SANDIA REPORT

SAND2017-10464
Unlimited Release
Printed September, 2017

Kokkos' Task DAG Capabilities

H. Carter Edwards and Daniel A. Ibanez

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-mission laboratory managed and operated by
National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc.,
for the U.S. Department of Energys National Nuclear Security Administration under contract DE-NA0003525.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2017-10464
Unlimited Release
Printed September, 2017

Kokkos' Task DAG Capabilities

H. Carter Edwards, *Scalable Algorithms*
Daniel A. Ibanez, *Computational Multiphysics*
Sandia National Laboratories
P.O. Box 5800 / MS 1318
Albuquerque, NM 87185

Abstract

This report documents the ASC/ATDM Kokkos deliverable "Production Portable Dynamic Task DAG Capability." This capability enables applications to create and execute a **dynamic task DAG**; a collection of heterogeneous computational tasks with a directed acyclic graph (DAG) of "execute after" dependencies where tasks and their dependencies are dynamically created and destroyed as tasks execute. The Kokkos task scheduler executes the dynamic task DAG on the target execution resource; *e.g.* a multicore CPU, a manycore CPU such as Intel's Knights Landing (KNL), or an NVIDIA GPU. Several major technical challenges had to be addressed during development of Kokkos' Task DAG capability: (1) portability to a GPU with it's simplified hardware and *micro*-runtime, (2) thread-scalable memory allocation and deallocation from a bounded pool of memory, (3) thread-scalable scheduler for dynamic task DAG, (4) usability by applications.

Acknowledgment

This work was funded in part by the Sandia National Laboratories' Laboratory Directed Research and Development (LDRD) program in the Computer and Information Science investment area.

Contents

List of Figures

Chapter 1

Introduction

Kokkos [1] refers to a performance portable, shared memory parallel programming model and its C++ library implementation. Initially Kokkos supported data parallel execution where a function can be called in parallel over a simple one-dimensional $[0..N)$ range. To this we added hierarchical thread-team data parallel execution where a function can be called in parallel over an NxM *league of teams of threads* One-dimensional and thread-team data parallelism addressed a large fraction of parallel algorithms' needs. However, a small fraction of performance critical algorithms could not be effectively implemented with just data parallelism.

Two *directed acyclic graph of tasks* (Task DAG) capabilities was recently added to Kokkos. Research and prototyping for these capabilities was performed within a three year laboratory directed research and development (LDRD) effort [2]. This prototype was subsequently matured into a production-quality capability within an Advanced Simulation and Computing (ASC) / Advanced Technology Development and Mitigation (ATDM) project.

Kokkos' Task DAG capabilities are the first time generalized support for Task DAG parallelism has been available on a GPU. This achievement required conquering major research and development challenges during both the LDRD and ASC/ATDM efforts. These challenges are a consequence of both the "leanness" and quantity of GPU cores. For example, a function executing on a GPU core cannot be efficiently context-switched to execute a different function - which is a common capability on "heavy" CPU cores.

Two Kokkos Task DAG capabilities have been deployed and are documented in this report. (1) A dynamic dag of heterogeneous tasks which provides the maximum set of Task DAG features. (2) A static dag using a single work function which provides the minimal set of Task DAG features with significantly less performance and complexity overhead. We name these Kokkos capabilities the *task-dag* and *work-dag*.

This report documents Kokkos' Task DAG capabilities in two parts. Chapter 2 gives the programming model abstractions, research challenges, and algorithmic design for these capabilities. Chapter 3 provides the application programmer interface (API) specifications for the memory pool, dynamic heterogeneous task DAG, and static homogeneous work DAG.

Chapter 2

Concepts and Design

Kokkos' previous programming model abstractions were limited to performance portable data parallelism with multidimensional array data structures. We enhanced this programming model with abstractions for parallel directed acyclic graph of tasks (task-dag). Kokkos' task DAG abstractions had to deviate from traditional task DAG programming models in order to be performance portable across diverse manycore architectures, especially GPUs.

We first introduce abstractions and our terminology for Kokkos' task DAG capabilities. Using this terminology we describe the research challenges that had to be accommodated by Kokkos' task DAG abstractions and design. Finally we give the conceptual design for the three major software components of Kokkos' task DAG capabilities: heterogeneous and dynamic task-dag, homogeneous and static work-dag, and finite memory pool with variable size allocations.

2.1 Conceptual Components and Terminology

Execution and Memory Spaces: An *execution space* identifies where and by what mechanism Kokkos executes parallel computations. For example, executing a computation on a subset of CPU cores using OpenMP [3] or executing on a GPU using CUDA [4]. A *memory space* identifies where and by what mechanism Kokkos manages data for parallel computations. For example, traditional CPU "main" memory, CPU high bandwidth memory, GPU memory, or GPU unified virtual memory (UVM). We refer to the implementation of execution and memory spaces as Kokkos' *back-ends*.

C++ Closure: A *C++ closure* is an application-defined C++ class and `operator()` member function for that class. A closure may be automatically generated by the C++ compiler from an application-defined C++11 *lambda* expression. An application calls a Kokkos parallel execution dispatch function, such as `parallel_for`, with a closure that Kokkos will execute in parallel.

Thread Team: A closure may concurrently execute on a *team of hardware threads* within an execution space. Such concurrent execution often uses data parallel operations

(`parallel_for`, `parallel_reduce`, and `parallel_scan`) that execute across that thread team. Thread teams typically share critical hardware resources such as registers and L1 cache. For example, the four hyperthreads sharing a core in the Intel Xeon Phi architecture and the warp lanes of an NVIDIA GPU architecture define a tightly bound group of hardware threads. When these tightly bound hardware threads execute different functions computing on different data they will compete for shared resources, which typically results in degraded performance compared to one of the hardware threads executing the same tasks sequentially.

Task: A *task* is implemented by closure whose function is scheduled for subsequent execution.

Task Dependence: A task may have an execute-after *dependence* on another task. For example, task B may only begin executing after task A has completed executing. We denote the directional "B execute-after A" dependence as mbox$B \leftarrow A$ or equivalently $A \rightarrow B$. Note that we use the term *dependence* and its plural *dependences* as opposed to the term dependency and its plural dependencies.

Task DAG: A collection of tasks and task dependences define a graph where tasks are graph-vertices and execute-after dependences are directed graph-edges. The graph cannot contain cycles of dependences, otherwise tasks within cycles could never be executed. Thus the collection tasks and dependences define a directed acyclic graph (DAG) of tasks; *i.e.*, a *Task DAG*.

Task-dag: We use the term *task-dag* to refer to a dynamic and heterogeneous collection of tasks and dependences: (1) tasks and dependences may be dynamically created and destroyed and (2) each task may be implemented by a different C++ closure.

Work-dag: We use the term *work-dag* to refer to a static and homogeneous collection of tasks and dependences: (1) all tasks are implemented by a single C++ closure, (2) the closure accepts an integer work index calling argument to identify the "task" to be executed, (3) all task dependences are declared before any task executes, and (4) each task dependence is expressed by "j execute-after i" work indices, $j \leftarrow i$.

Task Scheduler: A *task scheduler* is responsible for managing a task-dag or work-dag and executing tasks as constrained by dependences. A task scheduler executes a task by calling it's function on a single hardware thread or team of hardware threads.

Spawn: *Spawning* is the process of creating a new task within a task scheduler for future execution.

Future: When a task is spawned a handle to the created task is generated. For conformance with the C++ standard's vernacular we refer to this handle as the task's *future*. However, a Kokkos future has *shared ownership* semantics versus the C++ standard future's *unique ownership*semantics; details are given in Section 3.2.1. A task's future is used to to express execute-after dependences and obtain return values from tasks. To resolve a design challenge we also define a *when-all* future that bundles multiple execute-after dependences; details are given in Section 2.2.2.

Complete or Respawn: When a task's function returns from execution the task is either (1) *complete* and any execute-after dependences referring to that task are satisfied or (2) the task is *respawned* to execute again and all execute-after dependences referring to that task are not satisfied. During its execution a task may request to be respawned upon returning from its function. Task respawning is a major paradigm deviation from traditional task programming models that addresses a significant performance portability research challenge described in Section 2.2.1.

Ready Task: A task is *ready* for execution by the task scheduler when (1) the task that has no execute-after dependences or (2) all of its execute-after dependences are complete.

Task Priority: A task scheduler chooses tasks to execute from among ready tasks. An application's algorithm may assign tasks relative *priorities* such that the task scheduler will choose a higher priority ready task over a lower priority ready task for execution.

