System Analysis of the Parallel Execution Problem

Denys Goldiner
dept. of Applied Mathematics
Kharkiv National University of Radio Electronics
Kharkiv, Ukraine
denys.holdiner@nure.ua

Andriy Tevyashev
dept. of Applied Mathematics
Kharkiv National University of Radio Electronics
Kharkiv, Ukraine
tad45ua@gmail.com

***Abstract*—The article studies the prerequisites for the appearance and dissemination of the approach to optimizing the execution of program code through the simultaneous execution in several threads, analyzes the problem of parallel computing, identifies factors affecting the feasibility and effectiveness of the implementation of the architectural solution for multi-threaded program execution. Also, the problems that may arise during the implementation of optimization are considered. The main advantages and disadvantages of using the Go programming language for implementing parallel execution of a software product are considered.**

Keywords—parallelism; concurrency; execution thread; source code; memory allocation; optimization; multithreating; programming

IX. INTRODUCTION

Software becomes more and more complex, nowadays. It is extremely important to define the problem to be solved before start writing a new program. As a result, system design becomes one of the required conditions of a successful project. The term "system" implies a complex implementation that provides fundamental services to make software solve the determined goal. The following are common elements of a system design:

- Processes — design and redesign business processes.
- Architecture — defining business information and technical structures to support the design. The design of a system is often separated into an architecture that provides the height level structure and a design that provides enough detail to implement the system.
- Data — defining data models.
- Events — defining events and how they will be processed.
- Business rules — considering how business rules will be processed.
- Applications — design of applications. An application is a software component designed primarily to be used by people. A system may touch upon multiple applications.
- Services and components — defining the system as a series of services and components.
- Integration — designing how things work together including services, processes, events and data.
- Technology — defining the technologies that will be used including infrastructure, systems, applications, components, toolsets, libraries and APIs.
- Information security — defining how the system will be secured.
- Deployment — defining how the system will be logically and physically deployed.
- Constrains — additional constraints to guide implementation such as a set of principles, standards, and tools.
- Capabilities — defining the business and technology capabilities that the system provides or what performance load it should be able to handle. The estimated time required for the creation of the program is also a very important resource and there is no need to spend too much time on optimizations when the expected load does not require improvements. Also, to make effective optimization — research is needed to find the bottleneck. There is an important rule: "do not make optimizations when they are not needed".

Therefore, software architecture always depends on these factors. However, what can we do to speed up the execution of our program if we already faced the performance challenge?

The current reality tells us that single-core processors are dying out, and multicore processors are covering all sectors of the market, starting from home usage in personal computers and cell phones, and ending with server clusters in the Silicon Valley. The reason that almost all modern processors are multicore is that Moore's law is not working anymore. It is too expensive to add new transistors to a processor core and these actions do not provide expected efficiency. The logical solution was to make a multi-core processor, which will use a smaller number of transistors per core, but it will win the performance race via executing multiple tasks at the same time. This flow should increase the performance multiple times and at the same time, save money on production.

Nevertheless, any processor is useless itself, it only works with the software and should be supported by the operating system running over it and by programs executing by the OS.

If you run the program which simply doesn't use threads it will work on a single OS core at the same time, and its execution speed will not change. It means that the performance of executing programs depends on its architecture even more than of the processor's characteristics. So, the next challenge is to learn how to use the possibilities of running our code in parallel.

X. Parallelism

Programs are not communicating with the processor's cores directly. There is an operating system between them, taking the role of translator. If you need to translate the text into some language — you will need a translator who knows the required language, another case he would be useless. Therefore, to be able to use the possibilities of multicore processors, they should be supported by OS.

The programs themselves are just numbers of commands to OS to be executed on the processor, and as usual, modern programmers do not work with the low level of abstraction. They prefer writing programs on the height level programming languages. As the program is the number of programming language commands, then to make the software work in parallel the language should support this possibility, and it should have all needed instruments not only to run simultaneous execution but also to make the threads communicate between each other. Based on this we can make an assumption that the programming language is also an important part of the system called "parallel execution".

The main idea of the modern programming languages, which have been created during the last 10 years, is to give the programmers an opportunity to use the full potential of the new generation processors' possibilities and to make program execution as effective as it is possible.

Parallelism — is an approach when multiple operating system threads are executing tasks preassigned by one single program. At this moment it is important to understand that the tasks might perform different either the same actions, in fact, the main idea is simultaneous execution [1].

There are three flows how the programming languages spread program execution streams between OS threads:

- one to many,
- one to one,
- many to many.

One to many — N program threads are executing on one OS thread. The advantage is that the execution context is changing extremely fast, but it does not allow the program to use multicore processors' potential. For example, such flow is used in Python.

One to one — 1 program thread is executing on 1 OS thread. The advantage is that it allows us to use multicore processors' potential, but the execution context change takes a lot of time efforts and leads to execution interruptions. As a result, the tasks to be executed in parallel should be big enough to justify the accompanying expenses of parallelism. This flow is used in many languages, e.g. Java, C#, C++, etc.

