The Kokkos Lectures

Module 6: Fortran/Python interoperability, MPI and PGAS

August 21, 2020

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.
SAND2020-8766 PE

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.

- 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

SIMD Types

- SIMD types help vectorize code.
- In particular for **outer-loop** vectorization.
- There are **storage** and **temporary** types.
- Currently considered experimental at `https://github.com/kokkos/simd-math`: please try it out and provide feedback.

Blocking Behavior and Streams

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

Module 6: Learning objectives

Language Interoperability

- Writing hybrid Fortran applications
- And, writing hybrid Python applications with Kokkos

Interoperability with MPI

- Learning how to develop hybrid MPI+Kokkos applications
- And, how to implement overlapping communication with computation
- Handling buffers and sparse indexing

Interoperability with PGAS

- How to create globally accessible Views
- Insight into a use-case (SPMV)

Fortran InterOp

How to write hybrid Fortran - Kokkos code.

Learning objectives:

- Allocating data in Fortran and viewing it as Kokkos Views.
- Calling C++ functions with Kokkos in it from Fortran.
- Allocating DualView's from within Fortran.

HPC world owns many Fortran LOC!

- We generally cannot port it all at once.
- We need an incremental porting strategy
  - Keep our e.g. Fortran mains, drivers, physics packages
  - But port relevant infrastructure, or hotspot kernels to C++ and Kokkos

Why do we need this?

HPC world owns many Fortran LOC!

- We generally cannot port it all at once.
- We need an incremental porting strategy
  - Keep our e.g. Fortran mains, drivers, physics packages
  - But port relevant infrastructure, or hotspot kernels to C++ and Kokkos

How do we make Kokkos and Fortran talk with each other?

Fortran Language Compatibility Layer (FLCL)

- Pass multidimensional arrays accross the C++/Fortran boundary
  - See Fortran arrays as Kokkos Views and vice versa
- Create Kokkos View and DualView from Fortran
  - Allows Fortran to be the memory owner but call C++ functions with Kokkos kernels for CUDA/HIP
- Initialize and Finalize Kokkos from Fortran
- FortranIndex<T> scalar type to deal with 1 vs 0 based indexing in sparse data structures

Fortran Language Compatibility Layer (FLCL)

- Pass multidimensional arrays accross the C++/Fortran boundary
  - See Fortran arrays as Kokkos Views and vice versa
- Create Kokkos View and DualView from Fortran
  - Allows Fortran to be the memory owner but call C++ functions with Kokkos kernels for CUDA/HIP
- Initialize and Finalize Kokkos from Fortran
- FortranIndex<T> scalar type to deal with 1 vs 0 based indexing in sparse data structures

FLCL

The Fortran Language Compatibility Layer allows an incremental porting of a Fortran code to Kokkos.

Simple binding of `Kokkos::initialize` and `Kokkos::finalize`

- `kokkos_initialize()`
  - Call after `MPI_Initialize`
  - Parses the command line arguments of the executable
- `kokkos_initialize_without_args()`
  - Call after `MPI_Initialize`
  - Ignores command line arguments of the executable
  - Kokkos will still look up environment variables
- `kokkos_finalize`
  - Call before `MPI_Finalize`

nd_array_t

The compatibility glue between Fortran arrays and Kokkos Views.

nd_array_t

The compatibility glue between Fortran arrays and Kokkos Views.

Keeps Track of:

- An array's rank
- Extents of the array
- Strides of the array
- Pointer to the allocation

nd_array_t

The compatibility glue between Fortran arrays and Kokkos Views.

Keeps Track of:

- An array's rank
- Extents of the array
- Strides of the array
- Pointer to the allocation

How do we create an `nd_array_t`?

- Explicit routines like `to_nd_array_i64_d6`
- Simple interface taking a fortran array as argument

```
array = to_nd_array(foo)  ! Fortran
```

nd_array_t

The compatibility glue between Fortran arrays and Kokkos Views.

Keeps Track of:

- An array's rank
- Extents of the array
- Strides of the array
- Pointer to the allocation

How do we create an `nd_array_t`?

- Explicit routines like `to_nd_array_i64_d6`
- Simple interface taking a fortran array as argument

```
array = to_nd_array(foo)  ! Fortran
```

This allows us to write a simple hybrid Fortran/Kokkos code!

Everyone loves AXPBY so we do it here!

- Using a few modules including iso_c_binding and the flcl_mod provided by FLCL
- axpby is just a fortran subroutine taking fortran arguments (next slide)

```
program example_axpy
  use, intrinsic :: iso_c_binding
  use :: flcl_mod
  use :: axpy_f_mod
  implicit none

  real(c_double), dimension(:), allocatable :: c_y
  real(c_double), dimension(:), allocatable :: x
  real(c_double) :: alpha
  integer :: mm = 5000
  ... setup here ...
  call kokkos_initialize()
  call axpy(c_y, x, alpha)
  call kokkos_finalize()
end program example_axpy
```

AXPBY

```
module axpy_f_mod
    use, intrinsic :: iso_c_binding
    use :: flcl_mod
    public
      interface
        subroutine f_axpy( nd_array_y, nd_array_x, alpha ) &
          & bind(c, name='c_axpy')
          use, intrinsic :: iso_c_binding
          use :: flcl_mod
          type(nd_array_t) :: nd_array_y
          type(nd_array_t) :: nd_array_x
          real(c_double) :: alpha
        end subroutine f_axpy
      end interface
      contains
        subroutine axpy( y, x, alpha )
          use, intrinsic :: iso_c_binding
          use :: flcl_mod
          implicit none
          real(c_double), dimension(:), intent(inout) :: y
          real(c_double), dimension(:), intent(in) :: x
          real(c_double), intent(in) :: alpha
          call f_axpy(to_nd_array(y), to_nd_array(x), alpha)
        end subroutine axpy
end module axpy_f_mod
```

