SAND2011-1199C

Shared Memory and GPU Algorithms for Finite-Element Linear-System Assembly

Alan Williams
Sandia National Laboratories

SIAM CSE, March 3, 2011

Outline:

- Motivation
- MiniFE
- FE Linear-system assembly
- Threading, “fine-grain” parallelism
- Timing results
- Conclusions

Sandia National Laboratories is a multi-program laboratory operated and managed 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.

Sandia National Laboratories

Motivation

Why research shared-memory algorithms for finite-element applications when we (Sandia & community) already have a huge investment in MPI-parallel algorithms?

- **Parallel machines are trending towards having compute-nodes with many (10s to 100s) cores, and it may not be optimal or feasible to have an MPI process on each core.**
- **It may be better to have one MPI process per compute-node, and use "fine-grained" parallelism on cores within a node or on a coprocessor.**
- **It is preferable to do this research using 'mini-applications' rather than using our 'real' applications. Hence MiniFE…**

"miniFE" is a Finite-Element mini-application

Implements algorithms from an implicit finite-element application

- Assemble a sparse linear-system from the steady-state conduction equation on a domain of hexahedral elements.

$$Ku = f \qquad K = \sum_e K^e \qquad K^e = \int_{\Omega_e} \frac{\partial \psi}{\partial x_m} k_{mn} \frac{\partial \psi}{\partial x_n} dx$$

$$f = \sum_e Q^e \qquad Q^e = \int_{\Omega_e} \psi Q dx$$

- Solve the linear-system using the Conjugate Gradient algorithm:
  - Per iteration:
    - 2 dot-products
    - 3 axpys
    - 1 matrix-vector product

$r_0 = b - Ax_0$

Loop {

  If *k* == 1

    $p_1 = r_0$

  else

    $\beta_k = r_{k-1}^T r_{k-1} / r_{k-2}^T r_{k-2}$

    $p_k = r_{k-1} + \beta_k p_{k-1}$

  $\alpha_k = r_{k-1}^T r_{k-1} / p_k^T A p_k$

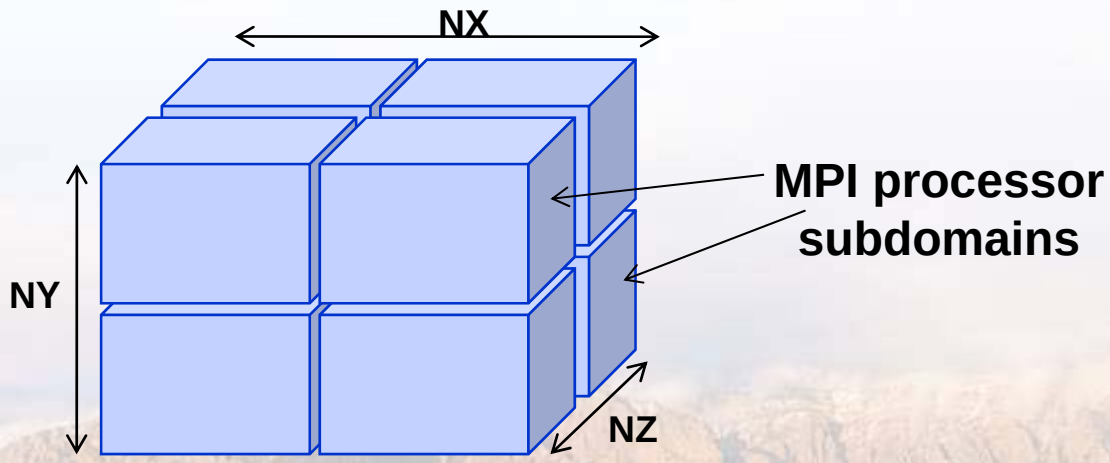
  $x_k = x_{k-1} + \alpha_k p_k$

  $r_k = r_{k-1} - \alpha_k A p_k$

}

Finite Element mini-app: miniFE

- **miniFE sets up a brick-shaped problem domain of hexahedral elements**
  - Parameters **nx, ny, nz** specify global number of elements in each dimension
  - Global number of equations in linear-system corresponds to finite element vertices (nodes): (nx+1)(ny+1)(nz+1)
- **RCB partitioning splits the global problem domain among MPI processors**
- **Vertices on processor boundaries are shared**



miniFE code origins, overview

- **Evolved from Mike Heroux's HPCCG**
  - Small, portable, self-contained implementation of linear Conjugate Gradient solve.
  - MPI parallel, with many derivative implementations that explored threading, etc.

