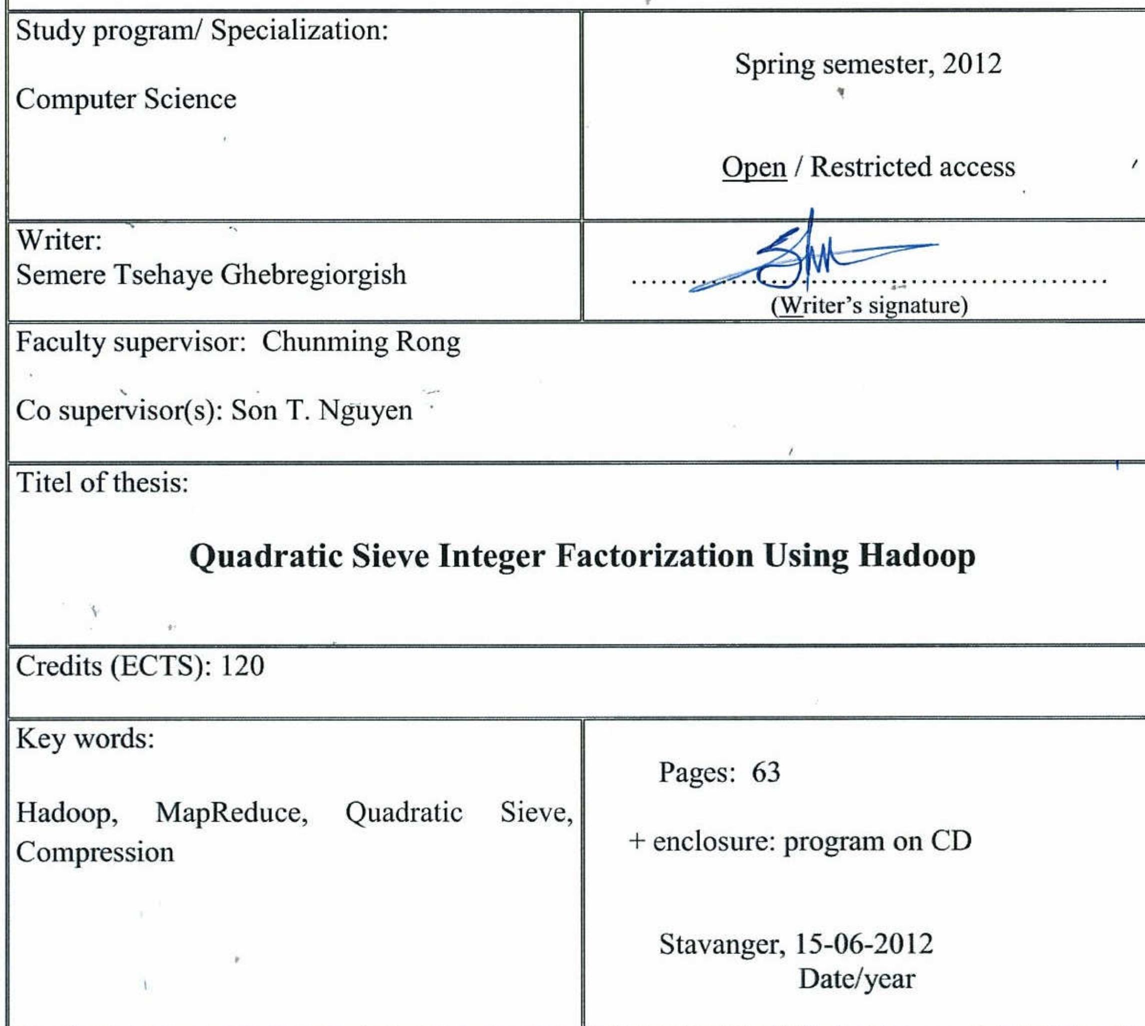
University of
Stavanger

Faculty of Science and Technology

MASTER'S THESIS



Study program/ Specialization: Computer Science	Spring semester, 2012 <u>Open</u> / Restricted access
Writer: Semere Tsehaye Ghebregiorgish	…………………………………… (Writer's signature)
Faculty supervisor: Chunming Rong Co supervisor(s): Son T. Nguyen	
Titel of thesis: **Quadratic Sieve Integer Factorization Using Hadoop**	
Credits (ECTS): 120	
Key words: Hadoop, MapReduce, Quadratic Sieve, Compression	Pages: 63 + enclosure: program on CD Stavanger, 15-06-2012 Date/year

Frontpage for master thesis
Faculty of Science and Technology
Decision made by the Dean October 30th 2009

Quadratic Sieve Integer Factorization using Hadoop

Semere Tsehaye Ghebregiorgish

June 15, 2012

Contents

Preface 1

1 Introduction **2**
1.1 Motivation 2
1.2 Related Work and Contribution of the Thesis 3
1.3 Organization of the Thesis 4

2 The Basics of Integer factorization **5**
2.1 introduction 5
2.2 RSA and Integer Factorization 6
2.3 Selected Special-Purpose Algorithms 7
2.4 Selected General-Purpose Algorithms 10

3 Quadratic Sieve Algorithm **12**
3.1 Introduction and the Algorithm 12
3.2 Example 13

4 Hadoop and MapReduce **17**
4.1 Motivation 17
4.2 What is Hadoop 18
4.2.1 MapReduce Programming Model 19
4.2.2 Hadoop Distributed File System (HDFS) 22
4.3 Data Compression 26

5 Implementation **28**
5.1 Program implementation 28
5.1.1 Choice of Factor Base Size 29
5.2 Cluster Setup 31
5.2.1 Hadoop Configuration 32

5.2.2 Data Compression configuration 33
5.3 Numbers Used 35

6 Results and Analysis 37
6.1 Size of Generated Data 37
6.1.1 comparison between new file size and previous one . . 38
6.2 Performance of Data collection and processing time 40
6.2.1 Performance improvement from the previous result . . 41
6.3 Effect of Cluster Size on Performance 43
6.4 Effect of Data Compression and Splitting Data into files . . . 44
6.4.1 Compressing the Input Data 45
6.4.2 Compressing the Output of Map 47

7 Discussion 48
7.1 Limitations of using Hadoop as a tool 48

8 Conclusion and Future Work 51
8.1 Conclusion 51
8.2 Future Work 52

List of Figures

3.1 Quadratic Sieve Algorithm 14

4.1 Map and Reduce Method in Nutshell. 19
4.2 Quadratic Sieve Algorithm 20
4.3 MapReduce Word Count 21
4.4 Overview of the HDFS Architecture 23

5.1 The effect of factor base size choice 29
5.2 comparison of factor bases 31

6.1 File Size trend in GB vs digit size 38
6.2 Imrpvement of file Size in the thesis 39
6.3 Imrpvement of file Size in the thesis 42
6.4 Factoring 36-digit in different cluster setting 44
6.5 Improvement of Compression 46

7.1 The two Phases in series (Hadoop's approach) 49
7.2 Doing the two phases of QS in parallel 49

List of Tables

4.1 Compression formats supported by Hadoop 26

5.1 Comparison of factor base sizes 30
5.2 Selected Parameters used in the configuration 33
5.3 Composite numbers and their corresponding factors 36

6.1 Size of File Generated for each number 38
6.2 File Size Improvements from previous works 39
6.3 Performance measurement for 11-node cluster 40
6.4 Performance improvement after optimizations 41
6.5 Performance comparison when factoring 40-digit number in different cluster sizes 44
6.6 Effect of Compression 45

Abstract

Integer factorization problem is one of the most important parts in the world of cryptography. The security of the widely-used public-key cryptographic algorithm, RSA [1], and the Blum Blum Shub cryptographic pseudorandom number generator [2] heavily depend on the presumed difficulty of factoring a number to its prime constituents. As the size of the number to be factored gets larger, the difficulty of the problem increases enormously. This fact has led to the development of many different algorithms to attack bigger number within polynomial time. However, there is no known efficient algorithm for very large numbers, to shake up the security of real-world cryptographic applications.

The fastest factoring algorithms to factor general numers are General Number Field Sieve (GNFS) and Quadratic sieve (QS). While QS is regarded as the fastest one for integers less than 110 digits, GNFS is by far the fastest for very big numbers. As of December 2009, the biggest number factored is 768-bit, 232-digit number taken from RSA challange [6] by using GNFS algorithm over a period of two years. The success came after a long hard work and complex programming development. This shows the complexity and difficulty of the Integer factorization problem.

As the size of the number to be factored increases, the data required to factor that number equally increases. Hadoop-MapReduce is a powerful tool for processing for such magnitude data in a reasonable time. Because of this advantage, the thesis investigates the possiblity of attacking the problem in a distributed fashion using MapReduce programming paradigm on Hadoop. A single quadratic sieve algorithm written in Java was used to test different composite numbers. The results were carefully analyzed, then the strength and limitations of the approach are thoroughly discussed.

Acknowledgements

I would like to thank professor Chunming Rong, my supervisor, for his advises and for providing necessary facilities in the project room which are very helpful to successfully finish the thesis. My deepest gratitute goes to Son T Nguyen, Post-Doc Reasearcher at UiS, for his relentless support and insightful comments. He was always available whenever i wanted to see him. It was him who first allowed me to work on the topic. I would also like to thank PHd, Tomasz Wiktor Wlodarczyk for persistently organizing meeting every friday through out the thesis work.

Last but not least, I would like to thank my family and friends for making my dream come true through their flowing love and encouragement.

Preface

This thesis has been submitted in partial fulfillment of the requirements to complete the Master of Science (M.Sc.) degree at the Department of Electrical and Computer Engineering at the University of Stavanger (UiS), Stavanger Norway.

The work has been done at the department of Computer Science under the supervision of Prof. Chunming Rong and Son T. Nguyen.

The initial work was published in Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science (IEEE CloudCom 2011), Nov 29 - Dec 01, 2011 [3]. And further studies were done during computer science project autumn course. As discussed in latter sections, further code and Hadoop configuration optimizations were performed during the writing of the thesis and the result is improved significantly. Furthermore, in order to reduce the requirement of disk space, data compression option is considered and evaluation is made.

Most of the Literature review part in the thesis is based on previous submitted report, which was wrote by the same author of the this thesis. Chapter 2, sections 2.1 through 2.3, explaining the basics of integer factorization is based on report of Computer Science Project Course (MID 240) sumitted to UiS. The discussion on Quadratic Sieve Algorithm (chapter 3) and Hadoop/MapReduce (Chapter 4) with the exception of section 4.3 are based on the report of the course, with only small modifications.

I would like the reader to be aware that whenever I say "previous works", it implies to [3] and the work done during computer science project. To differentiate between the two, I will refer initial work to the former, and project work to the latter.

June 15, 2012

Semere Tsehaye Ghebregiorgish

Chapter 1

Introduction

1.1 Motivation

Integer factorization is one of the ancient mathematical problems, meaning to describe a compoiste number interms of its prime constituents. In other words, it is the decomposition of a composite number down to smaller prime numbers. Multiplying all these primes gives back the original composite.

The Integer factorization problem is regarded very tough one, especially if the composite is very large and is a product of two almost equal-sized primes. we call such numbers, semiprimes. Until today, the largest semiprime to be factored is 768-bit (232-digit), yielding result after almost two years. They used General number Field Sieve (GNFS) algorithm to attack the number, therefore this makes the algorithm the fastest to factor general-purpose numbers. Note that there are numbers with special behavior and they can easily be factored by exploiting their special structure. In 2007, a 1039-bit number was factored based on the special form of the number[4]. Otherwise, there is no any algorithm which efficiently factors a general number within a polynomial time. Additionally, due to GNFS algorithm's extreme complexity and difficulty to understand, it has very small audience and is confined to people who are experts in mathematical fields such as number theory and algebra. This shows how difficult the problem is. isn't it?

