Extracted from:

Seven Concurrency Models in Seven Weeks

When Threads Unravel

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Seven Concurrency Models in Seven Weeks

When Threads Unravel

Paul Butcher

Series editor: *Bruce Tate*
Development editor: *Jacquelyn Carter*

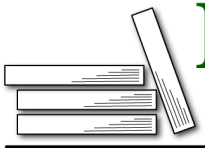
Seven Concurrency Models in Seven Weeks

When Threads Unravel

Paul Butcher

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Bruce A. Tate (series editor)
Jacquelyn Carter (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-937785-65-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2014

CHAPTER 1

Introduction

Concurrent programming is nothing new, but it's recently become a hot topic. Languages like Erlang, Haskell, Go, Scala, and Clojure are gaining mindshare, in part thanks to their excellent support for concurrency.

The primary driver behind this resurgence of interest is what's become known as the "multicore crisis." Moore's law continues to deliver more transistors per chip,[1] but instead of those transistors being used to make a single CPU faster, we're seeing computers with more and more cores.

As Herb Sutter said, "The free lunch is over."[2] You can no longer make your code run faster by simply waiting for faster hardware. These days if you need more performance, you need to exploit multiple cores, and that means exploiting parallelism.

Concurrent or Parallel?

This is a book about concurrency, so why are we talking about parallelism? Although they're often used interchangeably, *concurrent* and *parallel* refer to related but different things.

Related but Different

A *concurrent* program has multiple logical *threads of control*. These threads may or may not run in parallel.

A *parallel* program potentially runs more quickly than a sequential program by executing different parts of the computation simultaneously (in parallel). It may or may not have more than one logical thread of control.

1. http://en.wikipedia.org/wiki/Moore's_law
2. http://www.gotw.ca/publications/concurrency-ddj.htm

An alternative way of thinking about this is that concurrency is an aspect of the problem domain—your program needs to handle multiple simultaneous (or near-simultaneous) events. Parallelism, by contrast, is an aspect of the solution domain—you want to make your program faster by processing different portions of the problem in parallel.

As Rob Pike puts it,[3]

> Concurrency is about dealing with lots of things at once.
>
> Parallelism is about doing lots of things at once.

So is this book about concurrency or parallelism?

Joe asks:

Concurrent or Parallel?

My wife is a teacher. Like most teachers, she's a master of multitasking. At any one instant, she's only doing one thing, but she's having to deal with many things concurrently. While listening to one child read, she might break off to calm down a rowdy classroom or answer a question. This is concurrent, but it's not parallel (there's only one of her).

If she's joined by an assistant (one of them listening to an individual reader, the other answering questions), we now have something that's both concurrent and parallel.

Imagine that the class has designed its own greeting cards and wants to mass-produce them. One way to do so would be to give each child the task of making five cards. This is parallel but not (viewed from a high enough level) concurrent—only one task is underway.

Beyond Sequential Programming

What parallelism and concurrency have in common is that they both go beyond the traditional sequential programming model in which things happen one at a time, one after the other. We're going to cover both concurrency and parallelism in this book (if I were a pedant, the title would have been *Seven Concurrent and/or Parallel Programming Models in Seven Weeks*, but that wouldn't have fit on the cover).

Concurrency and parallelism are often confused because traditional threads and locks don't provide any direct support for parallelism. If you want to

3. http://concur.rspace.googlecode.com/hg/talk/concur.html

exploit multiple cores with threads and locks, your only choice is to create a concurrent program and then run it on parallel hardware.

This is unfortunate because concurrent programs are often *nondeterministic*—they will give different results depending on the precise timing of events. If you're working on a genuinely concurrent problem, nondeterminism is natural and to be expected. Parallelism, by contrast, doesn't necessarily imply nondeterminism—doubling every number in an array doesn't (or at least, shouldn't) become nondeterministic just because you double half the numbers on one core and half on another. Languages with explicit support for parallelism allow you to write parallel code without introducing the specter of nondeterminism.

Parallel Architecture

Although there's a tendency to think that parallelism means multiple cores, modern computers are parallel on many different levels. The reason why individual cores have been able to get faster every year, until recently, is that they've been using all those extra transistors predicted by Moore's law in parallel, both at the bit and at the instruction level.

Bit-Level Parallelism

Why is a 32-bit computer faster than an 8-bit one? Parallelism. If an 8-bit computer wants to add two 32-bit numbers, it has to do it as a sequence of 8-bit operations. By contrast, a 32-bit computer can do it in one step, handling each of the 4 bytes within the 32-bit numbers in parallel.

