The Kokkos Lectures

Kokkos Execution Space Instances

May 5, 2022

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.
SAND2020-8508 PE

This is NOT a general tutorial!

Today: three topics with exercises.

- Three Short Lectures
  - Execution Space Instances
  - PyKokkos
  - New KokkosKernels Capabilities
- Hands-On Exercise
  - Split Into Breakout Rooms
  - One per topic: Choose which one you want to attend

Tuesday 2022-05-10: Community BOF Days with presentations on more new capabilities.

https://www.exascaleproject.org/event/ecp-community-bof-days-2022

Online Resources:

- https://github.com/kokkos:
  - Primary Kokkos GitHub Organization
- https://github.com/kokkos/kokkos-tutorials/wiki/Kokkos-Lecture-Series:
  - Slides, recording and Q&A for the Lectures
- https://github.com/kokkos/kokkos/wiki:
  - Wiki including API reference
- https://kokkosteam.slack.com:
  - Slack channel for Kokkos.
  - Please join: fastest way to get your questions answered.
  - Can whitelist domains, or invite individual people.

Instructions for joining the exercises can be found here:
`https://github.com/kokkos/kokkos-tutorials/issues/60`

Issue: ECP Annualmeeting 2022 - Kokkos Tutorial AWS Instances

Claim an instance in the Google spreadsheet.

SSH to the instance.

The slides are available at:
`https://github.com/kokkos/kokkos-tutorials`
under `Other/ECP-Annualmeeting/2022`

Asynchronicity and ExecutionSpace Instances

The (non-)blocking behavior of Kokkos operations.

Learning objectives:

- What are blocking and non-blocking operations in Kokkos.
- What kind of work can overlap.
- How to wait for completion.
- How to run kernels simultaneously on a GPU.

Most operations in Kokkos are non-blocking

- The caller returns before the operation is finished
- The caller can do other things, while operations are executing

Most operations in Kokkos are non-blocking

- The caller returns before the operation is finished
- The caller can do other things, while operations are executing

So what is the ordering behavior?

- Execution Spaces have an ordered execution queue
- The queue is first-in/first-out (FIFO)

Most operations in Kokkos are non-blocking

- The caller returns before the operation is finished
- The caller can do other things, while operations are executing

So what is the ordering behavior?

- Execution Spaces have an ordered execution queue
- The queue is first-in/first-out (FIFO)

Important Point

Execution Spaces execute operations in dispatch order.

Execution Space Instances

- Each unique **Instance** of an execution space has its own queue
- Execution Policies can take an instance as the first argument
- deep_copy can take an instance as a first argument
- For every **Execution Space Type** there is a **default instance**
  - It is a singleton
  - The **default instance** is returned by the default constructor
  - Used if no specific instance is provided

Execution Space Instances

- Each unique **Instance** of an execution space has its own queue
- Execution Policies can take an instance as the first argument
- deep_copy can take an instance as a first argument
- For every **Execution Space Type** there is a **default instance**
  - It is a singleton
  - The **default instance** is returned by the default constructor
  - Used if no specific instance is provided

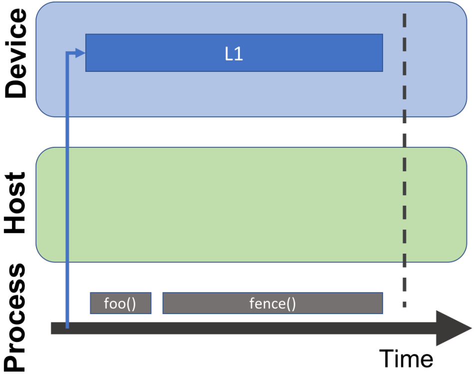
```
// This is equivalent:
RangePolicy<ExecSpace>
  policy_1(0, N);
RangePolicy<ExecSpace>
  policy_2(ExecSpace(), 0, N);
```

