Module 1: Introduction and Parallel Dispatch

April 24, 2024
Kokkos is C++ Performance Portability

- Write a *single source* implementation using C++
- Use a *descriptive* Programming Model
- Compile for GPUs and CPUs

Kokkos is Ready for Use

- Well established project since 2012
- Major buy-in by DOE National Labs
- Well over 100 projects with over 500 developers use Kokkos
- Dedicated developer staff at 5 National Labs
- Robust support for software stacks: GCC 8+, Clang 8+, NVCC 11+, ROCM 5.2, Intel 19+
07/17 Module 1: Introduction, Building and Parallel Dispatch
07/24 Module 2: Views and Spaces
07/31 Module 3: Data Structures + MultiDimensional Loops
08/07 Module 4: Hierarchical Parallelism
08/14 Module 5: Tasking, Streams and SIMD
08/21 Module 6: Internode: MPI and PGAS
08/28 Module 7: Tools: Profiling, Tuning and Debugging
09/04 Module 8: Kernels: Sparse and Dense Linear Algebra
09/11 Reserve Day
Exercises

- Exercises are small codes with places to do modifications.
- Access to GPUs helpful for most of them, but most can be done on pure CPU systems.
- Only dependent on standard compilers (e.g. Clang, NVCC)
## Introduction
What is Kokkos? Who is behind it? Why should you use it?

## Parallel Dispatch
Pattern, Policy and Body: how to parallelize simple code with Kokkos.
Introduction

Learning objectives:

▶ Why do we need Kokkos
▶ The Kokkos EcoSystem
**The HPC Hardware Landscape**

**Current Generation:** Programming Models OpenMP 3, CUDA and OpenACC depending on machine

- LANL/SNL Trinity
  - Intel Haswell / Intel KNL
  - OpenMP 3
- LLNL SIERRA
  - IBM Power9 / NVIDIA Volta
  - CUDA / OpenMP
- ORNL Summit
  - IBM Power9 / NVIDIA Volta
  - CUDA / OpenACC / OpenMP
- SNL Astra
  - ARM CPUs
  - OpenMP 3
- Riken Fugaku
  - ARM CPUs with SVE
  - OpenMP 3 / OpenACC

**Upcoming Generation:** Programming Models OpenMP 5, CUDA, HIP and DPC++ depending on machine

- NERSC Perlmutter
  - AMD CPU / NVIDIA GPU
  - CUDA / OpenMP
- ORNL Frontier
  - AMD CPU / AMD GPU
  - HIP / OpenMP
- ANL Aurora
  - Xeon CPUs / Intel GPUs
  - DPC++ / OpenMP
- LLNL El Capitan
  - AMD CPU / AMD GPU
  - HIP / OpenMP

(a) Initially not working. Now more robust for Fortran than C++, but getting better.
(b) Research effort.
(c) OpenMP 5 by NVIDIA.
(d) OpenMP 5 by AMD.
(e) OpenMP 5 by Intel.
(f) OpenMP 5 by HPE.
Industry Estimate

A full time software engineer writes 10 lines of production code per hour: 20k LOC/year.

**Conservative estimate:** need to rewrite 10% of an app to switch Programming Model
A full time software engineer writes 10 lines of production code per hour: 20k LOC/year.

**Conservative estimate:** need to rewrite 10% of an app to switch Programming Model

Just switching Programming Models costs multiple person-years per app!
What is Kokkos?

- A C++ Programming Model for Performance Portability
  - Implemented as a template library on top CUDA, HIP, OpenMP, ...
  - Aims to be descriptive not prescriptive
  - Aligns with developments in the C++ standard
- Expanding solution for common needs of modern science and engineering codes
  - Math libraries based on Kokkos
  - Tools for debugging, profiling and tuning
  - Utilities for integration with Fortran and Python
- It is an Open Source project with a growing community
  - Maintained and developed at https://github.com/kokkos
  - Hundreds of users at many large institutions
Knowledge of C++: class constructors, member variables, member functions, member operators, template arguments

