Funktionale Programmierung – Nebenläufigkeit & Parallelisierung

Jan-Philipp Willem, *1314162, Fakultät für Informatik, Hochschule Mannheim*

Zusammenfassung—Durch die immer weiter fortschreitende Vernetzung und die damit verbundene Fokussierung auf Big-Data, steigen auch die Erwartungen der User an produktiv genutzten Systemen. Um diesen Anforderungen gerecht zu werden, wird es immer wichtiger, effizient und nebenläufig programmieren zu können. Die bisher eingesetzten Lösungen haben durch den Einsatz imperativer Vorgehensweisen hohe Fehleranfälligkeit und Schwächen bei der Entwicklung von Multi-Tasking-Anwendungen bewiesen. Diese Arbeit hat sich das Ziel gesetzt, einige Konzepte der Nebenläufigkeit und Parallelisierung in der Funktionalen Programmierung an Hand der Sprache Elixir zu demonstrieren.

Abstract—The increasing Interconnection and the related Big-Data play an integrative part in the advancing expectations in productive-used systems by its users. To fulfill these requirements, the need to program efficiently and concurrent will be even more important. When building Multi-Tasking-Systems, previously used solutions were highly error-prone and had definitive weaknesses, due to the use of imperative strategies. This text strives to demonstrate some concepts of Concurrency and Parallelism with Functional-Programming in context of the language Elixir.

1 Einleitung

> The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. [...] Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. (G. E. Moore, 1965)

Seit Gordon Moore im April 1965 seine Beobachtung als einen Artikel in der Zeitschrift "Electronics" veröffentlichte, ist viel Zeit vergangen. Die auch als "Moore's Law" bekannte These beschreibt, dass sich jedes Jahr in Relation zu minimalen Komponentenkosten, die Integrationsdichte in Integrierten Schaltkreisen (englisch: integrated circuits, ICs) verdoppelt. [3]

In Abbildung 1 ist eine Relation zwischen CPU-Produkt-Veröffentlichung und deren jeweilige Transistoren-Anzahl zu betrachten. Man sieht, dass sich die Komplexität der ICs verdoppelt hat. Deswegen wird vermutet, dass sich die Moore's Law bewahrheitet hat. Jedoch scheint dieses Verhalten bald ein Ende zu haben. So hat Intel kürzlich mit dem Veröffenlichen der neuen Prozessorgeneration *Kaby Lake*, einen Prozessor auf den Markt gebracht, welcher eine exakt gleiche Performance wie sein Vorgänger *Skylake* aufweist.

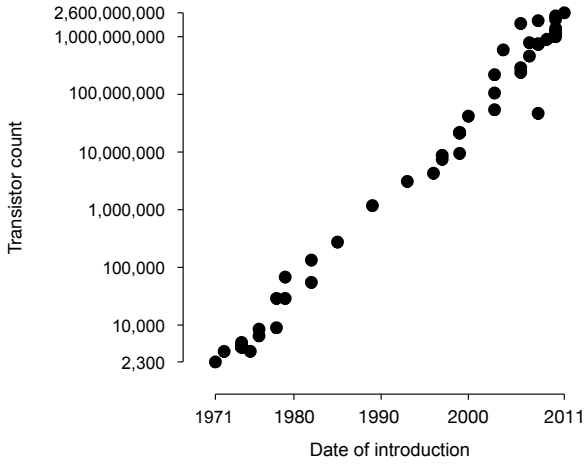

Abbildung 1. Transistor counts 1971-2011 & Moore's Law

1.1 Parallelisierung

Der deutsche Begriff *parallel* besitzt die Synonyme *nebeneinander* und *nebenläufig*. Dies trifft jedoch im Kontext der Informatik nicht zu. Die Parallelisierung hat zwar Gemeinsamkeiten mit der Nebenläufigkeit, allerdings auch Unterschiede.

