PARALLEL BOYER-MOORE STRING MATCHING ALGORITHM USING HADOOP

A thesis submitted in partial fulfillment of the requirements
For the degree of Master of Science in Computer Science

By

Cesar Zalzalah

December 2015
The thesis of Cesar Zalzalah is approved:

______________________________  __________
Dr. Gloria Melara  Date

______________________________  __________
Dr. Taehyung Wang  Date

______________________________  __________
Dr. Richard Covington, Chair  Date

California State University, Northridge
Dedication

I dedicate my thesis work to the love of my life my wife Irina Zalzalah, for being with me during the bad time and the good ones too!
Acknowledgement

I would like to express my sincere appreciation to my advisor and committee chair Prof. Richard Covington, for his wise advice and support, without which this thesis would not have been possible. His broad knowledge and expertise helped me immensely in understanding, structuring, and developing this thesis.

I would also like to thank my committee members Prof. Gloria Melara and Prof. Taehyung Wang for accepting to be on my committee and for reviewing and certifying the thesis. Prof. Taehyung Wang introduced me to Big Data, Hadoop, and Spark and for this I am very grateful. Also, I would like to thank Prof. Robert McIlhenny for introducing me to the area of parallel computing and allowing me to use the subject of string matching as a research project for his class COMP 620. Special thanks to Professor John Noga for reinforcing my algorithmic skills that helped me immensely during my research.

I would like to thank Google, The Apache Software Foundation specifically and the Open Source communities as a whole for sharing ideas, source code, and information as well as for advancing innovation all for the benefit of the public good.

Last but not least I’d like to thank my family for their love and full support. Without you, all this thesis and my graduate degree would not have been possible.
Table of Contents

Signature Page  ii
Dedication  iii
Acknowledgement  iv
List of Tables  vii
List of Figures  viii
Abstract  ix

1. Introduction  1
2. Related Work  3
3. Background  5
   3.1 Boyer–Moore Algorithm  5
      3.1.1 Time Complexity of the BM Algorithm  9
      3.1.2 Boyer-Moore-Horspool Algorithm  9
   3.2 Big Data  10
   3.3 Cloud Computing  12
   3.4 Amazon Web Services (AWS)  13
   3.5 Hadoop  15
      3.5.1 Hadoop Distributed File System (HDFS) Architecture  16
      3.5.2 MapReduce 2.0 Framework  18
   3.6 Apache Spark  20
4. Parallel Architecture  24
### Table of Contents

4.1 Text Partitioning and Data Distribution ........................................... 25
4.2 Boyer–Moore Parallelization on Multi-Core Microprocessors ................. 27
4.3 Boyer–Moore Parallelization on Many-Core Accelerators ...................... 28
  4.3.1 Using Graphics Processing Unit (GPU) ...................................... 29
  4.3.2 Using Intel Xeon Phi Coprocessor ......................................... 31

5. Implementation ................................................................................. 34
  5.1 Tools and Architectures Used .................................................... 34
  5.2 Parallelization of Boyer–Moore Algorithm Using Hadoop .................. 36
  5.3 Graphical Representation ........................................................ 42

6. Experimental Analysis ..................................................................... 47
  6.1 Experimental Results ................................................................... 47
  6.2 Validation Results ....................................................................... 53

7. Lessons Learned ............................................................................. 54

8. Future Work .................................................................................... 57

9. Conclusion ....................................................................................... 58

References .......................................................................................... 59
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>Boyer-Moore Bad Character Shift Example</td>
<td>7</td>
</tr>
<tr>
<td>Table 2</td>
<td>Sequential Processing VS Parallel Processing</td>
<td>48</td>
</tr>
<tr>
<td>Table 3</td>
<td>Speedup VS File Size</td>
<td>49</td>
</tr>
<tr>
<td>Table 4</td>
<td>Number of Nodes and Actual Time</td>
<td>51</td>
</tr>
<tr>
<td>Table 5</td>
<td>Processing Time for Different Chunk Sizes</td>
<td>52</td>
</tr>
</tbody>
</table>
List of Figures

| Figure 1: Simple Pseudo Code of Boyer–Moore Algorithm | 5 |
| Figure 2: Boyer-Moore-Horspool Algorithm | 10 |
| Figure 3: HDFS Architecture | 16 |
| Figure 4: MapReduce Example | 19 |
| Figure 5: MapReduce 2.0 Framework | 20 |
| Figure 6: Simplified Block Diagram of the NVIDIA Tesla K20X GPU | 30 |
| Figure 7: General Structure of the Intel Xeon Phi Coprocessor Architecture | 32 |
| Figure 8: System User Interface | 37 |
| Figure 9: AWS EC2 Instances View | 38 |
| Figure 10: Randomly Named Input Folders | 39 |
| Figure 11: How The Chunks are stored in an S3 bucket | 40 |
| Figure 12: AWS Cluster Configuration View | 41 |
| Figure 13: How the Final Result is Displayed | 42 |
| Figure 14: Simplified Execution Flow Model | 43 |
| Figure 15: MapReduce Class Diagram | 44 |
| Figure 16: Servlet Class Diagram | 44 |
| Figure 17: System Activity Diagram | 45 |
| Figure 18: System Sequence diagram | 46 |
| Figure 19: Sequential Processing VS Parallel Processing | 49 |
| Figure 20: Speedup vs File Size | 50 |
| Figure 21: Number of Nodes and Actual Time | 51 |
| Figure 22: Processing Time for Different Chunk Sizes | 53 |
Abstract

Parallel Boyer-Moore String Matching Algorithm Using Hadoop

By

Cesar Zalzalah

Master of Science in Computer Science

String matching is an imperative problem in computer science. This topic becomes increasingly complex when the search text is very large. In this thesis, I implemented a cloud-based parallel Boyer-Moore string matching algorithm using Apache Hadoop. I then evaluated, compared, and contrasted its performance with a sequential version.

Boyer-Moore algorithm is one of the most efficient exact string matching algorithms. A cloud-based parallel implementation of Boyer-Moore string matching algorithm using Hadoop provides significant processing capacity and speed when dealing with very large search text. This type of approach is highly necessary for many pattern matching applications, such as signature-based malware detection, computational biology, search engines, artificial intelligence, and many other fields.

Experimental results show that the parallel implementation is superior to the sequential method when the search text is large. However, for smaller files the sequential method is better with less overhead and faster processing speed.
1. Introduction

String matching is a primary task used in many applications such as signature-based malware detection, computational biology, web search engine, to name a few. It is a process of finding all appearances of a string $S$ in a search text $T$. Boyer-Moore algorithm is one of the most popular string matching algorithms that is mostly used in its sequential form. This problem becomes increasingly complex when dealing with large search data. Apache Hadoop is a very efficient system to store and process vast sets of data primarily in a parallel way.

In this thesis I implemented and evaluated a cloud-based parallel Boyer-Moore string matching algorithm using AWS Hadoop. This implementation is very useful for string matching on large data files.

The parallelization of Boyer-Moore algorithm was done using MapReduce methods on AWS (EMR) platform. The main program containing the implementation of the algorithm is packaged into a JAR file then stored in an S3 bucket. The user interface of the system is made of a JavaServer Pages (JSP) that takes in user input and passes the values to a Servlet at the back end to performs required operations on them. Both JSP page and Servlet run on Apache Tomcat web server that is installed on an EC2 instance. After the search text file is uploaded, the file is divided into chunks of smaller files equal to the number of nodes the EMR will be running.

The system uses two different types of EC2 instances: m1.large for the EMR and t2.micro for the web server. Each EMR EC2 instance is responsible for creating nodes that execute and run the JAR file on the chunk of data, which it is working on simultaneously. This method of search parallelizes both phases of the Boyer-Moore algorithm.
Experimental results demonstrate that the parallel implementation is superior to the sequential method when the search text is large. However, for smaller files the sequential method is better with less overhead and faster processing speed.