Using your own \texttt{{HOME}}

- Git
- GCC 8.2 (or newer) \textit{OR} Intel 19.0.5 (or newer) \textit{OR} Clang 8.0 (or newer)
- CUDA nvcc 11.0 (or newer) \textit{AND} NVIDIA compute capability 6.0 (or newer)
- \texttt{git clone https://github.com/kokkos/kokkos}
  into \texttt{{HOME}}/Kokkos/kokkos
- \texttt{git clone https://github.com/kokkos/kokkos-tutorials}
  into \texttt{{HOME}}/Kokkos/kokkos-tutorials

Slides are in
\texttt{{HOME}}/Kokkos/kokkos-tutorials/LectureSeries

Exercises are in
\texttt{{HOME}}/Kokkos/kokkos-tutorials/Exercises

\textit{Exercises' makefiles look for} \texttt{{HOME}}/Kokkos/kokkos
Online Resources:

- https://github.com/kokkos: Primary Kokkos GitHub Organization
- https://github.com/kokkos/kokkos-tutorials: Tutorial exercises
Kokkos’ basic capabilities:
- Simple 1D data parallel computational patterns
- Deciding where code is run and where data is placed
- Managing data access patterns for performance portability
- Multidimensional data parallelism

Kokkos’ advanced capabilities:
- Thread safety, thread scalability, and atomic operations
- Hierarchical patterns for maximizing parallelism
- Task based programming with Kokkos

Kokkos’ tools and Kernels:
- How to profile, tune and debug Kokkos code
- Interacting with Python and Fortran
- Using Kokkos Kernels math library
Tutorial Takeaways

- Kokkos enables **Single Source Performance Portable Codes**
- **Simple things stay simple** - it is not much more complicated than OpenMP
- **Advanced performance optimizing capabilities** easier to use with Kokkos than e.g. CUDA or HIP
- Kokkos provides data abstractions critical for performance portability not available in other programming models
- **Controlling data access patterns is key for obtaining performance**
- The **Kokkos Ecosystem** comes with tools (profiling, debugging, tuning, math libraries, etc.) needed for application development in professional settings
Assume you are here because:

- Want to use all HPC node architectures; including GPUs
- Are familiar with C++
- Want GPU programming to be easier
- Would like portability, as long as it doesn’t hurt performance

Helpful for understanding nuances:

- Are familiar with data parallelism
- Are familiar with OpenMP
- Are familiar with GPU architecture and CUDA
Target machine:
Important Point: Performance Portability

There’s a difference between *portability* and *performance portability*.

**Example**: implementations may target particular architectures and may not be *thread scalable*.

- (e.g., locks on CPU won’t scale to 100,000 threads on GPU)
There’s a difference between *portability* and *performance portability*.

**Example:** implementations may target particular architectures and may not be *thread scalable*.

(e.g., locks on CPU won’t scale to 100,000 threads on GPU)

**Goal:** write one *implementation* which:

- compiles and runs on multiple architectures,
- obtains *performant memory access patterns* across architectures,
- can leverage *architecture-specific features* where possible.
There’s a difference between *portability* and *performance portability*.  

**Example**: implementations may target particular architectures and may not be *thread scalable*.  
(e.g., locks on CPU won’t scale to 100,000 threads on GPU)

**Goal**: write one implementation which:

- compiles and runs on multiple architectures,
- obtains performant memory access patterns across architectures,
- can leverage architecture-specific features where possible.

**Kokkos**: performance portability across manycore architectures.
Concepts for Data Parallelism

Learning objectives:

▶ Terminology of pattern, policy, and body.
▶ The data layout problem.
for (element = 0; element < numElements; ++element) {
    total = 0;
    for (qp = 0; qp < numQPs; ++qp) {
        total += dot(left[element][qp], right[element][qp]);
    }
    elementValues[element] = total;
}
Concepts: Patterns, Policies, and Bodies

Pattern

\[
\text{for (element } = 0; \text{ element } < \text{ numElements}; \text{ ++element }) \{ \\
\text{ total } = 0; \\
\text{ for (qp } = 0; \text{ qp } < \text{ numQPs}; \text{ ++qp }) \{ \\
\text{ total } += \text{ dot(left[element][qp], right[element][qp]); } \\
\} \\
\text{ elementValues[element] } = \text{ total; } \\
\}
\]

Policy

Terminology:

- **Pattern**: structure of the computations
  
  for, reduction, scan, task-graph, ...

