Portable Manycore Sparse Linear System Assembly Algorithms and Performance Tradeoffs

H. Carter Edwards, Daniel Sunderland, and Christian Trott

SIAM Parallel Processing
February 19, 2014
SAND2014-0748C
Kokkos: C++ Library / Programming Model for Manycore Performance Portability

- **Portable to Advanced Manycore Architectures**
  - Multicore CPU, NVidia GPU, Intel Xeon Phi (potential: AMD Fusion)
  - Maximize amount of user (application/library) code that can be compiled without modification and run on these architectures
  - Minimize amount of architecture-specific knowledge that a user is required to have
  - Allow architecture-specific tuning to easily co-exist
  - Only require C++1998 standard compliant

- **Performant**
  - Portable user code performs as well as architecture-specific code
    - Thread scalable – not just thread safety (no locking!)

- **Usable**
  - Small, straight-forward application programmer interface (API)
    - Constraint: don’t compromise portability and performance
Kokkos: Collection of Libraries

- Core – lowest level portability layer
  - Portable data-parallel dispatch: parallel_for, parallel_reduce, parallel_scan
  - Multidimensional arrays with device-polymorphic layout for transparent and device-optimal memory access patterns

- Containers – built on core arrays
  - UnorderedMap – fast find and thread scalable insertion
  - Vector – subset of std::vector functionality to ease porting
  - Compress Row Storage (CRS) graph

- Linear Algebra
  - Sparse matrices and linear algebra operations
  - Wrappers to vendors’ libraries
  - Portability layer for Trilinos manycore solvers

- Examples – where the code for this presentation resides
  - MiniFENL: finite element solution of non-linear system of equations
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
MiniFENL: Parallel Computational Steps

- **Construct finite element mesh**
  - Simple unstructured finite element mesh data structure
  - Hexahedral elements, element-node connectivity array

- **Construct maps sparse linear system**
  - Sparse linear system graph: node-node map
  - Element-graph map for scatter-atomic-add assembly algorithm
    - Graph-element map for gather-sum assembly algorithm

- **Compute nonlinear residual and Jacobian**
  - Iterate elements to compute per-element residual and Jacobian
    - Scatter-atomic-add values into linear system
      - Save values in gather-sum scratch array
      - Iterate rows, gather data from scratch array, sum into linear system

- **Solve linear system for Newton iteration**
Scatter-Atomic-Add vs. Gather-Sum

Map: Mesh $\rightarrow$ Sparse Graph

Scatter-Atomic-Add Pattern

Element Computations + Scatter-Add

atomic_add

Sparse Linear System Coefficients

Finite Element Data

Element Computations

Scratch Arrays

Gather-Sum

Gather-Sum Pattern
Scatter-Atomic-Add Overview

- Compute element nonlinear residual and Jacobian
  - Parallel-for iteration of elements
  - Computational in element-local arrays, with element-local numbering
  - \( \text{ElemRes}(i) = \text{element residual for local node } \#i \)
  - \( \text{ElemJac}(i,j) = \text{element Jacobian for local nodes } \#i, \#j \)

- Add values into sparse linear system
  - \( \text{Res}(I) = \text{Residual for row } I \)
  - \( \text{Jac}(K) = \text{Jacobian value for row } I \text{ column } J \text{ in the sparse linear system} \)
  - \( \text{atomic_add}( \text{Res( node_map(e,i) )} , \text{ElemRes}(i) ) \)
  - \( \text{atomic_add}( \text{Jac( elem_graph_map(e,i,j) )} , \text{ElemJac}(i,j) ) \)

- Precompute \text{elem_graph_map}
  - Composition of element-node map and node-node map
  - Compute once and re-use in the nonlinear Newton iteration loop
  - Valid as long as the mesh and graph don’t change
Gather-Sum Overview

- Compute element nonlinear residual and Jacobian
  - Parallel-for iteration of elements
  - Computational in element-local arrays, with element-local numbering
  - $\text{ElemRes}(i) = \text{element residual for local node } #i$
  - $\text{ElemJac}(i,j) = \text{element Jacobian for local nodes } #i, #j$

- Save values in scratch arrays (large scratch space)
  - $\text{ScrRes}(e,i) = \text{ElemRes}(i) ; \text{ScrJac}(e,i,j) = \text{ElemJac}(i,j)$

- Gather-sum from scratch array into sparse linear system
  - Parallel-for iteration of rows, each thread has exclusive access to its row ‘$I$’
  - Iterate elements ‘$e$’ with node ‘$i$’ mapping to row ‘$I$’
    - $(e,i) \in \text{row}_\text{elem}_\text{map}(I) ; \text{uses a CRS graph data structure}$
    - $\text{Res}(I) += \text{ScrRes}(e,i)$
    - $\text{Jac}(\text{elem}_\text{graph}_\text{map}(e,i,j)) += \text{ScrJac}(e,i,j)$ element-local nodes $(i,j)$

- Precompute $\text{elem}_\text{graph}_\text{map}$ and $\text{row}_\text{elem}_\text{map}$
Scatter-Atomic-Add vs. Gather-Sum

