Kokkos Technical Review
Slides and Discussion Notes

Sept. 6-8, 2016
Albuquerque, NM

SAND2016-9870 PE
Unclassified Unlimited Release, UUR
Kokkos Comprehensive Technical Review

- Is a Technical Review Dialogue, is not a tutorial
  - Participants (you) should be active developers who use Kokkos
  - We will review Kokkos’ current and near-term-priority functionality and API
  - We want your feedback: what is working well, what is not working well
  - We want a dialogue regarding how to improve functionality and API

- Gather feedback, dialogue, recommendations, requests

- Goal: Kokkos Version 3.0 in FY17
  - Incorporate high priority recommendations & requests from this review
  - Breaking backward compatibility is “on the table” as necessary
Agenda

- **Sept. 6: Tuesday Afternoon 1-5pm**
  - Configuration management, testing, build
  - Initialization and finalization
  - Execution and memory space instances: multi-GPU

- **Sept. 7: Wednesday Morning 8:30-11:30am**
  - Utility types
  - View and subview, extensibility: tiling, padding

- **Sept. 7: Wednesday Afternoon 1-5pm**
  - Dynamic rank view
  - Dual View
  - Unordered map and Bitset
  - “Flat” data parallel patterns, 1D range policy, and user-supplied closures
  - Hierarchical data parallel patterns, team policy
  - Customized reductions
Agenda

- **Sept. 8: Thursday Morning 8:30-11:30pm**
  - Atomic operations
  - Algorithms: random number generator and sort
  - Multidimensional range policy
  - Memory pool and dynamic size view
  - Task DAG pattern and policy

- **Sept. 8: Thursday Afternoon 1-5pm**
  - Kokkos Kernels
  - Kokkos profiling tools
  - Examples, tutorials, documentation, ...
  - Open discussion to prioritize recommendations and requests
# Participants

## Kokkos Users

<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keita Teranishi</td>
<td>SNL</td>
</tr>
<tr>
<td>Brad Peterson</td>
<td>U-Utah</td>
</tr>
<tr>
<td>John Schmidt</td>
<td>U-Utah</td>
</tr>
<tr>
<td>Irena Demeshko</td>
<td>LANL</td>
</tr>
<tr>
<td>Geoff Womeldorff</td>
<td>LANL</td>
</tr>
<tr>
<td>James Elliot</td>
<td>SNL</td>
</tr>
<tr>
<td>Eric Phipps</td>
<td>SNL</td>
</tr>
<tr>
<td>Erik Boman</td>
<td>SNL</td>
</tr>
<tr>
<td>Stephen Olivier</td>
<td>SNL</td>
</tr>
<tr>
<td>Jimmy Carleton</td>
<td>SNL</td>
</tr>
<tr>
<td>James Overfelt</td>
<td>SNL</td>
</tr>
<tr>
<td>Alex Toth</td>
<td>SNL</td>
</tr>
<tr>
<td>Mehmet Deveci</td>
<td>SNL</td>
</tr>
<tr>
<td>Daniel Holladay</td>
<td>LANL</td>
</tr>
<tr>
<td>David Hollman</td>
<td>SNL</td>
</tr>
<tr>
<td>Alan Williams</td>
<td>SNL</td>
</tr>
<tr>
<td>Kendall Pierson</td>
<td>SNL</td>
</tr>
</tbody>
</table>

## Kokkos Users

<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mike Tupek</td>
<td>SNL</td>
</tr>
<tr>
<td>Roger Pawlowski</td>
<td>SNL</td>
</tr>
<tr>
<td>Matt Bettencourt</td>
<td>SNL</td>
</tr>
<tr>
<td>Tolu Okusanya</td>
<td>SNL</td>
</tr>
<tr>
<td>Chris Wilson</td>
<td>SNL</td>
</tr>
</tbody>
</table>

## Kokkos Developers

<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carter Edwards</td>
<td>SNL</td>
</tr>
<tr>
<td>Dan Sunderland</td>
<td>SNL</td>
</tr>
<tr>
<td>Mark Hoemmen</td>
<td>SNL</td>
</tr>
<tr>
<td>Nathan Ellingwood</td>
<td>SNL</td>
</tr>
<tr>
<td>Greg Mackey</td>
<td>SNL</td>
</tr>
<tr>
<td>Christian Trott</td>
<td>SNL</td>
</tr>
</tbody>
</table>
CONFIGURATION MANAGEMENT
Configuration Management

- [github.com/kokkos/kokkos](https://github.com/kokkos/kokkos)
  - Publicly viewable/cloneable repository, restricted permission to push
  - Issue tracking for questions, bug reports, feature requests

- "develop -> master" Development Process
  - Day to day development pushes to develop branch
  - Expect develop pull requests and pushes to pass Kokkos' internal tests
  - ... more on Kokkos internal testing later
  - All commits to master are develop->master promotion NO fast-forward merge
    - New: config/master_history.txt with tag, date, and upstream sha1
  - Expect master branch to pass internal and integration tests
  - ... more on Kokkos integrating testing later
  - Every merge commit on master branch is highly trustworthy
  - All merge commits on master branch are copied into Trilinos repository
  - Merge commits to master branch are expensive quality assurance efforts
    - Occur ~ every few weeks; our "sprints" do not have hardened schedule
Configuration Management

- **Namespaces are meaningful**
  - *Kokkos::* denotes stable public feature and API
  - *Kokkos::Impl::* denotes our internal “private” implementation API
    - NO promise for stability, we revise/refactor at will!
    - Is a feature request to move something from private to public
  - *Kokkos::Experimental::* denotes a planned to-become-public feature and API
    - Exposed for evaluation, feedback, and internal testing
    - Not stable until promoted to *Kokkos::* namespace
    - *Kokkos::Experimental::Impl::* is corresponding private implementation

- **Denote experimental feature and API in Kokkos:: classes?**
  - Prefix “experimental_” to those member names?
  - Simply comment that is “experimental”?
**Discussion Summary**

**Issue Tracking: Email list or Github issues?**
- Github issues: feature requests, enhancement requests, bug reports, questions
- Add to Github page “If you have a question, post an issue” and how to subscribe to email list
- Hard to find questions and answers
- Programming guide is go-to resource, needs to be updated
- Improve Github main page, could use wiki
- Announcements should be via kokkos-users email list (straw vote YES); e.g., breaking backward compatibility
- Bug reports and feature requests must be via Github issues; not email list

**develop -> master promotion process**
- Need to add explanation of version identifier
- \${MAJOR_VERSION}.${MINOR_VERSION}.${NUMBER_OF_WEEKS_SINCE_LAST_MINOR_VERSION_CHANGE}
- Changes to MAJOR require reassert copyright
- Changes to MINOR should required updating user documentation
- Usually 4-6 weeks between promotion
- Merge commit message should have the tag
- Pull requests to develop branch only
- Master branch is protected

**Experimental development, implementation namespace**
- Extensibility of View layout needs to be in public namespace
- Want a road map of what development is planned, more readable than issue list
- experimental_ prefix for experimental member functions that cannot go into Experimental namespace
- experimental API needs to be stable in at least one cycle of master before lifting to public namespace
TESTING AND COMPILERS
Outline

- Standalone Testing
- Integration Testing
- Develop/Master promotion
- Compiler lifecycle
Standalone testing

- Jenkins based
- Nightly test of develop branch
- 4 machines: Shepard (Haswell), Bowman (KNL), White (Power8) KokkosDev (IvyBridge + K40)
- Each machine tests multiple compilers and backend configurations
- Strong Warnings and Warnings as Errors on Intel, GCC and Clang
  - “-Wall, -Wshadow, -pedantic, -Werror, -Wsign-compare, -Wtype-limits, -Wignored-qualifiers, -Wempty-body, -Wclobbered, -Wuninitialized”
- Total configurations: 115 complete test-suite runs per night
- Script: kokkos/config/test_all_sandia
- Individual test on Pascal, ARM, PGI, Cray
Standalone Testing

- **Shepard (HSW):**
  - Intel 16.1 / 17.0
  - Serial, OpenMP, Pthread, Serial_OpenMP, Serial_Pthread

- **Bowman (KNL):**
  - Intel 16.2 / 17.0
  - Serial, OpenMP, Pthread, Serial_OpenMP, Serial_Pthread

- **White (Power8):**
  - GCC 4.9.2 / 5.3.0; IBM XL 13.1.3
  - Serial, OpenMP, Serial_OpenMP

- **KokkosDev (IvyBridge + K40):**
  - GCC 4.7.2 / 4.8.4 / 4.9.2 / 5.1.0; Intel 14.4 / 15.2 / 16.1; Clang 3.5.1 / 3.6.1; Cuda 6.5.14 / 7.0.28 / 7.5.18
  - Serial, OpenMP, Pthread, Serial_OpenMP, Serial_Pthread, Cuda_Serial, Cuda_OpenMP, Cuda_Pthread
Integration Testing

- **Currently Required:**
  - Pass all nightly tests
  - Pass Trilinos Build and test with about 120 packages enabled
  - Intel/Serial, GCC/OpenMP, Cuda

- **Occasionally done (i.e. when I do the testing I did it):**
  - LAMMPS pass all tests
  - NALU pass all tests
  - Should we try and do make this required and also more end applications?

