# Structured Dagger: Supporting Asynchrony with Clarity

Jonathan Lifflander, Laxmikant V. Kale

{jliff12, kale}@illinois.edu

University of Illinois Urbana-Champaign

March 16, 2015

## Motivation

- Parallel programming models are becoming more asynchronous
- To fully exploit asynchrony we need to define the dependencies
  - Describing DAGs in terms of nodes and edges is difficult to program and intuitively understand
  - Describing sequences tends to be more natural to program
- What is Structured Dagger?
  - A scripting language that can describe a subset of DAGs with an implicit ordering
  - Part of the Charm++ ecosystem (in C++)

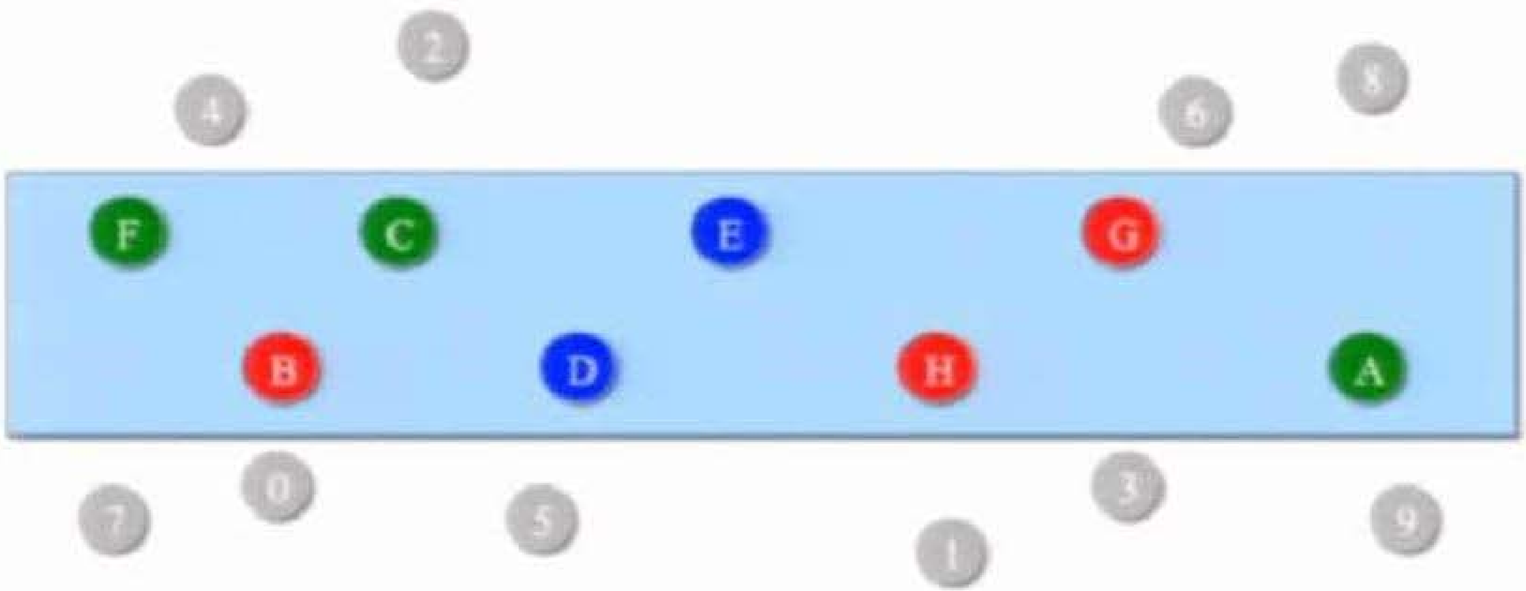
# What is Charm++?

