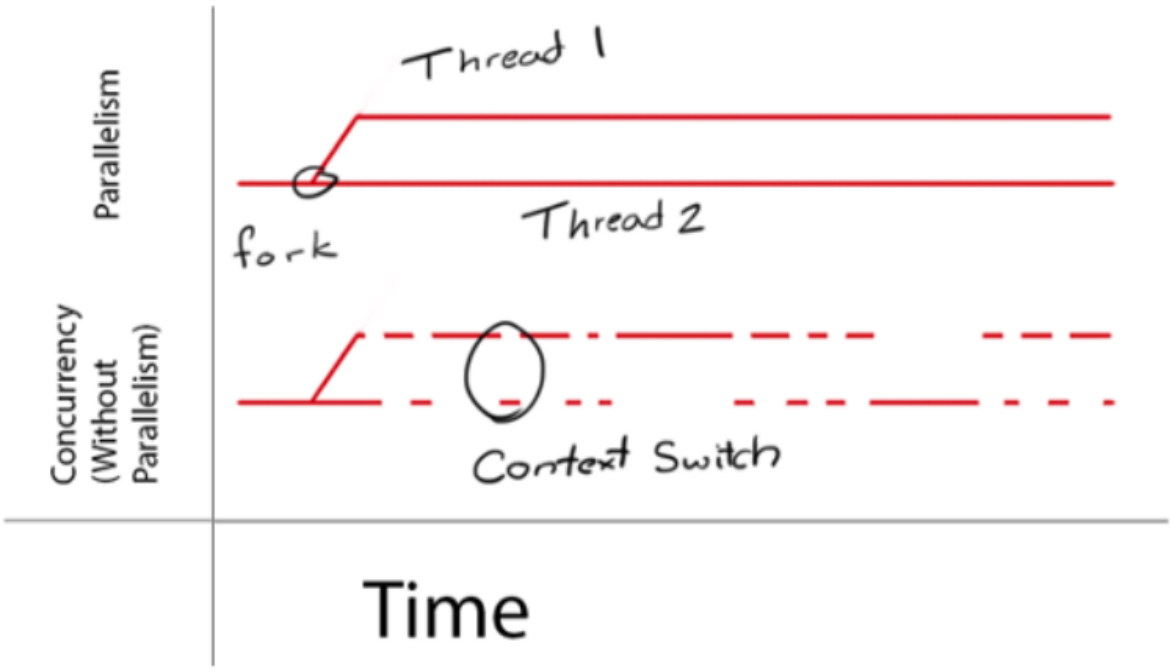
Compare Go Parallel/Concurrent Programming with Erlang

Parallel Programming vs Concurrent Programming

> Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.
>
> — Rob Pike, Heroku's Waza conference

Parallel Programming and Concurrent Programming are two concepts of Programming Languages that seem to refer to the same concept. They are used whenever multitasking is needed in coding. The problem with this is that they are not the same thing. Concurrent means that there are multiple tasks that are executed in overlapping time periods whereas parallel means multiple tasks are executed the same time. Concurrency is used in context of the way programs can run. Parallelism requires at least a dual-core processor or else the program can't be executed truly parallel. [Gupt]



Concurrency and Parallel Programming [Gup19]

An example for this could be the production of vehicles. A car consists of different parts e.g. the engine, the tires and the body. For the car to be assembled you first need to produce all separate parts. After that the car can be assembled. The task is the production of the vehicle while the subtasks are the assembly and the production of the parts. The production of the parts could be executed parallel. That means that the body, the engine and the tires are all produced at the same time. The whole system is concurrent as the production of the individual parts and the assembly can't be executed at the same time, but still are part of the same task.

Concurrent Programming in Golang

> Don’t communicate by sharing memory; share memory by communicating.
>
> — Golang Authors, Effective Go [eg]

The quote summarizes the way that concurrency in Golang works. Concurrency is achieved by using goroutines and channels. Goroutines are often referred to as 'lightweight threads'. That means that they are like threads in the way that threads are a series of instructions. The same thing goes for goroutines. In both cases the goal is to execute those instructions one after another. The main difference that makes Goroutines lightweight and threads not so much, is the approach to memory. Goroutines only get around 2 KB of stack size when they are first initialized. This stack size is increased in case it is needed. 'Normal' threads however get around 1 MB of stack size. This size varies depending on the Operating System (OS) that is used. And threads are running in the OS whereas Goroutines run in the go runtime. Go runtime starts an OS Thread which handles all the goroutines. They are all started in the same OS thread and share no memory. Because of the small stack size of goroutines they are very cheap and because of their flexible size they are also very scalable. [Kha18]

