Chares are reactive

- The way we described Charm++ so far, a chare is a reactive entity:
  - If it gets this method invocation, it does this action,
  - If it gets that method invocation then it does that action
  - But what does it do?
  - In typical programs, chares have a *life-cycle*
- How to express the life-cycle of a chare in code?
  - Only when it exists
    - ★ i.e. some chars may be truly reactive, and the programmer does not know the life cycle
  - But when it exists, its form is:
    - ★ Computations depend on remote method invocations, and completion of other local computations
    - ★ A DAG (Directed Acyclic Graph)!

Consider Fibonacci Chare

- The Fibonacci chare gets created
- If its not a leaf,
  - It fires two chares
  - When both children return results (by calling `response`):
    - ⋆ It can compute my result and send it up, or print it
  - But in our, this logic is hidden in the flags and counters ...
    - ⋆ This is simple for this simple example, but ...
  - Lets look at how this would look with a little notational support

Structured Dagger Constructs: `atomic`

- The `atomic` construct
  - A sequential block of C++ code
  - The keyword `atomic` means that the code block will be executed without interruption/preemption, like an entry method
  - Syntax: `atomic <optionalString> { /* C++ code */ }`
  - The `<optionalString>` is used for identifying the `atomic` for performance analysis
  - Atomics can access all members of the class they belong to
- Examples:

```
atomic "setValue" {
  value = 10;
}
```

```
atomic {
  thisProxy.invokeMethod(10);
  callSomeFunction();
}
```

Structured Dagger Constructs: `when`

- The `when` construct
  - Declare the actions to perform when a message is received
  - In sequence, it acts like a blocking receive
  - A `when` must have a corresponding declaration of an entry method
  - The actual body of the corresponding entry method is generated
- `when` semantics:

```
entry void someMethod() {
  atomic { /* block1 */ }
  when entryMethod1(parameters) {
    // ... further code ...
  }
  atomic { /* block2 */ }
}

entry void entryMethod1(parameters);
```

- Sequence
  - Sequentially execute `/* block1 */`
  - Wait for `entryMethod1` to arrive, if it has not, return control back to the Charm++ scheduler, otherwise, execute `/* further code */`
  - Sequentially execute `/* block2 */`

Structured Dagger Constructs: `when`

- Execute `/* further sdag */` when `myMethod` arrives

```
when myMethod(int param1, int param2)
  /* further sdag */
```

- Execute `/* further sdag */` when `myMethod1` and `myMethod2` arrive

```
when myMethod1(int param1, int param2),
       myMethod2(bool param3)
  /* further sdag */
```

- Syntactical sugar for:

```
when myMethod1(int param1, int param2)
  when myMethod2(bool param3)
    /* further sdag */
```

Fibonacci with Structured Dagger

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

  chare Fib {
    entry Fib(int n, bool isRoot, CProxy_Fib parent);
    entry void calc(int n) {
      if (n < THRESHOLD) atomic { respond(seqFib(n)); }
      else {
        atomic {
          CProxy_Fib::ckNew(n - 1, false, thisProxy);
          CProxy_Fib::ckNew(n - 2, false, thisProxy);
        }
        when response(int val)
          when response(int val2)
            atomic { respond(val + val2); }
      }
    };
    entry void response(int);
  };
};
```

Fibonacci with Structured Dagger

```
#include "fib.decl.h"
#define THRESHOLD 10

struct Main : public CBase_Main {
  Main(CkArgMsg* m) { CProxy_Fib::ckNew(atoi(m->argv[1]), true, CProxy_Fib()); }
};

struct Fib : public CBase_Fib {
  Fib_SDAG_CODE
  CProxy_Fib parent; bool isRoot;

  Fib(int n, bool isRoot_, CProxy_Fib parent_)
    : parent(parent_), isRoot(isRoot_) {
    __sdag_init();
    calc(n);
  }

  int seqFib(int n) { return (n < 2) ? n : seqFib(n - 1) + seqFib(n - 2); }

  void respond(int val) {
    if (!isRoot) {
      parent.response(val);
      delete this;
    } else {
      CkPrintf("Fibonacci number is: %d\n", val);
      CkExit();
    }
  }
};

