5-14-2017

Facilitating High Performance Code Parallelization

Maria Abi Saad
Syracuse University

Abstract

With the surge of social media on one hand and the ease of obtaining information due to cheap sensing devices and open source APIs on the other hand, the amount of data that can be processed is as well vastly increasing. In addition, the world of computing has recently been witnessing a growing shift towards massively parallel distributed systems due to the increasing importance of transforming data into knowledge in today's data-driven world. At the core of data analysis for all sorts of applications lies pattern matching. Therefore, parallelizing pattern matching algorithms should be made efficient in order to cater to this ever-increasing abundance of data. We propose a method that automatically detects a user's single threaded function call to search for a pattern using Java's standard regular expression library, and replaces it with our own data parallel implementation using Java bytecode injection. Our approach facilitates parallel processing on different platforms consisting of shared memory systems (using multithreading and NVIDIA GPUs) and distributed systems (using MPI and Hadoop). The major contributions of our implementation consist of reducing the execution time while at the same time being transparent to the user. In addition to that, and in the same spirit of facilitating high performance code parallelization, we present a tool that automatically generates Spark Java code from minimal user-supplied inputs. Spark has emerged as the tool of choice for efficient big data analysis. However, users still have to learn the complicated Spark API in order to write even a simple application. Our tool is easy to use, interactive and offers Spark's native Java API performance. To the best of our knowledge and until the time of this writing, such a tool has not been yet implemented.

Facilitating High Performance Code Parallelization

by
Maria Abi Saad

B.S. Lebanese American University, 2006
M.S. Lebanese American University, 2008

Dissertation

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Electrical and Computer Engineering

Syracuse University
May 2017

Acknowledgements

Firstly, I would like to express my deepest appreciation to my advisor Professor C.Y. Roger Chen for his continuous support and advice. I am very grateful to have been the student of such a knowledgeable, enthusiastic, and ambitious visionary. Without his guidance and encouragement over the years, this dissertation would not have been possible.

Besides my advisor, my sincere thanks go to my dissertation committee members for taking the time to review my work and for their valuable insights, comments and suggestions. They are and in alphabetical order Prof. Ehat Ercanli, Prof. Erin Mackie, Prof. Qinru Qiu, Prof. Yanzhi Wang, and Prof. Edmond Yu.

I would also like to thank my colleagues at the Design and Modeling Lab who have accompanied me at different intervals of my research, and who have offered valuable suggestions and stimulating discussions. In particular, I am grateful to Elie for his support, his encouragement, his sarcasm and his friendship. This would have been a much harder wave to ride without him.

I am also grateful to my friends who always remind me that there is more to life than work although at times the lines get blurred. I would like to specifically thank all the wonderful people whom I have met during my time at Syracuse. You have made the snowy winters much easier to bear. Sigue bailando!

Lastly, I would like to thank my parents, my sister, and my brother who have supported me throughout this journey and have always believed in me. I am grateful for their understanding, their patience, and mostly for their love. This is dedicated to them.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

Traditionally, programs were written to be executed in a serial manner. A problem is broken down into a series of instructions which are executed sequentially one after the other. Hence only one instruction can execute at a time. These instructions execute on a single processor. Over the past twenty years, microprocessor technology has seen significant advances, such as increased clock rates, capability of processors to execute multiple instructions in the same cycle, and improved average number of cycles per instruction (CPI) [1]. This has led to an increase in the peak floating point execution rate (FLOPS). One of the issues that have arisen because of this improvement is the inability of the memory system to supply the processor with enough data at the required rate. Interest in parallelism for high performance computing has increased in recent years due to the physical constraints that prevent frequency scaling. As power consumption and heat generation become more and more of an issue, researchers are focusing on parallel computing since it involves using multiple processing elements. In addition to that, parallel computing provides increased access to storage units whether memory or disk, has scalable performance, and potentially lower costs over high-end processing units.

1.1 Parallel Computing

In a nutshell, parallel computing consists of the simultaneous use of several compute resources to solve a large problem by dividing it into smaller sub-problems that can be solved at the same time. The processing elements usually consist of a single computer with multiple cores, several computers connected by a network, specialized hardware for acceleration, or a combination of all three.

Parallel computing is being used extensively in a wide range of applications. Large scale applications in science and engineering rely on large configurations of parallel computers to solve and model difficult problems in several areas such as physics, genetics, geology, seismology, circuit design, weaponry… Parallel computing is also widely used in industrial and commercial applications which require processing large amounts of data. We mention a few examples: database or web servers, transaction processing, data mining, financial and economic modeling, medical imaging, image processing, applications in graphics and visualization…

Software developers therefore need to be familiar with a variety of parallel computing platforms and technologies to write efficient code capable of harnessing the parallel processing power. However, the development of parallel software is tedious, error-prone, time consuming, and non-trivial for programmers, especially if the aim is to develop an application that works well across several platforms.

We propose a system that facilitates the parallelization of high performance code.

1.2 Proposal

We propose a system that relieves the burden of dealing with parallel programming from the application developer. Our system handles both categories of main memory in parallel systems: shared-memory and distributed, as shown in Figure 1-1. Shared-memory systems have a uniform memory access meaning that each processing element can access memory with equal latency and bandwidth. Distributed systems exhibit a non-uniform memory access behavior since memory is logically and physically distributed. Parallel techniques that target shared-memory systems include multithreading and GPGPU computation. Parallel techniques that target distributed systems include message passing and map-reduce.

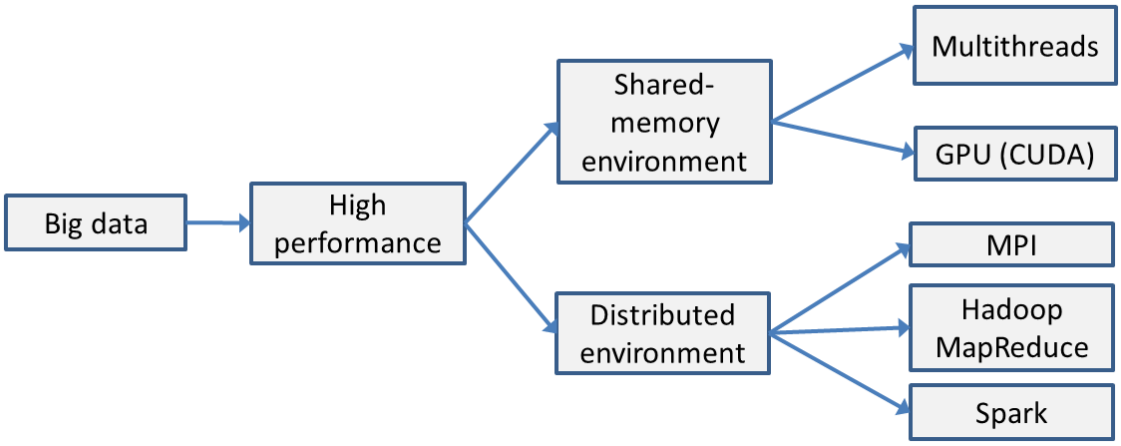


Figure 1-1: Parallel Systems

Our targets are Java developers. We have chosen Java since is mainly the language of choice for high performance applications, and it has gained popularity since its first inception due to several performance improvements to the Java Virtual Machine (JVM). Also and even though Java provides efficient yet difficult to program support for threads, it lacks complete and standardized support for GPUs and message passing. Hence, we saw an area where our contributions could be more significant.

Our central proposition revolves around parallelizing existing sequential Java code without having the user incur any extra effort other than using our library in order to obtain faster execution times. To validate our work, we have chosen the pattern matching application since pattern matching lies at the core of data analysis. However, our approach can be used with other applications, as we demonstrated in Chapter 2. We propose a method that detects a user's function call to search for a pattern using Java's standard regular expression library, and replaces it with our own parallel implementation using bytecode injection. Our parallel implementation consists of multithreading, using a GPU, and using MPI. We also perform feasibility testing with Hadoop.

In order to seamlessly parallelize pattern matching, we have used bytecode injection. Typically, injection can be done at the source code or at the bytecode level. We have selected the latter since the former suffers from the following shortcomings: (i) for the source code level injection to work, all the source code must be available which is not always the case, (ii) source code level injection is harder for complex transformations, (iii) with source code level, we will be manipulating strings, which have no particular meaning, and (iv) we would need a versatile parser. On the other hand, if injection is performed at the bytecode level, then (i) there is no need to have the actual source code, (ii) we can modify third party, closed-source classes, (iii) runtime performance is not affected, and (iv) we do not modify the original source files which is appropriate if we need the changes to be applied only for a limited time.

In addition to what we mentioned, we present a tool that facilitates creating Spark code for the user by automatically generating Spark Java code from minimal user-supplied inputs. Spark has emerged as the tool of choice for efficient big data analysis. However, users still have to learn the complicated Spark API in order to write even a simple application. Code generation for parallel programming is a well-studied field that has been recently used to teach undergraduate students parallel programming, and is reported to have reduced programming errors [2].

1.3 Objectives

Our system achieves the following objectives:

- Correctness: Using our method does not change the output of the code.
- Transparency: The modifications are hidden from the user, as in the user is not required to do any source code modifications or implement or learn a new interface or API.
- Performance: Using our method yields performance gains in terms of execution time.

1.4 Development Process

Our workflow can be summarized by the following steps:

- Detect and intercept the user's function call to search for a pattern by running java agent to intercept class files.
- Understand bytecode representations and use ASM to instrument class methods into parallelized version.
- Replace the original function call with our own parallel function (environment-dependent) that splits the input by injecting bytecode into the user's code.
- Return results and control back to the user.
- Measure performance gain.

1.5 Outline

In this work, we present a method that facilitates the parallelization of high performance code under both shared-memory and distributed environments. Sequential legacy code written in Java is seamlessly mapped into a parallel program. Specifically, we have targeted the pattern matching application.

In addition to facilitating the transformation of existing code into a parallel application, we also implement a tool that helps developers write Spark Java code in an attempt to promote efficient and easy to implement parallel programming.

In Chapter 2, we discuss pattern matching in a broader view, and we present our motivation behind selecting it as the application to use. We also discuss Java bytecode and explain how we used ASM to implement bytecode injection. As well, we present our first approach of facilitating

parallelization in a shared-memory environment by using multithreading and we show our results.

In Chapter 3, we present another approach to shared-memory consisting of GPUs. We discuss GPGPU in general, CUDA in particular, the previous work done in this area and how it is different from what we propose, the challenges we faced, and our implementation and results.

In Chapter 4, we introduce distributed systems. We discuss MPI as the prominent communication technology of choice in the industry. We show how we applied our method using MPI and display our results. In addition to that, we introduce Hadoop as an alternative to message passing. We do not use Hadoop as a target environment for our method for reasons mentioned in the chapter; however we do compare its output to our multithreading results.

In Chapter 5, we present a tool that helps developers write Spark code that can run on a distributed environment. We introduce the Spark ecosystem, show our reasoning and workflow, and provide a machine learning plug-in to facilitate generating machine learning algorithms.

Finally, in Chapter 6 we summarize our work and our findings, and we suggest some directions for future work.

Chapter 2: Automatic Parallel Matching using Java Bytecode Injection

Shared memory systems describe a computer architecture in which all processors share a single view of the memory module and access the same logical memory locations irrespective of where the physical memory actually is. Communication (data sharing) in such systems is efficient and fast since all the parallel tasks can share the same resources. In this chapter, we discuss implementing pattern matching in a shared memory environment using Java threads. Our contribution is summarized as follows: detect a user's Java call to search for a pattern, instrument a parallelized multithreaded pattern matching code, and finally return control back to the program.

2.1 Introduction

The importance of big data does not simply lie in the amount of data available. It stems from what can be done with that data, how to transform data into useful information, whether it be to reduce cost, reduce risk, make better calculated decisions, or increase efficiency. Pattern matching is the basis for data mining upon which other more complex algorithms are built. Pattern matching consists of finding one, or several, occurrences of a pattern within a larger dataset. It has been extensively used in several applications from various domains. Some applications include citation matching, plagiarism detection, malware detection, virus scanners, content monitoring filters, intrusion detection, digital forensics tools, genetics….

Finding a way to parallelize pattern matching algorithms will result in more efficient usage. Parallelizing pattern matching can be done in one of two ways: either rewriting the algorithm to

make it inherently parallel or using data parallelism which consists of splitting the input and running the same serial algorithm on all parts. We have opted to do the latter since it is typically what programmers would try to do.

Our implementation deals with searching for a pattern within a text using Java's standard regular expression library (regex) and parallelizing it using multiple threads without compromising the integrity of the library. We have chosen the standard library, as opposed to other implementations, since it is widely and commonly used. Parallelizing it will not change the implementation and hence will preserve the integrity and correctness of the original code.

The rest of the chapter is divided into the following sections. Section two gives a brief overview of pattern matching. Section three presents the motivation behind our work. Section four consists of a literature review. Section five gives a brief description of Java bytecode and the ASM library. Section six discusses the parallel approach in our implementation. Section seven displays our working environment and results. Section eight concludes the chapter.

2.2 Pattern Matching

Pattern matching can be divided into two basic categories: exact pattern matching and regular expression pattern matching.

Exact pattern matching is more commonly used as string matching, since the pattern is well defined and most search applications are text-based. String matching (or string searching) algorithms constitute a crucial subclass of text processing since they are widely used in many applications pertaining to various domains, such as parsers, spam filters, word processors, search engines, digital libraries, natural language processing, and molecular biology [3]. String matching consists of finding the first occurrence of a pattern of length m in a text of length n,

where typically $n >> m$. Both pattern and text are composed of a finite alphabet set denoted by Σ, whose size is denoted by σ. This alphabet set could be for example the modern English alphabet (letters a through z), the binary alphabet (0 and 1) or the DNA alphabet (A, C, G, T), depending on the application. Although nowadays data is stored in many formats, text remains the main form for exchanging information. The main algorithms for exact pattern matching are: brute-force (BF), Knuth-Morris-Pratt (KMP), Karp-Rabin, Boyer-Moore (BM), and Horspool [4].

The BF method, also called naïve method for the simplicity of its implementation, is the only one among these that does not require a pre-processing phase. The way the BF works is by comparing the pattern to every character in the text between *0* and *n-m* and then by shifting one position to the right. The search time complexity is O(*nxm*). BF is the method used in the Java standard String library implementation of the highly used method `indexOf`. The `indexOf` method returns the location of the first occurrence of the pattern. Therefore, in order to find all occurrences of a pattern without inherently changing the underlying implementation, one must repeatedly check for each occurrence using a loop.

The second category of pattern matching is regular expression (regex) pattern matching. The difference between this and the exact approach is that regex searches for occurrences of multiple patterns in a text. Some applications include natural language processing, scanning for virus signatures, accessing information from digital libraries, filtering text, validating data-entry fields, and searching for markers in human genome [3]. Regular expression notations specify a set of strings and can be very expressive. They can be used in nearly infinite ways and include characters, quantifiers, and meta-characters. For example, one can search for words that contain 2 *x*'s, or for fields representing a telephone number where the format is *ddd-ddd-dddd*, *d* being a digit, or for a part of a text not followed by another specific part of text.

We focus our research on regular expression pattern matching. According to the article in [5], there are two approaches to regular expression matching. The first is a backtracking implementation used in several standard languages, such as Perl, Python, Java, PCRE, Ruby, and several others. The second, mainly used in implementations of `awk` and `grep`, uses finite automata.

A finite automaton is another word for a finite state machine. Switching from a state to another depends on the sequence of characters of the string. There is a start state, and a matching state. If the matching state is reached, then a match has been found. If the machine ends in a state other than the matching state, then a match was not found.

There are two types of automata: deterministic (DFA) and non-deterministic (NFA). In any state in DFAs, each possible input character leads to just one new state. In NFAs, there could be multiple choices for the next state. Figure 2-1is an example that shows the difference for the pattern `a(bb)+a`.

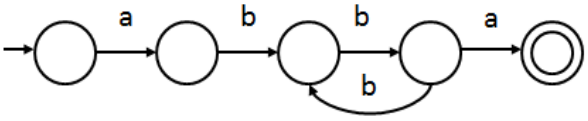
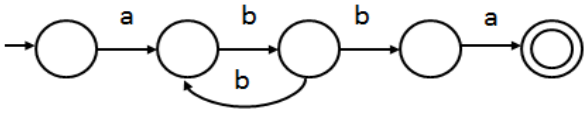
Partial DFA representation	**Partial NFA representation**
	

Figure 2-1: DFA vs NFA example diagram

The circles in the diagram represent the states, and the arrows represent the state transitions based on the characters. The matching state is denoted by a double circle. The difference between DFA and NFA can be shown from the transitions in state four. In the DFA, if an `a` is encountered, we reach the matching state; if a `b` is encountered, we go back to state three. In

contrast, in state three of the NFA, if a `b` is encountered, there are two possible solutions: either go back to state two or proceed to state four. The NFA does not know which decision is the correct one since it cannot peek to see the rest of the string. The machine will guess, and hence is labeled non-deterministic. To simulate guessing, one can allow the machine to guess one option and if that does not work, to try another option by backtracking. This is recursive though and might lead to a slow exponential runtime if there are many possible options. A better way would be to guess both options simultaneously. In this case, the NFA will be in multiple states at the same time, and runtime is linear.

DFAs are more efficient to execute because they can be in only one state at a time. DFAs are also faster than NFAs since there is a state transition for every possible character. This has not been shown in the diagram above. However, this also means that they require more memory to be stored. In NFAs, the number of states is at most equal to the length of the regular expression. An NFA can be converted into an equivalent DFA where each DFA state corresponds to several NFA states.

Java contains a regular expression standard library (java.util.regex). It is mainly composed of two classes: a Pattern class and a Matcher class. The Pattern class compiles the pattern, and the Matcher class contains the methods needed to perform and analyze the search for that pattern[1].

[1] It is also worth mentioning that the String class can also perform regular expression matching. However, it can only match the entire text against the pattern, using the *text.matches(regex)* function.

2.3 Motivation

Most of the times, a standard language implementation is fairly fast, however, there are some regular expressions that would make the algorithm run very slow. Most regular expressions library implementations (Java's regex included) use a backtracking algorithm. Given a set of bad inputs, albeit simple, such algorithms can take exponential time to execute. The article in [5] goes on to show that for a particular regex, and for a 100-character string, trends show that a Thompson NFA implementation takes 200 microseconds, while a Perl implementation will take over 10^{15} years. This is because standard library implementations support backtracking and back references, which makes matching NP hard. To illustrate, we have tested the inputs from [6] on our local machine. We used the Java regular expression library with the pattern "*(a|aa)*b*", and varied the length of the string text. Results are shown in Table 2-1 and Figure 2-2, where *N* is the number of characters of the text.

Table 2-1: Runtimes of pattern matching backtracking example

Text	N	Search time (ms)
aaaaaaaaaaaaaaaaaaaac	21	10
aaaaaaaaaaaaaaaaaaaaac	22	30
aaaaaaaaaaaaaaaaaaaaaac	23	30
aaaaaaaaaaaaaaaaaaaaaaac	24	50
aaaaaaaaaaaaaaaaaaaaaaaac	25	90
aaaaaaaaaaaaaaaaaaaaaaaaac	26	120
aaaaaaaaaaaaaaaaaaaaaaaaaac	27	200

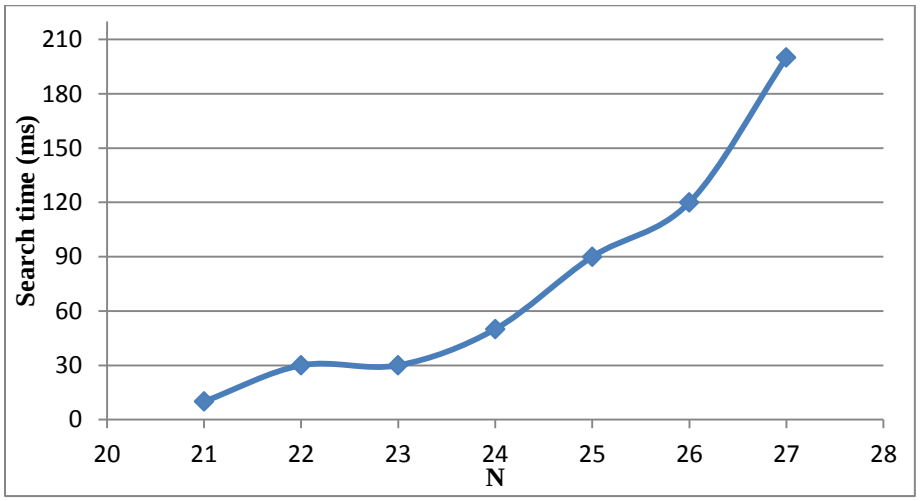


Figure 2-2: Runtime graph of pattern matching backtracking example

For every character increase in *N*, the runtime increases exponentially. This happens because the whole pattern was not found. However, the first part was found, so the algorithm backtracks and tries to match the remaining part. When it does not find it, it goes back one more step to the last matched part and tries to match the whole pattern, and so on. Other more realistic similar cases can result from spam attacks and cause a denial of service.

Another reason for the exponential searching is the use of back-references. A back-reference matches the string matched by an earlier expression surrounded by parenthesis. For example, the regex `(romeo|juliet)\1` matches `romeoromeo` or `julietjuliet` but not `romeojuliet` or `julietromeo`. Back-references provide additional power but at a potentially much higher cost.

We believe the reason the standard tools have not considered implementing the more efficient finite automaton approach is two-fold. First of all, finite state machines cannot handle sub-match extraction that requires recursive backtracking. This means that they cannot

distinguish the individual matches made by an expression surrounded by parenthesis for example. In the Java regex library, this can be done using `matcher.group()`. The other reason is back-references. Back-references, being an NP-complete problem with no current efficient implementation, are not represented in the finite state approach and are not implemented in `grep` for example. However, they are part of the POSIX standard for regular expressions and are available in standard tool implementations.

Since execution time increases exponentially with *N*, then having a smaller search space will immensely decrease runtime. Hence, dividing the string into smaller substrings will improve performance.

2.4 Literature Survey

We survey some of the work that involves parallelizing pattern matching in both shared and distributed environments. The paper in [7] implements parallel string matching with Java multi-threading with multi core processing, and performs a comparative study on KMP, BM and BF string matching algorithms on a gene sequence data set. In [8], the authors focus on detecting malware for unstructured data stored in Hadoop Distributed File System (HDFS) by developing a map-reduce approach to easily scan for viruses in HDFS in real time. They used Clam AV's virus signature database to perform their experiments. Their mapper extracts the file type to obtain the correct signature and produces a key-value pair, where the key is the signature and the values are the lines from the files. The reducers aggregate the lines per key (signature) and do pattern matching on the list of values. They used three different algorithms for string matching: BM, KMP and RabinKarp. In [9], the authors present a citation matching method and scale it up using Hadoop. In [10], the authors present a parallel implementation for string matching using

MPI. They partition the string to search into a number of subtexts and have used the brute force method for searching. In [11], given a text and a pattern, the authors try to find if the pattern occurs in the text using a Hadoop cluster. If it does, they return the starting index. If not, then they find the longest prefix, suffix and substring of the pattern in the document and find the similarity between documents and the longest common subsequence of all documents. The algorithms they used are KMP and cosine similarity. In the blog post [12], the author uses a grep script in a Hadoop cluster to search for a string in the WorldCat.org record. In [13], the author implements, tests, and tries to optimize the three string matching algorithms (Brute-force, QuickSearch, Horspool) using CUDA. The methods applied to the development are: finding ways to parallelize the sequential code, minimize data transfer between host and device, coalesce global memory as much as possible, and avoid branch divergence within a CUDA warp. The paper in [14] implements KMP using a multicore processor and using a GPU. They partition the string into the number of processors that they have; however, they do several extra iterations after that to account for in-between occurrences. This could have been avoided by doing better partitioning. They implement their algorithm using four techniques: serial, multithreaded CPU, multicore CPU using OpenCL, and on a GPU using OpenCL. In [15], the authors propose to offload the processing of digital forensics tools to a GPU and compare the speedups obtained by simple threading schemes appropriate for multicore CPUs. The application they use is file carving. For the multicore machines, a thread was spawned for every carving rule (used the BM technique). On the GPU, each thread is responsible for searching for approximately 160 bytes of the 10 MB block. In [16], the authors implement the BF, KMP, Boyer-Moore-Horspool, and Quick-Search string matching algorithms on a GPU and record the performance. The data set consisted of three reference sequences: the bacterial genomes of Yersinia pestis, Bacillus

anthracis, and a simulated BAC built of the first 200,000 characters of NCBI build 26 of Homo sapiens' chromosome 2. The query pattern was constructed of randomly chosen subsequences from each reference sequence. In [17], the authors propose the design, implementation, and evaluation of a pattern matching library running on the GPU. The library supports both string searching and regular expression matching using CUDA. They chose to implement a GPU-based pattern matching library for inspecting network packets in real time and returning any matches found back to the application. They have parallelized the DFA based matching process by splitting the input data stream into different chunks. Each chunk is scanned independently by a different thread using the same automaton that is stored in device memory.

2.5 Java Bytecode and ASM

When a Java code is compiled, the `javac` compiler generates a machine-independent intermediate format known as the bytecode. The bytecode is mainly composed of the instruction set that describes the program to the Java virtual machine. The JVM processes the bytecode instructions from the class files. Primitive types and objects are expressed with the representations shown in Table 2-2. These representations are later used in constructing the method descriptors such as the examples in Table 2-3 which show the method declarations in a Java source file and their bytecode counterparts.

