Extracted from:

Concurrent Data Processing in Elixir

Fast, Resilient Applications with OTP, GenState, Flow, and Broadway

The Pragmatic Bookshelf

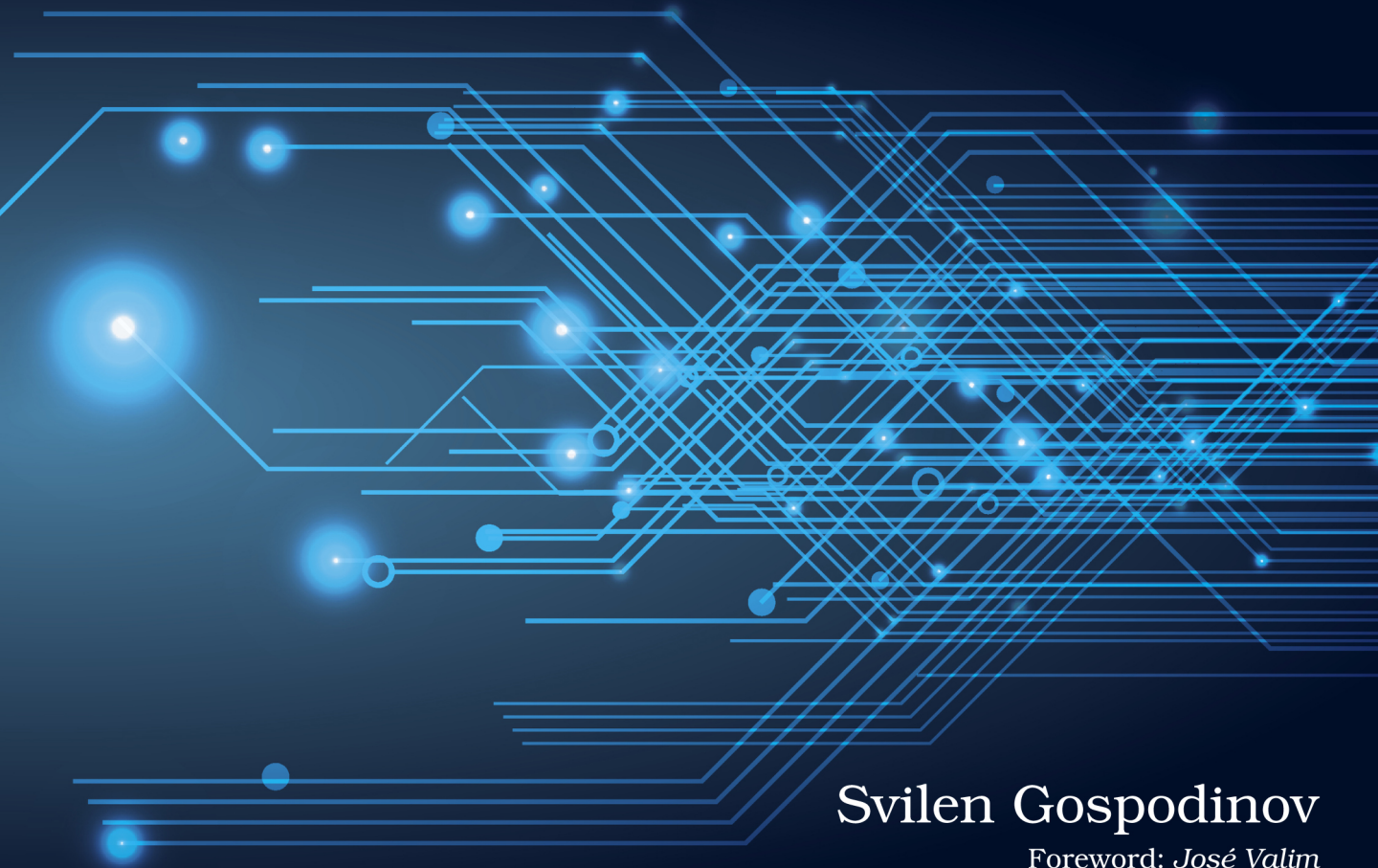
Raleigh, North Carolina

The Pragmatic Programmers

Your Elixir Source

Concurrent Data Processing in Elixir

Fast, Resilient Applications with OTP, GenStage, Flow, and Broadway



Svilen Gospodinov

Foreword: *José Valim*
Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

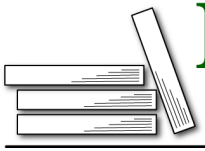
Concurrent Data Processing in Elixir

Fast, Resilient Applications with OTP, GenState, Flow, and Broadway

Svilen Gospodinov

The Pragmatic Bookshelf

Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Series Editor: Bruce A. Tate
Development Editor: Jacquelyn Carter
Copy Editor: Karen Galle
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

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

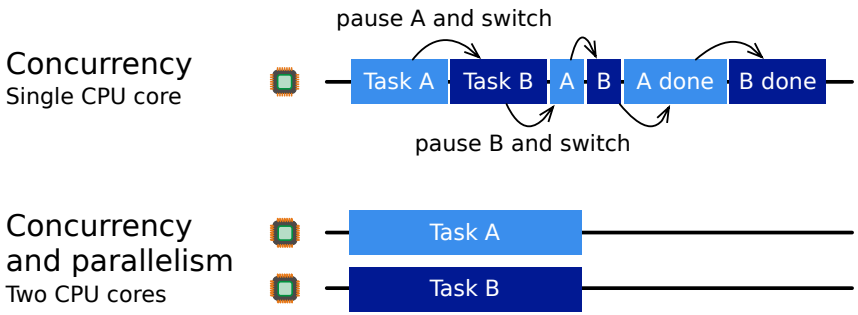
ISBN-13: 978-1-68050-819-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2021

CHAPTER 1

Easy Concurrency with the Task Module

Since the dawn of the computer industry, hardware manufacturers and computer scientists have tried to make computers faster at running programs. At first, multithreading was the only way to achieve concurrency, which is the ability to run two or more programming tasks and switch between them to collect the results. This is how computers appeared to be doing many things at once, when in fact they were simply multitasking.

Multi-core CPUs changed that. They brought parallelism and allowed tasks to run at the same time, independently, which significantly increased systems' performance. Multiprocessor architectures followed, enabling even greater concurrency and parallelism by supporting two or more CPUs on a single machine. The figure below shows a simple comparison between concurrency on a single-core CPU vs. a dual-core CPU. The latter also enables parallelism:



Of course, cutting-edge hardware always comes with a high price tag. But with the advent of cloud computing, things changed once again. Nowadays you can run code on cloud services using virtual machines with dozens of CPU cores, without the need to buy and maintain any physical hardware.

All these advancements are important to us as software engineers. We want to write software that performs well and runs quickly. After all, no one likes loading screens and waiting for the computer to finish. However, running code on a multi-core processor system does not automatically make it efficient. In order to take full advantage of the computer resources available to us, we need to write software with concurrency and parallelism in mind. Thankfully, modern programming languages try to help us as much as possible, and Elixir is no exception. In fact, thanks to Erlang, the Erlang Virtual Machine (BEAM), and the Open Telecom Platform (OTP), Elixir is a superb choice for building concurrent applications and processing data as you'll soon see in this and upcoming chapters.

In this book we're going to cover the most popular tools for performing concurrent work using Elixir. You will learn about the pros and cons of each one and see how they work in practice. Some of them, like the `Task` module and `GenServer`, come with Elixir. The others—`GenStage`, `Flow`, and `Broadway`—are available as stand-alone libraries on the `Hex.pm` package registry. Knowing how to utilize each of these tools will help you leverage concurrency in the most effective way and solve even the most challenging problems. Along the way, you will also learn how to build fault-tolerant applications, recover from failures, use back-pressure to deal with limited system resources, and many more useful techniques.

