SAND2016-3476C

Thread parallelism in sparse linear algebra and iterative solvers

Mark Hoemmen
Center for Computing Research
Sandia National Laboratories

19 Apr 2016

Sandia National Laboratories

Exceptional service in the national interest

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

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. SAND NO. XXXX-XXXX C

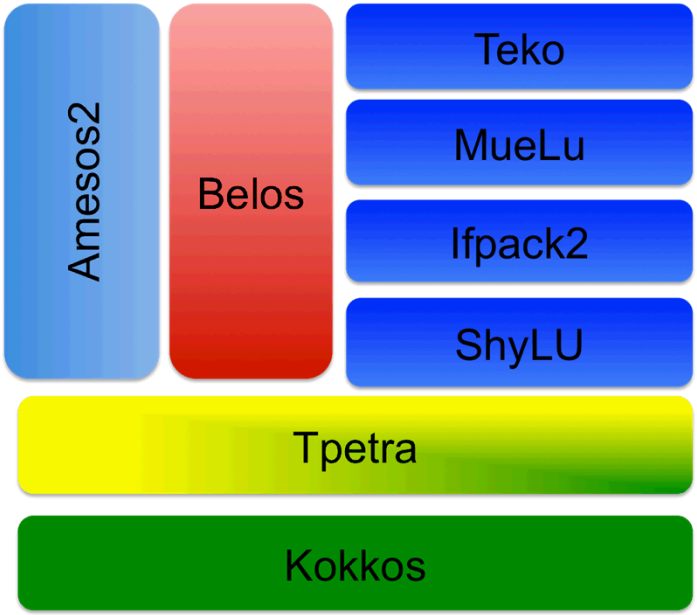
Outline

Sandia National Laboratories

- Our sparse matrix library: Tpetra (part of Trilinos project)
- Goal: Support MPI + X parallelism on current & future architectures, where X is OpenMP, CUDA, …
- Genericity (“X”) via Kokkos programming model
- Fill (creating / changing sparse matrices) interfaces complicate thread parallelization
- Current & planned thread-parallel capabilities

Trilinos' linear solvers

Sandia National Laboratories

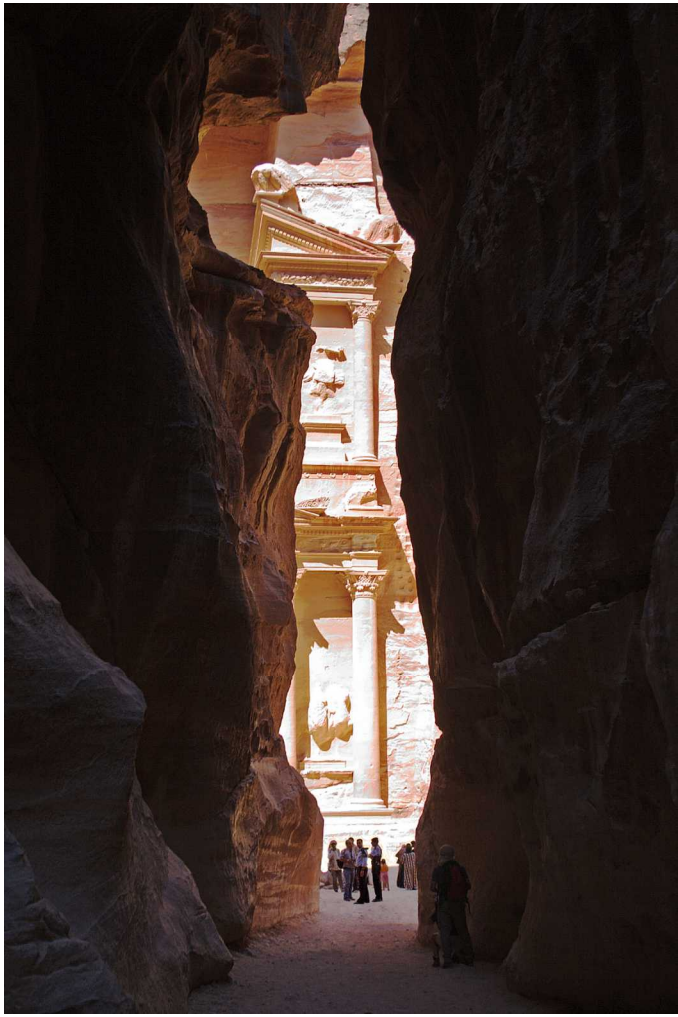


- Sparse linear algebra (Tpetra)
  - Sparse graphs, (block) sparse matrices, dense vectors, parallel solve kernels, parallel communication & redistribution
- Iterative (Krylov) solvers (Belos)
  - CG, GMRES, TFQMR, recycling methods
- Sparse direct solvers (Amesos2)
- Algebraic iterative methods (Ifpack2)
  - Jacobi, SOR, polynomial, incomplete factorizations, additive Schwarz
- Shared-memory factorizations (ShyLU)
  - LU, ILU(k), ILUt, IC(k), iterative ILU(k)
  - Direct+iterative preconditioners
- Segregated block solvers (Teko)
- Algebraic multigrid (MueLu)

Tpetra: parallel sparse linear algebra

Sandia National Laboratories

- Tpetra implements
  - Sparse graphs & matrices, & dense vecs
  - Parallel kernels for solving Ax=b & Ax=λx
  - Parallel communication & (re)distribution
- Key Tpetra features
  - Can manage > 2 billion (10^9) unknowns
  - Can pick the type of values:
    - Real, complex, extra precision
    - Automatic differentiation
    - Types for stochastic PDE discretizations
  - Center of growing support for MPI + X parallelism, for several X



Tpetra development goals

Sandia National Laboratories

- Scale from laptop to full supercomputer
- 1 implementation for all platforms & parallelism options
  - Very limited developer time (< 2 full-time staff)
  - Easier to debug solver (convergence) & performance issues
- Maintain backwards compatibility
  - Trilinos only allows breaking it at major releases (every 1-2 years)
  - Balance research, prep for new hardware, & support today's apps (often running on old hardware & software)
  - Sparse linear algebra central to apps & other Trilinos packages
  - Interfaces matter for performance & parallelism
- Exploit optimized kernels but minimize library dependencies
  - We need our own implementations that work everywhere
  - 3rd-party libraries may ignore features needed for e.g., MPI