- **Evolution to miniFE added several features:**
  - Finite-Element assembly from conduction equation
  - Optionally store mesh data in Sierra Toolkit Mesh (STK Mesh)
  - Entire code is parameterized (C++ templates) on floating-point and integer types
  - ComputeNode programming model (Baker et. al.) provides abstract interface to fine-grained parallelism (CPU threading, GPU co-processing).

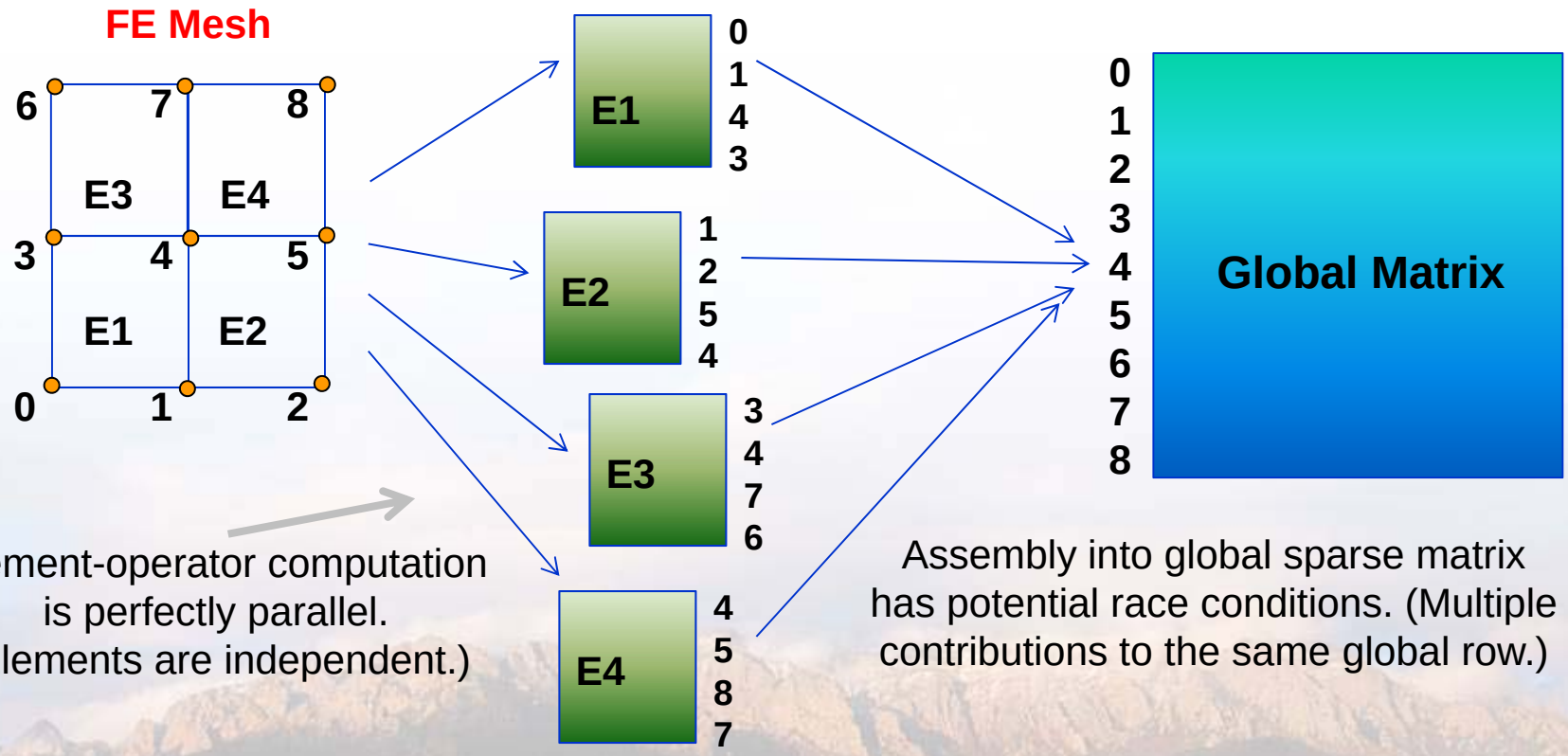
- **Previous work has focused on the parallel performance of the linear-solve; we now focus on the finite-element assembly into the linear-system.**