Table 2-2: Primitive types and class representations

Type	Representation
void	'V'
boolean	'Z'
char	'C'
byte	'B'
short	'S'
int	'I'
float	'F'
long	'J'
double	'D'
Class	L<class>;

Table 2-3: Method descriptors

Method declaration in source file	Method descriptor
void methodName(int i, float f)	(IF)V
int methodName(Object o)	(Ljava/lang/Object;)I
int[] methodName(int i, String s)	(ILjava/lang/String;)[I

Bytecode instructions use a stack to exchange the data. The code snippet in Figure 2-3 shows a basic example of how one would typically implement the pattern matching function in Java, and the code in Figure 2-4 shows how ASM (discussed shortly) uses the bytecode

instructions to store the objects and then when needed, identify and use them based on their locations, as shown in Table 2-4.

```
(23) String pattern = "pattern";
(24) String text = "the text to search ";

(26) Pattern  p = Pattern.compile(pattern);
(27) Matcher  matcher = p.matcher(text);
(28) if (matcher.find())
        ...
```

Figure 2-3: Pattern matching code snippet

Table 2-4: Object locations

Variable	Object
7	pattern
8	text
9	p
10	matcher

```
Label l0 = new Label();
mv.visitLabel(l0);
mv.visitLineNumber(23, l0);
mv.visitLdcInsn("pattern");
mv.visitVarInsn(ASTORE, 7);

Label l1 = new Label();
mv.visitLabel(l1);
mv.visitLineNumber(24, l1);
mv.visitLdcInsn("the text to search ");
mv.visitVarInsn(ASTORE, 8);

Label l2 = new Label();
mv.visitLabel(l2);
mv.visitLineNumber(26, l2);
mv.visitVarInsn(ALOAD, 7);
mv.visitInsn(ICONST_2);
mv.visitMethodInsn(INVOKESTATIC, "java/util/regex/Pattern", "compile", "(Ljava/lang/String;I)Ljava/util/regex/Pattern;", false);
mv.visitVarInsn(ASTORE, 9);

Label l3 = new Label();
mv.visitLabel(l3);
mv.visitLineNumber(27, l3);
mv.visitVarInsn(ALOAD, 9);
mv.visitVarInsn(ALOAD, 8);
mv.visitMethodInsn(INVOKEVIRTUAL, "java/util/regex/Pattern", "matcher", "(Ljava/lang/CharSequence;)Ljava/util/regex/Matcher;", false);
mv.visitVarInsn(ASTORE, 10);
```

Figure 2-4: ASM code of Figure 2-3

ASM [18] is a Java bytecode engineering library. It can be used to analyze and to manipulate bytecode by modifying classes. First, a Java agent is created. The agent contains a pre-main method that implements a class transformer. The class transformer is composed of a class reader and a class writer. The class reader reads the bytecode from the class files in order to analyze the classes or the methods using a class visitor. If a class or a method needs to be modified, the class visitor performs these modifications. Examples of such modifications include printing a method's description, displaying the different object types in a class… These modifications are then stored using the class writer. Bytecode injection consists of manipulating the bytecode by inserting constructs that change what a method does. A method visitor scans the classes in search of the desired method. It is identified based on its description. The description consists of the method return type and the types of its parameter list.

The overall process is shown in Figure 2-5. In our implementation, when we encounter the particular method that searches for a pattern, we use bytecode injection to replace it with our own parallel pattern matching implementation.

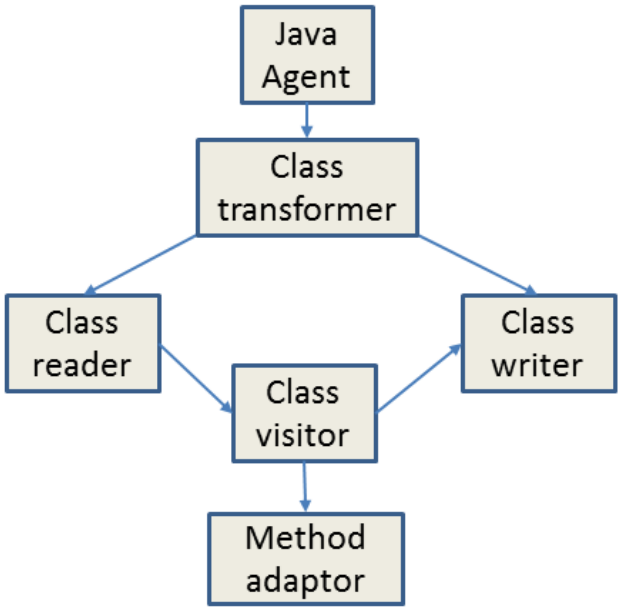


Figure 2-5: ASM process

2.6 Implementation

Because of the way it is implemented, the regex pattern matching `find()` function from the Java standard library can have an exponential search time based on the pattern and the text to be searched. We have devised a method that parallelizes this search without compromising the integrity of the library. Another main contribution is that this method is transparent to a user. Therefore, a coder does not need to put any extra effort using our method. All that is needed is for the user to run his/her code with an extra parameter that invokes our java agent. Our method will automatically detect the function to be parallelized and will instrument the parallel version of it instead of the user's function. Control is then returned to the user's code. All data structures,

objects, and primitives will be preserved. The advantage over other work is that users do not need to learn any new constructs, they do not need to implement any new interface or API, and they do not need to apply any changes to their code. A simple illustration of our process can be found in Figure 2-6. When we encounter the particular method that searches for a pattern, we use bytecode injection to replace it with our own parallel pattern matching implementation.

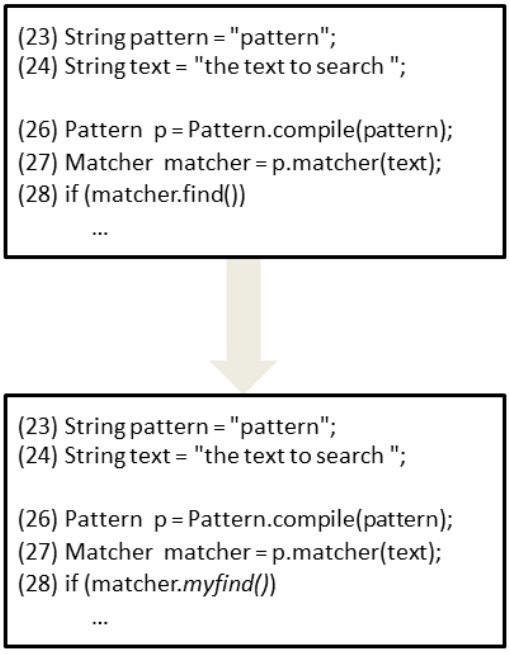


Figure 2-6: Process illustration

Multithreading is the ability of a CPU to execute several threads concurrently within the same program. It aims to increase the utilization of the CPU time using parallelism. The concept is similar to multitasking, but dealing with applications. A single application can be divided into separate operations, where each operation is assigned to a different thread. These threads can run in parallel with their own paths of executions but share resources. This sharing of computer and data resources is the main challenge of multithreading. Special attention needs to be given to threads wanting to read and write the same data simultaneously. However, the benefits of

multithreading are many. Some of them include: more efficient CPU usage, better system reliability, better resource utilization, and improved performance on multiprocessor computers.

Multithreading in Java is essential but not always simple. The Java platform is designed to support concurrent implementation. It supports basic concurrency in the language and the class libraries. Each application has at least one thread, the main thread, which has the ability to create other threads. In this chapter, we use multithreading to implement our parallel approach.

Originally, we had started off by trying to parallelize the Java Collections library. We focused on detecting and implementing the `sort` (library implementation is a merge sort), the `replaceAll`, the `min`, and the `max` functions. We used this as a basis to implement the pattern matching, and to prove that our algorithm works for several cases.

2.6.1 Challenges

The Java regex library proved to be more challenging than the Collections library, and we had to change the way we approached the problem several times in order to bypass both the standards privacy and restrictions.

The regex library contains two main classes: the Pattern class and the Matcher class. The Pattern class compiles the pattern, and the Matcher class contains the method `find` which searches for a pattern within the text.

The first issue we faced was having insufficient variables on the stack. When statement (28) from Figure 2-3 statement is executed, and if we look into the bytecode shown in Figure 2-4, only the matcher is on the stack. So if we wanted to replace the serial version of `find` with the parallel version, what we have on the stack will not be sufficient information. We will also need

to search for the ‘text’, push it on the stack and then pass it to our method. Preloading these variables was another issue we faced since it is not feasible without using some sort of dynamic parsing or creating an abstract syntax tree (AST). The reason for that is that when the compiler generates bytecode, it loads the variables into registers without using their names. Therefore, there is no simple way to know where the variable ‘text’ is stored. We could of course look it up after it is compiled, but that will remove the generality and the automatic detection property that we claim. Another way we thought about was to try and generate the ‘text’ from the matcher, since it is inherently used in the matcher class. However, we found that it is private and that we were unable to access it. Having all these troubles, we decided to implement the whole regex library with all its classes[2]. This would allow us to, first change the private property of ‘text’ to public, and second, to directly implement our parallel functions in the new library. Again, we hit a wall. We reached a point where we had to import Sun’s security libraries and we had absolutely no access to that. Therefore, we decided to write our own wrapper method that runs the `find` method but takes both the text and the pattern as parameters.

2.6.2 Implementation

In what follows we discuss how our implementation achieves load balancing, correctness, and synchronization.

The original user code contains a single thread pattern matching search function that processes the entire text[3]. In our multithreaded implementation, we use a static mapping

[2] The classes are: ASCII, Matcher, MatchResult, Pattern, PatternSyntaxException, and UnicodeProp

[3] Instead of using the pattern matching method ‘find’ from the standard library (java.util.regex), we had to write our own based on running the ‘find’ method within a loop. This is because ‘find’ will only return the first occurrence of the pattern in the text and exit. If we need to locate all the occurrences, we will have to iterate over the whole text. The reason why such a function is not found in the standard library is that the library leaves it up to the users to determine what to do with the data they obtain, i.e. the locations of the pattern. Users can choose to store these integers in the data structure of their choice. We have chosen to store

approach that achieves load balancing. We divide this input text into `t` semi-equal subtexts, where `t` is the number of threads. Each thread receives the same amount of data proportional to the number of available runtime threads before execution starts.

Correctness is achieved in the following two ways: (i) After we distribute the data among the threads, each thread then implements the same function as the original one on the reduced substring. Hence, we did not change the algorithm's implementation and can be sure of its correctness. (ii) However, we also need to account for boundary overlap otherwise strings that span over two partitions would be neglected and our output would be incorrect. Figure 2-7 shows a case where this might happen. Let us assume that the pattern to search for is 'went', which is composed of `m = 4` characters. In this very simple example, the searched text is composed of `n = 20` characters, and we are assuming that the system can run `t = 2` threads. If we partition the text equally, then each subtext will be composed of `n/t = 10` characters. The split will be where the arrow points to in Figure 2-7. Now when each thread performs the pattern matching on its subtext, the pattern will not be found, producing an incorrect output.

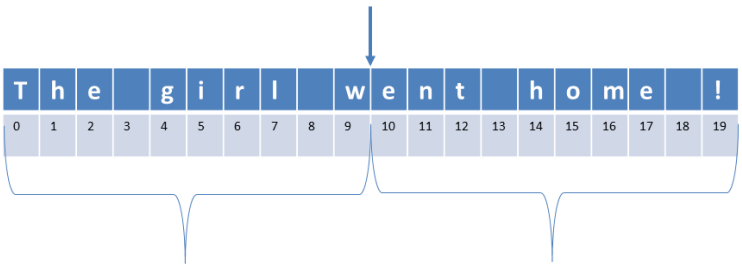


Figure 2-7: Partitions with no boundary overlap

the location in a Hashmap that has the pattern as the key. This is to be consistent with Hadoop's output format discussed in chapter 4.

Therefore, we need a more aware implementation that takes this issue into consideration. Figure 2-8 shows how this can be resolved. The modified partition takes the length of the pattern into consideration where each subtext will now be composed of `(n/t) + (m - 1)` characters. Therefore, the partition from the previous example will contain now 13 characters and our algorithm will correctly identify the pattern. This division excludes the last partition since it is unnecessary and will merely consist of the remaining characters. It is worth mentioning that this approach does not violate the load balancing property since typically pattern lengths are short, and are considered negligible especially when compared to large bodies of texts.

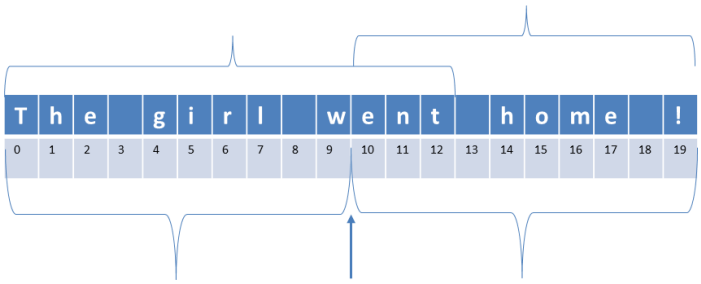


Figure 2-8: Partitions with boundary overlap

Each thread performs pattern matching independently on its own subtext, eliminating the need to access any shared variables. After all the threads complete their jobs, the results are synchronized before being aggregated and returned to the user application.

2.6.3 Working Environment and Benchmark

Implementation has been done using Java (jdk 8) on Eclipse IDE (Luna - release 4.4.0) and ASM v5. Single-threaded and multi-threaded testing was performed on a virtual machine with the specifications [19] shown in Table 2-5.

Table 2-5: VM specifications

Model	Intel(R) Xeon(R) CPU E5-4650 0 @ 2.70GHz
Architecture	x86_64
Number of CPU cores	8 *(2 logical cores per physical)*
Number of threads	16
Cache size	20480 KB

As a benchmark, a text file of 142 MB was used. The file consists of concatenating the 11th edition of the Encyclopedia Britannica downloaded from the project Gutenberg website [20].

2.7 Experimental Results and Interpretations

For the Collections library, testing was done on our local machine. We randomly generated a list of various sizes (ranging from 500,000 to 20,000,000 items). We implemented our multithreaded version using two and four threads. The `max` and the `min` functions attained lower performance over the serial implementation when run on smaller list sizes (<10,000,000 items), however achieved a performance gain of up to 32% when running on 20,000,000 items. The `sort` and the `replaceAll` methods showed an improvement when parallelized over all list sizes, with the highest improvement reaching 38%.

We now present the results for the pattern matching application. Table 2-6 shows the results for the average runtimes. The columns indicate the number of threads used while the first row is the average runtimes in milliseconds and the second row is the percent improvement over the single thread implementation. The first column corresponds to the single thread implementation. In this implementation, the whole input file is read into a string before the search is performed.

The other columns labeled as (*t = number*) correspond to the multithreaded implementation. We have plotted the results in Figure 2-9. Percent improvement is plotted in Figure 2-10 and is calculated as such:

*% improvement = ((sequential – multithreaded)/sequential)*100*

Table 2-6: Average runtimes and percent improvement

	t = 1	**t = 2**	**t = 4**	**t = 8**	**t = 16**	**t = 32**	**t = 64**
Average runtime (ms)	73,199	35,366	15,539	9,506	6,501	4,840	4,142
Percent improvement		51.68	78.77	87.01	91.11	93.38	94.34

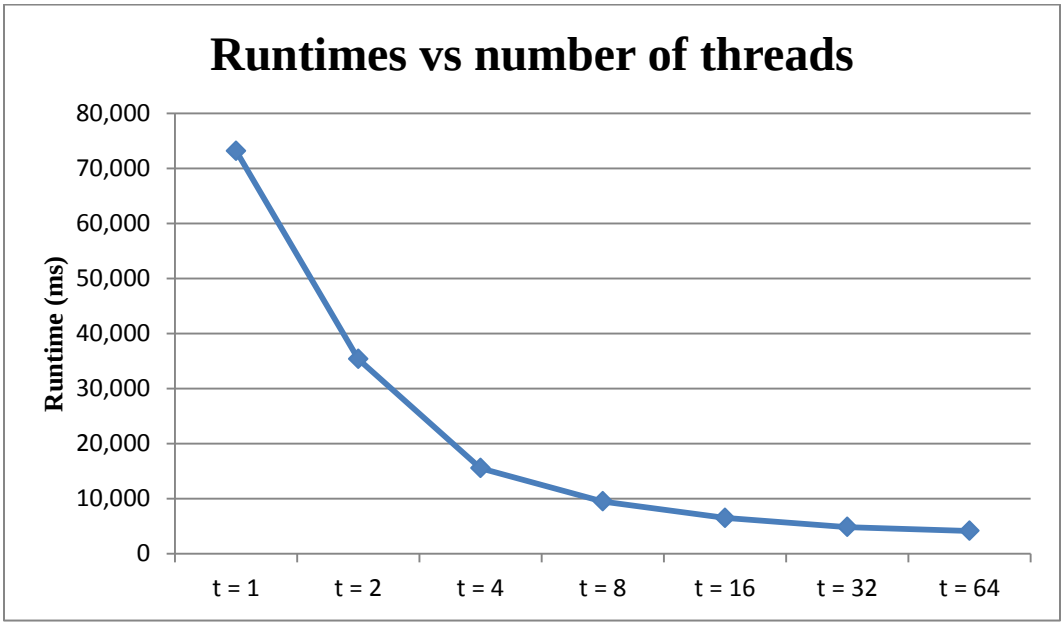


Figure 2-9: Runtimes vs number of threads

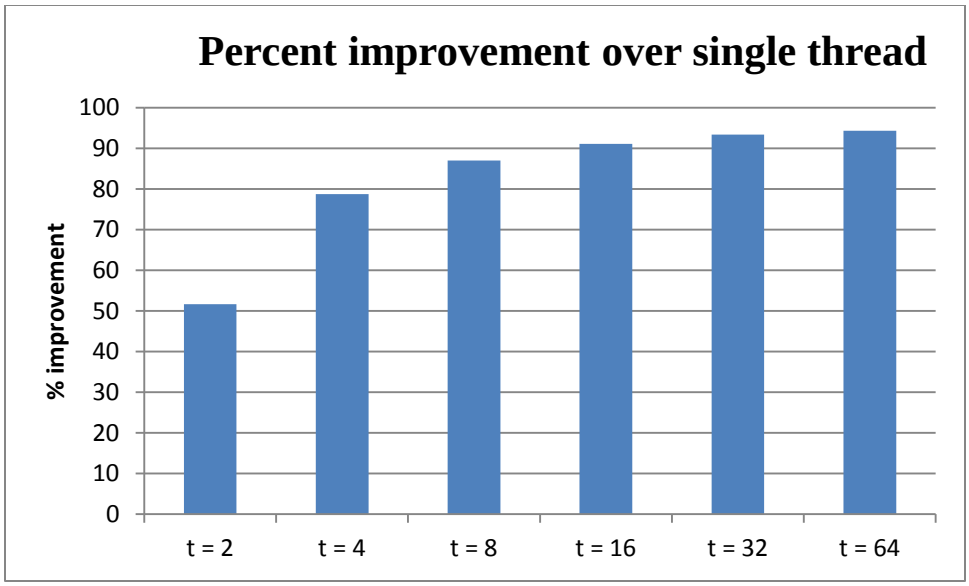


Figure 2-10: Percent improvement over single thread

Since the substring lengths for the multithreaded version are shorter than the original string, the multithreaded approach yields much better results than the single thread approach. We can observe that this gives an almost logarithmic improvement. When we double the number of threads, runtime is observed to be cut in about half. This is until we reach t = 16 threads, which is the number of threads supported by the test environment. Beyond 16 threads, the improvement is negligible.

2.8 Conclusion

In this chapter, we have presented a method that automatically detects a user's function call to search for a pattern using Java's standard regular expression library and seamlessly replaces it with our own parallel implementation. We developed a multithreaded implementation of the pattern matching function and have used bytecode injection to run our optimized implementation instead of the original function. Our experiments show a significant reduction in execution time, and we believe better speedup can be achieved with more powerful machines.

Chapter 3: Parallel Pattern Matching on GPUs

In this chapter, we discuss implementing pattern matching on a GPU. Even though pattern matching might not be considered a purely computational problem in the sense that the actual matching time might not overcome the overhead of I/O for very simple patterns, we believe that it can benefit from parallelization when run on a big dataset and a complex pattern. Therefore, we have attempted to implement it on a GPU. Our contribution is summarized as follows: detect a user's Java call to search for a pattern, instrument the pattern matching code to run on a GPU, and finally return control back to the program.

3.1 Introduction

General purpose computing on graphics processing units, or more commonly referred to as GPGPU, is a high performance computing approach that uses graphics processing units (GPUs) to perform extensive data operations in parallel, yielding better throughput. Programming GPUs is considered quite a challenging task to get right, since first of all it requires a low-level language (CUDA or OpenCL) that can interface with the hardware, and second of all, since it is very different from parallel programming on the CPU. For example, in Java, one might use threads and concurrency (such as thread pool executors or fork-join pools) and then split and distribute the work among the threads and/or processors. However with GPUs, one has other extra low-level issues to consider such as register mapping, shared memory, memory coalescing, and occupancy. So even though GPGPU programming can make computations faster, it is not always the case. If the code is not properly written taking into consideration the issues mentioned, and if the application is not suited to be run on a GPU, performance might degrade. Example applications that usually benefit from a GPU implementation include 3D rendering,

image and media applications, signal processing, and other data-parallel tasks that operate on large datasets in the fields of physics, computational finance or biology, to name a few.

Currently, there are two main programming models for GPU acceleration: OpenCL and CUDA. OpenCL is an open standard, cross-platform parallel programming model for GPGPU computing by the Khronos Group [21]. CUDA on the other hand was developed by NVIDIA and contains a more complete set of HPC libraries. We have chosen CUDA for two reasons. First, we have more experience writing it, and second, since it is the model of choice used in the pattern matching library we have used.

The rest of the document is organized into the following sections. Section two is an introduction to GPUs and to the CUDA programming model. Section three surveys the literature with respect to pattern matching implementation performed on a GPU. Section four discusses the available solutions to run Java code on a GPU. Section five describes our approach. Section six details our implementation. Section seven shows our experimental results, and section eight concludes the chapter.

3.2 Introduction to GPUs

GPGPU computing has become so popular and prevalent due to GPUs evolving so rapidly and efficiently in order to satisfy the increasing market demand for real-time, high-definition 3D graphics. Based on NVIDIA's definition, GPUs are "highly parallel, multi-threaded, many-core processors with tremendous computational horsepower and very high memory bandwidth. [21]".

3.2.1 Difference between CPUs and GPUs

GPUs are dedicated to parallel compute-intensive applications. This is accomplished by having more transistors to process data than to deal with caching and control flow. Hence, more ALUs are present. GPUs are considered single-instruction multiple-thread (SIMT) processors. The same operation can be executed on each data element. Therefore, there is no need for complex flow control. In addition, the memory access latency can be hidden by the computation. Table 3-1, based on NVIDIA's website, summarizes the differences between a GPU and a CPU.

Table 3-1: Basic differences between a CPU and a GPUS

CPUs	GPUs
Optimized for low-latency access to cached data sets	Optimized for data-parallel, throughput computation
Control logic for out-of-order, pipelining, and speculative execution	Architecture tolerant of memory latency: memory latency is hidden by very fast context switching
	More transistors dedicated to computation

In spite of the apparent advantages of GPUs, they do come with certain disadvantages. First, and in order to obtain speedup, algorithms need to be rewritten taking into account the GPU architecture. The challenge is to develop applications that scale parallelism to the many available cores. Not all algorithms are suitable to be run on a GPU. We will examine this further in the next section. Second, adding GPU hardware might increase power consumption, heat production, and cost.

3.2.2 CUDA Programming Model

NVIDIA introduced CUDA in 2006. CUDA is a low-level general purpose parallel computing programming model built as an extension of the C language. Developers use a high-level programming language, such as C or C++ to interact with the CUDA API. Developers run their sequential code using C/C++ on the CPU, and call a CUDA kernel to run their parallel computation-extensive code on the GPU. After running, the results are returned to the CPU. This is called heterogeneous programming.

Heterogeneous programming involves execution on both the CPU and the GPU. The CPU and its memory are referred to as the host, whereas the GPU and its memory are referred to as the device. Typically, the following summarizes a sequence of operations:

- Allocate host input memory
- Initialize host input data
- Allocate device input memory
- Copy host input data to device input
- Allocate host output memory
- Allocate device output memory
- Execute kernel
- Copy device output data to host output
- Free memory on device

3.2.3 CUDA Architecture

CUDA abstracts three elements: streaming multiprocessors (SMs), barrier synchronization, and memory hierarchy.

