Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Statistical Agent Based Modelling Approach (SABM) Towards Complex Financial and Economic Systems: Implementation in Go

Bachelor Thesis

Steven Battilana

Wednesday 8th November, 2017

Supervisors: Dr. Zhang Qunzhi, Zhuli He, Dr. Paolo Penna

Professors: Prof. Dr. Sornette, Prof. Dr. Widmayer

Department of Computer Science &
Department of Management, Technology and Economics,
ETH Zürich

Abstract

Statistical agent based model (SABM) aims at reverse-engineering financial markets. In order to do so, more than ten thousand of market strategies are sampled to represent the agents within the market. The SABM is very computationally intensive, especially the backtesting function within it, as identified by Professor Sornette's research group. Thus, the backtesting function needs to see some time cutting measures in order to reduce the execution time of SABM.

The main task of this thesis was to translate the backtesting function from Python to Go in order to profit from the speed of Go. In addition, after the naïve version of the backtesting function was written in Go, the implementation was analysed to find the lines that needed optimisation. After localising the most time-consuming functions using CPU profiling, the identified functions were optimised, tested, and benchmarked until an optimised version of the backtesting function was reached.

This optimised version realised a speed-up of a factor $2.9\times$ in comparison to the vectorised backtesting function in Python, that is to say it only uses 34.63% of the vectorised version's execution time.

Acknowledgements

I would like to thank my supervisor Zhuli He for his support and Vera Baumgartner for proofreading my thesis so thoroughly.

Contents

Chapter 1

Introduction

For some time, it has been a trend among specialists to increasingly regard economic systems as complex systems. Complex systems lie in between the traditional economic theory and econometrics. The traditional economic theory assumes that we only have rational agents, whereas Econometrics tries to find statistical correlation between the data to explain observed market behaviour. A complex system defines simple rules for the agents and attempts to aggregate a macro state out of all micro state interactions. This is the reason why it is not feasible to find a mathematical formula that describes the macro level with a few parameters without understanding how the macro level arises from the micro level. Here, agent based modelling (ABM) comes into play. It designs a virtual economic system with computer agents which in turn should simulate the behaviour from the real agents. As long as the artificial computer agents cover the main characteristics from the real agents, ABM should be able to reproduce the stylized facts of the real world [1] [2] [3].

Statistical agent based modelling aims at reverse-engineering financial markets. In order to do so, more than ten thousand of market strategies are sampled to represent the agents within the market. To construct the so-called meta-state of the market, the detailed information of each sampled market strategy at each time step is required. Using this approach is very difficult because of the computational complexity when doing out-of-sample predictions [4]. In addition to the computational complexity, the implementation of the statistical agent based modelling (SABM) was done in Python, which does not necessarily give the best performance among the available programming languages. In the SABM Python scripts, the before mentioned detailed information is retrieved by the so-called backtesting function. Backtesting is the process where a trading strategy gets tested on historical data to see how it performs and to analyse the profitability and risk of an agent [39]. The backtesting function is very computationally intensive, which limits the ability of SABM to sample more market strategies. This limitation might eventually reduce the predictive power of SABM. Thus we need to perform some time cutting measures to the backtesting function in order to make SABM fast and accurate.

Professor Sornette's research group came up with the idea to re-implement the backtesting function of SABM in another, yet, more efficient language, namely Go. It is a rather newly developed language based on C and designed by Google and it has already gained a reputation as being very fast. The goal is to exploit the performance gains of Go compared to Python.

The main task of this thesis was therefore to translate the backtesting function from Python into Go in order to profit from the overall higher performance of Go. In addition, after the naïve version was written, the implementation was analysed to find the subroutines that needed optimisation. This optimisation pattern was applied repeatedly until a satisfying speed-up was reached.

In Chapter 2 Methodology, the main characteristics of both Python and Go are presented. In the case of Go, this also includes some language specific highlights which come in handy in the subsequent chapter. In Chapter 3 Experiments, the Python implementation of the backtesting function is introduced and then its implementation in Go. Subsequently, the optimisation approach is outlined accompanied by the corresponding timing results. In Chapter 4, a conclusion of the results found in the thesis is given which also outlines some possibilities of future improvements and steps that might lie ahead. The longer listings of Chapters 2 and 3 were moved to the appendix. All expressions marked with a $^+$ are explained in the glossary which is also to be found in the appendix.

Chapter 2

Methodology

In the following subchapters, the two languages Python and Go are briefly described and then compared according to the benchmarks from [21]. Furthermore, the features of Go which are crucial for this thesis are discussed.

2.1 Python

Python was released in the year 1991 as an open source language. As it is a dynamic typed language, it supports multiple programming paradigms such as object oriented$^+$, imperative$^+$, functional$^+$ and procedural programming$^+$. Furthermore, there are a myriad of libraries available for Python as well as interfaces to many system calls. Altogether, Python is easily accessible and a very attractive programming language to work with.
Even though Python seems to be a totally interpreted language at first glance, it is actually also a rather compiled language at the second. However, Python is a lot more interpreted than Go, which is a completely compiled language [18] [19]:(What is Python?). For further details the reader should feel free to consult [23].

A program written in Python often runs slower than an equivalent implementation in C or C++. This is due to the fact that it is a higher level language compared to C and C++ and that it is highly dynamic and partly interpreted. [20]

2.2 Go

Go was released in 2012 as an open source project by Google. The introduction of multicore processors, networked systems, massive computation clusters, and the web programming model slowed down the software development at Google significantly and made it clumsier. All the well-known

languages, as for example Python and C or C++, were not equipped to suddenly be used on multicore processors and by various developers simultaneously via the web. For these languages, new libraries were released to adapt to these changes but none of them was able to do so completely satisfactory. Generally speaking, Go was developed to address exactly these problems and to introduce a language which works well on multicores and clusters from the beginning.
Furthermore, Google produced with Go a language which enhances the time used to compile a program immensely. Generally, today's server programs potentially contain millions of lines of code which take up a lot of time to be compiled ranging from minutes to hours. With Go, programs are compiled almost instantanously [15]:(Abstract, Introduction).

Go is a procedural language with pointers, which is compiled. Furthermore, it is concurrent, garbage-collected, statically typed and it includes a testing package which enables code profiling, testing and benchmarking [17]. Go is efficient, scalable and productive while bringing together the performance and security from C and the speed of working with a dynamic language like Python [16]. The goal of the dynamic part of Go is to create a working environment which is arguably even more productive than Python's. Additionally, it should be easy to learn, at least for those who are familiar with the C family [15]:(Enter Go).

Semantics in Go differ sligthly from the ones in C. According to [15]:(Semantics), the main differences are the following:

- It has no pointer arithmetic;
- there are no implicit numeric conversions;
- array bounds are always checked;
- there are no type aliases;
- ++ and -- are statements not expressions; and
- assignment is not an expression.

Additionally, there are also some larger adjustments to make when switching from C to Go, since Go includes concurrency and garbage collection. It is important to keep in mind that Go is not completely memory safe when programming concurrently. Further details are given in Subsection 2.6.2.

2.3 Python vs. Go

Setting the main characteristics of Python and Go side to side, we are able to see the key differences and commonalities immediately.

Python vs. Go		
Characteristics	Python	Go
Created in year	1991	2012
Garbage Collector	✓	✓
Built-in concurrency	✗	✓
Scalability [15]:(Concurrency)	(✓)	✓
Dynamic type system	✓	✗
Static type system	✗	✓
Interpreted language	(✓)	✗
Compiled language	(✗)	✓
Emphasises on code readability	✓	✓
Code blocks	indentation	curly brackets
Syntax that allows to express concepts in possibly fewer lines than C/C++	✓	✓
Functions may return multiple values	✓	✓

Table 2.1: Go vs. Python differences and commonalities

Considering Table 2.1, both languages are garbage collected and have functions that may return multiple values. Futhermore, they both emphasise on code readability and thus their syntax potentially allows for shorter codes than C or C++. Even though both claim to be scalable, Go generally seems to scale a bit better as for example the code blocks are indicated with curly brackets instead of indentations, which increases the chances of finding mistakes by the naked eye [15]:(Pain points). In contrast, Python is a dynamic typed and rather interpreted language [19] whereas Go is statically typed and a complied language. Additionally, Go provides built-in concurrency [15], which increases the efficiency significantly when used.

2.4 Benchmarking Python and Go

This section gives a quick overview on how Go compares to Python when running an equivalent program. The results are from 'The Computer Language Benchmarks Game' and there the benchmarks were run on a quad-core 2.4Ghz Intel® Q6600® with 4GB of RAM and 250GB SATA II disk drive; using Ubuntu™ 17.10 Linux x64 4.13.0-16-generic [22]. Subsequent the results are summarised in Table 2.2 and Figures 2.1 and 2.2.

Except for the regex-redux program, Python takes between 173% and 5111% of the time Go needs to run the corresponding program. Thus, Go is up to 51.11× faster than Python! Therefore, the hopes of Professor Sornette's research group to obtain substancial increase in speed when implementing SABM in Go are definitely justified. As already mentioned in Section 2.1, Python's rather slow performance can be explained by the fact that Python is a higher-level, dynamic and interpreted language compared to Go which is very close to C and C++.

Go versus Python 3 [21]			
Benchmark tasks	Go timing	Python timing	Go vs. Python
fasta	2.17s	110.91s	5111%
mandelbrot	5.48s	273.43s	4990%
spektral-norm	3.94s	188.83s	4793%
n-body	21.47s	787.02s	3666%
fannkuch-redux	14.44s	483.79s	3350%
k-nucleotide	14.98s	84.73s	566%
reverse-complement	0.54s	2.82s	522%
binary-trees	34.42s	86.90s	252%
pidigits	2.03s	3.51s	173%
regex-redux	28.49s	14.86s	52%

Table 2.2: Go vs. Python timing results by 'The Computer Language Benchmarks Game'

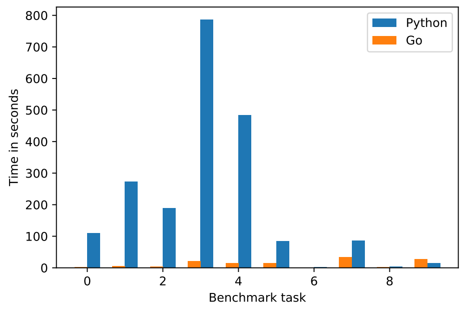

Figure 2.1: Go vs. Python timing results displayed as bars

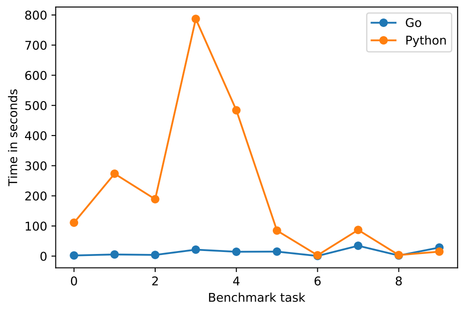

Figure 2.2: Go vs. Python timing results displayed as graphs

2.5 Profiling Go Programs

This chapter examines how to use profiling to optimise Go programs. Furthermore, it explains how to benchmark and test Go functions.

