Lessons Learned from

Porting the MiniAero Application to Charm++

David S. Hollman, Janine Bennett (PI), Jeremiah Wilke (Chief Architect), Ken Franko, Hemanth Kolla, Paul Lin, Greg Sjaardema, Nicole Slattengren, Keita Teranishi, Nikhil Jain, Eric Mikida

May 7, 2015
Introduction
Introduction

The Process: Porting an Explicit Aerodynamics Miniapp to the Chare Model
What was easy?
What was harder?
Introduction

The Process: Porting an Explicit Aerodynamics Miniapp to the Chare Model
  What was easy?
  What was harder?

Preliminary Results and Performance
Introduction

The Process: Porting an Explicit Aerodynamics Miniapp to the Chare Model
   What was easy?
   What was harder?

Preliminary Results and Performance

Next Steps
Introduction

The Process: Porting an Explicit Aerodynamics Miniapp to the Chare Model
What was easy?
What was harder?

Preliminary Results and Performance

Next Steps
The DHARMA Project

DHARMA: Distributed asynchronous adaptive resilient models for applications

Project mission: Assess and address fundamental challenges imposed by the need for performant, portable, scalable, fault-tolerant programming models at extreme-scale

Research in programmability, dynamic load-balancing, and fault-tolerance of AMT runtimes

Comparative analysis portion of the project:
Assess various asynchronous many-task (AMT) runtimes by implementing mini-apps of interest to Sandia using existing runtimes
First three runtimes to assess: Charm++, Legion, Uintah
First mini-app for assessment: MiniAero

May 7, 2015
The DHARMA Project

- **DHARMA**: Distributed asyncHronous Adaptive Resilient Models for Applications

- Project mission: Assess and address fundamental challenges imposed by the need for performant, portable, scalable, fault-tolerant programming models at extreme-scale

- Research in programmability, dynamic load-balancing, and fault-tolerance of AMT runtimes

- Comparative analysis portion of the project:
  - Assess various asynchronous many-task (AMT) runtimes by implementing mini-apps of interest to Sandia using existing runtimes
  - First three runtimes to assess: Charm++, Legion, Uintah
  - First mini-app for assessment: MiniAero

May 7, 2015
The DHARMA Project

- **DHARMA:** Distributed asyncHronous Adaptive Resilient Models for Applications

- Project mission: Assess and address fundamental challenges imposed by the need for performant, portable, scalable, fault-tolerant programming models at extreme-scale
DHARMA: Distributed asynchronous Adaptive Resilient Models for Applications

Project mission: Assess and address fundamental challenges imposed by the need for performant, portable, scalable, fault-tolerant programming models at extreme-scale

Research in programmability, dynamic load-balancing, and fault-tolerance of AMT runtimes
DHARMA: Distributed asynchronous Adaptive Resilient Models for Applications

Project mission: Assess and address fundamental challenges imposed by the need for performant, portable, scalable, fault-tolerant programming models at extreme-scale

Research in programmability, dynamic load-balancing, and fault-tolerance of AMT runtimes

Comparative analysis portion of the project:
DHARMA: Distributed asynchronous Adaptive Resilient Models for Applications

Project mission: Assess and address fundamental challenges imposed by the need for performant, portable, scalable, fault-tolerant programming models at extreme-scale

Research in programmability, dynamic load-balancing, and fault-tolerance of AMT runtimes

Comparative analysis portion of the project:
- Assess various asynchronous many-task (AMT) runtimes by implementing mini-apps of interest to Sandia using existing runtimes
The DHARMA Project

- **DHARMA: Distributed asyncHronous Adaptive Resilient Models for Applications**
- **Project mission:** Assess and address fundamental challenges imposed by the need for performant, portable, scalable, fault-tolerant programming models at extreme-scale
- **Research in programmability, dynamic load-balancing, and fault-tolerance of AMT runtimes**
- **Comparative analysis portion of the project:**
  - Assess various asynchronous many-task (AMT) runtimes by implementing mini-apps of interest to Sandia using existing runtimes
  - First three runtimes to assess:
    - Charm++
    - Legion
    - Uintah
The DHARMA Project

- **DHARMA:** Distributed asynchronous Adaptive Resilient Models for Applications

- Project mission: Assess and address fundamental challenges imposed by the need for performant, portable, scalable, fault-tolerant programming models at extreme-scale

- Research in programmability, dynamic load-balancing, and fault-tolerance of AMT runtimes

- Comparative analysis portion of the project:
  - Assess various asynchronous many-task (AMT) runtimes by implementing mini-apps of interest to Sandia using existing runtimes
  - First three runtimes to assess:
    - Charm++
    - Legion
    - Uintah
  - First mini-app for assessment: MiniAero

May 7, 2015
The DHARMA Project

- DHARMA: Distributed asynchronous Adaptive Resilient Models for Applications

- Project mission: Assess and address fundamental challenges imposed by the need for performant, portable, scalable, fault-tolerant programming models at extreme-scale

- Research in programmability, dynamic load-balancing, and fault-tolerance of AMT runtimes

- Comparative analysis portion of the project:
  - Assess various asynchronous many-task (AMT) runtimes by implementing mini-apps of interest to Sandia using existing runtimes
  - First three runtimes to assess:
    - Charm++
    - Legion
    - Uintah
  - First mini-app for assessment: MiniAero
MiniAero is a proxy app illustrating the common computation and communication patterns in unstructured mesh codes of interest to Sandia.
What is MiniAero?

- MiniAero is a proxy app illustrating the common computation and communication patterns in unstructured mesh codes of interest to Sandia.
- 3D, unstructured, finite volume computational fluid dynamics code.
  - Uses Runge-Kutta fourth-order time marching.
  - Has options for first and second order spatial discretization.
  - Includes inviscid Roe and viscous Newtonian flux options.
What is MiniAero?

- MiniAero is a proxy app illustrating the common computation and communication patterns in unstructured mesh codes of interest to Sandia.
- 3D, unstructured, finite volume computational fluid dynamics code:
  - Uses Runge-Kutta fourth-order time marching
  - Has options for first and second order spatial discretization
  - Includes inviscid Roe and viscous Newtonian flux options
  - Baseline application is about 3800 lines of C++ code using MPI and Kokkos.
What is MiniAero?

- MiniAero is a proxy app illustrating the common computation and communication patterns in unstructured mesh codes of interest to Sandia.
- 3D, unstructured, finite volume computational fluid dynamics code
  - Uses Runge-Kutta fourth-order time marching
  - Has options for first and second order spatial discretization
  - Includes inviscid Roe and viscous Newtonian flux options
  - Baseline application is about 3800 lines of C++ code using MPI and Kokkos
- Very little task parallelism; mostly a data parallel problem.
What is MiniAero?

- MiniAero is a proxy app illustrating the common computation and communication patterns in unstructured mesh codes of interest to Sandia.
- 3D, unstructured, finite volume computational fluid dynamics code:
  - Uses Runge-Kutta fourth-order time marching
  - Has options for first and second order spatial discretization
  - Includes inviscid Roe and viscous Newtonian flux options
  - Baseline application is about 3800 lines of C++ code using MPI and Kokkos
- Very little task parallelism; mostly a data parallel problem
- Communication: ghost exchanges, unstructured mesh
Background and Status

Porting MiniAero to Charm++ began with a “bootcamp”: March 9-12, 2015
Led by Nikhil Jain and Eric Mikida
About 10 Sandia scientists in attendance

Since the workshop, we’ve had one scientist working 50% time on the port and a couple others working 10-20% time

Current state of the code:
- running and passes test suite
- most immediately apparent optimizations done
- SMP version does not work (Kokkos incompatibility)
Porting MiniAero to Charm++ began with a “bootcamp”:
Porting MiniAero to Charm++ began with a “bootcamp”:
- March 9-12, 2015
Background and Status

Porting MiniAero to Charm++ began with a “bootcamp”:
  - March 9-12, 2015
  - Led by Nikhil Jain and Eric Mikida
Porting MiniAero to Charm++ began with a “bootcamp”:

- March 9-12, 2015
- Led by Nikhil Jain and Eric Mikida
- About 10 Sandia scientists in attendance

Since the workshop, we've had one scientist working 50% time on the port and a couple others working 10-20% time.

