Automatic C Code Parallelization

- Parallel code generation framework -

Bachelor Thesis Report

ED6-1-F16

Aalborg University Esbjerg
Electronics and Computer Engineering

This report was written in LaTeX.

Electronics and Computer Engineering
Aalborg University Esbjerg
http://esbjerg.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Automatic C Code Parallelization

Theme:
Distributed Systems

Project Period:
Spring Semester 2016

Project Group:
ED6-1-F16

Participant(s):
Mike Castro Lundin
Arturs Gumenuks
Aleksandrs Levi

Supervisor(s):
Daniel Ortiz-Arroyo

Page Number: 119

Date of Completion:
June 9, 2016

Abstract:

The project focuses on creating a system for automatic parallelization of a sequential code. The system is written in Java and distributed between a local machine and Google Cloud nodes.

The system is provided with a sequential C code as an input, written with respect to the defined subset of C.

Compiler techniques, such as a lexical analyzer and a part of parser are involved to analyze the input code and derive necessary data, which is then processed by an Apache Spark program, running on Google Cloud.

The parallel C code is generated using POSIX Threads library. A variety of tests of the system were performed.

The task of parallelizing a given program has been achieved. Conclusions regarding the performance of a parallel program have been drawn.

Contents

Preface

This report has been written as the documentation of the Bachelor project done by the group ED6-1-F16 at Aalborg University Esbjerg. The project covers the topic of automatic C code parallelization. Techniques used for solving this problem are described in the paper in detail.

The report starts with specifying the problem and defining specific concepts used in the context of the system. A glossary containing some of the used terminology is provided. The following chapters discuss design of the system and its implementation. A separate chapter is devoted to describing the tests conducted after the system has been developed. The results of the tests are then discussed.

The appendix at the end of the report contains a short guide on connecting a Java application to Google Cloud servers.

The source code with the developed application is available at the following github repository: `https://github.com/ED6/c-code-parallelizer`.

Aalborg University, June 9, 2016

Aleksandrs Levi
<alevi13@student.aau.dk>

Arturs Gumenuks
<agumen13@student.aau.dk>

Mike Castro Lundin
<mcastr13@student.aau.dk>

Glossary

- **Job (Cloud):** a collection of tasks determined by an application; in our case, a Spark program executed on a cluster of computers.
- **Bucket (Cloud):** a logical unit of storage used for storing objects (files) in the cloud.
- **Push down automaton:** an automaton with a stack.
- **Batch processing:** executing a series of non-interactive jobs all at one time[3].
- **Ecosystem of a framework:** an infrastructure of the framework's services.
- **First-class citizens:** in computer programming, entities which can be used as function parameters, returned from functions or assigned to variables. For example, functions are typically first-class citizens in functional programming languages.
- **Granularity of a thread:** a number of operations that should be performed by a thread[24].
- **Execution Table:** a list of lists which declares the order in which to execute sections. One dimension is the sections that will be executed in parallel, and the other dimension is sequential execution scheduling.
- **Fine-grain slicing:** implementation of a slicing algorithm that prioritizes getting the smallest possible slices.

Chapter 1

Introduction

The problem of making a program benefit from multi-core architecture has been extensively researched in the field of Computer Science. Although it is possible to manually identify parts of the code that can be potentially run in parallel and distribute their execution among multiple processors, it may take a lot of time and effort which could otherwise be spent on developing the application directly. With that in mind, it becomes clear that a system performing automatic program parallelization could be a useful tool for distributing software originally created without multithreading techniques in mind.

The purpose of the developed system is exactly that - automatic parallelization of the given source code. Since this task is considered an important topic in the field of computer programming, it was decided to study it thoroughly.

During the system development a variety of techniques have been learned and applied, such as: lexical analysis, cloud computing, Big Data analysis, and more. A range of algorithms have been developed in order to solve problems arising over the course of development, for example: optimal thread distribution for multi-core systems and identifying independent parts of a sequential program.

In essence, the main questions that were answered during the project development are:

- How to detect independent parts of the given sequential program?
- How can this information be converted into an efficient parallel program?

Chapter 2

Specification

2.1 System Overview

The purpose of this short chapter is to introduce the reader briefly to the system, whose details will be discussed in depth later in the report. The system overview is provided in figure 2.1.

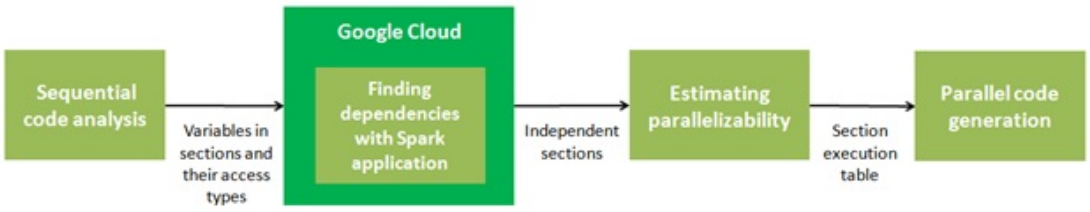


Figure 2.1: System schematics

The process starts with analyzing sequential input code using techniques common in compiler design. The purpose of this analysis is to find variables in all sections as well as access types of these variables (writing to or reading from). This information is then given to a Spark application running on Google Cloud in order to obtain list of independent sections, which is passed to the next stage, which is responsible for estimating potential ways of parallelizing the code. The flow of the program, section distribution across threads and other factors are considered here. The result of this part is an execution table, which contains information about how the sections should be organized for parallel execution. Based on this table, a parallel code is generated in the last stage of the system.

2.2 Requirements

There were several requirements for the project, introduced by the group in the beginning of development.

1. Analyze pieces of code written in C language. The C language uses the imperative programing paradigm, which makes it simpler to analyze compared with, for example, object-oriented paradigm, which may introduce more complex scenarios.

   However, there are still some difficulties with C language grammar. Some rules have been excluded from it as they were considered more complex to analyze: for example, pointers and interprocedural calls.

2. The next requirement is to involve Cloud computing and Big Data tools in the project. The program should be able to deal with Big Data, since a large input source code can potentially occur (when analyzing an operating system source code, for example).

3. The system is to be designed in Java programming language and should be able to do diverse tasks. Thus the next requirement is to cover the functionality of the chosen language.

4. The main point of the system is to produce a parallel version of a sequential code written in a reduced C language grammar. The requirement here is to have automatically generated parallel code running faster than manually written sequential one.

5. It is required that thoroughgoing testing of the system is done. The challenge is that many different variables participate in the process. That is, many parameters can be customized and tuned. The analyzed source code is an input itself, but it is also a C program, which can be compiled and executed with some input, whose size can, for instance, indicate the efficiency of parallelization. It is thus not considered possible to test every possible scenario. There are plenty of ways of parallelizing a program which will lead to some overheads and context switches. These factors should be taken into account while testing in order to obtain the correct conclusions.

6. As it is possible to predict, the project requires certain knowledge in compilers theory, parallelism and distributed computing. These topics were covered in the courses studied by the group in the university. So, one more requirement is to use taught material in the project development.

2.3 Section concept

For the purposes of parallelizing given source code, the concept of "sections" has been developed.

Each section is a block of code that is going to be executed in a separate thread after parallelization has been completed. Initially, the program is divided into sections according to the following principle: every line of code in main function consisting of a single instruction is a separate section, and conditional statements and loops are sections on their own, including all the iterations within them. This is also known as a fine-grained slicing[6].

All the sections are numbered starting from 1, while 0 is reserved for declaring all the variables used in the program – this section will be further referred to as "Section 0". This approach has been selected for the following reasons:

- It simplifies the task of finding data types of the variables.
- It avoids having variables with the same name declared in different scopes.
- It allows the multiple threads of the parallel version of the program to work as expected by ensuring that each thread has access to the variables it works with. The idea behind it is that the memory for the variables is allocated before any actions in the main program take place.

Figure 2.2 demonstrates an example of dividing a C program into sections. It can be seen that sectioning happens only within the main function. The return statement is not considered a section.

The idea behind dividing the program into sections in this way is that the sections can be further united and moved around until the best execution time is obtained. It was decided not to parallelize loops within themselves since this task was considered of a different nature.

```
#include<stdio.h>

int main()
{
   /* This is section 0. Variables are declared here. */
   int n = 30, first = 0, second = 1, next, c;
   //Section 1:
   printf("First %d terms of Fibonacci series are :-\n",n);
   //Section 2: a loop, so the section lasts as long as the loop does.
   for ( c = 0 ; c < n ; c++ )
   {
      if ( c <= 1 )
         next = c;
      else
      {
         next = first + second;
         first = second;
         second = next;
      }
      printf("%d\n",next);
   }
   //Section 3:
   printf("the value of a second variable is: %d", second);
   //Section 4:
   test[0]++;
   //A return statement: it is not considered a section.
   return 0;
}
```

Figure 2.2: A C program divided into sections

2.4 C Language Subset

This project will focus on the automatic parallelization of a subset of C. Following are the basic rules and format necessary for the parallelization to be possible.

The first specification in regards to the code is the structure that is expected. The structure is shown in figure 2.3.

The main point of this structure delimitation is that the code input must declare all its variables either before main, making them global variables, or at the start of main. This means that the framework only needs to scan these two sections to find variables and their data types, and ensures that variables can be accessed using memory referencing by any future sections.

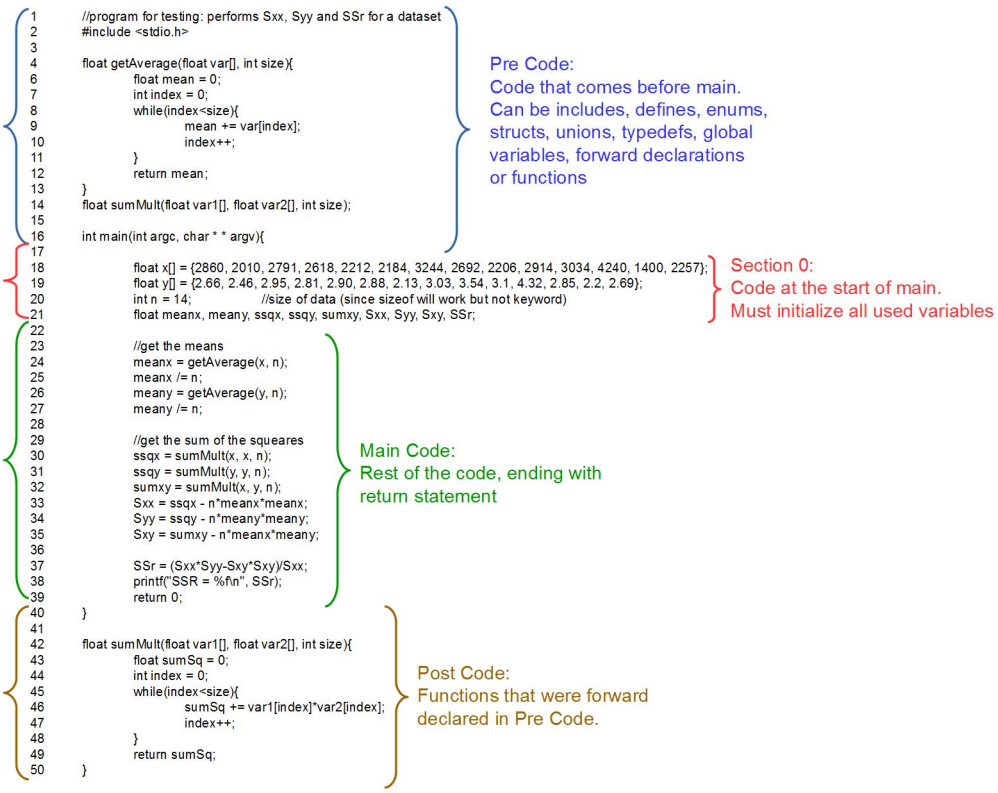


Figure 2.3: Structure of C Code

Another restriction is regarding returns and main. The current version of the framework assumes that the main function has a return type int and receives no arguments. This rule can easily be removed in a future version by getting the sig-

nature from precode processing, and using this information when generating code. The input code can only have one return statement at the end of main. This is because the parallelization is done by generating functions out of code sections, and thus executing a return statement would not terminate the program, but end the execution of the function. If different return values are desired, this can be done by using a variable in the return statement, which is modified if some conditions are detected. This rule can also be removed in the future, by replacing all return statements with a framework function which passes a message to the main thread that it should terminate the program with a defined output. The same concept can be used to ensure correct order when for example the *printf* function is used. In this scenario *printf* calls would be replaced with a framework function which takes as parameters the string to be printed and the section number that is calling, and stores them in order. Then, before the main thread is terminated, it would print all the strings that it has received in order. These concepts have not been applied in the current version, but could be implemented in the future.

Parallelization is implemented using pthreads and functions. Besides causing issues with return statements, this will also cause conflicts when using typedef, enums, structs, unions and defines that have been declared inside of main. If a function uses a struct that was created inside main, it will generate a compiling error because the size of the struct is not known. Thus, while implementing these statements in main would work in a sequential program, it will not work in the generated code. This is why defines, struct, union and enum declarations and typedef must be done outside main, in the precode section

All code was tested using the GCC 4.8.4 compiler inbuilt in Ubuntu 14.04.4. Since the program uses pthreads, which are POSIX threads, Windows OS computers will not be able to execute the code unless they have Cygwin or a similar application installed. This also causes issues when executing files automatically, since while UNIX uses -pthread to compile, Cygwin uses -lpthread. Even though we are working with the GCC compiler, the use of its data types (for example _int128) has not been considered, but could be implemented by adding them to the list of keywords.

The framework makes use of some reserved identifiers to function, and thus using the same name will cause issues. The reserved identifiers are timespecDiff, sdata_struct, sec_ followed by a number (such as sec_1, sec_2...), start, end, secNUM_thread (such as sec1_thread, sec2_thread), sdata, time.h library identifiers, pthread identifiers, inttypes.h identifiers and stdio identifiers.

Regarding C as a language, the framework implements only keywords defined in the C89 standard, and no pointer operations are allowed (no pointer operator

(*), no address operator (&) and no structure dereferencing (->)). They are forbidden because including them would require keeping track of what memory is being modified, which can be a very complex task when the code uses overflows or null pointer dereferencing. The goto statement is also not included in the grammar as the execution flow might be altered by the framework. While arrays are allowed, it is up to the user to ensure that there will not occur any overflows that will interfere with other variables. The grammar including the #ifdef and #ifndef usage has also not been considered in the current version.

In regards to functions, it is preferred for the function call to include any necessary data in its parameters, and it does not make use of global variables. Since the current version of the framework does not analyse functions, it will not realize that a function call might imply writing to a non local scope variable, which could cause two dependent sections to execute simultaneously. Thus if a function call must require reading or writing, this must be done via parameters and returns.

Casting is forbidden in the precode section. This means defines and global variables may not use casting in any form. This is to avoid issues with detecting line ends, which is explained more in-depth in section 4.4. While section 0 does not allow defines currently, if this rule was removed, then ending section 0 with a define would also be forbidden for the same reason.

A future implementation could include adding keywords according to the libraries that are being imported. Since this is not done yet, library datatypes may not be used, since the framework will not recognize them as data types. Functions have a different behaviour, so they will work without any issues.

To conclude, the rules for the program to be parallelizable in the framework are:

1. Have all variable declarations either outside main or at the start of main.
2. Use "int main( )" as signature for main.
3. Have only one return statement in main, located as the last statement in main.
4. Have all #define, typedef and struct/union/enum declarations in Pre Code.
5. Execute from a UNIX operating system.
6. Do not use timespecDiff, sdata_struct, sec_ followed by a number (such as sec_1, sec_2...), start, end, sec#_thread (such as sec1_thread, sec2_thread), sdata as identifiers, or any identifier used by time.h, pthread.h, inttypes.h or stdio.h libraries.

7. Do not use compiler specific data types such as GCC's _int128.
8. No goto statements, memory operations (* (pointer), & (address) or -> (structure dereference)), or #ifdef #ifndef usage.
9. Functions should not access global variables (either reading or writing to them).
10. No casting in the precode section.
11. No use of data types imported from libraries

2.5 Existing Solutions[28]

The generation of parallel code is a study area in computer science that has been in development since the early 80s. The concept was started with the concept of program slicing by Mark Weiser in 1984 [30]. The concept was to iterate backwards in a program to find which lines of the code directly or indirectly affected the outcome of a specific line of code. Soon the concept of forward slicing came, which involve iterating forward to see which lines where affected by a certain line of code.

Furthermore, slicing was started by static slicing, which meant that the slices should be correct for any possible execution or input. This meant that most slicing solutions where made on reduced sets of the C language, in particular excluding interprocedural calls, pointers, arrays, unstructured code and concurrency. Soon the concept of dynamic slicing was developed, for slicing code for only specific execution conditions [16]. This allowed extending the subset of C and making less conservative assumptions, and thus smaller slices.

This project makes use of static forward and backward slicing, thus the developments on dynamic slicing will not be disscussed.

The uses of slicing started from program debugging, understanding and verification, and have extended to maintenance, testing, fucntional cohesion-metric compuitation, dead code elimination, reverse engineering, software portability analysis, reversable component analysis and parallelization of sequential programs. [33]

The parallelization of sequential programs started with attempts to reconstruct sequential behaviour from parallel behaviour [31]. The issue was that even if slices were data independent, the behaviour could be changed in the parallel execution, caused by the non-deterministic nature of concurrent programs. This issue is discussed in C language subset, where our solution is proposed.

Later, the concept of Program Dependence Graphs was defined to generate accurate slices [13]. This type of graph combined data dependencies and control dependencies.

Then static slicing was extended to include interprocedural calls using System Dependence Graphs [26].

Most slicing applications make use of an intermediate representation, either a control flow graph, program dependence graph, system dependence graph or some newer type of graph. Since our framework does not use graphs as intermediate representation, we will not discuss the newest graph developments.

One of the topics that have filled many scientific papers is implementing pointers and arrays, as well as unstructured programs (use of goto, continue, break). A solution was proposed by Jiang [14] but later proven to generate incorrect slices [1]. An alternative solution was proposed by Agrawal [1].

Regarding arrays, this system implements a conservative solution, which assumes use of the whole array when updating one element of the array [20].

The main issue with allowing pointers is detecting multilevel pointer aliasing (two variables modifying the same memory). This problem has been studied and found to be NP-hard [17].

One of the proposed solutions is the implementation of a preliminary execution of the code to get the memory addresses being written to [21].

An implementation for slicing sequential programs in the full C has been developed [6]. Later it was extended to include slicing in a UNIX environment for concurrent programs excluding interference (communication) between threads [5].

The slicing algorithm has also been made parallel first for sequential programs [22] and then for concurrent programs [12]. This solution was implemented in a subset of C, using Lex and Yacc.

In this project it was decided to build the compiler parts from scratch, doing only the tasks that were required for the system to work.

Some of the latest development has come from research on methods to slice Object Oriented languages [15] and even concurrent OO programs [4].

Chapter 3

Theory on parallel computing

3.1 Parallel computing concept

This section will focus on specific aspects of parallel computing, which can find their application in the developed system.

Parallel computing implies running computations simultaneously on several cores of a processor. Compared with sequential computing, where one computation is performed at a time, parallel computing allows running many computations (for example, using threads) at the same time. This makes computation process faster, when the parallelized program is designed well, taking into account several points, described below.

There are issues which can occur with variables shared by several threads and modified at the same time. This is known as a race condition. Due to poor design, the eventual value of a shared variable will depend on the varying order of threads accessing it. A solution to this issue is using locks, for example. Once a thread starts working with a shared resource, the latter gets locked, so other threads cannot work with it too. The features of shared variables will not be further discussed, because the developed system does not make threads deal with shared variables. Instead, threads are composed in such a way that they work with their own variables, so are independent.

A key point of parallelism is the benefit one can get when running a parallel program instead of a sequential version of it. The advantage is usually represented by time: a parallel program is supposed to run faster than a sequential one. However, things are not as simple as they might appear.

The following question becomes relevant when working with parallel programs.

"How fast can a parallelized application run on multiple processors compared to a monolithic application running on a single processor?" [24]. Speedup calculation comes in. Speedup is defined as the ratio between time elapsed in sequential execution and time elapsed in parallel execution [25].

$$Speedup = SequentialExecutionTime / ParallelExecutionTime \tag{3.1}$$

It is proposed that the fastest version of the sequential program is used in this calculation. Speedup value should be greater than 1 for parallel program to be advantageous. Another formula allows calculating the maximum speedup of a parallel program run on P processors. T_s variable represents sequential execution time.

$$Speedup = \frac{T_s}{\frac{T_s}{P}} = P \tag{3.2}$$

This is an ideal case with a linear dependence. It means that the code can perfectly be split into P sections run by P processors. Nevertheless, the reality is different. A code will always have parts, which require to be executed sequentially, such as initialization section, return statement or any other parts demanding a strict flow respected. In order to consider these sequential parts, Amdahl's Law will be presented. The expected maximum speedup S achievable with P processors running the application with sequential fraction f is:

$$S \leq \frac{1}{f + \frac{(1-f)}{P}} \tag{3.3}$$

If one now assumes that infinitely many processors are used, the formula reduces to the following statement:

$$\lim_{P \to \infty} S(P) = \frac{1}{f} \tag{3.4}$$

This yields that no matter how many processors are used, the maximum speed up will be bounded by the sequential fraction of a program. The effect of sequential fraction is depicted in the figure 3.1.

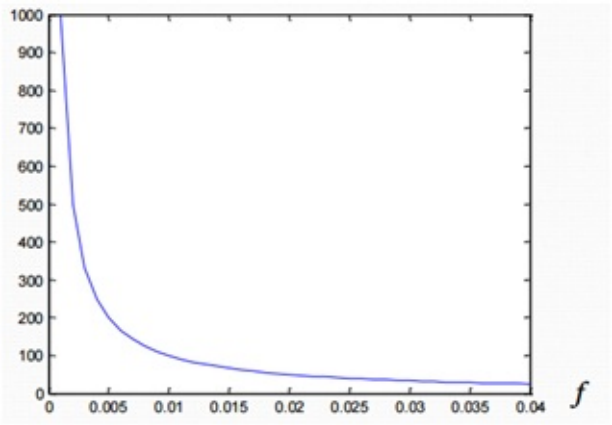


Figure 3.1: Speedup dependent of sequential fraction [24].

Efficiency is another related term. It is the measure of processor utilization [25] and is calculated like this:

$$Efficiency = \frac{Seq}{P * Parallel} = \frac{Speedup}{P} \tag{3.5}$$

In the formula, "Seq" is sequential execution time, "Parallel" is parallel execution time and P is the number of processors used.

A low value of efficiency indicates non-efficient usage or processors since they are idle. A high value, in its turn, signifies high-efficient processor utilization, meaning that they are actually busy with the computations. Let one take a look at the graph 3.2.