Memory Pool: Dynamic task creation and destruction must allocate and deallocate memory for the task's closure. These allocations are made from a *memory pool* which manages dynamic allocations of small blocks of memory from within a large pre-allocated chunk of memory.

2.2 Research Challenges

2.2.1 Respawn, Because GPUs Cannot Wait

Traditional task programming capabilities (*e.g.*, C++11 [5], OpenMP [3], Silk and Silk Plus [6], Qthreads [7], TBB [8], X10 [9], Chapel [10], and Habanero [11]) allow an executing task to spawn another task and then *wait* for that spawned task to complete. Such a wait

operation defines an implicit dependence – the waiting task stops executing (blocks) and resumes executing after the spawned task completes. In order to guarantee that execution of a collection of tasks will make progress, a blocked task's function must relinquish or yield its execution resource (hardware thread) so that other tasks can execute on that resource.

For task A to yield an execution resource to task B the complete execution state of task A must be stored, the complete execution state of task B loaded (if previously stored), and then task B executes. When the task that A is waiting on completes then A's execution state must be reloaded and then A can resume executing. A task's execution state includes its registers, stack memory, and instruction pointer. Storing and loading this state introduces both runtime overhead and complexity for mechanisms required to determine and store/load this state.

Given a GPU's *lean* core and runtime it is impractical (likely impossible) to store/load a task's execution state. Even on a CPU the task state store/load overhead is undesirable (likely unacceptable) with respect to the "micro" tasks that Kokkos' task-dag and work-dag capability is targeting. Thus the traditional task programming model's spawn-and-wait paradigm was excluded from Kokkos.

Kokkos replaces the traditional *wait* pattern with the *respawn* pattern. In Kokkos' task programming model a task's function may spawn one or more other tasks, request that its own task be respawned with new task dependences, and then return to be called again. A task's function is called only after the task's previous task dependences are complete. As such the function can process inputs from those execute-after tasks and is then free to declare new task dependences as part of the respawn request.

The respawn request is processed by the task scheduler after the task's function returns. At this point the task no longer has an execution state – it is not consuming registers, stack memory, or instruction pointer. As such the need to store/load an execution state is entirely eliminated. The task scheduler need only insert the task in the appropriate queue, as it does when a task is initially spawned.

Without automatic store/load of registers, stack memory, and instruction pointer a respawning task must maintain its own state within the task's closure. This state may be as simple as an integer variable indicating whether this is the first or a subsequent call to the task's function. The content of the task's state is entirely within the task's purview.

2.2.2 Futures, Dynamic Dependencies, and When-all

When a dynamic task is spawned the spawning function returns a future referencing the spawned task. When another task B is spawned, or requests to be respawned, the spawn and respawn functions may be passed a future referencing task A to declare that task B executes-after task A completes. However, a task may have more than one execute-after dependence when spawned or respawned.

Respawning with multiple execute-after dependences poses a design challenge for efficiently managing an unpredictable and potentially changing number of dynamic dependences. Our design solution is to (1) allow each task to have one dependence and (2) provide a *when-all* future in addition to the task future. A when-all future references multiple tasks and is complete when all of the referenced tasks are complete. Thus when a task is spawned or respawned with an execute-after dependence on a when-all future it will only execute after all referenced tasks are complete.

2.2.3 Thread Teams for Performance

A thread team is typically defined by a set of hardware threads that share critical hardware resources such as registers and L1 cache. On contemporary NVIDIA GPU architectures threads within a *warp* even share an instruction pointer. If different tasks' functions are called by threads that share such resources they will compete for those resources and subsequently impede each other's execution progress. Net performance is often improved by giving each task exclusive access to shared resources while its function is executing. In this scenario either the function is called on a single thread and all other threads within the team are idle, or the function is called on all threads of the team and those threads use shared resources cooperatively.

Kokkos' data parallel patterns have the capability to execute a function on an NxM *league of teams of threads*, *a.k.a.* the *thread team execution policy*. When a function is called by team $\#i \in [0..N)$ the function is called on M (team size) concurrently executing threads. The function is expected to use this thread team through a a sequence of data parallel operations that insures the thread team's memory accesses and control flow are fully cooperative. Note that on an NVIDIA GPU architecture M is at least the number of threads in a warp, on an Intel Xeon Phi or similar architecture M is the number hyperthreads per core or tile, and on convential "heavy" CPU architectures M is often one.

Kokkos' thread team abstraction is also used for task execution. Each task executes on a team of M concurrently executing threads, or on a single thread within that team while all other threads of the team remain idle. In contrast to data parallel thread teams, task parallel thread teams are independent so the league size N is one and the league rank $\#i \in [0..1)$ is zero.

2.2.4 Heterogeneous Architecture

A task-dag scheduler maintains a dynamic DAG of closures where each closure is a "blob" of data and a function pointer. The function pointer references instructions that execute on a particular execution architecture; e.g., CPU or GPU. The closure and its function may be compiled to more than one architecture with different architecture-specific instructions. Thus a task's function may have more than one implementation and a task-dag scheduler must chose the correct function for execution on the specified architecture.

When compiled to more than one architecture each compiler has the option of generating a different layout for the members of the closure; *i.e.*, the explicitly written or implicitly generated C++ class. This permissible flexibility results in interoperability concerns when constructing and executing a task on different architectures: does the constructing-architecture of the closure layout the closure's members as expected by the executing-architecture? Since Kokkos' inception, the layout of explicitly written C++ classes has been identical between NVIDIA's GPU compiler (nvcc) and the CPU compiler with which it interoperates. Interoperability relies upon the C++ *standard layout type* property: (1) non-static data members have the same access control, (2) no virtual functions or virtual bases classes, (3) no non-static data members of reference type, (4) all non-static data members and base classes are themselves standard layout types, and (5) other subtle constraints with respect to multiple inheritance. Interoperability of implicitly generated C++ lambda is problematic because the generating compiler has complete freedom when declaring class members. For the current implementation a task that is constructed on the CPU architecture for execution on the GPU architecture is required to explicitly write the task as a standard layout C++ class.

When constructing a closure on a CPU for execution on a GPU the closure's data blob and function pointer must be accessible to both the CPU and GPU. This requirement could be met by constructing the closure on the CPU and copying it to the GPU, or placing the closure in GPU *unified virtual memory* (UVM). A challenge for the non-UVM approach is managing updates to the ready queues and dependence queues as the CPU constructs multiple tasks with dependences. This complexity is avoided by using UVM. However, the UVM strategy introduces the constraint that while the task scheduler is executing a GPU task-dag the CPU is not permitted to modify or query the GPU task-dag.

2.2.5 Finite Memory Constraint

A dynamic and heterogeneous task-dag creates and destroys tasks' closures of varying sizes as the task-dag executes. Efficient execution requires efficient memory allocation and deallocation for those closures. Furthermore, memory resources are often limited such as GPU memory or CPU high bandwidth memory (HBM). Finally, the GPU's lightweight runtime does not have the rich memory management facilities of a heavyweight CPU runtime.

To manage these time-performance, space-performance, and lightweight runtime challenges we developed a new thread scalable and (essentially) lock free memory pool. This memory pool uses efficient atomic operations to support allocation and deallocation of blocks of memory from a preallocated, large, contiguous chunk of memory. The memory space and size of this preallocated chunk of memory under the application's control so that the application may effectively manage finite memory resources.

2.3 Task-dag: Dynamic and Heterogeneous

2.3.1 Dynamic Task Life-Cycle

Within a dynamic and heterogeneous task-dag individual tasks are created, executed, and destroyed according to the life-cycle illustrated in Figure 2.1.

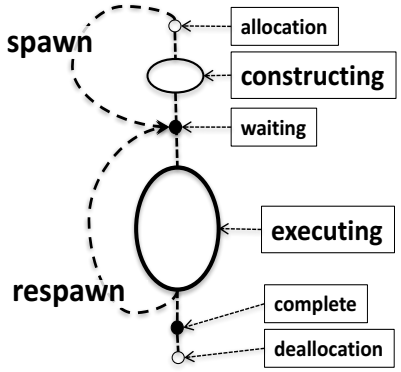

Figure 2.1: The life-cycle of a dynamic task consists of multiple states and state transitions between its allocation and deallocation. The spawn operation allocates memory for the task's closure, calls the closure's constructor, and enqueues the task in the task scheduler. When a task executes the closure's function is called. When that function returns the task is either complete or respawns back into the task scheduler's queue. When the closure of complete task is no longer needed the closure's allocated memory is deallocated.

