Learn Concurrent Programming with Go

James Cutajar

MANNING

Part 1

Foundations

How can we write instructions so that some actions are performed at the same time as others? In part 1 of this book, we'll explore the basics of how we can model concurrency in our programming. We'll see how modeling and executing concurrent programs require help from the hardware, the operating system, and the programming language.

When we develop concurrent programs, we encounter a new set of programming errors that are not present in sequential code. Known as race conditions, these errors can be some of the most difficult to identify and fix. A huge part of concurrent programming involves learning how to prevent these types of bugs in our code. In this part of the book, we'll learn about race conditions and then discuss various techniques for avoiding them.

1 Stepping into concurrent programming

This chapter covers

- Introducing concurrent programming
- Improving performance with concurrent execution
- Scaling our programs

Meet Jane Sutton. Jane has been working at HSS International Accountancy as a software developer for three months. In her latest project, she has been looking at a problem in the payroll system. The payroll software module runs at the end of the month after the close of business, and it computes all the salary payments for the HSS clients' employees. Jane's manager has arranged a meeting with the product owner, the infrastructure team, and a sales representative to try to get to the bottom of the problem. Unexpectedly, Sarika Kumar, CTO, has joined the meeting room via video call.

Thomas Bock, the product owner, starts: "I don't understand. The payroll module has been working fine for as long as I can remember. Suddenly, last month, the payment calculations weren't completed on time, and we got loads of complaints

from our clients. It made us look really unprofessional to Block Entertainment, our new and biggest client yet, with them threatening to go to our competitor."

Jane's manager, Francesco Varese, chimes in: "The problem is that the calculations are too slow and take too long. They are slow because of their complex nature, considering many factors such as employee absences, joining dates, overtime, and a thousand other factors. Parts of the software were written more than a decade ago in C++. There are no developers left in the firm who understand how this code works."

"We're about to sign up our biggest client ever, a company with over 30,000 employees. They've heard about our payroll problem, and they want to see it resolved before they proceed with the contract. It's really important that we fix this as soon as possible," replies Rob Gornall from the Sales and Acquisitions department.

"We've tried adding more processor cores and memory to the server that runs the module, but this made absolutely no difference. When we execute the payroll calculation using test data, it's taking the same amount of time, no matter how many resources we allocate. It's taking more than 20 hours to calculate all the clients' payrolls, which is too long for our clients," continues Frida Norberg from Infrastructure.

It's Jane's turn to finally speak. As the firm's newest employee, she hesitates a little but manages to say, "If the code is not written in a manner that takes advantage of the additional cores, it won't matter if you allocate multiple processors. The code needs to use concurrent programming for it to run faster when you add more processing resources."

Everyone seems to have acknowledged that Jane is the most knowledgeable about the subject. There is a short pause. Jane feels as if everyone wants her to come up with some sort of answer, so she continues. "Right. Okay. I've been experimenting with a simple program written in Go. It divides the payroll into smaller employee groups and then calls the payroll module with each group as input. I've programmed it so that it calls the module concurrently using multiple goroutines. I'm also using a Go channel to load-balance the workload. At the end, I have another goroutine that collects the results via another channel."

Jane looks around quickly and sees blank looks on everyone's faces, so she adds, "In simulations, it's at least five times faster on the same multicore hardware. There are still a few tests to run to make sure there are no race conditions, but I'm pretty sure that I can make it run even faster, especially if I get some help from accounting to migrate some of the old C++ logic into clean Go concurrent code."

Jane's manager has a big smile on his face now. Everyone else in the meeting seems surprised and speechless. The CTO finally speaks up and says, "Jane, what do you need to get this done by the end of the month?"

Concurrent programming is a skill that is increasingly sought after by tech companies. It is a technique used in virtually every field of development, from web development to game programming, backend business logic, mobile applications, crypto, and many others. Businesses want to utilize hardware resources to their full capacity, as this saves them time and money. To accomplish this, they understand that they have to hire the right talent—developers who can write scalable concurrent applications.