- **Execution Policy**: how computations are executed
  
  static scheduling, dynamic scheduling, thread teams, ...

- **Computational Body**: code which performs each unit of work; e.g., the loop body

  ⇒ The **pattern** and **policy** drive the computational **body**.
What if we want to thread the loop?

```c
for (element = 0; element < numElements; ++element) {
    total = 0;
    for (qp = 0; qp < numQPs; ++qp) {
        total += dot(left[element][qp], right[element][qp]);
    }
    elementValues[element] = total;
}
```

(Change the execution policy from "serial" to "parallel.")

OpenMP is simple for parallelizing loops on multi-core CPUs, but what if we then want to do this on other architectures? Intel PHI and NVIDIA GPU and AMD GPU and...
What if we want to thread the loop?

```c
#pragma omp parallel for
for (element = 0; element < numElements; ++element) {
    total = 0;
    for (qp = 0; qp < numQPs; ++qp) {
        total += dot(left[element][qp], right[element][qp]);
    }
    elementValues[element] = total;
}
```

(Change the execution policy from “serial” to “parallel.”)
What if we want to **thread** the loop?

```c
#pragma omp parallel for
for (element = 0; element < numElements; ++element) {
    total = 0;
    for (qp = 0; qp < numQPs; ++qp) {
        total += dot(left[element][qp], right[element][qp]);
    }
    elementValues[element] = total;
}
```

*(Change the *execution policy* from “serial” to “parallel.”)*

OpenMP is simple for parallelizing loops on multi-core CPUs, but what if we then want to do this on **other architectures**?

Intel PHI *and* NVIDIA GPU *and* AMD GPU *and* ...
Option 1: OpenMP 4.5

```c
#pragma omp target data map(...) 
#pragma omp teams num_teams(...) num_threads(...) private(...) 
#pragma omp distribute 
for (element = 0; element < numElements; ++element) { 
    total = 0 
#pragma omp parallel for 
    for (qp = 0; qp < numQPs; ++qp)
        total += dot(left[element][qp], right[element][qp]);
    elementValues[element] = total;
} 
```
Option 1: OpenMP 4.5

```c
#pragma omp target data map(...)
#pragma omp teams num_teams(...) num_threads(...) private(...)
#pragma omp distribute
for (element = 0; element < numElements; ++element) {
    total = 0
#pragma omp parallel for
    for (qp = 0; qp < numQPs; ++qp)
        total += dot(left[element][qp], right[element][qp]);
    elementValues[element] = total;
}
```

Option 2: OpenACC

```c
#pragma acc parallel copy(...) num_gangs(...) vector_length(...)
#pragma acc loop gang vector
for (element = 0; element < numElements; ++element) {
    total = 0;
    for (qp = 0; qp < numQPs; ++qp)
        total += dot(left[element][qp], right[element][qp]);
    elementValues[element] = total;
}
```
A standard thread parallel programming model *may* give you portable parallel execution *if* it is supported on the target architecture.

But what about performance?
A standard thread parallel programming model 
*may* give you portable parallel execution 
*if* it is supported on the target architecture.

But what about performance?

Performance depends upon the computation’s’s 
memory access pattern.
#pragma something, opencl, etc.
for (element = 0; element < numElements; ++element) {
    total = 0;
    for (qp = 0; qp < numQPs; ++qp) {
        for (i = 0; i < vectorSize; ++i) {
            total += left[element * numQPs * vectorSize + qp * vectorSize + i] * 
                    right[element * numQPs * vectorSize + qp * vectorSize + i];
        }
    }
    elementValues[element] = total;
}
#pragma something, opencl, etc.
for (element = 0; element < numElements; ++element) {
    total = 0;
    for (qp = 0; qp < numQPs; ++qp) {
        for (i = 0; i < vectorSize; ++i) {
            total +=
            left[element * numQPs * vectorSize +
                    qp * vectorSize + i] *
            right[element * numQPs * vectorSize +
                    qp * vectorSize + i];
        }
    }
    elementValues[element] = total;
}