With no shared memory the communication of goroutines depends on the direct communication. For these so-called channels are used. They are like pipes which transfer data from one function to another. A channel can only send or receive data of a specific type. That type is chosen when the channel is initialized. It can be of any type that can be understood by Golang. [cht] There are two different categories into which the channels can fall: unbuffered and buffered channels. The main difference between these two is that a buffered channel has a capacity to store some messages that are passed through it whereas the unbuffered one doesn’t have this. [unb] Because of that the channel types behave very differently. Unbuffered channels require that both the sender and the receiver are ready to send or receive a message. If one of those requirements is not true, then the channel will make the goroutine wait. That way the resource which is exchanged can be synchronized. Buffered channels will only behave like this if the buffer is full. In that case the sending functions is blocked until some of the space is free again. The synchronization is only guaranteed with the unbuffered channels because in that case the sent is blocked until the receiver gets the data. [Ken14]

As already explained earlier these features only make it possible to use concurrency in programming. Golang has a module which is called "runtime". With that it is possible to let the go runtime environment use more than one core to execute programs parallel. For this the developer can import the runtime module and use the function "GOMAXPROCS(n)" with n being the number of cores that the code should use. [run]

```
import "runtime" ①

func main() {
  runtime.GOMAXPROCS(4) ②
}
```

① Library import

② Function call

Concurrent Programming in Erlang

Erlang is a programming language that was created for the telecommunication company Ericsson. It was designed with concurrency in mind as this is a concept that is very important in telecommunication. [erf] The way that concurrency is implemented in Erlang is called the Actors Model. This means that programs are divided into separate parts and those parts communicate over direct messages. The different parts are the actors and are separate processes. The actors can act independently from each other. [Mil09] In Erlang the name for the thread like concept is process. The processes are spawned by executing a spawn function. This function gets a couple of parameters like the module name that executes the process and the function name that is spawned as a process. It also returns a unique ID called the pid. This stands for process identifier and can also be found in OS. There is also the possibility to register a specific name to a corresponding pid. This is necessary for two independently started processes that need to communicate with each other. For sending messages one of those identifiers must be known to the sending process. [con]

Like the goroutines in Golang, Erlangs processes don't share the same memory. That is the reason why processes in Erlang communicate with each other over direct messaging. The messages can be any valid Erlang term e.g. tuples, atoms or strings. Once a process sends the message it continues without waiting for an answer from the receiver. This means that the messaging is asynchronous. Similar to Golang's buffered channels Erlangs processes possess a queue where messages are stored and can be handled one after another. Every message includes a specific pattern which can be used by the receiving process to compare with given patterns. If a matching pattern is found, the process will execute the commands that correspond to the pattern and the message will then be deleted from the queue. If a pattern isn't found, then the next message in the buffer is tried against the given patterns. This continues until every message in the queue was tried. The process is than blocked and waits for a new message to arrive. With the arrival of this message the matching process begins again. The messages that can't be matched stay in the queue and get checked again every time a new message arrives. [con]

Comparison of the theoretical background

In direct comparison one can see that Golang and Erlang deal similarly with concurrency as a concept of programming. They both use processes or lightweight threads without shared memory and let them communicate via direct messaging. The differences are seen if one looks closely at the way the messages are passed. In Golang the goroutines don't get a specific ID and all the communication works with channels, whereas Erlang uses "real" direct communication. This just means that every process must know the pid of the process it wants to contact. This is not possible in Golang Code. Here the thing that is needed by the processes is the channel. Those are two very different concepts of transport. With the usage of channels more goroutines can keep in touch with each other. That way several goroutines can be started and get or send messages from or to the other goroutines without even knowing that they exist. At the same time, it is not possible to know from which exact goroutine the message was received, but this isn't needed. With pids the processes need to know the other processes or else they can't communicate with each other. That way the same two processes will always speak to one another.

