1

Shuffleless MapReduce in Go

Ivan Lugo, Darin Doria, William Adkins, Guy Bean
ivan.lugo@knights.ucf.edu, doria@knights.ucf.edu, wadkins@knights.ucf.edu

CONTENTS

***Abstract*—This project involves improving on the classic implementation of MapReduce by executing the three phases of map reduce as concurrently as the algorithm allows. The goal is to significantly improve the runtime over a sequential implementation running on a shared memory architecture machine. We chose to implement this solution using the Go programming language because it includes built in concurrent primitives that allows us to quickly and effectively write code that can run concurrently, while offloading tedious concepts like locking variable access to the language's runtime environment. The performance results are benchmarked not only against a sequential implementation, but also compared to running on a machine with different amounts of logical processors. Further, we show an example of the overuse of the same primitives available in Go that show a multiplicative increase in runtime.**

***Index Terms*—IEEEtran, journal, LaTeX, paper, template.**

I. INTRODUCTION

MAPREDUCE is a paradigm, or thought process, in computer programming that is used to process large data sets and convert them into some meaningful information for the computer programmer. The process works by taking several computers, called nodes, and organizing them into a group structure. There are two types of group structures that can be created when linking nodes. The first is a cluster, which is a grouping of nodes that all have similar physical specifications and are on the same local area network. The second is a grid, which is essentially the compliment of a cluster in that the nodes can be made up of many different types of hardware, and can be distributed over a distributed system. In either model the nodes do the majority of the work, with a main framework machine organizing the movement and distribution of data, data storage, and data back up in the instance of node failures. Each node executes several serial segments of code generally broken down into three phases: the Map phase, the Shuffle phase, and the Reduce phase.

A. *Map Phase*

Each node executes a map() function that takes the input data and parses it down into some usable batch of data hence called a token. The node then maps over all the input and creates a token for each user defined chunk. This can be something as simple as a character, or a word, or it can be something much larger and more complicated. Once tokens have been created for each chunk of the input data all the tokens are gathered into a single location for collection into the shuffle phase.

B. *Shuffle Phase*

The main framework gathers all of the mapped tokens and redistributes them so that all tokens of like category are sent to a single node. Usually this is done by examining the tokens, which are generally (key, value) pairs, and grouping the tokens by key.

C. *Reduce Phase*

Each node then begins executing its reduce() function. This takes all of the tokens and preforms some meaningful work, also user defined, on them in order to turn them into a single updated (key, value) pair. Examples of this can be summing up word counts over a document, calculating different statistics about every user in a website's directory, or even managing a database query. Once all the tokens have been reduced the main framework takes all the reduced values and compiles them into a list to display as the output of the program.

II. GOLANG

The Go language started as a purely research oriented project in Google Inc.'s Labs, its development headed by Robert Griesemer, Rob Pike, and Ken Thompson in 2007. [1] It was designed around the principles of simplistic and idiomatic concurrency, and done so through the creation of some of its primitive commands and data types: the 'goroutine' method of concurrent execution and its related 'go' command,

Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL 32816

'channels' used as blocking and safe interthread communication tools for concurrent data manipulation, and the 'select' command, a switching statement for the aforementioned channels that allows for arbitrary functions or concurrently executing goroutines to quickly and easily choose the most readily available input from a channel.

"Concurrency is not parallelism," [1] is the mainstay of the Go language. This guiding principle states that inherently concurrent challenges should be solved as such, and the realization that a parallel solution is something entirely different drives the underlying primitives. Building on this, the static typing of the language coupled with a modest form of object oriented capabilities helps to ensure that the very model of concurrency Go tries to build is not hampered by unsafe type conversions, memory accesses, or even simple programming errors at compile time.

One of the most fundamentally useful tools prescribed in the language is the goroutine: a construct that may be thought of as a lightweight thread of execution with a small enough footprint to be created in the magnitude of hundreds of thousands with a tiny memory overhead, yet still offering all the runtime benefits of thread scheduling and concurrent execution. Any function declared in Go - including anonymous functions and closures - may be commanded to run in one of these lightweight threads with the use of the prefix 'go'. So long as the program as a whole is capable of handling the concurrency introduced with this prefix, that is all that must be done.