3.2.3.1 Streaming Multiprocessors

CUDA architecture is built up of a number of multithreaded SMs which perform the actual computations. Each SM has cores, control units, schedulers, registers, execution pipeline and caches. The more multiprocessors a GPU has, the less the execution time will be. Each SM has an underlying logical model composed of grids of blocks, blocks of threads, and threads, as shown in Figure 3-1, taken from the NVIDIA website. When a kernel is invoked, the calling function has to specify the grid dimensions: the number of blocks in a grid, and the number of threads in a block. Based on available execution capacity, these blocks are distributed among the SMs. Threads of a thread block execute concurrently on one SM. Multiple blocks can execute concurrently on one SM. The dimension constitutes the parallelism, as in each kernel operation is executed *N* times in parallel, where

$$N = \textit{grid size x block size x number of threads per block}$$

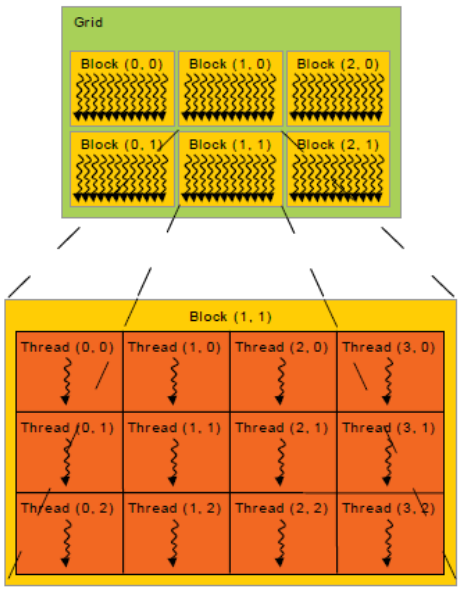


Figure 3-1: Grid of thread blocks, taken from NVIDIA website

3.2.3.2 Barrier Synchronization

Threads in a block execute concurrently in batches of 32, known as threads warps. Threads belonging to the same block can synchronize their execution to coordinate memory accesses. Developers can specify synchronization points in the code by explicitly calling a function to synchronize threads. This acts as a barrier where all threads in that block must wait before proceeding with execution.

Unlike threads in a block, thread blocks are expected to execute independently. They can be executed in any order, in parallel, or in series, and can be scheduled across any number of cores.

3.2.3.3 Memory Hierarchy

Following the CUDA abstraction, threads each have their own private per-thread memories, threads in a block can also access shared memory, and threads in grids can access global memory. Access to shared memory is typically much faster than global memory. That is why good CUDA programmers tend to place data in shared memory rather than in global memory. However, it is also much smaller. There are also two more memory modules: constant memory and texture memory. These are read-only and accessible by all threads. Each memory space is optimized for different usage.

3.2.4 Applications Suited for GPUs

Not all applications are suited for a GPU implementation. In other words, speedup while running on a GPU is not guaranteed. Certain types of problems will actually exhibit a reduction in performance if implemented on a GPU. We have mentioned a few of these applications in the introduction. Here we mention the properties that algorithms suited for a GPU implementation

share: data parallelism and throughput intensive. For problems that satisfy these properties, a GPU implementation can offer significantly faster computations.

3.2.4.1 Data Parallelism

There is a distinction between problems being task parallel or data parallel. Data parallelism involves several threads performing the same operation simultaneously, but on different parts of the data. GPUs are good for data parallel problems because they have many cores that can do the same operations on different parts of the input data.

3.2.4.2 Throughput Intensity

Problems can be either memory bound or compute bound. Memory bound refers to problems where I/O operations are predominant. These problems are not suited for GPU implementation because it is expensive, in terms of memory cycles, to transfer data back and forth from CPU to GPU. On the other hand, problems that are compute bound perform several instructions on a data element, hence offsetting the efforts of reading and writing to memory. These problems have a high order of complexity and will benefit from a GPU implementation.

3.3 Literature Survey

To the date of writing this dissertation, there has not been yet any official library form NVIDIA or the OpenCL community to perform pattern matching on GPUs. We believe the reason for that is that such a problem, if not used with massively data parallel computation, can be considered memory bound and will perhaps not map well to parallelism. However, several attempts have been made to efficiently perform pattern matching on a GPU. We differentiate

between three categories: commercial unavailable solutions, academic papers, and open source projects.

Commercially, HP [23] and IBM [24] have published presentations and videos on building finite automata and matching regexes respectively. However, the details were not exposed. The HP presentation simply mentions that they have achieved fast regex parsing; whereas the IBM video just reports on throughput, latency, and the effect of the pattern sizes. Their findings show that the GPU provides significant throughput boost compared to the CPU, the current GPU regex implementation they have is not suitable for latency sensitive applications, and that the problem is strongly workload dependent on both the data contents and the patterns.

The most prevalent work regarding pattern matching on GPUs has been purely academic. Our findings show that research in this area has been geared mostly towards network intrusion detection systems due to the nature of the problem wherein network packets are scanned against several known malicious attack patterns. This constitutes a problem of matching several patterns against several files of data, and hence tends well to a GPU implementation. The work in [25] provides a detailed evaluation study of GPU designs on practical datasets and explores advantages and limitations of several techniques. It also suggests some schemes to avoid the limitations discussed. In [26], the authors propose a hierarchical GPU based approach to accelerate regular expression matching and to resolve the problem of state explosion that can incur from a DFA implementation. They target network intrusion detection systems. Keeping their application in mind, they have reduced regular expressions into two categories: the first involves the '.*' wildcard and the second involves patterns with constraint repetitions. They have extended their previous PFAC (Parallel Failureless Aho-Corasick) algorithm to implement a hierarchical machine, where the master detects the first category and if found, the slave detects

the second from the intermediate result of the master. The same authors have expanded on their work in [27]. They introduced throughput efficiency techniques that include reducing global memory transactions, reducing the latency of the transition table lookup, eliminating output table accesses, avoiding bank conflicts, and enhancing communication between the CPU and the GPU. In [28], the authors present a ground up parallel NFA-based regular expression engine running on a GPU called iNFAnt. The difference between their work and previous implementations is that they were among the first to implement an NFA based state automata instead of a DFA one. Their work is general and does not only handle specific cases. Since they deal with NFAs that do not suffer from state explosion as opposed to DFAs, they were not constrained by the size or complexity of the rule sets. The authors in [29] propose a solution that tries to overcome the limitations of [28] by implementing a conceptual hybrid of a NFA-DFA implementation. However, their implementation could only work for smaller datasets. In [30], the authors propose the design, implementation, and evaluation of a pattern matching library running on the GPU. The library supports both string searching and regular expression matching using CUDA. They chose to implement a GPU-based pattern matching library for inspecting network packets in real time and returning any matches found back to the application. They have parallelized the DFA based matching process by splitting the input data stream into different chunks. Each chunk is scanned independently by a different thread using the same automaton that is stored in device memory. They use synthetic network traces and synthetic patterns, in order to control the impact of the network packet sizes and the size of the patterns to the overall performance. The content of the packets, as well as the patterns, are completely random, following a uniform distribution from the ASCII alphabet. Other publications by the same authors [31] detail their implementations of creating Gnort, which is the mapping of Snort to the GPU. They do so by

processing the DFA on the GPU, while implementing the NFA on the CPU. In [32], the authors implemented an XFA data structure on a GPU, and compared the performance of the XFA to a DFA. Their design is suited to a specific set of problems that can be broken down into non-overlapping sub-patterns separated by the '.*' wildcard. In [33], the authors present an implementation of the Aho-Corasick algorithm in CUDA, and discuss the various trade-offs and design decisions.

With respect to the open-source projects available that perform pattern matching on GPUs, we were able to only find one [34]. Cuda-grep is a CUDA implementation of the NFA automata on a GPU. We have decided to use it in our work. We will therefore postpone discussing it to a later section in which we provide a detailed analysis of the implementation and limitations.

3.4 Writing Java for GPUs

GPUs are programmed using usually either CUDA or OpenCL. These two languages support C and C++. However, Java programming is not inherently possible as a directly available language to run on GPUs. In the following, we discuss why one might need to program GPUs in Java and how it can be done.

3.4.1 Why Java

There are several reasons why one might want to run a Java application on a GPU. Java developers would rather code in Java than in C/C++. They do not want to worry about pointers, destructors, memory models, or allocating and freeing memory [35]. In order to use a GPU, they will have to deal with the previously mentioned hassles. Java offers a level of abstracting the low level hardware details, and typically developers used to that mode of programming would like to

maintain that level. Another reason why developers might want to write Java code running on a GPU is to take advantage of the performance benefits of GPUs in their already existing Java application. Some areas in a project that require huge computational power could be mapped to a GPU to obtain better performance. That can be done without having to rewrite the whole application in C or C++, just the relevant code segment.

3.4.2 Running Java code on a GPU

Basically, there exist two main ways of running code written in Java on a GPU.

The first method consists of using one of the available open-source projects that transform the Java code into low level code that runs on the GPU. This can be done in one of two ways: either use a library that performs the bindings for the users or use a library that hides the low level code from the users. This includes code to initialize the device, create data buffers, allocate memory on device, send the data to the device, execute the code, and transfer the results back to the host. The advantage of this is that we will not need to write any low level code ourselves. However, we will need to learn and use a new API. In addition to that, we will be restricted with the chosen solution's hardware and programming requirements. In addition to that, and based on our survey, such solutions are not flexible in a sense that in some cases, only primitive datatypes are supported.

The second and basic method consists of implementing native methods in the Java code, and running the GPU kernel code from within these C/C++ native methods. The advantage of doing so is that it is more flexible with respect to the datatypes that can be used. And since all implementation is 'home-made', we are not limited to an operating system, a programming model, a GPU device or a processor type. However, this requires us to write more complex code

consisting of JNI (Java Native Interface) functions and declarations, manually generate the object files, and correctly link everything together.

In what follows we survey the current available solutions, test some of them, and justify our method of choice.

3.4.3 Available Solutions

Several attempts have been made in order to allow programmers to write Java code that runs on GPUs. [21] and [36] have provided a summary of the most popular and widely used libraries. [21] also evaluates two representative Java GPGPU projects and draws some comparisons based on performance and programming effort. We have added some newer projects to the list and have summarized the most popular ones. They can be classified into three main categories based on their implementations:

3.4.3.1 Bytecode Translation and CUDA/OpenCL Code Generation

- Aparapi [37]: Aparapi is an open-source library by AMD. It translates Java code into OpenCL, without requiring users to know or write any OpenCL code. Users override a 'kernel' method which is to be implemented on the GPU, and the bytecode of this method is loaded at runtime and translated into OpenCL. If compilation to OpenCL is not possible, the code will still be executed in parallel using a thread pool.
- Rootbeer [38]: Rootbeer is an open source library that converts Java code embedded in a 'gpuMethod' function into CUDA code. Users do not need to know CUDA.
- java-gpu [39]: Java-gpu is a library that translates annotated Java code into CUDA.

3.4.3.2 Java CUDA/OpenCL Binding Libraries

- JavaCL [40]: JavaCL is an object-oriented OpenCL library that auto generates low level Java bindings for OpenCL.
- Jcuda [41]: Jcuda provides low level one-to-one bindings that map jcuda API function calls to CUDA API.
- Jocl [42]: Jocl provides low level one-to-one bindings that map jocl API function calls to OpenCL API.

3.4.3.3 Language Extensions

- Ateji – PX [43]: allows parallel constructs, such as OpenMP-style parallel for loops to be executed on the GPU using OpenCL. This project is no longer maintained.
- JCUDA [44]: JCUDA is a library that translates special Java code into CUDA C code.
- JaMP [45]: JaMP is a Java extension for OpenMP constructs with a CUDA backend.

There have been other projects such as CUDA4j and Project Sumatra. CUDA4j [46] is an IBM product that is heavily endorsed by NVIDIA. It aims to provide Java API to run CUDA code on NVIDIA GPUs. Impressive results have been shown, but currently not all features have been developed. Project Sumatra [47] started out as an OpenJDK initiative, but it seems that the project has been put on hold and development is currently not underway. No other information could be found.

Another categorization similar to that in [21] can be based on whether these approaches constitute a user-friendly API or whether the user needs to know CUDA language. They can be summarized in Table 3-2.

Table 3-2: Categorization of Java GPGPU solutions

	Java bindings	**User-friendly**
CUDA	JCUDA jcuda	Java-gpu Rootbeer CUDA4j
OpenCL	jocl	Aparapi jacc

3.4.4 Evaluation

After surveying and testing some of these solutions, and while we found them relatively easy to use, we have opted to implement the basic approach that is the underlying approach beneath most of the previously mentioned techniques. It consists of a JNI implementation, coupled with dynamic linked libraries that contain both C/C++ and CUDA object files. We have chosen this approach since it provides us with more flexibility specifying and using non-primitive object types, does not restrict us to certain limitations such as those mentioned in Table 3-3, and allows us to integrate our bytecode injection library.

Table 3-3: Limitations with current solutions

Most popular solutions	Limitations
Aparapi	- Works with arrays of primitive datatypes only
Rootbeer	- Requires additional build steps
java-gpu	- Code needs to be annotated - Only works on for-loops - Support has been discontinued
Jcuda	- Kernel function still needs to be written in C - Kernel parameters must be arrays of primitive datatypes - Becomes complex to program if application is not straight forward
Jocl	- Kernel function still needs to be written in C
JCUDA	- Library not publicly available - Outdated, does not support all features provided by newer devices
Jacc	- Complex to write and understand API - Not available as an open source project
Cuda4j	- Uses Power processor and IBM Java - Currently available in Java 7.1 and Java 8 on POWER 8 Little Endian, running Ubuntu 14.10 and CUDA 5.5-54 - Supported hardware is POWER 8 model 824L with one or two Tesla K40m GPUs - Not fully developed yet

3.5 Mapping Approach

We describe our workflow in this section. More specifically, we describe the libraries and interfaces we used to parallelize the Java code of interest on the GPU, and how the linking between the languages was performed. A diagram of our workflow can be found in Figure 3-2.

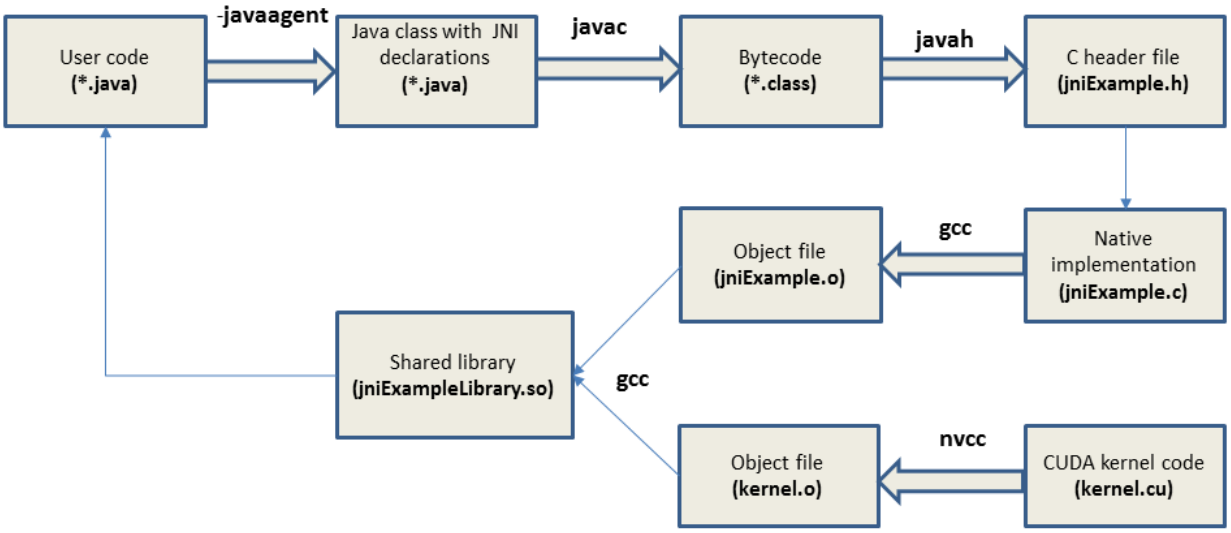


Figure 3-2: Workflow diagram

There are some scenarios where writing Java code alone will not meet the needs of an application. Developers use the JNI to implement Java native methods that take care of these situations. According to [49], some examples include: the Java standard library does not support required platform-dependent features, a library written in a different language is already available and needs to be used, and a small segment of performance critical code needs to be implemented in a lower-level language. The last case is exactly what we are dealing with. We want to implement the part of our code that does pattern matching in CUDA, a low level language. Therefore, in order to run CUDA kernels from Java, we need to use the Java Native Interface library. JNI is a standard programming framework that allows Java code running in a

JVM to call and be called by native applications (written in a different language such as C, C++, and assembly) that are both operating system and hardware specific.

Our workflow diagram starts off by detecting the call to perform pattern matching in the user code using the ASM library. Once the function is detected, bytecode injection is used to replace it with our own implementation. Our file which is still written in Java contains the JNI calls that implement the native functions that in turn call the CUDA kernel. We can divide the work into three parts: modifications to the Java code, creation of the native code which we have written in C, and creation of the CUDA code.

3.5.1 Modifications to the Java code

In order to call a native method from Java, that method has to be declared as a *native* method. This is done by preceding the method signature by the *native* keyword. Now, we are able to call this method like any other function. Another thing we'll have to add is the shared library that contains both the C and the CUDA objects. We do this by calling the `System.loadLibrary` method and passing the name of the library (the `.dll` if in Windows or the `.so` if in Unix). Loading the library maps the implementation of the native method to its definition. Now we are ready to compile the Java class.

Once we have the class compiled, we need to create the header file that will be used in the shared library. To do so, we will make use of the '`javah`' tool. It generates a header file that provides the signature for the native methods defined in that class. The name of a native language function consists of the prefix Java, the package name, the class name, and the name of the Java native method, all separated by an underscore. The method parameters include the following:

- a JNI environment pointer through which the native code accesses parameters and objects passed to it from Java
- a JNI class object which is similar to *this* in Java
- the method parameters and return types. However, these types will no longer be Java defined types. For example, an `int` type will now be mapped to a `jint` type, and a `float []` will be of type `jfloatArray`. This native method signature will be used to write the C implementation.

3.5.2 Writing the C code

This is the part where we implement our native methods. We have chosen to code in C rather than in C++ since we found that it will be easier to integrate with CUDA, but the concept is very similar for either. In addition to that, it proved easier in the linking phase. The first thing we must do is include the header file we had previously generated. Then, we implement the methods in the header file and make sure to preserve the signature descriptions (including the JNI environment pointer and the JNI class object). Inside this method we will need to perform appropriate conversions and assign pointers to variables depending on the application. We will also include a call to a function in the CUDA file that implements a kernel that does pattern matching.

With respect to pattern matching, the Java function takes two parameters: the input text file to search within, and the regular expression. These two variables are of type `String`, which in JNI are translated to a `jstring` type. As mentioned previously, we have used the Cuda-grep library. If we ignore the input flags required to run this library, the entry point is a C++ file that has as parameters the filename and the pattern, both defined as `char *` types. In order to map

the types correctly, we had to first convert the `jstring` type using the `GetStringUTFChars` method, which returns a `const char*` type. After that, we cast the `const char*` to `char*`. In addition to that, we changed parts of the Cuda-grep C++ code to be able to integrate it with our system, and to be able to return the total number of occurrences of the pattern, as opposed to just the lines in which the pattern occurs.

3.5.3 Writing the CUDA code

This is the `.cu` file that contains the allocation of inputs in device memory, the allocation of the output in both the host and the device memories, transferring the input from host to device, performing the computation, copying back the result from device to host, and freeing the memories. The computation step is done by invoking a kernel. This is done by calling the kernel method, passing the parameters, and specifying the number of threads and the number of thread blocks to perform the computations. The kernel definition is identified by the `__global__` prefix. The kernel contains the code that does the pattern matching.

We have used the .cu files provided by the Cuda-grep library, slightly adjusting them for our needs.

3.5.4 Putting It All Together

Once we have the C code and the CUDA code, we will need to create the shared library that we load in the Java program. To do so, we will first compile the C code into an object file using the ‘`gcc`’compiler, and the CUDA code into an object file using the ‘`nvcc`’ compiler. After that, we will link both files using ‘`gcc`’ into a shared library, making sure to include the necessary flags and directories. The last step is specifying the location of this shared library in

the Java application. We can now run the Java program and the native method will execute on the GPU and return the control to the Java application. Moreover, when we run the Java code with the `-javaagent` option, we will be able to automatically detect the pattern matching function, and replace it with its GPU counterpart.

3.6 Performing Pattern Matching on the GPU

As previously mentioned in the literature survey, CUDA-grep [34] was the only open-source implementation of regular expression pattern matching on the GPU that we found. It is based on Thompson's NFA implementation [48]. In what follows, we describe the approach the developers used, the issues they had to consider and the limitations they faced.

3.6.1 The Cuda-grep Library

The CUDA-grep library was written by two students from Carnegie Mellon. Their implementation yields the same results as if running the following from UNIX: `egrep -x <regex> <filename>`. In fact, to validate their results, they compare the outputs of their test bench to the output of `egrep`. If similar, the test passes, otherwise it fails.

`egrep` is similar to `grep -E`, where `E` denotes the Extended Regular Expression (ERE) as opposed to the basic set. The '`-x`' flag instructs grep to select only those matches that exactly match the whole line. Currently, the library does not support all the regexes of `egrep`. It supports the wildcards mentioned in Table 3-4, and based on [50].

Table 3-4: Supported wildcards in CUDA-grep

Supported wildcards	Significance
.	Any character except a new line
+	One or more occurrences
?	Zero or one time
*	Zero or more occurrences
|	Alternate
(…)	Group
[…]	Range, set of characters
\	Escape character preceding .+*?|

The library supports matching multiple patterns to a list of files. In fact, the developers claim that this scenario was the most suited for optimized results versus `egrep`, as compared to searching for one pattern in one file.

Their algorithm can be summarized in Table 3-5, based on the developers' documentation.

Table 3-5: Cuda-grep high level algorithm, taken from developers' documentation

```
Load dataset and regular expressions
Compute NFA from regular expression
for all regular expressions
     for all lines in dataset
          use NFA to pattern match each line
```

The algorithm starts off by loading the dataset and the regular expressions into memory. The files are read, and a table containing the indices of new lines is created. In this way, the whole file can be transferred to the GPU at once, without having to perform multiple transfers back and forth between CPU and GPU and thus optimizing performance. Similarly for the regular expressions, a table is created with indices pointing to each regular expression. In addition to that, the regexes are built and parsed in order to constitute the NFA later on.

The actual lines, patterns, and both tables are copied to the device. Then the matching kernel is called. The kernel specifies a fixed number of threads: 512 thread blocks, each containing 160 threads (5 warps per thread block). The developers have found that this configuration gave them optimal results. Also, they have set the cache configuration using `cudaFuncSetCacheConfig` to `cudaFuncCachePreferShared`. This instructs the program on how to dynamically split the GPU on-chip memory between the L1 cache and the shared memory on a per-kernel basis, since both memory modules use the same statically configured on-chip memory.

The matching process works as follows: each thread block constructs an NFA from the pattern. This is accomplished by checking if the thread id is 0. If so, this is a new block, since thread ids are particular to their corresponding blocks. Constructing an NFA constitutes of first converting the prefix notation of the built regexes to a postfix notation in order to form the NFA states. The threads are then synced. If the thread id is not 0, which means that we have already created an NFA for that thread block, the thread uses the NFA to match a different line from the input file. If a match is found in a line, the index of that line is set in a new table. This result table is then transferred back to the host.

To be consistent with the `egrep` output, the algorithm displays the lines in which a match occurred. We have added to that the number of occurrences.

3.7 Experimental Results and Interpretations

3.7.1 Different Regular Expression Syntax

While most tools share common definitions for wildcards, different tools have slightly different syntax for regular expressions. Implementations can also have extensions that are not supported by other tools. The result is that a regular expression that works for Perl will need to be modified before it works for Java for example. The main difference between tools is whether or not operators such as `*+?.|()[]{}` require a backslash. A backslash is an escape character in literal Java strings and in regular expressions. Therefore, the regular expression `\\` actually matches a `\`. To represent this as a Java string, it will become four backslashes `\\\\` in order to match just one. This is different from other tools. Another example is matching a word character. In `grep` for example, the regex `\w` is used. In Java, this becomes `\\w`. Another difference is the extensions that are supported beyond the basic ones.

Standard tools such as Java support unanchored matches. A substring of the input that matches the regular expression is commonly returned, instead of the whole string. For example, if the pattern is `Romeo|Juliet` and the text is `Juliet is the sun`, then the match returned is `Juliet`. On the other hand, in `grep` for example and using the same pattern and text, the whole line consisting of `Juliet is the sun` is returned. This is because the Unix search tools return the longest matching substring. In order to get comparable output, one must change the pattern to `.*(Romeo|Juliet).*`.

We mention all this because it affects our mapping between Java regex and Cuda-grep that is similar to `egrep -x`.

3.7.2 Working Environment and Benchmark

Implementation has been done using Java as a plugin (openjdk7) for NVIDIA's Nsight Eclipse Edition (version 5.5.0), ASM v5 and CUDA 5.5. We also used the `javac`, `gcc`, and `nvcc` compilers. Testing was performed on a CPU and a GPU with the specifications shown in Table 3-6.

Table 3-6: CPU and GPU specifications

	CPU	**GPU**
Model	AMD A6-5400K APU with Radeon(tm) HD Graphics	GeForce GT 640 Kepler, CUDA capability 3.0
Number of cores	2	384 (2 MPs, 192 cores/MP)
Cache size	1024 KB	L2: 262144 bytes
Clock rate	CPU: 1400 MHz	GPU: 902 MHz Memory: 891 MHz
Memory	8 GB	Global: 1024 Mbytes Shared per block: 49152 bytes Constant: 65536 bytes Registers per block: 65536
Number of threads	2	Maximum per MP: 2048 Maximum per block: 1024

As a benchmark, a text file of 142 MB was used. The file consists of concatenating the 11th edition of the Encyclopedia Britannica downloaded from the project Gutenberg website [20].