The thesis is structured as follows: Section 2 provides imperative related work. Section 3 presents an overview of the Boyer–Moore Algorithm with an example of how it works. Furthermore it introduces a very important variation of Boyer–Moore Algorithm, which is the Boyer–Moore-Horspool Algorithm. Also in this section, a quick overview is given about Big data, Cloud Computing, AWS, Hadoop framework, and finally Spark framework. Section 4 provides information about parallel architectures, specifically shared memory and distributed memory architecture. It also introduces different parallelization approaches for Boyer-Moore Algorithm on Multi-Core Microprocessors, the Many-Core Accelerator Nvidia Tesla K20 GPU and Intel Xeon Phi. Section 5 presents my proposed parallel Boyer–Moore string matching approach and explains the tools and architectures used. At the end of this section a graphical representation of the system is shown. Chapter 6 reports the experimental results of four different experiments. The first two compared the sequential method to the parallel method. The third test shows the effect of the search text size on the processing time, and the final test shows the processing time for different chunk sizes using the parallel approach. Chapter 7 presents valuable lessons learned while working on this project. Section 8 discusses various issues that can be addressed in the future. Finally, the research is concluded in Section 9.
2. Related Work

Much research has been done on different ways and methods to parallelize the task of string matching. String matching operations are ideal for parallelization because of the minimal data dependence. In [1] the authors proposed a new data parallelism method for exact string matching using a dedicated grid network of computers. They showed a significant improvement in running time and speed. In [2] the authors put forward a parallel string matching algorithm that is based on MapReduce method. They used the algorithm with ClamAV-free virus signature database to detect signature-based malware in unstructured data kept in Hadoop distributed file system environment. The report showed a correlation between the input file size and the execution speed. The bigger the file size the faster the execution speed becomes. In [3] the authors proposed and evaluated a parallel implementation of Boyer-Moore algorithm; they demonstrated it to be quicker and more useful than the sequential implementation. In [4] the team used a parallelized version of Boyer-Moore algorithm on Nvidia Tesla K20 GPU, the many-core accelerator Intel Xeon Phi, and the multi-core Xeon processor.

The parallel algorithm breaks up the input data into chunks then assigns each chunk of data to a different thread for parallel processing. The researchers [4] used multithreading techniques and dynamic scheduling to overcome the load balancing problem caused by data lying on the threads boundaries. They also used the algorithmic cascading method to lower the burden on the GPU shared memory. The scientists demonstrated that parallel version of Boyer-Moore algorithm showed around 17 times speedup on the many-core accelerators Intel Xeon Phi and around 45 times speedup on the Nvidia Tesla K20 GPU in comparison with a serial implementation on the multi-core Xeon processor.
S.V Raju and Vinay Babu [5] proposed an efficient parallel method for string matching algorithm. They considered an optical interconnection network the linear array with reconfigurable pipelined bus system (LARPBS) and 2D LARPBS for both exact and approximate string matching. This type of string matching has many existing uses such as Cellular Automata, string database, and computational biology. With the proposed method, the team was able to achieve the time complexity $O(1)$ for string matching on 2D LARPBS with the text and pattern that has not been previously processed. In this thesis I implemented and evaluated a cloud-based parallel Boyer-Moore string matching algorithm. The parallel architecture is built using AWS. Unlike other Boyer-Moore parallelization methods in this implementation both phases of Boyer-Moore algorithm were parallelized.
3. Background

This section introduces the Boyer-Moore algorithm. Additionally, the concepts of Big Data, Cloud Computing, and the cloud provider Amazon Web Services (AWS) are discussed. In section 3.5 Hadoop, the main engine of the Cloud Computing power, is examined. Finally, this chapter analyzes Apache Spark, which works similar to Hadoop but runs in memory.

3.1 Boyer–Moore Algorithm

Boyer-Moore (BM) algorithm [6] is a suffix-based, single-pattern string matching algorithm developed in 1977 by Robert S. Boyer and J. Strother Moore. It is easy to understand but a little hard to code. It works by trying to skip as many unnecessary comparisons as possible without losing any correct match. It is one of the most efficient general single pattern matching algorithms. Also, the algorithm performs increasingly better with big size alphabet if the search pattern is large. The reason for this is that the bigger the alphabet or search pattern size, the larger the shifts might be.

The working concept of this algorithm is very simple: given a search pattern (characters), the algorithm aligns the search pattern from the beginning of the text and starts searching from right to left shifting the search pattern from left to right. The central idea behind the comparison order is to shift the pattern as long as possible. The algorithm consists of two precomputed steps: a preprocessing step and a pattern matching step.

In the preprocessing step, the algorithm creates the bad character shift rule (used in case there is no match), which is a table with character shift rules calculated using the search pattern. It contains all of the characters in the alphabet “the alphabet of the text we are searching in” with
integers representing how far the algorithm will shift when there is no match. Any character in
the text that is not in the pattern will cause the algorithm to shift the pattern by a length equal to
the length of the pattern. If a character does exist in the pattern, then the algorithm will look up
the bad character shifts in the rule table and based on the shift number will realign the pattern to
the correct position. Figure 1 below illustrates the basic Pseudo Code of Boyer-Moore algorithm.

```
{ initialization of deltal and delta2 tables is omitted }
lastch. ← pat[patlen];
i ← patlen;
while i ≤ strlen do
  begin
    ch ← string[i];
    if ch = lastch then
      begin
        j ← patlen - 1;
        repeat
          if j = 0 then return i;
          j ← j - 1;
          i ← i - 1;
        until string[i] ≠ pat[j];
        i ← i + max(deltal[ch] , delta2[j]);
      end
    else
      i ← i + delt[ch];
  end;
return 0;
```

Figure 1: Simple Pseudo Code of Boyer–Moore Algorithm [10]

Example:

Text: mike went to the supermarket to buy an ice-cream

Pattern: ice-cream
Firstly, the algorithm creates the shift table that will contain the number of shifts for all characters. Characters that do not belong in the pattern “donated by a * “will have a value of 9 shifts as shown in Table 1.

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>c</th>
<th>r</th>
<th>e</th>
<th>a</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1: Boyer-Moore Bad Character Shift Example

Second, align the pattern with the beginning of the text. When we start comparing the pattern with the text we will notice that the character m in the pattern is mismatched with the t in the text:

mike went to the supermarket to buy an ice-cream

Since the character t is not part of the pattern we shift the pattern by 9 shifts:

mike went to the supermarket to buy an ice-cream

mike went to the supermarket to buy an ice-cream

Now the m in the pattern is mismatched with the s in the text. Since the s is not part of the pattern we again shift the pattern by 9 shifts:

mike went to the supermarket to buy an ice-cream

mike went to the supermarket to buy an ice-cream
Now since the character \( e \) is part of the pattern we shift the pattern based on the shift table we created earlier for each character in the pattern. In this case, we will shift the pattern by 2:

mike went to the supermarket to buy an ice-cream

\[
\text{ice-cream}
\]

Since the *empty space* is not part of the pattern we again shift the pattern by 9 shifts:

mike went to the supermarket to buy an ice-cream

\[
\text{ice-cream}
\]

Now, since the \( n \) is not part of the pattern we shift the pattern by 9 shifts:

mike went to the supermarket to buy an ice-cream

\[
\text{ice-cream}
\]

Lastly, we shift the pattern by 1 and we get exact matches so the algorithm stops:

mike went to the supermarket to buy an ice-cream

\[
\text{ice-cream}
\]

The second pre-computation step done by Boyer-Moore algorithm is called good suffix shift rule, and it is calculated when there is a character match. This action involves the creation of the good suffix shift table. This shift table is stored in an array with information about how the
pattern matches against suffix of itself. [7] After every mismatched character, Boyer-Moore algorithm picks the larger shift given by the two precomputed steps.

### 3.1.1 Time Complexity of the BM Algorithm

Most of the efficiency and performance Boyer-Moore algorithm achieves happens because of the time it spends $O(m+n)$ in its pre-computation phases. With a big size alphabet and pattern length, the algorithm is frequently expected to make long shifts of length equal to the length of the pattern bypassing a large number of unnecessary characters in the text.

In the worst case performance, that is when there are no matches in the text being searched, the time complexity of Boyer-Moore algorithm is $O(3m)$ ($m$: length of the pattern, $n$: length of text).

But if there is, at least, one match, the time complexity is $O(mn)$, which means that the larger the text that is being searched the slower the algorithm will run[8]. In the best case the algorithm has to check only $n/m$ characters; therefore, it is time complexity would be $O(n/m)[9]$. Lastly, the memory overhead of the algorithm is $O(m+n)$ bytes because of the two arrays created by the two pre-computation steps.