- Takes about 24 hours to get through the process if nothing goes wrong
Develop/Master cycle

- Upgrade to master in sync with Trilinos Integration
  - Trilinos always has the latest master
- Push to develop after testing subset of nightly tests
  - So far develop has been extremely stable
  - About 2-4 times a month nightly testing comes back non-clean
- Generally feeling:
  - Seems to be going pretty well
  - Not aware of breaking any existing feature with a new master drop
  - Sometimes new bugs not covered with test suite are found in field though
Compiler LifeCycle

- Compiler support is growing strongly
  - Two years ago 6 compilers
  - Now 16 in nightly, another 6-10 need to be added soon (GCC 6, Clang > 3.6, PGI, Cray, Cuda 8)

- Burden on testing resources and maintenance
  - Sometimes workarounds for compiler bugs necessary, the more compilers the more workarounds

- When can we drop older compilers?
  - Sems working on Sandia wide strategy
Discussion Summary

Testing
- Applications’ bitwise reproducibility requirement is a problem; need better success criteria
- test_all_sandia script to rigorously test patches before pull request
- gcc 6 not ready

When can we drop old compilers
- Challenge is big DOE machines – slow to update

Configuration and build
- Prefer that applications import Kokkos into their project and build in-place
- Configuration requires setting Kokkos macros through config header, -D command line, Makefile, Cmake
- Fine grain selection of debugging facilities? – yes
- Need memory consumption tracking; especially high-water-mark on GPU, UVM allocation count
- Stand-alone Cmake is not nightly tested; make a Kokkos.cmake for FIND_PACKAGE
- nvcc_wrapper – need to integration test snapshot into Kokkos
INITIALIZATION AND FINALIZATION
Initialization & Finalization

- Execution spaces must be initialize before use

  - Initialization should happen after MPI_Init(...) since it may change the process mask

  - Initialization creates threads, streams, singletons and other shared resources

  - Initialization of an execution space should only happen once
    Not valid to initialize-finalize-initialize...
Initialization

Preferred initialization methods

Initialize all enabled execution spaces

Kokkos::initialize()
Kokkos::initialize( int & argc, char *argv[] )
Kokkos::initialize( const InitArguments & )

```c
struct InitArguments {
    int num_threads = -1;  // number of host threads
    int num_numa   = -1;   // number of numa regions
    int device_id  = -1;   // GPU device id
};
```

Command line arguments (next slide)
Initialization

Kokkos::initialize( int & argc, char *argv[] )

The following arguments exist also without prefix 'kokkos' (e.g. --help). The prefixed arguments will be removed from the list by Kokkos::initialize(), the non-prefixed ones are not removed. Prefixed versions take precedence over non prefixed ones, and the last occurrence of an argument overwrites prior settings.

--kokkos-help : print this message
--kokkos-threads=INT : specify total number of threads or number of threads per NUMA region if used in conjunction with '--numa' option.
--kokkos-numa=INT : specify number of NUMA regions used by process.
--kokkos-device=INT : specify device id to be used by Kokkos.
--kokkos-ndevices=INT[,INT] : used when running MPI jobs. Specify number of devices per node to be used. Process to device mapping happens by obtaining the local MPI rank and assigning devices round-robin. The optional second argument allows for an existing device to be ignored. This is most useful on workstations with multiple GPUs of which one is used to drive screen output.
It is possible to initialize execution spaces explicitly

The host spaces must be initialized before the device spaces (Cuda)

Kokkos::ExecutionSpace::initialize()
Kokkos::ExecutionSpace::initialize(...)

The various execution spaces may take different arguments in their non-default initialization method
Initialization

Kokkos::Serial::initialize( ... ) // all arguments (up to 4) are ignored

Kokkos::OpenMP::initialize(
    unsigned thread_count = 0,
    unsigned use_numa_count = 0,
    unsigned use_cores_per_numa = 0
)

Kokkos::Thread::initialize(
    unsigned thread_count = 0,
    unsigned use_numa_count = 0,
    unsigned use_cores_per_numa = 0,
    bool allow_asynchronous_threadpool = false
)

Kokkos::Cuda::initialize(
    int cuda_device_id = 0,
    size_t num_cuda_streams = 1
)
Finalization

Finalization returns resources allocated in initialization

Should be called before returning EXIT_SUCCESS

Finalize all execution spaces initialized with Kokkos::initialize()
Kokkos::finalize()

Finalize all enabled execution spaces
Kokkos::finalize_all()

Explicitly finalize an execution space
Device spaces (i.e. Cuda) must be finalized before host spaces
Kokkos::ExecutionSpace::finalize()
Yield resources

All devices are initialized in a \textit{ready-to-work} state

Users can request that an execution space try to yield its resources by calling
\begin{verbatim}
Kokkos::ExecutionSpace::sleep()
\end{verbatim}

A corresponding wake command must be given before running parallel algorithms on the execution space
\begin{verbatim}
Kokkos::ExecutionSpace::wake()
\end{verbatim}
### Discussion Summary

**Initialization options**
- Desire to change CPU-architecture thread usage dynamically per kernel – defer to execution space instances
- Some kernels can effectively use hyperthreads, others cannot
- `hwloc` not working on KNL leading to problems with thread migration
- Multiple GPUs per MPI rank not necessarily a good idea, more work to manage
EXECUTION AND MEMORY SPACE INSTANCES
Execution and Memory Space Instances

- Current singleton-based spaces
  - ~ using MPI_COMM_WORLD
- Need for instance-based spaces
- Using instance-based spaces
  - ~ evolving to use MPI_Comm instances
- Potential strategies for managing instance-based spaces
  - ~ MPI_Comm topologies, splitting, ...
Current Singleton-based Memory Spaces

```cpp
struct MemorySpace {
    typedef MemorySpace memory_space;
    typedef /* preferred */ execution_space;
    void * allocate( size_t ) const;
    void deallocate( void *, size_t ) const;
    MemorySpace(); // access to singleton
};

Kokkos::HostSpace
Kokkos::CudaSpace
Kokkos::CudaUVMSpace
Kokkos::CudaHostPinnedSpace
Kokkos::Experimental::HBWSpace
```
Current Singleton-based Execution Spaces

```c++
struct ExecutionSpace {
    typedef ExecutionSpace execution_space;
    typedef /* preferred */ memory_space;
    typedef /* preferred */ array_layout;
    typedef /* preferred */ scratch_memory_space;
    typedef /* preferred */ size_type;
    static void fence();
    static int is_initialized();
    static void print_configuration( std::ostream &, bool);
    static void finalize();
    static void initialize( /* type dependent arguments */);
};
```

Kokkos::Serial
Kokkos::Threads
Kokkos::OpenMP
Kokkos::Cuda
Kokkos::Qthread
Using Current Singleton-based Spaces

\[
\text{parallel\_pattern( Policy<ExecSpace>(args), functor );}
\]
- Execute functor on \text{ExecSpace} according to pattern and policy

\[
\text{View< ArraySpec , Space >}
\]
- View of an array allocated in \text{Space::memory\_space}
- Allocation may execute element constructor in \text{Space::execution\_space}
- Deallocation may execute element destructor in \text{Space::execution\_space}

\text{Kokkos::Impl::VerifyExecutionCanAccessMemorySpace}<\text{ExecSpace::memory\_space , MemorySpace}>::\text{value}
- Can code executing in \text{ExecSpace} access memory in \text{MemorySpace} ?
- Used to catch memory access errors, instead of seg-fault
- Should something like this be public?
Need for Instance-Based Spaces

- Use multiple GPUs on a node
- Use multiple CUDA streams on a GPU
  - Enable overlap of deep copy and functor execution
  - Asynchronously launch multiple independent (large) parallel operations
- Group cores to use independently and asynchronously
  - Asynchronously launch multiple independent (large) parallel operations
  - Partition cores of manycore node, analogous to MPI_Comm_split
  - Group by NUMA affinity to reduce memory movement
- Required for inter-node asynchronous many-task (AMT)
  - AMT scheduler launches large tasks with internal Kokkos parallelism

Challenge: execution resource management

- Choosing GPU
- Selecting and partitioning CPU cores
- Execution resources <-> memory resource affinity
Using TBD Instance-based Spaces

\texttt{parallel\_pattern( policy(\textit{exec\_space},\textit{args}), \textit{functor} );}
- Execute functor on \textit{exec\_space} instance according to pattern and policy
- Define a default instance, \texttt{ExecSpace()}, for backward compatibility

\texttt{View< ArraySpec, SpaceType > \textit{a}( \textit{space}, \textit{args} );}
- View of an array allocated in \textit{space}.memory\_space\_instance()
- May execute element constructor in \textit{space}.execution\_space\_instance()
- May execute element destructor in \textit{space}.execution\_space\_instance()
- Default instance, \texttt{SpaceType:::memory\_space()}, for backward compatibility

\texttt{deep\_copy( \textit{exec\_space}, \textit{destination}, \textit{source} );}
- Synchronize asynchronous deep\_copy on \textit{exec\_space}

\texttt{verify\_accessible( \textit{exec\_space}, \textit{memory\_space} ) ?}
- Can code executing in \textit{exec\_space} access memory in \textit{memory\_space} ?
- Used to catch memory access errors, instead of seg-fault
- Space\_Type no longer sufficient to detect errors; e.g., multi-GPU scenario
TBD Instance-based Memory Spaces

```c
struct MemorySpace {
    typedef MemorySpace memory_space;
    typedef /* preferred */ execution_space;
    // Already converted to ‘instance’ API:
    void * allocate( size_t ) const;
    void deallocate( void * , size_t ) const;
    // preferred execution space instance:
    execution_space execution_space_instance() const;

    MemorySpace();     // default instance
    MemorySpace( identifier ); // particular instance
};
```

- Relationship of `identifier` to resource management?
TBD Instance-based Execution Spaces

```cpp
struct ExecutionSpace {
    typedef ExecutionSpace execution_space;
    typedef /* preferred */ memory_space;
    typedef /* preferred */ array_layout;
    typedef /* preferred */ scratch_memory_space;
    typedef /* preferred */ size_type;

    memory_space memory_space_instance() const;
    void fence() const;
    ExecutionSpace(); // default instance
    ExecutionSpace( identifier ); // particular instance

    static int is_initialized();
    static void print_configuration( std::ostream &, bool );
    static void finalize();
    static void initialize( init_args );
};
```

- **Relationship of identifier and init_args to resource management?**
  - Selection / partitioning of resources into instances
TBD Resource Management

- **ASC/ATDM Resource Manager; Prototype in FY17**
  - “Software Requirements for ATDM On-Node Resource Management” SAND2016-6357; Olivier, Pedretti, and Brightwell
  - Representing Machine Topology
    - execution resources, memory resources, and their relationships
  - Maintaining an Inventory of Node Resources
    - Present, available, assigned
    - Threads pinned to hardware location, blocks of memory with affinity
  - Assigning and Recovering Resource Allocations
    - “User” (e.g., Kokkos) requests and relinquishes resources

- **Kokkos / Resource Manager Integration**
  - Execution space is a collection of requested execution resources
  - Memory space is a requested memory resource

- **AMT / Kokkos / Resource Manager Integration**
  - AMT requests resources and gives them to Kokkos?
  - AMT works through Kokkos to request resources?
Assignment/converting Views with compatible memory spaces
- Allow fewer instantiations of algorithms
- Feature request is Github backlog