But why do we care then about Integer factorization? The problem resides at the heart of most widely-used crytographic algorithms, such as RSA public-key exchange, RSA digital signature schemes, Blum Blum Shub

Random number Generator. These algorithms use a number which is product of two prime numbers. The idea behind is that it is not easy to find the two primes given the composite number. Recovering the primes within reasonable time would annul the security of these algorithms and inturn compromise the dependability of applications that use the algorithms. For example, TLS/SSL encryption [26] which is used to establish secure TCP/IP connections over WWW heavily depends on RSA algorithm. Therefore, if large integers are factored quick, then there will no more be secure connection over the Internet, unless the mechanism is changed. Due to these fact, it is worthwhile to do research on integer factorization to make sure the confidentiality, authentisity and integrity of information moving accross the Internet.

Since the Introduction of computers, many mathematical problems have been solved by using computers as a tool. Very fast computers have been manufactured and as a result, many mathematical problems have been solved. However, even the fastest computer cannot factor a number beyond a certain size. Recently, the idea of distributed systems have become popular. Tanenbaum defines the concept as, " Distributed is a collection of independent computers that appears to its users as a single coherent system " [7]. It makes someone to use the power of all computers in the system as if the person is dealing with one machine. Different frameworks have been developed and Hadoop is one of them. Hadoop is a powerful tool for data-intensive applications. Hence, the motivation is to make use of this framework to attack very big number.

1.2 Related Work and Contribution of the Thesis

There are a number of papers related to the quadratic sieve algorithm with its variants, but not something in relation to Hadoop. The first paper we find is in [8] written by Javier Tordable. The author gives basic implementation of quadratic sieve in Hadoop and Maple and compares the results. His main goal is to show that the conecpt of MapReduce can solve the Integer factorization. The paper does not make detailed study concerning performance. This paper is the starting point for the thesis, so it can be considered more as work-in-parallel, instead of related work.

The paper [9], tries to show that the Hadoop implementation of Integer

factorization is much easier and scalable use a large number of commodity computers. The analysis are nicely presented, but the goal was just to compare that distribution is better than non-distribution, not the concept related to cracking big number. Numbers which have digit sizes less than 14 were mostly used during the test.

The papers [10], [11], [12], and [13] explains on fast implementation of Quadratic Sieve algorithm in Memory. They also talk about running the algorithm on multiple processors.

In this thesis thorough study have been made on dealing the integer factorization problem in Hadoop/MapReduce. It presents what are the strengths and weakneses of the approach. Tests were done from different angles and it spots the limiting factors in the approach. Therefore, it can serve as a reference to someone on how to deal the problem in Hadoop.

1.3 Organization of the Thesis

The thesis is organized in the following way

Chapter 2 presents the underlying concepts of Integer factorization that are helpful in understanding the Quadratic Sieve Algorithm. In this chapter, different factoring algorithms are discussed.

Chapter 3 discusses the Quadratic Sieve algorithm. In the chapter an example is given to make the ready easly master the algorithm.

Chapter 4 describes the Hadoop framework and the MapReduce programming model. It explains what are the basic constituents of Hadoop, how MapReduce works and others.

Chapter 5. presents the implementation part of the thesis. It discusses the hardware and software setup of the implementation.

Chapter 6 discusses the results obtained from results. The explanation is given from different point of view.

Chapter 7 gives the important discussions over all the tests made. It presents the strengths and limitations of the approach.

Chapter 8 concludes the thesis and also incites what can be done in the future.

Chapter 2

The Basics of Integer factorization

2.1 introduction

Integer factorization, also called prime factorization, is the process of finding prime numbers of a number; in which when multiplied together they yield the original number. It simply means decomposing a composite number down to its prime number constituents. At the moment, it is extremely difficult to find the primes of a very large number. Particularly, factoring semiprimes[1] are considered the hardest ones to factor. The fact that there is no efficient integer factorization algorithm for large numbers until today shows the complexity of the problem. So far, the largest semiprime known to be successfully factored is a 232-digit (768-bit) number [5]. The process took about 2 years to finish by employing the fastest known algorithm called General Number Field Sieve. In this report the focus will be on Quadratic Sieve algorithm.

The Integer factorization problem lies at the heart of cryptography such as RSA. The strength of one cryptographic algorithm depends on how quickly the required prime numbers can be found. Finding the factors of a number within reasonable time would make the security of many current cryptographic algorithms highly compromised, and in turn applications that employ these algorithms. For example, e-commerce systems highly rely on cryptographic algorithms to ensure confidentiality, authenticity, and in-

[1] A semiprime is a natural number that is the product of two prime numbers

tegrity of information going through the Internet. Breaking the algorithms means endangering the financial transactions taking place on the Internet in every single minute.

Integer factorization algorithms can be broadly classified into two groups; *special-purpose* and *general purpose algorithms*. While special-purpose algorithms are designed for numbers that fulfill specific behavior requirements, general-purpose are designed to work with any number. Because of the additional behavioral requirements, the running time of special-propose algorithms depend on particular behavior of the number to be factored in addition to its size. For example, if a composite number is a product of two prime numbers, and these prime numbers are close to each other, then it might be efficiently factored by one of the special-purpose algorithms called Fermat method. On the other hand, the running time of general-purpose algorithms solely depend on the size of the number and such algorithms are the types used to factor RSA numbers [6].

Among the general-purpose algorithms, the General Number field Sieve (GNFS) and the Quadratic Sieve (QS) algorithms are the fastest modern algorithms in practice. While the former is the most efficient one for numbers larger than 100 digits, the latter is best for numbers smaller than 100 digits.

In this chapter, I will discuss some of the factoring algorithms which are the basis for quadratic sieve algorithm. However, before proceeding, I would like to talk about the importance of prime numbers in RSA in relation to the problem

2.2 RSA and Integer Factorization

RSA [22](section 9.2) is one of the most popular public key cryptographic algorithms used in electronic commerce. Message exchange between two parties using RSA involves three steps: key generation, encryption on sender side and decryption on receiver side.

Suppose Alice wants to send an encrypted message to Bob. Then the process goes like this:

1. **Key Generation**: both Alice and Bob generate a public/private key pair by

    - Selecting two distinct large prime numbers at random: p and q.

The numbers are of similar bit-length

- Compute $n = pq$ where n is used as the modulus for the public and private keys
- Compute $\phi(n) = (p-1)(q-1)$
- Randomly select an integer e, such that $1 < e < \phi(n)$ and $gcd(e, \phi(n)) = 1$ [2]
- Solve the equation $d.e = 1 mod \phi(n) \Longrightarrow d = e^{-1} mode \phi(n)$
- Finally publish their encryption key $< e, n >$ and keep secret decryption key $< d, n >$. Now, letâĂŹs assume the public key of Bob, the message receiver, be $< e, n >$ and the private key of Bob be $< d, n >$. In this case, $< e, n >$ is accessible to anyone (e.g. Alice) who wants to send encrypted messages to Bob while d is kept secretly by Bob.

2. **Encryption**: To encrypt a message M for Bob, Alice has to compute $C = M^e(mod(n))$ where $0 \leq m < n$. and then send C to Bob. Note that the message M has to be converted to an integer using some reversible technique

3. **Decryption**: Bob then can decrypt C and recover the original message M by computing $M = C^d(modn)$. No one except Bob can decrypt C since d is only known to Bob.

The strength of RSA lies in choosing large enough public key. Because of factoring large numbers is a hard problem, full decryption is thought to be infeasible. In practice, the typical RSA keys are 1024 - 2048 bits long, and so far the largest number factored is 768 bits long. However, 1024 bit long could be factored in the near future and make the current length unsecure.

2.3 Selected Special-Purpose Algorithms

i. **Trial Divison**

Trial division is the easiest factorization algorithm to understand and yet is the fastest method for small composites. As its name indicates

[2] gcd refers to algorithm to find greatest common divisor for two numbers

it divides the number to be factored with possible factors to see if the reminder is zero.

The algorithm tries to factor an integer N, by checking if the number can be divided by an integer greater than 1 and less than $\sqrt{N}$. To perform the trial division algorithm, one simply checks whether s divides N for s = 2, ... , $\lfloor \sqrt{N} \rfloor$. When such a divisor s is found, then $t = N/s$ is also a factor, and a factorization has been found for N.

In fact, each divisor s found by trial division is prime. Therefore, composites can be skipped and only primes can be checked.

Let's try to illustrate the trial division algorithm using an example. Suppose, we want to factor an integer $N = 18525$

Solution: $\sqrt{N} \approx 136.11$ then $\lfloor \sqrt{N} \rfloor = 136$.

This implies that we can make trial division from s = 2, 3, 4, ..., 136. Again, since each divisor is prime, we ignore all composites in the range. Therefore, we make trial division on the 32 prime numbers distributed from 2 to 136.

Below is given each steps how trial division is performed

- 18525 is not divisible by 2
- 18525 is divisible by 3; so replace it with $18525/3 = 6175$ and print 3
- 6175 is not divisible by 3;
- 6175 is divisible by 5; replace with $6175/5 = 1235$ and print 5
- 1235 is divisible by 5; replace with $1235/5 = 247$ and print 5 again
- 247 is not divisible by 5
- 247 is not divisible by 7
- 247 is not divisible by 11
- 247 is divisible by 13; replace with $247/13 = 19$ and print 13
- 19 is not divisible by 13
- 19 is not divisible by 17
- 19 is divisible by 19; replace $19/19 = 1$ and print 19

This way we have learned that 18525 is 3 times 5 times 5 times 13 times 19.

You can imagine how cumbersome it can be for very big composite number.

ii. **Fermat's Factorization Method**

Fermat's factorization method [23] is another special method algorithm that tries to factor a composite number N, through representing it as a difference of two squares. In other words, it tries to resolve $N = a^2 - b^2$ where a and b are integers. From algebra, it is well known that $a^2 - b^2$ can be expressed as $(a - b)(a + b)$. Hence, if $N = (a - b)(a + b)$ then $a - b$ and $a + b$ are the factors of N.
For example, if we take $N = 8051$ and if we try to factor it using trial division, it would be a waste of time by making divisions that don't take us to the right result again and again. However, through Fermat's method, let's take first take $\left\lceil \sqrt{8051} \right\rceil = 90$. We then can write $8051 = 90^2 - b^2$, and after solving the equation, we find that $b = 49$. Luckily, 49 is a perfect square of 7. This means $8051 = 90^2 - 7^2 = (90 - 7)(90 + 7)$. We finally find the factors to be 83 and 97. This is definitely faster than the simple trial division. But, the biggest challenge is that it is not always easy to find difference of two squares. For example, taking $N = 1649$ wouldn't let us easily find squares of number. $\left\lceil \sqrt{1649} \right\rceil = 41$. If we express it, as $N = a^2 - b^2$, then we get $41^2 - 1641 = 32$. However, 32 is not a perfect square. Therefore, we keep on incrementing a by 1, until we get a perfect square. The steps are shown below:

$41^2 - 1649 = 32$
$42^2 - 1649 = 115$
$43^2 - 1649 = 200$
$44^2 - 1649 = 287$
. . . .
. . . .
$57^2 - 1649 = 1600$.

When $a = 57$ we find b which is a perfect square. We can then easily find the factors to be $(57 - 40) = 17$ and $(57 + 40) = 97$. This is not better than trial division, for it takes a lot of time.

A small improvement has been made to speed up the method by taking mod relationship with N. The approach is trying to find an integer a such as $a^2 modN = b^2$, where b is another integer number. This is equivalent to $(a-b)(a+b)modN = 0$ and thus, $N = gcd(a+b, N)(N = gcd(a + b, N))$. This factorization is trivial if a...............b mod N. In that case, the method tries different numbers until it gets a non-trivial pair of factors of N.

In the example above, if we take $a = 41$ and $a = 43$ and if we multiply their respective bâĂŹs, we get perfect square.

$(41x43)^2 \equiv 32x200 \equiv 80^2(mod(n))$
$(114)^2 \equiv 80^2(mod(n))$

2.4 Selected General-Purpose Algorithms

i. **Dixon's Factorization Method**

Dixon's method [24] is an improvement to the Fermat's method through allowing a much weaker condition. The algorithm introduces the concept of factor base, which is going to be the corner stone in quadratic sieve algorightm. Factor base is a set of small prime numbers which are usually used algorithms that involve sieving. In Fermat's method, for any randomly chosen x, we were checking the computation $x^2mod(N)$ to be a perfect square. However this task of finding congruence of squares is almost impractical for large numbers, as there are $\sqrt{N}$ squares less than N. For very big numbers, trying every possible $\sqrt{N}$ squares would be a wastage of resource. Dixon's method reduces this number by Introducing a weaker condition as explained below.

