Elixir in parallel

Nico Westerbeck

Contents

1 Introduction

In this paper we evaluate elixir as a parallel programming language, exploring the basic idea of the actor model and how it is used in elixir. Furthermore we compare the capabilities of the language as a parallel programming-language and compare it to C-family languages. We find that, though offering several advantages, Elixir lacks some features for being usable as a parallel programming language such as C++ in the general case but still worth looking at in special cases.

1.1 Terms and Definitions

For understanding of the paper we require basic knowledge of parallel programming models. Furthermore we distinguish the terms concurrency and parallelism based on the definition from HaskellWiki [1]

> The term Parallelism refers to techniques to make programs faster by performing several computations in parallel. This requires hardware with multiple processing units. In many cases the sub-computations are of the same structure, but this is not necessary. Graphic computations on a GPU are parallelism. Key problem of parallelism is

to reduce data dependencies in order to be able to perform computations on independent computation units with minimal communication between them. To this end it can be even an advantage to do the same computation twice on different units.

The term Concurrency refers to techniques that make program more usable. Concurrency can be implemented and is used a lot on single processing units, nonetheless it may benefit from multiple processing units with respect to speed. If an operating system is called a multi-tasking operating system, this is a synonym for supporting concurrency. If you can load multiple documents simultaneously in the tabs of your browser and you can still open menus and perform more actions, this is concurrency.

If you run distributed-net computations in the background while working with interactive applications in the foreground, that is concurrency. On the other hand dividing a task into packets that can be computed via distributed-net clients, this is parallelism.

2 Actor Model

The actor model after [2] is a model of parallel programming which builds upon the basic building block of actors.

2.1 Explanation

An actor is an independent unit in that model and each actors has the following three possibilities to act

- An actor can spawn a finite number of new actors
- An actor can send a finite number of messages to other actors it knows
- Upon the arrival of a message an actor can perform computations changing its behavior for future message arrivals ("state")

To launch the system, usually the starting setup is one existing actor which receives a startup message from the system. Effectively, these three constraints are enough to set up a system which enables complex parallel computations. Notice some more things:

- Actors do not share state, in fact the only way for actors to interact with each other are messages
- There are no assumptions made over the compution time, resembling experiences made in exploring parallel programming in other languages
- There is no specification about the transporting channel, messages can arrive in any order.

- An actor has to know the recipient of a message. For that, actors are identified by an address, which is transferrable through channels to inform actors about the existence of other actors.
- There are no available synchronization primitives, all synchronization needs to happen via message passing

This system has proven a powerful, lock-free way to model parallel applications. By extending that model to allow for message loss and crashing actors, this would closely resemble the real world situation of independent computation nodes in a distributed cluster. To obtain a better understanding we will shortly explain the model in an example

2.2 Fork-Join in Actor-language

The Fork-Join model is a commonly known pattern in parallel computing, we will try to "translate" that into an actor model. In the usual Fork-Join paradigm, an execution-flow branches up into several, concurrent execution flows, which will rejoin to one sequential one at a predefined point. For this, upon the fork, either the current state of the application is copied into every execution unit(process) or units work on a common, shared state(threads) and are rejoined by the spawning entity after their computation. For this, each computation unit terminates itself after completion of its work.

In an actor model, we will have a problem with copying or sharing the state to the units, as this is not explicitely possible in this model. Instead, the data relevant for each piece of work needs to be explicitely sent to an actor. Also, we do not have to spawn or terminate the actors at the point of fork or join, they can live before and after that, waiting for a message containing work. And at last, if synchronization is needed between the actors, this needs to happen based on messages too as no synchronization primitives are available.

3 Elixir

Elixir [3] is a functional programming language built upon the Erlang VM. It appeared in 2011 and currently is in its version 1.4.0. The aim of Elixir is to combine the advantages of functional programming upon the battle-tested Erlang-VM while allowing for easier to write syntax and a modern programming workflow. Erlang [4] itself is a programming language that has been around since 1986. It is a language developed by telecommunication providers to allow for a scalable, maintainable and fault-tolerant software used in telecommunications. Since those traits have become increasingly demanded in other fields of computing such as web development, the Elixir developers tried to build upon those features.

Some Key-features that are relatively unique to the Beam-VM are

- **Processes** Erlang created an own userspace-scheduler for Beam-VM processes which allows for spawning millions of processes in contrast to the overhead-rich operating system processes.
- **Hot code reloading** The language was designed to be maintainable without downtime. Thus, it is possible to insert updated code during runtime without restarting the application.
- **OTP-Framework** A feature-rich framework which at first was intended to provide telecommunication-related functionalities but now has leveraged to be some kind of standart library including functionalities for diverse use-cases.
- **OTP-Supervision tree** A key feature of OTP is the supervision tree, which has become a pattern to design Erlang applications. An application is structured in a tree-like fashion of supervisors and workers, where the task of each supervisor is to restart failing branches.