Current state of the code:
- running and passes test suite
- most immediately apparent optimizations done
- SMP version does not work (Kokkos incompatibility)
Porting MiniAero to Charm++ began with a “bootcamp”:  
- March 9-12, 2015  
- Led by Nikhil Jain and Eric Mikida  
- About 10 Sandia scientists in attendance  

Since the workshop, we've had one scientist working 50% time on the port and a couple others working 10-20% time
Porting MiniAero to Charm++ began with a “bootcamp”:
- March 9-12, 2015
- Led by Nikhil Jain and Eric Mikida
- About 10 Sandia scientists in attendance

Since the workshop, we've had one scientist working 50% time on the port and a couple others working 10-20% time

Current state of the code:
Porting MiniAero to Charm++ began with a “bootcamp”:
- March 9-12, 2015
- Led by Nikhil Jain and Eric Mikida
- About 10 Sandia scientists in attendance

Since the workshop, we've had one scientist working 50% time on the port and a couple others working 10-20% time

Current state of the code:
- running and passes test suite
Porting MiniAero to Charm++ began with a “bootcamp”:
- March 9-12, 2015
- Led by Nikhil Jain and Eric Mikida
- About 10 Sandia scientists in attendance

Since the workshop, we've had one scientist working 50% time on the port and a couple others working 10-20% time

Current state of the code:
- running and passes test suite
- most immediately apparent optimizations done
Background and Status

- Porting MiniAero to Charm++ began with a “bootcamp”:
  - March 9-12, 2015
  - Led by Nikhil Jain and Eric Mikida
  - About 10 Sandia scientists in attendance

- Since the workshop, we've had one scientist working 50% time on the port and a couple others working 10-20% time

- Current state of the code:
  - running and passes test suite
  - most immediately apparent optimizations done
  - SMP version does not work (Kokkos incompatibility)
Introduction

The Process: Porting an Explicit Aerodynamics Miniapp to the Chare Model
What was easy?
What was harder?

Preliminary Results and Performance

Next Steps
What was easy?

Load balancing
   - At the end of the timestep...
   - if (doLoadBalance && timestepCounter % loadBalanceInterval == 0) {
     - serial {
       - Do the actual load rebalancing
       - this->AtSync();
     }
     - Called when load balancing is completed (required)
     - when ResumeFromSync() {}
What was easy?

- Load balancing
What was easy?

- Load balancing (synchronous)

Checkpointing
To disk: CkStartCheckpoint(...)
In memory (to partner node): CkStartMemCheckpoint(...)
Must be done synchronously

Both of these were key features we wanted to test in AMT runtimes, both were done essentially on the first day of coding.
What was easy?

- Load balancing (synchronous)

```java
// at the end of the timestep...
if (doLoadBalance && timestepCounter % loadBalanceInterval == 0) {
    serial {
        // Do the actual load rebalancing
        this->AtSync();
    }
    // Called when load balancing is completed (required)
    when ResumeFromSync() { }
}
```
What was easy?

- **Load balancing (synchronous)**

  ```java
  // at the end of the timestep...
  if (doLoadBalance && timestepCounter % loadBalanceInterval == 0) {
      serial {
          // Do the actual load rebalancing
          this->AtSync();
      }
  }

  // Called when load balancing is completed (required)
  when ResumeFromSync() {} 
  ```

- **Checkpointing**

  - To disk: `CkStartCheckpoint(...)`
  - In memory (to partner node): `CkStartMemCheckpoint(...)`

  Both of these were key features we wanted to test in AMT runtimes, both were done essentially on the first day of coding.

  Both of these require only serialization on the user side.
What was easy?

- **Load balancing (synchronous)**

```java
// at the end of the timestep...
if (doLoadBalance && timestepCounter % loadBalanceInterval == 0) {
    serial {
        // Do the actual load rebalancing
        this->AtSync();
    }
    // Called when load balancing is completed (required)
    when ResumeFromSync() { }
}
```

- **Checkpointing**
  - To disk
What was easy?

- Load balancing (synchronous)

```java
// at the end of the timestep...
if (doLoadBalance && timestepCounter % loadBalanceInterval == 0) {
    serial {
        // Do the actual load rebalancing
        this->AtSync();
    }
    // Called when load balancing is completed (required)
    when ResumeFromSync() { } 
}
```

- Checkpointing
  - To disk: CkStartCheckpoint(...)

What was easy?

- Load balancing (synchronous)

  ```
  // at the end of the timestep...
  if (doLoadBalance && timestepCounter % loadBalanceInterval == 0) {
    serial {
      // Do the actual load rebalancing
      this->AtSync();
    }
    // Called when load balancing is completed (required)
    when ResumeFromSync() { }
  }
  ```

- Checkpointing
  - To disk: CkStartCheckpoint(...)
  - In memory (to partner node)
What was easy?

- **Load balancing (synchronous)**

```cpp
// at the end of the timestep...
if (doLoadBalance && timestepCounter % loadBalanceInterval == 0) {
    serial {
        // Do the actual load rebalancing
        this->AtSync();
    }
    // Called when load balancing is completed (required)
    when ResumeFromSync() { }
}
```

- **Checkpointing**

  - To disk: `CkStartCheckpoint(...)`
  - In memory (to partner node): `CkStartMemCheckpoint(...)`

Both of these were key features we wanted to test in AMT runtimes, both were done essentially on the first day of coding. Both of these require only serialization on the user side.
What was easy?

- Load balancing (synchronous)

```cpp
// at the end of the timestep...
if (doLoadBalance && timestepCounter % loadBalanceInterval == 0) {
    serial {
        // Do the actual load rebalancing
        this->AtSync();
    }
    // Called when load balancing is completed (required)
    when ResumeFromSync() { }
}
```

- Checkpointing
  - To disk: `CkStartCheckpoint(...)`
  - In memory (to partner node): `CkStartMemCheckpoint(...)`
  - Must be done synchronously
What was easy?

- Load balancing (synchronous)

```java
// at the end of the timestep...
if (doLoadBalance && timestepCounter % loadBalanceInterval == 0) {
    serial {
        // Do the actual load rebalancing
        this->AtSync();
    }
    // Called when load balancing is completed (required)
    when ResumeFromSync() { }
}
```

- Checkpointing
  - To disk: `CkStartCheckpoint(...)`
  - In memory (to partner node): `CkStartMemCheckpoint(...)`
    - Must be done synchronously

Both of these were key features we wanted to test in AMT runtimes, both were done essentially on the first day of coding
What was easy?

- Load balancing (synchronous)

```java
// at the end of the timestep...
if (doLoadBalance && timestepCounter % loadBalanceInterval == 0) {
    serial {
        // Do the actual load rebalancing
        this->AtSync();
    }
    // Called when load balancing is completed (required)
    when ResumeFromSync() { }
}
```

- Checkpointing

  - To disk: `CkStartCheckpoint(...)`
  - In memory (to partner node): `CkStartMemCheckpoint(...)`
  - Must be done synchronously

- Both of these were key features we wanted to test in AMT runtimes, both were done essentially on the first day of coding

Both of these require only serialization on the user side
MPI \implies Charm++ is relatively straightforward

Just like in MPI, data dependencies are expressed in terms of messages:

- Sends become message-like function calls on proxies of array members.
- Receives become when clauses.

```c
class OldStuffDoer {
    /* ... */
    void do_stuff() {
        generate data();
        /* ... */
        MPI_Irecv(data, n_send, MPI_DOUBLE, partner, /*...*/);
        MPI_Send(other_data, n_recv, MPI_DOUBLE, partner, /*...*/);
        use_other_data();
    }
};
```

```c
array[1D] NewStuffDoer {
    entry void receive_data(int src, int ndata, double data[ndata]);
    entry void do_stuff_1() {
        generate data();
        /* ... */
        thisProxy[partner].receive_data(n_send, data);
    }
    entry void do_stuff_2() {
        when receive_data(int partner, int n_send, double * other_data) {
            serial {
                memcpy(other_data_, data, n_send * sizeof(double));
                use_other_data();
            }
        }
    }
};
```

May 7, 2015
“Quick start” implementation: map one chare to one MPI process
“Quick start” implementation: map one chare to one MPI process
- Just like in MPI, data dependencies are expressed in terms of messages:
MPI → Charm++ is relatively straightforward