#include "fib.def.h"
```

Structured Dagger Constructs: `when`

- What is the sequence?

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

Structured Dagger Constructs: `when`

- What is the sequence?

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

- Sequence:
  - Wait for `myMethod1`, upon arrival execute body of `myMethod1`

Structured Dagger Constructs: `when`

- What is the sequence?

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

- Sequence:
  - Wait for `myMethod1`, upon arrival execute body of `myMethod1`
  - Wait for `myMethod2` and `myMethod3`, upon arrival of both, execute `/* sdag block1 */`

Structured Dagger Constructs: `when`

- What is the sequence?

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

- Sequence:
  - Wait for `myMethod1`, upon arrival execute body of `myMethod1`
  - Wait for `myMethod2` and `myMethod3`, upon arrival of both, execute `/* sdag block1 */`
  - Wait for `myMethod4`, upon arrival execute `/* sdag block2 */`
- Question: if `myMethod4` arrives first what will happen?

Structured Dagger Constructs: Reference Numbers

- Entry methods can be *tagged* with a *reference number*
- A reference number is a special field in the envelope of the message that is sent
- By default, the reference number is a `short`
- This can be changed when compiling charm:
  - Add this to the build flags: `--with-refnum-type=int`
  - For example, compiling on BG/P with the IBM XLC compiler:

```
./build charm++ bluegenep xlc --with-refnum-type=int -g -O0
```

Structured Dagger Constructs: `when`

- The `when` clause can wait on a certain reference number
- If a reference number is specified for a `when`, the first parameter for the `when` must be the reference number
- Semantic: the `when` will "block" until a message arrives with that reference number

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

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

Structured Dagger Constructs: `when`

- Another example:

.ci file:

```
chare MyChare {
  entry MyChare();
  entry void startWork() {
    atomic { myRef = 100; }
    when method1[myRef1](short ref, bool param1) /* block1 */
    when method2[myRef2](short ref, bool param1) /* block2 */
  };
}
```

.cpp file:

```
class MyChare : public CBase_MyChare {
   int myRef1, myRef2;
   MyChare() : myRef2(200) { }
};
```

Structured Dagger Constructs: `overlap`

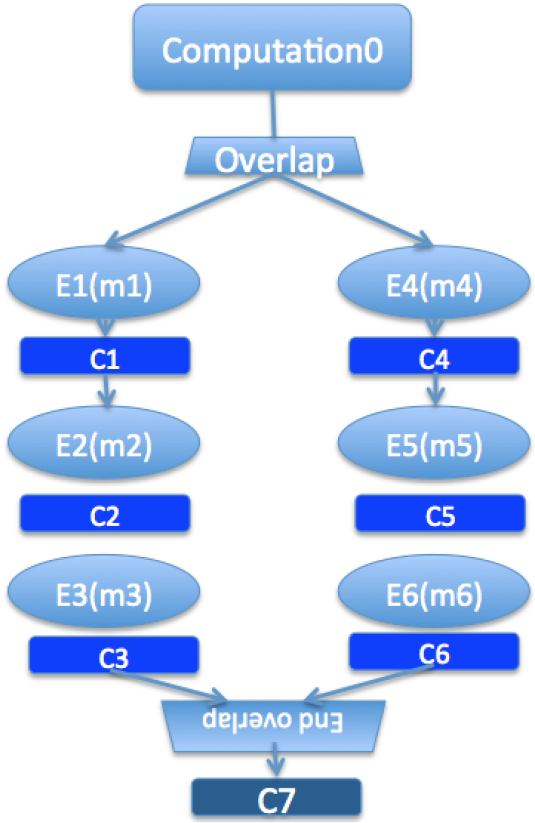
- The `overlap` construct:
  - By default, Structured Dagger defines a sequence that is followed sequentially
  - `overlap` allows multiple independent clauses to execute in any order
  - Any constructs in the body of an `overlap` can happen in any order
  - An `overlap` finishes in sequence when all the statements in it are executed
  - Syntax: `overlap { /* sdag constructs */ }`

What are the possible execution sequences?