3.7.3 Results and Interpretations

As mentioned in the previous section, the results returned from running Cuda-grep (and similarly `egep -x`) and from running Java regex library will be different if the same pattern is used. In the first case, the whole line is returned, and in the second, only the matching group is returned. In order to overcome this difference and to be able to obtain matching results, we have also altered our patterns to return the whole line in which a pattern is found.

We have performed a few warm-up runs and then averaged the times shown over several runs. Table 3-7 shows the execution times of running several patterns on both CPU and GPU. The percent improvement is calculated as follows:

% improvement = ((CPU time – GPU time)/CPU time) x 100

On average, the percent improvement of running the matching on the GPU is 98.11% faster than on the CPU. The data is also plotted in Figure 3-3.

Table 3-7: Runtimes and percent improvement with GPU

	Patterns	CPU time (ms)	GPU time (ms)	% improvement
1	.*romeo.*	78482	1745.6	97.77
2	.*ROMEO.*	84481.7	1387.1	98.35
3	.*Romeo.*	75728.4	1386	98.17
4	.*Juliet.*	74472.6	1378.5	98.15

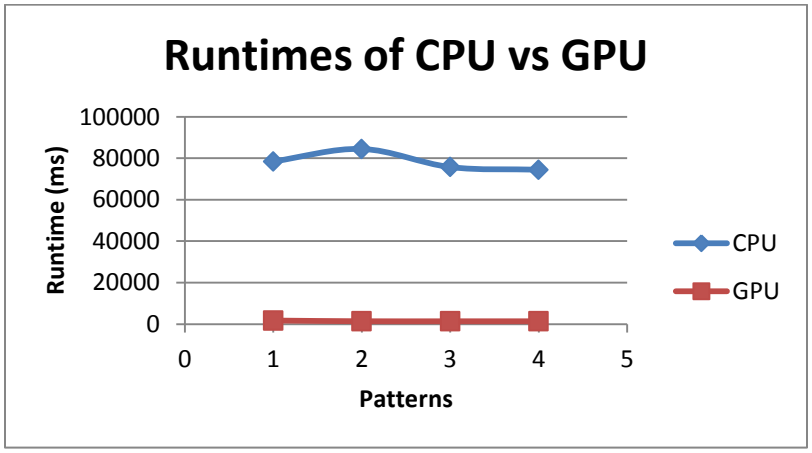


Figure 3-3: Runtimes of CPU vs GPU

Cuda-grep is case-sensitive. Therefore, we have performed an additional set of experiments to measure how including case sensitivity (and thus rendering the pattern more complex) will affect execution. We have compared the results to two Java implementations: the first uses the same pattern we used for the GPU (case 1), and the second uses the built-in flag (Pattern.CASE_INSENSITIVE) that can be added when compiling a pattern (case 2). Our results are summarized in Table 3-8. We have used the following definitions to make the tables more compact and clear:

- Case 1: (.*romeo.*)|(.*ROMEO.*)|(.*Romeo.*)
- Case 2: .*romeo.*, Pattern.CASE_INSENSITIVE

Table 3-8: Run times taking into consideration case sensitivity

CPU time case 1 (ms)	CPU time case 2 (ms)	GPU time case 1 (ms)
232,308.2	92,996.6	2,765.3

Note that GPU time with case 2 does not exist. We observe that using the flag in the CPU implementation is more efficient than listing the combinations by about 60%. This is because in the first case expensive backtracking and grouping were performed. In spite of that, the GPU implementation was still able to outdo both as shown in Table 3-9.

Table 3-9: Percent improvements of GPU over cases 1 and 2

	GPU over case 1	**GPU over case 2**
% improvement	98.81 %	97.02%

3.8 Conclusion

We have presented a tool that detects a user's function call to search for a pattern and seamlessly replaces it with our own parallel GPU-based implementation. We have used bytecode injection and JNI to run our optimized code. Our CUDA code is based on the CUDA-grep project. Experiments show a reduction in total execution time. As GPUs develop and as newer GPUs are able to overcome the overhead imposed by necessary but expensive I/O, we believe that there could be more room for optimized performance.

Chapter 4: Parallel Pattern Matching using MPI

As detailed previously, we have chosen to parallelize the pattern matching process. In this chapter, we discuss implementing pattern matching on a distributed environment using MPI. Parallelism in MPI is explicit. This means that the developer has to correctly identify areas for parallelism and to implement parallel algorithms using the MPI libraries. Our approach alleviates this burden from the developer since MPI programming is considered quite complex and has a high learning curve. Our contribution is summarized as follows: detect a user's Java call to search for a pattern, instrument the pattern matching code to run on a cluster while accounting for process communication, and finally return control back to the program. In addition to that, we implement another distributed approach using Hadoop's MapReduce and compare that to the multithreading results obtained in a previous chapter.

4.1 Introduction

Distributed systems are the other form of parallel systems that we explore. As opposed to shared memory systems wherein memory is shared, a distributed system consists of several autonomous computational nodes (commonly referred to as a cluster) that each have their own local memory module and that communicate with each other using a high speed communications network to relay information typically in the form of message passing.

In order for a distributed system to be reliable, it must be fault-tolerant, highly available, recoverable, consistent, scalable, have a predictable performance, and secure [51].

The key issue in programming such systems lies in distributing the data over the individual distinct processors' memories. Distributed computing is the field that studies computational problems performed on distributed systems. In short, a large complex problem is divided into

many smaller sub-problems. These sub-problems are then solved using one or more nodes which communicate with other nodes in the cluster. Computational tasks per node can only operate on the data local of each node. Hence, if remote data is needed, the task must communicate with other remote processors that contain the required data. The results from each processor are then reassembled into one solution.

The underlying network architecture that connects these nodes is out of the scope of our research. What we do discuss is the implementation vis a vis a programming perspective; i.e. message passing interface and implementations.

Several message passing libraries had been available prior to 1992. It was difficult for programmers to develop portable applications because these implementations varied completely. Therefore, an effort began by a small group of researches to establish a standard interface. The Message Passing Interface (MPI) version 1.0 was released in 1994. Later versions include MPI-2 (released in 1996) and MPI-3 (released in 2012). Since its release, MPI has become the industry standard for message passing, replacing all previous implementations. This has allowed parallel hardware vendors to have a clear set of defined routines that can be easily implemented and has made applications portable on several parallel platforms. The MPI standard was incorporated by adopting features from systems by IBM, Intel, nCUBE, PVM, Express, P4, and PARMACS [52].

Another form of communication used in distributed systems is based on the MapReduce model which was first introduced by Google. Instead of using messages to relay data between processors, MapReduce is based on reshuffling and duplicating data. MapReduce was developed to address the shortcomings of MPI. As mentioned previously, an important factor in distributed

systems is reliability. However, MPI is not very reliable. Therefore, MapReduce, and more specifically Hadoop, was developed to account for reliability.

Without going into details of both techniques since they will be covered extensively later on, the major differences between MPI and MapReduce can be summarized as follows:

- Reliability: In MPI, if one node dies, the whole application crashes. In MapReduce, data is duplicated as a precautionary measure and if a node is down, the system can adjust and re-compute that node's task. Fault tolerance can also be done in MPI, but it requires more effort and time to get it right.
- Programming efficiency: With MapReduce, setting up applications and getting them running in a scalable fashion is faster.
- Flexibility: MPI is more flexible in the sense that a developer has more control over data and data locality. With MapReduce, developers have no control over data locality. Data is randomly distributed over the nodes.
- Performance: MPI is generally more efficient in terms of performance than MapReduce, especially if an application does not fit the map-reduce model.

One can conclude that MPI is favorable for fast, reliable networks; whereas MapReduce is favorable for slow unreliable hardware.

The rest of the document is organized into the following sections. Section two is an introduction to MPI and the various available MPI implementations. Section three surveys the literature with respect to pattern matching implementation performed using MPI. Section four describes our approach and our implementation. Section five shows our experimental results and

analysis. Sections six and seven discuss Hadoop as an alternative solution for distributed systems and section eight concludes the chapter.

4.2 Introduction to the Message Passing Interface (MPI)

MPI is a standardized language-independent specification of what a message passing library should be. It attempts to be practical, portable and is supported on virtually all HPC platforms, efficient since vendor implementations can exploit native hardware features to optimize performance, functional with over 430 different routines in MPI-3, and flexible since a variety of both vendor and public domain are available. It defines the syntax and the semantics of a set of functions that can be used by applications to exchange messages between processes in C, C++, and Fortran. The protocol supports both types of communication: point-to-point and collective. Each CPU core gets assigned a process at runtime. Originally, MPI was designed for distributed memory architectures, but has been adapted to handle hybrid distributed memory and shared memory systems as well and can run on virtually any hardware platform. Different interconnects and protocols support has also been developed.

There are several implementations of the MPI standard, and they differ in which version and which features of the standard they support. Open MPI is perhaps the most robust and prevalent implementation available, and it is the one we have chosen to use. In what follows, we discuss the major architectural concepts and the several implementations available, as well as go into a little more detail into Open MPI and why we have selected to use it.

4.2.1 MPI Concepts

In MPI, all the processes that communicate with each other are included within an entity called a communicator. Once a communicator has been initialized, functions determining the total number of processes, the rank of the calling MPI process, the processor name, the MPI version used, elapsed wall clock time, and resolution can be performed. Also, calls to abort and finalize are available. The rank is a unique identifier for each process and is used for inter-process communication identification.

A typical program starts by initializing the MPI environment, determining the total number of processes and the rank of the current process, performing the communications and per process computations, and ending by terminating the communicator. A template is shown in Figure 4-1. Every process runs the same code. However, some variables locally defined will need to be broadcast or sent to the other processes that will need to handle the data. This is confusing for some programmers because contrary to common practice, defining and initializing a variable might not necessarily mean it is readily available in that process's local memory.

```
import mpi.*;

…

//class definitions

//variable declarations

final static int ROOT = 0;

//sequential code

…

//initialize MPI environment

MPI.Init(args);

//main functions used in almost all MPI programs

int myRank = MPI.COMM_WORLD.getRank();

int numProcs = MPI.COMM_WORLD.getSize();

//code executed on ROOT only

if(myRank == ROOT){

   …

}

//parallel code executed on each process

//example: do pattern matching

...

//clean up MPI environment

MPI.Finalize();
```

Figure 4-1: MPI code template

Message passing is fundamentally based on sending and receiving messages among processes. A process sends a message to another process by providing the rank and a tag to identify the message. The receiver posts a receive message along with the sender's rank and anticipated tag. Such communication that involves one sender and receiver is referred to as point-to-point communication, and can be synchronous or asynchronous. Communication can also be one-sided.

Another form of communication is collective communication. In collective communication, all the processes are required to take part in the process. Examples include broadcast, scatter, gather, and reduce operations among many others. Technically these operations could be implemented with send/receive variations (which is what typically happens underneath the surface). However, they are less cumbersome to write and they use the network in a more efficient manner. Collective calls are synchronous calls and an implicit barrier is internally implemented.

Complex parallel programs usually include a mix of point-to-point and collective communication and/or I/O where a set of MPI processes access storage subsystems.

4.2.2 MPI Implementations

As previously mentioned, MPI is an interface. It is not tied to any hardware or software environment and is therefore up to developers to implement for the different architectures. Many efficient, well tested, and open-source implementations of MPI exist. They might differ in which version and features of the standard that they support, and in how they are compiled and run on various platforms. MPI implementations consist of C, C++, and Fortran APIs. Most implementations also consist of C#, Java, or Python bindings, or any other language that is able to interface with the original libraries.

MPICH was the initial implementation of MPI-1. IBM also provided its own implementation. Another known implementation is Open MPI which was formed by merging several other previous versions. Commercial implementations are derived from these initiatives.

4.2.2.1 Open MPI

Open MPI is used by many TOP500 supercomputers. It is an open source implementation of MPI that was developed by merging the following previous well-known implementations: FT-MPI from the University of Tennessee, LA-MPI from the Los Alamos National Laboratory, LAM/MPI from Indiana University, and PACX-MPI from the University of Stuttgart [53]. The developers intended to use the best features and technologies from these independent projects in order to create a better overall open source implementation. Open MPI fully supports MPI-3 standards, thread safety and concurrency, several operating systems, and provides high performance on all platforms.

4.2.2.2 Java Bindings

Even though MPI specifications require a C or Fortran interface, the language used to implement MPI can be different. Most implementations use C, C++ and assembly to target C, C++, and Fortran programmers. However, bindings are available for many other languages such as Python and Java among others. Java bindings were developed due to the increasing interest of using Java for HPC. MPI can benefit as well due to the widespread use of Java that makes it likely to be applied further beyond traditional HPC applications [53].

Several implementations have developed Java bindings. We mention a few, while noting that we have used the Open MPI implementation in our work. The first attempt called mpiJava is from [54]. It comprises a set of JNI wrappers to a local C MPI library. The project suffered from limited portability and was not entirely complete. However, it formed the basis for other projects to build upon, including the Open MPI binding. MPJ Express [55], MPJava [56], and F-MPJ [57] are other projects that have taken a different route. The programmers developed the bindings on

a pure Java basis. This is based on Java sockets and specialized I/O interconnects, and requires significant coding effort.

One of the challenging parts of creating Java bindings stems from the language characteristic. Java does not support explicit pointers, and it is inefficient in transferring multidimensional arrays and complex objects because of how objects are stored and/or copied in the address space. Typical workarounds include deserialization and casting.

As mentioned previously, we have chosen the Java bindings by the Open MPI project that supports the complete MPI-3.1 standard minus a few exceptions that were irrelevant to our work. The approach is based on JNI and consists of an interface that lies ontop of a C native library [58]. The bindings are a 1-to-1 mapping to the MPI C bindings. We discuss further details in Section 4 in their relevancy to our implementation decisions. The authors state that MPI Java bindings perform well compared to C and Fortran code, stating that over the NAS Parallel Benchmarks (NPB), the results were satisfactory and had only a slight overhead.

4.3 Literature Survey

In this section, we survey three topics. First, we list some of the previous work that has been done for parallelizing pattern matching algorithms using MPI. Second, we explore previous work that has attempted to automate MPI code generation. And finally, we discuss some Java implementations in high performance computing.

In [59], the authors use both MPI and OpenMP separately to implement the Naïve, Karp and Rabin, Zhu and Takaoka, Baeza-Yates and Regnier, and the Baker and Bird exact two-dimensional on-line pattern matching algorithms. They concluded that for the same set of algorithms and data sets, OpenMP is more efficient since it does not impose any communication

costs. However, MPI might be favorable since it is more scalable as opposed to OpenMP which is limited by the number of cores on a single processor. In [60], the authors propose four master-worker models to perform parallel multi-pattern matching on a heterogeneous cluster. They further validate their theoretical models by experiments using MPI. In the first model, the documents are distributed among the workstations and stored on local disks. Each worker reads its subtext from its main memory and executes the searching algorithm. The master then collects the results from each worker. This model has low communication overhead since each worker searches its own subtext and there is no communication with other workers or the master. The drawback is the possibility of load imbalance. However, this can be eliminated with the use of a better partitioning technique. The second model consists of having the entire text reside on the master's local disk. The master sends the chunks of text to the workers that each perform their own search. This is a dynamic approach that has low load imbalance but higher inter-workstation communication overhead. The third model consists of a dynamic allocation of pointers. The whole text is stored on local disks of all the workstations and the master maintains pointers to the subtexts and sends these pointers to the workers. This reduces inter-workstation communication overhead but requires more space to store all the files on all the workers' local storage space. The fourth and most efficient model in terms of execution time and speedup consists of a hybrid implementation of the first model along with an optimal distribution method.

The tool in [61] automatically generates MPI source code from OpenMP. OpenMP parallel loops are detected and are distributed in a master-worker pattern. This is done by manipulating the AST, performing loop analysis and workload distribution. Only peer-to-peer send/receive protocols are supported. The user code to be transformed has to be originally parallel for the tool to detect pragmas. The authors also mention that in order to obtain significant performance gain

from MPI over OpenMP, many processes are needed. In [62], the authors deal with a sub-problem of MPI code generation: generating MPI type-maps from user source code. MPI derived datatypes are custom datatypes that are more complex than the primitive datatypes. Users need to manually maintain descriptions for their datatypes so that the MPI routines can properly transmit them. In [63], the authors propose a framework to generate MPI code that is type safe. The motivation behind this work is that the most common MPI programming error consists of communication mismatch between senders and receivers. Instead of verifying code correctness, the authors automate communication generation based on the user provided sequential code and on a supplied interaction protocol. Type safe code reduces lost messages, communication deadlocks, and calculation errors. In [64], the authors present a directive-based tool that transforms sequential C code to a parallel message passing form. They also introduce a loop scheduling method for load balancing. Directives are marked in the C code to indicate and enclose which for-loop to parallelize, which block will be initialized for all nodes, and which parts to synchronize. The authors take the following assumptions into consideration: loops are expressed by for-statements, only outermost loops are parallelized, parallelizable loops do not contain pointers, and the boundaries of each loop are considered static and known at compile time. The process consists of two passes. The first pass scans the source program to process directives, prepare loops, and create a def-use symbol table. In the second pass, the tool examines the loops based on the def-use table and generates the MPI code. The experimental results show that handwritten optimized codes performed better than codes generated by the system, and the authors list some factors that might have led to that. The authors in [65] perform event traces by using pattern matching in order to analyze an MPI program. They aim to detect groups of communication patterns that may resemble collective operations. Their system informs

the user of the potential match, and thus users can replace their point-to-point operations with the suggested collective functions in order to obtain better performance. The motivation behind their work is that inexperienced users might not know which collective functions already available to use instead of non-optimal send/receive would better fit their program behavior.

The authors of [57] also presented a study [66] of the state of Java for HPC for both shared and distributed memory. While their overview was current at the time their work was published, it has become incomplete at the time of writing this work. However, we mention their main observations: the interest in Java for HPC has led to several projects although still modest; Java can achieve almost similar performance to natively compiled languages and can be an alternative for HPC programming; the advances of Java communications in a shared memory environment can bridge the gap between Java and natively compiled HPC applications. In [67] the authors discuss their results in evaluating the performance of a parallel deterministic annealing clustering program using both Open MPI with Java bindings and FastMPJ as compared to native Open MPI and MPI.NET. They do so by migrating existing C# based code to Java by rewriting the algorithms.

4.4 Implementation

In this section, we discuss the reasons behind some of our implementation decisions and the approaches we took to perform pattern matching using the Open MPI Java bindings. The first part explains how we achieved function detection, how we linked the MPI code to the user's, and how we return the MPI results from the distributed environment back to the user main code and allow him/her to proceed to seamlessly run his/her code. The second and third parts deal with the MPI implementation. Because of some MPI standard restrictions and also because of Java

language restrictions, we faced some challenges that would have been otherwise easy to deal with in a C environment. We have developed three different approaches to our problem, each complete in its own way (as in each solution solves all the anticipated problems) and we compare these approaches in terms of efficiency and code complexity.

4.4.1 Overall Process

As mentioned in the introduction, the purpose of this chapter is to use MPI to parallelize an otherwise sequential user's function call for pattern matching without the user's intervention. In Chapter 2, we have achieved this for shared memory by using multiple threads. However, with distributed memory, our approach has to be slightly different in the sense that our data now resides on multiple machines, and these machines will each process part of the data. So far, this does not seem like an issue. However, recall that our purpose is to detect pattern matching (which would typically be in the middle of the program), replace the sequential implementation by a parallel one, and then return control to the original user code that will probably use the result of the matching (which will have to be in memory) in some way or another. To integrate the MPI implementation, we will need to execute the MPI application externally from Java, read the system results, and then parse them and send them back to the Java application. The reason for this is that MPI executables cannot be launched without the `mpirun` command. In addition to launching multiple copies of the MPI executable, `mpirun` also exports special environment variables to the launched MPI processes. Therefore, calling `mpirun` is necessary, and in Open MPI there is no way to embed its functionality in the program. It has to be explicitly called. To handle this, we have used the Java `exec` command. This command executes external applications from within the Java program. Figure 4-2 shows a code snippet.

```
…
Runtime rt = Runtime.getRuntime();
String cmd = "/usr/local/bin/mpirun -np " + np + " " + hostFile + " java MPI_Matcher "
            + textFile + " " + pattern + " |sort";
Process pr = rt.exec(cmd);
BufferedReader input = new BufferedReader(new InputStreamReader(pr.getInputStream()));
String line=null;
while((line=input.readLine()) != null) {
            //parse the output
      }
int exitVal = pr.waitFor();
System.out.println("Exited with error code "+ exitVal);
…
```

Figure 4-2: Sample exec code

First, we create a Runtime object and attach it to the system process. In this example, we are using the `mpirun` command. It takes the following arguments that are dynamically set based on the user's input:

- `-np`: number of processes to be run
- `-hostfile`: contains the names of the parallel machines and how many slots are available in each
- `textFile`: the text file name to process
- `pattern`: the pattern to search for

The call to `exec` launches `mpirun`. But we still need to collect its output. To do so, we use a buffered reader to read the input stream, which is then parsed and used (we do not show here how). Finally, the `waitFor()` method will make the current thread wait until `mpirun` finishes

and will return an exit value to the current thread. An exit value of ‘0’ means that the external program has executed successfully. Our overall process is shown in Figure 4-3.

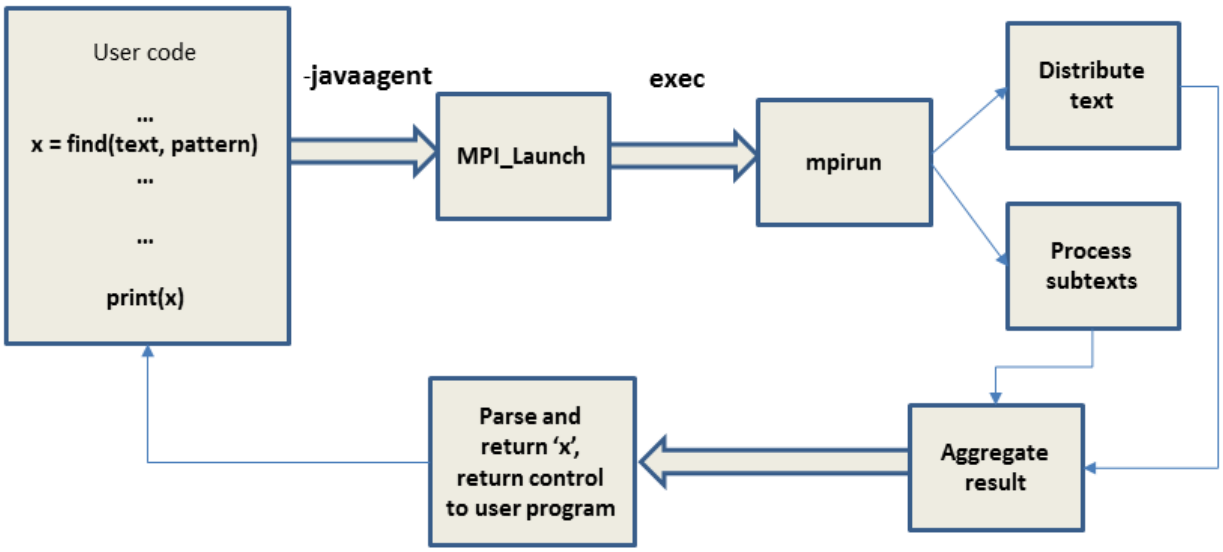


Figure 4-3: Overall execution process

The reason we did not use JNI as we did when parallelizing for GPUs, is that our function call is in Java since we are using Open MPI Java bindings. Had we wanted to use a C Open MPI library for example, we would have needed to use JNI instead.

4.4.2 MPI Design Decisions

Before developing the MPI code and worrying about which communication operations to use, there are two key issues to consider: text partitioning, and data distribution.

According to [68], there are two major steps in designing parallel algorithms: dividing a computation into smaller parts, and assigning these parts to different processes for parallel execution. Properly selecting methods to handle these issues will reduce the overhead of parallel algorithms by decreasing execution time in that it minimizes the time processors are idle due to an uneven load distribution.

There are two mapping techniques available: static and dynamic mapping. Static mapping is generally used in a homogeneous environment where the processes have the same characteristics; whereas dynamic mapping is generally used in a heterogeneous environment where processes might have different characteristics. In static mapping, each process receives the same amount of data proportional to the number of available processes before the execution of the algorithm starts. In dynamic mapping, data is divided into more parts than the available processes. Each process is assigned a chunk size depending on its capacity, and this is the amount of data each process receives during the execution of the algorithm. In [60] this is discussed in more detail. In order to achieve a good balanced distribution among heterogeneous nodes, the amount of text distributed should be proportional to its processing capability within the network. This is given in Equation (1) where L_i represents the processing capacity of process i, S represents the speed, and p is the total number of processes available.

$$\boldsymbol{L_i = \frac{S_i}{\sum_{j=1}^{p} S_j}} \quad \textbf{(1)}$$

Thus the amount of data distributed to each process i will be $L_i \, x \, (n + m - 1)$[4].

In our case, and since we have a homogenous distributed environment as discussed in the next section, we use a static approach. Using a dynamic approach would result in the same load balance as a static approach, since the S_i's are equal and hence L_i would reduce to $1/p$. We will delay discussing exact data distribution to the next section in which we propose the communication operations used since they are dependent.

