РЕАЛИЗАЦИЯ МОДЕЛИ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ GOMAPREDUCE НА ОПЕРАЦИОННОЙ СИСТЕМЕ PLAN9

GOMAPREDUCE PARALLEL COMPUTATION IMPLEMENTATION BASED ON PLAN9 OPERATING SYSTEM

Леохин Юрий Львович / Yury L. Leokhin,
доктор технических наук, профессор, начальник отдела, Национальный исследовательский университет «Высшая школа экономики» /
Doctor of Technical Sciences, prof., head of department,
National Research University Higher School of Economics (HSE),
yleokhin@hse.ru

Мягков Андрей Сергеевич / Andrey S. Myagkov,
аспирант, Национальный исследовательский университет «Высшая школа экономики» /
graduate student, National Research University Higher School of Economics (HSE),
myagkov.as@gmail.com

Аннотация

В статье рассмотрена реализация модели параллельного программирования MapReduce, активно применяемая в распределенных вычислениях. Представлены результаты исследования масштабируемости данной реализации, запущенной на операционной системе Plan9, и указаны ее направления развития.

Abstract

The implementation of the parallel programming model MapReduce, which is used in distributed computation, is considered in the article. The results of the scalability research of the implementation running on the Plan9 operation system are shown and the main lines of the development of this system are pointed out.

Ключевые слова: параллельные вычисления, MapReduce, язык программирования Go, GoLang.

Keywords: parallel computing, MapReduce, a programming language Go, GoLang.

Введение

MapReduce [1] – это популярная модель программирования. Она основывается на распределенных параллельных вычислениях. Существует множество ее реализаций. Наиболее популярная из них – ApacheHadoop, написанная на языке Java. В 2013 году группой исследователей из Университета Флориды был выпущен прототип GoMapReduce [2], написанный на языке Go. Отчет, опубликованный по результатам, свидетельствует о возможности успешно их использования, поэтому было принято решение перенести реализацию MapReduce на операционную систему Plan9, изучить особенности реализации и выявить перспективы развития GoMapReduce на Plan9.

Сам принцип MapReduce был изобретен в компании Google и создан в связи с потребностью параллельной обработки больших объемов данных: задачи построения обратного индекса, обработки логов веб-запросов, аналитики в социальных сетях, анализ рынков, а также задач, связанных с обработкой баз данных (запросы, ETL, etc), моделированием физических процессов и анализом графов. В настоящее время активно ведется решение задач машинного обучения с помощью MapReduce [3].

Операционная система Plan9 построена на трех принципах. Во-первых, ресурсы именуются, и к ним можно получить доступ как к файлам в иерархической файловой системе. Во-вторых, имеется стандартный протокол, называемый 9p, для доступа к этим ресурсам. И, наконец, несвязанные иерархии, обеспечиваемые различными службами, соединяются вместе в единое личное иерархическое

пространство имен файлов. Важные свойства Plan9 обусловлены целенаправленным последовательным применением этих принципов. Во многом эти качества позволяют рассуждать об использовании MapReduce на Plan9: канонический MapReduce требует распределенную файловую систему и тесную работу многих машин в одном пространстве, что Plan9 позволяет реализовать практически из коробки. Тем не менее, первой ставилась задача запуска MapReduce стандартными средствами.

Язык программирования Go (он же GoLang) был разработан в компании Google в 2007 году. Создавался он как язык программирования для современных машин с многоядерной архитектурой, спорно является объектно-ориентированным [4], имеет параллельные примитивы в стиле CSP [5, 6]. Документация является достаточно полной, несмотря на то, что язык новый. Все параллельные примитивы языка Go заслуживают внимания: асинхронные вызовы функций, каналы и мультиплексор многосторонних связей select. В языке Go используются так называемые горутины (goroutine). Горутина – это функция, выполняющаяся конкурентно с другими горутинами в том же адресном пространстве. Ключевое преимущество горутины – ее легковесность. Основные расходы на горутину – создание стэка в 4КБ, который при необходимости может расти. Таким образом, горутины значительно более легковесны, чем классические POSIX-нити (threads, pthreads, etc.), требующие выделения для них по умолчанию в Linux/x86-32 2 мегабайта памяти [14].Каналы являются, по сути, очередями с блокировками. Процесс А может положить данные в канал, а процесс Б может принять эти данные. При этом процесс А, желающий добавить данных в канал, блокируется до тех пор, пока Б не примет эти данные. Мультиплексор select по своей сути является конструкцией, ожидающей действия по одному из каналов. Как только данные придут по одному из каналов, реализуется соответствующий код. Таким образом, цикл select позволяет программе читать из разных каналов одновременно. Кроме того, компилятор Go выполнен для разных архитектур и операционных систем, в частности, для Plan9 [7].