```
atomic { /* block1 */ }
overlap {
  atomic { /* block2 */ }
  when entryMethod1[100](short ref_num, bool param1) /* block3 */
  when entryMethod2(char myChar) /* block4 */
}
atomic { /* block5 */ }
```

Illustration of a long "overlap"

- Overlap can be used to get back some of the asynchrony within a chare
  - But it is constrained
  - Makes for more disciplined programming,
    - ★ with fewer race conditions



Structured Dagger Constructs: `for`

- The `for` construct:
  - Defines a sequenced `for` loop (like a sequential C for loop)
  - Once the body for the ith iteration completes, the $i+1$ iteration is started

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

- `iter` must be defined in the class as a member

```
class Foo : public CBase_Foo {
  public: int iter;
};
```

Structured Dagger Constructs: `while`

- The `while` construct:
  - Defines a sequenced `while` loop (like a sequential C while loop)

```
while (i < numNeighbors) {
  when recvData(int len, double data[len]) {
    atomic {
      /* do something */
    }
    overlap {
      when method1() /* block1 */
      when method2() /* block2 */
    }
  }
  atomic { i++; }
}
```

Structured Dagger Constructs: `forall`

- The `forall` construct:
  - Has "do-all" semantics: iterations may execute an any order
  - Syntax:
    `forall [<ident>] (<min> : <max>, <stride>) <body>`
  - The range from `<min>` to `<max>` is inclusive

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

- Assume `block` is declared in the class as `public: short block;`

Structured Dagger Constructs: `if-then-else`

- The `if-then-else` construct:
  - Same as the typical C if-then-else semantics and syntax

```
if (thisIndex.x == 10) {
  forall [block] (0 : numBlocks − 1, 1) {
    if (isPrime(block))
      when method1[block](short ref, bool someVal) /* code block1 */
  }
} else {
  when method2(int payload) atomic {
    //... some C++ code
  }
}
```

Structured Dagger Boilerplate

- Structured Dagger can be used in any entry method (except for a constructor)
  - Can be used in a `mainchare`, `chare`, or `array`
- For any class that has Structured Dagger in it you must insert two calls:
  - The Structured Dagger macro: `[ClassName]_SDAG_CODE`
  - Call the `__sdag_init()` initializer in the constructor
  - For later: call the `__sdag_pup()` in the `pup` method

Structured Dagger Boilerplate

The .ci file:

```
[mainchare,chare,array] MyFoo {
  ...
  entry void method(parameters) {
    // ... structured dagger code here ...
  };
  ...
}
```

The .cpp file:

```
class MyFoo : public CBase_MyFoo {
  MyFoo_SDAG_CODE /* insert SDAG macro */
public:
  MyFoo() {
    __sdag_init(); /* call SDAG initialization in constructor */
  }
};
```

Determinant MP0 Solution: .ci file

```
mainmodule Determinants {
  mainchare Main {
    entry Main(CkArgMsg *m);
    entry void response(int index, int det);
  };
  chare DeterminantChare {
    entry DeterminantChare(CProxy_Main main, int i, int n, int matrix[n*n]);
  };
};
```

Determinant MP0 Solution: .cpp file (part 1)

```
#include "Determinants.decl.h"
#include <cstdlib>
#include <vector>

struct Main : public CBase_Main {
  int count; std::vector<int> dets;
  Main(CkArgMsg *msg) {
    if (msg->argc < 3) CkAbort("Usage: det <n> <m>");
    int n = std::atoi(msg->argv[1]), m = std::atoi(msg->argv[2]);
    std::srand(29);
    count = n + m;
    dets.resize(n + m);

    for (int i = 0; i < n + m; ++i) {
      int matrix[9];
      int size = i < n ? 2 : 3;
      for (int j = 0; j < size*size; ++j)
        matrix[j] = std::rand();
      CProxy_DeterminantChare::ckNew(thisProxy, i, size, matrix);
    }
  }

  void response(int index, int det) {
    dets[index] = det;
    if (--count == 0) {
      for (int i = 0; i < dets.size(); ++i)
        CkPrintf("Determinant of matrix %d is %d\n", i, dets[i]);
      CkExit();
    }
  }
};
```

Determinant MP0 Solution: .cpp file (part 2)

```
struct DeterminantChare : public CBase_DeterminantChare {
  DeterminantChare(CProxy_Main main, int i, int n, int *matrix) {
    int retVal;
    if (n == 2) {
      retVal = matrix[0]*matrix[3] - matrix[1]*matrix[2];
    } else if (n == 3) {
      retVal = matrix[0]*matrix[4]*matrix[8]
             + matrix[1]*matrix[5]*matrix[6]
             + matrix[2]*matrix[3]*matrix[7]
             - matrix[0]*matrix[5]*matrix[7]
             - matrix[1]*matrix[3]*matrix[8]
             - matrix[2]*matrix[4]*matrix[6]
        ;
    } else {
      CkAbort("Only supports determinants of size 2 or 3!");
    }
    main.response(i, retVal);
  }
};