The other thing to consider is the data distribution. Several models exist such as the Data-parallel model, the Task Graph model, the Work Pool model, and the Pipeline Model. In [69], it

[4] Recall that (m-1) is the boundary overlap, where m is the pattern size, and n is the length of the text to partition.

was concluded that the Master-Worker model was the most appropriate for pattern matching on both distributed and shared memory systems. Therefore, we have chosen this model for our implementation. It corresponds to the typical master-worker scheme in which a master node distributes data to the other worker nodes. Each worker node then processes the data and when finished, sends the results to the master node. The individual results are gathered by the master, and depending on the required output, a reduction might be performed to present the final overall results.

4.4.3 Open MPI Java Bindings Implementation Approaches

There are several communication approaches for message passing. In pattern matching, the text needs to be partitioned so that processing can occur on the subtexts. These subtexts are what needs to be passed among the nodes. We have chosen to implement the following three approaches to pass the subtexts and compare their performance:

- point-to-point communication using send/receive
- collective communication using scatter
- minimal communication where whole text is local to all nodes

We discuss each implementation next.

4.4.3.1 Point-to-point Communication (Approach 1)

In this scenario, one of the processes is labeled as root. The text to process is only in local storage of the node that contains the root. The root reads the text and processes it. It then divides the contents among the other processes which could be on other nodes as well. Therefore, a root process performs a send operation sending individual subtexts, and all other processes perform a

receive operation receiving the data upon which to perform the matching. Load balancing is achieved since the text is partitioned in equal parts. We have chosen to implement a static block scheduling approach instead of a cyclic one.

One of the issues we faced is specific to the Java language, the message passing interface, and the Java bindings. To send or receive a message, the following must be provided:

- an array (or buffer) holding the contents of the message to send or receive
- the number of elements to send or receive
- the type of the elements
- the source/destination process involved
- the tag identifier of the message

Here is an example for a simple send and a receive operation.

```
MPI.COMM_WORLD.send(subText, count, MPI.CHAR, processID, tag);
MPI.COMM_WORLD.recv(lsubText, count, MPI.CHAR, ROOT, tag);
```

To send a text, we will first need to convert it either into an array or into a buffer, because the Java bindings can only handle arrays or buffers. According to the documentation [58], it is better to use buffers when dealing with large messages due to the way copying and moving data is handled internally in the Java bindings implementation. We have actually tested both scenarios and when we implemented the array version, our program would hang due to the large message size. Hence, in this approach and the next one, we have used char buffers instead of arrays.

In Java, we have to initialize a buffer with its size before we can use it. This is not a problem on the send size, because the root process knows how big each partition is going to be. However,

this is not true on the receiving end. Each process needs to know how big its message is going to be in order to allocate memory to receive it. One way of doing that is to actually send each partition size to the corresponding process using another send operation. However, that would incur extra performance penalty due to increased communication. Another way would be to allocate a huge buffer size to handle all possible sizes of messages. However, that would require extra unused space. Therefore, we implemented an alternative dynamic approach. Each process would first probe the message before it is received in order to dynamically determine its size, allocate memory for it, send the size to the receive operation, then receive the message.

4.4.3.2 Collective Communication (Approach 2)

Instead of using send and receive messages, a scatter operation can be performed. Scatter is a collective operation that involves a root process sending chunks of data to all processes in a communicator including the root itself. Figure 4-4 shows an illustrative example. The communicator here has four processes. Process 0 is the root process. Scatter takes an array of elements and distributes the elements to all the processes in order of rank. The first element goes to process 0, the second to process 1 and so on so forth.

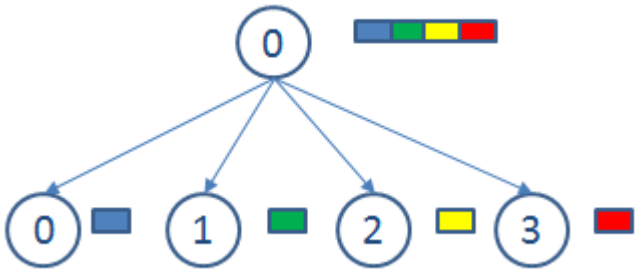


Figure 4-4: Scatter illustrative example

A scatter function requires the following parameters:

- send_data: array that resides on root
- send_count: how many elements will be sent
- send_datatype: type of data elements
- recv_data: receive buffer
- recv_count: how many elements will be received
- recv_datatype: type of data elements
- root process

If for example, `send_count` is 5 and `send_datatype` is `MPI_INT`, then process 0 will get the first 5 elements of the array, process one will get the next five, and so on. Therefore, `send_count` is usually equal to the number of elements in the array divided by the number of processes.

There are three things to note for the scatter operation. First, and similarly to all collective communication routines, it does not support derived datatypes. Messages should be of a primitive datatype. Hence, we cannot send an array of Java Strings for example. This was not a problem for us since we divide the string into characters and send those. Another issue is that scatter (and also gather) does not allow for boundary overlap. This is a restriction in the MPI standard, and failing to abide by it will result in unpredictable behavior. In pattern matching though, we need to send (m -1) overlap characters. The third issue arises when the number of elements in an array is not divisible by the number of processes. In that case, a `scatterv` function can be used that in addition to the previously discussed parameters, holds a reference to an array that contains the number of elements to send to each process. However, we did not use

this functionality. The solution we propose solves both the second and the third issue. Our solution performs a pre-processing step on the root process. The root reads the text, and appends the (m-1) overlapped characters into their corresponding place, then pads the last partition to fit the maximum partition length. What we have now is a new string that when divided equally among the number of processes will have the same partition length and the correct characters to search.

4.4.3.3 Local Communication (Approach 3)

In this scenario, the text to search for patterns within resides in the local storage of all the nodes. Hence, there is no need for a root process to partition the text. Instead, each process, based on its rank in the communicator, is able to calculate which parts of the text pertain to it and perform the matching.

4.4.3.4 Gathering Results

The three approaches mentioned earlier discuss how to partition and send the data. Once that is accomplished, each process has its part of the data upon which it will search for the pattern. The results can then be sent to the root process where they are aggregated using a reduce operation that performs similar to the addition reduction shown in Figure 4-5. One thing to note here is that since the root will have to wait until all other processes are done with their results, it will be dependent on the slowest process.

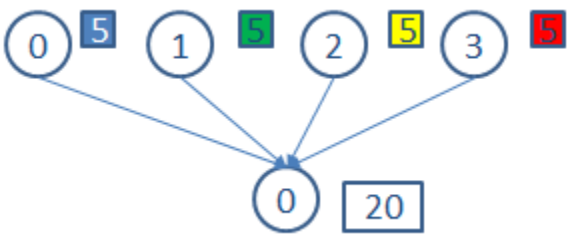


Figure 4-5: Reduce illustrative example

4.5 Experimental Results and Interpretations

4.5.1 Working Environment and Benchmark

Implementation has been done using Java (jdk 8) on Eclipse IDE (Luna - release 4.4.0) and ASM v5, Java bindings for Open MPI v2.0.1 which supports MPI-3.

Testing was performed on two connected virtual machines with the specifications shown in Table 4-1.

Table 4-1: VM specifications

Model	Intel(R) Xeon(R) CPU E5-4650 0 @ 2.70GHz
Architecture	x86_64
Number of CPU cores	8
Number of threads	16
Cache size	20480 KB

As a benchmark, a text file of 142 MB was used. The file consists of concatenating the 11th edition of the Encyclopedia Britannica downloaded from the project Gutenberg website [20].

4.5.2 Results and Interpretations

In this section, we will present the results from our three approaches and will perform an analysis of our findings. We have recorded four cases for each of our approaches, as shown in Table 4-2.

Table 4-2: Test cases process distribution

	Number of processes on VM1	**Number of processes on VM2**
Case 1: local	4	0
Case 2: local	8	0
Case 3: distributed	4	4
Case 4: distributed	8	8

VM1 is the master node, and VM2 is the slave node. In the first two cases, we run the program on the master node only in a local manner, over four processes and over eight processes. In the last two cases, we run the program over two distributed nodes, once on four processes on each node, for a total of eight processes, and once on eight processes on each node, for a total of sixteen processes.

We have performed a few warm-up runs and then averaged the execution times shown over several runs. The percent improvement is calculated as follows:

$$\% \text{ improvement} = ((\text{sequential time} - \text{parallel time})/\text{sequential time}) \times 100$$

Table 4-3 shows the execution times (in seconds) of running several patterns of all three approaches (send/receive, collective, and local) for the setup of case 1. It also shows the percent

improvement over the sequential program. Table 4-4, Table 4-5, and Table 4-6 show similar data for cases 2, 3, and 4 respectively.

Table 4-3: Case 1 runtimes and percent improvement: VM1: 4 processes - VM2: 0 processes

	Sequential	**Approach 1**	**Approach 2**	**Approach 3**	**% improvement of approach 1**	**% improvement of approach 2**	**% improvement of approach 3**
Pattern 1	41.615	14.458	11.635	11.8	65.25	72.04	71.64
Pattern 2	33.007	11.296	10.1	10.052	65.77	69.4	69.54
Pattern 3	118.468	41.903	31.34	30.413	64.62	73.54	74.32
Average					65.21 %	71.66 %	71.83 %

Table 4-4: Case 2 runtimes and percent improvement: VM1: 8 processes - VM2: 0 processes

	Sequential	**Approach 1**	**Approach 2**	**Approach 3**	**% improvement of approach 1**	**% improvement of approach 2**	**% improvement of approach 3**
Pattern 1	41.615	7.297	7.203	7.385	82.46	82.69	82.25
Pattern 2	33.007	6.474	6.811	6.696	80.38	79.36	79.71
Pattern 3	118.468	18.393	17.214	17.365	84.47	85.46	85.34
Average					82.43 %	82.5 %	82.43 %

Table 4-5: Case 3 runtimes and percent improvement: VM1: 4 processes - VM2: 4 processes

	Sequential	**Approach 1**	**Approach 2**	**Approach 3**	**% improvement of approach 1**	**% improvement of approach 2**	**% improvement of approach 3**
Pattern 1	41.615	11.639	11.335	11.795	72.02	72.76	71.64
Pattern 2	33.007	11.076	11.553	11.086	66.44	64.99	66.41
Pattern 3	118.468	30.922	30.052	29.87	73.89	74.63	74.78
Average					70.78 %	70.79 %	70.94 %

Table 4-6: Case 4 runtimes and percent improvement: VM1: 8 processes - VM2: 8 processes

	Sequential	**Approach 1**	**Approach 2**	**Approach 3**	**% improvement of approach 1**	**% improvement of approach 2**	**% improvement of approach 3**
Pattern 1	41.615	7.295	7.191	9.35	82.46	82.72	77.53
Pattern 2	33.007	6.708	7.014	8.872	79.67	78.74	73.11
Pattern 3	118.468	16.117	15.77	18.964	86.39	86.68	83.99
Average					82.84 %	82.71 %	78.2 %

As we can see from the previous tables, execution time decreases dramatically using MPI. It is worth noting that the three approaches exhibit similar behavior for the same pattern and the same case within a maximum of 5% difference. This is because for Approach 3 (where the file is stored locally on each node) there is no communication overhead between the nodes. It has been shown [64] that using some collective routines, such as scatter, does not necessarily improve performance. It can in fact incur an extra overhead compared to multiple send/receive operations

implemented with the same number of processes. However, the reason why our Approach 2 (using scatter) is similar to Approach 1 (using send/receive) is that in Approach 1, there is one less process that performs the matching, since the root process is not part of computation.

We can also observe that results for cases 1 and 3, and results for cases 2 and 4 follow the same trend. Consider cases 1 and 3. Case 1 runs on one virtual machine with four processes. Case 3 runs on two virtual machines with four processes each, for a total of eight. However, the execution times are similar. The reason for this is that execution time is dependent on the slowest process. Since case 3 is a distributed environment, the communication overhead incurs extra runtime. We have recorded the execution times for case 3 for each process in Table 4-7. As can be seen, processes running on the local machine (VM1) are faster than those on the remote node (VM2). This excludes the root process (process 0) which has to wait for the slowest process to terminate before it can aggregate the results. Similar analysis applies to cases 2 and 4.

Table 4-7: Execution times for Case 3 for every process

		VM1				**VM2**			
		Proc 0	**Proc 1**	**Proc 2**	**Proc 3**	**Proc 4**	**Proc 5**	**Proc 6**	**Proc 7**
Approach 1	*Pattern1*	10.963	6.281	6.460	6.459	10.963	10.927	10.909	10.909
	Pattern 2	10.396	5.324	5.451	5.322	10.396	10.169	10.392	10.264
	Pattern 3	30.242	16.439	17.321	17.258	30.242	29.824	29.563	28.563
Approach 2	*Pattern1*	10.634	6.063	6.382	6.316	10.635	10.594	10.273	9.705
	Pattern 2	10.814	5.287	5.501	5.497	10.814	10.592	10.534	10.235
	Pattern 3	29.249	16.542	15.816	15.742	29.248	28.784	28.564	28.365
Approach 3	*Pattern1*	10.971	5.517	6.211	6.126	10.971	10.940	10.771	10.771
	Pattern 2	10.273	4.708	4.925	4.749	10.273	10.179	10.114	9.888
	Pattern 3	29.015	16.088	15.622	15.572	29.015	28.991	28.542	28.542

We make another observation for Approach 1. As mentioned, in this approach process 0 does not take part in the computation. Its sole responsibility is to partition the text, distribute it, and collect the results once all the other processes have terminated. Contrary to the other approaches, the root does not perform pattern matching. Hence, partitioning the text is done over one less process compared to the other approaches. Table 4-8 shows the execution times with four, five, eight, and nine processes on one node. We observe that for five and nine processes, execution times get closer to those recorded for Approach 2. In Table 4-9 we compare the percent improvement after adding the extra process. Running over five processes (only four of which take part in computation) performs 20.47% better than running over four processes. And running over nine processes (only eight of which take part in computation) performs 7.29% better than running over eight processes.

Table 4-8: Execution times (seconds) for Approach 1

	Sequential	**np = 4**	**np = 5**	**np = 8**	**np = 9**
Pattern 1	41.615	14.458	11.362	7.297	6.748
Pattern 2	33.007	11.296	9.715	6.474	6.193
Pattern 3	118.468	41.903	31.001	18.393	16.553

Table 4-9: Percent improvement comparison for Approach 1

	% improvement np = 4 over sequential	% improvement np = 5 over sequential	% improvement between np = 4 and np = 5	% improvement np = 8 over sequential	% improvement np = 9 over sequential	% improvement between np = 8 and np = 9
Pattern 1	65.25	72.69	21.41	82.46	83.78	7.52
Pattern 2	65.77	70.56	13.99	80.38	81.23	4.34
Pattern 3	64.62	73.83	26.01	84.47	86.02	10.00
Average			20.47			7.29

We can conclude that all three approaches exhibit similar results. Therefore, selecting which approach to use depends on factors other than performance as shown in Table 4-10. If for example storage is not ample, then using Approach 3 is not feasible for large files.

Table 4-10: Advantages and disadvantages of the three approaches

	Advantages	**Disadvantages**
Approach 1	Easy to program	Root process does not take part in computation
Approach 2	Simplicity, programmability, expressiveness	File needs to be preprocessed
Approach 3	No communication overhead	File needs to reside on all nodes

4.6 Introduction to Hadoop

Hadoop is an Apache open source framework for reliable, scalable and distributed computing of large data sets [70]. It was created by Doug Cutting and Mike Cafarella in 2005. Cutting named it Hadoop after his son's toy elephant. Written in Java, Hadoop has three main objectives:

- Distributed computing: Hadoop is designed to work in an environment of distributed storage and computation of vast amounts of data (multi-terabyte data sets) across large clusters of computers.
- Reliability: Hadoop is designed to handle hardware failure and data congestion in a reliable, fault tolerant way.
- Scalability: Hadoop is designed to scale up from a single server to several thousands of servers.

The Hadoop framework is made up of the following four modules [71]:

- Hadoop Common: Java libraries and utilities required by the other modules to start Hadoop
- Hadoop Yarn: framework for job scheduling and cluster resource management
- Hadoop Distributed File System (HDFS)
- Hadoop MapReduce: parallel processing of large data sets

Since 2012, Hadoop also refers to the set of software frameworks that can be installed along with Hadoop. These include Apache Pig, Apache Hive, Apache HBase and Apache Spark.

Hadoop's main modules are HDFS and MapReduce. Both are similar to Google's File System (GFS) and MapReduce, respectively.

4.6.1 HDFS

The Hadoop Distributed File System, as the name suggests, is a distributed file system that is highly fault tolerant and is designed to be installed on low cost hardware and to run on large clusters. HDFS uses a master/slave architecture. The master consists of a single node, the NameNode. The slave(s) consist of DataNode(s). The NameNode manages the file system metadata, and the DataNodes store the data. A file in HDFS is split into several blocks which are stored in the DataNodes, based on a mapping provided by the NameNode. Like any other file system, HDFS has a shell and a list of commands to access and manipulate the file system.

4.6.2 MapReduce

MapReduce is the programming module used for processing data in parallel. A MapReduce job is divided into two phases. The map phase splits the input data into chunks (depending on the input file format specified) and processes these chunks as parallel independent tasks. The input is converted into a set of `<key, value>` output pairs. The framework then sorts these pairs based on their key values and sends them as input to the reduce task. The reduce phase aggregates the `<key, value>` pairs based on the reduction function, and produces an output consisting of a smaller set of `<key, value>` pairs. Both the primary input and resulting output are stored in the HDFS.

4.6.3 Procedure

The MapReduce framework consists of a single master ResourceManager, and several slave NodeManagers (per node). An application submits a job to Hadoop, specifying the input/output

locations, the map and reduce Java classes in the form of a jar file, and other parameters and configurations that might be needed. The Hadoop job client then sends the jar and configurations to the ResourceManager. The ResourceManager performs resource management, tracks resource consumption and availability, and schedules tasks to the slaves. The slaves execute the tasks and provide task-status information periodically to the master.

4.6.4 Usage and Advantages

Hadoop is a large scale distributed processing framework that can be used on a single machine. However, to obtain the full potential power of Hadoop, it must be scaled to hundreds or even thousands of nodes, each containing multiple cores. Hadoop has been demonstrated to work on clusters of up to 4000 nodes. The data that Hadoop deals with should also be huge. Hadoop was built to process web-scale data ranging from hundreds of gigabytes to terabytes or even petabytes [72]. Hadoop is not particularly known for being runtime efficient, yet it is widely used by industries and developers across the world. The reason for that is scale. The input data set that the HDFS can typically hold will in no way fit in the hard drive of one computer, or in the memory of a single machine. In most businesses, there is no choice but to use Hadoop's distributed file system to store and process their data. Another reason for Hadoop's popularity is its efficiency compared to the potential cost it can incur. For example, suppose one owns a computer with one thousand CPUs. That would be very expensive to obtain (assuming it exists of course), even though it would be fast and very efficient. As opposed to that, consider having one thousand single-CPU computers. Hadoop will connect these cheaper machines together to form a more cost effective, efficient, and reliable cluster. Other advantages of Hadoop include:

- It is open source and compatible on all platforms since it is Java based. And although it is implemented in Java, applications can be written in other languages. Hadoop Streaming allows developers to create and run jobs with many executables.
- It has a simplified programming model. Developers can quickly write and test distributed systems, since Hadoop automatically distributes the data and work across the nodes.
- It detects and handles failures without relying on hardware to provide fault tolerance.
- Nodes can be added dynamically to the cluster without interrupting operation.

The compute nodes and the storage nodes are the same. MapReduce and HDFS run on the same set of nodes, resulting in high data locality that leads to better performance. Data is mostly read from the local disk into the CPU. This reduces network bandwidth and prevents unnecessary transfers. Hence, computation is moved to the data, instead of moving the data to the computation.

4.7 Hadoop Comparison

We decided to investigate how our multithreading approach could compare to a distributed approach using Hadoop. Therefore, we implemented a pattern matching application in MapReduce. We also used the standard library's regex package. The mappers process the input and search for the pattern. The mapper produces the intermediate output `<key, value>` pairs. The key is a text object consisting of the pattern group and the value is an integer indicating the location of that group. These pairs are then passed to the reducer. The reducer aggregates the results based on the keys, and outputs another `<key, value>` pair, where the key is the pattern group and the value is a list of locations for that pattern.

Splits in Hadoop are of two types: physical and logical. Physical splits are how the HDFS chunks the files into blocks and distributes them among the nodes. This is based on a block size parameter that can be specified either in the `conf` files or when the input files are ported to HDFS. The logical splits are how the MapReduce splits these chunks to be processed by the mappers. This is defined by the specification of the input file format. The mapper in Hadoop processes the input file one line at a time, then performs the mapping function, which in our case, searches for the pattern location. The reducer then groups the patterns found with their locations and writes the result to a file. We have kept the default installation configurations and have chosen to use one reducer by setting 'setNumReduceTasks' to 1 so that the aggregated result will be in one file. The output file thus consists of the different patterns along with their locations in the text file.

The Hadoop library contains a RegexMapper<K> class. The implementation of the map function searches for the pattern. Hence, each line is processed separately, and the pattern is searched for within that line. We have based our implementation on the RegexMapper class but have tweaked it to fit our needs. We have also implemented a reducer class that aggregates the multiple patterns found and outputs the patterns with their locations to file.

Testing on Hadoop was performed on a cluster of five virtual machines with the same specifications as those mentioned in Table 4-1. One machine served as the master and slave node concurrently, and the other four machines served as slave nodes.

Execution time using Hadoop is measured from the time the input is read up until the result is written to file. The results are recorded in Table 4-11. The first row indicates the average runtimes in milliseconds and the second row is the percent improvement over the single thread

implementation. The first column corresponds to the single thread implementation. The column in the table labeled Hadoop gives the Hadoop runtime and improvement. We obtained an 89% reduction in runtime. This is due to the fact that Hadoop reads the input one line at a time and performs its map function on each line, which might be shorter than even the substrings in the multithreaded option.

Table 4-11: Average runtimes and percent improvement for Hadoop implementation

	t = 1	**Hadoop**	**Single lines**
Average runtime (ms)	73,199	7,670	20,880
Percent improvement		89.52	71.47

In order to better compare, we have devised another approach that consists of the single threaded option reading the input from file and searching for the pattern one line at a time. This is comparable to the way Hadoop reads input files. The results are in the last column of Table 4-11. This method performed worse than Hadoop due to the fact that Hadoop has a more efficient file system and does reading and writing to files more efficiently. However, this method performed better than our single-threaded method of reading the whole input into a string. Comparing these two methods in Table 4-12, we see that Hadoop outperforms the single line implementation by 63%.

Table 4-12: Runtime improvements

	Single Lines	**Hadoop**
Average runtime (ms)	20,880	7,670
Percent improvement		63.26

However, our multithreaded approach still gave better results. We believe this is because the data that we tested is not big enough to compensate for the expensive writes to disk. This led us to explore another alternative to MapReduce. Spark is a fast general compute engine that can run on top of Yarn and HDFS. In addition to achieving Hadoop's objectives, Spark is able to evolve past these main attributes and solve several of Hadoop's main issues. The discussion on Spark is beyond the scope of this chapter, but will be further discussed in the next one.

4.8 Conclusion

We have presented a tool that detects a user's function call to search for a pattern and seamlessly replaces it with our own distributed MPI-based implementation. We have used bytecode injection and the Open MPI Java bindings to run our optimized code. We have used three different approaches for message passing: point-to-point using multiple sends and receives, collective using the MPI scatter routine, and local in which the data resides on all nodes. Our experiments show a reduction in total execution time for all three approaches, and we have discussed the advantages and disadvantages of selecting one over the other. In addition to that, we have explored another form of distributed communication represented by Hadoop. Because data exchange using Hadoop is based on the filesystem, we have chosen to compare its performance to our multithreaded approach. This allowed us to learn how Hadoop works and what its limitations are.

Chapter 5: Generating Spark Java Code

With the growing shift towards massively parallel distributed systems on one hand, and the increasing importance of transforming data into knowledge in today's data-driven world on the other, Spark has emerged as the tool of choice for efficient big data analysis. However, users still have to learn the complicated Spark API in order to write even a simple application. We present a tool that facilitates this job for the user by automatically generating Spark Java code from minimal user-supplied inputs. Our tool is easy to use, interactive and offers Spark's native Java API performance.

5.1 Introduction

Hadoop is an Apache open source framework for reliable, scalable and distributed computing of large data sets and has been around for over 10 years [73]. Written in Java, the Hadoop framework is made up of the following four modules: Hadoop Common – the Java libraries and utilities required by the other modules to start Hadoop, Hadoop Yarn – the framework for job scheduling and cluster resource management, Hadoop Distributed File System (HDFS), and Hadoop MapReduce – for parallel processing of large data sets.

Since 2012, Hadoop also refers to the set of software frameworks that can be installed along with Hadoop. These include Apache Spark. Spark is a fast general compute engine that can run on top of Yarn and HDFS. It consists of an expressive programming model that is able to support a wide range of applications, including among others, machine learning and graph computation. In addition to achieving Hadoop's objectives, consisting of distributed computing, reliability, and scalability, Spark is able to evolve past these main attributes and solve several of Hadoop's main issues. The main problem with Hadoop's MapReduce model is that it is unable to handle

iterative and multi-pass algorithms that need to reuse a working set of data across parallel operations in an efficient manner. This is because each step consists of either one map phase and/or one reduce phase. First of all, this is restrictive in the sense that a program will need to be converted into such a pattern to be run. Second, if the user wanted to do something a bit complicated, he/she would have to run a series of map-reduce operations, each of which are high latency and cannot start until the previous job has finished execution. Third, and most importantly, the data between each step needs to be stored in HDFS, before being processed by the next step. This is slow because of the replication factor in HDFS and because disk is being accessed. For iterative algorithms, such as machine learning algorithms, this incurs a high latency. In contrast to Hadoop's MapReduce model, Spark allows developers to write complex, multi-pass algorithms in an efficient manner since it supports in-memory data sharing, hence bypassing the expensive need to store to disk at every single map or reduce process. A dataset can be stored in memory, and operations being run on that dataset do not need to fetch it from disk every time it is called. Spark has been shown to provide a 10 fold faster performance time than Hadoop running iterative machine learning jobs [74].