Execution Spaces
- The concurrency function is ambiguous for some spaces – GPU, does not mean the same thing as occupancy
- Possible to get actual GPU occupancy? We have an Impl:: function

Properties of Spaces
- For example bandwidth – have been holding off on this until instances are introduced
- A string label for spaces would be helpful for messages

Space Instances
- Why other than 1 MPI rank per NUMA? Don’t want to require MPI. NUMA regions are only the current complexity.
- Uintah has a working example of using execution space instances
- For backwards compatibility don’t require an instances – assume a default instance exists
- “MPI Sessions” disjoint stream of execution of MPI to support task parallelism – Ryan Grant
- Aggregate heterogeneous execution spaces with heuristics on which space to execute a functor?
  - Not in Kokkos. Higher layers of software could do this.
  - Uintah has worked this for years for the special case of structured meshes. Would never want to try and generalize it.
- Users need to propagate execution spaces through their code, a function to “get me the current execution space instance?”
- Does the complexity of multiple GPUs per MPI rank outweigh the benefits?
  - For simple use cases will keep the current interface via defaults, keep simple things simple
  - Enable more sophisticated applications to use multiple instances
- Mechanism to ask if an asynchronous dispatch is done?
- Accessibility between spaces needs more property information
- Allow running on a subset of hardware
- Not an AMT feature

Participants in future design dialogue: Kokkos team, Stephen Olivier, Brad Peterson (Utah), DARMA team, James Elliot, Geoffrey Womeldorff (LANL),
UTILITY TYPES
Kokkos Utility Types

- **Required because standard utility types**
  - Are not portable to CUDA, not marked with `__device__`
  - Do not have volatile functions for reductions (e.g., complex)
  - Are not layout manipulate-able

- `Kokkos::pair` <-> `std::pair`
- `Kokkos::complex` <-> `std::complex`
- `Kokkos::Array` <-> subset of `std::array` ; C++11
Kokkos::pair

template< class T1, class T2 >
struct pair {
    // match std::pair API, add KOKKOS_INLINE_FUNCTION
    // plus std::pair interoperability
    pair( const std::pair<U,V> & );
    std::pair<T1,T2> to_std_pair() const ;
};
Kokkos::complex

template< class T >
struct complex {
    // match std::complex API, add KOKKOS_INLINE_FUNCTION
    // add operator overloads for volatile qualifier
    // plus std::complex interoperability
    pair( const std::complex<T> & );
    operator std::complex<T> () const ;
};
Kokkos::Array

template< class T, size_t N , class Proxy = void >
struct Array {
    T & operator[](index);
    const T & operator[](index) const;
    T * data();
    const T * data() const;
    constexpr size_t size() const;
};

View< Array<T,N>**, Layout > a("A",N0,N1);

- For performance Layout should include the Array’s N dimension
- Storage should be as if View<T**,[N],Layout>, and it is.
  a(i,j) returns Array<T,N,proxy>

View<...>::reference_type s = a(i,j);

- Which acts just like Array<T,N>&
- Pattern for View of embedded AD and UQ types
Discussion Summary

- Can I reinterpret_cast to standard types? Can probably get away with it, may be fragile, Tpetra is getting away with it
- Kokkos::Array’s use of proxy really calls for the use of autor

Other standard library functions needed
- std::min, std::max, std::numeric_limits
- std::tuple (C++14 (CE says it's hairy))
- constexpr doesn't work (yet)
- Some CUDA functions want a nonconst reference when they should take a const reference; it's a compiler bug (JE, DS)
- quad support on CUDA
- cmath and some C libraries are not constexpr. CUDA has overloads for simple transcendental functions, but they are not constexpr. Would be nice to compute things at compile time.

- CUDA cannot throw exceptions, utilities don’t throw exceptions

- SIMD types? e.g., Mike Tupek’s wrapper types for AVX, etc. Github project libvec?
VIEW, SUBVIEW, AND EXTENSIBILITY
View, Subview, and Extensibility

- **Kokkos::View**: recent refactoring to
  - Support non-POD data construction and destruction
    - View-of-View, embedded AD and UQ, ... extensible
  - Reduce (minimize) burden of extension
  - Introduce extensible allocation properties
  - Align with C++20 proposal

- **Kokkos::subview**
  - How “auto” makes it easy
  - When explicit construction is feasible

- Extensibility for new layouts and special “scalar” types
Kokkos::View Template Parameters

template< typename ArrayType, typename ... Properties >
class View;

- **ArrayType**: scalar type, dynamic dimensions, static dimensions
  - `double**[8][3]` : 4D array with two dynamic and two static dimensions
  - Working to change C++20 standard to allow: `double[][][8][3]`

- **Properties**: Layout, Space, and Access Traits
  - In this order
  - Omitting Space causes Space = default Execution Space
  - Omitting Layout causes Layout = Space::array_layout
  - Omitting Traits is ”ordinary” managed View
Kokkos::View Multidimensional Shape

- **Current / old interface; deprecated & to be removed**

```cpp
class View { public:
    enum { Rank = /* number of dimensions */ };  
    size_t dimension_#() const;
    size_t size() const;
};
```

- **Revised interface; leveraging C++11 and aligning with C++20**

```cpp
class View { public:
    constexpr unsigned rank() const;  // soon
    constexpr unsigned rank_dynamic() const; // soon
    constexpr size_t extent( # ) const;
    constexpr int extent_int( # ) const;
    constexpr size_t size() const;
};
```

- **When ”# >= rank()” then extent(#) == 1**
Kokkos::View Multidimensional Shape

- **Reoccurring issue: return value of extent(#) and size()**
  - The standard way is size_t, an unsigned 64bit integer
  - But a for loop with signed integer is best performance
  - Either cast return value of extent(#) or get a warning
  - Requirement motivating ‘extent_int(#)’ function

- **Related issue: View may not need 64bit dimensions and strides**
  - Extra storage can add up with many Views in a functor

- **Possible solution:**

  ```
  View<ArrayType, Layout, Space, SizeType<int> >
  ```

  - View Properties... is easily extensible
  - Optional specification of View’s size_type, default to size_t
  - Verify at construction that size() <= maximum<

  size_type>
### Kokkos::View Multidimensional Layout

#### Current

class View { public:
    typedef /* */ array_layout;
    array_layout layout() const;
    size_t stride_#() const;
};

#### Planned interface; leveraging C++11 and aligning with C++20

class View { public:
    using array_layout = /* */;
    array_layout layout() const;
    constexpr size_t stride(#) const;
    constexpr size_t span() const;
};
Kokkos::View Member Access

class View { public:
    typedef /* */ reference_type;
    reference_type operator()( indices... ) const ;
    reference_type operator[]( index ) const ; // 1D only
};

- Requires number of indices >= rank()
- Extra indices are ignored
- Optional bounds checking
  - CMake: Kokkos_ENABLE_Debug_Bounds_Check=on
  - Make: KOKKOS_DEBUG=yes
  - #define KOKKOS_ENABLE_DEBUG_BOUNDS_CHECK 1
- Bounds checking requires 0 <= index# < extent(#)
- Bounds checking requires extra indices == 0
Kokkos::View Raw Memory Access

- **Current / old interface; deprecated & to be removed**

```cpp
class View { public:
    pointer_type_type ptr_on_device() const
    size_t capacity() const;
};
```

- **Revised interface; leveraging C++11 and aligning with C++20**

```cpp
class View { public:
    constexpr pointer_type data() const;
    constexpr size_t span() const;
    constexpr bool span_is_contiguous() const;
};
```

- Raw memory: `[ data() ... data() + span() )`  
- If `! span_is_contiguous()` then not every entry is a member  
- E.g., subview may skip entries, padding skips entries
Kokkos::View Memory Allocation

class View { public:
    View( label, dimensions... );
    View( label, layout ); // for strided layout
    View( view_alloc( properties... ), dimensions... );
};

- Requires number of dimensions == rank_dynamic()
- Initializes every allocated entry in the execution space
- properties... are any (or none) of the following
  - label, a string label
  - space, an execution or memory space instance
  - AllowPadding; default is to not allow padding
  - WithoutInitializing; default is to initialize entries

- For non-POD types allocation a destruction functor executes in the execution space immediately prior to deallocation
Kokkos::View Reference Counting

- Views are references to memory; Views are NOT containers
  - Pass Views to functions/functors by value, meta-data is small

- Reference counting: destroying or assigning last reference triggers deallocation
  - Destructor / assignment operator
  - Only applied to View allocated memory

- Subviews reference count the original allocation
  - Entire original view allocation is deallocated

- Views captured in a parallel functor stop reference counting
  - Avoid high-collision rate on atomic reference count operation
  - Reference count is on CPU, GPU cannot touch it

- Common error: View outlives Kokkos::finalize()
  - Attempt to deallocate on “shut down” execution space
Kokkos::View Your Raw Memory

class View { public:
    View( ptr , dimensions... );
    View( ptr , layout ); // for strided layout
    View( view_wrap( ptr ), dimensions... );
    static constexpr size_t required_allocation_size( dimensions... );
    static constexpr size_t required_allocation_size( layout );
};

- Requires number of dimensions == rank_dynamic()
- Raw memory is at least required_allocation_size, in bytes
- NO initialization, reference counting, deallocation, destruction
  - Your raw memory may contain ”garbage”
  - A problem for non-POD data types
Kokkos::View of Scratch Memory

class View {
    public:
        View( scratch_memory_space, dimensions... );
        static constexpr size_t required_allocation_size( dimensions... );
};

- Requires number of dimensions == rank_dynamic()
- Raw memory is at least required_allocation_size, in bytes
- No initialization, reference counting, deallocation, destruction
  - Memory may contain "garbage"
  - A problem for non-POD data types
- Execution space’s scratch memory spaces discussed later...
Kokkos::deep_copy

depth_copy( destination_value , source_view );
depth_copy( destination_view , source_value );
depth_copy( destination_view , source_view );

- **Views from different memory spaces**
  - Must have the same value type
  - Must be contiguous and equal size
  - Must be the same layout if rank() > 1

- **Views from the same (or compatible) memory space**
  - Must have assignable value types; e.g., double = int
  - Must be the same rank
  - May have different extents, the minimum is used
  - May have different layout
  - May trigger element-by-element parallel copy (e.g., transpose)
Kokkos::create_mirror_view

ViewType::HostMirror host_space_view = 
create_mirror_view( view );

- Returns a View accessible from host code that
  - Has the exact same value type, dimensions, and layout
  - Is ready for deep_copy
- If the original View memory space is accessible from host code then it returns a View to the same memory
- If the original View memory space is not accessible then it allocates a new View in HostSpace
  - Is a reference counted View
Kokkos::resize and Kokkos::realloc

```cpp
resize( view &, dims... );
realloc( view &, dims... );
```

- **Both reallocate memory**
  - Only the input view is modified
  - Other view’s of the previous allocated array are not effected
  - Previous allocated array is deallocated only if the last view
  - NO testing for same size

- **Resize**
  - Allocate, element-by-element copy, *possibly* deallocate
  - Two allocations – memory “high water mark”

- **Reallocate**
  - *Possibly* deallocate, NO copying, allocate
  - If deallocating avoids memory “high water mark”
Kokkos::subview

```cpp
auto s = subview( view , arg... );
```

- The number of `arg`... must equal `view.rank()`
- Returns a View; may have different rank, dimensions, and layout
- Each `arg` is an index, a range, or `Kokkos::ALL`
  - Index: corresponding dimension is contracted out
  - Range: corresponding dimension remains a dimension
  - ALL: corresponding dimension is preserved
  - `s.rank() ==` number of ranges in `arg`...