- A parallel runtime system used for scientific applications
  - NAMD: molecular dynamics app for large biomolecular systems
  - OpenAtom: quantum chemistry app (CPAIMD)
  - ChaNGa: n-body app for cosmological simulations
  - EpiSimdemics: contagion app for simulating spread of disease

# What is Charm++?

- The computation is decomposed into objects
  - Objects that are parallel are notated by the user as parallel objects, called **chares**
  - Chares are parallel entities that can migrate between processors
- Some methods for each parallel object are notated as **entry methods**
  - Entry methods can be invoked remotely if you have a proxy to the object
  - When an entry methods is called it causes the method parameters to be packed and sent over the network and then invoked on the other end

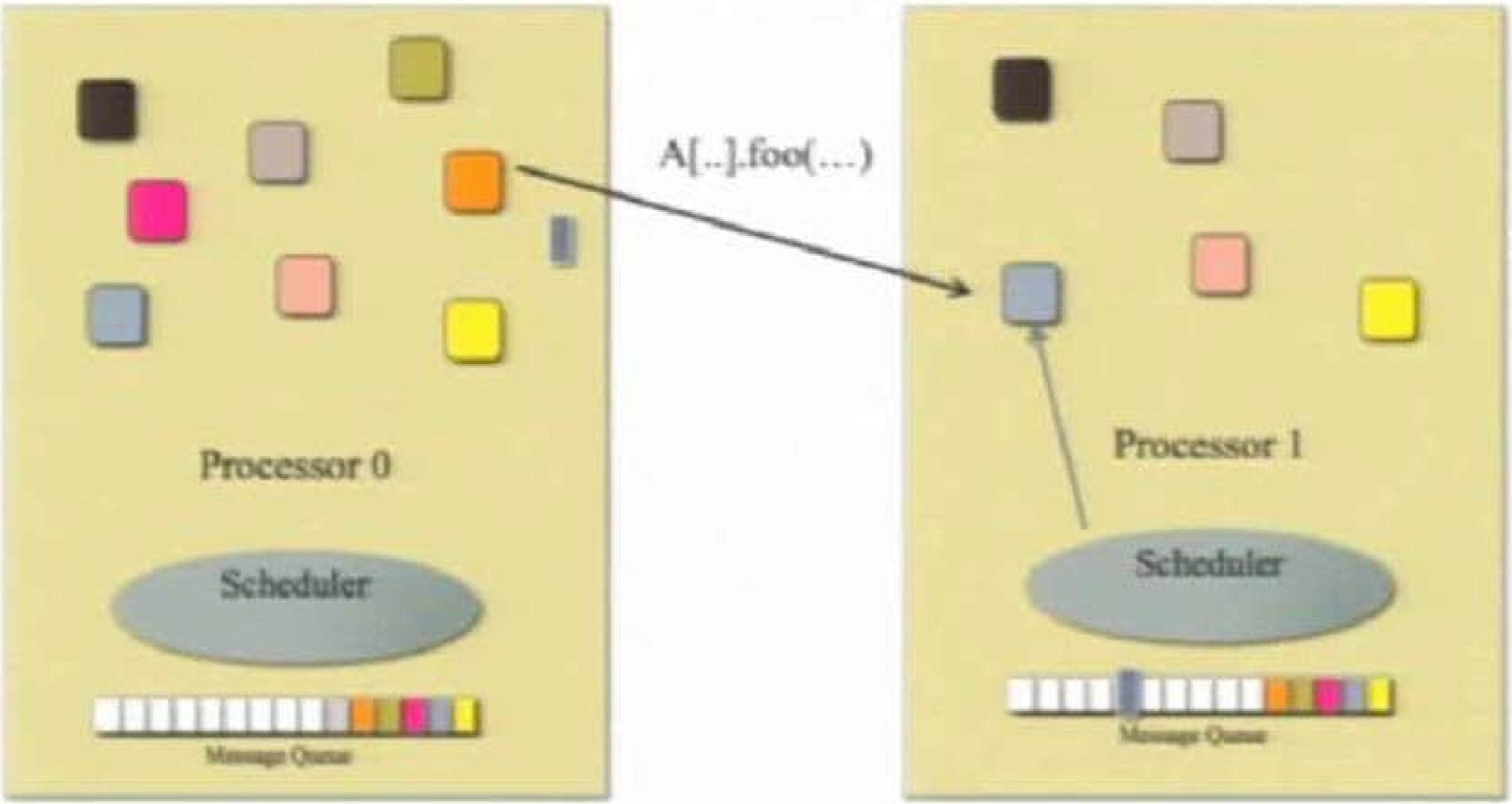
# What is Charm++?



