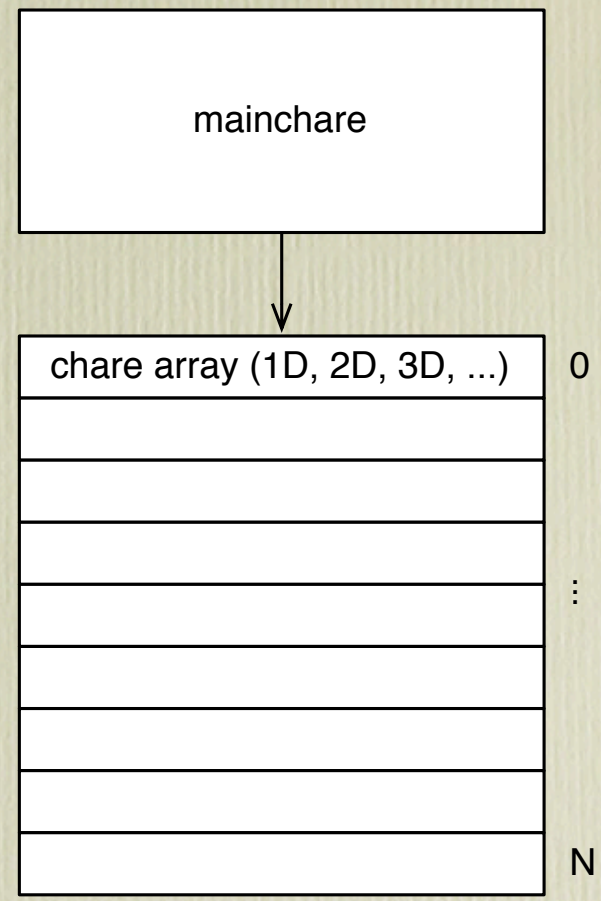
Charm++ Objects

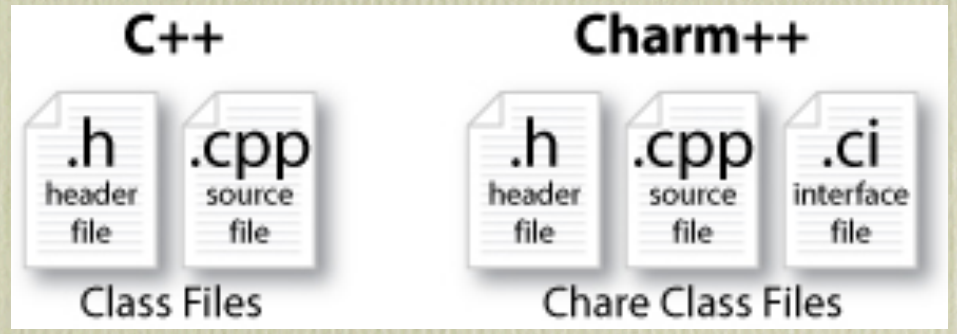
- A “chare” is a C++ object with methods that can be remotely invoked
- The “mainchare” is the chare where the execution starts in the program
- A “chare array” is a collection of chares of the same type
- Typically the mainchare will spawn a chare array of workers

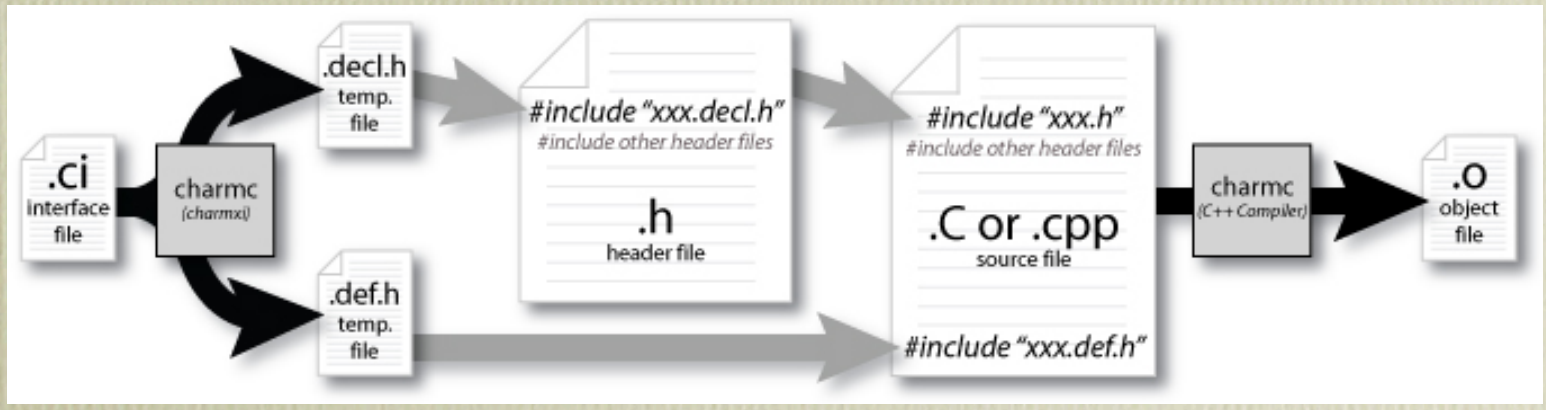


Charm++ File Structure

- The C++ objects (whether they are chares or not)
  - Reside in regular .h and .cpp files
- Chare objects, messages, message coordination, and entry methods (methods that can be called asynchronously and remotely)
  - Are defined in a .ci (Charm interface) file
  - And are implemented in the .cpp file

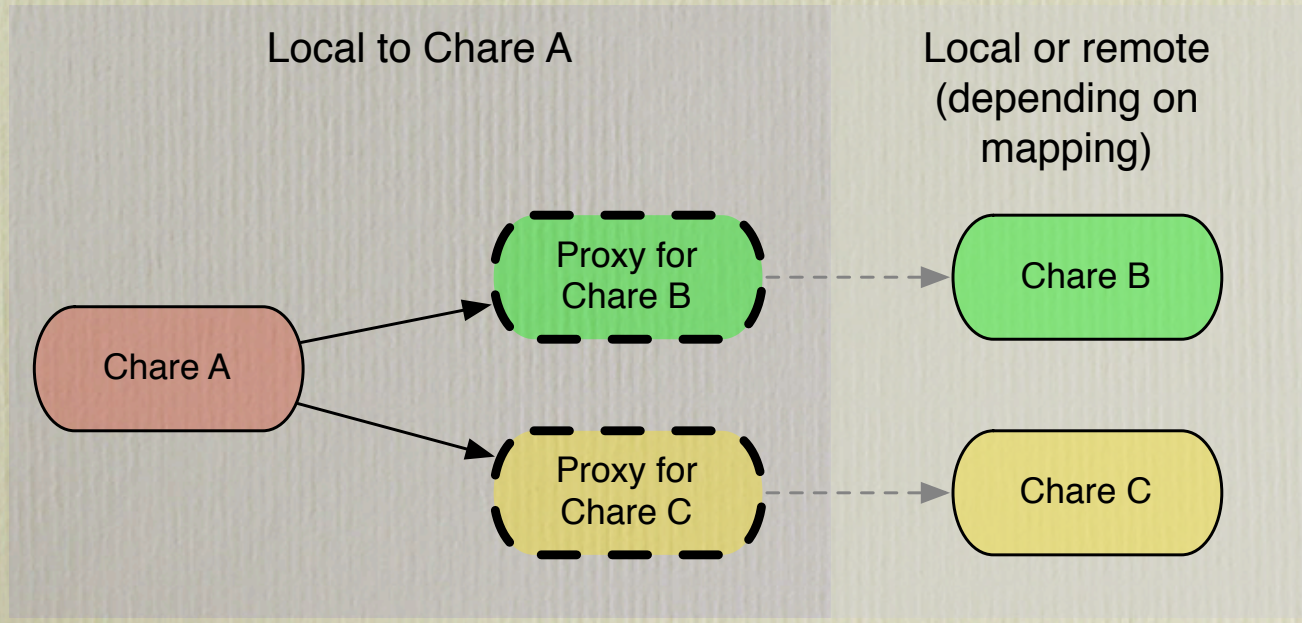
Charm++ File Structure





Charm++ Proxy Objects

- Proxy objects are handles for invoking entry methods on other chares
- These chares may be local or remote depending on the mapping