We use the following conventions in subsequent slides

```
// Execution Space types
using device = Kokkos::DefaultExecutionSpace;
using host = Kokkos::DefaultHostExecutionSpace;

// Execution Space instances
device dev1(..), dev2(..)
device host1(..), host2(..);

// Execution Policies
RangePolicy<device> policy_d(0,N), policy_device(0,N);
RangePolicy<host> policy_h(0,K), policy_host(0,K);
RangePolicy<device> policy_d1(dev1, 0,N), policy_d2(dev2, 0,N);
RangePolicy<host> policy_h1(host1, 0,N), policy_h2(host2, 0,N);

// Functors/Lambda for parallel_for
auto L1 = KOKKOS_LAMBDA(int i) {...};
auto L2 = ...; auto L3 = ...; auto L4 = ...; auto L5 = ...;
// Functors/Lambda for parallel_reduce
auto R1 = KOKKOS_LAMBDA(int i, double& lsum) {...};
```

Most Kokkos Operations are Asynchronous

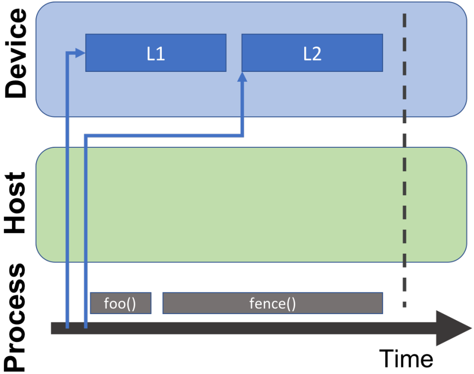
- Best to assume all of them are asynchronous
- They overlap with work in the process thread
- Use `Kokkos::fence()` to wait for completion




```
RangePolicy<>
  policy_device(0,N)
FunctorL1 L1(...);

parallel_for("L1",
  policy_device, L1);
foo();
fence();
```

Execution Spaces execute in dispatch order

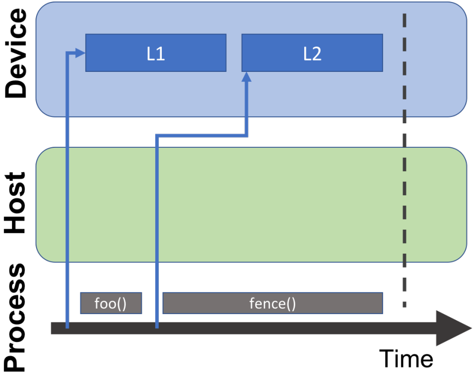
- Dispatches to the same space instance will never overlap
- Executed in order FIFO
- Use `Kokkos::fence()` to wait for completion




```
RangePolicy<>
  policy_device(0,N)
FunctorL1 L1(...);
FunctorL2 L2(...);

parallel_for("L1",
  policy_device, L1);
parallel_for("L2",
  policy_device, L2);
foo();
fence();
```

Execution Spaces execute in dispatch order

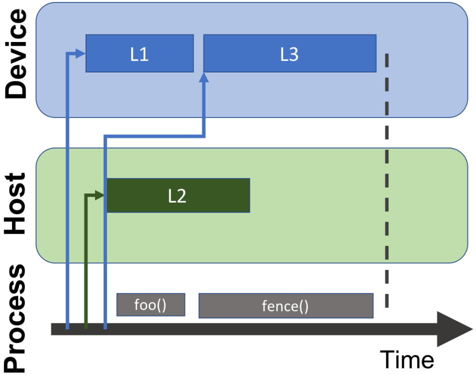
- Dispatches to the same space instance will never overlap
- Executed in order FIFO
- Use `Kokkos::fence()` to wait for completion




```
RangePolicy<>
  policy_device(0,N)
FunctorL1 L1(...);
FunctorL2 L2(...);

parallel_for("L1",
  policy_device, L1);
foo();
parallel_for("L2",
  policy_device, L2);
fence();
```

ExecutionSpaces are Independent

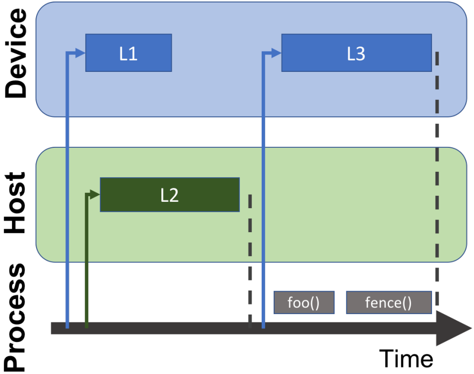
- Dispatches into different ExecutionSpaces may overlap.
- Overlap with process thread functions and each other
- Use `Kokkos::fence()` to wait for completion of all




```
RangePolicy<>
  policy_d(0,N)
RangePolicy<Host>
  policy_host(0,N)
FunctorL1 L1(...);
FunctorL2 L2(...);
FunctorL3 L3(...);
parallel_for("L1",
  policy_d, L1);
parallel_for("L2",
  policy_host, L2);
parallel_for("L3",
  policy_d, L3);
foo();
fence();
```

Reality: Some Host Backends Block

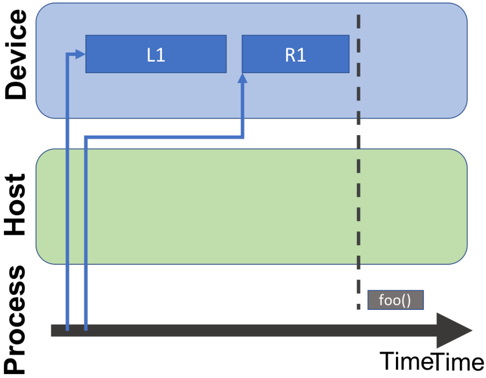
- Most host backends are blocking dispatches (except HPX)
- They never overlap with process thread functions
- But: **Do NOT rely on blocking behavior!!**




```
RangePolicy<>
  policy_d(0,N)
RangePolicy<Host>
  policy_host(0,N)
FunctorL1 L1(...);
FunctorL2 L2(...);
FunctorL3 L3(...);
parallel_for("L1",
  policy_d, L1);
parallel_for("L2",
  policy_host, L2);
parallel_for("L3",
  policy_d, L3);
foo();
fence();
```

Reductions to Scalars are Blocking

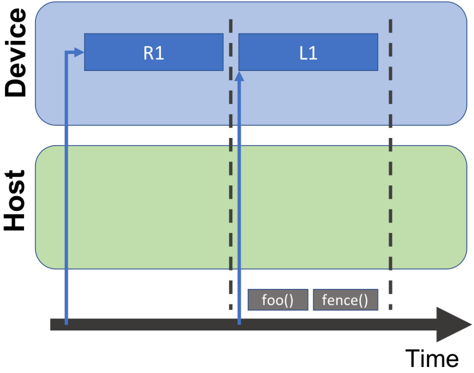
- The call only returns after result is available.
- FIFO implies, every other kernel in the same ExecutionSpace instance will be done too




```
RangePolicy<>
  policy_d(0,N)
FunctorL1 L1(...);
FunctorR1 R1(...);

double result;
parallel_for("L1",
  policy_d, L1);
parallel_reduce("R1",
  policy_d, R1, result);
foo();
```

Reductions to Scalars are Blocking

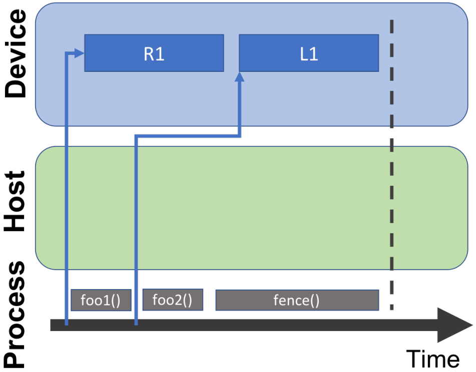
- The call only returns after result is available.
- For subsequent dispatches previous rules apply




```
RangePolicy<>
  policy_d(0,N)
FunctorL1 L1(...);
FunctorR1 R1(...);

double result;
parallel_reduce("R1",
  policy_d, R1, result);
parallel_for("L1",
  policy_d, L1);
foo();
fence();
```

