Kokkos – Portability, Performance, Productivity

Christian Trott, Carter Edwards, Nathan Ellingwood, Si Hammond

crtrott@sandia.gov

Center for Computing Research
Sandia National Laboratories, NM
Kokkos: Performance, Portability and Productivity

Multi-Core  Many-Core  APU  CPU + GPU
Kokkos: Performance, Portability and Productivity

- A programming model implemented as a C++ library
- Abstractions for Parallel Execution and Data Management
  - Execution Pattern: What kind of operation (for-each, reduction, scan, task)
  - Execution Policy: How to execute (Range Policy, Team Policy, DAG)
  - Execution Space: Where to execute (GPU, Host Threads, PIM)
  - Memory Layout: How to map indices to storage (Column/Row Major)
  - Memory Traits: How to access the data (Random, Stream, Atomic)
  - Memory Space: Where does the data live (High Bandwidth, DDR, NV)
- Supports multiple backends: OpenMP, Pthreads, Cuda, Qthreads, Kalmar (experimental)
- Sandia application teams committed to Kokkos as its path for transitioning legacy codes, and as part of its new codes
  - Trilinos, LAMMPS, Albany, Sierra Mechanics, ...
ASC L2 2015 Codesign Milestone

- Evaluate the performance and productivity tradeoff when using the Kokkos C++ programming model
- Chose LLNL mini-App Lulesh to demonstrate broad applicability

**Source Code Lines Added/Removed Compared to Serial**

<table>
<thead>
<tr>
<th>System</th>
<th>Lines Added</th>
<th>Lines Removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kokkos Minimal CPU</td>
<td>500</td>
<td>100</td>
</tr>
<tr>
<td>Kokkos Minimal GPU</td>
<td>700</td>
<td>200</td>
</tr>
<tr>
<td>Kokkos Optimized v1</td>
<td>800</td>
<td>300</td>
</tr>
<tr>
<td>Kokkos Optimized v2</td>
<td>900</td>
<td>400</td>
</tr>
<tr>
<td>Kokkos Optimized v3</td>
<td>1000</td>
<td>500</td>
</tr>
<tr>
<td>OpenMP Optimized</td>
<td>1100</td>
<td>600</td>
</tr>
<tr>
<td>OpenMP Original</td>
<td>1200</td>
<td>700</td>
</tr>
</tbody>
</table>

**LULESH Benchmark Figure of Merit on Multi-Core, Many-Core and GPU Systems (Problem Size 90)**

<table>
<thead>
<tr>
<th>System</th>
<th>Zones per Second per Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haswell Single Socket MPI 1 x 16 Threads (Problem 90)</td>
<td>5000</td>
</tr>
<tr>
<td>Haswell Single Socket MPI 1 x 32 (Problem 90)</td>
<td>4500</td>
</tr>
<tr>
<td>Knights Corner MPI 1 x 224 (Problem 90)</td>
<td>4000</td>
</tr>
<tr>
<td>Sandy Bridge Single Socket MPI 1 x 8 (Problem 90)</td>
<td>3500</td>
</tr>
<tr>
<td>NVIDIA K40 (Problem 90)</td>
<td>3000</td>
</tr>
<tr>
<td>APM XGene1 MPI 1 x 8 Threads (Problem 90)</td>
<td>2500</td>
</tr>
<tr>
<td>POWER8-XL Dual Socket Node MPI 8 x NUMA Domain MPI 1 x 40 Threads (Problem 90)</td>
<td>2000</td>
</tr>
<tr>
<td>POWER8-XL Single Compute Element NUMA Domain MPI 1 x NUMA Domain MPI 1 x 40 Threads (Problem 90)</td>
<td>1500</td>
</tr>
</tbody>
</table>
Going Production

- Kokkos released on github in March 2015
  - Develop / Master branch system => merge requires application passing
  - Testing Nightly: 11 Compilers, total of 90 backend configurations, warnings as errors
  - Extensive Tutorials and Documentation > 300 slides/pages

- Trilinos NGP stack uses Kokkos as only backend
  - Tpetra, Belos, MueLu etc.
  - Working on threading all kernels, and support GPUs