AXPY - The C++

In C++ create an unmanaged view from the `nd_array_t` handle:

```cpp
#include "flcl-cxx.hpp"
extern "C" {
  void c_axpy( flcl_ndarray_t *nd_array_y,
               flcl_ndarray_t *nd_array_x,
               double *alpha )
  {
    auto y = flcl::view_from_ndarray<double*>(*nd_array_y);
    auto x = flcl::view_from_ndarray<double*>(*nd_array_x);

    Kokkos::parallel_for( "axpy", y.extent(0),
      KOKKOS_LAMBDA( const size_t idx) {
      y(idx) += *alpha * x(idx);
    });
  }
}
```

- All the arrays are `LayoutLeft` i.e. Fortran Layout
- The data type needs to match.

FLCL allows allocating DualViews from Fortran

```
real(c_double), dimension(:), pointer :: array_x
type(c_ptr) :: v_x
... setup here ...
call kokkos_allocate_dualview(array_x, v_x, "array_x", length)
```

- array_x is an array aliasing the host view of the dualview
- v_x is a pointer to the DualView itself.

FLCL allows allocating DualViews from Fortran

```
real(c_double), dimension(:), pointer :: array_x
type(c_ptr) :: v_x
... setup here ...
call kokkos_allocate_dualview(array_x, v_x, "array_x", length)
```

- `array_x` is an array aliasing the host view of the dualview
- `v_x` is a pointer to the DualView itself.

In C++ take a pointer to a pointer of a DualView as argument:

```
#include "flcl-cxx.hpp"
void c_foo( flcl::dualview_r64_1d_t**  v_x ) {
  flcl::dualview_r64_1d_t dv_x = **v_x;
  dv_x.sync_device();
  ...
}
```

- Can assign to an instance, DualView is reference counted
- Note: there is NO type safety in the Fortran/C++ boundary, better make sure you get the types right!

Write the same gluecode as for AXPY example:

```
module interface_f_mod
  use, intrinsic :: iso_c_binding
  use :: flcl_mod
  implicit none
  public
  interface
    subroutine foo( v_x ) &
      & bind(c, name="c_foo")
      use, intrinsic :: iso_c_binding
      use :: flcl_mod
      implicit none
      type(c_ptr), intent(inout) :: v_x
    end subroutine foo
  end interface
end module interface_f_mod

void c_foo( flcl::dualview_r64_1d_t**  v_x ) {
  flcl::dualview_r64_1d_t dv_x = **v_x;
  dv_x.sync_device();
  ...
}
```

Summary

- Fortran Language Compatibility Layer provides facilities for interoperability of Kokkos and Fortran
- Initialize Kokkos from Fortran via `kokkos_initialize` and `kokkos_finalize`
- `nd_array_t` is a representation of a `Kokkos::View`
- Create `nd_array_t` from a Fortran array via `to_nd_array`
- Allocate `Kokkos::DualView` in Fortran with `kokkos_allocate_dualview`

Available at
`https://github.com/kokkos/kokkos-fortran-interop`.

- Feedback is appreciated!

Python InterOp

How to write hybrid Python - Kokkos code.

Learning objectives:

- Allocating data in Python and viewing it as Kokkos Views in C++.
- Allocating data in C++ and viewing it as Numpy Arrays in Python.

Work-flows orchestrated by Python with the heavy lifting in C++ is increasing in popularity

- Python is excellent for data pre-processing, post-processing, and visualization
  - Easy to manipulate data into other forms
  - Easy to import packages which handle various I/O formats (JSON, YAML, etc.)
  - Standard library has rich set of packages for operating system services, file/directory access, networking, statistics, etc.
- Python is inefficient at computationally-intensive tasks
  - Dynamic type system requires a lot of type-checking, even in simple `c = a * b`
  - Python statements are not optimized for execution on specific architecture

Work-flows orchestrated by Python with the heavy lifting in C++ is increasing in popularity

- Python is excellent for data pre-processing, post-processing, and visualization
  - Easy to manipulate data into other forms
  - Easy to import packages which handle various I/O formats (JSON, YAML, etc.)
  - Standard library has rich set of packages for operating system services, file/directory access, networking, statistics, etc.
- Python is inefficient at computationally-intensive tasks
  - Dynamic type system requires a lot of type-checking, even in simple `c = a * b`
  - Python statements are not optimized for execution on specific architecture

How do we make Kokkos and Python talk with each other?

PyBind11 is a C++ template library for mapping C++ types and functions to Python

- Despite the syntax of Python having more similarities to C++ than C (e.g. classes), the most popular implementation of the Python interpreter is written in C (CPython)
  - C++ code needs to be translated into implementations of the CPython API
  - PyBind11 provides this translation through template meta-programming
- NumPy is the *de facto* standard for arrays in Python
  - NumPy `ndarray` is quite similar to `Kokkos::DynamicView` in many respects
  - Goal is to provide Kokkos Views which can be treated as NumPy arrays: `array = numpy.array(view, copy=False)`

Similar to Fortran, Kokkos initialize and finalize will be available in Python

- The primary caveat will be how to invoke `kokkos.finalize()`
  - Invoking `Kokkos::finalize()` in C++ requires all Kokkos data structures to no longer have reference counts
  - Python scoping rules are quite different than C++ scoping rules
  - `kokkos.finalize()` will run the garbage collector but the invocation must be in a different function outside of any variables holding a reference to a Kokkos View.