Reductions to Views are Non-blocking

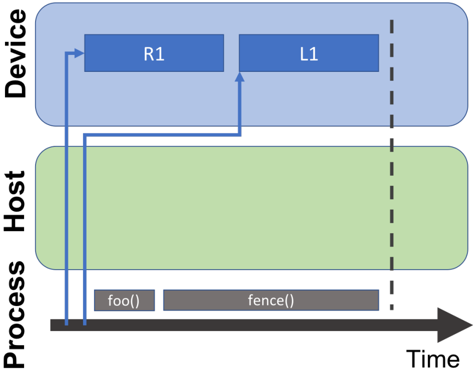
- Behave like a `parallel_for`
- Results are only available after a `Kokkos::fence()`
- Even true for unmanaged `Views` of host variables!




```
...
double result;
View<double, HostSpace>
  v_result(&result);
parallel_reduce("R1",
  policy_d,R1,v_result);
foo1();
parallel_for("L1",
  policy_d, L1);
foo2();
fence();
```

Simple Parallel Loop

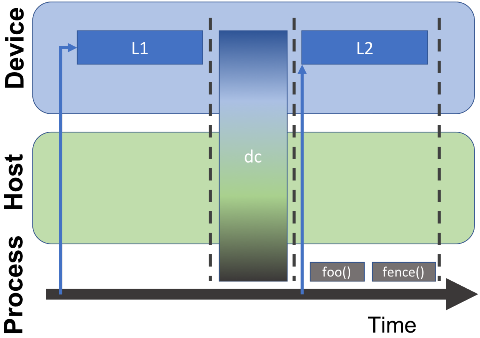
- Asynchronous
- Overlaps with host functions
- Use `Kokkos::fence()` to wait for completion




```
RangePolicy<>
  policy_device(0,N)
FunctorL1 L1(...);
FunctorL2 L2(...);

parallel_for("L1",
  policy_device, L1);
parallel_for("L2",
  policy_device, L2);
foo();
fence();
```

2-Argument deep_copy is fully blocking

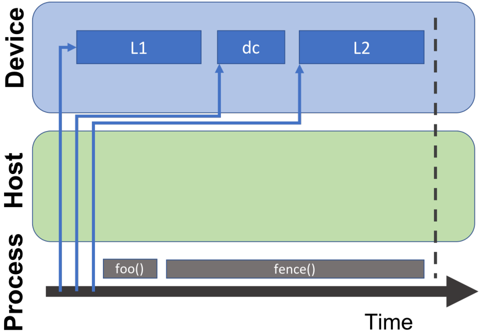
- Implies a full `fence` before the copy
- Copy is done by the time call returns.
- Even if it is a no-op due to `src == dst`!




```
parallel_for("L1",
  policy_device, L1);
deep_copy(dest,src);
parallel_for("L2",
  policy_device, L2);
foo();
fence();
```

deep_copy with space argument are non-blocking

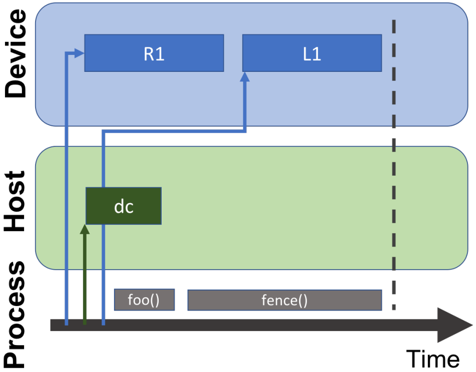
- Execute in dispatch order in the queue of the space
- Overlap with host process functions
- Use `Kokkos::fence()` to wait for completion




```
parallel_for("L1",
  policy_device, L1);
deep_copy(device,
  dest,src);
parallel_for("L2",
  policy_device, L2);
foo();
fence();
```