- Sandia Sierra Mechanics going to transition to Kokkos
  - Decided to go with Kokkos instead of OpenMP (only other realistic choice)
  - FY 2016: prototyping threaded algorithms, explore code patterns
  - Data management postponed to FY 2017 and follow on

- Sandia ATDM has Kokkos as big component
  - All ATDM Apps are using Kokkos
  - Add System level Tasking with Dharma later
Improved View Capabilities

- View now allow allocatable types
  - Enables View-of-View concept (View<View<double*> >)
  - Simplifies implementation for enabling UQ types
- Utilize const expression C++11 functionality
  - Better optimization; often smaller binary size
- Improvement in underlying assembly/instruction generation
  - Makes debugging and profiling mapping much cleaner
KokkosP Profiling Interface

- Dynamic Runtime Linkable profiling tools
  - Not LD_PRELOAD based (hurray!)
  - Profiling hooks are always enabled (i.e. also in release builds)
    - Compile once, run anytime, profile anytime, no confusion or recompile!
  - Tool Chaining allowed (many results from one run)
  - Very low overhead if not enabled

- Simple C Interface for Tool Connectors
  - Users/Vendors can write their own profiling tools
  - VTune, NSight and LLNL-Caliper

- Parallel Dispatch can be named to improve context mapping

- Initial tools: simple kernel timing, memory profiling, thread affinity checker, vectorization connector (APEX-ECLDRD)
Enhancing Productivity: Using C++ Lambdas

- C++11 Feature which simplify using abstraction layers

**Pragma Based OpenMP:**
```
#pragma omp parallel for
for(int i=0; i<N; i++) {
    a[i] += b[i];
}
```

**Functor Based Kokkos:**
```
struct vector_add {
    View<double*> a;
    View<double*> b;
    vector_add(View<double*> a_, View<double*> b_): a(a_), b(b_){}
    KOKKOS_INLINE_FUNCTION
    void operator() (const int&i) const {
        a(i) += b(i);
    }
};

parallel_for( N, vector_add(a,b));
```

**LAMBDA Based Kokkos:**
```
parallel_for( N, KOKKOS_LAMBDA (const int& i) {
    a[i] += b[i];
});
```
Hierarchical Parallelism

- Expose more parallelism
- Provides API for efficient scratch space usage and collaboration

TeamPolicy<> policy =
TeamPolicy<>(N, AUTO()).set_scratch_size(1,team_size,thread_size);

parallel_for( policy, KOKKOS_LAMBDA (TeamPolicy<>::~member_type& team) {
  View<double*,ScratchSpace> team_shared(team.team_scratch(1),M);
  View<double*,ScratchSpace> thr_private(team.thread_scratch(1),P);

  // Fill shared and thread private data
  team.team_barrier();

  parallel_for(TeamThreadRange(team,0,M), [&] (int& j) {
    // Do Something
  });

  double sum;
  parallel_for(TeamThreadRange(team,0,M), [&] (int& j, double& lsum) {
    for(int k=0; k<P; k++) lsum += thr_private(k);
  });

  single(PerTeam(team), [&] () { global(team.league_rank()) = sum; } );
});
Hierarchical Parallelism

- Expose more parallelism
- Provides API for efficient scratch space usage and collaboration

```cpp
TeamPolicy<> policy = 
TeamPolicy<>((N, AUTO()).set_scratch_size(1,team_size,thread_size);

parallel_for( policy, KOKKOS_LAMBDA (TeamPolicy<>::member_type& team) {
    View<double*,ScratchSpace> team_shared(team.team_scratch(1),M);
    View<double*,ScratchSpace> thr_private(team.thread_scratch(1),P);

    // Fill shared and thread private data
    team.team_barrier();

    parallel_for(TeamThreadRange(team,0,M), [&] (int& j){
        //Do Something
    });

    double sum;
    parallel_for(TeamThreadRange(team,0,M), [&] (int& j, double& lsum){
        for(int k=0; k<P; k++) lsum+= thr_private(k);
    });

    single(PerTeam(team), [&] () { global(team.league_rank()) = sum; }
});```
Hierarchical Parallelism

- Expose more parallelism
- Provides API for efficient scratch space usage and collaboration

