Charm++ Tutorial

Presented by Phil Miller

2012-04-30

Outline

- Basics
  - Introduction
  - Charm++ Objects
  - Chare Arrays
  - Chare Collectives
  - SDAG
  - Example
- Intermission
- Advanced
  - Prioritized Messaging
  - Interface file tricks
    - Initialization
    - Entry Method Tags
  - Groups & Node Groups
  - Threads

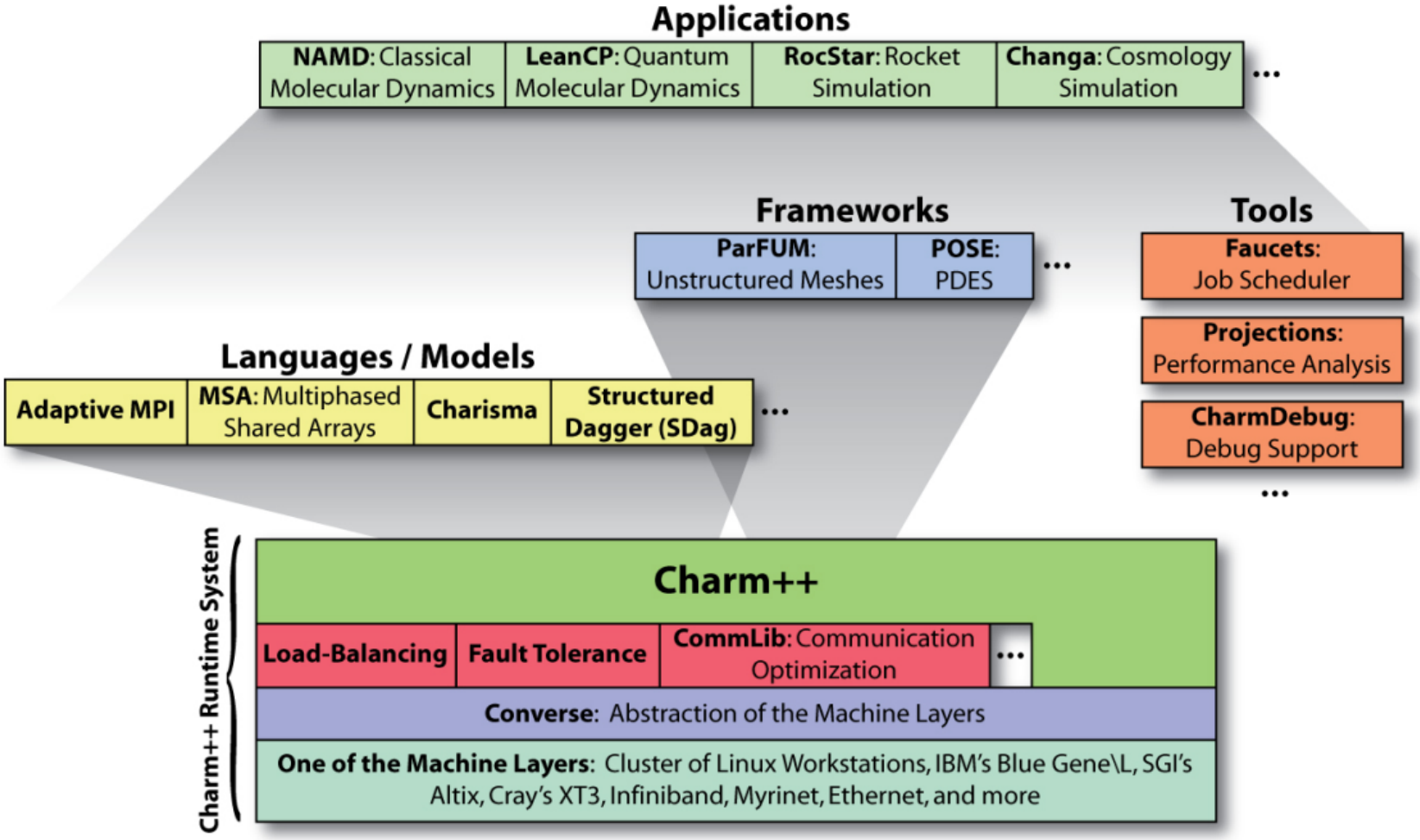
Expectations

- Introduction to Charm++
  - Assumes parallel programming aware audience
  - Assume C++ aware audience
  - AMPI not covered
- Goals
  - What Charm++ is
  - How it can help
  - How to write a basic charm program
  - Provide awareness of advanced features

What Charm++ Is Not

- Not Magic Pixie Dust
  - Runtime system exists to help you
  - Decisions and customizations are necessary in proportion to the complexity of your application
- Not a language
  - Platform independent library with a semantic
  - Works for C, C++, Fortran (not covered in this tutorial)
- Not a Compiler
- Not SPMD Model
- Not Processor Centric Model
  - Decompose to individually addressable medium grain tasks
- Not A Thread Model
  - They are available if you want to inflict them on your code
- Not Bulk Synchronous