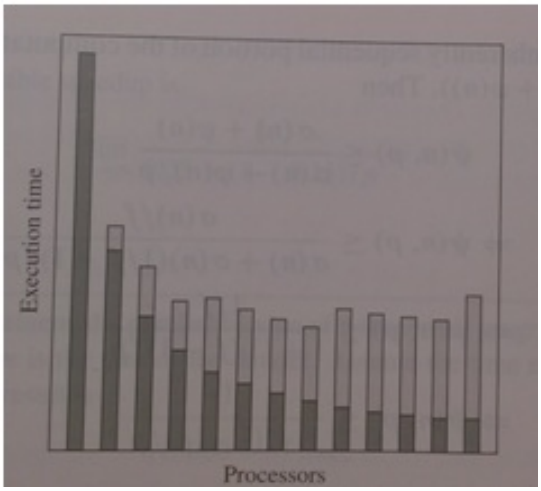


Figure 3.2: Computation vs. communication time distribution [25].

As it is possible to observe, the overall execution time is decreasing for the first 8 bars. Even though the actual computation time (marked with black color) falls, the communication overhead time (marked with gray color) becomes larger making slight the change in overall execution time. What is more interesting is that involving even more processors to the computation makes the execution time larger due to the high communication between many processors. Thus, it makes no sense to involve more processors than needed, because the communication overhead will discount the actual computation time, so no efficiency is achieved. According to the book, for any fixed input size there exists an optimal number of processors that minimizes total execution time of a program.

The mentioned communication overhead is a factor which is not considered by Amdahl's Law described earlier. Experimenting with increasing input size will result into an Amdahl's Effect observed. As the book states, parallel overhead time k(n,p) typically has lower complexity than parallel portion of the computation φ(n). This means that communication time will increase slower than speedup with an increasing problem size n [25]. A graphical representation of Amdahl's Effect is available in the figure 3.3.

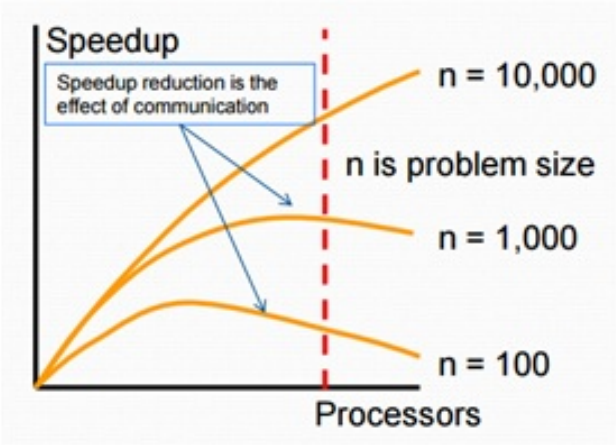


Figure 3.3: Amdahl's Effect demonstration [24].

Thus, for a fixed amount of processors, there is a problem size, at which communication time is so small, that it does not cause speedup reduction and can be omitted.

3.2 Apache Spark theory

3.2.1 Features

This chapter will be dedicated to the theoretical aspect of Apache Spark framework, used in the project. The framework was developed by UC Berkley (University of California, Berkley) and is an engine for large-scale data processing. Apache Spark employs Hadoop Distributed File System (HDFS) and Hadoop Map-Reduce algorithm. Apart from similarities, there are also differences.

The main features of Apache Spark are listed below and will later be discussed to a certain extent:

1. Besides Map and Reduce functions, more than 80 high-level operators are available [8].
2. Written in Scala and runs on JVM.
3. Provides API for Scala, Java and Python. Functional programming is recommended.
4. Interactive shell programming is available for Scala and Python.
5. Data is represented in Resilient Distributed Datasets (RDDs), which support two kinds of operations: transformations and actions. RDDs are fault-tolerant and can be processed in parallel.

Some other details are also worth mentioning. Similar to Hadoop, Spark has its own ecosystem presented in the figure 3.4.

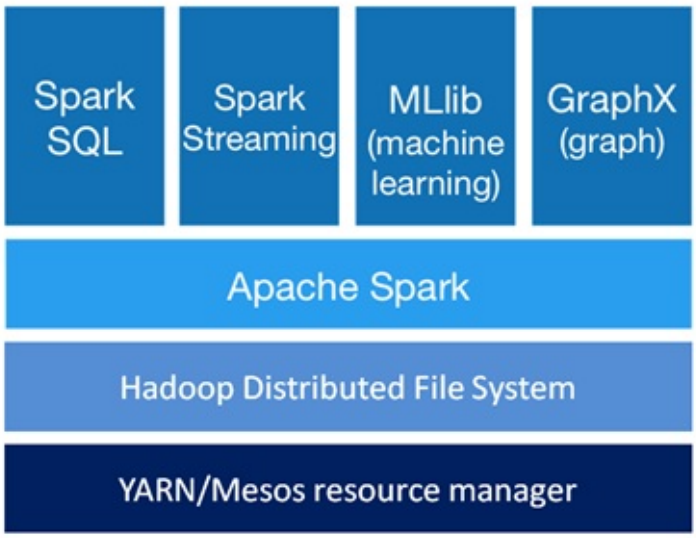


Figure 3.4: Spark ecosystem [8].

The major advantage of Spark over Hadoop is speed. According to the official Apache Spark website, Spark programs run "100x faster than Hadoop MapReduce in memory, or 10x faster on disk". A visual example is shown.

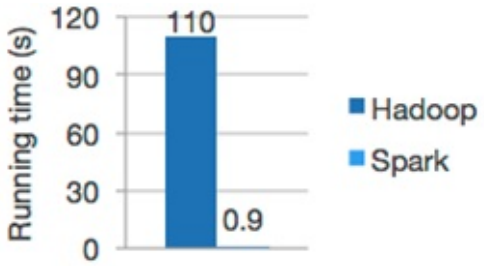


Figure 3.5: Spark vs. Hadoop running time [8].

Spark also managed to win Daytona Gray Sort 100TB Benchmark (sorting competition), beating the top score of Hadoop [9]. Details are provided on the blog of a winner team participant [32].

	Hadoop MR Record	Spark Record	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**	**22.5 GB/min**

Figure 3.6: Spark and Hadoop performance comparison [32].

The conclusion, which the author comes up with, tells about a speed advantage of Spark: "using Spark on 206 EC2 machines, we sorted 100 TB of data on disk in 23 minutes. In comparison, the previous world record set by Hadoop MapReduce used 2100 machines and took 72 minutes. This means that Apache Spark sorted the same data 3X faster using 10X fewer machines. All the sorting took place on disk (HDFS), without using Spark's in-memory cache".

Nevertheless, sometimes it is not very beneficial to use Spark's speed. An article on InfoWorld Journal says, that it can simply be fine to use MapReduce, if the data operations are static and the program can afford some time spent on batch-processing. If the data is retrieved in real time (sensor readings, for instance) or multiple operations are used, then Spark is the right choice [23].

3.2.2 Overview and the working principle

Again, according to official Apache Spark implementation, a typical Spark application contains a driver program, which is responsible for running user's main function and executing various parallel operations on a cluster [7]. Running applications as independent sets of processes on a cluster in coordinated by the SparkContext object, initialized in the driver program. SparkContext is an essential element of any Spark application, since it is basically a client of Spark's execution environment and gives a developer an opportunity to work with RDDs and other objects, use Spark services and run jobs [18]. There are several types of cluster managers, which the SparkContext connect to: Spark's own standalone

cluster manager, YARN or Mesos). The role of a cluster manager is to distribute resources between applications. Once the connection is established, executors are introduced on nodes in the cluster. Executors are processes that execute computations and store data for the application. After this, it sends the application code to the executors. Finally, SparkContext supplies executors with the tasks (application commands sent from the driver) [11]. A figure below depicts the described architecture.

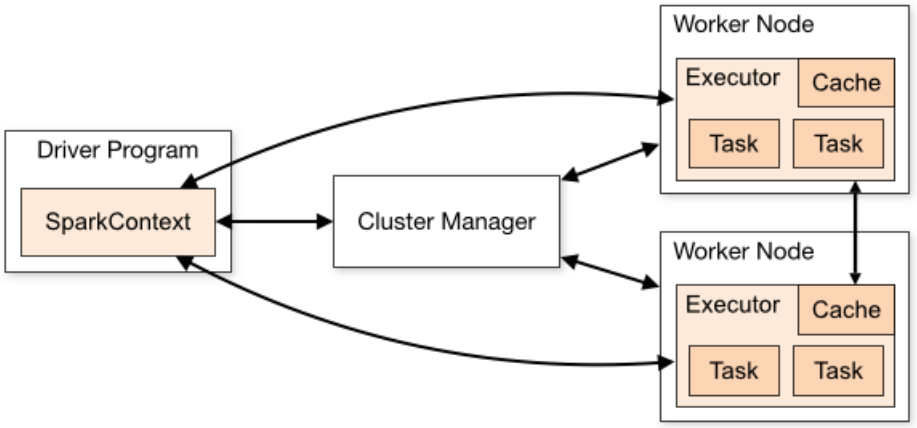


Figure 3.7: Spark cluster overview [11].

Two more common terms used in Spark are jobs and stages. Jobs are parallel computations consisting of multiple tasks. They are spawned in response to Spark actions. A job is split into smaller sets of tasks. These sets are called stages and they are dependent on each other. Both terms will typically appear in driver's logs [11].

3.2.3 RDDs concept

A resilient distributed dataset is an essential abstraction, which Spark applications are built around. It is created based on a data from a Scala collection (already initialized in the code) or a file in a Hadoop-supported file system. An RDD is split into partitions (amount is customizable), which are given to several nodes, so can be operated in parallel. RDDs are fault-tolerant, which means that if a partition is lost, it will be automatically recomputed based on the transformations which produced it. The collection is parallelized by calling *parallelize(data: list)* or *textFile("data.txt": String)* method from SparkContext object. The process is shown in the figure (note that the arrows are mixed up in order to show that partition-to-node assignment can be various).

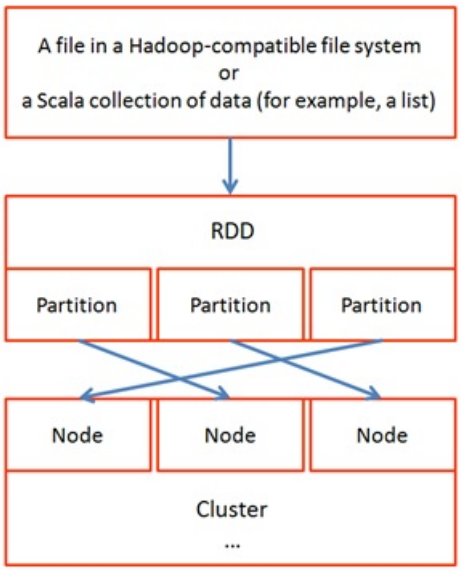


Figure 3.8: Creation and distribution of an RDD.

There are two types of operations, which can be performed on an RDD. These are transformations and actions. Transformations will be discussed first.

The point of a transformation is to do certain manipulations on RDD and return a new RDD, which is a result of those manipulations. Note, that the contents of the original RDD do not change after a transformation, since RDDs are immutable. An example of a transformation can be a common *map(func)* operation, where *func* is a function passed as a parameter. This, by the way, explains the benefit of using a functional paradigm when creating a Spark application, since functions are treated as first-class citizens. Transformation *map(func)* returns a new dataset formed by passing elements of the original RDD through the function *func*.

In Spark, Transformations are lazy, which means that they do not happen immediately. Transformations called on a dataset are remembered instead. They are actually computed, when an action applied to the dataset requires a result to be given back to the driver. As it is reasoned in Spark website, this design implies a higher efficiency. Let one assume, that a developer is working with a known example of applying *map(func)* and *reduce(func)* operation to a dataset. The new dataset formed by calling *map* transformation will be utilized in *reduce* action. The larger resulting dataset of *map* is changed (i.e. reduced), so there is no need to return it to the driver program right away. The program waits for an action, so the result of

reduce is returned to the driver instead.

Actions are also different from transformations in what they return. Actions return a value (of some other type, not an RDD) to the driver after performing a computation on the dataset the action was applied to. Operation *take(n: int)* can be an example of an action. It returns an array of first n elements of the RDD. Spark documentation on API for RDDs states that this action "works by first scanning one partition, and use the results from that partition to estimate the number of additional partitions needed to satisfy the limit" [10], which gives a reader a clue of how an action can work.

A small remark about RDD operations is that some of them may trigger *shuffle* event. The point of shuffling is to reorganize data in such a way that it is distributed differently between the partitions. As it is explained on Spark website, "In Spark, data is generally not distributed across partitions to be in the necessary place for a specific operation. During computations, a single task will operate on a single partition - thus, to organize all the data for a single *reduceByKey* reduce task to execute, Spark needs to perform an all-to-all operation. It must read from all partitions to find all the values for all keys, and then bring together values across partitions to compute the final result for each key - this is called the shuffle" [7]. One should remember that shuffling involves replicating the data across executors and machines, which makes it a complex and heavy operation. Other shuffle-triggering operations as well as additional RDD organization mechanisms are listed in the provided source.

A developer may want to use the same RDD several times. Whenever an action is performed on a transformed RDD, the latter will be recomputed. There is a special technique designed for this case. The developer can *persist* (or *cache*) an RDD. It will lead to keeping elements of the RDD around on the cluster, allowing a faster access the next time the RDD is appealed to. Caching is done on memory level by default. It is also possible to persist RDDs on a disk or copy them across multiple nodes.

Other abstractions in Spark are accumulators and broadcast variables. The only information, which a node can work with, is a partition of an RDD assigned to it. However, sometimes extra data is required to be read or created within a node, especially when designing sophisticated algorithms in Spark. It would be extremely inefficient to carry this additional information in the RDD, because it would violate its format and occupy large space. A solution is to introduce variables in a driver program, which all nodes are able to read or to write to. Thus, Spark supports two types of shared variables: write-only accumulators and read-only broadcast

variables.

An accumulator is a shared variable, which can be safely modified from tasks inside actions only. Accumulator concept is an extended version of MapReduce counters, which are a typical application of accumulators as well as sums. Tasks running in parallel on a cluster are then able to add numbers to accumulator, but they cannot read its value. The value of an accumulator can only be read in the driver program.

Accumulators have a native support for numerical types and the developers can create accumulators of other types by implementing AccumulatorParam class. The limitation here is that the added data must be of the same type as the accumulated value. A more general approach is to implement AccumulableParam class, supporting the resulting type different from the type of elements added. Note, "adding" is not necessary used in an arithmetical sense. It is called so because of a corresponding method *add*. So, with the aforementioned interface it becomes possible to forge, for instance, an accumulated HashMap of RDD elements.

Broadcast variables are read-only variables kept in a cache of every worker (see figure 3.7), which if preferred over delivering a copy of it to tasks. Spark promises that broadcast variables are attempted to be distributed efficiently by using broadcast algorithms for communication cost reduce.

A broadcast variable is created from a variable (passed as a parameter) and is called from SparkContext. Its value can then be accessed in nodes by calling *value* method. If the initial variable is called by its name instead, then it will be shipped to the nodes more than once, which is not efficient. At last, the contents of the initial variable should not be modified after it is broadcast to nodes to make sure that they all receive the same value of the broadcast variable.

Finally, it is possible to execute Spark application on a local machine (where workers are represented by threads) or on a distributed cluster of computers, which can be run on Amazon EC2 or Google Cloud Platform.

Chapter 4

Design

4.1 System architecture

4.1.1 Overview

In order to describe the overall flow of the system, a series of UML Sequence Diagrams is presented and described in this section. Each diagram and its corresponding explanation represents a closer look at some part of the system, aiming to allow a better understanding of the complete architecture.

4.1.2 Obtaining tokens

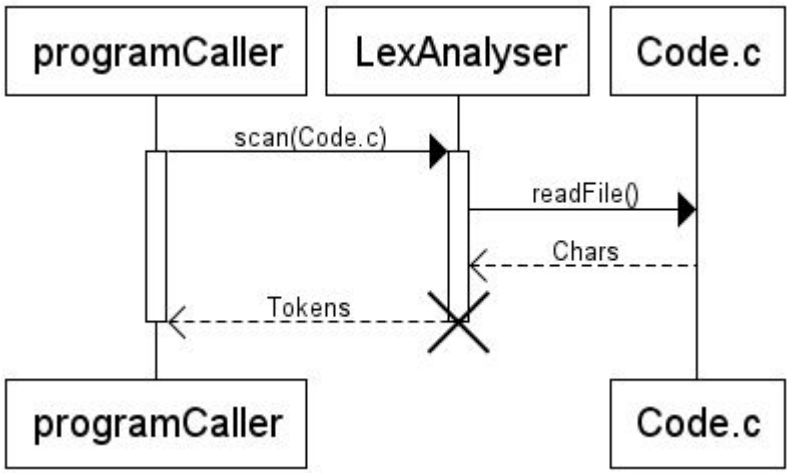


Figure 4.1: Getting the tokens from the input code

The system starts by acquiring tokens of the input source code. Figure 4.1 illustrates the process. programCaller, which contains the main method of the system,

calls *_scan* method of the LexAnalyser class, which reads the input code, contained in "Code.c" file, and returns the detected tokens to the programCaller.

4.1.3 Splitting the code and getting new keywords

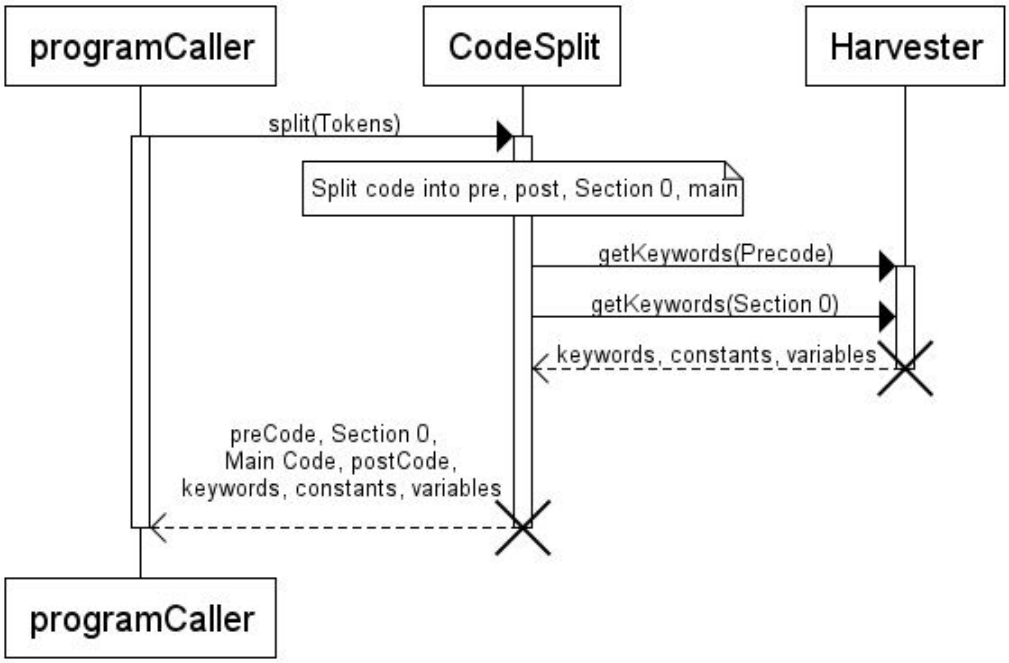


Figure 4.2: Splitting the tokens and fetching keywords

Once the tokens of the input code are obtained, it is required to separate them into tokens of preCode, postCode, Section 0 and Main code. This is performed by CodeSplit class.

Aside from that, information about the new keywords, constants and variables, which can be defined by the user in Precode or Section 0, is retrieved by the Harvester class. All the obtained information is then sent back to the programCaller.

4.1.4 Separating sections of the main code

The obtained tokens of the main code must then be separated into sections. For this purpose, SectionSeparator class is used. Not only the Main tokens, but also the new keywords and constants are passed to the SectionSeparator. After processing, the separated Main tokens are returned to the programCaller. Figure 4.3 provides visual illustration of the process.

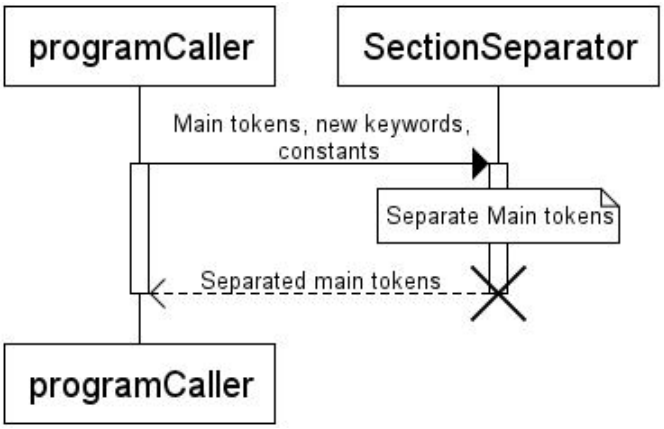


Figure 4.3: Separating the tokens of the main code

4.1.5 Obtaining information about Writes and Reads

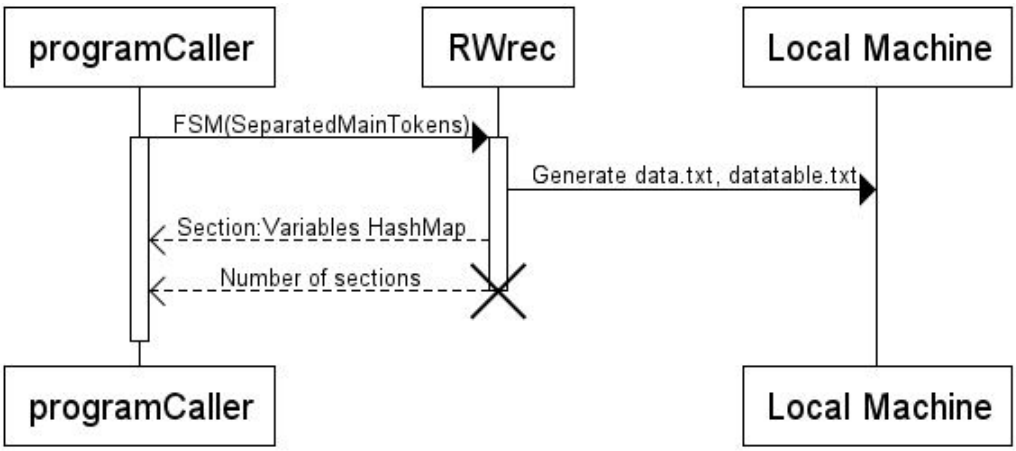


Figure 4.4: Obtaining read-write information

Class RWrec (stands for "ReadWriteRecognizer") is responsible for analyzing the separated tokens of the main code and retrieving information about variables being written to and being read in the code. It is discovered which operations each of the sections performs on its variables. This information is converted to a format required for further code processing by the DepFinder class and stored in 2 text files on the local machine: "data.txt" and "datatable.txt".