```cpp
TeamPolicy<> policy = TeamPolicy<>(N, AUTO()).set_scratch_size(1,team_size,thread_size);

parallel_for( policy, KOKKOS_LAMBDA (TeamPolicy<>::member_type& team) {
    View<double*,ScratchSpace> team_shared(team.team_scratch(1),M);
    View<double*,ScratchSpace> thr_private(team.thread_scratch(1),P);

    // Fill shared and thread private data
    team.team_barrier();

    parallel_for(TeamThreadRange(team,0,M), [&] (int& j){
        //Do Something
    });

    double sum;
    parallel_for(TeamThreadRange(team,0,M), [&] (int& j, double& lsum){
        for(int k=0; k<P; k++) lsum+= thr_private(k);
    });

    single(PerTeam(team), [&] () { global(team.league_rank()) = sum; } );
```
Hierarchical Parallelism

- Expose more parallelism
- Provides API for efficient scratch space usage and collaboration

```cpp
TeamPolicy<> policy =
TeamPolicy<>::(N, AUTO()].set_scratch_size(1,team_size,thread_size);

// TeamPolicy::policy, KOKKOS_LAMBDA (TeamPolicy<>:::member_type& team) {
View<double*,ScratchSpace> team_shared(team.team_scratch(1),M);
View<double*,ScratchSpace> thr_private(team.thread_scratch(1),P);

// Fill shared and thread private data
team.team_barrier();

parallel_for(TeamThreadRange(team,0,M), [&] (int& j){
    // Do Something
});

double sum;
parallel_for(TeamThreadRange(team,0,M), [&] (int& j, double& lsum){
    for(int k=0; k<P; k++) lsum+= thr_private(k);
});

single(PerTeam(team), [&] () { global(team.league_rank()) = sum; })
```

Hierarchical Parallelism

- Expose more parallelism
- Provides API for efficient scratch space usage and collaboration

TeamPolicy<> policy =
TeamPolicy<>(N, AUTO()).set_scratch_size(1,team_size,thread_size);

parallel_for( policy, KOKKOS_LAMBDA (TeamPolicy<>::member_type& team) {
  View<double*,ScratchSpace> team_shared(team.team_scratch(1),M);
  View<double*,ScratchSpace> thr_private(team.thread_scratch(1),P);

  // Fill shared and thread private data
  team.team_barrier();

  parallel_for(TeamThreadRange(team,0,M), [&] (int& j){
    //Do Something
  });

  double sum;
  parallel_for(TeamThreadRange(team,0,M), [&] (int& j, double& lsum){
    for(int k=0; k<P; k++) lsum+= thr_private(k);
  });

  single(PerTeam(team), [&] () { global(team.league_rank()) = sum; });
});
Hierarchical Parallelism

- Expose more parallelism
- Provides API for efficient scratch space usage and collaboration

```cpp
TeamPolicy<> policy = 
TeamPolicy<>((N, AUTO()).set_scratch_size(1, team_size, thread_size);

parallel_for( policy, KOKKOS_LAMBDA (TeamPolicy<>::member_type& team) {
  View<double*, ScratchSpace> team_shared(team.team_scratch(1), M);
  View<double*, ScratchSpace> thr_private(team.thread_scratch(1), P);

  // Fill shared and thread private data
  team.team_barrier();

  parallel_for(TeamThreadRange(team, 0, M), [&] (int& j){
    // Do Something
  });

  double sum;
  parallel_for(TeamThreadRange(team, 0, M), [&] (int& j, double& lsum){
    for(int k=0; k<P; k++) lsum+= thr_private(k);
  });

