Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Distributed Execution of Communicating Sequential Process-Style Concurrency

Golang Case Study

James Whitney[*] · Chandler Gifford [*] · Maria Pantoja

Received: date / Accepted: date

Abstract In the last decade the majority of new Central Processing Units (CPU) have become multicore. To take advantage of these new architectures we need programming languages that can express parallelisms. The programming language Golang is well known for providing developers with an easy programming model for Communicating Sequential Process (CSP)-style concurrency enabling programmers to easily write functions that will execute on the different cores of a modern multicore CPUs. Unfortunately, Golang does not support distributed execution of *goroutines* on clusters or distributed systems. In this paper, we extend the concurrency capabilities of Golang to a distributed cluster by providing a library called Gluster, that is simple and easy to use. We developed a programming model that allows users to easily distribute work between machines, in a similar way as workloads are distributed in multicore CPUs by using operating system threads with libraries similar to Pthreads and OpenMP. Our Gluster solution is based on a single master node that connects to peers over a network and distributes work to these peers. The master node is able to send function arguments over the network to worker nodes as well as receive return values. Results using matrix multiplication show that our distributed implementation can speed up the *goroutines* by 5x in a small 16 nodes cluster but more importantly, it shows that the results are scalable to cluster size.

Keywords Concurrent Programming · Distributed Computing · Cluster · HPC

James Whitney
Cal Poly San Luis Obispo College of Engineering, San Luis Obispo California, USA
E-mail: Contact@JasWhitney.com

Chandler Gifford
Cal Poly San Luis Obispo College of Engineering, San Luis Obispo California, USA
E-mail: chgiffor@calpoly.edu

Maria Pantoja
Cal Poly San Luis Obispo College of Engineering, San Luis Obispo California, USA
Tel.: +001 805 756 2824
E-mail: mpanto01@calpoly.edu

[*] These authors contributed equally to this work

1 Introduction

Golang (Go) [1], the programming language created by Google in 2009 to enhance their production systems was created by Robert Griesemer, Rob Pike, and Ken Thompson; but is now Open Source and free to use. The key defining feature of Go, is the built-in concurrency primitives in the shape of *goroutines* (light-weight processes), and channels that allow for CPU asynchronous execution. These concurrency primitives are derived from Hoare's [2] Communicating Sequential Processes (CSP). In Go, *goroutines* are mapped on OS threads and communicate with each other through data structures buffering input data (channels). The advantage of Go's *goroutines* is their independence from the operating system scheduler. Go has its own scheduler that allows go threads to execute just like Operating Systems (OS) threads but without the overhead of system calls and the OS scheduler. This native support makes Go an important tool in a user's toolbox for parallel programming. Since the creation in 2009 Go has gathered a fast-growing community of developers specially for server-side programming, making Go one of the fastest growing (in number of users) programming languages [3]-[4] in the last couple of years. The main goal of this paper is to make it simple for users of the Go programming language to run code on a distributed system as well in a multicore one, by implementing a library (Gluster) that abstracts the details and difficulties of distributing code by just adding five new API calls to the original Go language. In its current state, Go supports writing multithreaded programs using the goroutine API, which allows called functions to be run in separate threads, and the use of channels for thread safe message passing. What Go lacks is a way of running functions in a distributed manner across multiple physical machines. In this paper we focus on different aspects of the distributed execution and ease of use by providing the Gluster library that allows the connection to peers on a local network and the distribution of work across those peers with the design goal of being as code light as possible. The design of Gluster has been inspired by trends in modern parallel programming, mainly the appreciation of ease of use over performance. At the GTC 2018 conference, Nvidia's Stephen Jones announced plans to move CUDA to a unified memory model in the near future [5]. The primary reason for the change being that CUDA is highly complex and often difficult to debug especially around memory. Nvidia found that this problem was compounded by the fact that the growing number of research scientists using their GPUs were not skilled programmers, and that the performance cost of significant simplification of the CUDA memory architecture is minor in comparison to the amount of time saved from development. Our design philosophy with Gluster is similar. We acknowledge that a robust implementation of an algorithm with classical distributed programming languages, for example MPI, may for greatly optimize code get better results than our Gluster implementation; however, the ability for a user to turn an existing implementation into one that is distributed in only five API calls is a very valuable proposition that this library brings.

