

Amrita Vishwa Vidyapeetham
TIFAC-CORE in Cyber Security

20CYS312 - Principles of Programming Languages
Assignment-01: Exploring Programming Paradigms

M NAGA RAVI CHANDRA

21st January, 2024

Paradigm 1: Concurrent

Concurrent is about execution of multiple tasks at same time effectively to create the illusion of parallelism and achieve better overall performance. If only one processor core is actively executing instructions at any given moment. We use rapid context switching, where the processor interchange between multiple tasks by giving each a small slice of time to execute which results in parallelism illusion.

Sequential



Figure 1: example of sequential in real-world

here, let's consider girls as one unit task and boys as another unit. Cafe has only one coffee machine. Boys can only get the coffee after last girl finish up fetching her coffee.

This is similar to having one processor and tasks can only be done one after the other.

Here let's say they have to board a bus in 4 minutes and it takes 1 minute to fill coffee per person. So now what about boys?? None of them gets coffee.

So, Now let's say boys and girls agree to fill the coffee as- 1 boy, 1 girl, 1 boy, 1 girl.....

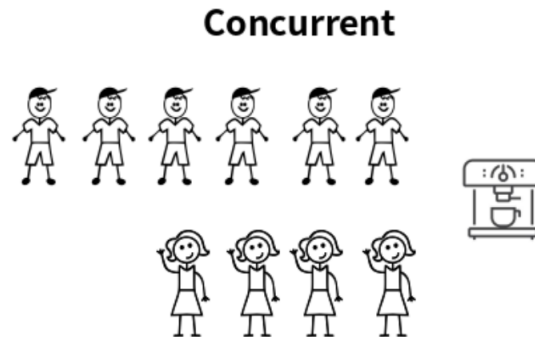


Figure 2: example of concurrent in real-world

In concurrent condition for same 4 minutes 2 boys and 2 girls can fetch their coffee. Which increases the performance. Which also increases Responsiveness to explain this if suppose we are running a restaurant we first take order at first customer and then we will take second customers order. If concurrency is not present here second customer has to wait till first customer pays the bill which reduces Responsiveness.

If Cafe has two coffee machines then boys can fetch coffee at the other coffee machine. Which is similar to having two processors other processor can take care of the remaining task.

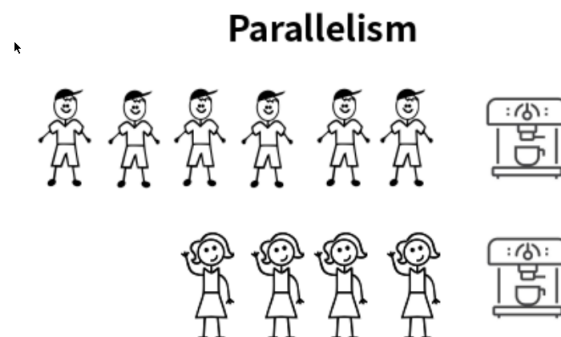


Figure 3: example of parallel in real-world

I hope above example explains the difference between parallelism and concurrent (uncovered illusion magic of the concurrent).

We may face some issues during implementation of concurrency such as sequencing of the interactions or communications between different computational executions and coordinating access to resources that are shared among executions.

Multitasking is fairly simple when all tasks are independent from each other. However, when several tasks try to use the same resource, or when tasks try to share information, it can lead to confusion and inconsistency. Locks can deal with this issue but it may also result in "Deadlock" problem.

Making sure of synchronization together with above all may seem complicated but this will produce good results such as

- increased program throughput
- High responsiveness for input/output

Combination of parallelism and concurrency may produce good results if implemented carefully.