The spawn operation (1) allocates memory for the task's closure from the memory pool, (2) calls the closure's constructor on that memory, (3) enqueues the task into the task scheduler, and (4) returns a future that references the spawned task. These steps are immediately executed by the thread that called the spawn operation; either on the CPU or on the GPU. The spawn operation fails if the memory pool cannot fulfill the memory allocation request and the task scheduler returns a null future.

Once enqueued the task is waiting to be executed. A task may be waiting for its execute-after dependences to be satisfied, or may be ready and simply waiting for the task-scheduler to select it for execution.

A task is executed when a thread or thread team call the closure's function. Execution may occur on the same thread A in which the task was spawned; however, the more threads available to the task-scheduler the more likely the task will execute on a different thread B. It is even possible that the task will execute on thread B before the spawn operation returns a future referencing that task on thread A.

While executing a task may request that itself be respawned and then re-enqueued in the task-scheduler. The respawn request may introduce new execute-after dependences – replacing the previous execute-after dependences that were guaranteed to be fulfilled when

the task started executing. When the task's function returns, the respawn operation re-enqueues the task and it is waiting again.

A task may respawn itself any number of times; for example it may respawn itself such that its function is called N times. The task itself is responsible for tracking respawn / re-execution. For example, a task that respawns and re-executes N times is responsible for incrementing a counter $i \in [0..N)$ so that whenever $i < N$ the task requests respawning. The type and meaning of this tracking data is at the discretion of the task, and may be maintained and updated as a member parameter of the task's closure.

A task that does not request to be respawned is complete and all execute-after dependences that reference the task are fulfilled. The complete task is retained as long as there is a future that references the task. This allows the task's result to be queried through the future. When the last future referencing the task is destroyed the task is deallocated and its memory returned to the memory pool.

2.3.2 Thread Teams

A task executes on either a single thread or team of threads, as illustrated in Figure 2.2. Tasks executing on a thread team will have internal data parallel operations that are spread across the thread team. When executing on a thread team the task's function must control whether a code block is executed just by one thread, by all threads, and when synchronization of the thread team is necessary.

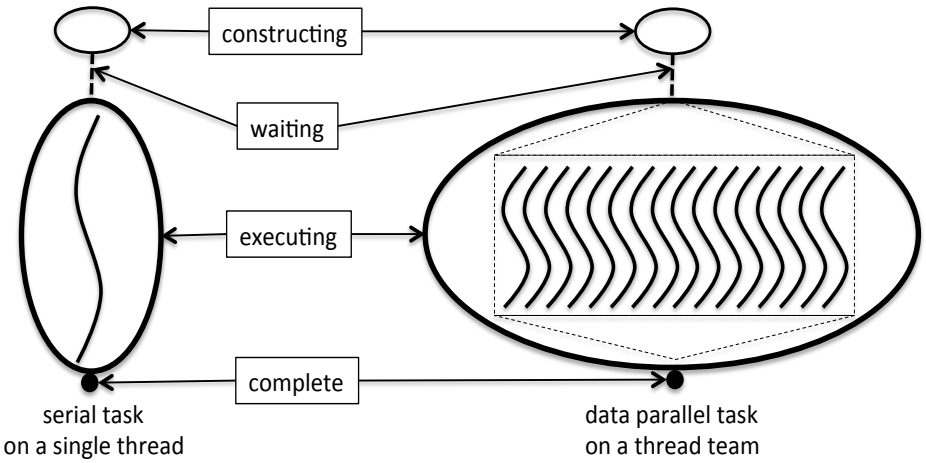

Figure 2.2: Tasks execute serial or data parallel. A task scheduler selects a task that is ready to execute from the waiting collection and executes the task on a single thread or thread team.

When a task executes on a single thread the task's function may spawn new tasks, request respawning, and modify parameters within its closure. When a task executes on a thread

team the task's function should use only one thread of that team to (1) spawn a particular new task or (2) request that the task be respawned.

2.3.3 Scheduler

When a dynamic task is spawned or respawned it is assigned a priority and possibly an execute-after dependence. The task-dag scheduler maintains queues of ready-to-execute tasks, tasks waiting on an incomplete task or when-all entity, and when-all entities waiting on incomplete tasks. When a dynamic task is spawned or respawn the scheduler inserts the task into the appropriate ready or waiting queue.

A queue is last-in-first-out (LIFO) implemented with a simple linked list. Each member of the linked list is a task or a when-all entity. Push and pop operations use atomic operations to add or remove members to the head of the linked list. Thus these operations are thread-safe and thread-scalable.

The scheduler's set of queues consists of one head of a linked list for each priority (high, regular, and low). Each task or when-all entity holds a head of a waiting linked list. The relationship between ready queues, waiting queues, and members (tasks and when-all entities) is illustrated in Figure 2.3.

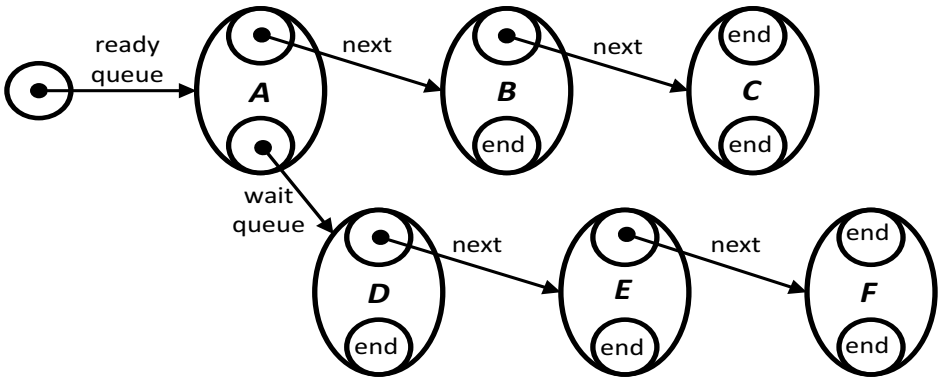

Figure 2.3: Each *ready queue* is defined by the head of a linked list of tasks. Each task or when-all entity is a member of exactly one linked list and holds the head of a *wait queue*. Since a task or when-all entity is a member of a linked list it also holds a *next* reference.

In Figure 2.3 task A, B, and C are members of ready queue. Task A is the head of this queue and references task B as the next member. In turn task B references task C, which is the end of the ready queue. Task D, E, and F are members of the waiting queue for task A. Note that a task or when-all entity is a member of at most one queue so the "next" reference is unambiguous.

When a task is spawned, respawned, or its execute-after dependence completes the scheduler pushes the task onto the appropriate queue. The spawn or respawn operation on task B interrogates whether it has a dependence on an incomplete task or when-all entity A. If so

then B is pushed onto A's waiting queue. Otherwise task B is pushed onto the ready queue of B's assigned priority.

When a when-all entity is created its referenced tasks are interrogated to find an incomplete task. If when-all entity B has an incomplete task A then B is pushed onto A's waiting queue. Otherwise B is complete and it is not pushed onto a queue.

When a task completes the scheduler processes its queue of waiting tasks and when-all entities. Similarly when a when-all entity is determined to be complete the scheduler processes its queue of waiting tasks. Each formerly waiting task is pushed to a ready queue. If a when-all entity B references an incomplete task A then B is pushed onto A's waiting queue. Otherwise the when-all entity B is determined to be complete and the scheduler processes its queue of waiting tasks.

The scheduler is implemented with a set of collaborative, concurrent workers executing on hardware threads. All workers are equal partners in the scheduler; *i.e.*, there is no "master" or "foreman" among the workers. Concurrent workers are organized into thread teams; on "heavy" CPUs the team is a single thread, on "light" CPUs such as Intel Xeon Phi a team consists of all hyperthreads on a core, and on GPUs the team consists of a GPU warp of threads. One worker from each team iterates an array of ready queues from high to low priority and attempts to pop a task. If successful the task is broadcast to the thread team and the thread team executes the task. If all ready queues are empty and there are no executing tasks, which could spawn new tasks, then entire task-dag is complete.

The scheduler is thread-safe and thread-scalable through the use of atomic operations at all points of inter-worker collaboration. Tasks and when-all entities are allocated and deallocated with a memory pool using atomic operations. Members are pushed and popped on queues with atomic operations.