Must support > 3 architectures

Sandia National Laboratories

- Coming systems to support
  - Trinity (Intel Haswell & KNL)
  - Sierra (CORAL): NVIDIA GPUs + IBM multicore CPUs
  - Clusters, workstations, etc.
- 3 different architectures
  - Multicore CPUs (big cores)
  - Manycore CPUs (small cores)
  - NVIDIA GPUs
- MPI only, & MPI + threads
  - Threads don't always pay on common CPU architectures
  - Porting to threads must not slow down the MPI-only case

CORAL COLLABORATION
OAK RIDGE • ARGONNE • LIVERMORE

SIERRA

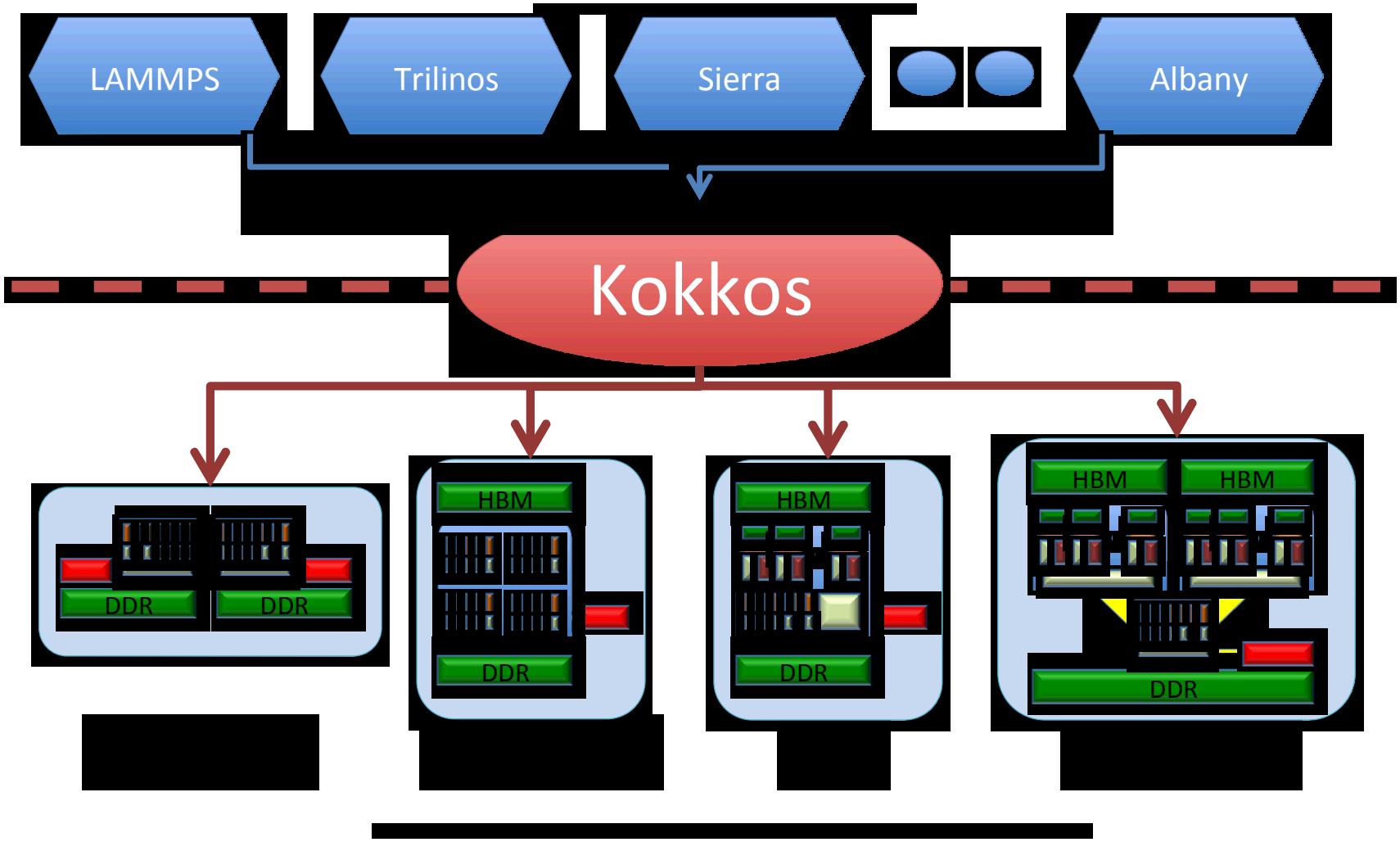
Kokkos: Common C++ - based programming model for thread parallelism on GPUs, CPUs, …

Sandia National Laboratories

- Parallel {for, reduce, scan} w/ custom user code
- Exposes different levels of parallelism
  - Flat [0,N), or hierarchical (team, thread, vector)
  - Experimental task parallelism too!
- Different memory & execution spaces
  - Control where data live & code executes
  - Enable “hybrid” (host + GPU) parallelism
- Multidimensional arrays (Kokkos::View) w/ slices
  - Decouple array layout (row/column-major, tiled, …) from app
  - Default layout optimized for the architecture (SoA / AoS)
  - Unified interface to shared memory, texture fetch, atomic access, …
- Goal: write code once, run well on many different back-ends

