The need to optimise the procedures that computers use to perform calculations is growing as a result of the advancement of technologies that are used in the development of processors for personal computers and the rise in the demand for calculations to be performed quickly. Customers of software systems have the right to demand from developers that the maximum amount of data possible be processed in the shortest amount of time. This is done with the intention of lowering the cost of software support and the processing of a single unit of information. Processing times cannot be reduced simply by using more efficient sequential algorithms because of the massive amounts of user data that must be analysed, for example, by stakeholders. A significant number of algorithms are based on the method of sequentially carrying out particular operations on a list of entities. These kinds of algorithms can be sped up by a factor of several by utilising the resources of multi-core processors and the tools that are designed to work with them. This is possible because the actions that are performed on each element of the processed sequence will probably remain the same regardless of the number of iterations.

One of the technologies that allows you to perform multithreaded calculations is the programming language developed by Google since 2007 and presented in 2010 - Go (or Golang). The language is low-level enough to handle and interact with threads, for example, but is much easier to learn and use than any of the other low-level programming languages popular today (such as C++) . Also, the advantages of Golang include such aspects as:
1. Static typing;
2. The garbage collector is a tool that eliminates the developer's need to manage the memory and resources used by the software manually;
3. Custom binaries;
4. Fast and efficient compilation;
5. Simple operation with multithreading.

The main concept that will be considered is competitiveness. Concurrency is the ability of various elements of the program, algorithms or tasks to be performed in an unordered or partial order without affecting the original result. This significantly affects the efficiency of the software and the speed of volume calculations. The main tools for working with concurrency in the Golang programming language are Goroutines - easy to execute methods or functions that are executed independently of the method or function that calls them. The effectiveness of Goroutines is due, in large part, to their behavior, which consists in scheduling the processing of a given number of system threads, which, in theory, allows processing any number of Goroutines . It is worth noting that one of the advantages of using goroutines in the Go programming language is to minimize the effort on the part of the developer to write code. Because in order to force a method or function to be executed in a goroutine, you need to use the go keyword before calling the method. This will allow them to be executed independently of the calling thread, and competing with each other between the threads of the processor cores.
The next basic concept for working with concurrency in Go is channels - a way of communication between individual goroutines. Channels are a conditional "bus" for communication and synchronization of executed routines. With the help of channels, there is no need to create intermediate conditional variables that would control and check each of the executed goroutines for the completion of a certain stage of the method or its complete execution. Channels, like any container, must be pre-created before use like this: make( chan int ). The given code creates a channel to transfer integers. To process "events" created by transferring data to channels from the goroutine, you can use the select operator, which blocks the execution of the goroutine until at least one of the described situations is triggered. As an example, we give the method of obtaining the Nth Fibonacci number:

```go
package main
import "fmt"

func fibonacci(c, quit chan int) {
    x, y := 0, 1
    for {
        select {
            case c <- x:
                x, y = y, x+y
            case <-quit:
                fmt.Println("quit")
                return
        }
    }
}

func main() {
    c := make(chan int)
    quit := make(chan int)
    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(<-c)
        }
    }()
    quit <- 0
    fibonacci(c, quit)
}
```

The method for obtaining the Fibonacci number is called 10 times in the for loop, which is executed in the goroutine. After each iteration, the next value is obtained from the resulting channel c, which is passed as a parameter to the number calculation method, using the <- operator, and upon completion of the routine, the value 0 is transferred to the quit channel, which initiates the printing of the word "quit" and completes the execution of the method fibonacci. The fibonacci method uses the select { case } construct, which expects one of two events to occur:

1. Inputting value to channel c;
2. Getting the value from the quit channel.

It becomes possible to receive a value from the quit channel, the execution of the routine is completed, the word "quit" is displayed on the screen. If the event of entering a value into channel
c is triggered, then the calculation of the next two numbers of the Fibonacci sequence is performed and the execution of the infinite cycle continues. Thus, the developer is freed from the need to create intermediate variables to check the readiness of the execution of the routine. As a result, the algorithm is performed much more efficiently than a similar sequential one.

To demonstrate the effectiveness of using competitive methods over sequential ones, sequential and competitive algorithms for calculating the sum of numbers from 1 to a given limit were developed (as an example, the sum from 1 to 1000000000 was calculated), tested, and the execution time was measured. The research results and algorithm codes are given below.

Code for sequential sum calculation:
```go
func SerialSum() int {
    sum := 0
    for i := 0; i < limit; i++ {
        sum += i
    }
    return sum
}
```

Code for competitive sum calculation:
```go
func ConcurrentSum() int {
    n := runtime.GOMAXPROCS(0)
    res := make(chan int)

    for i := 0; i < n; i++ {
        go func(i int, r chan int) {
            sum := 0
            start := (limit / n) * i
            end := start + (limit / n)
            if i == n - 1 {
                end = limit
            }
            for j := start; j < end; j += 1 {
                sum += j
            }
            r <- sum
        }(i, res)
    }
    sum := 0
    for i := 0; i < n; i++ {
        sum += <- res
    }
    return sum
}
```

In both examples given, the value of the limit constant is set equal to 1000000000. After the program is started, the execution time of the competitive method and the sequential method are measured. The results are shown in Picture 1.
The result already reflects the efficiency of using all available processor cores to perform massive calculations, speeding up the program by approximately 2.8 times compared to sequential execution.

Thus, the Golang programming language is a rather promising tool for the development of massive systems, which can speed up the execution of software code several times by correctly converting sequential algorithms to competitive ones.

References: