A comparison between C and Go implementations of a Traffic Control System Simulator

Guilherme H. K. Martini[1], Jean Marcelo Simão[1], João Fabro[1]

[1] Centro de Pós Graduação em Engenharia Elétrica e Informática Industrial (CPGEI)
Universidade Tecnológica Federal do Paraná (UTFPR) – Curitiba, PR - Brasil

ghk.martini@gmail.com, jeansimao@utfpr.edu.br, fabro@utfpr.edu.br

Abstract. *This paper presents a comparison between C and Go implementations of a Traffic Control System Simulator. The analysis focuses on comparing both languages from a time efficiency standpoint. Also, implementation complexity, multi-threading capabilities, techniques, and technical constraints of both languages are shown. A quantitative analysis of the simulation runs is used to clarify the obtained results, which show that Go is a usable, flexible, and well-performing language for applications, having a good trade-off between productivity and efficiency, being a choice over C for some cases. C is still opted as a reliable language for low-level and strict-timing programming.*

1. Introduction

Modern computing deployment drives the need for the creation of new programming languages that would simplify and accelerate development of software. Doing more with less without losing control of what is being coded is one of the key reasons for continuous improvement of programming languages. This efficiency gain can mean either coding less and faster to solve a specific problem or it can mean that the same hardware architecture can become capable of doing more tasks simply by using a better suited programming paradigm [Rojas 2000].

In the recent years, Go (which is also referred as Golang) is becoming popular due to its pragmatism [TIOBE 2018]: it brings features such as good memory management, error handling capabilities, simplified debugging, high readability while still being concise and suited for performance-driven applications [Rouse 2017]. Its syntax, which inherits some concepts from C, was designed with clarity in mind, having only 25 keywords. Thus, it is minimalistic and easy to write [Pike 2012].

In this context, a comparison between Golang and other programming languages is necessary. In this paper, moreover, it is presented the comparison in the context of system with time restriction for embedded systems. Thus, the C language was chosen for this comparison because it is still the most used one in this sort of application [TIOBE 2018]. This C characteristic is useful to determine how efficiently other programming languages deal with time performance: if the same software is coded in C and also in any other language and then compared in terms of how fast they can run in the same architecture, it becomes clear how much time-efficient that other programming language is.

Besides time efficiency, but in the same work scope, an analysis of how well multi-threading is dealt is also made necessary given the actual widespread usage of multi-core computers architectures [El-Seoud 2017].

2. Background comparison

Table 1 shows a top-level comparison of both languages where it can be seen that they make use of the same programming paradigm: both are Imperative and Procedural [Van Roy 2004]. Differences appear with object orientation support. C only supports structs, and they can only contain sets of data, not methods. On the other side Go does not have an "object" declaration, but it allows the programmer to declare "types" which can describe methods and data sets. Also, Go has "interfaces" that can be used to create methods that take in generic parameters, which is a type of polymorphism. Class inheritance is also substituted by type embedding, which can also be used on interfaces. Those design characteristics aim for boilerplate reduction [Pike 2018] [Lämmel 2003].

They are both compiled and do not support scripting. C has a straightforward preprocessor which is very helpful for code characterization, a feature that is not yet available in Go. C is considered a mid-to-low level language since it isn't very far from machine code [Kernighan 1988] and doesn't provide any memory management while Go has an automatic garbage collector. When working with parallelism, Go has native support to worker threads, thread syncing and data channels [Pike 2014], features that are not native in C, although Pthreads are available as a POSIX standard.

Table 1. Go and C main features comparison

Language Name	***Go***	***C***
Creation date	2007	1972
Paradigm	Imperative, Procedural	Imperative, Procedural
OOP support	Yes[a]	No
Interpretation	Compiled	Compiled
Keywords	25	32
Preprocessor	No	Yes
Memory Management	Automatic	Manual
Abstraction level	High	Low
Script support	No	No
Parallelism	CSP based, channels	Pthreads, OpenMP, MPI

[a.] Type embedding and interfaces are used to replace class inheritance and polymorphism

Deep into Go's programming paradigm, it can be considered a multi-paradigm language when using its library modules [Clark 2004], suited for multi-threaded with asynchronous messaging applications. The Go language is classified as Active-Object/Object-Capability-Programming because of its following properties: it has named state, it has closures in the form of anonymous functions and it creates threads with the use of the "go" keyword. Also, it can create ports by instantiating data streams and it supports dynamic object relations by manipulating local and shared cells.