- "Quick start" implementation: map one chare to one MPI process
  - Just like in MPI, data dependencies are expressed in terms of messages:
    - sends become message-like function calls on proxies of array members
“Quick start” implementation: map one chare to one MPI process

- Just like in MPI, data dependencies are expressed in terms of messages:
  - sends become message-like function calls on proxies of `array` members
  - receives become `when` clauses
MPI ⇒ Charm++ is relatively straightforward

“Quick start” implementation: map one chare to one MPI process
- Just like in MPI, data dependencies are expressed in terms of messages:
  - sends become message-like function calls on proxies of \texttt{array} members
  - receives become \texttt{when} clauses

```cpp
class OldStuffDoer {
    /* ... */
    void do_stuff() {
        generate data();
        /* ... */
        MPI_Irecv(data, n_send,
                   MPI_DOUBLE, partner, /*...*/);
        MPI_Send(other_data, n_recv,
                  MPI_DOUBLE, partner, /*...*/);
        use_other_data();
    }
};
```
MPI ⇒ Charm++ is relatively straightforward

- “Quick start” implementation: map one chare to one MPI process
  - Just like in MPI, data dependencies are expressed in terms of messages:
    - sends become message-like function calls on proxies of array members
    - receives become when clauses

```cpp
class OldStuffDoer {
    /* ... */
    void do_stuff() {
        generate data();
        /* ... */
        MPI_Irecv(data, n_send,
                   MPI_DOUBLE, partner,
                   /*...*/);
        MPI_Send(other_data, n_recv,
                  MPI_DOUBLE, partner,
                  /*...*/);
        use_other_data();
    }
}

array [1D] NewStuffDoer {
    entry void receive_data(int src, int ndata, double data[ndata]);
    entry void do_stuff_1() {
        generate data();
        /* ... */
        thisProxy[partner].receive_data(
            n_send, data);
    }
    entry void do_stuff_2() {
        when receive_data(int partner, int n_send, double* other_data)
            serial {
                memcpy(other_data_, data,
                       n_send * sizeof(double));
                use_other_data();
            }
    }
};
```

May 7, 2015
“Quick start” implementation: map one chare to one MPI process

- Just like in MPI, data dependencies are expressed in terms of messages:
  - Sends become message-like function calls on proxies of array members
  - Receives become when clauses

```
class OldStuffDoer {
    /* ... */
    void do_stuff() {
        generate data();
        /* ... */
        MPI_Irecv(data, n_send, MPI_DOUBLE, partner, /*...*/);
        MPI_Send(other_data, n_recv, MPI_DOUBLE, partner, /*...*/);
        use_other_data();
    }
};
```

```
// Gotchas:

array [1D] NewStuffDoer {
    entry void receive_data(int src, int ndata, double data[ndata]);
    void receive_data(
        int src, int ndata, double data[ndata]);
    entry void do_stuff_1() {
        generate data();
        /* ... */
        thisProxy[partner].receive_data(n_send, data);
    }
    entry void do_stuff_2() {
        when receive_data(int partner, int n_send, double* other_data) {
            serial {
                memcpy(other_data_, data, n_send * sizeof(double));
                use_other_data();
            }
        }
    }
};
```

May 7, 2015
MPI ⇒ Charm++ is relatively straightforward

- “Quick start” implementation: map one chare to one MPI process
  - Just like in MPI, data dependencies are expressed in terms of messages:
    - sends become message-like function calls on proxies of array members
    - receives become when clauses

```
class OldStuffDoer {
    /* ... */
    void do_stuff() {
        generate data();
        /* ... */
        MPI_Irecv(data, n_send, MPI_DOUBLE, partner, /*...*/);
        MPI_Send(other_data, n_recv, MPI_DOUBLE, partner, /*...*/);
        use_other_data();
    }
};
```

```
array [1D] NewStuffDoer {
    entry void receive_data(int src, int ndata, double data[ndata]) {
        thisProxy[partner].receive_data(n_send, data);
    }
    entry void do_stuff_1() {
        generate data();
        /* ... */
        thisProxy[partner].receive_data(n_send, data);
    }
    entry void do_stuff_2() {
        when receive_data(int partner, int n_send, double* other_data) serial {
            memcpy(other_data_, data, n_send * sizeof(double));
            use_other_data();
        }
    }
};
```

“Gotchas”:
- static variables
- conditional communication
- a lot of size and metadata communication
- setup can be skipped
MPI ⇒ Charm++ is relatively straightforward

- "Quick start" implementation: map one chare to one MPI process
  - Just like in MPI, data dependencies are expressed in terms of messages:
    - sends become message-like function calls on proxies of array members
    - receives become when clauses

```
class OldStuffDoer {
    /* ... */
    void do_stuff() {
        generate data();
        /* ... */
        MPI_Irecv(data, n_send,
                  MPI_DOUBLE, partner,
                  /*...*/);
        MPI_Send(other_data, n_recv,
                  MPI_DOUBLE, partner,
                  /*...*/);
        use_other_data();
    }
};
```

```
array [1D] NewStuffDoer {
    entry void receive_data(int src, int ndata, double data[ndata]);

    entry void do_stuff_1() {
        generate data();
        /* ... */
        thisProxy[partner].receive_data(
            n_send, data);
    }

    entry void do_stuff_2() {
        when receive_data(int partner, int n_send, double* other_data) {
            serial {
                memcpy(other_data_, data,
                       n_send * sizeof(double));
                use_other_data();
            }
        };
    }
};
```

"Gotchas":
- static variables
- conditional communication
MPI ⇒ Charm++ is relatively straightforward

- “Quick start” implementation: map one chare to one MPI process
  - Just like in MPI, data dependencies are expressed in terms of messages:
    - sends become message-like function calls on proxies of array members
    - receives become when clauses

```cpp
class OldStuffDoer {
    /* ... */
    void do_stuff() {
        generate data();
        /* ... */
        MPI_Irecv(data, n_send,
                    MPI_DOUBLE, partner,
                    /*...*/);
        MPI_Send(other_data, n_recv, 
                  MPI_DOUBLE, partner, /*...*/);
        use_other_data();
    }
};
```

```
array [1D] NewStuffDoer {
    entry void receive_data(int src, 
                            int ndata, double data[ndata]);
    entry void do_stuff_1() {
        generate data();
        /* ... */
        thisProxy[partner].receive_data(n_send, data);
    }
    entry void do_stuff_2() {
        when receive_data(int partner, 
                          int n_send, double* other_data) 
        serial {
            memcpy(other_data_, data, 
                   n_send * sizeof(double));
            use_other_data();
        }
    }
};
```

“Gotchas”:
- static variables
- conditional communication
- a lot of size and metadata communication setup can be skipped
MPI ⇒ Charm++ is relatively straightforward …at first!

- Is this the best approach for our workloads, or does it lead to unnecessary synchronizations being left over from the MPI version?
Is this the best approach for our workloads, or does it lead to unnecessary synchronizations being left over from the MPI version?

Two approaches:
MPI ⇒ Charm++ is relatively straightforward …at first!

- Is this the best approach for our workloads, or does it lead to unnecessary synchronizations being left over from the MPI version?

- Two approaches:
  - “Bottom up”: Map sends and receives to function calls and
  - “Top down”: Think about task structure and dependencies of code, write this into the .ci file

Clearly, “top down” approach will lead to better, more efficient code in most cases, but…

For production code, a complete “top down” overhaul is completely impractical

Is there a good middle ground?

“Bottom up”-ness vs. “top down”-ness of approach should be assessed before writing too much code (in any porting project)
MPI ⇒ Charm++ is relatively straightforward ...at first!

- Is this the best approach for our workloads, or does it lead to unnecessary synchronizations being left over from the MPI version?
- Two approaches:
  - “Bottom up”: Map sends and receives to function calls and
  - “Top down”: Think about task structure and dependencies of code, write this into the .ci file

Clearly, “top down” approach will lead to better, more efficient code in most cases, but…

For production code, a complete “top down” overhaul is completely impractical

May 7, 2015
Is this the best approach for our workloads, or does it lead to unnecessary synchronizations being left over from the MPI version?

Two approaches:

- “Bottom up”: Map sends and receives to function calls
- “Top down”: Think about task structure and dependencies of code, write this into the .ci file

Clearly, “top down” approach will lead to better, more efficient code in most cases, but...
MPI ➞ Charm++ is relatively straightforward ...at first!

