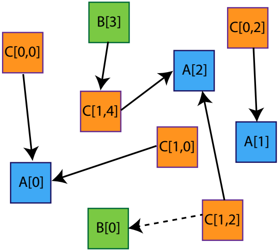
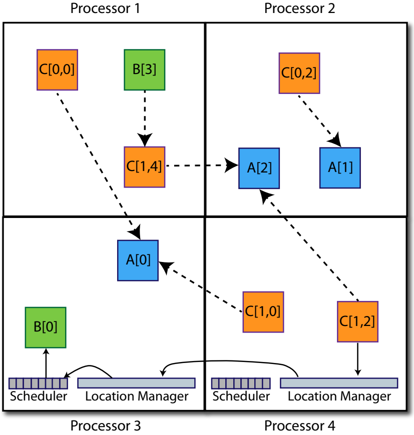
Chare Arrays

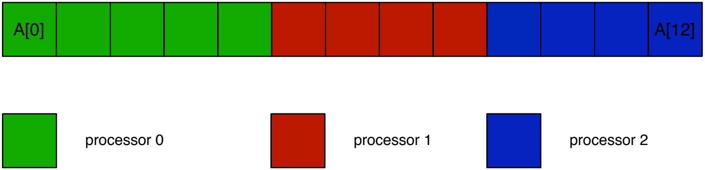
- Indexed collections of chares
  - Every item in the collection has a unique index and proxy
  - Can be indexed like an array or by an arbitrary object
  - Can be sparse or dense
  - Elements may be dynamically inserted and deleted
- For many scientific applications, collections of chares are a convenient abstraction
- Instead of creating networks of chares that learn about each other (by sending proxies to each other), each element in a chare array knows about all the others


B[3]
C[0,0]
C[0,2]
A[2]
C[1,4]
A[1]
C[1,0]
A[0]
C[1,2]
B[0]


Processor 1
Processor 2
C[0,0]
B[3]
C[0,2]
C[1,4]
A[2]
A[1]
A[0]
C[1,0]
C[1,2]
B[0]
Scheduler
Location Manager
Scheduler
Location Manager
Processor 3
Processor 4

Chare Array Location

- By default, chare arrays are distributed to the processors in a “blocked” distribution



- A initial mapping function can be specified (input is the index, output is the processor)
  - Called the *home PE* of the element
- Chare array elements can be migrated by the user or the runtime (load balancing)

Declaring a Chare Array

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

```
struct foo : public CBase_foo {
  foo() { }
  foo(CkMigrateMessage*) { }
};
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

```
CProxyElement_foo elm = myFoo[5];
elm.invokeEntry();
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

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

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

Charm Array: Hello Example

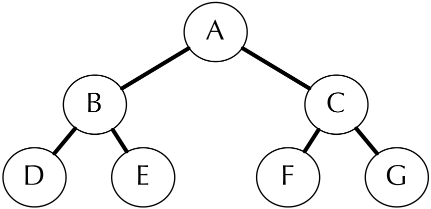
```
mainmodule arr {
  readonly int arraySize;

  mainchare Main {
    entry Main(CkArgMsg∗);
  }

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

Charm Array: Hello Example

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

/*readonly*/ int arraySize;

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

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

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

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:

```
thisProxy.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.
```