The first steps to factor a composite number N is to first chose a bound B, and a factor base P, which is a set of prime numbers less than or equal to B. Then randomly pick a positive integer z such that $z^2mod(N)$ is B-smooth. A positive integer is called B-smooth if none of its prime

factors is greater than B . mathematically it can be writen as

$$z^2 \equiv \prod_{p_i \in P} {p_i}^{q_i} \mod N \tag{2.1}$$

Then after collecting relations a little bit more than the size of the factor base, check a combination on the right hand side that their multiplication is a square. Remeber, In order for a number to be a square, the exponents of the primes on the right hand side must be even. You can use Gaussian elimination to determine this step. Finally the calculation produces a congruence of the form $a^2 \equiv b^2 \mod N$ and the Fermat theorem can then be applied. For detailed explanation and example, please refer [3]

Chapter 3

Quadratic Sieve Algorithm

3.1 Introduction and the Algorithm

Quadratic sieve algorithm is an Integer factorization algorithm developed by Carl Pomence as an improvement to Dixon's factorization method[25]. As of today, it is the second fastest algorithm after General number field sieve and still is regarded as the fastest for numbers less than 100 digits. Since its inception, it is developed into various forms, and we see today implementations of the different variants available. It is a general purpose algorithm where its running time is soleley dependent on the sizes of the input numbers.

The quadratic sieve algorithm has two phases: the *data collection* and *data processing* phase. During the data collection phase, all necessary information that may lead to factorization of a number is gathered. These information are pairs of integers x and $Q(x)$ that meet the condition $x^2 \equiv y^2 \bmod N$. During data processin phase, the data is placed in matrix then the algorithm attempts to find the factors by search for congruence of squares.

As in Dixon's factorization method, the basic idea is, in order to factor a number N, find two numbers x and y such that $x^2 \equiv y^2 \bmod N$ and $x \not\equiv \pm y \bmod N$. This can be implied as $(x-y)(x+y) \equiv 0 (modN)$, and can be simply computed $(x-y; N)$ using the Euclidean Algorithm to check if this divisor is nontrivial. At least, there is half chance that the factor is nontrivial.

During the first data collection pahse, the algorithm attempts to find

pairs of integers x and $Q(x)$ that satisfy the condition $x^2 \equiv y^2 \bmod N$. It chooses a set of primes called *factor base*, and tries to find x such that the remainder of $Q(x) = x^2 \text{mod } N$ factorizes completely over the factor base. Then x values are said to be smooth over the factor base. The steps of the Quadratic sieve are described briefly in 3.1. One technique used by QS to speed-up finding the above relations is to take x as close as the square root of n. This secures that $Q(x)$ is smaller and as a result have greater chance to being smooth.

$$Q(x) = (\left\lceil \sqrt{N} \right\rceil + x)^2 - N \text{ (x issmall integer)} \tag{3.1}$$

$$Q(x) \approx 2x \left\lceil \sqrt{N} \right\rceil \tag{3.2}$$

Simply increasing the size of the factor base can also increase the chance of smoothness.

next step is to compute $Q(x_1), Q(x_2), ..., Q(x_k)$. Deciding the values for x_i will be shown in the example. Then from the set of values of Q(x), pick a subset such that $Q(x_{i1}))Q(x_{i2})...Q(x_{ir})$ is a square, y^2. Finally, if the following condition holds, we have the factors.

$Q(x_{i1}))Q(x_{i2})...Q(x_{ir}) \equiv (x_{i1}x_{i2}...x_{ir})^2 (\text{mod } N)$

3.2 Example

Explaining mathematical concepts through examples is easy to understand. In this section I will try to simplify for someone to understand the algorithm by giving an example.

Let's say we want to factor a number $N = 87463$ [2]. The ceiling of the square root of N is 296. Let me divide the task according to the two phases of QS

[2]number is taken from
http://www.math.colostate.edu/ hulpke/lectures/m400c/quadsievex.pdf

Algorithm 1 Quadratic Sieve Algorithm

1: **Input**: number N to be factored
2: **Output**: two factors of N
3: Choose a bound B
4: $\Pi(B)$ = the number of primes less than or equal to B. Call those prime numbers as $p_1 \dots p_k$
5: Find $\Pi(B) + 1$ numbers a_i so that ${a_i}^2$ mod $N = b_i$ is B-smooth, meaning that $b_i = {p_1}^{q_{i1}} \times {p_2}^{q_{i2}} \dots \times {p_k}^{q_{ik}}$
6: Represent b_i as matrix: $b_i = (q'_{i1}, q'_{i2}, \dots q'_{ik})$ where $q' = q$ mod 2
7: Use linear algebra to find a subset of those vectors b_i which add to the zero vector
8: Multiply to all corresponding a_i as well as all corresponding b_i together to produce the form $a^2 \equiv b^2$ mod N where $a = \prod a_i$ and $b^2 = \prod b_i$.
9: **if** $\gcd(a + b, N)$ is non-trivial **then**
10: **return** $\gcd(a + b, N)$ as one factor of N
11: **return** $N/\gcd(a + b, N)$ as the other factor of N
12: **else**
13: Try different set of b_i or different a_i
14: **end if**

Figure 3.1: [1] (taken from [[?]

- Data Collection
  We find the factor base through considering the values $\left(\frac{N}{p}\right)$ where $p = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37)$

  Definition: **Quadratic Residue** an integer q is called a quadratic residue modulo n if it is congruent to a perfect square modulo n; i.e., if there exists an integer x such that:
  $x^2 \equiv q(\text{mod} n)$
  Otherwise, q is called a quadratic nonresidue modulo n.

  Definition: **The Legendre symbol**: Let a be an integer and $p > 2$ a prime. Then the Legendre symbol $\left(\frac{a}{p}\right)$ is defined as

$$\left(\frac{a}{p}\right) = \begin{cases} 0, & \text{if } p|a \\ 1, & \text{if } a \text{ is quadratic residue mod p} \\ -1, & \text{if } a \text{ is non-residue mod p} \end{cases}$$

After computation, the factor base will be 2,3,13,17,19,29.

Then Construct the sieve by usinng Equation 3.1 for $0 \leq x < 100$.

$V = [Q(0)Q(1)Q(3)Q(4)...Q(99)]$

Then the next step is to solve $x^2 \equiv n$ mod p. For all the primes in the factor bases, we will have the following results
P = { 2 ..... 3 .... 13 .... 17 .... 19 .... 29 }
x = { 1..... 1,2 ...... 5,8 ..... .7,10...... 5,14......12,17}

We now start sieving, using a sieving interval of length $2 * 30$ around $\lceil \sqrt{N} \rceil = 295$.

- Data Processing
  For the values of x for which x^2âĹŠN splits completely,the exponent vector modulo 2 is:

x	-1	2	3	13	17	19	29
265	1	1	1	0	1	0	0
278	1	0	1	1	0	0	1
269	0	0	0	0	1	0	0
299	0	1	1	0	1	1	0
307	0	1	0	1	0	0	1
316	0	0	0	0	1	0	0

Then Solve the matrix in a transposed way : $Av = 0$ and not $vA = 0$: modulo 2. One solution is v = (1,1,1,0,1,0). These rows are selected, because if we add them they give us an even number, which is a square modulo.

We thus take the 1st, 2nd, 3rd and the 4th x-value and get

$x = 265 * 278 * 296 * 307 = 6694540240 \equiv 34757(modN)$

$$
\begin{pmatrix}
1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0
\end{pmatrix} \cdot \underline{\mathbf{v}} = \underline{\mathbf{0}}
$$

$y = \sqrt{(265^2 - N) * (278^2 - N) * (296^2 - N) * (307^2 - N)}$

$= 2 * 3^4 * 12^2 * 17 * 29 = 13497354 \equiv 28052(\text{mod} N)$

finally, by using gcd we get : $gcd\{x - y, N\} = 149 and gcd\{x + y, N\} = 587$

Chapter 4

Hadoop and MapReduce

4.1 Motivation

In the current world where digital equipments are massively used, the amount of data is growing at a fast pace. This tide of data is coming from different parties ranging from big business companies to a single person. Even, if we only consider the amount of videos uploaded to YouTube, it is easy to understand how the digital world is exploding. It is estimated that about 15 petabytes of data is generated every day [14]. According to IDC, the size of the "digital universe" was around 281 exabytes in 2007, and was forecasted to reach 1,800 exabytes by 2011, which is almost a 6 fold growth within a period of 4 years [15]. An Exabyte is 10^{18} bytes, or equivalently one billion gigabytes.

This crazy growth-rate makes data management cumbersome. We are capturing almost every data we want, but the challenge is where to store it and how to extract useful information in real-time. The main problem with storage is the rate at which disk drives are accessed. During the last two decades, disk capacity growth has outperformed to the access speeds. As Tom White explains, during the 1900s, 1.34 GB of disk capacity with 4.4 MB/s transfer speed was typical and it would only need five minutes to read all the data in the disk [16](pp. 3)[1]. And at this time, a terabyte hard disk with a transfer speed around 100MB/s has become a commodity. While drive capacity increased tremendously by about 1000 times, access-speed could not catch up. Because of this, it requires more than two and a

[1]The specifications are for Seagate ST-41600n hard disks

half hours to read all data resided in the terabyte disk.

One way to solve this problem is using the distributed system, which is to read from multiple disks in parallel. Considering the current rate we have, employing 100 drives would enable us to read the terabyte disk in less than 2 minutes. However, there are a number of issues, such as fault-tolerance and distribution that needs to be addressed with this kind of set-up. Here comes Hadoop into picture, and is explained in the next section.

4.2 What is Hadoop

Hadoop [6] is an open-source programming framework for large-scale data processing across clusters of computers using a simple programming model. Being developed mainly by Yahoo, it is now an Apache project. It is mostly inspired by Google's MapReduce and GFS (Google File System). Currently, Hadoop is immensely used by big companies like Yahoo, Facebook, Microsoft, and Amazon.

One important characteristic of Hadoop is the automatic handling of data replication and node failure. Hadoop tries to partition data and computation across the nodes, and executes application computations in parallel close to their data. The feature to make data processing locally is known as data locality, and is the reason for Hadoops' outstanding performance with very big data, by preserving network bandwidth.

As it is concisely pointed in [18] (Section 1.2), Hadoop has the following key distinctions. It is

- **Accesible**: it runs either on large clusters of machines or cloud computing services such as in Amazon
- **Robust**: it graciously handles node failures due to network or other problems
- **Scalable**: it only requires adding more nodes (usually cheap commodity machines) to handle larger data
- **Simple**: it doesnâĂŹt complicate writing programs, rather it allows to write simple and efficient code

Handling the complex nature of distribution is provided by Hadoop; therefore the programmer does not have to worry about making his program

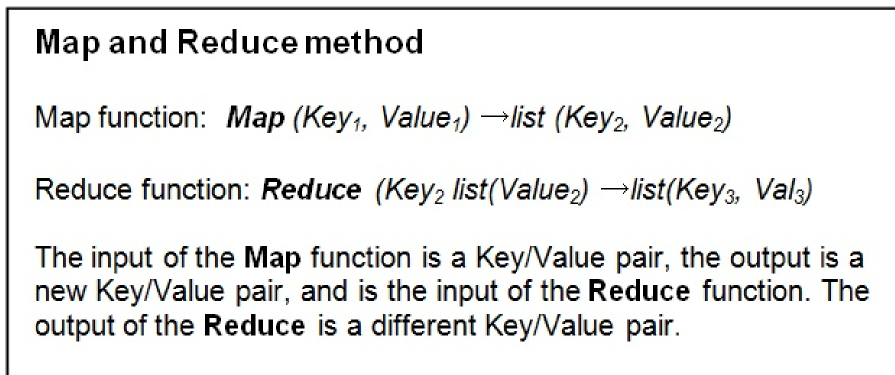


Figure 4.1: Map and Reduce Method in Nutshell.

distributed and can only focus on the data processing. Hadoop relies on the Hadoop file system, called HDFS, and the MapReduce programming model to abstract the problem from disk reads and writes, transforming it into a computation over sets of keys and values.