Channels, combined with the concurrency enabled by a goroutine, provides an efficient and safe way for those lightweight threads to communicate with one other. Channels may either be used to receive and send arbitrary blocks of data from and to any goroutine with a reference to the channel and with interest in its contents. Instantiation is done in the same way as any other language level primitive, and once instantiated, may be used indefinitely until it has been closed. The ability to communicate on a channel, while still a nicety, is not its most interesting quality. A channel will block further execution of a goroutine on a send until a receive counterpart has acknowledged the send operation, and likewise, a receive operation will block further execution until a send operation has sent its data and raised a flag. While this is happening, the runtime environment will actually move whichever routine is blocking off of its own thread of execution onto a burner thread, allowing the rest of the program as a whole to continue until that thread is able to continue with its matched channel send or receive. This optimization of execution is something exploited heavily in the implementation discussed, and indeed, one of the most powerful features of lightweight threading. Combining this with the the 'select' statement, a complex structure of handling communication events can 'fan in' from a number of channels and only pop into the main execution thread when it is ready, rather than sitting and spinning on a mutex or lock.

III. Challenge

A. Problem Statement

Map reduce is normally done in three sequential phases [4]. The problem is that each phase depends on the data being received from the previous phase. The process is entirely limited in how fast you can finish by how quickly the previous phase computes all its data and passes it on to the next phase. Current implementations attempt to increase overall speed by computing each phase of data concurrently, whether on a shared memory architecture or across a distributed system of computers, then passing it on to the next phase and repeating those steps until the process completes. The goal of our project is not only to compute each one of these phases concurrently, but to maximize how much of the computing can be done in parallel. There is only so much that can be computed in parallel since, by the very nature of map reduce algorithms, you need all the data from the previous phase before you can proceed.

B. Why Go?

Initially the thought of using Go came after researching how previous versions of map-reduce had been implemented. The most notorious version was the one implemented by Google of course. A bit more research led to finding and verifying that google's own programming language is a great fit for map reduce. Go provides built in primitives that allow the language to handle and optimize complex concurrent concepts. This allows us to easily get around common bottlenecks that are introduced when writing multithreaded programs by thinking in a multithreaded fashion without having to explicitly deal with the threads, monitors or locks.

We gain a huge advantage in debugging and programming time by allowing the language to handle threads and locks. The biggest gain is achieved when maximizing the use of goroutines and channels. Passing data through the use of channels helps us get around the use of traditional locks. Because our solution revolves primarily around running several functions concurrently, Go is a perfect fit for the solution.

IV. Our Solution

Our implementation on MapReduce focuses on when the user only cares about some subset of the data being searched. The user inputs a list of the key terms in addition to the data to be searched, and we return the values of just those key. While our implementation provides less information than standard MapReduce, it gives us a key advantage. In standard MapReduce, the program doesn't know how many reducers to create until it's finished the shuffle phase, but in our MapReduce, we know we want one reducer per keyword (or maybe some multiple). This knowledge allows us to create map, shuffle, and reduce phases that can run concurrently with each other. We first create our reducers and the channels they receive off of, then create mappers which have access to these channels and can thus communicate with the reducers directly. The shuffle phase then becomes an implicit part of the map phase, while the reduce phase is simply receiving a stream of data from mappers as they finish. This extra concurrency, we propose, will speed up runtime, even when the worst case runtime of our phases is the same as the standard implementation.

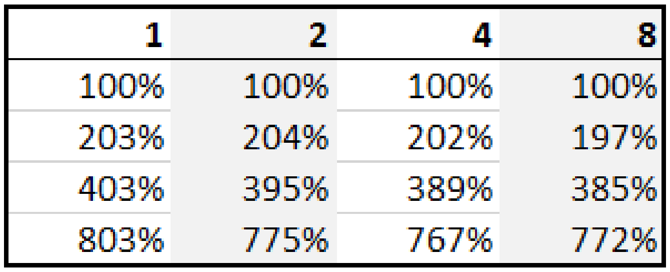
We start by taking in the files our user wants to search and the keywords they care about. We create the reducers and channels for those keywords, and set them to run in goroutines. We then create a mapper for each input file, each with the ability to send data across each of the reducer channels. They then iterate through their assigned input, hashing each word and incrementing the index of the result. This means the mapping phase has a runtime of O(n), where n is the number of words in the input file.

When a mapper finds the end of their input, they send the keyword's counts through the appropriate channels to their corresponding reducer. They then send a flag back to main, which is used in judging when to call for the final results. This phase has a runtime of O(m), where m is the number of keywords the user input.

The reducers each keep a running total of the data points they have received. They treat their channel like a queue; constantly popping values and adding them to their total. If their channel is emptied, they must wait for further instruction. The channels will be closed when all mappers are done, until that point no reducer can be guaranteed to have a correct final result, as any mapper's completion can affect all reducers result. The reducers each have buffer size limit on their channel, and are nonblocking until their buffer is filled. When main receives a flag from every mapper, it closes the receiver's channels and starts listening for results. Any receiver that is already done will send its final result across another channel to main, which will output data as it receives it. Reducers that are not done will continue to compute until emptying out their channel, and send their final result to main as they finish. The reducers have a runtime of O(k), where k is the number of mappers created, as each mapper can only send data to any given reducer once.

