Benefits of Golang for Distributed Computing

Alex Nassif
Computer Engineering and Computer Science
California State University Long Beach
Alex.Nassif@student.csulb.edu

Dennis La
Computer Engineering and Computer Science
California State University Long Beach
Dennis.La@student.csulb.edu

ABSTRACT

Go is a relatively new programming language developed by Google whose main strength comes from how it handles concurrency. In this paper, we will discuss Go's language characteristics, concurrency model and features such as Goroutines, channels and the Go Runtime. We will explore how Go offers benefits in terms of writability and performance compared to other languages like Java, C++, Erlang and Scala by surveying some case studies. We will also go over some drawbacks of Go, namely the lack of inheritance and generic types and how to overcome them.

Lastly, we implemented our own experiment to test the execution runtime performance of Goroutines against Java Threads by performing a parallelized inorder tree walk. We found that Go performed the task faster.

1. INTRODUCTION

Go was invented in 2009 by Google as an open source project and has grown over the years with hundreds of contributors. It was born from the frustration of other languages used at Google such as Java and C++ because they were not able to meet the performance requirements that Google needed [10].

Furthermore, many large scale companies today require distributed systems to handle their traffic and have resorted to using Go in their systems to help meet their needs. For example, with the popularity of containers, Go's concurrency plays a big role in major container companies such as Docker and Kubernetes [5]. Both companies use Go as their language of choice. Go is well known primarily for its distributed computing performance.

1.1 Language Characteristics

First of all, Go is a statically typed, compiled language and has garbage collection. Having static types helps Go's execution runtime as dynamic type checking incurs an overhead cost. In addition, since Go compiles to machine code, it will have an advantage over interpreted languages and languages that compile to an intermediate code like bytecode in Java and Scala. Some other benefits of Go are that it was designed for fast compilation and ease of programming [13]. Having garbage collection allows the language to be safer than a language like C++ and avoid memory leaks. This helps with the ease of programming in Go.

In terms of concurrency, Go follows the principle of "Don't communicate by sharing memory, share memory by communicating" [5] otherwise known as the CSP model (Communicating Sequential Processes). In other words, Go is a language that handles concurrency by favoring message passing over shared memory. This principle was designed to help create concurrent programs that are simpler to understand and better suited for automatic verification of the absence of concurrency errors such as deadlocks or starvation [5]. Overall, CSP's goal is to make concurrent programming more intuitive and thus help reduce the chances of deadlocks happening. This idea is implemented with constructs called Goroutines and channels which will be discussed in more detail later in this paper. Intuitively, Goroutines are lightweight threads that send and receive messages through a pipe called a channel as depicted in figure 1.

Figure 1: Visual representation of Goroutines and Channels [7].

1.2 Communicating Sequential Processes

The reason for Go's successes in concurrent computing is due to paradigm of communicating sequential processes or CSP which is Go's concurrency model. It was created in 1978 by Charles Antony Richard Hoare and it is a formal language used to mathematically model the interactions between concurrent processes [3]. Go utilizes CSP mechanisms in order to abstract the complexity of concurrent processes [13]. A complete and comprehensive discussion of CSP is beyond the scope of this paper.

However, we will discuss the three main benefits of this concurrency model.

1. Sequential code
2. Message passing
3. Scalability

The first one is that when writing the code for the concurrent processes, the programmer can just write sequential code, hence the name sequential processes. This improves ease of use as most programmers are used to writing sequential code already.

The second is that CSP promotes Go's preference of message passing over shared memory, hence the communicating part. Go accomplishes this by passing references through channels. Therefore, the receiver of that reference is the only process that has access to it which enforces mutual exclusion. Message passing helps prevent issues like deadlocks and starvation since the programmer does not have to explicitly use locks to control access to shared memory [14] which also helps improve ease of use for the programmer.

Lastly, CSP allows for greater scalability as adding more Goroutines improves throughput since message passing provides greater scalability potential over shared memory [2].

2. CONCURRENCY FEATURES

Here we will give some background knowledge on the mechanisms used in Go concurrency. Goroutines are the processes and channels provide a means to send and receive messages. The runtime manages the Goroutines.

