Threaded Construction and Fill of Tpetra Sparse Linear System using Kokkos

Mark Hoemmen and Carter Edwards

Trilinos User Group
October 28, 2014
SAND2014-19125 C
Tpetra: Parallel sparse linear algebra

- **Advantages of Tpetra**
  - User can choose data type: float, complex, AD, UQ, ...
  - Can solve problems with over 2 billion unknowns
  - MPI+X parallel (X: any shared-memory parallel programming model)
    - Performance-portable to different node hardware

- **What IS thread-parallel now (& was since ~2008):**
  - Sparse matrix-vector multiply (& any preconditioner that uses it)
  - (Multi)Vector dot, norm, and update (axpy)

- **What’s NOT thread-parallel now:**
  - Some kernels: e.g., sparse {triangular solve, matrix-matrix multiply}
  - **Fill**: Creating or modifying the entries of a graph, matrix, or vector
  - We’re fixing this using Kokkos programming model & data structures
Tpetra fill not thread-scalable

- Dynamic memory allocation ("dynamic profile")
  - Impossible in some parallel models; slow on others
  - Allocation implies synchronization (must agree on pointer)
  - Better: Count, allocate (== thread collective), fill, compute

- Throw C++ exceptions 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
  - Teuchos::RCP: like std::shared_ptr but not thread safe
  - Not hard to make thread safe, but updating the ref count serializes!
  - Better: Use Kokkos’ thread-scalable reference counting
How will Tpetra make fill MPI+X?

- Tpetra will use Kokkos data structures & kernels
  - Kokkos objects are natively thread-safe & scalable
  - You’ll be able to extract & work with the Kokkos objects
- Tpetra objects will behave like Kokkos::DualView
  - Simplify port to use GPUs or multiple memory spaces
  - No overhead (& can ignore) if you only have 1 memory space
- Tpetra objects will have view semantics
  - Copy constructor and operator= will do shallow copy
  - Deep copy is explicit, just like Kokkos
    - Tpetra::deep_copy (nonmember function)
    - 2-argument copy constructor with Teuchos::Copy
  - Will make thread-scalable operations w/ Tpetra objects easier
Initially, use Kokkos directly for fill

- Initially, only Kokkos interface will be thread-safe & scalable
  - Simplifies backwards compatibility & gradual porting
  - Ensures low overhead

- Create a Tpetra object by passing in a Kokkos object
  - Create & fill local Kokkos data structure in thread-parallel way
  - Pass it into Tpetra object’s constructor
  - Tpetra object will wrap & own the Kokkos object

- Modify existing Tpetra object through Kokkos interface
  - Ask Tpetra for its Kokkos local data structure, & modify that

- Tpetra’s interface for modifying data will evolve
  - Only serial (no threads) at first; this will change gradually
  - Will always work on host (not CUDA device) version of data
Tpetra DualView semantics

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

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

- **Done**
  - Crs{Graph, Matrix} & MultiVector use Kokkos objects & kernels
  - Their constructors take Kokkos objects; see TrilinosCouplings FENL
  - MultiVector implements DualView semantics
  - Map implements view semantics; new Map works on CPUs

- **Interface changes left:** MUST finish by Trilinos 12.0 in April
  - Make Crs{Graph, Matrix} & Map implement DualView semantics
  - Firm up Kokkos objects’ interfaces & typedefs in Tpetra

- **Critical thread-parallel kernels to implement**
  - Not needed for Trilinos 12.0, but we want them soon!
  - Sparse triangular solve (prototypes exist) & relaxations
  - Multigrid setup: Sparse matrix-matrix multiply, aggregation
How did we do it?

- 2 Tpetra versions now coexist: “Classic” & “Kokkos Refactor”
  - Both build & pass all Tpetra & downstream solver tests
  - We maintained backwards compatibility if possible (mostly was)
  - You can test both versions in the same code!

- Partial specialization on the Node template parameter
  - Classic Tpetra uses Nodes in KokkosClassic subpackage
  - Kokkos Refactor Tpetra uses Nodes in KokkosCompat subpackage
  - Look in packages/tpetra/src/kokkos_refactor for implementations

- How do I enable Kokkos Refactor RIGHT NOW?
  - Look in packages/tpetra/ReleaseNotes.txt
  - Enable KokkosCore & 5 other subpackages explicitly
  - Set CMake option (Tpetra_ENABLE_Kokkos_Refactor:BOOL=ON)
  - Set default Node type via CMake, or specify it explicitly
Schedule for releasing new Tpetra

- Will deprecate Classic at 11.12 release in January
  - 11.12: Kokkos will build in Trilinos by default (Primary Tested)
  - Using old Node types will emit deprecate warnings
- Plan to remove Classic at 12.0 release in April
  - Don’t worry, MPI only will remain default (Kokkos::Serial device)
  - Best practice: **Do NOT specify Node template parameter explicitly**
    - Rely on default Node type unless you really care
    - Prefer to change default Node type via CMake option
    - Find default from typedefs: e.g., Tpetra::Map<>::node_type
- Early adopters: Get in touch with us!
Coarse- & fine-grained operations

- Prefer *fine-grained* interface for fill
- Coarse-grained: Serial interface, lots of parallel work inside
  - Sparse matrix-vector multiply, dot products, etc.
  - Focus on *linear solve* performance
  - Interface doesn’t matter because all the work is inside
- Fine-grained: Call in parallel, not much work inside
  - Convert an index from global to local
  - Change an entry of a sparse matrix or dense vector
  - Add an entry to a sparse graph
  - Focus on *matrix assembly & preconditioner setup*
  - Interface matters for thread safety & performance
Thread scalable create & fill of Kokkos data structures for sparse matrix: MiniFENL example
MiniFENL: Mini (proxy) Application