  single(PerTeam(team), [&] () { global(team.league_rank()) = sum; })
});
```
Hierarchical Parallelism

- Expose more parallelism
- Provides API for efficient scratch space usage and collaboration

TeamPolicy<> policy =
TeamPolicy<>((N, AUTO()).set_scratch_size(1,team_size,thread_size);

parallel_for( policy, KOKKOS_LAMBDA (TeamPolicy<>::member_type& team) {
  View<double*,ScratchSpace> team_shared(team.team_scratch(1),M);
  View<double*,ScratchSpace> thr_private(team.thread_scratch(1),P);

  // Fill shared and thread private data
  team.team_barrier();

  parallel_for(TeamThreadRange(team,0,M), [&] (int& j){
    // Do Something
  });

  double sum;
  parallel_for(TeamThreadRange(team,0,M), [&] (int& j, double& lsum){
    for(int k=0; k<P; k++) lsum+= thr_private(k);
  });

  single(PerTeam(team), [&] () { global(team.league_rank()) = sum; } ));
Hierarchical Parallelism

- Expose more parallelism
- Provides API for efficient scratch space usage and collaboration

TeamPolicy<> policy =
TeamPolicy<>(N, AUTO()).set_scratch_size(1,team_size,thread_size);

parallel_for( policy, KOKKOS_LAMBDA (TeamPolicy<>::member_type& team) {
    View<double*,ScratchSpace> team_shared(team.team_scratch(1),M);
    View<double*,ScratchSpace> thr_private(team.thread_scratch(1),P);

    // Fill shared and thread private data
    team.team_barrier();

    parallel_for(TeamThreadRange(team,0,M), [&] (int& j){
        //Do Something
    });

    double sum;
    parallel_for(TeamThreadRange(team,0,M), [&] (int& j, double& lsum){
        for(int k=0; k<P; k++) lsum+= thr_private(k);
    });

    single(PerTeam(team), [&] () { global(team.league_rank()) = sum; })
});
Hierarchical Parallelism

- Expose more parallelism
- Provides API for efficient scratch space usage and collaboration

TeamPolicy<> policy =
TeamPolicy<>(N, AUTO()).set_scratch_size(1, team_size, thread_size);

parallel_for( policy, KOKKOS_LAMBDA (TeamPolicy<>::member_type& team) {
    View<double*, ScratchSpace> team_shared(team.team_scratch(1), M);
    View<double*, ScratchSpace> thr_private(team.thread_scratch(1), P);

    // Fill shared and thread private data
    team.team_barrier();

    parallel_for(TeamThreadRange(team, 0, M), [&] (int& j){
        // Do Something
    });

    double sum;
    parallel_for(TeamThreadRange(team, 0, M), [&] (int& j, double& lsum){
        for(int k=0; k<P; k++) lsum+= thr_private(k);
    });

    single(PerTeam(team), [&] () { global(team.league_rank()) = sum; })
});
Hierarchical Parallelism

- Expose more parallelism
- Provides API for efficient scratch space usage and collaboration

```c++
TeamPolicy<> policy =
TeamPolicy<>(N, AUTO()).set_scratch_size(1,team_size,thread_size);

parallel_for( policy, KOKKOS_LAMBDA (TeamPolicy<>::member_type& team) {
  View<double*,ScratchSpace> team_shared(team.team_scratch(1),M);
  View<double*,ScratchSpace> thr_private(team.thread_scratch(1),P);

  // Fill shared and thread private data
  team.team_barrier();

  parallel_for(TeamThreadRange(team,0,M), [&] (int& j){
    //Do Something
  });

  double sum;
  parallel_for(TeamThreadRange(team,0,M), [&] (int& j, double& lsum){
    for(int k=0; k<P; k++) lsum+= thr_private(k);
  });

  single(PerTeam(team), [&] () { global(team.league_rank()) = sum; } )
});
```
Hierarchical Parallelism

- Expose more parallelism
- Provides API for efficient scratch space usage and collaboration

TeamPolicy<> policy =
TeamPolicy<>((N, AUTO()).set_scratch_size(1, team_size, thread_size);

parallel_for( policy, KOKKOS_LAMBDA(TeamPolicy<>::member_type& team) {
    View<double*, ScratchSpace> team_shared(team.team_scratch(1), M);
    View<double*, ScratchSpace> thr_private(team.thread_scratch(1), P);

    // Fill shared and thread private data
    team.team_barrier();

    parallel_for(TeamThreadRange(team, 0, M), [&] (int& j) {
        // Do Something
    });

    double sum;
    parallel_for(TeamThreadRange(team, 0, M), [&] (int& j, double& lsum) {
        for(int k=0; k<P; k++) lsum += thr_private(k);
    });

    single(PerTeam(team), [&] () { global(team.league_rank()) = sum; })
});
Hierarchical Parallelism

- Expose more parallelism
- Provides API for efficient scratch space usage and collaboration