1.1 About concurrency

In this book, we will focus on principles and patterns of concurrent programming. How can we program instructions that happen at the same time? How can we manage concurrent executions so they don't step over each other? What techniques should we use to have executions collaborate toward solving a common problem? When and why should we use one form of communication over another? We will answer all these questions and more by making use of the Go programming language. Go gives us a full set of tools to illustrate these concepts.

If you have little or no experience in concurrency but have some experience in Go or a similar C-style language, this book is ideal. This book starts with a gentle introduction to concurrency concepts in the operating system and describes how Go uses them to model concurrency. We'll then move on to explain race conditions and why they occur in some concurrent programs. Later, we'll discuss the two main ways we can implement communication between our executions: memory sharing and message passing. In the final chapters of this book, we'll discuss concurrency patterns, deadlocks, and some advanced topics such as spinning locks.

Apart from helping us to get hired or promoted as developers, knowing concurrent programming gives us a wider set of skills that we can employ in new scenarios. For example, we can model complex business interactions that happen at the same time. We can also use concurrent programming to improve our software's responsiveness by picking up tasks swiftly. Unlike sequential programming, concurrent programming can make use of multiple CPU cores, which allows us to increase the work done by our programs by speeding up their execution. Even with a single CPU core, concurrency offers benefits because it enables time-sharing and lets us perform tasks while we're waiting for I/O operations to complete. Let's now look at some of these scenarios in more detail.

1.2 Interacting with a concurrent world

We live and work in a concurrent world. The software that we write models complex business processes that interact concurrently. Even the simplest of businesses typically have many of these concurrent interactions. For example, consider multiple people ordering online at the same time or a consolidation process grouping packages together while coordinating with ongoing shipments, as shown in figure 1.1.

In our everyday life, we deal with concurrency all the time. Every time we drive a car, we interact with multiple concurrent actors, such as other cars, cyclists, and pedestrians. At work, we may put a task on hold while we're waiting for an email reply and pick up the next task. When cooking, we plan our steps so we maximize our productivity and shorten the cooking time. Our brain is perfectly comfortable managing concurrent behavior. In fact, it does this all the time without us even noticing.

Concurrent programming is about writing code so that multiple tasks and processes can execute and interact at the same time. If two customers place an order simultaneously and only one stock item remains, what happens? If the price of a flight ticket goes up every time a client buys a ticket, what happens when multiple tickets are

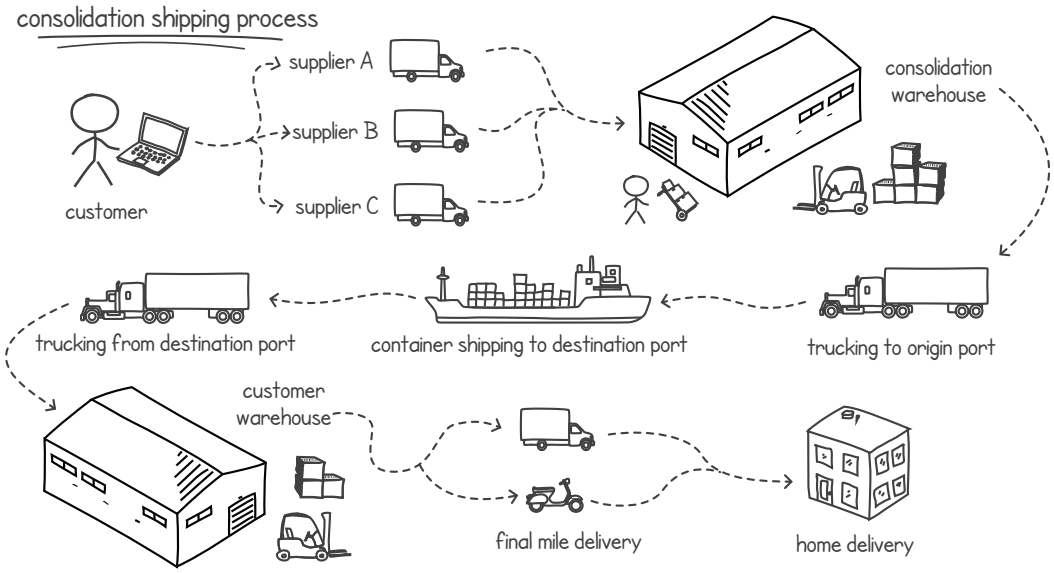

Figure 1.1 A consolidation shipping process showing complex concurrent interactions

booked at the same exact instant? If we have a sudden increase in load due to extra demand, how will our software scale when we increase the processing and memory resources? These are all scenarios that developers deal with when they are designing and programming concurrent software.