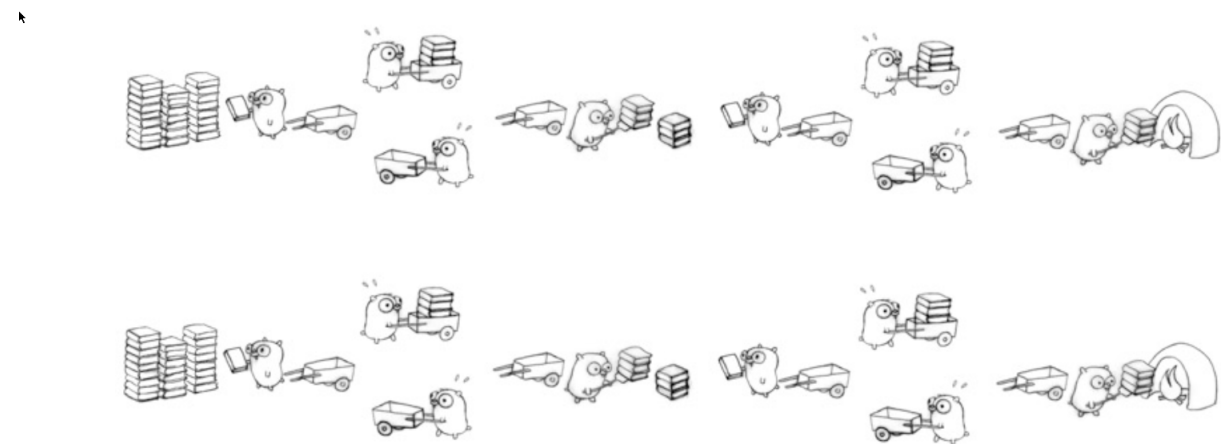


Figure 4: example of concurrent in real-world

In the above picture gophers work in the combination of both parallel and concurrent which produces good performance.

Concurrency can be two types-

- Implicit concurrency
- Explicit concurrency

Implicit concurrency– concurrency is managed automatically by the programming environment. Without the programmer having to explicitly manage threads.

example– Applications can run concurrently on a computer, and the operating system scheduler decides how to allocate processor time.

Explicit concurrency– concurrency behaviour will be defined by the programmer.

example– Programmer has to specify the concurrent behaviour in Go programs.

Every concurrent program needs process interactions for

- As they compete for exclusive access of shared resources.
- Also they interact to exchange the data.

In these both cases they need to maintain the synchronization to avoid any possible conflicts.

There are two ways for process interaction

- shared variables.
- message passing.

Some the Safety properties of concurrency-

- Mutual exclusion
- No Deadlock
- Partial correctness

Some examples of concurrency in real world–

- restaurant having single maid.
- Beehives.
- Downloads can be concurrent.
- question paper correction.
- Manufacture factories.

Concurrency may increase complexity of your program, requiring careful design and management of threads, processes, synchronisation, potential race conditions and Deadlocks. Debugging errors in can be a complex and time-consuming task due to the potential for non-deterministic behaviour and race conditions. Certain tasks may be best solved without introducing the complexities of concurrency.

concurrent is supported in many languages like:

- c++
- Clojure
- Go
- Erlang
- Haskell
- java
- Rust

Language for Paradigm 1: Go

GO is a statically typed, compiled high-level programming language designed at Google by Robert Griesemer, Rob Pike, and Ken Thompson. It is syntactically similar to C, but also has memory safety, garbage collection and CSP-style concurrency.

GO is a Multi-paradigm Language consisting:

- concurrent
- imperative
- functional
- object-oriented

Go was designed at Google in 2007 to improve programming productivity in an era of multi-core, networked machines and large codebases. The designers wanted to address criticism of other languages in use at Google, but keep their useful characteristics:

- Static typing and run-time efficiency (like C)
- Readability and usability (like Python)
- High-performance networking and multiprocessing.

Its designers were primarily motivated by their shared dislike of C++. Although C++ is one of the most widespread programming languages, many prominent software engineers criticize C++ (the language, and its compilers) for being overly complex and fundamentally flawed.

The lack of support for generic programming in initial versions of Go drew considerable criticism. The designers expressed an openness to generic programming.

Go has these following features to support concurrency:

- Goroutines: Lightweight threads managed by the runtime.
- Channels: Typed message-passing channels for synchronization and communication between goroutines.
- Built-in support: Concurrency is a core feature of the language, making it easy to write concurrent code.
- Simple syntax: `go func()` starts a goroutine, and `channel <- value` sends data to a channel.

About Goroutines–

It is a fundamental feature of the Go programming language. They are lightweight, concurrent units of execution that are managed by the Go runtime.

It is kind of Green thread (is a thread that is scheduled by a virtual machine (VM) instead of natively by the underlying operating system (OS)).

`go func()` syntax for launching goroutines, and runtime handles scheduling and context switching efficiently.