To summarize, Hadoop provides a reliable storage and analysis system through HDFS and MapReduce, respectively. I will talk on the two features in the next sub-sections.

4.2.1 MapReduce Programming Model

MapReduce is a programming model introduced by Google in 2004 to process and to generate large data sets [19].It abstracts the large-scale distributed data processing to one platform and two user-defined functions, **Map** and **Reduce**. Users implement the interfaces of the two functions. The Map function is responsible for processing sub-data sets and produce intermediate results, and the Reduce function is responsible for reduction of the intermediate results and generates the final results of the processing.

Applications written in this functional style are automatically parallelized and executed on the Hadoop cluster. The details of partitioning the input data, scheduling the execution across a set of machines, handling failures, and managing communication between nodes is taken care by the system. This encourages programmers without any prior experience with parallel and distributes systems to easily interact with the large distributed system.

A MapReduce program usually takes a set of `key/value` pairs as input.

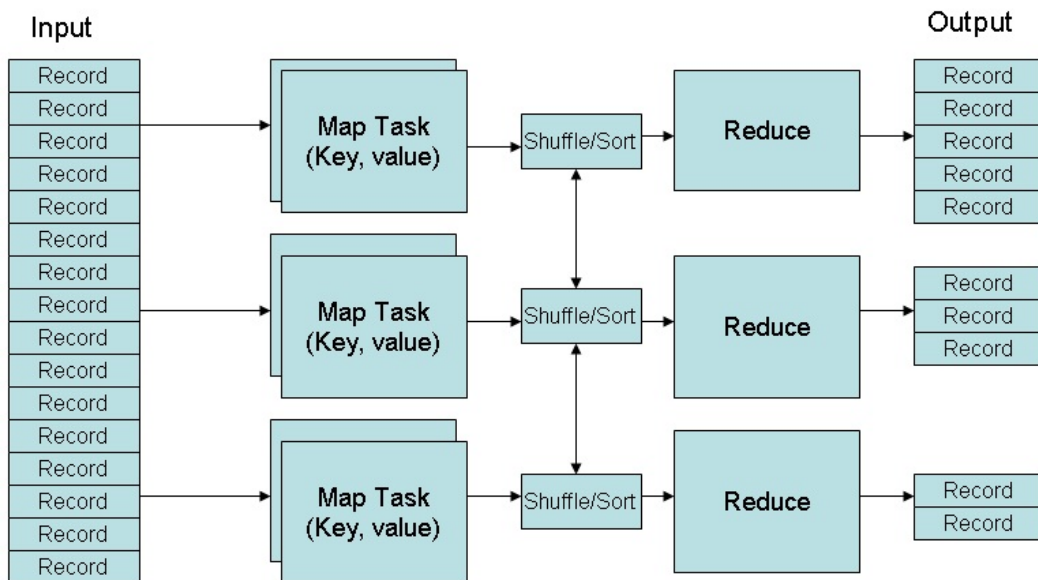


Figure 4.2: Quadratic Sieve Algorithm
[3] (source: http://sundar5.wordpress.com/2010/03/19/hadoop-basic/)

It then splits the input data-set into independent fixed-sized chunks called *input splits*. These pieces are processed by the map tasks in a completely parallel manner. After completing its task, the map function outputs another intermediate `key/value` pairs, which is going to be an output to the reduce function. It is good to note that this result is written to a local file-not the HDFS file system, because it will not be necessary to hold it after it is processed by the output function. This intermediate values associated with the same intermediate key are grouped together by the system and are fed into the user-defined Reduce function. The reduce function then merges the values and finally outputs possibly a smaller set of `key/value` pairs. The map and reduce functions are concisely given in Fig. 4.1.

The execution flow of the MapReduce model is given in Fig. 4.2. The `"Record"` boxes indicate to the data split which are inputs to the Map function. The output of the Map function is an input to the Reducers. However, it is important to note that the additional shuffle/sort step is required to correctly associate the key/value pairs with the responsible nodes running. For every call to the Reduce function, zero or more output value can be produced.

Let me try to elaborate the operation of the MapReduce using a word

```
map(filename, file-contents)
  for each word in file-contents:
      emit(word, 1)

reduce(word, values):
   sum = 0
  for each value in values:
      sum = sum + value
emit(word, sum)
```

Figure 4.3: MapReduce Word Count

count example. Let's assume that we have two files, each containing the texts `"computer science students"` and `"many students study computer"`. After counting the words in each file, we expect the output to look like: `(computer 2, science 1, students 2, many 1, study 1)`.

Based on the Fig. 4.3, I will try to show how the Map and Reduce steps are executed.

The map function instances are created on the different nodes in the cluster. Each instance receives a different input file. The map function outputs (word, 1) paris, in which word is the key and 1 is the value. For our example, the output of the map function may look like as shown below:

```
(computer 1)
(science 1)
(students 1)
(many 1)
(students 1)
(study 1)
(computer 1)
```

All this word/ones pairs are then an input to the reducer function. Each reducer is responsible for processing the list of values associated with each word. The function sums up all ones with the same word into a final count. The final out put looks like

```
(computer 2)
(science 1)
```

```
(students 2)
(many 1)
(study 1)
```

4.2.2 Hadoop Distributed File System (HDFS)

Hadoop Distributed File system (HDFS) [20] is a filesystem used in Hadoop. As it is well defined in [16], “HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware”. Its design enables it to store big data of thousands of gigabytes and stream them with in feasible time. The other advantage of HDFS is that it is economical for it can simply be implemented in commodity computers.

It looks like the UNIX file system and is inspired by the Google file system (GFS). It stores file system metadata and application data separately. It has two types of node operating in a master-worker pattern: the namenode (the master) and a number of datanodes (workers). Metadata is stored in the namenode and the application data is stored in workers called datanodes.

Figure 4.4 shows the basic architecture of the HDFS. The main constituents are shortly explained below.

i. **NameNode**

   The NameNode is responsible for managing the filesystem namespace tree and maps the location of all blocks to the DataNodes. However, this block location is not stored persistently, since it is constructed from DataNodes when the system starts.

   When an HDFS client wants to read a file located in the system, it first contacts the NameNode for the locations of data blocks comprising the file and then reads block contents from the DataNode closes to the client. Similarly, when the client wants to write data, it first queries the NameNode to nominate a suite of three DataNodes to host block replicas. The client then writes data to the DataNodes in a pipeline fashion. The replication number three is the default; it can be changed to any numbers.

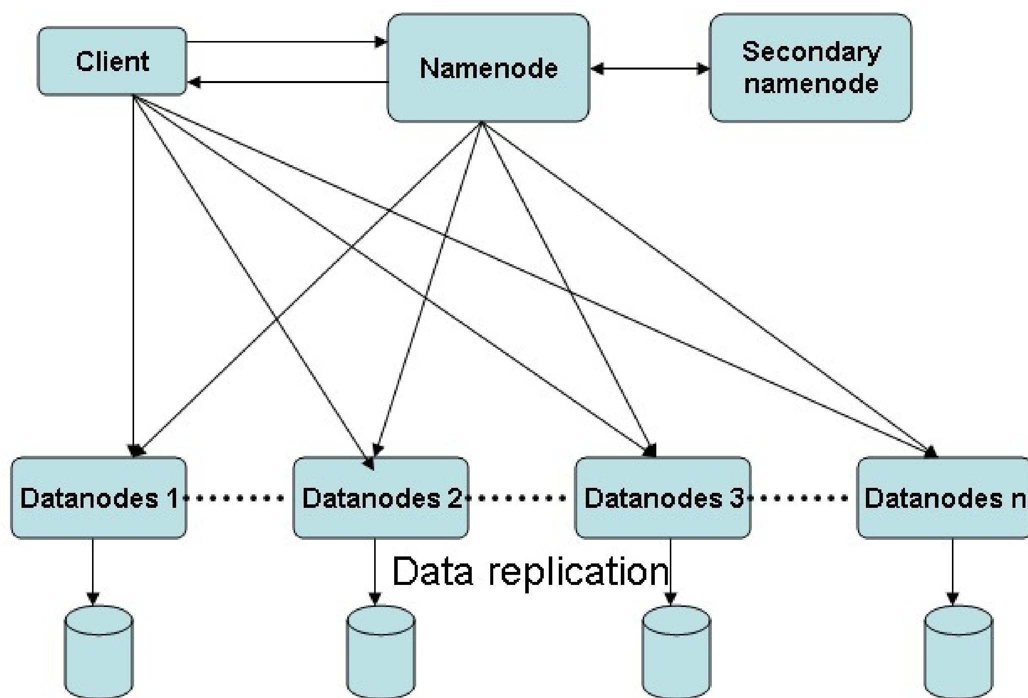


Figure 4.4: Overview of the HDFS Architecture [5](source: http://sundar5.wordpress.com/2010/03/19/hadoop-basic/)

The activities done by Namenode are memeory and I/O intensive. Because of this, it is not good practice to store any user data and perform computations in the NameNode in order to lower the workload on the server.

Without the NameNode, the filesystem is of no use. If the it crashes, there is no way to reconstruct the block information from DataNodes.

ii. **DataNode**

DataNodes are the real workers; they store data and, when told to do, retrieve blocks. Additionally, they have to periodically send report containing information on which blocks they are storing back to the NameNode.

DataNodes send heartbeats to the NameNode to confirm that they are still functioning and block replicas they host are available. Heartbeat interval is 3 seconds by default and if the NameNode doesn't receive a heartbeat from a DataNode within 19 minutes, then it is considered dead [21]. The NameNode also considers the replicas hosted by that specific DataNode to be available and consequently schedules creation of new replicas of those blocks on other DataNodes.

In HDFS datanodes, data is replicated on multiple datanodes for reliability, instead of using protection mechanism such as RAID.

iii. **Secondary NameNode (SNN)**

The Secondary NameNode is an assistant to NameNode for monitoring the state of the cluster's file system. Its main task is to take snapshots of the HDFS metadata from the NameNode memeory structures. By doing this it helps in preventing file system corruption and reducing loss of data. Unlike NameNode, the SNN dosn't record real-time information about any changes to the HDFS. It only takes snapshots from the NameNode according to some specified time interfal in the configuration.

The SNN doesn't overtake the management of the system, if the NameNode fails. The NameNode is a single point of failure in the Haduoop cluster. However, the images saved in SNN can help to a failed NameN-

ode when it is restarted. The NameNode can refer to SNN to create an up-to-date structure.

iv. **HDFS client**

HDFS client is a code library that provides an interface to user applications to access the HDFS filesystem. Like other conventional filesystems, it supports read, write, and delete operations. Always when reading and writing, the HDFS client first contacts the Namenode. An HDFS client creates a new file by giving its path to the Namenode. For each block of the file, the Namenode returns a list of Datanodes to host its replicas. The client then pipelines data to the chosen Datanodes, which eventually confirm the creation of the block replicas to the Namenode

Apart from the above storage daemons we discussed, there are two computing daemons worth discussing: *JobTracker* and *TaskTracker*. Like the storage daemons, they follow a master/slave architecture. The JobTracker daemon acts as a master node that provides a link between users application and Hadoop. It prepares the execution plan, assigns tasks to nodes, and then monitors the tasks. If a task fails, the JobTracker automatically restart the task possibly on a different node. TaskTrackers accepts execution plan for tasks from JobTracker and manage those tasks on each slave node. The TaskTracker continuously communiates with JobTracker to declare that it is alive. There can only be one JobTracker for every Hadoop Cluster and one TaskTracker per slave node. However, each TaskTracker can start multiple JVMs to handle a number of map or reduce tasks in parallel.

HDFS does not work so well with

- *Low-latency data access*: Remember, HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency.
- *Lots of small files*: Since the name node holds filesystem metadata in memory, the limit to the number of files in a filesystem is governed by the amount of memory on the namenode
- *Multiple writers, arbitrary file modifications*: Files in HDFS may be written to a single writer. Writes are always made at the end of file.