1.3 Increasing throughput

For the modern developer, it is increasingly important to understand how to program concurrently. This is because the hardware landscape has changed over the years to benefit this type of programming.

Prior to multicore technology, processor performance increased proportionally to clock frequency and transistor count, roughly doubling every two years. Processor engineers started hitting physical limits due to overheating and power consumption, which coincided with the explosion of more mobile hardware, such as notebooks and smartphones. To reduce excessive battery consumption and CPU overheating while increasing processing power, engineers introduced multicore processors.

In addition, the rise of cloud computing services has given developers easy access to large, cheap processing resources where they can run their code. This extra computational power can only be harnessed effectively if our code is written in a manner that takes full advantage of the extra processing units.

DEFINITION *Horizontal scaling* is when we improve system performance by distributing the load over multiple processing resources, such as processors and server machines (see figure 1.2). *Vertical scaling* is when we improve the existing resources, such as by getting a faster processor.

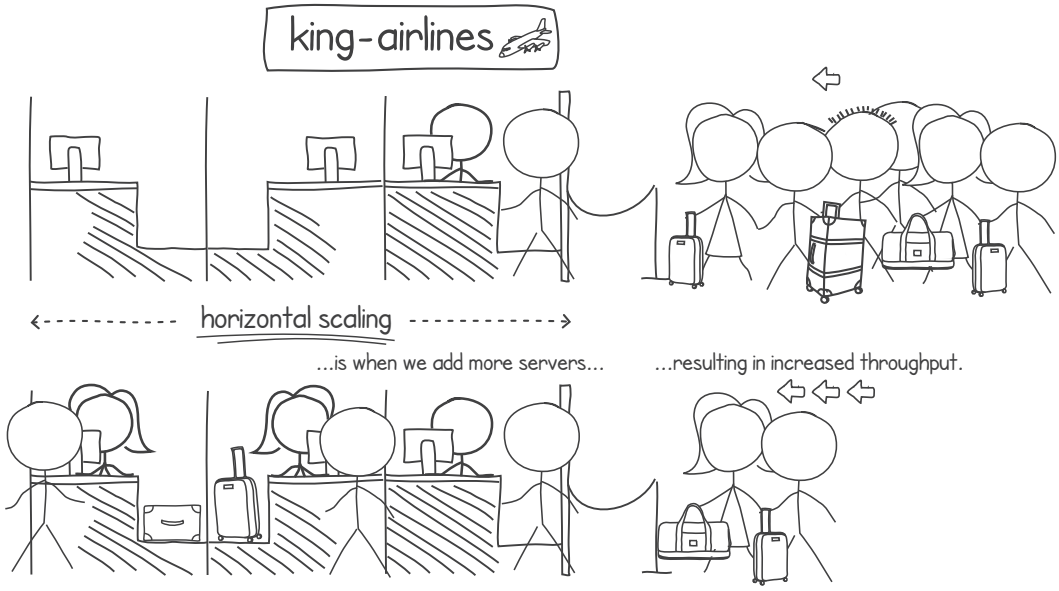

Figure 1.2 Improving performance by adding more processors

Having multiple processing resources means we can scale horizontally. We can use the extra processors to execute tasks in parallel and finish them more quickly. This is only possible if we write code in a way that takes full advantage of the extra processing resources.

What about a system that has only one processor? Is there any advantage to writing concurrent code if our system does not have multiple processors? It turns out that writing concurrent programs has a benefit even in this scenario.

Most programs spend only a small proportion of their time executing computations on the processor. Think, for example, about a word processor that waits for input from the keyboard, or a text-file search utility that spends most of its running time waiting for portions of the text files to load from disk. We can have our program perform different tasks while it's waiting for I/O. For example, the word processor could perform a spell check on the document while the user is thinking about what to type next. We can have the file search utility look for a match with the file that we already loaded in memory while we are reading the next file into another portion of memory.