Для реализации MapReduce создается инструмент (например, Apache Hadoop, Amazon Cloud MapReduce, Disco, Phoenix и т.д.), а программисту следует создать интерфейс в виде двух функций: map и reduce. Для выполнения конкретной задачи в модели MapReduce достаточно описать только эти две функции. В функции map происходит преобразование входных данных в пары *ключ-значение* (например, для задачи подсчета количества вхождений каждого слова в тексте функция map будет генерировать такие пары: <слово>:1). На вход функции reduce подаются сгруппированные по ключу пары *ключ-множество значений*, на выходе – результирующие пары *ключ-значение* (например, для вышеуказанной задачи на вход будут подаваться пары типа <слово>:[1, 1, 1, 1, 1]; на выходе будет <слово>:5). Рассмотрим исследуемую нами реализацию GoMapReduce.

Реализация GoMapReduce

У данного варианта есть ряд ограничений по причине того, что на данном этапе GoMapReduce является прототипом. На данном этапе система работает нестабильно. Например, в случае отключения одной рабочей машины выполнение задачи останавливается. Если где-то произойдет переполнение памяти, то прототип не будет работать. Кроме того, рабочие машины не производят чтение и запись на диск и с диска, что ограничивает объем входных данных. Так, например, при нашем запуске подсчета слов с объемом входных данных в 150 мегабайт во время запуска пары «один мастер – один рабочий» у рабочей машины наступало переполнение памяти. На текущий момент GoMapReduce не может выполнять каскадный MapReduce, в частности, реализацию подсчета индекса цитируемости. В ходе тестирования GoMapReduce был обнаружен ряд оши-

бок, связанных с вычислением хэш-функции по алгоритму adler32 [8], что нам удалось исправить. Помимо этого, формат распределения заданий, а также функцию перемешивания в данной реализации невозможно изменить посредством API, не изменяя основной код.

У GoMapReduce есть две реализации: многопоточная на одной машине и многомашинная версия. Мы изучали принцип работы многомашинной версии. В листинге 1 показана реализация пользовательских функций Mapper и Reducer для подсчета слов.

Листинг 1. Реализация подсчета слов в GoMapReduce

```
// The input key, value will be filename, text
// The output would be word, count
func (wc WC) Mapper(key, value string, out chanmr.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 chanmr.Pair) {
        count := 0
        for _, v := range value {
                //fmt.Println("k: ", key, "v:", v)
                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)}
}
```

