Hello World Example

- `hello.ci` file

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

- `hello.cpp` file

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

class Main : public CBase_Main {
  public: Main(CkArgMsg* m) {
    ckout << "Hello World!" << endl;
    CkExit();
  };
};

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

Hello World with Chares

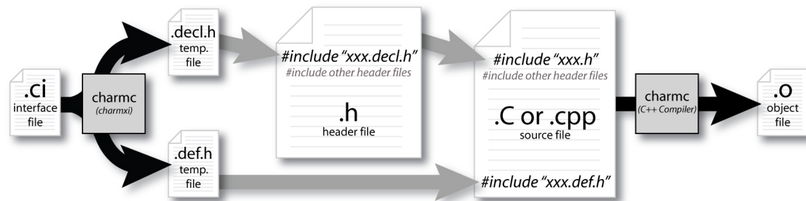
`hello.ci` file

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

`hello.cpp` file

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

class Main : public CBase_Main {
  public: Main(CkArgMsg* m) {
    CProxy_Singleton::ckNew();
  };
};

class Singleton : public
      CBase_Singleton {
  public: Singleton() {
    ckout << "Hello World!" << endl;
    CkExit();
  };
};
#include "hello.def.h"
```

Compiling a Charm++ Program



Building Charm++

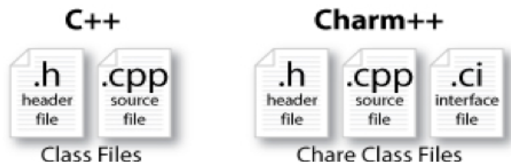
- git clone http://charm.cs.uiuc.edu/gerrit/charm
- ./build <TARGET> <ARCH> <OPTS>
- TARGET = Charm++, AMPI, bgampi, LIBS etc.
- ARCH = net-linux-x86_64, multicore-darwin-x86_64, pamilrts-bluegeneq etc.
- OPTS = –with-production, –enable-tracing, xlc, smp, -j8 etc.
- http://charm.cs.illinois.edu/manuals/html/charm++/A.html

Hello World Example

- Compiling
  - `charmc hello.ci`
  - `charmc -c hello.C`
  - `charmc -o hello hello.o`
- Running
  - `./charmrun +p7 ./hello`
  - The +p7 tells the system to use seven cores

Charm++ File structure

- C++ objects (including Charm++ objects)
  - Defined in regular `.h` and `.C` files
- Chare objects, entry methods (asynchronous methods)
  - Defined in `.ci` file
  - Implemented in the `.C` file



Charm Interface: Modules

- Charm++ programs are organized as a collection of modules
- Each module has one or more chares
- The module that contains the *mainchare*, is declared as the `mainmodule`
- Each module, when compiled, generates two files: `MyModule.decl.h` and `MyModule.def.h`

.ci file

```
[main]module MyModule {
  //... chare definitions ...
};
```

Charm Interface: Chares

- Chares are parallel objects that are managed by the RTS
- Each chare has a set *entry methods*, which are asynchronous methods that may be invoked remotely
- The following code, when compiled, generates a C++ class `CBase_MyChare` that encapsulates the RTS object
- This generated class is extended and implemented in the `.C` file

`.ci file`

```
[main]chare MyChare {
  //... entry method definitions ...
};
```

`.C file`

```
class MyChare : public CBase_MyChare {
  //... entry method implementations ...
};
```

Charm Interface: Entry Methods

- Entry methods are C++ methods that can be remotely and asynchronously invoked by another chare

`.ci` file:

```
entry MyChare(); /* constructor entry method */
entry void foo();
entry void bar(int param);
```

`.C` file:

```
MyChare::MyChare() { /*... constructor code ...*/ }

MyChare::foo() { /*... code to execute ...*/ }

MyChare::bar(int param) { /*... code to execute ...*/ }
```

Charm Interface: `mainchare`

- Execution begins with the `mainchare`'s constructor
- The `mainchare`'s constructor takes a pointer to system-defined class `CkArgMsg`
- `CkArgMsg` contains `argv` and `argc`
- The `mainchare` will typically creates some additional chares

Creating a Chare

- A chare declared as `chare MyChare {...};` can be instantiated by the following call:

```
CProxy_MyChare::ckNew(... constructor arguments ...);
```

- To communicate with this class in the future, a *proxy* to it must be retained

```
CProxy_MyChare proxy =
  CProxy_MyChare::ckNew(... constructor arguments ...);
```

Chare Proxies

- A chare's own proxy can be obtained through a special variable `thisProxy`
- Chare proxies can also be passed so chares can learn about others
- In this snippet, `MyChare` learns about a chare instance `main`, and then invokes a method on it:

`.ci` file

```
entry void foobar2(CProxy_Main main);
```

`.C` file

```
MyChare::foobar2(CProxy_Main main) {
  main.foo();
}
```

Charm Termination

- There is a special system call `CkExit()` that terminates the parallel execution on all processors (but it is called on one processor) and performs the requisite cleanup
- The traditional `exit()` is insufficient because it only terminates one process, not the entire parallel job (and will cause a hang)
- `CkExit()` should be called when you can safely terminate the application (you may want to synchronize before calling this)

Chare Creation Example: .ci file

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

  chare Simple {
    entry Simple(int x, double y);
  };
};
```

Chare Creation Example: .C file

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

class Main : public CBase_Main {
public: Main(CkArgMsg* m) {
     ckout << "Hello World!" << endl;
     if (m->argc > 1) ckout << " Hello " << m->argv[1] << "!!!" << endl;
     double pi = 3.1415;
     CProxy_Simple::ckNew(12, pi);
  };
};
class Simple : public CBase_Simple {
public: Simple(int x, double y) {
    ckout << "Hello from a simple chare running on " << CkMyPe() << endl;
    ckout << "Area of a circle of radius" << x << " is " << y*x*x << endl;
    CkExit();
 }
};

#include "MyModule.def.h"
```

Asynchronous Methods

- Entry methods are invoked by performing a C++ method call on a chare's proxy

```
CProxy_MyChare proxy =
  CProxy_MyChare::ckNew(... constructor arguments ...);

proxy.foo();
proxy.bar(5);
```

- The `foo` and `bar` methods will then be executed with the arguments, wherever the created chare, `MyChare`, happens to live
- The policy is one-at-a-time scheduling (that is, one entry method on one chare executes on a processor at a time)

Asynchronous Methods

- Method invocation is not ordered (between chares, entry methods on one chare, etc.)!
- For example, if a chare executes this code:

```
CProxy_MyChare proxy = CProxy_MyChare::ckNew();
proxy.foo();
proxy.bar(5);
```

- These prints may occur in **any** order

```
MyChare::foo() {
  ckout << "foo executes" << endl;
}

MyChare::bar(int param) {
  ckout << "bar executes with " << param << endl;
}
```

Asynchronous Methods

- For example, if a chare invokes the same entry method twice:

```
proxy.bar(7);
proxy.bar(5);
```

- These may be delivered in **any** order

```
MyChare::bar(int param) {
  ckout << "bar executes with " << param << endl;
}
```

- Output

```
bar executes with 5
bar executes with 7
```

OR

```
bar executes with 7
bar executes with 5
```

Asynchronous Example: .ci file

```
mainmodule MyModule {
  mainchare Main {
    entry Main(CkArgMsg *m);
  };
  chare Simple {
    entry Simple(double y);
    entry void findArea(int radius, bool done);
  };
};
```

Asynchronous Example: .C file

- Does this program execute correctly?