#include "Determinants.def.h"
```

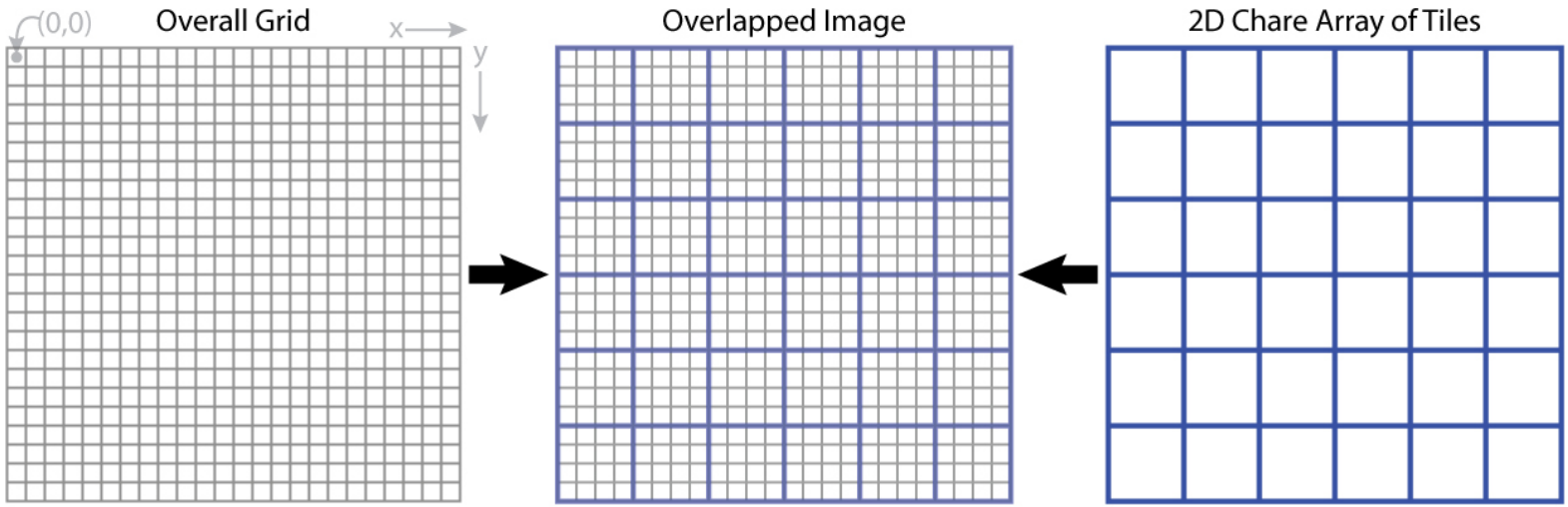
Determinant MP0 Structered Dagger: .ci file