1.1 Previous work

The Go language has been the subject of research in recent years, due to its concurrency features and easy of use. There are several works that focus on verification and performance comparison with other concurrent programming languages. Prasertsang and Pradubsuwun present formal verification of the Go language to assure the correctness of the program[6]. Togashi and Klyuev analyzes and compare the performance of Java Threads versus Go's *goroutines* and channels[7]. From the experiment, Go derived better performance than Java in both compile time and concurrency and since is also easier to use the authors conclude that Go will be used much more in the coming years. The Ueda and Ohara paper also compares the Go language with two dynamically compiled languages, JavaScript and Java[8]. Experimental results show that the Go implementation achieved a 3.8x and 2.4x higher throughput than the JavaScript and Java implementations respectively, primarily because Go suffers less from polymorphism due to static typing than JavaScript and because the Web framework for Go causes less overhead to process Web service requests than that for Java. The articles by Lange, Toninho, and Yoshida [9] and Midtgaard, Nielson, and Nielson [10] use theoretical statistics analysis of imperative processes communicating, lifting traditional sequential analysis techniques to a concurrent setting.

Some other works focus on adding new data structures to Go to improve its performance on specific algorithms. Jenkins, Zhou, and Spear [11] introduce the Interlocked Hash Table(IHT), a highly concurrent lock-based map, to add built-in data structures that allow fine-grained concurrent access. The scalability of the IHT, and its optimized implementation inside of the Go compiler and runtime, enable it to outperform all known alternatives in Go, to include lock-free and lock-based open-source maps. Obtaining performance up to 7the performance of the default map at high thread counts. Pasarella, Vidal, and Zoltan [12] implement an alternative approach to implement the Divide and Conquer paradigm, named dynamic pipeline, using the problem of counting triangles in a graph to prove the concept. To evaluate the properties of pipeline, a dynamic pipeline of processes and an ad-hoc version of MapReduce were implemented in the language Go. Observed results suggest drastically reducing the execution time with respect to the MapReduce implementation. In the paper by Binet [13] a High Energy and Nuclear Physics (HENP) multi threaded library is developed, Go-HEP, to easily write concurrent software to interface with legacy HENP C++ physics libraries. In a more recent paper Togashi and Klyuev [14] discuss an approach used to develop a better schedule management system. It is designed using Google App Engine (GAE) and the Go programming language. Developers can concentrate on writing applications because they can skip traditional construction of the development environment.

There is also research to add features to the language so it can work on different distributed and RealTime environments. Fang et al. [15] present the design of Go-RealTime, a lightweight framework for real-time parallel programming based on Go language. Goroutines are adapted to provide scheduling for real-time tasks, with resource reservation enabled by exposing Linux APIs to Go. Results show nearly full utilization on 32 processors scheduling periodic heavy tasks using Least Laxity First (LLF) algorithm. Go-RealTime is implemented by modifying open source Go runtime, implementing Go packages and by importing important Linux system calls into Go. It uses Linux thread scheduler for resource reservation. Goroutine running on top of thread is the unit of concurrency. This framework greatly simplifies the implementation and deployment of RT programs, and improves the flexibility in system extension. A paper by Scionti and Mazumdar [16] proposes a first attempt to map goroutines on a data-driven based Program eXecution Models (PXM). Modifying the Go compilation procedure and the run-time library to exploit the execution of fine-grain threads on an abstracted parallel machine model.

The main goal of this paper is to make it simple for users of the Go programming language to run code on a distributed system. There is research on how to use Go together with Representation State Transfer (REST) web standard architecture and HTTP protocol to write distributed Web applications using Go [17], but the setup required is intensive and is mainly applied to web based distributed applications mainly to search data and not to distribute computations. In the work developed by Proto-Actor [18], a private company that does not publish its research, an API for cross platform actor support between Go and C# is introduced. This is based in the concept of actors developed by Hewitt et al. [19]. The implementation is a work in progress and is company property. In gleam [20] Lu, et al. tried to implement similar functionality to the one we are proposing. The proposed work has several shortcomings and restrictions in function calls. In this paper, we attempt to solve the shortcoming and allow users to execute any kind of function on a distributed system. Gleam is high performance, but it is difficult for programmers to use gleam since it focuses its solution on MapReduce [21] paradigm for splitting up a distributed program. Our approach focuses on allowing the user to execute arbitrary functions on a distributed system based in a model similar as the one used in Message Passing Interface (MPI) [22] by creating a library that allows for generalized function execution on any distributed system. To the best of the authors' knowledge there is no other work done on adding distributed programming paradigms for Go.