A function call prefixed with the **go** keyword starts a function in a new goroutine.

```
package main

import (
    "fmt"
    "time"
)

func main() {
    start := time.Now()
    defer func() {
        fmt.Println(time.Since(start))
    }()
    evilninja := []string{"Tommy", "jhonny", "bobby", "andy"}
    for _, evilninja := range evilninja {
        go attack(evilninja)
    }
    time.Sleep(time.Second * 1)
}

func attack(target string) {
    fmt.Println("throw ninja stars", target)
    time.Sleep(time.Second)
}
```

Figure 5: example of goroutine

Go language specification doesn't prescribe how goroutines should be implemented internally, current implementations of Go often use a strategy of multiplexing many goroutines onto a smaller set of operating-system threads. This allows Go to efficiently manage concurrent execution while abstracting away some of the complexities of thread management from the programmer.

About Channels-

Channels are used for communication between multiple different goroutine. Channels are typed, so that a channel of type `chan T` can only be used to transfer messages of type `T`. channels are ideal for sending and receiving data between goroutines without the need for shared memory. They ensure certain tasks finish before others begin, which allows to control the flow of execution. because of them efficient distribution of work between goroutines is maintained which allows maximizing resource utilization.

- `ch <- "hello"`
This command is used to send the data through channel.
`ch-channel, "hello,sir"`-message which is needed to be sent.
- `message := <-ch`
This command is used to receive the data.
`<-ch` is an expression that causes the executing goroutine to block until a value comes in over the channel `ch`.

In Fig 6 code explains about how to use channel. Here previously we used `sleep` so main function will sleep so it gives time to complete attack functions. But because of channel function and main function can communicate with each other. which will be helpful in suspension of the main code till the function to execute. Which saves execution time. In this way channel helps in synchronization of concurrent process.

```
// Click here and start typing.
3 package main
4
5 import (
6     "fmt"
7     "time"
8 )
9
10 func main() {
11     now := time.Now()
12     defer func() {
13         fmt.Println(time.Since(now))
14     }()
15
16     channel := make(chan bool)
17     evilninja := "tommy"
18     go attack(evilninja, channel)
19     fmt.Println(<-channel)
20 }
21
22 func attack(target string, channel chan bool) {
23     time.Sleep(time.Second)
24     fmt.Println("Throwing ninja stars", target)
25     channel <- true
26 }
27 }

Throwing ninja stars tommy
true
1s
```

Figure 6: example of channel

Paradigm 2: Functional

Functional programming treats functions as first class citizens. They can be passed as parameters. It encourages to keep the data and functions separate. It encourages us to create a new variable instead of overwriting the variable.

It is a declarative programming paradigm style which mainly tells "What to do?" instead of "How to do?" where one applies pure functions in sequence to solve complex problems. Functions take an input value and produce an output value without being affected by the program. Functional programming mainly focuses on what to solve and uses expressions instead of statements.

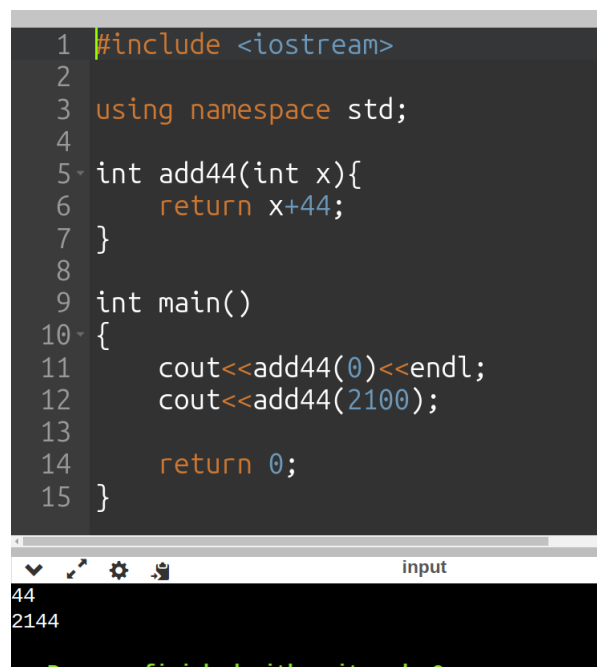
The image shows a code editor window with a dark background. The code is written in C++ and includes a function named 'add44' and a 'main' function. The 'add44' function takes an integer 'x' as input and returns 'x+44'. The 'main' function calls 'add44(0)' and 'add44(2100)', printing the results. Below the code editor, there is a terminal window showing the output of the program: '44' and '2144'. The terminal also shows a message 'Program finished with exit code 0'.