Given the importance of Spark, we have implemented a tool that helps developers write Spark Java code. We have not attempted to implement something similar for Hadoop's MapReduce for the reasons mentioned above. Spark has a Scala API, a python API and a Java API. The Java API is the most complex since first, Spark was written in Scala and to use the Java API, some wrappers were implemented; and second, because Java is a strongly typed language and data types need to be explicitly stated. Our tool consists of a user-friendly, intuitive and interactive graphical user interface with error handling capabilities. It supports the creation of general purpose Spark code (using the core library) and machine learning Spark code (using the

mLlib library). Our tool requires minimal user input and will guide the user throughout the programming process. Nevertheless, some basic programming understanding is still required. The intended users of our tool include first time Spark users who do not wish to learn the API, first time users who would like to learn the API in a simple instructional manner, one-time users who need Spark for a single specific application, non-technical users who are not interested in learning Spark but require to perform some analysis on big data, and of course professional developers who want to quickly prototype a proof of concept.

The rest of the chapter is divided into six sections. Section two consists of a literature review. Sections three and four give a brief description of Spark, of machine learning in general, and of Spark's machine learning library. Section five discusses the work presented. Section six describes the graphical user interface proposed. Section seven concludes the chapter.

5.2 Related Work

According to [75], automatic programming consists of programming at a higher level of abstraction than the one available to the programmer. More specifically, software code generation is the process of generating source code based on a predefined model using a tool such as an IDE which could use templates or wizards and can accept several different forms of user inputs. Several examples are listed, most of which are commercial tools.

The automatic generation of programs is also discussed in [76]. The user tells the computer what to do, and the computer builds a program that does that. The author states that in some situations this is easier to do than write the program by hand, and is specifically useful in complex domains where human beings find it difficult to write programs.

Several tools have been developed that perform software code generation. These tools span various programming languages and various domains. We mention a few that deal with parallel systems. In [62], the authors introduce TypemapGenerator, a tool that automatically generates MPI typemaps from user source code. In [77], the authors have developed STARGATES, a tool that automatically generates source code for finite-differencing simulations for the following parallel platforms: Message Passing Interface (MPI), Threading Building Blocks (TBB) and Compute Unified Device Architecture (CUDA).

Perhaps the most related tools to our project are Weka and MemSQL. Weka [78] is an open source software that contains several machine learning algorithms for data mining tasks. Users can either incorporate Weka API into their Java code or can use the graphical user interface tool. The tool can perform data pre-processing, classification, regression, clustering, association rules, and visualization. Recently, Weka has added support for big data by developing basic distributed wrappers and has incorporated that with both Hadoop and Spark. The major disadvantage that developers complain about with Weka is that it primarily uses a specific format for an input file called Attribute Relation File Format (ARFF). ARFF files contain several information about a machine learning problem. Weka uses this format to create its header data. Users will have to rewrite their input files to fit this format. The objective of Weka is to offer a solution to writing and designing machine learning algorithms, therefore its applications are restricted to that domain.

MemSQL [79] is a database platform for real-time analysis. It is available in a free community version. MemSQL Streamliner uses Spark to manage real-time data pipelines. It provides a rich graphical user interface for the ETL (Extract, Transform, Load) process. However, it is strictly database-oriented.

5.3 Spark

Spark is a parallel open source big data framework. It was originally developed at UC Berkeley's AMPLab in 2009, was open sourced in 2010, and was later licensed as an Apache project. Since then, Spark has gained popularity and it has been reported that as of early 2015, more than 500 companies are using Spark in production [80].

The designers of Spark realized the importance of existing applications (such as MapReduce) in implementing large scale data intensive applications on commodity clusters. However, the prevailing systems were unable to handle the reuse of a working set of data across parallel operations. Hence, Spark was developed. Spark can support iterative algorithms (such as machine learning algorithms) and still retain the scalability and fault tolerance of MapReduce. In addition to that, Spark has been shown to provide a 10 fold faster performance time than Hadoop (the prevailing tool at that time) running iterative machine learning jobs [74]. What follows is a brief description of Spark.

5.3.2 Spark Ecosystem

Spark is made up of the following components/libraries:

- Spark Core API: This is the underlying execution engine upon which all other functionalities are built. It provides distributed task dispatching, scheduling, in-memory computing, and basic IO functionalities.
- Spark MLlib: This library is used to implement machine learning algorithms. It will be further discussed in a following section.

- Spark Streaming: This library can be used to perform analytics of real time streaming data by processing mini-batches of data.
- Spark SQL: This library provides support for structured and semi-structured data. It allows running SQL-like queries on Spark data.
- Spark GraphX: GraphX is a distributed graph-processing framework.

5.3.3 Underlying Framework

Spark requires two main components to run. The first is a cluster manager. Spark can support a standalone native Spark cluster, a Hadoop Yarn management system, SIMR (Spark Inside MapReduce), or Apache Mesos. The second component is a distributed storage system. Spark can run with Hadoop Distributed File System (HDFS), HBase, Hive, MapR File System (MapR-FS), Cassandra, Open Stack Swift, Amazon S3, Kudu, or even with a customized system. In addition to a cluster, Spark can run on the local filesystem (a single machine) as a pseudo-distributed local mode. The purpose for this is for development and testing.

3.4 Spark Building Blocks

A Spark application consists of a ‘driver program’ that runs a main function and that runs the parallel code on a cluster. This is achievable due to an abstraction that Spark developers introduced called a distributed resilient dataset (RDD) [81].

An RDD is a read-only parallel collection of objects that can be partitioned across a cluster, and that can be rebuilt and recovered if one of the partitions is lost. There are basically two ways of transforming data into an RDD. It can be created either by reading an input file (based on the filesystems mentioned in the previous section) and transforming its contents, or by calling a

parallelize function on an already existing dataset. RDDs that are to be reused (such as datasets used in iterative programming) should be persisted. This means that the RDD will be cached in memory. This allows operations working on that dataset to be faster. In addition to that, persisted RDDs can be stored in memory only, in memory and on disk (if dataset does not fit in memory, store the remaining partition on disk), on disk only, and several other options consisting of serialization or of data replication [80].

Once an RDD is created, there exist two operations that can be performed on it: either a transformation, or an action. A transformation creates a new RDD by applying a certain operation on the source RDD. For example, a `mapToPair` transformation creates a new (key, value) paired dataset passing each element of the source dataset through a function that generates a tuple (Scala representation that is used in the Java implementation) representative of the operation. Transformations are defined as 'lazy'. This means that the results are not computed when a transformation is defined. They are instead 'remembered' and will be applied when they are actually needed. Unless an action is called upon a dataset, that transformation will not be implemented.

An action is a function that performs a computation on an RDD and returns a value to the driver program. They are necessary in that they allow the transformations to actually take place. If the application does not contain an action, albeit a dummy one, the transformations will not execute. For example, `reduce` is an action that aggregates the RDD elements into a single variable based on an associative reduction function.

5.3.5 Usage and Advantages

Spark is written in Scala and runs on a Java Virtual Machine (JVM). It provides built-in APIs in Java, Scala and Python.

It encompasses the functionalities of several separate tools into one workload system, making it easy for the user to manage different systems and to maintain separate tools. It does so by extending the MapReduce model to include broader types of computations, such as batch applications, iterative algorithms, interactive queries and stream processing.

The main feature of Spark is fast processing. This is achievable due to the fact that storage and processing can be done in-memory. Therefore, performance can be orders of magnitude faster than other big data technologies, such as Hadoop, which opts to store all intermediate data in disk instead. Spark holds intermediate results in memory rather than writing to disk. This reduces the number of expensive read/write operations. This is very useful with iterative algorithms, where the same dataset is used multiple times. Spark will attempt to store as much data in memory as possible. When data is too large to fit in memory, it will be spilled to disk. When this happens, performance can be degraded.

Other advantages of Spark include:

- Supporting functions other than simple Map and Reduce
- Lazy evaluation of big data. This helps with the optimization of the overall data processing.
- Library support for SQL queries, machine learning, streaming, and graph algorithms.
- Interactive shell for Scala and Python.

5.4 Machine Learning

Recently, and mostly due to the availability of big data based on the increase in social media usage and the heavy reliance on web-related applications, machine learning has been driven back to the front as a means of understanding all this data and transforming it to knowledge. Another contributor to the resurfacing of machine learning is the available processing power, the speed of processing and the low-cost of programming units.

Machine learning is generally divided into two categories: supervised learning and unsupervised learning. Supervised learning consists of training the program on a pre-defined set of training examples, which then facilitate its ability to predict the outcome when given new data. Some example problems are classification and regression. Unsupervised learning consists of giving the program a bunch of data wherein the program must find patterns and relationships, and group the data accordingly. Some example problems are clustering, dimensionality reduction and association rule learning.

Machine learning algorithms are of an iterative nature and have not been widely implemented on big data frameworks (such as Hadoop) previously, since such frameworks were unable to efficiently support iterations. However, this has changed with Spark.

5.4.1 Machine Learning with Spark

In 2007, [82] published a paper on which machine learning algorithms could be applied to a MapReduce framework. Since then developers at Hadoop set forth to try and implement these algorithms as a Hadoop based library called Mahout. They were successful in building the library; however, had issues with performance.

According to Spark developer Zaharia, Spark emerged from observing the problems that users had with the MapReduce model and trying to improve it [83]. More specifically, Zaharia stated that machine learning algorithms were the main bottleneck with Hadoop. Hadoop users at UC Berkeley were running machine learning algorithms on big data, and were unsatisfied with the slow runtime results, since their algorithms were performing several iterative scans over their data. Hence, Zaharia and his lab mates designed Spark, from what started as a solution to machine learning algorithms to covering other use cases beyond machine learning.

MLlib is Spark's machine learning library that runs on top of Spark core [80]. MLlib is as much as nine times faster than the disk-based Apache Mahout (Hadoop's machine learning library). Even Mahout now has a Spark interface [84]. MLlib consists of learning algorithms and utilities, such as classification, regression, clustering, collaborative filtering, dimensionality reduction, and optimization primitives.

5.5 Implementation

This section will describe our implementation, more specifically, our design, our workflow, some of the problems we encountered, how they were solved and our limitations.

The most widely used example for map-reduce applications, whether it be Hadoop or Spark, is the word count example. The user provides a file and the program outputs the distinct number of words along with how many times they occur in the file. We will use this example as a basis for our discussion.

In order to use our tool, we only require two things from the user. First, we require the user to provide us with an input file that contains the original dataset. As previously mentioned, there are two ways in Spark to provide datasets: either loading it from a file, or applying a

parallelization transformation on a previously defined dataset. We have chosen the option to load the dataset from file for the simple reason that we do not want the user to write code (by generating the dataset) and then try to incorporate that with the code that we generate. We believe that would be both cluttered and error prone. The second thing we require from the user is to provide us with the tokenization granularity for parsing. Parsing is usually considered a tedious task; therefore we have chosen to abstract it from the user. Instead, we handle it internally ourselves, and the user does not need to know which Spark-specific operations to use. These are our only two main requirements. Other than that, the user can use our tool freely.

To demonstrate our claim, we go back to the word count example. The code in Figure 5-1shows the loading and parsing code that we generate based on the user providing us with the name of the text file, and that parsing granularity is strings separated by spaces. Therefore, the parsing (which in this case uses a `flatMap` transformation, but could use another transformation based on granularity) produces a new data set, labeled words, that contains the individual words.

```
…
JavaRDD<String> data = jsc.textFile("text.txt");
JavaRDD<String> words = data.flatMap(
    new FlatMapFunction<String, String>() {
    @Override
    public Iterable<String> call(String input) {
        return Arrays.asList(input.split(" "));
        }}).cache();
```

Figure 5-1: Loading and parsing code example for word count

The next part would be the code that performs the word count. One way to do that is based on the Spark documentation implemented as shown in Figure 5-2. The first transformation, which is a `mapToPair` transformation, creates a new dataset that contains a pair of the words and assigns a count of one to each. The second transformation is a `reduceByKey`. It looks for distinct keys and adds their values, hence obtaining the word count.

```
…
JavaPairRDD<String, Integer> ones = words.mapToPair(
    new PairFunction<String, String, Integer>() {
        @Override
        public Tuple2<String, Integer> call(String s) {
            return new Tuple2<String, Integer> (s, 1);
            }});
JavaPairRDD<String, Integer> count = ones.reduceByKey(
    new Function2<Integer, Integer, Integer>() {
        @Override
        public Integer call(Integer i1, Integer i2) {
            return i1 + i2;
            }});
…
```

Figure 5-2: Word count code based on Spark implementation from documentation

However, this is not the only way to implement word count. Using our tool, we have implemented it using two other methods: the first performs a `countByValue` action on the 'words' dataset, and the other performs a `countByKey` action on the 'ones' dataset as shown in Figure 5-3. This requires minimal user input. It also serves to show that our tool has limited

restrictions, and can support different implementations depending on the individual users' lines of thought.

```
//Alternative Method 1

Map<String, Long> count = words.countByValue();

//Alternative Method 2

Map<String, Object> count = ones.countByKey();
```

Figure 5-3: Alternative implementations for word count

Table 5-1 represents our currently supported operations.

Table 5-1: Currently supported operations

	Supported operations
Transformations	*cogroup, distinct, filter, flatMap, groupBy, groupByKey, intersection, join, leftOuterJoin, rightOuterJoin, fullOuterJoin, keys, map, mapToPair, reduceByKey, sortByKey, subtractByKey, union, values*
Actions	*collect, collectAsMap, count, countByKey, countByValue, first, foreach, id, isEmpty, name, reduce, take, top, saveAsTextFile, saveAsSequenceFile, saveAsObjectFile*

Basically, our tool works similarly to the Weka interface: the user interface will gather information about the operations the user intends to perform and a code generator will create the necessary wrapper representations, which will make up the necessary constructs to produce the Spark code. Our approach can be divided into two main phases. Phase one consists of our internal representation, and translates into phase two which consists of the code generation, as shown in Figure 5-4.

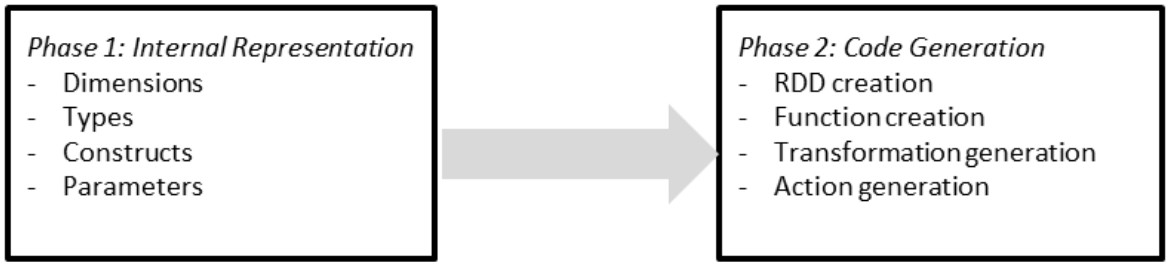


Figure 5-4: Program phases

5.5.1 Internal Representation

There were several issues we needed to deal with in order to generate correct code that adhered to the types and dependencies. More specifically, we mention how we handled dimensions, types, constructs and parameters.

5.5.1.1 Dimensions

Most Spark transformations and actions run on datasets of any type. However, there are a set of operations that run on datasets that are of `<key, value>` pairs, such as for example, grouping by a key. These `<K, V>` pairs are represented using the Scala Tuple2 class and the corresponding dataset is represented by the JavaPairRDD class. In our implementation, we had to categorize these operations and set the dimensions of their datasets accordingly. A distinction is made whether transformations create a pair dataset or operate on one. Some transformations can operate on either a pair dataset or a single type dataset. For these operations that do and that also happen to require two datasets such as a `union` operation, we have to make sure that both datasets have the same dimension.

5.5.1.2 Types

We had initially used the Spark Java API documentation [85] to obtain the various modifiers and return types for the operations. However, we have added on to that based on dataset dependencies to obtain the correct specific types. In order to do that, we generate a list of RDDs that is populated based on the dependencies created by the transformations. For example, in the documentation it states that the return type of a `filter` transformation is an RDD of any type `<T>`. In our implementation, we take this further into declaring what `T` actually represents (Integer, String, Tuple …) based on the source dataset. So for example, if a `filter` transformation is called on a dataset of type Integer, it will create a new filtered dataset of type Integer. For the same `filter` transformation, if the source dataset is a `<K, V>` pair, then the new dataset will also be a `<K, V>` pair. This relationship between the input types and the return types is different for each transformation and depends on the transformation purpose and description. A list of some transformations and their return types based on their source datasets can be found in Table 5-2. We also deduce types for actions similarly. The user does not need to know what the type of the variable is.

Table 5-2: Transformation input and output types

Transformation	Input type	Return type
cogroup	dataset1 (K,V) & dataset2 (K,W)	(K, Tuple2<Iterable<V>, Iterable<W>>)
distinct	- input_type - (K,V)	- input_type - (K,V)
filter	- input_type - (K,V)	- input_type - (K,V)
flatMap	any input_type	any input_type
groupBy	(K,V)	(any input_type, Iterable<K>)
groupByKey	(K,V)	(K, <Iterable<V>)
intersection	- input_type - dataset1 & dataset2 (K,V)	- input_type - (K,V)
join	dataset1 (K,V) & dataset2 (K,W)	(K, Tuple2<V,W>)
leftOuterJoin	dataset1 (K,V) & dataset2 (K,W)	(K, Tuple2<V, Optional<W>>)
rightOuterJoin	dataset1 (K,V) & dataset2 (K,W)	(K, Tuple2<Optional<V>,W>)
fullOuterJoin	dataset1 (K,V) & dataset2 (K,W)	(K, Tuple2<Optional<V>, Optional<W>>)
keys	(K,V)	K
map	any input_type	any input_type
mapToPair	any input_type	any (K,V) pair types
reduceByKey	(K,V)	(K,V)
sortByKey	(K,V)	(K,V)

subtractByKey	dataset1 & dataset2 (K,V)	(K,V)
union	- input_type - dataset1 & dataset2 (K,V)	- input_type - (K,V)
values	(K,V)	V

5.5.1.3 Constructs

Most transformations and a few actions require a function that defines what the operation should do. For example, to obtain a list of employees with salaries greater than $100,000, one might perform a `filter` transformation on an `<employee_id, salary>` dataset to produce that new dataset. This description is specified in a function that is passed as a parameter to the transformation. In this case, the function is a `Boolean` function, and will return the entries where salary >100,000. Several other types of functions are used in Spark, such as `Function` (which takes one input and generates an output), `Function2` (which takes two inputs instead of one and generates an output), `PairFunction` (which takes one input and generates an output with the types of the keys and values), `VoidFunction` (which takes one input but does not return an output)…

The creation of these functions is confusing to the user; therefore, we have tried to abstract that by inferring and detecting the types from code and variable dependencies whenever possible. For example, in the code for the ‘ones’ dataset shown in Figure 5-2, the user does not need to know which function to use, how its parameters are described, the types of its input dataset, or the call method it overrides. All the user needs to do is supply us with the information in Table 5-3.

Table 5-3: User information needed

Out Type	Operation
String	s
Integer	1

Another issue we had to handle was that in some cases users might lose track of types pertaining to which datasets, and might specify wrong types to transformations or actions whose types we are able to either know before-hand or based on the source dataset. Therefore, we implemented an error handling mechanism to automatically infer these types when possible and present them to the user. For example, the `filter` transformation requires a Boolean function; therefore we preload that value in our code as a precautionary measure and will pass it to the function even if the user mistakenly specifies it to be an Integer for example. This is similarly done for all the transformations and actions where the inputs to the functions are determined by the source dataset or by the operation itself, such as for instance `reduce` and `reduceByKey`.

Another form of error handling that we deal with and is very important for correct program execution is making sure that an action is eventually called in the application. As previously stated, transformations are lazy and require actions for them to be executed. Thus, if a program does not at some point create an action that chains the transformations, then nothing will happen. To catch this potential slip, we make sure to implement a ‘collect’ action (which simply fetches the RDD as an array to the driver program) in case none is specified by the user.

5.5.1.4 Parameters

Transformations and some actions require input parameters for execution. We mention specifically those transformations that operate on two datasets, such as joins, unions, intersection…These transformations are different than those that operate on solely one dataset in that special care needs to be given to how their types are assigned. For example, to perform a join on two datasets, first they both have to be `<K, V>` pair datasets. In addition to that, both datasets should have the same type of key; otherwise a join cannot be performed.

5.5.2 Code Generation

Our tool is written in Java and consists of four major classes: SparkRDD, SparkTransformation, SparkFunction, and SparkAction. The SparkRDD class creates the base RDD objects, which are the datasets. It sets their dimension and their input types. In what follows we will use the terms RDD and dataset interchangeably. A SparkTransformation creates a SparkRDD object based on a transformation and a source dataset. To achieve this, it implements a base class that extends two other template files: one that implements a parsing transformation, and one that implements a transformation that requires two input datasets. A SparkAction creates a variable that is the result of an operation on a dataset. Some transformations and actions require a function as a parameter, and that is represented by the SparkFunction class. We show a simplified class diagram in Figure 5-5.

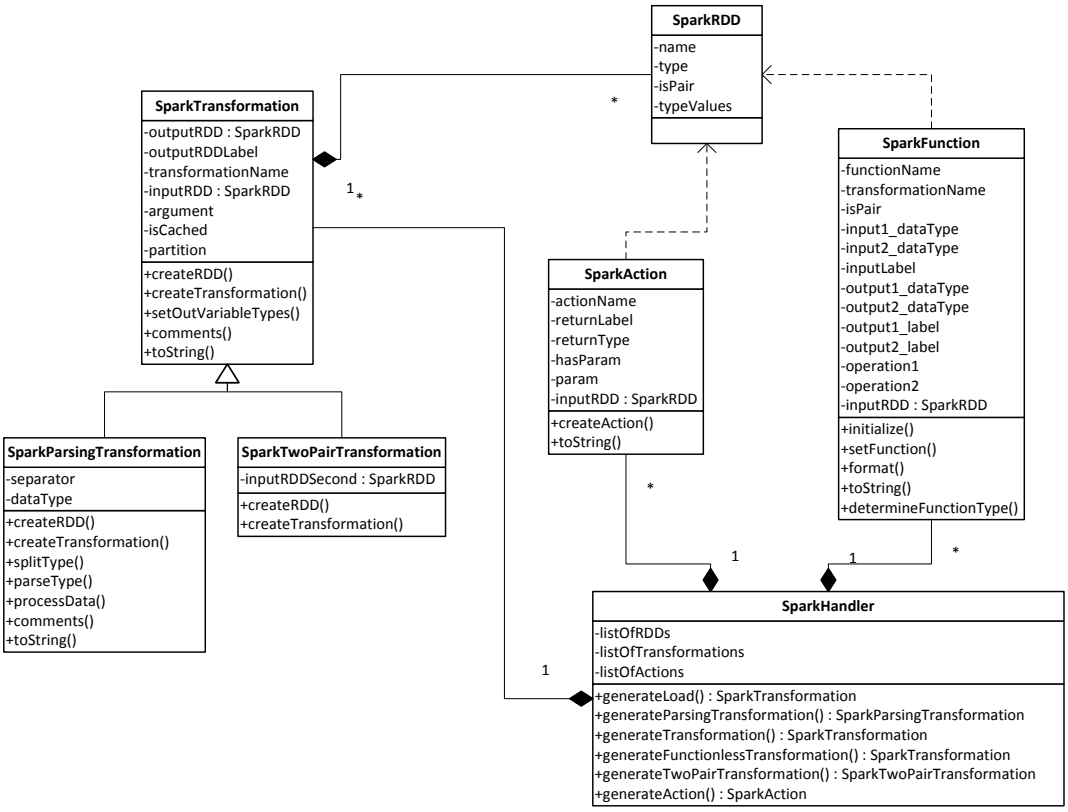


Figure 5-5: Spark tool class diagram

Our workflow consists of a series of operations that are performed based on a user input file representing the initial data. The entry point of our tool is the user-supplied data file. A typical flow of execution would start off by loading a file from storage (HDFS for e.g.) into the application. The data now is represented in a dataset wherein each entry is populated with a single line from the data file. The next step consists of parsing the data. Since Spark, similar to MapReduce, originally splits the data from a file based on a new line, in most cases, we will need to parse the data before using it. Now that the data is parsed, it can be operated on by using either transformation calls or action calls. A transformation will require user input to detect dependencies, form a function or supply a parameter, and will generate a new dataset that will be added to the dataset repository. An action will require user input to detect a source dataset, form a function in some cases or supply a parameter, and will generate a variable (if the action does

not have a void return type) that will be added to the variable repository. There is no limit or order (except dependency) on creating transformations and actions, though transformations are typically chained and actions are called at the end. Our workflow can be summarized in Figure 5-6.