3. The Simulator Specification

3.1. Overview

A traffic control system simulator is a software that is capable of simulating traffic behavior. It consists of a certain number of streets, crossings, lanes, cars, and traffic lights.

During a simulation run, cars would move across the streets and obey standard rules such as: stop on red lights, follow street's flow direction, do not collide with other cars, do not block crossings, move when lights are green, turn on crossings from time to time. Cars are inserted at the beginning of every street on a determined timeframe according to each simulation purpose. Still, logically, they also leave the streets after moving past them. This simulation scenario allows for a study of traffic jams and also for a study of traffic lights synchronization strategies to avoid them. Moreover, it also allows some comparison between programing languages of same or different paradigms.

3.2. Specification

For the comparison of C and Go implementations, a square map of 10 north-south streets and 10 west-east streets with 100 crossings was implemented. Cars would enter this map following a Poisson distribution that varies between 0.1 to 0.5 cars per second, per street lane. The simulation timeframe is of 2000s, which is not the execution time but the time of the simulated real-time clock. The number of lanes per street vary from 1 to 4 and every crossing has a turn rate probability from 10 up to 35%. Two traffic light strategies were implemented: 1 – traffic lights have a fixed cycle time and are not synchronized with each other, called Independent Control (IC); and 2 – depending on the block congestion level, traffic lights can have its red time shortened so car flow is increased, called Control based on Congestion Level (CBCL).

A full and better detailed specification is available in a specification where all rules are set, including semaphore timings, street lengths, car speed, car size, etc. [Renaux 2014].

3.3. Motivation

This simulator has a well-defined set of rules for its implementation. Its requirements ask for the use of data sets and has enough need for mathematical operations that multi-threading can be used to shorten execution time, leading to a good usage of different features of both languages, being then considered well-suited for this work.

4. C Development Process

Based on the fact that C is traditionally a good language for light-weight applications, it was chosen for the first implementation so the focus would be to have a well-performing version of the traffic simulator for a comparison basis with the Go version. Some spartan programming techniques [Lalouche 2016] were used to reach the following needs:

- No use of stack and dynamic memory allocation to reduce execution time. Manual memory mapping was used instead.
- Encapsulation and decoupling are used as directives to minimize variable scoping, but this cannot increase execution time significantly.

- Minimum use of variables and code branching.
- Functional blocks should have the minimum possible external calls/entry points but the number of functional blocks should not be big enough so that the boilerplate becomes a burden.

With these directives in mind, a group of data sets, functions and definitions were developed, aiming at a minimalistic implementation that fulfills all requirements.

4.1. Software structure

The structure designed for the C version is shown in Figure 1. The package called "street" holds most of the data. This data is declared straight into a memory address, making them public inside the street package but not visible to the other ones. Set functions were written so that a car insertion in the streets were made possible. Cars are passed as reference from the traffic manager package, which holds a list of all cars also mapped in the memory. Whenever a car reference is received, the street package uses pointers to indicate in which "car slot" of the streets every car is situated.

The algorithm that takes most of the execution time is in the same package and is responsible for moving all cars during the simulation. The chosen approach was to make "car slots" in the streets that would point to cars whenever they are on the map. Cars change from slot to slot. The algorithm also takes into consideration the street direction. Thus, cars that are closer from the end of the streets are moved first and then the collision testing between cars become simplified. This also leads to a small change in execution time when the streets are filled with cars when compared to an empty map at the beginning of a simulation.