Charm++ Ecosystem

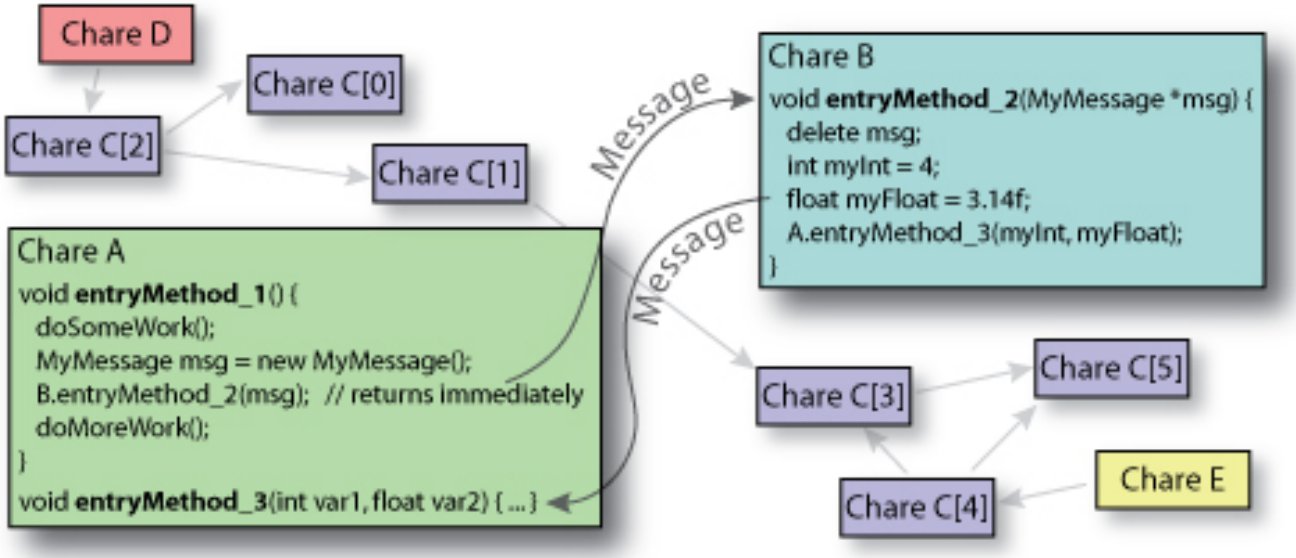


The Charm++ Model

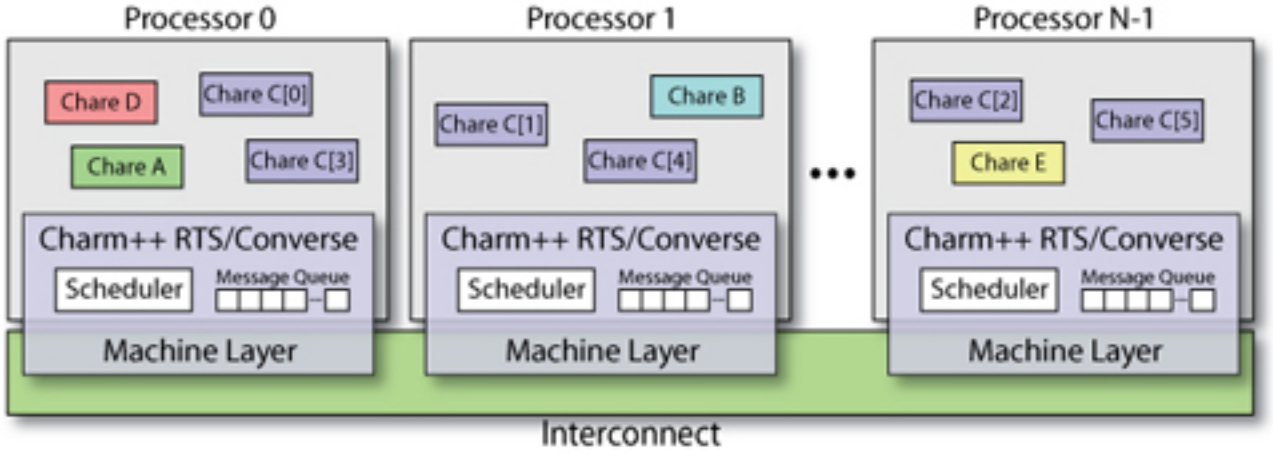
- Parallel objects (chares) communicate via asynchronous method invocations (entry methods).
- The runtime system maps chares onto processors and schedules execution of entry methods.
- Similar to Active Messages or Actors

User View vs. System View

User View:



System View:



Architectures

- Runs on:
  - Clusters with Ethernet (UDP/TCP)
  - Clusters with Infiniband
  - Clusters with accelerators (GPU/CELL)
  - IBM & Cray Supercomputers
  - Windows
  - Any machine with MPI installation
  - …
- To install
  - “./build”

Portability

Cray XT (3|4|5|6)
- Kraken

Cray X(E|K) 6
- Titan, Blue Waters

BlueGene (L|P|Q)
- Intrepid, Mira

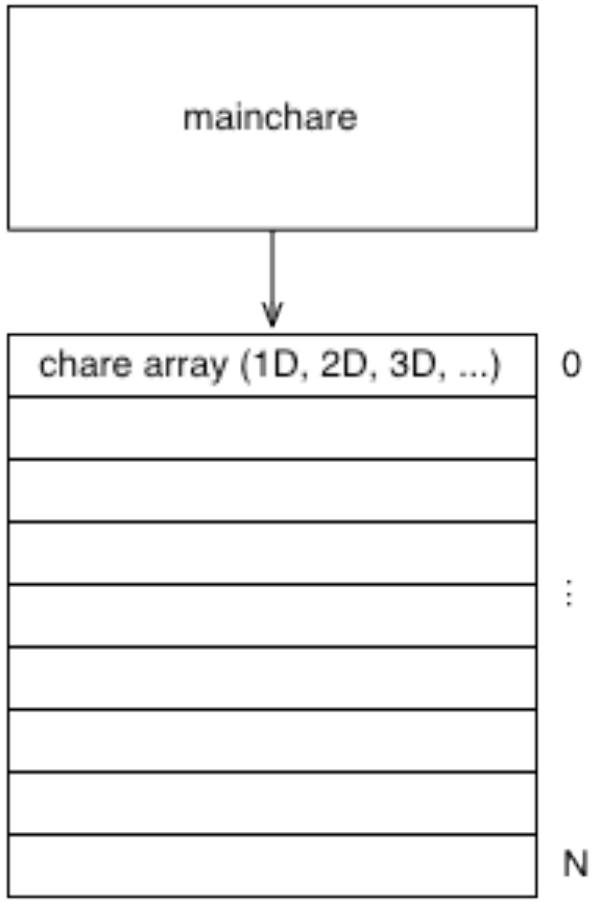
SGI/Altix
- Ember

Clusters
- X86_64, POWER
- MPI, UDP, TCP, LAPI, Infiniband

Accelerators
- GPGPU
- Cell

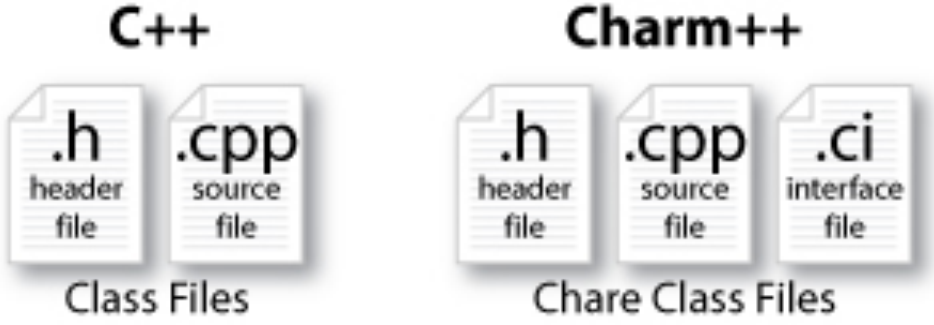
Charm++ Objects

- 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 and entry methods (methods that can be called asynchronously and remotely)
  - Defined in a `.ci` (Charm interface) file
  - And are implemented in the `.cpp` file



Hello World: .ci file

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

- .ci: Charm Interface
- Defines which type of chares are present in the application
  - At least a *mainchare* must be set
- Each definition is inside a *module*

Hello World: the code

Interface

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

Implementation

```
#include “hello.decl.h”

class Main : public CBase_Main {
  public:
  Main(CkArgMsg* m) {
    CkPrintf(“Hello World!\n”);
    CkExit();
  }
};

#include “hello.def.h”
```