- Finite element method to solve of nonlinear problem via Newton iteration
  - Simple scalar nonlinear equation: \(-k \Delta T + T^2 = 0\)
  - 3D domain: simple XYZ box
  - Restrict geometry and boundary conditions to obtain analytic solution to verify correctness
  - Linear hexahedral finite elements: 2x2x2 numerical integration
    - Non-affine mapping of vertices for non-uniform element geometries
  - Compute residual and Jacobian (sparse matrix)
  - Solve linear system via simple conjugate gradient iterative solver

- Focus: **Construction** and **Fill** of sparse linear system
  - Thread safe, thread scalable, and performant algorithms
  - Evaluate thread-parallel capabilities and programming models
Thread-Scalable Fill of Sparse Linear System

- **MiniFENL**: Newton iteration of FEM: \( x_{n+1} = x_n - J^{-1}(x_n)r(x_n) \)

- Thread-scalable pattern: Scatter-Atomic-Add or Gather-Sum?

  - **Scatter-Atomic-Add**
    - Simpler
    - Less memory
    - Slower HW atomic

  - **Gather-Sum**
    - Bit-wise reproducibility

- Performance win?
  - Scatter-atomic-add
  - ~equal Xeon PHI
  - 40% faster Kepler GPU

- Pattern chosen
  - Feedback to HW vendors: performant atomics

![Diagram showing Scatter-Atomic-Add and Gather-Sum processes with performance metrics](image)
Using Kokkos atomic-add interface

Kokkos::View<double*> residual; // global
Kokkos::CrsMatrix<...> jacobian; // global

// parallel compute of element[ielem] residual and jacobian
double elem_res[NN];
double elem_jac[NN][NN];

for ( int i = 0 ; i < NN ; ++i ) {
    const int row = elem_node_index( ielem , i );
    if ( row < residual.dimension_0() ) { // locally owned row
        Kokkos::atomic_fetch_add( & residual(row), elem_res[i] );
        for ( int j = 0 ; j < NN ; ++j ) {
            const int entry = elem_graph_map( ielem , i , j );
            Kokkos::atomic_fetch_add( & jacobian.values(entry)
                                      , elem_jac[i][j] );
        }
    }
}
Performance Overhead of Atomic Add

- Performance analysis: replace atomic-add with “y += x;”
  - Numerical errors due to thread unsafe race condition
  - Approximate performance of “perfect” atomics or coloring algorithm
- Kepler: Large overhead for double precision “CAS loop” atomic
- Phi: Small overhead versus element computation
Thread Scalable CRS Graph Construction

- Thread scalable construction (reconstruction) pattern
  a. **Count** in parallel
  b. **Allocate** (resize)
  c. **Fill** in parallel
  d. **Compute** in parallel

- Construction of CRS Graph from Element-Node connectivity
  a. Generate elements’ unique node-node pairs (graph edges)
     - Count non-zeros per row (graph vertices’ degree)
  b. Prefix-sum non-zeros per row for CRS row offset array
  c. **Allocate** CRS column index array
  d. Fill CRS column index array
  e. Sort each row of CRS column index array

- Following pseudo-code is simplified to illustrate algorithm
CRS Graph Edges (non-zero entries)

- Generate elements’ unique node-node pairs

```cpp
Kokkos::UnorderedMap< pair<int,int> > nodenode("nn",capacity);

Kokkos::parallel_for( num_elem , [=]( int ielem ) {
    for ( int i = 0 ; i < NN ; ++i ) {
        const int row = elem_node_id( ielem , i );
        for ( j = i ; j < NN ; ++j ) {
            const int col = elem_node_id( ielem , j );
            const pair<int,int> key( min(row,col), max(row,col) );
            auto result = nodenode.insert( key );
            if ( result.success() ) { // First time inserted
                if ( row < row_count.dimension_0() )
                    Kokkos::atomic_fetch_add( & row_count(row) , 1 );
                if ( col != row && col < row_count.dimension_0() )
                    Kokkos::atomic_fetch_add( & row_count(col) , 1 );
            }
        }
    }
});
```
CRS Graph Row Offset Array

- Parallel prefix-sum row counts and allocate column index array

```cpp
cr::parallel_scan( num_rows ,
    [=]( int irow , int & update , bool final ) {
        // parallel scan is a multi-pass parallel pattern
        // In the 'final' pass 'update' has the prefix value
        if ( final ) graph.row_map( irow ) = update ;
        update += row_count( irow );
        if ( final && num_rows == irow + 1 )
            graph.row_map(irow+1) = update ; // total non-zeros
    });

    graph.columns =
        View<int*>("columns", graph.row_map(num_rows) );
```

---

17
**CRS Graph: Fill Column Index Array**

- Fill column index array with rows in non-deterministic order

Kokkos::fill( row_count , 0 );
Kokkos::parallel_for( node.node.capacity() ,
    [=]( int ientry ) {
    if ( node.node.valid_at(ientry) ) {
        const pair<int,int> key = node.node.key_at(ientry);
        const int i = key.first ;
        const int j = key.second ;
        if ( i < num_rows )
            graph.columns( graph.row_map(i) +
                atomic_fetch_add(&row_count(i),1) );
        if ( j != i && j < num_rows )
            graph.columns( graph.row_map(j) +
                atomic_fetch_add(&row_count(j),1) );
    }
    });

- Sort each row of column index array
See

- packages/kokkos/example/fenl
  - Create and fill kokkos/linalg data structures

- packages/trilinoscouplings/examples/fenl
  - Create and fill kokkos/linalg data structures
  - Use these objects to create Tpetra objects
  - Variant for embedded UQ “scalar” type