Aside from that, this class links each section to the variables contained in it, which is required for further processing by Generator class in the last stage of the system. A HashMap containing this information and the number of sections in the input code are returned to the programCaller.

4.1.6 Finding independent sections utilizing Google Cloud

The next stage of the system deals with finding independent sections within the input source code. The class responsible for this is DepFinder. It is located in a DependencyFinder.jar file stored on a Google Cloud Storage bucket.

The Cloud operations executed by the system are explained in form of interactions between the Local system and the Cloud. The overall flow of the Cloud part

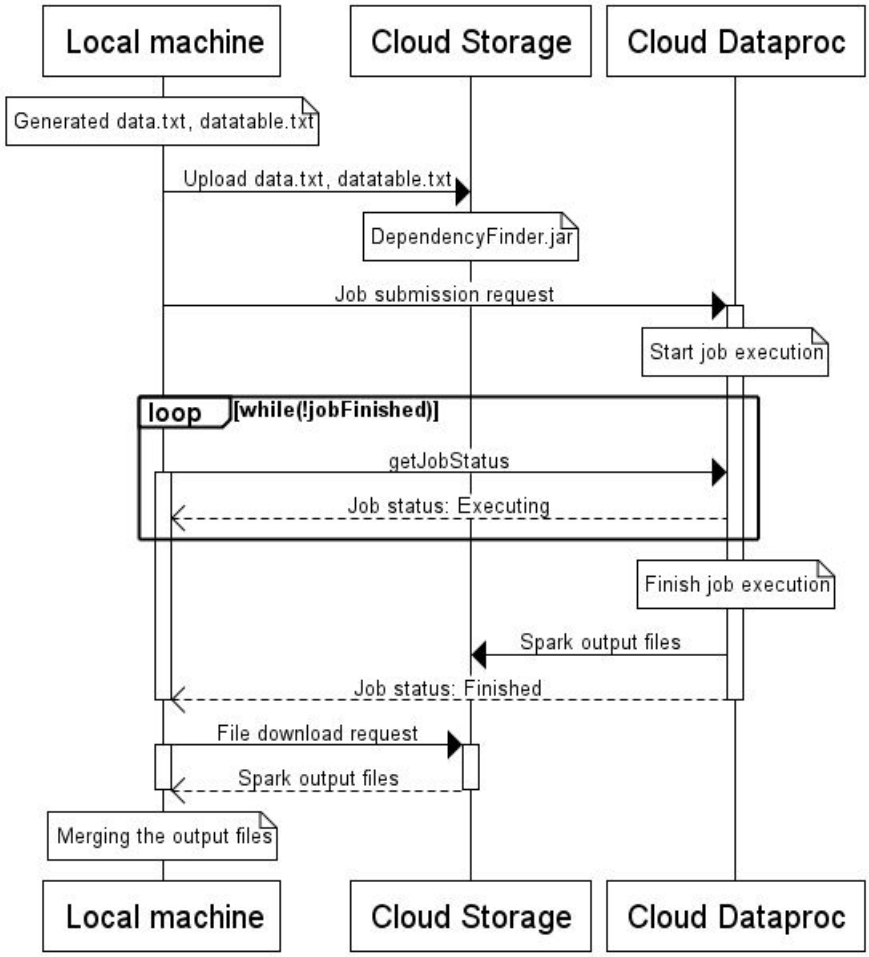


Figure 4.5: Cloud Sequence Diagram

of the system can be illustrated by a UML Sequence Diagram, depicted in the figure 4.5. Whenever a DepFinder class must be executed, the files that are input to it ("data.txt" and "datatable.txt") are first uploaded to the Cloud Storage bucket. Then, the Spark job is submitted for execution to the Cloud Dataproc cluster. The local machine is querying the job status, in order to know when the output is

ready. Finally, when the job is complete, the output files (containing the information about the independent sections) are generated in Cloud Storage, from which they are then fetched by the local machine and merged into one. The resultant file is called "merged_file.txt" and it contains the number of sections in the program and the information about the independent sections.

All the file manipulation and job submission operations are performed by the system automatically, utilizing Google Cloud API client libraries. The classes responsible for actions related to Cloud operations are: FileUploader, JobSubmitter and FileDownloader, while the class responsible for merging the output files is FileMerger.

4.1.7 Assigning threads for executing independent sections

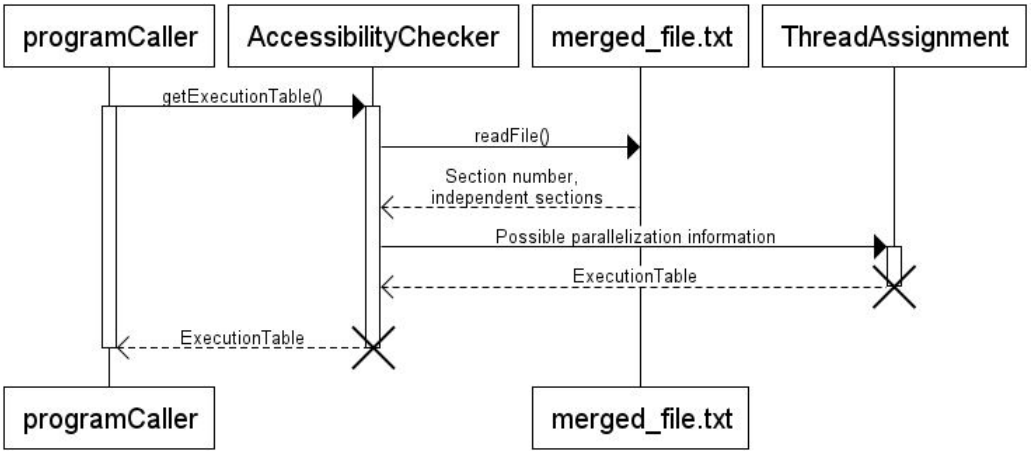


Figure 4.6: Getting ExecutionTable

Once the information about the independent sections is obtained, it becomes possible to decide on how to distribute the sections in a parallel code: i.e. which sections should be executed in parallel and how to order them.

The AccessibilityChecker class is used to find potentially parallelizable sections based on the information obtained from DepFinder class. It gets this information from "merged_file.txt" file and creates a table, containing possibly parallelizable sections. This table is passed to ThreadAssignment class.

ThreadAssignment uses information obtained from AccessibilityChecker to create an "ExecutionTable", which contains a complete execution order of the sections. Finally, the "ExecutionTable" is passed to programCaller.

4.1.8 Generating code

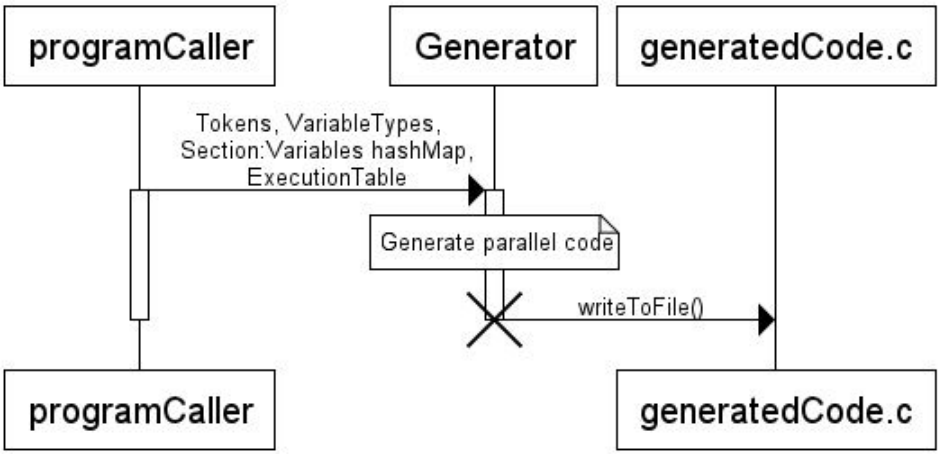


Figure 4.7: Generating parallel code

At this point, the system has all the information required for generating parallel code. Tokens of different parts of the program have been obtained from LexAnalyser, CodeSplit and SectionSeparator classes, while variable types have been detected by Harvester.

The HashMap linking sections to variables it contains has been returned by RWrec class. Finally, ExecutionTable has been created by AccessibilityChecker and ThreadAssignment classes, utilizing DepFinder class, which was executed in the cloud. Thus, figure 4.7 shows the last stage of the system, in which Generator class, having all of the needed information, generates a "generatedCode.c" file, containing the generated parallel code.

4.2 Cloud Overview

The Google Cloud Platform is used to execute a DepFinder class, which utilizes the Apache Spark framework and is designated for finding independent sections in the source code.

Out of multiple services offered as a part of Google Cloud platform, the ones used for the project are:

- Cloud Storage
- Cloud Dataproc

4.2.1 Cloud Dataproc

Cloud Dataproc is an Apache Spark, Hadoop, Pig and Hive service. It allows creating and managing clusters of computers, allowing efficient Big Data manipulations. In the context of the developed system this service is the one responsible for executing an Apache Spark Job (Action).

When submitting a Job to the cluster, it must be contained within a .jar file, which must contain a .java file with the source code to execute, as well as any third party libraries used in that code. A benefit of Dataproc is that Spark is already preinstalled on all cluster computers: therefore, it is not required to add a massive (approximately 150 MB) .jar file with the prebuilt Spark library when submitting a Job for execution.

4.2.2 Cloud Storage

Cloud Storage is an object storage service. It can be used to store files with different levels of availability and durability.

In order for a Spark application to be able to receive input in form of files, these files must be first uploaded to a Cloud Storage bucket - a logical unit of storage. The same applies to the .jar file containing the application source code, in case it needs to be submitted for execution. The output generated by the Spark application is stored in the Storage as well.

4.3 Lexical Analyser

A scanner or lexical analyser is typically the first part of a compiler, which has the task of transforming the stream of characters which make the source code into groups of characters in meaningful sequences called lexemes. [2]

Some compilers implement a **(token-name, attribute value)** output, which is used to keep track of known variables, as well as their sequence, and mapping operators and constants. Our implementation of lexical analysis simply marks certain lexemes that are of special interest to the syntax analysis that follows. The lexemes are variables, whose identifiers are marked by a '$' character preceding the name, and function calls, which are marked with a '?' before the token-name.

C89 keywords are automatically detected and left unmarked. Custom keywords (datatypes and constants), which are *defines*, *enums*, *typedefs*, *structs* and *unions*, are not known at the time lexical analysis happens, so they are temporarily marked as

auto	case	const	default
double	enum	float	goto
int	register	short	sizeof
struct	typedef	unsigned	volatile
break	char	continue	do
else	extern	for	if
long	return	signed	static
switch	union	void	while

Figure 4.8: List of C89 keywords[27]

variables, but later filtered after a syntax analysis process which targets the relevant grammars related to these types.

Another issue for the lexical analyser is the removal of white space characters (space, '\n' , '\t') and comments. Many languages, including C, allow arbitrary amounts of white spaces (with exception of after a define identifier), which can be removed, unless they are explicit chars or part of strings. Comments are irrelevant to the syntax analysis, and thus can be ignored completely. By eliminating these tokens at the lexical level, the system avoids having to compute and detect additional grammars later in the parser, which might be complicated.[2] Usually, line numbers are tracked for error messages, however, the system eliminates newlines using the same rules as other white space character. Almost all C grammar is indifferent to line breaking, though there are two exceptions: includes which must be on a separate line, and defines which must be one line, unless the break line characters are escaped, as seen in the following code.

```
...
//Example of a legal multiline define
#define MAX(a, b) \
       a > b ?    \
       a : b
...
```

This lexical analyser uses lookahead in order to distinguish, for example, function calls from variables.

The first step in creating the lexical analyser was to design a state machine which would be able to detect all token combinations. In order to determine the transitions in states, groups of characters and groups of characters where defined. The resulting groups were (in either list or regex):

- Whitespace:

  '\n ' ' ' (Unicode U+0020)
  '\t '

- Lowercase alphabetic:

  $[a..z]$

- Uppercase alphabetic:

  $[A..Z]$

- Digit:

  $[0..9]$

- Alphanumeric:

  $[A..Z] \cup [a..z] \cup [0..9]$

- Special Symbol:

[	{	?
]	}	:
(	,	
)	;	

- Operator:

+	/	!
−	%	^
|	<	~
&	>	
*	=	

- Double operator:

++	*=	>=
−−	/=	^=
==	&=	
||	|=	~=
&&	%=	<<
+=	!=	
−=	<=	>>

- Triple operator:

<<=	>>=

- Comment:

//	/*	*/

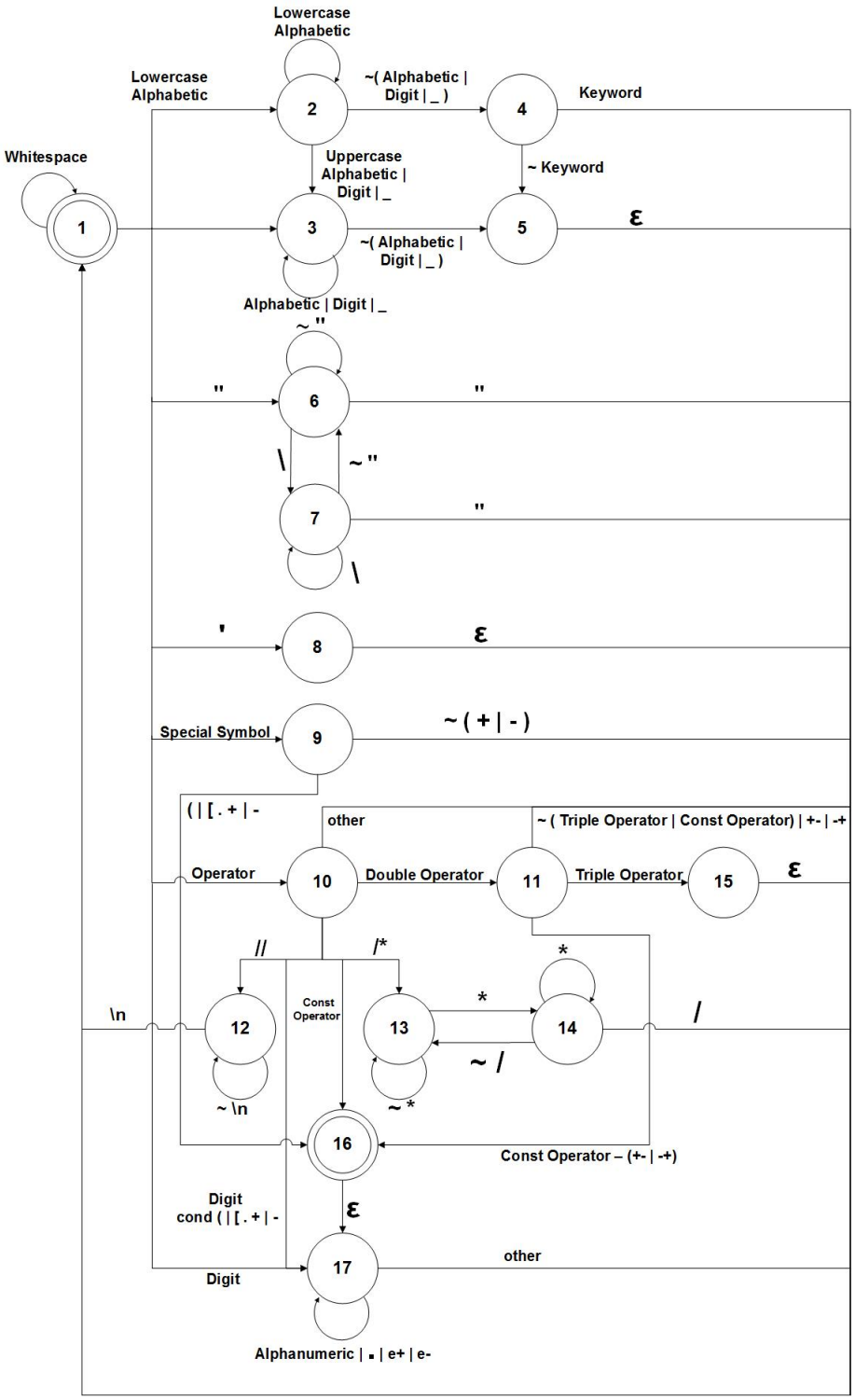
- Constant Operator:

+−	|+	^+
−+	|*	^−
=+	&−	
=−	&+	!+
*+	%+	!−
*−	%−	~+
/+	<+	
/+	<−	~−

One group that is slightly different from the rest is the members of the constant operator group, which are technically not tokens, but a list of the possible scenarios where two consecutive operators are part of separate tokens, and the second token is a constant starting with a sign (+ or -). The same scenario must also be detected as a transition from the special symbol group, in particular round and square brackets, but excluding the scenario where there is a pre-increment after the bracket.

The resulting state machine is in figure 4.9. Following is a general description of the actions of each of the states:

Figure 4.9: State Machine for Lexical Analyser



State 1: Start of new token

Data: Current Char *current*, Accumulated Chars *accumulated*, List of Tokens *tokens*
Result: Next State *state*
if *accumulated* $\neq$ "" **then**
  add *accumulated* to *tokens*;
  accumulated $\leftarrow$ "";
end
return *state* according to transitions and *current*;

State 2/3/10/11/15/17: Keyword or Identifier/Only Identifier Possible/Operator/Double char Operator/Triple char Operator/Numeric Constant

Data: Current Char *current*, Next Char *next*, Accumulated Chars *accumulated*
Result: Next State *state*
accumulated $\leftarrow$ *accumulated* + *current*;
current $\leftarrow$ *current.next*;
return *state* according to transitions and *current*;

State 4: End of a Keyword or Identifier

Data: Accumulated Chars *accumulated*
Result: Next State *state*
if *accumulated* ***is*** *C89 keyword* ***OR*** *"define"* ***OR*** *"include"* ***OR*** *"main"* **then**
  state $\leftarrow$ 1;
else
  state $\leftarrow$ 5;
end
return *state*;

State 5: End of an Identifier or Function

Data: Current Char *current*, Next Char *next*, Accumulated Chars *accumulated*
Result: Next State *state*
if $current.next = '('$ **then**
  $accumulated \leftarrow '?' + accumulated$;
else
  $accumulated \leftarrow '\$' + accumulated$;
end
$state \leftarrow 1$;
return *state*;

State 6: Start of a String

Data: Current Char *current*, Next Char *next*, Accumulated Chars *accumulated*
Result: Next State *state*
$accumulated \leftarrow accumulated + current$;
if $next = '"'$ **then**
  $accumulated \leftarrow accumulated + current.next$;
  $current \leftarrow current.next.next$;
  $state \leftarrow 1$;
else if $next = '\backslash\backslash'$ **then**
  $state \leftarrow 7$;
  $current \leftarrow current.next$;
else
  $state \leftarrow 6$;
  $current \leftarrow current.next$;
end
return *state*;

State 7: Excaped char state to avoid early string termination in case of \"

Data: Current Char *current*, Next Char *next*, Accumulated Chars *accumulated*
Result: Next State *state*
$accumulated \leftarrow accumulated + current + current.next$;
$current \leftarrow current.next.next$;
if $current = '\backslash\backslash'$ **then**
  $state \leftarrow 7$;
else if $current = '"'$ **then**
  $accumulated \leftarrow accumulated + current$;
  $current \leftarrow current.next$;
  $state \leftarrow 1$;
else
  $state \leftarrow 6$;
end
return *state*;

State 8: Start of a char

Data: Current Char *current*, Next Char *next*, Accumulated Chars *accumulated*
Result: Next State *state*
$accumulated \leftarrow accumulated + current$;
$current \leftarrow current.next$;
if $current = '\backslash\backslash'$ **then**
  $accumulated \leftarrow accumulated + current + current.next + current.next.next$;
  $current = current.next.next.next$;
else
  $accumulated \leftarrow accumulated + current + current.next$;
  $current = current.next.next$;
end
$state \leftarrow 1$;
return *state*;

State 9: Special Symbol

Data: Current Char *current*, Next Char *next*, Accumulated Chars *accumulated*
Result: Next State *state*
accumulated $\leftarrow$ *accumulated* + *current*;
if (*current* $= '($ ***OR*** $'[')$ ***AND*** (*current.next* $= '+'$ ***OR*** $'-')$ ***AND*** (*current.next.next* $\neq '+'$ ***OR*** $'-')$ **then**
  state $\leftarrow$ 16;
else
  state $\leftarrow$ 1;
end
current $\leftarrow$ *current.next*;
return *state*;

State 12: Single Line comment

Data: Current Char *current*, Next Char *next*, Accumulated Chars *accumulated*
Result: Next State *state*
accumulated $\leftarrow$ "";
current $\leftarrow$ *current.next*;
if *current* $= '\backslash n'$ **then**
  state $\leftarrow$ 1;
else
  state $\leftarrow$ 12;
end
return *state*;

State 13: Multi Line comment

Data: Current Char *current*, Next Char *next*, Accumulated Chars *accumulated*
Result: Next State *state*
accumulated $\leftarrow$ "";
current $\leftarrow$ *current.next*;
if *current* $= '*'$ **then**
  state $\leftarrow$ 14;
else
  state $\leftarrow$ 13;
end
return *state*;

State 14: Multi Line comment possible end

Data: Current Char *current*, Next Char *next*
Result: Next State *state*
current ← *current.next*;
if *current* = $'/'$ **then**
  state ← 1;
  current ← *current.next*;
else if *current* = $'*'$ **then**
  state ← 14;
else
  state ← 13;
end
return *state*;

State 16: Transition from operator into a constant

Data: Current Char *current*, Accumulated Chars *accumulated*, List of Tokens *tokens*
Result: Next State *state*
if *accumulated* ≠ "" **then**
  add *accumulated* to *tokens*;
  accumulated ← "";
end
state ← 17;
return *state*;

One of the most complex tasks of the scanner is detecting lexical errors in lexemes for numbers. This is because the possible grammars for numbers is very extensive. The basic format can vary depending on if the number is signed, the base (hexadecimal, octal, decimal), in scientific notation and optional additional markers to note whether the number is long or unsigned (which can be in any order and uppercase of lowercase). The format can be synthesized with the following regular expression:

$$(+|-)?.(0|(0.(x|X)))?.([0..9] \cup [a..f] \cup [A..F])^* \\ .((e|E).(+|-).[0..9]^*)? \\ .(((L|l)?.(U|u))?|((U|u)?.(L|l)?))$$

Since our framework does not evaluate errors, it assumes the code input is compilable; then the lexical analyser only needs to detect when a number starts, and where it ends.