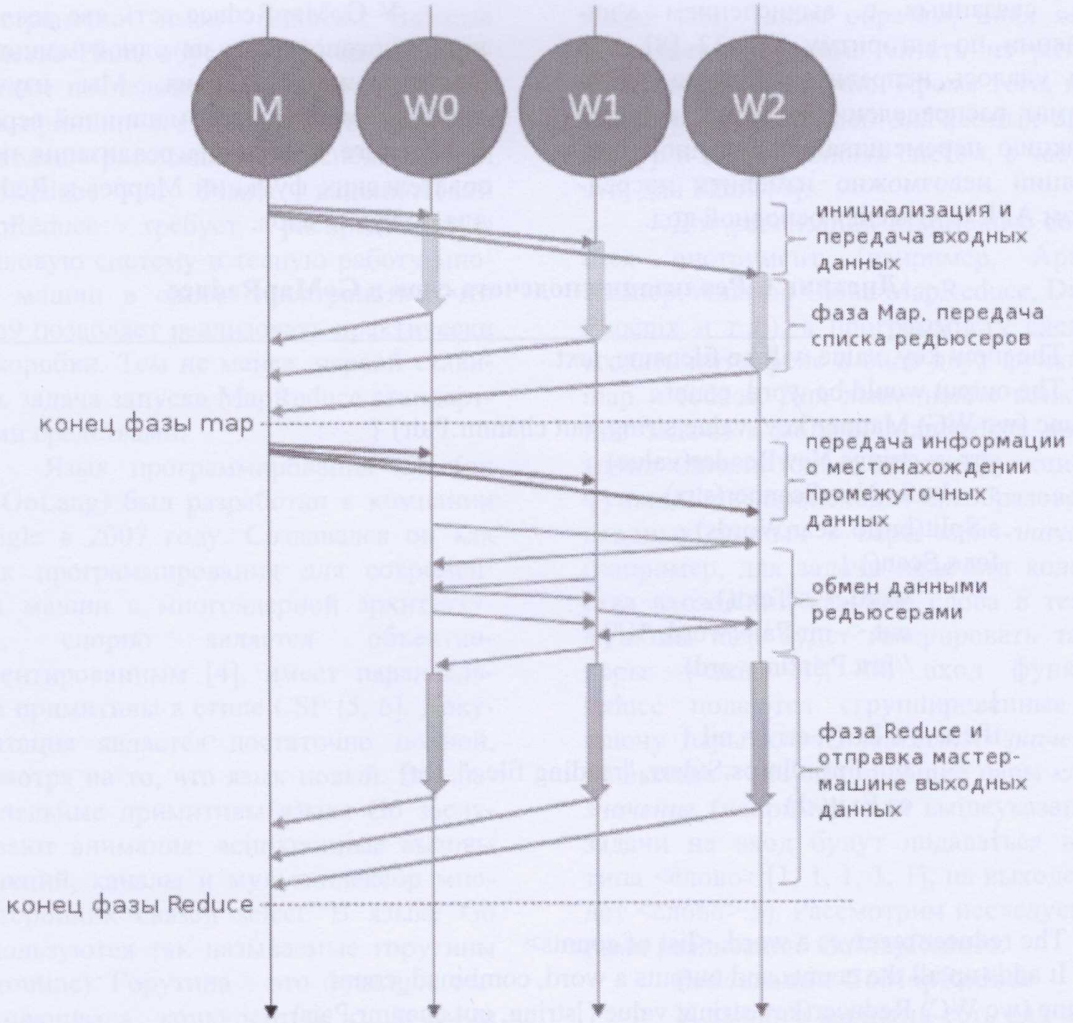

Рис. 1. Схема работы многомашинного варианта GoMapReduce

Схема **работы многомашинного варианта GoMapReduce** показывает стадии работы алгоритма данной реализации MapReduce. Буквой M показана центральная машина-мастер, отвечающая за распределение заданий и сбор результатов. Рабочие машины обозначены как W[0…2…n]. Во время запуска каждый узел (рабочая машина) получает свой ранг, зная который, мастер и другие рабочие машины впоследствии будут распределять задания. Каждая машина с запущенным GoMapReduce узнает конфигурацию запуска (номер, IP-адреса и порт) всех машин из конфигурационного файла.

В самом начале мастер подключается ко всем рабочим машинам и удостоверяется в том, что все они запущены. Затем считывает входные файлы из директории и разделяет их между всеми рабочими машинами. В этот момент все рабочие машины переходят в фазу map (становятся т.н. «мэпперами»). После обработки входных данных рабочие машины хранят вывод в структуре языка Go «map» [9] – несортированной коллекции пар «ключ-значение».