Many to many — arbitrary number of program threads are executing on arbitrary number of OS threads. This technique is nearly new and is supported mainly by the newest programming languages, for example in Go. It tries to combine all the advantages of both previous flows, leveling disadvantages. This means that we might use all potential of multicore processors, and at the same time, we should be able to switch between program trades fast [1, 2].

XI. Synchronization

When the background is covered, we need to start planning the architecture for our future program, and we should take into account all specific moments of parallel execution, like the fact that the simultaneous run of threads leads to additional consequences. The programmers need to be concerned about the next things:

- synchronization,
- sharing information between threads,
- memory management,
- garbage collection.

Synchronization is when the program thread needs to wait until some synchronization event will be triggered. For example, the main thread should wait for all worker threads to finish their calculations, we cannot continue execution of the program till then. An issue can be that there is a possible scenario when due to wrong synchronization implementation program comes to a deadlock state. Deadlock means that all existing threads are blocked forever and cannot continue their execution. This block is instant, and as a result, that program will fall into a critical state. Therefore, it is very important to know how this process should work. To avoid such situations, we need synchronization primitive called semaphore, it is an atomic signal value which can be changed by one thread showing all other threads that they should react on its state change, for example, to continue or to stop execution until the semaphore will be switched back [1, 3].

Sharing information is closely related to synchronization and it is also an important theme. As usual, parallel execution generates some results and we need to have a possibility to share the data with other treads. This requirement leads to the question: "How can we share the memory between threads in a safe way?". There could be a situation when two different threads are trying to modify one same memory element. Such situations mean that the variable value is not determined, and it cannot be defined because treads are in conflict. This kind of situations is called: "race conditions", and we need to generate rules and architectural solutions to avoid them in the program software we provide. Also, synchronization primitives should work in a thread-safe mode, so we can be sure that only one thread had performed an action at once this is called atomic operations. Synchronization primitives are using data types of special size to guarantee that the operation of value change can be performed in a single processor action.

XII. Memory menegment

One more important theme is memory management, while writing programs, we use variables, which are in fact aliases to

memory layouts [2, 4]. RAM is not endless, and we should control the number of allocated resources for our program by using it with sense and collecting unneeded memory for farther reallocation. Different languages provide a variety of workflows to handle this task, we can split them into two groups:

- manual memory management,
- garbage collectors.

Manual procedure is when the programmer should explicitly decide when and which memory can be collected. As a result, such flow leads to needing to know if the memory is still needed and this task is assigned to the developer. It can be a source of mistakes and bugs because it is a very hard task to know everything about all variables created in the program. But there is also a benefit, this flow takes fewer efforts to be processed and as a result programs with manual memory management are working faster.

The garbage collector is a special language process that should make deep analysis of which variables are still in use and which of them can be released. It is a difficult task, especially in the context of parallel execution, because we should be sure that none of the currently working threads will need the variable we check. Garbage collectors make this work automatically and thus minimize the possibility of a mistake, so they make the software more reliable [5]. As a disadvantage, they are also execution on the processor and are taking part in its performance. This fact means that as usual, such a solution will be executing slower than the one without the garbage collector.

XIII. Implementation with Go

Now, when we briefly have discussed the backstage of parallelism and possible problems connected to it, we can start research on how to solve all mentioned tasks with the minimum efforts. First, we need to choose the programming language to be used. After a long research I decided to choose Go language, this decision is based on the next arguments:

- modern language with great native parallelism,
- reach synchronization toolset out of the box,
- garbage collection optimized for parallel execution.

Go is a modern programming language created to cover 21st century needs, as a result it uses “many to many” threads scheduler. Engineers who created Go decided to implement the concept called CSP which was first described in a 1978 paper by Tony Hoare. The idea of CSP — communicating sequential processes, is that the program consists of two types of entities:

- simple operations,
- synchronization events.

Go was not the only implementation of this theoretical idea, but it is the most successful. Go language matches the CSP by concurrency principle, which can be interpreted as the architectural idea of lightweight threads called goroutines, which should be easy to execute in parallel and communicating between each other thought special synchronization mechanism called channel [5]. Channel is a primitive consisting of reader treads queue, writer threads queue, data buffer and mutex to make all communication actions thread safe. This element makes the communication between threads safe and easy to handle. Also, as a standard tool, there is a “select” operation. It is needed for orchestration between multiple channels, and it is widely used to handle the communication between threads in Go. The “select” statement consists of cases, each of them is waiting for its condition to be triggered, as an example, the condition could be the write or read into the channel event. The select construction will be waiting for any of the conditions to be triggered, synchronizing the execution. As channels are asynchronous – it is a possible scenario when multiple cases are ready to be executed at the same time. In this case, a random one would be chosen to be processed. If we need only to check if the case is ready or not, without stopping the execution of a function, we can use the "default" operator, and it will be chosen automatically if none of the other cases are ready to go.