### 3.1.2 Boyer-Moore-Horspool Algorithm (BMH)

New faster and more efficient string(s) matching algorithm has been proposed within a short time after the release of Boyer-Moore algorithm. The first improved one was made by Horspool [10]. BMH is a simplified and improved version of the original Boyer-Moore algorithm introduced by R. N. Horspool. He observed that the second complex shift table is unnecessary as it does not usually provide long enough shift to make it worth the effort. Horspool modified the
original BM algorithm to use only the bad character shift (Figure 2) that is the rule that usually leads to the longest shift. When used with large alphabet size BMH performs efficiently.

```plaintext
delta12[*] \leftarrow \text{patlen}; \{\text{initialize whole array}\}
\text{for } j \leftarrow 1 \text{ to } \text{patlen} - 1 \text{ do }
delta12[\text{pat}[j]]. \leftarrow \text{patlen} - j;
lastch \leftarrow \text{pat}([\text{patlen}];
i \leftarrow \text{patlen};
\text{while } i \leq \text{stringlen do}
\begin{align*}
    &ch \leftarrow \text{string}[i];
    &\text{if } ch = \text{lastch then}
    &\quad \text{if } \text{string}[i - \text{patlen} + 1 \ldots ] = \text{pat then}
    &\quad \quad \text{return } i - \text{patlen} + 1 ;
    &\quad i \leftarrow i + \delta\text{ma}[ch];
\end{align*}
\text{end;}
\text{return } 0;
```

Figure 2: Boyer-Moore-Horspool Algorithm [10]

In [10] the performance of BMH has been compared to Boyer-Moore algorithm, and the researcher stated that BM and BMH have a similar performance with the advantage that BMH has a simpler preprocessing step. In [11] researchers experimentally demonstrated that BMH is faster than Boyer-Moore algorithm on randomly generated patterns and input strings. The time complexity for BMH preprocessing step is $O(m + n)$, and the time complexity of the search phase is $O(nm)$ in the worst case. In the best case the time complexity is $O(n/m)$, and in the standard case it is sub-linear.

### 3.2 Big Data

Big Data is a term that describes a type of data that is very large. This data can be Structured, unstructured, or semi-structured. It is very difficult to process the significant amount of data or/and the fast speed in which it moves using standard software, tools, and techniques.[14]
Big Data is characterized by five terms beginning with the letter "V". Initially, three conditions sufficed: *volume, velocity, and variety*, but more recently two more names have been added: *veracity* and *value*. *Volume* is the size of the data available for processing. The volume of the data is often expressed in terabytes or even petabytes, so it is very difficult to store and even harder to process. *Velocity* refers to the rate of data generation, which, as previously stated, is steadily increasing. This is important because many Big Data applications require very fast processing to derive information from data in a short enough time frame to be useful. A good example is detecting a computer intrusion event by analyzing connection patterns. If it takes 1 minute to detect the attempt, it is too late. The third V, *Variety*, references the fact that data comes in countless different forms. Some are carefully structured and can be easily parsed while others are completely freeform and need advanced techniques to process. *Veracity* refers to the reliability and accuracy of the data. Data like a set of sports scores may approach 100% efficiency, but dictated text may have a much higher error rate. It is possible to analyze a user’s Twitter profile, but users might fill it out with incorrect information. Lastly, *Value* refers to the potential gain that a set of data can provide after being analyzed. The techniques of Big Data can be compared to attempting to find a needle in a haystack, except it is uncertain whether there even is a needle to begin with.

The ecosystem of Big Data consists of a base layer of technologies like Hadoop, Spark, and HBASE. Above the layer is the infrastructure layer, composed of analytics solutions (such as prepackaged virtual appliances with Hadoop pre-installed), structured databases, and service providers (like AWS and Google Cloud).

Utilizing Big Data in a software project first requires a large dataset or datasets to analyze. After all, if the amount of data to be analyzed is small enough (say less than 1GB), the overhead
of the approach is probably going to outweigh any performance gains. Assuming a software project does indeed operate on a large data set, the next requirement is to select a processing framework. The traditional processing framework of choice is Hadoop or Spark. Next, one must decide whether to self-host the cluster of machines or use a cloud hosting providers like AWS or Google Cloud. After that, one should write any necessary algorithms within the chosen framework (usually in Java, but other languages can be used as well), deploying the JAR file (or other build artifacts), and running the procedure.

3.3 Cloud Computing

As cloud computing models became more needed and popular, it became necessary to define what precisely a cloud is. A precise terminology was put forth by the National Institute of Standards and Technology (NIST), which stated that a system must contain five essential elements to be considered a cloud: self-service on-demand, broad network access, resource pooling, rapid elasticity, and measured service. [15]

The NIST classifies the different cloud models into four types: private, public, hybrid, and community clouds. While all the models share the main characteristics of a cloud, they differ in how the system infrastructure is hosted and owned, and by the ways in which it is accessed. [16]

The concept of virtualization allows clouds to consolidate servers and maximize infrastructure by virtually dividing and pooling resources. As a result, abstracting workload from hardware allows for rapid and efficient scalability. Virtualization is a building block of cloud computing, but it is not an instance of a cloud, as it does not include all five of the essential cloud characteristics.
The NIST categorizes the different cloud architectures into three service types: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). Unlike the cloud models, which are closely related and can be combined, consumers typically choose a cloud application service based on their particular need, which can vary tremendously. [15]

All of the cloud types and services are highly cost-effective, accessible, scalable, and available, with public clouds having only a few possible disadvantages. Consumers pay for the services as needed and, therefore, save on the costs of the initial setup up.

Cloud services have higher availability than traditional computing environments. Most of the significant disadvantages, such as lack of security and control, pertain to the public cloud computing model. Since the cloud provider controls the infrastructure, the consumer cannot control the security and reliability of the cloud service. Other disadvantages, such as high upfront costs due to the cost of private cloud, can be amortized over time.

A cloud computing model can be applied to many existing software systems. Traditional hardware-centric client-server systems can easily transition to a private, public, or hybrid cloud based on theirs. With so many cost-effective solutions and new capabilities that can only be implemented in a cloud, such as big data solutions, there are many incentives for an organization or an individual to adopt a cloud service model.

3.4 Amazon Web Services (AWS)

Amazon Web Services (AWS) is a growing group of remote computing services offered over the internet that are the building block of Amazon Cloud platform. AWS makes transitioning to the cloud easy with world-class documentation. With AWS an acquisition of storage,
computation, and other services is fast and efficient. It is also very affordable “with no upfront fees, pay for what you use type of plan”, flexible, scalable, and reliable.

The most relevant AWS services for this research are Amazon EC2, Amazon EMR, Amazon S3, Amazon CloudWatch, and Elastic Load Balancing below is detail description of each.

Amazon Elastic Compute Cloud (EC2) is a web service with a simple interface that provides access to cloud-based server instances for discrete compute capacity. EC2 is the compute center for AWS cloud platform. Amazon offers many types of server images to run on EC2 users can even run their images. The best part of EC2 is the ability to scale up or down computing resources in a matter of minutes or automatically using Auto Scaling.

Amazon Elastic MapReduce (EMR) is a web service for processing large data sets in a quick and cost-efficient way. It is great for managing certain types of problems, for example, biological data analysis, data mining, web indexing and image processing and many others. Amazon EMR utilizes Apache Hadoop framework and other AWS products, to both distribute and process data sets across a distributed cluster of Amazon EC2 instances.

Amazon Simple Storage Service (Amazon S3) is an online cloud based storage service designed to be extremely scalable. It supports unlimited data storage and bandwidth usage. Amazon S3 is very reliable it saves its stored data in multiple facilities and multiple machines within those facilities. It has a guaranteed server uptime of 99.99%. Users can access it online anytime through its simple web services interface.

