CMSC416: Introduction to Parallel Computing

Topic: Charm ++
Date: April 11, 2024

.CI File:

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);
  };

};
```

The left image shows an example of the .CI file. This is the first file you write when you create a chart program and is how the user tells the runtime that he is going to make some c++ objects or charm ++ arrays so the runtime should help in migrating them and load balancing them. Normally, the .CI file is the first file you write. The utility of it is to logically organize your objects in charm or your array classes in charm into modules. DO NOT forget to provide the semicolons in the CI file.

The last chare is a 1d array called Hello that hast 2 functions, the constructor and the other one is sayHi. Both can be created remotely. The constructor must be remote so you can call the object from any process. In charm++, even the main function happens in a chare, this is why we have the mainchare class. Inside we have the constructor of the mainchare and other functions like “done” (can be used to communicate to mainchare to do a clean exit). Can also create read only global variables used to define some information at the beginning of the program and then use it in the program without changing it again. These will be available among all cores.

Compiling a charm program

Translates the ci file into 2 files: .decl.h and .def.h. You should include the first one at the top of the header file and the second one at the bottom of the cpp file. Use of charmc as wrapper to compile the code (like mpicc).

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) {
      ckout << "All done" << endl;
      CkExit();
    }
};
```

Standard C++ class with some differences. When you create the CI file and the compiler translates it, it creates some classes automatically for you. For example, when you create a mainchare called MyMain, the Cbase_MyMain is automatically available to you and MyMain class is going to inherit from Cbase_MyMain, which has some helper functions. The CkExit() function is like MPI_Finalize(), it is used to make a clean exit of the program. Once you put arguments in the command line, all these get packed in a message so that mainchare inspects the message and retrieves the command line arguments, in this case, the number of elements in the 1d array we want to create. This value will be available to all chares in the program. To create the chare array object, we use ckNew using the proxy of the array object (Cproxy_Hello) that returns a proxy that we can store in "helArrProxy". Now, we can call functions on the chare array elements like where we call the function sayHi for the first element in the chare array.

Hello Class

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

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

    void sayHi(int num) {
      ckout << "Chare " << thisIndex << "says Hi!" << num << endl;

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

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

Remember to be able to access classes that the runtime has created like CBase_Hello. This class has some helper functions and the Hello constructor and sayHi function. When a chare enters the sayHi function, the chare is going to do a print of a message. The important thing is "thisIndex" is a keyword that returns the index of that chare in the chare array. It is important to understand that in the function, each chare will call sayHi on the next chare. If we get the to the last chare because it is the last, then it tells the proxy of the Main chare that we are done so the last chare is going to call the done function on the main chare.

Proxy class

The proxy class is useful to know where a chare is located (physical core). Users can just index chares by proxy, and it is the runtime that abstracts the user of knowing where each chare is. Also, the runtime

creates the messages to communicate to other chares by remote entry methods and packs and unpacks the messages appropriately.