Sample Python Kokkos

```python
import numpy
import kokkos

def main():
    # 2D double-precision view in host memory space
    view = kokkos.array([10, 2],
        dtype=kokkos.double,
        space=kokkos.HostSpace)
    arr = numpy.array(view, copy=False)
    print("Kokkos View : {} (shape={})".format(
        type(view).__name__, view.shape))
    print("Numpy Array : {} (shape={})".format(
        type(arr).__name__, arr.shape))

if __name__ == "__main__":
    kokkos.initialize()
    main()
    # gc.collect() <-- implicitly run in finalize()
    kokkos.finalize()
```

Sample User Bindings - CMake

```cmake
cmake_minimum_required(VERSION 3.10 FATAL_ERROR)

project(Kokkos-Python-Example LANGUAGES C CXX)

find_package(Kokkos REQUIRED)
find_package(pybind11 REQUIRED)

# user library using Kokkos
add_library(user SHARED user.cpp user.hpp)
target_link_libraries(user PUBLIC Kokkos::kokkos)

# python bindings to user library
pybind11_add_module(example
    ${PROJECT_SOURCE_DIR}/example.cpp)
target_link_libraries(example PRIVATE user)

# copy example script to build directory
configure_file(${PROJECT_SOURCE_DIR}/example.py
    ${PROJECT_BINARY_DIR}/example.py COPYONLY)
```

Sample User Bindings - user.hpp

```cpp
#include "Kokkos_Core.hpp"

// views returning to python must explicitly
// specify memory space
//
using view_type = Kokkos::View<double**, Kokkos::HostSpace>;

// This is meant to emulate some function that exists
// in a user library which returns a Kokkos::View and will
// have a python binding created in example.cpp
//
view_type generate_view(size_t n, size_t m);
```

Sample User Bindings - user.cpp

```
#include "user.hpp"

view_type
generate_view(size_t n, size_t m)
{
  view_type _v("random_view", n, m);
  // populate some data
  // ...
  // v(1, 0) = 0
  // v(1, 1) = 1
  // v(2, 0) = 2
  // v(2, 1) = 0
  // v(3, 0) = 0
  // v(3, 1) = 3
  // v(4, 0) = 4
  // ...
  for (size_t i = 0; i < n; ++i)
  {
    _v(i, i % m) = i;
  }
  return _v;
}
```

Sample User Bindings - example.cpp

```
#include "user.hpp"
#include <pybind11/pybind11.h>

namespace py = pybind11;

PYBIND11_MODULE(example, ex) {
  ///
  /// This is a python binding to the user-defined
  /// 'generate_view' function declared in user.hpp
  /// which returns a Kokkos::View. Default arguments
  /// are specified via py::arg(...) and are optional.
  ///
  ex.def(
    "generate_view",            // python function
    &generate_view,             // C++ function
    "Generate a random view",   // doc string
    py::arg("n") = 10,          // default arg
    py::arg("m") = 2            // default arg
  );
}
```

Sample User Bindings - example.py

```python
import argparse
import numpy
import kokkos

# pybind11 will generate dynamic python module:
#   example.cpython-37m-darwin.so
# and just import normally
import example

def main(args):
    view = example.generate_view(args.n, args.m)
    arr = numpy.array(view, copy=False)
    # should see printout of data set in C++ code
    print(arr)

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("-n", default=10, type=int)
    parser.add_argument("-m", default=2, type=int)

    kokkos.initialize()
    main(parser.parse_args())
    kokkos.finalize()
```

This is in pre-release: ask us for access.

The Python Interop provides:

- Initialize and Finalize Kokkos from Python
- Create Views from Python
- Alias Kokkos Views with NumPy arrays

- For now relies on pybind11.
- We are looking for feedback on functionality and usability!

MPI - Kokkos Interoperability

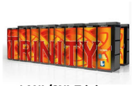
Writing a hybrid MPI - Kokkos program.

Learning objectives:

- How to send data from Kokkos Views.
- How to overlap communication and computation.
- Buffer packing strategies.
- How to generate sparse index lists.

MPI + Kokkos

Current supercomputers expose node-level parallelism

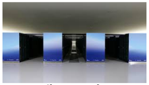


LANL/SNL Trinity
Intel Haswell / Intel KNL

LLNL SIERRA
IBM Power9 / NVIDIA Volta

ORNL Summit
IBM Power9 / NVIDIA Volta

SNL Astra
ARM CPUs



Riken Fugaku
ARM CPUs with SVE

Upcoming Generation: MPI + {OpenMP 5, CUDA, HIP or DPC++ depending on machine}

NERSC Perlmutter
AMD CPU / NVIDIA GPU

ORNL Frontier
AMD CPU / AMD GPU

ANL Aurora
Xeon CPUs / Intel GPUs

LLNL El Capitan
AMD CPU / AMD GPU

- Today supercomputers are clusters with disjoint address spaces
- One additional level of concurrency: node-level
- Allow MPI+Kokkos hybrid applications (using patterns, views, spaces, etc.)

Why mix MPI with Kokkos

- Need to address internode data transfers
- MPI is the de-facto standard
- MPI is well supported on all platforms
  - MPI also knows how to talk to GPUs
- The legacy code you want to port is already using MPI
- Programming explicitly to the parallelism hierarchy can help

Why mix MPI with Kokkos

- Need to address internode data transfers
- MPI is the de-facto standard
- MPI is well supported on all platforms
  - MPI also knows how to talk to GPUs
- The legacy code you want to port is already using MPI
- Programming explicitly to the parallelism hierarchy can help

Are there any alternatives?