States 5, 14, 15 and 16 are epsilon transition states, meaning that no progress is made in them in terms of advancing the current char. Thus it would be possible to incorporate their actions into the transitions of the previous states. This would result in fewer states in the machine, but the trade off would be making some already complex states even larger. Having these transition states also allows for easier debugging and more intuitive comprehension of the process, specially when transitions require multiple actions to be taken.

Another issue with the lexical analyser is that it attempts to mark variable identifiers, however as mentioned before, it will mark user defined data types and constants as variables, even though they are not.

The lexical analyser will detect and properly tokenize two of the operators outside the C subset, the pointer to operator (*) and the address operator (&). The only operator that is not properly tokenized is the structure dereference operator (->), which could be detected by adding it to the list of legal double operators if the C subset is expanded. Furthermore, compiler specific types, such as GCC's __complex__ and integer types; or keywords, such as the GNU extension's _FUNCTION__, __PRETTY_FUNCTION__, __alignof, etc.

4.4 Sectioning and Pre Code/Section 0 Processing

4.4.1 Splitting the Code

The input to the framework, as established by the rules of our C subset is split into 4 sections:

Precode: All code before main

Section 0: Code inside main that initializes variables or creates defines

Maincode: Code inside main that is not part of Section 0

Postcode: Code that is after main, for example functions that have been predeclared.

Thus, one of the first tasks is recognizing the delimitation between these sections, in order to further process the precode and Section 0 tokens, and send maincode further through the system. This task is done after tokenizing, since it avoids any wrong detection of specific patterns that are searched for in the program.

Delimiting the first section (precode) involves iterating through the tokens until the declaration of main is found, and tokens are included until the main function

starts (first "{" after the token "main"). Since the declaration of main is standardized by the defined C subset grammar, the system must remove the whole declaration of main. This is done by iterating backwards while removing elements until the "main" token is found. Then while the tokens are consecutive keywords (most commonly "int", but not limited to it) it will remove them, since these are the data type which is returned by the program. A considered alternative would be simply to iterate backwards until the ";" token is found, however this is not a viable alternative because define and include do not use ";" to mark the end of their grammar.

Section 0 is detected because we have the knowledge that every grammar it uses will start with a keyword, more specifically either "#define", "typedef", a basic data type or custom defined data type (name of a struct, union, enum, typedef). Thus detection of Section 0 must occur after the processing of the Pre Code, considering the new keywords when it verifies if the start of the grammar is a keyword. It must also add the keywords as it finds them to the list of keywords, although this has not been implemented in the current version. This would require doing successive Section 0 delimitation and Section 0 processing for keywords in order to verify that Section 0 did in fact end where the program said it did. An example of when this scenario would happen is the following code:

```
#include <stdio.h>
typedef unsigned int uint;
int main(int argc, char ** argv){
    //uint is a keyword detected at Pre Code processing
   uint foo = 1;
   //typedef is a keyword because of C89
   typedef unsigned long ulong;
   //ulong is a keyword, but has not been processed yet, so it will be
       incorrectly allocated into Main Code
   ulong bar = 10;
   //Main Code should start here
   foo++;
   return 0;
}
```

and the output tokens are:
Section 0: [$uint, $foo, =, 1, ;, typedef, unsigned, long, $ulong, ;]
Main Code: [$ulong, $bar, =, 10, ;, $foo, ++, ;, return, 0, ;]

While the program does detect after the processing that ulong is a data type, it has already sectioned the code, and thus the line will stay in Main Code. This causes an issue, because the framework only looks for variable initializations in Section 0. Then the hash map containing all the variables and their types which

is used for code generation will not contain the variable **bar**, which will cause the generated code to fail. This is the reason behind the rule in our C subset that typedefs and structs/unions/enums must be defined in precode, although a later version could allow these inside of main. If the rule is removed, there will still be a problem with implementing the program in a parallel form. Since parallelization is done using functions, these will not know the size of, for example, structs that they are using, because they are defined later in the program; and will generate errors at compile time. Thus typedefs/structs/unions/enums must be written in precode as established by the rule, or alternatively the framework must have an extra module for processing Section 0, which will detect these grammars and move them to precode. This alternative would of course also require the previously discussed interaction between Section 0 delimitation and processing the Section 0 code.

Returning to the process of finding where section 0 ends: this can be done by finding the first time "typedef", a basic datatype or custom defined data type does not come after a semicolon. This will be able to detect the end of Section 0 in every scenario with the exception of the one forbidden by the C subset rule that states that Section 0 must not end with a define statement. This once again is because the define grammar does not end with a semicolon, but with a line break that is not after a \.

Since the tokenizer has eliminated line breaks, it is not possible to detect the end of the define, and thus it would include the next line in Section 0. While intuitively this would only mean that the first line will be sequentially executed, it will cause a bigger issue for code generation. A define statement covers everything in the same line, so it is very important to detect where it ends, and add a line break. The way this is handled is by detecting when there is a keyword after a define, and adding a line break before it. This is why there is a rule that defines are not allowed to use casting, to prevent detecting keywords inside the defines. Because the code in main can start with a variable, it is very complex to find the end of the define without having tracked line numbers. The rule that the last line in section 0 may not be a define, prevents this scenario. Alternatively, generating a parse tree or tracking line numbers could allow the removal of this rule, but this is not implemented in the current version.

The next task is to detect where Main Code ends, and this is done by tracking "{" and "}" tokens, until the main has been closed. Finally, postcode is everything that comes after Main Code.

4.4.2 Precode and Section 0 Processing

Processing of precode and Section 0 has three functions. The first one is the recognition of grammars which create constants or new data types, so they are not incorrectly marked as variables later. The second task is to make a hash map of the created variables and their data types, and mark variables that are arrays by a '%' char after the data type. The third task is to put "\n" tokens at the start of a new grammar. This is necessary especially for statements after defines or includes, which do not mark the end of the statement by a semicolon.

The statements we are looking for are include, define, typedef, enum, struct, union and global variables.

Handling of include grammar

Once a "#include" token is encountered, the program can jump ahead 4 tokens, since the include statements will have 4 tokens ("#include", "<", "library.h", ">" OR "#include", """, "library.h", """). The program does not introduce break lines between include statements, since this is done in the code generation process.

Handling of enum grammar

When an "enum" token is found, a two token lookahead is used to check if it is a "{" token. This is done to distinguish between a global variable of a predefined enum, or a declaration of a new enum type. If it is a global variable, it will behave as a normal global variable, with behaviour as described in the global variable handling subsection described later this chapter.

Otherwise a "\n" token is inserted before the enum token to mark the start of a new statement. Then the token after the "enum" token must be the type name of the enum, and thus is added as a new keyword. The next task is to add all the enum constants identifiers as detected constants, and this is done by adding the first token after "{", and subsequently every token that is after a "," token, before the "}" token is found. This ensures that only the identifiers are added, and the optional assignment tokens are skipped. The next task is to find the end of the statement by iterating through tokens until a semicolon is found. There may be additional tokens between the "}" and ";" tokens, but these will be ignored by the compilers, since the typedef grammar was not used. Following is an example of an enum with the keywords and constants that would be added.

```
enum days {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY= -2,
    SUNDAY = -1,
};
//New Keywords: days
//New Constants: MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
    SUNDAY
```

Handling of struct/union grammar

When a "struct" or "union" token is encountered, it is checked if it is a global variable or a declaration by looking two tokens ahead for a "{" token. Since unions have several variables located in the same memory, they are dependent, and must be treated as the same variable. Thus the program uses the conservative concept that modification of any field in a struct or union is considered to be using the whole struct/union, in a similar way to how arrays are handled. Thanks to this assumption, nesting structs/unions will never cause any issues.

Thus structs and unions are handled by adding the token after the struct/union token to the list of keywords, and not adding anything from inside it by skipping until the curly bracket is closed, unless it is an embedded structure (like in the example). Also aliases are skipped since no typedef is used. An example of a struct is shown in the following code, as well as the generated output.

```
struct date {
    int yearNum;
    int monthNum;
    int dayNum;
    enum days weekday;
    union data{
        int i;
        float f;
        char c;
    };
    struct other_struct instance;
} Invalid_alias;
//New Keywords: date, data
//New Constants:
```

A "\n" token is inserted before the struct/union token too, to mark the start of the statement.

Handling of typedef grammar

The typedef statement starts by adding a "\n" token before the "typedef" token. Then it evaluates according to the enum, struct, union grammars, with the difference that it also adds aliases, by adding non comma tokens after "}" and before ";". If the grammar in the typedef statement does not include the declaration of a new enum, stuct or union; then all non comma tokens before the semicolon are added as keywords, since if the token is already a keyword this will simply do nothing.

```
typedef struct point_struct {
    int x;
    int y;
    int z;
}point, p;
//New Keywords: point_struct, point, p

typedef unsigned int uint;
//New Keywords: uint (unsigned and int are already keywords!)
```

A "\n" token is inserted before the "typedef" token too, to mark the start of the statement.

Handling of define grammar

Defines can be used to create a constant or a function. Since the tokens that are functions have already been marked with a '?' char, it can be verified that it is a constant, and if so add it to the list of constants which have been found. No matter if it is a function or a constant, we insert a line break before the define token. An example of the define both in the case of a function and constant are shown in the following code.

```
#define ?add(a, b) (a+b) //the ? char has been added by the lexical
    analyser, and is removed at code generation

#define NUMTHREADS 4

//New Constants: NUMTHREADS
```

4.5 Minimum Section Generation

The issue of dividing the code into the most efficient separate sections is a complex task. The whole code in a section will be executed sequentially. Having too small sections will lead to a high overhead caused by initializing many threads, while too large sections can lead to code that cannot be parallelized because of high cohesion between sections (this means that sections may not be parallelized). Another layer of optimization involves attempting to get sections to have similar execution times, so there is maximum usage of the computer cores. The task of generating the optimal sections will require different approaches, depending on the program, and thus it would be better to solve this task by generating a variety of possible sections, and testing the execution times in order to find the optimal solution. In order to be able to generate the most options, it is best to start with the smallest possible sections, and then merge sections gradually to find the optimal solution. In a C program, the smallest sections possible would be one line of code or one loop. Following is an example of how the code in main would be split into sections.

```
//Section 0
i = 0;                  //Section 1
while(i<20){            //Section 2
    printf("%d\n", i);
    i++;
}
point1.x = 1;           //Section 3
...
```

Thus the minimum section generation marks lines and loops with a @ token, and later in the process some of these tokens can be removed to create larger sections. Detection of code lines and loops is done by tracking curly brackets and semicolons. The only special behaviour that must be taken into account is the for loop, which uses semicolons in its syntax.

Also, since this program iterates through the tokens and has all the new keywords and constants from precode and section 0 processing, it can remove the wrongly marked tokens, which were thought to be variables, but were not.

4.6 Pattern Recognizer

Pattern recognizer program is responsible for determining access types of the variables based on the tokens they are surrounded by. Pattern recognizer deals with tokens provided by Scanner and can be considered a minimalistic version of a parser. It involves similar techniques but for a much more narrow task. Pattern recognizer does not find or correct errors in the input source code, since the latter is assumed do be initially correct (that is, successfully compiled). The program was designed as a state machine, which is provided in the figure 4.10.

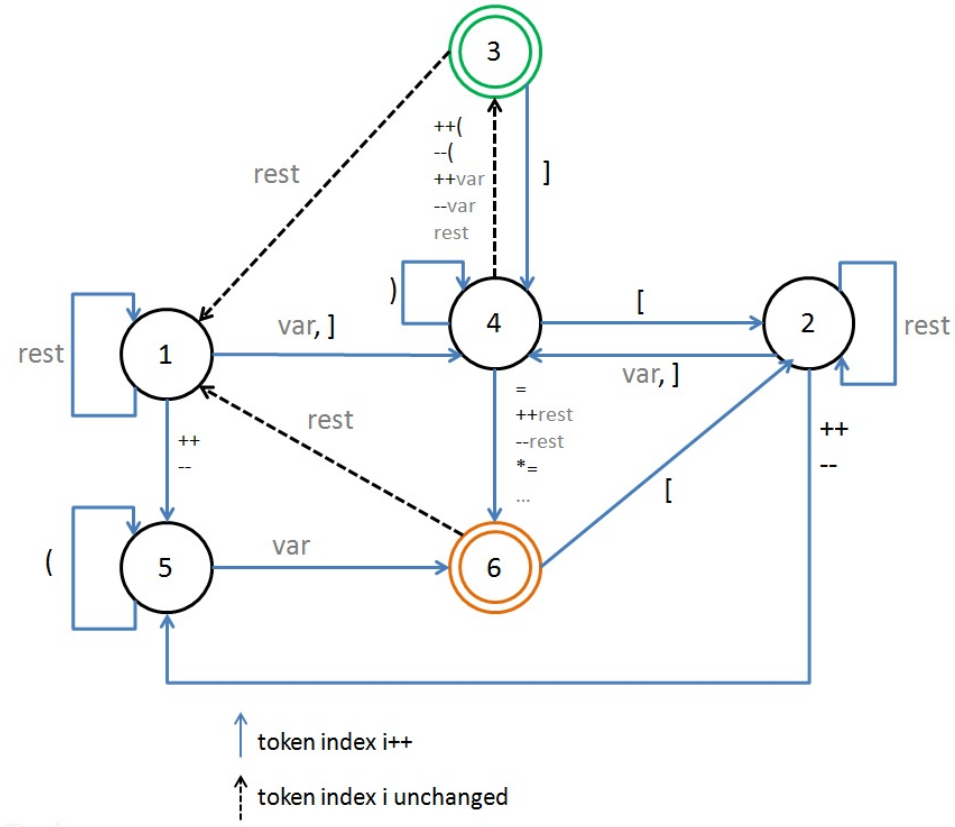


Figure 4.10: Pattern recognizer implemented as a state machine.

Gray captions do not represent tokens themselves, but rather types of tokens. Thus, "rest" means the rest - all other tokens, and "var" stands for a token which is a variable name. At last, three dots "..." on 4 to 6 transition say that there are more tokens which will lead from 4 and 6. These are listed in the corresponding data holder **wrtops** of Pattern recognizer class (RWrec.java). A token like "++(" or simi-

Table 4.1: Table of states of Pattern Recognizer state machine

State	General information	Input action	To state
1	Start state, looking for w/r expression.	Var: temp = var; i++	4
		"]": temp = stack.peek(); stack.pop(); i++	4
		"++", "−−": i++	5
		"@": secnum++; i++	1
		The rest: i++	1
2	Array handler.	Var: temp = var; i++	4
		"]": i++	4
		"++", "−−": i++	5
		The rest: i++	2
3	Reading detected. temp goes to the holder.	"]": temp = stack.peek(); stack.pop(); i++	4
		The rest: no action	1
4	Intermediate state.	Writing operator: i++	6
		"[": stack.push(temp); i++	2
		")": i++	4
		"++", "−−" followed by "(" or var as look ahead: no action	3
		"++", "−−" followed by any other token as look ahead: i++	6
		The rest: no action	3
5	Writing token before variable.	Var: temp = var; i++	6
		"(": i++	5
6	Writing detected. temp goes to the holder.	"[": stack.push(temp); i++	2
		The rest: no action	1

lar (state 4 to state 3 transition) is not one token. Instead, it is a "++" token followed by a "(" token, which is obtained thanks to the look ahead technique described in implementation part. Similarly, in state 4 to state 6 transition, "++rest" is "++" followed by any token different from "(" and "var", since these are considered in 4 to 3 transition.

The state machine table is also available in table 4.1. It describes shortly what each state does. The details are discussed in Pattern Recognizer implementation section.

4.7 Spark Application

4.7.1 Overview

For determining section dependencies of the input code it was decided to use Apache Spark, which is an engine for big data processing. Even though the tested input codes are not big enough to be considered big data, it is still reasonable to use Spark, because even a relatively short program can deal with many variables and have many inner dependencies. Moreover, the algorithm stays the same regardless to the size of input, which makes a truly big data analysis be a further application of the developed system.

The basic idea behind choosing Spark was imagining input source code as text. A known example of Spark is counting words in a large text or processing a large database. Imagining source code as text makes the process similar to what Spark has proven to be good at.

Spark application runs on a cluster on Google Cloud platform and takes three inputs: number of section, data.txt and datatable.txt text files. Both files are also placed on cloud. Figure 4.11 shows the algorithm of the developed Spark application.

4.7.2 Algorithm

Phase 1

The process starts making a pair RDD out of a text file "data.txt". The format of the RDD is <variable : section> , which signifies a variable and a section it is found in. Since the same variable can be found in several section, it makes sense to use to group values by key, using the variable as a key and its section as a value. The result of this operation is an RDD of format <variable : Iterable <section».

Phase 2

An important manipulation is performed on the produced RDD here. It is needed to take all unique combinations of two elements of the variable collection. As long as Spark is also capable of creating pair of elements of two RDDs, it is reasonable to first convert iterable collection of sections to a list of sections and then to an RDD. By performing some additional list-to-RDD and RDD-to-list conversions (described further in the implementation section) an RDD of format <variable : section : section> is obtained. It represents a collection of two sections, which have the common variable.

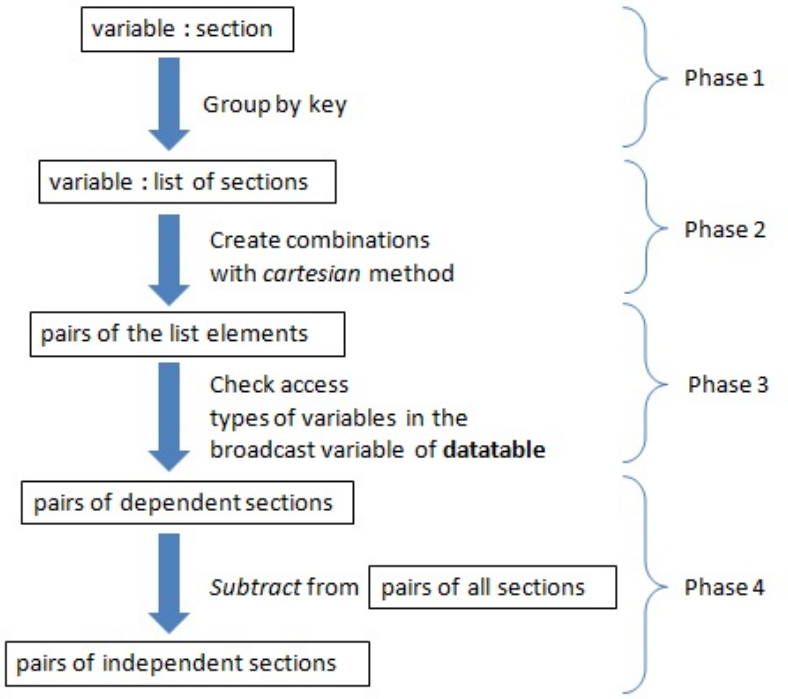


Figure 4.11: Spark application schematics.

Phase 3

Phase 3 is responsible for checking, whether two sections, having a common variable, are dependent with each other. On one hand, if both read from the same variable, then this variable is shared by these sections, but it does not make them dependent, so they will be able to run in parallel. On the other hand, if at least one section from the <variable : section : section> format RDD reads from the variable, then these sections are considered dependent. In this way, RDD elements of independent sections are filtered out at this stage, whose outcome is an RDD of dependent section pair with format <section : section>.

Phase 4

The overall number of section in the input code is given to the Spark application as a parameter. The number of sections is used to compose an RDD with elements of format <section : section> which represent unique two element combinations of all sections. In phase 4, the RDD of dependent section pairs (produced at stage 3) is subtracted from the RDD of all section pairs. Thus, an RDD of independent section pairs is the result of this subtraction. Independent sections are then saved in a text file and are passed to the next the next part of the system afterwards. It

will decide how exactly to run these independent sections in parallel.

4.7.3 Spark application run on an example

Consider the following code example. Note, that initialization section is omitted, and the code itself does not serve any meaningful purpose. It is simply a bunch of operations made up for the proof of concept.

```
main{
....
   v = 2.0;                A
   x = 1;                  B
   print(x);               C
   z = func1();            D
   if (z > x) {
      y = false;           E
      print(y);
   }
   while (x==1){
      func2(v);            F
   }
}
```

Figure 4.12: An example of code.

Sections are marked on the right side and are represented by capital letters instead of ordinal numbers to avoid ambiguity. The whole Spark application process is shown for the provided code. Analysis of section B and section E dependency is marked with red color as an example.

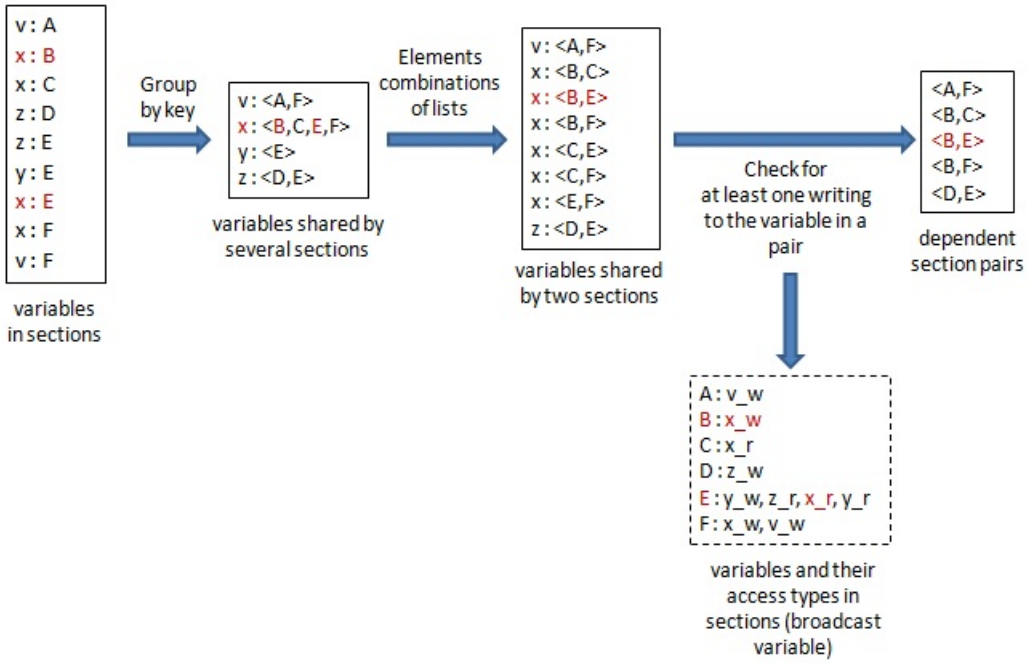


Figure 4.13: Spark application working principle. Part 1.