Man spricht auch von *Multi-Processing-Systemen*, welche einen Geschwindigkeitsvorteil durch gleichzeitiges Ausführen von *Anweisungen*, gegenüber eines sequenziellen Systems erzielen. [1] Nach einer Aufteilung eines Problems in Teilprobleme, werden diese auf verfügbaren Prozessor-Kernen *gleichzeitig* gelöst.

1.2 Nebenläufigkeit

Die im Englischen als "Concurrency" bezeichnete Nebenläufigkeit, beschäftigt sich dahingegen um das Ausführen von mehreren *Aufgaben* zur gleichen Zeit. [1]

Der Fokus liegt hierin an der Betrachtung der Architektur hinter dem System, weniger auf deren internen Aufbau. Die Einzelnen Komponenten selbst, müssen nicht zwingend eine Parallelisierung durchführen. Durch eine starke und vor allem sinnvolle Kapselung innerhalb dieser *Multi-Tasking-Systeme* kann jedoch in den meisten Fällen eine Parallelisierung auf trivialer Weise vorgenommen werden. Teilen

sich die einzelnen Komponenten des Systems referenzierten Zustand gegenseitig, so sollte auf Unveränderlichkeit der enthaltenen Daten geachtet werden.

1.3 „Concurrency is not Parallelism“

Rob Pike, ein Entwickler der Go-Lang bei Google hat die Unterschiede von Nebenläufigkeit und Parallelisierung in einem Talk auf der “Heroku’s Waza conference“ im Januar 2012 sehr interessant zusammengefasst..

> - Concurrency is about dealing with lots of things at once.
> - Parallelism is about doing lots of things at once.
> - Concurrency is about structure, parallelism is about execution.
> (Rob Pike, 2012)

2 Funktionales Paradigma

2.1 Unveränderlichkeit

```
a = 3        a = 3
a += 2       a' = add(a, 2)
a -> 5       a' -> 5
```

Listing 1. Mutability vs. Immutability

Viele Fehler traten in der Vergangenheit durch die Veränderung von bestehendem Zustand auf, welcher typischerweise in verschiedenen Bereichen der Applikation referenziert wurde. Die Ergebnisse sind meist von unerwarteter Natur. Das Prinzip der Unveränderlichkeit beschreibt die Vorgehensweise, bei nötigen Zustandsänderungen stattdessen eine veränderte Kopie zurückzuliefern. Falls gewünscht, so kann man diese Zustandskopien wie Snapshots in einem Cache betrachten und sich beliebig schrittweise zwischen diesen bewegen.
Listing 1 zeigt eine typische Vorgehensweise beim Zuweisen von Variablen. Der Variable `a` wird jeweils die 3 zugewiesen. Imperativ würde man anschließend den Wert verändern um die gewünschte Funktionalität zu erreichen. Im Gegenzug dazu würde der funktionale Ansatz vorsehen, das Ergebnis der Addition einer neuen Variable zuzuweisen als den alten Zustand zu verändern.

2.2 Funktionen

Die Funktionale Programmierung fokussiert sich in erster Linie auf Funktions-Komposition, bei der komplexe Schachtelungen von Teilfunktionen miteinander vereint werden. Dabei wird jedoch sonst auf gebräuchliche Seiten-Effekte verzichtet. Man möchte so ein möglichst deterministisches Verhalten bewerkstelligen. Somit sollte eine Funktion auch immer nur die eine Sache tun, welche man beim Lesen der Funktionssignatur auch erwarten würde. Im Idealfall können die einzelnen Funktionen als reine Daten-Transformationen aufgefasst werden, welche die übergebenen Daten kopieren, transformieren und wieder zurückgeben. Dies wird in der Literatur auf verschiedene Arten beschrieben: “pure“ und “data-in, data-out“ sind Synonyme für einander. In der Informatik ist dieses Prinzip auch als Eingabe-Verarbeitung-Ausgabe (EVA) fest verankert.

