CSE 746 - Parallel and High Performance Computing Lecture 13 - Parallel Languages: Charm++

Pawel Pomorski, *HPC Software Analyst*

SHARCNET, University of Waterloo

ppomorsk@sharcnet.ca

http://ppomorsk.sharcnet.ca/

Parallel programming languages

- Definition of the concept is not precise, all programming approaches we have been using so far (MPI,OpenMP,CUDA) obviously enable parallel programming
- Language/approach is "more parallel" if programmer needs to do less work in arranging the details of the parallel computation, in particular how the parallel work is distributed among hardware

CUDA - Parallel programming language?

- For example, CUDA is very parallel in that the programmer does not have to control which threads go to which multiprocessor inside the GPU
- On the other hand, it's not parallel in that the programmer has to:
  - decide which tasks run on GPU and which on CPU
  - decide which kernels go to which GPU (if there are more than 1)

MPI - Parallel programming language?

- MPI computation involves launching a set of serial programs that communicate with each other
- The number of these programs is (typically) fixed for the duration of the computation, and each program is running on a particular machine.
- The division of work between physical processors has to be done by the programmer, who distributes work between MPI processes
- Key issue: load balancing. If processors do an unequal amount of work, it is then necessary for the programmer to add code which moves work/data around in a more balanced way

OpenMP - Parallel programming language?

- the details of thread launching are handled by runtime environment
- thread creation relatively easy on the programmer
- work is divided between a small number of threads then assigned to individual processors, in a shared memory paradigm the operating system has some opportunities for load balancing
- nevertheless, programmer has to ensure that each thread is doing a similar amount of work to have an efficient program

Parallel programming languages

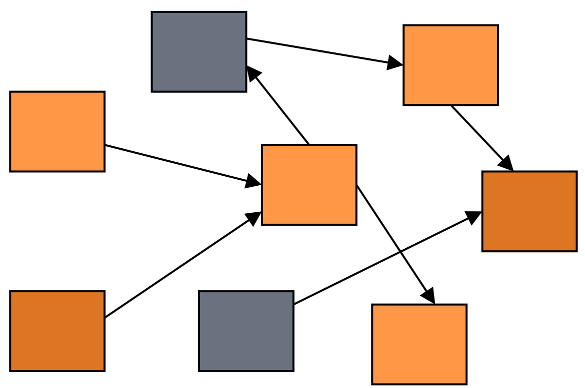
- A high-level programming language would only have the programmer define a set of work tasks that can be done in parallel, with some dependencies between them
- The compiler and runtime environment would then handle all the details of starting the tasks in parallel and distributing them efficiently on the available hardware

Charm++

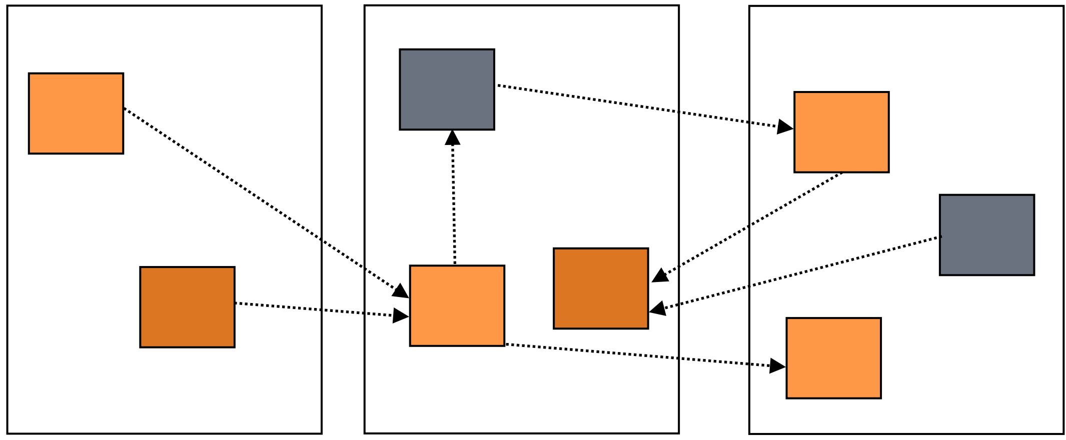
- Parallel object-oriented programming language
- http://charm.cs.uiuc.edu/tutorial/
  great starting page with tutorials and links to documentation
- based on C++ but with additional syntax
- comes with its own compiler, charmc

