Kokkos Tutorial

Christian R. Trott [1], Dan Ibanez [1], David S. Hollman [1], Dan Sunderland [1], Duane Labreche [1], Nathan Ellingwood [1], Steve Bova [1], Graham Lopez [2], Galen Shipman [3], and Geoffrey Womeldorff [3]

[1]Sandia National Laboratories, [2]Oak Ridge National Laboratory

[3]Los Alamos National Laboratory

Lawrence Berkeley National Laboratories March 26-29, 2019

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 Energy's National Nuclear Security Administration under contract DE-NA0003525.

SAND2019-xxxx

Task parallelism

Fine-grained dependent execution.

Learning objectives:

- Basic interface for fine-grained tasking in Kokkos
- How to express dynamic dependency structures in Kokkos tasking
- When to use Kokkos tasking

Recall that **data parallel** code is composed of a pattern, a policy, and a functor

```
Kokkos::parallel_for(
  Kokkos::RangePolicy<>(exec_space, 0, N),
  SomeFunctor()
);
```

Task parallel code similarly has a pattern, a policy, and a functor

```
Kokkos::task_spawn(
  Kokkos::TaskSingle(scheduler, TaskPriority::High),
  SomeFunctor()
);
```

```
struct MyTask {
  using value_type = double;
  template <class TeamMember>
  KOKKOS_INLINE_FUNCTION
  void operator()(TeamMember& member, double& result);
};
```

- Tell Kokkos what the value type of your task's output is.
- Take a team member argument, analogous to the team member passed in by `Kokkos::TeamPolicy` in hierarchical parallelism
- The output is expressed by assigning to a parameter, similar to with `Kokkos::parallel_reduce`

What policies does Kokkos tasking provide?

- `Kokkos::TaskSingle()`
  - Run the task with a single worker thread
- `Kokkos::TaskTeam()`
  - Run the task with all of the threads in a team
  - Think of it like being inside of a `parallel_for` with a `TeamPolicy`
- Both policies take a scheduler, an optional predecessor, and an optional priority (more on schedulers and predecessors later)

What patterns does Kokkos tasking provide?

- `Kokkos::task_spawn()`
  - `Kokkos::host_spawn()` (same thing, but from host memory space)
  - Soon, we'll have just `scheduler.spawn()`
- `Kokkos::respawn()`
  - Argument order is backwards; policy comes second!
  - Soon, we'll have just `scheduler.respawn()`
- `task_spawn()` and `host_spawn()` return a `Kokkos::Future` representing the completion of the task (see next slide), which can be used as a predecessor to another operation.

```
struct MyTask {
  using value_type = double;
  Kokkos::Future<double, Kokkos::DefaultExecutionSpace> dep;
  int depth;
  KOKKOS_INLINE_FUNCTION MyTask(int d) : depth(d) { }
  template <class TeamMember>
  KOKKOS_INLINE_FUNCTION
  void operator()(TeamMember& member, double& result) {
    if(depth == 1) result = 3.14;
    else if(dep.is_null()) {
      dep =
        Kokkos::task_spawn(
          Kokkos::TaskSingle(member.scheduler()),
          MyTask(depth-1)
        );
      Kokkos::respawn(*this, dep);
    }
    else {
      result = depth * dep.get();
    }
  }
};
```

```
template <class Scheduler>
struct MyTask {
  using value_type = double;
  Kokkos::BasicFuture<double, Scheduler> dep;
  int depth;
  KOKKOS_INLINE_FUNCTION MyTask(int d) : depth(d) { }
  template <class TeamMember>
  KOKKOS_INLINE_FUNCTION
  void operator()(TeamMember& member, double& result);
};
```

Available Schedulers:

- `TaskScheduler<ExecSpace>`
- `TaskSchedulerMultiple<ExecSpace>`
- `ChaseLevTaskScheduler<ExecSpace>`

```
using execution_space = Kokkos::DefaultExecutionSpace;
using scheduler_type = Kokkos::TaskScheduler<execution_space>;
using memory_space = scheduler_type::memory_space;
using memory_pool_type = scheduler_type::memory_pool;
size_t memory_pool_size = 1 << 22;

auto scheduler =
  scheduler_type(memory_pool_type(memory_pool_size));

Kokkos::BasicFuture<double, scheduler_type> result =
  Kokkos::host_spawn(
    Kokkos::TaskSingle(scheduler),
    MyTask<scheduler_type>(10)
  );
Kokkos::wait(scheduler);
printf("Result␣is␣%f", result.get());
```

- Tasks always run to completion
- There is no way to wait or block inside of a task
  - `future.get()` does not block!
- Tasks that do not `respawn` themselves are complete
  - The value in the `result` parameter is made available through `future.get()` to any dependent tasks.
- The second argument to `respawn` can only be either a predecessor (future) or a scheduler, not a proper execution policy
  - We are fixing this to provide a more consistent overload in the next release.
- Tasks can only have one predecessor (at a time)
  - Use `scheduler.when_all()` to aggregate predecessors (see next slide)

```
using void_future =
  Kokkos::BasicFuture<void, scheduler_type>;
auto f1 =
  Kokkos::task_spawn(Kokkos::TaskSingle(scheduler), X{});
auto f2 =
  Kokkos::task_spawn(Kokkos::TaskSingle(scheduler), Y{});
void_future f_array[] = { f1, f2 };
void_future f_12 = scheduler.when_all(f_array, 2);
auto f3 =
  Kokkos::task_spawn(
    Kokkos::TaskSingle(scheduler, f_12), FuncXY{}
  );
```

- To create an aggregate `Future`, use `scheduler.when_all()`
- `scheduler.when_all()` always returns a `void` future.
- (Also, any future is implicitly convertible to a `void` future of the same `Scheduler` type)

Formula

$$F_N = F_{N-1} + F_{N-2}$$
$$F_0 = 0$$
$$F_1 = 1$$

Serial algorithm

```
int fib(int n) {
  if(n < 2) return n;
  else {
    return fib(n-1) + fib(n-2);
  }
}
```

Details:

- Location: `Intro-Full/Exercises/08`
- Implement the `FibonacciTask` task functor recursively
- Spawn the root task from the host and wait for the scheduler to make it ready

Hints:

- Do the F_{N-1} and F_{N-2} subproblems in separate tasks
- Use a `scheduler.when_all()` to wait on the subproblems