Another difference is the type of messages that can be transferred. As already explained a channel only passes messages of a specific type. So, if different types of messages must be sent between goroutines, there has to be more than one channel for this to succeed. To be specific this means that in Golang to be able to send a string and a float64 between two goroutines they both need access to two channels, one for strings and the other for float64. In Erlang a process only needs the pid of the other process as messages can be of any type. Another difference is the handling of more than one message. As mentioned earlier Golang knows two types of channels. Only buffered channels can handle more than one message at a time. Erlangs processes each have a message queue. [con] That way it is possible to send a message even though a prior message was faulty and couldn't get matched or wasn't yet processed. As already explained in Golang a channel needs the affirmation that the message was received. This means that the communication in Golang code is synchronous. The sender wants a reply from the receiver. Erlangs message passing is asynchronous. If a process sends a message, it will then continue with other instructions. It is asynchronous. That is why Erlangs processes need a message queue whereas Golang's don't need one.

But it shows that Golang has a feature that Erlang lacks on purpose. Golang is able to use mutual exclusion (mutex). Mutex is one way to tackle the problem of deadlocks in programs. Deadlocks are one of the main problems in concurrent programming. It occurs if e.g. two threads try to get the same two resources but because each of them already got one, the other thread can't reach the second resource it needs. That way both threads stay in a state where they try to get the second resource but don't let go of the first one. This way they block each other. By using mutex in code, the accessibility of a variable can be controlled. In Golang this is managed through a standard library that provides two methods called "Lock" and "Unlock". [syn] Erlang is not meant to use such things. There is not even the possibility to have variables that are reachable in every process. Erlang tries to work against such things by not sharing any variables. Even the messages that are sent are only copies of the values and not pointers to the address of a variable. There aren't even global variables in Erlang, and it is best practice to send the information that is needed through the parameters of a function. That is why there is no shared memory. The processes are meant to be individual and not connected. Still there is the possibility of deadlocks in Erlang. This happens e.g. if a receive block doesn't get any messages that it can match. That way it's possible that a process waits for an answer and stays in the receive block forever. [Chr11] In Golang the same thing can happen, but in this case the deadlock happens because of messages not being received. That is the case if there are more working, sending goroutines than there are waiting, receiving goroutines at a time. They can be found by the deadlock detector and send a panic. [Bla20]

Table 1. Comparison of Erlang and Go

	Golang	**Erlang**
Message Passing	direct, synchronous → locks	direct, asynchronous
Message Passing - Method	message passed through channels	message directly send to other process via Pid (Process Identifier)
Message Passing - Message Types	all Golang types possible	all Erlang types possible
Memory	no shared memory between goroutines	no shared memory between processes

	Golang	Erlang
Thread "type"	"lightweight thread" called goroutine	processes called actors (Actor Model)
Blocking mechanism	channels - goroutines (sender) blocked until receiver received message and answered the sender; if channel is buffered, this only happens if buffer is full	receive block - if message can't be matched to given pattern in receiver, process stopped until new message arrives
Mutual exclusion	possible with "sync" Module → mutex lock and unlock work the same way as mutexes in OS	not provided → Erlang is build without any shared memory, the processes don't use the same resources
Deadlocks	possible can happen if messages can't pass through the channels → deadlock detector blocks those goroutines if there is no chance to unblock the goroutine (more working and active goroutines than wating and inactive goroutines)	possible - if send messages don't match the patterns of the expected messages, receiver is blocked and can't be unblocked - if no messages are send to the receiver, the proecess gets blocked → the problem is the receive

Comparison of Code Example

For the purpose of the comparison of Erlang and Golang in view of concurrency programming, the following example was chosen. A program should calculate the main properties of a sphere. The only value that should be taken into consideration is the radius. It should be a float value. The main properties, that were chosen, are the surface area, the circular area and the volume of the inside of the sphere. Every property should be calculated in a separate process/goroutine. The following source code examples are from crucial parts of the program. The complete source code can be found in the folders "erlangCode" and "goCode".

```
func main() {

  Sphere1 := new(Sphere)                                              ①
  Sphere1.number = 0

  ready := make(chan string)                                          ②
  areaMath := make(chan float64)                                      ②
  volumeMath := make(chan float64)                                    ②
  surfaceMath := make(chan float64)                                   ②

  go allResults(ready, volumeMath, areaMath, surfaceMath, *Sphere1) ③
  go circularArea(2.5, areaMath)                                      ④
  go volume(2.5, volumeMath)                                          ④
  go surface(2.5, surfaceMath)                                        ④

  end:= <- ready                                                      ⑤

  fmt.Println(end)
}
```

① New Sphere

② New unbuffered channels are created, one is a string channel and the other ones are float64.

③ Goroutine is started with the function allResults, the parameters are all channels and a pointer to a sphere.

④ Goroutines, which calculate the properties, start with a channel and the radius that is needed for the calculations.