2.5.1 Profiling

One advantage of Go are its built-in profiling tools with which a program can be analysed and bottlenecks can be identified and addressed. Thus, using these tools, the program can be improved and its running time decreased. To enable profiling however, the code has to be adjusted as shown

in Listing A.1.
First, the libraries needed for profiling have to be loaded, in particular, `flag` and `runtime/pprof` as done on the first few lines of Listing A.1. Second, line 13 has to be added right above the main function. Third, lines 18 to 28 have to be inserted in the beginning of the main function. What these changes actually generate is explained in a Go post as following:

> 'The new code defines a flag named `cpuprofile`, calls the Go flag library to parse the command line flags, and then, if the `cpuprofile` flag has been set on the command line, starts CPU profiling redirected to that file. The profiler requires a final call to `StopCPUProfile` to flush any pending writes to the file before the program exits; we use `defer` to make sure this happens as `main` returns.' [24]

In order to run the CPU file in Listing A.1, a makefile as in Listing 2.1 is used, which takes care of all the steps needed and sets the correct flags. The neat thing about a makefile is, that as soon as all parts of the program are saved, one only needs to type `make <keyword>` into the shell for everything to run smoothly. For the makefile in Listing 2.1 the `keyword` is one of either `main_cpu_profiling`, `run_main`, `run_cpu_profiling`, or `main_cpu_profiling_run`. Omitting the keyword altogether just runs `all`.
When `all` is run, in lines 3 to 4, the directive `go build` compiles the specified program `CPU_profiling.go` and returns a executable called `main_cpu_profiling`. As a next step, in lines 6 to 7, the directive `./main_cpu_profiling` runs the executable obtained before.
Lines 10 to 13 are then responsible for the CPU profiling. First, `main_cpu_profiling` is compiled again in line 10 to make sure any possible changes are included. Subsequently, the directive `main_cpu_profiling` is run, which invokes the code on the lines 12 to 13. Line 13 then runs the modified main function whilst doing the CPU profiling and saving the data into the file `main_cpu_profiling.prof`.
During profiling, the Go program stops roughly 100 times per second to record a sample of the current state in order to count how long a particular function is running [24].

Listing 2.1: CPU profiling makefile

```
1  all: main_cpu_profiling run_main
2
3  main_cpu_profiling: CPU_profiling.go
4    go build -o $@ $^
5
6  run_main:
7    ./main_cpu_profiling
8
```

```
# cpu profiling
run_cpu_profiling: main_cpu_profiling main_cpu_
    profiling_run

main_cpu_profiling_run:
  ./main_cpu_profiling -cpuprofile main_cpu_profiling
      .prof
```

After running the makefile, it is time to call
`go tool pprof main_cpu_profiling main_cpu_profiling.prof`
from the shell to start up the performance analysis tool and interpret the profile. Important commands for doing so are `topN` and `web`. The first command displays the top `N` samples with regard as to how many times the counter stopped while these functions were in action. Thus, the functions at the top of this ranking were taking up the majority of the execution time. The latter command writes a graph of the profile data in SVG format and opens it in a web browser [24], but note that this necessarily requires graphviz. To plot the graph as a PDF use `pdf` instead of `web`, an example of which is shown in Figure 2.3 (refer to this link for a larger example[1]). Every box represents a single function and its size corresponds to the number of samples of the particular function counted during the profiling. An edge from box A to B corresponds to A calling B. The colors range from red for boxes and edges which were called the most over orange to grey for the ones called the least. Edges that were used a lot are also marked by being wider. Using this intuitive graph to spot the time intensive functions is fast and simple. Hence, it helps to set the optimisation target in an efficient way.

2.5.2 Benchmarks [25] [27]

Benchmarks are very useful to measure the performance of a function in Go and they are a good way to track any performance improvement after an optimisation attempt. Thus a concise overview on how to write and run a benchmark is given in the following paragraphs.
Note that according to Dave Cheney, modern CPU rely heavily on active thermal management which can add noise to benchmark results [25].

One method to write a benchmark is to first create a file with an arbitrary name, for example the same name as the file containing the functions which are to be benchmarked. Then, very importantly `_test` has to be added to the name. Concretely, consider the file of the function to be benchmarked called `zurich.go`, then a suitable name for the file containing the benchmarks would be `zurich_test.go`.
A benchmark function always has to start with `Benchmark`, otherwise the

[1] https://battilanablog.files.wordpress.com/2017/11/cpu_profile.pdf

testing driver will not recognise it as such. The test driver runs the benchmarks several times, every time increasing `b.N` until the driver is satisfied with the stability of the benchmark. All benchmarks contain a `for` loop which runs the function `b.N` times. An example of an actual benchmark running for various inputs is given below. In order to enable the benchmark function to run on different inputs to the original program, a helper function can be introduced. This helper function allows for different input values to be fed to the program without having to hard code every single one of them.

This process is easier to understand when seen on a concrete example. Thus, consider the Listings A.2 and A.3. There, the function `benchmark_concurrent_binary_slice` is declared on line 24 and takes `b *testing.B`, `num_goroutines int`, `input []float64`, and `fn concurrent_binary_slice` as arguments. Note that the function name starts with a lower case `b`, hence it is a function which is only visible in the `main` package. The `for` loop mentioned above is stated on lines 26 to 28. The variables on line 23, 25 and the assignment on line 27 are needed to avoid elimination during compiler optimisations.

In the Listing A.3 on line 11, the function `BenchmarkAnnualVolFromDailyReturns_concurrent_1` is declared, which is the benchmark function run by the driver. This can be seen as it is written with a capital `B` and satisfies the requested signature by only having `b *testing.B` as an argument. The function `annualVolFromDailyReturns` is run with different `num_goroutines` on lines 11 to 21.

While being in the same directory as the benchmark files are saved, run the following command in the shell to invoke the benchmark functions from above: `go test -bench=.`
The `-bench=<function_name>` flag passes a chosen benchmark function to the driver and if `.` is used instead of `<function_name>`, all valid functions in the benchmark file are passed. The output produced is shown in Listing 2.2, where the results from the benchmark functions are displayed on lines 4 to 8. On these lines, the second entries show how many times the loop body in Listing A.2 was executed. The third entries display the arithmetic mean over the `b.N` runs per function call. Line 4 states that the mean execution time of the function `BenchmarkAnnual...Returns_serial` is 6133749ns (6.1ms) on the machine named in 3.0.1 (i).

Listing 2.2: Output of go test −bench=.

```
1  $ go test -bench=.
2  goos: darwin
3  goarch: amd64
4  BenchmarkAnnual...Returns_serial-4    200     6133749  ns/op
5  BenchmarkAnnual...Returns_conc_1-4    200     6119399  ns/op
6  BenchmarkAnnual...Returns_conc_4-4    500     2719709  ns/op
7  BenchmarkAnnual...Returns_conc_8-4    1000    2768626  ns/op
```

```
BenchmarkAnnual...Returns_con_32-4      500     2376696 ns/op
PASS
ok      _/Users/battilanast/.../Benchmarking    32.330s
```

2.5.3 Tests [27]

Testing functions in Go is quite similar to benchmarking with just a few additional tweaks.
Equivalently to producing benchmarks, the name of the file has to end with `_test.go`, otherwise the testing driver will not be able to recognise the testing function. Using again `zurich_test.go` as an example, a test unit is written by choosing a function name which starts with `Test` and has only one calling argument, namely `t *testing.T`. Otherwise, the compiler throws an error. To increase usability, the test unit was split up into a function body in the template file and the actual test file. Note that all the function names in the template start with lower case `t`, thus they are only visible within the `main` package. In comparison, the function names in the test file start with an upper case `T`. In the end, the unit test gets called by the driver and runs the specified tests, as for example seen in Listing A.5 on lines 8 to 10.

As done similarly in Subsection 2.5.2, run the following command in the shell to invoke the test functions from above when being in the same directory as the test files are saved: `go test -bench=.`
As immediately obvious, this directive is the same as the one used for benchmarks. However, Subsection 2.5.2 failed to mention that apart from benchmarks, it also runs all test functions which then results in an output as given in Listing 2.2. There, the testing result is summarised on line 9 with `PASS`, as all tests have finished successfully. If this was not the case, the shell would enlist an output as specified by the programmer. For instance, in Listing A.4, the output in case of failure is specified on lines 22 and 25.

2.6 Concurrency

Go provides built-in concurrency in the form of goroutines (light-weight processes$^+$) [15]:(Abstract) [28]:(Concurrency/Goroutines). However, before fully analysing concurrency in Go, it is important to make sure the difference of concurrency and parallelism is well understood.

2.6.1 Concurrency vs. Parallelism

Definition 1 (Concurrrency) [29]
A program which is a composition of independently executing light-weight processes.

Definition 2 (Parallelism) [29]
A program which executes simultaneous (possibly related) computations on different CPUs.

It is imoprtant to note that concurrency is about *dealing* with lots of things at once in one process, that is to say multiple light-weight processes running on one CPU. In comparison, parallelism is about *doing* lots of things at once with multiple kernel threads running at once on different CPUs. In other words, concurrency describes the structure of a code and parallelism is a certain form of execution [29].

2.6.2 Concurrency in Go

This subsection aims at explaining how Go handles concurrency by first describing how to share data in a concurrent environment and then introducing goroutines. Thirdly, channels$^+$ are looked at in more detail and last but not least, parallelisation in Go is explained.

Concurrent Communication

In most languages, it is difficult to correctly access shared data without causing race conditions$^+$. Go takes another approach than many by exchanging shared data over channels$^+$ to make sure that different light-weight processes$^+$ cannot access the data at the same time. When sticking to channels$^+$, data races$^+$ are by design not allowed to occur [28]:(Concurrency/Share by communicating). The Go developers reduced the idea behind this design to the following slogan, which turns up in every corner of the web when searching for Go and concurrent:

> 'Do not communicate by sharing memory; instead, share memory by communicating.' [32] [15]:(Concurrency) [28]:(Concurrency/Share by communicating)

Programmers have to use channels$^+$ with caution, since not every task is served best using them. Go comes with a built-in library called `sync`, which also provides mutex$^+$. A mutex$^+$ is best used for small problems like for instance increasing the reference count, where using channels$^+$ would be less efficient. Clearly, the high-level approach using channels$^+$ makes developing correct concurrent programs easier, since it inherently manages data access, that is to say there is no synchronisation needed. This approach was inspired by Hoare's Communicating Sequential Processes (CSP)$^+$ [28]:(Concurrency/Share by communicating).

Goroutines

Google chose the name *goroutine* because already existing terms with a similar meaning have been used inconsistently throughout literature and are thus not clearly defined.

Definition 3 *(Goroutine) [28]:(Concurrency/Goroutines)*
A function executing concurrently with other goroutines in the same address space.

A goroutine is a light-weight process$^+$, which costs a bit more than the allocation of stack space. The initial stacks are small and when more space is needed, it is allocated (or freed) on the heap as required.
The goroutines are multiplexed onto multiple kernel threads such that when one routine blocks because it is waiting, another can be scheduled and run. To spawn a new goroutine within a program, just prefix a function or method call with the `go` keyword. When the forked function or method completes, the goroutine exits silently [28].

Channels

To enable goroutines to exchange information, channels$^+$ are used to send and receive data. There are two types of these channels$^+$, buffered and unbuffered (or *synchronous*) ones. When initialising a buffered channel$^+$, a capacity number `N` is defined, which determines the size of the buffer. When a buffered channel$^+$ is used for communication of two goroutines, the sending one is able to send as many as `N` data points which then wait in the buffer until the receiving goroutine drains them. Thus, the sending goroutine is not blocked and can continue with execution. If the buffer is full however, the sending goroutine has to wait until a space is freed up on the buffer and then, it therefore blocks further execution for some time. An unbuffered channel$^+$ is the special case of a buffered channel$^+$ with capacity number `N=0`. When using an unbuffered channel$^+$, a sending goroutine therefore always blocks until the receiving one is ready. Receiving goroutines always block until there is an element to drain available on the channel$^+$, whether it is a buffered or an unbuffered one.
Unbuffered channels$^+$ are useful to guarantee that two goroutines are in a known state as either one has to wait for the other when exchanging information. This fact can be used by exchanging values insignificant to the respective computations only to signal that the goroutines have reached a certain state.