As another example, think of cooking or baking a favorite dish. We can make more effective use of our time if, while the dish is in the oven or on the stove, we perform some other actions instead of just waiting around (see figure 1.3). In this way, we are making more effective use of our time, and we are more productive. This is analogous to our program executing other instructions on the CPU while it waits for a network message, user input, or a file to be written. This means our program can get more work done in the same amount of time.

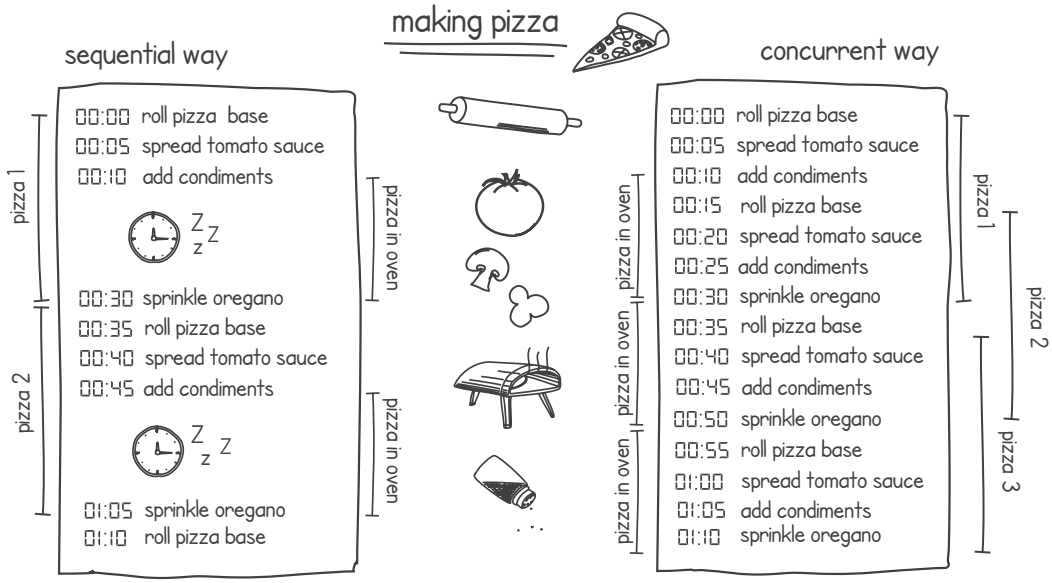

Figure 1.3 Even with one processor, we can improve performance if we utilize idle times.

1.4 Improving responsiveness

Concurrent programming makes our software more responsive because we don't need to wait for one task to finish before responding to a user's input. Even if we have one processor, we can always pause the execution of a set of instructions, respond to the user's input, and then continue with the execution while we're waiting for the next user's input.

If we again think of a word processor, multiple tasks might be running in the background while we are typing. There is a task that listens to keyboard events and displays each character on the screen. We might also have a task that checks our spelling and grammar in the background. Another task might be running to give us stats on our document (word count, page count, etc.). Periodically, we may have another task that autosaves our document. All these tasks running together give the impression that they are somehow running simultaneously, but what's happening is that these tasks are being fast-switched by the operating system on CPUs. Figure 1.4 illustrates a simplified timeline showing these three tasks executing on a single processor. This interleaving system is implemented by using a combination of hardware interrupts and operating system traps.

We'll go into more detail on operating systems and concurrency in the next chapter. For now, it's important to realize that if we didn't have this interleaving system, we would have to perform each task one after the other. We would have to type a sentence, then click the spell check button, wait for it to complete, and then click another button and wait for the document stats to appear.

