CS696 Concurrency

Kyle Clapper

November 29, 2023

Major ideas I want to discuss:

1. The various levels of concurrency abstraction absolve the programmer from having to worry about the hardware implementation and temporal dependencies (is there a better way to phrase temporal dependencies?).
2. The various levels of concurrency abstraction are best suited to different types of tasks, here are some examples.
3. Concurrent abstractions help define (1) how independent pieces of the program communicate and (2) how to think about WHEN these pieces are running.
4. The difference between concurrency and parallelism.

Other topics it would be nice to cover:

1. Theory of concurrent programs (CSP, Pi calculus).
2. Parallel algorithms.

Questions I want to keep thinking about:

1. The basis of many concurrent abstractions on the CPU side is the OS (which itself is an abstraction over hardware). How do these abstractions change or how are they mirrored with GPU programming?

1 Introduction

Concurrency is a notoriously tricky subject. Mentions of concurrent programs are often met with trepidation and fear because of their reputation for being difficult to program. These fears are not unwarrented because concurrent programs upset the mental model many programmers have for their programs. That being of sequential operations that happen (more or less) in the order their program specifies. These are typical sequential programs. Adding concurrency decouples the lexical order of the program from the order in which parts of the program are executed. This requires a very different mental model and a host of new systems and abstractions to manage it.

For many years programs were written almost exclusively in a sequential fasion. Programmers at the time could rely on Moores law to increase the speed of their programs without doing any additional work. Then, sometime around 2005, programmers began to realize the limits of Moores law. At some point, the "free lunch" will be over and they won't be able to rely on their programs getting faster as faster processors come out. As we've seen in recent years, the increase in CPU clock speed has started to slow and the focus is more and more on CPUs with multiple cores. With this shift in computer hardware, it's becoming increasingly necessary for programmers to write concurrent programs to take advantage of all system resources.

1.1 Concurrency vs Parallelism

A common point of confusion is the difference between concurrency and parallelism. These terms are very related, but they cannot necessarily be used interchangeably. In a similar way that a square is a rectangle but a rectangle is not necessarily a square, paralellism is a form of concurrency but concurrency does not necessarily imply parallelism.

Concurrency is when you have multiple independent threads of computation happening in a program. Parallelism is when two or more of those threads are being processed at the same time. Concurrency without parallelism often takes the form of time sharing. For example, two concurrent processes executing on a single CPU machine will appear to run at the same time, but they are not parallel because they aren't actually running at the same time but instead sharing time slices allocated by the operating system.

For example, if two operating system processes are running at the same time on separate CPU cores, then they are running in parallel. If instead, you have a program with two functions, one that requires waiting for an I/O device and another that does not, and the non-I/O function runs while the other function is waiting, then this is concurrency but not parallelism. In this example, both functions are running on the same thread of computation, but the time is shared between them.

It's worth mentioning that distributed systems are concurrent as well. If you have a web application that's built as a collection of independent services communicating over HTTP, these services act in a concurrent fashion, and their interactions proceed concurrently. These services are likely running in parallel, but there's nothing stopping these services from both running on a single core server, in which case they would not be in parallel and would instead be taking turns on the one processor.

2 Concurrent Abstractions

Concurrency is tricky, and as programmers are want to do, many levels of abstraction have been built to make the task easier. These abstractions start at roughly the operating system level, which abstracts over an arbitrary number of hardware cores, and ends with programming languages which end up imitating an operating system in order to get a similar level of abstraction.

2.1 The Operating System

Back when computers often only had a single CPU core, it was necessary for the operating system to provide abstractions to allow multiple programs to run at once. In order to multitask, the operating system needed to give programs a 'process' abstraction, in which the program could view itself as the only running program on the machine, or rather the exclusive user of a hardware thread. With only one hardware thread and many programs, the OS switched between them quickly in order to simulate multiple threads. To users, especially of operating systems with graphical user interfaces, it appeared as though the programs were all running at once instead of time sharing.

This abstraction was extended with the widespread release of multi-core machines. Instead of time sharing over one hardware thread, the operating system could now spread the workload out over multiple. This means programs physically running at the same time, as opposed to switching back and forth. The operating system still provides process abstractions. In addition, operating systems typically also provide a 'threads' abstraction. Programs can split themselves up into multiple threads of execution, each thread having less overhead than a 'process'. These threads often share a memory space and can communicate between each other using this shared memory.

2.2 Shared State

The concurrency model of lanugages like C, C++, and Java is primarily threads with shared state or shared memory. In this model, different threads of execution communicate amoungst each other using shared memory. With this model comes additional abstractions to handle safely interacting with shared memory.

Threads are the primary abstraction in the shared state model. These threads largely line up with the concept of operating system threads. Indeed, in C and C++ these are exactly the threads that are used. In languages like Java however, they have their own concept of threads which are lighter weight that OS threads. Java, running on a virtual machine, is able to create and manage this abstraction on it's own. In that case, the Java threads are scheduled on the JVM and the JVM translates this to interactions with operating system / hardware.

