CMSC416: Introduction to Parallel Computing

Topic: Performance Issues & Charm++
Date: April 9, 2024

Performance Issues

Performance variability

Performance variability in high-performance computing is often caused by operating system jitter. This occurs when background system processes, known as daemons, intermittently activate, disrupting the main program's execution. To measure this variability, a programmer can execute a set workload repeatedly to observe variations in completion time. Each daemon operates on its unique schedule and can be active for varying durations that can delay communication. For instance, if one process experiences more significant delays, it can cause a bottleneck, as other processes dependent on it are consequently held up, affecting the efficiency of message passing.

One way to reduce variability includes using a lightweight operating system kernel, which involves removing non-essential daemons or using a basic version of the Linux kernel that retains only the necessary services for running an HPC environment. Alternatively, you can also reduce daemon frequency or confine OS services to a subset of cores and dedicate the remaining cores to the application. Network congestion also causes performance variability. While each job may have dedicated access to a node, they all share the network infrastructure. The network's topology determines how resources are shared among jobs, which can lead to congestion.

Charm++

Introduction

Charm++ is a programming model designed for distributed memory systems. It can also be considered a hybrid programming model because it supports both distributed and shared memory paradigms. This means that Charm++ programs can be executed in a purely distributed memory environment, similar to how MPI operates. Additionally, Charm++ allows execution in a combined shared and distributed memory mode. This hybrid mode uses shared memory within a computing node to optimize communication and computation efficiency, while still enabling communication across nodes in a distributed system. This way, Charm++ can utilize the computational resources available within each node and across the cluster.

Task-Based Programming Models

Previously in this class, when designing parallel programs, the approach has involved distributing data among various threads or processes, as seen in models like MPI and OpenMP. These models typically use a one-to-one mapping, where each process or thread is mapped directly to a physical core. Specifically, MPI assigns one process per physical core, and OpenMP follows a similar pattern with threads, though it allows for multiple threads per core in hyperthreading modes.

On the other hand, task-based programming introduces a different model type. It shifts the focus from how data is divided among processing units to how the programmer designs the program's structure and the tasks that compose it. These tasks, which can vary in size from fine-grained to coarse-grained, represent the units of programming. The responsibility of mapping these tasks to the physical cores lies with the program's runtime system, not the programmer. This means that the runtime system can allocate more tasks than there are available cores, optimizing the utilization of hardware resources.

Task-based programming models significantly increase the potential for parallelism in a program. Unlike models we have previously explored in this class, where the number of processes or threads is often tied to the number of available cores, task-based models break this limitation. They allow for a more flexible allocation of computational tasks, enabling the programmer to assign a large number of tasks independent of the core count. This flexibility means that problems can be divided into numerous small tasks, optimizing parallel efficiency.

Furthermore, in task-based programming, tasks can be designed to execute specific code regions and then terminated upon completion, or they can be made persistent, depending on the program's implementation. Similar to OpenMP and MPI, the runtime environment in task-based models takes on the responsibility of task mapping and scheduling. Charm++ is a task-based parallel programming system founded by UIUC.

Charm++

In the Charm++ programming model, the programmer breaks down the data and computational tasks into units known as chares. Note that both data *and* tasks can be divided into chares. This allows for the creation of an arbitrary number of tasks, which can even be specified as command-line arguments when executing a program. The runtime system then dynamically assigns these tasks to physical cores.

Charm++ adopts a distributed memory model where each object can only access its own data. To access data from another chare, explicit requests must be made. In

Charm++, data in another object is accessed through remote method invocation, due to Charm++ objects being C++ based. For example, if there is an object named foo with a get_data() method, the programmer can directly call foo.get_data() to retrieve its data, unlike the traditional message-sending and receiving protocols found in MPI. Furthermore, objects calling methods on each other do not need to be on the same process or node.

Additionally, Charm++'s design has been asynchronous, while MPI originally had a synchronous design. This asynchronous nature allows Charm++ to efficiently handle non-blocking operations and task scheduling.