- Is this the best approach for our workloads, or does it lead to unnecessary synchronizations being left over from the MPI version?

- Two approaches:
  - “Bottom up”: Map sends and receives to function calls and when
  - “Top down”: Think about task structure and dependencies of code, write this into the .ci file

- Clearly, “top down” approach will lead to better, more efficient code in most cases, but...

- For production code, a complete “top down” overhaul is completely impractical
Is this the best approach for our workloads, or does it lead to unnecessary synchronizations being left over from the MPI version?

Two approaches:
- “Bottom up”: Map sends and receives to function calls and
- “Top down”: Think about task structure and dependencies of code, write this into the .ci file

Clearly, “top down” approach will lead to better, more efficient code in most cases, but...

For production code, a complete “top down” overhaul is completely impractical

Is there a good middle ground?
MPI \(\Rightarrow\) Charm++ is relatively straightforward \textit{...at first!}

- Is this the best approach for our workloads, or does it lead to unnecessary synchronizations being left over from the MPI version?

- Two approaches:
  - “Bottom up”: Map sends and receives to function calls and when's
  - “Top down”: Think about task structure and dependencies of code, write this into the .ci file

- Clearly, “top down” approach will lead to better, more efficient code in most cases, but...

- For production code, a complete “top down” overhaul is completely impractical

- Is there a good middle ground?
  - “Bottom up”-ness vs. “top down”-ness of approach should be assessed before writing too much code (in any porting project)
What was harder: Kokkos integration and templated code

Kokkos is a performance portability layer aimed primarily at on-node parallelism. It handles memory layout and loop structure to produce optimized kernels on multiple devices. Application developer implements generic code, Kokkos library implements device-specific specializations.

MiniAero was originally written in "MPI+Kokkos". What happens when you need to write templated code that uses Kokkos?

Explicitly listing all specializations can get out of hand quickly. For instance:

```cpp
template <typename Device>
struct ddot {
  const Kokkos::View<Device>& A, B;
  double result;

ddot(const Kokkos::View<Device>& A_in, const Kokkos::View<Device>& B_in) : A(A_in), B(B_in), result(0) {}

inline void operator() (int i) {
  result += A(i) * B(i);
}

void do_stuff() {
  /* ... */
  Kokkos::parallel_for(num_items, ddot<Kokkos::Cuda>(v1, v2));
}
```
What was harder: Kokkos integration and templated code

- Kokkos is a performance portability layer aimed primarily at on-node parallelism.
What was harder: Kokkos integration and templated code

- Kokkos is a performance portability layer aimed primarily at on-node parallelism
  - handles memory layout and loop structure to produce optimized kernels on multiple devices
Kokkos is a performance portability layer aimed primarily at on-node parallelism
- handles memory layout and loop structure to produce optimized kernels on multiple devices

Application developer implements generic code, Kokkos library implements device-specific specializations

What was harder: Kokkos integration and templated code

May 7, 2015
What was harder: Kokkos integration and templated code

- Kokkos is a performance portability layer aimed primarily at on-node parallelism
  - handles memory layout and loop structure to produce optimized kernels on multiple devices

- Application developer implements generic code, Kokkos library implements device-specific specializations

```cpp
template <typename Device>
struct ddot {
  const Kokkos::View<Device>& A, B;
  double result;

  ddot(
    const Kokkos::View<Device>& A_in,
    const Kokkos::View<Device>& B_in
  ) : A(A_in), B(B_in), result(0) {
  }

  inline void operator() (int i) {
    result += A(i) * B(i);
  }
};

void do_stuff() {
  /* ... */
  Kokkos::parallel_for(
    num_items,
    ddot<Kokkos::Cuda>(v1, v2)
  );
}
```
What was harder: Kokkos integration and templated code

- Kokkos is a performance portability layer aimed primarily at on-node parallelism
  - handles memory layout and loop structure to produce optimized kernels on multiple devices
- Application developer implements generic code, Kokkos library implements device-specific specializations
- MiniAero was originally written in “MPI+Kokkos”

```cpp
1 template <typename Device>
2 struct ddot {
3   const Kokkos::View<Device>& A, B;
4   double result;
5
6   ddot(
7       const Kokkos::View<Device>& A_in,
8       const Kokkos::View<Device>& B_in
9   ) : A(A_in), B(B_in), result(0) {
10  }
11
12   inline void operator()(int i) {
13       result += A(i) * B(i);
14   }
15  
16  void do_stuff() {
17     /* ... */
18     Kokkos::parallel_for(
19       num_items,
20       ddot<Kokkos::Cuda>(v1, v2)
21     );
22  }
```
What was harder: Kokkos integration and templated code

Kokkos is a performance portability layer aimed primarily at on-node parallelism
- handles memory layout and loop structure to produce optimized kernels on multiple devices

Application developer implements generic code, Kokkos library implements device-specific specializations

MiniAero was originally written in “MPI+Kokkos”

What happens when you need to write templated code that uses Kokkos?

```
template <typename Device>
struct ddot {
    const Kokkos::View<Device>& A, B;
    double result;

    ddot(
        const Kokkos::View<Device>& A_in,
        const Kokkos::View<Device>& B_in
    ) : A(A_in), B(B_in), result(0) {
    }

    inline void operator()(int i) {
        result += A(i) * B(i);
    }
};

void do_stuff() {
    /* ... */
    Kokkos::parallel_for(
        num_items,
        ddot<Kokkos::Cuda>(v1, v2)
    );
}
```
What was harder: Kokkos integration and templated code

- Kokkos is a performance portability layer aimed primarily at on-node parallelism
  - handles memory layout and loop structure to produce optimized kernels on multiple devices
- Application developer implements generic code, Kokkos library implements device-specific specializations
- MiniAero was originally written in “MPI+Kokkos”
- What happens when you need to write templated code that uses Kokkos?
  - Explicitly listing all specializations can get out of hand quickly. For instance...

```cpp
template <typename Device>
struct ddot {
  const Kokkos::View<Device>& A, B;
  double result;

  ddot(
    const Kokkos::View<Device>& A_in,
    const Kokkos::View<Device>& B_in
  ) : A(A_in), B(B_in), result(0) {
  }

  inline void operator() (int i) {
    result += A(i) * B(i);
  }
};

void do_stuff() {
  Kokkos::parallel_for(
    num_items,
    ddot<Kokkos::Cuda>(v1, v2)
  );
}
```

May 7, 2015
The MiniAero solver has five different ghost exchanges.
The MiniAero solver has five different ghost exchanges. Each communicates a different \texttt{Kokkos::View} type.
The MiniAero solver has five different ghost exchanges.
Each communicates a different `Kokkos::View` type, so we want an entry method prototype that looks something like this:
Template specialization explosion

- The MiniAero solver has five different ghost exchanges.
- Each communicates a different `Kokkos::View` type, so we want an entry method prototype that looks something like this:

```cpp
1 template <typename viewType>
2 entry [local] void receive_ghost_data(viewType& v);
```
The MiniAero solver has five different ghost exchanges.