CkArgMsg in the Main::Main Method

- Defined in charm++
- struct CkArgMsg {

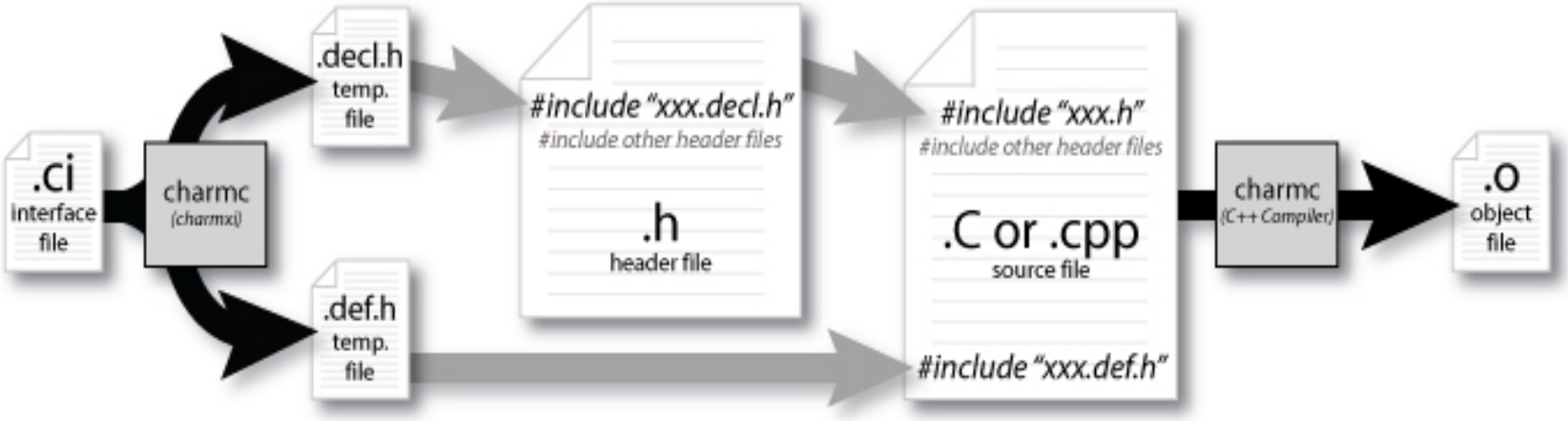
```
  int argc;
  char **argv;
};
```

Compilation Process

- charmc hello.ci
- charmc –o main.o main.C # compile

link

- charmc –language charm++ -o pgm main.o

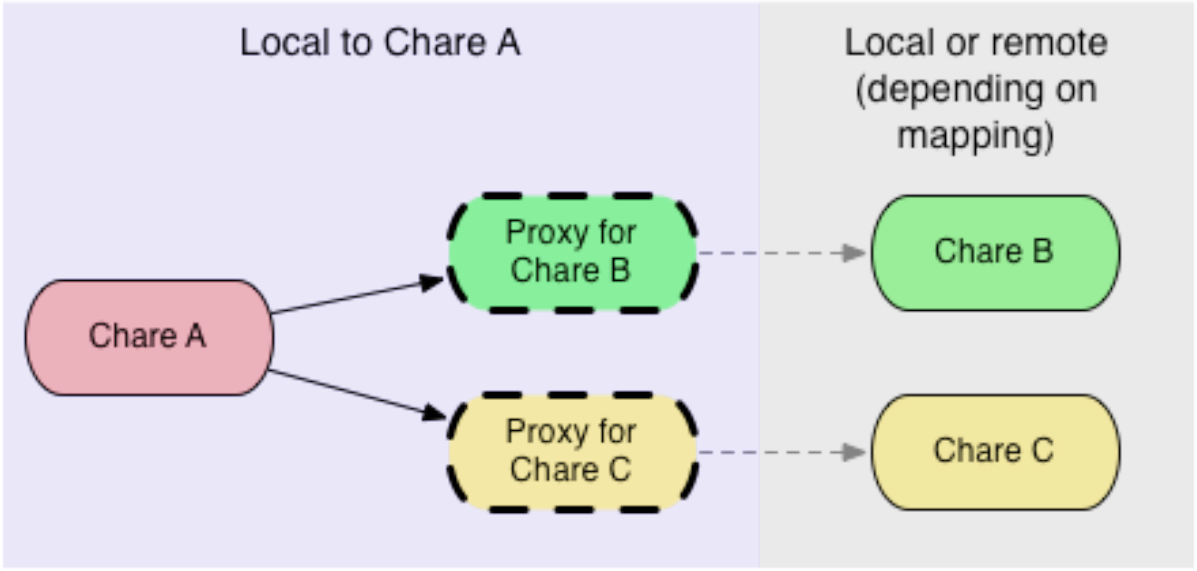


Execution

- `./charmrun +p4 ./pgm`
  - Or specific queueing system
- Output:
  - *Hello World!*
- Not a parallel code :(
  - Solution: create other chares, all of them saying “Hello World”

How to Communicate?

- Chares spread across multiple processors
  - It is not possible to directly invoke methods
- Use of Proxies – lightweight handles to potentially remote chares



The Proxy

- A *proxy* class is generated for every chare
  - For example, `CProxy_Main` is the proxy generated for the `chare Main`
  - Proxies know where to find a chare in the system
  - Methods invoked on a Proxy pack the input parameters, and send them to the processor where the chare is. The real method will be invoked on the destination processor.
- Given a proxy `p`, it is possible to call the method
  - `p.method(msg)`