Промежуточные ключи создаются «мэпперами», и затем (с точки зрения модели) должны уходить на «редьюсер» – машину, выполняющую GoMapReduce в фазе reduce. На вход в редьюсер подаются сгруппированные пары по ключам в формате *ключ – множество значений.* Для того, чтобы подать эти пары на вход, необходимо собрать их с узлов-мэпперов. Отправка промежуточных данных обрат-

но узлу-мастеру, чтобы он в дальнейшем распространял их редьюсерам, создаст «бутылочное горлышко» и снизит возможность масштабирования. Чтобы избежать этой проблемы, выбирается функция разделения данных между узлами. К каждому ключу пары промежуточных данных *ключ – значение* применяется хэш-функция adler-32 [8]. Как обещают разработчики GoMapReduce, в будущем появится возможность изменять внутри API функцию разделения данных. Затем берется остаток от деления хэша на количество рабочих машин-редьюсеров. Получившееся значение показывает, на какой машине будет обрабатываться данная пара *ключ – значение*. Мэпперы собирают информацию, чтобы определить каким редьюсерам понадобится промежуточные данные, хранящиеся у них, и отправляют эту информацию мастеру. После получения собранной информации мэпперы отправляют ее мастеру, который должен определить, какой редьюсер требует данные и от какого мэппера. Далее мастер отправляет информацию каждому редьюсеру о том, с каких мэпперов ему нужно собирать информацию.

После сбора необходимых промежуточных данных редьюсеры начинают выполнять стадию «reduce» – уменьшать и обрабатывать данные: из пары «ключ-множество значений» делать пару «ключ-значение».

Реализация эксперимента

Ввиду отсутствия работающего кластера было решено запускать GoMapReduce на виртуальных машинах VirtualBox. Виртуальным машинам предоставлялось максимально 40% процессорного времени одного ядра процессора IntelCore i7 и 1024 – 2048 мегабайтов памяти. Всего в машине было 24 гигабайта оперативной памяти, таким образом, раздел подкачки не использовался. Условно «безболезненно» удалось запустить 8 виртуальных машин, что позволило оценить нам рост производительности и пределы роста для программы.

В ходе выполнения серий экспериментов замерялось общее время выполнения алгоритма. Однако у нас возникла проблема, не дающая адекватно измерить время выполнения задачи. Мы предположили, что такая проблема возникает из-за того, что реализация виртуальной машины оптимизирована под максимально быстрое исполнение в ущерб аккуратности измерения времени. Так, при запуске задачи подсчета слов (входной файл 20 мегабайт) на четырех машинах встроенный счетчик времени GoMapReduce в серии экспериментов выдавал два варианта результата. Округлив до целой секунды, среднее время выполнения задачи было либо 13, либо 24 секунды. Внутрисистемная утилита time [10] считала реальное время выполнения в диапазоне 50-58 секунд. При этом по обычному секундомеру выполнение происходило за 16-18 секунд. Решением этой проблемы явился запуск системы посредством ssh. Подсчет времени мы вели с помощью утилиты time [11], но уже в операционной системе Linux. При этом вычитали время задержек удаленного запуска.

В качестве исходных данных мы взяли файлы англоязычного текста и разбили их на блоки данных. Для того, чтобы предотвратить возникновение проблем, связанных с обработкой разных кодировок на разных ОС, взят был англоязычный текст.

Эксперименты запускались с помощью автоматизированных скриптов. Размер входных данных был 5, 35 и 150 мегабайт. Для каждого объема входных данных были запущены эксперименты с количеством рабочих машин от 2 до 7. Для каждого количества машин было произведено по 20 запусков. Мы провели две серии экспериментов. Первая проводилась на виртуальных машинах под управлением Plan9, вторая — xUbuntuLinux. Версия компилятора Go – 1.2.1.

Результаты

Для оценки ограничения производительности системы можно использовать закон Амдала, который говорит о том, что при условии равной скорости

всех вычислителей, когда задача разделяется на несколько частей, суммарное время её выполнения на параллельной системе не может быть меньше времени выполнения самого длинного фрагмента [12]. Математически он описывается формулой,

$$S_p = \frac{1}{\left(\alpha + \frac{1-\alpha}{P}\right)},$$

где α – доля от всех вычислений, которая может быть получена только последовательными расчетами, p – количество задействованных узлов, а Sp – ускорение, определяемое отношением времени работы задачи на одной машине ко времени выполнения задачи на p машинах.