Each communicates a different Kokkos::View type, so we want an entry method prototype that looks something like this:

```cpp
template <typename ViewType>
entry [local] void receive_ghost_data(ViewType& v);
```

The solver chare is already parameterized on the Kokkos device type:
The MiniAero solver has five different ghost exchanges. Each communicates a different Kokkos::View type, so we want an entry method prototype that looks something like this:

```cpp
1 template <typename ViewType>
2 entry [local] void receive_ghost_data(ViewType& v);
```

The solver chare is already parameterized on the Kokkos device type:

```cpp
/* solver.ci */
1 template <typename Device>
2 array [1D] RK4Solver {
3    /* ... */
4 };
```
Template specialization explosion

- The MiniAero solver has five different ghost exchanges.
- Each communicates a different `Kokkos::View` type, so we want an entry method prototype that looks something like this:

```cpp
template <typename ViewType>
entry [local] void receive_ghost_data(ViewType& v);
```

- The solver chare is already parameterized on the Kokkos device type:

```cpp
/* solver.ci */
template <typename Device>
class RK4Solver {
    /* ... */
};

/* solver.h */
template <typename Device>
class RK4Solver :
    public CBase_RK4Solver<Device>
{
    Kokkos::View<Device,double*,5> m_data1;
    Kokkos::View<Device,double*,5,3> m_data2;
    Kokkos::View<Device,int*> m_data3;
    /* etc... */
};
```
Template specialization explosion

- The MiniAero solver has five different ghost exchanges.
- Each communicates a different `Kokkos::View` type, so we want an entry method prototype that looks something like this:

```cpp
template <typename ViewType>
entry [local] void receive_ghost_data(ViewType& v);
```

- The solver chare is already parameterized on the Kokkos device type:

```cpp
/* solver.ci */
template <typename Device>
class RK4Solver {
    array [1D] RK4Solver {
        /* ... */
    };
}
```

- The devices we'd like to test include `Kokkos::Serial`, `Kokkos::Threads`, `Kokkos::Cuda`, and `Kokkos::OpenMP`
Template specialization explosion

■ The MiniAero solver has five different ghost exchanges.
■ Each communicates a different `Kokkos::View` type, so we want an entry method prototype that looks something like this:

```cpp
template <typename ViewType>
entry [local] void receive_ghost_data(ViewType& v);
```

■ The solver chare is already parameterized on the Kokkos device type:

```cpp
/* solver.ci */
template <typename Device>
class RK4Solver {
  /* ... */
};
```

■ The devices we'd like to test include `Kokkos::Serial`, `Kokkos::Threads`, `Kokkos::Cuda`, and `Kokkos::OpenMP`
■ That already leads to 20 different explicit signatures for receive_ghost_data().

May 7, 2015
Template specialization: our workaround

- Pattern: templated setup, non-templated entry method, templated cleanup
Template specialization: our workaround

- Pattern: templated setup, non-templated entry method, templated cleanup

```cpp
/* comm_stuff.h */
template <typename Device>
class CommStuffDoer : public CBase_CommStuffDoer<Device> {
    Kokkos::View<Device, double*, 3> my_data_1_;  // Templated view for device-specific double data
    Kokkos::View<Device, int*, 3, 5> my_data_2_; // Templated view for device-specific int data
    /* ... */
};
```
Template specialization: our workaround

**Pattern:** templated setup, non-templated entry method, templated cleanup

```cpp
/* comm_stuff.h */
template <typename Device>
class CommStuffDoer :
public CBase_CommStuffDoer<Device>
{
  Kokkos::View<Device, double*, 3> my_data_1_;  
  Kokkos::View<Device, int*, 3, 5> my_data_2_;  
  /* ... */
};
```

```cpp
/* comm_stuff.ci */
template <typename Device>
array [1D] CommStuffDoer {
  entry void recv_it(int src, int size, double data[size]);
  entry void do_recv_done();
  entry [local] void do_recv(int src) {
    when recv_it[src](int s, int size, double data[size])
    {  
      memset(recv_buffers[src], data, size*sizeof(double));
      do_recv_done();
    }
  }
  entry void do_stuff() {
    /* ... */
    when do_recv_done()
    {  
      finish_recv(src, my_data_1_);
    }
  }
};
```
Template specialization: our workaround

- Pattern: templatized setup, non-templatized entry method, templatized cleanup

/* comm_stuff.h */
template <typename Device>
class CommStuffDoer : public CBase_CommStuffDoer<Device>
{
    Kokkos::View<Device,double*,3> my_data_1_
    Kokkos::View<Device,int*,3,5> my_data_2_
    /* ... */
}

/* comm_stuff.ci */
template <typename Device>
array [1D] CommStuffDoer {
    entry void do_stuff() {
    
    };
Template specialization: our workaround

Pattern: templated setup, non-templated entry method, templated cleanup

/* comm_stuff.h */
template <typename Device>
class CommStuffDoer : 
  public CBase_CommStuffDoer<Device>
{
  Kokkos::View<Device,double*,3> my_data_1_; 
  Kokkos::View<Device,int*,3,5> my_data_2_; 
  /* ... */
};

/* comm_stuff.ci */
template <typename Device>
array [1D] CommStuffDoer {

entry void do_stuff() {
  /* ... */
  serial {
    int src = /*...*/ , dest = /*...*/ ;
    send_it(dest, my_data_1_ );

  }

};
Template specialization: our workaround

Pattern: templated setup, non-templated entry method, templated cleanup

```cpp
/* comm_stuff.h */
template <typename Device>
class CommStuffDoer :
    public CBase_CommStuffDoer<Device>
{
    Kokkos::View<Device,double*,3> my_data_1_;
    Kokkos::View<Device,int*,3,5> my_data_2_; /* ... */
}

/* comm_stuff.ci */
template <typename Device>
array [1D] CommStuffDoer {

    entry void do_stuff() {
        /* ... */
        serial {
            serial {
                int src = /*...*/, dest = /*...*/;
                send_it(dest, my_data_1_);
                setup_recv(src, my_data_1_);
                /* ... */
            }
        }
    }
};
```
Template specialization: our workaround

Pattern: templated setup, non-templated entry method, templated cleanup

/* comm_stuff.h */
template <typename Device>
class CommStuffDoer :
    public CBase_CommStuffDoer<Device>
{
    Kokkos::View<Device,double*,3> my_data_1_;  
    Kokkos::View<Device,int*,3,5> my_data_2_;  
    /* ... */
};

/* comm_stuff.ci */
template <typename Device>
array [1D] CommStuffDoer {

    entry void do_stuff() {
        /* ... */
        serial {
            serial {
                int src = /*...*/, dest = /*...*/;
                send_it(dest, my_data_1_);
                setup_recv(src, my_data_1_);
                do_recv(src);
            }
        }
    }
};
Template specialization: our workaround

Pattern: templated setup, non-templated entry method, templated cleanup

/* comm_stuff.h */

```cpp
template <typename Device>
class CommStuffDoer :
    public CBase_CommStuffDoer<Device>
{
    Kokkos::View<Device,double*,3> my_data_1_;  
    Kokkos::View<Device,int*,3,5> my_data_2_;  
    /* ... */
};
```

/* comm_stuff.ci */

```cpp
template <typename Device>
array [1D] CommStuffDoer {

    entry void do_recv_done();

    entry void do_stuff() {
        /* ... */
        serial {
            int src = /*...*/, dest = /*...*/;
            send_it(dest, my_data_1_);
            setup_recv(src, my_data_1_);
            do_recv(src);
        }
        when do_recv_done() serial {
        }
    }
};
```

May 7, 2015
Template specialization: our workaround

- Pattern: templated setup, non-templated entry method, templated cleanup

```cpp
/* comm_stuff.h */
template <typename Device>
class CommStuffDoer :
  public CBase_CommStuffDoer<Device>
{
  Kokkos::View<Device,double*,3> my_data_1_;  
  Kokkos::View<Device,int*,3,5> my_data_2_;  
  /* ... */
};

/* comm_stuff.ci */
template <typename Device>
array [1D] CommStuffDoer {

  entry void do_recv_done();

  entry void do_stuff() {
    /* ... */
    serial {
      int src = /*...*/, dest = /*...*/;
      send_it(dest, my_data_1_);
      setup_recv(src, my_data_1_);
      do_recv(src);
    }
    when do_recv_done() serial {
      finish_recv(src, my_data_1_);
    }
  }
};
```
Template specialization: our workaround

- Pattern: templated setup, non-templated entry method, templated cleanup

```cpp
/* comm_stuff.h */
template <typename Device>
class CommStuffDoer : public CBase_CommStuffDoer<Device>
{
    Kokkos::View<Device,double*,3> my_data_1_;  
    Kokkos::View<Device,int*,3,5> my_data_2_;  
} /* ... */

    template <typename ViewT>
    void send_it(int dst, const ViewT& data) {
        size_t size = get_size(data, dst); 
        double* data = extract_data(data, dst); 
        this->thisProxy[dst].recv_it(
            this->thisIndex, size, data); 
    }

/* comm_stuff.ci */
template <typename Device>
array [1D] CommStuffDoer {
    entry void recv_it(int src, 
                      int size, double data[size]); 
    entry void do_recv_done(); 

    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_stuff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff() { 
    entry void do_staff()
```
Template specialization: our workaround