2.4 Work-dag: Static and Homogeneous

Kokkos' task-dag provides applications with a maximum amount of flexibility, a dynamic number of arbitrary tasks and dependences. An application pays for this flexibility with the runtime overhead of managing an arbitrary set of functions, dynamic memory allocation / deallocation, and queues with dynamic size. If an application's task DAG has a single *work* function for all of its tasks and all execute-after dependences are expressed prior to executing the task DAG then the runtime overhead can be significantly reduced by using Kokkos' static and homogeneous work-dag.

The work-dag combines the simplicity of data parallel operation over a range of integers with the execute-after dependences of a DAG. Instead of a dynamic and heterogeneous collection of tasks, the work-dag has a single work closure that is called over a predetermined range of work indices; $F(i)$ where $i \in [0..N)$. In contrast to a `parallel_for` operation, a work-dag is also given a DAG of execute-after dependences defined with "work item j

executes after work item i" declarations. The work-dag scheduler guarantees that for every execute-after dependence the call $F(j)$ will begin only after the call $F(i)$ has returned.

Execution of a work-dag requires only a single work closure and does not require dynamic allocation / deallocation. Thus the static and homogeneous work-dag is significantly simpler with significantly lower overhead than the dynamic and heterogeneous task-dag.

2.4.1 Scheduler

The work-dag scheduler uses a set of pre-allocated and pre-initialized queue data structures. These data structures consist of the following.

- A compressed row storage (CRS) array representing the work item execute-after dependence graph.
- An array for the waiting queue of work items.
- An array and range for the ready queue of work items.

Ready Queue

A thread pops a work item i from a ready queue and calls the work function with that work index, $F(i)$. A work item j becomes ready to execute when all of its execute-after dependences are complete. When a work item becomes ready a thread pushes that work item onto a ready queue.

A ready queue is implemented with an array of work indices $Q(N)$ and range within that array $R = (b, e)$. The ready work indices reside in the queue as $Q(k) \forall k \in R$. A new work item j pushed into the array at the end of the range $Q(e) = j, e = e + 1$. A work item i popped from the array at the beginning of the range, $i = Q(b), b = b + 1$. Push and pop operations are implemented with atomic updates so the ready queue is lock-free.

The ready queue's array is sized to the maximum work count N so that (1) the queue cannot overflow even if all N work indices are ready to execute and (2) push and pop operations need not be concerned with managing the reuse of array locations (*e.g.*, a smaller circular queue) and thus have simple and time-efficient implementations.

Dependence Graph

The DAG of execute-after dependences (work item j executes after work item i) is stored in a CRS array. Given CRS row-offset and column-index arrays R and C, these dependences

are stored as

$$j = C\left(k\right) \forall k \in \left[R(i)..R(i+1)\right).$$

This storage scheme enables efficient updates to the waiting queue when the call to $F(i)$ completes.

It is often simpler for an algorithm to generate execute-after dependences as a set of "after" work item j paired with all of their "before" work items i. If this information is stored in a CRS array it is the transpose of the CRS array required by the scheduler,

$$i = C^T\left(k\right) \forall k \in \left[R^T(j)..R^T(j+1)\right).$$

Such an CRS array must be transposed before it is usable by the work-dag scheduler.

Waiting Queue

The waiting queue is implemented with an array of counts $W(N)$ denoting the number of incomplete execute-after dependences for each work item. When the call to $F(i)$ returns all counts for work items waiting on i are atomically decremented.

$$W(C(k)) = W(C(k)) - 1 \; ; \forall k \in \left[R(i)..R(i+1)\right)$$

When a $W(j)$ count becomes zero the work item is ready to execute and is pushed onto the ready queue.

2.4.2 Potential Improvements

Priority

As noted in the dynamic heterogeneous task-dag an algorithm may have different priorities for different tasks. However, the work-dag scheduler currently maintains a single first-in-first-out (FIFO) queue of ready work items. A potential improvement is to allow algorithms to supply relative priorities for work times (*e.g.*, high, regular, and low) and the scheduler maintain on FIFO ready queue per priority. Work item priorities would also be static so an upper bound for each priority queue N_p would be pre-computed. Thus each priority can be given a range $R_p = [i_p..j_p)$ within the ready array. Then given priority values $p \in [0..P)$ each range is initialized as

$$i_p = \sum_{k=0}^{p-1} N_k \; ; \; j_p = i_p + N_p$$

2.5 Memory Pool: Finite and Variable Size

The dynamic and heterogeneous task-dag uses a memory pool to allocate / deallocate tasks and when-all entities of varying sizes. When these allocations occur in GPU memory, or other limited memory spaces, the application typically needs to control its total memory consumption. To meet requirements of finite memory, thread scalability, and portability to CPU and GPU architectures we developed a new memory pool. This memory pool leveraged ideas from existing GPU compatible allocators such as ScatterAlloc [12, 13], Halloc [14], and Xmalloc [15].

Our design addresses the following constraints and performance goals.

- Use only an application specified finite amount of memory.
- Maximize thread scalability.
- Minimize time to find, acquire, and release memory.
- Minimize "lost" memory, the difference between the actual allocation size and requested allocation size.
- Minimize overhead memory, memory which is used to track allocations and not available to be allocated.

2.5.1 Superblock and Block Partitioning

Our memory pool manages allocatable blocks of memory within a contiguous span of memory; as illustrated in Figure 2.4. These blocks are organized in a hierarchy where the span memory is partitioned into superblocks of a uniform size and each superblock is partitioned into allocatable blocks. These blocks are also of a uniform size within a particular superblock, but each superblock may hold blocks of a different size.

Memory is allocated by blocks. When a memory allocation is requested for a particular size an unallocated block that is equal to or greater than the requested size is located, claimed as allocated, and returned to fulfill the request. When a prior allocation is deallocated the allocation marking for that block must be located and released.

The size of the blocks within a superblock is assigned to the superblock as allocation requests are processed. For simplicity and execution performance, block sizes and superblock size are chosen as a power of two. Thus each superblock is assigned to N, where blocks within the superblock are size 2^N. This insures that a superblock contains a power of two number of blocks; given block size 2^N and superblock size 2^M then there are $2^{(M-N)}$ blocks in a superblock. This also allows expensive integer division and modulo instructions to be replaced with inexpensive bit shift and mask instructions.

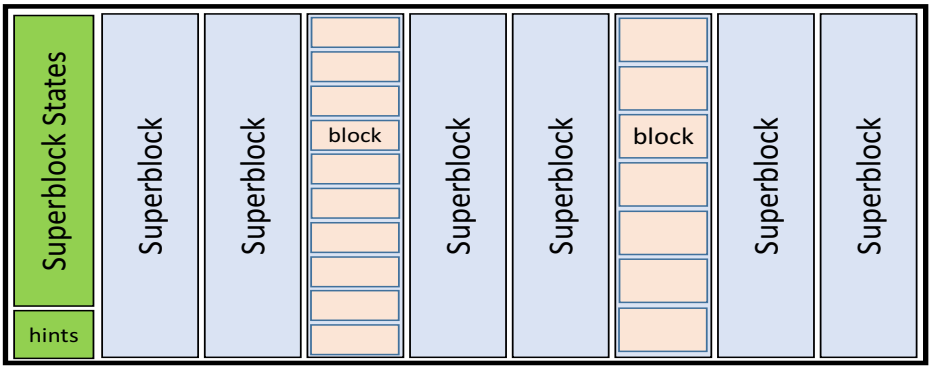

Figure 2.4: A memory pool's contiguous span of memory is partitioned into superblocks, which are partitioned into individually allocatable blocks.

2.5.2 Superblock State

The set of superblock states is maintained within a contiguous span of memory within a memory pool, as illustrated in Figure 2.4. A superblock's state is the size of blocks that it contains (2^N) and an array of flags indicating which of those blocks have been claimed for an allocation. The state changes in three ways: (1) an unused block is claimed (allocated), (2) a claimed block is released (deallocated), and (3) the block size 2^N is reassigned. A superblock is *full* if all of its blocks are claimed, *partially full* if some but not all of its blocks are claimed, and *empty* if none of its bocks are claimed. Only an empty superblock may be reassigned to a different block size.

A superblock contains 2^{M-N} blocks, where 2^N is the block size and where 2^M is the superblock size. The array of "claimed block" flags is implemented as an array of bits in order to minimize the memory consumed by these flags. Because a superblock may be reassigned to any block size the array must be sized to the smallest block size; *i.e.*, the largest number of blocks that the superblock could contain.