```
mainmodule Determinants {
  mainchare Main {
    entry Main(CkArgMsg *m);
    entry void response(int index, int det);
    entry void run() {
      atomic {
        for (i = 0; i < n+m; ++i) {
          int matrix[9];
          int size = i < n ? 2 : 3;
          for (int j = 0; j < size*size; ++j) matrix[j] = rand();
          CProxy_DeterminantChare::ckNew(thisProxy, i, size, matrix);
        }
      }
      for (i = 0; i < n+m; ++i)
        when response[i](int index, int det) atomic {
          CkPrintf("Determinant of matrix %d is %d\n", i, det);
        }
      atomic { CkExit(); }
    };
  }
  chare DeterminantChare {
    entry DeterminantChare(CProxy_Main main, int i, int n, int matrix[n*n]);
  }
}
```

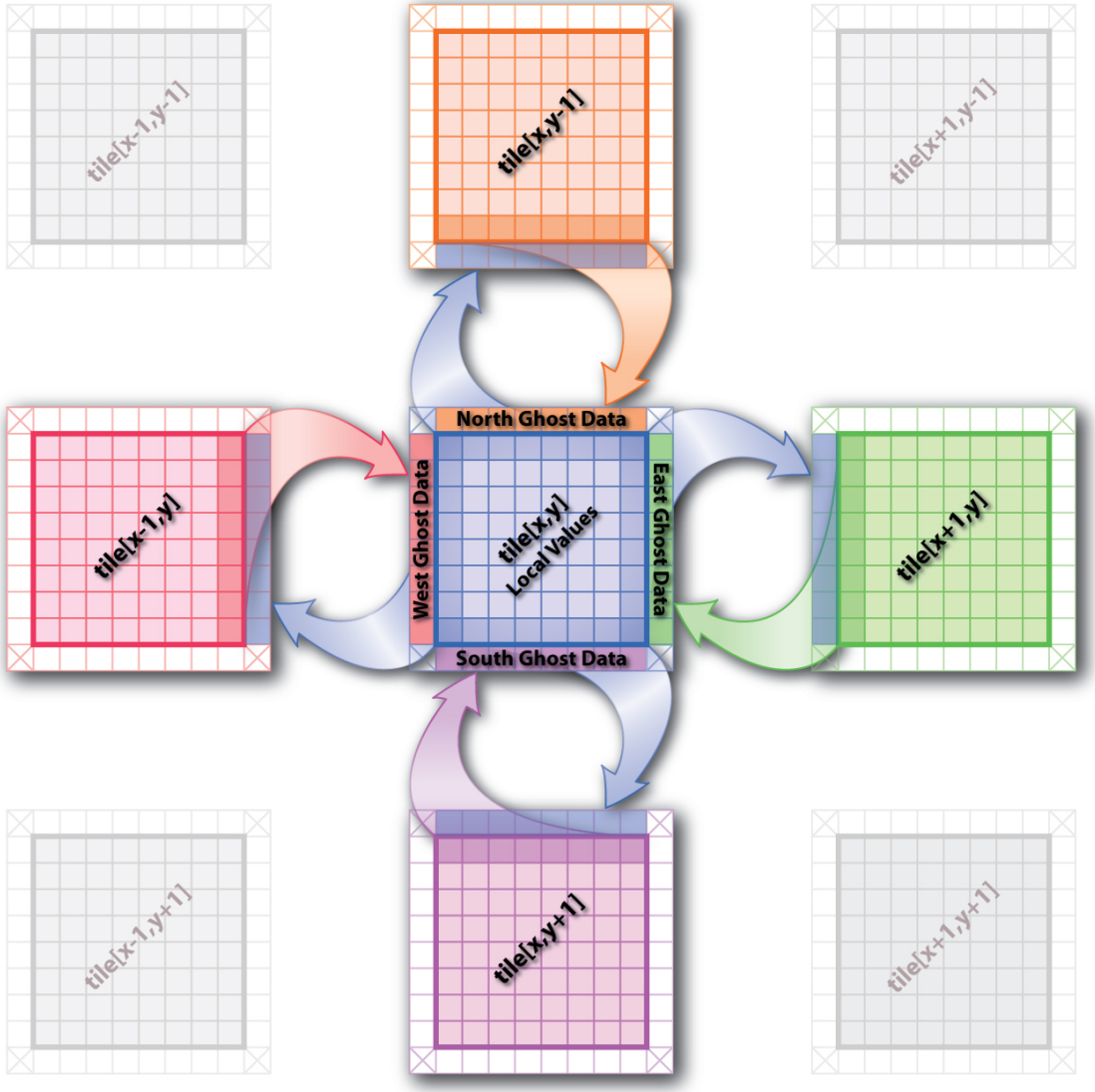
Determinant MP0 Structered Dagger: .cpp file

```
#include "Determinants.decl.h"
#include <cstdlib>
using std::atoi; using std::rand; using std::srand;

struct Main : public CBase_Main {
  Main_SDAG_CODE
  int i, n, m;
  Main(CkArgMsg *msg) {
    __sdag_init();
    if (msg->argc < 3) CkAbort("Usage: det <n> <m>");
    n = atoi(msg->argv[1]); m = atoi(msg->argv[2]);
    srand(29);
    run();
  }
};
struct DeterminantChare : public CBase_DeterminantChare {
  DeterminantChare(CProxy_Main main, int i, int n, int *matrix) {
    int retVal;
    if (n == 2) retVal = matrix[0]*matrix[3] - matrix[1]*matrix[2];
    else if (n == 3)
      retVal = matrix[0]*matrix[4]*matrix[8]
             + matrix[1]*matrix[5]*matrix[6]
             + matrix[2]*matrix[3]*matrix[7]
             - matrix[0]*matrix[5]*matrix[7]
             - matrix[1]*matrix[3]*matrix[8]
             - matrix[2]*matrix[4]*matrix[6];
    else CkAbort("Only supports determinants of size 2 or 3!");
    main.response(i, retVal);
  }
};