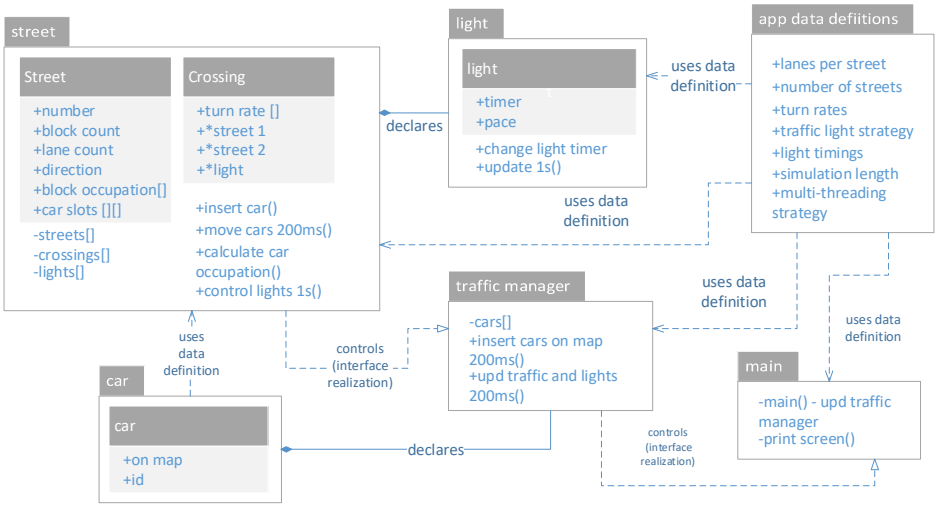


Figure 1.Software structure

Since there are no copies of data and all information is passed as reference, there is little object copying and lesser stack usage. Still, a lights package was created to manipulate traffic lights timings and to grant the interlock between lights of a same crossing. This package declares structures and functions that would take in light structures as parameters and control them independently. The layer that call the functions inside this package is the one responsible for lights synchronization, i.e. the street package.

All data inputs such as turn rates, number of streets in the map, number of lanes per street are in the “app data definitions” package. It holds a group of definitions and look up tables that are public as read-only information to whichever another package that imports it. Car package declares only the attributes that constitute the car. In turn, the traffic manager package is the one responsible for controlling the car insertion according to a Poisson curve along the simulation. It also has the loop control that does the time increment of the simulation.

The main package is responsible for initializing the structures by calling init() functions of the other packages and by calling the traffic manager package functions that run the entire simulator. Besides, there are three different ways to run the simulation:

1: The simulation can run in real-time so that 1 second of simulation represents 1 second of time, taking 2000s to finish. Being then synchronized with an application clock.
2: The simulation can run in real-time so that 1 second of simulation represents 10 seconds of time, taking 200s to finish. Being also synchronized with an application clock.
3: The calls are not synchronized with an application clock and function calls are called again as soon as they return from their previous calls, being then limited to the CPU processing capacity. The time it takes depends only on processing capability and on code efficiency.

For this study, runs that were limited to an application clock, that is, cases 1 and 2, were used only for debugging and troubleshooting purposes during the development. All execution time evaluations were made under case 3.

5. Go development process

After this structure was coded and debugged in C, a translation to Go was made. Since both languages were designed under the same computational paradigm, it was expected that the same software structure would make it possible to attend all requirements. The results of code translation were found satisfactory, here are the main difference points:

- Go still does not have a preprocessor (there can be directives to not compile a file, but there are not tools to embed compiling directives in the code), so all #define and #ifdef statements had to be replaced by read-only enumerations stored in RAM memory. Then, the code had some extra “if()” statements added.
- There are no header files in Go. So, packages became a single *.go file instead of one or more *.h and *.c files. Dependencies became clearer.
- The syntax between them is similar, but differences in parameter passing, error capturing and data declaration led to a longer than expected translation.
- Thread control and application clock control is much easier to implement in Go, its native libraries make it very easy to programmers kick-start their applications.
- Error back-tracing, hence debugging, is very clear and helpful in Go when compared to the little support in C.
- Go has a lot of syntactic sugar in its syntax making it more concise.
- Data encapsulation and object orientation is supported in Go. So, while translating the code, some data that was once public inside structures could become private in the Go application.

6. Multi-threading

As dividing work between multiple computer cores was also of interest, a deeper explanation on how this was made is necessary. All descriptions for multi-threading fit both implementations, C and Go. Figure 2 shows the execution flow of the program under a workflow representation. It can be noted that there is not a lot of conditional branching, the program is sequential most of the time, with "while" statements represented by gateways. These while statements are used to iterate through the repetitive data sets such as all cars in the map, all traffic lights in every crossing, etc.