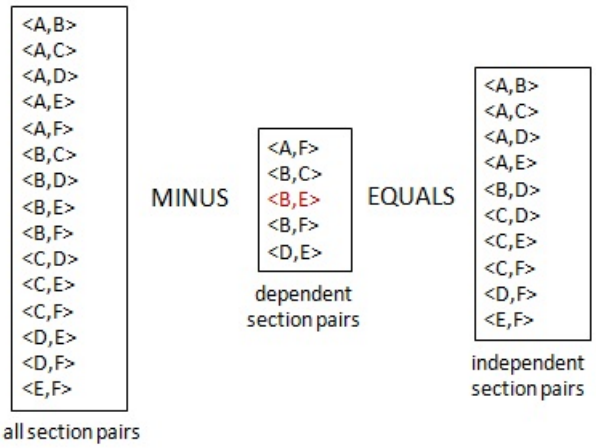


Figure 4.14: Spark application working principle. Part 2.

4.8 Parallelization Reachability Analysis

Once the analysis of independence of sections has finished, the program has available a list of code sections which are not data dependent. The simplest method to get a parallel program that has the same result as the sequential code would be to implement parallel adjacent execution. This means that if section n is independent to section n+1, then they will be executed in parallel, and if section n+2 is independent to both sections, then it will be executed in parallel with both sections. This concept is shown in figure 4.15.

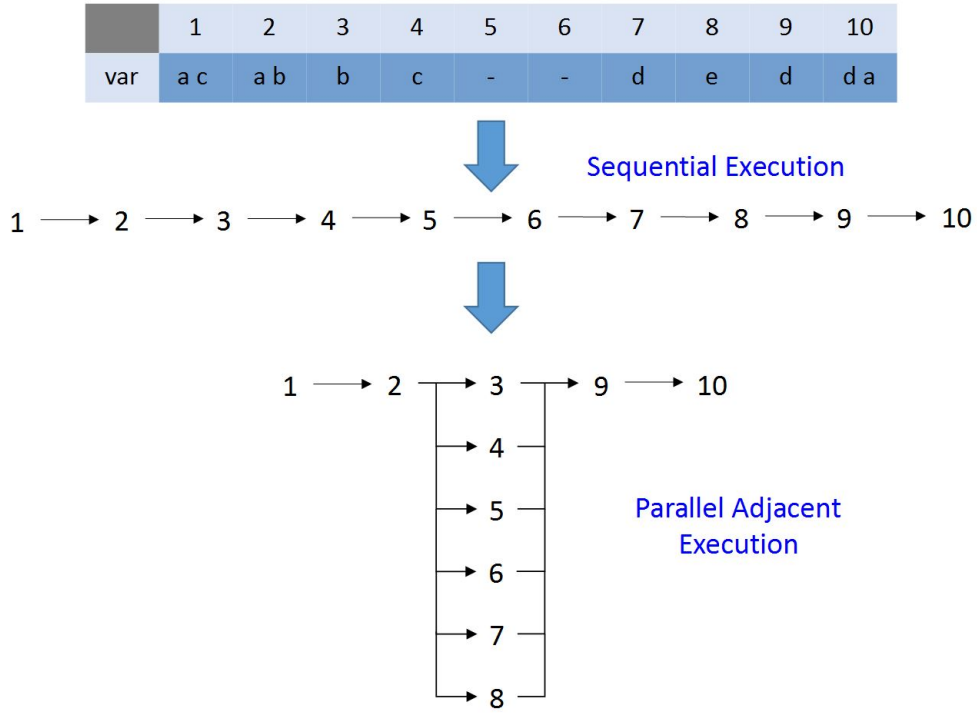


Figure 4.15: Parallel Adjacent Execution

From the parallel adjacent execution there are three optimization operations that can lead to faster execution times.

1. Merging contiguous dependent sections

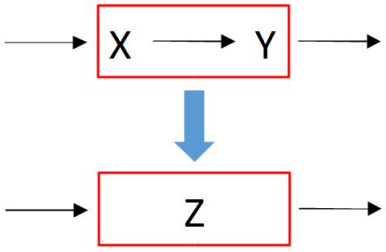


Figure 4.16: Since execution is sequential, there is no need for overhead caused by parallelism

2. Sequentially merge independent parallel sections to fit optimal thread amount

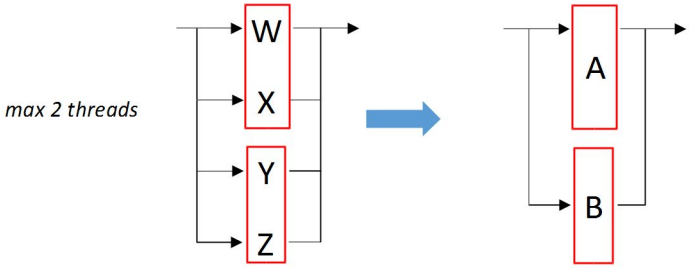


Figure 4.17: Adapt to finite resources

3. Independent section throwing backward/forward

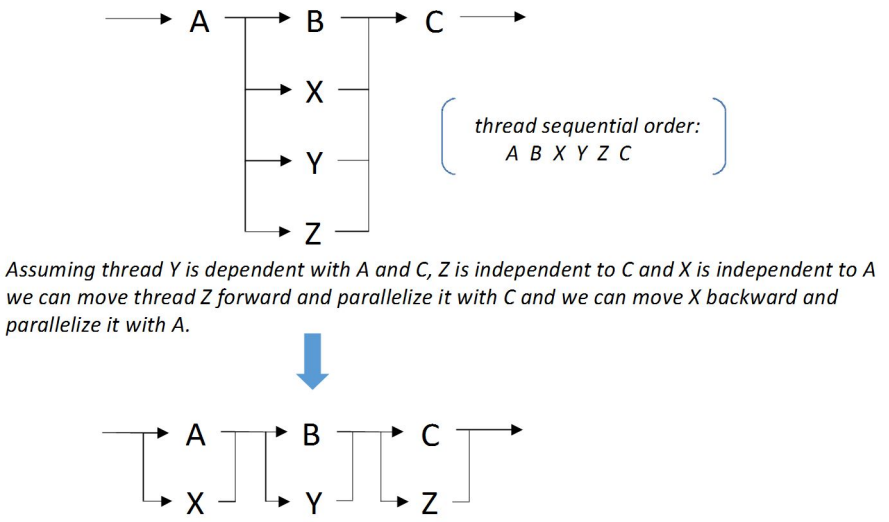


Figure 4.18: Create a more even distribution of sections in each execution cycle to use threads efficiently

The focus of this sub chapter is to explain how to detect the possible third operations in the code, using the independent section list. This section will describe the algorithm used using the variable dependencies given in figure 4.15. The main issue is that independence of two sections does not imply parallelizability. This is because although two sections can be data independent, they can still be dependent on control flow. Consider the following code:

```
...
i = 0;              //section n
if (i != 1){        //section n+1
    a = 10;
}
a++;                //section n+2
...
```

In this example section n is independent to n+2, but both are dependent on section n+1. If the code was executed sequentially, i would be 0, and a would be 11. On the other hand, if section n and n+2 are executed in parallel before section n+1, i would be 0 and a would be 10. This is one of the scenarios which are detected by the Parallelization Reachability Analysis.

Parallelization Reachability Analysis starts by generating a table from the independent pairs, and then checks backwards and forwards whether the independent pairs are parallelizable. If an independent pair cannot reach each other, they are in red color, and if they can they are in green. Figure 4.19 and 4.20 show both the

forwards and the backward reachability analysis.

section	1	2	3	4	5	6	7	8	9	10
Independent to (section > index)	=									
	-	=								
	3	-	=							
	-	4	4	=						
	5	5	5	5	=					
	6	6	6	6	6	=				
	7	7	7	7	7	7	=			
	8	8	8	8	8	8	8	=		
	9	9	9	9	9	9	-	9	=	
	-	-	10	10	10	10	-	10	-	=

Figure 4.19: Backward execution check

Both tables are generated by iterating through the sections, which have been ordered using an insertion sort algorithm, ordering relative to the first number and then the second for backward execution, and opposite for forward execution. Once the non-reachable independent pairs are removed, we get the union of both tables, which is the possible parallelization. In this case the result is in figure 4.21.

The advantage of generating this table is that if sections are merged later in the process, this will only require intersecting the possible parallelizations of both sections. In terms of computations this means that merging sections as in optimization operations 1 and 2 would not require recalculating dependencies and reachability.

Furthermore, this table is used in the process of assigned threads, since it is simple to find where a thread may be allocated in parallel. It also allows quantifying, comparing and optimizing the allocation of threads in the execution table. The way this table is used to generate an execution schedule is described in the next section.

The backwards and forwards tables should be equivalent to the results when performing backwards static slicing and forward static slicing, since now we consider both data and control flow dependencies.

section	1	2	3	4	5	6	7	8	9	10
Independent to (section < index)	=	-	1	1	1	1	1	1	1	-
		=	-	2	2	2	2	2	2	-
			=	3	3	3	3	3	3	3
				=	4	4	4	4	4	4
					=	5	5	5	5	5
						=	6	6	6	6
							=	7	-	-
								=	8	8
									=	-
										=

Figure 4.20: Forward execution check

An overview of the Parallelization Reachability Analysis process is on page 62.

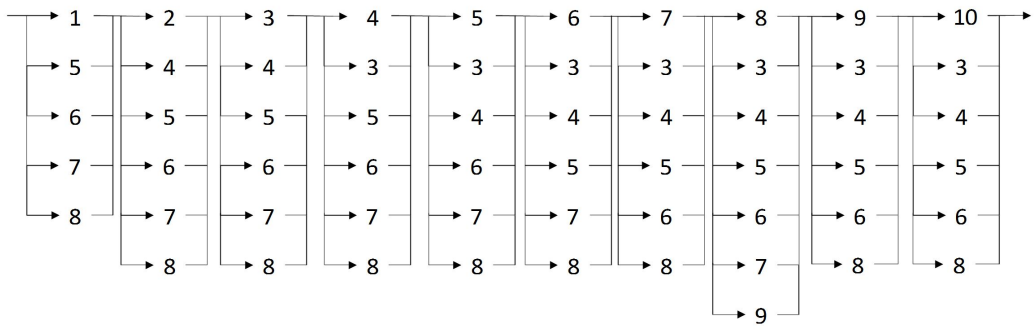
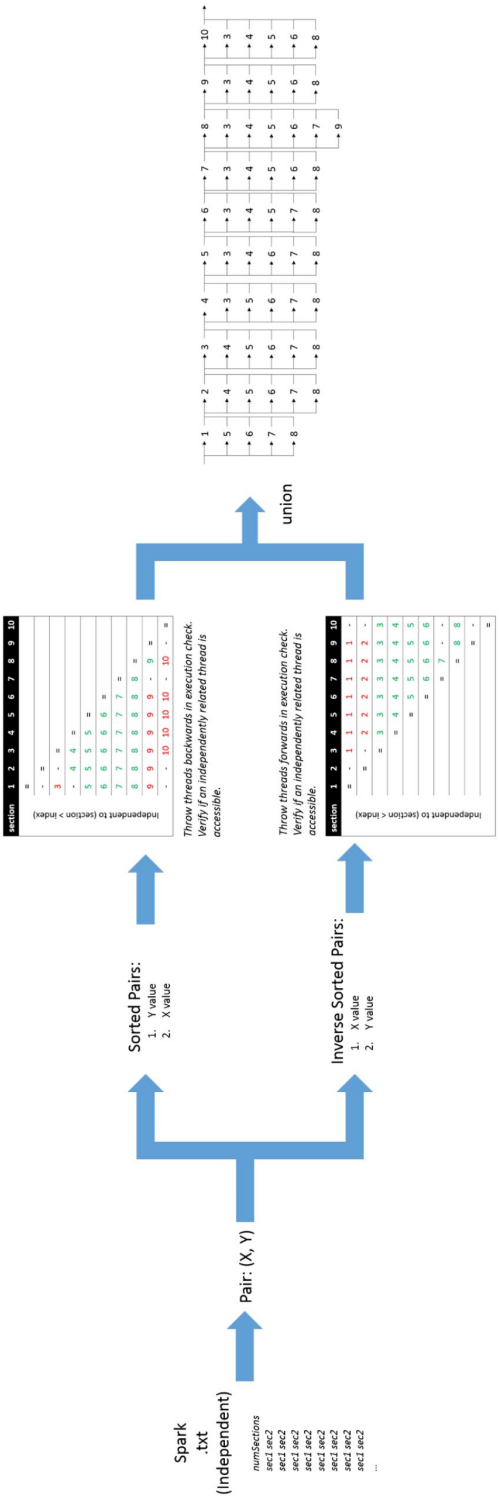


Figure 4.21: Possible Parallelization


Spark
.txt
(Independent)
numSections
sec1 sec2
Pair: (X, Y)
Sorted Pairs:
1. Y value
2. X value
Inverse Sorted Pairs:
1. X value
2. Y value
Throw threads backwards in execution check.
Verify if an independently related thread is accessible.
Throw threads forwards in execution check.
Verify if an independently related thread is accessible.
union

4.9 Thread Assignment

Thread assignment is the task of deciding which sections will be the base sections to parallelize on, and assign the remaining sections to these base sections, until an execution table is achieved. The input to this program is the union table, as shown in figure 4.21.

The first task is to decide the maximum amount of threads to run in parallel, and this is done by using the Runtime instance to detect the amount of processors that can be used by the JVM. Next it is necessary to quantify the characteristics of each section. The information that is of interest is how difficult it might be to assign a thread in parallel to another, so we count how many sections have each number in the list of threads that they can be parallelized with. In the example in figure 4.21, we get the following coefficients:

External Linking Coefficients (ELC):

Section 1: 0

Section 2: 0

Section 3: 7

Section 4: 8

Section 5: 9

Section 6: 9

Section 7: 7

Section 8: 9

Section 9: 1

Section 10: 0

From these coefficients, it is concluded that sections 1, 2 and 10 cannot be parallelized into any sections, and thus they must be base threads or parent threads, and have other threads assigned to them. Intuitively, if we have a maximum of n parallel threads executing simultaneously, then there must be at least x/n parent threads where x is the number of sections.

To explain the algorithm, n will be set at 2, and x is 10. In this example, there have to be at least 5 parent/base threads, and there are only 3 sections which must

be parents, so the program must decide on two more parents. This is done by taking the next 2 lowest external linking coefficient sections, which would be section 9 with coefficient of 1, and a tie between section 3 and section 7 with coefficient 7. When there is a tie, the deciding factor will be the internal linking coefficient (ILC), which is simply the size of the list. Since they have the same ILC, then the first number is taken, in this case 3.

An example of ILC and ELC is shown in figure 4.22

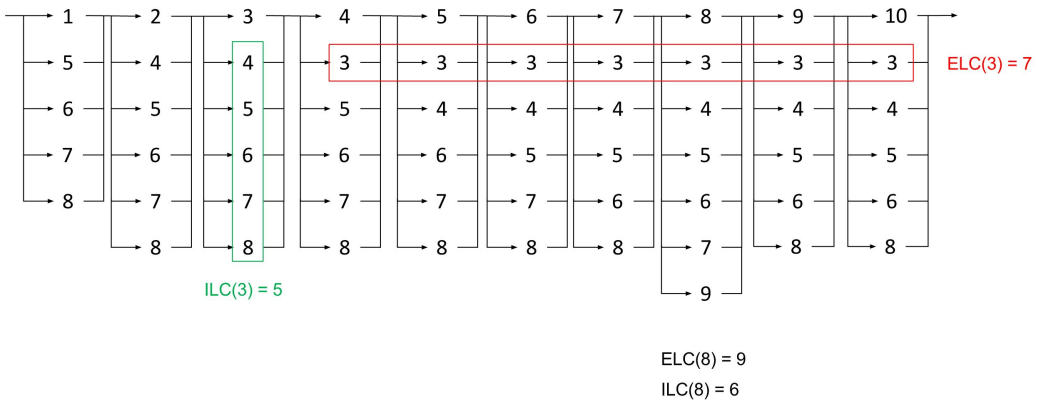


Figure 4.22: ILC and ELC

The next step is to sort the parents in order of execution, instead of ELC, so in this example the parent used to be 1, 2, 10, 9, 3; but after a merge sort algorithm are 1, 2, 3, 9, 10. Once this is done, the next step is to iterate through the other sections, and place them optimally. The first step is to make a list of current parents which could be parallelized with the current section to be assigned. From this list, parents that already have the maximum amount of parallel sections connected to them (in this case 1 section, since n = 2) must be filtered out. In case that there are no possible parents, the section must be added to the list of parents for sequential execution, and potentially getting parallelized with a section that will be assigned later. If there are more than one parent that the thread can be assigned to, then they must be evaluated, and there are two possible approaches.

The first approach is to always place the section to the parent that has the lowest ILC. This means that parents that can be parallelized with few other sectoins will reach the maximum thread size first, and the scenario where a section cannot be parallelized with any parent will be less likely. In the worst case scenario this could lead to an uneven distribution of parallel threads among the parents if the maximum amount of sections in parallel is large. The opposite approach is to prioritize equal distribution of parallel threads between parents. This means that

the algorithm will try to give every parent a parallel thread, and then try to give every parent two parallel sections and so on. If the maximum amount of parallel sections is larger than the amount of cores, then this approach could bring performance advantages, but in a bad scenario it can lead to having to create additional parent threads. The program uses the first approach, and the generated execution table is the following:

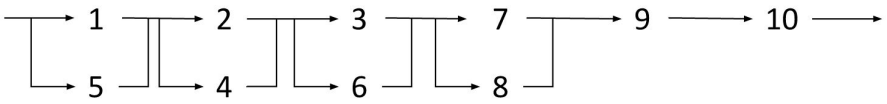


Figure 4.23: Execution Table

4.10 Principles of Code Generation

4.10.1 Overview

Since multithreading techniques are used in order to parallelize the given sequential program, specific rules have to be respected to ensure that the obtained program functions as expected (i.e. its output is the same as the output of the sequential code).

In order for code generation to be automatic, the output code follows a defined format.

4.10.2 Token processing for code generation

When processing tokens generated by LexAnalyser and converting them into new, parallel code, whitespaces are inserted after every token. This is done because tokens do not contain any information about whitespace locations in the sequential code. It does not interrupt program execution, because C grammar is, in general, indifferent to redundant spaces; however, some exceptions exist, such as:

- Any file name included by #include must be located in between "<>" symbols without spaces.
  For example: "#include <time.h>" – correct. "#include < time.h >" – incorrect.
- Every #define that defines a function must not have a space between the function name and the first parenthesis.
  For example: "#define ADD(a, b)" – correct. "#define ADD (a, b)" – incorrect.

With that in mind, the description of the generated code format follows.

4.10.3 Generated code format

Mandatory includes

In order for the parallelized code to function, the following includes are added to the start of the generated file:

– #include <stdio.h>
  Output the program execution time

– #include <pthread.h>
  Pthreads API

– #include <time.h>
  Execution time measurment

– #include <inttypes.h>
  Use of uint64_t data format for storing precise execution time (in nanoseconds)

Precode

Precode is any code that comes in the program before *main()*. In the generated code, it is located in the same place as in the sequential code. It may contain functions, includes, defines, struct/union/enum declarations, typedefs and global variables.

The includes defined by the user are considered a part of this section. Even if some of them are the same as the mandatory includes, the program compiles and executes successfully.

When processing precode tokens and writing them into a generated file, a special attention is paid to #define and #include tokens, for the reasons described in subsection 4.10.2. The details about how it is handled by the program are described in section 5.16 in the Implementation chapter.

***timespecDiff* function definition**

It is a function that is used in the parallel program in order to measure the difference between two time measurements. It is called at the beginning and at the end of the main code of the program in order to measure its execution time.

sdata_struct struct declaration

A struct that stores pointers to the variables used in the program. The pointers are void pointers because it is not yet known what types the variables have.

This struct is needed because one of the parameters that must be passed to *pthread_create* function is a void pointer to an argument that the specific thread needs to work with. In case the thread works with more than one argument, a struct containing pointers to them must be passed instead. By passing a struct containing pointers to all of the program variables, it is ensured that each thread has access to all the variables it may need to manipulate.

Definitions of section functions

The function pointers that will be executed by corresponding threads are defined. Here each section of a sequential program becomes a separate function, ready to be assigned to a pthread. Each of these functions follows a special pattern:

If a section does not contain any operations with variables, the code of the section is generated without any changes within the function.

If a section contains any operations with variables, a pointer to sdata_struct is defined first. After that, pointers to all the variables the section operates on are obtained from the pointer to sdata_struct, casted to appropriate types and defined.

Finally, the code of the section is generated. The code requires some formatting in order to work with the pointers of the variables instead of the variables directly. To achieve that, all the variable pointers need to be dereferenced. The type of formatting is different depending on the variable type:

– If the type of a variable is a basic type, such as int, float, char, and so on, it is dereferenced by being put into parenthesis and having an asterisk placed in front of it.

Example 4.1 (Dereferencing a variable of a basic type)
Sequential code:

```
for(c = 0; c < n; c++) //do something
```

Parallel code:

```
for ( (*c) = 0 ; (*c) < (*n) ; (*c) ++ ) //do something
```

– If the type of a variable is an array, it is dereferenced by being put into parenthesis.

Example 4.2 (Dereferencing an array)
Sequential code:

```
printf("Value: %d", ++test[0]);
```

Parallel code:

```
printf("Value: %d", ++ (test)[0]);
```

– Finally, if the type of a variable is a field of a struct or a union, it is dereferenced by the variable name of the struct being put into parenthesis and having an asterisk placed in front of it, but leaving the field being referenced (i.e. the part after the dot) unchanged.

Example 4.3 (Dereferencing a struct or union)
Sequential code:

```
printf("A struct value is \%d", struct.a);
```

Parallel code:

```
printf("A struct value is \%d", (*struct).a);
```

Main function

The main function of the parallel program starts with declaring variables **start** and **end** of type "struct timespec" (defined in time.h library). They store the recorded clock values at, respectively, start and end of the program.

Next, *clock_gettime* function is called (defined in time.h library), that assigns the current CLOCK_MONOTONIC value (absolute elapsed time since some unspecified point in the past) to the **start** variable, in order to calculate the program runtime later.

Section 0 containing all the variable declarations then gets generated, in the same format as it is defined in the sequential version of the program.

Variables of type "pthread_t" then get declared – one for each thread/section.

Next, the variable **sdata** of type "struct sdata_struct" (declared earlier) gets defined. It is then filled with addresses of all of the variables in the program.

Thread creation

Finally, pthreads for each section are created. For that, *pthread_create* function is used. It is passed the required parameters, i.e. pointer to pthread_t variable storing the thread, NULL for thread attributes to create it with default attributes, pointer to function that the specific thread is going to be executing, and pointer to struct sdata struct, containing pointers to all the program variables.