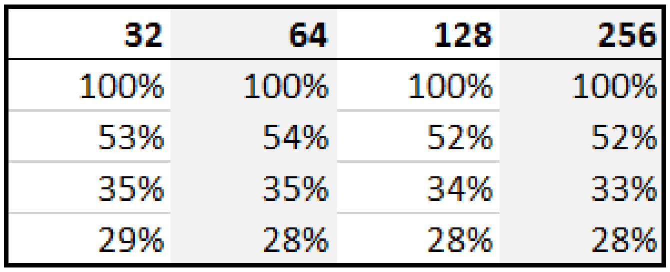
V. Results

See ExecutionComparison. [Preliminary]



1	2	4	8
100%	100%	100%	100%
203%	204%	202%	197%
403%	395%	389%	385%
803%	775%	767%	772%

Fig. 1: Percent deltas vs Core Count



32	64	128	256
100%	100%	100%	100%
53%	54%	52%	52%
35%	35%	34%	33%
29%	28%	28%	28%

Fig. 2: Percent deltas vs Input Size

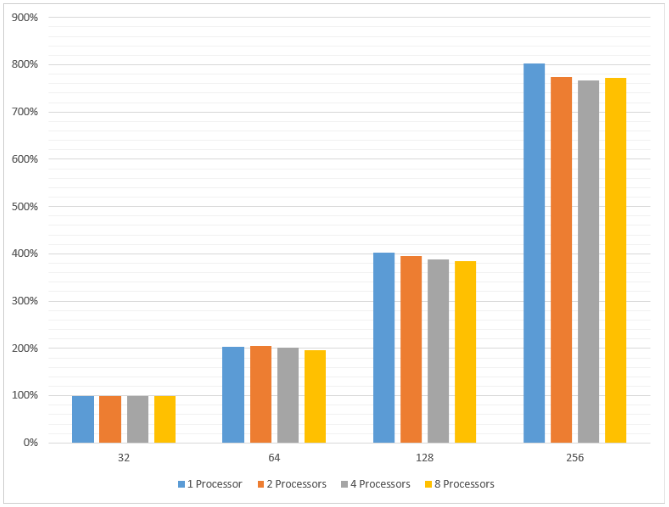


Fig. 3: Runtime delta vs Input Size

File Count	Parallel	Sequential	1.x Delta
32 Files	1.909212533	2.6275413	1.37624348
64 Files	3.682723233	5.158865333	1.400828954
128 Files	7.283922367	10.2794731	1.41125517
256 Files	14.33229347	20.5493898	1.433782377
512 Files	28.83089667	41.0223975	1.422862354
1024 Files	55.89146183	81.83026183	1.464092352
2048 Files	111.1820113	162.9768163	1.465855981
4096 Files	222.8098348	329.5016414	1.478846935

TABLE I: Overall Execution Comparison [3]