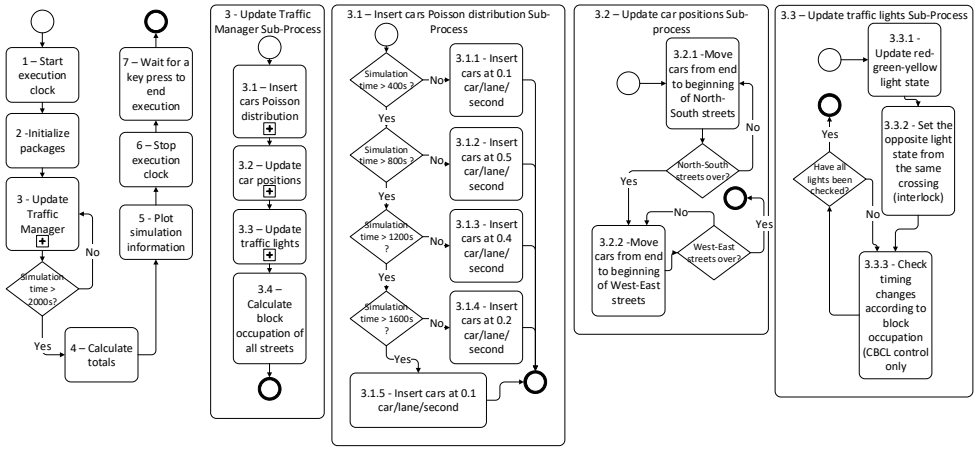


Figure 2.Execution flow of the application

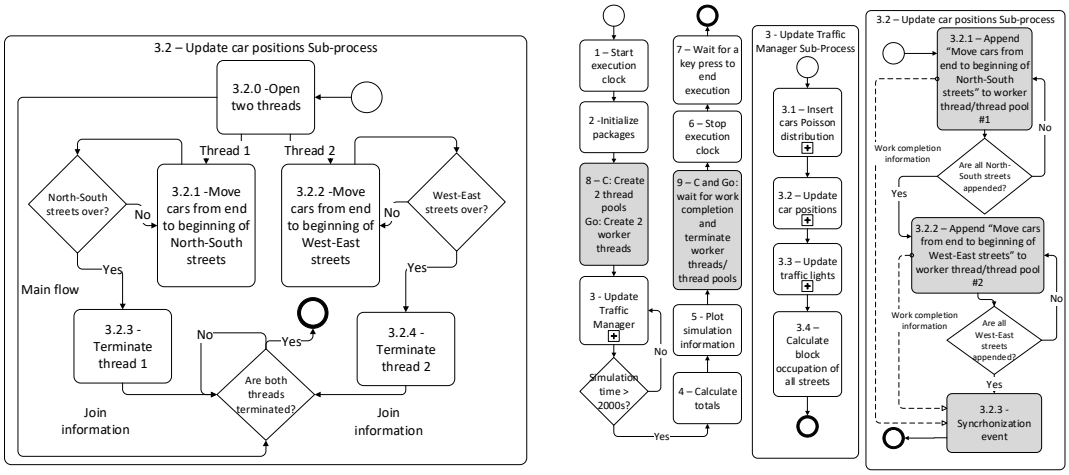
Around 85% of the work done by the processor during the simulation happens on sub-process 3.2. This was measured by checking the execution clock that is started on task 1 of Figure 2 and is stopped on task 6. When a bypass is added over 3.2, timing measurements are reduced by 80 to 90% of what it was without the bypass. Logically, this would be the best place to work on execution time improvements. Figure 3 shows a two-thread approach that was implemented: Whenever 3.2 is called, two extra threads are created, each one takes half of the "car slots" processing. Since they operate with pointers and changing where they point to, the threads can share the access to data without causing any race conditions or deadlocks.

The approach shown in Figure 3 did not have the expected outcome in C (using Pthreads): During a 2000s simulation, sub-process 3.2 is called 10000 times, so the creation and termination of threads became an unexpected overhead, causing no improvement when compared to the single-thread approach. Detailed results are shown in the next sections.

This led to another design: the use of thread pooling in C and of worker threads in Go, depicted in Figure 4. Blocks that were painted in gray are the ones that changed/were added over what was shown in Figure 4. For the C implementation, a library had to be added [Seferidis 2018] since thread pooling is not a natively supported resource. When the application starts, two thread pools are initialized with no work on them, at this point the program has three threads, one running the application and two other threads idling. Whenever the main thread reaches the call of the 3.2 sub-process, half of the work is appended to one idling thread pool and half to another. The time taken to append the work is very small so at this point two threads are moving cars and the main

one is free to start working on the traffic lights update. After the 2000s simulation timeframe ends, the program waits for the pools to be idling and then delete their instances.

In the Go implementation, worker threads are already available as a package. Two worker threads are initialized at the beginning of the program execution and both are linked to a data channel.