- Pattern: templated setup, non-templated entry method, templated cleanup

```cpp
/* comm_stuff.h */
template <typename Device>
class CommStuffDoer : public CBase_CommStuffDoer<Device>
{
    Kokkos::View<Device,double*,3> my_data_1_;  
    Kokkos::View<Device,int*,3,5> my_data_2_;  

    std::vector<double*> recv_buffers_;  

template <typename ViewT>
void send_it(int dst, const ViewT& data) {
    size_t size = get_size(data, dst);  
    double* data = extract_data(data, dst);  
    this->thisProxy[dst].recv_it(
        this->thisIndex, size, data);  
}

template <typename ViewT>
void setup_recv(int src, ViewT& data) {
    recvBuffers_[src] = get_buffer(data, src);  
}

};

/* comm_stuff.ci */
template <typename Device>
array [1D] CommStuffDoer {
    entry void recv_it(int src, 
        int size, double data[size]);  
    entry void do_recv_done();

    entry void do_stuff() {  
        /* ... */
        serial {
            int src = /*...*/, dest = /*...*/;  
            send_it(dest, my_data_1_);  
            setup_recv(src, my_data_1_); 
            do_recv(src);  
        }  
        when do_recv_done() serial {
            finish_recv(src, my_data_1_);  
        }
    }
};
```
Template specialization: our workaround

- Pattern: templated setup, non-templated entry method, templated cleanup

```cpp
/* comm_stuff.h */
template <typename Device>
class CommStuffDoer :
public CBase_CommStuffDoer<Device>
{
Kokkos::View<Device,double*,3> my_data_1_;  
Kokkos::View<Device,int*,3,5> my_data_2_;  
/* ... */  
std::vector<double*> recv_buffers_;  
template <typename ViewT>
void send_it(int dst, const ViewT& data) {
size_t size = get_size(data, dst);  
double* data = extract_data(data, dst);  
this->thisProxy[dst].recv_it(  
this->thisIndex, size, data);  
}

template <typename ViewT>
void setup_recv(int src, ViewT& data) {
recv_buffers_[src] =  
get_buffer(data, src);  
}

template <typename ViewT>
void finish_recv(int src, ViewT& data) {
insert_data(data, recv_buffers_[src], src);  
delete recv_buffers_[src];  
}
};

/* comm_stuff.ci */
template <typename Device>
array [1D] CommStuffDoer {
entry void recv_it(int src,  
int size, double data[size]);  
entry void do_recv_done();  
}

entry void do_stuff() {
/* ... */  
serial {
int src = /*...*/;  
send_it(dest, my_data_1_);  
send_it(dest, my_data_1_);  
setup_recv(src, my_data_1_);  
do_recv(src);  
}
when do_recv_done() serial {
finish_recv(src, my_data_1_);  
}
};
```
Template specialization: our workaround

- Pattern: templated setup, non-templated entry method, templated cleanup

```cpp
/* comm_stuff.h */
template <typename Device>
class CommStuffDoer :
  public CBase_CommStuffDoer<Device>
{
  Kokkos::View<Device,double*,3> my_data_1_;  
  Kokkos::View<Device,int*,3,5> my_data_2_;  
  /* ... */
  std::vector<double*> recv_buffers_;   
  template <typename ViewT>
  void send_it(int dst, const ViewT& data) {
    size_t size = get_size(data, dst);  
    double* data = extract_data(data, dst); 
    this->thisProxy[dst].recv_it(       
      this->thisIndex, size, data); 
  }  
  template <typename ViewT>
  void setup_recv(int src, ViewT& data) {
    recv_buffers_[src] = get_buffer(data, src); 
  }  
  template <typename ViewT>
  void finish_recv(int src, ViewT& data) {
    insert_data(data, recv_buffers_[src], src);  
    delete recv_buffers_[src]; 
  }
};

/* comm_stuff.ci */
template <typename Device>
array [1D] CommStuffDoer {
  entry void recv_it(int src,    
    int size, double data[size]);
  entry void do_recv_done();
  entry [local] void do_recv(int src) {
    when recv_it[src](int s, int size,    
      double data[size]) serial { 
      memcpy(recv_buffers[src], data,    
        size*sizeof(double)); 
      do_recv_done();
    }
  }
};

entry void do_stuff() {
  /* ... */
  serial {
    int src = /*...*/ , dest = /*...*/;
    send_it(dest, my_data_1_); 
    setup_recv(src, my_data_1_); 
    do_recv(src);
  }
  when do_recv_done() serial {
    finish_recv(src, my_data_1_);
  }
};
```
Template specialization: our workaround

Pattern: templated setup, non-templated entry method, templated cleanup

/* comm_stuff.h */

```cpp
template <typename Device>
class CommStuffDoer :
    public CBase_CommStuffDoer<Device>
{
    Kokkos::View<Device, double*, 3> my_data_1;
    Kokkos::View<Device, int*, 3, 5> my_data_2;
    /* ... */
    std::vector<double*> recv_buffers_; 

    template <typename ViewT>
    void send_it(int dst, const ViewT& data) {
        size_t size = get_size(data, dst);
        double* data = extract_data(data, dst);
        this->thisProxy[dst].recv_it(
            this->thisIndex, size, data);
    }

    template <typename ViewT>
    void setup_recv(int src, ViewT& data) {
        recv_buffers_[src] = get_buffer(data, src);
    }

    template <typename ViewT>
    void finish_recv(int src, ViewT& data) {
        insert_data(data, recv_buffers_[src], src);
        delete recv_buffers_[src];
    }
};
```

/* comm_stuff.ci */

```cpp
template <typename Device>
array [1D] CommStuffDoer {
    entry void recv_it(int src, int size, double data[size]);
    entry void do_recv_done();
    entry [local] void do_recv(int src) {
        when recv_it[src](int s, int size, double data[size]) serial {
            memcpy(recv_buffers[src], data, size*sizeof(double));
            do_recv_done();
        }
    }

    entry void do_stuff() {
        /* ... */
        serial {
            int src = /*...*/, dest = /*...*/;
            send_it(dest, my_data_1_);
            setup_recv(src, my_data_1_);
            do_recv(src);
        }
        when do_recv_done() serial {
            finish_recv(src, my_data_1_);
        }
    }
};
```

Is this ideal? Obviously not

May 7, 2015
Pattern: templated setup, non-templated entry method, templated cleanup

```cpp
/* comm_stuff.h */
template <typename Device>
class CommStuffDoer :
  public CBase_CommStuffDoer<Device> {
  
    Kokkos::View<Device,double*,3> my_data_1_; 
    Kokkos::View<Device,int*,3,5> my_data_2_; 
    
    std::vector<double*> recv_buffers_; 

    template <typename ViewT>
    void send_it(int dst) {
      size_t size = get_size(data, dst);
      double* data = extract_data(data, dst);

      this->thisProxy[dst].recv_it(
        this->thisIndex, size, data);
    } 

    template <typename ViewT>
    void setup_recv(int src, ViewT& data) {
      recv_buffers_[src] =
        get_buffer(data, src);
    }

    template <typename ViewT>
    void finish_recv(int src, ViewT& data) {
      insert_data(data, recv_buffers_[src], src);
      delete recv_buffers_[src];
    }
  };

/* comm_stuff.ci */
template <typename Device>
array [1D] CommStuffDoer {
  
  entry void recv_it(int src, 
                   int size, double data[size]);
  entry void do_recv_done();
  entry [local] void doRecv(int src) {
    when recv_it[src](int s, int size, 
                   double* data) serial {

      memcpy(recv_buffers[src], data,
                    size*sizeof(double));

      do_recv_done();
    }
  }

  entry void do_stuff() {
  
    int src = /*...*/, dest = /*...*/;
    send_it(dest, my_data_1_);
    setup_recv(src, my_data_1_);
    do_recv(src);

    when do_recv_done() serial {

      finish_recv(src, my_data_1_);
    }
  }
};

May 7, 2015
```

Is this typical of the effort required to make templated code work with an asynchronous many-task runtime system (AMT RTS)?

*Maybe*
Does Charm++ even support templated entry methods inside templated chares? *(We couldn't figure out how to do it)*
Distinguishing Entry Methods from Regular Method Calls

Suppose all of the `do_stuff_*()` methods are ordinary, non-entry methods. What happens first?

Now suppose `do_stuff_1()` is an entry method and `do_stuff_2()` is a normal method. Now what happens first?