Description of Finite-Element linear-system assembly: for each element in the mesh, 3 operations

1. Get node-ids & coordinates

2. Compute dense element-operators

3. Scatter-assemble into global sparse linear-system



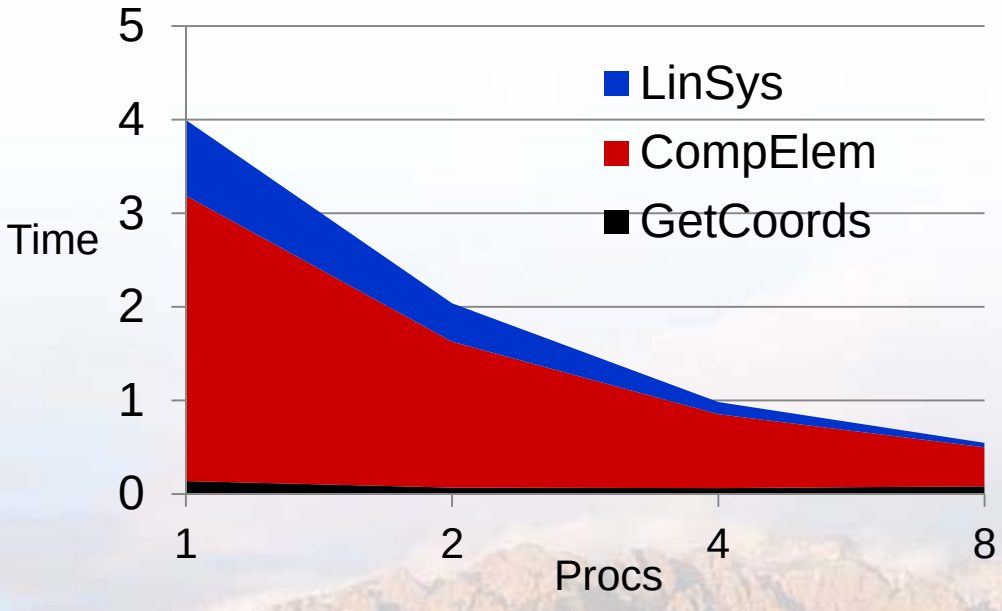
Element-operator computation is perfectly parallel. (Elements are independent.)

Assembly into global sparse matrix has potential race conditions. (Multiple contributions to the same global row.)

Base-line FE Assembly Timings: MPI parallelism, no threading

Problem size: 100x100x100 == 1 million elements
Matrix rows: 1,030,301
The finite-element assembly performs 8 million matrix-row sum-into operations (8 per element) and 8 million vector-entry sum-into operations.

Linux dual quad-core workstation.



- Assembly time is dominated by elem-operator computation.
- Scaling is nearly perfect.

Fine-grain parallelism: within an MPI process (within a multicore compute-node)