Kokkos: *Performance, Portability, & Productivity*

Sandia National Laboratories

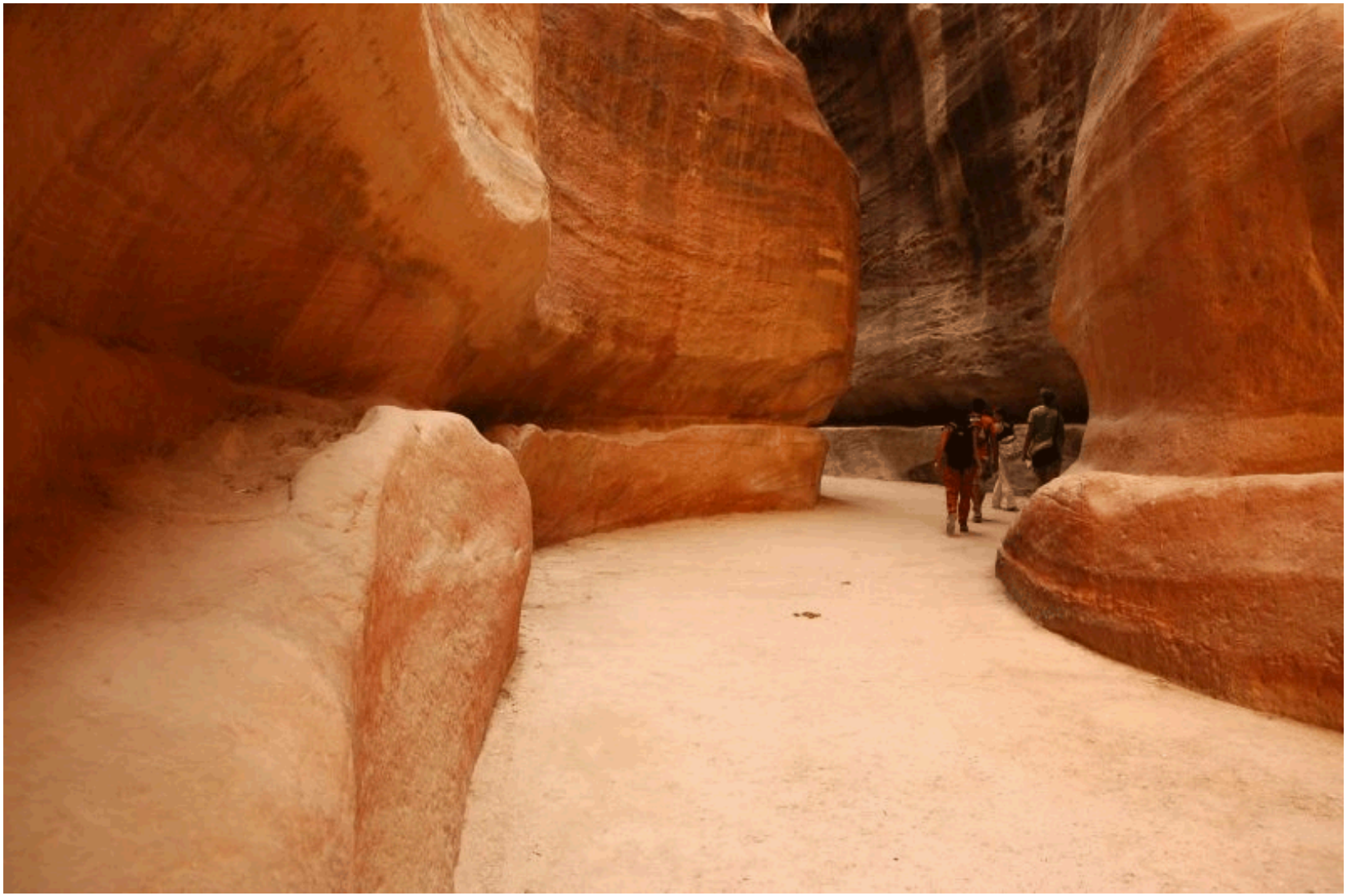


Kokkos as hedge against…

- Hardware divergence
- Parallel programming model
  - OpenMP, OpenACC, CUDA, TBB, Pthreads, Qthreads, …
- Traditional shared memory
  - vs. PGAS / distributed shared
- Threads at all
  - "Serial" back-end
  - Kokkos' semantics require vectorizable (ivdep) loops
- Kokkos protects our HUGE time investment of porting Tpetra

Why is thread parallelization hard?

Sandia National Laboratories

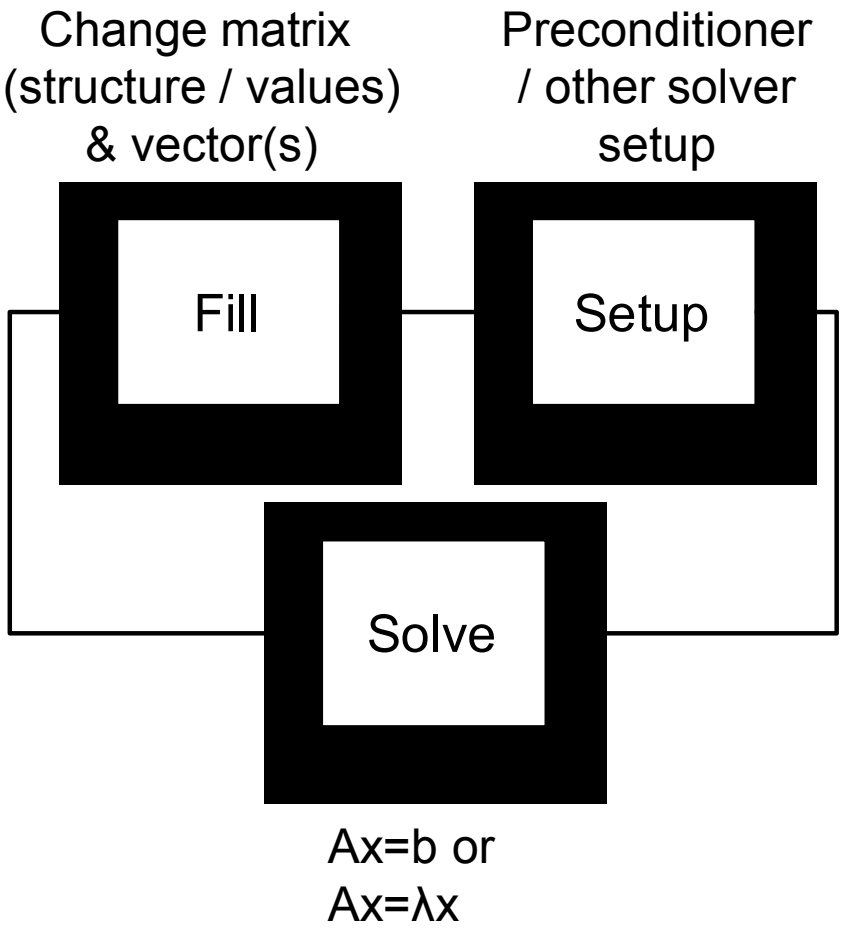


Sparse linear algebra use pattern

Sandia National Laboratories

- Fill: Create / modify matrix & vector data structures
  - As many ways to do this as there are applications
  - e.g., iterate over rows, entries, mesh points, elements (FEM), volumes (FVM), aggregates (AMG), …
  - Software interfaces affect performance A LOT
- Setup for solve (e.g., build preconditioner)
- Solve linear system(s), eigenvalue problems, etc.
  - Coarse-grained computational kernels (e.g., sparse mat-vec)
  - Software interfaces affect performance less
- Repeat (nonlinear iteration, time steps, parameter study, …)
  - Trilinos data structures & solvers optimized for reuse, e.g., of
    - Data structures (graph, basis vectors, allocations) &/or
    - MPI communication patterns (where to send / receive what)