How does the programmer who didn't write `do_stuff_1()` know this? Perhaps using naming conventions? (e.g., `EM_*()`)

```
1 entry void do_stuff() {
2     serial {
3         do_stuff_1();
4         do_stuff_2();
5     }
6 }

1 entry void EM_do_stuff() {
2     serial {
3         EM_do_stuff_1();
4         do_stuff_2();
5     }
6 }
```

In short, mixing entry method calls and regular method calls without using naming conventions makes it difficult to write self-documenting code.

May 7, 2015
Distinguishing Entry Methods from Regular Method Calls

Suppose all of the do_stuff_*() methods are ordinary, non-entry methods. What happens first?

Now suppose do_stuff_1() is an entry method and do_stuff_2() is a normal method. Now what happens first?

How does the programmer who didn’t write do_stuff_1() know this?

Perhaps using naming conventions? (e.g., EM_*())

```
entry void do_stuff() {
  serial {
    do_stuff_1();
    do_stuff_2();
  }
};

entry void EM_do_stuff() {
  serial {
    EM_do_stuff_1();
    do_stuff_2();
  }
};
```

In short, mixing entry method calls and regular method calls without using naming conventions makes it difficult to write self-documenting code.
Suppose all of the `do_stuff_*()` methods are ordinary, non-entry methods.

```java
entry void do_stuff() {
    serial {
        do_stuff_1();
        do_stuff_2();
    }
};
```

In short, mixing entry method calls and regular method calls without using naming conventions makes it difficult to write self-documenting code.
Suppose all of the `do_stuff_*( )` methods are ordinary, non-entry methods.

- What happens first?

```java
entry void do_stuff() {
    serial {
        do_stuff_1();
        do_stuff_2();
    }
};
```

In short, mixing entry method calls and regular method calls without using naming conventions makes it difficult to write self-documenting code.
Distinguishing Entry Methods from Regular Method Calls

- Suppose all of the `do_stuff_*()` methods are ordinary, non-entry methods.
  - What happens first?
- Now suppose `do_stuff_1()` is an entry method and `do_stuff_2()` is a normal method.

```plaintext
entry void do_stuff() {
    serial {
        do_stuff_1();
        do_stuff_2();
    }
}
```

In short, mixing entry method calls and regular method calls without using naming conventions makes it difficult to write self-documenting code.
Distinguishing Entry Methods from Regular Method Calls

- Suppose all of the `do_stuff_*()` methods are ordinary, non-entry methods.
  - What happens first?

- Now suppose `do_stuff_1()` is an entry method and `do_stuff_2()` is a normal method.
  - Now what happens first?

```java
event void do_stuff() {
  serial {
    do_stuff_1();
    do_stuff_2();
  }
}
```

In short, mixing entry method calls and regular method calls without using naming conventions makes it difficult to write self-documenting code.

May 7, 2015
Distinguishing Entry Methods from Regular Method Calls

- Suppose all of the `do_stuff_*( )` methods are ordinary, non-entry methods.
  - What happens first?

- Now suppose `do_stuff_1( )` is an entry method and `do_stuff_2( )` is a normal method.
  - Now what happens first?

- How does the programmer who didn't write `do_stuff_1( )` know this?

```java
1  entry void do_stuff() {
2     serial {
3         do_stuff_1();
4         do_stuff_2();
5     }
6 }
```
Suppose all of the `do_stuff_*( )` methods are ordinary, non-entry methods.

- What happens first?

Now suppose `do_stuff_1( )` is an entry method and `do_stuff_2( )` is a normal method.

- Now what happens first?

How does the programmer who didn't write `do_stuff_1( )` know this?

- Perhaps using naming conventions? (e.g., `EM_*()`)
Distinguishing Entry Methods from Regular Method Calls

Suppose all of the `do_stuff_*()` methods are ordinary, non-entry methods.
- What happens first?

Now suppose `do_stuff_1()` is an entry method and `do_stuff_2()` is a normal method.
- Now what happens first?

How does the programmer who didn't write `do_stuff_1()` know this?
- Perhaps using naming conventions? (e.g., `EM_*()`)

In short, mixing entry method calls and regular method calls without using naming conventions makes it difficult to write self-documenting code.
Further complication: non-blocking calls from a blocking context
Further complication: non-blocking calls from a blocking context

```c
/* stuff_doer.ci */
char StuffDoer {
  entry void EM_do_stuff () {
    serial {
      EM_do_stuff_1();
      do_stuff_2();
      do_stuff_3();
    }
  }
};
```

```c
/* stuff_doer.h */
class StuffDoer :
  public CBase_StuffDoer {

  /*...*/
  void do_stuff_2 () {
    // uh-oh
    thisProxy.EM_do_stuff_4();
  }
};
```
Further complication: non-blocking calls from a blocking context

In fact, `do_stuff_2()` may only be blocking *most* or the time, but occasionally contain non-blocking calls

---

```c
/* stuff_doer.ci */
chare StuffDoer {
  entry void EM_do_stuff() {
    serial {
      EM_do_stuff_1();
      do_stuff_2();
      do_stuff_3();
    }
  }
};

/* stuff_doer.h */
class StuffDoer :
  public CBase_StuffDoer {
    /*...*/
    void do_stuff_2() {
      // uh-oh
      thisProxy.EM_do_stuff_4();
    }
  }
};
```
Distinguishing Entry Methods from Regular Method Calls

- Further complication: non-blocking calls from a blocking context
- In fact, `do_stuff_2()` may only be blocking *most* or the time, but occasionally contain non-blocking calls

```c
/* stuff_doer.ci */
char StuffDoer {
    entry void EM_do_stuff() {
        serial {
            EM_do_stuff_1();
            do_stuff_2();
            do_stuff_3();
        }
    }
};

/* stuff_doer.h */
class StuffDoer : public CBase_StuffDoer {
    /*...*/
    void do_stuff_2() {
        if(some_rare_condition) {
            thisProxy.EM_do_stuff_4();
        }
        /*...*/
    }
};
```
Distinguishing Entry Methods from Regular Method Calls

- Further complication: non-blocking calls from a blocking context
- In fact, `do_stuff_2()` may only be blocking *most* or the time, but occasionally contain non-blocking calls
- In this case, how does the programmer make the control flow of the program apparent to future programmers?

```c
/* stuff_doer.ci */
chare StuffDoer {
  entry void EM_do_stuff() {
    serial {
      EM_do_stuff_1();
      do_stuff_2();
      do_stuff_3();
    }
  }
}

/* stuff_doer.h */
class StuffDoer : public CBase_StuffDoer {
  /*...*/
  void do_stuff_2() {
    if(some_rare_condition) {
      thisProxy.EM_do_stuff_4();
    }
    /* ... */
  }
};
```
Distinguishing Entry Methods from Regular Method Calls

- Further complication: non-blocking calls from a blocking context

- In fact, `do_stuff_2()` may only be blocking *most* or the time, but occasionally contain non-blocking calls

- In this case, how does the programmer make the control flow of the program apparent to future programmers?
  - Avoid writing code like this?

```c
chare StuffDoer {
  entry void EM_do_stuff() {
    serial {
      EM_do_stuff_1();
      do_stuff_2();
      do_stuff_3();
    }
  }
};
```

```c
class StuffDoer : public CBase_StuffDoer {
  /*...*/
  void do_stuff_2() {
    if(some_rare_condition) {
      thisProxy.EM_do_stuff_4();
    } /* ... */
  }
};
```
Further complication: non-blocking calls from a blocking context

In fact, `do_stuff_2()` may only be blocking most of the time, but occasionally contain non-blocking calls

In this case, how does the programmer make the control flow of the program apparent to future programmers?