In order to make sure that the threads are executed in the proper order, the *pthread_join* function is used. The following example shows how the program flow looks like if Thread 1 and Thread 2 should be executed in parallel.

> **Example 4.4 (Thread creation)**
> Creating thread 1
> Creating thread 2
> Joining thread 1
> Joining thread 2

This way, it is ensured that these threads will finish their execution before any other action defined in main occurs.

If there are more threads that are to be executed in parallel, but that are supposed to run after the first executed set of threads, they will get created and joined in the same way, after joining the previous threads.

After all of the threads have been joined, *clock_gettime* gets called again, and the time obtained from it is assigned to **end** variable. At this point, defined *timespecDiff* function is used to calculate time elapsed by the program. This time is then printed.

The main generation is finished with "return 0".

Postcode

Postcode refers to any code coming after *main()*. It starts with "}" and the rest of it may contain function definitions and defines. Just as with precode, a special attention is paid to #define and #include tokens, as described in subsection 4.10.2.

4.10.4 Basic outline of a parallelized program

Following is an outline of the structure of a parallelized program.

```
1  //Mandatory defines
2  #include <stdio.h>
3  #include <pthread.h>
4  ...
5  //Precode
6   struct Alex { int a ; } ;
7   ...
8  //Defining function for execution time measurment
9  int64_t timespecDiff(struct timespec *timeA_p, struct timespec
       *timeB_p){...}
10 //Defining struct for arguments
11 typedef struct sdata_struct{...} sdata_struct;
12 //Section threads definition
13 void *sec_1(void *void_ptr){sdata_struct *data_str_thread = (sdata_struct*)
       void_ptr; ...}
14 ...
15 //Starting main
16 int main()
17 {
18 struct timespec start, end;
19 //Start execution time measurment
20 clock_gettime(...);
21 //Sec 0
22  int next = 30 , first = 0 , ...
23 //Threads for sections
24 pthread_t sec1_thread;
25 pthread_t sec2_thread;
26 ...
27 //Defining sdata_struct struct for all the variables used
28 struct sdata_struct sdata;
29 sdata.arg1 = &next;
30 ...
31 //Creating pthreads
32 if (pthread_create (&sec1_thread, NULL, sec_1, (void *)&sdata)){...}
33 if (pthread_create (&sec2_thread, NULL, sec_2, (void *)&sdata)){...}
34 pthread_join(sec1_thread, NULL);
35 ...
36 //Finishing execution time measurment
37 clock_gettime(...);
38   uint64_t totalTimeElapsed ...
39 ...
40 return 0;
41 //Postcode
42 }
```

Figure 4.24: Parallel program outline

Chapter 5

Implementation

5.1 Class description: LexAnalyser

5.1.1 Overview

The LexAnalyser class is in charge of reading the chars in the code located in a file, and grouping them in strings, called tokens. It also marks some tokens of special interest, functions calls and tokens that could be variables.

Input

– Path to the file with code to be analysed.

Output:

– List of all tokens in the file

5.1.2 Fields

String tripleOps[]

Stores all operators made out of 3 characters.

String constOps[]

Stores all combinations of two characters that mean there is an operator working with a constant (for example: "a =-10", = is an operator, - is part of a constant number).

String doubleOps[]

Stores all operators made out of 2 characters.

String keywords[]

Stores C89 keywords, as well as #define, #include and main.

5.1.3 Methods

_scan **(String fileNameParam): ArrayList<String>**

Gets all the characters from the file located at the path given in the parameters. Iterates through the characters creating tokens according to the state machine described in section 4.3. Marks function calls by adding a ? char at the start of the token, and variables by adding a $ at the start. Returns an ArrayList of these tokens.

isKeyword **(String str): Boolean**

Checks if the given string is in the keywords field. Returns true if there is a match, otherwise false.

isDoubleOperator **(String str): Boolean**

Checks if the given string is in the doubleOps field. Returns true if there is a match, otherwise false.

isConstOperator **(String str): Boolean**

Checks if the given string is in the constOps field. Returns true if there is a match, otherwise false.

isTripleOperator **(String str): Boolean**

Checks if the given string is in the tripleOps field. Returns true if there is a match, otherwise false.

5.2 Class description: CodeSplit

5.2.1 Overview

The CodeSplit class has the task of splitting the tokens into precode, section 0, maincode and postcode Tokens, and calling the necessary processing of precode and section 0.

Input

– ArrayList of all code tokens.

Output:

– List of precode tokens, section 0 tokens, maincode tokens and postcode tokens; List of new keywords and list of constants that were found in the processing; and a hash map storing variables and their data types.

5.2.2 Fields

String keywords[]

Stores C89 keywords, as well as #define, #include and main.

ArrayList<String> PreCode

Stores Pre Code tokens.

ArrayList<String> PostCode

Stores Post Code tokens.

ArrayList<String> MainCode

Stores Main Code tokens.

ArrayList<String> Section0

Stores Section 0 Code tokens.

ArrayList<String> NewKeywords

Stores new keywords found (data types).

ArrayList<String> constants

Stores constants (enumerations, define constants).

HashMap variables

Stores variable identifiers, mapping to their datatype (where arrays have an appended % character).

5.2.3 Methods

***Split* (ArrayList<String> tokens): ArrayList<String>**

Splits the tokens into the different categories, and calls processing of Pre Code and Section 0 to obtain new keywords, constants and the variable hash map. The methods used to detect the start/end of sections are described indepth in section 4.4.1.

***isKeyword* (String str): Boolean**

Removes the $ marker if present. Checks if given string is in the keywords or newKeywords field. Returns true if there is a match, otherwise false.

***getSection0* (): void**

Splits the remaining tokens into Section 0 tokens, and not Section 0 tokens using methods described in section 4.4.1.

5.3 Class description: Harvester

5.3.1 Overview

The Harvester class is in charge of detecting include, define, typedef struct, enum, union and global variable grammar and processing the tokens to extract the new data types, constants and the hash map of variables. It is also in charge of marking array variables, and inserting break lines at the start of a new statement.

Input

– List of tokens to process.

Output:

– List of new keywords and list of constants that were found during the processing; a hash map storing variables and their data types.

5.3.2 Fields

String keywords[]

Stores C89 keywords, as well as #define, #include and main.

ArrayList<String> CodeTokens

Stores Processed Code tokens.

ArrayList<String> NewKeywords

Stores new keywords found (data types).

ArrayList<String> constants

Stores constants (enumerations, define constants).

HashMap variables

Stores variable identifiers, mapping to their datatype (where arrays have an appended % character).

5.3.3 Methods

***getKeywords* (ArrayList<String> preCodeTokens): ArrayList<String>**

Looks for certain tokens, and gathers information from them as established by section 4.4.2. Returns the new tokens, with added break lines.

***fix* (String str): String**

Removes the $ marker if present. Returns the remaining String.

***isKeyword* (String str): Boolean**

Calls fix(str). Checks if resulting string is in the keywords or newKeywords field. Returns true if there is a match, otherwise false.

***uniqueAdd* (String str): void**

Adds str to newKeywords if it is not already there. This is done to avoid adding a keyword multiple times.

5.4 Class description: SectionSeparator

5.4.1 Overview

The SectionSeparator class is in charge of inserting @ tokens at the end of each line or loop and if condition; it also fixes wrongly marked variables.

Input

– List of tokens to process, list of constants, list of newKeywords.

Output:

– List of tokens with added @ tokens.

5.4.2 Fields

ArrayList<String> NewKeywords

Stores new keywords found (data types).

ArrayList<String> constants

Stores constants (enumerations, define constants).

5.4.3 Methods

***separate* (ArrayList<String> tokens, ArrayList<String> extraKeywords, ArrayList<String> constantsP): ArrayList<String>**

Iterates until it finds the end of a statement of a loop, and adds a @ token to mark it. If it encounters a token of a keyword or constant marked as a variable, it removes the $ character.

Returns the modified token list.

***fix* (String str): String**

Removes the $ marker if present. Returns the remaining String.

***isKeyword* (String str): Boolean**

Calls fix(str). Checks if resulting string is in the keywords or newKeywords field. Returns true if there is a match, otherwise false.

5.5 Class description: RWrec.java

5.5.1 Overview

Pattern recognizer class is called like this, because its task is to recognize variable reading and writing patterns in the code. All other manipulations within a code are not important for the program, which is why its functionality is limited and it is not a complete parser, even though it works like one. The most important features of the class will be discussed in this section.

Input

– List of tokens.

Output:

– "data.txt" (derived from "fulldata" strorage) : a text file with entries of format "variable, section number". Example: "$area, 8". Every new entry is placed on a new line. Such format is introduced in order to apply Spark RDD operations on the data.

– "datatable.txt" (derived from"fulldatatable" storage) : a text file with entries of format "number of a section : variable1 variable2 . . . ". Example: "7: $p_r $z_r $m_r $v_w". This file allows accessing sections by their numbers as indices. It is done on Spark stage in order to obtain information about access type of a particular variable. A standalone holder is needed for this purpose due to two reasons:

    1. It is more convenient to keep the data about sections and corresponding variables in a structured way;

    2. It is essential to omit readings and writings in data.txt. Otherwise, the contents will not fit Spark transformations and actions used.

– *HashMap* : a map built for code generation part of the program. Similar to the contents of "fulldatatable" data holder, but without variable access type ("_r" or "_w") mentioned. Some more details are provided in the corresponding method description.

5.5.2 Key features

Pattern recognizer class is called like this, because its task is to recognize variable reading and writing patterns in the code. All other manipulations within the code are not important for the program, which is why its functionality is limited and it is not a complete parser, even though it works like one. The most important features of the class will be discussed in this section of the report.

A variable is read from when its value is accessed but not changed. It can, for example, happen in a loop condition: *if (a < 10)*. The value of **a** is read and compared to 10. Reading from **a** also happens, when its value is assigned to another variable: *int b = a*. The accessed variable can be followed by different tokens, and there are several ways of reading from a variable, as it is possible to observe. Within the project, these are called patterns.

The same goes for writings. A token, which entails writing to a variable, can

be placed before this variable (pre-increment or pre-decrement) or after the variable. There is a set of writing operators, stored in **wrtops** data holder of the class. Pre-increment and pre-decrement operators are special to some extent, because if a "++" or "−−" token is met in the code with no variable prior to it, then it means, that the next coming variable will be written to. So, the state machine chooses the following path: State 1 -> State 5 -> State 6.

All the other patterns are recognized based on a token, which a variable is followed by. The process starts at state 1. Once a variable is found, its name is stored in a **temp1** variable, and a transition to state 4 is performed. If the variable is followed by a round bracket token ")" or a sequence of those, the state does not change, it is looped until a different token is met. If the token is one of the writing operators, then writing pattern was recognized, and the machine takes state 6. If it is a "[" token, then an array is going to be dealt with, and a transition from state 4 to state 2 happens. All the other tokens met at state 4 will imply reading to the variable and will lead to state 3.

Nevertheless, there is an ambiguous case. Consider an expression:

if(x<**y) ++**z;

The issue of these codes is that the state machine (which processes one token per while-loop iteration) does not know, which variable the "++" token refers to. A reader can the whole picture at once, and it is clear to him, that the mentioned token refers to the variable **z**. Here is another piece of code: "(y**)++**;". It is similar to the first one in terms of tokens, but this time the variable **y** is written to. One may notice, that there is a whitespace after ")" in the first example. That is true. However, Scanner class returns a list of tokens with no whitespaces, Pattern Recognizer has no idea about them. This is how the ambiguity occurs.

The second expression does not require round brackets, since they do not change the behavior. However, it is allowed by C grammar. It would in addition require Pattern Recognizer class to analyze the actual meaning of the processed code. This procedure is normally done by a semantic analyzer in a classic compiler, a complete version of which was not needed in the project. Another solution could be previous token memorizing. For the first example, the state machine would memorize would memorize "if" token and would know, that **y** variable belongs to the condition, so the "++" token could not refer to **y** variable. This approach would though occupy some additional space and make the analysis more complex, since a condition can be long and composite. That is why it was decided to use the look ahead technique used in compiler design.

The aforementioned technique works in the following way. Say, a "++" or "−−" is found at index **i** of the token list, when the process is at state 4. The program will then read the token at index **i**+1 to have a wider picture of the processed code. For the example before:

```
if  (  x  <  y   )   ++   z   ;
                  i-1   i   i+1
```

If the token at index **i**+1 is either a variable or is equal to "(" , then the "++" or "−−" token at index **i** refers to an upcoming variable placed somewhere at index **i**+**n**. One remark is that the index is not just **i**+1 because any variable can be surrounded by any amount of round brackets.

Another feature of the state machine is the stack used to handle multidimensional arrays and nested arrays (when the value of an element of one array is an index of another array, like "arr1[arr2[0]]"). The usage of memory classifies the state machine as a push down automaton. Let us take a look at the following example: "arr1 [arr2 [5 + num]] = 8;" . The process starts at state 1 as usually.

Tokens	arr1	[	arr2	[	5	+	num	]	]	=	8	;
States	4	2	4	2	2	2	4	3->4	3->4	6	1	1
Stack	empty	arr1	arr1	arr2 arr1	arr2 arr1	arr2 arr1	arr2 arr1	arr1	empty	empty	empty	empty

Figure 5.1: The change of stack during processing

The table above depicts, which state a token leads to. Current stack trace is also provided. A "3->4" expression means that "]" first leads to state 3 and then to state 4, because it needs to be reprocessed (to see the further behavior), so index **i** is not increased and the state machine does not take the next token yet. So, the principle is quite straight forward. When a variable is followed by a "[" token, it is pushed to stack. A "]" token pops the top of the stack. Thus, it is possible to determine which array variable a reading or writing token refers to by peeking the top element of the stack.

One more characteristic of this state machine is reprocessing of a token, which has already been mentioned. Continuing with the arrays, consider the following statement: "arr3[m+n]−−;" .The variable **n** is placed after "+" token and is followed by the "]" token, which means it is read from. So, the state machine goes to state 3 and marks **n** variable is reading. However, it does not make sense to return to state 1, because the statement is not finished and **arr3** variable is yet to be processed. It cannot jump to the next token "−−" either, because then "−−" token will be lost without an "owner". The solution is to move from state 4 to state 3 without jumping to the next token and then reprocess the current token at state 3. In the

example, "]" token at state 3 will lead to the intermediate state 4. Afterwards, the token "−−" will cause a transition to state 6, where **arr3** variable will be marked as writing.

In general, reprocessing means that index **i** will not be incremented at some transitions, as it is possible to observe in figure 4.1. In this way, the state machine (or an automaton) includes epsilon transitions. An epsilon transition is a change of automaton state without reading the input symbol [29]. The presence of ε - transitions makes the given state machine be an NFA, which stands for non-deterministic finite automaton. Nevertheless, it is still possible to program this kind of state machine, since all transitions from any state are unique. Furthermore, an NFA can be converted to a DFA (deterministic finite automaton), according to automata theory. This is right for the pattern recognizer state machine: it could easily become a DFA with no damage to the behavior. Such conversion would nonetheless merge some states in one. For example, states 1, 3 and 6 could become one "bulky" state carrying out several actions. More transitions would also come out of one state. The state machine would then become needlessly tight and complex. That is why it was decided to make separate states for certain actions (state 3 for reading and state 6 for writing, for instance) and keeping ε - transitions.

5.5.3 Methods

***FSM* (ArrayList<String> scode): void**

The whole state machine described is implemented in *FSM* method. This is not the only method contained in the class. Some other additional methods are created in order to prepare the data for the next stage of processing as well as perform some manipulations on this data. These functions are doing their job within the same task, so they are not shifted to a standalone class.

***isvar* (String token): boolean**

Checks if a token, passed to it, is a variable. A token of type String is a variable, if it starts with a '$' character, since it was decided to mark variables of the input source code with this symbol.

***iswriting* (String token): boolean**

Checks if a token is contained in the writing operator list.

***isinlist* (Tuple tuple, ArrayList<Tuple<String, Integer» varsec): boolean**

Method *isinlist* does not allow adding duplicates to the **fulldata** storage. If section 1 had a variable **n**, a custom tuple <$n, 1> was added to the storage. Now if another variable **n** is detected in the same section, an extra <$n, 1> will not be added to the data holder. It is not important for **fulldata** storage, how many times a particular variable is dealt with inside the same section. It is enough to know, that a variable is present in a section. Combined with access type records from **fulldatatable** storage, this information will be used later to fined dependencies between sections. A custom condition is used instead of *list.contains()* , because a content-wise equality of two custom objects is checked for. Override *Equals()* would be required otherwise.

***isstrongreading* (String var, ArrayList<String> list): boolean**

Method *isstrongreading* is responsible for keeping the strongest access of a variable in the **fulldatatable** data holder. Namely, writing is stronger than reading. This method is called on state 3 before attaching a read access type variable to the **fulldatatable** storage. Let one assume, that reading from some variable **g** was detected. It is now important to check, if any element of **fulldatatable** (which is also an array list) contains either $g_r or $g_w. Note, that these two are about the same variable in spite of being different as strings. So, if an element of **fulldatatable** already includes a reading from **g**, there is no need to add one more reading from the same variable, since reading is not stronger than reading. If the data holder already has a writing to **g**, then the program does not add reading from **g** either, because reading is not stronger than writing. Thus, a variable with reading access type is added to an element of **fulldatatable** if it is the first time the variable is detected in a particular section.

***isstrongwriting()* (String var, ArrayList<String> list): boolean**

Method *isstrongwriting* behaves in a similar manner. It is called on state 6, which is responsible for writing access type variables. Let the third element of **fulldatatable** look like this: [$z_r, $h_r, $b_w]. It is worth reminding, that index of the element corresponds to the number of the section, which these variables are found in. Now, if a writing to variable **h** is detected further is section 3, it should no doubt be added to the data holder, since writing is stronger than reading. However, there it is no longer needed to keep "$h_r" in the holder. That is why "$h_r" is removed from the list and "$h_w" is added to the end of the list (order of the variables in the list does not matter). So, reading from a variable is basically replaced by a stronger operation - reading. The contents of the third element of **fulldatatable** will then be the following: [$z_r, $b_w, $h_w]. Another case is when a writing to a variable was already in the list. If one more writing to the variable is detected in the same

section, it will not be added to the list. At last, it goes without saying, that if it is the first time a variable is faced in a section, it is added to the list regardless to its access type.

***writetofiles* (): void**

Method *writetofiles* does exactly what it is called after. It writes the contents of **fulldata** and **fulldatatable** data holders to two text files, which will later be uploaded to Google Cloud and fed to the dependency finder (Spark part). Some modifications are performed. C language structures are an example, which will be described in a particular method.

***structCheck* (String var, boolean accessEnabled): String**

Method *structCheck* takes care of structures and unions. Consider a structure **struct1** with fields **c** and **d** of type int. Let section 4 write to **struct1.c** variable and section 5 write to **struct1.d** variable. These two are perceived by the program as two different strings and, consequently, two different variables. This is true. However, the situation is different for unions. The grammar of calling a member of a union is the same as for calling a member of a structure. Unlike structures, unions allow storing variables of different data types in the same memory location [19]. So, commands "printf("%d", union1.e);" in section 5 and "union1.f = 18.3;" in section 6 will access the same memory, which is why their sections cannot be executed in parallel. The task of *structCheck* method is to make different members of a structure or a union (including nested ones) to be treated by Spark program as the same variables. This is done by taking into account only the first part (before "." symbol) of a string-represented variable. Getting back to two commands above, <$union1, 5>, <$union1, 6> will be written to "data.txt" file, and "$union1_r" , "$union1_w" will be written to "datatable.txt" file. With this input, the dependency finder program will consider sections 5 and 6 dependent between each other.

***removeReturns* (): void**

Method *removeReturns* is called just before writing to files. The issue is, that main method of a C program has a return-statement, which is considered by lexical analyzer as a separate section. Nevertheless, this section cannot be parallelized with anything else. Just the opposite: it should be sequentially run as the last section to indicate the end of main method. That is why as many elements are removed from **fulldata** holder as many variables there are in return statement section. It is enough to remove only the last element from **fulldatatable** holder, because any element of it represents the whole section at once.

***getMapForCodeGeneration* (): HashMap<Integer, ArrayList<String»**

Method *getMapForCodeGeneration* creates a HashMap of section numbers as keys and ArrayLists of variables in the sections as values. The contents of the generated map are the same as of **fulldatatable** holder. The only difference is that strings of variables in the generated map do not include access type in the end, which means "_r" and "_w" are ignored by taking substrings of **fulldatatable** elements. Variable indicators "$" are also omitted at this point. The map is afterwards passed to code generator, where access type of a variable is not important.

5.6 Class description: FileUploader

5.6.1 Overview

The main purpose of the FileUploader class is uploading data.txt and datatable.txt files to the Google Cloud bucket, so that they can be passed to the program to be executed on the cluster.

Input

– data.txt and datatable.txt files on the local system

Output:

– data.txt and datatable.txt files in the Google Cloud bucket

5.6.2 Fields

Storage storageService

Stores Storage object used to connect to the Cloud Storage.

5.6.3 Methods

***getStorageService* (): Storage**

Obtains Google Credentials for giving the application access to the Cloud and creates a Storage object.

GoogleCredential is obtained from GoogleCredential.*getApplicationDefault()* method. This method retrieves the address of the file with the credentials from the **GOOGLE_APPLICATION_CREDENTIALS** environment variable .

Storage.Builder is then used to build a new Storage object with the obtained credentials. This object is returned.

***uploadSimple* (Storage storage, String bucketName, String objectName, InputStream data, String contentType): StorageObject**

Uploads a file to the bucket with the specified parameters.

New InputStreamContent object is instantiated. This object stores information about the type of file being uploaded, as well as the FileInputStream with the file contents.

A new Storage.Objects.Insert object gets instantiated, using a storageService obtained from *getStorageService* method. It contains the information about the file being uploaded and the bucket the file is being uploaded to.

A default file "gzipping"(compression) by MediaHttpUploader gets disabled, so that the contents of the InputStream are stored on storage "as is". The file then gets uploaded.

***uploadFile* (String bucketName, String storageName, String localFilePath): void**

Uploads a file to the cloud bucket.

Storage object is obtained and passed to the *uploadSimple* method, together with the other parameters.

5.7 Class description: JobSubmitter

5.7.1 Overview

The main purpose of the JobSubmitter class is submitting a Spark Job to the Google Cloud cluster. It contains methods for getting a Google Cloud Dataproc service, submitting the job and checking its status.

Input

- The name of the Google Cloud project
- The scope of the project ("global")
- The name of the cluster to submit the job to
- URI (Uniform Resource Identifier) of the Main jar file on the Cloud
- List of arguments to pass to the main class of the job jar

Output:

– Submitted job status

5.7.2 Fields

String jobID

Stores jobID randomly assigned to the job that gets submitted.

Job ourjob

Stores currently executing Job object.