Memory access pattern problem: CPU data layout reduces GPU performance by more than 10X.
#pragma something, opencl, etc.
for (element = 0; element < numElements; ++element) {
    total = 0;
    for (qp = 0; qp < numQPs; ++qp) {
        for (i = 0; i < vectorSize; ++i) {
            total +=
            left[element * numQPs * vectorSize +
            qp * vectorSize + i] *
            right[element * numQPs * vectorSize +
            qp * vectorSize + i];
        }
    }
    elementValues[element] = total;
}

Memory access pattern problem: CPU data layout reduces GPU performance by more than 10X.

Important Point

For performance the memory access pattern must depend on the architecture.
Data parallel patterns

Learning objectives:

▶ How computational bodies are passed to the Kokkos runtime.
▶ How work is mapped to execution resources.
▶ The difference between `parallel_for` and `parallel_reduce`.
▶ Start parallelizing a simple example.
Data parallel patterns and work

```c
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex) {
    atomForces[atomIndex] = calculateForce(...data...);
}
```

Kokkos maps **work** to execution resources
Data parallel patterns and work

```cpp
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex) {
    atomForces[atomIndex] = calculateForce(...data...);
}
```

Kokkos maps `work` to execution resources

- each iteration of a computational body is a **unit of work**.
- an **iteration index** identifies a particular unit of work.
- an **iteration range** identifies a total amount of work.
Data parallel patterns and work

```c++
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex) {
    atomForces[atomIndex] = calculateForce(...data...);
}
```

Kokkos maps work to execution resources

- each iteration of a computational body is a unit of work.
- an iteration index identifies a particular unit of work.
- an iteration range identifies a total amount of work.

Important concept: Work mapping

You give an iteration range and computational body (kernel) to Kokkos, and Kokkos decides how to map that work to execution resources.
How are computational bodies given to Kokkos?

As functors or function objects, a common pattern in C++. Quick review, a functor is a function with data. Example:

```cpp
struct ParallelFunctor {
    ...
    void operator ()(a work assignment) const {
        /* ... computational body ... */
    }
};
```
How are computational bodies given to Kokkos?

As **functors** or *function objects*, a common pattern in C++.
How are computational bodies given to Kokkos?

As **functors** or **function objects**, a common pattern in C++.

Quick review, a **functor** is a function with data. Example:

```cpp
struct ParallelFunctor {
    ...
    void operator()(a work assignment) const {
        /* ... computational body ... */
        ...
    }
};
```
How is work assigned to functor operators?

A total amount of work items is given to a Kokkos pattern, `ParallelFunctor` functor:

```cpp
Kokkos::parallel_for(
    numberOfIterations, functor);
```

Warning: concurrency and order

Concurrency and ordering of parallel iterations is not guaranteed by the Kokkos runtime.
How is work assigned to functor operators?

A total amount of work items is given to a Kokkos pattern,

```cpp
ParallelFunctor functor;
Kokkos::parallel_for(numberOfIterations, functor);
```

Warning: concurrency and order
Concurrent and ordering of parallel iterations is not
guaranteed by the Kokkos runtime.
How is work assigned to functor operators?

A total amount of work items is given to a Kokkos pattern,

```cpp
ParallelFunctor functor;
Kokkos::parallel_for(numberOfIterations, functor);
```

and work items are assigned to functors one-by-one:

```cpp
struct Functor {
    void operator()(const int64_t index) const {...}
};
```
How is work assigned to functor operators?

