GOMapReduce Parallel Computing Model Implementation on a Cluster of Plan9 Virtual Machines

Yury L. Leokhin\textsuperscript{a}, Andrey S. Myagkov\textsuperscript{a}, Peter B. Panfilov\textsuperscript{a,\textit{b}}

\textsuperscript{a}National Research University – Higher School of Economics, Myasnitskaya St. 20, Moscow 101000, Russian Federation
\textsuperscript{b}University of Applied Sciences Technikum Wien, Hochstaedtplatz 6, Vienna 1200, Austria

Abstract

In this paper, we present results of a computational evaluation of goMapReduce parallel programming model approach for solving distributed data processing problems. In some applications, particularly data center problems, including text processing the programming models can aggregate significant number of parallel processes. We first discuss the implementation of these approaches using both Linux and Plan9 operating system and conduct a comparative scalability study of the both. From these results, we empirically show that, in practical implementation and evaluation of a goMapReduce model, the degree of OS’s support for distributed processing encountered in solving the resulting word counting problem is crucial. We conclude that the goMapReduce approach under Plan9 may be useful in developing a heuristic approach for the data center problems.

Keywords parallel computing; MapReduce; programming language Go; GoLang; Plan9

1. Introduction

MapReduce [1] - is a popular programming model targeted at distributed and parallel processing. Among multiple realizations of MapReduce, the most popular is Apache Hadoop, written in Java. goMapReduce is a relatively new prototype implementation, written in Go in 2013 by a group of researchers from the University of Florida [2]. Their study report has shown encouraging results, and brought us to the research idea of porting their implementation of MapReduce to the Plan9 operating system, studying the characteristics of the implementation, and identifying prospects for application development in goMapReduce on Plan9.

The principle of MapReduce was invented in Google and developed into powerful programming model in response to the growing needs of parallel processing of large amounts of data: the problem of constructing a reverse index, log processing Web requests, analytics on social networks, market analysis, the tasks associated with processing databases (queries, ETL, etc.), physics-based simulation, graph theory analysis etc.. At present, MapReduce is actively being used in machine learning applications [3].

The programming language Go (or GoLang) was developed in the Google company in 2007 as a programming language for contemporary machines of multicore architecture. It is still doubtful whether this is an object-oriented language [4] but it has parallel primitives in the CSP style [5, 6]. The documentation is sufficiently complete, despite the fact that this is a new language. All parallel primitives of the Go language such as asynchronous function calls, channels, and the multilateral connection multiplexor Select deserve attention. The so-called goroutines are used in the Go language which are the functions realized in competition with other goroutines in the same address space. The key advantage of goroutine is its lightness. The main costs associated with the goroutine is the creation of a 4KB stack which can grow if necessary. Thus, the goroutines are much lighter than the classical POSIX-threads (pthreads) which require 2MB of memory by default on Linux/x86-32 [14]. The Channels are essentially queues with locks. Process A can put some data in the channel, and process B can receive these data. In this case, process A trying to add the data to the channel is blocked until process B accepts these data. The multiplexer Select is in fact a structure waiting for any action in one of the channels, i.e., the corresponding code is executed as soon as the data are obtained through one of the channels. Thus, the Select cycle allows the program to read from different channels simultaneously. Moreover, the Go compiler is implemented for different architectures and operation systems, in particular, for Plan9 [7, 15].

Plan9 operating system is based on three principles. First, the resources are named and can be accessed as files in a hierarchical file system. Second, there is a standard protocol called 9p, to access these resources. Third, unbounded hierarchies provided by the various services are linked together into a proprietary hierarchical file namespace. Important properties of Plan9 are maintained due to targeted consistent application of these principles. In many respects, these properties make it feasible to use MapReduce on Plan9: a “standard” MapReduce requires the distributed file system and the close-coupled work of many machines in the same address space that Plan9 allows to do virtually effortlessly. Nevertheless, the first step on the way to a full-fledged implementation of goMapReduce on Plan9 was to make MapReduce run using standard means.

There are number of challenges associated with the current embodiment of goMapReduce that we had to overcome in the course of study. Since goMapReduce is a prototype at this stage, it is still unstable and imposes some restrictions on application developers. For example, if one of the working machines in the system goes off, then the whole task execution stops. Also if the memory overflow occurs somewhere in the system, then the prototype ceases to operate. Moreover, the working machines cannot perform disk read and write operations, and this limits the volume of input data. For example, in the test implementation of the word counting problem with the input data volume of 150MB, the memory overflow occurred in a working machine when system was composed of a pair ‘one master–one worker’. The goMapReduce cannot currently operate as the cascade MapReduce; in particular, it cannot count the citation index. When testing goMapReduce, we discovered several errors associated with the hash-function computations by the adler32 algorithm [8] that we were to fix. In addition, the task distribution format and the hashing function cannot be changed in this implementation by using API - it requires changing the makeup code. As the goMapReduce developers promise, the possibility of changing the data separation/sharing function inside API will appear in the future.