It was critically important to minimize resources consumed by the garbage collector in Go, so creators of the language resort to stack/heap system where the compiler always tries to allocate memory on the stack. Stack allocation means that the memory could be collected just when the function in which the variable created will finish its work. Garbage collector works only with memory allocated on the heap, these resources are shared and can be used by multiple goroutines. Goroutines have their own stack memory, and it is isolated for other goroutines to be safe [6].

The disadvantage of the Go programming language is that the garbage collector is also a process with own goroutines, and it consumes processor resources to scan the heap memory and clean it. If the program allocates a lot of memory on the heap, the GC will increase its needs in memory and processor execution time, and it will slow down the execution of the program in general.

Memory management and garbage collector optimizations are very perspective places to be researched but the difficulty is that it is a tradeoff, the code quality of the optimized program would be decreased. Therefore, channels have a backdoor to write memory directly to other goroutines, which is the only such situation available in the language. In this case, the compiler can allocate memory on the goroutine stack only and then copy the value of the variable directly to another goroutine stack, minimizing the amount of allocated memory on the heap. This feature should be used for optimizations.

Scheduler in Go is a very important mechanism, its role is planning goroutine execution. One of the main reasons for it to be integrated into the language is that we need clear rules of goroutine context switches.

Context switch conditions in Go runtime are:

- garbage collection,
- system calls (blocking, non-blocking, CGo, etc.),
- function calls,

- goroutine creation,
- synchronization operations,
- special function “Gosched” of the “runtime” package.

It needs to be noticed that execution context triggers mean that currently executing goroutines might stop the execution, but also it is possible that they will continue working.

Another reason is a minimization of the context required by the goroutine to continue its execution. In Go, goroutines are executed on the virtual runtime threads, implicitly soft-connected to OS threads. It makes context lightweight and it becomes chipper to change executing goroutine when we need it.

To implement communication thought channels there are so-called concurrency patterns. Pattern — is a blueprint describing a possible solution to a common problem. In our case, these patterns describe how can we switch the architecture into concurrency using optimized synchronization primitives and main advantages of the Go programming language [7, 8]. Next patterns can be implemented using standard Go library:

- generator,
- fun-out,
- fun-in,
- workers,
- subscriber.

Generator — is a function which returns channels, it is very helpful for dynamic algorithms when for a new receiver program can provide new channel for communication [8].

Fun-out — pattern when the function splits data coming from one channel into multiple output channels. Through output channels, child goroutines should receive values.

Fun-in — is construction to aggregate data coming from multiple channels into one channel. It is handy to combine execution results from multiple goroutines into the parent goroutine thread.

Worker — is a mechanism that is needed to spread data to be executed between goroutines (often with limitation of threads number), which send results into channels. Usually, the worker is implemented by an improved combination of fun-out and fun-in patterns [8, 9].

Subscriber — the pattern is used to give goroutines to subscribe for a notification from the channel and be able to take actions on successful read from it. It can be implemented with blocking when the subscriber waits for notification, or it can continue execution until notification would be triggered [9].

CONCLUSIONS

The parallel program execution is inexorably becoming dominant in the computing field. Next-generation technologies with native parallelism do an excellent job in solving the objectives of the programming language given tasks. Creating a software product in Go language, we are provided with a large pool of tools, ideally suited for solving problems of synchronization, optimization, and parallelism. The developer gets many new opportunities to solve common problems such as deadlocks, data races, and synchronization of parallel execution. The internal architecture of the language runtime package is perfectly designed to support concurrency.

REFERENCES

[7] William Kennedy. Go in Action. / William Kennedy Brian Ketelsen, Erik St. Martin. – Shelter Island: Manning Publications Co, 2015. - 241p.

[8] Caleb Doxsey. Introducing Go. / Caleb Doxsey. – O’reilly, 2016. – 111p.

[9] Alan A. A. Donovan. The Go programming language / Alan A. A. Donovan, Brian W. Kernighan. – New York. etc: Addison-Wesly, 2016. – 1087p.

[10] Matt Butcher. Go in practice. / Matt Butcher, Matt Farina. – Shelter Island: Manning Publications Co, 2016. – 287p.

[11] Mat Ryer. Go Programming Blueprints. / Mat Ryer. – Birmingham: Packt Publishing Ltd, 2015. – 255p.

[12] Mihalis Tsoukalos. Go Systems Programming. / Mihalis Tsoukalos. – Birmingham: Packt Publishing, 2017. – 585p.

[13] Katherine Cox-Buday. Concurrency in Go. / Katherine Cox-Buday. – O’reilly, 2017. – 223p.

[14] Shiju Varghese. Go Recipes. / Shiju Varghese. – Cheranalloor: Apress, 2016. – 237p.

[15] Mario Castro Contreras. Go design patterns. / Mario Castro Contreras. – Birmingham: Packt Publishing, 2017. – 376p.