- Avoid writing code like this?
- Avoid naming conventions?
  Makes the programmer “get used to” the idea that any method invocation in a `.ci` file could be non-blocking

```c
/* stuff_doer.ci */
chare StuffDoer {
  entry void EM_do_stuff() {
    serial {
      EM_do_stuff_1();
      do_stuff_2();
      do_stuff_3();
    }
  }
};

/* stuff_doer.h */
class StuffDoer :
  public CBase_StuffDoer {
    /*...*/
    void do_stuff_2() {
      if(some_rare_condition) {
        thisProxy.EM_do_stuff_4();
      }
      /* ... */
    }
    /*...*/
  };
```
Distinguishing Entry Methods from Regular Method Calls

- Further complication: non-blocking calls from a blocking context
  - In fact, `do_stuff_2()` may only be blocking *most* or the time, but occasionally contain non-blocking calls
- In this case, how does the programmer make the control flow of the program apparent to future programmers?
  - Avoid writing code like this?
  - Avoid naming conventions?
    - Makes the programmer “get used to” the idea that *any* method invocation in a `.ci` file could be non-blocking
  - Just use comments?

```c
/* stuff_doer.ci */
chare StuffDoer {

    entry void EM_do_stuff() {
        serial {
            EM_do_stuff_1();
            do_stuff_2();
            do_stuff_3();
        }
    }
};

/* stuff_doer.h */
class StuffDoer : public CBase_StuffDoer {

    /*...*/

    void do_stuff_2() {
        if(some_rare_condition) {
            thisProxy.EM_do_stuff_4();
        }
        /* ... */
    }
};
```
Introduction

The Process: Porting an Explicit Aerodynamics Miniapp to the Chare Model
  What was easy?
  What was harder?

Preliminary Results and Performance

Next Steps
Performance vs. MPI Version: Weak Scaling

MiniAero Ramp (small) 50 time steps

MiniAero FlatPanel (50 timesteps)
Performance: Overdecomposition and Runtime Overhead

256 Chares on 128 PEs

1024 Chares on 128 PEs

Application code in green, runtime overhead in red, idle time in white
(Insets are enlargements of y-axes)

May 7, 2015
Performance: Overdecomposition and Runtime Overhead

Application code in green, runtime overhead in red, idle time in white
(Insets are enlargements of y-axes)
Introduction

The Process: Porting an Explicit Aerodynamics Miniapp to the Chare Model

What was easy?
What was harder?

Preliminary Results and Performance

Next Steps
Next Steps

- More performance analysis
- PAPI?
- More Kokkos devices (Cuda?)
- More miniapps
Next Steps

- More performance analysis
Next Steps

- More performance analysis
  - PAPI?
Next Steps

- More performance analysis
  - PAPI?
- More Kokkos devices (Cuda?)
Next Steps

- More performance analysis
  - PAPI?
- More Kokkos devices (Cuda?)
- More miniapps
Overdecomposition and Zero-Copy Semantics

The "Charm + X" model:
Charm++:
dynamic, inter-node parallelism
"X": static, on-node parallelism; vectorization

Zero-copy semantics and some shared data model or data warehouse are critical to mitigating the AMT runtime overhead from overdecomposition. Charm++ allows zero copy transfer of data between chares on-node using PackedMessages. But these are an "advanced feature," much more difficult than PUPing. The concept of a shared data block is missing. For instance, PackedMessages have no access privileges (e.g., read only, shared read/write, exclusive read/write).

Dynamic, on-node parallelism arising from overdecomposition...
The “Charm + X” model:

- Charm++: dynamic, inter-node parallelism
- “X”: static, on-node parallelism; vectorization

Zero-copy semantics and some shared data model or data warehouse are critical to mitigating the AMT runtime overhead from overdecomposition. Charm++ allows zero-copy transfer of data between chares on-node using PackedMessages. But these are an “advanced feature,” much more difficult than PUPing. The concept of a shared data block is missing. For instance, PackedMessages have no access privileges (e.g., read-only, shared read/write, exclusive read/write).

Dynamic, on-node parallelism arising from overdecomposition.
The “Charm + X” model:

- Charm++: dynamic, inter-node parallelism
- "X": static, on-node parallelism; vectorization

Zero-copy semantics and some shared data model or data warehouse are critical to mitigating the AMT runtime overhead from overdecomposition. Charm++ allows zero copy transfer of data between chares on-node using PackedMessage. But these are an “advanced feature,” much more difficult than PUPing. The concept of a shared data block is missing. For instance, PackedMessage has no access privileges (e.g., read only, shared read/write, exclusive read/write).
The “Charm + X” model:

- Charm++: dynamic, inter-node parallelism
- "X": static, on-node parallelism; vectorization

Dynamic, on-node parallelism arising from overdecomposition

Zero-copy semantics and some shared data model or data warehouse are critical to mitigating the AMT runtime overhead from overdecomposition
The “Charm + X” model:

- Charm++: dynamic, inter-node parallelism
- “X”: static, on-node parallelism; vectorization

Zero-copy semantics and some shared data model or data warehouse are critical to mitigating the AMT runtime overhead from overdecomposition.

Charm++ allows zero copy transfer of data between chares on-node using PackedMessage s.
The “Charm + X” model:

- Charm++: dynamic, inter-node parallelism
- "X": static, on-node parallelism; vectorization

Zero-copy semantics and some shared data model or data warehouse are critical to mitigating the AMT runtime overhead from overdecomposition.

Charm++ allows zero copy transfer of data between chares on-node using PackedMessage's.

But these are an “advanced feature,” much more difficult than PUP'ing.
Overdecomposition and Zero-Copy Semantics

- The "Charm + X" model:
  - Charm++: dynamic, inter-node parallelism
  - "X": static, on-node parallelism; vectorization

- Zero-copy semantics and some shared data model or data warehouse are critical to mitigating the AMT runtime overhead from overdecomposition

- Charm++ allows zero copy transfer of data between chares on-node using PackedMessage s
  - But these are an "advanced feature," much more difficult than PUP ping

- The concept of a shared data block is missing
The “Charm + X” model:

- Charm++: dynamic, inter-node parallelism
- “X”: static, on-node parallelism; vectorization

Zero-copy semantics and some shared data model or data warehouse are critical to mitigating the AMT runtime overhead from overdecomposition.

Charm++ allows zero copy transfer of data between chares on-node using PackedMessage s.

- But these are an “advanced feature,” much more difficult than PUP ing.

The concept of a shared data block is missing.

- For instance, PackedMessage s have no access privileges (e.g., read only, shared read/write, exclusive read/write)
Other minor issues

- The Charm++ compiler

Would be nice if it could run like a preprocessor to generate code, then regular compiler could be used after that.

SMP version: no way to initialize libraries on main thread?
Other minor issues

- The Charm++ compiler
  - .ci file compiler issues
Other minor issues

- The Charm++ compiler
  - .ci file compiler issues
    - (e.g., } inside C++ style comments not ignored?)
Other minor issues

- The Charm++ compiler
  - .ci file compiler issues
  - (e.g., `}` inside C++ style comments not ignored?)
  - Would be nice if it could run like a preprocessor to generate code, then regular compiler could be used after that.
Other minor issues

- The Charm++ compiler
  - .ci file compiler issues
    - (e.g., } inside C++ style comments not ignored?)
  - Would be nice if it could run like a preprocessor to generate code, then regular compiler could be used after that.

- SMP version: no way to initialize libraries on main thread?
Conclusions

How does the programming experience in Charm++ compare to other runtimes?
Inconclusive so far. Charm++ MiniAero was a port, others were complete rewrites.

How does the performance of Charm++ compare to other runtimes?
Inconclusive so far. Other MiniAero versions are in various states of completeness.

But… our current implementation is already comparable to MPI.
How does the programming experience in Charm++ compare to other runtimes?
Conclusions

- How does the programming experience in Charm++ compare to other runtimes?
  - *Inconclusive* so far. Charm++ MiniAero was a port, others were complete rewrites
Conclusions

- How does the programming experience in Charm++ compare to other runtimes?
  - *Inconclusive* so far. Charm++ MiniAero was a port, others were complete rewrites

- How does the performance of Charm++ compare to other runtimes?
Conclusions

- How does the programming experience in Charm++ compare to other runtimes?
  - *Inconclusive* so far. Charm++ MiniAero was a port, others were complete rewrites

- How does the performance of Charm++ compare to other runtimes?
  - *Inconclusive* so far. Other MiniAero versions are in various states of completeness
Conclusions

- How does the programming experience in Charm++ compare to other runtimes?
  - *Inconclusive* so far. Charm++ MiniAero was a port, others were complete rewrites

- How does the performance of Charm++ compare to other runtimes?
  - *Inconclusive* so far. Other MiniAero versions are in various states of completeness
  - *But...* our current implementation is already comparable to MPI