2. Implementation of goMapReduce

2.1. MapReduce model implementation example

According to the MapReduce programming model, which is targeted at large data sets processing and generating in a highly parallel and distributed manner on a cluster, an application program is composed of two functions (procedures or methods), namely: a map function for data filtering and sorting operations and a reduce function for “summary” operations. Both functions are defined with respect to data structured in a special way as a ‘key-value’ pairs. In the map function, the input data are transformed into the ‘key-value’ pairs. In the reduce function, a grouped (with respect to the key) ‘key-set of values’ pairs are used as the input, and the resulting ‘key-value’ pairs appear at the output. For example,
in the prototypical MapReduce application problem of counting the appearance of each word in a text (i.e., word counting application), the map function will generate such pairs as \(<\text{word}>, 1\). In turn, the reduce function accepts pairs of the form \(<\text{word}>,[1, 1, 1, 1, 1]\) as the input, and produces a \(<\text{word}>, 5\) pair as the output.

```go
// The input key, value will be filename, text
// The output would be word, count
func (wc WC) Mapper(key, value string, out chan mr.Pair) {
    strr := strings.NewReader(value)
    s := bufio.NewScanner(strr)
    s.Split(bufio.ScanWords)
    for s.Scan() {
        word := s.Text()
        out <- mr.Pair{word, "1"}
    //fmt.Println(word)
    }
    if err := s.Err(); err != nil {
        fmt.Fprintln(os.Stderr, "reading file :", err)
        os.Exit(-1)
    }
}

// The reducer receives a word, <list of counts>
// It adds up all the counts and outputs a word, combined count
func (wc WC) Reducer(key string, value []string, out chan mr.Pair) {
    count := 0
    for _, v := range value {
        c, err := strconv.Atoi(v)
        if err != nil {
            fmt.Fprintln(os.Stderr, "error converting ", v, " to integer, err: ", err)
            os.Exit(-1)
        }
        count += c
    }
    out <- mr.Pair{key, strconv.Itoa(count)}
}
```

Fig. 1. Implementation of word count application in goMapReduce.

Figure 1 shows the implementation of user-defined functions Mapper and Reducer for counting words. Here, a text is split into words, and each word is counted by the map function (the word is a key). All pairs with the same key are put together and fed to the same call to reduce. The reduce function sums up all of its input values to find the total appearances of that word.

2.2. Multicomputer version of goMapReduce

There are two implementations of goMapReduce: a multi-threaded single computer version and a single-threaded multicomputer version. In our study we focused on the operational principles of the latter.

Figure 2 shows the operational stages of the algorithm for the multi-computer implementation of goMapReduce. Here M denotes the central master machine responsible for the task distribution and collections of results. The worker machines are denoted by W \(\{1, 2, 3\}\). At the starting point, each node (a worker) in the system acquires its own rank which is further used by the master and the workers to distribute tasks. Each machine with goMapReduce recognizes the starting configuration (number, IP-address, and port) of all machines in the configuration file.
At the beginning, the master is connecting to all workers and verifies if they are up and running. Then it reads the input files from the directory and distributes them between all workers. At this point, all workers enter the map phase (become the so-called «mappers»).

After the input data are processed, the worker machines store the results in the “map” structure of the Go language [9] (as an unsorted collection of “key–value” pairs). The intermediate keys are created by the «mappers», and then, according to the model, are sent to the «reducer» (a worker machine executing goMapReduce in the reduce phase). The reducer gets as an input a grouped (with respect to the key) ‘key–set of values’ pairs. For this, it is necessary to collect them from the mapper-nodes. One possible solution is shipping the intermediate data back to the master-node for their further distribution between the reducers. This creates a «bottleneck» in the system and affects its scalability.

To avoid this problem, we choose the function of data separation between the nodes. The hash-function adler-32 [8] is applied to each key of the ‘key–value’ pair of intermediate data. Then the remainder of the hash division by the number of worker machines-reducers is taken. The obtained value indicates the computer node where this particular ‘key–value’ pair is to be processed.

The mappers collect information to decide which of the reducers needs the intermediate data stored in them and send this information to the master. Then, after the whole information sent by all mappers is collected, the master decides which reducer needs information and from which of the mappers. Further, the master sends information to each reducer indicating the mappers from which this reducer must collect information.

After having collected the necessary intermediate data reducers enter the Reduce phase: they reduce the ‘key–set of values’ pair to the ‘key–value’ pair.

3. Experiment description

3.1. Experimental Setup

All test runs of the goMapReduce were accomplished on VirtualBox virtual machines. The virtual machines were allowed maximally 40% of CPU time of a single core of the Intel Core i7 processor and 1024–2048MB of the memory. The machine total main memory was 24GB, and therefore no swapping was required. We were able to run seamlessly 8 virtual machines simultaneously, which allowed us to obtain pretty confident results on the efficiency increase estimates as well as on the upper bounds for this increase in the program.