- You can potentially use PGAS models (discussed later).
- Global tasking models may work for you
  - Kokkos has been used with Uintah for years
  - LANL explores combining Legion and Kokkos with our support

Motivating Example - Vector Shift

Simple Shifting of data:

```
View<double*> A("A",N), B("B",N);
// Single Device
parallel_for("Shift",N,KOKKOS_LAMBDA(int i) {
  B((i+K)%N) = A(i);
});
```

Motivating Example - Vector Shift

Simple Shifting of data:

```
View<double*> A("A",N), B("B",N);
// Single Device
parallel_for("Shift",N,KOKKOS_LAMBDA(int i) {
  B((i+K)%N) = A(i);
});
```

Lets assume we have R ranks:

- Now each rank owns N/R elements
- Rank j needs to send K elements to rank j+1
  - j is sending the last K elements of A
- Rank j needs to receive K elements from rank j-1
  - j is receiving into the first K elements of B

- The MPI Interface uses raw pointers

```
Kokkos::View<...> recv_view(...);
Kokkos::View<...> send_view(...);
void* recv_ptr = recv_view.data();
void* send_ptr = send_view.data();
```

- Data needs to be stored contiguously ⇒ `LayoutStride` not possible
- Data stored on the device requires GPU-aware MPI implementations. Otherwise, copying to the host is necessary

```
auto recv_view_h = Kokkos::create_mirror_view_and_copy(
  Kokkos::DefaultHostExecutionSpace{}, recv_view_d);
auto send_view_h = Kokkos::create_mirror_view_and_copy(
  Kokkos::DefaultHostExecutionSpace{}, send_view_d);
void* recv_ptr = recv_view_h.data();
void* send_ptr = send_view_h.data();
```

Then the usual MPI functions can be used:

```
MPI_Request requests[2];
MPI_Irecv(recv_ptr,recv_view.size(),MPI_DOUBLE,
          source,1,MPI_COMM_WORLD,&requests[0]);
// Send the buffer
MPI_Isend(send_ptr,send_view.size(),MPI_DOUBLE,
          target,1,MPI_COMM_WORLD,&requests[1]);
// Wait for communication to finish
MPI_Waitall(2,requests,MPI_STATUSES_IGNORE);
```

Then the usual MPI functions can be used:

```
MPI_Request requests[2];
MPI_Irecv(recv_ptr,recv_view.size(),MPI_DOUBLE,
          source,1,MPI_COMM_WORLD,&requests[0]);
// Send the buffer
MPI_Isend(send_ptr,send_view.size(),MPI_DOUBLE,
          target,1,MPI_COMM_WORLD,&requests[1]);
// Wait for communication to finish
MPI_Waitall(2,requests,MPI_STATUSES_IGNORE);
```

In our example send_view and recv_view are just subviews:

```
auto send_view = Kokkos::subview(A,std::make_pair(myN-K, myN));
auto recv_view = Kokkos::subview(B,std::make_pair(0, K));
```

Simple Overlapping Communication

Overlap communication with computation if possible!

- Make sure compute kernel don not access send/recv buffers
- Post sends and recvs first
- Launch kernel
- Wait on MPI

Simple Overlapping Communication

Overlap communication with computation if possible!

- Make sure compute kernel don not access send/recv buffers
- Post sends and recvs first
- Launch kernel
- Wait on MPI

Vector-Shift Example:

```
auto send_view = Kokkos::subview(A,std::make_pair(myN-K, myN));
auto recv_view = Kokkos::subview(B,std::make_pair(0, K));
MPI_Request requests[2];
// Post sends/recv
MPI_Irecv(recv_ptr,recv_view.size(),MPI_DOUBLE,
          source,1,MPI_COMM_WORLD,&requests[0]);
MPI_Isend(send_ptr,send_view.size(),MPI_DOUBLE,
          target,1,MPI_COMM_WORLD,&requests[1]);
parallel_for("ShiftA",RangePolicy<>(K,myN),
  KOKKOS_LAMBDA(int i) { B(i) = A(i-K); });
// Wait for communication to finish
MPI_Waitall(2,requests,MPI_STATUSES_IGNORE);
```

Technical requirements

- Initialize MPI before Kokkos

```
int main(int argc, char* argv[]) {
  MPI_Init(&argc,&argv);
  Kokkos::initialize(argc,argv);
  [...]
  Kokkos::finalize();
  MPI_Finalize();
  }
```

- By default, GPUs are distributed in a round-robin fashion if there are multiple.
- Use `mpicxx` as compiler and `OMPI_CXX=<path-to-kokkos-install>/nvcc_wrapper` (for OpenMPI) or use `find_package`(MPI REQUIRED) with `CMake`.

Exercise: Send Data between MPI Processes

- Location: `Exercises/mpi_pack_unpack/`
- Add missing MPI calls to `RunPackCommUnpackTest::run_comm()`.
- Compile and run on CPU, and then on GPU.

```
mkdir build && cd build
export Kokkos_DIR=<path-to-kokkos-install>
cmake .. && make
# Run exercise
mpiexec -np 2 MPIPackUnpack
```

Command line arguments

- Vary size of data
- Vary size of buffers
- Number of repeats for timing
- Copy to host first

Explicit Message Buffers

Sometimes extra send/recv buffers are needed

- Buffer data which is getting written to again
- Sparse data needs to be sent or received
  - In particular if isn't regular strided
  - Will discuss some best practices later
- The system doesn't allow MPI to access some memory space

Explicit Message Buffers

Sometimes extra send/recv buffers are needed

- Buffer data which is getting written to again
- Sparse data needs to be sent or received
    - In particular if isn't regular strided
    - Will discuss some best practices later
- The system doesn't allow MPI to access some memory space

The Pack-Send/Recv-Unpack Cycle:

- Post Irecvs
- Pack buffers
- Post Isends
- Wait on message completion
- Unpack buffers

Explicit Message Buffers

Based on our Kokkos knowledge of Execution and Memory Spaces the following question arises:

Where should the pack kernel run, and where should it pack to?