Figure 7: example of pure function

here we can see "add44" function which will not have any side effects as it doesn't modify or is affected by anything. Its output fully depends on the input given by the user, which is a deterministic behavior where the same input will always produce the same output.

Next we will see how to pass a function as a parameter. "functions are treated as first-class citizens."

```

1  #include <iostream>
2  #include <functional>
3
4  using namespace std;
5
6  int firstfun() {
7      cout << "first" << endl;
8      return 0;
9  }
10
11 int secondfun(function<int()> fun) {
12     cout << "second" << endl;
13     return fun();
14 }
15
16 int main() {
17     function<int()> fc = firstfun;
18     secondfun(fc);
19     return 0;
20 }

```

input

second
first

Figure 8: example of passing function as a parameter

In the above example firstfun is passed as an parameter to the secondfun.

Functional programs avoid situations that create different outputs on every execution (so it doesn't use "while" or "for" or "if-else"). Instead, recursive functions call themselves repeatedly until they reach the desired solution.

In functional programming we will not modify variables after created. This is to maintain the program's state through-out the runtime. we can run the code and check the results as we know that variables remain constant which will produce a determined value.

High order function

A function that accepts other functions as parameters or returns functions as outputs is called a high order function.

Advantages of functional programming:

- Functional programming uses immutable variables, which will make creating parallel || concurrent programs is easy as they reduce the amount of change within the program. Each function only has to deal with an input value and have the guarantee that the program state will remain constant.
- Functional programs are efficient because they don't rely on any external sources or variables to function, they are easily reusable across the program.
- we can evaluate computations only at the moment it is needed. As same inputs produce same outputs which results in ability to reuse results produced from previous computations.
- It will be easy to debug as there will be no hidden modifications because of pure functions.
- Functions in functional programming are easy to read. Since functions are treated as values, immutable, and can be passed as parameters, it is easier to understand the codebase and purpose.
- By following immutability we don't have to worry about accidental modification of variables and we can refer to the previous variables when needed.

Disadvantages of functional programming:

- recursion is one of the good property in functional programming, it is very expensive to use. Writing recursive functions requires higher memory usage which can be costly. They can also sometimes cause difficulty when we try to understand the code.

Real-time applications of functional programming:

- Financial Departments
- Calculator
- concurrent and parallel programming
- Data processing

Functional is supported in many languages:

- Go
- Kotlin
- Perl
- Rust
- Python
- Java

Language for Paradigm 2: Scheme

Scheme is a Multi-paradigm programming language

- functional
 - imperative
 - meta
-
- Scheme is a dialect of the Lisp family of programming languages. Scheme was created during the 1970s at the MIT Computer Science and Artificial Intelligence Laboratory and released by its developers, Guy L. Steele and Gerald Jay Sussman.
 - Scheme started in the 1970s as an attempt to understand Carl Hewitt's Actor model, for which purpose Steele and Sussman wrote a "tiny Lisp interpreter" using Maclisp and then "added mechanisms for creating actors and sending messages".
 - Scheme is primarily a functional programming language. It shares many characteristics with other members of the Lisp programming language family. Scheme's very simple syntax is based on s-expressions, parenthesized lists in which a prefix operator is followed by its arguments.
 - Scheme has an iteration construct, `do`, but it is more idiomatic in Scheme to use tail recursion to express iteration.

```
;; Building a list of squares from 0 to 9:
;; Note: loop is simply an arbitrary symbol used as a label.
Any symbol will do.

(define (list-of-squares n)
  (let loop ((i n) (res '()))
    (if (< i 0)
        res
        (loop (- i 1) (cons (* i i) res)))))

(list-of-squares 9)
==> (0 1 4 9 16 25 36 49 64 81)
```

Figure 9: example of iteration in scheme

As we can see iteration in scheme is implemented using tail recursion here loop is just a label we can name it whatever we want. This situation is functional programming where we use recursion instead of loops.

Scheme supports delayed evaluation through the delay form and the procedure force.

```
(define a 10)
(define eval-ajplus2 (delay (+ a 2)))
(set! a 20)
(force eval-ajplus2)
==> 22
(define eval-ajplus50 (delay (+ a 50)))
(let ((a 8))
  (force eval-ajplus50))
==> 70
(set! a 100)
(force eval-ajplus2)
==> 22
```

Figure 10: example of delay evaluation in scheme

The lexical context of the original definition of the promise is preserved, and its value is also preserved after the first use of force. The promise is only ever evaluated once.