2.1 Goroutines

Goroutines are one of the basic programming structures in Go and they enable the use of concurrency in a Go program. They are easily created by using the keyword "go" in front of a function declaration. The code inside the Goroutine runs concurrently to other pieces of code. They are implemented as nonpreemptive coroutines which are concurrent subroutines. There is always at least one goroutine running in a go program and that is the main program.

A main strength of Goroutines is that they are super lightweight. The Go runtime allocates a stack of 2 KB per goroutine [1]. In addition, the stack is resizeable depending on how much space is required of the Goroutine [4]. Therefore, the cost of creating Goroutines is much lower compared to the cost of creating threads in Java where the overhead is very high [13]. Java uses OS threads which can have a large stack size of 1MB or greater [9].

As mentioned earlier, Goroutines are run in a single process that uses multiple threads. Consequently, this allows for less computational overhead as Goroutines minimize the need for the OS to context switch between the kernel and user space [6]. As a result, Goroutines provides improvements over threads in Java.

2.2 Channels

Channels are the mechanism by which Goroutines communicate. For example, we can create a channel and pass it to a couple of Goroutine functions. One Goroutine will pass some information to that channel and the other Goroutine will wait until its channel receives the information. In the code below we create a channel of type string using the built-in make() function. We call the functions hello() and world() and make them into Goroutines by adding the keyword "go" to the front of call. The hello() function will send "hello" to the messages channel. The function world() will receive the message and and append "world" to it. The function will then send the string back into messages and main() will print "hello world".

```
func main() {
        messages := make(chan string)
        go world(messages)
        go hello(messages)
        time.Sleep(3000 * time.Millisecond)
        fmt.Println(<- messages)
}
func hello(messages chan string){
        hello := "Hello"
        messages <- hello
}
func world(messages chan string)  {
        world := <-messages
        world += " World"
        messages <- world
}
```

Essentially, channels acts as a pipe for data flow in order to facilitate message passing.

2.3 Go Runtime

In order to manage Goroutines, channels and garbage collection, a runtime environment for Go is needed. The runtime is written in C and it is linked to user code after compilation and the result is an executable file in the user space of the operating system [4]. The Go program is run on one OS process which creates threads as Goroutines are run on these threads. If a Goroutine blocks then the scheduler will create another thread and allow other Goroutines to run. The Go runtime is made up of three C structs. The G struct, M struct and SCHED structs. The G structs are Goroutines, while the M structs are the threads that they run on. The SCHED struct keeps track of all the G and M structs. It keeps a queue of M's that need a G to run and a queue of what G's are available to work. The Go runtime acts as the intermediary between the user code and the operating system [4].

3. ADVANTAGES OF GO

Now we will discuss some advantages of Go, focusing primarily on its writability and performance by comparing it to other programming languages.

3.1 Writability

Go is a statically typed language, but also has dynamically typed capabilities such as Python. For some, this may make Go easier to write as the programmer does not always need to specify an exact type. In the example below we declare a variable with the number 7 in two ways. One using the keyword var and declaring the type int. In the second example no type is declared and Go uses type inference to infer the variable x is of type int.

```
func main() {
        var i int = 7
        x := 7
}
```

An advantage that Go has over other programming languages in terms of concurrency is its ease of use. To write a concurrent function, one simply has to use the "go" keyword

in front of a function call. In comparison to Java threads, one has make sure to either extend the Thread class or implement the Runnable interface which is more complicated than simply invoking one keyword.

In addition, Java and C++ utilizes an explicit concurrency model which means that shared memory must be protected with locks and mutexes which can potentially cause race conditions and deadlocks if used incorrectly [14]. As a result, the programmer must explicitly manage those constructs when writing concurrent code. Since Go uses the CSP concurrency model, message passing helps circumvent these issues and provides a safer mode of handling concurrency. It it also worth mentioning that C++, unlike Java and Go, has no garbage collection. Therefore, the programmer will also have to worry about memory allocation on top of managing the concurrency mechanisms, adding another layer of complexity.

NUMBER OF LINES AND CODE SIZE

Benchmarks	Number of Lines	Code Size(byte)
matrix.go	40	743
Matrix.java	46	937
parallel_matrix.go	67	1283
ParallelMatrix.java	82	1979

Figure 2: Comparison between the number of lines of code in Go and Java [13].