There are three way to initialise channels$^+$, two for an unbuffered channel$^+$ and one for a buffered one. On the first line in Listing 2.3, an unbuffered channel$^+$ for the exchange of integer values is allocated. Since an unbuffered channel$^+$ is just a buffered one with capacity zero, it can also be initialised as shown on the second line. When choosing to use a buffered channel$^+$ of capacity larger than 0, it has to be initialised as shown on line 3 [28]:(Concurrency/Channels).

Listing 2.3: Examples of channel initialisations

```
channel1 := make(chan int)                    // unbuffered channel of intergers
channel2 := make(chan int, 0)                 // unbuffered channel of intergers
channel3 := make(chan int, 13)                // buffered channel of intergers
```

To illustrate the behaviour of channels$^+$ in more detail, an implementation of a semaphore$^+$ with channels$^+$ is shown in Listing 2.4. When choosing the capacity of a channel$^+$ to be 1, a mutex$^+$ is obtained. Hence, channels$^+$ can also be used for synchronisation. The limit a semaphore$^+$ puts on the throughput of data is here realised by limiting the number of running goroutines, that is to say the size of the buffer of the channel$^+$, to `N`. Every additional goroutine blocks when calling `acquire(1)` until there is a spot left in the semaphore$^+$ (i.e. a free spot in the buffered channel$^+$) which happens when a running goroutine calls `release(1)`.

Listing 2.4: Semaphores implemented with channels [7]

```
type empty struct{}
type semaphore chan empty

sem = make(semaphore, N)    // N = buffer size

// acquire n resources
func (s semaphore) acquire(n int) {
  e := empty{}
  for i := 0; i < n; i++ {
    s <- e
  }
}

// release n resources
func (s semaphore) release(n int) {
  for i := 0; i < n; i++ {
    <-s
  }
}
```

Parallelisation

When the computation at hand can be parallelised, start as many goroutines as the number of available CPU cores to do so. For instance, split up a `for` loop into chunks and assign a chunk to every goroutine [28]. Then, the

goroutines are multiplexed onto kernel threads and since there are exactly the same number of goroutines as cores, every core should get exactly one kernel thread to run. Unfortunately, it is not easy to force Go to use all available cores since there are always some processes running in the background. Even tough [34] suggest there might be a workaround, the Go designers seem not to have intended to provide one easily.
Looking at the parallelisation example from [28] in Listing 2.5, it launches exactly `numCPU` goroutines and divides the `for` accordingly. The buffered channel+ `c` with buffer size `numCPU` is used as a wait synchronisation in the second `for` loop [28]:(Concurrency/Parallelisation).

Listing 2.5: Example for a parallel implementation [28]:(Concurrency/Parallelization)

```go
import "runtime"

numCPU := runtime.NumCPU() // available number of CPU
    cores

func (v Vector) DoAll(u Vector) {
    c := make(chan int, numCPU)  // Buffering
       optional but sensible.
    for i := 0; i < numCPU; i++ {
        go v.DoSome(i*len(v)/numCPU, (i+1)*len(v)/
           numCPU, u, c)
    }
    // Drain the channel, i.e. wait till a goroutines
        finished
    for i := 0; i < numCPU; i++ {
        <-c    // wait for one task to complete
    }
    // All done.
}
```

2.6.3 Different parallelisation paradigms

This section briefly describes the two (or three) main approaches taken whilst optimising during the experiment as described in the next chapter.

For illustration, consider the function `rolling max` as shown in Listing 3.1, which was tested during the optimisation process. The serial `rolling max` takes as arguments a `window_size` and an `input` slice+ in which it looks for the maximum. It does so by sliding a window of size `window_size` from left to right over the input slice+ and continuously filling the output slice+ with the found maximum in the respective window. The first `window_size-1` entries in `output` are filled with zeroes by definition.

Listing 2.6: Serial rolling max function

```
func rolling_max(input []float64, window_size int) []
    float64 {
  var output = make([]float64, len(input))
  if len(input) > 0 {
    for i := range input {
      if i-(window_size-1) >= 0 {
        output[i] = max_in_slice(input[i-(window_size
            -1) : i+1])
      }
    }
  } else { // empty input
    fmt.Println("rolling_max is panicking!")
    panic(fmt.Sprintf("%v", input))
  }
  return output
}
```

Splitting up Loops

In principle, it uses the same approach as the parallelisation in Listing 2.5. However, there are some details that have to be taken care of concerning possible false sharing.

Consider Listing A.6 where we admit false sharing. Adopting the approach from Listing 2.5, the concurrent function splits the `for` loop into the same number of chunks as there are goroutines available. On lines 14 to 21 the new `for` loop boundaries for each goroutine are being computed. In the `for` loop on line 23, the goroutine with `id` computes only its own chunk and saves the result into `output`. Note that as long as all the goroutines run concurrently, this will not pose any problem, but when running parallel, false sharing could occur. The `barrier_wg.Wait()` on line 28 synchronises the function such that it only returns when all goroutines are finished.

To address the false sharing issue in Listing A.6, the function gets adjusted in the following way: Every goroutine computes the rolling max in its chunk and saves the result into the local slice$^+$ `loc_max` instead of directly into the `output`. Then, every goroutine saves its `loc_max` into the slice$^+$ `go_max[go_id]` which was originally initialised on lines 7 to 10. Finally, the results saved in `go_max` are aggregated into `output`.

Master Worker Paradigm

The master worker paradigm is a tasking model where the master splits the `for` loop into tasks, which the worker pool executes one after the other until there are none left to do.

The concurrent version of rolling max in Listing A.8 uses the master worker paradigm. The computation of the maximum in a certain window in the `for` loop forms its own function here. This new rolling max function contains three main building blocks. First, it forks a goroutine in which new tasks are created in each iteration and loaded into the channel$^+$ `pending`. The task `NewWork` takes a function, an input and output slice$^+$, and the first and last indices of the window in the input slice$^+$ as arguments. Since there is no synchronisation, the main goroutine continues after spawning the new goroutine on line 23 and starts executing the `for` loop on line 29. This `for` loop starts the worker pool, of which each take the channels$^+$ `pending` and `done` as input arguments. Hidden behind the function `Worker` lie goroutines which drain in the forever `for` loop one `Work` element from the channel$^+$ `pending`. Then they start the computation and when they are finished, the altered `Work` element is sent to the channel$^+$ `done` to signal that one task has been finished. On line 37, the function waits until all tasks have been executed.

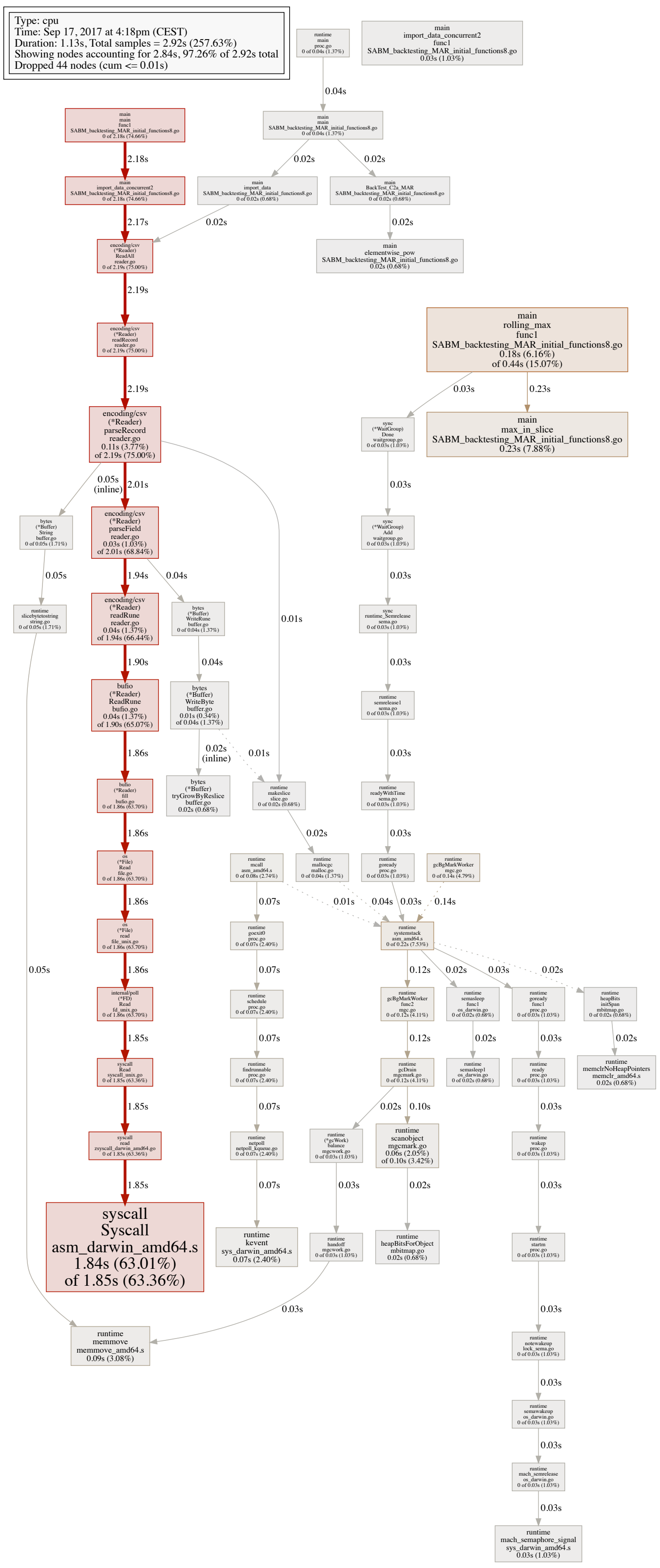
Type: cpu
Time: Sep 17, 2017 at 4:18pm (CEST)
Duration: 1.13s, Total samples = 2.92s (257.63%)
Showing nodes accounting for 2.84s, 97.26% of 2.92s total
Dropped 44 nodes (cum <= 0.01s)
main import_data_concurrent2 func1 SABM_backtesting_MAR_initial_functions8.go 0.03s (1.03%)
main rolling_max func1 SABM_backtesting_MAR_initial_functions8.go 0.18s (6.16%) of 0.44s (15.07%)
main max_in_slice SABM_backtesting_MAR_initial_functions8.go 0.23s (7.88%)
encoding/csv (*Reader) parseRecord reader.go 0.11s (3.77%) of 2.19s (75.00%)
encoding/csv (*Reader) parseField reader.go 0.03s (1.03%) of 2.01s (68.84%)
encoding/csv (*Reader) readRune reader.go 0.04s (1.37%) of 1.94s (66.44%)
bufio (*Reader) ReadRune bufio.go 0.04s (1.37%) of 1.90s (65.07%)
syscall Syscall asm_darwin_amd64.s 1.84s (63.01%) of 1.85s (63.36%)
runtime memmove memmove_amd64.s 0.09s (3.08%)
runtime kevent sys_darwin_amd64.s 0.07s (2.40%)
runtime scanobject mgcmark.go 0.06s (2.05%) of 0.10s (3.42%)
runtime heapBitsForObject mbitmap.go 0.02s (0.68%)
runtime memclrNoHeapPointers memclr_amd64.s 0.02s (0.68%)
runtime mach_semaphore_signal sys_darwin_amd64.s 0.03s (1.03%)

Figure 2.3: Go program profile

Chapter 3

Experiments

Explaining the main task of this thesis is split up into the following blocks: First, the original Python code is looked at in detail, particularly `BackTest_C2a_MAR`. This discussion includes both the loop based and the vectorised implementation, and compares their running times. Secondly, the workings of the initial naïve translation into Go are shown, including initial testing and timing. Thirdly, the CPU profiling results are discussed and the most time consuming functions are highlighted. Fourthly, the optimisation part is explained which includes the rewriting of the most time consuming function as well as discussing testing and timing. Lastly, the results are summarised by displaying the new improved CPU profiling plot and examining the time improvement.
The data used for the backtesting function is the historic data from S&P 500^{+}.