Hello World with a New Chare

Program's asynchronous flow

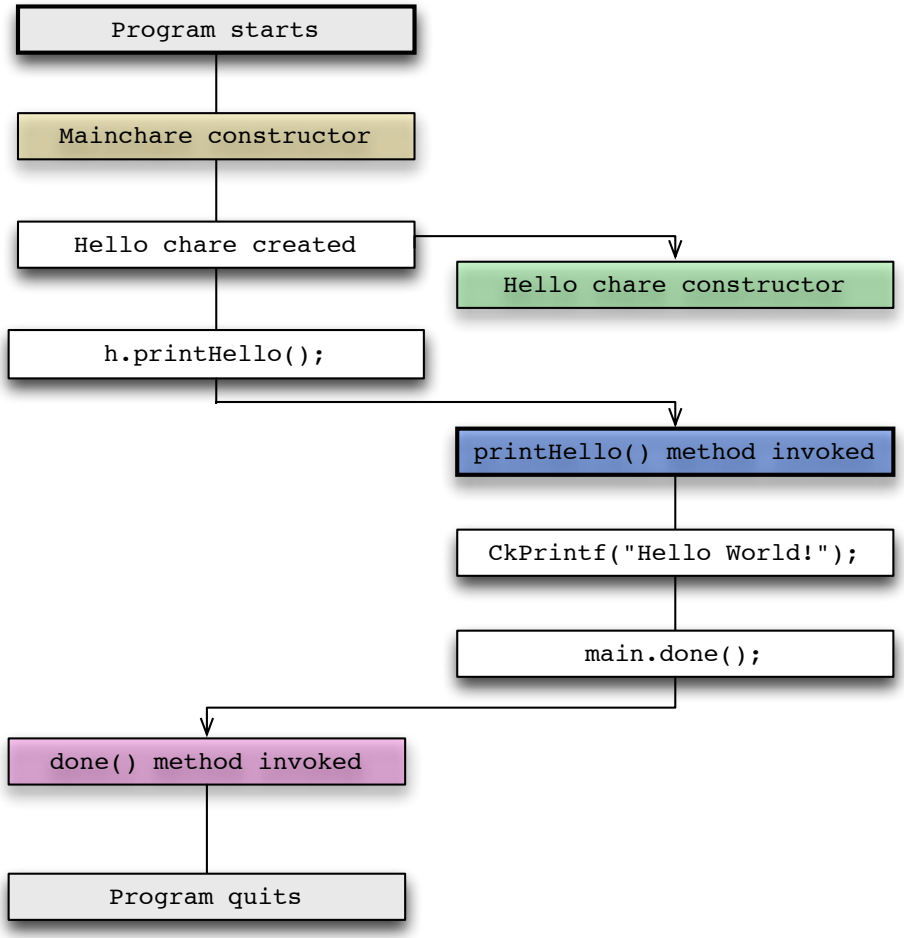
- Mainchare sends message to `Hello` object
- `Hello` object prints "*Hello World!*"
- `Hello` object sends message back to the mainchare
- Mainchare quits the application

Code

```
mainmodule hello {
  mainchare Main {
    entry Main(CkArgMsg* m);
    entry void done();
  };
  chare Hello {
    entry Hello(CProxy_Main main);
    entry void printHello();
  };
};
```

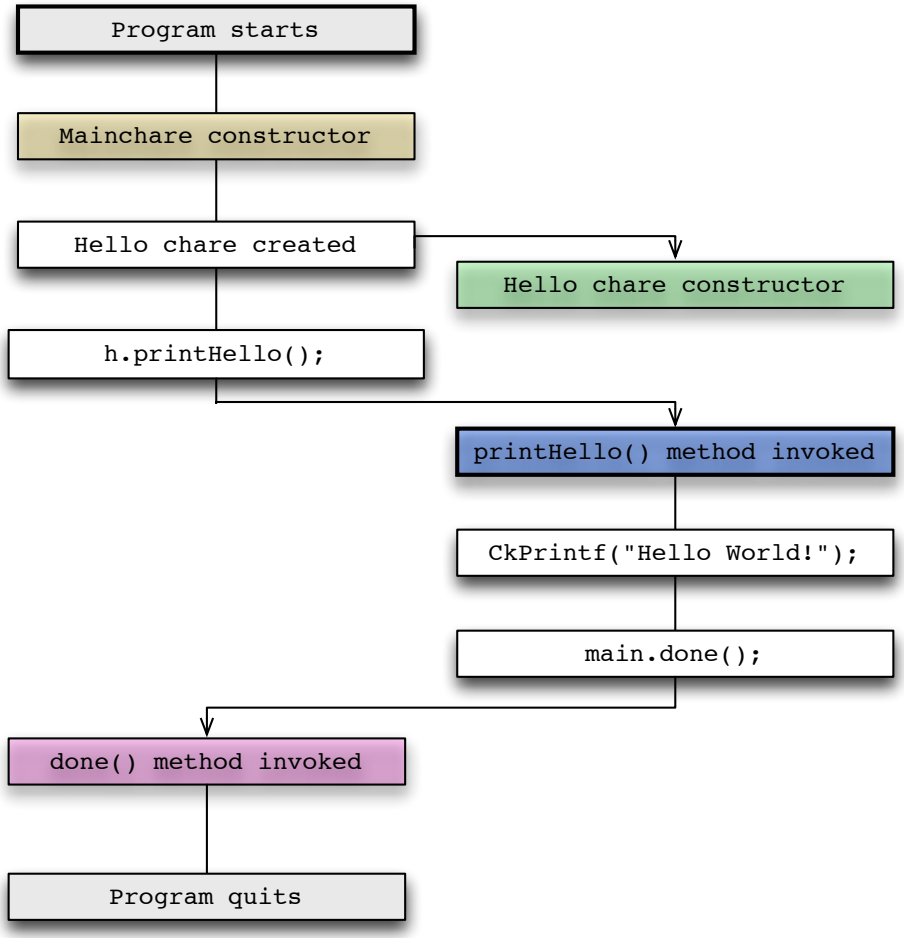
```
#include “hello.decl.h”
struct Main : public CBase_Main {
  Main(CkArgMsg* m) {
    delete m;
    CProxy_Hello  h =
      CProxy_Hello::ckNew(thisProxy);
    h.printHello();
  }
  void done() { CkExit(); }
};
struct Hello : public CBase_Hello {
  CBase_Main main;
  Hello(CProxy_Main main_)
    : main(main_) { }
  void printHello() {
    CkPrintf(“Hello World!\n”);
    main.done();
  }
};
#include “hello.def.h”
```