- Run the pack kernel wherever the data lives.
- The best memory space for the pack buffer depends.
- Sometimes packing into a device buffer, and still explicitly copying to the host is best.

There are a number of options for CUDA for example:

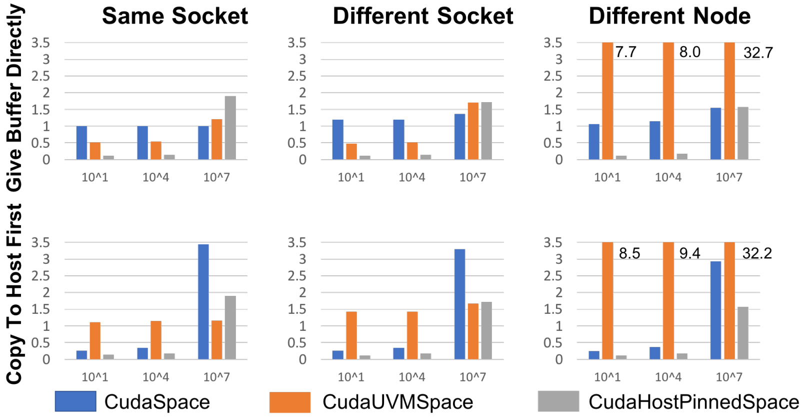
Data Space	Pack Buffer Space	Explicit HostCopy
CudaSpace	CudaSpace	yes
CudaSpace	CudaSpace	no
CudaSpace	CudaUVMSpace	no
CudaSpace	CudaHostPinnedSpace	no

CudaSpace vs CudaUVMSpace vs CudaHostPinnedSpace

- Time relative to CudaSpace on single socket (lower is better).
- Performance is very sensitive to system and configurations!!

CudaSpace vs CudaUVMSpace vs CudaHostPinnedSpace

- Time relative to CudaSpace on single socket (lower is better).
- Performance is very sensitive to system and configurations!!



Leveraging streams allow calculations and communication with explicit buffers to overlap!

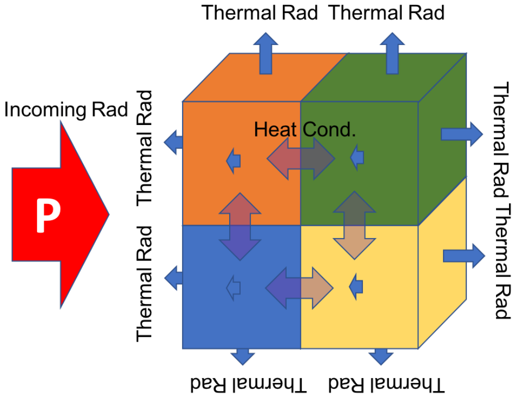
- Execute packing, unpacking and deep_copies with different streams than interior kernels.
- Submission order is important though
  - Worksets are largely worked on in submission order even with CUDA streams
  - Need to submit pack kernels first, then interiors kernel
  - Unless priority streams are a thing ...
- Fence the pack execution space instances only before issuing sends
- ExecutionSpace instances, and buffers need to be persistent - allocations add fencing

Code Skeleton:

```
// Create execution space instances
ExecSpace exec_pack(..), exec_comp(..);
using exec_policy = RangePolicy<ExecSpace>;
// Post Receives
MPI_Irecv(...);
// Launch pack kernel in pack exec instance
// Likely this uses only few cores
parallel_for("PackBuffer",
  exec_policy(exec_pack,0,N),fpack);
// Launch compute kernel independent of message exchange
parallel_for("Interior",
  exec_policy(exec_comp,0,N),finterior);
// Wait for pack kernel to finish before sending data
exec_pack.fence();
MPI_Isend(...);
// Wait for communication to finish
MPI_Waitall(...);
// Unpack received data - may still overlap with "Interior"
parallel_for("UnpackBuffer",
  exec_policy(exec_pack,0,N),funpack);
// Wait for all work to finish
Kokkos::fence();
```

3D Heat Conduction

- Heat conduction inside the body
- Thermal radiation (Black Body) on surface
- Incoming power flow from one direction



No Overlapping

Data Structures:

- T(x,y,z): temperature in cell (x,y,z)
- dT(x,y,z): temperature change in time increment dt
- T_left,right,up,down,front,back: recv buffers for boundaries
- T_left_out,...: send buffers for boundaries

Time approach:

- deep_copy boundary layers as needed to contiguous send buffers
- Post MPI send/recv with send/recv buffers
- Launch kernel for interior elements doing heat conduction only
- Wait for MPI
- Compute updates for boundary elements using the recv buffers

Overlapping packing/unpacking with interior compute

Getting better performance can be achieved by staging calls correctly, and using execution space instances:

- Use 7 instances: interior, 6x boundary (left, right, ...)
- Issue Irecv
- Run up to 6 pack kernels using different exec space instances
- Launch interior kernel into its own instance
- Fence first boundary pack instance, issue Isend
- Fence other boundary packs and issue Isends one by one
- Wait for MPI operations to finish
- Issue boundary temperature update kernel
- Fence everything

Exercise: Optimize 3D Heat

Optimize the basic MPI implementation of the 3D heat conduction code for GPU systems.

Details:

- Location: `Exercises/mpi_heat_conduction/`
- Use Execution Space instances for more overlapping
- Order operations for maximum overlapping
- Run with correct GPU mapping

Things to try:

- Try strong vs weak scaling
- Change Problem Size -X , -Y , -Z
- Play with buffer memory space
- Compare same socket, vs different socket, vs multi node perf

Sparse Data to be Send

In the heat conduction example some surfaces were sparse, but regular.