3.2. Experimental Scenarios

The total computation time of the word counting algorithm has been measured in a series of experiments. Here, we have faced a problem of the time inaccuracy due to a fact that the virtual machine implementation is optimized for maximum computing performance at the expense of the time measurement accuracy. Thus, when running the word counting program with the input file of 20MB on four machines, we obtained two different results from the built-in timer...
for the goMapReduce execution time. Rounding off the results to integer seconds, we saw that the mean time of the program execution was equal to either 13 or 24 seconds. The intrasystem utility ‘time’ [10] counted the real time in the range of 50–58 seconds. At the same time, the usual stopwatch showed approximately 16–18 seconds. This problem was solved by using ‘ssh’ regime to launch the system. The time was counted by the ‘time’ utility [11] in the Linux operation system, and the delay time of the remote run was also calculated.

As the initial data we took the files of an English text and divided them into several data blocks. The English text was taken to avoid possible problems due to different encodings on different OSs. The experiments were launched by automated scripts. The size of the initial data was equal to 5 MB, 35 MB, and 150 MB. For each volume of the initial data, the experiments were carried out for the number of worker machines varying from 2 to 7, and 20 runs were performed for each number of the machines. We also performed two series of experiments. The first series was completed on the virtual machines with Plan9 operating system, and the second one, on VMs with xUbuntu Linux OS. The Go compiler version was 1.2.1.

4. Results and Analysis

The restrictions on the system capability can be estimated by Amdahl’s law which states that, under the assumption of equal speed of all computations in the case where the problem is divided into several parts, the total time of its execution by a parallel system cannot be less than the time required for the execution of the longest fragment [12]. Mathematically, this law is described by the formula

\[
S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}}, \tag{1}
\]

where \(\alpha\) is the fraction of all computations which can be performed only in succession, \(p\) is the number of compute nodes, and \(S_p\) is the speedup determined by the ratio of the time of the program execution on a single machine to the time of the program execution on \(p\) machines.

Figure 3 illustrates the scalability of the MapReduce program execution with the increasing number of machines from one to seven. The graph shows that the scalability increases with an increase in the input data volume regardless the operating systems used. In figure 3, the maximum increase (of scalability, \(S_p\)) by a factor of 1.8 is observed for Plan9-based implementation with input data volumes of 5MB and 35MB.

A bar graph in figure 3 illustrates the execution times of the test problems. It follows from the GoMapReduce report [2] that the increase in the capability agrees well with the obtained results (\(\alpha=0.355\) obtained on Linux for the initial file of 519MB, and \(\alpha=0.358\) obtained in our studies on Plan9 for the initial data of 150MB). In our studies on Linux, the coefficient \(\alpha\) is equal to 0.422 for the input data of 150MB.
4. Conclusions

The problem of the goMapReduce implementation on Plan9 operating systems and the issue of the scalability of Plan9-based solution were the major foci of our research. In the course of our study, we succeeded in running a MapReduce word counting application on a cluster of virtual machines under operating systems Linux and Plan9. The goMapReduce framework operating on Plan9 shows similar scalability compared with the studies performed on the Amazon cloud services and with the results obtained on virtual machines under Linux OS. In our experiments, the goMapReduce exhibits better scalability on Linux than on Plan9 for a small volume of input data (5MB), while for a greater volume of input data, the results are converse. The cause of such a difference was discovered in the further profiling of goMapReduce, as well as some errors were discovered and corrected in the source code of goMapReduce existing implementation.

Nevertheless, goMapReduce has proved to be a good (open source) prototype realizing the MapReduce principles that can be easily ported to Plan9. The execution of the prototype on Plan9, which is comparable in scalability with the one on Linux both on our virtual machines and on Amazon, demonstrates similar performance of applications written in Go on both operating systems.

For the future study, we assume that, in addition to the implementation of the fault tolerance mechanism as it is declared in the goMapReduce report [2], it is expedient to implement some of its parts by using the 9p protocol [13] of Plan9 OS. In particular, the input data should be split between several machines to avoid the bottleneck effect at the stage of task distribution. Moreover, at the intermediate stage of MapReduce, all data are currently stored in the main memory which restricts the input data volume. That is why our maximal volume of input data was 150MB. It is also necessary to develop the data transfer and intermediate storage using file system inside the worker machines.

5. Acknowledgements

The paper summarizes the results of the project «Development and organization of high-technology production of power efficient multiprocessor hard- and software server complexes for the governmental and corporative information systems and data processing centers» which was won in an Open competition between organizations for the right to obtain grants for implementation of complex projects of high-technology production (RF Government Resolution №218, April 9, 2010).

6. References