A case study compared the difference in lines of code and code size between programs implemented in Go and Java [13]. In reference to figure 2, all programs perform the same task of matrix multiplication. The first two are not parallelized while the last two are. This figure showed that less lines of code are needed in Go compared to Java to perform matrix multiplication. This could help programmers write less code which offers an improvement in terms of writability. In addition, the code size in bytes was also smaller for all the Go files which can offer savings in space. Interestingly, the major differences in number of lines of code and code size come from the parallelized versions of the matrix multiplication program. This further evidences the benefits of concurrent programming with Go compared to Java.

Overall, writing concurrent programs in Go is much easier as it is supported natively using the "go" keyword. Other languages such as Java and C++ does not have native concurrency support and require outside libraries. Go streamlines the process of writing concurrent programs by letting the programmer not worry about creating explicit mutexes and locks.

3.2 Performance

Here we will go over some experiments done in other studies that compare the concurrency performance between Go and other languages, namely Java, C++, Erlang and Scala.

3.2.1 Comparison to Java

Java is a widely popular object oriented programming language used in many applications. It has support for concurrency through the concurrent library. Concurrency is accomplished through the use of the Thread class and Runnable interface. In order to compare the concurrency performance between Go and Java, an experiment comparing matrix multiplication implementations in Go and Java has been done to benchmark compile time and execution time [13].

COMPILE TIME

Benchmarks	Compile Time (millisec)
matrix.go	0.214
Matrix.java	0.670
parallel_matrix.go	0.225
ParallelMatrix.java	0.694

Figure 3: Comparison between Go and Java with regards to compilation time [13].

As show in figure 3, the compilation time for Go was much faster compared to Java for both the non parallelized and parallelized implementations of matrix multiplication. The reason for this is due to how Go was designed for fast compilation. It avoids a lot of the overhead of C-style dependency analysis with imported libraries and files [13]. Therefore, faster compilation times with Go becomes a great benefit for companies with large applications since Go can potentially increase compilation speed by a factor of about three, as seen figure 3.

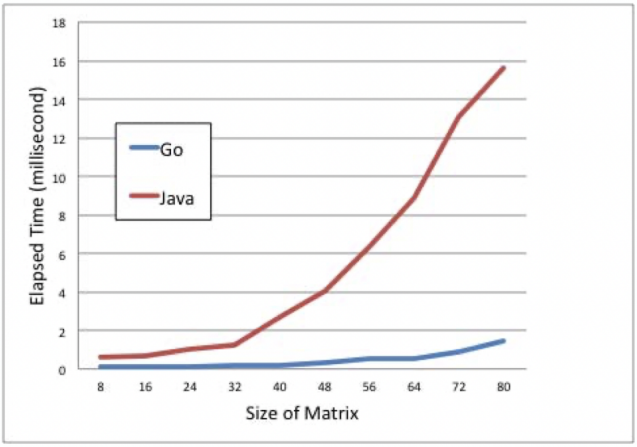


Figure 4: Comparison between Go and Java with regards to parallelized matrix multiplication performance [13].

Moreover, in terms of execution, figure 4 reveals that Go decidedly outperforms Java in the parallelized matrix multiplication benchmark. As the size of the matrix grew, the execution time of Go remained relatively the same. On the other hand, Java's execution time grew dramatically. This discrepancy was partly due to the high creation overhead of Java Threads [13]. Goroutines, by contrast, are very lightweight

and thus improve execution performance. As a result, Go shows great promise in becoming a mainstream programming language like Java especially in the area of distributed computing due to its strengths in concurrency.

3.2.2 *Comparison to C++*

C++, like Java, is another popular object oriented programming language with support for concurrency. Go also outperforms C++ with regards to compilation time due to the avoidance of processing the included files and libraries in C++ header files and Go's lack of inheritance which saves the compiler from defining class hierarchy relationships [12]. This allows the compiler to avoid much of the compilation overhead seen in C-style languages. Again, Go's faster compilation time can provide potential time savings when compiling large applications.

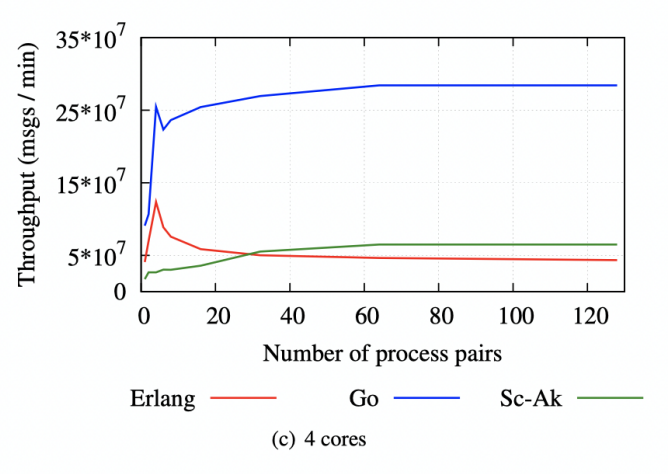
In terms of execution time however, C++ performs faster than Go. A case study compared the concurrency performance of Go and C++ Threading Building Blocks (TBB) on a dynamic programming problem known as the optimal binary search tree problem and found that C++ was about 1.6 to 3.6 times faster [11]. Despite Go not performing as fast as C++, Go offers other features such as garbage collection and ease of use. Thus, there is a trade off between execution times and safety and writability.

3.2.3 *Comparison to Erlang*

Erlang is a programming language used for distributed computing in popular multiplayer games such as League of Legends and Call of Duty and the messaging app, WhatsApp [14] which is owned by Facebook. Unlike Java, C++ and Go, Erlang is a functional, dynamically typed programming language. Like Go, concurrent Erlang processes also communicate through message passing [14]. However, Erlang utilizes an actor concurrency model instead of CSP. The actors pass messages to each other which helps reduce the pitfalls of shared memory like deadlocks and race conditions. Thus, Go and Erlang share the use of message passing in coordinating concurrent processes.

Despite their similarities in preferring message passing, in figure 5, Go outperforms Erlang by a relatively wide margin in terms of throughput. This benchmark measured the message exchange rate between a worker process and a client process. A possible reason for this result could be attributed to Go's statically typed channels [14]. Since Erlang is a dynamically typed language, types that are passed around as messages need to be checked during runtime which incurs an overhead cost. Therefore, having statically typed objects in Go enables a better yield in message passing throughput since types are primarily predetermined at compile time.

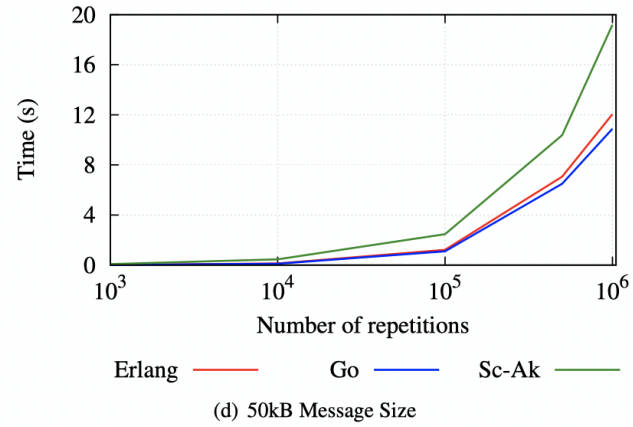
Moreover, in terms of process communication latency, the performance between Go and Erlang are similar as shown in figure 6. This benchmark measured the time taken to pass a message of a given size back and forth between two processes over a certain number of repetitions [14]. Go may seem to have a slightly lower communication latency compared to Erlang. However, since the margin is very small, it may not be a significant difference. Thus, any benefit of using Go over Erlang for reduced communication latency may be minimal. In addition, the same could be said when comparing the



(c) 4 cores

Figure 5: Comparison of concurrency throughput between Go, Erlang and Scala. Throughput was measured by messages per minute. This graph shows the results from a computer with 4 cores [14].

process creation time between Go and Erlang when referring to figure 7. The benchmark used is simple, as it just measures the time taken to spawn a given number of processes [14]. As a result, the difference in process creation speed is minimal as well as Go is only slightly faster than Erlang.