3.0.1 Configuration

In this subsection, the configuration used for this thesis is described by first including the machines used and then the versions of Python and Go, respectively.

Used Machine

The optimisations were done on the second machine and all the timings were produced on the first one. To put the results into perspective, both configurations are enlisted below:

(i) MacBook Pro (13-inch, 2017, Four Thunderbolt 3 Ports), 3.5 GHz Intel Core i7, 16 GB 2133 MHz LPDDR3, macOS Sierra Version 10.12.6

(ii) MacBook Pro (13-inch, Early 2011), 2.7 GHz Intel Core i7, 16 GB 1333 MHz DDR3, macOS Sierra Version 10.12.6

Python Version

The timing was run on the following python version.

```
$ python --version
Python 3.6.2 :: Anaconda, Inc.
```

Go Version

The current version as displayed below has been used throughout the experiment.

```
$ go version
go version go1.9 darwin/amd64
```

3.1 Implementation in Python

Given was a tested and correctly running Python implementation. Subsequently, this code is looked at and briefly described as to what happens in both, the loop based and the vectorised implementation.

3.1.1 Backtesting in Python

The `BackTest_C2a_MAR` function is based on the moving average ratio and takes eight arguments. Among these arguments are a fast and a slow window, an entry and an exit level, profit, loss exit, and a prices and a returns array. All these are processed and turned into a hash map `res` in which several outputs are saved, such as the trading returns, the trading states, trading profit and loss, trading drawdowns, hold profit and loss, trading annualised returns, trading trailing max drawdown, fast and slow moving average, and indicator (MAR ratio).

The loop based backtesting is divided into three stages.

(i) First up is the computation of the fast and slow moving average, `fastMA` and `slowMA`, as well as the MAR ratio, which is here named `indicator`. Next, the arrays (to be precise: the pandas series) for the second stage are initialised and prepared. This already concludes the first stage.

(ii) In the second stage, the trading returns, trading states, trading profit and loss, trading drawdowns and the hold profit and loss are computed. The trading returns, trading states and profit and loss current hold are computed while iterating through the entire time length. That is every day, it is computed in which of the following states the agent is: idle, to enter, hold or to exit.

(iii) Then, the backtesting results for different backtesting lengths are computed, in particular the trading annualised returns and the trading trailing max drawdown.

These three stages describe all there is to the loop based backtesting function in Listing A.9.

The vectorised backtesting function using the pandas library's vectorised functions works basically in the same way (see Listing A.10). The only difference lies in the fact that it is vectorised and therefore much more efficient compared to using simple loops as above. In the timing results below, it is shown how much of a difference in efficiency there is.

3.1.2 Timing in Python

For the following timing results of the Python implementation of the backtesting function, the computer as described in 3.0.1 (i) was used and the timings of 10 runs were averaged (see listing A.11).

Timing in Python			
Backtesting	averaged in ms	std. dev. in ms	pct.
Python loop based	2742.3350	47.406406	100%
Python vectorised	23.7179	4.562519	0.86%

Table 3.1: Timing of the Python implementation

Hence, vectorisation has already bought a speed-up of staggering 115×.

3.2 Implementation in Go

Finally, the main part of the thesis is discussed, namely the translation from Python to Go and the subsequent optimisations. This chapter explains the naïve Go implementation and its timing results. Then, using the profiling tools, it is shown in which order functions should be targeted to optimise. In each optimisation step, unit tests were written to ensure correctness. Afterwards, a timing round was run using benchmarks to show the improvement. This approach was repeated until there were no significant improvements to make time-wise.

3.2.1 First Naïve Implementation in Go

The naïve Go implementation of the backtesting function basically mimics the loop based version implemented in Python. To be precise, only serial functions were used. Rather quickly, it turned out that a large number of library functions given in Python were not to be found for Go and had to be implemented one by one. After having done so, the correctness of the implementation was checked by comparing the output with the one from the Python implementation.

In the following, the timing results of the naïve implementation are run on 3.0.1 (i).

Timing in Python and Go				
Backtesting	averaged in ms	std. dev. in ms	pct.	pct.
Python loop based	2742.3350	47.406406	100%	
Python vectorised	23.7179	4.562519	0.86%	100%
Go serial	21.12	2.054707235	0.77%	89.05%

Table 3.2: Timing of the Python and Go implementation

Keep in mind that only a naïve serial implementation in Go was used, which already yields an improvement factor of roughly 1.1× compared to the vectorised Python version.

3.2.2 CPU Profiling of the Naïve Implementation in Go

When examining the profiling results and the output graph in Figure 3.1 (see here for a larger version[1]), it is clearly detectable which functions are inefficient by means of using a larger portion of the running time.

From the profiling graph, it is quite straight forward to realise that the backtesting actually uses a significant portion of the execution time, namely 59.77%, in `max_in_slice`. This suggests that `max_in_slice` should be optimised first. However, it is not efficiently parallelisable, but the calling function `rolling_max` can call it concurrently and indeed this is how the optimisation was approached.

3.2.3 Optimisation of `rolling_max`

The `rolling_max` function in Listing 3.1 returns a slice$^+$ containing the moving maximum (rolling max) of the input slice$^+$. It essentially loops through the entire input slice$^+$ moving a window from left to right and computing the maximum within this window in each iteration.

Listing 3.1: Serial implementation of the `rolling_max` function

```go
func rolling_max_serial(input []float64, window_size
    int) []float64 {
  var output = make([]float64, len(input))
  if len(input) > 0 {
```

[1] https://battilanablog.files.wordpress.com/2017/11/cpuprofile01.pdf

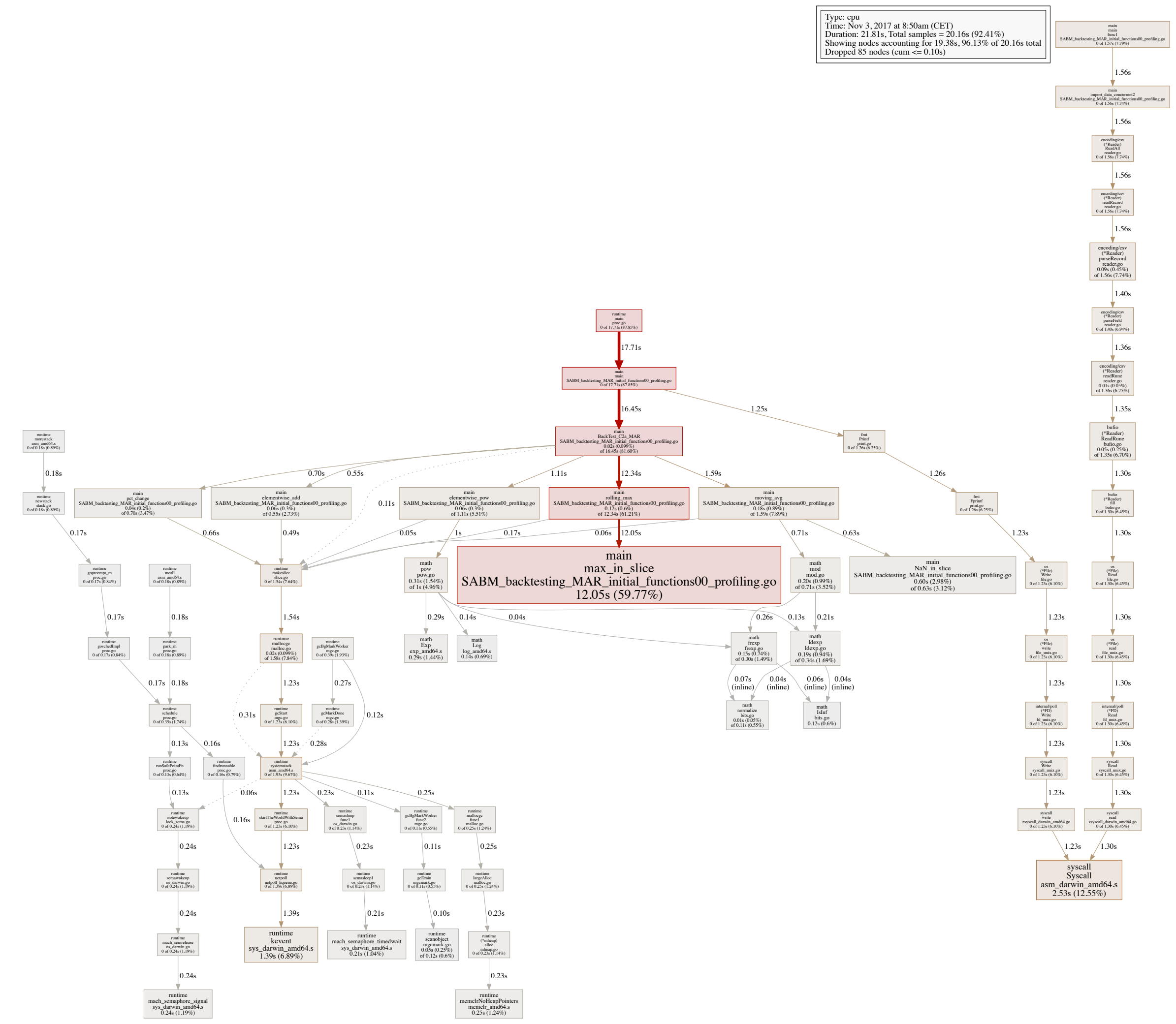

Figure 3.1: CPU profile of the naïve implementation in Go

```
   for i := 0; i < window_size; i++ {
    output[i] = 0.0
   }
   for i := range input {
    if i-(window_size-1) >= 0 {
     output[i] = max_in_slice(input[i-(window_size
       -1) : i+1])
    }
   }
  } else { // empty input
   fmt.Println("rolling_max is panicking!")
```

```
    panic(fmt.Sprintf("%v", input))
  }
  return output
}
```

Optimisation 1: Divide and Conquer

Opting for dividing and conquer in the same manner as explained in Subsection 2.6.3, that is splitting up the input slice+ for the different goroutines, which each have a slot in `go_max` to save their result. In the end, this new now concurrent function in Listing A.12 aggregates `go_max` to form the `output` slice+. Note that this function also mitigates possible false sharing+ issues.

Optimisation 2: Divide and Conquer

This optimisation is almost the same as the first one, the only difference being that it admits possible false sharing+ in the hope of saving time by directly saving the result into the `output` as shown in Listing A.13.

Optimisation 3: Master Worker Paradigm

This optimisation uses the master worker paradigm in which the master loads all tasks into the `pending` channel. From there, the worker pool, consisting of several goroutines, drains task by task until there is nothing left to do. For illustration, consider Listing A.14. In this case, a task corresponds to the computation of one window which is done in the `compute_window_max` function. In other words, it is an adopted version of the in 2.6.3 presented master worker paradigm.

3.2.4 Tests on `rolling_max`

Next, it is important to check whether the optimised `rolling_max` functions return the same output as the serial version. To do so, `rolling_max` is checked for correctness by the test functions in Listing A.15. After running these tests, the following message in Listing 3.2 is output, where *PASS* implies that all tests ran successfully. Otherwise, an error message would have been printed.

Listing 3.2: Results after checking `rolling_max`

```
$ go test
PASS
ok    _/Users/battilanast/.../Benchmarking  0.148s
```

3.2.5 Benchmarks

In order to determine which optimisation yields the best performance, benchmarks have to be run on them. These benchmarks are created and run as explained in Subsection 2.5.2. The benchmark results are displayed in Listing 3.3 with `window_size = 261x5` and `num_goroutines = a`, where $\mathtt{a} \in \{1, 4, 8, 32\}$.