There is no support for multiple writers, or for modifications at arbitrary offsets in the file

4.3 Data Compression

Basically, Compression means encoding data/information using fewer bits than the original representation, thereby reducing disk space. Every compressed data must be decompressed during reading; therefore we might have some performance penalty. We have always to trade-off between performance and space. Compression usually enables the computation load to shift between the processor and IO. A thorough study is made in [30] on how and when to use compression.

Compressing a data can give us various benefits. A list of three advantages of compression is given in [29], and are listed below

- It significantly reduces the number of bytes written to/read from HDFS
- It improves efficiency of disk space and network bandwidth
- It reduces the size of data needed to be read when issuing read

Hadoop supports different compression algorithms, such as Gzip, Lzo and others. Some of them are optimized in terms of space while others are interms of time. Table 4.1 summarizes the different compression formats supported by Hadoop.[6]

Compression Format	**Tool**	**Algorithm**	**file name extension**	**Multiple files**	**Splittable**
DEFLATE	N/A	DEFLATE	.deflate	NO	NO
gzip	gzip	DEFLATE	.gz	NO	NO
bzip2	bzip2	bzip2	.bz2	NO	YES
LZO	lzop	LZO	.lzo	NO	NO

Table 4.1: Compression formats supported by Hadoop

The gzip compression format is the same as DEFLATE, except it adds extra headers and a footer. Out of the 4 formats, gzip tries to equally balance the trade off between space and time. In the case of bzip2, space is given more focus, as a result it is slower. However, its decompression speed

[6]table taken from [16] page 78

is faster than its compression, but not yet as good as the other formats. On the other hand, LZO's focus is more on speed rather than space. It compresses less effectively, but works faster. Additionally, the LZO libraries are GPL-licensed and you may need to get them from external source and integrate with your system.

Hadoop provides compression option for outputs of both map and reduce. Compression can be done via a configuration file or programmatically. Additionally, there are two compression types: BLOCK and RECORD. Group of keys and values are compressed together in BLOCK compression, while each value is compressed individually in Record comrpession.

Chapter 5

Implementation

5.1 Program implementation

All the experiments in these thesis are done by using a Quadratic Sieve MapReduce implementation written in Java. The program was originally written by Javier Tordable on the work in [8] and I used it under the permission of the Apache license. I made further modifications to the program to get better results. Before explaining some of the improvements, let me describe the basic structure of the program.

The program have to first collect data that may lead to factorization of a number and then perform the MapReduce task to finally get the factors. I divided the two parts into separate programs and worked on each of them. For the first part, I modified the program to be able to generate data in parallel across all the nodes. All the data is stored in separate files in HDFS [20]. The number of files can be specified as a parameter when running the program to collect data. During data collection, the factor base is generated first, then is serialized and passed to the nodes as counter. After that, the program generates the full sieve interval and makes it ready for MapReduce Processing.

The second part of the program performs the map and reduce tasks and finally tries to yield the two factors. The master node first splits the input data into basic blocks and distributes them to the slave nodes. All this task is handled by Hadoop's MapReduce framework. This part has two main functions: the mapping and reducing. During mapping, the slave nodes do the sieve. "Each one of them receives an interval to sieve, and they return

a subset of the elements in that input which are smooth over the factor base. All output elements of all mappers share the same key." These smooth numbers then passed to the reducers and the reducers attempt to find a subset of them whose produce is a square by solving the system module 2 by using direct bit manipulation. If suitable subset is found, then the reducer tries to factor the original number. If that doesn't factor, another subset is tried, as long as we have many subsets to chose from.

5.1.1 Choice of Factor Base Size

The factor base size is one of the critical factors in the speed of the factorization process. Choosing huge set of factor bases and choosing less have significant effect in the performance of the process. Small factor base size implies that only few numbers of Quadratic relations, say f(x), must be found. However, these relations happens rare and consequently very time consuming to find. On the other hand, Larger factor base size means, easy way of finding f(x). However, more number of these relations are required.

We find in [27], an illustration of execution times of 40 digit number under different factor base sizes. The result is shown in figure 1. It can easily be noticed that choosing too small size or too large yields to slow execution time.

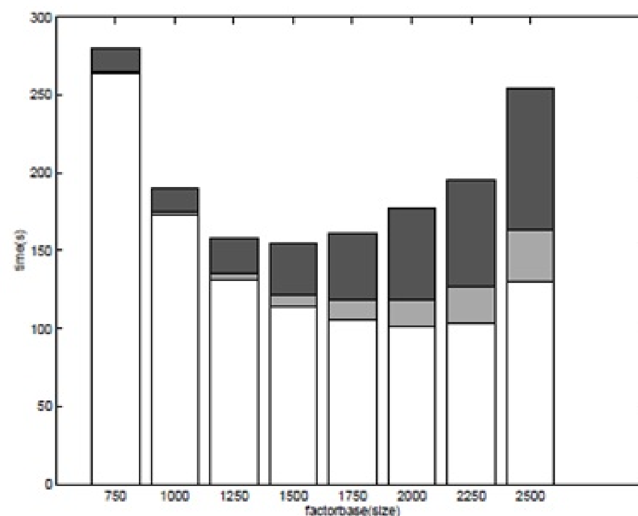


Figure 5.1: The effect of factor base size choice

As it can be easly seen from the figure, choosing a factor base more or less than 1500 would not make the factoring speed better. The choice of optimal factor base size becomes more critical in the approach used in the thesis than any other methods. The reason is that in our approach, data must be prepared first and then processed, unlike other systems where both tasks go in parallel. Therefore, if the factor base is poorly choosen, we might prepare redundant data or unsufficient data.

Choosing the optimal factor base size is not an easy task, because choosing fixed size would not suit to all numbers. After doing some research, I found some recommendation in one project menioned in [28], for numbers bigger than 25 digits. The formula they gave is

$$factorbasesize = 2.93x^2 - 164.4x + 2455 \tag{5.1}$$

For numbers less than 22-digit, I used the old factor-bases sizes, as the new one is greater than them. for numbers between 22 and 26 digits, I fixed the size to the minimum 140, after some experimentation. And last, for numbers greater than 26, I employed the formula recommended above.Table 5.1 and Figure 5.2 show comparison of factor base sizes between the new size used during thesis and from previous results we had.

Digit size	New	Previously used
20	108	108
22	140	140
24	140	196
26	140	108
28	158	316
30	170	429
32	204	516
34	262	665
36	343	880
38	448	1081
40	577	1323

Table 5.1: Comparison of factor base sizes

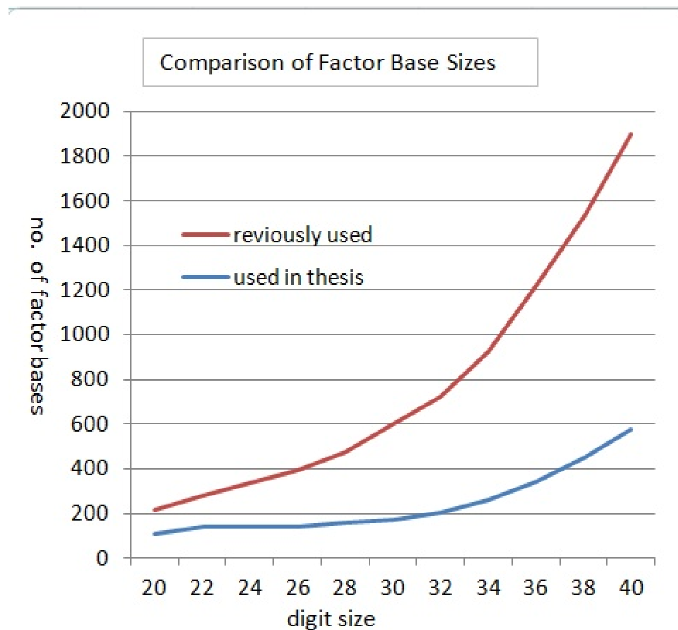


Figure 5.2: comparison of factor bases

The effect of the choice will be displayed, in the Results and Analysis chapter.

5.2 Cluster Setup

All implementations and experiments were done on Hadoop cluster containing 11 nodes. Each node has a specification of AMD Phenom II six-core 3.2GHz processor, 16 GB of ECC DDR-2 RAM, 1 TB secondary hard-disk. Each machines have network card with specification of HP ProCurve 2650 at the network bandwidth of 100 BaseTx-FD. In addition to that the cluster contains two racks with three datanodes in each rack. The racks are connected through a 1Gbps Realtek Semiconductor Co., Ltd. RTL8111/8168B PCI Express Gigabit Ethernet link and a Gigabit Ethernet switch is used to interconnect all the nodes. All nodes are loaded with Linux CenOs (Linux version 2.6.18-274.12.1.el5.centos.plus (mockbuild@builder10.centos.org) (gcc version 4.1.2 20080704 (Red Hat 4.1.2-51))) Operating system. All the experiments were conducted by using hadoop-0.20.203.0 release. Compactly, the cluster is formed with:

- 11 nodes

- 11 nodes each with 6 cores, totaled to 66 cores
- 11 nodes each with 16 GB memory, totaled to 176 GB.

Out of these 11-nodes, one was configured to serve as NameNode, and the rest were used as DataNodes. The Namenode is responsible for coordinating the tasks and the datanodes are the ones that actually perform the work.

5.2.1 Hadoop Configuration

All the above mentioned nodes have to be properly configured with Hadoop for the tests to be conducted. As it will be explained under the Results and Analysis section, Hadoop's configuration values have an important role to play. In the work done in [3], the default values of Hadoop configuration were mostly used. However, during *Computer Science Project* test, the parameters were more optimized and better result were registered. Further optimizations have been done during the thesis work.

In Hadoop, in order to override parameters with new values, you must make changes in one of three files under the configuration folder. These files are core-site.xml, mapred-site.xml and hdfs-site.xml. Some selected parameters which I think are worth mentioning are summarized in Table 5.2.

Table 5.2: Selected Parameters used in the configuration

File	Parameters	Value	Description
Core-site.xml	io.file.buffer.size	131072	(= 128M) Size of read/write buffer used in Sequence Files in Bytes Default: 4096 bytes (4K)
	fs.inmemory.size.mb	200	Larger amount of memory allocated for the in-memory file-system used to merge map-outputs at the reduces
	io.sort.factor	100	More streams merged at once while sorting files
	io.sort.mb	200	Higher memory-limit while sorting data
	io.compression.codecs	list of codecs	Different compression codecs were specified (explained below)
Mapred-site.xml	mapred.map.tasks	100	The default number of map tasks per job. Typically set to a prime several times greater than number of available hosts
	mapred.tasktracker.map.tasks.maximum	5	maximum number of map spawned on a task tracker. one node has 6 cores- so I used, one less
	mapred.reduce.tasks	10	This is number of reduce tasks per job. I chose 10, so that all 10 slave nodes do the reduce task
	mapred.tasktracker.reduce.tasks.maximum	5	maximum number of reduce spawned on a task tracker. Each node has 6 cores- so I used, one less
	mapred.child.java.opts	Xmx1024m	Java opts for the task tracker child processes. default is 200, so I gave larger heap-size for child jvms of maps/reduces
	mapred.child.ulimit	4194304	it is in bytes. should be 2âĂŞ3x higher than the heap size specified in mapred.child.java.opts
hdfs-site.xml	dfs.block.size	134217728	HDFS block size of 128MB for large file-systems. Default is 64M. "The larger this value, the fewer individual blocks will be stored on the DataNodes, and the larger the input splits will be".
	dfs.replicaton	3	I left this to default value, because there was no effect when changed.