Sparse Data to be Send

In the heat conduction example some surfaces were sparse, but regular.

How best to generate index lists if data is not regular sparse?

Sparse Data to be Send

In the heat conduction example some surfaces were sparse, but regular.

How best to generate index lists if data is not regular sparse?

Example problems:

- Send all particles which crossed the boundary.
- Send all elements in contact with the surface.

Sparse Data to be Send

In the heat conduction example some surfaces were sparse, but regular.

How best to generate index lists if data is not regular sparse?

Example problems:

- Send all particles which crossed the boundary.
- Send all elements in contact with the surface.

Use indirect pack kernel:

```
parallel_for("Pack",num_send, KOKKOS_LAMBDA(int e) {
  pack(e) = data(send_list(e));
});
```

Sparse Data to be Send

In the heat conduction example some surfaces were sparse, but regular.

How best to generate index lists if data is not regular sparse?

Example problems:

- Send all particles which crossed the boundary.
- Send all elements in contact with the surface.

Use indirect pack kernel:

```
parallel_for("Pack",num_send, KOKKOS_LAMBDA(int e) {
  pack(e) = data(send_list(e));
});
```

How to generate the send_list?

In the heat conduction example some surfaces were sparse, but regular.

How best to generate index lists if data is not regular sparse?

Example problems:

- Send all particles which crossed the boundary.
- Send all elements in contact with the surface.

Use indirect pack kernel:

```
parallel_for("Pack",num_send, KOKKOS_LAMBDA(int e) {
  pack(e) = data(send_list(e));
});
```

How to generate the send_list?
=> Use a `parallel_scan`

Generating Index Lists via `parallel_scan`

- `bool needs_send(int e)` is true if element e needs to be sent:

```
parallel_scan("GenIDX",num_elements,
  KOKKOS_LAMBDA(int e, int& idx, bool final){
  if(needs_send(e)) {
    if(final) send_list(idx) = e;
    idx++;
  }
});
```

Generating Index Lists via `parallel_scan`

- `bool needs_send(int e)` is true if element `e` needs to be sent:

```
parallel_scan("GenIDX",num_elements,
  KOKKOS_LAMBDA(int e, int& idx, bool final){
  if(needs_send(e)) {
    if(final) send_list(idx) = e;
    idx++;
  }
});
```

What if you don't know how large send_list needs to be?
=> Use parallel_scan with return argument; Repeat if count exceeds size.

Merged Count - Allocate - Fill pattern

```
// Initial Count Guess
int count = K;
send_list.resize(count);
parallel_scan("GenIDX1",N,KOKKOS_LAMBDA(int e, int& idx, bool f) {
  if(needs_send(e)) {
    // Only add if its smaller but keep counting
    if(final && idx<count) { send_list(idx)=e; } idx++;
  }
},count);
// If count indicates you ran over redo the kernel
if(count>send_list.extent(0)) {
  send_list.resize(count);
  parallel_scan("GenIDX2",N,KOKKOS_LAMBDA(int e, int& idx, bool f)
    if(needs_send(e)) { if(final) { send_list(idx)=e; } idx++; }
  },count);
}
```

- Worst case scenario: 2x cost
- If you remember count you will reach often steady state
- More complex memory pool based algorithms are often costly

CPU Core Assignment:

- Don't oversubscribe your CPU cores!
- By default for example each rank will use all cores in OpenMP
- Set process masks appropriately
  - OpenMPI: mpirun -np R –map-by socket=PE:4
  - mpich:
  - SLURM:

GPU Assignment:

- Tell Kokkos the number of GPUs used per node
  - –kokkos-num-devices=K
  - env varibale KOKKOS_NUM_DEVICES=K
- Kokkos will assign GPUs round robin i.e. MPI_Rank%K

Note: `jsrun` on `Summit` needs `--smpiargs="-gpu"` for GPU-aware MPI communication.

Simple MPI and Kokkos Interaction is easy!

- Simply pass `data()` of a View to MPI functions plus its size.
  - But it better be a contiguous View!
- Initialize Kokkos after MPI, and finalize it before MPI

Overlapping communication and computation possible

- Use Execution Space instances to overlap packing/unpacking with other computation.
- Order operations to maximize overlapping potential.

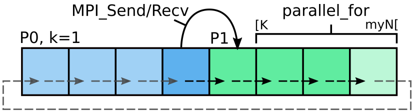
Kokkos Remote Spaces: Support for PGAS in Kokkos

How to write a PGAS application with Kokkos.

Learning objectives:

- How to create global Views.
- How access global data.
- Taking a closer look at SPMV (CG).

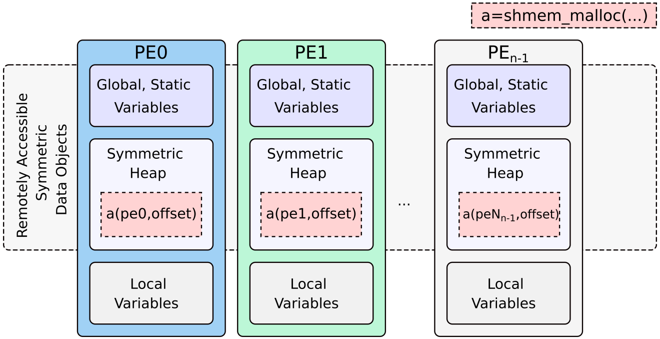
Previous Example: `Vector-Shift`



```
auto send_view = Kokkos::subview(A,std::make_pair(myN-K, myN));
auto recv_view = Kokkos::subview(B,std::make_pair(0, K));
...
MPI_Request requests[2];
MPI_Irecv(recv_ptr,recv_view.size(),MPI_DOUBLE,
          source,1,MPI_COMM_WORLD,&requests[0]);
MPI_Isend(send_ptr,send_view.size(),MPI_DOUBLE,
          target,1,MPI_COMM_WORLD,&requests[1]);
parallel_for("Shift",RangePolicy<>(K,myN),
  KOKKOS_LAMBDA(int i) { B(i) = A(i-K); });
MPI_Waitall(2,requests,MPI_STATUSES_IGNORE);
```

- **How to simplify communication, and reduce host-device data movement?**

PGAS (Partitioned Global Address Space)



- Variable a is globally accessible through `Put` and `Get` operations, `PE` and `Offset` are used for global addressing.