- We try to return View with the best performing layout
  - Is a non-trivial meta-programming challenge
  - Ongoing incremental improvements changes return View type
- Easiest to capture return type with `auto`
Kokkos::subview – Explicit Construction

\[
\text{auto } s = \text{subview}( \text{view}, \text{arg}... ); \\
\text{View}<...> s( \text{view}, \text{arg}... ); \text{ // explicit}
\]

- **When you know the subview-View type**
  - A contiguous rank-one subview or simple rank-two subview
  - Can construct subview-View in-place
  - Explicit construction of class’ subview-View member

- **View has C++11 move constructor and assignment**
  - \[
  \text{auto } s = \text{subview}(\text{view},\text{arg}...); \\
  \hspace{1em} \text{o Does not touch reference count} \\
  \hspace{1em} \text{o Does copy meta-data: pointer, dimensions} \\
  \hspace{1em} \text{o Small overhead}
  \]
Kokkos::View Extensibility

- Recent View implementation refactoring a complete redesign
  - Support non-POD data construction and destruction
    - View-of-View, embedded AD and UQ
  - Significantly reduce burden of extension: layout and special types
  - Introduce extensible allocation properties

- Redesign still needs refinement
  - Leverage C++11 and house cleaning to further reduce burden
  - Alignment with C++20 multidimensional array proposal

- Extension points in the Kokkos::Impl:: namespace
  - Subject to improvement / change

- Template meta-programming required
Kokkos::View Extensibility

- A little knowledge of View implementation:

```cpp
class View {
private:
    // Allocation reference counter and deallocator:
    Impl::SharedAllocationTracker m_track;
    // Mapping indices to array entry:
    Impl::ViewMapping<traits,void> m_map;
public:
    reference_type operator()( index... ) const
    {
        KOKKOS_VIEW_OPERATOR_VERIFY(( m_map, index... ))
        return m_map.reference( index... );
    }
};
```

- Extension == Specialization of Impl::ViewMapping
Kokkos::View Extensibility

Overview of how Impl::ViewMapping is used

- Mapping indices to array data, data member of View
  
  \[ \text{ViewMapping}\langle \text{view}\_\text{traits} \, , \, \text{void} \rangle \]

- Assignment of compatible Views:
  
  \[ \text{ViewMapping}\langle \text{dst}\_\text{view}\_\text{traits} \, , \, \text{src}\_\text{view}\_\text{traits} \rangle \]

- Type deduction and assignment of Subviews:
  
  \[ \text{ViewMapping}\langle \text{void} \, , \, \text{src}\_\text{view}\_\text{traits} \, , \, \text{subview}\_\text{arguments}... \rangle \]
Extension of Layout: Layout Data

- Create your *layout* type:

```c
struct layout {
    // Declare yourself to be an array layout,
    // std::is_same<layout, layout::array_layout>
    typedef layout array_layout;

    // Construct from list of integer values:
    layout( dimensions... );
};
```

- See Kokkos_Layout.hpp for examples
- Layout objects are temporary, used during View construction
  - Don’t do memory allocation / deallocation
  - Size members with Kokkos::ARRAY_LAYOUT_MAX_RANK
Extension of Layout: Offset Computation

- A little knowledge of ViewMapping implementation:
  ```cpp
  class ViewMapping<traits, enable> { 
  private:
    value_type * m_handle;
    ViewOffset<...> m_offset;
  public:
    constexpr value_type & reference(index... ) const 
    { return m_handle[ m_offset(index... ) ]; } 
  };
  ```

- m_handle
  - Simply a pointer for ordinary Views
  - Something special for CUDA texture cache, Atomic access, ...
  - Yet another extension point
Extension of Layout: Offset Computation

- Currently create specialization of:

  ```cpp
  ViewOffset< dimension, layout, enable > { 
    // Index mapping:
    constexpr size_t operator()( index... ) const ;
    constexpr size_t span() const ;
    constexpr size_t stride_#() const ;

    // Construct with optional padding:
    ViewOffset( const padding & , const layout & );
    // Compatible copy construction:
    ViewOffset( const ViewOffset<...>& );
    // Compatible subview construction:
    ViewOffset( const ViewOffset<...>& 
                 , const SubviewExtents<...>& );
  };
  ```
Extension of Layout: Subview Mapping

- Currently create specialization of:

  `ViewMapping< enable, src_traits, Args... >` {

    // Subview type deduced from `src_traits` and `Args...`
    typedef `View<...>` type;

    // Subview assignment:
    static void assign(
      `ViewMapping< dst_traits >` &
      , `ViewMapping< src_traits >` const &
      , Args ... args);

  };

- Non-trivial meta-programming to deduce `best` subview type
Extension for Layout of Special Types

- Specialization of Impl::ViewMapping\(<traits, void>\) 
  - \(\text{reference}_\text{type} \ \text{View::operator}()\(\text{(indices...)}\) \ const\;\)
  - returned \(\text{reference}_\text{type}\) may be a proxy, not value_type &

- Specialization of Impl::ViewMapping\(<dst_traits, src_traits>\)
  - Compatible View assignment

- Specialization of Impl::ViewMapping\(<void, src_traits, args...>\)
  - Subview type deduction and assignment

- Specialization of Impl::ViewDataAnalysis
  - Given View\(<\text{ArrayType}, \ldots>\)
  - Analyzes storage shape of \text{ArrayType} with respect to layout

- Working example in Kokkos for View\(<\text{Array<T,N>}, \ldots, \ldots>\)
- Working example in Trilinos for AD and UQ types
Discussion Summary

**Template parameters**
- Must be in proper positional order. This is different than execution policies where positional order not enforced
- Straw poll: more participants preferred uniformity of ordering over flexibility

**Multidimensional array shape**
- If size_type could be specified then would not need the extent_int(#) function
- No objection to deprecating dimension_#() functions
- Objection to non-uniformity of integer types; e.g., rank() is unsigned not size_type
- Signed types can give better loop performance due to language requirements on unsigned types
- Change to signed type by default? Straw poll YES. Have a macro that configures as signed or unsigned
- View< ... , SizeType<T> > ? Straw poll YES. Must be able to assign between View with different size_type

**Proposed C++ multidimensional array**
- Won’t be a container, won’t be allocator aware
- Link to proposal: [https://github.com/kokkos/array_ref](https://github.com/kokkos/array_ref)

**Layout**
- span() includes padding
- stride_#() replaced by stride(#)
- stride only meaningful if is_regular()

**Member access**
- Ignoring extra indices sounds error prone; bounds checking enforces extra indices to be zero
- Bounds checking on a per-View basis?
- Want option to require number of indices to match the rank

**Raw memory access**
- Expose alignment
Discussion Summary

Allocation
- label is for debugging
- can use DeviceType<ExecSpace,MemSpace> to specify both memory space and where construction and destruction occur
- If you don’t want default first-touch construction-initialization then you can allocate without initializing and initialize yourself
  - Must use placement-new for non-simple data types, such as in View-of-View
- Is View trying to do too much? Should it be split into different classes?
  - All features are tightly wound together, would give up performance, increase complexity of implementation, introduce code duplication
- How much of View do most users use? Try to keep simple use cases easy, and hard use cases not horrible.
- Common way to use View is set of typedef declarations
- Should label be optional?
- Need better evaluation of what is a simple type; Kokkos::complex does not pass the current evaluation

Could Kokkos expose its reference counting mechanism?

Viewing raw memory
- Will Kokkos provide View iterator interface? NO
- Could layout have a span() method?

Extensibility
- Vectorization? For standard layouts and simple data types we hoisted implementation into View manually to enable vectorization. Otherwise compilers have a hard time vectorizing through implementation.
- HostMirror does not match member typedef style, should be changed for consistency
- Custom layouts and special data types are dominant use cases for extensibility
  - (non-Kokkos developer) It’s not too bad, except for subview. I’ve done three implementations, most pain is subview
DYNAMIC RANK VIEW
Motivation

- Kokkos’ multidimensional array views designed with static rank for performance improvements
- Design assumption for various user codes: dynamic rank
- Several codes were redundantly developing this capability

Developed, in Kokkos, shared capability

- Compatible with static rank view
- Compatible with user codes’ design
- Expedited refactoring efforts
- Shared capability reduces user codes’ maintenance
API Comparison

**Static Rank View (3D)**

```c++
typedef T double;
typedef L Kokkos::LayoutRight;
typedef E Kokkos::HostSpace;
typedef MT void; //MemoryTraits

typedef
Kokkos::View<T**[3],L,E,MT>
ViewType;

ViewType v("v", 10, 20);
```

**Dynamic Rank View (3D)**

```c++
typedef T double;
typedef L Kokkos::LayoutRight;
typedef E Kokkos::HostSpace;
typedef MT void; //MemoryTraits

typedef
Kokkos::DynRankView<T,L,E,MT>
DynRankViewType;

DynRankViewType dv("dv", 10, 20, 3);
```