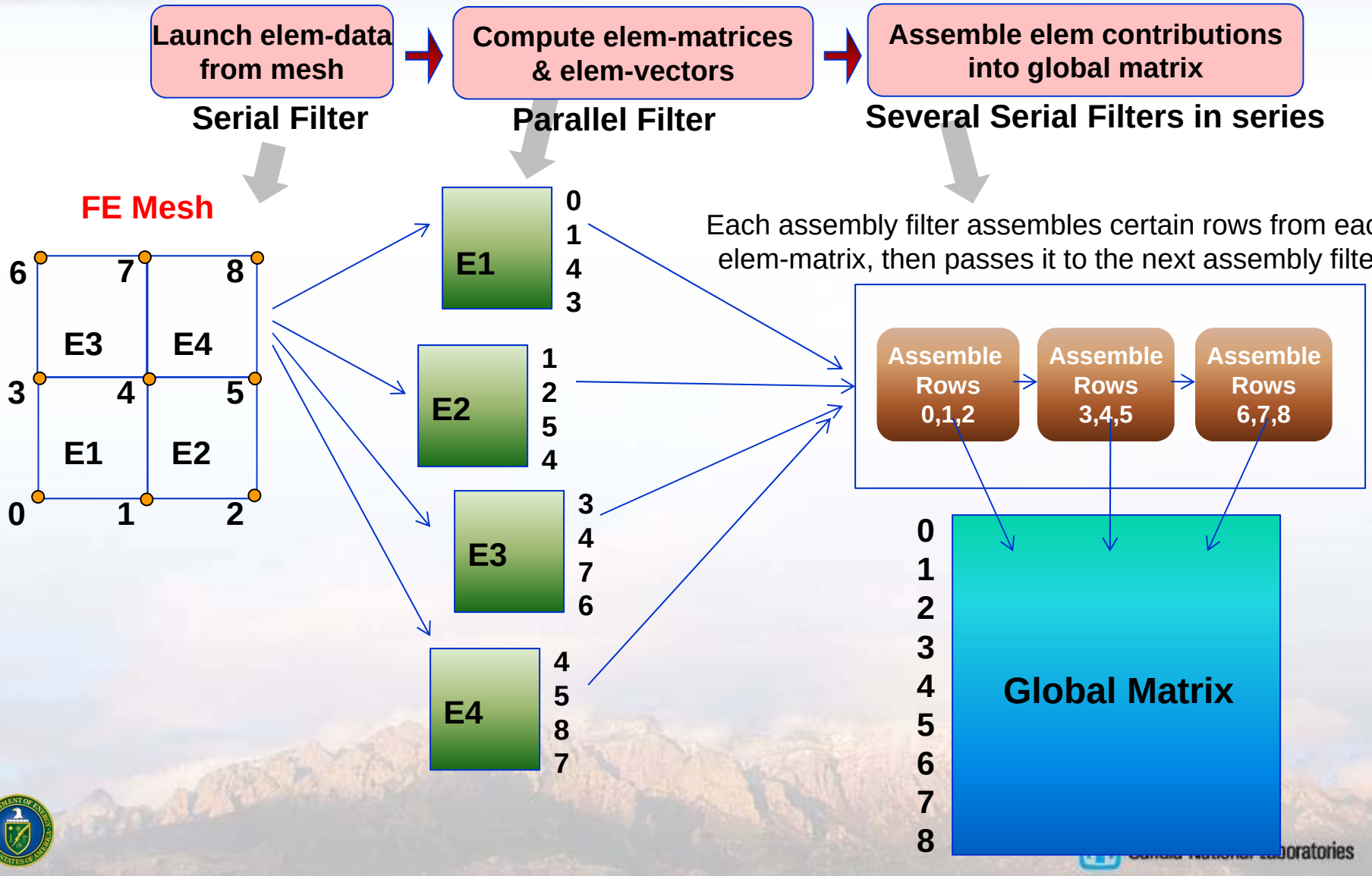
Different ways to parallelize algorithms using threading:

- Parallel for-loop
  - For-loop can be split among threads such that each thread visits a different subset of local mesh elements.

- Pipeline (from Intel Thread Building Blocks (TBB))
  - Pipelined application of a series of “filters” to a stream of “items”.
  - Think of the 3 element-computation operations as filters…

- The race-condition on matrix assembly is a significant consideration.
  - Depending on how the algorithm is organized, thread locking may be necessary to prevent simultaneous data updates from different threads.
  - A number of algorithms are considered, to compare the effects of locking.