Charm++ paradigm

- user view: programmer expresses communication between objects, with no reference to processors



system view:
objects distributed between processors



Charm++ paradigm

- Charm++ program is built around a set of parallel objects, called *chares*
- One of these is the special chare, declared as mainchare, which serves as the main function of the program, and invokes all other chares
- There is no main function in Charm++ program (actually, Charm++ will create its own main function which the user does not see directly)

Charm++ paradigm

- Chares are parallel objects which can communicate with each other by invoking each other's methods, with messages as arguments
- The methods of communication are so important that they are declared in their own separate file, with extension .cli
- This file is processed to produce C++ .h files, which are then included in source code

Charm++ paradigm

- Chares are distributed on the computed resources of the system, the programmer does not care where they are physically located. They do not share memory (hence like MPI)
- Chares can be identified by proxies, which serve as “pointers” to chares
- Chare execution is asynchronous by default. If one chare invokes method A followed by method B of another chare, there is no guarantee in what order they will execute
- Hence synchronization has to be taken care of by the programmer

Charm++ - tutorial exercises

- /home/ppomorsk/CSE746_lec11
- from: https://charm.cs.illinois.edu/tutorial/
- use **make** to compile. Charm++ has already been compiled and Makefile is modified to find it
- use **charmrun** to run (generated during compiling)
- Basic "Hello, world!" program
- Array "Hello, world!" program
- Jacobi 2D relaxation
- charm++ generates additional files during the build. If you want to see just the essential files, unpack the .tar.gz files provided (remember to modify the Makefile so it can find charm++ before you run **make**)

Basic "Hello, world!" program in Charm++

- /home/ppomorsk/CSE746_lec11/BasicHelloWorld
- https://charm.cs.illinois.edu/tutorial/BasicHelloWorld.htm

Header File (*main.h*)

```
#ifndef __MAIN_H__
#define __MAIN_H__

class Main : public CBase_Main { (1)

 public:
  Main(CkArgMsg* msg); (2)
  Main(CkMigrateMessage* msg); (3)

};

#endif //__MAIN_H__
```

Source File (*main.C*)

```
#include "main.decl.h" (6)
#include "main.h"

// Entry point of Charm++ application
Main::Main(CkArgMsg* msg) {

  // Print a message for the user
  CkPrintf("Hello World!\n"); (4)

  // Exit the application (5)
  CkExit();
}

// Constructor needed for chare object migration (ignore
// for now) NOTE: This constructor does not need to
// appear in the ".ci" file
Main::Main(CkMigrateMessage* msg) { }

#include "main.def.h" (6)
```

Interface File (*main.ci*)

```
mainmodule main { (7)

  mainchare Main { (8)
    entry Main(CkArgMsg* msg); (9)
  };

};
```

Makefile

```
CHARMDIR = [put Charm++ install directory here] (10)
CHARMC = $(CHARMDIR)/bin/charmc $(OPTS) (11)

default: all
all: hello

hello : main.o
   $(CHARMC) -language charm++ -o hello main.o (12)

main.o : main.C main.h main.decl.h main.def.h
   $(CHARMC) -o main.o main.C (13)

main.decl.h main.def.h : main.ci
   $(CHARMC) main.ci (14)

clean:
   rm -f main.decl.h main.def.h main.o hello charmrun
```

C

Array “Hello, world!” program in Charm++

- /home/ppomorsk/CSE746_lec11/ArrayHelloWorld
- observe the communication pattern, shown in: https://charm.cs.illinois.edu/tutorial/ArrayHelloWorld.htm
- try varying the number of processors

Creating chare arrays

- define chare Hello
- create arrays
  ```
  CProxy_Hello helloArray = CProxy_Hello::ckNew(numElements);
  ```
- call chare method
  ```
  helloArray[0].some_method();
  ```

Special variables

- thisProxy - variable points to the array that this chare object is in
- thisIndex - variable holds the index in the the chare array that this chare object occupies
- think of analogous variables in MPI, OpenMP and CUDA
- Eg. If **this** chare (the one executing the code) which has index **thisIndex** wants to call a method of a "neighbour" chare in its chare array (index **thisIndex + 1)**, it would do:

```
thisProxy[thisIndex + 1].some_method();
```

This will execute method some_method from another chare

CkMyPe()

- returns the PE (processing element) that the chare is running on

Jacobi 2D relaxation program in Charm++

- /home/ppomorsk/CSE746_lec11/ArrayHelloWorld
- note the communication pattern shown on:
- https://charm.cs.illinois.edu/tutorial/Basic2DJacobi.htm
- try varying the number of processors