Type T and Rank info required

Only type T required

Type T and Rank info required
API Comparison

Static Rank View (3D)

typedef T double;
typedef L Kokkos::LayoutRight;
typedef E Kokkos::HostSpace;
typedef MT void; //MemoryTraits

typedef
    Kokkos::View<T**,[3],L,E,MT>
    ViewType;

ViewType v("v", 10, 20);

Dynamic Rank View (3D)

typedef T double;
typedef L Kokkos::LayoutRight;
typedef E Kokkos::HostSpace;
typedef MT void; //MemoryTraits

typedef
    Kokkos::DynRankView<T,L,E,MT>
    DynRankViewType;

Rank deduced from dimension args

DynRankViewType dv("dv", 10, 20, 3);
Features and Limitations

Features

- View – DynRankView
- Interoperability
  - Possesses most features of static rank views
  - Memory tracking, portability between execution spaces, subviews, memory traits
- Maximum Rank 7

Limitations

- 1-way assignment
  - Assign View to DynRankView
  - Copy View to DynRankView
- Implementation is like that of a rank 7 view – performance will be limited
- Maximum Rank 7
Discussion Summary

- Does dynamic-rank view replace use case of extra arguments at the end of View::operator()?
  - Similar motivation, but actual dynamic rank gives up too much performance
  - Dynamic rank view has larger storage; on GPUs this costs registers

- Views of FAD types with runtime size have extra hidden dimension
DUAL VIEW
DualView: Choose space at run time

- Kokkos lets you choose where data live, at compile time
- What if I want run-time choice?
  - LAMMPS: Users may write custom physics modules. They need not run on GPU, but must have access to data structures. Run-time script controls which modules run, in what order.
  - Trilinos: Not all solvers ported to GPUs. Some kernels faster on CPUs than GPUs, so solvers may want to run on CPU anyway.
  - Don’t want to rebuild entire app for every space; only rebuild low-level computational kernels for different architectures
- Kokkos::DualView makes this work
  - Users say where they want data’s most recent version to live
  - Users mark data as modified; “sync” only moves data if marked
Kokkos::DualView semantics

**Work on host (CPU)**
- Get a host view of the data & treat it as read only
- Manually mark host as modified
- Get Kokkos::View of host data; read & write data
- Tell DualView to sync to device

**Work on device (GPU)**
- Get a device view of the data & treat it as read only
- Manually mark device as modified
- Get Kokkos::View of device data; read & write data
- Tell DualView to sync to host

Host & device data are in sync
Idioms for using Kokkos::DualView

```cpp
template<class DevType>
void someUserCode (DualView<int*, DevType> dv) {
    typedef typename decltype(dv)::size_type size_type;
    {
        // Read-only access on host
        dv.template sync<HostSpace> (); // no modify
        auto dv_h = dv.template view<HostSpace> ();
        const size_type N = dv_h.dimension_0 ();
        int sum = 0;
        for (size_type i = 0; i < N; ++i) {
            sum += dv_h(i);
        }
        std::cout << "Sum on host: " << sum << std::endl;
    }
    {
        // Read-and-write access on device
        dv.template sync<DevType> ();
        dv.template modify<DevType> ();
        auto dv_d = dv.template view<DevType> ();
        typename typename DevType::execution_space exec_space;
        parallel_for (RangePolicy<exec_space, int> (0, 10),
        KOKKOS_LAMBDA (const int i) { dv_d(i)++; });
        // Convention: Leave DualView unsync'd
    }  
}
```

Sync just before use. No-op if not marked modified.
Leave DV unsync'd after modifying. Next code will sync for you.

Host View is a HostMirror of the device View.
Desired extensions

- Support for KNL HBM in progress
- Support for >2 memory spaces?
  - Interface supports this; impl. currently does not (easily fixed)
- Support for memory space instances?
  - Interface currently selects space as a compile-time type
  - What about multiple GPUs? MPI / PGAS windows?
- Lazy allocation?
  - Don’t want to consume limited HBM / GPU memory if not using it
  - Sync to “host” (DRAM) implies not using HBM / GPU; could free it
  - Current DualView design prevents this; could fix in either of 2 ways:
    - Store Views by std::shared_ptr (would prevent DV use on GPU)
    - “View of Views” (let Kokkos handle reference counting)
- Alternate “use” syntax? (see next slide)
Alternate syntax: "Use"

- `dv.template use (memorySpace, {Read, Write, ReadWrite});`
- Semantically equivalent to `sync + modify`
  - `use (M, ReadWrite) == sync<M>, modify<M>`
  - `use (M, Read) == sync<M>`
  - `use (M, Write) == modify<M> (SEE BELOW)`
- Some users find "sync" confusing (it’s 1 way: `sync to M`)
  - "Use" makes clear that you intend to access in that space
- DualView currently forbids `modify<M>` if not sync’d to `M`
  - It checks both spaces’ modified counts, & throws if both nonzero
  - This is useful for catching bugs, but forbids write-only access
  - Lazy allocation is the main use case for write-only access
    - Unallocated in `M` ➔ "not sync’d to `M""
    - "Sync to M" ➔ allocate in `M`, then copy; option to free source memory
Discussion Summary

- DualView pattern that was being repeated in applications

- Can I use more than 2 memory spaces? “MultiView” not “DualView”; API could support this but implementation does not
- Request for multiple memory spaces

- Prefer not to have to put template in front of methods like sync and modify

- Would like lazy allocation

- Alternative API to sync/modify: use( Space, { Read | Write | ReadWrite } )
  - More intuitive, more clear
  - A blocking call
  - ReadWrite unclear, perhaps InputOutput
  - Better for lazy allocation
  - more aligned with AMT drivers
  - Read/Write traits on Views?
  - Straw poll: this interface is preferred

- Two different memory spaces with same execution space?

- Can DualView be unmanaged? yes
UNORDERED MAP / SET
Atomic Bitset

Bitset<MemorySpace> b(size)

Length of bitset — host & device
b.size()

Number of bits which are set — host & device
b.count()

Set all bits to 1 — host
b.set()

Set all bits to 0 — host
b.reset()

Set bit 'i' to 1 — device
b.set(i)

Set bit 'i' to 0 — device
b.reset(i)

Is bit 'i' set — device
b.test(i)

Find any set bit in same block as hint
returns true if found, otherwise return new hint — device
pair<bool, unsigned> result =
   b.find_any_set_near(hint, direction)

Find any unset bit in same block as hint
returns true if found, otherwise return new hint — device
pair<bool, unsigned> result =
   b.find_any_unset_near(hint, direction)

Direction is used to reduce contention of find
BIT_SCAN_FORWARD_MOVE_HINT_FORWARD
BIT_SCAN_FORWARD_MOVE_HINT_BACKWARD
BIT_SCAN_BACKWARD_MOVE_HINT_FORWARD
BIT_SCAN_BACKWARD_MOVE_HINT_BACKWARD
Unordered Map/Set

A performance-portable, lock-free unordered map/set
For an unordered set let Value equal void

template < typename Key
, typename Value
, typename Device = Kokkos::DefaultExecutionSpace
, typename Hasher = hash<Key>
, typename EqualTo = equal_to<Key>
>
class UnorderedMap;

Capacity hint is an initial guess of how many unique keys will be inserted into the map

UnorderedMap( size_t capacity_hint
, Hasher hasher = Hasher()
, EqualTo equal_to = EqualTo() )
Unordered Map/Set

Index based instead of pointer based to allow copying between memory spaces

For a 'set' the value may be omitted — device only

InsertResult r = map.insert(key, value)

struct InsertResult {
    bool success();
    bool existing();
    bool failed();
    unsigned index();
};
Unordered Map - Iterate

- To iterate, invoke a parallel_for over the capacity of the map
- Check an index with valid_at() before accessing the key/value

```cpp
parallel_for( map.capacity(), KOKKOS_LAMBDA( unsigned i ) { if (map.valid_at(i)) {
    auto key = map.key_at(i);
    auto value = map.value_at(i);
    ...
}
});
```
Unordered Map - Find / Exists

- To find the value associated with a key

```cpp
unsigned idx = map.find(key);
if (map.valid_at(idx)) {
    ...
}
```

- To check if a key exists

```cpp
bool exists = map.exists(key);
```
Unordered Map - erase

- To erase, call begin_erase() on the host
- Run a parallel functor to mark keys as erased
- call end_erase() on the host to delete the keys

```cpp
// on host
map.begin_erase();
...
// on device
bool erased = map.erase(key);
...
// on host
map.end_erase();
```
Unordered Map - rehash

- If there are failed inserts
  Rehash the map to a larger size and retry insertion algorithm

```cpp
while (map.failed_inserts()) {
    map.rehash( 2*map.capacity() );
    parallel_for(..., KOKKOS_LAMBD A(int i) {
        ...
        map.insert(...);
    });
}
```
Discussion Summary

- Confusing that insertion has three possible results; standard library throws on failure, we can’t throw
- TODO: add emplace function
- TODO: map.size() should throw if map is in erasable phase
- Sum-into (+) interface in addition to assignment interface? Could be handled by proxy object, just like View<...,Atomic>
- Want to converge with KokkosKernels hash map; KokkosKernels hash map can work on scratch memory
PARALLEL PATTERNS
Outlook

- Patterns
- Policies
- Functors
- Reducers
Patterns

- **Parallel_for**
  - Execute a policy specified “index range”
  - All iterations are independent
  - No order or concurrency guarantee

- **Parallel_reduce**
  - Like parallel_for except:
  - Reduction Dependency
  - By default does a sum operation
  - Custom Reductions Ops possible

- **Parallel_Scan**
  - Like parallel_for except
  - Pre-/postfix scan dependency between iterations
Patterns

- `parallel_for ( Policy , Functor )`
- `parallel_for ( Label , Policy , Functor )`

- `parallel_reduce ( Policy, Functor )`
- `parallel_reduce ( Label, Policy, Functor )`
- `parallel_reduce ( Policy, Functor, Result/Reducer )`
- `parallel_reduce ( Label, Policy, Functor, Result/Reducer )`

- `parallel_scan ( Policy , Functor )`
- `parallel_scan ( Label , Policy , Functor )`
Policies

- RangePolicy
- TeamPolicy
- An Integer
  - This is like RangePolicy<DefaultExecutionSpace>
template<class ... Properties>
class RangePolicy public Impl::PolicyTraits<Properties ... > {

  //! Tag this class as an execution policy
  typedef RangePolicy execution_policy;
  typedef typename traits::index_type member_type;