FE assembly algorithm with no locking required: TBB Pipeline



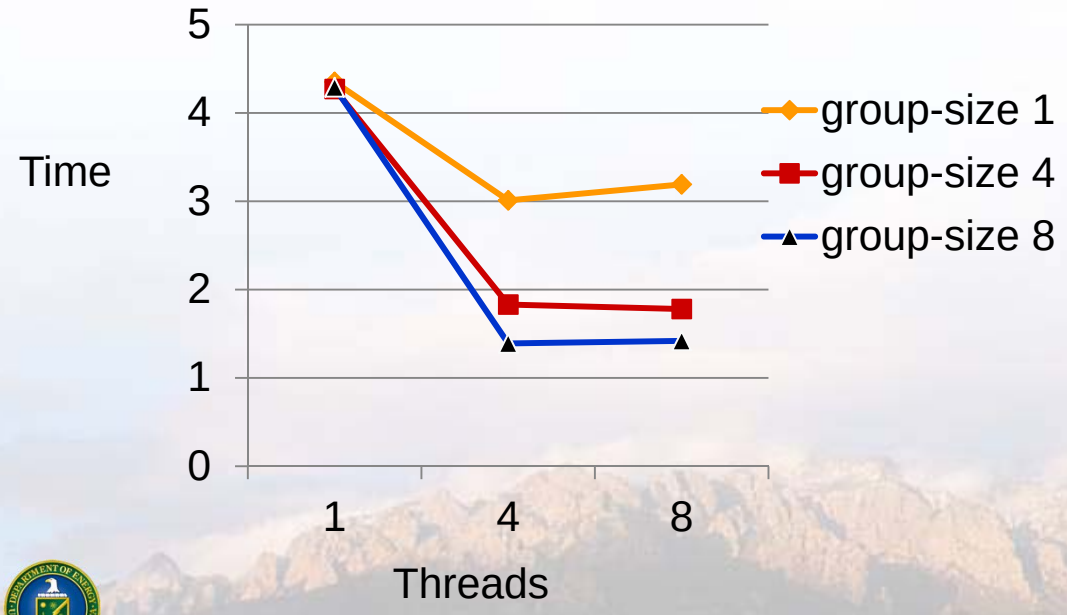
FE Assembly Timings: Intel TBB, pipeline, no locking, serial assembly filters

Problem size: 100x100x100

Elem-group-size: num-elems processed at-a-time by each filter.

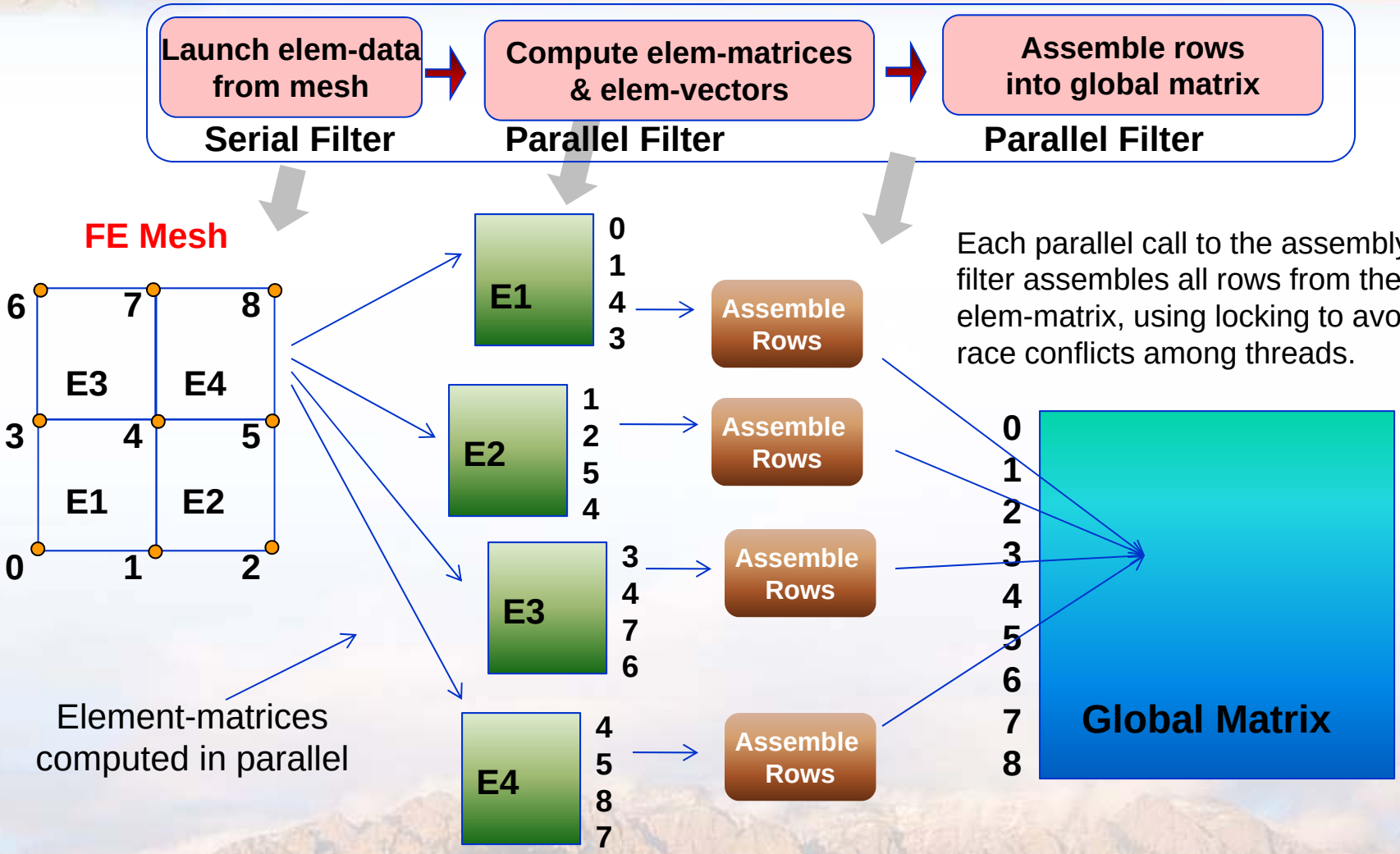
Performance not affected much by varying the number of assembly filters.