Workflow of Hello World



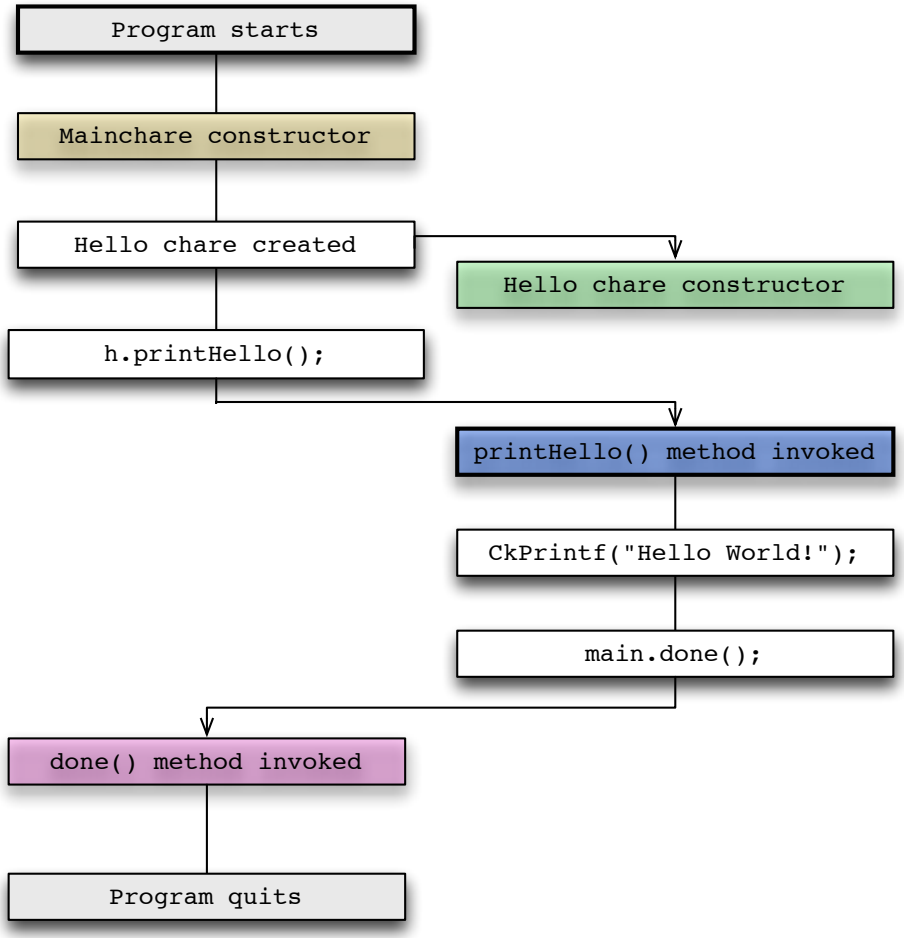

```
#include “hello.decl.h”
struct Main : public CBase_Main {
  Main(CkArgMsg* m) {
    delete m;
    CProxy_Hello  h =
      CProxy_Hello::ckNew(thisProxy);
    h.printHello();
  }
  void done() { CkExit(); }
};
struct Hello : public CBase_Hello {
  CBase_Main main;
  Hello(CProxy_Main main_)
    : main(main_) { }
  void printHello() {
    CkPrintf(“Hello World!\n”);
    main.done();
  }
};
#include “hello.def.h”
```

Workflow of Hello World



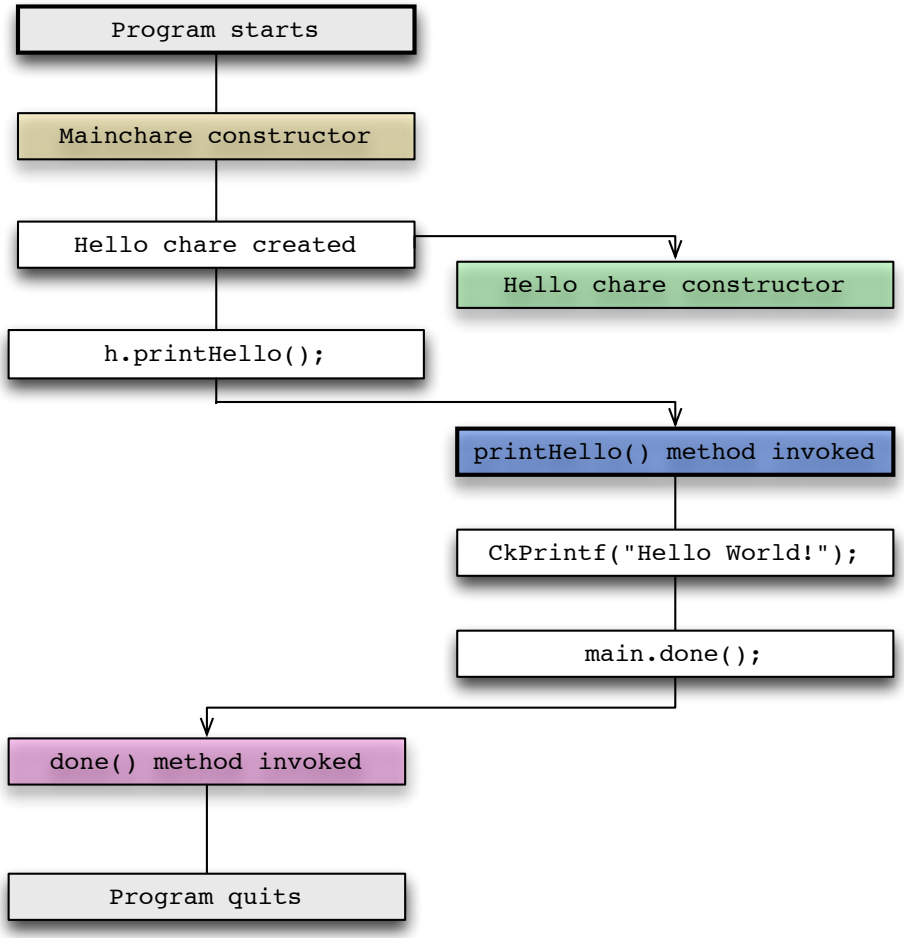

```
#include “hello.decl.h”
struct Main : public CBase_Main {
  Main(CkArgMsg* m) {
    delete m;
    CProxy_Hello  h =
      CProxy_Hello::ckNew(thisProxy);
    h.printHello();
  }
  void done() { CkExit(); }
};
struct Hello : public CBase_Hello {
  CBase_Main main;
  Hello(CProxy_Main main_)
    : main(main_) { }
  void printHello() {
    CkPrintf(“Hello World!\n”);
    main.done();
  }
};
#include “hello.def.h”
```

Workflow of Hello World




```
#include “hello.decl.h”
struct Main : public CBase_Main {
  Main(CkArgMsg* m) {
    delete m;
    CProxy_Hello  h =
      CProxy_Hello::ckNew(thisProxy);
    h.printHello();
  }
  void done() { CkExit(); }
};
struct Hello : public CBase_Hello {
  CBase_Main main;
  Hello(CProxy_Main main_)
    : main(main_) { }
  void printHello() {
    CkPrintf(“Hello World!\n”);
    main.done();
  }
};
#include “hello.def.h”
```

Workflow of Hello World




```
#include “hello.decl.h”
struct Main : public CBase_Main {
  Main(CkArgMsg* m) {
    delete m;
    CProxy_Hello  h =
      CProxy_Hello::ckNew(thisProxy);
    h.printHello();
  }
  void done() { CkExit(); }
};
struct Hello : public CBase_Hello {
  CBase_Main main;
  Hello(CProxy_Main main_)
    : main(main_) { }
  void printHello() {
    CkPrintf(“Hello World!\n”);
    main.done();
  }
};
#include “hello.def.h”
```

Limitations of Plain Proxies

- In a large program, keeping track of all the proxies is difficult
- A simple proxy doesn't tell you anything about the chare other than its type.
- Managing collective operations like broadcast and reduce is complicated.