  //! Not really public interface
  KOKKOS_INLINE_FUNCTION const typename traits::execution_space & space() const;
  KOKKOS_INLINE_FUNCTION member_type begin() const;
  KOKKOS_INLINE_FUNCTION member_type end() const;

  //! Constructors
  RangePolicy(const RangePolicy&) = default;
  RangePolicy(RangePolicy&&) = default;

  inline RangePolicy()
  inline RangePolicy( const typename traits::execution_space & work_space
                        , const member_type work_begin, const member_type work_end)
  RangePolicy( const member_type work_begin, const member_type work_end)

  /** \brief return chunk_size */
  inline member_type chunk_size() const;
  /** \brief set chunk_size to a discrete value*/
  inline RangePolicy set_chunk_size(int chunk_size_) const;
};
Template Properties

- Arguments can come in any order, all are optional
- ExecutionSpace: some execution space
- Schedule: a HINT for Kokkos how to schedule iterations
  - This does not guarantee any behavior, but should be treated as a performance optimization
  - Kokkos::Schedule<Kokkos::Static> (default), Kokkos::Schedule<Kokkos::Dynamic>
- IndexType: what internal integer type should be used
  - Kokkos::IndexType<TYPE>
  - Deprecated: just giving TYPE as a raw argument
- TagType: some tag type which is used to call operator of functor
  - Lets you have multiple operators in same class

RangePolicy<OpenMP,Kokkos::Schedule<Dynamic>,IndexType<int>, InitialInitializeTag>
Deeper down ...

```cpp
struct PolicyTraitsBase {
    using execution_space = ExecutionSpace;
    using schedule_type = Schedule;
    using work_tag = WorkTag;
    using index_type = IndexType;
    using iteration_pattern = IterationPattern;
};
```
How to Set Optional Arguments

- Currently it is policy.set_OPTION()
- Returns new policy object

- Alternatives
  - Modify existing policy
  - Optional argument on constructor: policy(..., Schedule(Dynamic))
    - This was tested initially, but is worse for compile time
Functor Parallel For

- Assume from now that
  - member_type is PolicyType::member_type
  - work_tag is PolicyType::work_tag

```cpp
class Functor {
    typedef ExecSpace execution_space; // Like to get rid of this
    KOKKOS_INLINE_FUNCTION
    void operator() (const member_type& i) const;
    void operator() (const work_tag, const member_type& i) const;
};
```

- execution_space
  - RangePolicy typedef has precedence
  - Matters only for launching with “N”
class Functor {
  typedef ExecSpace execution_space;
  typedef Scalar value_type;

  KOKKOS_INLINE_FUNCTION
  void operator() (const member_type& i, value_type& val) const;
  void operator() (const work_tag, const member_type& i, value_type& val) const;
};

• value_type
  • Is optional
  • Usually deducted from operator (and or Cuda return argument)
  • Maybe we should get rid of this?
Functor Parallel Reduce: Array

```cpp
class Functor {
  typedef ExecSpace execution_space;
  typedef Scalar value_type[];
  unsigned value_count;

  KOKKOS_INLINE_FUNCTION
  void operator() (const member_type& i, value_type val[]) const;
  void operator() (const work_tag, const member_type& i, value_type val[]) const;
};
```

- Do a reduction where each operand is a runtime sized array
  - For example small histogram, dimension dependent virial
  - Reduction is element wise (i.e. the result is a vector of length `value_count`)
Functor Custom Reduction

class Functor {
    ...
    void init(value_type& val) const;
    void join(volatile value_type& val, const volatile value_type& update) const;
    void final(value_type& val); // optional
    ...
};

- Init is called on each thread private variable once
  - I.e. the update value in the operator doesn’t necessarily have the init value

- Join is used to combine thread contributions
  - Updating the value in the operator should do the same operation

- Final is called once at the end
  - For example take sqrt on reduction value for norm
  - Can be used to store away the result
  - Final is necessary if no result or reducer argument is given to parallel_reduce

- All three can have a work_tag argument in first place
Functor with work_tag

```cpp
struct Integrator {
    struct InitialTag{};
    struct FinalTag{};
    void operator() (InitialTag&, const int &i) {
        ...
    }
    void initial_integrate() {
        parallel_for(RangePolicy<InitialTag>(0,N),*this);
    }
    void operator() (FinalTag&, const int &i) {
        ...
    }
    void final_integrate() {
        parallel_for(RangePolicy<FinalTag>(0,N),*this);
    }
};
```
Return Arguments

- Remember:
  - parallel_reduce (Label, Policy, Functor)
  - parallel_reduce (Label, Policy, Functor, Scalar)
  - parallel_reduce (Label, Policy, Functor, Pointer)
  - parallel_reduce (Label, Policy, Functor, View)
  - parallel_reduce (Label, Policy, Functor, Reducer)

- Restriction:
  - Nothing: functor must provide final function
  - Pointer: functor must have value_count
  - Must match operator argument
Reducer

- Provide Custom Reduction Operator for Lambdas
- Provide a set of build-in reducers

```cpp
class Reducer {
    typedef Reducer reducer_type;
    typedef Scalar value_type;
    typedef View<...> result_view_type;

    void init(value_type& val) const;
    void join(value_type& val, const value_type& update) const;
    void join(volatile value_type& val, const volatile value_type& update) const;
    void finale(value_type& val);
    result_view_type result_view() const;
};
```
Build-In Reducers

- Take a scalar or a view of a scalar as construction argument
  - May take an initialization argument if the type is not arithmetic type known to C++
- Based on MPI Reduction Ops
- Sum, Prod, Min, Max, MinLoc, MaxLoc, MinMaxLoc, LAnd, LOr, LXor, BAnd, BOr, BXor
- Templated on Scalar Type (and optional a space argument for ReturnView)
- TODO: Put in appropriate safe guards other than “It doesn’t compile” for Scalar types (i.e. only integer types work for some reducers)
Reducer Example:

double result;
parallel_reduce(N, KOKKOS_LAMBDA (const int& i, double& val) {
    if(val<a[i]) val = a[i];
}, Max<double>(result));
Discussion Summary

- Do scans other than sums make sense? A ”max” instead of “+” use case

RangePolicy
- returns execution space instance by const reference, imply that instances have shallow copy semantics? Current thinking, but haven’t committed yet
- DARMA passes around equivalent of space instance by value
- Why does RangePolicy have tagged template parameters but View has positional template parameters?
  - Tagging clarifies IndexType
  - Relies heavily on C++11, View came before C++11
  - Is there an intuitive ordering of RangePolicy template parameters?
- Optional runtime policy arguments will be given to constructor – post review decision
  - Prior experiments showed compiler performance penalty
  - New implementation strategy resolved this
- TODO: merge in multidimensional range policy

TeamPolicy
- Tightly nested loops with fixed ranges can use (experimental) multidimensional range policy
- This is for non-trivial nested parallelism
- Selected team size too large for hardware: Straw poll TODO generate error instead of silently reducing team size
- TODO Team reduce accept new reducer, default to sum
- Nested parallel operations are team collectives
- CPU vs. GPU for vector parallelism is confusing; document incorrect behavior in programming guide
- Vector parallelism is not “performance portability” it is writing to the hardest architecture
  - This is the complicated feature for the most difficult architecture
- Team policy is overkill when simple multidimensional range policy would be sufficient
- Unstructured codes need team policy with variable length nested parallelism

Scratch Memory
- Smaller level means closer memory
- team_shmem_size inputs int, outputs unsigned, don’t like this
- Prefer keyword arguments to integer list
ATOMICS
Atomic Functions

- Functions work on a memory address and a operand
- May return old or new value (atomic_fetch_OP, atomic_OP_fetch)
- Work on any size object (does it work on NonPOD?)

```c
T atomic_compare_exchange( volatile T * const dest, const T compare, const T val )
T atomic_exchange   (volatile T * const dest, const T val)
void atomic_assign   (volatile T * const dest, const T val)
void atomic_increment(volatile T * const dest) // Can be faster than fetch_add
void atomic_decrement(volatile T * const dest) // Can be faster than fetch_add
T atomic_fetch_add  (volatile T * const dest, const T val) // No add_fetch
T atomic_fetch_sub  (volatile T * const dest, const T val) // No sub_fetch
T atomic_fetch_max  (volatile T * const dest, const T val)
T atomic_fetch_min  (volatile T * const dest, const T val)
T atomic_fetch_mul  (volatile T * const dest, const T val)
T atomic_fetch_div  (volatile T * const dest, const T val)
T atomic_fetch_mod  (volatile T * const dest, const T val)
T atomic_fetch_and  (volatile T * const dest, const T val)
T atomic_fetch_or   (volatile T * const dest, const T val)
T atomic_fetch_xor  (volatile T * const dest, const T val)
T atomic_fetch_lshift(volatile T * const dest, const unsigned int val)
T atomic_fetch_rshift(volatile T * const dest, const unsigned int val)
```
Atomic Views

- Add memory trait
- Currently size restricted to 64bit, can be changed
- Views are non-referenceable, they give back a meta object
- Believe all operators are implemented (arithmetic, logical, comparison, shift)
- Volatile and non-volatile overloads

Kokkos::View<DataType,...,Kokkos::MemoryTraits<Kokkos::Atomic> > a_atomic = a;
Discussion Summary

- Why are these not tied to execution space?
- These are not C++11’ish; C++11 atomics don’t work for use because you must have atomic types
- What about large types? Use a lock array
- Cannot compare-exchange on types that pointer chase
- memory order? strong / sequential

Atomic Views
- Cannot do atomic views of data types larger than 64bits; could remove this restriction
- Aliasing to non-atomic is legal, just cannot access through both views
- Not an allocation, atomic view of an existing allocation
RANDOM NUMBERS
Random Number Generators

- They live in algorithms subpackage, must be explicitly included Kokkos_Random.hpp (not included with Kokkos_Core.hpp)
- Two generators: 1024bit and 64bit state (XorShift1024 XorShift64)
- Both are as good or better than Mersenne Twister => should be suitable for Monte Carlo and other high random quality needs

General principal:
- Create a pool of random number generators on host (Random_NAME_Pool)
- Threads grep a specific generator from the pool (Random_NAME)
- Threads use their generator independently
- Threads give the generator instance back to the pool
- There is also a function interface