Need thread-parallel fill



- Fill & setup not free
- Some solves are cheap, so fill & setup time matter
- Amdahl's Law:
  - Threading just solves makes cute, easy-to-publish papers
  - Solves do take most app time
  - But: 90% time w/ 1 thread ➔ 50% time w/ 10 threads
- Preconditioners create sparse matrices, so they also need fill

Fill makes gradual porting hard

Sandia National Laboratories

- Iterative linear solves in practice
  - Do NOT need 1000s of iterations
  - ARE preconditioned, often nontrivially
  - Occur in context of nonlinear solves / time stepping (matrix changes!)
  - May take lots of memory
- Implications for linear algebra data structures
  - Fill interfaces affect performance
  - Prefer standard data structures to avoid reimplementation & copying
  - Can't just plug in kernels (e.g., sparse matrix-vector multiply)
- I'm not excited that you made sparse matrix-vector multiply 5% faster with your funny new data structure

Thread-parallel fill interface options

Sandia National Laboratories

Coarse-grained (batched)

- Pass many items into linear algebra interface at once
- Library parallelizes inside
- (+) Need not be thread safe
- (+) Hides complexity
- (-) Hard for gradual porting
- (-) No cross-kernel reuse
- (-) Need enough parallelism to keep whole device busy

Fine-grained (Tpetra prefers)

- 1 item at a time
- User parallelizes outside
- (-) Interface must be thread safe & scalable
- (-) Limited parallelism inside
  (+) Easy to add to existing code – even pre-parallel
- (+) Users can exploit reuse
- (+) Works w/ team/thread

How we ported, & current progress

Sandia National Laboratories

A brief history of Tpetra

Sandia National Laboratories

- 2008: Tpetra started becoming usable
- 2009: Explorations of thread parallelism (computational kernels only, no fill interfaces)
- 2009-2010: Initial efforts at preconditioners
- I started working on Tpetra in late 2010
- "Productionization" (several staff): 2011-2013
  - Fix bugs & improve performance of (single-threaded) solvers & fill
  - We integrated into an internal engineering numerical simulation
  - Fruits of our effort in Nalu: https://github.com/spdomin/nalu
- "Kokkos refactor" (1.5 full-time staff): Late 2013 – present
  - Stage 1 (FY14-15): Keep interface, replace data structures & kernels
  - Stage 2 (FY15-now): Continue kernels work; evolve fill interface

Keys to gradual porting

Sandia National Laboratories

- Abstract away memory allocation & deallocation
  - Kokkos::View (multidimensional arrays) as building block
  - Kokkos manages deallocation automatically via ref counting
    - Thread safe; table of references on host, updated via host atomics
    - Off inside parallel_*; can turn off per array (handy for slices in loops)
  - Handles NUMA first-touch initialization too
- Abstract away data-parallel loops & computational kernels
  - Loops: Kokkos::parallel_{for, reduce, scan}
    - I write loop body as functor or C++11 lambda (new CUDA feature)
    - Kokkos semantics force me to write vectorizable & parallelizable loops
  - Computational kernels separated into “KokkosKernels” package
- Manage data movement between memory spaces
  - Kokkos’ abstractions make everything look like GPU
  - CUDA UVM means I can port one kernel at a time

Pattern for parallel dynamic allocation

Sandia National Laboratories

- Pattern:
  1. Count / estimate allocation size; may use Kokkos parallel_scan
  2. Allocate; use Kokkos::View for best data layout & first touch
  3. Fill: parallel_reduce over error codes; if you run out of space, keep going, count how much more you need, & return to (2)
  4. Compute (e.g., solve the linear system) using filled data structures
- Compare to Fill, Setup, Solve sparse linear algebra use pattern
- Fortran <= 77 coders should find this familiar
- Semantics change: Running out of memory not an error!
  - Generalizes to other kinds of failures, even fault tolerance

Fill in 2012 was not thread-scalable

Sandia National Laboratories

- Dynamic memory allocation ("dynamic profile")
  - Impossible in some parallel models; slow on others; implies sync
  - Better: Count, Allocate (thread collective), Fill, Compute
- Throw C++ exceptions on error / when out of space
  - Either doesn't work (CUDA) or hinders compiler optimization
  - Prevents fruitful retry in (count, allocate, fill, compute)
  - Better: Return success / failed count; user reduces over counts
- Unscalable reference counting implementation
  - Tpetra's interface relied heavily on something like std::shared_ptr
  - Not hard to make thread *safe*, but updating the ref count serializes!
  - Returning std::shared_ptr (or our thing) updates ref count
  - Better: Hide ref counting inside; make objects have "view semantics"
- Had to fill Tpetra data structures on host (copy), sequentially

Refactor plan: Stage 1 (FY14-15)

Sandia National Laboratories

- Replace all internal data structures & kernels w/ Kokkos
  - Sparse matrix-vector multiply & vector ops first
  - Later, we factored out local kernels into KokkosKernels
  - Tpetra already had (de)allocation abstraction ("smart pointers"), so it was easier to introduce Kokkos' arrays & CUDA device allocations
- Assume CUDA UVM so only have to port 1 function at a time
  - Pain point: UVM allocations can't coexist w/ device kernels
- Thread-parallel fill into Kokkos, & hand off to Tpetra
- C++ partial specialization let pre- & post-refactored versions of Tpetra coexist – users could select which at compile time
- Old version built with older compilers (no need for C++11)