2.5.3 Block Allocation and Deallocation

Allocation

Memory allocation is requested for a block of at least $Size$ bytes. The preferred block allocation size is 2^N, where $2^{N-1} < Size \leq 2^N$. The *preferred superblock* is one that is assigned to N (blocks of size 2^N) and is partially full. If a preferred superblock is not available the next best option is to find an empty superblock and reassign it to N. If an empty superblock is not available then the last option is to find a partially full superblock assigned to larger block size. The last option wastes memory by claiming a block that is larger than necessary; however, wasting memory is preferred to failing the allocation request.

The memory pool allocation algorithm that satisfies these preferences is summarized in Figure 2.5. The algorithm is designed to be performant even with a large number of concurrent threads (thread scalable) and a small number of superblocks (potentially high contention). All superblock state updates are performed through atomic operations and are lock-free.

```
function ALLOCATE( Size bytes )
    N ← such that 2^(N−1) < Size ≤ 2^N
    while trying to allocate do
        if SB is undefined then SB ← SBHint(N)
        if success claiming block from SB then return block
        for all SB ← each superblock, begin at SBStart(N) do
            if SB assigned to N and not full then
                search succeeds, exit for loop
            else if SB empty superblock then
                if first then SBEmpty ← SB
            else if SB assigned to > N and not full then
                if first then
                SBLarger ← SB
            end if
        end for
        if search failed then
            SB ← undefined
            if found SBEmpty then
                if success reassigning SBEmpty to N then
                    SB ← SBEmpty
                end if
            else if found SBLarger then
                SB ← SBLarger
            end if
        end if
        if SB assigned to N and not full then
            SBHint(N) ← SB
        end if
    end whilereturn failed allocation
end function
```

Figure 2.5: The block allocation algorithm must find a superblock that is (1) assigned to a block size suitable for the allocation request and (2) has an unclaimed block. A superblock may be reassigned to the required block size in order to fulfill the request.

A memory pool may have a large number of superblocks to search for the preferred or necessary superblock. We attempt to minimize the number of superblocks searched during an allocation in two ways. First, we maintain a preferred superblock *hint* for each block size assignment N (recall Figure 2.4). Second, we designate a "starting superblock" within the set of superblocks for each block size.

The superblock hint is intended to reference a preferred superblock, assigned to N and partially full. A superblock hint becomes invalid whenever the superblock is full or the superblock is reassigned to a different block size. Thus (1) a hint cannot be trusted to be correct and (2) a hint should be updated when it is invalid.

To claim a block from a superblock the array of bits is searched for an unset bit, indicating an unclaimed block. This bit is set with an atomic operation. If the atomic operation succeeds the block is claimed and the allocation succeeds.

When a block cannot be claimed from the hinted superblock the superblocks are linearly searched. This starting point for this search is a function of the requested block size, SBStart(N) in Figure 2.5. These starting points are spread out among the set of superblocks so that searches for different sizes are likely to start at different superblocks.

Each iteration of the search queries for the three usable states of a superblock: preferred, empty, and larger than necessary. If a preferred superblock (assigned to N and partially full) is found then the search is successful and ends. If the superblock is empty and is the first empty superblock encountered then this superblock is remembered. If the superblock is larger than necessary, is partially full, and is the first such superblock encountered then this superblock is remembered. If the search iteration completes without finding a suitable superblock then the *current* search fails. Because allocations and deallocations may occur concurrently another thread *may* have deallocated a block that could satisfy the allocation request. Thus the algorithm may try again any number of times, as directed by the allocation request.

If the search succeeds in finding a preferred superblock then the hint is updated with an atomic operation to help accelerate subsequent allocation requests. If instead the search finds an empty superblock then an attempt is made to reassign, with an atomic operation, the empty superblock to the desired block size. If this reassignment succeeds the hint is updated.

Deallocation

Deallocation is a straightforward constant-time operation. First the superblock containing the to-be-deallocated memory is computed from the memory's location. Second the block within that superblock is computed from the memory's location. Finally the block is released by unsetting its corresponding bit with an atomic operation.

2.5.4 Performance Parameters

A memory pool's span of memory (Figure 2.5) is allocated from a memory space and the superblock states are initialized according to a set of construction parameters. These parameters are used to "tune" the memory pool's performance.

Minimum total allocation size S_{tot} is the minimum amount of memory within the span that is reserved for superblocks. The actual allocation will be rounded up so that each superblock is 2^M bytes. This is the total memory available for individual allocations from the memory pool.

Minimum block allocation size S_{min} defines the minimum size of an allocation request. The default for this parameter is (currently) 64 bytes. The allocation algorithm does not enforce this lower bound during allocation. This parameter establishes the size of the smallest block that will be allocated as 2^N such that $2^{N-1} < S_{min} \leq 2^N$. Each allocations smaller than 2^N will consume an entire minimum-size block and thus waste memory. This parameter also establishes the maximum number of blocks in a superblock 2^{M-N}, which is used to size the array of bit-flags in the superblock state.

Maximum block allocation size S_{max} defines the maximum size of an allocation request. The default for this parameter is (currently) 4096 bytes. This upper bound is enforced by the allocation algorithm. This parameter is used determine the superblock size, if a superblock size is not specified. A large upper bound allows larger allocations but reduces the number of superblocks and thus reducing the ability of the memory pool to efficiently support a wide range of allocation sizes. This parameter is also establishes number of superblock hints needed, $N_{max} - N_{min}$, and thus memory consumed by the hints values.

Minimum superblock size S_{sup} defines the minimum superblock size where the actual size is 2^M such that $2^{M-1} < S_{sup} \leq 2^M$. The default for this parameter is the maximum block allocation size.

These performance parameters are constrained as follows.

$$0 < S_{min} \leq S_{max} \leq S_{sup} \leq S_{tot}$$

The memory pool is designed to support variable block-size allocations. However, it can be tuned for single block-size allocations by (1) setting the minimum and maximum block size to the same value and (2) setting the total allocation size and superblock size to the same value. These settings will establish a single superblock state with a single array of bits for the block allocations. The superblock search will occur only if the one superblock is full and the search will terminate after a single iteration.

Chapter 3

Specification

The application programmer interface (API) specifications for Kokkos' dynamic heterogeneous task-dag, static homogeneous work-dag, and memory pool are given here. The on-line Kokkos documentation (github.com/kokkos/kokkos/wiki) should also be consulted as these APIs may evolve.

3.1 Finite and Variable Size Memory Pool

```
1  namespace Kokkos {
2  template < typename Space >
3  class MemoryPool {
4  public:
5    void * allocate( size_t alloc_size , int attempt_limit = 1 ) const ;
6    void deallocate( void * ptr , size_t alloc_size ) const ;
7    size_t allocate_block_size( size_t alloc_size ) const ;
8
9    MemoryPool( const typename Space::memory_space &
10             , size_t  min_total_alloc_size
11             , size_t  min_block_alloc_size = 0
12             , size_t  max_block_alloc_size = 0
13             , size_t  min_superblock_size  = 0 );
14
15   MemoryPool();
16   MemoryPool( MemoryPool && );
17   MemoryPool( const MemoryPool & );
18   MemoryPool & operator = ( MemoryPool && );
19   MemoryPool & operator = ( const MemoryPool & );
20
21   size_t capacity() const ;
22   size_t min_block_size() const ;
23   size_t max_block_size() const ;
24
25   using usage_statistics = /* ... */ ;
26
27   void get_usage_statistics( usage_statistics & ) const ;
28 };
29 }
```

Figure 3.1: The MemoryPool provides variable size allocations from a fixed size span of memory.

```
void * MemoryPool::allocate( size_t alloc_size
                           , int attempt_limit = 1 );
```

- Effects: Makes `attempt_limit` attempts to obtain a memory block that is at least `alloc_size` bytes, when `alloc_size` ¿ 0. If successful returns a pointer to that block of memory. Otherwise returns NULL.

```
size_t MemoryPool::allocate_block_size( size_t alloc_size )
```

- Effects: For `alloc_size` bytes return the anticipated memory block size that would be allocated.

```
void MemoryPool::deallocate( void * ptr , size_t alloc_size )
```

- Requires: `ptr` was obtained from a prior call `ptr = allocate( alloc_size );`. Effects: Releases the previously allocated memory block referenced by `ptr`.

```
MemoryPool::MemoryPool( const typename Space::memory_space &
                      , size_t  min_total_alloc_size
                      , size_t  min_block_alloc_size = 0
                      , size_t  max_block_alloc_size = 0
                      , size_t  min_superblock_size  = 0 );
```