Amazon CloudWatch monitors AWS and customer’s applications and services running on it. It makes it possible to collect and track metrics data and react quickly to any potential problem in the cloud system.
Elastic Load Balancing (ELB) automatically routes incoming application traffic across the available EC2 instances in the cloud. Elastic Load Balancing also monitors EC2 instances to check their health, if it detects any unhealthy instances. It automatically reroutes traffic to other available healthy instances, when and if the unhealthy instances become healthy again it restores traffic to them. Elastic Load Balancing can be enabled within a single or across multiple availability zones, for even better application performance. [17] [18]

3.5 Hadoop

Hadoop is a framework that is written in Java and which provides expandable distributed storage and processing platform. An entirely open source Apache software framework based on Google’s Map and Reduce functions. Hadoop is designed to enable users to efficiently process vast sets of data in mostly parallel fashion using a cluster of commodity machines that scale to hundreds or thousands of nodes. It coordinates work among the groups of nodes. If one node fails, Hadoop shifts the task to another available node that enables it to operate without losing data or interrupting the work process. Hadoop solves problems that require examination of all the available data. For example, string matching and image processing require every single record to be read, and often interpreted in the context of similar records. [19]

Hadoop structure includes the processing portion, which is called MapReduce, and the storage portion, which is Hadoop distributed file system HDFS or any other supported file type such as Amazon S3 file system, CloudStore, FTP File system, mapR FS file system. Hadoop has grown to be a very useful popular tool to manage the needs of Big Data and cloud computing in a cost-effective manner.
3.5.1 Hadoop Distributed File System (HDFS) Architecture

Hadoop Distributed File System HDFS is a distributed, scalable, portable and extremely fault-tolerant file system that was originally derived from Google File System (GFS). It is written mostly in Java programming language, with some native code in C and command line utilities written as shell-scripts. HDFS is based on Master/Slave architecture and uses the TCP/IP layer for communication. Clients use a remote procedure call (RPC) to communicate with each other. Replicating the data across multiple hosts makes the system reliable. [20][21]

HDFS Structure includes Namenode, which is a master machine that manages the storage cluster, executes namespace operations of the file system (i.e. renaming, opening, closing, files, and directories), regulates access to files by clients, and determines the mapping of data to the DataNodes, which is another HDFS component, that handles reading and writing requests from the clients, blocking creation, deletion, and replication when the NameNode request it.

![HDFS Architecture](Figure 3: HDFS Architecture)
HDFS can store very large files across several nodes. It complements MapReduce because jobs can be scheduled according to which nodes have stored which pieces of data, which in turn save I/O bandwidth.

The file system namespace supports a traditional tree-based directory structure e.g. /users/mark/file7. The NameNode manages the file system namespace. Files are broken up into blocks of a particular size, often 64 MB except for the last one. Redundancy of each file can be set by the application interfacing with the NameNode.

When a client application wishes to create a file, it first saves the data into a temporary local file, and all writes to this file are transparently redirected to it. When this local file exceeds the set HDFS block size, the client asks the NameNode to store it. The NameNode then adds the filename to the HDFS hierarchy, allocates a block in the system, and tells the client about it. Data is replicated across multiple nodes by some specified value, for example, the: value of 7 implies that any piece of data is available on at least seven machines.

At any time, replication value is set by the application storing the data. All data that clients receive from DataNodes have checksums if corruption occurs the client retrieves the data from another DataNode. NameNode periodically pings the DataNodes to make sure they are functioning, If the NameNode fails to detect one or more DataNodes, it marks them as ‘dead’ and doesn’t send any more traffic to them The NameNode then figures out what data no longer meets the replication factor and re-replicates it. NameNode can be failed over manually to a backup in the event of failure through the implementation of a secondary Name Node.[20]
3.5.2 MapReduce 2.0 Framework

MapReduce is a deeply scalable, parallel processing framework that leverages the concept of a map and reduces functions with some Shuffle and Sort phase. MapReduce follows similar logic like functional programming. It runs mappers and reducers on the nodes themselves, alongside the data in the HDFS. With MapReduce and Hadoop, compute is executed at the location of the data, rather than moving data to the compute location; data storage and computation coexist on the same physical nodes in the cluster. A typical program in MapReduce contains two primary functions: the Map function and the Reduce function. The map function computes an intermediate result of different parts of the input, and the reduce function merges the intermediate results into an end product.

The Map operation processes input data, and the Reduce operation puts together intermediate results into a final result. The Shuffle and Sort are done automatically by Hadoop. Both Map and Reduce functions use key-value pairs specified by the programmer as input and output. Data is broken and stored in chunks of 64MB or 120MB. The data chunks then get replicated for fault tolerance and reliability.
Figure 4: MapReduce Example

In MapReduce, an application is either one job or a DAG (directed acyclic graph) of jobs. MapReduce architecture includes ResourceManager, a per-application ApplicationMaster, a per-node slave Node Manager and a Container. The ResourceManager and the NodeManager form the distributed data computation model. The ResourceManager acts as the master controller that watches and allocates resources to all the applications in the system.

The ResourceManager contains two elements: Scheduler and Applications Manager. The Scheduler allocates resources to the various running applications based on their resource requirements. The Applications Manager handles managing incoming jobs, deciding which
container should start running application specific to ApplicationMaster. The Applications Manager also performs the task of restarting the ApplicationMaster container if it failed.

The per-application ApplicationMaster acts as an arbitrator that facilitates applications requests for specific resources and works with the Node Managers(s) in executing and monitoring jobs. Below is an example of a resource request:

   <resource-name, priority, resource-requirement, number-of-containers>

The per-node slave Node Manager monitors the resource usage (memory, CPU, network, disk) of containers and reports the information to the ResourceManager/Scheduler. The container acts as a resource allocator granting the rights for the applications to use things like memory, CPU, and other resources on a particular host. [22][23][24]

Figure 5: MapReduce 2.0 Framework

3.6 Apache Spark

In the business world, Big Data is applied to many things from tracking business processes and transactions to spam and virus detection. However, whatever the procedure, the data is only
as valuable as the decisions that it generates. Big Data provides the link between data and decisions by enabling interactive queries on both historical and live (i.e. streaming) data, thus permitting higher quality decisions to be made more quickly.

Most current Big Data frameworks focus on on-disk data analysis, which allows investigation of large datasets, but severely hampers speed. In contrast, Apache Spark seeks to improve on this by focusing heavily on in-memory processing. Since memory access is significantly faster than disk access, and since memory density is still growing in agreement with Moore’s law (unlike disk capacity that has run into design barriers), this is now a feasible approach. That is exemplified by the fact that ninety percent of the jobs in clusters used by Facebook, Yahoo!, and Bing can now fit into memory [25].

Like Hadoop, Spark is capable of being highly parallelized, permitting faster processing and a higher degree of fault-tolerance and failure mitigation (e.g. if a node fails, another node can take over). While this does not guarantee interactive (real-time) processing, it does allow one to get very close. Spark is compatible with existing data frameworks like the HDFS file system used by Hadoop, and can even run within the Hadoop framework. Recently, in fact, Spark became available on AWS. The level of interest can be gauged by the fact that more than 14 large companies from Adobe Systems to Yahoo! are contributing to its development.

The most interesting development Spark Streaming can be described as “Hadoop on steroids.” It permits large-scale, near real-time stream processing, is capable of scaling to hundreds of nodes, and can achieve latencies on the order of seconds. Spark Streaming can integrate with regular Spark’s batch and interactive processing facilities, and itself provides a simple batch-like API for implementing sophisticated algorithms. Near real-time stream processing is well suited
for applications such as live social network trends, intrusion detection systems, as these require both very large clusters and very low latencies.

Traditional stream processing frameworks like Storm and Trident utilize event-driven, record-at-a-time processing models. If a node goes down, the state is lost, making fault tolerance difficult. Storm replays a record if not processed by a node, but can still lose a mutable state to node failure, while Trident processes each record only once using transactions to update state, thus making the process fairly slow.

In contrast, Spark Streaming runs a streaming computation as a series of subtle, deterministic batch jobs generated by breaking up a stream into segments of X seconds (where X is some small number). Data is loaded into a data structure called a Resilient Distributed Dataset (RDD) and processed by a series of transformations into other types of RDDs. They are designed for parallel and fault-tolerant operations, since they can be reconstructed if the node running an operation on them fails. Spark Streaming remembers the series of steps used to create the RDD and simply repeats them.

In comparison with Storm and another framework from Apache called S4, Spark Streaming holds the advantage in speed, being more than five times faster than Storm and more than 60 times faster than S4 [26]. At the same time, it keeps most of the simple syntax used by traditional Spark, thus making it relatively easy to use.