Stage 2 (FY15-16): Thread-safe fill

Sandia National Laboratories

- Done for CrsMatrix & (Multi)Vector, for methods that
  - Don't change graph structure (no "insert" yet)
  - Don't cause MPI communication (+= values for off-process rows)
- Return error code / success count; don't throw on error
- No more internal temporary array dynamic allocation
- Atomic update option for methods that do += to values
- Creating / modifying sparse graph
  - Fill into Kokkos data structures, & hand off to Tpetra
  - Tpetra has example showing thread-parallel iteration over finite-element mesh to create graph structure of sparse matrix
  - Tpetra interface to simplify this is in progress, lower priority

Hierarchical parallelism

Sandia National Laboratories

Under development: KokkosKernels

Sandia National Laboratories

- Local computational kernels used by Trilinos, usable outside
  - Dense & sparse matrix, graph (e.g., coloring), & tensor kernels
  - Local (no MPI) – Trilinos / users responsible for MPI
  - No required software dependencies other than Kokkos
  - Hooks for 3rd-party libraries like cu{Blas,Sparse} if available
- Multi-year effort w/ many contributors, mostly Trilinos devs
- Provide kernels for all levels of hierarchical parallelism:
  - **Global:** all available execution resources (e.g., whole GPU)
  - **Team:** single block / team, use shared memory
  - **Thread:** single "thread" (/ warp), vectorization inside
  - **Serial:** "elemental" function (`omp declare simd`)

Why kernels for different levels?

Sandia National Laboratories

- Many apps do many small computations in parallel
  - How often do real apps need 10k x 10k DGEMM? (very rarely)
  - e.g., Small dense matrix operations (BLAS & factorizations)
    - PDE discretizations w/ multiple unknowns per mesh point: ~10 x 10
    - Multifrontal sparse matrix factorizations: ~100 x 100 (NOT square)
- CUDA, OpenMP 4, OpenACC all expose (team, thread)
- Remember coarse (batched) vs. fine-grained fill?
  - Lets users exploit locality & amortize kernel launch overhead
  - Matches what many e.g., finite-element codes already do
    - “Worksets,” “bucket loop”
    - Break loop into data-parallel chunks, sized to fit in cache
    - Do many operations to each chunk before moving on

Multiple memory spaces?

Sandia National Laboratories

>1 memory or execution spaces

Sandia National Laboratories

- Upcoming NNSA platforms
  - Trinity (KNL): 2 memory spaces (HBM, DDR4)
  - CORAL (Sierra): 2 execution & memory spaces (NVIDIA GPUs, IBM multicore CPUs)
- Common hardware features
  - 2 memory spaces: “fast & small” vs. “slow & big”
  - Can access each memory space from each exec space (acts like NUMA)
  - “Fast” memory limited; must use as temporary workspace

- Support via some comb of {Kokkos, Tpetra, solvers, app}
- Use cases to support
  - Gradual port (mix new & legacy)
  - Concurrently use 2 exec spaces (e.g., MPI pack & compute)

Strategies for limited GPU memory

Sandia National Laboratories

- All app data, even whole linear system, may not fit on GPU
- Prefer algorithmic solutions over auto-magic
  - Don't want different libraries to need to arbitrate limited resource
  - Painful run-dependent debugging & performance variation
- Strategy 1: Stage in individual linear systems temporarily
  - Real physics is multiphysics ➔ solve multiple linear systems at same time
  - Works with block preconditioners or nonlinear (loose) coupling
  - Tpetra's current interface could support this, w/ more impl work
- Strategy 2: Domain decomposition (divide up single solves)
  - Affects convergence; doesn't work well for all linear systems
  - Subdomain solvers need enough reuse to amortize data transfer
  - Would take more software work to avoid e.g., data reformatting

Next steps

Sandia National Laboratories

Questions left to answer

Sandia National Laboratories

- Dependence on atomic updates (esp. +=)
  - Easy way to parallelize sequential loops (e.g., finite-element assembly)
  - Some uncertainty about their efficiency on different hardware
  - Algorithmic fixes take extra memory (store all terms before summing)
- No single pattern for exploiting hybrid (CPU+GPU) parallelism
  - Kokkos exposes hybrid parallelism, but Tpetra + users must exploit it
  - Overlap communication & computation?
  - Treat GPU (+ 1 CPU core?) as another MPI process?
  - Auto-magic load-balancing run-time scheduler?
- For full network bw, may need >1 threads/node using MPI
  - MPI_THREAD_MULTIPLE or its proposed MPI > 3 successors
  - MPI 1-sided avoids locks, but only works well on CPUs
  - No obvious single programming model for all architectures

Thanks!