Charm++ Hello World

- Hello world program asynchronous flow
  - Mainchare sends message to Hello object
  - Hello object prints “Hello World!”
  - Hello object sends message back to the mainchare
  - Mainchare quits the application

Charm++ Hello World

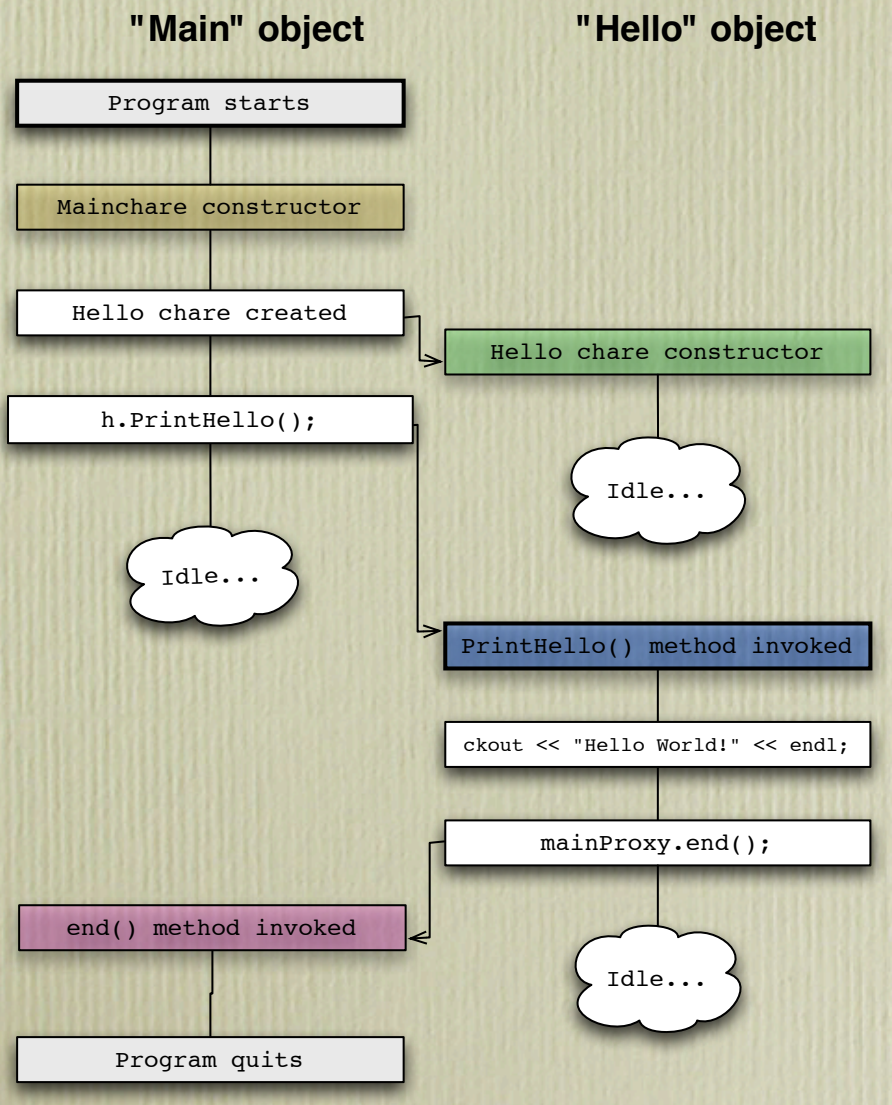
hello.ci

```
mainmodule hello {
  readonly CProxy_Main mainProxy;

  mainchare Main {
    entry Main(CkArgMsg*);
    entry void end(void);
  };

  chare Hello {
    entry Hello();
    entry void PrintHello(void);
  }
};
```

hello.cpp

```
#include "hello.decl.h"

/*readonly*/ CProxy_Main mainProxy;

class Main : public Chare {
public:
  Main(CkArgMsg* m) {
    delete m;
    mainProxy = thishandle;

    CProxy_Hello h = CProxy_Hello::ckNew();
    h.PrintHello();
  }

  void end() {
    CkExit();
  }
};

class Hello : public CBase_Hello {
public:
  Hello() {}

  void PrintHello(void) {
    ckout << "Hello World!" << endl;
    mainProxy.end();
  }
};

#include "hello.def.h"
```

Charm++ Hello World




```
#include "hello.decl.h"

/*readonly*/ CProxy_Main mainProxy;

class Main : public Chare {
public:
  Main(CkArgMsg* m) {
    delete m;
    mainProxy = thishandle;

    CProxy_Hello h = CProxy_Hello::ckNew();
    h.PrintHello();
  }

  void end() {
    CkExit();
  }
};

class Hello : public CBase_Hello {
public:
  Hello() {}

  void PrintHello(void) {
    ckout << "Hello World!" << endl;
    mainProxy.end();
  }
};

#include "hello.def.h"
```

Charm++ Asynchronous Methods

- Does this code produce the desired output?

```
class Main : public CBase_Main {
public:
  Main(CkArgMsg* m) {
    double pi = 3.1415;
    CProxy_Simple sim = CProxy_Simple::ckNew(pi);

    for (int i = 1; i < 10; i++)
      sim.findArea(i, false);

    sim.findArea(10, true);
  };
};

class Simple : public CBase_Simple {
private:
  float y;

public:
  Simple(double pi) {
    y = pi;
    ckout << "Hello from a simple chare running on " << CkMyPe() << endl;
  }

  void findArea(int r, bool done) {
    ckout << "Area of a circle of radius " << r << " is " << y*r*r <<
endl;

    if (done)
      CkExit();
  }
};
```

```
mainmodule simple {
  mainchare Main {
    entry Main(CkArgMsg *m);
  };

  chare Simple {
    entry Simple(double y);
    entry void findArea(int radius, bool);
  };
};
```

Charm++ Asynchronous Methods

- Maybe, however the program might terminate after printing some number of messages, because the order is unknown
- This is probably not the desired output

```
class Main : public CBase_Main {
public:
  Main(CkArgMsg* m) {
    double pi = 3.1415;
    CProxy_Simple sim = CProxy_Simple::ckNew(pi);

    for (int i = 1; i < 10; i++)
      sim.findArea(i, false);

    sim.findArea(10, true);
  };
};

class Simple : public CBase_Simple {
private:
  float y;

public:
  Simple(double pi) {
    y = pi;
    ckout << "Hello from a simple chare running on " << CkMyPe() << endl;
  }

  void findArea(int r, bool done) {
    ckout << "Area of a circle of radius " << r << " is " << y*r*r <<
endl;

    if (done)
      CkExit();
  }
};
```

The order that these `findArea(...)` messages are received is not guaranteed or known

`CkExit()` will terminate the program on all processors

Charm++ Chare Arrays

```
mainmodule hello {
  readonly CProxy_Main mainProxy;
  readonly int nElements;

  mainchare Main {
    entry Main(CkArgMsg *m);
    entry void done(void);
  };

  array [1D] Hello {
    entry Hello(void);
    entry void SayHi(int hiNo);
  };
};
```

Charm++ Chare Arrays

```
#include <stdio.h>
#include "hello.decl.h"

/*readonly*/ CProxy_Main mainProxy;
/*readonly*/ int nElements;

/*mainchare*/
class Main : public CBase_Main {
public:
  Main(CkArgMsg* m) {
    //Process command-line arguments
    nElements=5;
    if(m->argc >1 ) nElements=atoi(m->argv[1]);
    delete m;

    //Start the computation
    ckout << "Running Hello on " << CkNumPes() << " processors
for " << nElements << " elements" << endl;

    mainProxy = thisProxy;

    CProxy_Hello arr = CProxy_Hello::ckNew(nElements);

    arr[0].SayHi(17);
  };

  void done(void) {
    ckout << "All done" << endl;
    CkExit();
  };
};
```