Hello World in Charm++ (Slides 6-9)

In Charm++, the programming structure often includes arrays of objects, with 'hello' being the array in this example. This array defines specific functions, with their implementations provided separately. A distinct interface file, typically known as a ".ci" file, is required for the definition of Charm++ objects, also referred to as chares. These chares are special because they have the ability to communicate with other remote chares by sending messages. In contrast, standard C++ objects and functions are defined in traditional C++ source files.

The 'hello' array in the program is one-dimensional and utilizes "entry" methods to designate which functions of the Hello class can be invoked remotely. Functions not meant for remote invocation are kept in standard C++ files. In the accompanying standard C++ file, function definitions follow the usual syntax. Moreover, the program includes a main function encapsulated within a Main class, which has a constructor. During execution, the runtime determines the number of Hello object instances to create. The ckNew constructor is called to instantiate these objects. To invoke the sayHi function across all Hello instances, one would use helloArray.sayHi(). Alternatively, it's possible to call sayHi on a specific instance by using its index.

In Charm++, chares are persistent, meaning they continue to exist throughout the lifetime of the program. In this example, the model employs an SPMD style of programming, where the sayHi function is a common instruction executed by all objects but operates on different data, highlighting the parallelism inherent in the Charm++ model.

Compilation

To compile a program in Charm++, you use the charmc compiler wrapper to compile the Charm++ interface files (these have a .ci extension). The charmc wrapper acts as a translator, converting .ci files into valid C++ code. It produces two types of files:

declaration files (.decl.h) and definition files (.def.h), which contain the necessary declarations and definitions for the Charm++ objects.

Once the interface file has been processed, you can proceed as if working with standard C++ files. The generated declaration files should be included at the beginning of your header files (*.h), and the definition files should be included at the bottom of your C++ source files (*.C or *.cpp). This ensures that the compiler is aware of the Charm++ constructs and can appropriately handle them during the compilation process, ultimately leading to the generation of the executable (object) file.

Chare Arrays

Chare arrays in Charm++ enable the creation of an indexed collection of chares, but you can also create independent chares. The process of defining and instantiating a chare array involves two key steps: Firstly, within the Charm++ interface file (.ci), you declare the intention to create an array with a certain dimensionality, such as a 1D array named hello. Secondly, using the translated proxy class, you instantiate the array in your code with the ckNew function, specifying the desired number of elements. This size can be dynamically determined, often passed as a command-line argument upon program execution.
Chare arrays can be 1D, 2D, 3D, and beyond—usually, this just depends on the physical space available. The runtime system (RTS) is responsible for the assignment of these array elements to physical cores. By default, the RTS employs a load-balancing scheme aimed at distributing an approximately equal number of objects to each core.

Object-based virtualization

Users write programs with chares or objects within a global object space ("User View). This space allows programmers to focus on the problem rather than the underlying hardware architecture. On the other side, the "System View" shows how the runtime system maps these abstract objects onto physical processors (CPU A, CPU B, CPU C). The objects are distributed across the CPUs, with the arrows symbolizing communication between them. This separation between the user's perspective and the system's view demonstrates how in Charm++, the user is freed from direct concerns about object location and communication, which are managed by RTS.

Message-Driven Execution

In task-based programming, the quantity of objects can exceed the number of available processors. Instantiating significantly more objects than there are physical cores is called over-decomposition. For example, having eight objects per core means we have an over-decomposition factor of eight. Because there are so many objects, task-based

models perform object migration between cores, allowing automatic load balancing. Programmers can also access provided load-balancing algorithms or create their own custom migration algorithms.

In scenarios where multiple objects are assigned to a single processor, the Charm scheduler orchestrates their execution, using a first-come-first-served policy. This means that the order of object execution is determined by the sequence of incoming messages, with scheduling based on a first-in-first-out queue. Once an object completes its task, it is then replaced by the next in line.

Cost of Creating More Objects

This task-based model, while being fine-grained compared to more coarse-grained approaches like MPI, does introduce the potential for context switching overheads. However, there can also be advantages, such as improved cache performance, because smaller tasks are more likely to fit into and utilize the cache effectively.