```c
TeamPolicy<> policy = TeamPolicy<>(N, AUTO()).set_scratch_size(1,team_size,thread_size);

parallel_for( policy, KOKKOS_LAMBDA (TeamPolicy<>::member_type& team) {
    View<double*,ScratchSpace> team_shared(team.team_scratch(1),M);
    View<double*,ScratchSpace> thr_private(team.thread_scratch(1),P);

    // Fill shared and thread private data
    team.team_barrier();

    parallel_for(TeamThreadRange(team,0,M), [&] (int& j){
        //Do Something
    });

    double sum;
    parallel_for(TeamThreadRange(team,0,M), [&] (int& j, double& lsum){
        for(int k=0; k<P; k++) lsum+= thr_private(k);
    });

    single(PerTeam(team), [&] () { global(team.league_rank()) = sum; } );

});
```
Hierarchical Parallelism

- Expose more parallelism
- Provides API for efficient scratch space usage and collaboration

```cpp
TeamPolicy<> policy =
TeamPolicy<>(N, AUTO()).set_scratch_size(1,team_size,thread_size);

parallel_for( policy, KOKKOS_LAMBDA (TeamPolicy<>::member_type& team) {
    View<double*,ScratchSpace> team_shared(team.team_scratch(1),M);
    View<double*,ScratchSpace> thr_private(team.thread_scratch(1),P);

    // Fill shared and thread private data
    team.team_barrier();

    parallel_for(TeamThreadRange(team,0,M), [&] (int& j){
        //Do Something
    });

    double sum;
    parallel_for(TeamThreadRange(team,0,M), [&] (int& j, double& lsum){
        for(int k=0; k<P; k++) lsum+= thr_private(k);
    });

    single(PerTeam(team), [&] () { global(team.league_rank()) = sum; })
});
```
Dynamic Scheduling

- Addresses simple load balancing issues
- Affinity aware Work Stealing Mechanism (note: OpenMP is work sharing)
- Up to 100x faster scheduling than Intel OpenMP (based on scheduling stress test)
- Modifier on execution policy e.g.: RangePolicy<Schedule<Dynamic> >(0,N)
Directed Acyclic Graph (DAG) of Tasks
Directed Acyclic Graph (DAG) of Tasks

- **A Task**
  - Is a C++ closure (e.g., functor) of data + function
  - Executes on a single thread or thread team
  - May only execute when its dependences are complete (DAG)

- **A Task’s Life Cycle:**

  - Serial task on a single thread
  - Data parallel task on a thread team

- Tasks can be in states of: constructing, waiting, executing, and complete.
Task Execution Policy

- **Manages a Heterogeneous Collection of Tasks**
  - Memory allocation and deallocation in a memory space
  - Execution on a thread or thread team in an execution space
  - Scheduling according to dependence directed acyclic graph (DAG)

- **Challenges**
  - Portability across multicore/manycore architectures: CPU, Xeon Phi, GPU, ...
  - Dynamic – creating tasks within executing task
  - Performance – thread scalable allocation/deallocation within finite memory
  - Performance – execution overhead and thread scalable scheduling

- **Portability and Performance Constraint: Non-blocking Tasks**
  - Eliminate overhead of saving execution state: registers, stack, ...
  - Reduce overhead of context switching
Managing a Non-Blocking Task’s life-cycle

- **Create: allocate and construct**
  - By the main process or another task
  - Allocate from task policy memory pool
  - Construct internal data
  - Assign DAG dependences

- **Spawn: enqueue to scheduler**
  - By the main process or another task

- **Respawn: re-enqueue to scheduler**
  - By the executing task itself
  - Reassign DAG dependences
  - Replaces task spawning and waiting upon “child” task(s)
    - Create and spawn child task(s)
    - Assign new DAG dependence(s) to new child task(s)
    - Re-enqueue to be executed again after child task(s) complete
The Way Forward

- Stabilize Capabilities
  - Support tasking on all platforms
  - Make sure compilers optimize through layers
  - Harden KNL support for High Bandwidth Memory
- Support Production Teams in Adoption
- Develop more Documentation
- Extend profiling tools to help with transition

www.github.com/kokkos/kokkos: Kokkos Core Repository
www.github.com/kokkos/kokkos-tutorials: Kokkos Tutorial Material
www.github.com/kokkos/kokkos-profiling: Kokkos Profiling Tools (soon)