A total amount of work items is given to a Kokkos pattern,

```cpp
ParallelFunctor functor;
Kokkos::parallel_for(numberOfIterations, functor);
```

and work items are assigned to functors one-by-one:

```cpp
struct Functor {
    void operator()(const int64_t index) const {...}
};
```

**Warning: concurrency and order**

Concurrency and ordering of parallel iterations is *not* guaranteed by the Kokkos runtime.
How is data passed to computational bodies?

for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex) {
    atomForces[atomIndex] = calculateForce(...data...);
}

struct AtomForceFunctor {
    ...
    void operator()(const int64_t atomIndex) const {
        atomForces[atomIndex] = calculateForce(...data...);
    }
    ...
}
How is data passed to computational bodies?

```cpp
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex) {
    atomForces[atomIndex] = calculateForce(...data...);
}
```

```cpp
struct AtomForceFunctor {
...
    void operator()(const int64_t atomIndex) const {
        atomForces[atomIndex] = calculateForce(...data...);
    }
...
}
```

How does the body access the data?

**Important concept**

A parallel functor body must have access to all the data it needs through the functor’s **data members**.
Putting it all together: the complete functor:

```cpp
struct AtomForceFunctor {
    ForceType _atomForces;
    DataType _atomData;
    AtomForceFunctor(/* args */) {...}
    void operator()(const int64_t atomIndex) const {
        _atomForces[atomIndex] = calculateForce(_atomData);
    }
};
```
Putting it all together: the complete functor:

```cpp
struct AtomForceFunctor {
    ForceType _atomForces;
    DataType _atomData;
    AtomForceFunctor(/* args */) {...}
    void operator()(const int64_t atomIndex) const {
        _atomForces[atomIndex] = calculateForce(_atomData);
    }
};
```

Q/ How would we reproduce serial execution with this functor?

```cpp
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex){
    atomForces[atomIndex] = calculateForce(data);
}
```
Putting it all together: the complete functor:

```cpp
struct AtomForceFunctor {
    ForceType _atomForces;
    DataType _atomData;
    AtomForceFunctor(/* args */) {...}
    void operator()(const int64_t atomIndex) const {
        _atomForces[atomIndex] = calculateForce(_atomData);
    }
};
```

Q/ How would we reproduce serial execution with this functor?

```cpp
Serial
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex){
    atomForces[atomIndex] = calculateForce(data);
}
```

```cpp
Functor
AtomForceFunctor functor(atomForces, data);
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex){
    functor(atomIndex);
}
```
The complete picture (using functors):

1. Defining the functor (operator+data):

```cpp
struct AtomForceFunctor {
    ForceType _atomForces;
    DataType _atomData;

    AtomForceFunctor(ForceType atomForces, DataType data) :
        _atomForces(atomForces), _atomData(data) {}

    void operator()(const int64_t atomIndex) const {
        _atomForces[atomIndex] = calculateForce(_atomData);
    }
}
```

2. Executing in parallel with Kokkos pattern:

```cpp
AtomForceFunctor functor(atomForces, data);
Kokkos::parallel_for(numberOfAtoms, functor);
```
Functors are tedious $\Rightarrow$ **C++11 Lambdas** are concise

```cpp
atomForces already exists
data already exists
Kokkos::parallel_for(numberOfAtoms, 
  [=] (const int64_t atomIndex) {
    atomForces[atomIndex] = calculateForce(data);
  }
);
```
Functors are tedious \( \Rightarrow \) **C++11 Lambdas** are concise

```cpp
atomForces already exists
data already exists
Kokkos::parallel_for(numberOfAtoms,
    [=] (const int64_t atomIndex) {
        atomForces[atomIndex] = calculateForce(data);
    }
);
```

A lambda is not *magic*, it is the compiler **auto-generating** a **functor** for you.
Functors are tedious ⇒ **C++11 Lambdas** are concise

```
atomForces already exists
data already exists
Kokkos::parallel_for(numberOfAtoms, [=] (const int64_t atomIndex) {
    atomForces[atomIndex] = calculateForce(data);
});
```

A lambda is not *magic*, it is the compiler **auto-generating** a **functor** for you.

**Warning: Lambda capture and C++ containers**

For portability to GPU a lambda must capture by value `[=]`. Don’t capture containers (e.g., `std::vector`) by value because it will copy the container’s entire contents.
How does this compare to OpenMP?