5.2.2 Data Compression configuration

During data collection phase of the implementation, a large amount of data is generated. Hence, I configured hadoop to get the benefits of compression by reducing space requirements and speeding up data transfer across the network. While the gzip comes with the Hadoop Apache distributions, the LZO does not. The LZO libraries are GPL-licensed, as a result, I had to install and configure the libraries on top of Hadoop. To enable compression on the output of mapper's, the mapred-compress.map.output parameter should be

set to true. The configuration done in mapred-site.xml file is shown below:

```
<property>
<name>mapred.compress.map.output</name>
<value>true</value>
<description></description>
</property>
```

For the LZO compression to work, the LZO libraries must be properly installed. After installation, there are two properties that should be defined. First, all the available codecs must be listed in the core-site.xml file, as shown below:

```
<property>
   <name>io.compression.codecs</name>
   <value>org.apache.hadoop.io.compress.GzipCodec,
      org.apache.hadoop.io.compress.DefaultCodec,
      org.apache.hadoop.io.compress.BZip2Codec,
      com.hadoop.compression.lzo.LzoCodec,
      com.hadoop.compression.lzo.LzopCodec</value>
</property>
```

Then in the mapred-site.xml in addition to enabling the compression property, the compression codec which is going to use in compressing the output should be specified as shown below:

```
<property>
   <name>mapreduce.map.output.compress.codec</name>
   <value>com.hadoop.compression.lzo.LzoCodec</value>
</property>
```

The above configuration will tell Hadoop, to compress all outputs of map

by using the LzoCodec.

5.3 Numbers Used

All the numbers used during the test are product of two prime numbers with nearly-equal sizes interms of digit size. These kind of numbers are the hardest ones to factor. In addition to that, in order for me to be able to compare with our previous results, I used the same numbers ranging from 20 to 40 digits. But, all numbers above 40-digit are newly generated. These numbers are given in Table 5.3. When I make analysis, I will refer to these numbers by their digit-size. For example, if I say 20 digit number, then I am implying to the number 50031469295581191569 from Table 5.3.

In each of the tests, I was measuring the time needed to prepare the file and the data processing. Adding these two components would give the total time required to factor a number. The running time and the required disk space to store temorary files increase with respect to the size of the input numbers. The tests conducted and their analysis are given in the following section.

Table 5.3: Composite numbers and their corresponding factors

Digit size	Bit size	Composite number	Factor number 1	Factor number 2
20.	66	50031469295581191569	5911885873	8462861153
22.	72	3093234287444441437763	54761954833	56485096211
24.	80	862453752119754745160851	792406500433	1088398128547
26.	86	49011889928728242228979543	6422306627983	7631508859321
28.	92	3518600851308845646375884429	60323532586829	58328826254401
30.	98	668113306130414322957880778059	792306979677701	843250562303759
32.	104	39020183337692330125364561421937	5920597772026877	6590581701403781
34.	112	4127273800774582589008897597045849	64730046451170773	63761329197994613
36.	120	814953198814747109042639381041932671	974179817226728669	836553154154574859
38.	126	46290891461674506619188664102675408291	9031123274871755341	5125706963880619951
40.	132	3738498394415495504811095025706246229629	59070320211691524599	63288947495421756971
42.	140	885696632089484883110677058539286807328389	1128694497756857601379	7847089126860268335191
44.	145	39451743650742425837313602239140165404434849	7834911716796649127831	5035378199114221670279
46.	152	4475168651644614627902597500997543594279149181	62811622721428808569063	71247461182337300468987
48.	160	879220724720998982398721996043809035486062674831	988159202798848607621321	889756146813900259964311
50.	166	53468946676763197941455249471721044636943883361749	5974436590343814450125977	8949621586608250991047837

Chapter 6

Results and Analysis

6.1 Size of Generated Data

In order to factor a number, a data based on the factor base information have to be placed in HDFS. This data varries in size according to the number of digits forming the number to be factored. Table 6.1 shows the total size of the data for each numbers in giga byte and mega byte and Figure 6.1 shows the file trend as the number of digits increase. From these two figures, it can easily be noticed that the size grows-up exponentially. For example, if we carefully look into numbers above 30 digit, we see that for every increase of 2 digits, the size of the data doubles. The reason for this sharp rise is the factor base size we generate using equation (5.1). In the program, the full sieve interval size is determined using the factor base size to the power of three. Changing the power to less than three, was not good enough to factor for most of the numbers. At least, the factor base size is much more reduced than the previous, so taking to the power of three was fair. So, having such setup makes the the groth exponential.

This increase brings a serious concern to the disk capacity required to factor bigger digit. With this trend, for instance to factor 60 digit number, about 8 TB disk space will be required for the necessary sieve interval to be prepared.

Digit size	Bit size	File size (GB)	File size (MB)
20	66	0.04	37
22	72	0.08	86
24	80	0.09	92
26	86	0.09	97
28	92	0.14	147
30	98	0.19	193
32	104	0.34	351
34	112	0.76	781
36	120	1.8	1,855
38	126	4.2	4,295
40	132	9.5	9,628
42	138	20	20,480
44	144	39	39,936
46	150	74	75,780
48	156	134	137,232
50	164	229	234,496

Table 6.1: Size of File Generated for each number

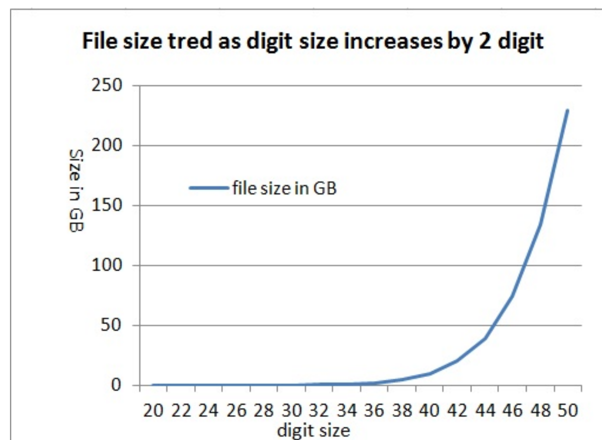


Figure 6.1: File Size trend in GB vs digit size

6.1.1 comparison between new file size and previous one

In this section, I present comparison between the file sizes, produced in the previous works and data used in the thesis. In both cases, it was possible to factor the respective numbers, this shows we were generating redundant data and as a result were doing extra computation.

From Table 6.2 and Figure 6.2, it can easily be noticed the difference, especially as the number of digits grow. For instance, were were generating a data which is 12 times greater that what we have now. Even at this time, for some numbers the data generated might be redundant. The problem with

this approach is that, data must be prepared first, then stored in HDFS, and finally start processing. However, in other methods such as using MPIs, the data processing is done in parallel with the data collection. In such cases, you don't have to worry about how much data you have to prepare, because the program stops when enough relations are collected.

As it will be discussed, in the next section, the size of the data will defninely impact the speed of the factorization.

Digit size	Bit size	New generated data size (GB)	Previously generated data size (GB)
20	66	0.04	0.04
24	80	0.09	0.25
28	92	0.14	1.17
30	98	0.19	2.25
32	104	0.34	5.21
34	112	0.76	12.84
36	120	1.8	31.3
40	132	9.5	115.8

Table 6.2: File Size Improvements from previous works

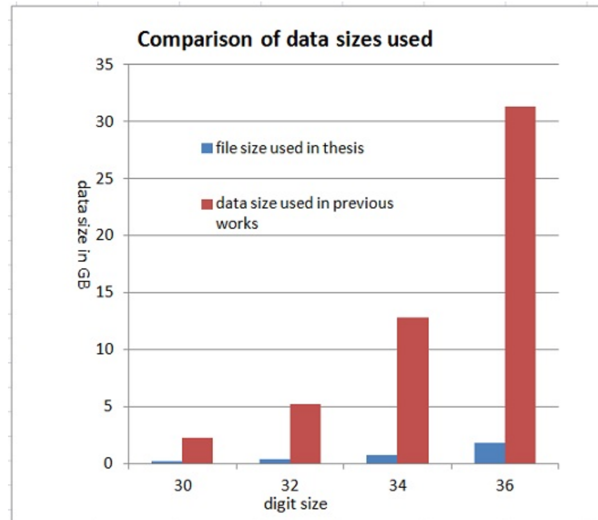


Figure 6.2: Imrpvement of file Size in the thesis

6.2 Performance of Data collection and processing time

In this section, I present the time taken to collect data and to process it. Table 6.3 gives the time taken to factorize different numbers under 11-node cluster. Out of these 11 machines, one is assigned the role of a NameNode in order to control all the process accross the rest 10 nodes. During the data-collection phase, each node out of the 10 slaves was assigned a part from the total sieve interval to prepare the necessary data. The whole sieve interval is equally divided into 10, and each node prepares the data and saves into the HDFS as a separate file. As a result, I had to see 10 different files in the HDFS with almost equal size. Data collection phase is computational intensive where the CPU is intensively used. In order not to overload one node, the program is slightly modified all other nodes to take their share in the computation. In this test, all 10 prepared data in parallel, so the time improved by about 10 times than when it is done in one machine.

Digit size	**Bit size**	**Data collection time (min)**	**MapReduce time (min)**	**Total time taken (min)**
20	66	0.05	0.65	0.70
22	72	0.05	0.66	0.71
24	80	0.05	0.66	0.71
26	86	0.05	0.66	0.71
28	92	0.05	0.81	0.86
30	98	0.05	0.85	0.90
32	104	0.09	0.89	0.98
34	112	0.19	1.09	1.28
36	120	0.42	1.25	1.67
38	126	1.05	1.82	2.87
40	132	2.25	2.60	4.85
42	138	4.42	4.91	9.33
44	144	8.5	9.41	17.91
46	150	17	19	36
48	156	31	41	72
50	164	50	90	140

Table 6.3: Performance measurement for 11-node cluster

In Table 6.3 the MapReduce column is time taken from running the second part of the program to do processing to getting the final result. Then the summation of the two parts is given under the last column. The time gap between data collection and data processing is ignored. This gap

can be the time spent to check the HDFS for the proper placement of files, and may be, removing unnecessary extra files in the HDFS. As it can also be noticed in Figure 6.3, the MapReduce time increases exponentially, just as the file size and data collection time do.

6.2.1 Performance improvement from the previous result

During the initial work, mostly the default hadoop configuration parameters were used and data was only generated in one machine, then on the computer science project, the program was slightly modified to distribute data collection and some of the used Hadoop configuration parameters were optimized. In this thesis, it is easily possible to distribute the data collection task as per the users wish and the hadoop configuration parameters were further optimized. Additionally, the data processing phase is optimized, for instance, the way the factor base size is determined. As can be seen in Table 6.4, the MapReduce time is improved enormously from the initial one.

In Table 6.4, the third and fouth columns compare the time taken to collect data when it is distributed across the 10 nodes or only performed by one machine. Additionally the file used during the previous works was the same for all numbers, but smaller during the thesis. The last three columns show time recorded during the initial work, computer science project course, and this thesis respectively.

During all these tests, the data replication option was left to the default value, which is 3, because changing it didnt bring any performance issue. The reason for this might be due to the fact that all nodes are connected to the same switch, and this reduces the probability of data loss because of network failure.

Digit size	**Bit size**	**Init. Data coll. time (min) in 10 nodes**	**Data coll. in 1 node (min)**	**Initial MR (mins)**	**MR-time in cs project (mins)**	**Latest MR-time (mins)**
20	66	0.05	0.10	2.12	0.70	0.65
24	72	0.05	0.57	2.62	0.71	0.66
28	80	0.05	2.62	3.18	0.86	0.81
32	86	0.09	10.58	9.55	0.98	0.89
36	92	0.42	64.15	55.03	1.67	1.25
40	98	2.25	236.30	234.88	4.85	2.60

Table 6.4: Performance improvement after optimizations

The improvement is tremendous as the digit size increases. If the numbers are very small, then the map reduce time is almost the same. With all the optimizations done, the time required to prepare sieve intervals and to process the matrix , for instance, for 40 digit number have improved by around 90 times.