1.2 Structure of the paper

The paper is structured as follows. Section 2 presents the implementation strategy of our library, Section 3 explains and summarizes the results, and in Sections 4 and 5 we present future work and conclusions of the project.

2 Implementation

Before we begin, we would like to define some vocabulary used in this paper. Our system specifies a cluster as consisting of multiple networked nodes. The master node is a single node that distributes work over the network to worker nodes. The runner is the program that executes jobs on the worker nodes. Before discussing the implementation of the master and the runner, we first examine the rational behind using Go as the implementation language.

2.1 Language Choice

The design goals and feature requirements needed to implement Gluster limits the number of languages they could be written in. We ultimately chose to write the library in Go due to Go's extensive library support and also because its relative popularity. The language feature requirements we need are explained as follows:

Reflection. When arguments are passed in to a distributed function call, the types of those arguments must be checked against the function prototype of the requested function. The arguments passed in are initially of type interface{}, meaning they could be of any type. To execute the function we need to find their actual type at runtime, and we can do this by using reflections. Reflection are not supported by many programming languages. Reflection helps programmers make generic software libraries to process different formats of data.

Shared objects and runtime linking. Compiling code from function files is a necessary step for the functions to be run on the distributed nodes. These functions must be compiled as shared object files so they can be runtime linked into the already running worker nodes. This compilation and runtime linking is supported in Go via Go plugins. Go plugins have some shortcomings, such as only supported in Unix operation systems.

In addition to the above required language features for Gluster, there are features of Go that are very desirable for the implementation of this project, a list of them include:

Built in encoding libraries. Gluster makes use of Go's Gob encoding protocol. Gob is flexible, allows for encoding complex structures easily, encodes data in an efficient over the wire format, and handles flattening of pointers automatically [25]. This allows us to send arguments and return values between nodes with just a few calls to Gob encode and decode.

Goroutines for easy concurrency. When a distributed function call is made, communication with the worker node and execution of the function is done concurrently in the background and the distributed function call returns immediately. This is to be expected with any distributed system and this functionality could be done in almost any language. However, go provides a very easy to use concurrency model via go routines which simplifies the implementation of Gluster.

Garbage collection. Implementing Gluster in a garbage collected language has a number of benefits. First, it simplifies some of our implementation and the user's program. The biggest advantage though, is on the worker nodes. The workers will be executing arbitrary functions implemented by the user and and without automatic garbage collection we will need to rely on the user to free all memory and avoid leaks. There is still the potential for the user to abuse memory allocation but having garbage collection means that when the worker exits the user's function memory will be free and prevents memory misuse on the worker nodes.

2.2 Master and Worker

The implementation of Gluster can be split into two main components: A master program that connects to and sends the function with the required data to worker nodes. A worker node that waits with a runner (program that executes jobs on the worker nodes) for any jobs delivered by a master node. The worker node when prompted by a master node, returns the called functions return value. This design choice of having the Master/Worker node connect, enables us to lighten the complexity of the library while also granting the user control to decide how many worker nodes the system has access to. Worker nodes wait idle with our runner component until they are contacted by a master node and giving a function file to run. Since most programs are broken into different functions, with our implementation is easy to assign this functions with different ranges for the input parameter to different nodes in the distributed system. While this approach may not be as efficient as a message-passing system like MPI [22], it is significantly more dynamic, allowing users who work with clusters to easily issue distributed calls in Gluster library into existing code without needing to redesign the structure of the original code, which MPI requires. This call and response model also allows for the worker to send back a return value from the function. This allows users to see the library call as a function call, simplifying the implementation. The work of how

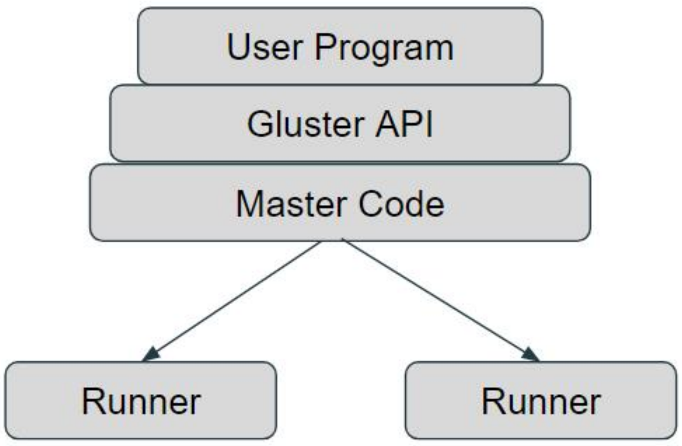

Fig. 1 High Level Design For Main Gluster Library API Components, one Master and one or several Runners/Workers.