**Serial**

```c
for (int64_t i = 0; i < N; ++i) {
    /* loop body */
}
```

**OpenMP**

```c
#pragma omp parallel for
for (int64_t i = 0; i < N; ++i) {
    /* loop body */
}
```

**Kokkos**

```c
parallel_for(N, [=] (const int64_t i) {
    /* loop body */
});
```

**Important concept**

Simple Kokkos usage is **no more conceptually difficult** than OpenMP, the annotations just go in different places.
Riemann-sum-style numerical integration:

\[ y = \int_{\text{lower}}^{\text{upper}} \text{function}(x) \, dx \]
Riemann-sum-style numerical integration:

\[ y = \int_{lower}^{upper} function(x) \, dx \]

```c
double totalIntegral = 0;
for (int64_t i = 0; i < numberOfIntervals; ++i) {
    const double x =
        lower + (i / numberOfIntervals) * (upper - lower);
    const double thisIntervalsContribution = function(x);
    totalIntegral += thisIntervalsContribution;
}
totalIntegral *= dx;
```
**Riemann-sum-style numerical integration**:

\[ y = \int_{lower}^{upper} function(x) \, dx \]

```cpp
double totalIntegral = 0;
for (int64_t i = 0; i < numberOfIntervals; ++i) {
    const double x =
        lower + (i/numberOfIntervals) * (upper - lower);
    const double thisIntervalsContribution = function(x);
    totalIntegral += thisIntervalsContribution;
}
totalIntegral *= dx;
```

How do we **parallelize** it? *Correctly?*
Riemann-sum-style numerical integration:

\[ y = \int_{\text{lower}}^{\text{upper}} \text{function}(x) \, dx \]

Pattern?

double totalIntegral = 0;
for (int64_t i = 0; i < numberOfIntervals; ++i) {
    const double x = lower + (i/numberOfIntervals) * (upper - lower);
    const double thisIntervalsContribution = function(x);
    totalIntegral += thisIntervalsContribution;
}

Policy?

totalIntegral *= dx;

Body?

How do we **parallelize** it? **Correctly**?
An (incorrect) attempt:

double totalIntegral = 0;
Kokkos::parallel_for(numberOfIntervals, [=] (const int64_t index) {
  const double x =
    lower + (index/numberOfIntervals) * (upper - lower);
  totalIntegral += function(x);},
);
totalIntegral *= dx;

First problem: compiler error; cannot increment totalIntegral
(lambdas capture by value and are treated as const!)
An (incorrect) solution to the (incorrect) attempt:

double totalIntegral = 0;
double * totalIntegralPointer = &totalIntegral;
Kokkos::parallel_for(numberOfIntervals, 
  [=] (const int64_t index) {
    const double x =
      lower + (index/numberOfIntervals) * (upper - lower);
    *totalIntegralPointer += function(x);}, 
  );
totalIntegral *= dx;
An (incorrect) solution to the (incorrect) attempt:

double totalIntegral = 0;
double * totalIntegralPointer = &totalIntegral;
Kokkos::parallel_for(numberOfIntervals,
                    [=] (const int64_t index) {
                        const double x =
                            lower + (index/numberOfIntervals) * (upper - lower);
                        *totalIntegralPointer += function(x);},
                );
totalIntegral *= dx;

Second problem: race condition

<table>
<thead>
<tr>
<th>step</th>
<th>thread 0</th>
<th>thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>load</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>increment</td>
<td>load</td>
</tr>
<tr>
<td>2</td>
<td>write</td>
<td>increment</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>write</td>
</tr>
</tbody>
</table>
Root problem: we’re using the **wrong pattern**, for instead of *reduction*
Root problem: we’re using the **wrong pattern**, *for* instead of *reduction*

**Important concept: Reduction**

Reductions combine the results contributed by parallel work.
Root problem: we’re using the **wrong pattern**, *for* instead of *reduction*

**Important concept: Reduction**

Reductions combine the results contributed by parallel work.