- Requires: $0 <$ `min_block_alloc_size`
  `min_block_alloc_size` $\leq$ `max_block_alloc_size`
  `max_block_alloc_size` $\leq$ `min_superblock_size`
  `min_superblock_size` $\leq$ `min_total_alloc_size`
- Effects: Allocates a span of memory from the memory space to provide at least `min_total_alloc_size` bytes of allocatable memory. Initializes the superblock partitions and states as guided by the remaining input parameters.

```
MemoryPool::MemoryPool();
MemoryPool::MemoryPool( MemoryPool && );
MemoryPool::MemoryPool( const MemoryPool & );
MemoryPool & MemoryPool::operator = ( MemoryPool && );
MemoryPool & MemoryPool::operator = ( const MemoryPool & );
```

- Effects: The `MemoryPool` class follows Kokkos' shared ownership semantics for the copy constructors and assignment operators.

```
size_t MemoryPool::capacity() const ;
```

- Effects: Returns the amount of allocatable memory where `min_total_alloc_size` $\leq$ `capacity()`.

```
size_t MemoryPool::min_block_size() const ;
```

- Effects: Returns the minimum allocation block size where `min_block_alloc_size` $\leq$ `min_block_size()`.

```
size_t MemoryPool::max_block_size() const ;
```

- Effects: Returns the maximum allocation block size where `max_block_alloc_size` $\leq$ `max_block_size()`.

```
void MemoryPool::get_usage_statistics( usage_statistics & ) const ;
```

- Effects: Outputs usage statistics such as memory allocated and number of memory blocks allocated. See the Kokkos memory pool header file for current statistics output.

3.2 Dynamic and Heterogeneous Task-DAG

Kokkos' dynamic heterogeneous task-dag has two major classes: a `Future` which is a reference to a task and a `TaskScheduler` which manages the queue of waiting tasks and execution of ready tasks.

3.2.1 Kokkos::Future

```
1  namespace Kokkos {
2  template< typename Arg1 = void , typename Arg2 = void >
3  class Future {
4  public:
5    using execution_space = /* ... */ ;
6    using value_type      = /* ... */ ;
7
8    Future();                          /* null future */
9
10   template< typename A1 , typename A2 >
11   Future( const Future<A1,A2> & ); /* copy future */
12
13   template< typename A1 , typename A2 >
14   Future & operator = ( const Future<A1,A2> & ); /* assign future */
15
16   ~Future();                         /* destroy future */
17
18   bool is_null() const ;             /* is null future */
19   const value_type & get() const ; /* get value of completed task */
20 };
21 }
```

Figure 3.2: The Future provides a reference counted handle to a spawned task.

```
template< typename Arg1 = void , typename Arg2 = void > class Future ;
Future::execution_space
Future::value_type
```

A `Future` references a task that executes in the `Future::execution_space` and has a result of the `Future::value_type`. The execution space and value type are specified through the future's template parameters. If the value type parameter is omitted the `value_type` is `void` and the referenced task has no result value. If the execution space parameter is omitted the `execution_space` is Kokkos' default execution space.

```
Future::Future();
template< typename A1 , typename A2 >
Future::Future( const Future<A1,A2> & );
template< typename A1 , typename A2 >
Future & Future::operator = ( const Future<A1,A2> & );
Future::~Future();
```

The `Future` class follows Kokkos' shared ownership semantics for the copy constructors and assignment operators. A futures may be assigned from another future with a compatible execution space and values type. For example, a future with `value_type` of `void` may be assigned from a future with a non-`void` `value_type`.

```
bool Future::is_null() const ;
```

- Effects: Returns whether the future references a task.

```
const value_type & Future::get() const ;
```

- Requires: The future references a completed task.
- Effects: Returns the result of the referenced task.

3.2.2 Kokkos::TaskScheduler

```
namespace Kokkos {
template< typename ExecutionSpace >
class TaskScheduler {
public:
  using execution_space = ExecutionSpace ;
  using memory_space    = /* ... */ ;
  using memory_pool     = /* ... */ ;
  using member_type     = /* ... */ ;

  TaskScheduler();
  TaskScheduler( TaskScheduler && );
  TaskScheduler( const TaskScheduler & );
  TaskScheduler & operator = ( TaskScheduler && );
  TaskScheduler & operator = ( const TaskScheduler & );
  TaskScheduler( const memory_pool & );

  memory_pool const * memory() const ;

  template< typename A1 , typename A2 >
  static Future< execution_space >
  when_all( Future<A1,A2> const dependence[] , int number_dependence );

  template< typename DependenceGenerator >
  Future< execution_space >
  when_all( int number_dependence , DependenceGenerator && );

  template< typename FunctorType >
  static void respawn( FunctorType * self
                     , TaskScheduler const & scheduler
                     , TaskPriority  const & priority );

  template< typename FunctorType , typename A1 , typename A2 >
  static void respawn( FunctorType * self
                     , Future<A1,A2> const & dependence
                     , TaskPriority  const & priority );
};

enum class TaskPriority { High , Regular , Low };

template< typename Future_Or_Scheduler >
Impl::TaskPolicyData<...>
TaskTeam( Future_Or_Scheduler const & , TaskPriority = TaskPriority::Regular );

template< typename Future_Or_Scheduler >
Impl::TaskPolicyData<...>
TaskSingle( Future_Or_Scheduler const & , TaskPriority = TaskPriority::Regular );

template<..., typename FunctorType >
Future<...> host_spawn( Impl::TaskPolicyData<...> const & , FunctorType && );

template<..., typename FunctorType >
Future<...> task_spawn( Impl::TaskPolicyData<...> const & , FunctorType && );

template< typename Space >
void wait( TaskScheduler< Space > const & );
}
```

Figure 3.3: The TaskScheduler provides an interface for spawning tasks, respawning tasks, and executing those tasks according to their execute-after dependences.

```
TaskScheduler::execution_space
TaskScheduler::memory_space
TaskScheduler::memory_pool
TaskScheduler::member_type
```

The task scheduler allocates memory for tasks from a `memory_pool`, which obtains its large span of memory from a `memory_space`, and executes tasks in a `execution_space`. When tasks execute they are called with a `member_type`.

```
TaskScheduler::TaskScheduler();
TaskScheduler::TaskScheduler( TaskScheduler && );
TaskScheduler::TaskScheduler( const TaskScheduler & );
TaskScheduler & TaskScheduler::operator = ( TaskScheduler && );
TaskScheduler & TaskScheduler::operator = ( const TaskScheduler & );
```

- Effects: The `TaskScheduler` class follows Kokkos' shared ownership semantics for the copy constructors and assignment operators.

```
TaskScheduler::TaskScheduler( const memory_pool & );
```

- Effects: Constructs a task scheduler which will use the input memory pool to allocate tasks.

```
TaskScheduler::memory_pool const * TaskScheduler::memory() const ;
```

- Effects: Returns the memory pool object with which the task scheduler was constructed. Returns NULL if the memory pool object was default constructed.

```
template< typename Future_Or_Scheduler >
Impl::TaskPolicyData<...>
TaskTeam( Future_Or_Scheduler const &
        , TaskPriority = TaskPriority::Regular );

template< typename Future_Or_Scheduler >
Impl::TaskPolicyData<...>
TaskSingle( Future_Or_Scheduler const &
          , TaskPriority = TaskPriority::Regular );
```

- Requires: The first argument is a non-NULL `Future` or non-NULL `TaskScheduler`.

- Effects: Construct task policy data required to spawn a task with either `host_spawn` or `task_spawn` functions. The policy data returned by `TaskTeam` specifies that a spawned task will execute on a thread team. The policy data returned by `TaskSingle` specifies that a spawned task will execute on a single thread. The policy references a `TaskScheduler` that is either explicitly input or that spawned the task referenced by the input future.

```
template<..., typename FunctorType >
Future<...> host_spawn( Impl::TaskPolicyData<...> const & policy
                      , FunctorType && );

template<..., typename FunctorType >
Future<...> task_spawn( Impl::TaskPolicyData<...> const & policy
                      , FunctorType && );
```

- Requires: The policy argument is the output of either the `TaskTeam` or `TaskSingle` function. The `FunctorType` is copyable or moveable, and conforms to the interface and semantics given in Section 3.2.3. The `host_spawn` function is called in host code outside of of any Kokkos data parallel or task parallel execution. The `task_spawn` function is called within a task executing in the `execution_space`. Two separate task spawning functions are required to obtain a function pointer for the `execution_space`. In particular, the mechanisms for obtaining a GPU execution space function pointer from host code or GPU code are are different and incompatible.