На рис. 2 показаны ускорения работы задачи MapReduce на машинах от одной до семи. Из графика видно, что с ростом входных данных масштабируемость растет. Для каждой операционной системы Sp увеличивается с ростом входных данных: наибольшая разница наблюдается для Plan9 5МБ и 35МБ: в 1,8 раз.

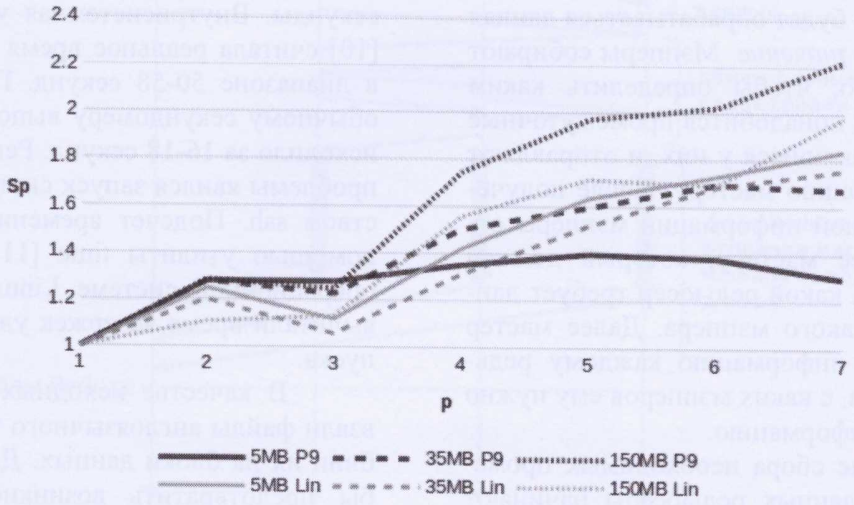

Рис. 2. Ускорение вычислений при увеличении количества рабочих машин

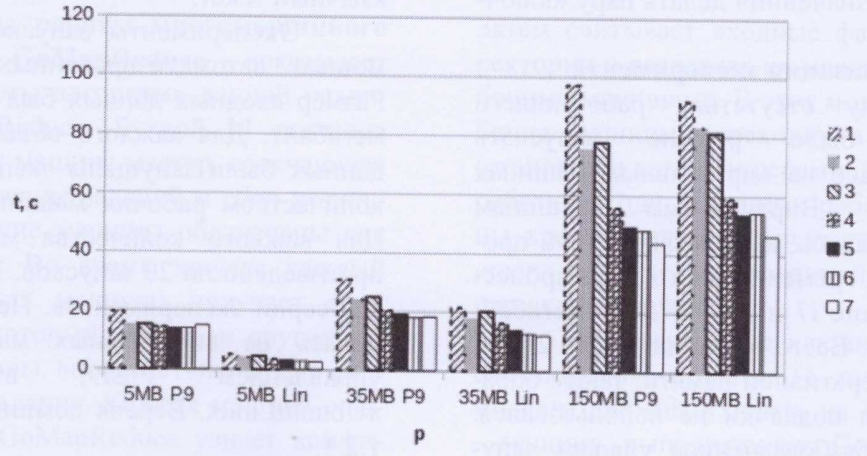

Рис. 3. Вычисления подсчета слов

На рис. 3 показана гистограмма, на которой отражены времена исполнения тестовых задач. Из отчета GoMapReduce: MapReduceinGoLang [2] следует, что повышение производительности сходно полученным результатам (α=0,355 в случае

запуска под Linux с исходным файлом в 519 МБ в вышеупомянутом отчете; α=0,358 в случае наших исследований под plan9 со входными данными в 150 МБ). Из нашего исследования под Linux коэффициент α равен 0,422 для входных данных размером 150 МБ.