to handle the division of data across the cluster solely on the user. This may be consider a drawback, but most Parallel and Distributed programming languages prefer to do this since it makes it more flexible and adaptable to the different problems needed to be solved. The design of our library makes the data partition logic nearly identical to other concurrent and parallel programming [22]-[24] languages by allowing users to simply divide data across machines before dividing it across threads. To implement the functionality of sending data to nodes and having them execute work, we took inspiration from Gos built in Remote Procedure Call (RPC) functionality but adapted it to a distributed system that would run jobs on separate clients. Our system consists of a library, in which the user writes their "master" program to distribute work. In addition, the system has a "runner that must be transferred and executed on each of the worker nodes. Unfortunately using standard RPC functionality would require accepting several drawbacks; primarily that both the master and runner code must be precompiled, and functions could only have a single argument and a single return value. This will be incredibly restrictive especially since we want the library to be dynamic. Because of this, we implemented our own request reply protocol that could do exactly what RPC

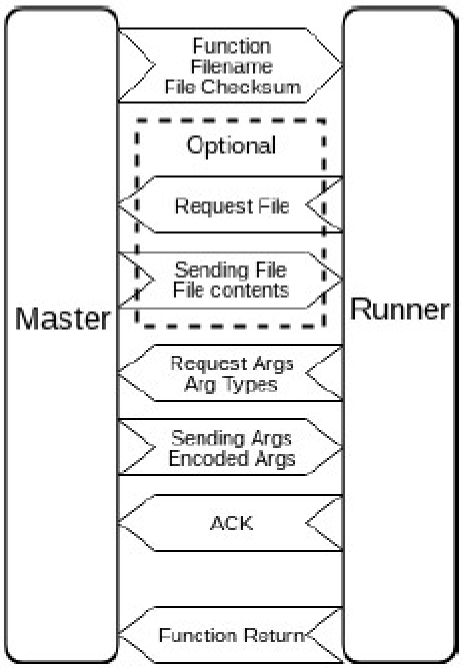

Fig. 2 Request/Reply Gluster Flow

does, with less overhead and none of the drawbacks. Our implementation is as seen in Figure 1. The user program will make Gluster API calls to add nodes to the distributed system that will be used to execute the user program, add function to be executed by the nodes, distribute, and gather the results back to the user program. A list of these API calls can be found in the Appendix. The Gluster implementation has a master node that will send the work to the worker nodes. The worker nodes contain a runner program waiting for the work to arrive from the master to start execution. The system consists of two main components:

– an API that is called by the user and acts as a master node
– Runners which run independently of any user program. The runners do not need to be compiled with the user program and simply waits on a cluster node for the master to send functions to be executed.

This has a great advantage over RPC since the runner needs to be compiled once, and then just executed as often as needed. The runners do not need to be aware of what functions they will be executing.

A Gluster program begins with the user connecting to the Runners on the network using the AddRunner API call. Then desired function files are loaded using the ImportFile API call. These function files contain the code that the user intends to be run on a distributed system. To execute a function on the cluster, the user uses a RunDist() call. It is worth noting that RunDist is designed as a variadic function so it supports calling functions with any number of arguments of any type. The library is able to interpret pointers to data passed as arguments to the function. It will automatically dereference the pointers as part of the encoder/decoder stage when the data are sent to the Runner. When calling RunDist() on your function an important thing to note is that the function cannot return a pointer. None of the data are moved between the users system and the nodes on the cluster until the RunDist() call. This API call triggers a number of interactions between the master node and one of the runners that eventually result in the function's return value being sent back to the master. After this the user program can access the return value.

The RPC implemented in this paper starts by connecting to one of the runner nodes. Then a file name, the file's hash, and a function name are sent to that runner. The runner checks if it already has the necessary function file and if not, it requests it. Then the runner requests from the master the encoded arguments. In addition, the runner sends over the expected function signature which is derived from the shared object file so that the master can type check the validity of the arguments before sending them. Once all arguments are received, the runner sends an acknowledgement and begins executing the function. Once the function is finished, the return value is encoded and sent to the master. This concludes the interaction and the network link is closed. The runner then goes back to waiting for more requests. A figure of the request/reply flow between the master and the runner can be seen in Figure 2.

On the runners. Execution of the code is handled using runtime linking. When the runner gets the function file from the master it will either be in the form of a shared library (.so) file or a source code (.go) file. In the case of the source files, the runner executes Go build to compile the code into a Go plugin, which is just a shared object file. With the shared object file, the runner uses the Go plugin system to runtime link the shared library. The runner looks up the function signature of the requested function and sends this off to the master so the master can check that the provided arguments are valid. Once all the arguments are received, the runner proceeds to call the requested function using the arguments sent by the master. At the end of the function execution the return value of the function is encoded and sent back to the master.

On the master. RunDist returns to the user a job ID. This id can be used along with the JobDone(id) API call to determine if a given job has completed. Once, a job is known to be completed, the return value can be retrieved using GetReturn(id) API call. Example of User Program in Gluster:

Listing 1 Gluster Pseudocode

```
//Add runners
gluster.AddRunner("127x17.csc.calpoly.edu")
gluster.AddRunner("127x18.csc.calpoly.edu")

//Reads in functions file to be sent to each runner
gluster.ImportFunctionFile("functions.go")

//launch runners
for jobID := 0; jobID < jobCount; jobID ++ {
    runnerID := gluster.RunDist( //Unique ID for each runner
    "functions.MatrixMultiply", //Function to be called
    reflect.TypeOf(output),   //Return Variable type
    inputA, inputB, ArrayWidth, //Function inputs
    jobID, jobCount)
    runnerList = append(runnerList, runnerID)
}
//collect runners
for _, runnerID := range runnerList {
    for !(gluster.JobDone(runnerID)) {}
        var partialOutput = gluster.GetReturn(runnerID).([]int)
        mergeArray(output, partialOutput)
}
```

3 Results

To validate our implementation in two different ways:

1. Matrix Multiplication for different matrix sizes.
2. Word count

We chose the above tests since they will show different scaling capabilities of the proposed Gluster library.

We first performed matrix multiplication with different matrix sizes. This is a standard test for most parallel programming languages to prove that parallelism can efficiently be done at the data level (all nodes execute same code with different data). We tested all our experiments on our university lab. The configuration of the nodes on the lab is as follows: CPU Intel Xeon 14 cores; nodes are connected using TCP/IP. The lab is the parallel distributed system lab in the university with no special/fast interconnect network as would be found on a dedicated cluster. This means that any improvement in time in our distributed system will scale on a cluster too. Since the project is to add distributed system capability to a general purpose programming language, this setup will allow us to better mimic the behavior of how our potential users will utilize the library in a distributed system than to use a cluster which is in general only available to research labs. Having a better intercommunication (dedicated protocol) as in cluster will do likey improve the execution time.

We developed a matrix multiplication function on square matrices of different sizes. This program was chosen among other things because it requires sending large amount of data for each of the matrices to and from each of the participating nodes on the network. To get the comparison results we use standard serial O(n^3) matrix multiplication, where X rows of the initial matrix were assigned evenly to the nodes in the cluster. In addition we ran an implementation where each node utilizes all of its available threads by parallelizing its subsection of the matrix using Goroutines. In other words, we utilize two levels of parallelism. The first level parallelizes work among many machines (distributed) and the second level uses multiple threads on each machine (multicore).

As a comparison, we ran matrix multiplication on a single machine using the same algorithm as above but not utilizing the Gluster library. There was some variance in the times due to slightly different system load during different tests due to the fact that the machines are university resources available to all students. Our results show that there is very little overhead of using the Gluster library compared to a normal Go implementation since the times on single machine non-Gluster and single machine Gluster are comparable with very little improvement

Table 1 Execution Time Standard Serial Matrix Multiplication on University Lab. Intels Xeon 14 core CPU.

Matrix Size	Single Machine Serial	Single Machine Goroutines
1024 x 1024	3.09s	0.293s
4096 x 4096	704s	53.3s
8192 x 8192	6080s	478s

Table 2 Execution Time on University Labs. 16 Computers with Intels Xeon 14 core CPUs.