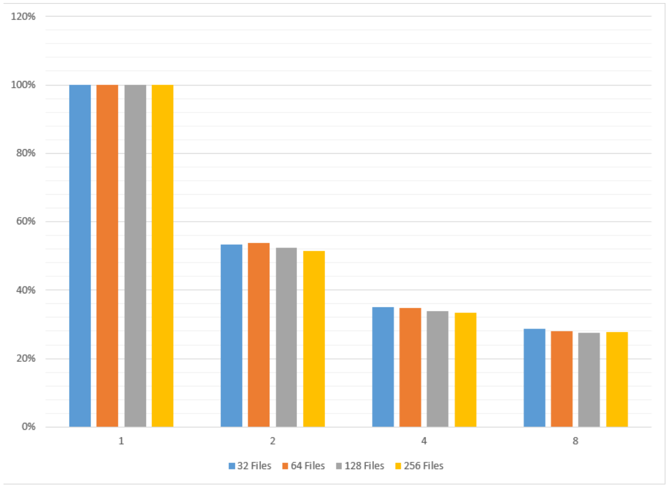


Fig. 4: Runtime decrease vs. Processor Count

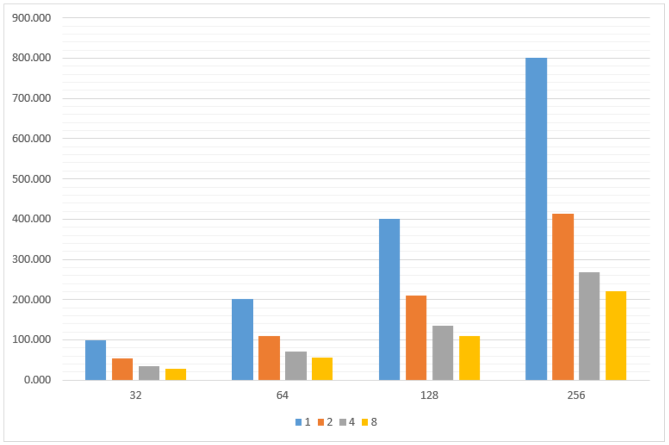


Fig. 5: Runtime increase vs. Input size

VI. Conclusion

We have found the following in our less-naive implementation of MapReduce across many goroutines: Per Fig. 1, we can see that as the number of processors increased, the average runtime of decreased somewhat linearly. From this, we also see that as the number of true processor cores doubled, the execution time almost halved. Per Fig. 2, we see this expressed more readily; core counts 1, 2, 4, and 8 all experience an approximate doubling of runtime as the input sizes increase (100-percent, 203-percent, etc.) However, of note is the decreased true percentage changes as the processor count increased. Note that processor count 1 end at approximately 800-percent of initial runtime, while processor count 8 ends at approximately 775-percent. This is a fairly impressive difference, but to put this in another context of input-size deltas, Fig. 3 shows that, as input sizes increase (double), the percent difference between the input sizes is nearly identical. Following this pattern, running our implementation on increasingly doubled input sizes would see a slowly increasing delta between the single core and multi-core runtimes, implying that, to some extent, a level of scalability has been achieved. Fig'.s 4 and 5 show graphical representations of the differences in execution by input and cores. It seems vaguely exponentially decreasing from this perspective (albeit with a very high constant), and should the number of cores continue to increase, so to would the runtime decrease. Of note, the above table expressing the differences in parallel and sequential execution time shows that there is a very definite decrease in runtime when using our parallel implementation. A figure not shown is an execution attempt made by the group to create a recursive and sub-mapped solution to our MapReduce problem. In it, we changed our single Mapper() function to spawn multiple sub-mappers that ran per-line, as opposed to a single Mapper() goroutine executing for every individual file. It seems that, at least with our implementation, that memory usage shot up more than exponentially; a 1 GB input text file used more than 8 GB of memory to execute correctly(!). Through brief analysis, the cause of this was found to be a sublety of golang that was not carefully accounted for. As mentioned, a key expectation of the golang is that data is to be sent via channels as opposed to methods; that being the case, this implementation actually copied every single string it processed until the machine ran out of memory. The go-approved method for this would have been even more channels to send parsing strings on that channel.

VII. Future Work

There are a few potential optimizations and utilities we found that might improve upon our algorithm. One potential optimization would be to generate multiple reducers per word, instead of only one. This would allow for effectively larger channel buffers, as when a channel fills its buffer it blocks until room is created. It could also be expanded into dynamically resizing reducers, where more reducers are created for a word when its buffer fills. The potential pitfalls of this would involve each word needing a reducer hub or controller, to manage the reducers for a word, distribute incoming tokens, and merge the results when all reducers are done. This could cause bottlenecking; as every request would go through the hub to be distributed across the multiple reducers for a word. Additionally, it would add another level of blocking to the end of the algorithm; each reducer hub would need to block sending its results until every one of its reducers returned a value. A potential utility to expand on our algorithm would be the creation of a generic framework that could take in a map and reduce function from the user and use our algorithm in conjunction with these functions to derive an answer to whatever question. This might slow down the algorithm, as doing so would require changing our mapping logic to operate on the output of the supplied mapping logic, instead of processing the data as its iterated through. As mentioned in the conclusion, one of the largest improvements would be to find a way to implement a finer-grained approach to Map() and Submap(), allowing even more goroutines to execute and totally utilize the increasing number of cores on a machine.

References

[1] ”Frequently Asked Questions (FAQ).” - The Go Programming Language. February 4, 2015. Accessed March 25, 2015. http://golang.org/doc/faq.

[2] ”Tool: Random String Generator [Test Data Creation].” Random String Generator. Accessed March 25, 2015. http://textmechanic.com/Random-String-Generator.html.

[3] Bravendar, Dan. ”MapReduce for Go.” MapReduce for Go. November 24, 2009. Accessed March 25, 2015. http://dan.bravender.net/2009/11/24/MapReduce_for_Go.html.

[4] Dean, Jeffrey, and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI, 2014.