First, we are going to look at the `Task` module, which is part of the Elixir standard library. It has a powerful set of features that will help you run code concurrently. You are also going to see how to handle errors and prevent the application from crashing when a concurrent task crashes. The chapter provides a foundation on which the following chapters will be built upon, so let's get started!

Introducing the Task Module

To run code concurrently in Elixir, you have to start a process and execute your code within that process. You may also need to retrieve the result and use it for something else. Elixir provides a low-level function and a macro for doing this—`spawn/1` and `receive`. However, using them could be tricky in practice, and you will likely end up with a lot of repetitive code.

Elixir also ships with a module called `Task`, which significantly simplifies starting concurrent processes. It provides an abstraction for running code concurrently, retrieving results, handling errors and starting a series of processes. It packs a lot of features and has a concise API, so there is rarely (if ever) need to use the more primitive `spawn/1` and `receive`.

In this chapter, we are going to cover everything that the Task module has to offer. You will learn how to start tasks, and different ways to retrieve results. You will tackle processing large lists of data. We will talk about handling failure and explain how process linking works in Elixir. You will then see how to use one of the built-in Supervisor modules for isolating process crashes, and finally, discuss Elixir's approach to error handling.

Before we dive in, let's create an Elixir project to work on first and get familiar with some of the development tools we're going to use throughout this and the following chapters.

What Is an Elixir Process?

Processes in Elixir are Erlang processes, since Elixir runs on the Erlang Virtual Machine. Unlike operating system processes, they are very lightweight in terms of memory usage and quick to start. The Erlang VM knows how to run them concurrently and in parallel (when a multi-core CPU is present). As a result, by using processes, you get concurrency and parallelism for free.

Creating Our Playground

We are going to create an application called sender and pretend that we are sending emails to real email addresses. We are going to use the Task module later to develop some of its functionality.

First, let's use the mix command-line tool to scaffold our new project:

```
$ mix new sender --sup
```

This creates a sender directory with a bunch of files and folders inside. Notice that we also used the --sup argument, which will create an application with a *supervision tree*. You will learn about supervision trees later in this chapter.

Next, change your current directory to sender with cd sender and run iex -S mix. You should see some Erlang version information and the following message:

```
Interactive Elixir (1.11.0) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

We're now running the *Interactive Elixir* shell, also known as IEx. We are going to use it to test code frequently throughout the book. Most of the time, when we make a code change using our text editor, we can call the special recompile/0 function available in IEx, and the code will reload:

```
iex(1)> recompile()
:noop
```

We haven't actually added or changed any code yet, so the function returned just :noop for *no operation*.

In some cases, you may need to restart the IEx shell entirely, for example, when making fundamental application changes in the application.ex file. You can restart IEx by pressing Ctrl-C twice to quit and then running the iex -S mix command again.

To keep our project and examples simple, we're not actually going to send real emails. However, we still need some business logic for our experiments. We can use the Process.sleep/1 function to pretend we're sending an email, which is normally a slow operation and can take a few seconds to complete. When called with an integer, Process.sleep/1 stops the current process for the given amount of time in milliseconds. This is very handy, because you can use it to simulate code that takes a long time to complete. You can also use it to test various edge cases, as you will see later. Of course, in real world production applications, you will replace this with your actual business logic. But for now, let's pretend that we're doing some very intensive work.

Let's open sender.ex and add the following:

sender/lib/sender.ex

```
def send_email(email) do
  Process.sleep(3000)
  IO.puts("Email to #{email} sent")
  {:ok, "email_sent"}
end
```

Calling this function will pause execution for three seconds and print a message, which will be useful to debugging. It also returns a tuple {:ok, "email_sent"} to indicate that the email was successfully sent.

Now that everything is set up, we're ready to start. I suggest you keep one terminal session with IEx open and your favorite text editor next to it, so you can make and run changes as we go.