Because of its key features for this article we will talk more in detail about Processes and the OTP Supervision tree. Notice that, though these features were implemented for Erlang, they are also available in Elixir. In fact, Elixir is able to include any Erlang-code and thus provides a superset of features over Erlang. Also it holds the design principles and workflow of a functional programming language, which incorporates the idea of "everything is a function" in contrast to OOP-based "everything is an object" approaches. The design philosophy is to, instead of thinking in objects and functionalities to think in dataflows and transformations.

As most of the functional programming languages, Elixir has immutable data but unlike others it allows for side-effects to happen inside a function. Thus it is not suited for mathematically proving code or optimizations based on the no-side-effect assumption. However it is more comfortable to write code, as input/output or database communication can happen in place and does not need to be wrapped into I/O monads. It does support some lazy evaluation through the Stream datastructure but lazyness is not hardwired into the language such as in the case of some other functional programming languages.

3.1 Processes

The Beam-VM implements it's own scheduled concurrency through so-called processes. Though they are called the same as the OS-level processes, they are different in quite some aspects. The first is the low overhead per process, which allows programmers to use a high number of processes for concurrent execution, ranging in the millions on a single machine. It is not possible to share memory between processes, in fact there is only one recommended way to pass data to a process, which are messages. Elixir also offers a dictionary per process, but advises to use it with care or not at all.

```
parent = self()
spawn fn -> send(parent, {:hello, self()}) end
receive do
  {:hello, pid} -> "Got hello from #{inspect pid}"
end

"Got hello from #PID<0.48.0>"
```

The above listing spawns a process which sends back a hello message to the parent process and then terminates. `spawn` creates a new process which in this case launches an anonymous function, `send` and `receive` are methods to send and receive messages. To send a message the receiving process needs to be identified by its pid, which is obtained in the first line by `self`. This syntax is enough to create a distributed instance of a hello world, as it is transparent to the programmer wether processes are launched on the same machine or on another machine in the cluster.

All in all processes are maybe one of the most powerful tools of Erlang/Elixir and though being the only building block for concurrency, allowing for large scale computing with very simple syntax.

3.2 Supervision tree

Because of its relevance for distributed systems we also want to shortly present the idea behind the supervision tree of erlang. Supervisors like supervisord [5] are an essential part of todays production system, as they allow for user-interaction free recovery in case of application failure. However, the idea of supervision trees is not widely adapted yet. The basic concept behind that is, that whenever a part of an application spawns new processes, it has the responsibility of monitoring them and respawning them in case of failure. Those parts however can spawn new parts themselves, which results in a supervision tree. Further more, each supervisor can decide if it wants to only respawn a single failed child or, upon one failer, respawn a whole group of them.

This idea results in the design of a composed application with a high fault-tolerance, allowing individual components to fail and restart without affecting the whole application. In general, a fine-grained supervision and composition of the application has proven a viable solution for reaching high fault-tolerance and scalability, such as in the design-paradigm of microservices [6] outside of Erlang/Elixir. These ideas however existed in Elixir for several years already and have been developed and tested over the course of the existence of the Beam VM.

4 Actor-model parallelism in Elixir

It is quite notable that Elixir's notion of processes is very close to the notion of actors in the actor model, with two subtle differences. As mentioned,

erlang/Elixir processes have a process dictionary though taking use of it is disregarded, and upon spawn of a process automatic variable capturing is executed. The first does invalidate the actor-model, the second can be simulated by sending a message to each newly spawned actor and parsing the captured variables from it. This is rather a convenience feature and does not necessarily need to be used. Because of this similarity, most parallel applications written in Elixir follow the actor model. To illustrate that, we will shorty present a parallel "for" in Elixir. As for is not really a functional programming construct, we will instead use map which basically does the same by applying a function to all elements of a dataset.

4.1 Example

```
parent = self()

1..1000
  |> Enum.map(fn(x) -> {x, (spawn fn ->
       receive do
         x -> send parent, :math.pow(x, :math.pi)
       end
     end)} end)
  |> Enum.map(fn({x, pid}) -> {x, (send pid, x)} end)
  |> Enum.map(_) -> (receive do x -> x end) end)
  |> IO.inspect
```

This listing will print out the numbers between 1 and 1000 raised to the power of pi. This code is a rather sloppy example for the beginning, as 9 lines of code for that are quite a lot and the code is rather but we will provide a sleaker example later, this is for understanding the concepts. We will not give a introduction into the Elixir syntax here as it would exceed the scope of the paper and is not necessary for explaining the basic concepts of parallelism in elixir, for further information please consult their website [3].

We launch 3 map calls on a dataset of 1000 elements, which is transformed by the map call.