How would we do this with **OpenMP**?

```c
double finalReducedValue = 0;
#pragma omp parallel for reduction(+:finalReducedValue)
for (int64_t i = 0; i < N; ++i) {
    finalReducedValue += ...;
}
```

How will we do this with **Kokkos**?

```c
double finalReducedValue = 0;
parallel_reduce(N, functor, finalReducedValue);
```
Scalar integration (3)

Root problem: we’re using the **wrong pattern**, *for* instead of *reduction*

**Important concept: Reduction**
Reductions combine the results contributed by parallel work.

How would we do this with **OpenMP?**

```c
double finalReducedValue = 0;
#pragma omp parallel for reduction(+:finalReducedValue)
for (int64_t i = 0; i < N; ++i) {
    finalReducedValue += ...;
}
```

How will we do this with **Kokkos**?

```c
double finalReducedValue = 0;
parallel_reduce(N, functor, finalReducedValue);
```
Example: Scalar integration

```c
double totalIntegral = 0;
#pragma omp parallel for reduction(+:totalIntegral)
for (int64_t i = 0; i < numberOfIntervals; ++i) {
    totalIntegral += function(...);
}
```

```c
double totalIntegral = 0;
parallel_reduce(numberOfIntervals,
    [=] (const int64_t i, double & valueToUpdate) {
        valueToUpdate += function(...);
    },
    totalIntegral);
```

The operator takes **two arguments**: a work index and a value to update.

The second argument is a **thread-private value** that is managed by Kokkos; it is not the final reduced value.
Warning: Parallelism is NOT free

Dispatching (launching) parallel work has non-negligible cost.

\[ \text{Time} = \alpha + \beta \times N \]

\[ \alpha = \text{dispatch overhead} \]
\[ \beta = \text{time for a unit of work} \]
\[ N = \text{number of units of work} \]
\[ P = \text{available concurrency} \]

\[ \text{Speedup} = \frac{P}{1 + \alpha \times P \beta \times N} \]

You should have \( \alpha \times P \ll \beta \times N \)

All runtimes strive to minimize the dispatch overhead \( \alpha \).

Find more parallelism to increase \( N \).

Merge (fuse) parallel operations to increase \( \beta \).
Warning: Parallelism is NOT free

Dispatching (launching) parallel work has non-negligible cost.

Simplistic data-parallel performance model: Time = $\alpha + \frac{\beta N}{P}$

- $\alpha$ = dispatch overhead
- $\beta$ = time for a unit of work
- $N$ = number of units of work
- $P$ = available concurrency
Warning: Parallelism is NOT free

Dispatching (launching) parallel work has non-negligible cost.

Simplistic data-parallel performance model: \[ \text{Time} = \alpha + \frac{\beta \times N}{P} \]

- \(\alpha\) = dispatch overhead
- \(\beta\) = time for a unit of work
- \(N\) = number of units of work
- \(P\) = available concurrency

\[ \text{Speedup} = P \div \left(1 + \frac{\alpha \times P}{\beta \times N}\right) \]

- Should have \(\alpha \times P \ll \beta \times N\)
- All runtimes strive to minimize launch overhead \(\alpha\)
- Find more parallelism to increase \(N\)
- Merge (fuse) parallel operations to increase \(\beta\)
**Results**: illustrates simple speedup model = \[ P \div \left(1 + \frac{\alpha P}{\beta N}\right) \]
Always name your kernels!

Giving unique names to each kernel is immensely helpful for debugging and profiling. You will regret it if you don’t!

- Non-nested parallel patterns can take an optional string argument.
- The label doesn’t need to be unique, but it is helpful.
- Anything convertible to ”std::string”
- Used by profiling and debugging tools (see Profiling Tutorial)

Example:

```cpp
double totalIntegral = 0;
parallel_reduce("Reduction", numberOfIntervals,
   [=] (const int64_t i, double & valueToUpdate) {
      valueToUpdate += function(...);
   },
   totalIntegral);
```
Recurring Exercise: Inner Product

**Exercise:** Inner product \(< y, A \ast x >\)

**Details:**
- \(y\) is \(N\times1\), \(A\) is \(N\times M\), \(x\) is \(M\times1\)
- We’ll use this exercise throughout the tutorial
The **first step** in using Kokkos is to include, initialize, and finalize:

```cpp
#include <Kokkos_Core.hpp>
int main(int argc, char* argv[]) {
    /* ... do any necessary setup (e.g., initialize MPI) ... */
    Kokkos::initialize(argc, argv);
    {
        /* ... do computations ... */
    }
    Kokkos::finalize();
    return 0;
}
```