Dataproc dataprocService

Stores Dataproc object used to connect to the Cloud Dataproc.

JsonFactory jf

Stores JsonFactory object used to build a dataprocService.

5.7.3 Methods

***getDataprocService* (): Dataproc**

Obtains Google Credentials for giving the application access to the Cloud and creates a Dataproc object.

GoogleCredential is obtained from GoogleCredential.*getApplicationDefault()* method. This method retrieves the address of the file with the credentials from the **GOOGLE_APPLICATION_CREDENTIALS** environment variable .

Dataproc.Builder is then used to build a new Dataproc object with the obtained credentials. This object is returned.

***submitJob* (String projectName, String scope, String clusterName, String mainJarUri, List args): void**

Submits a job to the cluster with the given parameters.

A random UUID (universally unique identifier, which is simply a 128-bit value) gets created in order to be able to query the results of job submission.

A new Dataproc object is then obtained from *getDataprocService* method. This dataprocService is then used to submit the job with the specified parameters.

***getJobStatus* (): String**

Gets a status of the submitted job.

Dataproc object is obtained and the instance of the job currently being executed is assigned to **ourjob** object. The current state of the job is then obtained from this object and returned.

5.8 Class description: Dependency Finder

In the beginning of the Spark Application, necessary variables are declared and initialized.

```
private static SparkConf conf = new SparkConf().setAppName("Dependency
    Finder").setMaster("local[*]");
private static JavaSparkContext sc = new JavaSparkContext(conf);
private static Broadcast<HashMap<Integer, ArrayList<String>>>
    datatableBrVar = sc.broadcast(new HashMap<>());
```

The application is written in Java, so Scala entities should be converted to their Java representations. In addition, "local[*]" parameter of *setMaster* method means that Spark will run with as many worker threads as the logical core on the machine.

The main method calls *findIndependent* method, where the main calculation happens. It takes four parameters:

- **datapair** of type JavaPairRDD<String,Integer>: an RDD of format <variable,section> produced in *handleinput1* method.
- **datatable** of type HashMap<Integer, ArrayList<String» : assembled in *handleinput2* method HashMap of variables and their access types in sections. A broadcast variable will be created out of this variable.
- **sectionsList** of type ArrayList<Integer> : produced by makeSections collection of all section listed one after another.
- **pathToSave** of type String : a path of the output file.

The process described in Spark application design starts with grouping values (sections) by their keys (variables).

```
JavaPairRDD<String, Iterable<Integer>> datapairgrouped =
    datapair.groupByKey().filter(new Function<Tuple2<String,
    Iterable<Integer>>, Boolean>() {
           public Boolean call(Tuple2<String, Iterable<Integer>> par)
               throws Exception {
               long size = par._2.spliterator().getExactSizeIfKnown();
               return size > 1;
           }
       });
```

The point of interest is the *filter* transformation call. It means that only these elements of *groupByKey* created RDD will be added to **datapairgrouped** RDD, for which the *filter call* function returns "true". In this case, if size of Iterable<Integer> **par._2** is greater than 1, then the element will be added. It is done like this because if a certain variable is only found in one section of the input source code and nowhere else, only this section is independent on all other sections. It can already be known after *groupByKey* transformation applied, so there is no need to process that single variable further.

Operations like this will only be described once.

```
JavaRDD<Tuple2<String, Iterable<Integer>>> datatuplegrouped =
    JavaRDD.fromRDD(JavaPairRDD.toRDD(datapairgrouped),
    datapairgrouped.classTag());
```

This is done in order to convert a JavaPairRDD<K,V> to JavaRDD<Tuple2<K,V» in order to apply a necessary operation.

Once the conversion is done, the elements of **datatuplegrouped** dataset are collected to a list using *collect* action.

```
List<Tuple2<String, Iterable<Integer>>> dtGroupedList =
    datatuplegrouped.collect();
```

This action picks up elements of an RDD from multiple workers, forms a list of those and returns it to the driver program. A remark can be done: combined with *filter*, collecting allows building custom lists from RDDs. However, *collect* is a costly operation and can be altered with adding RDD's elements to an Accumulable of list type. The theory behind it is covered in theoretical part on Spark. One of the older versions of developed Spark application used accumulators instead of collecting. However, it was not very advantageous due to the relatively small (in big data standards) size of input source code. Contrariwise, it made the code much longer and more complex to understand, so it was decided to work with *collect* for

the moment.

Getting back to the code, the following piece of code converts **datatuplegrouped** JavaRDD<Tuple2<String, Iterable<Integer»> to **forCartesian** of type ArrayList<Tuple2<String, JavaRDD<Integer»>. Iterables should become RDDs, so it is possible to call an RDD transformation *cartesian* on them.

```
transformation cartesian on them.
ArrayList<Tuple2<String, JavaRDD<Integer>>> forCartesian = new
    ArrayList<>();

       for (Tuple2<String, Iterable<Integer>> elem : dtGroupedList) {
           List<Integer> iterableAsList = new ArrayList<>();
           for (Integer item : elem._2) {
               iterableAsList.add(item);
           }
           forCartesian.add(new Tuple2<>(elem._1,
               sc.parallelize(iterableAsList)));
       }
```

Then JavaRDD<Integer> components of **forCartesian** elements are combined with themselves using *cartesian* transformation.

```
JavaPairRDD<Integer, Integer> tempCommonsPair =
    forCartesian.get(i)._2.cartesian(forCartesian.get(i)._2).filter(new
    Function<Tuple2<Integer, Integer>, Boolean>() {
               public Boolean call(Tuple2<Integer, Integer> par) throws
                   Exception {
                   return par._1 < par._2;
               }
           });
```

Filtering also takes place here because it is necessary to take unique combinations with no repetitions. The order of section does not matter at this point, so an element <section1, section2> is the same as <section2, section1>. That is why *filter call* function returns "true" if and only if number of the first section (par._1) is less than number of the second section (par._2). Let one take a look at the concrete example. Let forCartesian.get(i)._2 have the following value: (1 4 5 8). Combining this dataset with itself and then applying the filtering will make **tempCommonsPair** be equal to (<1,4> <1,5> <1,8> <4,5> <4,8> <5,8>). As it is possible to observe, first components of elements are smaller than second components, and the amount of combinations respects the formula (n=4 for this case):

$$C_2(n) = \frac{n!}{(n-2)!2!} \tag{5.1}$$

Getting back to the code, the unique pairs are added to **commonslist** of type ArrayList<Tuple3<String, Integer, Integer», where the first component of a Tuple3 element is the variable common for two section of a unique pair. A **commonsrdd** of type JavaRDD<Tuple3<String, Integer, Integer» is then obtained by parallelizing (*sc.parallelize*) **commonslist** variable.

The program is now ready to distinguish pairs of sections dependent on each other. Any two dependent sections have a common variable which makes them dependent, but not any two sections sharing a variable are dependent on each other. The latter happens when both sections read from the same variable. The application can now check the access types or variables common for two sections. Access types of variables are stored in **datatable** variable, which has to be read from the nodes. That is why a broadcast variable is made out of it.

```
datatableBrVar.unpersist();
datatableBrVar = sc.broadcast(datatable);
```

Broadcast variable **datatableBrVar** is first unpersisted in order to remove its old value - an empty HahsMap. It was assigned on purpose during creation of the broadcast variable, because it had to be global and it could not be declared without initialization. Now, a new value (**datatable** of type HashMap<Integer, ArrayList<String») is assigned to **datatableBrVar**.

Transformation *filter* is called on **commonsrdd** of type JavaRDD<Tuple3<String, Integer, Integer» followed by *mapToPair* transformation in order to obtain **dependentrdd** of type JavaPairRDD<Integer, Integer>. Filtering takes care of access types. If at least one section (either the second component of a **commonsrdd** element or the third one) writes to the variable (the first component of a **commondsrdd** element), then "true" is returned by the *call* function. "False" is returned otherwise. Accesses are taken from **datatable** by calling *value* method on the broadcast variable. In the **datatable**, keys are represented by section numbers, which makes it easy to manage.

For example, command "datatableBrVar.value().get(par._2()) ..." will get a list of variables and their access types contained in the section with number par._2 contains. So, with a **commondsrdd** element <var, 4, 9>, the command above will return the list of variables in section 4. It is then possible to find “var” in this list and check its access type (“r” or “w”). Let one save it in the variable **access1**. The same is done for section 9, so the access type of variable “var” in section 9 is stored in the variable **access2**. The return value can now be determined.

```
if (access1.equals("w") || access2.equals("w")) return true;
```

```
return false;
```

Transformation *mapToPair* is applied to the result of *filter* transformation for the purpose of converting JavaRDD<Tuple3<String, Integer, Integer» type variable to JavaPairRDD<Integer, Integer> type variable **dependentrdd**. Variables get removed from RDD elements because their accesses were checked, and dependencies were distinguished, so it is no longer needed to process the variables.

After this, an RDD is created from the list of sections **sectionsList**. This RDD is combined with itself using *cartesian* and *filter* exactly in the same way as described before. The result of these two operations is JavaPairRDD<Integer,Integer> type **secpairs**, which contains unique pairs of all sections.

Finally, an RDD of independent section pairs is obtained by subtracting the RDD of dependent section pairs from the RDD of all section pairs.

```
JavaPairRDD<Integer, Integer> independentrdd =
    secpairs.subtract(dependentrdd);
```

At last **independentrdd** is saved as a text file (the path was provided as a parameter), and SparkContext is closed.

```
independentrdd.saveAsTextFile(pathToSave);
sc.close();
```

5.9 Class description: FileDownloader

5.9.1 Overview

The main purpose of FileDownloader class is downloading the output files produced by the Spark job. It contains methods for downloading separate files, as well as a method for obtaining the names of all the output files.

Input

– The output files on Google Cloud Storage

Output:

– The output files downloaded to the local system

5.9.2 Fields

Storage storageService

Stores Storage object used to connect to the Cloud Storage.

5.9.3 Methods

***getStorageService* (): Storage**

Obtains Google Credentials for giving the application access to the Cloud and creates a Storage object.

GoogleCredential is obtained from GoogleCredential.*getApplicationDefault()* method. This method retrieves the address of the file with the credentials from the **GOOGLE_APPLICATION_CREDENTIALS** environment variable .

Storage.Builder is then used to build a new Storage object with the obtained credentials. This object is returned.

***getFileNames* (): ArrayList<String>**

Retrieves names of all the files from the desired Google Cloud Storage directory.

A new Storage object is instantiated in order to connect to the Storage. Storage.Objects.List object is also instantiated and set to list names of all of the files in the project bucket that begin with the prefix "output.txt/p" - since all the output files are generated in output.txt directory and start with a "p" in the following fashion: "part00000", "part00001", and so on.

A **foreach** loop within a **do-while** loop is used to iterate through every obtained file name and add it to the list, until there are no more result pages left. When all the file names have been fetched, the list containing them is returned.

***downloadToOutputStream* (Storage storage, String bucketName, String objectName, OutputStream data): void**

Downloads an object to the output stream.

A Storage.Objects.Get object is instantiated and set to download a desired file from a desired bucket. A direct download gets enabled: it means that the file is going to be downloaded in one HTTP request, which is acceptable since the size of the files that are going to be downloaded using this method is small. The file is then downloaded into the output stream.

***downloadFile* (String bucketName, String storageName): ByteArrayOutputStream**

Downloads a desired file from the cloud bucket to a ByteArrayOutputStream.

A Storage object is obtained and passed to *downloadToOutputStream* method, along with the bucket name, file name and a ByteArrayOutputStream object, which is then returned, containing the downloaded file.

The reason why a ByteArray stream is used to store the downloaded file contents is that the files that Spark outputs are of binary format. That is also why a FileOutputStream is used when writing the downloaded files into text files on the local system (in the programCaller class).

5.10 Class description: FileMerger

5.10.1 Overview

FileMerger class is responsible for merging the multiple output files obtained as a result of the Spark job. It also edits the first output file by adding a number of sections in the input code to its start (it is required for further processing by AccessibilityChecker).

Input

– Output files with independent sections produced by Spark job

Output:

– merged_file.txt file containing information about number of sections in the program and all the independent sections

5.10.2 Methods

***addSecNum* (String secNum): void**

Adds a number of sections in the input program to the start of the first output file (so that the merged file will contain it as well).

The contents of the original file are obtained using BufferedReader. The FileWriter is then used to create a new file with the same name, append the number of sections to it and fill it with the contents of the original file line by line. This way, the new file contains both the number of sections and the original contents, while the old file is removed.

***mergeFiles* (ArrayList<String> fileNames): void**

Merges the downloaded output files.

In order to merge the files, java.nio API is used. Using a passed ArrayList containing the names of the output files to merge, the file Paths get stored in a new List. This list is then iterated through, and contents of each file are appended to a new file, called "merged_file".

5.11 Class description: AccessibilityChecker

5.11.1 Overview

AccessibilityChecker is the class that transforms the list of data independent sections into a table that can be used to generate every possible parallelization. Then it calls an object of the ThreadAssignemnt class to generate an execution schedule which is returned.

Input

– File with number of sections and independent section pairs.

Output:

– Execution table.

5.11.2 Methods

***getExecutionTable* (): ArrayList<ArrayList<Integer> >**

Gets number of sections and independent section pairs form the file and uses the forward and backward table operation as described in section 4.8. The union is passed to the ThreadAssignment object, and the result is returned. It also uses classes Pair and Sort2D.

5.12 Class description: Pair

5.12.1 Overview

The Pair class is for creating pairs of sections that are independent. Stores two values of type int.

Input

– Initialization values for x and y.

Output:

– None

5.12.2 Methods

***Pair* (int px, int py): void**

Creates this object with values given.

***getX* (): int**

Returns value of X.

***getY* (): int**

Returns value of Y.

***toString* (): String**

Returns a printable version of the Pair using the format " (x , y) ". Used for debugging.

5.13 Class description: Sort2D

5.13.1 Overview

Holds methods to sort a list of Pairs. Uses insertion sort.

Input

– List of pairs.

Output:

– Ordered list of pairs.

5.13.2 Methods

***_Sort2D* (ArrayList<Pair> arr): ArrayList<Pair>**

Orders the list, using X as priority, then Y. Returns the ordered list.

***_Sort2DInv* (ArrayList<Pair> arr): ArrayList<Pair>**

Orders the list, using Y as priority, then X. Returns the ordered list.

5.14 Class description: ThreadAssignment

5.14.1 Overview

Gets the amount of cores availabel to the VM, and sets that number as the maximum parallel sections. The it assigns parent threads, and distributes the remaining sections using the External and Internal Linking Coefficients as described in section 4.9.

Input

– Posible parallelizations of each section.

Output:

– Execution sequence.

5.14.2 Methods

***ThreadDistribute* (ArrayList<ArrayList<Integer» AccesibilityChecked): ArrayList<ArrayList<Integer> >**

Creates a schedule of execution, both sequentially and in parallel. Uses methods from the MergeSort class.

5.15 Class description: MergeSort

5.15.1 Overview

Standard merge sort algorithm and some methods required for thread assignment

Input

– List of integers.

Output:

– List of ordered integers.

5.15.2 Methods

***MergeSort* (ArrayList<Integer> unsorted): ArrayList<Integer>**

Splits the list in two, and calls recursively sorting on the new lists, then merges the result.

***merge* (ArrayList<Integer> left, ArrayList<Integer> right: ArrayList<Integer>**

Merges two lists in order.

***FormatFix* (ArrayList<Integer> unformated: ArrayList<ArrayList<Integer»**

Fixes the format of a list from 1 dimensional to two dimensions.

***addParent* (ArrayList<ArrayList<Integer» threadExecutionTable, int currentIndex: ArrayList<ArrayList<Integer»**

Method for correctly adding a new parent to execution schedule.

5.16 Class description: Generator

5.16.1 Overview

The purpose of the Generator class is generation of the parallel code after all other necessary operations with the code have been performed. It contains methods for retrieving information required for code generation from tokens produced by other classes. Some of those methods format the tokens so that they can be written into a file in a desired manner. The rest of the methods are responsible for generating different parts of the code.

This section focuses on describing the technical details of code generation; the reasoning behind why the code is generated exactly in this format is given in the Design chapter of the report, section 4.10.

Input

- Tokens of main function of the program from LexAnalyser class
- Tokens of other parts of the program from CodeSplit class
- HashMap with variable types from Harvester class
- HashMap with variables of each section from RWrec class

– Table with information about thread execution order (Execution Table) from ThreadAssignment class.

Output:

– "generatedCode.c" file

5.16.2 Fields

ArrayList<String> Sections

Stores sections formatted appropriately for code generation.

ArrayList<String> Variables

Stores variables detected in the input code.

ArrayList<ArrayList<Integer» ExecutionTable

Contains information about thread execution order.

5.16.3 Methods

***getVars* (ArrayList<String> tokens): ArrayList<String>**

Retrieves and appropriately formats variables of the program from tokens obtained from the *main()* function of the program.

Foreach loop is used to iterate through tokens. First, the check if the token is a variable is performed by seeing if the first element of the token is a dollar sign. Next check deals with the detected struct and union fields – if the detected variable contains a dot, only the part before dot is stored in a variable list. ArrayList containing detected variables is returned by this method.

***getSections* (ArrayList<String> tokens, HashMap<String, String> varTypes): ArrayList<String>**

Formats tokens of the program obtained from the *main()* function of the program to Strings in a form needed for generated code.

StringBuilder object is instantiated for the purpose of building a formatted String from the input tokens.

Foreach loop is used to iterate through tokens. First, the check if the token is

not a section separator (“@”) is performed. After that, formatting of variable tokens takes place.

To appropriately format structs and unions, the tokens containing a dot get split in the parts before and after the dot. They then get appended to the stringBuilder in the following format: “(*variable).field ”.

Tokens that do not contain a dot are checked for the case of being arrays. This is done utilizing **varTypes** HashMap.

```
else if (varTypes.containsKey(string.replace("$", ""))
   && varTypes.get(string.replace("$", "")).contains("%")) {
       stringBuilder.append(string.replace("$", "(")).append(") ");
}
```

In the code snippet above it is checked if the current token is an array, by checking whether its datatype contains an array marker "%". If this is the case, then the current token gets appended to the stringBuilder in the following format: “(variable) ”.

If the variable is not a struct, union or array, it gets appended in the following format: “(*variable) ”.

Next, function tokens are taken care of.

```
else if (string.charAt(0) == '?' && (string.length() > 1))
   stringBuilder.append(string.replace("?", "")).append(' ');
```

The code above checks that the current token starts with a question mark and that its length is larger than one (to make sure it is a function, not a ternary operator). If this is true, the question mark gets removed and the token gets appended to the stringBuilder.

The final else block is executed when a section separator (“@”) token is met. In that case, the accumulated stringBuilder gets converted to a String and added to the List of sections. Lastly, it gets cleared in order to get ready to be used for the formatting of the next section.

getCode (ArrayList<String> codeTokens): String

Retrieves and appropriately formats code from the preCode, sec0 and postCode tokens.

"count" variable is introduced in order to prevent appending spaces after tokens

for the cases of "#include" and "#define" tokens.

Foreach loop is used to iterate through tokens. In case the token is "#include" it gets appended together with a space, and the "count" variable is assigned a value of 3 – so for the next three tokens spaces will not get appended, and the resultant part of the String will be of the following format: "#include <file.h>".

Next check deals with “#define” tokens.

```
else if ((string.equals("#define")) && codeTokens.get(index + 1).charAt(0)
    == '?') {
    stringBuilder.append("\n").append(string).append(' ');
    count = 1;
}
```

The code above checks if the current token is a define token and the next token is a function symbol ("?") – if this is the case, the current token gets appended with a newline symbol preceding it. A space is then appended and the “count” variable is set to 1: space will not get appended after the next token, and the resultant part of the String will be of the following format: “\n#define funcName( )”.

The last three checks in this method check if the token is a variable, function or something else. In case it is a variable or a function, a corresponding marker "$" (for variables) or "?" (for functions) gets removed, and the token gets appended with or without a space, depending on the value of the “count” variable. In case it is neither a variable nor a function, the token simply gets appended with or without a space, depending on the value of the “count” variable.

At the end of the method, the accumulated stringBuilder gets converted to String and returned.

generateStruct (int varNum): String

Generates a struct called "sdata_struct" for storing pointers to all the variables in the program. varNum parameter is a number of variables in the program.

First, a String that starts the struct is defined. Then, using a stringBuilder, arguments for the struct get appended, with the names "arg1", "arg2" and so forth, as many as is needed. Finally, the generated String gets appended with a stringBuilder.toString() and a String finishing the struct, and the result gets returned.

***generateThreadFunc* (int secNum): String**

Generates a thread function pointer for a section that contains no variables. secNum is an ordinal number of the thread function to generate.

First, a String that starts the function is defined. Name of the function includes the provided section number.

Then, the corresponding formatted code that is to be executed by that function is obtained from the Sections ArrayList. Finally, a String that finishes the function is defined, and the concatenation of all the three created Strings gets returned.

***generateThreadFunc* (int secNum, HashMap<String, String> varTypes, String... varNames): String**

Generates a thread function pointer for a section that contains variables. secNum is an ordinal number of the thread function to generate, varTypes is a HashMap linking each variable to its type, and varNames is an array of variable names that are contained in the section.

First, a String that starts the function is defined. Name of the function includes the provided section number.

Next, a StringBuilder object is instantiated, and a Foreach loop is used to generate instantiations of all the pointers to the variables used in the section.

```
for (String varName : varNames) {
    String varStrTemplate = " #TYPE# *#VAR# =
        (#TYPE#*)(data_str_thread->arg"
        + (Variables.indexOf(varName.split("\\.")[0]) + 1) + ");\n";
    varStr = varStrTemplate.replace("#TYPE#",
        varTypes.get(varName.split("\\.")[0]).replace("%", ""))
        .replace("#VAR#", varName.split("\\.")[0]);
    varStrBuilder.append(varStr);
}
```

The code above first defines a template for the String that is used to generate a pointer to every variable used in the section function being generated. The String is always of the following format: "varType *varName = (varType *) (data_str_thread->argNo);", where argNo is linked to the index of the variable pointer to which is being generated (the index is taken from obtained in *generateCode* method List **Variables**, described further).

The following checks are conducted: if the variable name contains a dot, only

the part before the dot gets taken into account when getting varType or varName, since the struct or union fields do not require generating pointers to them. Taking this into consideration, the String with the template is replaced by the String with the actual values of the corresponding varName and varType (percentage signs are removed from variable types since they are only used as array markers for main code formatting). It is then appended to the StringBuilder.

Then, the corresponding formatted code that is to be executed by that function is obtained from the Sections ArrayList. Finally, a String that finishes the function is defined, and the concatenation of all the created Strings gets returned.

***generateMainStart* (): String**

Generates start of main function, including declaration of variables (Section 0) and Pthreads, as well as defining a struct for storing pointers to the declared variables.