#include "Determinants.def.h"
```

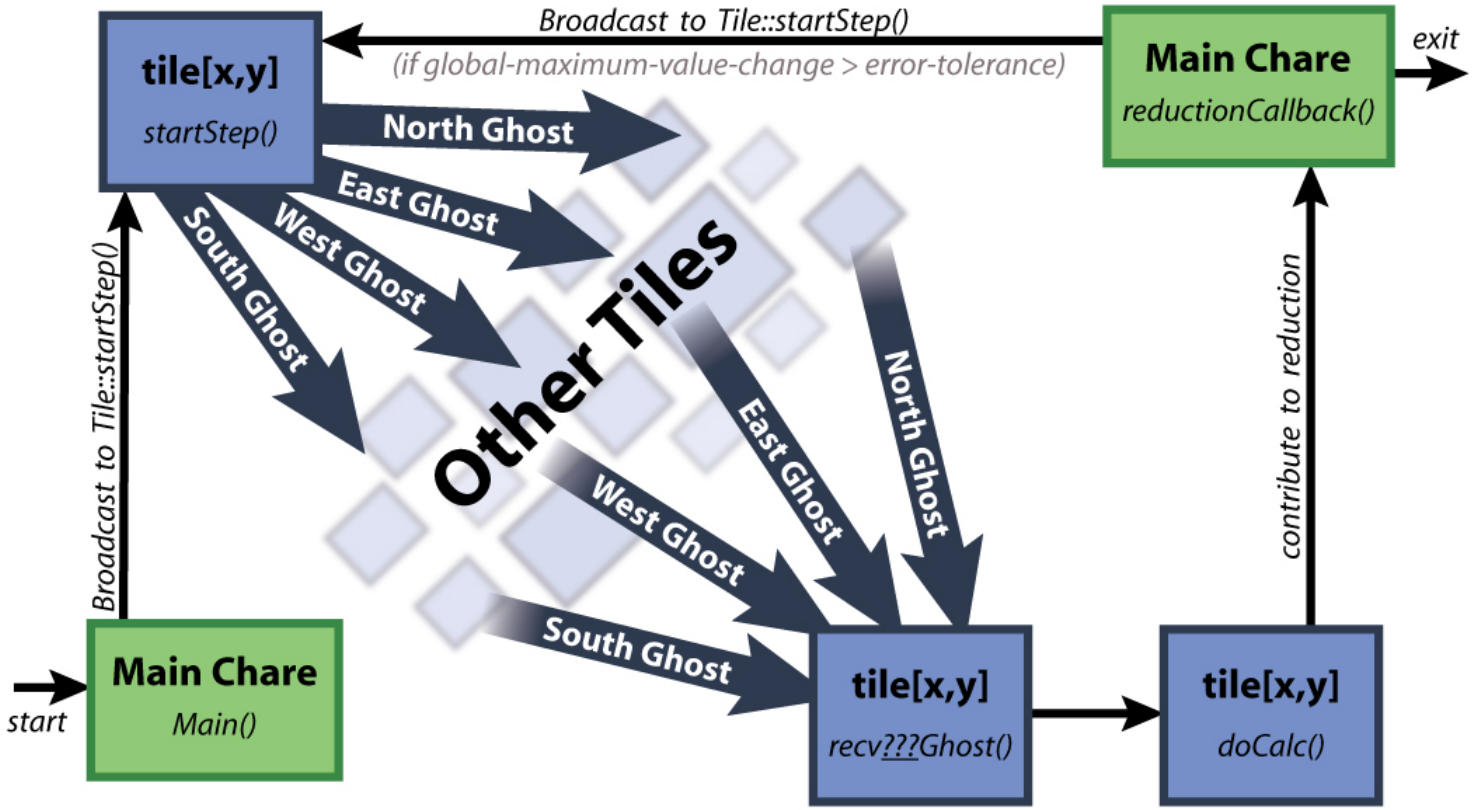
5-point Stencil



5-point Stencil



5-point Stencil



Jacobi: .ci file

```
mainmodule jacobi3d {
  readonly CProxy_Main mainProxy;

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

  array [3D] Jacobi {
    entry Jacobi(void);
    entry void updateGhosts(int ref, int dir, int w, int h, double gh[w*h]);
    entry [reductiontarget] void checkConverged(bool result);
    entry void run() {
      // ... main loop (next slide) ...
    };
  };
};
```

Jacobi: .ci file