Details:
- On CPU backends there is 1-to-1 relationship of generators to threads => reproducible
- On GPUs potentially billions of threads, but only 30k active => threads pull lock a generator from pool atomicly => runs are non-deterministic
Generator Pool Public Interface

template<class DeviceType = Kokkos::DefaultExecutionSpace>
class Random_XorShift64_Pool {
    Random_XorShift64_Pool();
    Random_XorShift64_Pool(uint64_t seed);
    Random_XorShift64_Pool(const Random_XorShift64_Pool& src);

    KOKKOS_INLINE_FUNCTION
    Random_XorShift64<DeviceType> get_state() const;

    KOKKOS_INLINE_FUNCTION
    void free_state(const Random_XorShift64<DeviceType>& state) const;
};
Generator Public Interface

class Random_XorShift64 {
    typedef DeviceType device_type;
    enum {MAX_URAND = 0xffffffffU};
    enum {MAX_URAND64 = 0xffffffffffffffffULL-1};
    enum {MAX_RAND = static_cast<int>(0xffffffff/2)};
    enum {MAX_RAND64 = static_cast<int64_t>(0xffffffffffffffffLL/2-1)};
    Random_XorShift64 (uint64_t state, int state_idx = 0)
    uint32_t urand() // [0 .. MAX_URAND)
    uint32_t urand(const uint32_t& range) // [0 .. Range)
    uint32_t urand(const uint32_t& start, const uint32_t& end ) // [start .. end)
    uint64_t urand64() // [0 .. MAX_URAND64)
    uint64_t urand64(const uint64_t& range) // [0 .. range)
    uint64_t urand64(const uint64_t& start, const uint64_t& end ) // [start .. end)
    int rand() // [0 .. MAX_RAND)
    int rand(const int& range) // [0 .. range)
    int rand(const int& start, const int& end ) // [start .. end)
    int64_t rand64() // [0 .. MAX_RAND64)
    int64_t rand64(const int64_t& range) // [0 .. range)
    int64_t rand64(const int64_t& start, const int64_t& end ) // [start .. end)
    float frand() // [0 .. 1]
    float frand(const float& range) // [0 .. range)
    float frand(const float& start, const float& end ) // [start .. end)
    double drand() // [0 .. 1]
    double drand(const double& range) // [0 .. range)
    double drand(const double& start, const double& end ) // [start .. end)
    double normal() // normal gaussian distributed with mean=0 and std_dev = 1.0
    double normal(const double& mean, const double& std_dev=1.0)
};
Rand class static functions

- Main purpose: support use case templated on Scalar type
- Still requires a generator
- Implemented for: `char, short, int, unsigned int, long, unsigned long, long long, unsigned long long, float, double, complex<float>, complex<double>`

```cpp
template<class Generator, Scalar>
struct rand {
    static Scalar max ()
    static Scalar draw (Generator& gen)
    static Scalar draw (Generator& gen, const Scalar & range)
    static Scalar draw (Generator& gen, const Scalar & start, const Scalar & end)
};
```
Discussion Summary

- Not feasible to be reproducible on GPUs
- How seeded? Initialize pool of generators with seeded random numbers
- Does NVIDIA intend to support new C+11 standard library random number generation? TODO to inquire
- Determinism? On CPU yes, on GPU no
SORTING
Sorting

- Part of Algorithms
- Implemented bin based sorting
  - Uses extra memory, but is well parallelizable and O(N)
- Multi Step: Create Permute Vector, then sort based on vector
  - Advantage: sort any number of arrays, but only once establish order
- You can implement your own Binning Operator
  - BuildIn is 1D and 3D
- A simple sort(view) function exist for 1D views
Public Interface BinSort

template<class KeyViewType, class BinSortOp,
    class ExecutionSpace = typename KeyViewType::execution_space,
    class SizeType = typename KeyViewType::memory_space::size_type>
class BinSort {

    BinSort(key_view_type keys, BinSortOp bin_op, bool sort_within_bins = false);
    void create_permute_vector();

    template<class ValuesViewType>
    void sort(ValuesViewType values);

    // Get the permutation vector
    KOKKOS_INLINE_FUNCTION
    offset_type get_permute_vector() const { return sort_order;}

    // Get the start offsets for each bin
    KOKKOS_INLINE_FUNCTION
    offset_type get_bin_offsets() const { return bin_offsets;}

    // Get the count for each bin
    KOKKOS_INLINE_FUNCTION
    bin_count_type get_bin_count() const { return bin_count_const;}
};
BinOp Interface

template<class KeyViewType>
struct DefaultBinOp1D {
  DefaultBinOp1D(int max_bins__, typename KeyViewType::const_value_type min,
                  typename KeyViewType::const_value_type max);

  // Determine bin index from key value, takes view + index instead of scalar
  // to support multi dim views
  template<class ViewType>
  KOKKOS_INLINE_FUNCTION
  int bin(ViewType& keys, const int& i) const;

  // Return maximum bin index + 1
  KOKKOS_INLINE_FUNCTION
  int max_bins() const;

  // Compare to keys within a bin if true new_val will be put before old_val
  template<class ViewType, typename iType1, typename iType2>
  KOKKOS_INLINE_FUNCTION
  bool operator() (ViewType& keys, iType1& i1, iType2& i2) const;
};
Performance of bin sorting is data dependent
Targeting physics applications with spatial data that has balanced histogram

- TODO: Move BinOp to public namespace
- Needs 3 kinds of sort: general purpose, Radix, Bin
- Switch from bubble sort to insertion sort within bins
- Graph algorithms need comparison sort
MULTIDIMENSIONAL RANGE POLICY
Multidimensional Range Policy

Motivation

- Tightly nested loops
- Efficient support for structured grid
- Need parallelism across multiple dimensions
- Kokkos to provide an optimized capability
  - Architecture and/or Layout specific optimizations

Experimental capability currently available

- kokkos/core/src/KokkosExp_MDRangePolicy.hpp
Experimental::MDRangePolicy

- **New PolicyTraits: IterationPattern**

  Rank< N, Outer = Iterate::Default, Inner = Iterate::Default>

  ```cpp
  enum class Iterate { Default, Left, Right, Flat };
  ```

  - Outer specifies direction between tiles
  - Inner specifies direction within tiles
  - Tile dimensions default to reasonable values
  - *Still experimenting to determine good defaults*

- **Construction – initializer lists:**

  - `MDRangePolicy< Rank<N> >(upper);`
  - `MDRangePolicy< Rank<N> >(lower, upper);`
  - `MDRangePolicy< Rank<N> >(lower, upper, tiles);`
Experimental::md_parallel_for

**Set only the upper bound (lower defaults to 0)**
md_parallel_for(
    MDRangePolicy( Rank<3>, ...)({I,J,K}),
    KOKKOS_LAMBDA (int i, int j, int k) {
        ...
    });

**Set the lower and upper bound**
md_parallel_for(
    MDRangePolicy( Rank<3>, ...)({I0,J0,K0}, {I1,J1,K1}),
    KOKKOS_LAMBDA (int i, int j, int k) {
        ...
    });

**Set the tile (workset) size**
md_parallel_for(
    MDRangePolicy( Rank<3>, ...)({0,0,0}, {I,J,K}, {T0,T1,T2}),
    KOKKOS_LAMBDA (int i, int j, int k) {
        ...
    });
**Optimization Approach - Tiling**

**Tiling** – Partition loop ranges into subsets (tiles) and run a parallel algorithm over tiles

A tile is comparable to the 1D chunk_size
Discussion Summary

- NOT for ragged index spaces, extents must be independent
- Outer loop over tiles, inner loop within tiles
- Why “Left” and “Right” not “LayoutLeft” and “LayoutRight”? Still experimental
- Plan to merge into RangePolicy
- Doesn’t tiling performance depend on the device? Defaults chosen according to device.
- If “Flat” means no tiling then just call it “NoTiling”
- Could transpose iteration space instead of array layout for memory access pattern; requires tightly nested loop access
- parallel_scan doesn’t make sense due to ambiguous ordering
MEMORY POOL ALLOCATOR
Superblocks only assigned to block sizes at first allocation
Once a block size is assigned a superblock, it will always be assigned a superblock
Each superblock has bitset indicating allocated status of each block
template <typename Device>
class MemoryPool {
    MemoryPool( const Device::memory_space & memspace,
                size_t total_size,
                size_t log2_superblock_size = 20 );

    void * allocate( size_t alloc_size );

    void deallocate( void * alloc_ptr, size_t alloc_size );

    bool is_empty();
};
MemoryPool( const Device::memory_space & memspace,
    size_t total_size,
    size_t log2_superblock_size = 20 );

- Memory controlled by pool is divided into superblocks
- total_size rounded up to nearest multiple of superblock size
- Max superblock size is $2^{31}$
- For many usages, the default superblock size is fine
- For performance reasons, a superblock is full before all locations are allocated, so you need some cushion in total_size
Allocation / Deallocation

```c
void * allocate( size_t alloc_size );
void deallocate( void * alloc_ptr, size_t alloc_size );
```

- Allocate returns NULL on failure
- Can allocate any size up to superblob size
- Chooses smallest block size >= allocation size
- Both involve atomic Boolean on page’s bitset word
- Parallelism comes from scattering allocations / deallocations across pages, so superblob needs enough pages to get parallelism
- Locking only occurs when switching a block size’s superblob during allocation and only on that block size
- Only one thread per warp does allocation or deallocation
Checking for Empty

bool is_empty();

- Expensive to fully check, so do quick partial check
- Returns true only if all superbblocks are “full”
- Allocations could still happen from the active superbblock if it isn’t completely full even when is_empty() returns true.
- When using multiple block sizes, could be empty for one block size when returns false because other block size has allocations available
- Most accurate would be to just to try to allocate
Discussion Summary

- Designed / optimized for small allocations and GPU; with decent performance on CPU

- Relaxing restriction of only one thread per warp allocating would cost performance
  - Allocation should be collective operation over a warp (team)

Scratch Memory for RangePolicy?
- KokkosKernels has its own memory pool
- Iteration-private scratch memory is a common pattern
- Could develop a different range policy that gives index and handle for scratch memory

Compared to std::allocator
- add typedefs
- Is not a stateless class, so wouldn’t work in std::vector

- Two different implementations, optimized for CPU and for GPU
- Apply memory pool to scratch / shared memory?
- Use shared memory until exhausted then switch to global memory? Very different design
DYNAMIC SIZE VIEW
Dynamic Size View

