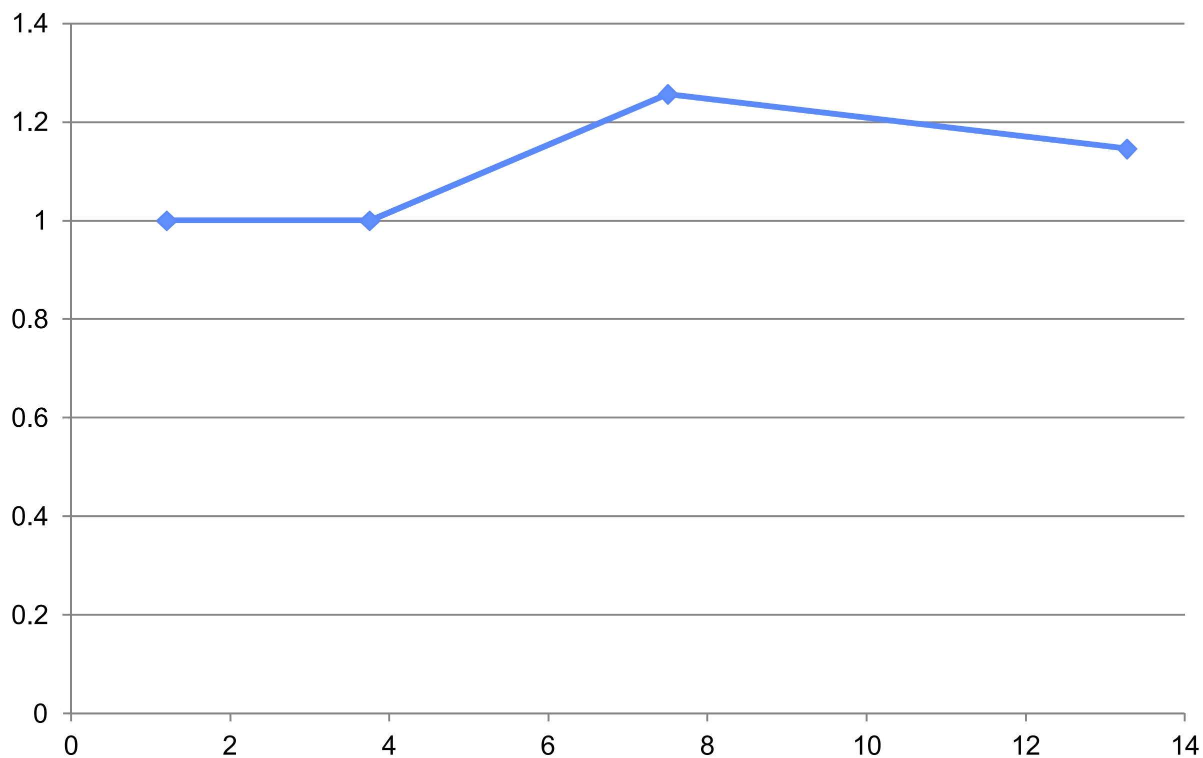
Sandia National Laboratories

Parallel Program on Very Fine Grain



- # Volumes: 125
- # Surfaces: 750
- Mesh size: 0.25 to 0.075
- # Nodes: 1.2 to 13.27 million