Listing 3.3: Results after benchmarking `rolling_max`

```
1  $ go test -bench=.
2  goos: darwin
3  goarch: amd64
4  BenchRMax_ser_261x5        200    6915321 ns/op
5  BenchRMax_ser2_261x5       200    7166906 ns/op
6  BenchRMax_con_261x5_1      200    6913682 ns/op
7  BenchRMax_con_261x5_4      500    3729162 ns/op
8  BenchRMax_con_261x5_8      500    3634100 ns/op
9  BenchRMax_con_261x5_32     500    3739790 ns/op
10 BenchRMax_con2_261x5_1     200    6767008 ns/op
11 BenchRMax_con2_261x5_4     500    3570324 ns/op
12 BenchRMax_con2_261x5_8     500    3461646 ns/op
13 BenchRMax_con2_261x5_32    500    3452789 ns/op
14 BenchRMax_con3_261x5_1     100   11408390 ns/op
15 BenchRMax_con3_261x5_4     100   10295883 ns/op
16 BenchRMax_con3_261x5_8     200    7705298 ns/op
17 BenchRMax_con3_261x5_32    300    6165166 ns/op
18 PASS
19 ok    _/Users/battilanast/.../Benchmarking  173.560s
```

As the number of goroutines should best not blow-up in order to control the size of the multiplexing overhead, the most efficient configuration is the one displayed on line 12. There, the number of goroutines is `num_goroutines=8` and the resulting execution time equals `3461646 ns/op`. Thus, the best optimisation is obtained when optimising `rolling_max` with optimisation 2 of Subsection 3.2.3 and using 8 gorutines. Finally, the backtesting function and its helper function are adjusted such that these results are mirrored in their performance.

3.2.6 Timing After First Optimisation in Go

After plugging in the optimised `rolling_max` into the backtesting function, the timing has to evaluated anew. In the following Table 3.3 the timing results are enlisted.

Timing in Python and Go					
Backtesting	averaged in ms	std. dev. in ms	pct.	pct.	pct.
Python loop based	2742.3350	47.406406	100%		
Python vectorised	23.7179	4.562519	0.86%	100%	
Go serial	21.12	2.054707235	0.77%	89.05%	100%
Go opt. rolling max	14.172	1.246583319	0.52%	59.75%	67.10%

Table 3.3: Timing of the Python and Go implementation after the first optimisation

According to Table 3.3, the backtesting now only uses 59.75% of the running time used by the vectorised approach in Python which is an improvement by a factor of about 1.7×.

3.2.7 CPU Profiling of the First Optimisation in Go

Looking at the CPU profiling in Figure 3.2 after running the backtesting with the optimised `rolling_max` function, it is observable that the portion of the execution time spent in `max_in_slice` has now reduced from 59.77% to 36.88%. (A larger version of Figure 3.2 is available on [2])

Subsequently, the optimisation steps described in Subsections 3.2.3 to 3.2.7 are repeated with all the other functions which use a lot of the execution time of the backtesting function.

3.2.8 Timing of `BackTest_C2a_MAR`

After very the time consuming optimisation of all time intensive functions, it is time to benchmark the entire `BackTest_C2a_MAR` function. The results thereof are enlisted in Table 3.4.

Timing in Python and Go					
Backtesting	averaged in ms	std. dev. in ms	pct.	pct.	pct.
Python loop based	2742.3350	47.406406	100%		
Python vectorised	23.7179	4.562519	0.86%	100%	
Go serial	21.12	2.054707235	0.77%	89.05%	100%
Go opt. rolling max	14.172	1.246583319	0.52%	59.75%	67.10%
Go concurrent	8.214	1.045608577	0.30%	34.63%	38.89%

Table 3.4: Timing of the Python and Go implementation after the optimisation

Table 3.4 shows that the concurrent backtesting in Go only takes 34.63% of the time of the vectorised version in Python which is a speed-up factor of 2.9×.

[2] https://battilanablog.files.wordpress.com/2017/11/cpuprofile02.pdf

When examining the profiling results of the fully optimised `BackTest_C2a_MAR` in Figure 3.3, it is noticeable that the total duration came down from 14.93s to 9.22s. In addition, when looking at the `rolling_max`, the percentage of the total execution time rose from 36.88% to 37.33%, however, the time spent within it decreased from 9.96s to 8.34s. (A larger version of Figure 3.3 is available on [3])

The complete and fully optimised code is available on [4].

3.3 Outlook

The next step is the integration of the optimised function into Python, in other words, make the Go function directly callable from Python. This integration is already in the working but, unfortunately, it could not be finished before the deadline of this thesis. I will continue working with the Chair to finish the integration in the near future. Additionally, I am planning on benchmarking my implemented helper functions against the gonum floats library[5] for possible further optimisation of the backtesting function. Then, the research group of Professor Sornette's could swap the old and time intensive backtesting function implemented in Python with the optimised function implemented in Go in order to profit from the speed-up without rewriting the entire Python codebase.

[3]https://battilanablog.files.wordpress.com/2017/11/cpuprofile03.pdf
[4]https://battilana.uk/bachelor-thesis/
[5]https://github.com/gonum/gonum/tree/master/floats

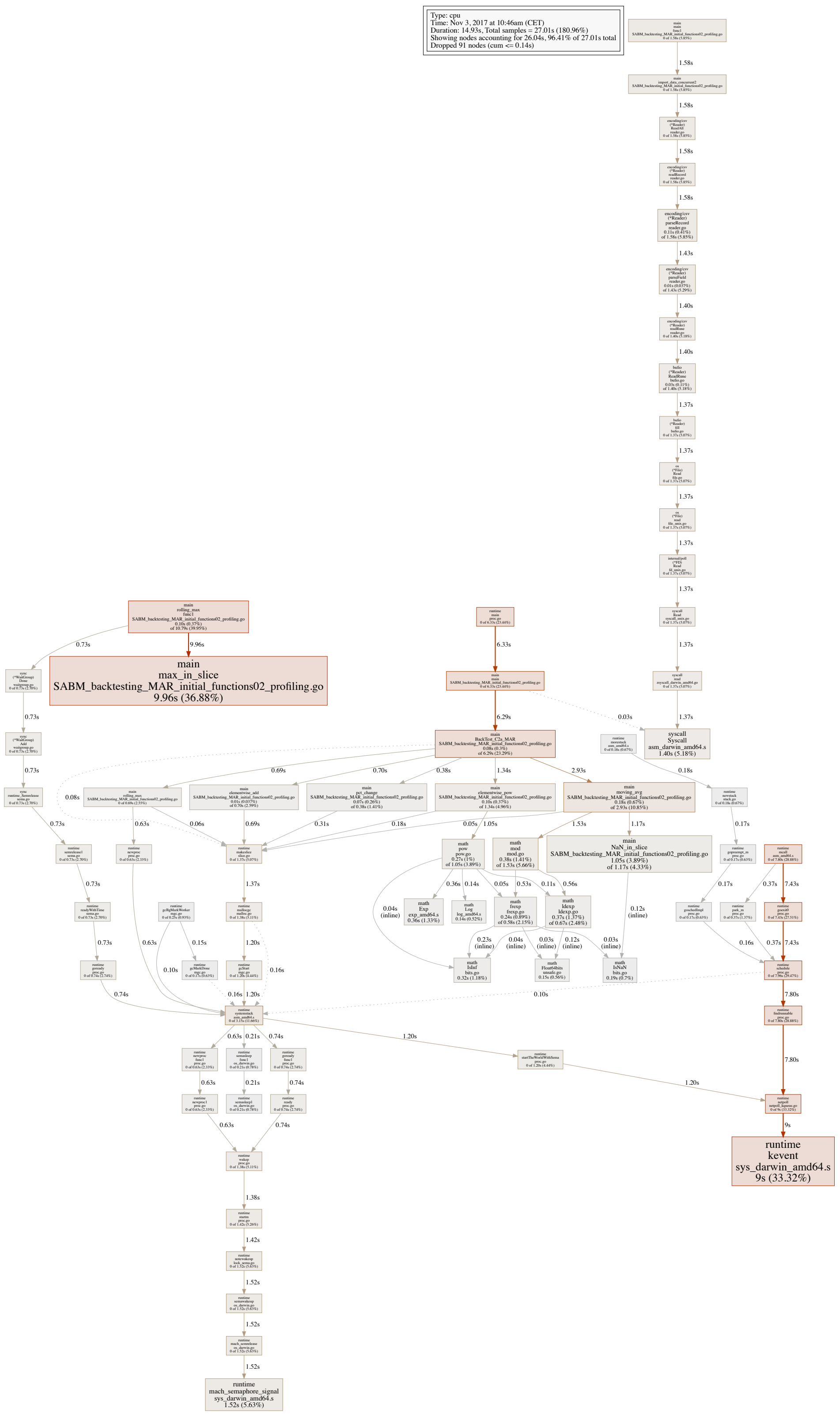
Type: cpu
Time: Nov 3, 2017 at 10:46am (CET)
Duration: 14.93s, Total samples = 27.01s (180.96%)
Showing nodes accounting for 26.04s, 96.41% of 27.01s total
Dropped 91 nodes (cum <= 0.14s)
main max_in_slice SABM_backtesting_MAR_initial_functions02_profiling.go 9.96s (36.88%)
runtime kevent sys_darwin_amd64.s 9s (33.32%)
syscall Syscall asm_darwin_amd64.s 1.40s (5.18%)
runtime mach_semaphore_signal sys_darwin_amd64.s 1.52s (5.63%)