That's why the history of computing has seen us move from 8- to 16-, 32-, and now 64-bit architectures. The total amount of benefit we'll see from this kind of parallelism has its limits, though, which is why we're unlikely to see 128-bit computers soon.

Instruction-Level Parallelism

Modern CPUs are highly parallel, using techniques like *pipelining*, *out-of-order execution*, and *speculative execution*.

As programmers, we've mostly been able to ignore this because, despite the fact that the processor has been doing things in parallel under our feet, it's carefully maintained the illusion that everything is happening sequentially.

This illusion is breaking down, however. Processor designers are no longer able to find ways to increase the speed of an individual core. As we move into a multicore world, we need to start worrying about the fact that instructions

aren't handled sequentially. We'll talk about this more in *Memory Visibility*, on page ?.

Data Parallelism

Data-parallel (sometimes called SIMD, for "single instruction, multiple data") architectures are capable of performing the same operations on a large quantity of data in parallel. They're not suitable for every type of problem, but they can be extremely effective in the right circumstances.

One of the applications that's most amenable to data parallelism is image processing. To increase the brightness of an image, for example, we increase the brightness of each pixel. For this reason, modern GPUs (graphics processing units) have evolved into extremely powerful data-parallel processors.

Task-Level Parallelism

Finally, we reach what most people think of as parallelism—multiple processors. From a programmer's point of view, the most important distinguishing feature of a multiprocessor architecture is the memory model, specifically whether it's shared or distributed.

In a *shared-memory* multiprocessor, each processor can access any memory location, and interprocessor communication is primarily through memory, as you can see in Figure 1, *Shared memory*, on page 5.

Figure 2, *Distributed memory*, on page 5 shows a *distributed-memory* system, where each processor has its own local memory and where interprocessor communication is primarily via the network.

Because communicating via memory is typically faster and simpler than doing so over the network, writing code for shared memory-multiprocessors is generally easier. But beyond a certain number of processors, shared memory becomes a bottleneck—to scale beyond that point, you're going to have to tackle distributed memory. Distributed memory is also unavoidable if you want to write fault-tolerant systems that use multiple machines to cope with hardware failures.

Concurrency: Beyond Multiple Cores

Concurrency is about a great deal more than just exploiting parallelism—used correctly, it allows your software to be responsive, fault tolerant, efficient, and simple.

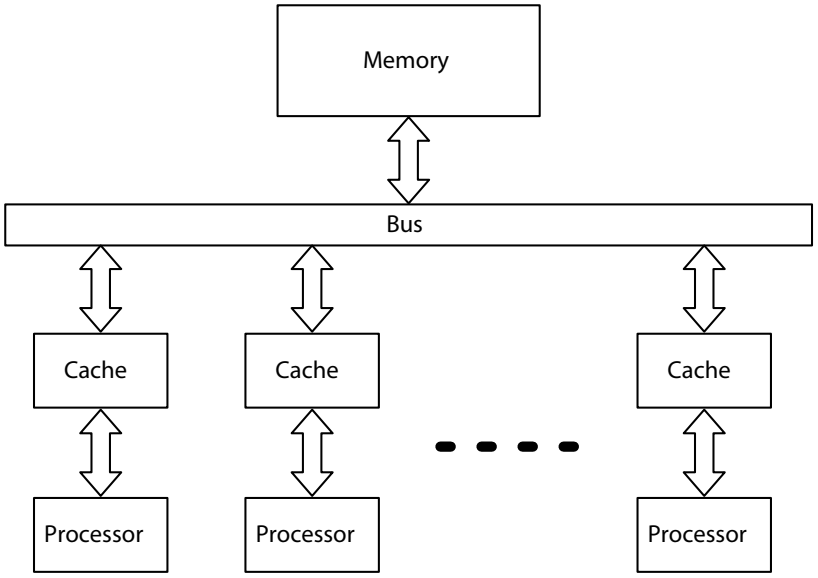


Figure 1—Shared memory

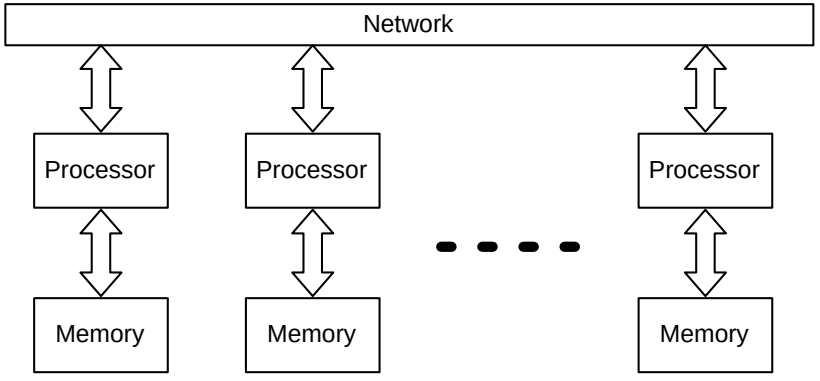


Figure 2—Distributed memory

Concurrent Software for a Concurrent World

The world is concurrent, and so should your software be if it wants to interact effectively.