Performance doesn't scale well



Num-threads	Elem-group-size	Num-asm-filters	Assembly-time
1	1	1	4.35s
1	4	1	4.27s
1	8	1	4.29s
4	1	1	3.01s
4	1	2	3.31s
4	4	1	1.83s
4	4	2	1.84s
4	8	1	1.42s
4	8	2	1.39s
8	1	2	3.19s
8	4	2	1.78s
8	8	1	1.48s
8	8	2	1.46s
8	8	4	1.42s

FE assembly algorithm that requires locking: TBB Pipeline for FE assembly

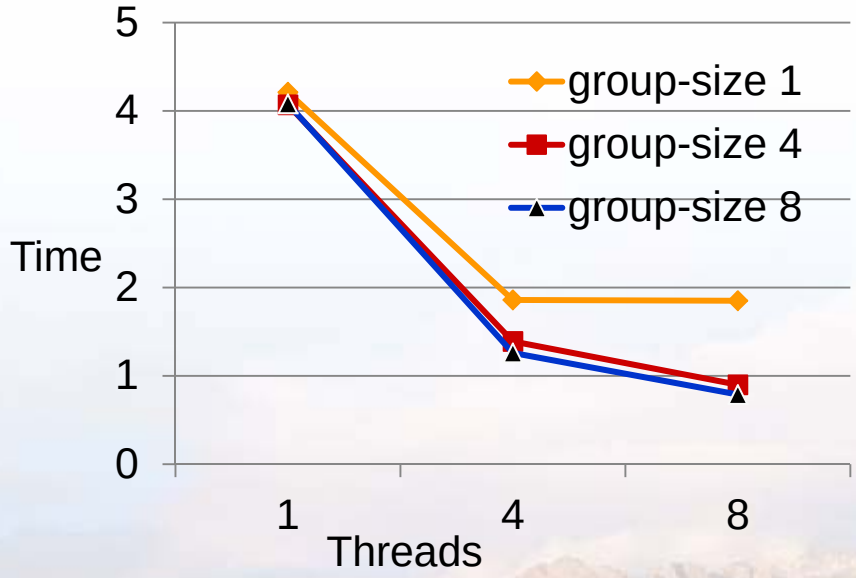


FE Assembly Timings: Intel TBB, pipeline with linear-system locking

Problem size: 100x100x100
Parallel matrix-assembly filter uses locking to protect against data races

"Matrix conflicts", "Vector conflicts" shows number of thread contentions for locks.

