Charm++

CS315B
Lecture 11
History

• Charm++ designed in the early 1990’s
  - Based on Charm from the late 1980’s

• Parallel machines of the time were
  - Custom architectures, fading in importance
  - Networks of commodity workstations
    • Much cheaper
    • Eventually became the dominant compute platform
History (Cont.)

• This is the environment that led to the rise of **MPI**
  - Two-level programming model
  - On-node managed with standard programming
  - Off-node managed by message passing

• **Charm++** has a similar top-level design
  - With a focus on integrating object-oriented features
Chares

• The basic unit of computation and parallelism in Charm++ is a *chare*

• An object
  - A set of *entry* methods
    • Take a single *message* argument
  - Entry methods can be invoked by other chares
Message Passing Model

• A chare responds to one message at a time
  - Chares are single-threaded
  - Entry point methods always run to completion
    • No interrupts

• Flexibility in which message is handled next
  - When multiple entry point methods could be invoked, configurable policies determine choice
  - E.g., messages can have priorities
Chare Classes

- Chares are special in Charm++

```cpp
chare MyChareType {
    entry MyChareType(args);
    entry void MyMethod(args);
}
```

- The Charm++ preprocessor/compiler generates C++ classes and methods from this spec
Creating Chares

• Chares can be created individually
  \[ C_{proxy\_X} \times = X::ckNew(\text{args}); \]

• To create a chare on a specific processor:
  \[ C_{proxy\_X} \times = X::ckNew(\text{args},\text{proc}); \]
What Are Proxies?

• Handles on remote objects
  - The chara itself is in some unknown location, usually not local
  - The programmer interacts with proxy objects

• To invoke a method on a chara, invoke the method on its local proxy

• Proxies are an artifact of being embedded in C++
  - Could be avoided in a language with its own syntax/semantics
Method Invocation on Chares

\[ \text{chareProxy.EntryMethod(args)} \]

- Asynchronous, does not block
  - Calling thread continues

- And one-sided, no explicit acknowledgment
Creating Chares

• Chares can be created individually

• More commonly, *chare arrays* are used

\[
carray = \text{ClassName}::\text{cknew}(\text{numElements}) \\
carray[0].\text{entry}(\text{msg})
\]
Advantages of Chare Arrays

• Easy to create lots of chares
  - Which are automatically distributed around the machine

• Easy to name chares
  - A chare can easily refer to its neighbors, a distinguished chare, etc. using array indices
Hello World, Version 1

```cpp
helloArray = Hello::cknew(numElements);
helloArray[0].sayHi(-1);

...

Hello {
    void Hello::sayHi(int from) {
        printf(“Hello from %d\n”, thisIndex);
        if thisIndex < (numElements - 1)
            thisProxy[thisIndex + 1].sayHi(thisIndex);
    ...
```
Hello World, Version 2

Main {
    helloArray = Hello::cknew(numElements);
    helloArray.sayHi(-1);

    void done() {
        if (++doneCount >= numElements) CkExit();
    }

    ...

Hello {
    void Hello::sayHi(int from) {
        printf("Hello from %d\n", thisIndex);
        mainProxy.done();
    }

    ...
}
Chare Arrays vs. MPI

- Chare arrays provide an MPI-like model
- Message passing
- Collective operations
  - E.g., reductions
  - Global names for elements of the collection
Reductions

```c
int myInt = 1;
contribute(sizeof(int), &myInt, CkReduction::sum_int);
```

- `contribute` is a built-in method on chare arrays
- All members of a chare array must call `contribute`
- `contribute` can also be used as a barrier:

  ```c
  contribute()
  ```
Comments on Control in Charm++

• Because message sends and receives are asynchronous, programs tend to be written in an event-driven style
  - Many entry point methods, each doing a small part of a larger task

• This leads to difficult-to-understand control flow
  - Hard to reason about order of entry point execution
Structured Dagger

- A mechanism for showing/enforcing intended order of entry point calls

```java
chare ComputeObject {
    entry void start() {
        when first(T x)
            when second(T y)
                doPair(x,y)
    }
    entry void first(T x);
    entry void second(T j);
}
```
Another Problem ...

• Charm++ is based on message passing in C++

• Most C++ things are objects

• So we’ll want to send objects in messages ...
PUP

• PUP = pack/unpack

• A serialization/deserialization framework
  - One declaration of both

```cpp
void T::pup(PUP::er &p) {
  p|field1;
  p|field2;
  ...
}
```
But …

• No in order message delivery

• All messages are one-sided
  – Chare does not block on a message send

• Not limited to one array of chares

• Location of chares is transparent
  – And can change (e.g., for load balancing)
Read Only Data

- Can declare read only data
  - With global name, globally accessible

```typescript
readonly Type ReadonlyVariable;
```

- `readonly` is really “write once”
  - In main chare

- An important facility
  - Underlying system makes sure read-only data is available everywhere
Load Balancing

• Because location of a chare is kept abstract, it is possible to migrate chares

• Charm++ has built-in load balancing
  – Runtime moves chares
  – Uses the chares’ PUP methods
  – Many load balancing policies
  – And users can write their own
Load Balancing (Cont.)

• To balance load, need chores > processors

• Called over partitioning
  - Create more units of work than processors
  - If one processor is too heavily loaded, move some of its units of work to a lightly loaded processor

• Good if compute cost is linear in data size
  - Not so good if compute cost is superlinear
Other Mapping Policies

• User can set policies for

• Initial assignment of chares to processors
• Migration of chares
  – i.e., load balancing
• Locality
  – Affinity of chares to each other

• Reminiscent of dynamic mapping decisions in Regent
Critique of Charm++

- Consider:
- Programmability
- Control model
- Data model
Programming

• Race conditions
  - No shared memory, so no traditional races
  - But easy to miss needed synchronization
  - E.g., have all shares in a local stencil calculation contributed?

• Deadlocks
  - Easy to get with out-of-order message handling

• Tradeoff
  - Can improve performance by being more asynchronous
  - But take the risk of introducing concurrency issues
Memory Management

• Programmer is responsible for managing message allocation & deallocation

• No way for the runtime to know when a program is finished with a message

• Programmer must manage all other memory explicitly as well
  - Like MPI
Control

• Parallelism expressed at the level of chares
  - One level of parallelism
  - Well suited to clusters of sequential processors

• Ability to express hierarchy unclear
  - Early versions of Charm++ had hierarchy
  - Now in the “experts only” feature list
Data

• **Minimal facilities for describing structure of data**
  - Chare arrays are the main mechanism
  - Note they unify control & data decomposition

• **No support for defining multiple views of data**
  - Can be done, but programmer has to do it “by hand”, and system will not take advantage of it

• **Support for locality in load balancing/scheduling policies**
  - But nothing higher level
Summary

- **Charm++**
  - Minimalist view: Object-Oriented MPI
  - But can do more

- **Mature**
  - Many ports
  - Good documentation
  - Significant applications and libraries
  - Some applications run on very large machines