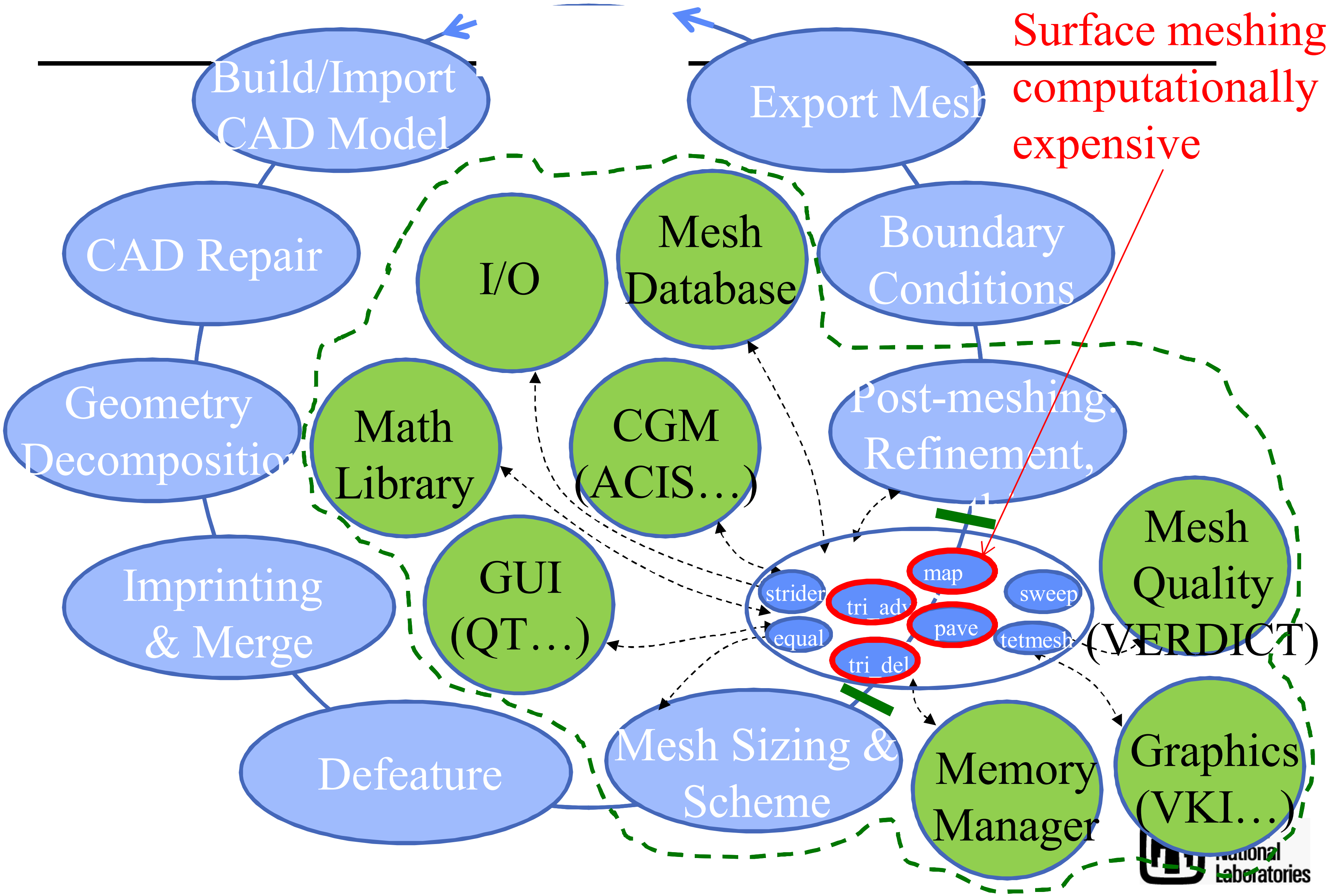
Speedup in Very Fine Grain Data



Number of nodes in million

Major Bottlenecks in Threading:

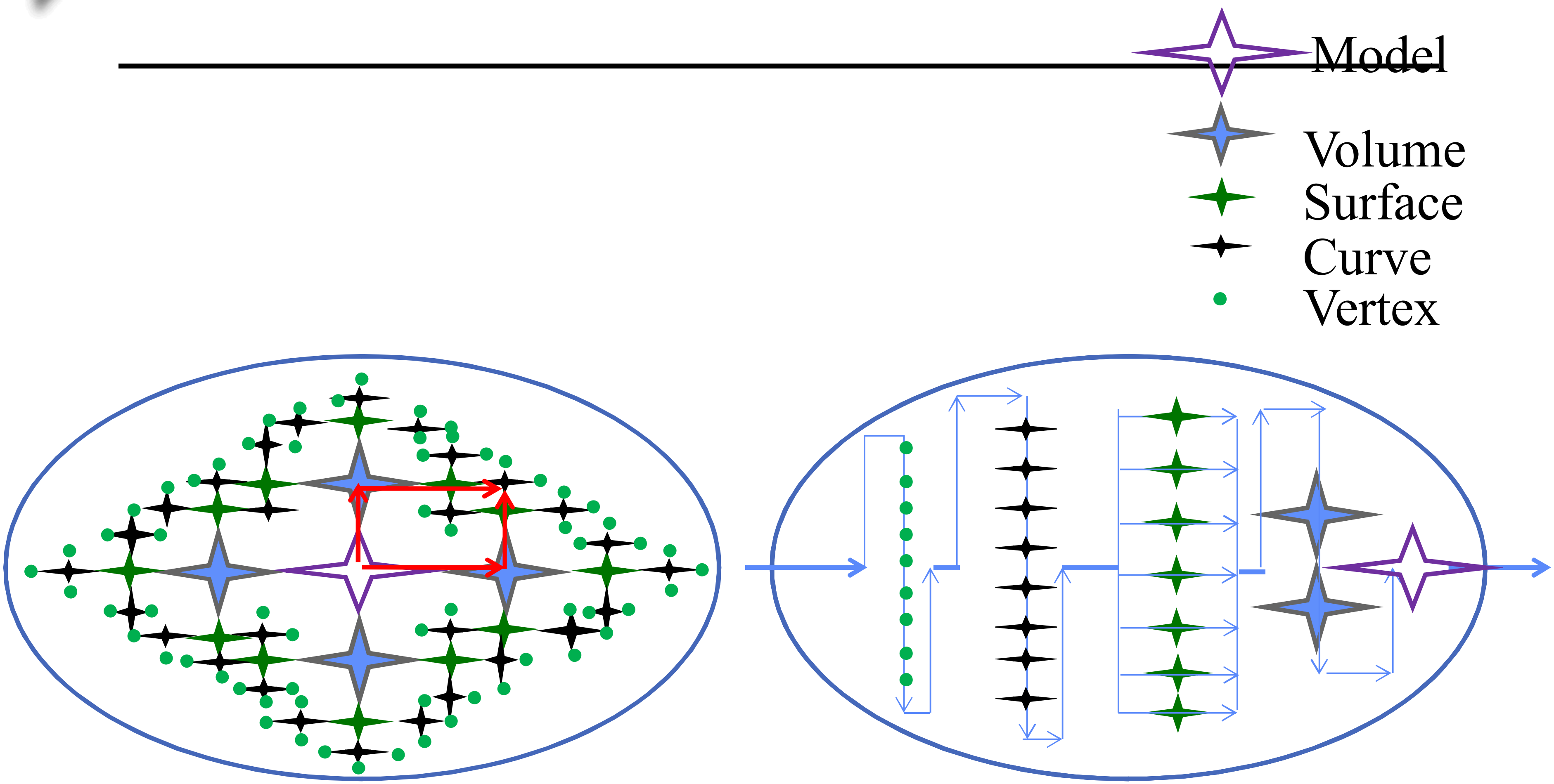
Ordering, Interdependency,& Thread Safety



Solutions to Bottlenecks

- Ordering
  - Use bottom-up instead of depth-first traversal
- Interdependency
  - Disable non-critical information reporting
  - Use native system memory manager
- Non-thread safety
  - Wrap geometry kernel in a thread safe interface using locks
  - Replace global data using local data
  - Handle all graphics, GUI, and mesh database access in master thread, and perform meshing in slave threads

Solution for Ordering



Old Ordering: Depth-First

New Ordering: Bottom-Up

Results

Windows 7, AMD Four Core Machine

Tri Meshing

Number of Surfaces	Type	Serial (tris/sec)	Parallel (tris/sec)	Speed Up
6	Planar	3321	9300	2.8
66	Planar	19279	62989	3.27
182	Mixed*	28048	68068	2.43

Paving

Number of Surfaces	Type	Serial (quads/sec)	Parallel (quads/sec)	Speed Up
6	Planar	2459	7092	2.88
66	Planar	15086	48116	3.19
182	Mixed*	5914	13497	2.28

* Mixed = Planar, Periodic, Spline

Sandia National Laboratories

Future Work

- OpenMP
  - Explore other OpenMP clauses and directives
  - Parallelize other hot-spots: mesh quality, geometry, associativity, diagnostics, imprint and merge, CAMAL smoother, layer-by-layer paving, and import/export mesh
- ThreadPool
  - Improve thread safety
  - Enable information reporting
  - Remove CUBIT memory manager
  - Parallelize other algorithms such as volume meshing

Conclusion

- **OpenMP**
  - **+ small developer time for a relatively big gain in speedup**
  - **+ good for domain decomposition at hot-spots**
  - **+ easy maintenance as serial and parallel share same code**
  - **- requires detecting hot-spots in the algorithm**
- **ThreadPool**
  - **+ fine control on task decomposition (graphics, meshing, …)**
  - **+ no need to understand details of the algorithm**
  - **+ good scalability can be achieved**
  - **- requires significant refactoring to achieve thread safety**

Sandia National Laboratories