Different implementations, same conceptual API

- OpenSHMEM
  - void *shmem_malloc(size_t size);
  - void shmem_T_p(T *dest, T value, int pe);
  - TYPE shmem_T_g(T *src, int pe);
- NVSHMEM
  - void *nvshmem_malloc(size_t size);
  - void nvshmem_T_p(T *dest, T value, int pe);
  - TYPE nvshmem_T_g(TYPE *srct, int pe);
- MPI One-Sided
  - int *MPI_Win_allocate(size_t size);
  - int MPI_Put(T *src, int count, MPI_TYPE, int target_pe,...);
  - int MPI_Get(T *target, int count , MPI_TYPE, int source_pe,...);

Programming with Kokkos Remote Spaces: Vector Shift

- Allocate a remote View

```
using RemoteSpace_t = Kokkos::Experimental::SHMEMSpace;
Kokkos::View<T**,RemoteSpace_t> a("A",numPEs,myN);
```

Programming with Kokkos Remote Spaces: Vector Shift

- Allocate a remote View

```
using RemoteSpace_t = Kokkos::Experimental::SHMEMSpace;
Kokkos::View<T**,RemoteSpace_t> a("A",numPEs,myN);
```

- Access global memory

```
a(0,0) = 6; //Writes 6 to view a on PE 0 at offset 0
a(1,8) = 3; //Writes 3 to view a on PE 1 at offset 8
```

Programming with Kokkos Remote Spaces: Vector Shift

- Allocate a remote View

```
using RemoteSpace_t = Kokkos::Experimental::SHMEMSpace;
Kokkos::View<T**,RemoteSpace_t> a("A",numPEs,myN);
```

- Access global memory

```
a(0,0) = 6; //Writes 6 to view a on PE 0 at offset 0
a(1,8) = 3; //Writes 3 to view a on PE 1 at offset 8
```

- Fence

```
RemoteSpace_t().fence();
```

Programming with Kokkos Remote Spaces: Vector Shift

- Allocate a remote View

```
using RemoteSpace_t = Kokkos::Experimental::SHMEMSpace;
Kokkos::View<T**,RemoteSpace_t> a("A",numPEs,myN);
```

- Access global memory

```
a(0,0) = 6; //Writes 6 to view a on PE 0 at offset 0
a(1,8) = 3; //Writes 3 to view a on PE 1 at offset 8
```

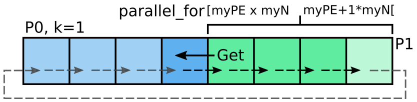
- Fence

```
RemoteSpace_t().fence();
```

- Copy data to other memory space

```
Kokkos::View<T**,Kokkos::HostSpace_t> a_h("A_h",1,myN);
Kokkos::Experimental::deep_copy(a_h,a);
```

Vector Shift with Kokkos Remote Spaces



```
RemoteView_t a("A",numPEs,myN);
RemoteView_t b("B",numPEs,myN);

Kokkos::parallel_for("Shift",Kokkos::RangePolicy<>
(myPE*myN,(myPE+1)*myN), KOKKOS_LAMBDA(const int i) {
  int j = i+k; //Shift
  b((j/myN)%numPEs, j%myN) = a(myPE, i);
});

RemoteSpace_t().fence();
```

Example Sparse Matrix Multiply $y = A * x$: Sparse Matrix Representation in Compressed Row Storage (CRS):

- Store non-zero matrix elements sorted by occurence.
- Store the actual column index of each value.
- Store the offsets of where each row begins.

Example Sparse Matrix Multiply $y = A * x$: Sparse Matrix Representation in Compressed Row Storage (CRS):

- Store non-zero matrix elements sorted by occurence.
- Store the actual column index of each value.
- Store the offsets of where each row begins.

Single Node Implementation:

```
Kokkos::parallel_for("SPMV",nrows, KOKKOS_LAMBDA(int row) {
for(j=A.row_offset(row); j< A.row_offset(row+1); j++)
y(row) += A.val(j)* x(A.idx(j));
});
```

Example Sparse Matrix Multiply $y = A * x$: Sparse Matrix Representation in Compressed Row Storage (CRS):

- Store non-zero matrix elements sorted by occurence.
- Store the actual column index of each value.
- Store the offsets of where each row begins.

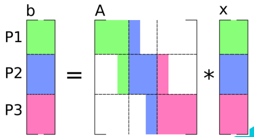
Single Node Implementation:

```
Kokkos::parallel_for("SPMV",nrows, KOKKOS_LAMBDA(int row) {
for(j=A.row_offset(row); j< A.row_offset(row+1); j++)
y(row) += A.val(j)* x(A.idx(j));
});
```

How to distribute data:

- The matrix is distributed by rows.
- The vectors are distributed by elements.

Problem:$A.idx(j)$ **may be on a remote node!**



MPI based SPMV needs a lof setup:

- Find all values of A.idx(j) outside this ranks owned range.
- Find the ranks of each of those values which own them.
- Create for each of those ranks the list of indicies needed.
- Send the list to each rank, where it becomes the send_list.