Figures 3 (left) and 4(right) Execution flow for multi-threaded and thread pooling versions of sub-process 3.2

Whenever 3.2 subroutine is called, two jobs are appended to the channel, each job containing half of the work of the subroutine. The worker threads then automatically read this channel and split the job between them. The only difference here is that there is not a clear delegation of which worker thread will do each job, as opposed to the C implementation where in each append it is clear which one is doing what. This automatic delegation in Go depends on how overloaded each worker is, so it is a little bit smarter than the thread pool used in C, being more suitable in cases where many more threads and delegations are needed. Just like in C, at the end of the simulation the program waits for the workers to finish their jobs and then the channel and workers are deleted from the memory. Task 3.2.3 from Figure 4 created an opportunity for more test scenarios, depending on the kind of multi-threading synchronization that is developed:
1: After appending the work from 3.2.1 and 3.2.2, the program would wait for both to end their work before proceeding. This method was given the name of Hard Sync.
2: When the program reaches 3.2.3 it would wait for the work completion of 3.21. and 3.2.2 only once every 10 times. Being then called Soft Sync.
3: All the work appended to any of the worker threads/thread pools would work independently, hence it would provide a higher throughput. This method was called No Sync.

From the implementation standpoint, all are very easy to code since either Go and C applications provide information about how much work is left on each thread. This is represented in Figure 4 by the information arrows named "Work completion information". Both projects were made available online [Martini 2018].

7. Data Analisys & Metrics

The time analysis over all test cases already explained over the previous sections was made on a single computer, for consistency. A 64-bit, dual core Athlon processor running at 3.01GHz with 8Gb of RAM memory was used. The application ran under the Windows 10 operating system with only the very essential operation tasks running on the background, the executable file was called on the operating system with the highest priority possible so that there would be little influence of the task scheduler over execution time.

Also, 10 runs for every simulation case were made so that any non-determinism could be captured. Average time, deviation and median value of the 10 runs are provided to clarify when a time-variation happens. Time to code the software in both languages and total line count is also provided for a development analysis.

For both applications the time spent to run the 2000s simulation was calculated the same way: the program itself would open up a timer counter that starts on task 1 and stops on task 6. It was considered that, since both languages use the CPU clock count to measure time and that the simulation time is reasonably short, this method would provide information precise enough for a comparison. Constants were used on both languages to have a common output that would translate processor clock count to millisecond. CBCL control and IC control, explained in section 3.2, are compared only in the single tread control mode for simplicity.

8. Results

Table 2 shows the first results from the simulation runs in milliseconds. It is possible to see that CBCL control takes more time to run on both languages since all block occupation percentages have to be calculated from time to time so that traffic lights can shorten their red time whenever blocks start getting full of cars. On IC control, block occupation is calculated only once at the end of the simulation only to display this information on the terminal.

Standard deviation values and median values from the 10 simulation runs show that the application is deterministic in all cases. For IC control, Go was 8.1% slower and for CBLC control, 5.8% than the C version. Interestingly, the increase in complexity added 13% of time on the C application, but only 10.6% to the Go one.

Tables 2 (left) and 3 (right). IC/CBLC control and compiling results, respectively.

	C			Go		
	Average	*Std. Deviation*	*Median*	*Average*	*Std. Deviation*	*Median*
IC - Single Thread [ms]	1836	41	1826	1985	50	1985
CBCL - Single Thread [ms]	2076	28	2074	2196	54	2176

	C	Go
Development time [h]	30	30
Total line count	1112	1060
Executable size	117KB	2.38MB

On table 3 some development and compiling information is shown. Development time for the C version, which was the first version to be coded, summed 30 hours. Since the Go version was just a translation of what was already coded in C, it was expected to have a shorter development time, but a learning curve time caused some overhead to the

Go total time given the author's little experience with the language, indicating that Go might be more productive than C, but this affirmation cares for more study. Line count was very similar on both implementations, Go has a lot of syntactic sugar embedded on it to make the code more concise, but as C is already a very concise language, the difference ended up being small.

Executable size ended being very different between both applications but this didn't become a constraint for the development, the end result is shown just as a standard output of the compilers in their "as is" configurations. It is also noticeable that in order to target ROM-constrained architectures, the Go compiler would need to be properly set. In some applications, Go ROM usage can reduce up to seven times when compared to its standard configuration compilation [Pike 2018].

Table 4 shows the main differences in the multi-threading contexts. Standard deviations are acceptable in all scenarios, but a little bit higher than expected on "CBCL – Multi-thread pool – No sync" for the Go language. Median and Average values are close to each other, meaning that there were no outlier values spoiling the average value. The difference column shows how much slower the average of Go runs were in comparison to the C ones.