⋮

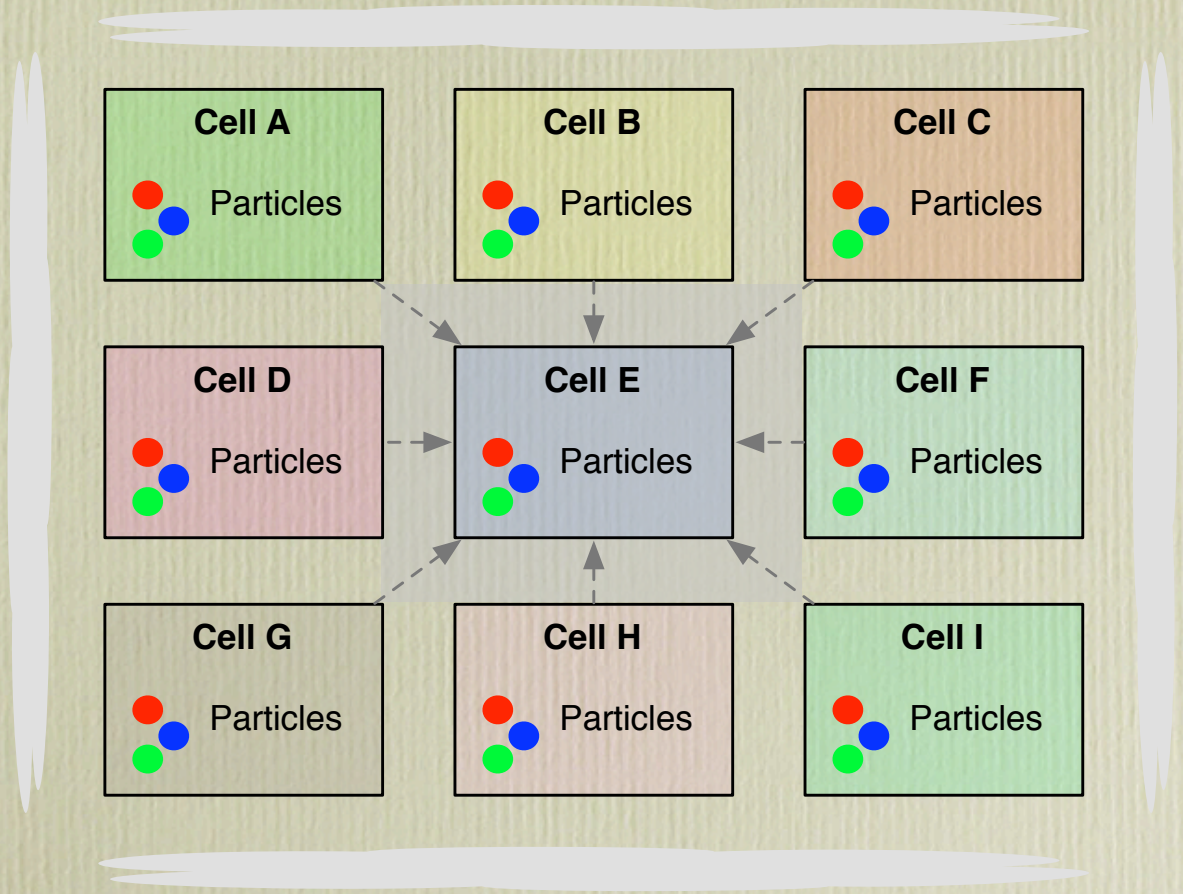
⋮

```
/*array [1D]*/
class Hello : public CBase_Hello {
public:
  Hello() {
    ckout << "Hello " << thisIndex << " created"
<< endl;
  }

  Hello(CkMigrateMessage *m) {}

  void SayHi(int hiNo) {
    ckout << "Hi[" << hiNo << "] from element "
<< thisIndex << endl;
    if (thisIndex < nElements-1)
      //Pass the hello on:
      thisProxy[thisIndex+1].SayHi(hiNo+1);
    else
      //We've been around once-- we're done.
      mainProxy.done();
  }
};

#include "hello.def.h"
```

Charm++ Van der Waals Example

- Basic Idea
  - 2D Van der Waals simulation
  - 2D world broken into Cells
  - Each Cell is a chare, which will be mapped to a processor
  - Each Cell has a collection of particles
    - Each Cell communicates with its neighbors to compute particle interactions and update its particles
  - World is wrapped around

Charm++ Van der Waals Example



"Cell E" updates particles based on remote and local interactions

Charm++ Van der Waals Example

vanderwaals.ci

```
mainmodule parallel {
  readonly CProxy_Main mainProxy;
  readonly int numIterations;
  readonly int numParticles;

  mainchare Main {
    entry Main(CkArgMsg*);
    entry void finishInit();
    entry void end();
  };

  array [2D] Cell {
    entry Cell();
    entry void initialize(int);
    entry void sendAll(int);
    entry void migrateParts(int);
    entry void stepMine(void);
    entry void process(void) {
      // ... Cell's lifecycle ...
    };
    entry void updateParticles(int t0,
                               CkVec<Particle> &parts);
    entry void tradeParticles(int t0,
                              CkVec<Particle> &parts);
  };
};
```

Broadcasts `process()` to Cell array

Charm++ Broadcasts

- "arr" is a 2D chare array
- Broadcasts are a simple way to invoke the same method on every element of the array

```
class Main : public Chare {
public:
  CProxy_Cell arr;

  Main(CkArgMsg* m) {
     // ...
     arr = CProxy_Cell::ckNew(nrows, ncols);
     // ...
  }

  void finishInit(void) {
    arr.process();
  }

  // ...

};
```

Charm++ Van der Waals Example

vanderwaals.ci

```
mainmodule parallel {
  readonly CProxy_Main mainProxy;
  readonly int numIterations;
  readonly int numParticles;

  mainchare Main {
    entry Main(CkArgMsg*);
    entry void finishInit();
    entry void end();
  };

  array [2D] Cell {
    entry Cell();
    entry void initialize(int);
    entry void sendAll(int);
    entry void migrateParts(int);
    entry void stepMine(void);
    entry void process(void) {
      // ... Cell's lifecycle ...
    };
    entry void updateParticles(int t0,
                               CkVec<Particle> &parts);
    entry void tradeParticles(int t0,
                              CkVec<Particle> &parts);
  };
};
```

Broadcasts `process()` to Cell array

Calls `CkExit()`

Charm++ Van der Waals Example

vanderwaals.ci

```
mainmodule parallel {
  readonly CProxy_Main mainProxy;
  readonly int numIterations;
  readonly int numParticles;

  mainchare Main {
    entry Main(CkArgMsg*);
    entry void finishInit();
    entry void end();
  };

  array [2D] Cell {
    entry Cell();
    entry void initialize(int);
    entry void sendAll(int);
    entry void migrateParts(int);
    entry void stepMine(void);
    entry void process(void) {
      // ... Cell's lifecycle ...
    };
    entry void updateParticles(int t0,
                               CkVec<Particle> &parts);
    entry void tradeParticles(int t0,
                              CkVec<Particle> &parts);
  };
};
```

Broadcasts `process()` to Cell array

Calls `CkExit()`

Initializes random particles, performs reduction back to `finishInit()`

`initialize()` Method

- The next step should not proceed until all the chares in the array have finished initialization

```
void initialize(int size) {
  double init_range = InitRange * CellSize;

  centerx = CellSize * thisIndex.x + 0.5 * CellSize;
  centery = CellSize * thisIndex.y + 0.5 * CellSize;

  // Randomly initialize particles to be some distance from the
center of the square
  for (unsigned i = 0; i < size; i++) {
    srand48(thisIndex.x * 10 + thisIndex.y * 1000 + i + size * 20);

    // ...

    Particle p(drand48()*1000000, vx, vy, nrows, ncols, max_vel);

    p.x = centerx + (drand48() - 0.5) * init_range;
    p.y = centery + (drand48() - 0.5) * init_range;
    p.mass = drand48() * 500 + 300;
    particles.push_back(p);
  }

  contribute(CkCallback(CkIndex_Main::finishInit(), mainProxy));
}
```