When you have multiple threads of execution accessing the same memory at the same time, you run into new classes of issues not present in sequential programs. One such class of issues is race conditions, where two threads access a shared resource at the same time. In this scenario, the threads may have different effects on that resource resulting in conflicting updates to the shared state. For example, if two threads access a counter variable at the same time, they both might read the same value from memory, but if they go to update the value with different numbers, only one of them will actually be stored and the other will be overwritten.

2.2.1 Locks

Locks are an abstraction that is primarily used to resolve race conditions and other problems arising from multiple threads trying to access the same resource at the same time. In it's simplest form, a lock on a resource can be acquired by a single thread, which prevents other threads from accessing that resource until the lock is released. In this way, locks allow for mutual exclusion when accessing shared resources (and this kind of lock is often called a mutex). There are various types of locks, for example a read-write lock is a lock that allows either multiple read-only locks on a resource or a single write lock. This is useful because the value of a shared resource can be read by multiple threads simultaneously without issue so long as the value is not changed. If the value is going to be changed, it must happen under mutual exclusion to ensure there is no confusion over it's value.

Locks help solve race conditions in shared memory concurrency environments, but they can introduce their own problems. Consider the scenario where multiple threads need to access a shared resource to make progress. If one of the threads aquires a mutual exlusion lock on the resource then fails to halt or to release the lock, then the other threads are stuck in a blocked state and will fail to make progress. If the thread with the lock fails to halt or needs to access a lock held by one of the blocked threads, then the whole program will hang. This situation is called a deadlock. The resources needed to make progress are held under locks that will not be released.

Similarly, consider the scenario where two threads need access to two shared resources in order to make progress. Consider that each thread may try to place a lock on differing resources. If they then try to place a lock on the other resource, they may find it's being held and so they release their lock then try again from the beginning. In this scenario, it's possible that both threads keep trying to lock the shared resources but are unable and so release their locks, constantly oscillating between locked and unlocked and unable to make progress. This situation is called live lock.

2.2.2 Semaphore

2.2.3 Atoms / Atomic Operations

2.2.4 Synchronization

2.3 Event loop / Time Sharing

Some languages provide an event loop abstraction. This is similar to the time sharing an operating system gives to processes. The abstraction allows for concurrency but without parallelism. This is how Javascript operates. Golang is specialized for things like this too. Most languages provide some level of support for this kind of abstraction, but in some cases it's built on top of the language, not an integrated language feature.

- How does the term asynchronous fit into all of this?

2.4 Message Passing

Some languages rely on shared state need mechanisms like locks and semaphores to manage it.

Other languages use no shared state and rely on message passing to share data between parallel tasks.

These abstractions enforce separation between different threads of computation and have strict ways in which they can interact. They sometimes have no shared state and communicate entirely via message passing.

This is necessarily the mechanism by which physically separate machines/threads interact. For example, in a high performance computing cluster or in a distributed system.

- Example: Instant messaging app. Inter process and inter host.

2.4.1 Channels

2.4.2 Messages

- Maybe talk about how GPU programming is basically a combination of Shared state programming and message passing? Like, it's shared state on the GPU side but communication between the GPU and CPU is (necessarily?) message passing.
- Maybe talk about distributed programming here?
- Maybe talk about client/server paradigms here?

2.5 Actors

2.6 Recreating the Operating System

Some lanugages give abstractions to the programmer that have strict separation and communication requirements but that mirrors an operating system in the level of control and the structure it imposes. See Racket.

- Racket, Revenge of the Son of the Lisp Machine. Custodians, threads, etc.
- Has threads which are threads of control with their own I/O, control flow, and resources.
- How do threads communicate?
- Uses custodians for resource control. Threads, I/O ports, eventspaces.
- Custodians are hierarchical.

3 Example Use Cases for Concurrency

This section will be a showcase of various concurrency abstractions and how they are typically used. Maybe this would be a good place to talk about parallel algorithms?

3.1 Shared State

3.2 Event Loop

3.3 Message Passing

3.4 Recreating the Operating System

4 *DEPRECATED?* Concurrency in Modern Languages

This section will be an overview of the various parallel programming concepts modern day languages use. Each section will have an example of a typical (toy) parallel program written in one of these languages. Each section will talk about what types of parallel programming tasks each language/system is and isn't good for.

4.1 C, C++, and Java

Shared state and traditional CPU parallelism problems.

4.2 CUDA, OpenGL, and OpenMPI

Modern day GPU programming

4.3 Erlang

Message passing and no shared state.

4.4 Rust and Go

Not sure, but I know both of these languages promise to make parallel computing effortless. I think they both are a combination of C and Erlang style parallelism.

4.5 Javascript

Not parallel but concurrent (with the exception of web workers). Talk about the event loop and maybe talk about web workers too.

4.6 Racket

Go over Racket's implementation of threads and the Concurrent ML concepts they employ.

5 Current State of the Art

Depending on time, this section would talk about some of the research underway on new parallel programming paradigms, techniques, and technologies. Maybe mention Sam Caldwell.