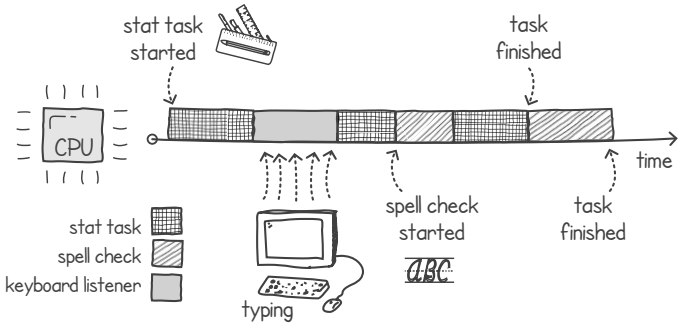

Figure 1.4 Simplified task interleaving in a word processor

1.5 Programming concurrency in Go

Go is a very good language to use when learning about concurrent programming because its creators designed it with high-performance concurrency in mind. Their aim was to produce a language that was efficient at runtime, readable, and easy to use.

1.5.1 Goroutines at a glance

Go uses a lightweight construct, called a *goroutine,* to model the basic unit of concurrent execution. As we shall see in the next chapter, goroutines give us a system of user-level threads running on a set of kernel-level threads and managed by Go's runtime.

Given the lightweight nature of goroutines, the premise of the language is that we should focus mainly on writing correct concurrent programs, letting Go's runtime and hardware mechanics deal with parallelism. The principle is that if you need something to be done concurrently, create a goroutine to do it. If you need many things done concurrently, create as many goroutines as you need, without worrying about resource allocation. Then, depending on the hardware and environment that your program is running on, your solution will scale.

In addition to goroutines, Go provides us with many abstractions that allow us to coordinate the concurrent executions on a common task. One of these abstractions is known as a *channel.* Channels allow two or more goroutines to pass messages to each other. This enables the exchange of information and synchronization of the multiple executions in an easy and intuitive manner.

1.5.2 Modeling concurrency with CSP and primitives

In 1978, C.A.R. Hoare first described *communicating sequential processes* (CSP) as a formal language for expressing concurrent interactions. Many languages, such as Occam and Erlang, have been influenced by CSP. Go tries to implement many of CSP's ideas, such as the use of synchronized channels.

This concurrency model of having isolated goroutines communicating and synchronizing using channels (see figure 1.5) reduces the risk of race conditions—types of programming errors that occur in bad concurrent programming and that are typically very hard to debug and lead to data corruption and unexpected behavior. This

type of modeling concurrency is more akin to how concurrency happens in our everyday lives, such as when we have isolated executions (people, processes, or machines) working concurrently, communicating with each other by sending messages back and forth.

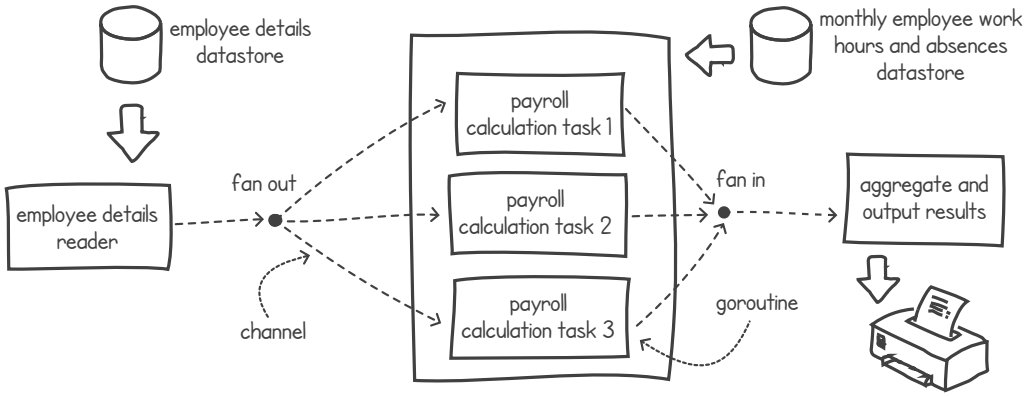

Figure 1.5 A concurrent Go application using CSP

Depending on the problem, the classic concurrency primitives used with memory sharing (such as mutexes and condition variables, found in many other languages) will sometimes do a better job and result in better performance than using CSP-style programming. Luckily for us, Go provides us with these tools in addition to the CSP-style tools. When CSP is not the appropriate model to use, we can fall back on the other classic primitives.