Scheme is a popular target for language designers, hobbyists, and educators, and because of its small size, that of a typical interpreter, it is also a popular choice for embedded systems and scripting.

Scheme is widely used by several schools; in particular, several introductory computer science courses use Scheme in conjunction with the textbook *Structure and Interpretation of Computer Programs* (SICP).

advantages	disadvantage
improved performance increased Responsiveness best Resource Utilization Scalability	Increased Complexity Potential for Deadlocks Synchronization Overhead Debugging issues

Table 1: advantages and disadvantages of Concurrent

advantages	disadvantage
Easy to debug Concurrency and Parallelism Pure Functions immutability	Limited Mutability Not always best recursion may cause confusion

Table 2: advantages and disadvantages of Functional

Analysis

Go	Scheme
Go is a statically-typed Go has a static typing system Concurrency is a central feature of Go Go has automatic memory management Go has a growing and active community	Scheme is a dynamically-typed Scheme is dynamically typed Scheme supports concurrency manual memory management Scheme has smaller community

Table 3: Go vs Scheme

Comparison

Both concurrent and functional programming paradigms bring unique strengths and approaches to software development, offering solutions for specific challenges. While distinct in their core principles, they also share some interesting overlap in their ideas.

Functional constructs like higher-order functions and map-reduce can be effectively parallelized under certain conditions.

Concurrent environments can benefit from immutable data structures for improved consistency and avoiding shared state pitfalls.

Both paradigms encourage developers to focus on what needs to be done, leaving the execution details to the underlying system.

Differences:

Concurrency:

Mainly Focuses on Handling multiple tasks seemingly simultaneously, maximizing resource utilization and responsiveness.

Key concepts: Threads, processes, synchronization mechanisms, communication channels.

Advantages are Improved performance, handling multiple inputs/outputs efficiently, responsiveness to external events.

issues faced Complexity, debugging challenges, race conditions, potential for unpredictable behavior.

Functional:

Mainly Focuses on Immutability, pure functions, and side-effect-free computations, leading to predictable and modular code.

Advantages are Easier reasoning about code, improved testability, modularity, and composability.

issues faced May not be readily applicable to all problem domains, may require different problem-solving approaches.

Comparing Go and Scheme

Similarities:

Both are statically typed, compiled languages which leads to offering better performance and type safety compared to dynamically typed languages. Both emphasize simplicity and expressiveness favoring clarity, conciseness, and readability in code.

Both support functional programming features by utilizing higher-order functions, recursion, and immutable data structures, albeit to different degrees.

Both are well-suited for specific domains like Go shines in concurrent and network programming, while Scheme excels in symbolic manipulation and AI research.

Differences:

Go is built for concurrency with lightweight goroutines and channels, fostering efficient parallel execution. Scheme, while capable of concurrency, uses traditional threads and requires more manual setup.

Go uses automatic garbage collection for memory management, while Scheme relies on manual allocation and deallocation, giving programmers finer control but increasing complexity.

Challenges Faced

- I have faced issue with finding concepts for scheme.
- After finding some references i cannot understand the scheme concepts so well.
- I don't know how to fill required number of pages with knowledge i understand
- i have faced an latex code issue as you can see at Analysis.
- I tried my best other than above i am comfortable.

Conclusion

- It depends upon situations to decide which language is best for that condition.
- concurrency can increase performance and utilize the available resources as much as it can
- functional is useful when same code is needed multiple times.
- but both has their share of advantages and disadvantages

References

- I referred about concepts of Concurrent from https://en.wikipedia.org/wiki/Concurrent_computing and also from the document uploaded in the Github.
- I took image used for concurrent from <https://github.com/nikhilkumarsingh/concurrent-programming/blob/master/concurrency.ipynb>
- I referred Go concepts from [https://en.wikipedia.org/wiki/Go_\(programming_language\)#](https://en.wikipedia.org/wiki/Go_(programming_language)#)
- I took code for channel from this following video <https://youtu.be/LgCmPHqAuf4?feature=shared>
- I referred functional programming from <https://youtu.be/dAPL7MQGjyM?feature=shared> and https://en.wikipedia.org/wiki/Functional_programming#
- I referred passing function as a parameter from <https://youtu.be/-Qotqv80azk?feature=shared>
- I referred concepts of scheme from [https://en.wikipedia.org/wiki/Scheme_\(programming_language\)](https://en.wikipedia.org/wiki/Scheme_(programming_language))