Matrix Size	1 CPU Seq	1 CPU GoR	2 CPU Seq	2 CPU GoR
1024 x 1024	4.04s	0.486s	2.13	0.30
4096 x 4096	747s	55.9s	383s	29.6s
8192 x 8192	6410s	456s	3280s	334s
Matrix Size	8 CPU Seq	8 CPU GoR	16 CPU Seq	16 CPU-GoR
1024 x 1024	0.852s	0.212s	0.567s	0.252s
4096 x 4096	134s	9.06s	51.2s	6.75s
8192 x 8192	859s	70.8s	485s	42.8s

(see Table 1 and Table 2). These results were expected as Gluster was created to run on a cluster not a single node; nevertheless, we wanted to make sure for a sanity check that our implementation doesnt slow down single node execution either, and that can be confirmed by Table 1 results. The most significant comparison from our results is the speedup seen as more and more Gluster nodes are utilized/connected together as seen in Table 2 below. We see that doubling the number of machines halves the run time up to a point. We can attribute this decaying performance improvements for a large number of machines to the network bandwidth required for the master to send data to each machine. This is a very common problem in distributed systems where the memory management becomes crucial compared to computations for bigger problems. Generally this is a very difficult problem to solve with some supercomputers in research labs having complex memory hierarchy with at least seven layers. Any improvement in the memory hierarchy will benefit the implementation we present.

Figure 3 compares the execution times of using two different distributed system programming languages: Gluster and MPI. For both cases, we use an implementation of matrix multiplication written in C and execute the code using the same cluster of machines. It is clear from Figure 3 that MPI exceeds the current speed of Gluster when it comes to distributing work across machines, that demonstrates that there are still opportunities for improvement within Gluster in regards to setup and networking. The results obtained in Figure 4 are very interesting where we perform the same comparison except that this time we also implement parallelism at the machine/thread level. In other words, we compare a program implemented with OpenMP(multicore) and OpenMPI(distributed) to Goroutines(multicore) and Gluster(distributed) respectively. This demonstrates that despite the clear disadvantage per machine seen in Figure 3 Gluster paired with Goroutines is capable of out performing MPI with OpenMP.

Note on results: There is a performance difference in the library between sending precompiled functions file vs. an uncompiled one. A precompiled binary allows the runner to dynamically link in a couple of milliseconds, while the uncompiled

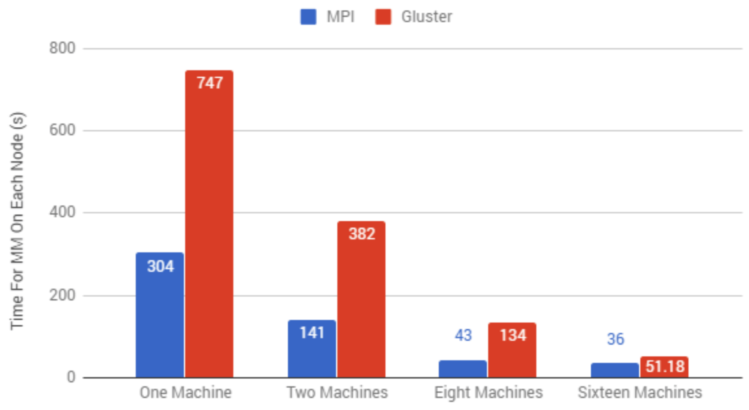

Fig. 3 Distributed Execution Time for 4096x4096 Matrix Multiplication

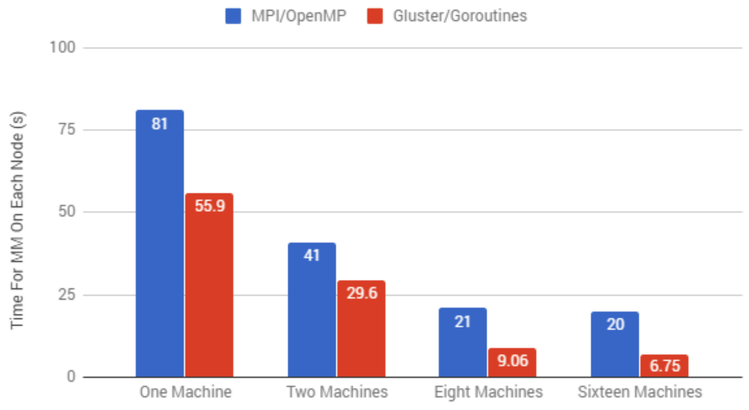

Fig. 4 Distributed and Parallelized Execution Time for 4096x4096 Matrix Multiplication

functions file goes through the compiling process, which took multiple seconds to complete with our relatively simple matrix multiplication test. With this difference in mind, users who are using clusters of similar architecture on their master machines will be able to use precompiled binaries without worry. Users will also have the flexibility of working across a diverse cluster and ensure that the functions are built properly on each system. This design choices allows for increased flexibility in the make-up of a Gluster cluster. In either case, the overhead of loading the function file is only done once, after a single function in that file is called any further function calls will happen without any building or linking overhead. This is due to the fact that the function file is hashed and the hash is checked to see if the file is already loaded before the master sends the function file to a node.