As a point of reference, Spark has beaten Hadoop in a contest called the Daytona Gray Sort 100TB Benchmark, where the challenge was to sort 100TB of data. Hadoop achieved a time of 72 minutes using a cluster of 2100 nodes, but Spark ran the procedure in a mere 23 minutes using only 206 nodes [27]. Thus, they achieved three times the performance at less than a tenth of the cost, an impressive result.
Spark would be most appropriate in a software system where Hadoop would be an option as it can implement the MapReduce paradigm with much greater speed. However, if extensive processing on a disk is required, the advantages of Spark over Hadoop might be negligible. Spark-Streaming appears to be the best solution for large sets of streaming data, assuming it can mostly be loaded into memory.
4. Parallel Architecture

Parallel architecture falls under two basic categories depending on how the processing elements communicate: the shared memory architecture and the distributed memory architecture. Shared memory parallel systems (multiprocessor, multicore processors, GPU) read and write to a shared memory area. While distributed memory parallel systems use communications network to send and receive information messages enabling them to coordinate work between each other.

The most popular API for shared memory parallel architecture processing is OpenMP, which is a set of compiler directives, environmental variables, and library routines for parallel applications. OpenMP makes it easier to write parallel multi-threaded applications in C++, C, and Fortran and it is jointly supported by the largest computer hardware and software vendors. OpenMP accomplishes parallelism through a special use of threads. Threads communicate in a shared address model by sharing variables. The Programming structure is based on Fork-Join model where the code is made of parallel and serial portions and the application alternates between a serial and a parallel state with a master thread creating a team of parallel threads as needed.

The most popular standard for distributed memory processing is Message Passing Interface (MPI), which is a message passing library standard that is vendor-independent with a big collection of collective operations for data movement, point-to-point communication routines, and synchronization. It was defined by a large group of computer vendors, researchers, software companies and users. The primary goal of the Message Passing Interface is to establish a portable, flexible, and adequate standard for message passing. Message Passing Interface runs on any hardware platform “Distributed, Shared, or Hybrid memory”. Unlike the shared memory
architecture, distributed memory architecture depends on nodes, where each node has its partition of the memory location that is not accessible by any other node. As noted in [28] the two crucial attributes of message passing are the support for only explicit parallelization and the partitioning of the address space between each node which enforces the communication and synchronization between them [29].

To parallelize an algorithm, we first have to determine if it’s parallelizable or not if it’s indeed parallelizable then we have to clearly identify the parallel parts because the performance increase will be proportional to the percent of parallelization we were able to do to the algorithm.

The identification of parallelism includes finding sequences of code that might be executed in parallel by different processors [30].

According to Amdahl’s Law [31], the top performance increase that can be gained is equal to

\[
\frac{1}{1 - \frac{P}{N} + \frac{P}{N}}
\]

where \( P \) is the section of the parallelizable part of the algorithm and \( N \) is the total number of the processing nodes. [32]

4.1 Text Partitioning and Data Distribution

There are two primary stages in the design of parallel algorithms: breaking a computation into smaller pieces and appropriating them to many processing units for parallel execution. Since the load can be assigned unevenly, each part might have different processing time. To improve the performance and minimize the processing time of parallel data applications it is necessary to organize the available data in a map.
The static and the dynamic mapping are the two load balancing mapping techniques that can reduce the overhead. The static mapping can minimize the processing period and delays in communications when the information about the speed of execution and available resources are known. The dynamic mapping is used for balancing the workload by taking a load of heavily used processors and redistributing it into the units with a smaller load. Dynamic algorithms may be executed either by centralized or distributed methods, which differ by the amount of processors responsible for making the scheduling decisions. The best way to break the data into smaller parts is to use a line partitioning that separates data on a per-line basis, in which a two-dimensional array is cut into segments of lines for the purpose of reducing the number of overlapping characters.

In the following equation, let $p$ be the number of existing processes and $s$ the number of the text segments. The static mapping will allow each process to take a fragment of the sentence that consists of

$$\left\lceil \frac{n^2}{p} \right\rceil + n(m - 1)$$

text characters. In the case of dynamic mapping the text is broken into more segments than the amount of the available processes ($s > p$). Each process will consist of $n sf + m - 1$ characters where $sb$ is a specific size of a fragment at the time of the execution. The overlap $nm - 1$ is needed to make sure that each process gets the necessary data out of which we can derive $snm - 1$ added characters that can be processed for the dynamic mapping and $pnm - 1$ for the static. [33]

Additionally, there is a communication model Master-Worker that has a master (a node) that assigns data fragments to the workers (other nodes). The master collects the results after the workers finish processing those data segments. The communication overhead, which occurs
during the data assignment among the worker and master nodes, can be minimized by a dynamic allocation of text pointers [34].

4.2 Boyer–Moore Parallelization on Multi-Core Microprocessors

As mentioned earlier, Boyer-Moore algorithm consists of the preprocessing step and the pattern matching step. In Multi-Core Microprocessors systems, the pattern matching step can be parallelized. Three problems need to be addressed before attempting to parallelize the algorithm. The first problem: as the length of the search text increases so does the time complexity of the algorithm if there is a match. If there is no match, then the time complexity of the algorithm depends only on the length of the pattern.

The second problem that needs to be addressed is the big variances in the workload distribution on the working threads. The main cause of this problem is when distributing the chunks of search text amongst a group of working threads for the parallel processing some of those threads would find matches so they will finish later than the threads that didn’t find any matches. The solution to this problem would be to balance the workload amongst the working threads. The third problem that needs to be addressed is finding the pattern strings that are lying on the boundaries of the chunks of data. It is impossible to redundantly copy the data because as the number of threads and the pattern size increase so does the big overheads associated with the redundant data copy.

The algorithm is parallelized by splitting the search text into many small chunks. Then distribute the small chunks of data into a group of available threads. The threads attempt to find a match for the pattern in the chunk of data it is handling.
When making the search text chunks, a copy (pattern_length-1)-bytes of data need to be redundantly copied from its neighboring chunks. Experimental results show that the number of pattern matches in each thread may vary widely; therefore, the final execution time is bottlenecked by the completion of the last thread.

One solution to minimize the inefficiency of the idle threads is to split the search text into smaller chunks and delegate multiple chunks to each thread also to use dynamic scheduling policy so that threads that finish its assigned chunk can take another available chunk [4].

4.3 Boyer–Moore Parallelization on Many-Core Accelerators

Many-core accelerator chips such as Intel’s Many Integrated Core (MIC) architectures, and the Graphic Processing Units (GPUs) from Nvidia, among others, provide orders of magnitude more parallelism and processing power than regular CPUs.

These chips are becoming very popular in the High-Performance Computing (HPC) server market where the need for parallel processing power is always in demand. Today, application developers for these many-core chips are reporting 10X-100X speedup over sequential code on traditional microprocessors [35]. Both Nvidia GPUs and Intel Many-Core Accelerators have a large number of cores, multiple threads per core, large peak performance for the double precision arithmetic, large amounts of onboard memory, and advanced cache hierarchies.

Currently, the accelerator market is dominated by AMD /Nvidia but with the release of Intel’s accelerator the Xeon Phi coprocessor, AMD market dominance might be numbered. By the time Intel made its official debut in 2012 seven systems on the Top500 list were already using Intel accelerator chip. According to Intel, its coprocessors have advantages over GPGPUs. They are easier to program and works independently of CPUs [36].
4.3.1 Using Graphics Processing Unit (GPU)

Graphical Processing Units (GPU), introduced in the 1980s were made to offload the processing tasks from the CPU. GPUs usually consist of the multi-core architecture where each core consists of thread processors having the capacity of processing and executing hundreds of threads simultaneously. These threads are grouped into blocks, where each block talks with a shared memory exchanging and handling the data to be processed. The number of blocks, as well as the number of threads per block, is assigned as per the capacity of the GPU. Nvidia Tesla K20 GPU has overwhelmingly improved the performance of the earlier GPUs.

In [37], the authors were able to successfully parallelize Boyer-Moore algorithm using NVIDIA Tesla K20 GPU. Tesla K20 GPU chip contains multiple thread blocks (also called Streaming multiprocessor) where each block contains multiple thread processors that execute in Single Instruction multiple data mode (SIMD). That means each thread operates on the same instruction on their own data streaming from the device memory to the cache memories and the registers of the thread processor. Each thread processor has its own registers and local memories. For example, shared memory, Level-1 cache and read-only data cache are integrated with the thread block of the Nvidia Tesla K20 GPU. Whereas level-2 cache is integrated in the GPU chip and is used by all the thread blocks (See Figure: 6).
Figure 6: Simplified Block Diagram of the NVIDIA Tesla K20X GPU [40].

GPU communicates with the host CPU through the Global memory located in the off-chip device memory. Data in this memory can be stored in the L-2 cache and the Read-Only data cache.

