SAND2014-2282C

Plans for Tpetra

Mark Hoemmen

April 2014

Acknowledgments

- Sandia National Laboratories is a multiprogram 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.
- Thanks to fellow Kokkos and Tpetra developers, especially
  - Christian Trott
  - Dan Sunderland
  - Carter Edwards
  - Eric Phipps
  - Siva Rajamanickam

What is Tpetra?

- Parallel sparse {graphs, matrices} & dense vectors
- Parallel data redistribution facility
- Works with Trilinos' {linear, eigen}solvers
- Features:
  - Choice of value type
    - Complex arithmetic & extended precision
    - (128-bit real works including multigrid)
    - Automatic differentiation & uncertainty quantification
  - 64-bit global indices (only if you want)
  - MPI+X (topic of this talk)

History of Tpetra

- $\approx$ 2000: Petra split: E(ssential), T(emplated)
  - E: We had to support a pre-C++98 compiler
  - T became "experimental" (mostly neglected)
- 2008 revision (Chris Baker)
  - MPI+X sparse mat-vec & vector ops
  - *No* thread-safe matrix / vector fill
  - Generic thread (+X) programming model
  - Inspired Kokkos (Carter Edwards et al.)
- 2011-3: Sierra / Tpetra integration (done May 2013)
  - Tpetra solvers into Sierra Thermal Fluids prototype
  - Tpetra had users; Sierra first to demand performance
  - MPI only (no +X), but made Tpetra correct & fast
- 2012-4: Kokkos programming model
  - Basis for Tpetra's cross-platform +X support
  - Parallel dispatch & optimal data layout

General +X model: "count, allocate, fill, compute"

1. Count (in parallel)
   - e.g., # entries per matrix row
   - Kokkos' FENL prototype demonstrates
   - "Fill" can catch underestimate
2. Allocate (outside of parallel region)
   - Dynamic allocation necessarily synchronizes
   - Prefer large arrays; avoid "array of pointers"
3. Fill (in parallel)
   - Compute and fill pre-allocated data structures
   - Reduce over error codes (e.g., ran out of room)
   - Repeat Steps 2-3 if you run out of space
4. Compute with filled data structure
   - Optimized sparse matrix-vector multiply
   - Input to factorization or preconditioners

Current Tpetra interface not thread-{safe, scalable}

- Issue: *fine-grained* ops (e.g., change one matrix entry)
  - Small ops over which *you* want to parallelize
  - *Not* big ops that hide parallelism *inside*
- Fine-grained dynamic memory allocation
  - Impossible on some devices; slow on others (Phi)
  - Prefer: count, allocate, fill, compute
- C++ exceptions to signal "out of space"
  - Either don't work (on CUDA) or hinder compiler
  - Instead:
    - Fine-grained ops return error code
    - Parallel reduce over returned codes
- Non-thread-safe reference counting
  - Teuchos memory management classes
  - Good: Encapsulate storage
  - Bad: Not thread safe, hard to make scalable

Proposed Tpetra interface changes (1 of 2)

1. Make fine-grained ops thread-{safe, scalable}
   - Global-to-local index conversion
   - Change matrix or vector entries
     - Atomic update option
   - Construct sparse graph
2. Move from container to view semantics
   - Container: deep copy, like `std::vector`
   - View: shallow copy, like a pointer
   - Tpetra currently has container semantics
     - To share Tpetra objects without copying, forces use of nonscalable ref-counted pointers (`std::shared_ptr` / `Teuchos::RCP`) that complicate interface
   - Kokkos has thread-scalable reference counting
     - No effort for Tpetra to leverage this

Proposed Tpetra interface changes (2 of 2)

3. "DualView" semantics: Host mirrors device
   - Simplify app's gradual port to GPU
     - e.g., fill on host, solve on GPU
   - User must explicitly synchronize
     - Kokkos: No hidden costs!
     - UVM => no explicit synch needed
4. Tpetra vs. "device" interface
   - Default Tpetra interface
     - DualView *must* have host interface (Else how does it know whether to modify device or host mirror?)
   - Can access "device object"
     - Its data live in that device's memory
     - Restricted interface suited for all devices
     - Less abstraction; easier for compiler
   - Gradual migration path for Tpetra and users

Timeline

1. Kokkos programming model mature (summer 2013)
2. Proven Kokkos +X sparse matrix kernels (2012,3)
3. Preserve Tpetra interface, with Kokkos backend
   - Done end Feb 2014
   - Easy to test Tpetra and downstream solvers
   - Passes tests including algebraic multigrid!
4. Start changing Tpetra interface
   - Existing interface preserved through partial template specialization (must opt in explicitly)
   - Map (global-to-local index lookup): DONE
   - Graph, matrix, vector: PROTOTYPED
   - Summer 2014: "device" interface done
   - Preconditioner setup will take longer

Simplifying assumptions

- MPI can read GPU (CUDA) device memory
  - True since mid-2012
  - We can work around not having this
- UVM (CPU can read GPU memory and vice versa)
  - May improve performance (esp. for sparse access)
  - Simplifies Tpetra porting; limited use
- C++ (98, not C++11)
  - Approach depends on templates
  - Works fine with many different compilers
  - Analogous approaches in other languages need
    - Language extensions, or
    - Code generation outside the language, or
    - Some other "code that writes code" technique
  - C++11 *not* required but would help users
    - Type inference would save lines of typedefs
    - Lambdas simplify loop body injection
  - Traits help cross-platform programming

Unforeseen issue: Compilation time

- Not just about C++ templates
- Kernel specializations for performance
  - Sparse mat-vec: b - A*x (not just A*x)
  - Sparse mat-vec with multiple right-hand sides
- We match or outperform third-party libraries
- Suggested solutions
  1. Configure option to enable specializations selectively
     - Risk of not testing rarely used options
  2. Refine #1 by collecting application statistics
     - Identify rarely used and marginal specializations
  3. Extreme: Just-in-time compilation
     - Generate code for a specialization at run time, only if the application asks for it
     - See e.g., UC Berkeley SEJITS project
     - Poor language and system support
     - Great idea; not feasible now

Unresolved questions

- Outer loop SIMD?
  - Use SIMD to solve multiple problems at once
  - Entirely different software organization
- Changing MPI support for threads?
  - Mismatch between network and compute hardware
  - May call still for multiple MPI processes per node
  - MPI standard may change to make $\text{MPI}_{\text{THREADMULTIPLE}}$ viable
- Is global shared memory the right on-node model?
  - Memory allocation forces global synchronization
  - Does not force 1-to-1 data-to-thread binding
  - PGAS (partitioned global address space) language?
    - e.g., UPC, Fortran 2008 (with co-arrays)
  - 1-to-1 data-to-thread binding not always wise
    - Prevents dynamic load balancing
    - (between OS, run-time system, and app)