In this book, we will purposely start with memory sharing and synchronization using classic primitives. The idea is that by the time we get to discussing CSP-style concurrent programming, you will have a solid foundation in the traditional locking and synchronization primitives.

1.5.3 Building our own concurrency tools

In this book, you will learn how to use various tools to build concurrent applications. This includes concurrency constructs such as mutex, condition variables, channels, semaphores, and so on.

Knowing how to use these concurrency tools is good, but what about understanding their inner workings? Here, we'll go one step further and take the approach of building them together from scratch, even if they are available in Go's libraries. We will pick common concurrency tools and see how they can be implemented using other concurrency primitives as building blocks. For example, Go doesn't come with a bundled semaphore implementation, so apart from understanding how and when to use semaphores, we'll go about implementing one ourselves. We'll also do this for some of the tools that are available in Go, such as waitgroups and channels.

This idea is analogous to having the knowledge to implement well-known algorithms. We might not need to know how to implement a sorting algorithm to use a sorting function; however, learning how the algorithm works exposes us to different scenarios and new ways of thinking, making us better programmers. We can then apply those scenarios to different problems. In addition, knowing how a concurrency tool is built allows us to make better-informed decisions about when and how to use it.

1.6 Scaling performance

Performance scalability is the measure of how well a program speeds up in proportion to the increase in the number of resources available to the program. To understand this, let's try to make use of a simple analogy.

Imagine a world where we are property developers. Our current project is to build a small multi-story residential house. We give the architectural plan to a builder, and they set off to finish the small house. The work is all completed in a period of eight months.

As soon as that project is finished, we get another request for the same build but in another location. To speed things up, we hire two builders instead of one. This time around, the builders complete the house in just four months.

The next time we are asked to build the same house, we hire even more help, so that the house is finished quicker. This time we pay four builders, and it takes them two and a half months to complete. The house has cost us a bit more to build than the previous one. Paying four builders for two and a half months costs more than paying two builders for four months (assuming they all charge the same rate).

We repeat the experiment twice more, once with 8 builders and then with 16. With both 8 and 16 builders, the house takes two months to complete. It seems that no matter how many hands we put on the job, the build cannot be completed in less than two months. In geek speak, we say that we have hit our *scalability limit*. Why does this happen? Why can't we continue to double our resources (people, money, or processors) and always reduce the time spent by half?

1.6.1 Amdahl's law

In 1967, Gene Amdahl, a computer scientist, presented a formula at a conference that measured speedup with regard to a problem's parallel-to-sequential ratio. This became known as Amdahl's law.

> **DEFINITION** *Amdahl's law* states that the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used.

In our house build scenario, the scalability is limited by various factors. For starters, our approach to solving the problem might be limiting us. For example, one cannot construct the second floor before constructing the first. In addition, several parts of the build can only be done sequentially. For example, if a single road leads to the

building site, only one transport can use the road at any point in time. In other words, some parts of the building process are sequential (one after the other), and other parts can be done in parallel (at the same time). These factors influence and limit the scalability of our task.

Amdahl's law tells us that the non-parallel parts of an execution act as a bottleneck and limit the advantage of parallelizing the execution. Figure 1.6 shows this relationship between the theoretical speedup obtained as we increase the number of processors.