Kokkos::Experimental::
DynamicView< type *, properties... >

- Restricted to rank-one array
- Growable from within a parallel kernel
  - RangePolicy (not), TeamPolicy, TaskPolicy
- Growable and shrinkable from host process
- Thread safe and thread scalable
  - Uses memory pool
  - Uses chunking strategy
**Chunking Strategy**

```cpp
reference operator()( index ) const
{ return pa[index/chunk][index%chunk]; }
```

- \( pa \): array of pointers to chunks of array entries
- Chunks are individually allocated / deallocated as necessary
  - Fixed chunk size

**DynamicView<...>**

```cpp
a( label, memory_pool, N_upper_bound )
```

- **Construct with** \( N_{\text{upper bound}} \) potential number of entries
  - Allocation: \( pa[N_{\text{upper bound}}/\text{chunk}] = \{0,...\} \)
  - Memory not allocated for array entries
Resizing

DynamicView::resize_serial( N );
- Only called outside of parallel kernel
- \( N \leq N_{\text{upper\_bound}} \)
- Allocate or deallocate chunks as necessary

DynamicView::resize_parallel( N );
- Only called inside of parallel kernel
- \( N \leq N_{\text{upper\_bound}} \)
- Allocate chunks until: \( N \leq \text{size()} \)

DynamicView::size();
- Outside of parallel kernel is actual size
- Inside of parallel kernel is size at instant of call
Discussion Summary

- Work in RangePolicy? No, allocation is a team collective but no need to broadcast result
- Performance comparison to segmented view? TBD
TASK DAG
Dynamic Directed Acyclic Graph (DAG) of Tasks

- **Extension of Parallel Pattern**
  - Tasks: Heterogeneous collection of parallel computations
  - DAG: Tasks may have acyclic “execute after” dependences
  - Dynamic: New tasks may be spawned by executing tasks

- **Extension of Execution Policy**
  - Schedule tasks for execution
  - Manage tasks’ dynamic lifecycle
Anatomy and Life-cycle of a Task

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

**Life-cycle:**

- **constructing**
- **waiting**
- **executing**
- **complete**

- **serial task** on a single thread
- **task with internal data parallelism** on a thread team
Dynamic Task DAG Challenges

- Portability, especially GPU’s simple cores and minimal runtime
  - Executing tasks cannot block, wait, or yield
  - GPU function pointer accessibility on host and device
  - Dynamic – creating tasks within executing tasks on GPU
  - Requires CUDA 8 with relocatable device code option

- Performance
  - Thread scalable allocation/deallocation within finite memory
  - Execution overhead and thread scalable scheduling

- Non-blocking Tasks
  - *Respawn* instead of wait
  - Reduces complexity and overhead of implementation
Managing a Non-blocking Task’s Lifecycle

- **Spawn: enqueue to scheduler**
  - By main process or within another task
  - Allocate from a memory pool
  - User constructs internal data
  - Assign DAG dependences
  - Enqueue to scheduler

- **Respawn: re-enqueue to scheduler**
  - Instead of the task waiting or yielding
  - Can reassign DAG dependences
  - Reconceived wait-for-child-task use case
    - Create & spawn child task(s)
    - Reassign DAG dependence(s) to new child task(s)
    - Re-spawn to execute again after child task(s) complete
API via (silly) Fibonacci Illustration

template< typename ExecSpace > struct FibonacciTask;

template< typename ExecSpace >
long fibonacci( long n ) // F(n) = F(n-1) + F(n-2)
{
    typename ExecSpace::memory_space mem_space_inst;
    const size_t memory_pool_size = 10000 ; // bytes
    Kokkos::TaskPolicy<ExecSpace>
        policy( mem_space_inst , memory_pool_size );
    Kokkos::Future<ExecSpace,long> f =
        policy.host_spawn( FibonacciTask(policy,n) , Kokkos::TaskSingle );
    Kokkos::wait( policy ); // all tasks complete
    return f.get();
}
API via (silly) Fibonacci Illustration

template< typename ExecSpace > struct FibonacciTask {
    using value_type = long; // Future's value type
    using policy_type = Kokkos::TaskPolicy<ExecSpace>;
    using future_type = Kokkos::Future<ExecSpace,long>;
    using member_type = typename policy_type::member_type;

    // NOT required to be const
    KOKKOS_INLINE_FUNCTION
    void operator() ( const member_type &, value_type & );

    policy_type policy;
    future_type child[2];
    value_type N;

    KOKKOS_INLINE_FUNCTION
    FibonacciTask ( const policy_type & argP, long argN )
        : policy(argP), child{}, N(argN) {}
};
API via (silly) Fibonacci Illustration

```cpp
void operator()( const member_type & , value_type & result ) {
    if ( N < 2 ) { result = N ; }
    else if ( child[0].is_null() ) { // Called first time
        child[0] = policy.task_spawn( FibonacciTask(policy,N-1), Kokkos::TaskSingle );
        child[1] = policy.task_spawn( FibonacciTask(policy,N-2), Kokkos::TaskSingle,
                                      Kokkos::TaskHighPriority );
        policy.respawn( this , policy.when_all(2,child), Kokkos::TaskLowPriority );
    }
    else { // Called second time
        result = child[0].get() + child[1].get();
    }
};
```
**Spawn**

Future< TaskPolicy::execution_space, functor::value_type >
TaskPolicy::host_spawn( functor, options... );

Future< TaskPolicy::execution_space, functor::value_type >
TaskPolicy::task_spawn( functor, options... );

- **host_spawn**: called outside of a task, by host process
- **task_spawn**: called inside of a task; e.g., within GPU

**Spawn options...**

- Kokkos:: TaskSingle or TaskTeam
- Kokkos:: TaskHighPriority, TaskRegularPriority, TaskLowPriority
- Kokkos::Future<...>
  - Execute-after dependence on another task
  - Spawned by the same TaskPolicy instance
Respawn

```cpp
void TaskPolicy::respawn( this , options... );
```

- Called once within a running task on itself
- Respawn options...
  - Kokkos:: TaskHighPriority, TaskRegularPriority, TaskLowPriority
  - Kokkos::Future<...>
    - Execute-after dependence on another task
    - Spawned by the same TaskPolicy instance
- Respawn action
  - Upon returning `this` task is rescheduled
  - Changed priority, dependence, but NOT single/team
Single Thread vs. Thread Team Task

```cpp
void operator()( const member_type & member,
                value_type & result );
```

TaskPolicy<...>::member_type
used just like TeamPolicy<...>::member_type

- member.league_size() == 1 and member.league_rank() == 0
- member.team_size() and member.team_rank()
  - spawn( ... , Kokkos::TaskSingle ) => team_size() == 1
  - spawn( ... , Kokkos::TaskTeam ) => team_size() == architecture dependent
- parallel_op with TeamThreadRange and ThreadVectorRange
  - parallel_for done
  - parallel_reduce and parallel_scan implementation in progress
- Team scratch memory : TBD
Multiple Dependences: when\_all

\begin{verbatim}
Future< TaskPolicy::execution_space >
TaskPolicy::when_all( N, Future<...>[]);
\end{verbatim}

- Returned Future is complete when all input futures are complete
- No ‘get’ value
- Allows respawn to change the number of dependences
Algorithmic Concern: Finite Memory Pool

- Spawn, respawn, and when_all allocate from memory pool
  - End-to-end task DAG may be much larger than memory pool
  - Large end-to-end task DAG does execute all at once
  - Waiting and completed tasks unnecessarily consume memory
  - Tasks do not (most often) execute until `Kokkos::wait( policy )`

- Futures are reference counted, last reference deallocates

- Algorithm has control of TaskPolicy memory
  - Release no-longer used Future: `f = Future<...>()`
  - `host_spawn` a `driver` task that
    - Calls `task_spawn` for the rest of the task DAG
    - If iterative it can periodically `respawn` at low priority to let backlog of waiting tasks drain and be deallocated
    - Strategy used by sparse matrix factorization mini-app
Task DAG Functionality is In-Progress

- LDRD FY14-FY16, ending Sept 22, 2016
  - Two prototypes thrown away; third time seems to be the charm
  - Qthreads back-end: on again / off again as Kokkos evolved
  - Qthreads prototyped thread team tasks: still evolving
- Thread Team Task
  - Internal data parallelism not finished
- Scratch Memory – not started, requirements?
  - Thread team scratch memory – one task
  - Task DAG subgraph scratch memory
    - Leading task allocates from memory pool
    - Dependent tasks use scratch memory
    - Trailing task deallocates
Discussion Summary

- Tasks not required to be idempotent
- Respawn implies that this interface is not useful for driving a polling task (where you really want to wake up every N [time unit])
- Why not have the execution space instance own a queue of tasks?
  - Algorithm’s task DAG exists independent of execution space; have execution space without task DAG but not vice-versa
  - Task queue requires scratch memory that does not live beyond the task DAG algorithm
  - Can have multiple independent task queues for a given execution space that at exist at different times

- Cannot concurrently execute two different task queues on the same execution space

Task API

- Rename from “TaskPolicy” to “TaskScheduler” or “TaskQueue”; “TaskScheduler” preferred and aligns with other libraries
- Use operator() argument introspection instead of ‘typedef value_type’
- when_all: the array length typically goes after the array, not before
- Tags for operator(), as in other functor interfaces
- Future::clear() instead of having to assign to default constructed Future

- Want ability for application to set task team size

- Want ability to start task queue execution without waiting; blocking/nonblocking would be architecture dependent

- Want scratch memory for tasks
DOCUMENTATION, EXAMPLES, TUTORIALS
Discussion Summary

- Programming guide is out of date
- Tutorial examples don’t show full API / capabilities
- Straw poll, what is most useful: 6 update examples, 6 update programming guide, 1 update tutorial, 1 formal spec

Programming guide
- Pull in examples’ code
- Good to have it in plain English
PRIORITIES
Meeting Retrospective

- Align future reviews with other major event; e.g., Trilinos spring developers meeting
- Better advertising in advance of meeting
- Agenda/schedule
  - More precise and firm timing for schedule
  - Helpful to let schedule be flexible so discussion could expand as needed
- Repeat on ~ two year cycle
- Kokkos workshop with presentations, comparable to Trilinos User Group meeting