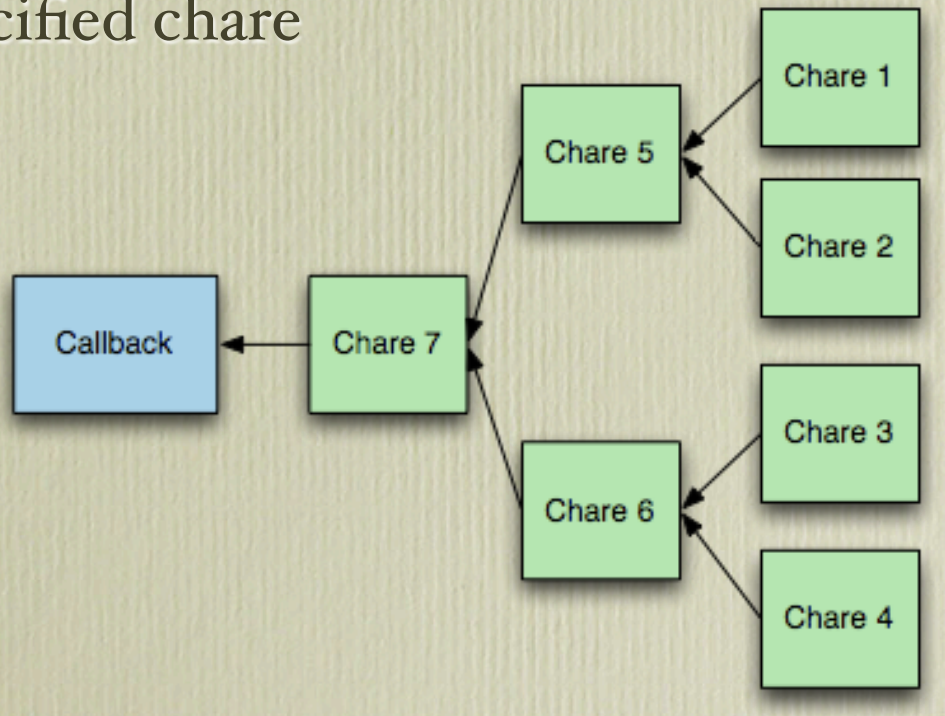
Charm++ Reductions

- No global flow of control, so each chare must contribute with the `contribute()` function
- A user callback (created using `CkCallBack`) is invoked when the reduction is complete

```
void initialize(int size) {
  double init_range = InitRange * CellSize;

  centerx = CellSize * thisIndex.x + 0.5 * CellSize;
  centery = CellSize * thisIndex.y + 0.5 * CellSize;

  // Randomly initialize particles to be some distance from the
center of the square
  for (unsigned i = 0; i < size; i++) {
    srand48(thisIndex.x * 10 + thisIndex.y * 1000 + i + size * 20);

    // ...

    Particle p(drand48()*1000000, vx, vy, nrows, ncols, max_vel);

    p.x = centerx + (drand48() - 0.5) * init_range;
    p.y = centery + (drand48() - 0.5) * init_range;
    p.mass = drand48() * 500 + 300;
    particles.push_back(p);
  }

  contribute(CkCallback(CkIndex_Main::finishInit(), mainProxy));
}
```

Charm++ Reductions

- Runtime system builds a reduction tree
- User specifies the reduction operation
- At the root of the tree, a callback is performed on a specified chare



Charm++ Van der Waals Example

vanderwaals.ci

```
mainmodule parallel {
  readonly CProxy_Main mainProxy;
  readonly int numIterations;
  readonly int numParticles;

  mainchare Main {
    entry Main(CkArgMsg*);
    entry void finishInit();
    entry void end();
  };

  array [2D] Cell {
    entry Cell();
    entry void initialize(int);
    entry void sendAll(int);
    entry void migrateParts(int);
    entry void stepMine(void);
    entry void process(void) {
      // ... Cell's lifecycle ...
    };
    entry void updateParticles(int t0,
                               CkVec<Particle> &parts);
    entry void tradeParticles(int t0,
                              CkVec<Particle> &parts);
  };
};
```

Broadcasts `process()` to Cell array

Calls `CkExit()`

Initializes random particles, performs reduction back to `finishInit()`

Sends data to each of cell's neighbors

Sending the Particles

- Point-to-point messages between chare array elements

```
typedef struct {
  int x, y;
} sendPair;

class Cell : public CBase_Cell {
// ...
public:
  CkVec<Particle> particles;
  sendPair sends[8];
  // ...

  Cell() {
    wrapAround(sends, 0, thisIndex.x-1, thisIndex.y);
    wrapAround(sends, 1, thisIndex.x-1, thisIndex.y-1);
    wrapAround(sends, 2, thisIndex.x, thisIndex.y-1);
    wrapAround(sends, 3, thisIndex.x+1, thisIndex.y);
    wrapAround(sends, 4, thisIndex.x+1, thisIndex.y+1);
    wrapAround(sends, 5, thisIndex.x, thisIndex.y+1);
    wrapAround(sends, 6, thisIndex.x-1, thisIndex.y+1);
    wrapAround(sends, 7, thisIndex.x+1, thisIndex.y-1);
  }

  void sendAll(int t0) {
    for (int i = 0; i < 8; i++) {
      thisProxy(sends[i].x, sends[i].y).updateParticles(t0, particles);
    }
  }
};
```

Charm++ Van der Waals Example

vanderwaals.ci

```
mainmodule parallel {
  readonly CProxy_Main mainProxy;
  readonly int numIterations;
  readonly int numParticles;

  mainchare Main {
    entry Main(CkArgMsg*);
    entry void finishInit();
    entry void end();
  };

  array [2D] Cell {
    entry Cell();
    entry void initialize(int);
    entry void sendAll(int);
    entry void migrateParts(int);
    entry void stepMine(void);
    entry void process(void) {
      // ... Cell's lifecycle ...
    };
    entry void updateParticles(int t0,
                               CkVec<Particle> &parts);
    entry void tradeParticles(int t0,
                              CkVec<Particle> &parts);
  };
};
```

Broadcasts `process()` to Cell array

Calls `CkExit()`

Initializes random particles, performs reduction back to `finishInit()`

Sends data to each of cell's neighbors

Migrates particles that are now out of range

migrateParts() Method

```
void migrateParts(int t0) {
  const int cellsize2 = CellSize / 2;

  map<int, CkVec<Particle> > tosend;
  map<int, bool> remove_list;

  // ... code for building these maps ...

  for (int i = 0; i < 8; i++) {
    thisProxy(sends[i].x, sends[i].y).tradeParticles(t0, tosend[i]);
  }

  // ... code for removing the send elements from local particles
}
```

Charm++ Van der Waals Example

vanderwaals.ci

```
mainmodule parallel {
  readonly CProxy_Main mainProxy;
  readonly int numIterations;
  readonly int numParticles;

  mainchare Main {
    entry Main(CkArgMsg*);
    entry void finishInit();
    entry void end();
  };

  array [2D] Cell {
    entry Cell();
    entry void initialize(int);
    entry void sendAll(int);
    entry void migrateParts(int);
    entry void stepMine(void);
    entry void process(void) {
      // ... Cell's lifecycle ...
    };
    entry void updateParticles(int t0,
                               CkVec<Particle> &parts);
    entry void tradeParticles(int t0,
                              CkVec<Particle> &parts);
  };
};
```

Broadcasts `process()` to Cell array

Calls `CkExit()`

Initializes random particles, performs reduction back to `finishInit()`

Sends data to each of cell's neighbors

Migrates particles that are now out of range

Computes local interations

`stepMine()` Method

```
void stepMine() {
  int size = (int)particles.size();
  for (int i = 0; i < size-1; i++) {
    for (int j = i + 1; j < size; j++) {
      interact(particles[i], particles[j], CellSize, A, B);
      interact(particles[j], particles[i], CellSize, A, B);
    }
  }
}
```

Charm++ Van der Waals Example

vanderwaals.ci

```
mainmodule parallel {
  readonly CProxy_Main mainProxy;
  readonly int numIterations;
  readonly int numParticles;

  mainchare Main {
    entry Main(CkArgMsg*);
    entry void finishInit();
    entry void end();
  };

  array [2D] Cell {
    entry Cell();
    entry void initialize(int);
    entry void sendAll(int);
    entry void migrateParts(int);
    entry void stepMine(void);
    entry void process(void) {
      // ... Cell's lifecycle ...
    };
    entry void updateParticles(int t0,
                               CkVec<Particle> &parts);
    entry void tradeParticles(int t0,
                              CkVec<Particle> &parts);
  };
};
```

- Broadcasts `process()` to Cell array
- Calls `CkExit()`
- Initializes random particles, performs reduction back to `finishInit()`
- Sends data to each of cell's neighbors
- Migrates particles that are now out of range
- Computes local interations
- Cell's top-level lifecycle

High Level Coordination in Charm++

Phil Miller

Charm++ Asynchronous Methods

- Does this code produce the desired output?

```
class Main : public CBase_Main {
public:
  Main(CkArgMsg* m) {
    double pi = 3.1415;
    CProxy_Simple sim = CProxy_Simple::ckNew(pi);

    for (int i = 1; i < 10; i++)
      sim.findArea(i, false);

    sim.findArea(10, true);
  };
};

class Simple : public CBase_Simple {
private:
  float y;

public:
  Simple(double pi) {
    y = pi;
    ckout << "Hello from a simple chare running on " << CkMyPe() << endl;
  }

  void findArea(int r, bool done) {
    ckout << "Area of a circle of radius " << r << " is " << y*r*r <<
endl;

    if (done)
      CkExit();
  }
};
```

```
mainmodule simple {
  mainchare Main {
    entry Main(CkArgMsg *m);
  };

  chare Simple {
    entry Simple(double y);
    entry void findArea(int radius, bool);
  };
};
```

Counting Messages

```
class Simple : public CBase_Simple {
  float y;
  int received;       // Added counter
  bool doneReceived; // Added flag

  void findArea(int r, bool done) {
    ckout << "Area of a circle of radius " << r << " is " << y*r*r << endl;

    // Update conditions
    received++;
    doneReceived = doneReceived || done;

    if (received >= 10 && doneReceived)
      CkExit();
  }
};
```

Counting with Coordination

.ci file

```
entry void work(int messages) {
  for (i = 1; i <= messages; i++) {
    when findArea (int r, bool done) atomic {
      printArea(r);
      doneReceived = doneReceived || done;
    }
  }

  if (doneReceived) atomic {
    CkExit();
  }
}
```

.cpp file

```
void Simple::Simple(double pi) {
  // . . .
  thishandle.work(10);
  // . . .
}

void Simple::printArea(int r) {
  ckout << "Area of a circle of radius " << r << "
is " << y*r*r << endl;
}
```

Counting with Coordination

.ci file

```
entry void work(int messages) {
  for (i = 1; i <= messages; i++) {
    when findArea (int r, bool done) atomic {
      printArea(r);
      doneReceived = doneReceived || done;
    }
  }

  if (doneReceived) atomic {
    CkExit();
  }
}
```

.cpp file

```
void Simple::Simple(double pi) {
  // . . .
  thishandle.work(10);
  // . . .
}

void Simple::printArea(int r) {
  ckout << "Area of a circle of radius " << r << "
is " << y*r*r << endl;
}
```

Counting with Coordination

.ci file

```
entry void work(int messages) {
  for (i = 1; i <= messages; i++) {
    when findArea (int r, bool done) atomic {
      printArea(r);
      doneReceived = doneReceived || done;
    }
  }

  if (doneReceived) atomic {
    CkExit();
  }
}
```

.cpp file

```
void Simple::Simple(double pi) {
  // . . .
  thishandle.work(10);
  // . . .
}

void Simple::printArea(int r) {
  ckout << "Area of a circle of radius " << r << "
is " << y*r*r << endl;
}
```

Counting with Coordination

.ci file

```
entry void work(int messages) {
  for (i = 1; i <= messages; i++) {
    when findArea (int r, bool done) atomic {
      printArea(r);
      doneReceived = doneReceived || done;
    }
  }

  if (doneReceived) atomic {
    CkExit();
  }
}
```

.cpp file

```
void Simple::Simple(double pi) {
  // . . .
  thishandle.work(10);
  // . . .
}

void Simple::printArea(int r) {
  ckout << "Area of a circle of radius " << r << "
is " << y*r*r << endl;
}
```

Counting with Coordination

.ci file

```
entry void work(int messages) {
  for (i = 1; i <= messages; i++) {
    when findArea (int r, bool done) atomic {
      printArea(r);
      doneReceived = doneReceived || done;
    }
  }

  if (doneReceived) atomic {
    CkExit();
  }
}
```

.cpp file

```
void Simple::Simple(double pi) {
  // . . .
  thishandle.work(10);
  // . . .
}

void Simple::printArea(int r) {
  ckout << "Area of a circle of radius " << r << "
is " << y*r*r << endl;
}
```

Counting with Coordination

.ci file

```
entry void work(int messages) {
  for (i = 1; i <= messages; i++) {
    when findArea (int r, bool done) atomic {
      printArea(r);
      doneReceived = doneReceived || done;
    }
  }

  if (doneReceived) atomic {
    CkExit();
  }
}
```

.cpp file

```
void Simple::Simple(double pi) {
  // . . .
  thishandle.work(10);
  // . . .
}

void Simple::printArea(int r) {
  ckout << "Area of a circle of radius " << r << "
is " << y*r*r << endl;
}
```

Counting with Coordination

.ci file

```
entry void work(int messages) {
  for (i = 1; i <= messages; i++) {
    when findArea (int r, bool done) atomic {
      printArea(r);
      doneReceived = doneReceived || done;
    }
  }

  if (doneReceived) atomic {
    CkExit();
  }
}
```

.cpp file

```
void Simple::Simple(double pi) {
  // . . .
  thishandle.work(10);
  // . . .
}

void Simple::printArea(int r) {
  ckout << "Area of a circle of radius " << r << "
is " << y*r*r << endl;
}
```

Counting with Coordination

.ci file

```
entry void work(int messages) {
  for (i = 1; i <= messages; i++) {
    when findArea (int r, bool done) atomic {
      printArea(r);
      doneReceived = doneReceived || done;
    }
  }

  if (doneReceived) atomic {
    CkExit();
  }
}
```

.cpp file

```
void Simple::Simple(double pi) {
  // . . .
  thishandle.work(10);
  // . . .
}

void Simple::printArea(int r) {
  ckout << "Area of a circle of radius " << r << "
is " << y*r*r << endl;
}
```

Counting with Coordination

.ci file

```
entry void work(int messages) {
  for (i = 1; i <= messages; i++) {
    when findArea (int r, bool done) atomic {
      printArea(r);
      doneReceived = doneReceived || done;
    }
  }

  if (doneReceived) atomic {
    CkExit();
  }
}
```

.cpp file

```
void Simple::Simple(double pi) {
  // . . .
  thishandle.work(10);
  // . . .
}

void Simple::printArea(int r) {
  ckout << "Area of a circle of radius " << r << "
is " << y*r*r << endl;
}
```

Charm++ Van der Waals Example

vanderwaals.ci

```
mainmodule parallel {
  readonly CProxy_Main mainProxy;
  readonly int numIterations;
  readonly int numParticles;

  mainchare Main {
    entry Main(CkArgMsg*);
    entry void finishInit();
    entry void end();
  };

  array [2D] Cell {
    entry Cell();
    entry void initialize(int);
    entry void sendAll(int);
    entry void migrateParts(int);
    entry void stepMine(void);
    entry void process(void) {
      // ... Cell's lifecycle ...
    };
    entry void updateParticles(int t0,
                               CkVec<Particle> &parts);
    entry void tradeParticles(int t0,
                              CkVec<Particle> &parts);
  };
};
```

- Broadcasts `process()` to Cell array
- Calls `CkExit()`
- Initializes random particles, performs reduction back to `finishInit()`
- Sends data to each of cell's neighbors
- Migrates particles that are now out of range
- Computes local interations
- Cell's top-level lifecycle

Cell's Lifecycle

```
entry void process(void) {
  while (current_time < numIterations) {
    atomic "send particle positions" {
      beginStep();
      sendAll(current_time);
    }
    overlap {
      atomic "interact local particles" {
        stepMine();
      }
      for (i = 0; i < 8; i++) {
        when updateParticles (int t0, CkVec<Particle> &parts)
        atomic "process particles" { step(parts); }
      }
    }
    for (i = 0; i < particles.size(); i++) {
      atomic "update all" { particles[i].update(); }
    }
    atomic "migrate particles" { migrateParts(current_time); }
    for (i = 0; i < 8; i++) {
      when tradeParticles (int t0, CkVec<Particle> &parts)
      atomic "add new particles" { addParticles(parts); }
    }
    atomic "increment time" {
      current_time++;
    }
  }
  atomic "finalize" {
    contribute(CkCallback(CkIndex_Main::end(), mainProxy));
  }
};
```