Your mobile phone can play music, talk to the network, and pay attention to your finger poking its screen, all at the same time. Your IDE checks the syntax

of your code in the background while you continue to type. The flight system in an airplane simultaneously monitors sensors, displays information to the pilot, obeys commands, and moves control surfaces.

Concurrency is the key to *responsive* systems. By downloading files in the background, you avoid frustrated users having to stare at an hourglass cursor. By handling multiple connections concurrently, a web server ensures that a single slow request doesn't hold up others.

Distributed Software for a Distributed World

Sometimes geographical distribution is a key element of the problem you're solving. Whenever software is distributed on multiple computers that aren't running in lockstep, it's intrinsically concurrent.

Among other things, distributing software helps it handle failure. You might locate half your servers in a data center in Europe and the others in the United States, so that a power outage at one site doesn't result in global downtime. This brings us to the subject of resilience.

Resilient Software for an Unpredictable World

Software contains bugs, and programs crash. Even if you could somehow produce perfectly bug-free code, the hardware that it's running on will sometimes fail.

Concurrency enables resilient, or fault-tolerant, software through *independence* and *fault detection*. Independence is important because a failure in one task should not be able to bring down another. And fault detection is critical so that when a task fails (because it crashes or becomes unresponsive, or because the hardware it's running on dies), a separate task is notified so that it can take remedial action.

Sequential software can never be as resilient as concurrent software.

Simple Software in a Complex World

If you've spent hours wrestling with difficult-to-diagnose threading bugs, it might be hard to believe, but a concurrent solution can be simpler and clearer than its sequential equivalent when written in the right language with the right tools.

This is particularly true whenever you're dealing with an intrinsically concurrent real-world problem. The extra work required to translate from the concurrent problem to its sequential solution clouds the issue. You can avoid this extra work by creating a solution with the same structure as the problem:

rather than a single complex thread that tries to handle multiple tasks when they need it, create one simple thread for each.

The Seven Models

The seven models covered in this book have been chosen to provide a broad view of the concurrency and parallelism landscape.

Threads and locks: Threads-and-locks programming has many well-understood problems, but it's the technology that underlies many of the other models we'll be covering and it is still the default choice for much concurrent software.

Functional programming: Functional programming is becoming increasingly prominent for many reasons, not the least of which is its excellent support for concurrency and parallelism. Because they eliminate mutable state, functional programs are intrinsically thread-safe and easily parallelized.

The Clojure Way—separating identity and state: The Clojure language has popularized a particularly effective hybrid of imperative and functional programming, allowing the strengths of both approaches to be leveraged in concert.

Actors: The actor model is a general-purpose concurrent programming model with particularly wide applicability. It can target both shared- and distributed-memory architectures and facilitate geographical distribution, and it provides particularly strong support for fault tolerance and resilience.

Communicating Sequential Processes: On the face of things, Communicating Sequential Processes (CSP) has much in common with the actor model, both being based on message passing. Its emphasis on the channels used for communication, rather than the entities between which communication takes place, leads to CSP-based programs having a very different flavor, however.

Data parallelism: You have a supercomputer hidden inside your laptop. The GPU utilizes data parallelism to speed up graphics processing, but it can be brought to bear on a much wider range of tasks. If you're writing code to perform finite element analysis, computational fluid dynamics, or anything else that involves significant number-crunching, its performance will eclipse almost anything else.

The Lambda Architecture: Big Data would not be possible without parallelism—only by bringing multiple computing resources to bear can we

contemplate processing terabytes of data. The Lambda Architecture combines the strengths of MapReduce and stream processing to create an architecture that can tackle a wide variety of Big Data problems.

Each of these models has a different sweet spot. As you read through each chapter, bear the following questions in mind:

- Is this model applicable to solving concurrent problems, parallel problems, or both?
- Which parallel architecture or architectures can this model target?
- Does this model provide tools to help you write resilient or geographically distributed code?

In the next chapter we'll look at the first model, Threads and Locks.