(d) 50kB Message Size

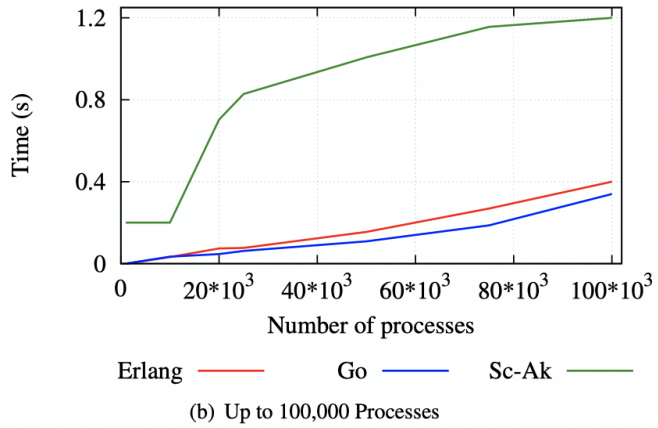
Figure 6: Comparison of process communication latency between Go, Erlang and Scala. The message being passed was 50kB [14].

3.2.4 *Comparison to Scala*

Scala is an statically typed, objected oriented and functional language used for supporting scalable high-load services [14]. Since both Go and Scala are statically typed, they both have a runtime performance advantage over dynamically typed languages as types are not checked at runtime as often. Scala, similar to Erlang uses an actor based concurrency model. As a result, Scala also utilizes message passing over shared memory like Go and thus has the advantages that come

with message passing. However, Scala differs from Go in that it runs on the Java Virtual Machine as it compiles to byte code just like Java. Naturally, since Go compiles to machine code, it will run faster than Scala.

Scala was compared to Go in the same case study discussed in section 3.2.3 using the same benchmarks of concurrency throughput, communication latency and process creation time. The study used Scala with the Akka framework. It is important to note that When Scala is used with the Akka framework, the messages sent between actors are dynamically type checked [14] which incurs a runtime cost. With this in mind, Scala trailed behind Go in every benchmark as seen in figures 5, 6 and 7. Thus, companies that use Scala can benefit greatly from improved distributed computing performance by switching to Go.



(b) Up to 100,000 Processes

Figure 7: Comparison of process creation time between Go, Erlang and Scala. The max number of processes was set to 100,000 [14].

4. DISADVANTAGES OF GO

There are some disadvantages to Go as well. Here we focus on its lack of inheritance and generics as those are common features offered in many popular programming languages.

4.1 No inheritance

There are no classes in Go and is therefore a procedural language rather than an objected oriented one. Instead it uses structs. Methods can be added onto structs like so.

```
type Foo struct {
    f int
}

func (f Foo) Name() string {
        return "I'm foo"
}
```

Since there is no inheritance, Go promotes the principle of programming to an interface rather than an implementation and favors composition over inheritance. Composition is achieved by creating struct variables within structs.

```
type Tool struct {
        // more code here
}

func(t* Tool) Turn(){
    fmt.Println("I'm turning")
}

type Wrench struct {
        Tool
        // more code here
}
```

By embedding a Tool struct within Wrench we are able to use Tools Turn method, otherwise known as delegation. In this sense, we can get around the lack of inheritance as reuse is accomplished by composing structs with other structs. This can actually be seen as an advantage as composition can be defined dynamically at runtime by passing references of the same type which allows for more flexible code. On the other hand, inheritance relationships are static and cannot be altered at runtime.

4.2 No generics

Golang does not have generics in the same sense that, for example, Java does. We cannot create a function that can take in various objects. For example, in Java we can create a generic function to add all the numbers in an array. We can pass in Integer, Double, and Float arrays.

In Golang, we would have to create an interface and our data type would have to implement the interface's methods and then we would have to bind our data type to the interface. Golang's sort method is implemented that way as shown in the following code [8]. This code would only work for arrays of type int and we would have to do the same for any other type we would like to sort. It can become very tedious and the code can be very verbose. However, this is a work around for the lack of generics.

```
type Interface interface {
        Len() int
        Less(i, j int) bool
        Swap(i, j int)
}

func Sort(data Interface) {
        n := data.Len()
        quickSort(data, 0, n, maxDepth(n))
}

type IntSlice []int

func (p IntSlice) Len() int {
        return len(p)
}
func (p IntSlice) Less(i, j int) bool {
        return p[i] < p[j]
}
func (p IntSlice) Swap(i, j int) {
        p[i], p[j] = p[j], p[i]
}
```

```
func (p IntSlice) Sort() { Sort(p) }
```

[8]

5. OUR EXPERIMENT

We decided to conduct our own experiment to see if goroutines were quicker than Java threads. Our experiment consisted of creating Binary Trees for both Go and Java. We included two methods, one for insert and one for an inorder walk. Initially, we implemented recursive methods for both Java and Go, but Java would give us a stack overflow error when we tried creating more than 10k nodes. We had to refactor insert and inorder walk into iterative functions. Interestingly, enough Go allowed us to create a Tree with 1 million nodes without any problem.