The final test program we ran was to better evaluate the scaling of Gluster with a program that counts the number of different words in a document. Our wordCount program builds a histogram of each unique word found in Herman Melvilles, *Moby Dick* on each machine before merging the histogram together on

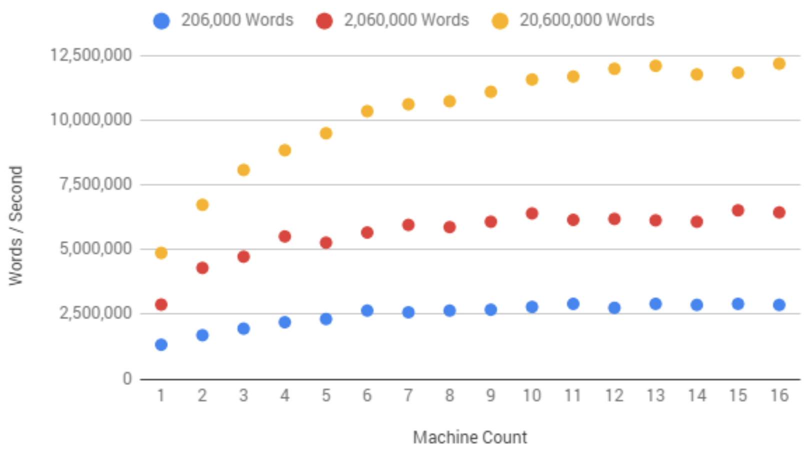

Fig. 5 Words Counted per Second with Increasing amount of Machines

the master machine. This problem demonstrates Gluster's ability to work natively with Go's standard library data structures by passing slices of strings and returning maps, well also demonstrating the scalability across multiple machines[26]. As can be seen in Figure 5 we scaled the number of copies of *Moby Dick* by a factor of 1, 10, and 100 in order to better visualize the scaling in relation to individual execution times and network overhead. From our results, represented in Figure 5, we can see that for large size documents (20 million words) our implementation scales by a factor directly related to the number of machines, when this number is less than eight machines. Beyond eight machines an increasing amount of time is spent by basic send and receive functions of the Gluster library and the execution time doesn't improve as much. Performance improvements in the library can likely be found by including smarter branch broadcast and gather APIs, to burden of network transfers.

4 Future Work

For future enhancements, we would like to support more operating systems and architectures. At the present time, Go plugins which are used for runtime linking are only supported on Linux operating systems. This limits the worker machines to only machines running Linux OS. It is possible in theory to support runtime linking is operating systems such as Windows and Mac OSX but changes would have to be made to the Go source.

We will add failure detection. There is some failure reporting to the user; GetReturn() function will return null if there was an error or if the function is not done executing. We plan to include a more verbose API that can report more status information to the user and gracefully handle errors.

We are also working on a better load balancing. The downside of Gluster's flexible design is that there is no easy way to implement better data distribution routines in the way that more structured systems such as MPI can. This is a trade-off that the user must accept for ease of use currently.

5 Conclusion

Based on the results of our experiments we conclude that Gluster can successfully accelerate execution of functions by utilizing a cluster of machines. The performance scales fairly well and adding more machines provides a near linear speedup within a cluster. There is, of course, falloff points where there are too many machines and not enough work. This is due the fact that networking overhead will eventually match or exceed the computation time if jobs are too small; but this is normal and assumed behavior on any parallel programming library.

Our proposed implementation is much more flexible than previous works and allows the parallelization of basically any kind of function and argument lists. Gluster is scalable to the number if nodes in the cluster and allows an easy way to programmers to add distributed capabilities to their go source code. In [27] the user can find a repository of the current version of the library.

