CMSC416: Introduction to Parallel Computing

Topic: Charm++
Date: April 11th, 2024

Charm++ Refresher

Charm++ is a task-based programming model where computational work is split across objects (also known as *chares*) that can communicate asynchronously with each other. From the programmer's viewpoint, this model allows you to focus more on the logical structure of your program as opposed to the lower level details of parallel programming. Things like load balancing, for example, are handled automatically and dynamically, the runtime will handle redistributing computation across processors for better performance.

Tasks are code regions that can be executed in parallel. Different tasks can be defined by defining objects known as chares which encapsulate their own data and behavior. The behavior of different chares can be defined by their methods (similar to methods of a traditional object in object oriented programming). Execution of tasks is scheduled by the runtime system only when a message is received by a chare (message-driven execution).

The Charm++ interface (.ci) File

The Charm++ interface file is where we specify chares and their associated data and method headers. It serves as a blueprint for our Charm++ program. Groups of chares can be organized into **modules**, but for the purposes of this course we will only ever work with a single module with a Charm++ interface file. See slide 13 of the Charm++ lecture slides for an example .ci file.

All execution within a Charm++ program happens within a chare. The "main" chare can be defined with the **mainchare** keyword and is the entry point for your Charm++ program. It defines the initial chare that'll be created and generally handles initializing other chares and the top-level control flow. Typically, the main chare will contain an entry method to stop program execution, a done() method.

Other chare types include **array** which can be used to define an array of chares, where each element has its own data and can do work asynchronously. There are different types of chare arrays: 1D, 2D, and 3D.

Chares can contain both data members and methods like an object. There are different types for both data and methods. A **readonly** variable is immutable after initialization, for example. An **entry** method is used for passing messages between chares, entry methods are invoked to process messages and execute the corresponding behavior according to the receive message.

Compiling the Charm++ interface file provides us with two header files: ***.decl.h** and ***.def.h,** where * is the name of the module. These header files can be used as imports in a C++ program to access classes that allow you to interface with your defined chares and define behavior for the chares' methods.

Writing C++ file using Charm++ Header Files

Now that you've defined the structure of your various chares, you can define their behavior and actual computational work within a C++ file. The header files define a base class for each chare type named as **CBase_*,** where * is the name of the chare. We define classes that extend off of these base classes, for each chare, to define the behavior of that chare's methods. See slides 14 and 15 of the Charm++ lecture slides.

The Charm++ header files define **proxy** classes which provide a handle to access particular chares. These proxies keep track of where the chare is actually stored across a distributed memory system, so that the programmer can just rely on the provided handle. Invoking methods on these proxy classes sends a message to the actual chare on a node/core. These proxy classes are names as **Cproxy_*,** where * is the name of the chare.

The base classes for each chare type define a **thisProxy** keyword which contains the handle for the associated chare. **Array** type chares also have a **thisIndex** keyword that contains that chare element's index within the array. It's common to have a global main proxy variable defined within a Charm++ program so that every chare can communicate with the main chare.

Passing data between chares (analogous to passing data between processes or threads) is done by passing data through the arguments of entry methods. This is known as the **parameter marshalling** method.