```
entry void run() {
  while (!converged) {
    atomic {
      copyToBoundaries();
      int x = thisIndex.x, y = thisIndex.y, z = thisIndex.z;
      int bdX = blockDimX, bdY = blockDimY, bdZ = blockDimZ;
      thisProxy(wrapX(x−1),y,z).updateGhosts(iter, RIGHT, bdY, bdZ, rightGhost);
      thisProxy(wrapX(x+1),y,z).updateGhosts(iter, LEFT, bdY, bdZ, leftGhost);
      thisProxy(x,wrapY(y−1),z).updateGhosts(iter, TOP, bdX, bdZ, topGhost);
      thisProxy(x,wrapY(y+1),z).updateGhosts(iter, BOTTOM, bdX, bdZ, bottomGhost);
      thisProxy(x,y,wrapZ(z−1)).updateGhosts(iter, BACK, bdX, bdY, backGhost);
      thisProxy(x,y,wrapZ(z+1)).updateGhosts(iter, FRONT, bdX, bdY, frontGhost);
      freeBoundaries();
    }
    for (remoteCount = 0; remoteCount < 6; remoteCount++)
      when updateGhosts[iter](int ref, int dir, int w, int h, double buf[w∗h]) atomic {
        updateBoundary(dir, w, h, buf);
      }
    atomic {
      double error = computeKernel();
      int conv = error < DELTA;
      contribute(sizeof(int), &conv, CkReduction::logical_and, CkCallback(CkReductionTarget(Jacobi,
          checkConverged), thisProxy));
    }
    when checkConverged(bool result)
      if (result) atomic { mainProxy.done(iter); converged = true; }
    atomic { ++iter; }
  }
};
```

Jacobi: .ci file (with **asynchronous** reductions)

```
entry void run() {
  while (!converged) {
    atomic {
      copyToBoundaries();
      int x = thisIndex.x, y = thisIndex.y, z = thisIndex.z;
      int bdX = blockDimX, bdY = blockDimY, bdZ = blockDimZ;
      thisProxy(wrapX(x−1),y,z).updateGhosts(iter, RIGHT, bdY, bdZ, rightGhost);
      thisProxy(wrapX(x+1),y,z).updateGhosts(iter, LEFT, bdY, bdZ, leftGhost);
      thisProxy(x,wrapY(y−1),z).updateGhosts(iter, TOP, bdX, bdZ, topGhost);
      thisProxy(x,wrapY(y+1),z).updateGhosts(iter, BOTTOM, bdX, bdZ, bottomGhost);
      thisProxy(x,y,wrapZ(z−1)).updateGhosts(iter, BACK, bdX, bdY, backGhost);
      thisProxy(x,y,wrapZ(z+1)).updateGhosts(iter, FRONT, bdX, bdY, frontGhost);
      freeBoundaries();
    }
    for (remoteCount = 0; remoteCount < 6; remoteCount++)
      when updateGhosts[iter](int ref, int dir, int w, int h, double buf[w∗h]) atomic {
        updateBoundary(dir, w, h, buf);
      }
    atomic {
      double error = computeKernel();
      int conv = error < DELTA;
      if (iter % 5 == 1)
        contribute(sizeof(int), &conv, CkReduction::logical_and, CkCallback(CkReductionTarget(Jacobi,
            checkConverged), thisProxy));
    }
    if (++iter % 5 == 0)
      when checkConverged(bool result)
        if (result) atomic { mainProxy.done(iter); converged = true; }
  }
};
```

Jacobi: .cpp file

```
class Main : public CBase_Main {
public:
  CProxy_Jacobi array;
  int iter;

  Main(CkArgMsg* m) {
    // ... initialization code ...
    // Create new array of worker chares
    array = CProxy_Jacobi::ckNew(num_chare_x, num_chare_y, num_chare_z);

    //Start the computation
    array.run();
    startTime = CkWallTimer();
  }

  void done(int iterations) {
    CkPrintf("Completed %d iterations\n", iterations);
    endTime = CkWallTimer();
    CkPrintf("Time elapsed per iteration: %f\n", (endTime - startTime) / iterations);
    CkExit();
  }
};
```

Jacobi: .cpp file

```
class Jacobi: public CBase_Jacobi {
  Jacobi_SDAG_CODE

public:
  int iter;
  int remoteCount;

  double *temperature;
  double *new_temperature;
  bool converged;
  double *leftGhost, *rightGhost, *topGhost, *bottomGhost, *frontGhost, *backGhost;