- 1000 processes are launched, which each wait for a received piece of data and then raise that piece of data to the power of pi and send it back to their parent
- The second function accepts a number and a process handle and sends that number to the process handle
- THe last map does not take any data and just performs 1000 receive calls to receive data from any source, to pass it on to the output call

Each of those 1000 spawned processes and additionally the parent process are implementing the actor model requirements. This way we implicitely implemented that calculation for a parallel, lockfree execution on a cluster. We did

not perform any scaling tests on this example code as we expect transferring the data to be the bottleneck in this application, but we believe to actually see a scaling on a more complex application.

As elixir is focussing on readable syntax, they provide a convenience wrapper named `Task` which can be used to increase the readability of the code above and even provide the convenience of in-order output.

```
1..1000
  |> Enum.map(&(Task.async(fn -> :math.pow(&1, :math.pi) end)))
  |> Enum.map(&Task.await(&1))
  |> IO.inspect
```

4.2 Analysis

In general, actor-model based parallelism enforces a parallel-optimized design of an application, as the actors and their interface needs to be clear during the design period. This has the advantage that those language constructs push the designers to applying these design-constraints but has the disadvantage that it often needs a redesign to transfer a sequential code into a parallel one. Also, Elixir offers less choices in parallel computing compared with other languages, so adjusting the framework/setup to match the application is difficult to not possible.

In the end especially communication-intensive applications such as mathematical computations will be hard to get to performance in Elixir though as the explicit channel communication is unavoidable and not necessary the best solution if you handle large data volumes in a shared memory context. Thus we do not expect Elixir to take a bigger portion in the HPC market in the near future, as those applications are the major use-case in there. However, Elixir gains increasing popularity in web-development and soft-realtime systems as the high fault-tolerance, low latency and the communication-optimized framework of the language are of great benefit there.

5 Comparison to C-family langugages

The C-family, here represented by C++ has become a preferred language for parallel applications and a wide set of frameworks provides routines for many use-cases. Compiler extensions like OpenMP[7] or communication frameworks like MPI [8] have made parallel programming easier in these languages. In Elixir, some basic concurrency-frameworks are available and the included OTP is very superior in terms of handling many concurrent and even failure-prone execution to C++-frameworks, but for parallelism there are not many available frameworks.

Furthermore C-like languages such as CUDA or OpenCL allow for including of accelerator devices such as GPUs or Xeon Phis, which is currently rather difficult as the only way to enable for that is to get the BeamVM running on

the accelerator device and there has been very little evidence of this happening or happening efficiently.

C++ has an explicit memory-management and thus allows the programmer to control where and how the data is stored. This is not possible in Elixir, the Beam-VM takes care of memory-management and garbage-collection. Though garbage-collection has proven advantageous on implementing concurrent datastructures, the explicit memory-management is advantageous if the memory usage is known prior to the execution as the programmer can decide on how to structure allocations or deletes and can optimize those things. In a soft-realtime system or a reactive system, this normally is not the issue of concern so Elixir abstracts these concerns away and lets the runtime environment decide on those, but in compute-intensive applications this is a concern, which is a clear drawback.

At next Elixir provides a userspace-scheduler that is superior to OS-level schedulers in terms of maximum number of units. Though this should theoretically improve performance, the reality of parallel programming seldomly shows the need of allocating more than N*4 threads, N being the number of processor cores available, as performance-gain usually drops around this number. This is by far not the level where the scheduling advantage of Elixir kicks in, these number of thread is usually only needed when waiting for different IO-events like in the case of a communication hub or a webserver.

Though the design of Elixir enforces lockfree codedesign, the same is also possible but not enforced in C++. Thus this can not really be considered an advantage in this case, as it is in the responsibility of the programmer to choose the best option and in some cases, the choice might conciously be a synchronization directive over lockfree code.

6 Conclusion

Elixir offers a great way to access the powerful Beam-VM through a sleek and elegant syntax with and enforces a concurrent design by its functional nature. However compared to the C-Family, it lacks some features that would make it the preferred choice for tackling massively parallel applications. Elixir's strengths lay more in concurrency than parallelism, which makes it a great language for reactive systems like a web application but not so well suited for the traditional parallel applications such as Matrix Multiplication or other mathematical constructs. For those we would rather recommend the C-family or, if you want to still utilize the benefits of functional programming, Haskell.

References

[1] Parallelism vs concurrency after haskellwiki. `https://wiki.haskell.org/Parallelism_vs._Concurrency`. Accessed: 2017-05-16.

[2] Carl Hewitt. Actor model for discretionary, adaptive concurrency. *CoRR*, abs/1008.1459, 2010.

[3] Elixir homepage. http://elixir-lang.org. Accessed: 2017-05-04.

[4] Erlang homepage. https://www.erlang.org/. Accessed: 2017-05-04.

[5] Supervisord homepage. http://supervisord.org/. Accessed: 2017-05-05.

[6] Martin fowler on microservices. https://martinfowler.com/articles/microservices.html. Accessed: 2017-05-05.

[7] Openmp website. http://www.openmp.org/. Accessed: 2017-05-16.

[8] Message passing interface, openmpi implementation. https://www.open-mpi.org/. Accessed: 2017-05-16.