Section 0 tokens are obtained and passed through *getCode* method in order to be converted to code. Then, a **for** loop is used to generate as many Pthread declarations as the number of sections. Finally, a struct storing pointers to the declared variables is generated, using a **for** loop iterating through Variables list. All the generated Strings are then concatenated and returned.

***generateThreads*(int row): String**

Generates Pthreads and joins them for every "row" of ExecutionTable. Row in this context is a set of sections that need to be executed simultaneously, all stored in the same List within the ExecutionTable.

First **for** loop iterates through the specified row of the Table and generates "Pthread_create" functions for every section in that row.

Once the first **for** loop is finished, the second **for** loop iterates through the specified row of the Table again, this time to generate needed "Pthread_join"s for the threads created before.

All the generated Strings then get concatenated and returned.

***generateCode*(ArrayList<String> mainTokens, HashMap<String, String> varTypes, HashMap<Integer, ArrayList<String» secVars, ArrayList<ArrayList<Integer» executionTable): void**

Generates a file with the parallel code. This is the method that gets called from the programCaller.

postCode and preCode tokens are obtained and passed through *getCode* method in order to get converted to code. Then, *getVars* and *getSections* methods are called with mainTokens in order to retrieve the variables and formatted sections of the program. In order to get rid of the duplicates of the variables, they are converted to HashSet and then back to ArrayList, which is assigned to "Variables" field.

BufferedWriter is declared and instantiated in order to write resultant generated code into a file. The code is then generated using methods previously described in this section.

In order to distinguish between using method for generating threads with or without variables, the double **for** loop is used. Every section gets checked to find if it has variables, in which case all of them get added to a list which is then converted to an array to be passed to *generateThreadFunc* for sections with variables method. Otherwise, a *generateThreadFunc* for no variable scenario is called.

Chapter 6

Testing

6.1 Sequential versus parallel code performance

The first performed series of tests is aimed at checking the performance difference between the sequential source code (input into the system) and the parallel code (system's output).

A code used for testing purposes has a purpose of finding values required for solving a range of Statistics problems, which means it may potentially involve analyzing large amounts of data. The sequential code follows:

```
//program for testing: performs Sxx, Syy and SSr for a dataset
#include <stdio.h>

float getAverage(float var[], int size){
   float mean = 0;
   int index = 0;
   while(index<size){
      mean += var[index];
      index++;
   }
   return mean;
}

float sumMult(float var1[], float var2[], int size){
   float sumSq = 0;
   int index = 0;
   while(index<size){
      sumSq += var1[index]*var2[index];
      index++;
   }
   return sumSq;
```

```
}

int main(int argc, char * * argv){

   float x[] = {2860, 2010, 2791, 2618, 2212, 2184, 3244, 2692, 2206, 2914,
       3034, 4240, 1400, 2257,2860, 2010, 2791, 2618, 2212, 2184, 3244,
       2692, 2206, 2914, 3034, 4240, 1400, 2257};
   float y[] = {2.66, 2.46, 2.95, 2.81, 2.90, 2.88, 2.13, 3.03, 3.54, 3.1,
       4.32, 2.85, 2.2, 2.69, 2.66, 2.46, 2.95, 2.81, 2.90, 2.88, 2.13,
       3.03, 3.54, 3.1, 4.32, 2.85, 2.2, 2.69};
   int n = 28; //size of data (since sizeof will work but not keyword)
   float meanx, meany, ssqx, ssqy, sumxy, Sxx, Syy, Sxy, SSr;

   x[0]=1000; //test of array recognition
   //get the means
   meanx = getAverage(x, n);
   meanx /= n;
   meany = getAverage(y, n);
   meany /= n;

   //get the sum of the squeares
   ssqx = sumMult(x, x, n);
   ssqy = sumMult(y, y, n);
   sumxy = sumMult(x, y, n);

   Sxx = ssqx - n*meanx*meanx;
   Syy = ssqy - n*meany*meany;
   Sxy = sumxy - n*meanx*meany;
   SSr = (Sxx*Syy-Sxy*Sxy)/Sxx;
   printf("SSR = %f", SSr);
  return 0;
}
```

Arrays **x[]** and **y[]** contain numbers that were modified multiple times during testing, and their sizes are referred to as "input size" (stored in **n** variable).

Since it is clear that performance of the parallel program is influenced by an overhead caused by introducing multiple threads, it was decided to include a third program version into the test: a sequential program with added overhead. The way this was done was making each section of the program run by a separate thread: i.e. in a way similar to the parallel version of the program, but without allowing multiple sections to execute at once.

Thus, the three tested program versions are:

- Sequential

– Parallel

– Sequential with overhead

In order to make tests as fair as possible, each program was automatically run 1000 times per trial, and the mean execution time was calculated based on the results of these 1000 runs. Each program was tested for 12 different input sizes, in order to analyze differences in scalability. For every trial new input values were randomly generated. Table 6.1 contains the results of the test.

Table 6.1: Parallel vs. Sequential vs. Sequential with overhead: Program average execution time comparison (per 1000 runs)

Trial number	*N*	*Sequential*	*Parallel*	*Sequential with overhead*
1	28	15455	898744	1154123
2	2028	44594	898261	1138483
3	4028	74646	908698	1246757
4	6028	107315	1098230	1171256
5	8028	138424	1002351	1368530
6	10028	167835	1018939	1225656
7	12028	203965	1000753	1437071
8	14028	281613	1070369	1257458
9	16028	264297	1243006	1387558
10	30028	488795	1144445	1599969
11	90028	1434258	1562560	2531780
12	180028	2962063	2285471	4062718

Execution times are in nanoseconds. N is the input size.

It can be observed that for small input sizes, sequential version of the program is significantly faster than the parallel, while the sequential program with overhead is slower than the other two.

However, as the input size is increased, performance of the sequential program decreases significantly. The parallel code, on the contrary, scales much better, becoming faster than the sequential at large number of inputs.

The same data is plotted on the scatter graph in figure 6.1. It can be observed that for both versions of the sequential program (regular and with overhead) a steeper growth of execution time is observed compared to the parallel version of the program, which demonstrates its better scalability, becoming more efficient at higher input sizes.

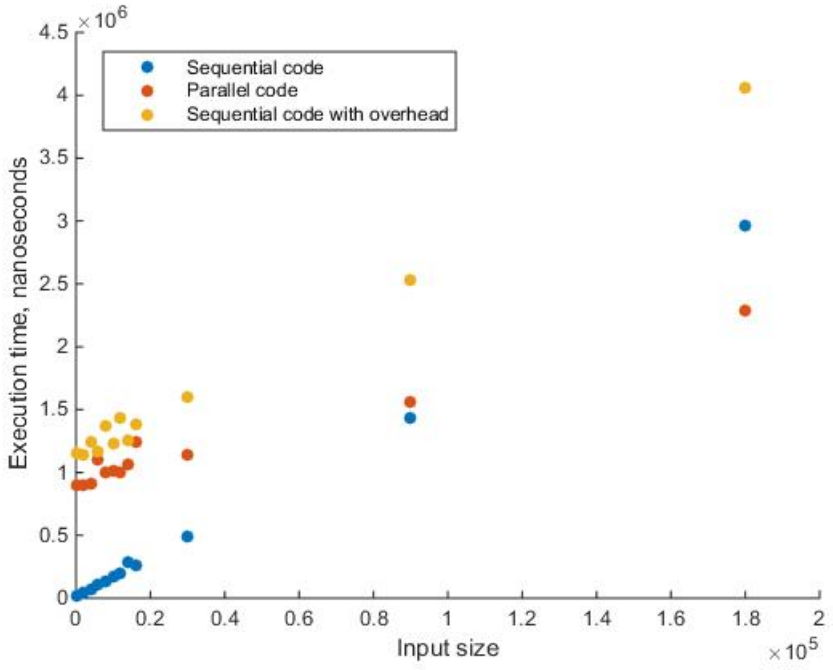


Figure 6.1: Input size vs. execution time

It is thus possible to conclude that the observed behaviour follows Amdahl's Effect, which states that a speedup for a fixed number of processors increases with the problem size, since less time is spent on communication among the processors and more - on parallel processing. [24]

6.2 Automatic parallel code sectioning versus manual parallel code sectioning

This series of tests was aimed at testing whether the parallel code produced by the system is more efficient compared to the parallel code section distribution of which has been manually changed (two pairs of sections were merged).

The test conditions remained the same as in the previous section: 1000 runs per trial, recording the means of these runs. The programs were tested for the same input sizes as in the previous series of tests. Table 6.2 contains the results of the test.

An original assumption made before conducting this test series was that manual

Table 6.2: Parallel program with normal vs. manual sectioning: average execution time comparison (per 1000 runs)

Trial number	N	$Parallel\ (normal\ sectioning)$	$Parallel\ (manual\ sectioning)$
1	28	898744	764920
2	2028	898261	861721
3	4028	908698	837603
4	6028	1098230	761260
5	8028	1002351	879292
6	10028	1018939	930890
7	12028	1000753	977332
8	14028	1070369	975675
9	16028	1243006	895186
10	30028	1144445	1101441
11	90028	1562560	1667703
12	180028	2285471	2609394

Execution times are in nanoseconds. N is an input size.

sectioning of the program is more efficient than the automatic, since the sectioning algorithm is not taking into account all the factors that may affect code parallelization. However, the data obtained do not show a clear trend. When the input size is small, the manually sectioned program seems to behave better; however, for large input size the automatically sectioned program takes a lead in the performance.

A natural assumption may be that the manually sectioned program became more sequential in its nature: that would explain better performance for smaller input sizes and slower runtimes for larger inputs compared to the normally sectioned code.

It is concluded that it is not clear how good the developed thread assignment algorithm is compared to the manual thread distribution, and more testing is required to make reliable conclusions regarding this question.

6.3 Apache Spark: Cloud versus Local Mode performance

The final series of performed tests was used to check whether the utilization of Google Cloud for finding independent sections in the input source code is more efficient than using a local system.

The tests were performed the following way: an input program was fed into the

system 5 times and the average execution time was recorded for both local machine and the Cloud. The input program size (amount of sections) was then increased. This was done in order to see the scalability difference between Spark in local mode and Spark in the Cloud.

Table 6.3: Apache Spark: Local mode vs. Cloud: average execution time comparison (per 5 runs)

Trial number	*Input size (sections)*	*Local*	*Cloud*
1	14	3.6838	20.3468
2	40	3.600	17.5746
3	118	3.906	16.6848
4	196	4.2488	22.2184
5	586	7.3982	27.8854

Execution times are in seconds.

Table 6.3 contains the results of this test series. It can be observed that the execution of the program on the Cloud takes much longer time than it does on the local machine. Possible explanation for that is that Cloud cluster is executing the program on multiple computers, thus increasing a communication overhead.

When increasing the size of sections in the input program, the execution times for both local machine and Cloud do not show noticeable changes for small alterations in the input size; however, the runtimes increase significantly when the input size is increased first to 196, and then to 586 sections.

Although more tests are definitely required to find the input size for which using Cloud over Local system is reasonable, it is naturally assumed that scalability of Cloud is better than that of the local mode. From the obtained data, it can be seen that with the increase of input size from 14 to 586, Local mode execution time has doubled, while the Cloud runtime has only faced a 1.37 times increase.

Thus, it is concluded that it is not feasible to use Cloud for the developed system, as programs with source code sizes large enough to consider its usage are unlikely to be parallelized with the help of this framework. Although the task of finding dependencies in the input code seemed like it could be considered a Big Data problem at the beginning of the development, it was later understood that this is not the case. However, Cloud seems to be an appropriate solution for applications dealing with truly large amounts of data.

Chapter 7

Conclusion

As stated in the Introduction chapter, the main problems this project was aiming to solve are detecting independent parts of the given sequential program and converting this information into a parallel code.

The first issue was successfully tackled by our partial implementation of a compiler's front-end (Lexical and Syntax analysis) in order to tokenize the program and find the relevant information (variables being read and written to). Furthermore, cluster computing was involved to analyze the combinations of program sections and detect the independent pairs.

The second problem was solved by utilizing the obtained independent sections information to create an execution table with the help of a thread scheduling algorithm.

The requirements that were set by the group before the start of the project were also met. The C code analysis has been implemented for a reduced set of C grammar. Cloud Computing has also been involved, just as it was desired.

The system was indeed designed in Java programming language; however, it is recognized that implementing a more defined structure before the start of the development could lead to a more cohesive program design, meaning that some of the existing classes could be reimplemented in a better way.

The goal of producing a parallel version of a sequential program was achieved; however, the parallel code was not always performing better than the sequential one. The reasons for that were considered a large overhead cost combined with a little benefit from introduction of multiple threads. Nevertheless, it was observed that for large input sizes the parallel program may run faster, as is explained by

Amdahl's effect.

Extensive testing of the system has been performed, analyzing sequential and parallel program performances, as well as testing the benefit of using Cloud.

Finally, the knowledge obtained from Compilers and Databases course, as well as Distributed Computing course was successfully utilized in the project.

Further steps

- In the current version of the system, the whole process is only run once. A certain parallel program is produced as output. It can then be executed with measuring the elapsed time. However, there is no guarantee that this parallel program is the most optimal compared to other parallel program possibilities. That is why it makes sense to loop the process, which means start all over again having different section organizations and also skipping some stages whose output will not change. Execution time of the first parallel program version can be used in the second and further iterations of the process as a factor in thread assignment part.
- It was difficult to plan the structure of the system in advance. The structure was formed over the course of program development. Once the system was developed, it became possible to pinpoint the ways it could be improved and refactored with no behaviour violated. For the given version, it is clear for the developers that several parts respect similar patterns. This is a reason for making these parts unified (of the same format) by utilizing objective-oriented paradigm of Java language in a more efficient way.
- Multiple rules of C language grammar were excluded due to the complexity and irrelevance (within the current project) of analyzing those. It is a future work to introduce the forbidden rules back to the system and analyze these, covering wider and wider aspects of C language.

Bibliography

[1] H. Agrawal. “On slicing programs with jump statements”. In: *SIGPLAN Notices* 29.6 (1994), pp. 302–312.

[2] Ravi Sethi Jeffrey D. Ullman" "Alfred V. Aho Monica S. Lam. *”Compilers: Principles, Techniques, and Tools”*. 2. ed. Pearson Education, 2007.

[3] Vangie Beal. *batch processing*. 2016. URL: http://www.webopedia.com/TERM/B/batch_processing.html.

[4] Z. Chen and B. Xu. “Slicing concurrent Java programs”. In: *ACM SIGPLAN Notices* 36 (2001), pp. 41–47.

[5] R. Mall D. Goswami and P. Chatterjee. “Static slicing in Unix process enviroment”. In: *Software Practice and Experience* 30.1 (2000), pp. 17–36.

[6] Michael Ernst. “Practical fine-grained static slicing of optimized code”. In: *Technical Report MSR-TR-94-14, Microsoft Research* (1994).

[7] Apache Software Foundation. *Apache Spark programming guide*. 2016. URL: http://spark.apache.org/docs/latest/programming-guide.html#overview.

[8] Apache Software Foundation. *Spark short*. 2016. URL: http://spark.apache.org/.

[9] Apache Software Foundation. *Spark wins*. 2016. URL: http://spark.apache.org/news/spark-wins-daytona-gray-sort-100tb-benchmark.html.

[10] Apache Spark Foundation. *Class RDD*. 2016. URL: https://spark.apache.org/docs/1.6.1/api/java/org/apache/spark/rdd/RDD.html.

[11] Apache Spark Foundation. *Cluster Mode Overview*. 2016. URL: http://spark.apache.org/docs/latest/cluster-overview.html.

[12] D. Goswami and R. Mall. “A parallel algorithm for static slicing of concurrent programs”. In: *Concurrency Practice and Experience* 16.8 (2004), pp. 751–769.

[13] J. Warren J. Ferrante K. Ottenstein. “The program dependence graph and its use in optimization”. In: *ACM Transactions on Programming Languages and Systems* 9.3 (1987), pp. 314–349.

[14] X. Zhou J. Jiang and D. Robson. “Program Slicing for C – The problems in implementation”. In: *IEEE Computer Society Press* 12.1 (1991), pp. 182–190.

[15] J. Cheng J. Zhao and K. Ushijima. “A dependence-based representation for concurrent object-oriented software maintenance”. In: *Proceedings of the Second Euromicro Conference on Software Maintenance and Re-engineering: IEEE Computer Society Press* (1998), pp. 60–66.

[16] B. Korel and J. Rilling. “Dynamic program slicing methods”. In: *Information and Software Technology* 40 (1998), pp. 647–649.

[17] William Landi and Barbara G. Ryder. “Pointer-induced aliasing: A problem classification”. In: *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages* (1991), pp. 93–103.

[18] Jacek Laskowski. *SparkContext - Entry Point to Spark*. 2016. URL: `https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-sparkcontext.html`.

[19] Tutorials Point (I) Pvt. Ltd. *C - Unions*. 2016. URL: `http://www.tutorialspoint.com/cprogramming/c_unions.htm`.

[20] J. Lyle. “Evaluating variation on program slicing for debugging”. In: *PhD thesis, University of Maryland, College Park.* ().

[21] J. R. Lyle and D. Binkley. “Program slicing in the presence of pointers”. In: *Proceedings of the Third Annual Software Engineering Research Forum. IEEE Computer Society Press* (1993).

[22] S. Daninic M. Harman and Y. Sivagurunathan. “A parallel algorithm for static program slicing”. In: *Information Processing Letters* 56.6 (1996), pp. 307–313.

[23] Katherine Noyes. *Five things you need to know about Hadoop v. Apache Spark*. 2016. URL: `http://www.infoworld.com/article/3014440/big-data/five-things-you-need-to-know-about-hadoop-v-apache-spark.html`.

[24] Dr. Daniel Ortiz-Arroyo. *Operating Systems and Data Communication lecture material*. Aalborg University Esbjerg, Fall 2015.

[25] "Michael J. Quinn". *"Parallel programming in C with MPI and OpenMP"*. 1. ed. McGraw-Hill, 2004.

[26] T. Reps S. Horwitz and D. Blinkey. “Interprocedural slicing using dependence graphs”. In: *ACM Transactions on Programming Languages and Systems* 12.1 (1990), pp. 26–61.

[27] "International Organization for Standardization (ISO)". *ISO/IEC 9899:1990*. 1990.

[28] Frank Tip. “A Survey of Program Slicing Techniques”. In: *Journal of Programming Languages* 3.3 (1995), pp. 121–189.

[29] Portland State University. *NFA with epsilon transitions*. 2016. URL: http://web.cecs.pdx.edu/~sheard/course/CS311/Fall2013/ppt/NfaEpsilonDefined.pdf.

[30] M. Weiser. "Program Slicing". In: *IEEE Transactions on Software Engineering* 10.4 (1984), pp. 352–357.

[31] M. Weiser. "Reconstructing sequential behavior from parallel behavior projections". In: *Information Processing Letters* 17.10 (1983), pp. 129–135.

[32] Reynold Xin. *Apache Spark officially sets a new record in large-scale sorting*. 2016. URL: https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html.

[33] Priti Shankar" "Y. N Srikant. *"The compiler design Handbook: optimizations and machine code generation"*. 2. ed. Taylor Francis Group, 2008.

Appendix A

Connecting application to Google Cloud

This appendix aims at providing a short guide for the interested about how to connect one's application to Google Cloud servers.

1. The first step is to register and create a new project. The simple way to do it is through a web browser: `http://console.cloud.google.com`. It is possible to sign up for a free account, which allows to use all Google Cloud platform services for 60 days with a virtual credit of 300 dollars and some limitations, such as the maximum number of cores in Compute Engine (8).

2. After creating the project, it is necessary to activate certain Google APIs depending on which services are desired. To access the API Manager, press on the "Enable and manage APIs" menu in the console.

   Now, the APIs that need to be accessed by the application should be activated. For the case of the APIs used by this specific project, Google Cloud Dataproc API is needed: to activate it, type "dataproc" in the search field, select the appearing API and click "Enable". This will enable not only Dataproc API, but also APIs required for it to function, such as Compute Engine and Storage.

3. The next step involves creating credentials in order to be able to use the activated API. To do that, go to "Credentials" menu on the left of the API Manager page, and click on "Create credentials". In the drop-down menu, choose "Service account key" - this type of key is used to authenticate applications rather than people, and that is exactly what is needed in our case.

   In the "Service account" drop-down menu, select "Compute Engine default

service account". Leave the key type as JSON and click "Create". The key is then going to be downloaded. It is important to note that the created key must be stored securely, as it cannot be downloaded again.

4. Now that the API is enabled and the key is stored on the local computer, it is required to create an environment variable called GOOGLE_APPLICATION_CREDENTIALS which will be storing the path to the key.

   One way to create it on a Linux systems is to open the terminal and type in the following command: "gedit .bashrc". Then add the following line to the end of the file:
   export GOOGLE_APPLICATION_CREDENTIALS='/path/to/the/key'
   Save the file and restart the terminal session.

5. At this point, it becomes possible to authenticate applications using the "default credentials". In order to do that, GoogleCredential class is used. An example of creating a Google Storage object and authenticating it using the default credentials follows.

```
1   /* Code from FileUploader class*/
2
3   public static Storage getStorageService() throws IOException,
        GeneralSecurityException {
4       if (null == storageService) {
5       //Getting default credentials
6       GoogleCredential credential =
            GoogleCredential.getApplicationDefault();
7       //In case the obtained credentials need specifying the required
            scopes, insert the scopes requested
8       if (credential.createScopedRequired()) {
9           credential = credential.createScoped(StorageScopes.all());
10      }
11      HttpTransport httpTransport =
            GoogleNetHttpTransport.newTrustedTransport();
12      JsonFactory jf = JacksonFactory.getDefaultInstance();
13      storageService = new Storage.Builder(httpTransport, jf,
            credential).
14          setApplicationName("DepFinder").build();
15      }
16      return storageService;
17  }
```

In the code snippet above a Storage object is obtained in *getStorageService* method. This object is authenticated using Google default credentials and

can be used to perform operations with Google Storage.

Lines 8-10 make sure that if the credentials require to have their scope specified, the scopes requested are inserted. This means that if the credentials are used to, for example, modify Storage content, they may require having a corresponding permission scope specified. Since we want to allow all operations with Google Storage for our application, the *StorageScopes.all()* method is used to pass all possible Storage scopes to the credentials.

In order to access Cloud servers from the local machine, Google Cloud API libraries must be installed.

Google libraries used in this specific project are (with corresponding links):

- Google API Client Library
  `https://developers.google.com/api-client-library/java/google-api-java-client/download`
- Google Cloud Dataproc API
  `https://developers.google.com/api-client-library/java/apis/dataproc/v1`
- Google Cloud Storage API
  `https://developers.google.com/api-client-library/java/apis/storage/v1`