Task-based Prog. Models and Charm++

Alan Sussman, Department of Computer Science
Announcements

• Assignment 4 posted today, due May 2 at 11:59 pm

• Quiz 2 available tomorrow at 11AM, and you have 24 hours to take it
Task-based programming models

- Describe program / computation in terms of tasks
- Notable examples: Charm++, StarPU, HPX, Legion, Cilk, OpenMP(!)
- Attempt at classification of programming models: https://link.springer.com/article/10.1007/s11227-018-2238-4
- From that paper a task is defined as “a sequence of instructions within a program that can be processed concurrently with other tasks in the same program. The interleaved execution of tasks may be constrained by control- and data-flow dependencies.”
Task-based programming models

- Enable exposing high degree of parallelism
- Number of tasks is independent of the number of processors
- Tasks might be short-lived or persistent throughout program execution
- Runtime system handles distribution and scheduling of tasks
Charm++: Key principles

• Programmer decomposes data and work into objects (called *chares*)
  • Decoupled from number of processes or cores

• Runtime assigns objects to physical resources (cores and nodes)

• Each object can only access its own data
  • Request data from other objects via remote method invocation: `foo.get_data()` – similar to RMI in Java

• Asynchronous message-driven execution
Hello World in Charm++

mainmodule hello {
    array [1D] Hello {
        entry Hello();
        entry void sayHi();
    };
};

void Hello ::sayHi() {
    CkPrintf("Hello from chare %d on processor %d.\n", thisIndex, CkMyPe());
}

Main::Main(CkArgMsg* msg) {
    numObjects = 5; // number of objects
    CProxy_Hello helloArray = CProxy_Hello::ckNew(numObjects);
    helloArray.sayHi();
}

Charm++ Tutorial: http://charmplusplus.org/tutorial/ArrayHelloWorld.html
Compiling a charm program

- Charm translator for .ci file (a Charm++ interface file)
  - Generates charm_hello.decl.h and charm_hello.def.h

```
charmc hello.ci
```

- C++ code:

```
charmc -c hello.C
carmlc -o hello hello.o
```
Chare arrays

• User can create indexed collection of data-driven objects

\[ \text{CProxy}_\text{Hello} \text{ helloArray} = \text{CProxy}_\text{Hello}::\text{ckNew}(\text{numElements}); \]

• Different kinds: 1D, 2D, 3D, ...

• Mapping of array elements (objects) to hardware resources handled by the runtime system (RTS)
  • By default in round-robin fashion
Object-based virtualization

- User programs in terms of shares/objects
Over-decomposition

- Create lots of “small” objects per physical core
  - Objects grouped into arrays: 1D, 2D, ...
- System assigns objects to processors and can migrate objects between physical resources
- Facilitates automatic load balancing
Message-driven execution

• An object is scheduled by the runtime scheduler only when a message for it is received

• Facilitates adaptive overlap of computation and communication
Cost of creating more objects?

- Context switch overhead
- Cache performance
- Memory overhead
- Fine-grained messages
Hello world: .ci file

mainmodule hello {

    readonly CProxy_MyMain myMainProxy;
    readonly int numChares;

    mainchare MyMain {
        entry MyMain(CkArgMsg *msg);
        entry void done(void);
    };

    array [1D] Hello {
        entry Hello(void);
        entry void sayHi(int);
    };

};
Hello world: MyMain class

/*readonly*/ CProxy_MyMain myMainProxy;
/*readonly*/ int numChares;

class MyMain: public CBase_MyMain {
    public:
        MyMain(CkArgMsg* msg) {
            numChares = atoi(msg->argv[1]); // number of elements
            myMainProxy = thisProxy;
            CProxy_Hello helArrProxy = CProxy_Hello::ckNew(numChares);

            helArrProxy[0].sayHi(20);
        }

    void done(void) {
        cout << "All done" << endl;
        CkExit();
    }
};
Hello world: Hello class

#include "hello.decl.h"
extern /*readonly*/ CProxy_MyMain myMainProxy;

class Hello: public CBase_Hello {
    public:
        Hello(void) { }

        void sayHi(int num) {
            cout << “Char “ << thisIndex << “ says Hi!” << num << endl;

            if(thisIndex < numChars-1)
                thisProxy[thisIndex+1].sayHi(num+1);
            else
                myMainProxy.done();
        }
};

#include "hello.def.h"
Proxy class

• Runtime needs to pack/unpack data and also figure out where the chare is

• Proxy class generated for each chare class
  • Proxy objects know where the real object is
  • Methods invoked on these proxy objects lead to messages being sent to the destination process/thread where the real object lives
Broadcast, barrier, and reduction

• Entry method called on a chare proxy without subscript is essentially a broadcast:

```cpp
chareProxy.entryMethod();
```

• Barrier: reduction without arguments:

```cpp
contribute();
```

• Reduction with arguments:

```cpp
void contribute(int bytes, const void *data, CkReduction::reducerType type);
```
Callback for reduction

• Where does the output of the reduction go?

• Use a callback object known as a reduction client

CkCallback* cb = new CkCallback(CkIndex_myType::myReductionFunction(NULL), thisProxy);
contribute(bytes, data, reducerType, cb);

• Use the reduction data in the callback:

    void myType::myReductionFunction(CkReductionMsg *msg) {
        int size = msg->getSize() / sizeof(type);
        type *output = (type *) msg->getData();
        ...
    }

2D Stencil in Charm++