Выводы

В исследовании удалось запустить прикладную задачу на кластере виртуальных машин, работающих под операционными системами Plan9 и Linux, измерена масштабируемость. Фреймворк GoMapReduce, запущенный на Plan9, показывает сходную масштабируемость по сравнению с исследованиями, запущенными на облачных сервисах Amazon, а также с его же результатами под Linux на виртуальных машинах. Однако на малом объеме входных данных (5МБ) GoMapReduce показывает большую производительность под Linux по сравнению с Plan9. На большем объеме результаты обратные. Причину такой разницы удастся выявить при дальнейшем профилировании GoMapReduce.

Кроме того, были исправлены ошибки в исходном коде GoMapReduce. При этом следует констатировать, что GoMapReduce показал себя как удачный и единственный прототип в открытом доступе, реализующий принципы MapReduce и легко переносимый на Plan9. Работа прототипа в Plan9, сравнимая по масштабируемости с работой под Linux на наших виртуальных машинах и Amazon, показывает, что работа программ на языке программирования go в Plan9 сходна по скорости с Linux.

Мы считаем, что помимо реализации механизма отказоустойчивости, заявленного в отчете GoMapReduce [2], следует реализовать некоторые части с помощью протокола 9p [13]. В частности, входные данные задач раскладывать на несколько машин, чтобы не создавать узкое место на этапе распределения задач.

Кроме того, в настоящий момент, на промежуточном этапе MapReduce все данные хранятся в оперативной памяти, что создает ограничения по объему входных данных. Так, в результате этого ограничения нам пришлось остановиться на максимальном объеме в 150 МБ. Также следует организовать переброску и промежуточное хранение данных, основываясь на файловой системе внутри рабочих машин.

Статья написана по результатам выполнения проекта «Разработка и организация высокотехнологичного производства энергоэффективных многопроцессорных аппаратно-программных серверных комплексов для государственных и корпоративных информационных систем и центров обработки данных», выигранного в рамках Открытого конкурса по отбору организаций на право получения субсидий на реализацию комплексных проектов по созданию высокотехнологичного производства (Постановление Правительства РФ от 09 апреля 2010 г. № 218).

Литература

1. S. G. Jeffrey Dean. «MapReduce: Simplified Data Processing on Large Clusters», OSDI'04 Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation. – Т. 6. – PP. 10-10, 2004.

2. J. N. K. N. M. S. K. S. T. T. Bethu J. «GoMapReduce: MapReduce in GoLang», 2013.

3. «MapReduce Patterns, Algorithms, and Use Cases – Higly Scalable Blog», 01 02 2012. / [Электронный ресурс]. – Режим доступа: URL: http://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/.

4. S. Francia. «Is Go an Object Oriented language? », 09 06 2014. / [Электронный ресурс]. – Режим доступа: URL: http://spf13.com/post/is-go-object-oriented.

5. C. A. R. Hoare. «Communicating sequential processes». – Communications of the ACM. – Т. 21. – № 8 – PP. 666-677, 1978.

6. «Effective Go» / [Электронный ресурс]. – Режим доступа: URL: http://golang.org/doc/effective_go.html.
7. «Go Porting Efforts». / [Электронный ресурс]. – Режим доступа: URL: http://go-lang.cat-v.org/os-ports.
8. «Adler-32». / [Электронный ресурс]. – Режим доступа: URL: http://en.wikipedia.org/wiki/Adler-32 (дата обращения: 14 05 2014).
9. C. Doxsey, An Introduction to Programming in Go, 2012.
10. Bell Labs. «Plan 9 /sys/man/1/time». / [Электронный ресурс]. – Режим доступа: URL: http://plan9.bell-labs.com/magic/man2html/1/time (дата обращения: 11 06 2014).
11. «Time (1) - Linux man page». / [Электронный ресурс]. – Режим доступа: URL: http://linux.die.net/man/1/time.
12. G.M. Amdahl. «Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities», 1967.
13. «Map Reduce Go in University of Florida». / [Электронный ресурс]. – Режим доступа: URL: https://github.com/EEL6935022ATeam10/GoMapReduce.
14. «PTHREAD_CREATE(3) - Linux Programmer's Manual». / [Электронный ресурс]. – Режим доступа: URL: http://man7.org/linux/man-pages/man3/pthread_create.3.html.