- Both are thread-safe and thread-scalable

Scatter-Atomic-Add
- Simple implementation
- Fewer global memory reads and writes
  - Atomic operations much slower than corresponding regular operation
  - Non-deterministic order of additions – floating point round off variability
  - Double precision atomic add is a looped compare-and-swap (CAS)

Gather-Sum
- Deterministic order of additions – no round off variability
  - Extra scratch arrays for element residuals and Jacobians
  - Additional parallel-for

Performance comparison – execution time
- Neglecting the time to pre-compute mapping(s), assuming re-use
- Cost of atomic-add vs. additional parallel-for for the gather-sum
Performance Comparison

- Single “Device” Performance Tests
  - NVidia Kepler K40 (Atlas), 12Gbytes
  - Intel Xeon Phi (Knights Corner) COES2, 61 cores, 1.2 GHz, 16Gbytes
    - Limit use to 60 cores, 4 hyperthreads/core

- MiniFENL – portable source code via Kokkos

- Kokkos chooses multidimensional array layouts to match device
  - NVidia : coalesced memory access
  - Intel : caching and vectorization

- Scale problem size (number of nodes)
  - Small problem size – parallel dispatch overhead dominate
  - Large problem size – computations dominate

- Measure total time-to-fill normalized by problem size
  - Element Computation + ( Scatter-Atomic | Gather-Sum )
  - Double precision data and computations
Performance Comparison: Element+Fill

- **Phi**: ScatterAtomicAdd ~equal to GatherSum
  - ~2.1x speed up from 1 to 4 threads/core – hyperthreading
- **Kepler**: ScatterAtomicAdd ~40% faster than GatherSum
  - Performant double precision atomic-add via compare-and-swap algorithm
  - Fewer global memory writes and reads
Performance Overhead of Atomic Add

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

1. Fill unordered map with elements’ (row-node, column-node)
   - Parallel-for of elements, iterate node-node pairs
   - Successful insert to node-node unordered map denotes a unique entry
   - Column count = count unique entries for each row-node

2. Construct (row-node, column-node) sparse graph
   - Parallel-scan of row-node column counts
     - This is now the CRS row-offset array
   - Allocate CRS column-index array
   - Parallel-for on node-node unordered map to fill CRS column-index array
   - Parallel-for on CRS graph rows to sort each row’s column-indices

3. Construct elem_graph_map
   - Parallel-for of elements
   - For each element (row-node, column-node) search CRS graph row for column-index entry
**Performance: Graph and Map Construction**

- Graph construction ~5x longer than Element+Fill
  - Multiple parallel kernels performing random-access queries and updates
  - Recall finite element computation is
    - Linearized hexahedron finite element for: 
      \[ -k \Delta T + T^2 = 0 \]
    - 3D spatial Jacobian with 2x2x2 point numerical integration
Performance: Graph Construction on Phi-240

- Performance for each phase of construction
  - “Hot spot” is fill of node-node unordered map (hash map)
    - Dominated by memory access and integer atomic operations
    - Extensive analysis and optimizing has been done here ...
    - Performance very sensitive to hash map capacity, must be < 75% full
Performance: Graph Construction on Kepler

- Performance for each phase of construction
  - No single “hot spot”
  - Opportunity to improve parallelism for sort of CRS row columns: change from serial to parallel sorting within a row
Conclusions: Sparse Linear System Assembly

- Scatter-atomic-add is the winning pattern
  - Less memory consumed, faster with performant atomics
  - If you can tolerate floating point round off variability / nondeterminism

- CRS graph construction can be thread scalable
  - Pattern
    - Parallel count array lengths
    - Allocate arrays
    - Parallel fill arrays
    - Parallel post-process arrays (e.g., sort CRS rows’ column indices)
  - Essential tools
    - Parallel-for, parallel-scan, and atomics
    - Thread scalable unordered map

- Graph construction time > sparse linear system fill time
  - Separate graph construction from sparse linear system fill
  - Reuse graph whenever possible
A little more about Kokkos

- **Core abstractions**
  - Dispatch parallel kernels to a manycore device
    - Parallel for, parallel reduce, parallel scan
  - Device-polymorphic layout of multidimensional arrays in device memory
    - Choose layout for optimal memory access patterns
    - Layout changes are transparent to user code

- **A Library using Standard C++, not a Language extension**
  - In *spirit* of Intel’s TBB, NVIDIA’s Thrust & CUSP, MS C++AMP, ...
  - *Not* a language extension: OpenMP, OpenACC, OpenCL, CUDA

- **Via C++ template meta-programming**
  - Compile-time polymorphism for devices and array layouts
  - C++1998 standard; would be nice to *require* C++2011 ...
A little more about Kokkos

- Recent capabilities
  - Parallel scan using arbitrary user-supplied kernels
  - Unordered map container
    - Thread scalable insert and erase
    - Use NVidia texture fetch for random access queries
  - League of thread teams
    - Team shared scratch memory and synchronization functions

- Here at SIAM PP14
  - MS7 – Embedded UQ on manycore architectures
  - MS33 – Overview and use in other miniapplications
  - MS70 – Research on unified task-data manycore parallelism
  - MS73 – DSL layered on Kokkos