A Survey of Concurrency

Kyle Clapper

October 5, 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.

- This section should probably also motivate why we need concurrency in programs.

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.

- Should I mention that concurrency could also refer to distributed systems like web servers? HTTP being a good example where you have concurrent processes communicating with each other (not necessarily in parallel).

1.2 Task Parallelism vs Data Parallelism

This is basically the difference between GPU parallelism and CPU parallelism. SIMD vs running different tasks on different CPUs.

2 Concurrent Abstractions

This section will discuss various levels of concurrent programming abstractions.

2.1 The Operating System

A brief explaination of how concurrency and parallelism work from an OS point of view. This forms the basis of concurrent programming for most programs.

2.2 Shared State

The concurrency model of languages like C and Java. This section will talk about shared state concurrency and how the language abstractions largely mirror the operating system abstractions but with more control given to the programmer.

2.2.1 Atoms / Atomic Operations

2.2.2 Locks

2.2.3 Mutex

2.2.4 Semaphore

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 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.