(Optional) Command-line arguments or environment variables:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--kokkos-num-threads=INT</code></td>
<td>total number of threads</td>
</tr>
<tr>
<td><code>KOKKOS_NUM_THREADS</code></td>
<td></td>
</tr>
<tr>
<td><code>--kokkos-device-id=INT</code></td>
<td>device (GPU) ID to use</td>
</tr>
<tr>
<td><code>KOKKOS_DEVICE_ID</code></td>
<td></td>
</tr>
</tbody>
</table>
Exercise: Inner product $<y, A \ast x>$

Details:
- Location: Exercises/01/Begin/
- Look for comments labeled with “EXERCISE”
- Need to include, initialize, and finalize Kokkos library
- Parallelize loops with `parallel_for` or `parallel_reduce`
- Use lambdas instead of functors for computational bodies.
- For now, this will only use the CPU.
Compiling for CPU

```bash
cmake -B build -DKokkos_ENABLE_OPENMP=ON \ 
    -DCMAKE_BUILD_TYPE=Release

cmake --build build
```

Running on CPU with OpenMP backend

```bash
# Set OpenMP affinity
export OMP_NUM_THREADS=8
export OMP_PROC_BIND=spread OMP_PLACES=threads

# Print example command line options:
./build/01_Exercise -h

# Run with defaults on CPU
./build/01_Exercise

# Run larger problem
./build/01_Exercise -S 26
```

Things to try:

- Vary problem size with command line argument `-S s`
- Vary number of rows with command line argument `-N n`
- Num rows = $2^n$, num cols = $2^m$, total size = $2^s = 2^{n+m}$
Basic capabilities we haven’t covered

- Customizing `parallel_reduce` data type and reduction operator
  - e.g., minimum, maximum, ...
- `parallel_scan` pattern for exclusive and inclusive prefix sum
- Using *tag dispatch* interface to allow non-trivial functors to have multiple “operator()” functions.
  - very useful in large, complex applications
Simple usage is similar to OpenMP, advanced features are also straightforward.

Three common data-parallel patterns are parallel_for, parallel_reduce, and parallel_scan.

A parallel computation is characterized by its pattern, policy, and body.

User provides computational bodies as functors or lambdas which handle a single work item.
Building Applications with Kokkos

Learning objectives:

▶ Kokkos-docs: https://kokkos.org/kokkos-core-wiki/building.html
▶ NERSC-docs: https://docs.nersc.gov/development/programming-models/kokkos/
Building Applications with Kokkos

Learning objectives:

▶ Kokkos-docs: https://kokkos.org/kokkos-core-wiki/building.html
▶ NERSC-docs: https://docs.nersc.gov/development/programming-models/kokkos/

Ignore This For Tutorial Only

The following details on options to integrate Kokkos into your build process are NOT necessary to know if you just want to do the tutorial.
Kokkos EcoSystem:

- C++ Performance Portability Programming Model.
- The Kokkos Ecosystem provides capabilities needed for serious code development.
- Kokkos is supported by multiple National Laboratories with a sizeable dedicated team.
Data Parallelism:

- Simple things stay simple!
- You use **parallel patterns** and **execution policies** to execute computational bodies
- Simple parallel loops use the **parallel_for** pattern:

```cpp
parallel_for("Label", N, [=] (int64_t i) {
    /* loop body */
});
```

- Reductions combine contributions from loop iterations

```cpp
int result;
parallel_reduce("Label", N, [=] (int64_t i, int& lres) {
    /* loop body */
    lres += /* something */
}, result);
```
Kokkos::View:
- Solving the data-layout issue.
- Controlling data life-time.

Execution and Memory Spaces:
- How to control where data lives.
- How to control where code executes.
- How to manage data transfers.

Don’t Forget: Join our Slack Channel and drop into our office hours on Tuesday.

Updates at: 
https://github.com/kokkos/kokkos-tutorials/issues/38