  // Constructor, initialize values
  Jacobi() {
    __sdag_init();

    usesAtSync = CmiTrue;
    converged = false;

    // allocate a three dimensional array
    temperature = new double[(blockDimX+2) * (blockDimY+2) * (blockDimZ+2)];
    new_temperature = new double[(blockDimX+2) * (blockDimY+2) * (blockDimZ+2)];

    for(int k=0; k<blockDimZ+2; ++k)
      for(int j=0; j<blockDimY+2; ++j)
        for(int i=0; i<blockDimX+2; ++i)
          temperature[index(i, j, k)] = 0.0;

    iter = 0;
    constrainBC();
  }
```

Jacobi: .cpp file

```
class Jacobi: public CBase_Jacobi {
  Jacobi_SDAG_CODE

public:
  int iter;
  int remoteCount;

  double *temperature;
  double *new_temperature;
  bool converged;
  double *leftGhost, *rightGhost, *topGhost, *bottomGhost, *frontGhost, *backGhost;

  // Constructor, initialize values
  Jacobi() {
    __sdag_init();

    usesAtSync = CmiTrue;
    converged = false;

    // allocate a three dimensional array
    temperature = new double[(blockDimX+2) * (blockDimY+2) * (blockDimZ+2)];
    new_temperature = new double[(blockDimX+2) * (blockDimY+2) * (blockDimZ+2)];

    for(int k=0; k<blockDimZ+2; ++k)
      for(int j=0; j<blockDimY+2; ++j)
        for(int i=0; i<blockDimX+2; ++i)
          temperature[index(i, j, k)] = 0.0;

    iter = 0;
    constrainBC();
  }
```

Jacobi: .cpp file

```
void copyToBoundaries() {
  // Copy different faces into messages
  leftGhost = new double[blockDimY*blockDimZ];
  rightGhost = new double[blockDimY*blockDimZ];
  topGhost = new double[blockDimX*blockDimZ];
  bottomGhost = new double[blockDimX*blockDimZ];
  frontGhost = new double[blockDimX*blockDimY];
  backGhost = new double[blockDimX*blockDimY];

  for(int k=0; k<blockDimZ; ++k)
    for(int j=0; j<blockDimY; ++j) {
      leftGhost[k*blockDimY+j] = temperature[index(1, j+1, k+1)];
      rightGhost[k*blockDimY+j] = temperature[index(blockDimX, j+1, k+1)];
    }

  for(int k=0; k<blockDimZ; ++k)
    for(int i=0; i<blockDimX; ++i) {
      topGhost[k*blockDimX+i] = temperature[index(i+1, 1, k+1)];
      bottomGhost[k*blockDimX+i] = temperature[index(i+1, blockDimY, k+1)];
    }

  for(int j=0; j<blockDimY; ++j)
    for(int i=0; i<blockDimX; ++i) {
      frontGhost[j*blockDimX+i] = temperature[index(i+1, j+1, 1)];
      backGhost[j*blockDimX+i] = temperature[index(i+1, j+1, blockDimZ)];
    }
}
```

Jacobi: .cpp file

```
void updateBoundary(int dir, int height, int width, double* gh) {
  switch(dir) {
  case LEFT:
    for(int k=0; k<width; ++k)
      for(int j=0; j<height; ++j) { temperature[index(0, j+1, k+1)] = gh[k*height+j]; }
    break;
  case RIGHT:
    for(int k=0; k<width; ++k)
      for(int j=0; j<height; ++j) { temperature[index(blockDimX+1, j+1, k+1)] = gh[k*height+j]; }
    break;
  case BOTTOM:
    for(int k=0; k<width; ++k)
      for(int i=0; i<height; ++i) { temperature[index(i+1, 0, k+1)] = gh[k*height+i]; }
    break;
  case TOP:
    for(int k=0; k<width; ++k)
      for(int i=0; i<height; ++i) { temperature[index(i+1, blockDimY+1, k+1)] = gh[k*height+i]; }
    break;
  case FRONT:
    for(int j=0; j<width; ++j)
      for(int i=0; i<height; ++i) { temperature[index(i+1, j+1, 0)] = gh[j*height+i]; }
    break;
  case BACK:
    for(int j=0; j<width; ++j)
      for(int i=0; i<height; ++i) { temperature[index(i+1, j+1, blockDimZ+1)] = gh[j*height+i]; }
    break;
  default:
    CkAbort("ERROR\n");
  }
}
```