When Parallelizing Boyer-Moore algorithm on the GPU, the input data that was stored from the host memory to the Global memory is partitioned and loaded into the Streaming Multiprocessor (the blocks). Then as each block has multiple threads, each thread is assigned a chunk of that partitioned data and searches the pattern in their assigned target chunk in parallel to other threads. High degree of multithreading capability provided by the GPU also overcomes the irregular execution times of the different threads.
The data is mapped to each Streaming Multiprocessor according to the size of the Shared memory to generate a number of blocks which is assigned to it. In a case where the number of blocks increases beyond the capacity of the SMX, the SMX becomes saturated and the blocks stay idle. Therefore, to minimize this problem, multiple blocks are cascaded algorithmically [38], thus increasing the size of the blocks and the amount of work assigned to each thread in that block. This results in a decrease in the number of blocks that are mapped to each SMX, and, therefore, a reduction in the number of idle blocks that are not mapped, which in turn improves the multithreading and ultimately the performance.

4.3.2 Using Intel Xeon Phi Coprocessor

Based on Intel Many Integrated Core (MIC) architecture the Xeon Phi coprocessor combines many Intel CPU cores onto a single chip. In [4] the authors used Xeon Phi 5110P for the parallel implementation of the Boyer-Moore Algorithm.

Intel Xeon Phi Coprocessor is based on x86 technology with support for 64-bit architecture. It contains 60 in-order compute cores and one service core thus each chip contains 61 cores in total. The cores are connected by bidirectional ring interconnect (see Figure 7).
Each core can execute four threads through the four-way Hyper-Threading and has a 512-bit vector processing unit (SIMD unit) which corresponds to eight double precision or sixteen single precision floating point numbers. Each core has both L1 and L2 data cache, and it provides Maximum 16-channel GDDR memory interface with an ECC option and 8GB total system memory. The Xeon Phi coprocessor connects to the host Intel Xeon processor through a PCI Express 2.0 x 16 interfaces [41].

What makes this hardware accelerator chip very promising is its support of the same programming languages and same tools used for programming Xeon CPU; it also can run applications written in Fortran and C/C++ programming languages. An application can be run on the Xeon Phi Coprocessor using one of two modes of operations: an offload mode and a native mode.
In the offload mode the application must be modified to instruct the compiler to generate code that would allow it to run on the host side, and only the selected area are offloaded to the Xeon Phi coprocessor [42].

In the native mode the application runs entirely on the Xeon Phi. Native mode is a fast way to get an existing application suitable for native execution to run on the Xeon Phi with minimal changes. To use the native mode the –mmic option has to be used in the compile command.

The user process on the Xeon Phi has up to sixty cores (plus one service core). Since each core can execute four threads through the four-way Hyper-Threading [41], up to two hundred forty threads per every Xeon Phi Coprocessor can be utilized. The authors in [43] apply the dynamic scheduling on the Xeon Phi to be able to use up to two hundred forty threads.

For parallelizing Boyer-Moore algorithm on Intel Xeon Phi coprocessor the authors in [43] used the same parallelization methodology as the one used for the Multi-Core Microprocessors. The only exception is that they used the off-load mode of the Xeon Phi so that only selected parallel kernel regions can be offloaded from the host multi-core processor to the Xeon Phi Coprocessor [42].

The algorithm they used first copied the search text from the host system memory to the memory of the Xeon Phi. The host CPU controlled both: the parallel kernel execution on the Xeon Phi and the copying of the execution result from the Xeon Phi to the host system memory.
5. Implementation

Information presented in this section includes identification and descriptions of the tools and architectures used also the detail implementation of the system. This section also introduces graphical models of the system.

5.1 Tools and Architectures Used

The project software was designed to follow the Model–View–Controller (MVC) architecture to maintain the independence between the different modules used in the application. A typical software developed using MVC architecture is usually divided into three parts: Model, View, and Controller. The Model module contains the business logic of the application where the core logic is implemented. The View module contains all the files that directly or indirectly create the user interface. The Controller module acts as an event handler that connects the View module with the Model module. Keeping the modules separated helps in building a robust and well-maintained application.

The View module uses a JSP page called “index.jsp” as a front-end technology, which is a single entry point for the user. This JSP page contains a simple HTML form that asks the user to upload the text file, enter the search phrase, select the number of EC2 instances to use, and choose whether to use a sequential or parallel processing. Once the user submits the form, the control is redirected to a Servlet at the back end that receives the values of the form parameters and performs required operations on them. This JSP file and the Servlet at the back end runs on Apache Tomcat web server that is installed over an EC2 instance.
The EC2 instances powering the system are of two sizes: m1.large for the EMR and t2.micro for the Apache Tomcat web server. Each of the EMR EC2 instances is responsible for creating nodes that execute and run the job JAR file simultaneously. These EC2 instances follow the 64-bit architecture with two vCPUs for processing, memory of 7.5 GiB and 2 X 420 GiB for storage.

Elastic MapReduce (EMR) creates and configures a cluster upon which all the JAR files can run simultaneously. The user employs the JSP form to specify the number of EC2 instances utilized in the EMR, which is responsible for configuring those EC2 instances and executing them in a parallel. Currently, AWS EMR cluster uses Amazon Machine Image (AMI) version 2.4.2 and the Hadoop version of 1.0.3.

Simple Storage Service (S3) acts as a repository service. A bucket within S3 is a simple directory that contains files and folders. In this S3 directory the user needs to upload the JAR file that contain the logic behind the parallel Boyer-Moore implementation and the the search text file.

The system user interface is accessible using a public URL. Both the JSP file at the front end and the Servlet file at the back end require a web server to be installed on a EC2 instance. Apache Tomcat Web Server was selected because of its active support community. It is installed over a t2.micro EC2 instance. This t2.micro instance uses one vCPU with six CPU credits per hour and a memory of one GiB; and for repository it utilizes Elastic Block Storage (EBS). The physical processor employed in t2.micro is from Intel Xeon family with a clocking speed of 3.3GHz. To install Apache Tomcat on an EC2 instance, an SSH client like PuTTY has to be used.
Because Amazon provides a toolkit specifically designed to work with Eclipse IDE as a plugin, Eclipse is used to write necessary code and to make the process of JAR file creation easy. This toolkit can be installed from the Eclipse plugin market.

AWS uses the concept of private and public key to provide a remote connection to its instance. Upon creation of the “t2.micro” instance, AWS provides a private key with the “.pem” extension. A tool called WinSCp is implemented to convert this “.pem” file into “.ppk” file because the software PuTTY only supports “.ppk” extension private keys.

Text files from the works of Shakespeare were used to test the system. The tool fJoiner combined multiple text files to create large ones. The files were obtained from the URL below:

http://www.textfiles.com/etext/AUTHORS/SHAKESPEARE/

As this Hadoop application is written in Java using Eclipse IDE, the Java SDK needs to be installed on the computer where the program is written. For using Amazon Web Services remotely, AWS provides a special Amazon SDK that contains necessary libraries for creating EC2 instances, setting up clusters, storing and reading data from the S3 storage. The implementation of the Eclipse Amazon toolkit allows all the necessary JAR files to be automatically added to the program as a part of Maven dependency in the application.

5.2 Parallelization of Boyer–Moore Algorithm Using Hadoop

The parallelization of Boyer-Moore algorithm was done using MapReduce methods through AWS (EMR). The main program containing the implementation of the algorithm is packaged into a JAR file then stored in an S3 bucket. The user interface of the system is made of a JSP page that is hosted on Apache Tomcat Web Server, This JSP page is publicly accessible through the URL:
Figure 8: System User Interface

This URL is dynamically assigned; if the EC2 instance gets stopped or rebooted then a new URL gets assigned. Upon visiting the URL, the user can upload any type of text file including a compress one. Once the user specifies the file, he/she then specifies the number of EC2 instances to use in the EMR cluster: see Figure 9
After the search text file is uploaded, the file is divided into chunks of smaller files equal to the number of nodes the EMR will be running. These chunks of files are then stored in a unique folder within the Input folder. To create a unique name for this folder, the code below is used:

```java
UUID RANDOM_UUID = UUID.randomUUID();
String inputfilename = RANDOM_UUID.toString();
fileItems.write(new File(UPLOAD_DIRECTORY + File.separator + fileName));
String filelocation = fileItems.getName();
File inputFile = new File(UPLOAD_DIRECTORY + "/" + fileName);
```

Figure 10 below shows the view of the folder in AWS:
The size of each chunk is equal to: $(\text{Size of file}) / (\text{Number of EC2 instances} \times 2)$.