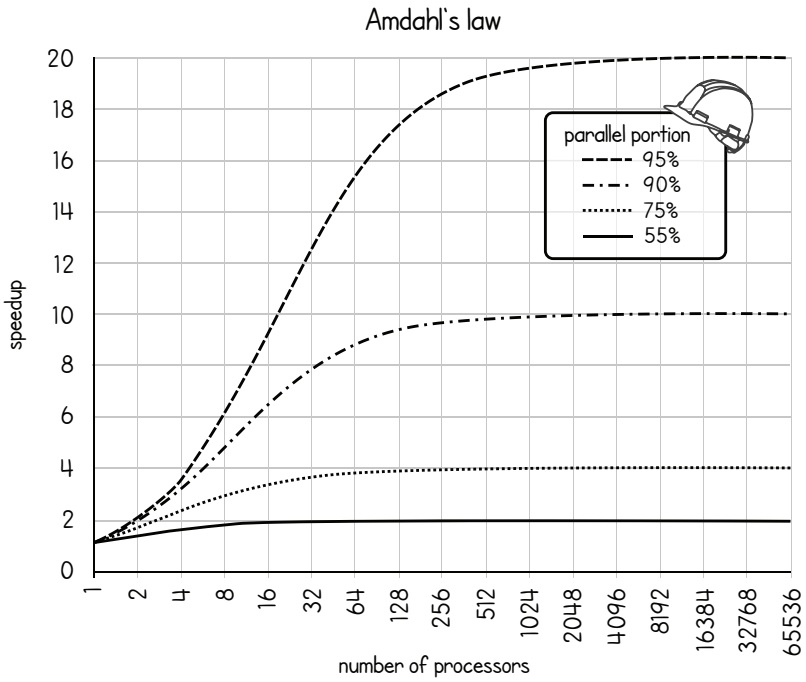

Figure 1.6 The speedup against the number of processors according to Amdahl's law

If we apply this chart to our construction problem, when we use a single builder and they spend 5% of their time on the parts that can only be done sequentially, the scalability follows the topmost line in our chart (95% parallel). This sequential portion is the part that can only be done by one person, such as trucking in the building materials through a narrow road.

As you can see from the chart, even with 512 people working on the construction, we would only finish the job about 19 times faster than if we had just 1 person. After this point, the situation does not improve much. We'll need more than 4,096 builders to finish the project just 20 times faster. We hit a hard limit around this number. Contracting more workers does not help at all, and we would be wasting our money.

The situation is even worse if a lower percentage of work is parallelizable. With 90%, we would hit this scalability limit around the 512-workers mark. With 75%, we

get there at 128 workers, and with 50% at just 16 workers. Notice that it's not just this limit that goes down—the speedup is also greatly reduced. When the work is 90%, 75%, and 50% parallelizable, we get maximum speedups of 10, 4, and 2, respectively.

Amdahl's law paints a pretty bleak picture of concurrent programming and parallel computing. Even with concurrent code that has a tiny fraction of serial processing, the scalability is greatly reduced. Thankfully, this is not the full picture.

1.6.2 Gustafson's law

In 1988, two computer scientists, John L. Gustafson and Edwin H. Barsis, reevaluated Amdahl's law and published an article addressing some of its shortcomings ("Reevaluating Amdah's Law," https://dl.acm.org/doi/pdf/10.1145/42411.42415). The article gives an alternative perspective on the limits of parallelism. Their main argument is that, in practice, the size of the problem changes when we have access to more resources.

To continue with our house-building analogy, if we did have thousands of builders available at our disposal, it would be wasteful to put them all into building a small house when we have future projects in the pipeline. Instead, we would try to put the optimal number of builders on our house construction and allocate the rest of the workers to other projects.

Suppose we were developing software and we had a large number of computing resources. If we noticed that utilizing half the resources resulted in the same software performance, we could allocate the extra resources to do other things, such as increasing the accuracy or quality of that software in other areas.

The second argument against Amdahl's law is that when you increase the problem's size, the non-parallel part of the problem typically does not grow in proportion with the problem size. In fact, Gustafson argues that for many problems, this remains constant. Thus, when you take these two points into account, the speedup can scale linearly with the available parallel resources. This relationship is shown in figure 1.7.

Gustafson's law tells us that as long as we find ways to keep our extra resources busy, the speedup should continue to increase and not be limited by the serial part of the problem. However, this is only true if the serial part stays constant as we increase the problem size, which, according to Gustafson, is the case in many types of programs.

To fully understand both Amdahl's and Gustafson's laws, let's take a computer game as an example. Let's say a particular computer game with rich graphics was written to make use of multiple computing processors. As time goes by and computers become more powerful, with more parallel processing cores, we can run that same game with a higher frame rate, giving us a smoother experience. Eventually, we get to a point where we're adding more processors, but the frame rate is not increasing further. This happens when we hit the speedup limit. No matter how many processors we add, the game won't run with higher frame rates. This is what Amdahl's law is telling us—that there is a speedup limit for a particular problem of fixed size if it has a non-parallel portion.