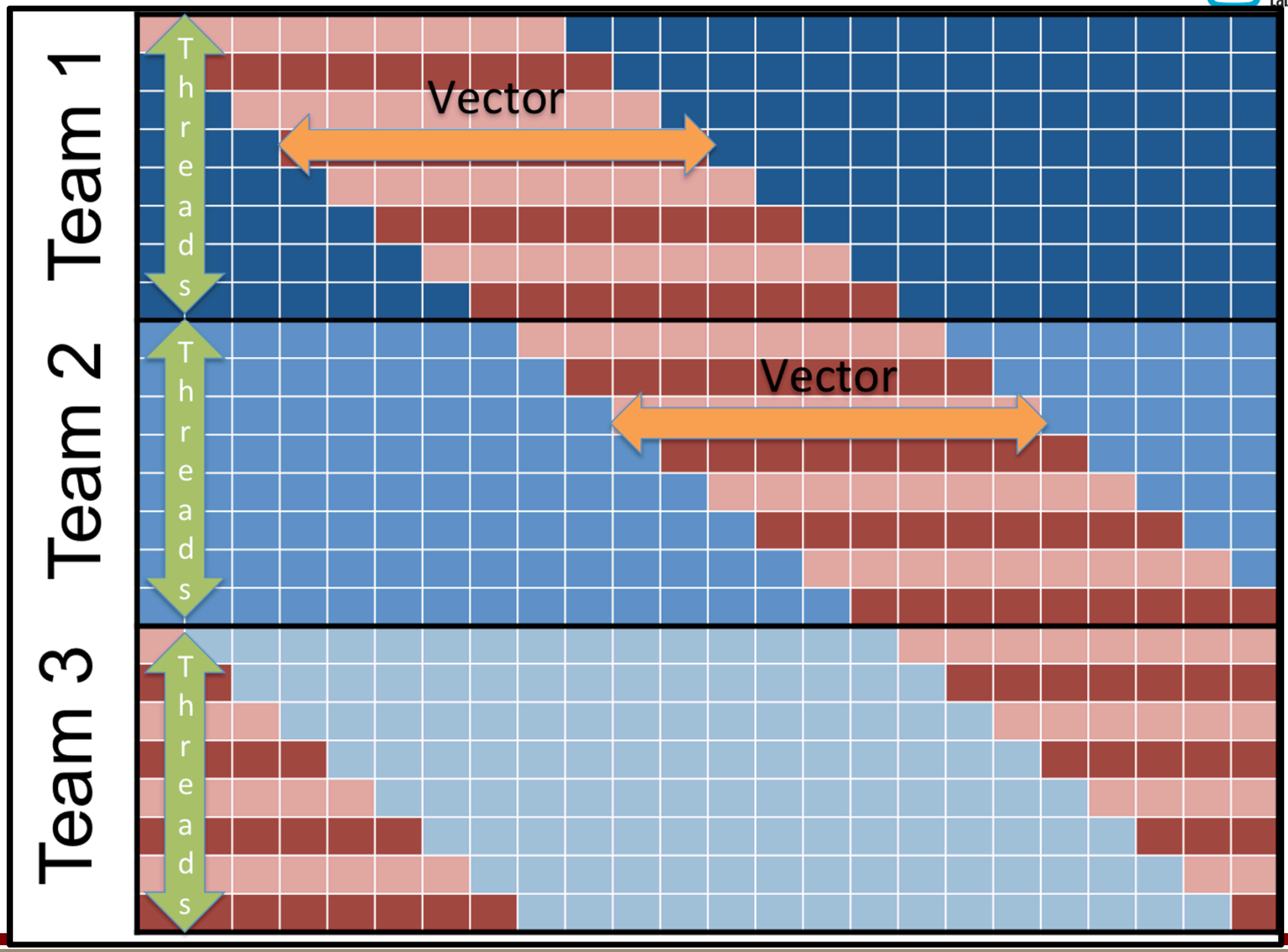
Sandia National Laboratories

- Trilinos' thread parallelism has been / still is a HUGE effort
- Dozens of colleagues & collaborators have contributed
- Super thanks to Christian Trott for LOTS of Tpetra help!