References

1. GoLang: accessed in May 1 2018 Go -Lang reference. https://golang.org/doc/ (2018)
2. C. A. R. Hoare. 1978. Communicating sequential processes. Commun. ACM 21, 8 (August 1978), 666-677. DOI=http://dx.doi.org/10.1145/359576.359585
3. Go usage: accessed 1 May 2018. https://research.swtch.com/gophercount (2018)
4. Go Usage: accessed 5 may 2018. (2018) https://www.zdnet.com/article/googles-go-beats-java-c-python-to-programming-language-of-the-year-crown/
5. S. Jones. CUDA - New Features and Beyond, Developer Talk, GTC 2018 - ID S8278
6. A. Prasertsang and D. Pradubsuwun,"Formal verification of concurrency in go," 2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE), Khon Kaen, 2016, pp. 1-4.
7. N. Togashi and V. Klyuev, "Concurrency in Go and Java: Performance analysis," 2014 4th IEEE International Conference on Information Science and Technology, Shenzhen, 2014, pp. 213-216.
8. Y. Ueda, M. Ohara,"Performance competitiveness of a statically compiled language for server-side Web applications, Performance Analysis of Systems and Software (ISPASS) 2017 IEEE International Symposium on, pp. 13-22, 2017.
9. J. Lange, N. Ng, B. Toninho, and N. Yoshida. "A Static Verification Framework for Message Passing in Go using Behavioural Types", accepted draft at ICSE 2018
10. J. Midtgaard, F. Nielson, and H. Nielson, "Process-Local Static Analysis of Synchronous Processes". 25th Static Analysis Symposium (2018).
11. L. Jenkins, T. Zhou and M. Spear, "Redesigning Gos Built-In Map to Support Concurrent Operations," 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), Portland, OR, 2017, pp. 14-26.
12. E Pasarella, M. E. Vidal, and C.Zoltan, "Comparing MapReduce and Pipeline Implementations for Counting Triangles," Proceedings XVI Jornadas sobre Programación y Lenguajes,PROLE 2016, Salamanca, Spain, 14-16th September 2016.
13. S. Binet. Go-HEP: writing concurrent software with ease and Go. arXiV:1808.06529 https://go-hep.org
14. N. Togashi and V. Klyuev, "A novel approach for web development: A schedule management system using GAE/Go," 2015 IEEE 7th International Conference on Awareness Science and Technology (iCAST), Qinhuangdao, 2015, pp. 55-59.
15. Z. Fang, M. Luo, F. M. Anwar, H. Zhuang, and R. K. Gupta. 2018. Go-realtime: a lightweight framework for multiprocessor real-time system in user space. SIGBED Rev. 14, 4 (January 2018), 46-52.
16. A. Scionti, and S. Mazumdar. 2017. "Let's Go: a Data-Driven Multi-Threading Support." In Proceedings of the Computing Frontiers Conference (CF'17). ACM, New York, NY, USA, 287-290.
17. V.N. Anurag. Distributed Computing with Go: Practical concurrency and parallelism for Go applications, Packt Publishing, Feb 2018

18. Proto-actor 2018 accessed May 1 2018, http://proto.actor/
19. C. Hewitt; P. Bishop and R. Steiger (1973). "A Universal Modular Actor Formalism for Artificial Intelligence". IJCAI.
20. C. Lu, Gleam, accessed Jan 1 2018. https://github.com/chrislusf. 2017.
21. J. Dean and S. Ghemawat "MapReduce: Simplified Data Processing on Large Clusters OSDI04 Proceedings of the 6th conference on Symposium on Operating Systems Design and Implementation. Volume 6
22. OpenMPI forum website: http://www.mpi-forum.org/.
23. l. Dagum, and R. Menon. "OpenMP: an industry standard API for shared-memory programming." Computational Science and Engineering, IEEE 5.1 (1998): 46-55
24. B. Lewis, and D.Berg, Multithreaded Programming with Pthreads. Prentice-Hall, Inc. Upper Saddle River, NJ, USA 1998
25. R. Pike, Gobs of Data, website: https://blog.golang.org/gobs-of-data. The Go Blog, Mar. 24 2011.
26. Mark Algee-Hewitt. Counting words in HathiTrust with Python and MPI. Stanford Literary Lab
27. Gluster Github. https://github.com/James-Whitney/gluster

Appendices

Appendix Gluster exposes the following functions to the user:

- AddRunner(ip string) - Adds a runner with given ip on the default port
- AddRunnerPort(ip string, port int) Adds runner with specified port
- ImportFunctionFile(filename string) -imports .go file containing functions to be run on distributed machines
- RunDist(funct string, reply reflect.Type, args ...interface) int - runs a function on a runner and returns an id for the job
- JobDone(id int) bool - returns true if a job with the given id is finished and the result is set