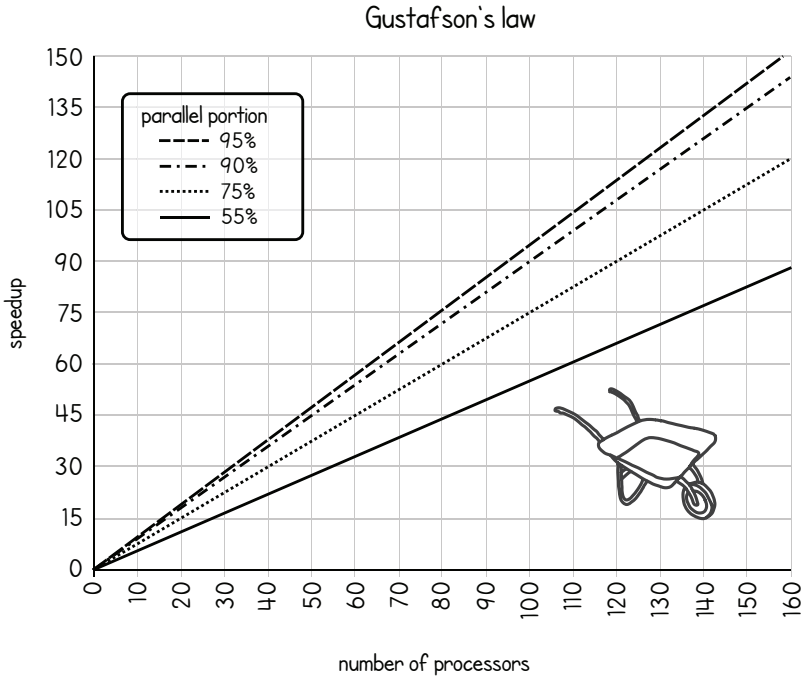

Figure 1.7 The speedup against the number of processors according to Gustafson's law

However, as technology improves and processors get more cores, the game designers will put those extra processing units to good use. Although the frame rate might not increase, the game can now contain more graphic detail and higher resolution due to the extra processing power. This is Gustafson's law in action. When we increase the resources, there is an expectation of an increase in the system's capabilities, and the developers will make good use of the extra processing power.

Summary

- Concurrent programming allows us to build more responsive software.
- Concurrent programs can also provide increased speedup when running on multiple processors.
- We can increase throughput even when we have only one processor if our concurrent programming makes effective use of the I/O wait times.
- Go provides us with goroutines, which are lightweight constructs for modeling concurrent executions.
- Go provides us with abstractions, such as channels, that enable concurrent executions to communicate and synchronize.
- Go allows us the choice of building our concurrent application either using the communicating sequential processes (CSP)-style model or using the classical primitives.

- Using a CSP-style model, we reduce the chance of certain types of concurrent errors; however, for certain problems, using the classical primitives will give us better results.
- Amdahl's law tells us that the performance scalability of a fixed-size problem is limited by the non-parallel parts of an execution.
- Gustafson's law tells us that if we keep on finding ways to keep our extra resources busy, the speedup should continue to increase and not be limited by the serial part of the problem.

2 Dealing with threads

This chapter covers

- Modeling concurrency in operating systems
- Differentiating between processes and threads
- Creating goroutines
- Differentiating between concurrency and parallelism

The operating system is the gatekeeper of our system resources. It decides when and which processes are given access to the various system resources, including processing time, memory, and network. As developers, we don't necessarily need to be experts on the inner workings of the operating system. However, we need to have a good understanding of how it operates and the tools it provides to make our lives as programmers easier.

We'll start this chapter by looking at how the operating system manages and allocates resources to run multiple jobs concurrently. In the context of concurrent programming, the operating system gives us various tools to help manage this concurrency. Two of these tools, processes and threads, represent the concurrent actors in our code. They may execute in parallel or interleave and interact with each other. We will look, in some detail, at the differences between the two. Later,