Extra slides

Sandia National Laboratories

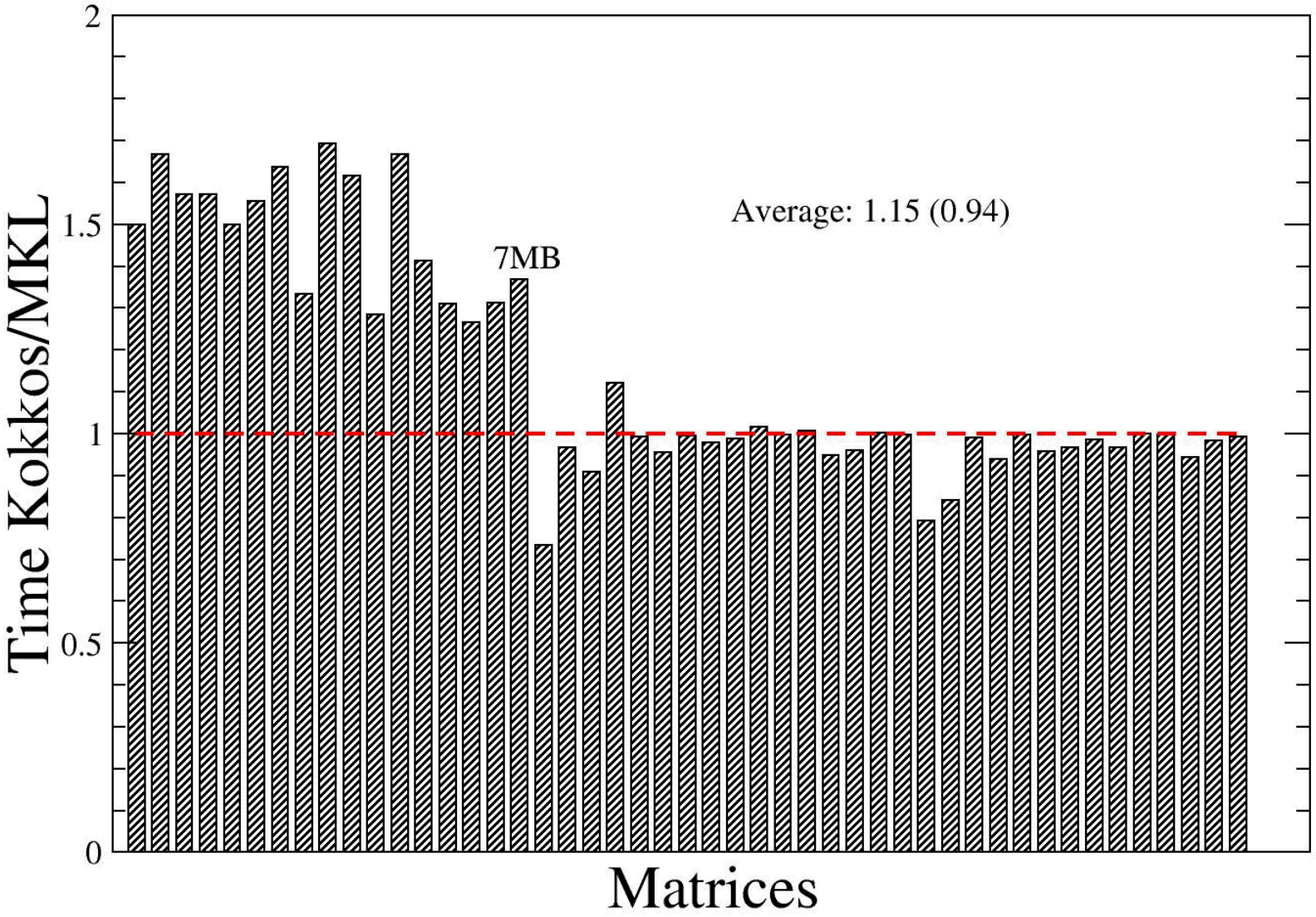
SPMV – Using Hierarchical Parallelism

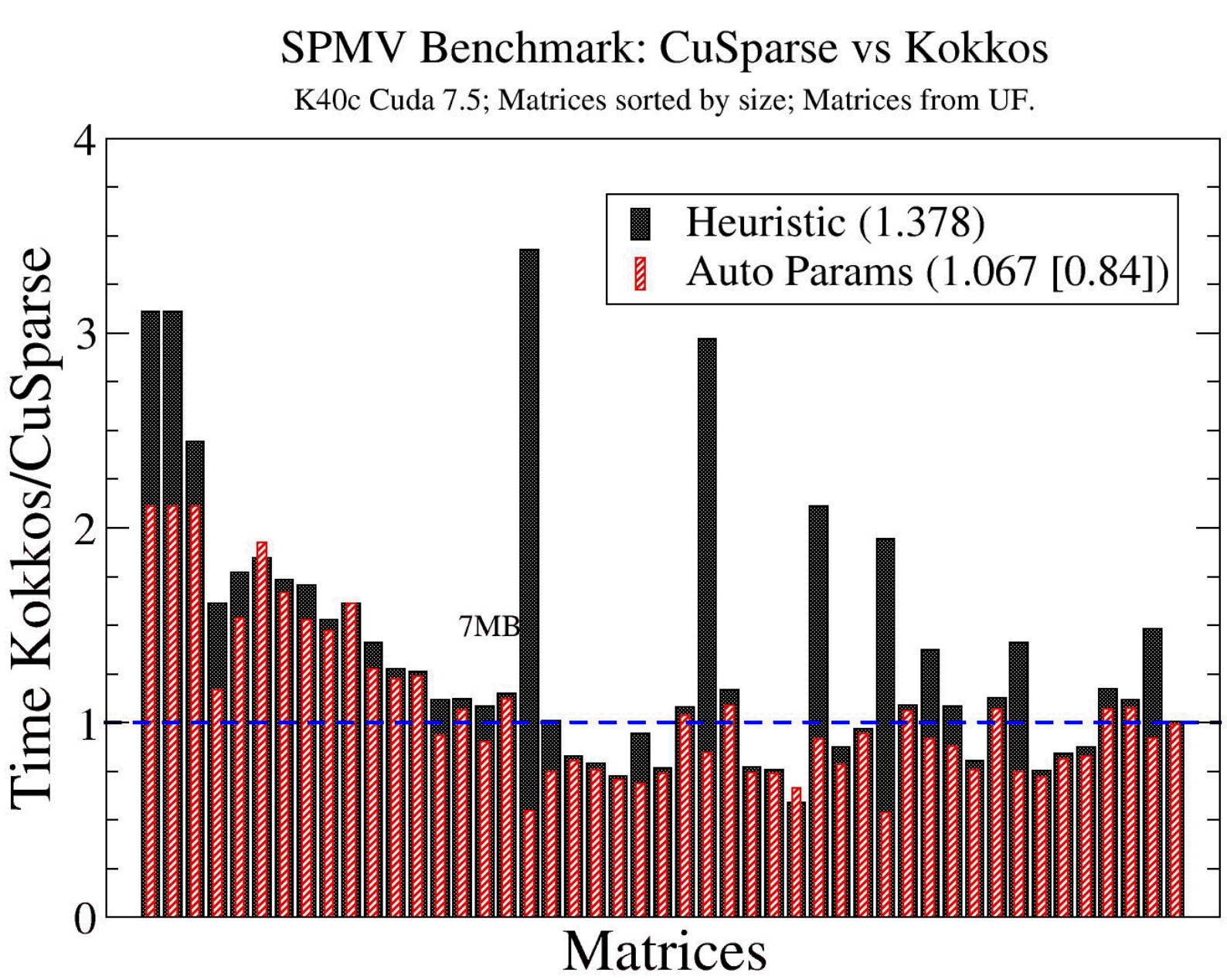


SPMV – Using Hierarchical Parallelism

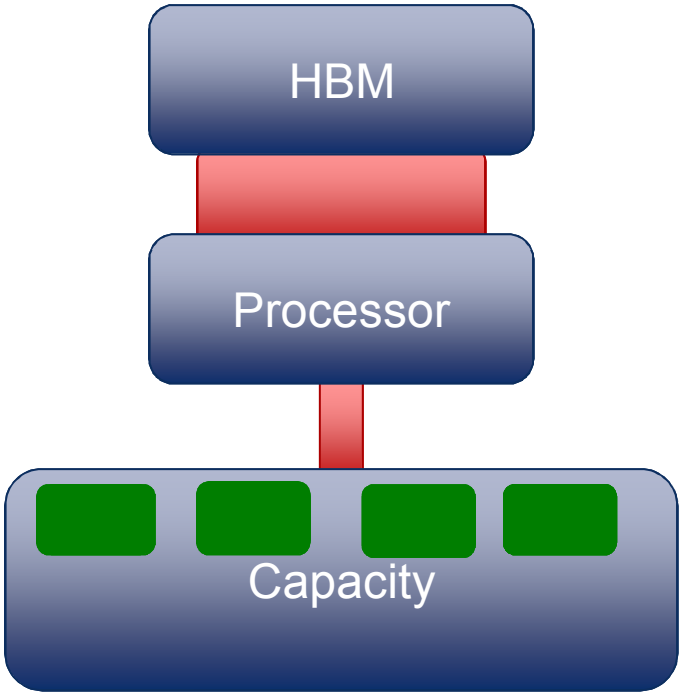
Sandia National Laboratories

```
void spmv(Matrix A, Scalar alpha, XType x, Scalar beta, YType y) {
 int nnz_per_team = 2048;
 int conc = execution_space::concurrency();
 while((conc * nnz_per_team * 4> A.nnz())&&(nnz_per_team>256)) nnz_per_team/=2;

 int nnz_per_row = A.nnz()/A.numRows();
 int rows_per_team = (nnz_per_team+nnz_per_row-1)/nnz_per_row;
 int vector_length = GetVectorLength(A);
 const int nworkset = (y.dimension_0()+rows_per_team-1)/rows_per_team;

 parallel_for(TeamPolicy<Schedule<Dynamic> >(nworkset, AUTO(),vector_length),
  KOKKOS_LAMBDA (const TeamPolicy<>::member_type& team)  {
  const int startRow = team.league_rank() * rows_per_team;
  const int endRow = startRow + rows_per_team < A.numRows() ?
                startRow + rows_per_team : A.numRows()

  parallel_for(TeamThreadRange(team,startRow,endRow), [&] (const int& loop) {
   const SparseRowViewConst<MatrixType,SizeType> row = A.template rowConst<SizeType>(iRow);
   const int row_length = row.length;
   Scalar sum = 0;

   parallel_reduce(ThreadVectorRange(team,row_length), [&] (const int& iEntry, Scalar& lsum) {
    const Scalar val = conjugate ?
          ATV::conj (row.value(iEntry)) :
          row.value(iEntry);
    lsum += val * x(row.colidx(iEntry));
   },sum);

   single(PerThread(team), [&] () {
    sum *= alpha;
    y(iRow) = beta * y(iRow) + sum;
   });
  });
 }
```