To specify the size of the chunk and to store it in a unique folder within the S3 bucket, the code below has been used:

```c
int fileSize = (int) inputFile.length();
long tempsize = (inputFile.length() / (nodes * 2));
int nChunks = 0, read = 0, readLength = PART_SIZE;
createFolder("stringsearch", inputfilename, s3client);
s3client.putObject(new PutObjectRequest("stringsearch", folderName, new
File(newFileName)).withCannedAcl(CannedAccessControlList.PublicRead));
```

Figure 11 below shows how the chunks are stored in an S3 bucket:
After dividing the file into chunks and storing them in an S3 bucket, the EMR cluster with the exact number of EC2 instances specified by the user gets created.

Once the cluster is set up, the next step is to configure the Hadoop on this cluster.

The cluster setup and Hadoop configuration is achieved by the code below:

```java
RunJobFlowRequest request = new RunJobFlowRequest
(FLOW_NAME, configInstance())
.withServiceRole("EMR_DefaultRole")
.withJobFlowRole("EMR_EC2_DefaultRole").withName("String Matching");
request.setLogUri(S3N_LOG_URI);

// Configure the Hadoop jar to use
HadoopJarStepConfig jarConfig = new
HadoopJarStepConfig(S3N_HADOOP_JAR);
jarConfig.setArgs(ARG_LIST);
jarConfig.setMainClass("com.searchstring/StringSearch");
```

Figure 12 below shows the cluster configuration page within AWS:
After setting up the cluster and configuring Hadoop on it, the number of nodes required to operate on the chunks of files are calculated by the below equation:

\[
\text{Number of nodes} = \text{Number of EC2 instances} \times 2
\]

Each node operates on one chunk of data available from the S3 bucket. Each node runs Boyer-Moore search algorithm on the chunk it is working on. Because the operations on all the nodes are executed simultaneously on all of the assigned chunks, parallelization is achieved.

Instead of reading the file line by line, EMR uses the Mapper class of Hadoop to map the entire file for quick access. After executing the Mapper method, the reducer combines the output per key and returns the final result in the form \(<\text{search-phrase}, \text{time}>\).

The reducer method returns whether the search phrase requested by the user is present in the specified text file or not. It also shows the time in milliseconds it took to execute the parallel algorithm.
Figure 13: How the Final Result is Displayed

This method of search parallelizes both phases of Boyer-Moore algorithm. Experimental results shows that the number of patterns found in each node may differ widely; as a result the execution final time is bottlenecked by the last node that finishes finding the match.

5.3 Graphical Representation
Figure 14: Simplified Execution Flow Model
Figure 15: MapReduce Class Diagram

Figure 16: Servlet Class Diagram
Figure 17: System Activity Diagram
Figure 18: System Sequence diagram
6. Experimental Analysis

In this section, four different experiments will be presented. The first two are comparing the sequential Boyer-Moore implementation to the Hadoop-based parallel implementation. The third experiment shows the effect of the search text size on the processing time. And the final experiment shows the processing time for different chunk sizes using the Hadoop-based parallel algorithm. All experiments were performed using Amazon Web Services (AWS). For all of the four experiments search text files from the work of Shakespeare were used; and for the Elastic MapReduce three to four Elastic Compute Cloud (EC2) were used. Each EC2 instance used is of type m1.large. These EC2 instances follow the 64-bit architecture with two vCPUs and the memory of 7.5 GiB. Each EC2 instance runs two processing nodes.

6.1 Experimental Results

Table 2 and Figure 18 demonstrate the first experiment conducted: comparing the processing time in a millisecond between the sequential implementation and the parallel implementation. For this experiment different search text files have been used ranging from 4.8 Mb to 115 Mb, and the search keyword used is “zzzzz”. This keyword was chosen because it does not occur in the search text file, which will force Hadoop to search the whole file. Also, for the parallel processing three EC2 instances were used.

As Table2 demonstrates, the performance of the sequential algorithm is consistent and predictable. The processing time almost constantly doubles every time the size of the search text file doubles. The performance of the parallel processing, demonstrated by Table 2, requires deeper analysis. The findings from Table 2 show that the performance of the parallel processing
is inconsistently fluctuating within a range of numbers. For example, when the input text increased by more than \( \frac{1}{3} \) the processing time did not increase by much and in some cases when the input file increased the processing time decreased. For instance, when using 48 MB file the processing time was 612 MS, but when using 38.4 MB file the processing time was 932 MS.

In conclusion, for smaller files the sequential method is superior with less overhead and faster processing speed than the parallel implementation. The other conclusion is that the parallel implementation works better with larger text files.

<table>
<thead>
<tr>
<th>Search Text File in (MB)</th>
<th>Sequential Time (Millisecond)</th>
<th>Parallel Time (Millisecond)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.8</td>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>9.61</td>
<td>11</td>
<td>132</td>
</tr>
<tr>
<td>19.2</td>
<td>21</td>
<td>134</td>
</tr>
<tr>
<td>28.8</td>
<td>32</td>
<td>407</td>
</tr>
<tr>
<td>38.4</td>
<td>51</td>
<td>932</td>
</tr>
<tr>
<td>48</td>
<td>74</td>
<td>612</td>
</tr>
<tr>
<td>57.0</td>
<td>79</td>
<td>925</td>
</tr>
<tr>
<td>67.2</td>
<td>105</td>
<td>1026</td>
</tr>
<tr>
<td>76</td>
<td>179</td>
<td>1383</td>
</tr>
<tr>
<td>86.5</td>
<td>194</td>
<td>1540</td>
</tr>
<tr>
<td>96.1</td>
<td>226</td>
<td>1096</td>
</tr>
<tr>
<td>115</td>
<td>250</td>
<td>1191</td>
</tr>
</tbody>
</table>

Table 2: Sequential Processing vs. Parallel Processing
Figure 19: Sequential Processing vs. Parallel Processing

For the second experiment, the data obtained from the first experiment was used to calculate the speedup achieved by using the parallel processing. Table 3 and Figure 19 demonstrate again that the performance of the parallel implementation increases as the input text file increases though it is not consistent when the input files are small due to framework overhead.

<table>
<thead>
<tr>
<th>Search Text File in (MB)</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.8</td>
<td>0.05</td>
</tr>
<tr>
<td>9.61</td>
<td>0.08</td>
</tr>
<tr>
<td>19.2</td>
<td>0.15</td>
</tr>
<tr>
<td>28.8</td>
<td>0.07</td>
</tr>
<tr>
<td>38.4</td>
<td>0.05</td>
</tr>
<tr>
<td>48</td>
<td>0.12</td>
</tr>
<tr>
<td>57.6</td>
<td>0.08</td>
</tr>
<tr>
<td>67.2</td>
<td>0.1</td>
</tr>
<tr>
<td>76</td>
<td>0.12</td>
</tr>
<tr>
<td>86.5</td>
<td>0.12</td>
</tr>
<tr>
<td>96.1</td>
<td>0.2</td>
</tr>
<tr>
<td>115</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Table 3: Speedups File Size
Figure 20: Speedup vs. File Size

For experiment three a compressed text file of size 187 megabytes and four EC2 instances were used. These four EC2 instances run eight processing nodes. Similar to experiment one, the search keyword “zzzzz” was used because it does not occur in the search text file which will force Hadoop to search the whole file. Table 4 and Figure 20 demonstrate an amazing finding that is adding more nodes will significantly increase the processing speed when the number of processing nodes is greater than two. But when the number of the processing nodes is greater than four, adding more nodes will not necessarily increase the processing speed, but it will increase the processing capacity. This experiment highlight a very important fact that too many unnecessary nodes will cause framework overhead also unnecessary financial cost on the other hand using a small number of nodes will cause a slowdown in processing.
Finally, we arrive at experiment four, which is an extension of experiment one, but for this experiment a larger text files and solely the parallel processing been used. The file sizes used range from 4.8 megabytes to 1140 megabytes; Elastic MapReduce was setup using four m1.large EC2 instances that yielded eight processing nodes. Same like in the past experiments the search

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>Processing Time (Milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5386</td>
</tr>
<tr>
<td>4</td>
<td>2320</td>
</tr>
<tr>
<td>6</td>
<td>659</td>
</tr>
<tr>
<td>8</td>
<td>2291</td>
</tr>
<tr>
<td>10</td>
<td>2420</td>
</tr>
<tr>
<td>12</td>
<td>1809</td>
</tr>
<tr>
<td>14</td>
<td>731</td>
</tr>
<tr>
<td>16</td>
<td>2200</td>
</tr>
<tr>
<td>18</td>
<td>1846</td>
</tr>
<tr>
<td>20</td>
<td>1246</td>
</tr>
<tr>
<td>22</td>
<td>658</td>
</tr>
<tr>
<td>24</td>
<td>836</td>
</tr>
<tr>
<td>26</td>
<td>925</td>
</tr>
</tbody>
</table>

Table 4: Number of Nodes and Actual Time

Figure 21: Number of Nodes and Actual Time
keyword “zzzzz” was used for the same reason as above. Table 5 and Figure 21 demonstrate the findings. Overall, as the size of the search text increased, so does the processing time. For small files, the increase in the processing time is inconsistent due to default framework overhead. However, as the search text gets larger the increase in the processing time becomes more consistent.