```
struct Main : public CBase_Main {
  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);
  };
};

struct Simple : public CBase_Simple {
 float y;
 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();
 }
```

Data types and entry methods

- You can pass basic C++ types to entry methods (`int`, `char`, `bool`, etc.)
- C++ STL data structures can be passed by including `pup_stl.h`
- Arrays of basic data types can also be passed like this:
- `.ci` file:

```
entry void foobar(int length, int data[length]);
```

- `.C` file:

```
MyChare::foobar(int length, int* data) {
  // ... foobar code ...
}
```

Collections of Objects: Concepts

- Objects can be grouped into indexed collections
- Basic examples
  - Matrix block
  - Chunk of unstructured mesh
  - Portion of distributed data structure
  - Volume of simulation space
- Advanced Examples
  - Abstract portions of computation
  - Interactions among basic objects or underlying entities

Collections of Objects

- Structured: 1D, 2D, . . . , 6D
- Unstructured: Anything hashable

Collections of Objects

- Structured: 1D, 2D, . . . , 6D
- Unstructured: Anything hashable
- Dense
- Sparse

Collections of Objects

- Structured: 1D, 2D, . . . , 6D
- Unstructured: Anything hashable
- Dense
- Sparse
- Static - all created at once
- Dynamic - elements come and go

Chare Array: Hello Example

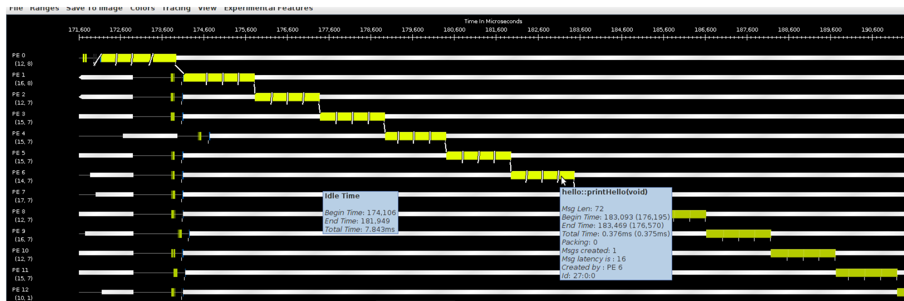
```
mainmodule arr {


  mainchare Main {
    entry Main(CkArgMsg*);
  }

  array [1D] hello {
    entry hello(int);
    entry void printHello();
  }
}
```

Chare Array: Hello Example

```
#include "arr.decl.h"

struct Main : CBase_Main {
  Main(CkArgMsg* msg) {
    int arraySize = atoi(msg->argv[1]);
    CProxy_hello p = CProxy_hello::ckNew(arraySize, arraySize);
    p[0].printHello();
  }
};

struct hello : CBase_hello {
  hello(int n) : arraySize(n) { }
  hello(CkMigrateMessage*) { }
  void printHello() {
    CkPrintf("PE[%d]: hello from p[%d]\n", CkMyPe(), thisIndex);
    if (thisIndex == arraySize - 1) CkExit();
    else thisProxy[thisIndex + 1].printHello();
  }
  private:
  int arraySize;
};

#include "arr.def.h"
```

Hello World Array Projections Timeline View

- Add `-tracemode projections` to link line to enable tracing
- Run Projections tool to load trace log files and visualize performance



- arrayHello on BG/Q 16 Nodes, mode c16, 1024 elements (4 per process)

Declaring a Chare Array

.ci file:

```
array [1d] foo {
  entry foo(); // constructor
  // ... entry methods ...
}
array [2d] bar {
  entry bar(); // constructor
  // ... entry methods ...
}
```

.C file:

```
struct foo : public CBase_foo {
  foo() { }
  foo(CkMigrateMessage*) { }
  // ... entry methods ...
};
struct bar : public CBase_bar {
  bar() { }
  bar(CkMigrateMessage*) { }
```

Constructing a Chare Array

- Constructed much like a regular chare
- The size of each dimension is passed to the constructor

```
void someMethod() {
    CProxy_foo::ckNew(10);
    CProxy_bar::ckNew(5, 5);
}
```

- The proxy may be retained:

```
CProxy_foo myFoo = CProxy_foo::ckNew(10);
```

- The proxy represents the entire array, and may be indexed to obtain a proxy to an individual element in the array

```
myFoo[4].invokeEntry();
```

thisIndex

- 1d: `thisIndex` returns the index of the current chare array element
- 2d: `thisIndex.x` and `thisIndex.y` returns the indices of the current chare array element

`.ci` file:

```
array [1d] foo {
  entry foo();
}
```

`.C` file:

```
struct foo : public CBase_foo {
  foo() {
    CkPrintf("array index = %d", thisIndex);
  }
};
```

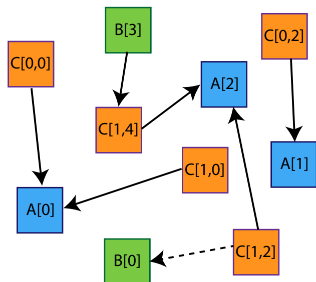
Collections of Objects: Runtime Service

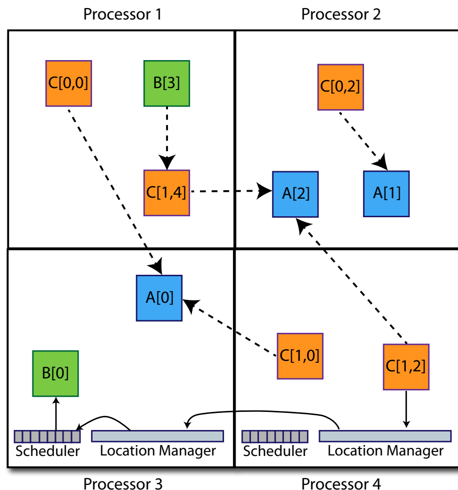
- System knows how to 'find' objects efficiently: $(collection, index) \rightarrow processor$
- Applications can specify a mapping, or use simple runtime-provided options (e.g. blocked, round-robin)
- Distribution can be static, or dynamic!
- Key abstraction: application logic doesn't change, even though performance might

Collections of Objects: Runtime Service

- Can develop and test logic in objects separately from their distribution
- Separation in time: make it work, then make it fast
- Division of labor: domain specialist writes object code, computationalist writes mapping
- Portability: different mappings for different systems, scales, or configurations
- Shared progress: improved mapping techniques can benefit existing code

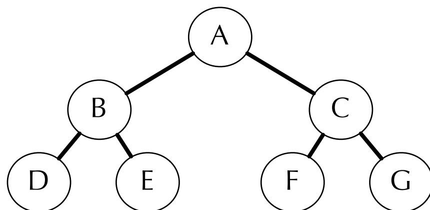
Collections of Objects





Collective Communication Operations

- Point-to-point operations involve only two objects
- Collective operations that involve a collection of objects
- Broadcast: calls a method in each object of the array
- Reduction: collects a contribution from each object of the array
- A spanning tree is used to send/receive data



Broadcast

- A message to each object in a collection
- The chare array proxy object is used to perform a broadcast
- It looks like a function call to the proxy object
- From the main chare:

```
CProxy_Hello helloArray = CProxy_Hello::ckNew(helloArraySize);
helloArray.foo();
```

- From a chare array element that is a member of the same array:

```
thisProxy.foo()
```

- From any chare that has a proxy p to the chare array

```
p.foo()
```

Reduction

- Combines a set of values: sum, max, aggregate
- Usually reduces the set of values to a single value
- Combination of values requires an operator
- The operator must be commutative and associative
- Each object calls `contribute` in a reduction

Reduction: Example

```
mainmodule reduction {
  mainchare Main {
    entry Main(CkArgMsg* msg);
    entry [reductiontarget] void done(int value);
  };
  array [1D] Elem {
    entry Elem(CProxy_Main mProxy);
  };
}
```

Reduction: Example

```
#include "reduction.decl.h"

const int numElements = 49;

class Main : public CBase_Main {
public:
  Main(CkArgMsg* msg) { CProxy_Elem::ckNew(thisProxy, numElements); }
  void done(int value) {
    CkAssert(value == numElements * (numElements - 1) / 2);
    CkPrintf("value: %d\n", value);
    CkExit();
  }
};

class Elem : public CBase_Elem {
public:
  Elem(CProxy_Main mProxy) {
    int val = thisIndex;
    CkCallback cb(CkReductionTarget(Main, done), mProxy);
    contribute(sizeof(int), &val, CkReduction::sum_int, cb);
  }
  Elem(CkMigrateMessage*) { }
};

#include "reduction.def.h"
```

Output:

```
value: 1176
Program finished.
```