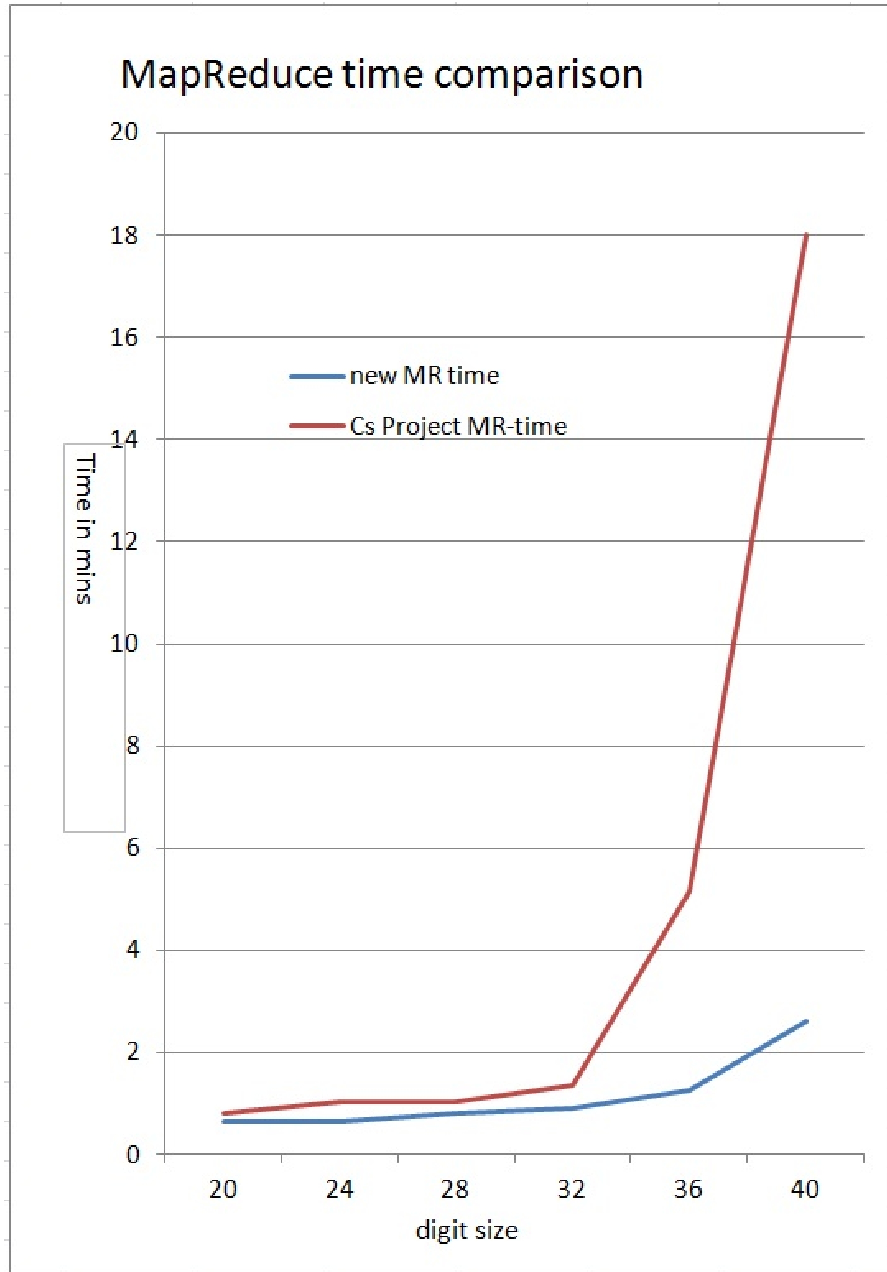


Figure 6.3: Imrpvement of file Size in the thesis

From Figure 6.3, still the MapReduce time increases exponentially in all cases, eventhough after all the optimization it goes at a steady pace. This indicates that the resource requirement, interms of disk space and processing times will be a big constraint to factor bigger numbers.

Another observation from Table 6.4 is that the ratio of the MapReduce time to file size dcreases sharply. This can show us that hadoop performs better as the size of the data to be processed is bigger.

6.3 Effect of Cluster Size on Performance

In this section, I try to show the effect of having different number of workers (slave nodes) on the performance of Hadoop-MapReduce. I left one separate namenode in each test cases, that means the cluster size is one plus number of nodes in Table 6.5 under the "No. of Slaves" column. In the table, the time required to do the data processing for 40 digit number is measured when the cluster has 1, 2, 4, 6, 8, and 10 slave nodes. From the result, it can be noticed that as the cluster size increase, the cluster gets more procesing power, hence the time is going down. This is due to the workload being shared by the additional nodes. Figure 6.4 shows how the trend is as the cluster size increase.

This can be considered as the biggest advantage of this approach from others interms of simplicity and scalability. To improve performance, you simply need to only add more nodes. You don't need to modify any part of the code. and It can also scale to any number of nodes you may have. All the distribution system headaches, such as coordination, is handled by Hadoop.

However, one thing I would like to mention is that all the tests are done on machines which are connected to one Gigabit switch. Threrefore, it will not be wise to ignore the network delay that might happen when the nodes are connected to many switches. The data will need to be distributed across a scattered network and the nodes will continuously need to send message that they are alive to the master.

No of Slaves	MapReduce Time (mins)
Factorization of 40-digit number	
1-node	16.76
2-node	9.11
4-node	4.95
6-node	3.58
8-node	3.05
10-node	2.72

Table 6.5: Performance comparison when factoring 40-digit number in different cluster sizes

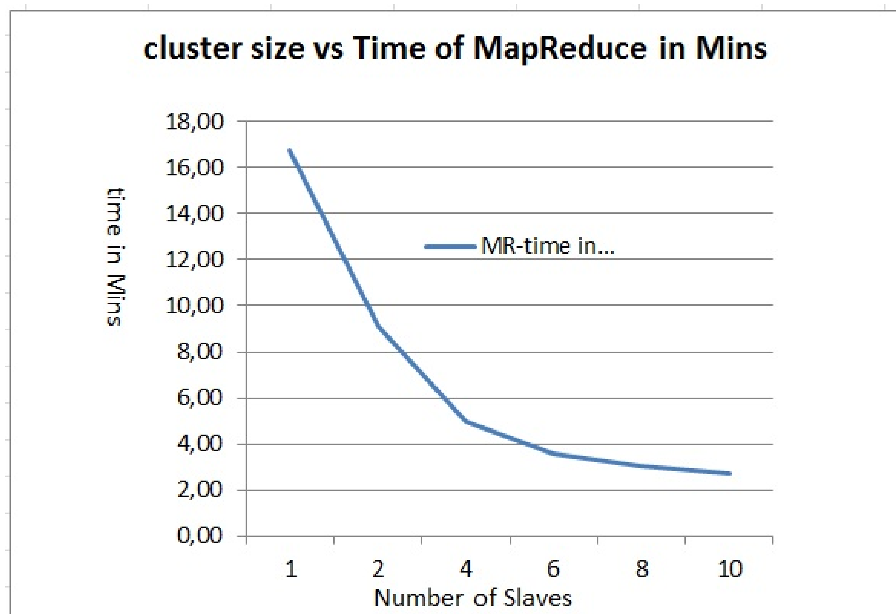


Figure 6.4: Factoring 36-digit in different cluster setting

6.4 Effect of Data Compression and Splitting Data into files

As the data size grows fast, the disk requirement is a big concern. The concern is both on the input data and output of map as, the Mapper function may produce very big temporary output. I made the test 40 digit number, by dividing the data into a number of files. In the approach, there should be around 9.5 GB data placed in HDFS in total. I put this data into 1, 5, 10, 20, 40, 80, and 100 files, and tested each cases. In Table 6.6, the columns

which say size of each file, refers to size of a single file, not to the total. For example, size of each uncompressed file when the data is divided amont 10 files is 0.93GB. This means, the total data size placed in HDFS is 10 times 0.93 GB, which is 9.3GB. Such multiplication should give us the same value, whether we slice the data into small parts.

From the table, if we compare the size of each uncompressed file against the compressed ones, then we see different sizes. Gzip algorithm performs better than LZO interms of disk space gain. While Gzip reduces the size by about 70%, LZO shrinks by 50%. In the next two subsections, I will try to briefly discuss the performance on compressing both the input and the map output.

No. of Files	**Size of each file (Uncomp)**	**MR-time (Mins)**	**size of each LZO file (GB)**	**MR-time LZO (mins)**	**Size of each Gzip file (GB)**	**MR-time Gzip file (GB)**
1	9.5	2.75	4.3	34.95	2.9	34.99
5	1.9	2.75	0.87	7.77	0.56	7.75
10	0.93	2.75	0.43	4.45	0.28	4.43
20	0.47	2.75	0.22	2.75	0.14	2.77
40	0.23	2.75	0.11	2.11	0.07	2.19
80	0.12	2.75	0.06	2.51	0.04	4.56
100	0.09	2.75	0.04	2.54	0.03	2.67

Table 6.6: Effect of Compression

6.4.1 Compressing the Input Data

The third column in Table 6.6 showing the MapReduce time for the uncompressed ones have the same value for all kinds of slices. For uncompressed data, Hadoop simply divides into small fixed block sizes and assigns them to the workers (mappers). Therefore, whether the input is one file or file, there wouldn't be huge difference. How ever, for compressed data, each compressed entity is assigned to one mapper, so it lacks the benefits of distribution. That is the reason, why as the data is divided in to smaller files, the performance improves simultaneously.

One thing I noticed during the tests is that, compressing either in Gzip or in LZO, the result of time needed to do processing is the same. But, the Gzip saves more disk space, there fore, Gzip seems the better option in our case. However, as it is also explained in the literature review in Chapter 4, LZO has a lot of benefits.

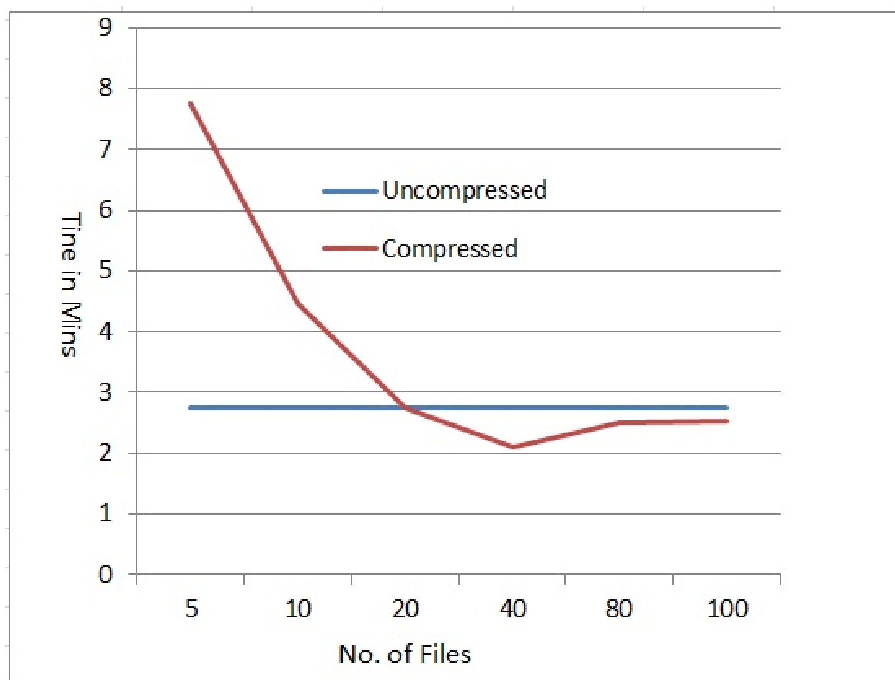


Figure 6.5: Improvement of Compression

Figure 6.5 compares the MapReduce time for compressed and uncompressed data in different files. The blue color shows the MapReduce time for uncompressed data. The fact that it is straight line indicates that the time is relatively stable whether it is sliced into few or many files. However, for compressed data, the time improves as the data is sliced into more files. it reaches its best time when it is sliced into 40 files. The main reason for this might be due to the buffer and block size I used in the configuration. I used 128MB for block size and buffer size, and this means each time, a 128 MB slice of data is processed at a time. As the slicing continues towards 40, the file size nears to the buffer size, as a result it becomes fast to decompress and process. But for the less number of files, it takes a lot of time to decompress the file before start processing.

If we keep on dividing the data into, say 100 files, the size of each file is less than the configured Hadoop's buffer size. This means, the decompression is of course much faster, but the decompression function is called many times. There fore, map-reduce time is not the best, eventhough it is better than the uncompressed one.