Chare Arrays

- *Chare Arrays* organize chares into *indexed collections.*
- One single name for the whole collection
- Each chare in the array has a proxy for *all* the other array elements, accessible using simple syntax
  - `sampleArray[i]  // i'th proxy`
  - `sampleArray(i, j) // (i, j) proxy`

Array Dimensions

- Anything can be used as array indices
  - integers
  - Tuples (e.g., 2D, 3D array)
  - bit vectors
  - user-defined types
- Dense or sparse in index space
- Can insert and delete elements on the fly

Array Elements Mapping

- Automatically by the runtime system
- Programmer could control the mapping of array elements to PEs.
  - Round-robin, block-cyclic, etc
  - User defined mapping

Broadcasts

- Simple way to invoke the same entry method on each array element.
- Example: A 1D array “Cproxy_MyArray arr”
  - arr[3].method(): a point-to-point message to element 3.
  - arr.method(): a ***broadcast*** message to every elements

Hello World: Array Version

```
mainmodule hello {
  mainchare Main {
    entry Main(CkArgMsg* m);
    entry void done();
  };
  array [1D] Hello {
    entry Hello(CProxy_Main m,
                int n);
    entry void printHello();
  };
};
```

```
struct Hello : public CBase_Hello {
  CBase_Main main;
  int numChares;
  Hello(CProxy_Main m, int n)
    : main(m), numChares(n)
   { }
  void printHello() {
    CkPrintf("Hello World from %s!\n",
             thisIndex);
    if (thisIndex < numChares - 1)
      thisProxy[thisIndex+1].printHello();
    else
      main.done();
  }
};
```

Result

Running "Hello World" with 10 elements using 3 processors.

$ ./charmrun +p3 ./hello 10

Hello world from 0!

Hello world from 1!

Hello world from 2!

Hello world from 3!

Hello world from 4!

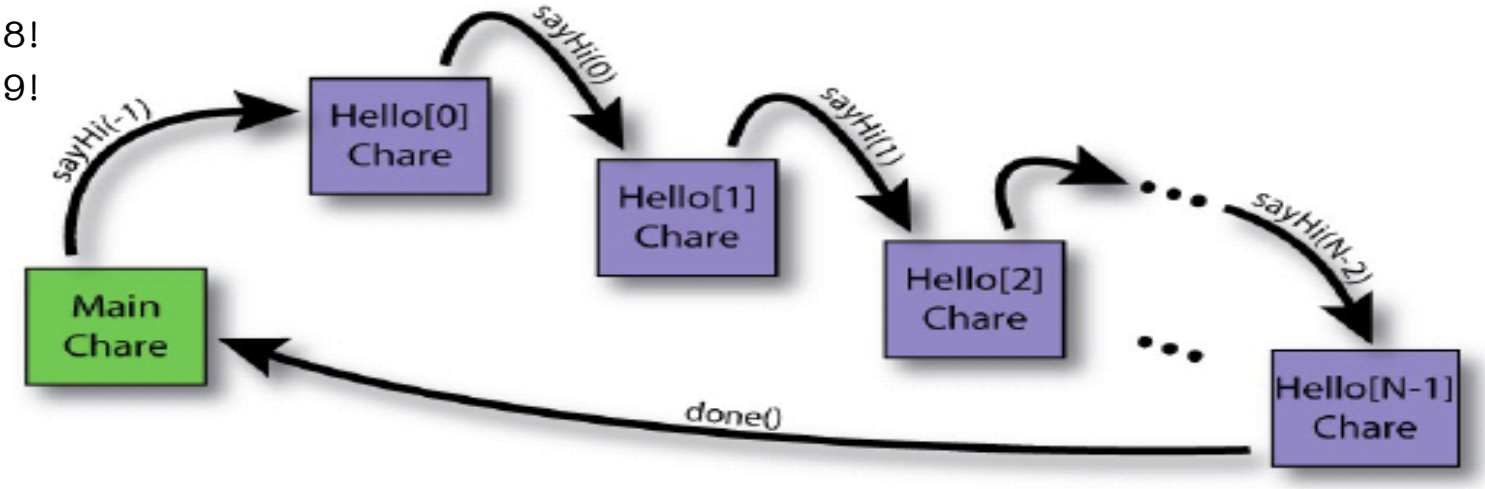
Hello world from 5!

Hello world from 6!

Hello world from 7!

Hello world from 8!

Hello world from 9!



readonly Variables

- Defines a global constant
  - Everyone gets its value
- Must be set only in the mainchare

```
mainmodule hello {
  readonly CProxy_Main main;
  readonly int numChares;
  mainchare Main {
    entry Main(CkArgMsg* m);
    entry void done();
  };
  array [1D] Hello {
    entry Hello();
    entry void printHello();
  };
};
```

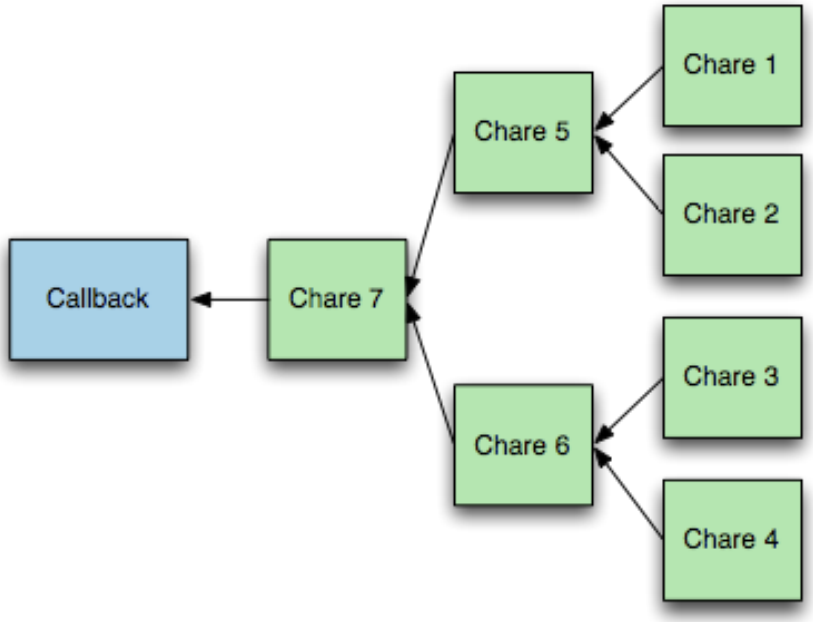
```
CBase_Main main;
int numChares;

Main::Main(CkArgMsg *m) {
  numChares = atoi(m->argv[1]);
  main = thisProxy;

  CProxy_Hello h = CProxy_Hello::ckNew();
  h.printHello();
  delete m;
}
```

Reductions

- Every chare element will contribute its portion of data to someone, and data are combined through a particular operation.
- Naïve way:
  - Use a “master” to count how many messages need to be received.
  - Potential bottleneck on the “master”

Reductions

- Runtime system builds reduction tree
- User specifies reduction *operation*
- At root of tree, a *callback* is triggered



Reduction in Charm++

- No global flow of control, so each chare must contribute data independently:

```
void contribute(int nBytes, const void *data,
                CkReduction::reducerType type,
                CkCallback cb);
```

- Callback `cb` is invoked when the reduction is complete.