Table 4. C and Go comparison – Multi-Threading

	C			Go			
	Average	*Std. Deviation*	*Median*	*Average*	*Std. Deviation*	*Median*	*Difference*
CBCL - Single thread [ms]	**2076**	28	2074	2196	54	2176	5.77%
CBCL - Multi thread [ms]	3840	141	3831	**1590**	65	1596	-58.61%
CBCL - Multi thread pool - Hard Sync [ms]	**1597**	59	1597	1756	97	1739	9.94%
CBCL - Multi thread pool - Soft Sync [ms]	**1388**	52	1363	1633	60	1647	17.61%
CBCL - Multi thread pool - No Sync [ms]	**1075**	40	1075	1560	117	1533	45.09%

"CBCL – Multi thread" line shows the overhead caused by thread creation and destruction in every loop for the C implementation, in the Go implementation, the use of "go func_name" feature in every call of the 3.2 sub-routine did not cause any unexpected overheads, making the Go implementation run 58% faster. On the other hand, all other simulation scenarios executed in less time on the C version. For both languages *Hard Sync* was slower than *Soft Sync*, which was slower than *No Sync*, with bigger gains on C than on Go. Since the Go version used a lot of manual RAM mapping and static allocation, its garbage collector was not very much used, so this work doesn't exploit how much slower the application would run by using its automatic memory management, being left as a suggestion for future study.

9. Conclusion

Worker threads in Go and thread pooling in C do not work exactly the same way, but the final result is comparable and both are suited for multi-threading applications. By comparing the average value of the runs, it is possible to notice that C is indeed faster than Go, but not to a point where it becomes a burden to most of the applications. Since Go is becoming a much more productive environment due to its many packages available on online repositories, C would only be a choice over Go when execution time

is a big constraint, or when low-level resources are still needed to control hardware functions, like on hard-time embedded systems and other time-critical applications.

References

Rojas, Raúl; et al (2000). “Plankalkül: The First High-Level Programming Language and its Implementation”, in Institut frame Informatik, Freie Universität Berlin, Technical Report B-3/2000, ftp://ftp.mi.fu-berlin.de/pub/reports/TR-B-00-03.pdf, February.

TIOBE (2018). “Index programming languages”, https://www.tiobe.com/tiobe-index/, June.

Rouse, Jeff (2017). “Why go is skyrocketing in popularity”, in Opensource.com, open article, https://opensource.com/article/17/11/why-go-grows, November.

Pike, Rob (2012). “Go at Google: Language Design in the Service of Software Engineering”, https://talks.golang.org/2012/splash.article, October.

El-Seoud, Samir Abou; et al (2017). “Big Data and Cloud Computing: Trends and Challenges”, in International Journal of Interactive Mobile Technologies, November.

Van Roy, Peter; Haridi, Seif (2004). “Concepts, Techniques, and Models of Computer Programming”, in the MIT Press.

Pike, Rob; Ken, Thompson (2018). “The Go Programming Specification”, https://golang.org/ref/spec#Assignability, February.

Lämmel. Ralf; Jones, Simon Peyton (2003). “Scrap your boilerplate: a practical design pattern for generic programming”, in Proceedings of the 2003 ACM SIGPLAN TLDI’03, January.

Kernighan, Brian W.; Ritchie, Dennis (1988). “C programming language” in Prentice Hall, March.

Pike, Rob (2014). “Hello Gophers” in Gophercon Opening Keynote, https://talks.golang.org/2014/hellogophers.slide#1, February.

Clark, K.L.; McCabe, F.G. (2004). “Go! – A Multi-paradigm Programming Language for Implementing Multi-threaded Agents” in Annals of Mathematics and Artificial Intelligence, v41, p171-206, August.

Renaux, Douglas; Linhares, Robson R.; Simão, Jean M.; Stadzisz, Paulo C. (2014). “CTA Simulator”, http://www.dainf.ct.utfpr.edu.br/~douglas/CTA_CONOPS.pdf, June.

Lalouche, Gal. (2016). “Spartan Programming”, https://webcourse.cs.technion.ac.il/236700/Spring2016/ho/WCFiles/06-Spartan%20Programming.pdf.

Seferidis, Johan Hanseen (2017). “C-Thread-Pool”, https://github.com/Pithikos/C-Thread-Pool, April.

Martini, Guilherme (2018). “C and Go CTA” https://github.com/ghkmartini/CandGoCTA , July.