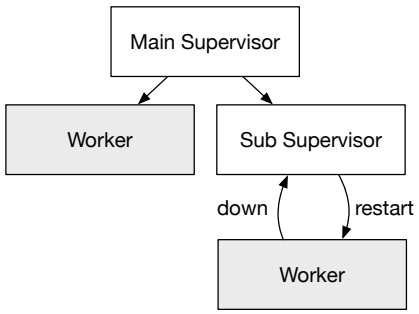

Abbildung 2. komplexe Supervision-Bäume

Funktionen können selbst auch als ein Datentyp definiert werden. Das sogenannte Konzept “functions as first-class citizens“, ermöglicht es Funktionen als Parameter anderer Funktionen zu übergeben. Die Ausführende Funktion kann jedoch selbst entscheiden, ob die übergebene Funktion schon ausgewertet werden soll. (lazyness)
Lamdas sind anonyme Funktionen, welche inline definiert werden und somit keinen Namen besitzen. Sie werden oft als Callback anderer Funktionen genutzt oder für die spätere Weiterverwendung einer Variable zugewiesen.

2.3 reine funktionale Sprachen

Es wird zwischen Sprachen unterscheiden, welche eine Unterstützung für die funktionale Programmierung bieten und solchen, welche ausschließlich auf deren Konzepte beruhen. Seiten-Effekte sind somit laut Definition gar nicht möglich auszuführen. Dies kann sehr hilfreich sein, da in der Regel auch solche Teile einer Programmiersprache genutzt werden, welche man eigentlich meiden sollte. Gerade bei der Arbeit in einem größeren Team, stellen sich die zusätzlichen Einschränkungen eher als eine Erleichterung heraus.
Viele rein funktionale Sprachen wie beispielsweise *Haskell* setzen zudem eine strikte Typisierung voraus. Im Gegensatz zu beispielsweise *Java* hat der Programmierer einen tatsächlichen Nutzen von der Typisierung, da ihm so der Compiler beim beseitigen von Fehlern besser helfen kann.

3 Elixir

Elixir [2] ist eine seit dem Jahre 2011 entwickelte Variante von Erlang. Es besteht eine dynamische Typisierung mit Typen-Interferenz jedoch weißt sie weiterhin Features einer reinen funktionalen Sprache auf. Es kann weitestgehend Seiten-Effekt-Frei programmiert werden. Elixir nutzt Erlangs Beam-VM um eine Fehler-Tolerante Umgebung zu bieten. Dazu wird mit einem Kompiler ein Erlang-Bytecode erstellt.
Erlang wurde von Ericsson schon im Jahre 1987 entwickelt, um die Vernetzung von Telefonie-Systemen zu vereinfachen. Grundlage sind sogenannte Akteure, welche aus leichtgewichtigen Erlang-Prozessen bestehen. Diese erzeugen und nutzen wiederum unzählige Prozesse um ihre Aufgaben zu lösen. Es sind je nach Anwendungsfall sowohl geteilte wie auch verteilte Speicher zwischen den Akteuren möglich. Im Rahmen des *Open-Telephony-Protocol (OTP)* ist es üblich komplexe Akteur-Bäume aufzubauen. (Abbildung 2) Ein Akteur kann so auch als ein Supervisor eines anderen Prozesses dienen. Die Fehlerbehandlung kann

damit erheblich vereinfacht werden. Es besteht die Philosophie "let it crash", da man bei einem Fehlerfall, den Prozess einfach neu starten kann. Es existieren viele Arten an Vorgehensweisen, welche Teile der Applikation vom neu starten betroffen sind. Wird das Konzept der sinnvollen Aufteilung in Akteure verfolgt, so erhält man granulare Teilsysteme, die sich nicht beeinflussen können.