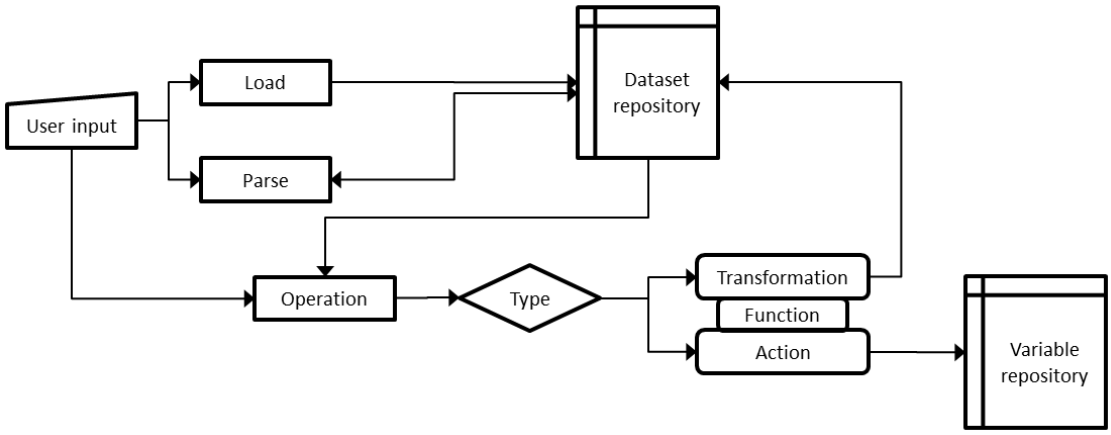


Figure 5-6: Spark tool workflow

5.5.3 Machine Learning Specific

The reason that machine learning cannot be generalized as other map-reduce applications is due to the fact that machine learning in Spark uses a specific library (mLlib) with its own set of functions and constructs. In some cases it can, but will not be as efficient as using the library, and the documentation advises against it.

5.5.3.1 Classification

Typically, classification problems follow the following procedure: train - validate - test. The training and validating phases are done on labeled data. Usually, training data is split (70%

training and 30% validating)[5]. So the user gets to train the data in order to create a model; then validates the model using the other part of the labeled data. This is all done before the actual testing (where typically labels are unknown). Validation also serves to justify whether the model is a good match for the given data set. After generating the model, the testing data is applied to obtain the prediction results.

The generated Spark code will produce the following:

- Load the training set
- Parse the data into features and labels
- Calculate the number of features and labels
- Split the data into training and validation sets
- Train the model
- Predict and validate
- Calculate prediction accuracy
- Load the test data
- Parse the test data
- Predict the output
- Return result to user with a percentage accuracy

Currently, the models in Table 5-4 along with their corresponding default values are supported[6]. We have provided most of the classification models that Spark implements and some of the regression models. We believe that for beginners to machine learning, this is quite a comprehensive list and should cover most of the cases.

[5] We have not mentioned cross-validation since Spark's mLlib does not support it.
[6] The default values are taken from the examples in the Spark documentation.

Table 5-4: Supported classification models and their default parameters

Model	Default values
Naive Bayes	lambda = 1.0
LinearRegressionWithSGD	number of iterations = 100
SVMWithSGD	number of iterations = 100
LogisticRegressionWithLBFGS	
DecisionTree	- impurity = "gini" - maximum depth = 5 - maximum number of bins = 32
RandomForest	- impurity = "gini" - maximum depth = 5 - maximum number of bins = 32 - number of trees = 3
GradientBoostedTrees	- boosting strategy number of iterations = 3 - boosting strategy maximum tree depth = 5

5.5.3.2 Clustering

The clustering problem is slightly less complicated to define than the classification problem. The generated Spark code will produce the following:

- Load the training set
- Parse the data
- Cluster the data into 'k' classes

- Evaluate clustering by computing within set sum of squared errors
- Return the cluster centers to the user

Currently, only K-Means is supported. It is however the most popular algorithm supported by Spark for clustering.

5.6 Graphical User Interface

In order to get the appropriate feedback from the user, and to be able to better display the steps needed to generate the Spark code, an interactive graphical interface was developed. The tool automatically generates Spark code based on minimal user-input, and can be divided into two main parts: a part that generates Spark code for machine learning algorithms, and another part that generates Spark code for general purpose applications. The two parts share a common functionality, the Spark-specific parameters. This section will first discuss the common constructs for both parts, and then delve into the specifics of generating machine learning code and generating general purpose code.

5.6.1 Spark Parameters

Some Spark parameters will need to be provided by the user. These values are needed in order to run the generated code in a Spark environment.

- Master: This is the master configuration running Spark. If run on a cluster, then `master` is a Spark, Mesos or YARN cluster URL. If run locally, then `master` can be set to `local`. Additionally, `master` can be set to `local[n]` where `n` is the number of threads.

- Partition: This is the number of partitions that Spark uses to distribute the dataset. Spark will run one task for each partition of the cluster. Typically, the user will want 2-4 partitions for each CPU in the cluster. Normally, Spark tries to set the number of partitions automatically based on the cluster. To apply this default setting in the tool, the user can set the value for partition as the `default` keyword. Otherwise, it can be manually set it by specifying the required number of partitions.

5.6.2 Spark Machine Learning Code Generation

Our tool explores the problems of classification and clustering. In addition to setting the Spark parameters mentioned above, the user will need to specify some application-specific variables. A snapshot of the machine learning part of our tool is shown in Figure 5-7.

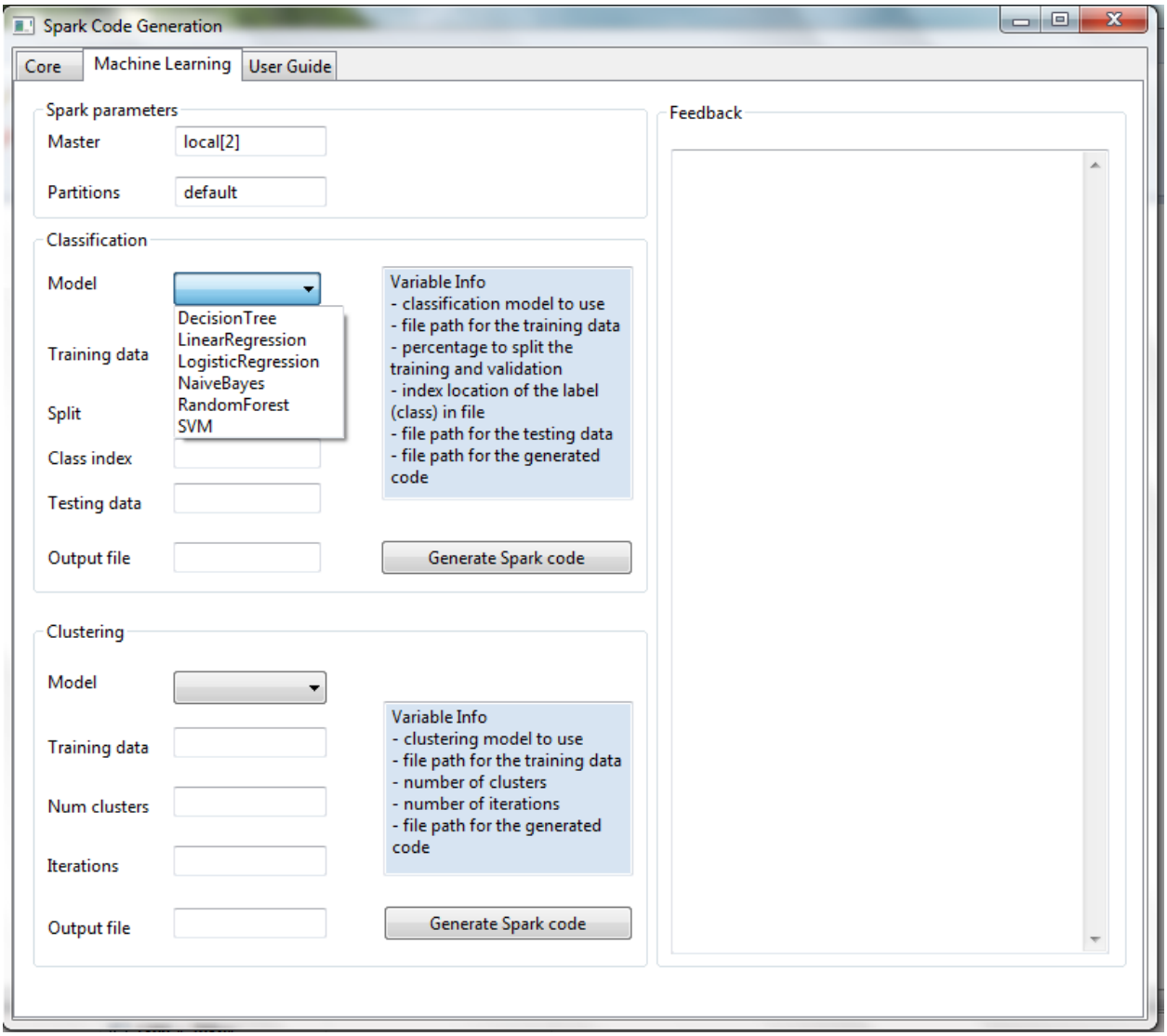


Figure 5-7: Tool snapshot - machine learning

5.6.2.1 Classification Variables and Models

In order for the code to be generated correctly, the following variables will need to be specified:

- file path for the training data
- percentage upon which to split the training and validation data in the input file
- model type: which classification model to use

- class index: location of the label in the file
- file path for the testing data

5.6.2.2 Clustering Variables and Models

The following variables will need to be specified:

- file path for the training data
- model type: which clustering model to use
- k: number of clusters
- number of iterations to run the algorithm

5.6.3 Spark General-Purpose Code Generation

This part of the tool generates general purpose Spark code based on user input and interaction. It assumes that the user is familiar with at least some sort of programming language and is capable of correctly writing basic code statements.

The tool can be divided into five main sections, which constitute the design flow: spark parameters, file handlers, datasets and variables, operations consisting of transformations and actions, and feedback. A snapshot of the general-purpose part of our tool is shown in Figure 5-8. Spark parameters will not be discussed since they are similar to those mentioned in the previous section.

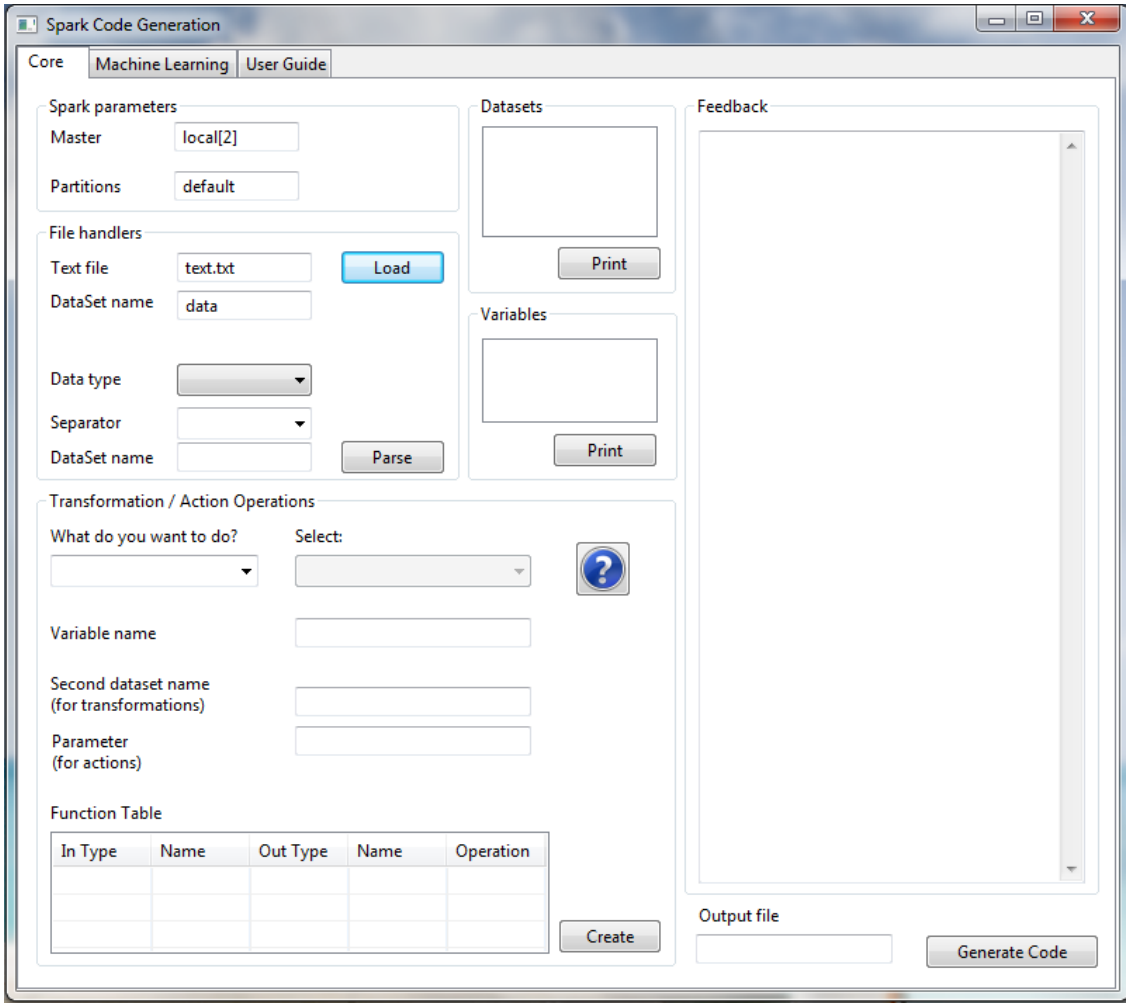


Figure 5-8: Tool snapshot – general purpose

5.6.3.1 File Handlers

The first level of entry to the program is the input file. It contains the original user data that requires modification and/or analysis. The user needs to specify the file path and the name of the dataset into which to load the data. Clicking on ‘Load’ will create the code to load the data file into a newly created dataset that will be displayed to the user in the "Datasets" section.

The second part consists of parsing the dataset. The user needs to specify the data type, the separator characters upon which the data is split, and the name of the dataset to parse the data

into. Additionally, the user needs to specify on which dataset to perform the parsing. Typically, it is the one loaded from file. The user can just click on its name from the ‘Datasets’ section. After clicking on ‘Parse’, the parsing code is generated and the new dataset is added to the list. The snapshot from Figure 5-9 shows how the loading and parsing of the word count example from Figure 5-1 is constructed.

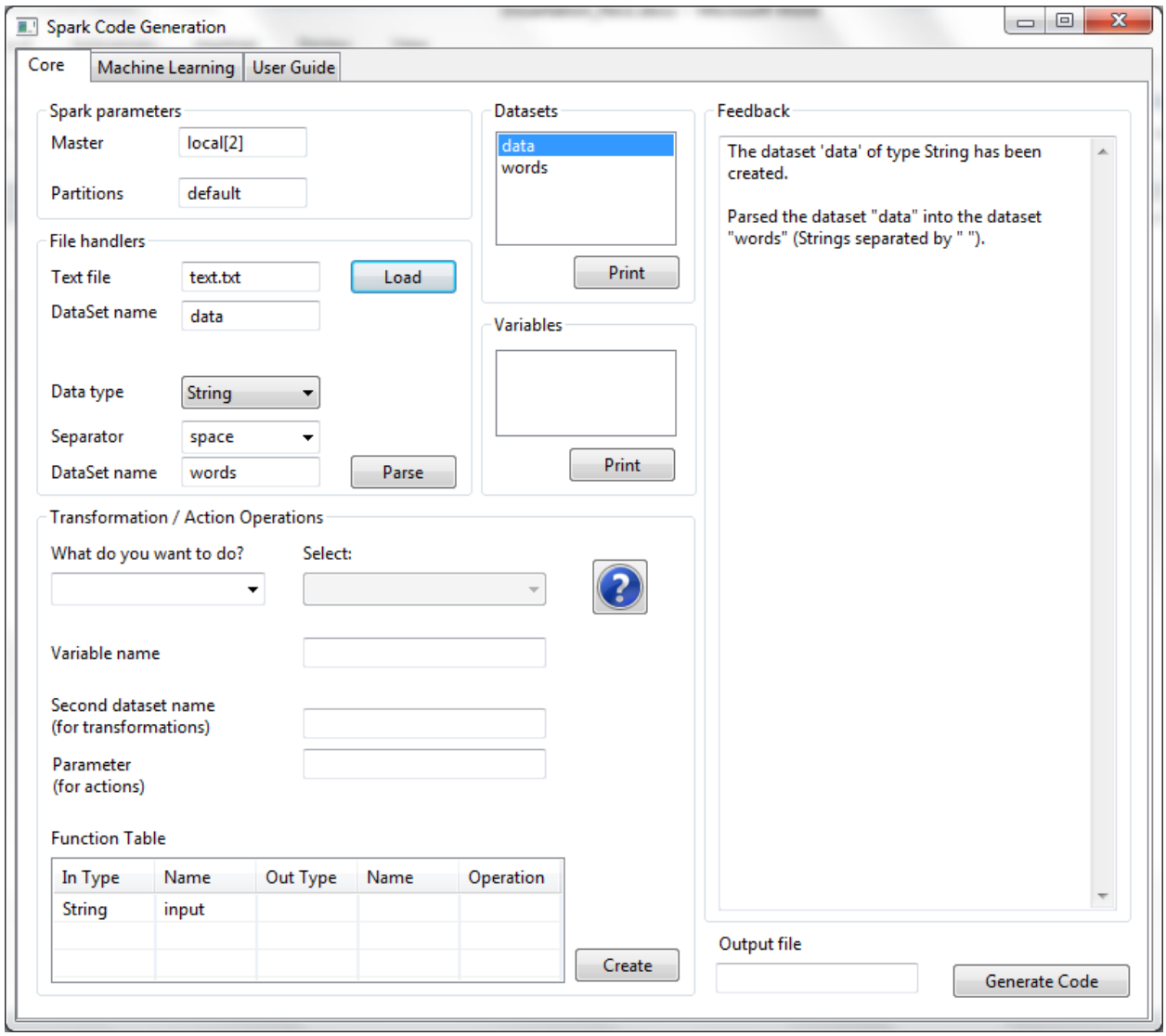


Figure 5-9: Word count example: loading and parsing

5.6.3.2 Datasets and Variables

This section consists of two lists that are dynamically populated:

- Datasets: once a new dataset is created, it is added to the list. The user should click on an item from the list to specify the source dataset to a transformation or an action. Information about that dataset will also be displayed.
- Variables: once a new variable is created by an action, it is added to a list.

5.6.3.3 Operations

This section handles the creation of transformations and actions. It also includes support and help for users who are not sure about which transformation or action to use, what parameters are required and how to use a certain operation.

The section starts off asking the user what he/she wants to do. The user can select an operation (either a transformation or an action) from a drop-down menu. Once that is selected, the user is prompted to select which transformation or which action he/she will want to perform.

Some instructions as to what parameters this operation needs will be displayed in the 'Feedback' section. Now the user knows exactly what he/she will need to input. Part of the instructions displayed in the 'Feedback' section will ask the user to make sure to select a source data set from the 'Dataset' section. Doing so will provide some information about the source dataset that the user might find helpful in order to correctly create the operation. We show this in Figure 5-10.

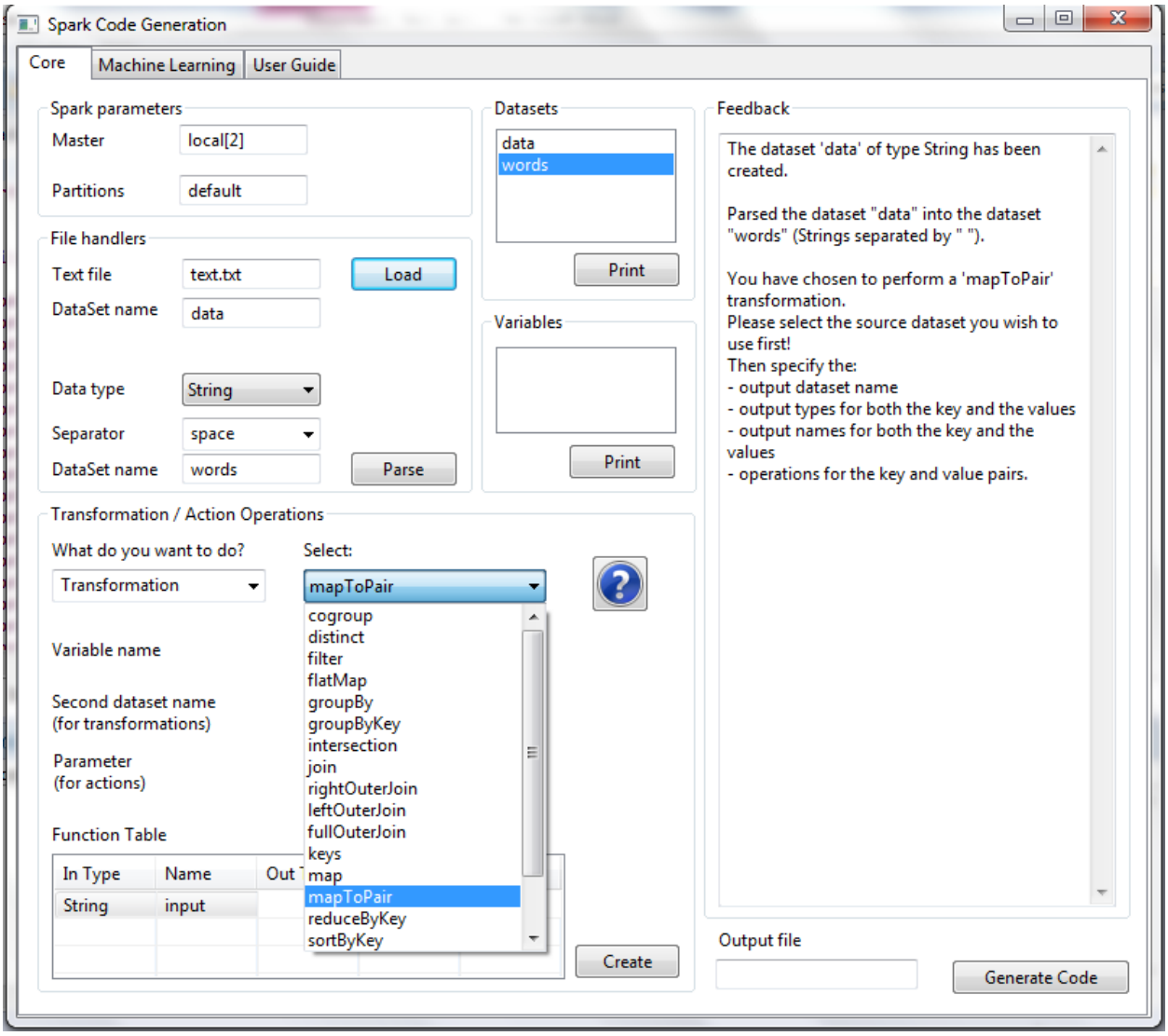


Figure 5-10: Example of operations and corresponding feedback

In addition to that, there is also a 'Help' button. Clicking it will display the definition of a transformation or action, some helpful tips and an example of how to use it.

In the case where a transformation has been selected, a table that the user might need to fill in is dynamically generated depending on the selected transformation. In addition to that, there is a field to enter the name of the other dataset when the transformation requires two datasets. For example, if the user chooses a 'map' transformation and chooses to call the generated dataset

'map_out', and has selected the source dataset 'testSet', which is for example of type String, then a table is displayed as such:

In Type	In Name	Out Type	Out Name	Operation
String	*input*			

Now assume that the user wants to perform a map that returns the length of a string. Then he/she will need to fill out the fields of the table as such:

In Type	In Name	Out Type	Out Name	Operation
String	*input*	Integer	len	input.length()

This will generate the code in Figure 5-11:

```
JavaRDD<Integer> map_out = testSet.map(new Function<String, Integer>()
            @Override
            public Integer call(String input) {
                   Integer len = input.length();
                   return len;
            }
      });
```

Figure 5-11: Map example code

In case an action is selected, the user must select a variable name for the action; that is, what the user wants the operation to be identified as. Some actions require a parameter, and the user is instructed to enter a value for that in the 'Feedback' section. Once all inputs are specified, the user can click the 'Create' button, and the code will be generated, and the variable name will be

added to the list. Other actions (such as `reduce`) will require a function to be filled in the table. Note that some actions (such as `saveToFile` or `foreach`) are void, and hence do not return a variable. Therefore, no variable name will be displayed in the 'Variable' section if any of these actions are chosen.

Once the user clicks on the 'Create' button, the code for creating a dataset or a variable is generated and the new dataset or variable is added to the corresponding list. We show how the word count `reduceByKey` function from Figure 5-2 is constructed in Figure 5-12.