The data structures (Using View of Views (VoV)):

- num_recv_ranks: Number of ranks you need data from.
- num_send_ranks: Number of ranks you need to send data too.
- recv_buffers (2D VoV): list of recv buffer for each rank
    - subviews into the end of x beyond owned elements.
- send_buffers (2D VoV): list of send buffer for each rank
- send_lists (2D VoV): list of indicies to send to each rank

Code Skeleton for SPMV in MPI

```
// Post all the recvs
for(int i=0; i<num_recv_ranks; i++) {
  MPI_Irecv(recv_buffers(i).data(),nrecv(i)...);
}
// Pack send buffers and send
for(int i=0; i<num_send_ranks; i++) {
  // Get the send buffer and list for this rank
  auto sb = send_buffer(i);
  auto sl = send_list(i);
  // Pack the data
  Kokkos::parallel_for(nsend(i),KOKKOS_LAMBDA(int j)
    { sb(i) = x(sl(j)); });
  Kokkos::fence();
  // Send the data
  MPI_Isend(sb.data(),nsend(i),...);
}
// Wait for all the communication to be done
MPI_Waitall(...);
// Run the local code
parallel_for("SPMV",...);
```

Sparse communication with PGAS is easy!

- Only x is distributed!
- Simply keep using global indicies in A.idx
- Compute PE and offset with div and mod for accessing x

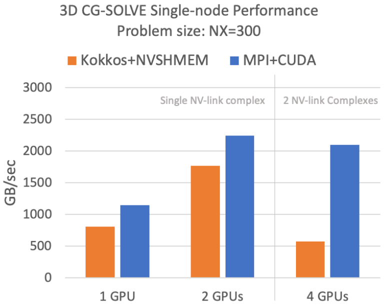
```
Kokkos::parallel_for("SPMV",my_nrows, KOKKOS_LAMBDA(int row) {
  for(j=A.row_offset(row); j< A.row_offset(row+1); j++) {
    int idx_j = A.idx(j);
    int pe = idx_j/N_local;
    int k = idx_j%N_local;
    y(row) += A.val(j)*x(pe,k);
  }
});
```

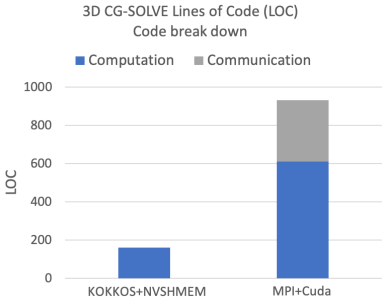
Remember the MPI Skeleton? This is the skeleton in PGAS:

```
parallel_for("SPMV",...);
```

CGSolve with Kokkos Remote Spaces

- Lassen Supercomputer (LLNL), IBM Power9, 4x NVidia Volta GPUs





Remote Spaces: Code and Configuration

Get it on GitHub

- https://github.com/kokkos/kokkos-remote-spaces

Configuration Example

- **NVSHMEM**: `cd $(kokkos_remote_spaces)`
```
cmake .  -DKokkos_DIR=$(KOKKOS_HOME)/install
-DCMAKE_CXX_COMPILER=mpicxx
-DNVSHMEM_ROOT=$(NVSHMEM_HOME)/install
-DKokkos_ENABLE_NVSHMEMSPACE=ON
```
- **SHMEM**: `-DKokkos_ENABLE_SHMEMSPACE=ON`
- **MPI One-Sided:**`-DKokkos_ENABLE_MPISPACE=ON`

Note: Setting the `-DKokkos_ENABLE_{SPACE}` CMAKE flag sets `Kokkos::Experimental::DefaultRemoteMemporySpace` to the given **PGAS backend**.

Exercise: Implementing a Distributed Vector-Shift

- Location: `Exercises/pgas_vectorshift/`
- Compile and run with one and two ranks using SHMEM (CPUs) or NVSHMEM (GPUs)

```
mkdir build && cd build
cmake . -DKokkosRemote_ROOT=<path-to-Kokkos-Remote-Spaces>/ins
cmake .. && make
# Run exercise
mpiexec -np 2 ./vectorshift --kokkos-num-devices=2
```

Summary: Kokkos Remote Spaces

- Adds distributed shared memory to Kokkos.
- View-templating defines memory space (SHMEM, NVSHMEM or MPI One-Sided).
- View `()-operator` implements the `Put/Get semantic`.
- Use `deep_copy(...)` to move data between memory spaces.

Module 6 Summary

Simple MPI and Kokkos Interaction is easy!

- Simply pass `data()` of a View to MPI functions plus its size.
  - But it better be a contiguous View!
- Initialize Kokkos after MPI, and finalize it before MPI

Overlapping communication and computation possible

- Use Execution Space instances to overlap packing/unpacking with other computation.
- Order operations to maximize overlapping potential.

Fortran Language Compatibility Layer

- Initialize Kokkos from Fortran via `kokkos_initialize` and `kokkos_finalize`
- `nd_array_t` is a representation of a `Kokkos::View`
- Create `nd_array_t` from a Fortran array via `to_nd_array`
- Allocate `Kokkos::DualView` in Fortran with `kokkos_allocate_dualview`

The Python Interop

- Initialize and Finalize Kokkos from Python
- Create Views from Python
- Alias Kokkos Views with NumPy arrays
- **This is in pre-release: ask us for access.**

Clang-Tidy Static Analysis

- Getting Kokkos specific warnings in your IDE

Kokkos Tools

- Debugging
- Profiling
- Tuning

Custom and 3rd party tools

- How to write distributed code using a global arrays like interface

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

Updates at: kokkos.link/the-lectures-updates

Recordings/Slides: kokkos.link/the-lectures