3.1 OTP / Actor-Model

Das Actor-Modell stellt das Concurrency-Model in Elixir dar. Es wurde oft implementiert, jedoch trug Erlang den weitesten Nutzen daraus, da es weitestgehend auf dessen Konzepten basiert.
Die Grundidee besteht aus unabhängigen Akteuren, welche jeweils eine eigene Mailbox und einen eigenen Zustand besitzen. Die Akteure können untereinander Nachrichten austauschen um miteinander zu kommunizieren. Die Reihenfolge der Abarbeitung der Nachrichten ist nach deren Eintreffen geregelt. Je nach Anwendungsfall, kann es sich anbieten die Akteure als vollwertige eigene Services zu betrachten und sie mit einem eigenen Cache und oder Persistenz (DB) auszustatten.
Als Alternativen gibt es Libraries, mit denen man mit Akteuren in beliebigen Sprachen programmieren kann. So gibt es beispielsweise Implementierungen für Java/Scala (Akka), Python (Pykka), Akka.NET, C++ (CAF) und Ruby (Celluloid).

3.2 List-Processing: map

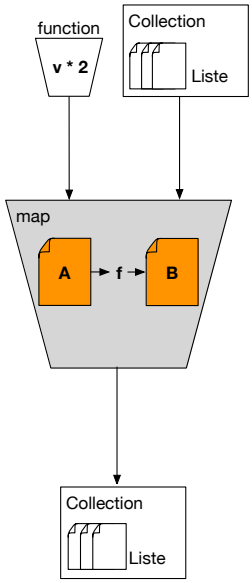

Abbildung 3. Die `map`-Funktion

Eines der wichtigsten Funktionen einer funktionalen Sprache, ist die `map`-Funktion. Sie ermöglicht eine schrittweise transformation einer Datenstruktur. Als Paramter werden in Elixir eine `transform`-Funktion und eine Collection übergeben. Beim Auswerten, wird über die Collection interiert, das jeweilige Element verändert und wieder einer neuen Collection hinzugefügt. Der Rückgabewert kann jedoch auch von einem anderen Typ sein.

```
iex> Enum.map [1, 2, 3], fn x -> x + 1 end
  [2, 3, 4]
iex> Enum.map [1, 2, 3], &(&1 * &1)
  [1, 4, 9]
iex> defmodule Math do
...> def multWithKey({k, v}), do: k * v
...> end
  ...
iex> list = Enum.with_index([1, 2, 3])
  [{1, 0}, {2, 1}, {3, 2}]
iex> Enum.map list, &Math.multWithKey/1
  [0, 2, 6]
```

Listing 2. Typische Einsatzmöglichkeiten von `map`

3.3 Streams

Mithilfe von Elixir Streams kann eine große (oder unendliche) Datenstruktur lazy verarbeitet werden. Dabei werden viele Funktionen aus dem `Enum`-Modul implementiert, so dass man ähnliche Verhalten abbilden kann. Mit function-composition wird die Collection nur einmal durchlaufen und dabei jeweilig transformiert. Sollten nicht alle Elemente gebraucht werden, so kann man mit `Enum.take` die Größe der Ergebnismenge bestimmt werden.
Das ganze Konzept stellt eine wichtige Optimierung unter Anderem bei der Parallelisierung dar. Im Detail gehen die Elixir-Streams und andere ähnliche Implementierungen in anderen Sprachen auf die Clojure Transducer/Reducer zurück.

```
iex> 1..10000
     |> Stream.map(&(&1 * &1))
     |> Stream.map(&(&1 + &1))
     |> Stream.map(&IO.inspect(&1))
     |> Enum.take(10)
  2
  8
  18
  ..
  200
  [2, 8, 18, 32, 50, 72, 98, 128, 162, 200]
```

Listing 3. Elixir-Streams

3.4 Processes

Im Gegensatz zu der fehleranfälligen Thread-Programmierung in anderen Sprachen kann in Elixir mit `spawn` direkt ein leichtgewichtiger Prozess gestartet werden. Der Rückgabewert ist dabei eine Process-Id. Mit `send` ist es möglich zwischen den Prozessen Nachrichten auszutauschen. Und mit einem `receive`-Block können die eingegangenen Nachrichten verarbeitet werden.

```
defmodule Parallel do
  def pmap(collection, fun) do
    me = self
    collection
    |> Enum.map(fn (elem) ->
         spawn fn ->
           (send me,
self, fun.(elem) )
         end
       end)
```

```
    |> Enum.map(fn (pid) ->
         receive do
           ^pid, result -> result
         end
       end)
  end
end

Parallel.pmap 1..1000, &(&1 * &1)
```