<table>
<thead>
<tr>
<th>Search Text File in (MB)</th>
<th>Processing Time (Milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.8</td>
<td>79</td>
</tr>
<tr>
<td>9.61</td>
<td>181</td>
</tr>
<tr>
<td>19.2</td>
<td>159</td>
</tr>
<tr>
<td>28.8</td>
<td>144</td>
</tr>
<tr>
<td>38.4</td>
<td>995</td>
</tr>
<tr>
<td>48</td>
<td>837</td>
</tr>
<tr>
<td>57.6</td>
<td>1665</td>
</tr>
<tr>
<td>67.2</td>
<td>743</td>
</tr>
<tr>
<td>76</td>
<td>1626</td>
</tr>
<tr>
<td>86.5</td>
<td>894</td>
</tr>
<tr>
<td>96.1</td>
<td>1800</td>
</tr>
<tr>
<td>115</td>
<td>1044</td>
</tr>
<tr>
<td>153</td>
<td>2903</td>
</tr>
<tr>
<td>187</td>
<td>2291</td>
</tr>
<tr>
<td>230</td>
<td>1924</td>
</tr>
<tr>
<td>442</td>
<td>2666</td>
</tr>
<tr>
<td>538</td>
<td>3430</td>
</tr>
<tr>
<td>635</td>
<td>3562</td>
</tr>
<tr>
<td>731</td>
<td>4272</td>
</tr>
<tr>
<td>885</td>
<td>3311</td>
</tr>
<tr>
<td>981</td>
<td>3635</td>
</tr>
<tr>
<td>1050</td>
<td>4098</td>
</tr>
<tr>
<td>1140</td>
<td>6459</td>
</tr>
</tbody>
</table>

Table 5: Processing Time for Different Chunk Sizes
6.2 Validation Results

Statistical integrity was validated by duplicating the same tests using different clusters that have been set up in a different region. The search text files used were entirely different from the ones previously been used. All tests has been carried out at different times than the original tests. The overall results obtained were similar to the first test results.
7. Lessons Learned

During the building of the project, I learned many valuable lessons. In this section, I will go over the most important ones. I have researched different cloud providers and found out that Amazon Web Services is the best available because it offers unbeatable technical support, and it has an active online community.

Due to my limited experience in writing Hadoop applications using Java SDK, I ran into many issues while coding the parallel approach of Boyer-Moore algorithm. There was a big learning curve involved as I had to read many articles and follow numerous online forums/threads to get a better understanding of how Hadoop works and how to exploit its features. The first thing important thing to understand when programming for Hadoop is how MapReduce works and what code it expects; the code has to be manually evaluated and formulated into MapReduce functions, there is no way to automatically do this. Also, it is crucial to understand how many Map and Reduce tasks are required to handle the particular problem.

When the system cluster is running, Hadoop creates a number of Map and Reduce tasks depending on the number of EC2 instances the user chooses to use. If too few EC2 instances is run, the processing of the task will be slow; on the other hand, if too many unnecessary EC2 instances are running, then some nodes will sit idle which will result in significant framework overhead and unnecessary financial cost, thus picking the right number of instances to use is very important.

Another important thing to note is that Hadoop has by default textinputFileFormat for input file processing that allows the mapper to read the file line by line and process it. But while doing
this project, I learned that using WholeFileInputFormat is better because it allows the mapper to read the file as whole rather than line by line.

After writing my version of the program, I thought of optimizing it for better performance. One idea I had was to remove the Reduce part from the “Map-Reduce” program because I thought there was no need for it and that the program’s performance was adversely affected by it. Working on this idea and removing the Reduce method proved to be a significant mistake as the program started to give erroneous results, and there was no performance improvement. After asking Amazon support for advice to remove the Reduce part, I realized that I have wasted eight days in vain. Based on the information I received from Amazon support, there is no way to implement the parallel algorithm without the Reduce section of the MapReduce approach. The lesson I learned was to always try to get advice from trusted experienced Hadoop developers before trying new ideas.

Implementing the algorithm and running it on AWS EMR was only a half of the journey. The next step was to develop a standalone application that automates the steps required to upload a file manually on an S3 bucket, create EC2 instances and configure, run, and shut down the EMR cluster. For this automation, AWS provides their Java packages which need to be imported while developing the applications code. The official documentation available on AWS website lacks proper explanation. As a result, I was forced to sign up for a paid support plan, which turned out to be a very helpful tool that guided me in completing the automation of the process. The only drawback was the time it took for the AWS technical support to respond, which lead to a delay in writing the application.
One advice I have for anybody choosing to use AWS is to keep an eye on the billing charges. It is imperative to terminate Elastic MapReduce (EMR) within an hour after a job has been submitted and processed because AWS charges an hourly rate for EMR.

I learned this the hard way: while I was learning how to use AWS I left EMR running after it finished processing a job, and it resulted in a $122 bill from Amazon.

Another important issue also is to pay close attention to the limitations imposed by the cloud provider on the account. For example, AWS allowed the use of only 20 EC2 instances at any given time. Also, it is necessary to learn the architecture of the different instances provided by AWS to be able to use the most suited one for the particular job.

Finally, running many EC2 instances does not mean faster results. Instead, it means more capacity and more overhead. Opening the excessive number of processing nodes does not mean faster processing speed; it just increases Hadoop processing capacity, not necessary the processing speed.
8. Future Work

In the future, I would like to test the MapReduce string matching algorithm on Spark framework and compare the speedup difference between it, Hadoop, and the sequential approach. The algorithm used can be optimized further to be more accurate precisely to solve the problem of data lying on the node’s boundaries so that the patterns lying on the chunks of data boundaries can be found. Furthermore, it would be interesting to examine the algorithms when executed on other cloud frameworks and examine the performance difference.
9. Conclusion

In this work, a cloud-based parallel Boyer-Moore string matching algorithm using AWS Hadoop was presented and evaluated. This implementation is very beneficial to string matching on large data files. The parallelization of Boyer-Moore algorithm was done using MapReduce methods on AWS (EMR) platform. Unlike other Boyer-Moore parallelization methods, in this implementation, both phases of Boyer-Moore algorithm were parallelized.

The experimental results show that for smaller files the sequential implementation is superior to the parallel implementation. The sequential implementation provides less overhead and faster processing speed. On the other hand, the parallel implementation fluctuates within a range of numbers due to default framework overhead, but for larger files the parallel implementation easily outperforms the sequential implementation.

In a simple speedup experiment, it was discovered that the speedup of the parallel implementation rises as the size of the input text file increases. Furthermore, adding more nodes will significantly increase the processing speed when the number of processing nodes is greater than two. However, when the number of the processing nodes is greater than four, adding more nodes will not necessarily raise the processing speed, but it will increase the processing capacity.

Based on the results presented above, I conclude that this parallel string matching approach provides compelling benefits to many string matching application that deal with large data sets.
References


shared memory parallel programming. The MIT Press


[33] Kouzinopoulos, C. S. & Margaritis, K. G. *Parallel implementation of exact two dimensional pattern matching algorithms using MPI and OpenMP*. Parallel and Distributed Processing Laboratory. Department of Applied Informatics, University of Macedonia