Figure 3.2: CPU profile with the optimised `rolling_max` function in Go

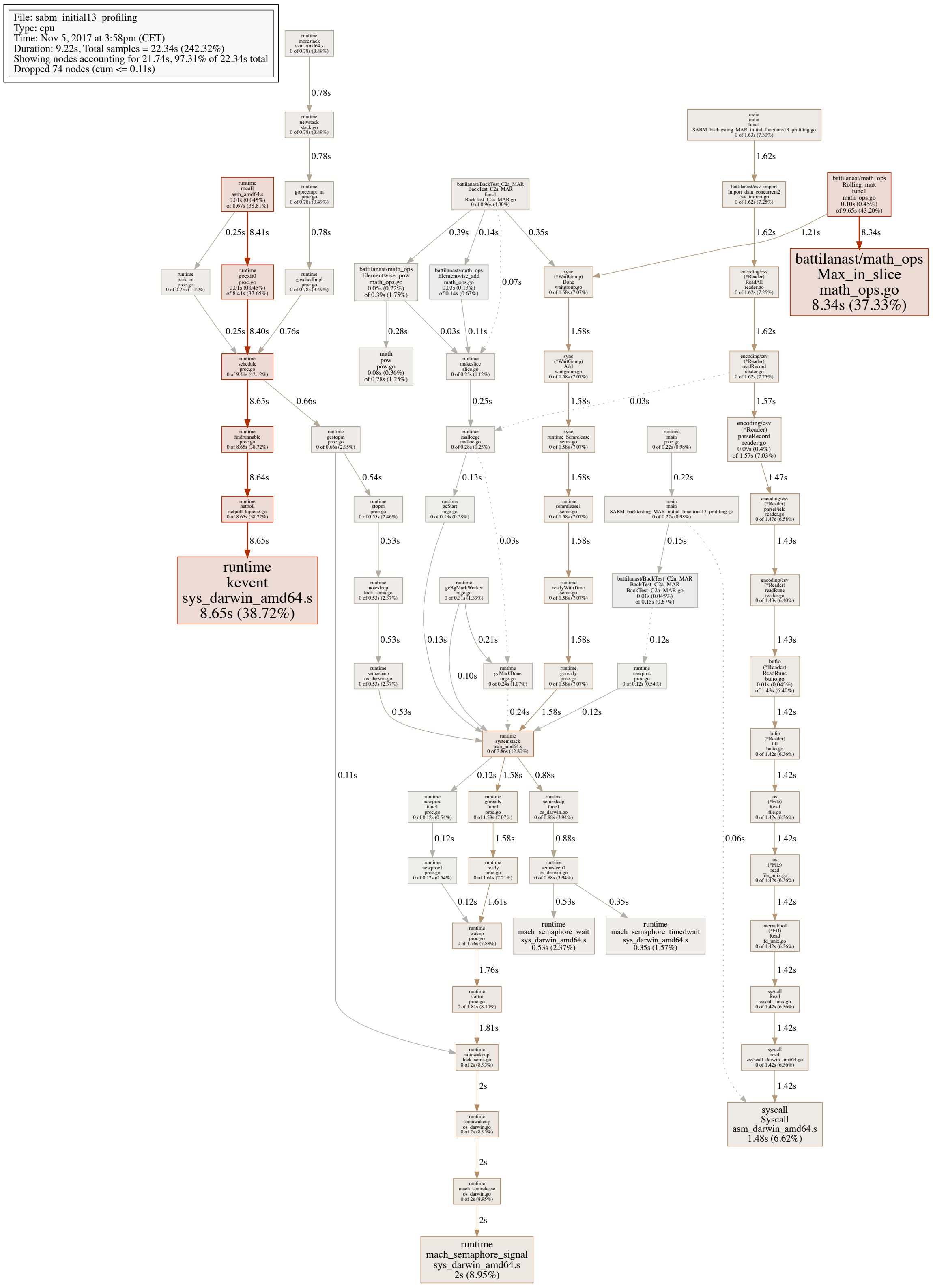
File: sabm_initial13_profiling
Type: cpu
Time: Nov 5, 2017 at 3:58pm (CET)
Duration: 9.22s, Total samples = 22.34s (242.32%)
Showing nodes accounting for 21.74s, 97.31% of 22.34s total
Dropped 74 nodes (cum <= 0.11s)
runtime kevent sys_darwin_amd64.s 8.65s (38.72%)
battilanast/math_ops Max_in_slice math_ops.go 8.34s (37.33%)
syscall Syscall asm_darwin_amd64.s 1.48s (6.62%)
runtime mach_semaphore_signal sys_darwin_amd64.s 2s (8.95%)
runtime mach_semaphore_wait sys_darwin_amd64.s 0.53s (2.37%)
runtime mach_semaphore_timedwait sys_darwin_amd64.s 0.35s (1.57%)
math pow pow.go 0.08s (0.36%) of 0.28s (1.25%)
encoding/csv (*Reader) parseRecord reader.go 0.09s (0.4%) of 1.57s (7.03%)

Figure 3.3: CPU profile with the fully optimised `BackTest_C2a_MAR` function in Go

Chapter 4

Conclusion

First, the loop based backtesting function in Python was translated to a naïve loop based implementation in Go. Astoundingly, this naïve loop based implementation is already as fast as the vectorised version of the backtesting function in Python. To be precise, it is even about two milliseconds faster. After localising the most time consuming function using CPU profiling, the identified function was optimised, tested and benchmarked. Reiterating these steps, namely CPU profiling, optimisation, testing and benchmarking, an optimised version of the backtesting function was reached. This version runs 2.9× faster than the vectorised backtesting in Python, that is to say it only uses 34.63% of the vectorised version's execution time.

However, there is still room for improvement of the backtesting function. For instance, the number of goroutines used in each function within the backtesting function could be tweaked further, such that it runs the fastest in its intended environment. This environments is likely a cluster with many more cores available than the machine used and described in Subsection 3.0.1 (i). Additionally, a timing test with randomly generated data should be conducted, instead of only using the historic data from the S&P 500 index. Doing so, the speed up projected in this thesis could be confirmed. Plus, introducing table based benchmarking could increase code readability immensely.

In summary, the implementation and optimisation of the backtesting function in Go achieves a substantial speed-up.

Appendix A

Appendix

A.1 Glossary

Channel 'is a model for interprocess communication and synchronization via message passing. A message may be sent over a channel, and another process or thread is able to receive messages sent over a channel it has a reference to, as a stream. Different implementations of channels may be buffered or not, and either synchronous or asynchronous.' [13]

Communicating Sequential Processes '(CSP) is a formal language for describing patterns of interaction in concurrent systems. It is [...] based on message passing via channels.' [33]

Critical Section 'refers to an interval of time during which a thread of execution accesses a shared resource, such as shared memory.' [30]

Data Race confer race condition below.

False Sharing 'is a performance-degrading usage pattern that can arise in systems with distributed, coherent caches at the size of the smallest resource block managed by the caching mechanism. When a system participant attempts to periodically access data that will never be altered by another party, but that data shares a cache block with data that is altered, the caching protocol may force the first participant to reload the whole unit despite a lack of logical necessity. The caching system is unaware of activity within this block and forces the first participant to bear the caching system overhead required by true shared access of a resource.' [36]

Functional programming 'is a programming paradigm [...] that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions or declarations instead of statements. In functional code, the output value of a function depends only on the arguments that are passed to the function, so calling a function f twice with the same value for an argument x produces the same result f(x) each time; this is in contrast to procedures depending on a local or global state, which may produce different results at different times when called with the same arguments but a different program state. Eliminating side effects, i.e., changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavio[u]r of a program, which is one of the key motivations for the development of functional programming.' [12]

Imperative programming 'is a programming paradigm that uses statements that change a program's state. In much the same way that the imperative mood in natural languages expresses commands, an imperative program consists of commands for the computer to perform. Imperative programming focuses on describing how a program operates.' [11]

Interpreted language 'is a programming language for which most of its implementations execute instructions directly, without previously compiling a program into machine-language instructions. The interpreter executes the program directly, translating each statement into a sequence of one or more subroutines already compiled into machine code.
The terms interpreted language and compiled language are not well defined because, in theory, any programming language can be either interpreted or compiled.' [18]

Light-weight process (LWP) 'is a means of achieving multitasking. In the traditional meaning of the term [...] a LWP runs in user space on top of a single kernel thread and shares its address space and system resources with other LWPs within the same process.' Sometimes LWP are also called user-level threads which are implemented directly on top of a kernel thread. [14]

Master/Worker '(or sometimes also Master/slave) is a model of communication where one device or process has unidirectional control over one or more other devices. In some systems a master is selected from a group of eligible devices, with the other devices acting in the role of slaves.' [35]

Multiplexer 'is a device that selects one of several analog or digital input signals and forwards the selected input into a single line.' [30]

Mutual Exclusion (also Mutex) 'is a property of concurrency control, which is instituted for the purpose of preventing race conditions$^+$; it is the requirement that one thread of execution never enter its critical section$^+$ at the same time that another concurrent thread of execution enters its own critical section$^+$.' [30]

Object-oriented programming '(OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self"). [...] There is significant diversity of OOP languages, but the most popular ones are class-based, meaning that objects are instances of classes, which typically also determine their type.' [9]

Procedural programming 'is a programming paradigm [...] based upon the concept of the procedure call. Procedures, also known as routines, subroutines, or functions [...], [which] simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including by other procedures or itself. [...] Procedural programming languages are also imperative languages'. Examples of procedural programming languages are C and Go. [10]

Race Condition (sometimes called data race) 'is the behaviour of an electronic, software, or other system where the output is dependent on the sequence or timing of other uncontrollable events. It becomes a bug when events do not happen in the order the programmer intended. The term originates with the idea of two signals racing each other to influence the output first.' [31]

Semaphore 'is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system such as a multiprogramming operating system.' [40]

Slices	'Go's slice type provides a convenient and efficient means of working with sequences of typed data. Slices are analogous to arrays in other languages [...]' [37]
S&P 500	'The Standard & Poor's 500, often abbreviated as the S&P 500, or just the S&P, is an American stock market index based on the market capitalisations of 500 large companies having common stock listed on the NYSE or NASDAQ.' [38]

A.2 Listings of Chapter 2 Methodology

Listing A.1: CPU profiling

```
package main

import (
  ...
  "flag"           // profiling
  "runtime/pprof" // profiling
)

// functions up here
...

//################ PROFILING ################
var cpuprofile = flag.String("cpuprofile", "", "write
    cpu profile to file")
//################ PROFILING end ############

func main() {
    //############ PROFILING ###############
  flag.Parse()
  if *cpuprofile != "" {
    f, err := os.Create(*cpuprofile)
    if err != nil {
      log.Fatal("could not create CPU profile: ", err
          )
    }
    if err := pprof.StartCPUProfile(f); err != nil {
      log.Fatal("could not start CPU profile: ", err)
    }
    defer pprof.StopCPUProfile()
  }
    //############ PROFILING end ###########
    ...
    // code here
    ...
}
```

Listing A.2: Example of a benchmark template

```
package main

import (
  "testing"
```

```
    "fmt"
)

// some type declaration for function for brevity
...


//########### BENCHMARK TEMPLATES ###########
// as above but for unary functions
var res_serial_unary_slice []float64
func benchmark_serial_unary_slice(b *testing.B, input
     []float64, fn serial_unary_slice) {
  var r []float64
  for n := 0; n < b.N; n++ {
    r = fn(input)
  }
  res_serial_unary_slice = r
}
// as above but for binary functions
var res_concurrent_binary_slice []float64
func benchmark_concurrent_binary_slice(b *testing.B,
    num_goroutines int, input []float64, fn concurrent
    _binary_slice) {
  var r []float64
  for n := 0; n < b.N; n++ {
    r = fn(input, num_goroutines)
  }
  res_concurrent_binary_slice = r
}
//######### BENCHMARK TEMPLATES end #########
```

Listing A.3: Example of a benchmark file containing benchmarks functions

```
package main

import (
  "testing"
)

//############### BENCHMARKS ###############
func BenchmarkAnnualVolFromDailyReturns_serial(b *
    testing.B) {
    benchmark_serial_unary_slice(b, returns,
       annualVolFromDailyReturns)
}
```

```
func BenchmarkAnnualVolFromDailyReturns_concurrent_1(b *testing.B) {
    benchmark_concurrent_binary_slice(b, 1, returns, annualVolFromDailyReturns_concurrent)
}
func BenchmarkAnnualVolFromDailyReturns_concurrent_4(b *testing.B) {
    benchmark_concurrent_binary_slice(b, 4, returns, annualVolFromDailyReturns_concurrent)
}
func BenchmarkAnnualVolFromDailyReturns_concurrent_8(b *testing.B) {
    benchmark_concurrent_binary_slice(b, 8, returns, annualVolFromDailyReturns_concurrent)
}
func BenchmarkAnnualVolFromDailyReturns_concurrent_32(b *testing.B) {
    benchmark_concurrent_binary_slice(b, 32, returns, annualVolFromDailyReturns_concurrent)
//############### BENCHMARKS end ###########
```

Listing A.4: Example of a test template

```
package main

import (
  "testing"
    "fmt"
)

// some type declaration for function for brevity
...


//############# TEST TEMPLATES ##############
// as above but now testing a concurrent function with binary input which returns a slice for correctness
func test_serial_concurrent_binary_slice(t *testing.T, input []float64, num_goroutines int, fn_serial serial_unary_slice, fn_concurrent concurrent_binary_slice) {
  var fw_serial, fw_concurrent, sw_serial, sw_concurrent []float64
  fw_serial = fn_serial(returns)
  sw_serial = fn_serial(returns)
  fw_concurrent = fn_concurrent(returns, num_
     goroutines)
  sw_concurrent = fn_concurrent(returns, num_
     goroutines)

  if !is_equal(fw_serial, fw_concurrent, accuracy_
     bench) {
        fmt.Printf("serial concurrent: fw_serial: %v
           fw concurrent:  t.Errorf("serial concurrent:  fw
           serial not equal fw concurrent!")
  }
  if !is_equal(sw_serial, sw_concurrent, accuracy_
     bench) {
        fmt.Printf("serial concurrent: sw_serial: %v
           sw concurrent:  t.Errorf("serial concurrent:  sw
           serial not equal sw concurrent!")
  }
}
//############# TEST TEMPLATES end ##########
```