Processing elements in "groups" reduces thread-contention, improves scaling.



Num-threads	Elem-group-size	Matrix-conflicts	Vector-conflicts	Assembly-time
1	1	0	0	4.21s
1	4	0	0	4.07s
1	8	0	0	4.08s
4	1	173672	2036	1.86s
4	4	23710	111	1.39s
4	8	3890	5	1.26s
8	1	85838	870	1.85s
8	4	12284	94	0.90s
8	8	2162	4	0.79s

parallel-for-loop FE assembly: 3 different ways to structure the loops

1. for each element {
   get node-ids & coords
   compute elem-operators
   sum into linear-system
   }

- The most obvious.
- only need memory for 1 elem-operator (per thread), reuse during loop

2. for each element { get node-ids & coords }
   for each element { compute elem-operators }
   for each element { sum into linear-system }

- Need to allocate memory for all elements
- Better for offloading 1 operation (elem-computation) to GPU

3rd looping option shown on next slide…

parallel-for-loop FE assembly: 3rd looping option

3. for each element {
get node-ids & coords
compute elem-operators
}
for each vertex {
for each element that shares this vertex {
sum this vertex's row of the elem-operators
into linear-system
}
}

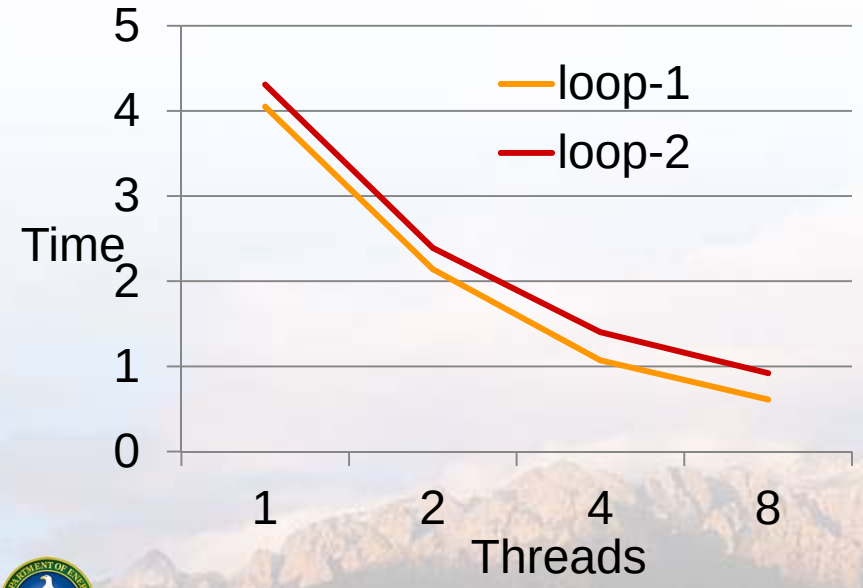
- Requires storing all elem-operators
- Requires a fancier mesh data structure (with node-to-element relations…)
  - miniFE can optionally use STK-mesh which provides these relations…
- No need for thread-locking, since vertex loop updates distinct matrix rows