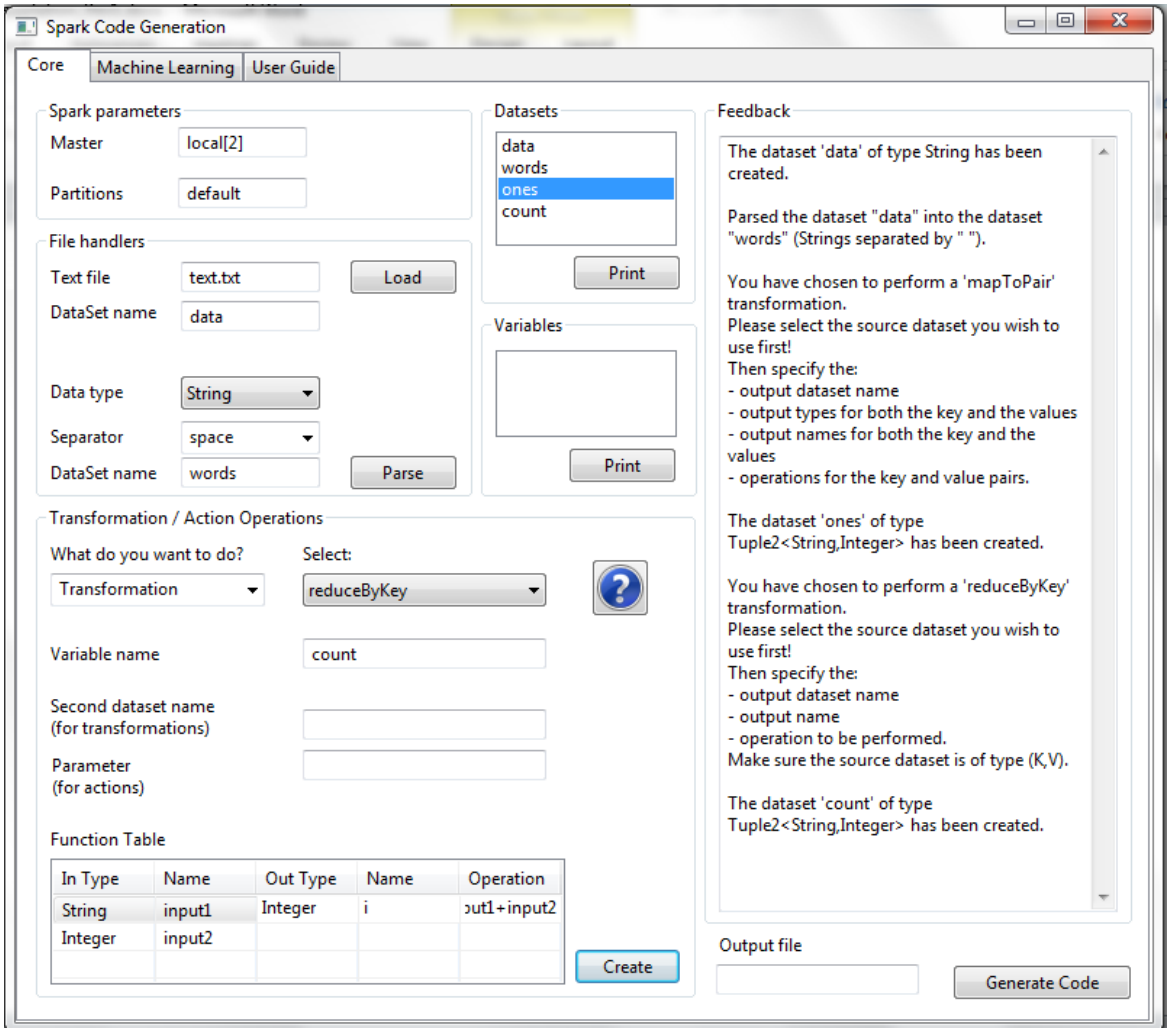


Figure 5-12: Word count example - reduceByKey transformation

5.6.3.4 Feedback

This section serves two purposes. First, it provides instructions to the user. It urges the user to specify the correct fields for his/her chosen operation. When the user selects a transformation or an action from the drop-down menu, this section will instruct the user as to what else is needed as input.

Second, it provides feedback to the user by displaying what has been created. When a transformation is created, the dataset name and types will be displayed to the user. When an action is created, the corresponding variable and its type are displayed to the user.

5.6.3.5 Error Handling

Our tool provides ways for handling potential compile-time errors that might be generated based on bad user input. In addition to the caught pitfalls that we have previously mentioned, we account for the following cases.

One issue that we think might occur is that the user chooses to perform a transformation or action that requires a source dataset to consist of `<key, value>` pairs on a dataset that does not represent a pair. Syntactically, that would be correct; however, this will generate an error while compiling. Therefore, if this happens, a warning message is given to the user to rectify the error.

Other issues include the user not entering all the appropriate information, and that is handled by the interface.

5.7 Conclusion

We have presented a tool that facilitates the generation of Spark code for the core and the mLlib libraries by providing a user-friendly, intuitive and interactive graphical user interface. While it does not support all the transformations and actions API calls since it is a proof of concept and not a commercial tool, we believe it covers most of what we believe are the most widely used and needed operations, especially for our intended users. We have extensively tested our tool by running the generated code under a Spark environment and have verified the correctness of our results. In the future, we hope to include more of the operations that have not been yet handled. Also, we hope to add more features for debugging purposes.

Chapter 6: Conclusion

6.1 Summary

Parallel computing has been used to model many difficult problems in science and engineering. Such fields include computer science, mechanical engineering, circuit design, geology, seismology, molecular sciences, genetics, physics…As well, several commercial applications such as in data mining, web-based services, medical imaging and diagnosis, financial and economic modeling, virtual reality, among many others, require processing of large amounts of data in a usually complex manner. The development of faster and more efficient parallel systems has stemmed from this need to satisfy the ever-increasing and growing body of data. In addition to being better suited to model real world systems compared to serial computing by being able to solve larger and more complex problems, parallel computing has the following advantages: (i) it can save both money and time since tasks will execute faster and concurrently, (ii) parallel systems can be built from cheap, commodity hardware, (iii) it can take advantage of non-local resources such as the internet or a wide area network.

Parallelism though comes at a price. It is not trivial to write parallel code. The process is tedious, complex, and error prone. Programmers have to worry about the environment, communication, concurrency, synchronization and efficiency. It is also somewhat counter intuitive to someone who is used to serial programming.

In this dissertation, we have presented methods to facilitate the parallelization of high performance code in order to alleviate the burdens associated with parallel programming. Making parallel computing easier is not a new endeavor. Several works have been successful in achieving this. However, it has always been left to the users to either locate potential code

fragments that could be parallelized, or specify pragmas, or learn new parallel and complicated APIs. In contrast, our approach does not require users to perform any of the mentioned tasks. Our method automatically detects a user's single threaded function call to search for a pattern using Java's standard regular expression library, and replaces it with our own data parallel implementation using Java bytecode injection. Our approach facilitates parallel processing on different platforms consisting of shared memory systems (using multithreading and NVIDIA GPUs) and distributed systems (using MPI and Hadoop). Our implementation produces correct results, is transparent to users, and reduces execution time. In addition to that, we have presented a tool that automatically generates Spark Java code from minimal user-supplied inputs. Using our tool, Spark users do not need to learn the complicated Spark API in order to write applications. Our tool is easy to use, interactive and offers Spark's native Java API performance.

In Chapter 1, we provided a summary on parallel computing, introduced our proposal, and stated our objectives and development process.

In Chapter 2, we further elaborated on the problem by discussing pattern matching and the motivation behind choosing it as an application to validate our work. We also discussed Java bytecode and how injection can be done using the ASM library. Then, we showed how our method works in a multithreaded environment, and reported the results we obtained, which show a somewhat logarithmic decrease in execution time with respect to the number of threads.

In Chapter 3, we discussed how our method works in a GPGPU environment. We surveyed the previous work done in this area and showed how JNI was used to offload Java programming on NVIDIA GPUs that only support CUDA. We used an external library as opposed to the Java

standard library to perform pattern matching on the GPU. Our results show a substantial decrease in execution times.

In Chapter 4, we implemented our method in a distributed environment using MPI. We used the Java bindings for the Open MPI implementation. We tested our method on several combinations of processes and virtual machines, and as predicted, our results consisted of decreased execution times. We also introduced Hadoop since it has been widely used recently in big data computing for it provides better storage and data management and computation speedups. We compared a Hadoop implementation to our multithreaded approach. However, we did not attempt a Hadoop implementation for our system because of the way data is shuffled around in HDFS.

In Chapter 5, we presented a tool that facilitates the generation of Spark Java code for the core and the mLlib libraries by providing a user-friendly, intuitive and interactive graphical user interface.

We have proposed a series of methods that can facilitate the parallelization of high performance code. We now briefly discuss how these methods compare with respect to each other and provide some advantages and disadvantages as to which approach might be of better benefit. Intuitively, multithreading might be considered the simplest and less costly approach since it does not require any extra hardware. However, to get good timing results, we will need high end CPUs that can also run several threads concurrently. Another disadvantage of multithreading is the lack of scalability between the memory and the CPU(s) which might result in access time degradation and lack of data coherence. In addition to that, parallelism is limited by the number of threads and improvement is negligible and could even be degraded beyond

that. In contrast, distributed systems can scale endlessly. One can always add a cheap node to the existing cluster to expand the span of computation. Communication overhead is considered minimal in computation extensive applications. However, there is always the issue of integration to be considered. GPUs on the other hand seem to be a middle solution. Currently, one can purchase mid-range GPUs for a minimal cost. In addition to that, their massive threading is very effective for CPU bound applications. Their disadvantage is that they perform poorly for I/O bound applications, and that they are complex to program.

6.2 Suggestions for Future Work

When we first started researching ways to facilitate parallelizing existing Java programs that perform pattern matching, our aim was quite ambitious. Instead of proposing to end-users several shared-memory and distributed methods as we have done, we had originally intended to provide a smart system that could probe the computing environment, detect significant parameters, and automatically replace the sequential desired Java library with a parallel implementation that provides optimal performance. Such a model is a suggestion for future work. The goal would be to build a model that is able to select a hardware architecture best suitable, in terms of execution time, to run a parallelized version of a selected Java library detected in a user's code by using bytecode manipulation. Based on analyzing performance measurements, input size, number of threads and cores, shared memory, number of GPU registers, available hardware, speed and bandwidth of the network, among other potential application and system parameters, the model will be able to predict which approach will give the best optimization with respect to execution time.

We have surveyed the literature and similar techniques can be classified into two approaches: an empirical approach and an analytical approach. The empirical approach such as that used in [86] is based on a training set. However, the application needs to be run at least once to obtain predictions. The analytical approach such as that used in [87] is based on statistical analysis. However, it imposes some challenges due to out-of-order execution, speculation, and prefetching. We provide a quick summary to some of the works in this area. In [86], Qilin is an automatic and adaptive technique that maps computations to the available processing elements in a heterogeneous environment. It is dynamic in that it can adjust to changes in the runtime environment and it focuses on CPU and GPU platforms. Programmers use the Qilin custom API to indicate to the complier the section of the code to be parallelized. In [87], the authors propose the MATE-CG system which is a map-reduce-like system that aims to accelerate map-reduce applications on a heterogeneous cluster. Their framework ports the generalized reduction models on hybrid CPU-GPU clusters to parallelize iterative data-intensive applications. They have also developed an auto-tuning strategy based on analytical models and offline training to identify the best partitioning parameter for heterogeneous execution, and to determine the best CPU/GPU chunk sizes that will optimize performance. In [88], the authors propose Merge, a framework that can automatically distribute computation across heterogeneous systems. The framework consists of a high-level parallel programming language, a predicate-based library system, and a compiler and runtime. In [89], the authors propose Dandelion, a model for heterogeneous systems where execution can be performed on CPUs, GPUs, and FPGAs. Programmers write sequential code in C# or F# (along with LINQ constructs) and the system executes it on the available parallel resources. In [90], the authors propose DryadLINQ, a system that can

automatically transform parallel sections of a sequential program written with LINQ constructs into a distributed model which is then executed on Dryad (large computing cluster).

Bibliography

[1] A. Grama, Introduction to parallel computing, Pearson Education, 2003, .

[2] C. Ferner, B. Wilkinson and B. Heath, "Toward using higher-level abstractions to teach Parallel Computing," in Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International, pp. 1291-1296, 2013.

[3] Pattern matching slides from Princeton: https://www.cs.princeton.edu/~rs/AlgsDS07/21PatternMatching.pdf

[4] C. Charras and T. Lecroq, “Exact String Matching Algorithms”, http://www-igm.univ-mlv.fr/~lecroq/string/index.html

[5] Russ Cox, Regular Expression Matching Can Be Simple And Fast, https://swtch.com/~rsc/regexp/regexp1.html

[6] Regular expression slides from Princeton: http://algs4.cs.princeton.edu/54regexp/

[7] C.S. Rao, K.B. Raju and S.V. Raju, "Parallel string matching with multi core processors-A comparative study for gene sequences," Global Journal of Computer Science and Technology, vol. 13, 2013.

[8] A.K. Sahoo, K.S. Sahoo and M. Tiwary, "Signature based malware detection for unstructured data in Hadoop," in Advances in Electronics, Computers and Communications (ICAECC), 2014 International Conference on, pp. 1-6, 2014.

[9] Large scale citation matching using Apache Hadoop (26 March 2013)

[10] P.D. Michailidis and K.G. Margaritis, "String matching problem on a cluster of personal computers: Experimental results," in Proc. of the 15th International Conference Systems for Automation of Engineering and Research, pp. 71-75, 2001.

[11] A. Ramya and E. Sivasankar, "Distributed pattern matching and document analysis in big data using Hadoop MapReduce model," in Parallel, Distributed and Grid Computing (PDGC), 2014 International Conference on, pp. 312-317, 2014.

[12] Adventures in Hadoop, #3: String Searching WorldCat (http://hangingtogether.org/?p=2199)

[13] R. Tay, "Demonstration of Exact String Matching Algorithms using CUDA," Unpublished Work,.

[14] A. Rasool and N. Khare, "Parallelization of KMP String Matching Algorithm on Different SIMD Architectures: Multi-Core and GPGPU's," International Journal of Computer Applications, vol. 49, 2012.

[15] L. Marziale, G.G. Richard and V. Roussev, "Massive threading: Using GPUs to increase the performance of digital forensics tools," Digital Investigation, vol. 4, pp. 73-81, 2007.

[16] C.S. Kouzinopoulos and K.G. Margaritis, "String matching on a multicore GPU using CUDA," in Informatics, 2009. PCI'09. 13th Panhellenic Conference on, pp. 14-18, 2009.

[17] G. Vasiliadis, M. Polychronakis and S. Ioannidis, "Parallelization and characterization of pattern matching using GPUs," in Workload Characterization (IISWC), 2011 IEEE International Symposium on, pp. 216-225, 2011.

[18] ASM official website: http://asm.ow2.org/

[19] Intel product website: http://ark.intel.com/products/64622/Intel-Xeon-Processor-E5-4650-20M-Cache-2_70-GHz-8_00-GTs-Intel-QPI

[20] Project Gutenberg website: https://www.gutenberg.org/ebooks/search/?query=encyclopedia+britannica

[21] NVIDIA: CUDA-C programming guide PG-02829-001_v5.5, www.nvidia.com

[22] OpenCL official website: https://www.khronos.org/opencl/

[23] D. Lehavi and S. Schein, “Fast Regex Parsing on GPUs”, HP Labs, 2011

[24] A. Housfater, “GPUs and Regular Expression Matching for Big Data Analytics”, IBM video presentation, Big Data Analytics, GTC 2014 - ID S4462

[25] X. Yu and M. Becchi, "GPU acceleration of regular expression matching for large datasets: exploring the implementation space," in Proceedings of the ACM International Conference on Computing Frontiers, pp. 18, 2013.

[26] C. Lin, C. Liu and S. Chang, "Accelerating regular expression matching using hierarchical parallel machines on GPU," in Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE, pp. 1-5, 2011.

[27] C. Lin, C. Liu, L. Chien and S. Chang, "Accelerating pattern matching using a novel parallel algorithm on gpus," IEEE Trans.Comput., vol. 62, pp. 1906-1916, 2013.

[28] N. Cascarano, P. Rolando, F. Risso and R. Sisto, "iNFAnt: NFA pattern matching on GPGPU devices," ACM SIGCOMM Computer Communication Review, vol. 40, pp. 20-26, 2010.

[29] Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng and Q. Dong, "GPU-based NFA implementation for memory efficient high speed regular expression matching," in ACM SIGPLAN Notices, pp. 129-140, 2012.

[30] G. Vasiliadis, M. Polychronakis and S. Ioannidis, "Parallelization and characterization of pattern matching using GPUs," in Workload Characterization (IISWC), 2011 IEEE International Symposium on, pp. 216-225, 2011.

[31] G. Vasiliadis, M. Polychronakis, S. Antonatos, E.P. Markatos and S. Ioannidis, "Regular expression matching on graphics hardware for intrusion detection," in International Workshop on Recent Advances in Intrusion Detection, pp. 265-283, 2009.

[32] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam and C. Estan, "Evaluating GPUs for network packet signature matching," in Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on, pp. 175-184, 2009.

[33] A. Tumeo, O. Villa and D. Sciuto, "Efficient pattern matching on GPUs for intrusion detection systems," in Proceedings of the 7th ACM international conference on Computing frontiers, pp. 87-88, 2010.

[34] CUDA-grep on github: https://github.com/bkase/CUDA-grep

[35] Gary Frost, I don’t always write GPU code in Java but when I do I like to use Aparapi. Online article retrieved 6/15/2016. http://developer.amd.com/community/blog/2011/09/14/i-dont-always-write-gpu-code-in-java-but-when-i-do-i-like-to-use-aparapi/

[36] J. Strnad and Z. Konfršt, "Java on CUDA architecture," in Conference on Computer Graphics, Visualization and Computer Vision (WSCG), 2013. Poster Proceedings.

[37] Aparapi on github: https://github.com/aparapi/aparapi

[38] P.C. Pratt-Szeliga, J.W. Fawcett and R.D. Welch, "Rootbeer: Seamlessly Using GPUs from Java," in High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on, pp. 375-380, 2012.

[39] P. Calvert, "Parallelisation of Java for Graphics Processors," 2010.

[40] JavaCL on github: https://github.com/ochafik/JavaCL

[41] Jcuda official website: http://jcuda.org/

[42] Jocl official website: http://jocl.org/

[43] ATEJI-PX website: http://www.ateji.com/px/index.htm

[44] JCUDA official website: http://www.habanero.rice.edu/Publications.html

[45] JaMP official website: https://www2.informatik.uni-erlangen.de/EN/research/JavaOpenMP/index.html

[46] The CUDA4J application programming interface: https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.lnx.80.doc/user/gpu_developing_cuda4j.html

[47] Project Sumatra homepage: http://openjdk.java.net/projects/sumatra/

[48] K. Thompson, "Programming techniques: Regular expression search algorithm," Commun ACM, vol. 11, pp. 419-422, 1968

[49] Oracle JNI docs: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/

[50] Perl regular expressions: https://www.cs.tut.fi/~jkorpela/perl/regexp.html

[51] Introduction to Distributed System Design: http://www.hpcs.cs.tsukuba.ac.jp/~tatebe/lecture/h23/dsys/dsd-tutorial.html

[52] MPI on Wikipedia: https://en.wikipedia.org/wiki/Message_Passing_Interface

[53] Open MPI official website: https://www.open-mpi.org/

[54] B. Carpenter, V. Getov, G. Judd, T. Skjellum and G. Fox, "MPI for Java," 1998.

[55] A. Shafi, B. Carpenter and M. Baker, "Nested parallelism for multi-core HPC systems using Java," Journal of Parallel and Distributed Computing, vol. 69, pp. 532-545, 2009.

[56] W. Pugh and J. Spacco, "Mpjava: High-performance message passing in java using java. nio," in International Workshop on Languages and Compilers for Parallel Computing, pp. 323-339, 2003.

[57] G.L. Taboada, J. Touriño and R. Doallo, "F-MPJ: scalable Java message-passing communications on parallel systems," The Journal of Supercomputing, vol. 60, pp. 117-140, 2012.

[58] Vega-Gisbert, Jeffrey M Squyres Oscar, J.E. Roman and J.M. Squyres, "Design and Implementation of Java bindings in Open MPI," 2014.

[59] C. Kouzinopoulos and K. Margaritis, "Parallel implementation of exact two dimensional pattern matching algorithms using MPI and OpenMP," in 9th Hellenic European Research on Computer Mathematics and its Applications Conference, 2009.

[60] P.D. Michailidis and K.G. Margaritis, "Parallelization of Multiple String Matching on a Cluster Platform," in Proceedings of the 8th Hellenic European Conference on Computer Mathematics and its Applications (HERCMA'2007), Athens, Greece, 2007.

[61] A. Saà-Garriga, D. Castells-Rufas and J. Carrabina, "OMP2MPI: Automatic MPI code generation from OpenMP programs," ArXiv Preprint arXiv:1502.02921, 2015.

[62] A. Schäfer, D. Fey and A. Knoth, "Tool for Automated Generation of MPI Datatypes,".

[63] N. Ng, de Figueiredo Coutinho, José Gabriel and N. Yoshida, "Protocols by Default-Safe MPI Code Generation Based on Session Types." Cc, vol. 15, pp. 212-232, 2015.

[64] C. Yang and K. Lai, "A directive-based MPI code generator for Linux PC clusters," The Journal of Supercomputing, vol. 50, pp. 177-207, 2009.

[65] D. Kranzlmüller, A. Knüpfer and W.E. Nagel, "Pattern Matching of Collective MPI Operations." in PDPTA, pp. 1243-1249, 2004.

[66] G.L. Taboada, S. Ramos, R.R. Expósito, J. Touriño and R. Doallo, "Java in the High Performance Computing arena: Research, practice and experience," Science of Computer Programming, vol. 78, pp. 425-444, 2013.

[67] S. Ekanayake and G. Fox, "Evaluation of Java message passing in high performance data analytics," 2014.

[68] V. Kumar, A. Grama, A. Gupta and G. Karypis, Introduction to parallel computing: design and analysis of algorithms, Benjamin/Cummings Redwood City, CA, 1994.

[69] J.K. Cringean, G.A. Manson, P. Willett and G.A. Wilson, "Efficiency of text scanning in bibliographic databases using microprocessor-based, multiprocessor networks," J.Inf.Sci., vol. 14, pp. 335-345, 1988.

[70] Apache Hadoop official website: https://hadoop.apache.org/

[71] Hadoop online Tutorial: http://www.tutorialspoint.com/hadoop/hadoop_introduction.htm

[72] Hadoop online Tutorial: https://developer.yahoo.com/hadoop/tutorial/module1.html

[73] Hadoop official website: http://hadoop.apache.org/

[74] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker and I. Stoica, "Spark: cluster computing with working sets." Hotcloud, vol. 10, pp. 10-10, 2010.

[75] Wikipedia: Automatic Programming, retrieved 4/24/2016

[76] R. Aler, Automatic Inductive Programming, Tutorial at the International Conference on Machine Learning, 2006. Carnegie Mellon, Pittsburgh, Pennsylvania, U.S.A.

[77] D.P. Playne and K.A. Hawick, "Auto-generation of parallel finite-differencing code for mpi, tbb and cuda," in Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on, pp. 1168-1175, 2011.

[78] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann and I.H. Witten, "The WEKA data mining software: an update," ACM SIGKDD Explorations Newsletter, vol. 11, pp. 10-18, 2009.

[79] MemSQL official website: http://www.memsql.com/

[80] Apache Spark official website: http://spark.apache.org/

[81] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, pp. 2-2, 2012.

[82] C. Chu, S.K. Kim, Y. Lin, Y. Yu, G. Bradski, A.Y. Ng and K. Olukotun, "Map-reduce for machine learning on multicore," Advances in Neural Information Processing Systems, vol. 19, pp. 281, 2007.

[83] Harris, Derrick (28 June 2014). "4 reasons why Spark could jolt Hadoop into hyperdrive". Gigaom. https://gigaom.com/2014/06/28/4-reasons-why-spark-could-jolt-hadoop-into-hyperdrive/

[84] Mahout official website: http://mahout.apache.org/

[85] Spark Java API documentation: http://spark.apache.org/docs/latest/api/java/

[86] Chi-Keung Luk, Sunpyo Hong and Hyesoon Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on, pp. 45-55, 2009.

[87] Wei Jiang and G. Agrawal, "MATE-CG: A Map Reduce-Like Framework for Accelerating Data-Intensive Computations on Heterogeneous Clusters," in Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International, pp. 644-655, 2012.

[88] M. Linderman, J. Collins, H. Wang and T. Meng, "Merge: A Programming Model for Heterogeneous Multi-Core Systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS XIII) , pp. 287-296, 2008.

[89] C. Rossbach, Y. Yu, J. Currey, J. Martin and D. Fetterly, "Dandelion: a Compiler and Runtime for Heterogeneous Systems," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (SOSP '13), pp. 49-68, 2013.

[90] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. Kumar Gunda and J. Currey, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (OSDI'08), pp. 1-14, 2008.

Vita

Maria Abi Saad

Electrical Engineering and Computer Science Department

Syracuse University

Graduate and Undergraduate Schools Attended:

- Syracuse University, Syracuse, New York, USA
- Lebanese American University, Byblos, Lebanon

Degrees Awarded:

- Master of Science in Computer Engineering, 2008, Lebanese American University
- Bachelor of Engineering in Computer, 2006, Lebanese American University

Professional Experience:

- Teaching Assistant, Department of Electrical and Computer Engineering, Syracuse University, 2008 – 2015
- Assistant Customer Application Analyst, International Turnkey Systems, Beirut, Lebanon, 2006 – 2008
- Part time faculty, Lebanese American University, Byblos, Lebanon, 2007 - 2008

Maria Abi Saad's dissertation was supervised by Dr. C.Y. Roger Chen.