# Charm++ Execution Model

- Several chares will live on a single processor
- As a result:
  - Method invocations directed at chares on that processor will have to be stored in a pool.
  - And a user-level scheduler will select one invocation from the queue and runs it to completion.
- Execution is triggered by availability of a "message" (a method invocation)
- When an entry method executes:
  - It may generate messages for other chares
  - The RTS deposits them in the message queue on the target processor

# What is Charm++?



# Reactive Nature of Method Description

- By just using entry methods, a chare's description is very reactive

# Structured Dagger

- Make implicit dependencies embedded in entry method's behavior more explicit
- Computations depend on remote method invocations, and completion of other local computations
- We assume a sequence by default, and allow the user to override it
- Wait for `A` to complete and then proceed to `B`

```
A
B
```

- If `A` and `B` can execute in any order, this must be explicitly defined using the `overlap` construct, implicit join

```
overlap {
  A
  B
}
```

# Structured Dagger: Waiting for Messages

- The `when` construct is used to wait for a remote method invocation
- The `when` construct allows you match a message using the name of the method and a reference number
- Syntax: `when` methods-to-wait-on block-to-execute

```
when recvData(int size, double data[size]) { /* do something */ }
```

- When a message is matched to a when, the associated block of code executes
- The data from that method is put into scope in the corresponding block

# Structured Dagger: Waiting for Messages

- The `when` construct can be nested

```
when myMethod1(int param1, int param2) {
  when myMethod2(bool param3),
       myMethod3(int size, int arr[size]) /* block1 */
  when myMethod4(bool param4) /* block2 */
}
```

- Sequence defined in the above example
  - Wait for `myMethod1`, upon arrival execute body of `myMethod1`
  - Wait for `myMethod2` and `myMethod3`, upon arrival of both, execute `/* block1 */`
  - Wait for `myMethod4`, upon arrival execute `/* block2 */`
- If messages arrive out of order, the Structured Dagger buffers them

# Fibonacci without Structured Dagger

```
chare Fib {
  int saved_val, response_counter;
  entry void dowork() {
    if (n < THRESHOLD) { respond(seqFib(n)); }
    else { Fib::ckNew(n - 1); Fib::ckNew(n - 2); }
  }
  entry void response(int val) {
    response_counter++;
    if (response_counter == 1) saved_val = val;
    else respond(val + saved_val);
  }
  void respond(int val) {
    if (!isRoot) parent.response(val);
    else printf("Fibonacci number is: %d\n")
  }
}
```

# Fibonacci with Structured Dagger

```
chare Fib {
  entry void dowork() {
    if (n < THRESHOLD) { respond(seqFib(n)); }
    else {
      Fib::ckNew(n - 1); Fib::ckNew(n - 2);
      when response(int val), response(int val2) {respond(val+val2);}
    }
  }
  void respond(int val) {
    if (!isRoot) parent.response(val);
    else printf("Fibonacci number is: %d\n")
  }
}
```

# Structured Dagger: Reference Numbers

- The `when` clause can wait on a certain reference number
- If a reference number is specified for a `when`, the first parameter for the entry method must be the reference number
- The `when` will not be matched until a message arrives with that reference number

```
when method1[100](short ref, bool param1) /* block */
```

Two sends:

```
proxy.method1(200, false); /* will not be delivered to the when */
proxy.method1(100, true); /* will be delivered to the when */
```