- Effects: A task scheduler is obtained from the policy. This task schedulure allocates task memory from the memory pool, copies the input functor into that task, obtains a function pointer to execute that task in the `execution_space`, sets task policy parameters, enqueues the task, and returns a future that references the spawned task.

```
template< typename A1 , typename A2 >
static
Future< execution_space >
TaskScheduler::when_all( Future<A1,A2> const dependences[]
                       , int number_dependence );
```

- Requires: All futures in the array `dependences[]` are not null and reference tasks spawned from the same scheduler.

- Effects: Return a future that is complete when all of the futures in the `dependences` array are complete.

```
template< typename DependenceGenerator >
Future< execution_space >
TaskScheduler::when_all( int number_dependence
                       , DependenceGenerator && );
```

- Requires: `DependenceGenerator` is a closure, preferrably a C++ lambda expression, with the following interface.

  ```
  scheduler.when_all( N ,
    [&](int i)->Future<execution_space> { ... } );
  ```

  The closure is called with $i \in [0..N)$ and returns a `Future`. The returned futures must either be null or reference a task that was spawned by the scheduler.

- Effects: Return a future that is complete when all of the futures returne by the dependence generator closure are complete.

```
template< typename FunctorType >
static
void TaskScheduler::respawn( FunctorType * self
                           , TaskScheduler const & scheduler
                           , TaskPriority  const & priority );

template< typename FunctorType , typename A1 , typename A2 >
static
void TaskScheduler::respawn( FunctorType * self
                           , Future<A1,A2> const & dependence
                           , TaskPriority  const & priority );
```

- Requires: A `respawn` function is called at most once from within an executing task closure with the first argument as `this`.

- Effects: Requests that this task be respawned with a new priority and optionally a new dependence. When the task's closure returns the task will be rescheduled with the requested priority and dependence.

```
template< typename Space >
void wait( TaskScheduler< Space > const & );
```

- Requires: Called from host code outside of any Kokkos data parallel or task parallel execution.

- Effects: Blocks the calling host code until all executing and ready tasks in the scheduler have completed, including tasks that are spawned by running tasks.

3.2.3 Requirements for Task Functor

A task's `FunctorType` must provide an execution function conforming to one of the two following interfaces.

```
void FunctorType::operator()( TaskScheduler::member_type & );
void FunctorType::operator()( TaskScheduler::member_type &
                            , value_type & );
```

A functor with the first interface does not have a result and a referencing future has a `void Future::value_type`. A functor with the second interface has a result and the referencing future has `Future::value_type` matching the `value_type` argument.

Task State and Respawning

A task that respawns should maintain state data to differentiate between the first and subsequent calls to the task. This state data may reside in global memory, or may reside within the task's closure. A task's `operator()` function is non-const so that member variables of the closure may be modified.

Tasks executing on teams

The `TaskScheduler::member_type` has the same functionality and interface as Kokkos' data parallel `TeamPolicy::member_type`. Thus a task spawned with a `TaskTeam` policy has access to all of the same nested data parallelism capabilities. Likewise a task executing on a thread team must also exercise team-versus-single discipline.

- When thread team task executes only one thread from that team may call the `respawn` function.
- Spawning new tasks is typically done by only one thread.
- Any updates to the task's closure must be carefully coordinated among the thread team to avoid race conditions.

Construction

The closure for a task will be constructed prior to being input to a spawn function. This closure is copy or move constructed into memory allocated by the task scheduler. When the spawn function returns the input closure is no longer needed. If this input closure is a local temporary it will be automatically destroyed.

3.2.4 Task-dag Example: Naive Fibonacci

We use the naive task-based Fibonacci algorithm for an example use of the task-dag capability.

$$\text{Fibonacci: } F(n) = \begin{cases} n \leq 1 & : \quad n \\ n > 1 & : \quad F(n-1) + F(n-2) \end{cases}$$

In this naive implementation each $F(n)$ computation is performed by its own task and each recursion (when $n > 1$) defines a new dependence. The purpose of this example is not the efficient computation of the Fibonacci function, but to use a trivial computation to illustrate the construction and execution of tasks and task dependences. This example also serves as a "stress test" for evaluating performance overhead for task allocation, deallocation, and scheduling.

The naive Fibonacci implementation using Kokkos' heterogeneous dynamic task-dag capability is illustrated in Figure 3.4. The code in this illustration has been simplified from the corresponding unit test case in the Kokkos repository. Simplification-omissions include checking whether memory pool allocation failed due to insufficient memory and macros required for portable compilation to GPUs.

In this implementation `Fibonacci` functor has two states: (1) the initial call where the $F(n-1)$ and $F(n-2)$ recursive tasks have not been spawned and (2) the second call where the recursive tasks are complete. This state is determined by whether the future handles to the recursive tasks (`fib[2]`) are null. If in the initial call $n > 1$ then the $F(n-1)$ and $F(n-2)$ recursive tasks are spawned, a `when_all` task is created for the completion of the two tasks, and the $F(n)$ task is respawned with a dependence on the `when_all` task. The second call merely queries the result of the recursive tasks and sets the result of the $F(n)$ tasks accordingly.

The $F(n-2)$ recursion will spawn fewer tasks than the $F(n-1)$ recursion. Thus the $F(n-2)$ recursion could complete sooner and deallocate the associated tasks – if it has the opportunity to execute before the $F(n-1)$ recursion. It is due to this algorithmic observation that the $F(n-2)$ task is assigned a high priority and the $F(n-1)$ branch is assigned the default regular priority. This strategy reduces the "high water mark" for memory consumption by the naive Fibonacci implementation.

```
template< typename Space >
struct Fibonacci {
  using namespace Kokkos ;
  using value_type = long ;
  using future_t   = Future< value_type , Space > ;
  using sched_t    = TaskScheduler< Space > ;
  using member_t   = typename sched_t::member_type ;

  sched_t    sched ;
  future_t   fib[2] ;
  value_type n ;

  Fibonacci( sched_t const & arg_sched ,
           , value_type const & arg_n )
   : sched( arg_sched ), fib{}, n( arg_n ) {}

  void operator()( member_t & , value_type & result )
  {
    if ( n <= 1 ) { result = n ; }
    else if ( ! fib[0].is_null() ) {
      result = fib[0].get() + fib[1].get();
    }
    else {
      fib[0] = task_spawn( TaskSingle(sched,TaskPriority::High), Fibonacci(n-2) );
      fib[1] = task_spawn( TaskSingle(sched), Fibonacci(n-1) );
      respawn( this , when_all( fib , 2 ) , TaskPriority::High );
    }
  }

  static void test( int n )
  {
    using memory_t = typename sched_t::memory_pool ;
    memory_t mempool( 16000 , 64 , 1024 );
    sched_t scheduler( mempool );
    future_t f = host_spawn( TaskSingle(scheduler), Fibonacci(n) );
    wait( scheduler );
    std::cout << "Fibonacci(" << n << ") = " << f.get() << std::endl ;
  }
};
```

Figure 3.4: Example use of Kokkos Task-dag capability for implementation of naive task-based Fibonacci algorithm.

3.3 Static and Homogeneous Work DAG

3.3.1 Kokkos::WorkGraphPolicy

```
namespace Kokkos {

template< typename ... Properties >
class WorkGraphPolicy ;

template< typename IndexType , typename Space >
class Crs ;

template< typename ... Properties , typename FunctorType >
parallel_for( WorkGraphPolicy<Properties...> const &
            , FunctorType const & );

template< typename ... Traits >
class WorkGraphPolicy {
public:
  using index_type      = /* */ ;
  using execution_space = /* */ ;
  using graph_type      = Crs< /* */ > ;

  WorkGraphPolicy( graph_type const & dependences );
};

template< typename ... >
class Crs {
public:
  using index_type   = /* */ ;
  using size_type    = /* */ ;
  using row_map_type = View<size_type*,Space> ;
  using entries_type = View<index_type*,Space> ;

  size_type numRows() const ;
  row_map_type row_map ;
  entries_type entries ;
};

template< typename ... Properties >
void transpose_crs( Crs< Properties... > & out
                  , Crs< Properties... > const & in );
}
```

Figure 3.5: Specification for Kokkos TaskScheduler which provides an interface for executing a work function on a range of work indices according to predefined execute-after dependences.