Listing A.5: Example of a benchmark file containing test functions

```
package main

import (
  "testing"
)

//############### TESTS ####################
func TestAnnualVolFromDailyReturns_concurrent(t *
   testing.T) {
    test_serial_concurrent_binary_slice(t, returns,
       4, annualVolFromDailyReturns,
       annualVolFromDailyReturns_concurrent)
}
func TestAnnualVolFromDailyReturns_concurrent11(t *
   testing.T) {
    test_serial_concurrent_binary_slice(t, returns,
       4, annualVolFromDailyReturns,
       annualVolFromDailyReturns_concurrent11)
}
func TestAnnualVolFromDailyReturns_concurrent2(t *
   testing.T) {
    test_serial_concurrent_binary_slice(t, returns,
        4, annualVolFromDailyReturns,
        annualVolFromDailyReturns_concurrent2)
}
func TestAnnualVolFromDailyReturns_concurrent3(t *
    testing.T) {
    test_serial_concurrent_binary_slice(t, returns,
        4, annualVolFromDailyReturns,
        annualVolFromDailyReturns_concurrent3)
}
//############### TESTS end ################
```

Listing A.6: Concurrent rolling max admitting possible false sharing

```
func rolling_max_concurrent(input []float64, window_
    size, num_goroutines int) []float64 {
  var output = make([]float64, len(input))
  if len(input) > 0 {
    num_items := len(input) - (window_size - 1)
    var barrier_wg sync.WaitGroup
    n := num_items / num_goroutines

    for i := 0; i < num_goroutines; i++ {
      barrier_wg.Add(1)
      go func(go_id int) {
        defer barrier_wg.Done()

        // computing boundaries
        var start, stop int
        start = go_id*int(n) + (window_size - 1) //
            starting index
        // ending index
        if go_id != (num_goroutines - 1) {
          stop = start + n // Ending index
        } else {
          stop = num_items + (window_size - 1) //
              Ending index
        }

        for i := start; i < stop; i++ {
          output[i] = max_in_slice(input[i-(window_
              size-1) : i+1])
        }
      }(i)
    }
    barrier_wg.Wait()

  } else { // empty input
    fmt.Println("rolling_max is panicking!")
    panic(fmt.Sprintf("%v", input))
  }
  return output
}
```

Listing A.7: Concurrent rolling max avoiding false sharing

```
func rolling_max_concurrent_noFalseSharing(input []
   float64, window_size, num_goroutines int) []float
   64 {
  var output = make([]float64, window_size-1, len(
     input))
  if len(input) > 0 {
    num_items := len(input) - (window_size - 1)
    var barrier_wg sync.WaitGroup
    n := num_items / num_goroutines
    go_max := make([][]float64, num_goroutines)
    for i := 0; i < num_goroutines; i++ {
      go_max[i] = make([]float64, 0, num_goroutines)
    }
    for i := 0; i < num_goroutines; i++ {
      barrier_wg.Add(1)
      go func(go_id int) {
        defer barrier_wg.Done()

        // computing boundaries
        var start, stop int
        start = go_id*int(n) + (window_size - 1) //
           starting index
        // ending index
        if go_id != (num_goroutines - 1) {
          stop = start + n // Ending index
        } else {
          stop = num_items + (window_size - 1) //
             Ending index
        }

        loc_max := make([]float64, stop-start)
        idx := 0
        for i := start; i < stop; i++ {
          loc_max[idx] = max_in_slice(input[i-(window
             _size-1) : i+1])
          idx++
        }
        go_max[go_id] = append(go_max[go_id], loc_max
           ...)

      }(i)
    }
    barrier_wg.Wait()

    for i := 0; i < num_goroutines; i++ {
      output = append(output, go_max[i]...)
    }

  } else { // empty input
    fmt.Println("rolling_max is panicking!")
    panic(fmt.Sprintf("%v", input))
  }
  return output
}
```

Listing A.8: Concurrent rolling max function using the master worker paradigm during optimisation attempt using the elements from [8]

```
func Worker(in chan *Work, out chan *Work) {
  for {
    t := <-in
    t.Fn(t.input, t.output, t.start, t.end)
    t.Completed = true
    out <- t
  }
}

func compute_window_max(input, output []float64,
   start, end int) {
    output[end-1] = max_in_slice(input[start:end])
}

func rolling_max_concurrent(input []float64, window_
   size, num_goroutines int) []float64 {
  var output = make([]float64, len(input))
  if len(input) > 0 {
        length := len(input)
        num_items := length - (window_size - 1)
```

```go
    pending := make(chan *Work)
    done := make(chan *Work)

    // load pending with work
    go func() {
      for i:=(window_size - 1); i<length; i++ {
        pending <- NewWork(compute_window_max, input, output, i-(window_size-1), i+1)
      }
    }()

    // start workers
    for i:=0; i<num_goroutines; i++ {
      go func() {
        Worker(pending, done)
      }()
    }

    // wait till all work is done
    for i:=(window_size - 1); i<num_items; i++ {
      <- done
    }

 } else { // empty input
  fmt.Println("rolling_max is panicking!")
  panic(fmt.Sprintf("%v", input))
 }
 return output
}
```

A.3 Listings of Chapter 3 Experiments

Listing A.9: Loop based backtesting implementation in Python

```python
def BackTest_C2a_MAR(fw, sw, entry, exit, pexit,
   lexit, price, returns):
    commissionRate = 0.0025
    fastMA = pd.rolling_mean(price, fw)
    slowMA = pd.rolling_mean(price, sw)
    indicator = fastMA / slowMA # MAR

    price = price["1980":]
    returns = returns["1980":]
    indicator = indicator["1980":]

    timeLen = len(price)
    tradingReturns = pd.Series(0.0, index = price.
       index)
    tradingpnls = pd.Series(0.0, index = price.index)
    pnlCurrentHold = pd.Series(index = price.index)
    tradingStates = pd.Series(0.0, index = price.
       index)
    tradingState_t0 = "IDLE"
    tradingState_t1 = "IDLE"

    holdStartTime = -1
    # 4 Available States, IDLE, TO_ENTER, HOLD, TO_
       EXIT
    for i in range(1, timeLen):
        if tradingState_t0 == "IDLE":
            if (indicator.iloc[i] > entry and
               indicator.iloc[i-1] < entry): # when
               MAR goes through entry_threshold from
               below
                tradingState_t1 = "TO_ENTER" # Update
                    the state for the next time step
                   t+1
                tradingReturns.iloc[i] = -
                   commissionRate # Commission Fee is
                    paid at t0

        if tradingState_t0 == "TO_ENTER":
            holdStartTime = i
            tradingState_t1 = "HOLD"
            tradingReturns.iloc[i] = returns.iloc[i]
            tradingStates.iloc[i] = 1.0

        if tradingState_t0 == "HOLD":
            tradingReturns.iloc[i] = returns.iloc[i]
            tradingStates.iloc[i] = 1.0
            pnlCurrentHold.iloc[i] = (price.iloc[i] - price.iloc[holdStartTime]) / price.iloc[holdStartTime] + 1

            if (indicator.iloc[i] < exit and indicator.iloc[i-1] > exit) or pnlCurrentHold.iloc[i] < lexit or pnlCurrentHold.iloc[i] > pexit:
                # when MAR goer through exit_threshold from above, or when stopProfit, stopLoss reached
                tradingState_t1 = "TO_EXIT"
                tradingReturns.iloc[i] = returns.iloc[i] - commissionRate
                # Commission Fee is paid at t0, also need to include returns at t0, since agent exit market at t1

        if tradingState_t0 == "TO_EXIT":
            tradingState_t1 = "IDLE"



        tradingState_t0 = tradingState_t1 # Update


    res = pd.DataFrame(index = price.index)
    res["Trading Returns"] = tradingReturns
    res["Trading States"] = tradingStates
    res["Trading PNLs"] = pnlsFromReturns(tradingReturns)
    res["Trading DrawDowns"] = drawdownsFromPnls(res["Trading PNLs"])
    res["Hold PNLs"] = pnlCurrentHold
```

```python
    ################### back-testing result for different back-testing length ##########
    ##### This part is vectorized ##########
    btLenNames = ["10y", "5y", "2y", "1y", "6m", "3m"]
    btLens = [261 * 10, 261 * 5, 261 * 2, 261 * 1, 130, 65 ]
    btLensInYear = [10.0, 5.0, 2.0, 1.0, 0.5, 0.25]
    # back-testing window lengths, 10y, 5y, 2y, 1y, 6m ,3m


    for i in range(0, len(btLens)):
        btLenName = btLenNames[i]
        btLen = btLens[i]
        btLenInYear = btLensInYear[i]
        # Annualized Returns
        res["Trading Annualized Returns " + btLenName] = (res["Trading PNLs"].pct_change(btLen).add(1.0) ** (1.0/btLenInYear) - 1.0 )

    for i in range(0, len(btLens)):
        btLenName = btLenNames[i]
        btLen = btLens[i]
        btLenInYear = btLensInYear[i]
        # MaxDrawDowns
        # btLen = Size of the moving window
        res["Trading Trailing MaxDrawDown " + btLenName] = pd.rolling_max(res["Trading DrawDowns"], btLen)


    #####################################
    res["fastMA"] = fastMA
    res["slowMA"] = slowMA
    res["Indicator"] = indicator
    return res
```

Listing A.10: Vectorised backtesting implementation in Python

```python
# VECTORIZATION WITH PANDAS
def BackTest_C2a_MAR_vec(fw, sw, entry, exit, pexit, lexit, price, returns):
    # Vectorized Pandas
    # stopLoss and stopProfit are not considered at current moment

    fastMA = pd.rolling_mean(price, fw)
    slowMA = pd.rolling_mean(price, sw)
    indicator = fastMA / slowMA # MAR

    price = price["1980":]
    returns = returns["1980":]
    indicator = indicator["1980":]

    indicatorShift = indicator.shift(1) # shift the row/column by one

    # some explanation would be nice ...
    tradingSignal = pd.Series(index = price.index)
    tradingSignal[(indicator >= entry) & (indicatorShift < entry)] = 1   # ... ?
    tradingSignal[(indicator <= exit) & (indicatorShift > exit)] = -1    # TO_EXIT?
    tradingSignal = tradingSignal.dropna().diff() / 2.0
    tradingSignal = tradingSignal[tradingSignal != 0]
    tradingSignal = tradingSignal.fillna(1.0)

    tradingStates = pd.Series(index = price.index)
    tradingStates[tradingSignal.index] = tradingSignal
    tradingStatesNotFilled = tradingStates.copy()  # this guy will be needed for adding commission fee to return series
    tradingStates = tradingStates.shift(1).ffill() # When a Signal Occurs, Trading is lagged by one time step
    tradingStates[tradingStates == -1] = 0
    tradingStates = tradingStates.fillna(0)

    tradingReturns = pd.Series(index = price.index)
    tradingReturns[tradingStates == 1] = returns
    tradingReturns[tradingStatesNotFilled ==  1] = -commissionRate
    tradingReturns[tradingStatesNotFilled == -1] = -commissionRate + returns
    tradingReturns = tradingReturns.fillna(0)


    res = pd.DataFrame(index = price.index)
    res["Trading Returns"] = tradingReturns
    res["Trading States"] = tradingStates
    res["Trading PNLs"] = (tradingReturns + 1).
       cumprod()
    res["Trading DrawDowns"] = 1 - res["Trading PNLs
       "].div(res["Trading PNLs"].cummax())
    res["Hold PNLs"] = pnlCurrentHold

    ################### back-testing result for
       different back-testing length ##########
    ##### This part is vectorized ##########
    btLenNames = ["10y", "5y", "2y", "1y", "6m", "3m
       "]
    btLens = [261 * 10, 261 * 5, 261 * 2, 261 * 1,
       130, 65 ]
    btLensInYear = [10.0, 5.0, 2.0, 1.0, 0.5, 0.25]
    # back-testing window lengths, 10y, 5y, 2y, 1y, 6
       m ,3m


    for i in range(0, len(btLens)):
        btLenName = btLenNames[i]
        btLen = btLens[i]
        btLenInYear = btLensInYear[i]
        # Annualized Returns
        res["Trading Annualized Returns " + btLenName
           ] = (res["Trading PNLs"].pct_change(btLen)
           .add(1.0) ** (1.0/btLenInYear) - 1.0 )

    for i in range(0, len(btLens)):
        btLenName = btLenNames[i]
        btLen = btLens[i]
        btLenInYear = btLensInYear[i]
        # MaxDrawDowns
        res["Trading Trailing MaxDrawDown " +
           btLenName] = pd.rolling_max(res["Trading
           DrawDowns"], btLen)


    #####################################
    return res
```