# Structured Dagger: More Advanced Constructs

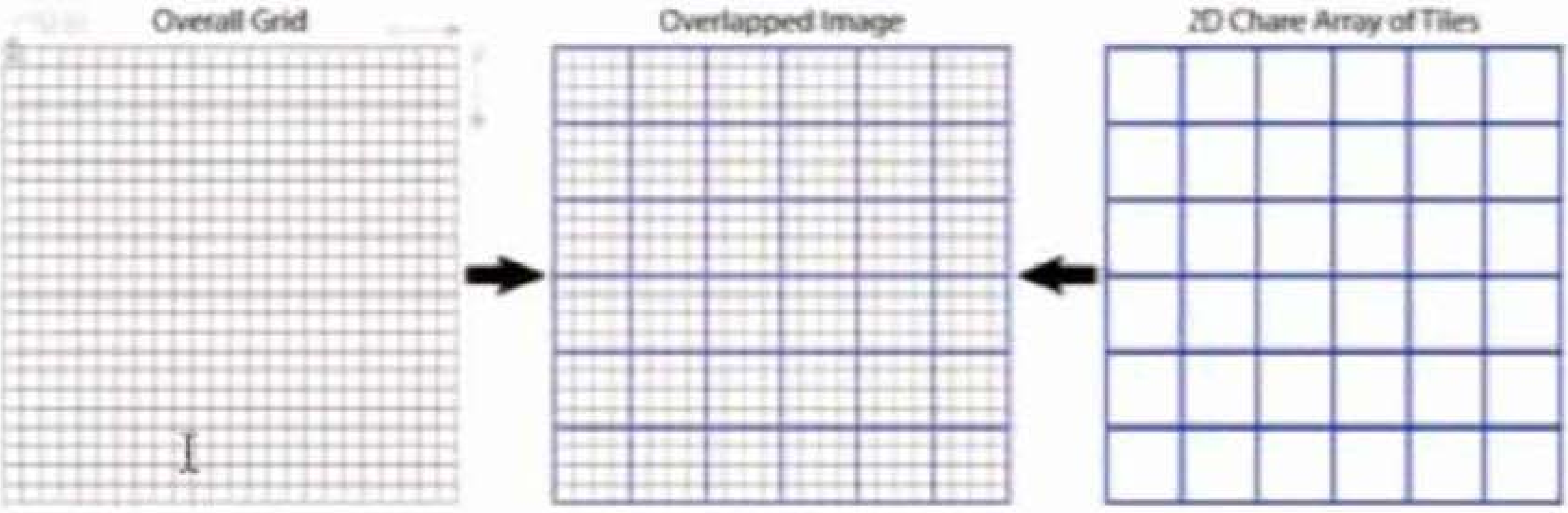
- The language includes `for`, `forall`, `while`
- `for` and `while` are ordered in their iteration space
- `forall` allows the iterations to execute in any order

```
for (iter = 0; iter < maxIter; ++iter) {
  overlap {
    when recvLeft[iter](short num, int len, double data[len]) { computeKernel(LEFT, data); }
    when recvRight[iter](short num, int len, double data[len]) { computeKernel(RIGHT, data); }
  }
}
```