Reduction Operations

- Predefined instances of CkReduction::reducerType:
  - Arithmetic (int, float, double)
    - CkReduction::sum_int, CkReduction::sum_double, ...
    - CkReduction::product_int, ...
    - CkReduction::max_int, ...
    - CkReduction::min_int, ...
  - Logic:
    - CkReduction::logical_and, logic_or
    - CkReduction::bitvec_and, bitvec_or
  - Gather:
    - CkReduction::set, concat
  - Misc:
    - CkReduction::random
- Defined by the user

Callback: where do reductions go?

- CkCallback(CkCallbackFn fn, void *param)
  - void myCallbackFn(void *param, void *msg)
- CkCallback(int ep, const CkChareID &id)
  - ep=CkReductionTarget(ChareName, EntryMethod)
- CkCallback(int ep, const CkArrayID &id)
  - The callback will be called on all array elements
- CkCallback(int ep, const CkArrayIndex &idx, const CkArrayID &id)
  - The callback will only be called on element[idx]
- CkCallback(CkCallback::ckExit)

Example

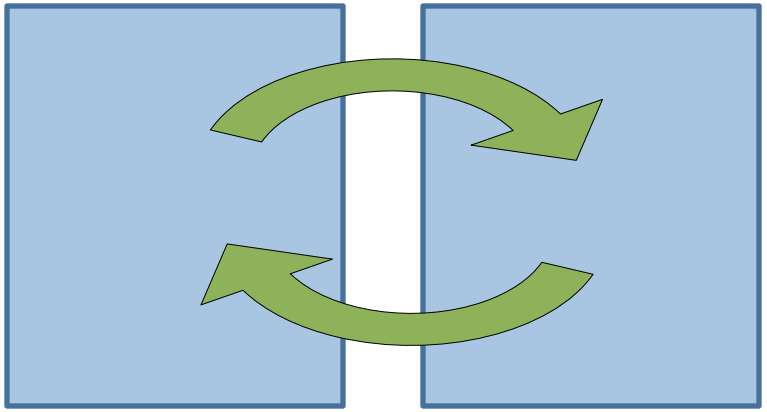
Sum local error estimators to determine global error

```
CkCallback cb (CkReductionTarget(Main, computeGlobalError),
                mainProxy);

contribute(sizeof(double), &myError, CkReduction::sum_double, cb);


// in .ci file for Main:
// . . . .
entry void [reductiontarget] computeGlobalError(double error);
```

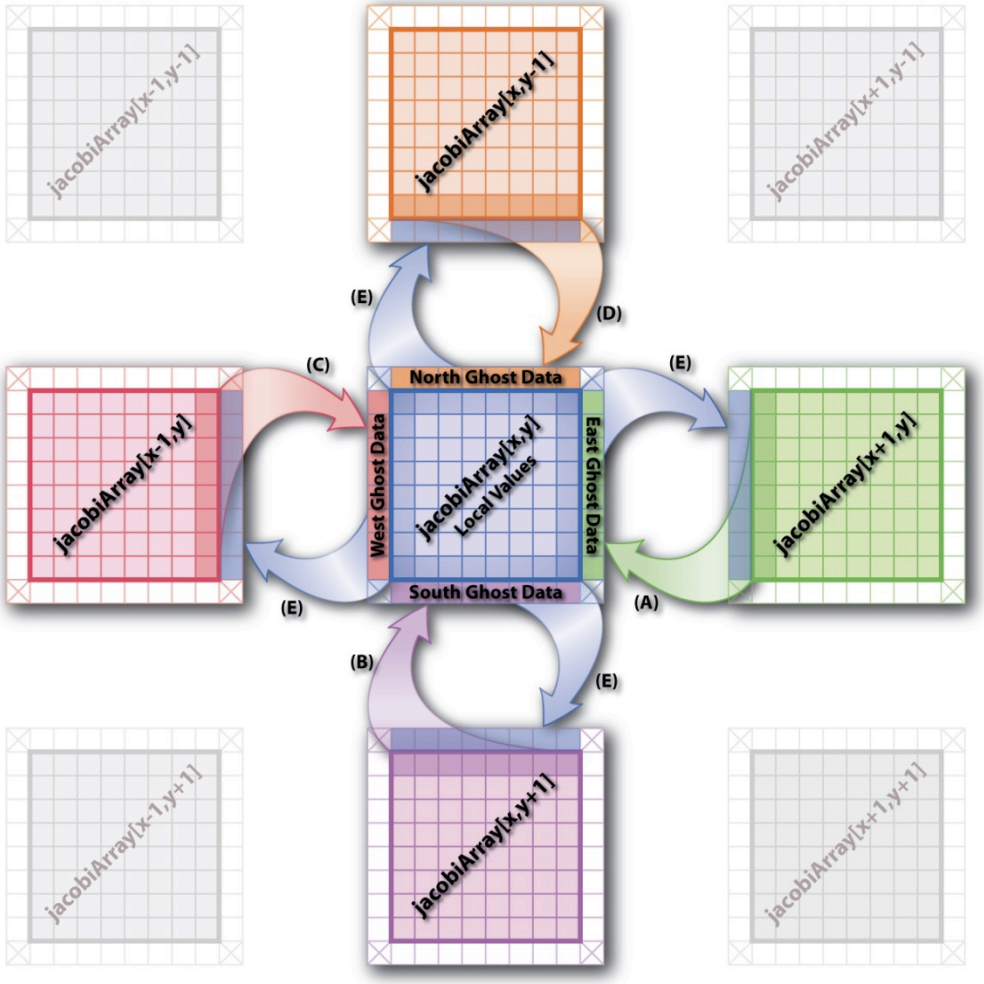
Example: Jacobi (Stencil)

Use two interchangeable matrices

```
do {
  computeKernel();
  maxDiff = max(abs (A - B));
} while (maxDiff > DELTA);

computeKernel() {
  foreach i,j {
    B[i,j] = (A[i,j] +
              A[i+1,j] +
              A[i-1,j] +
              A[i,j+1] +
              A[i,j-1]) / 5;
  }
  swap (A, B);
}
```



Jacobi in parallel



Control Flow using SDAG

- Structured DAGger
  - Directed Acyclic Graph (DAG)
- Express event sequencing and dependency
- Automate Message buffering
- Automate Message counting
- Express independence for overlap
- Differentiate between parallel and sequential blocks
- Negligible overhead

Structured Dagger Constructs

when <method list> {code}

Do not continue until method is called

Internally generates flags, checks, etc.

atomic {code}

Call ordinary sequential C++ code

if/else/for/while

C-like control flow

overlap {code1 code2 ...}

Execute code segments in parallel

forall

"Parallel Do"