Cell's Lifecycle

```
entry void process(void) {
  while (current_time < numIterations) {
    atomic "send particle positions" {
      beginStep();
      sendAll(current_time);
    }
    overlap {
      atomic "interact local particles" {
        stepMine();
      }
      for (i = 0; i < 8; i++) {
        when updateParticles (int t0, CkVec<Particle> &parts)
        atomic "process particles" { step(parts); }
      }
    }
    for (i = 0; i < particles.size(); i++) {
      atomic "update all" { particles[i].update(); }
    }
    atomic "migrate particles" { migrateParts(current_time); }
    for (i = 0; i < 8; i++) {
      when tradeParticles (int t0, CkVec<Particle> &parts)
      atomic "add new particles" { addParticles(parts); }
    }
    atomic "increment time" {
      current_time++;
    }
  }
  atomic "finalize" {
    contribute(CkCallback(CkIndex_Main::end(), mainProxy));
  }
};
```

Cell's Lifecycle

```
entry void process(void) {
  while (current_time < numIterations) {
    atomic "send particle positions" {
      beginStep();
      sendAll(current_time);
    }
    overlap {
      atomic "interact local particles" {
        stepMine();
      }
      for (i = 0; i < 8; i++) {
        when updateParticles (int t0, CkVec<Particle> &parts)
        atomic "process particles" { step(parts); }
      }
    }
    for (i = 0; i < particles.size(); i++) {
      atomic "update all" { particles[i].update(); }
    }
    atomic "migrate particles" { migrateParts(current_time); }
    for (i = 0; i < 8; i++) {
      when tradeParticles (int t0, CkVec<Particle> &parts)
      atomic "add new particles" { addParticles(parts); }
    }
    atomic "increment time" {
      current_time++;
    }
  }
  atomic "finalize" {
    contribute(CkCallback(CkIndex_Main::end(), mainProxy));
  }
};
```

Sending the Particles

```
typedef struct {
  int x, y;
} sendPair;

class Cell : public CBase_Cell {
// ...
public:
  CkVec<Particle> particles;
  sendPair sends[8];
  // ...

  Cell() {
    wrapAround(sends, 0, thisIndex.x-1, thisIndex.y);
    wrapAround(sends, 1, thisIndex.x-1, thisIndex.y-1);
    wrapAround(sends, 2, thisIndex.x, thisIndex.y-1);
    wrapAround(sends, 3, thisIndex.x+1, thisIndex.y);
    wrapAround(sends, 4, thisIndex.x+1, thisIndex.y+1);
    wrapAround(sends, 5, thisIndex.x, thisIndex.y+1);
    wrapAround(sends, 6, thisIndex.x-1, thisIndex.y+1);
    wrapAround(sends, 7, thisIndex.x+1, thisIndex.y-1);
  }

  void sendAll(int t0) {
    for (int i = 0; i < 8; i++) {
      thisProxy(sends[i].x, sends[i].y).updateParticles(t0, particles);
    }
  }
};
```

Cell's Lifecycle

```
entry void process(void) {
  while (current_time < numIterations) {
    atomic "send particle positions" {
      beginStep();
      sendAll(current_time);
    }
    overlap {
      atomic "interact local particles" {
        stepMine();
      }
      for (i = 0; i < 8; i++) {
        when updateParticles (int t0, CkVec<Particle> &parts)
        atomic "process particles" { step(parts); }
      }
    }
    for (i = 0; i < particles.size(); i++) {
      atomic "update all" { particles[i].update(); }
    }
    atomic "migrate particles" { migrateParts(current_time); }
    for (i = 0; i < 8; i++) {
      when tradeParticles (int t0, CkVec<Particle> &parts)
      atomic "add new particles" { addParticles(parts); }
    }
    atomic "increment time" {
      current_time++;
    }
  }
  atomic "finalize" {
    contribute(CkCallback(CkIndex_Main::end(), mainProxy));
  }
};
```

Cell's Lifecycle

```
entry void process(void) {
  while (current_time < numIterations) {
    atomic "send particle positions" {
      beginStep();
      sendAll(current_time);
    }
    overlap {
      atomic "interact local particles" {
        stepMine();
      }
      for (i = 0; i < 8; i++) {
        when updateParticles (int t0, CkVec<Particle> &parts)
        atomic "process particles" { step(parts); }
      }
    }
    for (i = 0; i < particles.size(); i++) {
      atomic "update all" { particles[i].update(); }
    }
    atomic "migrate particles" { migrateParts(current_time); }
    for (i = 0; i < 8; i++) {
      when tradeParticles (int t0, CkVec<Particle> &parts)
      atomic "add new particles" { addParticles(parts); }
    }
    atomic "increment time" {
      current_time++;
    }
  }
  atomic "finalize" {
    contribute(CkCallback(CkIndex_Main::end(), mainProxy));
  }
};
```

Cell's Lifecycle

```
entry void process(void) {
  while (current_time < numIterations) {
    atomic "send particle positions" {
      beginStep();
      sendAll(current_time);
    }
    overlap {
      atomic "interact local particles" {
        stepMine();
      }
      for (i = 0; i < 8; i++) {
        when updateParticles (int t0, CkVec<Particle> &parts)
        atomic "process particles" { step(parts); }
      }
    }
    for (i = 0; i < particles.size(); i++) {
      atomic "update all" { particles[i].update(); }
    }
    atomic "migrate particles" { migrateParts(current_time); }
    for (i = 0; i < 8; i++) {
      when tradeParticles (int t0, CkVec<Particle> &parts)
      atomic "add new particles" { addParticles(parts); }
    }
    atomic "increment time" {
      current_time++;
    }
  }
  atomic "finalize" {
    contribute(CkCallback(CkIndex_Main::end(), mainProxy));
  }
};
```

Cell's Lifecycle

```
entry void process(void) {
  while (current_time < numIterations) {
    atomic "send particle positions" {
      beginStep();
      sendAll(current_time);
    }
    overlap {
      atomic "interact local particles" {
        stepMine();
      }
      for (i = 0; i < 8; i++) {
        when updateParticles (int t0, CkVec<Particle> &parts)
        atomic "process particles" { step(parts); }
      }
    }
    for (i = 0; i < particles.size(); i++) {
      atomic "update all" { particles[i].update(); }
    }
    atomic "migrate particles" { migrateParts(current_time); }
    for (i = 0; i < 8; i++) {
      when tradeParticles (int t0, CkVec<Particle> &parts)
      atomic "add new particles" { addParticles(parts); }
    }
    atomic "increment time" {
      current_time++;
    }
  }
  atomic "finalize" {
    contribute(CkCallback(CkIndex_Main::end(), mainProxy));
  }
};
```

migrateParts() Method

```
void migrateParts(int t0) {
  const int cellsize2 = CellSize / 2;

  map<int, CkVec<Particle> > tosend;
  map<int, bool> remove_list;

  // ... code for building these maps ...

  for (int i = 0; i < 8; i++) {
    thisProxy(sends[i].x, sends[i].y).tradeParticles(t0, tosend[i]);
  }

  // ... code for removing the send elements from local particles
}
```

Cell's Lifecycle

```
entry void process(void) {
  while (current_time < numIterations) {
    atomic "send particle positions" {
      beginStep();
      sendAll(current_time);
    }
    overlap {
      atomic "interact local particles" {
        stepMine();
      }
      for (i = 0; i < 8; i++) {
        when updateParticles (int t0, CkVec<Particle> &parts)
        atomic "process particles" { step(parts); }
      }
    }
    for (i = 0; i < particles.size(); i++) {
      atomic "update all" { particles[i].update(); }
    }
    atomic "migrate particles" { migrateParts(current_time); }
    for (i = 0; i < 8; i++) {
      when tradeParticles (int t0, CkVec<Particle> &parts)
      atomic "add new particles" { addParticles(parts); }
    }
    atomic "increment time" {
      current_time++;
    }
  }
  atomic "finalize" {
    contribute(CkCallback(CkIndex_Main::end(), mainProxy));
  }
};
```