⑤ The variable end is initialized with the value that it gets from the channel "ready".

```
start(Radius) ->
    AllResults_PID=spawn(sphere, allResults, []),          ①
    spawn(sphere, volume, [Radius, AllResults_PID]),       ②
    spawn(sphere, circularArea, [Radius, AllResults_PID]), ②
    spawn(sphere, surface, [Radius, AllResults_PID]).      ②
```

① The allResults function is spawned as a process. The return value (pid) is then used as the value for the variable AllResults_PID.

② The other processes are spawned. One of the parameters is the pid of the allResults process.

The two functions with which the programs begin are the "main()" and the "start()" functions. In those two the main difference is clear. In Golang it is necessary to make channels if the goal is to let the goroutines communicate with each other. For this Golang provides the developer with "chan". It works in the same way as e.g. initializing maps or arrays. The goroutines themselves are started by using a "go" before the function call. It is important that the function gets access to the channels. That happens through the usage of parameters.

In the Erlang code the way that the processes are started is by using the function call "spawn()".

This returns the pid of the now "spawned" process. It can then be used as a parameter for the called function. That way there is no need to make a special construct to give to the process.

```
func allResults(ready chan string, volumeMath chan float64, areaMath chan float64,
    surfaceMath chan float64, Sphere1 Sphere){
    for{                                                    ①
        select{                                             ②
        case volume:= <-volumeMath:{                        ②
                Sphere1.volume = volume
                Sphere1.parts++
                fmt.Println("Volume is ", volume)
            }
            [...]                                           ③
        }
        if Sphere1.parts == 3{                              ④
        ready <- "All parts calculated"                     ④
        close(volumeMath)                                   ⑤
        [...]                                               ⑤
        }
    }
}
```

① The for-loop is used to make the goroutine wait for more than one message.

② The select makes it possible to differentiate between messages from different channels. In this case the channels that were given to the function are used.

③ The other cases (circular area, surface) are very similar to the volume case.

④ When all parts are calculated the main() function gets a message and this way ends the program.

⑤ All the channels are being closed.

```
allResults() ->
    receive                                                 ①
        {volume, Volume_PID} ->                             ②
            io:format("Volume was calculated~n"),
            Volume_PID ! okay,                              ③
            allResults();                                   ④
        [...]                                               ⑤
    end.                                                    ①
```

① The beginning and the end of the receive block.

② The pattern that is expected. The pattern consists of the string "volume" and a variable. The variable is than associated with the value which was sent. In this case the value is the pid of the sending process.

③ The variable is used to send the string "okay" back to the process that triggered the instructions.

④ The function is started once more. The message queue and the pid stay the same as the process was not ended and only a new function was called.

⑤ The other properties looked the same.

The function "allResults()" was used in different ways in the example code. The Golang "allResults()" function waits for a message from one of the calculating processes and is then able to print the received value in dependence of the name of the property. After all the properties were calculated and printed, the "main()"" function gets a message to be informed about the end of the task. All the channels are then closed. The goroutine doesn't die because it contains a for-loop. Otherwise it would end after receiving one message.

In the Erlang example the "allResults()" function is used to demonstrate the usage of patterns within a receive block. It also shows how a process can get the pid from another process without being told about it from the beginning. This process continues because after each message it starts the "allResults()" function all other again. This again shows an interesting attribute of Erlang processes. The pid and the message queue are not reset although the function is called again because the process is still the same and only the instructions changed. This is very logical but might seem strange in the beginning because calling the function again seems to be the same as starting a new process. But it is not. A process is always started by using the function "spawn()".

```
func volume(radius float64, volumeMath chan float64){
    powerOfThree := radius * radius * radius
    volume := float64((4/3) * math.Pi * powerOfThree)
    volumeMath <- volume                                    ①
}
```

① After the volume was calculated, it is send through the volumeMath channel. The select in the allResults() function awaits and receives it. Then this goroutine is closed.

```
volume(Radius, AllResults_PID) ->
    io:format("Volume Calculation begins~n",[]),
    Volume = (4/3) * math:pi() * Radius * Radius * Radius, ①
    AllResults_PID ! {volume, self()},                      ②
    receive                                                 ③
        okay ->                                             ④
            io:format("Volume is ~p ~n",[Volume])
    end.                                                    ③
```

① The calculation of the volume.

② The message with the string volume and the pid of this process are send to the allResults process.

③ The beginning and the end of the receive block.

④ The only message that is accepted by the recieve block is a string "okay". After this the process dies.