For a `Crs` (compressed sparse row) object to represent a DAG it must satisfy the following requirements.

- Let:
  - `dag` be a `Crs` object,
  - $N =$ `dag.numRows()`,
  - $NZ =$ `dag.entries.size()`,
  - $R(i) =$ `dag.row_map(i)`, and
  - $C(k) =$ `dag.entries(k)`.
- Requires:
  - $R(0) = 0$ and $R(N) = NZ$.
  - $R(i) \leq R(i+1)$, $\forall i \in [0..N)$.
  - $C(k) < N$, $\forall k \in [0..NZ)$.
  - $j = C(k)$ for some $k \in [R(i)..R(i+1))$ denotes a directed edge $j \leftarrow i$.
  - There are no cycles in the directed edges.

```
WorkGraphPolicy::WorkGraphPolicy(
  WorkGraphPolicy::graph_type const & dependences );
```

- Requires: `dependences` is a `Crs` object representing a DAG. An edge in the DAG denotes an execute-after dependence; $j = C(k)$ for some $k \in [R(i)..R(i+1))$ denotes work index j must execute-after work index i.
- Effects: Allocates and initializes a queue required to execute the `dependences` DAG of execute-after dependences. Retains a shared-ownership copy of the `dependences` input.

```
template< typename ... Properties >
void transpose_crs( Crs< Properties... > & out
                  , Crs< Properties... > const & in );
```

- Requires: `in` representes a DAG.
- Effects: Allocates and populates the `out` DAG as the transpose of `in` DAG.
- Note: It is often more straightforward for an application to construct a `Crs` object that is the transpose of the dependences DAG required to construct a `WorkGraphPolicy`; $j = C(k)$ for some $k \in [R(i)..R(i+1))$ denotes a that work index i must execute-after work index j. The transpose of such a graph must input to construct a `WorkGraphPolicy`.

```
template< typename ... Properties , typename FunctorType >
parallel_for( WorkGraphPolicy<Properties...> const & work_dag
            , FunctorType const & closure );
```

- Requires: Has `FunctorType::operator()(int i) const;`.
- Effects: Calls `closure(j)` for each work index j in the `work_dag` and according to the `work_dag` execute-after dependences; $j = C(k)$ for some $k \in [R(i)..R(i+1))$ denotes work index j must execute-after work index i.

3.3.2 Work-dag Example: Naive Fibonacci

The naive Fibonacci algorithm implementation using Kokkos' homogeneous static work-dag capability is illustrated in Figure 3.6. The code in this illustration has been simplified from the corresponding unit test case in the Kokkos repository.

```
template< class ExecSpace >
struct TestWorkGraph {
  using MemorySpace = typename ExecSpace::memory_space;
  using Policy  = Kokkos::WorkGraphPolicy<int, ExecSpace>;
  using Graph   = typename Policy::graph_type;
  using RowMap  = typename Graph::row_map_type;
  using Entries = typename Graph::entries_type;
  using Values  = Kokkos::View<long*, MemorySpace>;

  Graph  m_graph;
  Graph  m_depend;
  Values m_values;

  void operator()(int i) const {
    const int k = m_depend.row_map(i);
    if ( k < m_depend.row_map(i+1) ) {
      m_values(i) += m_values( m_depend.entries(k) ) +
                     m_values( m_depend.entries(k+1) );
    }
  }
  void run_test( long n ) {
    Kokkos::parallel_for(Policy(m_graph), *this);
    std::cout << "Fibonacci (" << n << ") = " << m_values(0) << std::endl;
  }
  TestWorkGraph( long arg_n ) {
    form_graph(arg_n);
    transpose_crs(m_depend, m_graph);
    run_test(arg_n);
  }
  struct HostEntry { long input; int parent; };
  void form_graph( long n ) {
    std::vector<HostEntry> hg;
    hg.push_back({ n , -1 });
    for (int i = 0; i < int(hg.size()); ++i) {
      auto e = hg.at(std::size_t(i));
      if (e.input < 2) continue;
      hg.push_back({ e.input - 1, i });
      hg.push_back({ e.input - 2, i });
    }
    m_graph.row_map = RowMap("row_map", hg.size() + 1);
    m_graph.entries = Entries("entries", hg.size() - 1);
    m_values = Values("values", hg.size());
    m_graph.row_map(0) = 0;
    for (int i = 0; i < int(hg.size()); ++i) {
      auto& e = hg.at(std::size_t(i));
      m_graph.row_map(i + 1) = i;
      if (e.input < 2) {
        m_values(i) = e.input;
      }
      if (e.parent == -1) continue;
      m_graph.entries(i - 1) = e.parent;
    }
  }
};
```

Figure 3.6: Example use of Kokkos Work-dag capability for implementation of naive task-based Fibonacci algorithm.

The `parallel_for` with the `WorkGraphPolicy` calls the functor's `operator()(i)` to compute $F(n)$. Before this call the two predecessor work items are guaranteed to have been called and the input values $F(n-1)$ and $F(n-2)$ have been computed. These predecessor work items (if they exist) are identified through the `m_depend` dependence graph in this example. The `operator()` work function queries `m_depend` to determine if there are predecessor tasks; *i.e.*, if the computation is for $n > 1$. If so then the the values from these tasks are queried, summed, and output.

In this example, the predecessor graph `m_depend` is the transpose of the `m_graph` execute-after graph used to construct the work-dag policy. Thus both forms of the dependence graph are required. For this naive Fibonacci algorithm we form `m_graph` and generate `m_depend` as its transpose because it is simpler than the reverse.

Each execute-after entry in `m_graph` is unique because, for this naive Fibonacci algorithm, each work item i computes one of the two inputs for exactly one work item j. The `form_graph` function pushes each of these j executes-after i dependences and the corresponding n_i predecessor values (if $n_i > 1$) into the temporary vector `hg`.

```
hg[i]   = { n_j - 1 , j }
hg[i+1] = { n_j - 2 , j }
```

The temporary `hg` vector is then converted to `m_graph` where the j executes-after i (i provides-input-to j) dependences are represented in compressed-row-storage object.

```
j = m_graph.entries[ m_graph.row_map[i] ]
```

Note (again) that this `form_graph` construction algorithm is based upon each work index i being a predecessor to a single work index j.

References

[1] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J. of Parallel and Distr. Comp.*, vol. 74, pp. 3202–3216, 2014. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2014.07.003

[2] H. C. Edwards, S. L. Olivier, G. E. Mackey, K. Kim, M. Wolf, G. W. Stelle, J. W. Berry, and S. Rajamanicham, "Hierarchical task-data parallelism using kokkos and qthreads," Sandia National Laboratories, Tech. Rep. SAND2016-9631, September 2016.

[3] "The OpenMP API Specification for Parallel Programming," openmp.org/, Jun. 2013.

[4] "CUDA home page," docs.nvidia.com/cuda/, Sep. 2017.

[5] "C++ Reference," cppreference.com, Sep. 2017.

[6] "Intel Cilk Plus," cilkplus.org, Sep. 2017.

[7] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *Proceedings of the 22nd International Symposium on Parallel and Distributed Processing*, ser. IPDPS/MTAAP, April 2008, pp. 1–8.

[8] J. Reinders, *Intel Threading Building Blocks - Outfitting C++ for Multi-core Processor Parallelism*. Sebastopol, CA: O'Reilly, 2007.

[9] "X10: Performance and Productivity at Scale," x10-lang.org, Sep. 2017.

[10] "The Chapel Parallel Programming Language," chapel.cray.com, Sep. 2017.

[11] "Habanero Extreme Scale Software Research Project," wiki.rice.edu/confluence/display/HABANERO/Habanero+Extreme+Scale+Software+Research+Pro Sep. 2017.

[12] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg, "ScatterAlloc: Massively parallel dynamic memory allocation for the gpu," in *Proceedings of 2012 Innovative Parallel Computing*, 2012.

[13] "ScatterAlloc: Massively parallel dynamic memory allocation for the gpu." [Online]. Available: https://github.com/ComputationalRadiationPhysics/scatteralloc

[14] "halloc: A fast and highly scalable gpu dynamic memory allocator." [Online]. Available: https://github.com/canonizer/halloc

[15] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W. mei W. Hwu, "Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines," in *CIT*, 2010.

DISTRIBUTION:

1	MS 0899	Technical Library, 9536 (electronic copy)
1	MS 0359	D. Chavez, LDRD Office, 1911

Sandia National Laboratories