SPMV Benchmark: MKL vs Kokkos
1S HSW 24 Threads, Matrices sorted by size, Matrices obtained from UF
Time Kokkos/MKL
2
1.5
1
0.5
0
Average: 1.15 (0.94)
7MB
Matrices


SPMV Benchmark: CuSparse vs Kokkos
K40c Cuda 7.5; Matrices sorted by size; Matrices from UF.
Heuristic (1.378)
Auto Params (1.067 [0.84])
7MB
Time Kokkos/CuSparse
Matrices
0
1
2
3
4

GPU / High-Bandwidth Memory



Cost Estimate (Bandwidth Bound):

Run From Main (capacity) Memory

Time = N_{iter} * Size / $BW_{Capacity}$

Run From HBM

Time = N_{iter} * Size / BW_{HBM} + Size / $BW_{Capacity}$

Expect

$BW_{HBM}/BW_{Capacity}$ ~ 5-20

Question: Generally need higher parallelism to achieve BW_{HBM} vs $BW_{Capacity}$

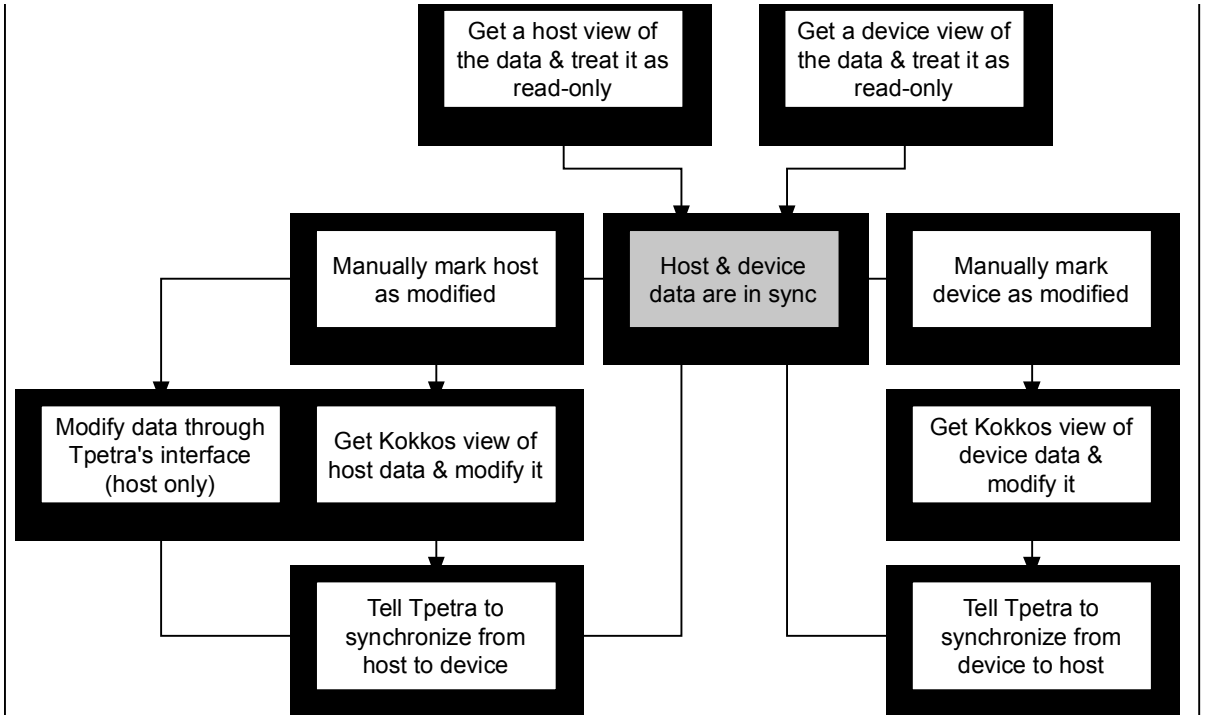
=> What about Direct Solvers?

Tpetra objects are “DualViews”

Sandia National Laboratories

- 2 memory spaces (“Host” & “Device”)
- 1 *preferred* execution space (associated w/ Device)
- 1 “host” execution space (associated w/ “Host” memory)
- Tpetra *may* execute in another space
  - e.g., overlap (un)pack of communication buffers, w/ computation
- User sets “modified” flags & “syncs” explicitly between spaces
- Successful use in LAMMPS (interactions btw user vs. GPU modules)

“DualView” example: Vector



Tpetra objects act just like Kokkos::DualView.
Tpetra’s evolution of legacy fill interface is host only.
To fill on (CUDA) device, must use Kokkos interface.

If you only have one memory space, you can ignore all of this; it turns to no-ops.

Preferred use with two memory spaces:

1. Assume unsync’d
2. Sync to memory space where you want to modify it (free if in sync)
3. Get & modify view in that memory space
4. Leave the Tpetra object unsync’d