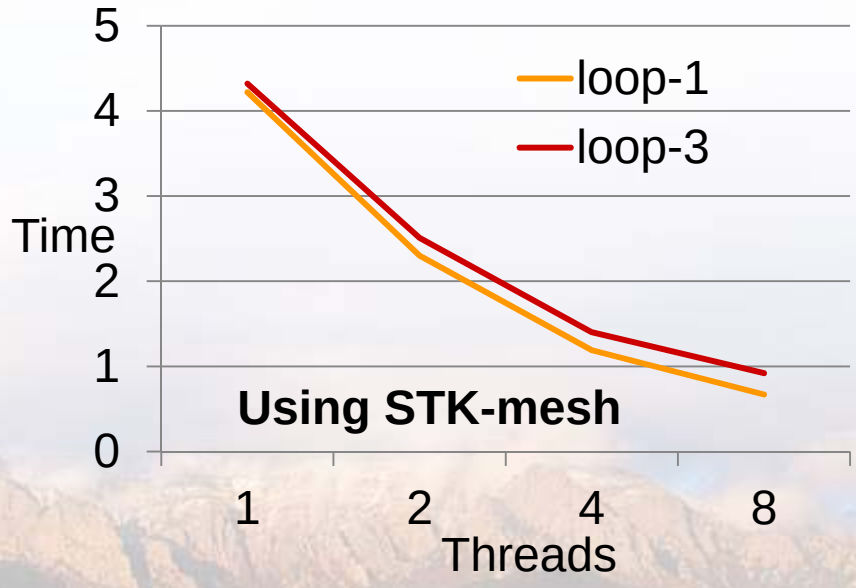
Timing results for parallel-for-loop FE assembly:

Problem size: 100x100x100
Matrix-assembly uses locking to protect against data races

No thread conflicts occurred in these cases. Threads happen to work “far apart” from each other on this large regularly-structured mesh.

Performance advantage of looping-structure 1 is probably due to better cache behavior (reusing memory for 1 elem-operator rather than allocating all).



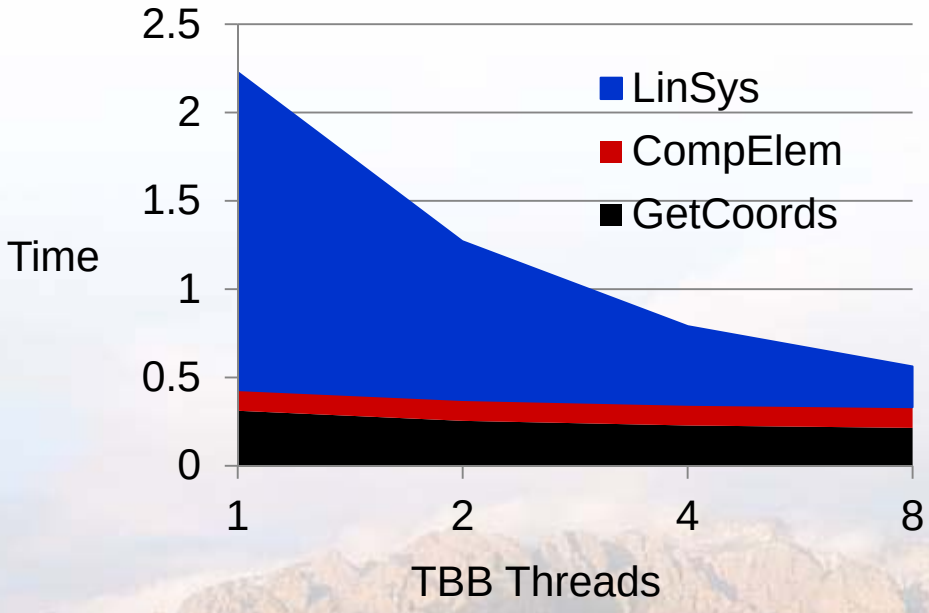


FE Assembly Timings: Intel TBB, parallel-for, with elem-matrices computed on Cuda GPU

Problem size: 100x100x100

Uses for-loop “option 2”, offloading elem-computation to GPU
Parallel matrix-assembly (on host using TBB) uses thread-locking to protect against data races

NOTE: In this case the scalar-type is float (single-precision)



- Elem-computation no longer dominates
- Threaded linsys assembly scales well, no thread conflicts in this case.

Sandia National Laboratories

Summary, Conclusions, Further work

- Parallel-for implementations scale slightly better than pipeline, but we expect a pipeline to be useful for certain hardware architectures

- Future work:
  - Develop pipeline implementation on host that orchestrates offload to devices like GPU, Intel “Knights Ferry”, etc.
  - May help hide latency to GPU if combined with asynchronous CUDA mem-copy operations…

- Thread-parallel doesn’t beat MPI-parallel for this application, at least not at these core-counts (8-core workstation)
  - We believe thread-parallel becomes more important at higher core counts (coming hardware architectures)