deep_copy with space argument are non-blocking

- Execute in dispatch order in the queue of the space
- Overlap with other execution spaces
- Use `Kokkos::fence()` to wait for completion




```
parallel_for("L1",
  policy_device, L1);
deep_copy(host(),
  dest,src);
parallel_for("L2",
  policy_device, L2);
foo();
fence();
```

So what about CUDA/HIP streams and SYCL queues?
Up to now we only used default execution space instances, but what if you want to have concurrent kernels on the GPU?

So what about CUDA/HIP streams and SYCL queues?

Up to now we only used default execution space instances, but what if you want to have concurrent kernels on the GPU?

Execution Space Instances

Execution Space instances behave largely like CUDA/HIP streams

So what about CUDA/HIP streams and SYCL queues?
Up to now we only used default execution space instances, but what if you want to have concurrent kernels on the GPU?

Execution Space Instances

Execution Space instances behave largely like CUDA/HIP streams

You can create different instances:

```
ExecSpace space;
std::vector<ExecSpace> instances =
  Kokkos::Experimental::partition_space(space, w1, ..., wN);
```

- Creates a vector of N new instances of ExecSpace

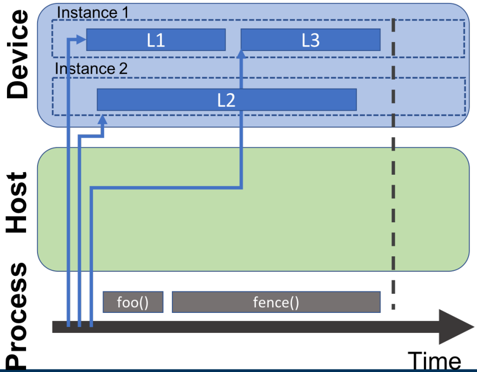
So what about CUDA/HIP streams and SYCL queues?

```
ExecSpace space;
std::vector<ExecSpace> instances =
  Kokkos::Experimental::partition_space(space, w1, ..., wN);
```

- The new instances will submit to a subset of the resources associated with `space`.
- The weights are hints as to the fraction of resources these instances are allowed to dispatch too.
  - Use integers or floating point numbers
  - Number of instances created is equal to number of weights
  - Note weights are hints, and might be ignored.
- Returns a `std::vector` of execution space instances
- The function may return just N copies of `space`
- all the created instances are independent of each other and from the original instance

Instances of Execution Spaces own an exec queue

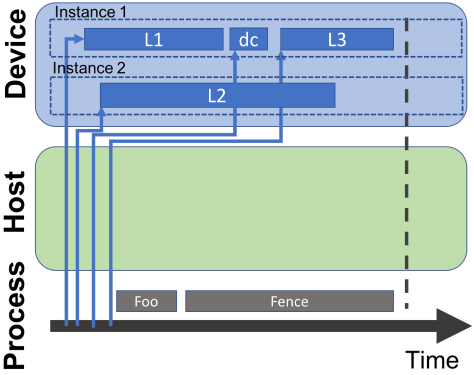
- Work dispatched to different instances overlaps with each other
- Overlaps with host process functions
- Use `Kokkos::fence()` to wait for completion of all




```
auto instances = partiti
Device dev1 = instances[
Device dev2 = instances[
RangePolicy<Device>
  policy_d1(dev1,0,N);
RangePolicy<Device>
  policy_d2(dev2,0,N);

parallel_for("L1",
  policy_d1, L1);
parallel_for("L2",
  policy_d2, L2);
parallel_for("L3",
  policy_d1, L3);
foo();
fence();
```