Cell's Lifecycle

```
entry void process(void) {
  while (current_time < numIterations) {
    atomic "send particle positions" {
      beginStep();
      sendAll(current_time);
    }
    overlap {
      atomic "interact local particles" {
        stepMine();
      }
      for (i = 0; i < 8; i++) {
        when updateParticles (int t0, CkVec<Particle> &parts)
        atomic "process particles" { step(parts); }
      }
    }
    for (i = 0; i < particles.size(); i++) {
      atomic "update all" { particles[i].update(); }
    }
    atomic "migrate particles" { migrateParts(current_time); }
    for (i = 0; i < 8; i++) {
      when tradeParticles (int t0, CkVec<Particle> &parts)
      atomic "add new particles" { addParticles(parts); }
    }
    atomic "increment time" {
      current_time++;
    }
  }
  atomic "finalize" {
    contribute(CkCallback(CkIndex_Main::end(), mainProxy));
  }
};
```

Cell's Lifecycle

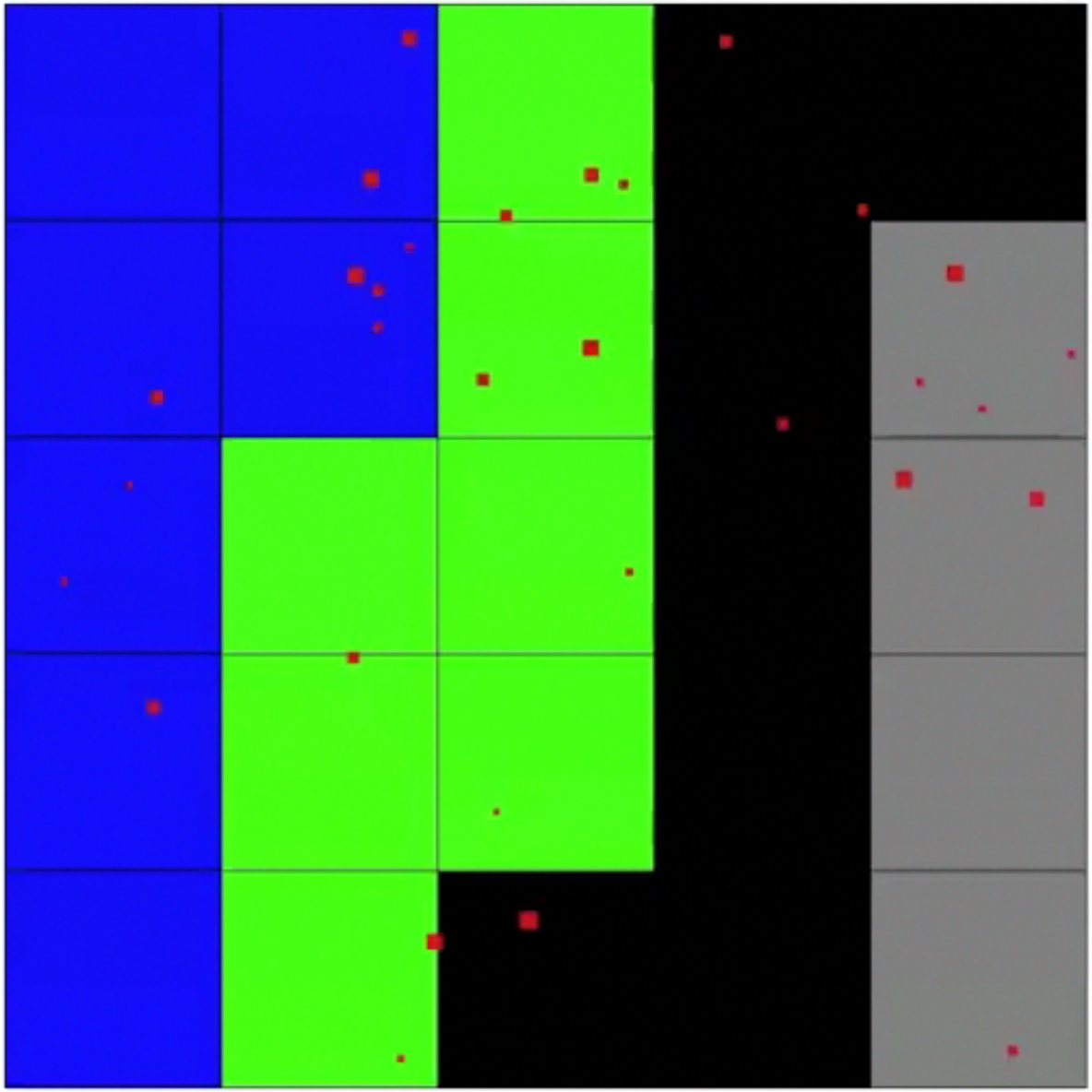
```
entry void process(void) {
  while (current_time < numIterations) {
    atomic "send particle positions" {
      beginStep();
      sendAll(current_time);
    }
    overlap {
      atomic "interact local particles" {
        stepMine();
      }
      for (i = 0; i < 8; i++) {
        when updateParticles (int t0, CkVec<Particle> &parts)
        atomic "process particles" { step(parts); }
      }
    }
    for (i = 0; i < particles.size(); i++) {
      atomic "update all" { particles[i].update(); }
    }
    atomic "migrate particles" { migrateParts(current_time); }
    for (i = 0; i < 8; i++) {
      when tradeParticles (int t0, CkVec<Particle> &parts)
      atomic "add new particles" { addParticles(parts); }
    }
    atomic "increment time" {
      current_time++;
    }
  }
  atomic "finalize" {
    contribute(CkCallback(CkIndex_Main::end(), mainProxy));
  }
};
```

Charm++ Van der Waals Example

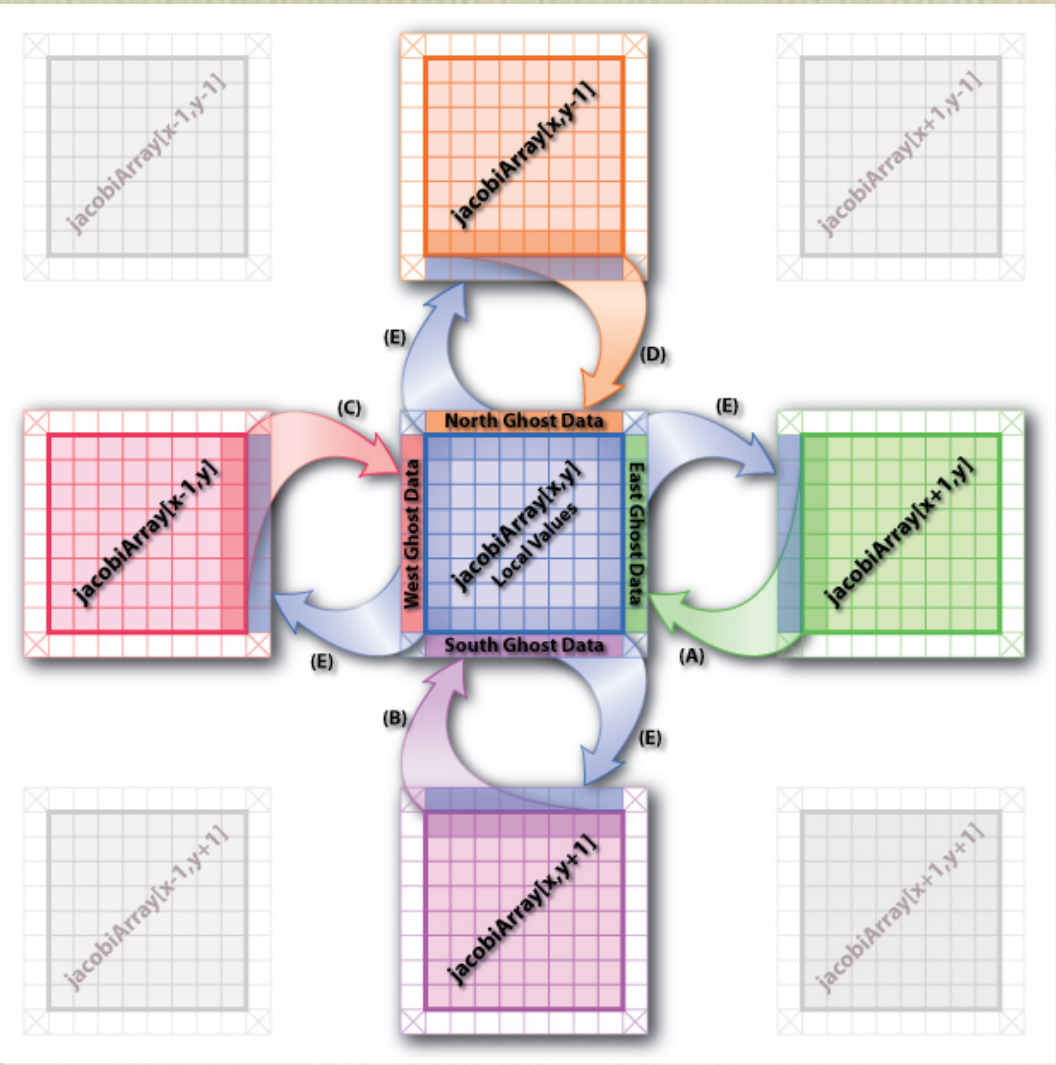
vanderwaals.ci

```
mainmodule parallel {
  readonly CProxy_Main mainProxy;
  readonly int numIterations;
  readonly int numParticles;

  mainchare Main {
    entry Main(CkArgMsg*);
    entry void finishInit();
    entry void end();
  };

  array [2D] Cell {
    entry Cell();
    entry void initialize(int);
    entry void sendAll(int);
    entry void migrateParts(int);
    entry void stepMine(void);
    entry void process(void) {
      // ... Cell's lifecycle ...
    };
    entry void updateParticles(int t0,
                               CkVec<Particle> &parts);
    entry void tradeParticles(int t0,
                              CkVec<Particle> &parts);
  };
};
```