In this part of the example the code is nearly the same. The only difference is that the Erlang code has a receive block. This is only the case because it was specifically written that way. It doesn't have to be like that and is only for reasons of demonstrating the possibility of using simple patterns. In both codes the goroutine or process is dead by the end of it. In the Golang example the goroutine

dies after getting the message that its message was received. For the goroutine to stay alive it would have needed something like a for-loop. In the Erlang code the process lives as long as it waits for the message "okay". If there wasn't the receive block, then the process would immediately end after sending the message to the "allResults()" process. The reason for the different behaviour of the two code examples is, that one is synchronous message passing, the other is asynchronous. But there is no real difference in the code itself.

References

Used directly in Text

- [Bla20] Vincent Blanchon. Go: How Are Deadlocks Triggered?. https://medium.com/a-journey-with-go/go-how-are-deadlocks-triggered-2305504ac019, 2020, Last Visit: 05.01.2021
- [Chr11] Maria Christakis and Konstantinos Sagonas. Static Detection of Deadlocks in Erlang. https://mariachris.github.io/Pubs/TFP-2011.pdf, 2011, Last Visit: 05.01.2021
- [cht] Channel types. https://nanxiao.gitbooks.io/golang-101-hacks/content/posts/channel-types.html, Last Visit: 04.01.2021
- [con] Concurrent Programming. https://erlang.org/doc/getting_started/conc_prog.html, Last Visit: 03.01.2020
- [eg] Effective Go. https://golang.org/doc/effective_go.html, Last Visit: 03.01.2021
- [erf] Frequently Asked Questions about Erlang. http://erlang.org/faq/, Last Visit: 08.01.2021
- [Gup19] Mayank Gupta. Understanding Golang and Goroutines. https://medium.com/technofunnel/understanding-golang-and-goroutines-72ac3c9a014d, 2019, Last Visit: 03.01.2021
- [Gupt] Lokesh Gupta. Concurrency vs. Parallelism. https://howtodoinjava.com/java/multi-threading/concurrency-vs-parallelism/, Last Visit: 28.12.2020
- [Ken14] William Kennedy. The Nature of Channels in Go. https://www.ardanlabs.com/blog/2014/02/the-nature-of-channels-in-go.html#:~:text=It%20is%20the%20channel%27s%20ability,is%20called%20a%20buffered%20channel., 2014, Last Visit: 04.01.2021
- [Kha18] Kartik Khare. Why goroutines are not lightweight threads?. https://codeburst.io/why-goroutines-are-not-lightweight-threads-7c460c1f155f, 2018, Last Visit: 02.01.2021
- [Mil09] Alex Miller. Understanding actor concurrency, Part 1: Actors in Erlang. https://www.infoworld.com/article/2077999/understanding-actor-concurrency-part-1-actors-in-erlang.html, 2009, Last Visit: 04.01.2021
- [run] Package runtime. https://golang.org/pkg/runtime/, Last Visit: 04.01.2020
- [syn] Package sync. https://golang.org/pkg/sync/, Last Visit: 05.01.2020
- [unb] Unbuffered and buffered channels. https://nanxiao.gitbooks.io/golang-101-hacks/content/posts/unbuffered-and-buffered-channels.html, Last Visit: 04.01.2021

Further used sources

- [All16] Mark Allen. Erlang and Go concurrency. https://github.com/mrallen1/erlang-n-go/blob/master/Erlang%20and%20Go%20Concurrency.pdf, 2016, Last Visit: 27.12.2020
- [Cha18] Joe Chasinga. Concurrency in Go vs Erlang. https://dev.to/pieohpah/concurrency-in-go-vs-erlang-595a#:~:text=Go%20has%20something%20close%20to,and%20it%27s%20called%20multiple%20returns.&text=One%20difference%20to%20note%20is,as%20image%20processing%20than%20Erlang, 2018, Last Visit: 28.12.2020
- [Lar08] Jim Larson. Erlang for Concurrent Programming, The Concurrency Problem Vol. 6 Nr. 5. https://www.cs.helsinki.fi/u/kerola/rio/papers/larson_erlang.htm, 2008, Last Visit: 03.01.2021
- [lye] The Hitchhiker's Guide to Concurrency. https://learnyousomeerlang.com/the-hitchhikers-guide-to-concurrency, Last Visit: 03.01.2021
- [Ram17] Naveen Ramanathan. Part 22: Channels. https://golangbot.com/channels/, 2017, Last Visit: 04.01.2021