We decided to test the inorder walk method on 10,000 nodes and 100,000 nodes. We timed how long it would take for two Goroutines to do an inorder walk on two trees. Each goroutine got one tree. We did the same for Java except with Threads. What we found is that Go took about 0.00007 seconds on average to walk both trees concurrently. Java on the other hand, took anywhere from 0.007 seconds to 0.02 seconds. As a result, Go was definitely faster. The same outcome held for 100k nodes. Go averaged around 0.0004 seconds and Java averaged 0.03 seconds.

6. CONCLUSION

Go is a programming language that compiles and/or executes fast compared to other languages such as Java, C++, Erlang and Scala. Its main draw comes from its concurrency performance as evidenced by the various case studies discussed in this paper and in our own experiment. This is attributed to multiple factors. It utilizes the CSP concurrency model and therefore is based on message passing. This helps reduce the occurrences of deadlocks and race conditions as there is no shared memory. It is statically typed and therefore does not need to dynamically type check messages passed through channels. Goroutines are super lightweight and thus lowers the cost of creating them. Lastly, Go compiles to machine code and will naturally have an advantage over interpreted languages and languages that compile to an intermediate language like bytecode or common intermediate language (CIL). Overall, Golang shows massive potential in becoming a much more mainstream programming language as the demand for distributed computing continues to grow.

7. REFERENCES

[1] V. Blanchon, "Go: How does the goroutine stack size evolve?" June 2019, https://medium.com/a-journey-with-go/go-how-does-the-goroutine-stack-size-evolve-447fc02085e5, accessed 2020-12-02.

[2] R. Carlsson, K. Sagonas, and J. Wilhelmsson, "Message analysis for concurrent programs using message passing," *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 4, p. 715–746, Jul. 2006. [Online]. Available: https://doi-org.csulb.idm.oclc.org/10.1145/1146809.1146813

[3] K. Cox-Buday, *Concurrency in Go*. O'Reilly Media, Inc., 2017.

[4] N. Deshpande and N. Weiss, "Analysis of the go runtime scheduler," 2012.

[5] N. Dilley and J. Lange, "An empirical study of messaging passing concurrency in go projects," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 377–387.

[6] G. Fisher and C. Yeh, "Comparing producer-consumer implementations in go, rust, and c," *ACM*, vol. 1, no. 1, pp. 1–7, 2017.

[7] GeeksforGeeks, "Channel in golang," https://www.geeksforgeeks.org/channel-in-golang/, accessed 2020-11-16.

[8] Google, "Sort," https://golang.org/src/sort/sort.go, accessed 2020-11-29.

[9] K. Khare, "Why goroutines are not lightweight threads?" May 2018, https://codeburst.io/why-goroutines-are-not-lightweight-threads-7c460c1f155f, accessed 2020-12-02.

[10] J. Meyerson, "The go programming language," *IEEE Software*, vol. 31, no. 5, pp. 104–104, 2014.

[11] D. Serfass and P. Tang, "Comparing parallel performance of go and c++ tbb on a direct acyclic task graph using a dynamic programming problem," in *Proceedings of the 50th Annual Southeast Regional Conference*, ser. ACM-SE '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 268–273. [Online]. Available: https://doi-org.csulb.idm.oclc.org/10.1145/2184512.2184575

[12] J. Stjernberg and J. Anneback, "A comparison between go and c++," 2011.

[13] N. Togashi and V. Klyuev, "Concurrency in go and java: Performance analysis," in *2014 4th IEEE International Conference on Information Science and Technology*, 2014, pp. 213–216.

[14] I. Valkov, N. Chechina, and P. Trinder, "Comparing languages for engineering server software: Erlang, go, and scala with akka," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, ser. SAC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 218–225. [Online]. Available: https://doi-org.csulb.idm.oclc.org/10.1145/3167132.3167144