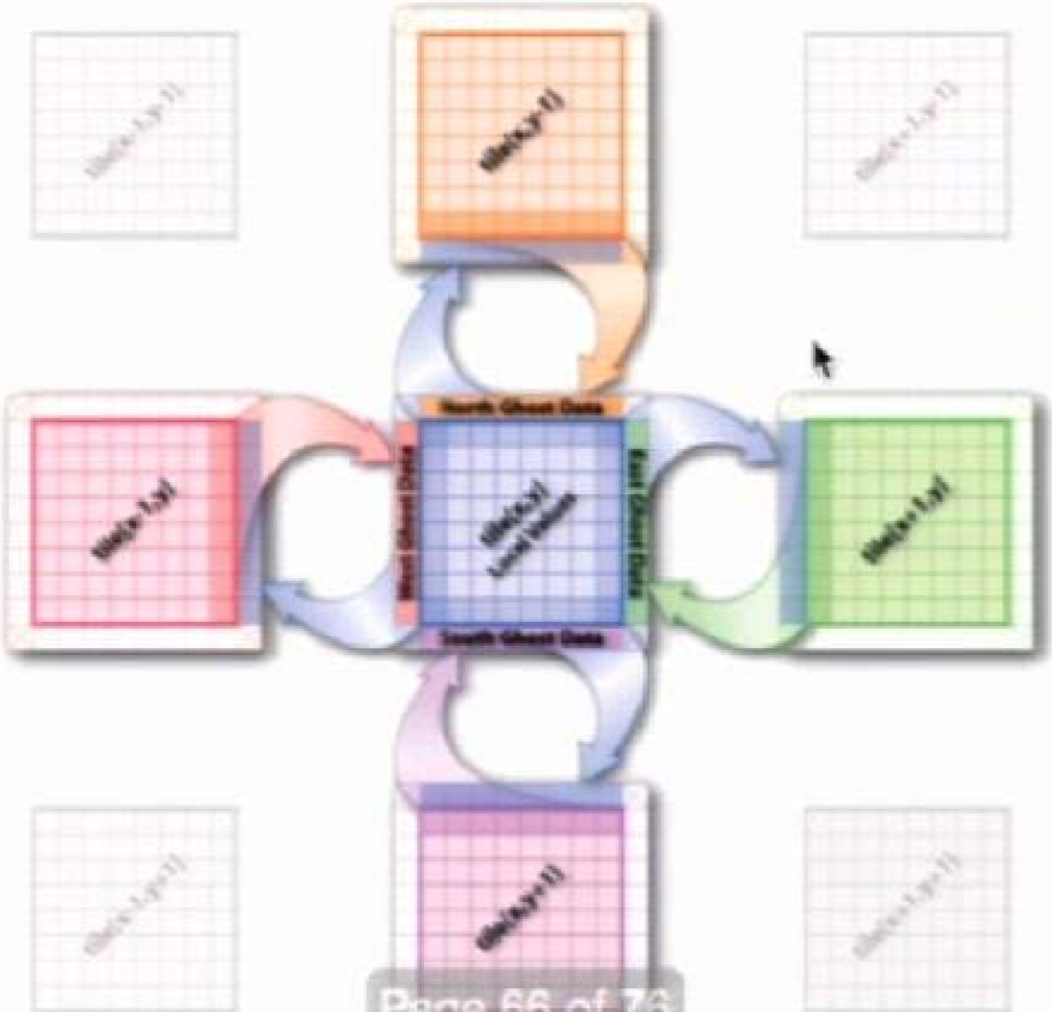
```
while (i < numNeighbors) {
  when recvData(int len, double data[len]) { /* execute kernel */ }
  i++;
}
```

```
forall [block] (0 : numBlocks - 1, 1) {
  when method1[block](short ref, bool someVal) /* code block1 */
}
```

# Example Program: Stencil 3D (Jacobi)



# Example Program: Stencil 3D (Jacobi)



Page 66 of 76

# Example Program: Stencil 3D (Jacobi)

```
while (!converged) {
  copyToBoundaries();
  sendToNeighbors();
  freeBoundaries();
  for (remoteCount = 0; remoteCount < 6; remoteCount++)
    when updateGhosts[iter](int ref, int dir, int w, int h, double buf[w*h]) {
      updateBoundary(dir, w, h, buf);
    }
  double error = computeKernel();
  int conv = error < DELTA;
  if (iter % 5 == 1)
    contribute_async_reduction(conv, logical_and, checkConverged);
  if (++iter % 5 == 0)
    when checkConverged(bool result)
      if (result) { mainChare.done(iter); converged = true; }
}
```

Page 67 of 76

# Conclusion

- Structured Dagger is the result of many years of research
  - Charm++ started with Dagger, which allowed any DAG to be specified and had not assumed order
  - We found that Dagger was difficult and error prone to program
- Structured Dagger makes DAGs natural to express
- Lessons learned
  - We may need to limit expressibility to some extent for better programmability
  - Most DAGs can be expressed in Structured Dagger, but not all
  - For many scientific programs implemented in Charm++, we've found that St[illegible]gger is sufficient

Questions?