Listing 4. Prozesse erzeugen

3.5 Tasks

Trotz des sauberen Codes beim Erzeugen von Prozessen, ist diese Herangehensweise ebenso Fehleranfällig, da man sich um selbst über Supervisioning-Verhalten und beispielsweise Timeouts kümmern muss. In viele Fällen ist es deswegen sinnvoll, die von Elixir mitgebrachte OTP-Abstraktion der `Tasks` zu nutzen. Dadurch hat man den zusätzlichen Vorteil, dass man den Task in bestehende Supervision-Bäume integrieren kann. Gerade bei so einem häufigen Anwendungsfall eines Elixir-Prozesses wird einem einiges an Arbeit abgenommen und man muss sich jedoch trotzdem nicht mit dem zwar mächtigen aber teilweise auch komplexen OTP beschäftigen.
Die dabei erhaltene Lösung wirkt noch etwas sauberer zu lesen und es wurden schon einige Mögliche Fallstricke optimiert. So kann mit `Task.async` ein Task erzeugt werden, welcher die übergebene Funktion nicht-blockierend ausführt. Anschließend kann mit `Task.await` auf die Ergebnisse von Tasks gewartet werden.

```
defmodule Parallel do
  def pmap(collection, func) do
    collection
    |> Enum.map(&(Task.async(fn -> func.(&1))))
    |> Enum.map(&Task.await/1)
  end
end

Parallel.pmap 1..1000, &(&1 * &1)
```

Listing 5. Tasks erzeugen

4 Fazit

Wenige Sprachen und deren Tool-Chains bieten dem Programmierer wirkliche Erleichterungen bei der Entwicklung von nebenläufigen Systemen. Oft fehlen weiterhin Konzepte die eine reine Parallelisierung ermöglichen. Die Thread-Programmierung ist dabei meist die einzigste Möglichkeit.
Je nach Sprache erinnert die Funktionale Programmierung eher an eine Zusammenstellung von Verhalten. Deren Art und Abfolge kann genau und deklarativ bestimmt werden. So beschäftigt sich *Higher-Order-Programing* genau mit dieser Tatsache. Der kompromisslose Einsatz von Unveränderlichkeit der Daten, erleichtert weiterhin die Vorgehensweise ein nebenläufiges oder paralleles System zu entwickeln. Die typische Fehleranfälligkeit (Dead-Locks, Race-Conditions,..) bei der Synchronisierung einzelner Teilsysteme, wie es bei Threads mit geteiltem Zustand der Fall war, werden einfach umgangen.
Bei der Wahl der Sprache Elixir, erhält man eine direkte Unterstützung für eine Verteilung der Anwendung auf mehrere Knoten. Dies wird mit dem von Erlang etablierten OTP mithilfe der Beam-VM erzielt. Die Programmierung findet in einer Umgebung ohne Seiten-Effekten oder sich verändernden referenzierten Daten, ohne auf moderne Syntax oder Sprache-Features wie Meta-Programming verzichten zu müssen.
Im Gegensatz zu den Vorteilen die geboten werden, steht die Tatsache, dass die Lernkurve um die funktionalen Konzepte zu verstehen, sehr steil sein kann. Man muss die bisherig genutzten Methoden teilweise komplett aufgeben um anstelle davon neues zu lernen.

Literatur

[1] Paul Butcher, *Seven Concurrency Models in Seven Weeks: When Threads Unravel*, 1.Auflage (18. Juli 2014), O'Reilly UK Ltd.
[2] Dave Thomas, *Programming Elixir*, 1.Auflage (19. Oktober 2014), Pragmatic Programmers
[3] G. E. Moore, *Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.,* in IEEE Solid-State Circuits Society Newsletter, vol. 11, no. 5, pp. 33-35, Sept. 2006.