Listing A.11: Backtesting timing implementation

```
########## backgtesting loop based ##########
time_loop_ms = []
for i in range(0,10):
    t0000 = time.time()
    res0 = BackTest_C2a_MAR(fw, sw, entry, exit,
       pexit, lexit, price, returns)
    t0001 = time.time()
    time_loop_ms.append((t0001-t0000)*1000)
    print("%1.0f, loop based:  print("averaged timing
       backtesting loop based:

########## backgtesting vec ##########
time_vec_ms = []
for i in range(0,10):
    t0000 = time.time()
    res1 = BackTest_C2a_MAR_vec(fw, sw, entry, exit,
       pexit, lexit, price, returns)
    t0001 = time.time()
    time_vec_ms.append((t0001-t0000)*1000)
    print("%1.0f, vec based:  print("averaged timing
       backtesting loop based:
```

Listing A.12: Concurrent rolling max with no false sharing

```
func rolling_max_concurrent(input []float64, window_
   size, num_goroutines int) []float64 {
  var output = make([]float64, window_size-1, len(
     input))
  if len(input) > 0 {
    num_items := len(input) - (window_size - 1)
    var barrier_wg sync.WaitGroup
    n := num_items / num_goroutines
    go_max := make([][]float64, num_goroutines)
    for i := 0; i < num_goroutines; i++ {
      go_max[i] = make([]float64, 0, num_goroutines)
    }
    for i := 0; i < num_goroutines; i++ {
      barrier_wg.Add(1)
      go func(go_id int) {
        defer barrier_wg.Done()

        // computing boundaries
        var start, stop int
        start = go_id*int(n) + (window_size - 1) // starting index
        // ending index
        if go_id != (num_goroutines - 1) {
          stop = start + n // Ending index
        } else {
          stop = num_items + (window_size - 1) // Ending index
        }

        loc_max := make([]float64, stop-start)
        idx := 0
        for i := start; i < stop; i++ {
          loc_max[idx] = max_in_slice(input[i-(window_size-1) : i+1])
          idx++
        }
        go_max[go_id] = append(go_max[go_id], loc_max...)

      }(i)
    }
    barrier_wg.Wait()

    for i := 0; i < num_goroutines; i++ {
      output = append(output, go_max[i]...)
    }

  } else { // empty input
    fmt.Println("rolling_max is panicking!")
    panic(fmt.Sprintf("%v", input))
  }
  return output
}
```

Listing A.13: Concurrent rolling max embracing false sharing

```
func rolling_max_concurrent2(input []float64, window_size, num_goroutines int) []float64 {
  var output = make([]float64, len(input))
  if len(input) > 0 {
    num_items := len(input) - (window_size - 1)
    var barrier_wg sync.WaitGroup
    n := num_items / num_goroutines

    for i := 0; i < num_goroutines; i++ {
      barrier_wg.Add(1)
      go func(go_id int) {
        defer barrier_wg.Done()

        // computing boundaries
        var start, stop int
        start = go_id*int(n) + (window_size - 1) // starting index
        // ending index
        if go_id != (num_goroutines - 1) {
          stop = start + n // Ending index
        } else {
          stop = num_items + (window_size - 1) // Ending index
        }

        for i := start; i < stop; i++ {
          output[i] = max_in_slice(input[i-(window_size-1) : i+1])
        }
      }(i)
    }
    barrier_wg.Wait()

  } else { // empty input
    fmt.Println("rolling_max is panicking!")
    panic(fmt.Sprintf("%v", input))
  }
  return output
}
```

Listing A.14: Concurrent rolling max embracing false sharing

```
func compute_window_max(input, output []float64, start, end int) {
    output[end-1] = max_in_slice(input[start:end])
}

func rolling_max_concurrent3(input []float64, window_size, num_goroutines int) []float64 {
  var output = make([]float64, len(input))
  if len(input) > 0 {
        length := len(input)
        num_items := length - (window_size - 1)
```

```
            pending := make(chan *Work)
            done := make(chan *Work)

            // laod pending with work
            go func() {
                for i:=(window_size - 1); i<length; i++ {
                    pending <- NewWork(compute_window_max
                        , input, output, i-(window_size-1)
                        , i+1)
                }
            }()

            // start workers
            for i:=0; i<num_goroutines; i++ {
                go func() {
                    Worker(pending, done)
                }()
            }

            // wait till all work is done
            for i:=(window_size - 1); i<num_items; i++ {
                <- done
            }

    } else { // empty input
      fmt.Println("rolling_max is panicking!")
      panic(fmt.Sprintf("%v", input))
    }
    return output
}
```

Listing A.15: Tests for checking the correctness of `rolling_max`

```
//############### TESTS ###############
func TestSlice_in_max_serial2(t *testing.T) {
    test_serial_serial_slice(t, returns, fw, sw,
        rolling_max_serial, rolling_max_serial2)
}
func TestSlice_in_max_concurrent1(t *testing.T) {
    test_serial_concurrent_slice(t, returns, fw, sw,
        32, rolling_max_serial, rolling_max_concurrent
        )
}
func TestSlice_in_max_concurrent2(t *testing.T) {
    test_serial_concurrent_slice(t, returns, fw, sw,
       32, rolling_max_serial, rolling_max_concurrent
       2)
}
func TestSlice_in_max_concurrent3(t *testing.T) {
    test_serial_concurrent_slice(t, returns, fw, sw,
       32, rolling_max_serial, rolling_max_concurrent
       3)
}
//############### TESTS end ###############
```

Bibliography

[1] Farmer, J Doyne, 2012, Economics needs to treat the economy as a complex system, Complexity Research Initiative for Systemic instabilities (CRISIS).

[2] Sornette, Didier, 2009, Why stock markets crash: critical events in complex financial systems (Princeton University Press).

[3] Sornette, Didier and Zhuli He, Statistical Agent-Based Modelling Approach (SABM) Towards Complex Financial and Economic Systems, Description March 2017

[4] Sornette, Didier and Zhuli He, Statistical Agent-Based Model Project Progress Report August 2016

[5] Go programs versus [...]
`https://benchmarksgame.alioth.debian.org`
retrieved April 25, 2017

[6] `https://www.cadmo.ethz.ch/education/thesis/guidelines.html`
retrieved April 25, 2017

[7] Go Language Patterns, Semaphores
`http://www.golangpatterns.info/concurrency/semaphores`
retrieved June 26, 2017

[8] Master Worker Pattern
`https://gist.github.com/mikewadhera/1492294`
retrieved September 6, 2017

[9] Object-oriented programming
`https://en.wikipedia.org/wiki/Object-oriented_programming`
retrieved October 28, 2017

[10] Procedural programming
https://en.wikipedia.org/wiki/Procedural_programming
retrieved October 28, 2017

[11] Imperative programming
https://en.wikipedia.org/wiki/Imperative_programming
retrieved October 28, 2017

[12] Functional programming
https://en.wikipedia.org/wiki/Functional_programming
retrieved October 28, 2017

[13] Channels
https://en.wikipedia.org/wiki/Channel_(programming)
retrieved October 29, 2017

[14] Light-weight process
https://en.wikipedia.org/wiki/Light-weight_process
retrieved October 29, 2017

[15] Go at Google: Language Design in the Service of Software Engineering
https://talks.golang.org/2012/splash.article
retrieved October 29, 2017

[16] Google's Go: A New Programming Language That's Python Meets C++
https://techcrunch.com/2009/11/10/google-go-language/
retrieved October 30, 2017

[17] Profiling Go Programs
https://blog.golang.org/profiling-go-programs
retrieved October 30, 2017

[18] Interpreted language
https://en.wikipedia.org/wiki/Interpreted_language
retrieved October 30, 2017

[19] General Python FAQ
https://docs.python.org/3/faq/general.html#
why-was-python-created-in-the-first-place
retrieved October 30, 2017

[20] Why are Python Programs often slower than the Equivalent Program Written in C or C++?
https://stackoverflow.com/questions/3033329/
why-are-python-programs-often-slower-than-the-
equivalent-program-written-in-c-or
retrieved October 30, 2017

[21] Go programs versus Python 3
https://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=go&lang2=python3
retrieved October 31, 2017

[22] The Computer Language Benchmarks Game
https://benchmarksgame.alioth.debian.org/how-programs-are-measured.html
retrieved October 31, 2017

[23] Is Python interpreted (like Javascript or PHP)?
https://stackoverflow.com/questions/745743/is-python-interpreted-like-javascript-or-php
retrieved October 31, 2017

[24] Profiling Go Programs
https://blog.golang.org/profiling-go-programs
retrieved October 31, 2017

[25] How to write benchmarks in Go
https://dave.cheney.net/2013/06/30/how-to-write-benchmarks-in-go
retrieved October 31, 2017

[26] Package testing
https://golang.org/pkg/testing/
retrieved October 31, 2017

[27] Multiplexer
https://en.wikipedia.org/wiki/Multiplexer
retrieved November 1, 2017

[28] Effective Go, Concurrency
https://golang.org/doc/effective_go.html#concurrency
retrieved November 1, 2017

[29] Concurrency is not Parallelism
https://talks.golang.org/2012/waza.slide#1
retrieved November 1, 2017

[30] Mutual Exclusion
https://en.wikipedia.org/wiki/Mutual_exclusion#cite_note-1
retrieved November 1, 2017

[31] Race Condition
https://en.wikipedia.org/wiki/Race_condition
retrieved November 1, 2017

[32] Go Concurrency Patterns
https://talks.golang.org/2012/concurrency.slide#1
retrieved November 1, 2017

[33] Communicating sequential processes
https://en.wikipedia.org/wiki/Communicating_sequential_
processes
retrieved November 1, 2017

[34] Is it possible to force a go routine to be run on a specific CPU?
https://stackoverflow.com/questions/19758961/
is-it-possible-to-force-a-go-routine-to-be-run-on-a-specific-cpu
retrieved November 1, 2017

[35] Master/slave (technology)
https://en.wikipedia.org/wiki/Master/slave_(technology)
retrieved November 1, 2017

[36] False sharing
https://en.wikipedia.org/wiki/False_sharing
retrieved November 1, 2017

[37] Go Slices: usage and internals
https://blog.golang.org/go-slices-usage-and-internals
retrieved November 3, 2017

[38] S&P 500 Index
https://en.wikipedia.org/wiki/S%26P_500_Index
retrieved November 4, 2017

[39] Backtesting
http://www.investopedia.com/terms/b/backtesting.asp
retrieved November 4, 2017

[40] Semaphore (programming)
https://en.wikipedia.org/wiki/Semaphore_(programming)
retrieved November 4, 2017

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Statistical Agent Based Modelling Approach (SABM) Towards Complex Financial and Economic Systems: Implementation in Go

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Battilana	Steven

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 8.11.2017	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.