Like a parameterized overlap

Jacobi Example

```
while (!converged) {
  atomic {
    int x = thisIndex.x, y = thisIndex.y, z = thisIndex.z;
    copyToBoundaries();
    thisProxy(wrapX(x-1),y,z).updateGhosts(i, RIGHT, dimY, dimZ, right);
    /* ...similar calls to send the 6 boundaries... */
    thisProxy(x,y,wrapZ(z+1)).updateGhosts(i, FRONT, dimX, dimY, front);
  }
  for (remoteCount = 0; remoteCount < 6; remoteCount++) {
    when updateGhosts[i](int i, int d, int w, int h, double b[w*h])
    atomic { updateBoundary(d, w, h, b); }
  }
  atomic {
    int c = computeKernel() < DELTA;
    CkCallback cb(CkReductionTarget(Jacobi, checkConverged), thisProxy);
    if (i % 5 == 1) contribute(sizeof(int), &c, CkReduction::logical_and, cb);
  }
  if (++i % 5 == 0) {
    when checkConverged(bool result) atomic {
      if (result) { mainProxy.done(); converged = true; }
    }
  }
}
```

Jacobi Example

```
while (!converged) {
  atomic {
    int x = thisIndex.x, y = thisIndex.y, z = thisIndex.z;
    copyToBoundaries();
    thisProxy(wrapX(x-1),y,z).updateGhosts(i, RIGHT, dimY, dimZ, right);
    /* ...similar calls to send the 6 boundaries... */
    thisProxy(x,y,wrapZ(z+1)).updateGhosts(i, FRONT, dimX, dimY, front);
  }
  for (remoteCount = 0; remoteCount < 6; remoteCount++) {
    when updateGhosts[i](int i, int d, int w, int h, double b[w*h])
    atomic { updateBoundary(d, w, h, b); }
  }
  atomic {
    int c = computeKernel() < DELTA;
    CkCallback cb(CkReductionTarget(Jacobi, checkConverged), thisProxy);
    if (i % 5 == 1) contribute(sizeof(int), &c, CkReduction::logical_and, cb);
  }
  if (i % lbPeriod == 0) { atomic { AtSync(); } when ResumeFromSync() {} }
  if (++i % 5 == 0) {
    when checkConverged(bool result) atomic {
      if (result) { mainProxy.done(); converged = true; }
    }
  }
}
```

Jacobi Example

```
while (!converged) {
  atomic {
    int x = thisIndex.x, y = thisIndex.y, z = thisIndex.z;
    copyToBoundaries();
    thisProxy(wrapX(x-1),y,z).updateGhosts(i, RIGHT, dimY, dimZ, right);
    /* ...similar calls to send the 6 boundaries... */
    thisProxy(x,y,wrapZ(z+1)).updateGhosts(i, FRONT, dimX, dimY, front);
  }
  for (remoteCount = 0; remoteCount < 6; remoteCount++) {
    when updateGhosts[i](int i, int d, int w, int h, double b[w*h])
    atomic { updateBoundary(d, w, h, b); }
  }
  atomic {
    int c = computeKernel() < DELTA;
    CkCallback cb(CkReductionTarget(Jacobi, checkConverged), thisProxy);
    if (i % 5 == 1) contribute(sizeof(int), &c, CkReduction::logical_and, cb);
  }
  if (i % lbPeriod == 0) { atomic { AtSync(); } when ResumeFromSync() {} }
  if (i % checkpointPeriod == 0) {
    atomic {CkStartMemCheckpoint(CkCallback(CkIndex_Jacobi::cpDone(), thisProxy));}
    when cpDone() { }
  }
  // . . .
```

Advanced Messaging

Prioritized Execution

- **Charm++ scheduler**
  - **Default - FIFO (oldest message)**
- **Prioritized execution**
  - **If several messages available, Charm will process the messages in the order of their priorities**
- **Very useful for speculative work, ordering timestamps, etc...**

Prioritized Messages

- **Number of priority bits passed during message allocation**

```
FooMsg * msg = new (size, nbits) FooMsg;
```

  - **Priorities stored at the end of messages**

- **Signed integer priorities**

```
*CkPriorityPtr(msg)=-1;
CkSetQueueing(msg, CK_QUEUEING_IFIFO);
```

- **Unsigned bitvector priorities**

```
CkPriorityPtr(msg)[0]=0x7fffffff;
CkSetQueueing(msg, CK_QUEUEING_BFIFO);
```

Prioritized Marshalled Messages

- **Pass "CkEntryOptions" as last parameter**
- **For signed integer priorities:**

```
CkEntryOptions opts;
opts.setPriority(-1);
fooProxy.bar(x,y,opts);
```

- **For bitvector priorities:**

```
CkEntryOptions opts;
unsigned int prio[2]={0x7FFFFFFF,0xFFFFFFFF};
opts.setPriority(64,prio);
fooProxy.bar(x,y,opts);
```

Advanced Message Features

- **Nokeep (Read-only) messages**
  - **Entry method agrees not to modify or delete the message**
  - **Avoids message copy for broadcasts, saving time**
- **Inline messages**
  - **Direct method invocation if on local processor**
- **Expedited messages**
  - **Message do not go through the charm++ scheduler (ignore any Charm++ priorities)**
- **Immediate messages**
  - **Entries are executed in an interrupt or the communication thread**
  - **Very fast, but tough to get right**
  - **Immediate messages only currently work for NodeGroups and Group (non-smp)**

Groups/Node Groups

Groups and Node Groups

- Groups
  - Similar to arrays: Broadcasts, reductions, indexing
  - Exactly one representative on each processor
    - Ideally suited for system libraries, caches, etc.
- Node Groups
  - One per OS process

Declarations

- **.ci file**

```
group mygroup {
    entry mygroup(); //Constructor
    entry void foo(foomsg *); //Entry method
 };
 nodegroup mynodegroup {
    entry mynodegroup(); //Constructor
    entry void foo(foomsg *); //Entry method
 };
```

- **C++ file**

```
class mygroup : public CBase_mygroup {
    mygroup() {}
  void foo(foomsg *m) { CkPrintf("Do Nothing");}
};
class mynodegroup : public CBase_mynodegroup {
    mynodegroup() {}
  void foo(foomsg *m) { CkPrintf("Do Nothing");}
};
```

Creating and Calling Groups

- **Creation**

```
p = CProxy_mygroup::ckNew();
```

- **Remote invocation**

```
p.foo(msg);    //broadcast
p[1].foo(msg);  //asynchronous
p.foo(msg, npes, pes); // list send
```

- **Direct local access**

```
mygroup *g=p.ckLocalBranch();
g->foo(….); //local invocation
```

  - **Danger: if you migrate, the group stays behind!**

Thank You!

Free source, binaries, manuals, and more information at:

http://charm.cs.illinois.edu/

Parallel Programming Lab at University of Illinois