6.4.2 Compressing the Output of Map

I configured hadoop to compress the outptut of map to Gzip compression codec first, then to LZO compression coded. As LZO is GPL licensed I had to make additional steps in order to make LZO work. After enabling compression on the output of map, there was not any improvement gain registered. In all cases, the result was the same as the time taken when it is not compressed, which is 2.75 minutes on average. This can be due to the fact that the cluster's disk space is more than enough fore the data. I tested the compression with data as big as 230 GB for 50-digit and there was no any performance penalty or gain. Each node in the cluster, has 1 TB secondary disk, that means collectively it is 10 TB.

Chapter 7

Discussion

As mentioned in the previoius chapter, the performance of the system to factor a number was affected by different factors. The factor base size is very important factor, because the file size depends on it. The factor base schema used in this thesis, is not of course the optimal one. For some numbers, redundant data was being prepared, and while the same size is not enough to factor other numbers. Finding a right balance in the hadoop configuration parameters is also an important factor.

In this approach, the trend how the data size increases is a big concern, as it grows exponentially. Eventhough, with the option of data compression reduces the size significantly, it doesn't change the trend.

Compared to the initial result we had, the result have improved enormously. However, compared with other approaches such as MPI options, using Hadoop as a tool has serious limitations. Let me discuss the limitations of the approach in the next section.

7.1 Limitations of using Hadoop as a tool

There are fastest implementations of the quadratic sieve algorithm using different approach. For example, the parallel quadratic implementation from [28], factors a 40 digit number in less than 10 minutes using a computer with 2 GB RAM and 2.4 dual core processor. There are two main reasons that I would like to mention why the Hadoop approach is slower than that.

1. In the Hadoop approach, the data collection and data processing phases of the quadratic sieve are done in series. It is a must to first

gather required information before starting processing. On the other hand, in the other approaches, they do these two phases in parallel. These implementations, they gather relations that may lead to factorization of a number and at the same time process these relations. If a set of relations that give the factors are found, the process is terminated and the result is displayed. Figures 7.1 and 7.2 shows the difference of the two approaches.

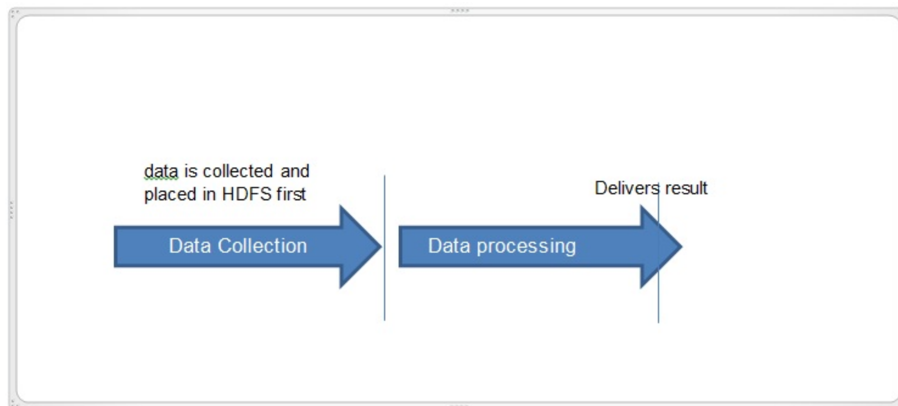


Figure 7.1: The two Phases in series (Hadoop's approach)

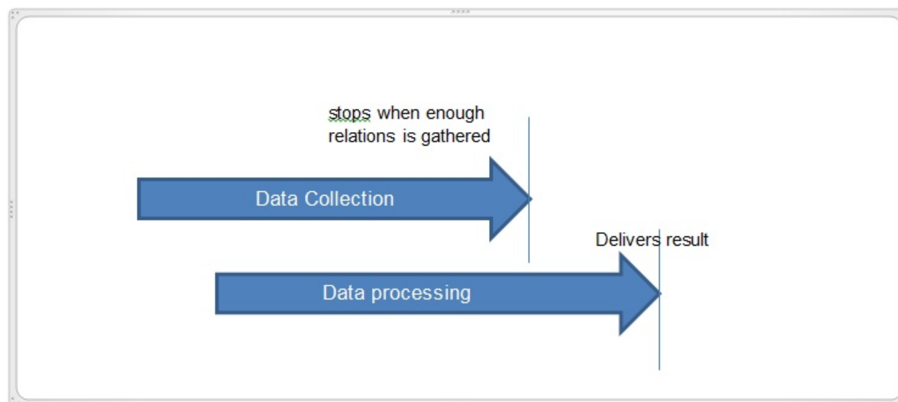


Figure 7.2: Doing the two phases of QS in parallel

2. In Hadoop, data is stored in HDFS, which is an abstraction of disk. However, in the case of MPI implementations, data is read from memory. Reading data from disk punishes the performance of the system a lot.

The approach, however, beats other approaches over its simplicity. In other conventional approaches, the task of managing the resources, such as memory or processor, falls on the hand of the programer. To the contrary, Hadoop handles the difficult tasks of distribution and fault-tolerance for the user.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this thesis, the possibility of factoring big integer number through exploiting the strenghts of hadoop was studied.

The approach is relatively simple to implement compared to other conventional implementations. One of the biggest features of Hadoop/MapReduce is its tolerance to fault. Therefore, it alleviates the pressure of managing and coordinating Hadoop's nodes. Secondly, the approach can easily be scalled to even thousands nodes, without worrying to make change in the design or code.

However, there are serious limitations that make it not the best option for the problem. Primarily hadoop is designed for data-intensive applications, but integer factorization is mainly computation-intensive though it has small data-intensive part. And the data gathered during the process grows exponentially as the digit size growth. This affects badly to the requirement of disk space and processing huge data would also take more time. The other limitation factor is data is written and read from disk, which is much slower than doing stuff in memory. The way data is also read and written are another problem. Data has first to be prepared, it is after that the processing starts.

Due to the above limitations, It seems changing to any variants of the quadratic algoritms would not bring any breakthrough. But, It may improve the current result to some degree.

8.2 Future Work

Working on a way to avoid the above limitations should be the primary goal in the future. At this time, the implementation of pure hadoop doesn't exactly fit to the problem. Message Passing Interface (MPI) implementations are widely accepted for high-performance computing purposes. If hadoop can adapt the benefits of MPI in the future, then a speed up in data intensive computing applications such as Integer factorization problem can be registered.

At this time, it is not clear what is the optimal value of the data to be prepared. For some of the numbers, it is optimal, for others it is not enough and for some numbers redundant data is computed and stored. Hence, looking into an option to define what is enough is interesting.

Algorithmic wise, implementing the other variants of Quadratic sieve algorithm, and may be General Number Field Sieve (GNFS) algorithms can also make a difference. Looking into that is, I think, wise move.

Bibliography

[1] R.L. Rivest. A. Shamir, and L. Adleman, “A method for obtaining digital signatures and Piublic-key Cryptosystems”, Commun. ACM, vol. 21, no. 2, pp. 120âĂŞ126, Feb. 1978. 1

[2] Junod, Pascal. "Cryptographic Secure Pseudo-Random Bits Generation: The Blum-Blum-Shub Generator", Mar 26, 2012. 1

[3] Son Nguyen, Semere Tsehaye Ghebregiorgish, Nour Alabasi, Chunming Rong. *Integer factorization using Hadoop*, submitted to the IEEE CloudCom 3rd International Conference, August 2011. 1, 11, 32

[4] K. Aoki, J. Franke, T. Kleinjung, A.K. Lenstra, D.A. Osvik, A kilobit special number field sieve factorization, Proceedings Asiacrypt 2007, Springer-Verlag, LNCS 4833 (2007) 1 -12 2

[5] Thorsten Kleinjung et. al. *Factorization of a 768-bit RSA Modulus*, http://eprint.iacr.org/2010/006.pdf. Retrieved Aug 15, 2011. 5

[6] RSA Numbers, http://en.wikipedia.org/wiki/RSA_numbers, formerly on http://www.rsa.com/rsalabs/node.asp?id=2093 1, 6

[7] Andrew S. Tanenbaum and Maarten van Steen. Distributed Systems: Principles and Paradigms. Prentice Hall, second edition, 2007. 3

[8] Javier Tordable.J. Tordable, “MapReduce for Integer Factorization,” arXiv:1001.0421, Jan. 2010., http://www.javiertordable.com/files/MapreduceForIntegerFactorization.pdf. Retrieved Apr 19, 2012. 3, 28

[9] Steven Arzt, “Distributed Number Factorization Using Hadoop”, `http://www.student.informatik.tu-darmstadt.de/s_arzt/Research/CloudFactorizatio` Apr 15, 2012 3

[10] H. Te Riele, W. Lioen, and D. Winter, âĂIJFactoring with the quadratic sieve on large vector computers,âĂİ Journal of Computational and Applied Mathematics, vol. 27, no. 1âĂŞ2, pp. 267âĂŞ278, 1989. 4

[11] J. A. Davis and D. B. Holdridge, âĂIJFactorization using the Quadratic Sieve algorithm,âĂİ pp. 103âĂŞ116, 1984. 4

[12] N.-F. Tzeng, âĂIJReliable butterfly distributed-memory multiprocessors,âĂİ Computers IEEE Transactions on, vol. 43, no. 9, pp. 1004âĂŞ1013, 1994. 4

[13] Cosnard and J. Philippe, âĂIJThe Quadratic Sieve Factoring Algorithm on Distributed Memory Multiprocessors,âĂİ Proceedings of the Fifth Distributed Memory Computing Conference 1990, pp. 254âĂŞ262, 1990. 4

[14] XZ Backup, LLC, Data and storage growth trends âĂŞ and how they affect online backup, (http://www.xzbackup.com/blog/company-news/data-and-storage-growth-trends-and-how-they-affect-online-backup/), Retrieved Sep 22, 2011. 17

[15] Grantz et al., âĂIJThe Diverse and Exploding Digital UniverseâĂİ, March 2008 (http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf), retrieved Sep 22,2011 17

[16] Tom White, âĂIJHadoop: The definitive GuideâĂİ, Second Edition June 2009, 17, 22, 26

[17] Hadoop Project. http://hadoop.apache.org/

[18] C. Lam, “Hadoop in Action", 1st ed. Manning Publications, 2010 18

[19] Jeffrey Dean, Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*, in Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI’04), Dec 2004. 19

[20] Hadoop Distributed Filesystem: http://hadoop.apache.org/hdfs/ 22, 28

[21] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, âĂIJThe Hadoop Distributed File SystemâĂİ,

http://storageconference.org/2010/Papers/MSST/Shvachko.pdf, retrieved Sep 24,211 24

[22] William Stallings, âĂIJCryptography and Netowrk Security Principles and PracticeâĂİ, 5th Edition 6

[23] Richard Crandall, Carl Pomerance. *Prime Numbers: A Computational Perspective*, 2nd Edition, Springer, 2005. 9

[24] John D. Dixon. *Asymptotically Fast Factorization of Integers*, in Mathematics of Computation, Vol.36, No.153, Jan 1981. 10

[25] Carl Ponence, Analysis and Comparison of Some Integer Factoring Algorithms, in Computational Methods in Number Theory, Part I, H.W. Lenstra, Jr. And R.Tiejdeman, eds., Math, Centre Tract 154, Amsterdam, 1982, pp 89-139. 12

[26] T. D. <tim@dierks.org>, âĂIJThe Transport Layer Security (TLS) Protocol Version 1.2.âĂİ [Online]. Available: http://tools.ietf.org/html/rfc5246. [Accessed: 07-Jan-2012]. 3

[27] Olof Osbrink and Joel Brynielsson, Factoring Large Integers Using Parallel Quadratic Sieve, ftp://ftp.nada.kth.se/Theory/Joel-Brynielsson/qs.pdf retrieved 29/03/2012 29

[28] Parallel Implementation of the Quadratic Sieve, http://www.bytopia.dk/qs/, retrieved on 25 Mar, 2012 30, 48

[29] Hadoop Wiki, http://wiki.apache.org/hadoop/UsingLzoCompression, retrieved on Apr 23, 2012 26

[30] Y. Chen, A. Ganapathi, and R. H. Katz, “To compress or not to compress - compute vs. IO tradeoffs for mapreduce energy efficiency,” in Proceedings of the first ACM SIGCOMM workshop on Green networking, New York, NY, USA, 2010, pp. 23 - 28. 26