deep_copy with an instance argument also overlap

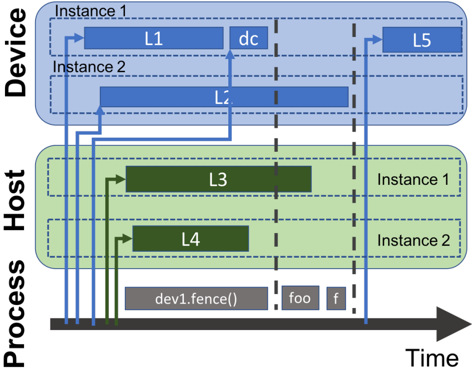
- deep_copy with an instance argument are like any other parallel operation
- Overlaps with parallel operations in other instance




```
parallel_for("L1",
  policy_d1, L1);
parallel_for("L2",
  policy_d2, L2);
deep_copy(dev1,
  dest, src);
parallel_for("L3",
  policy_d1, L3);
foo();
fence();
```

There are instance fences

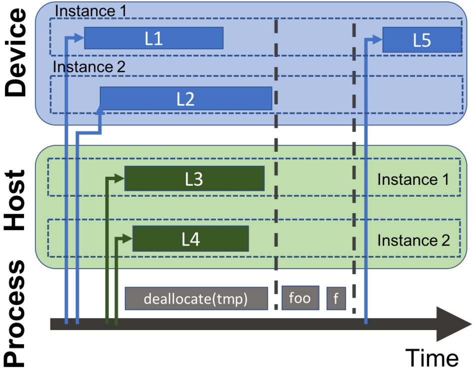
- Use instance specific fence to only wait on that instance
- Operations in other instances can overlap with that fence
- Use `Kokkos::fence()` to wait for all outstanding ops




```
parallel_for("L1",
  policy_d1, L1);
parallel_for("L2",
  policy_d2, L2);
deep_copy(dev1,
  dest, src);
parallel_for("L3",
  policy_h1, L3);
parallel_for("L4",
  policy_h2, L4);
dev1.fence();
foo();
fence();
parallel_for("L5",
  policy_d1, L5);
```

Reality Check: Kokkos Views deallocation implies fence!

- Due to limitations of reference counting, deallocations fence!
- **Important:** this is implementation limitation not semantic!
- **Do NOT rely on deallocations fencing!**




```
{
View<..> tmp(...);
parallel_for("L1",
  policy_d1, L1);
parallel_for("L2",
  policy_d2, L2);
parallel_for("L3",
  policy_h1, L3);
parallel_for("L4",
  policy_h2, L4);
}
foo();
fence();
parallel_for("L5",
  policy_d1, L5);
```

- Execution Space Instances execute work in order of dispatch.
- Operations dispatched to different Execution Space Instances can overlap.
- Each Execution Space type has a default instance as a singleton.
- Use `Kokkos::fence()` to wait for completion of ALL outstanding work.
- Use `exec_space_instance.fence()` to wait for completion of outstanding work dispatched to a specific execution space instance.

Exercise: Explore impact of using instances for concurrent execution.

Getting set up in your home directory:

```
mkdir Kokkos
cd Kokkos
git clone https://github.com/kokkos/kokkos
git clone https://github.com/kokkos/kokkos-tutorials
```

Find the exercise in the kokkos-tutorials/Exercises/instances folder.

The Begin subdirectory contains the code. Only instances_begin.cpp needs modifications.

Simply type `make -j 8` to build the exercise.

Look for EXERCISE comments to find places to modify.

Tasks:

- Create ExecutionSpace instances
- Use the instances to dispatch the kernels and `deep_copy`
- Remember to use device and host accessible views as results for reductions!

Things to try:

- Compare using the same instance for both data sets, and different ones.
- Compare performance for small (e.g. -N 5000) and large (e.g. -N 30000) problems.
- What happens if you create the result Views inside the repeat loop?