- Broadcasts `process()` to Cell array
- Calls `CkExit()`
- Initializes random particles, performs reduction back to `finishInit()`
- Sends data to each of cell's neighbors
- Migrates particles that are now out of range
- Computes local interations
- Cell's top-level lifecycle
- Sends particles from neighboring cells for interaction
- Sends particles from neighboring cells for migration



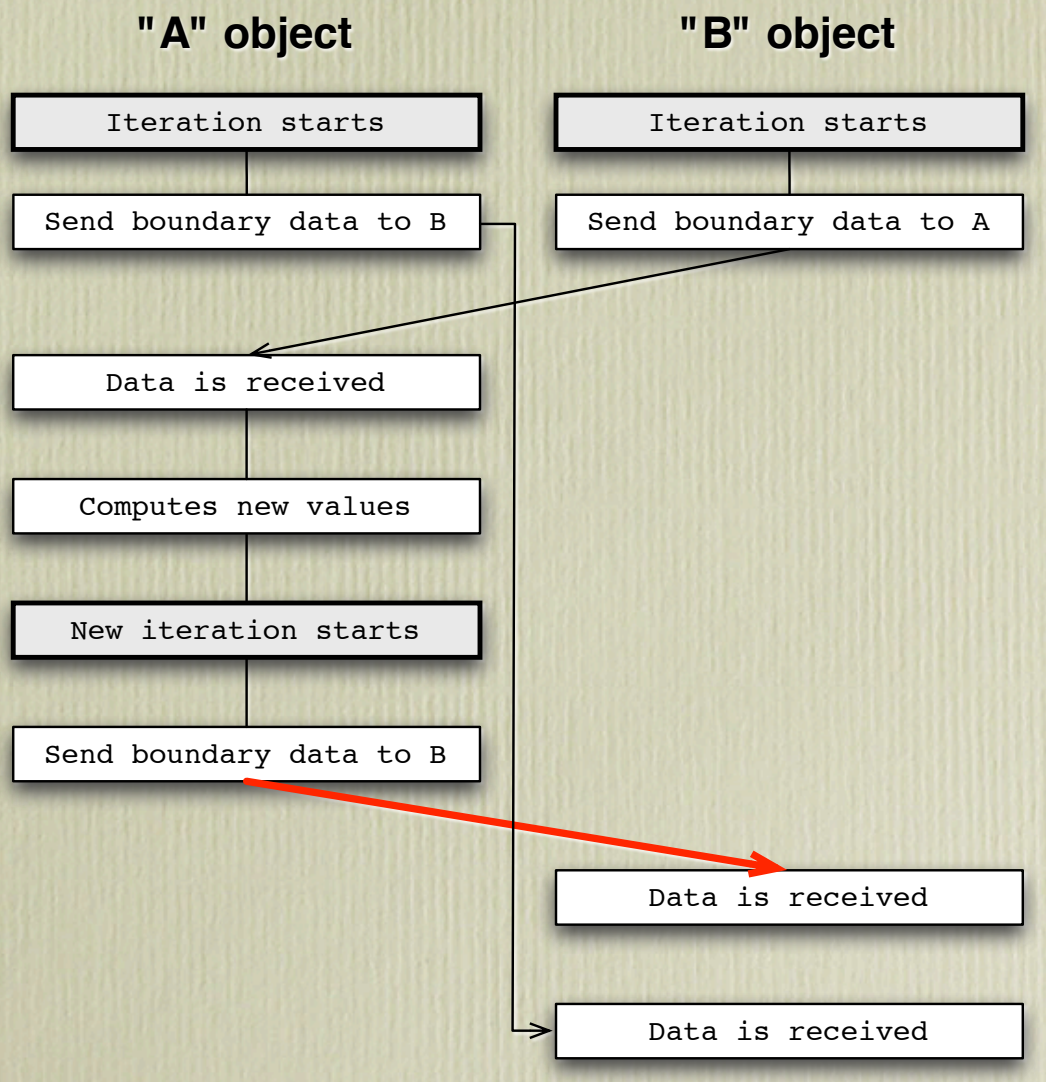
Jacobi Example



Incorrect Jacobi Code

```
array [3D] Jacobi {
  //. . .
  entry void doStep() {
    atomic "begin_iteration" {
        begin_iteration();
    }
    for (imsg = 0; imsg < 6; imsg++) {
        // "iterations" keeps track of messages across steps
        when receiveGhosts (int dir, int height, int width,
                            double ghosts[height*width])
        atomic "process ghosts" {
            processGhosts(dir, height, width, ghosts);
        }
    }
    atomic "doWork" {
        check_and_compute();
    }
  };
};

void begin_iteration() {
    //. . .
    double *leftGhost =  new double[blockDimY*blockDimZ];
    // Copy data to send . . .
    thisProxy(wrap_x(thisIndex.x-1), thisIndex.y, thisIndex.z)
        .receiveGhosts(iterations, RIGHT, blockDimY, blockDimZ,
                       leftGhost);
    //. . .
}
```

Incorrect Jacobi Code



Correct Jacobi Code

```
class Jacobi {
    // . . .
    std::map<int, std::list<Rec *> > received;
    int rec_this_iter;

    void checkReceived() {
        for (std::list<Rec *>::iterator i = received[iterations].begin();
             i != received[iterations].end(); ++i) {
            Rec *r = *i;
            processGhosts(r->dir, r->height, r->width, r->gh);
            delete r;
        }
        received.erase(iterations);
        if (6 == rec_this_iter) {
            check_and_compute();
        }
    }

    void receiveGhosts(int iter, int dir, int height, int width,
                       double ghosts[]) {
        if (iter == iterations) {
            processGhosts(dir, height, width, ghosts);
            checkReceived();
        }
        else
            received[iter].push_back(new Rec(dir, height, width, ghosts));
    }
    // . . .
};
```

Simplified Jacobi Code

```
array [3D] Jacobi {
  // . . .
  entry void doStep() {
    atomic "begin_iteration" {
      begin_iteration();
    }
    for (imsg = 0; imsg < 6; imsg++) {
      // "iterations" keeps track of messages across steps
      when receiveGhosts[iterations] (int iter, int dir, int height,
                                      int width, double ghosts[height*width])
      atomic "process ghosts" {
        processGhosts(dir, height, width, ghosts);
      }
    }
    atomic "doWork" {
      check_and_compute();
    }
  };
};
void begin_iteration() {
    //. . .
    double *leftGhost =  new double[blockDimY*blockDimZ];
    // Copy data to send . . .
    thisProxy(wrap_x(thisIndex.x-1), thisIndex.y, thisIndex.z)
        .receiveGhosts(iterations, RIGHT, blockDimY, blockDimZ, leftGhost);
    // . . .
}
```