ABSTRACT

Tyro: A First Step Towards Automatically Generating Parallel Programs From Sequential Programs

Arun Sanjel, M.S.

Mentor: Gregory D. Speegle, Ph.D.

Currently, MapReduce is used as the standard for automatic parallelization of programs. However, MapReduce restricts programs to a simple framework with limited parallelism but still requires the user to understand parallelism within the framework. In this thesis, we present Tyro, a new tool that automatically translates a sequential Python program into a parallel PySpark program. Tyro identifies potential code fragments where parallelism can be done and translates them. It uses Abstract Syntax Trees (AST) for fragment detection and gradual program synthesis to convert the Python operations into PySpark operations. Tyro also verifies the generated code against given user test cases. We evaluated Tyro by automatically converting different real world sequential Python programs into PySpark programs. The resulting PySpark programs perform up to 9x faster (on 9 parallel machines) compared to the original. The promising result of Tyro against these benchmarks shows how Tyro can utilize gradual synthesis and operation translation to go beyond MapReduce with automatic parallelization.

Tyro: A First Step Towards Automatically Generating Parallel Programs From Sequential Programs

by

Arun Sanjel, B.S.

A Thesis

Approved by the Department of Computer Science

Erich Baker, Ph.D., Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree
of
Master of Science

Approved by the Thesis Committee

Gregory D. Speegle, Ph.D., Chairperson

Pablo Rivas, Ph.D.

Enrique Blair, Ph.D.

Accepted by the Graduate School
December 2020

J. Larry Lyon, Ph.D., Dean

Page bearing signatures is kept on file in the Graduate School.

TABLE OF CONTENTS

LIST OF FIGURES vii

LIST OF TABLES ix

ACKNOWLEDGMENTS x

DEDICATION xi

1 Introduction 1

2 Background 4

2.1 Program Synthesis 4

2.2 Similar Tools 5

2.2.1 Casper 5

2.2.2 Parsl 7

2.2.3 Emma 9

2.3 AST 9

3 Methodology 11

3.1 Overview 11

3.2 Program Analyzer 13

3.2.1 AST Parser 14

3.2.2 Static Code Analysis 15

3.2.3 Meta-information 16

3.3 Feature Extractor 18

3.3.1 Loop Extractor 19

3.3.2 Operation Extractor 21
3.3.3 Operation Converter 23
3.4 Component Transformer 25
3.4.1 Map Transformer 27
3.4.2 Reduce Transformer 28
3.4.3 Filter Transformer 30
3.4.4 UDFs Transformer 33
3.4.5 Join Transformer 34
3.5 Code Generator 36
3.5.1 Static Code Generation 36
3.5.2 Dynamic Code Generation 37
3.6 Verifier 38

4 Results and Analysis 42
4.1 Evaluation 42
4.1.1 Test Suites 42
4.2 Aggregate Functions 43
4.3 Multidimensional Dataset 46
4.4 User Defined Function 47
4.5 Speed Up 49

5 Future Work and Conclusion 52
5.1 Future Work 52
5.1.1 Increased Loop Detection 52
5.1.2 Complex Data Types and File Handling 52
5.1.3 Optimized Nested Loop Handling 53

5.1.4 Extending The Search Space 53
5.1.5 Verification 53
5.1.6 Beyond MapReduce 54
5.1.7 Partitionable Functions 54
5.2 Conclusion 55

APPENDIX 57

A Additional Results 58
A.1 Count 58
A.2 Min 59
A.3 Join with no operation 60
A.4 Join with Operations 61

BIBLIOGRAPHY 62

LIST OF FIGURES

2.1 Casper Translation 6

3.1 Tyro's System Architecture 11

3.2 Tyro Translation 14

3.3 AST conversion of a Python function 15

3.4 Meta-Information Structure 17

3.5 Meta-Information Example 18

3.6 Loop Extractor Structure 20

3.7 AST structure of a filter operation 22

3.8 Temporary structure of Filter List 23

3.9 Complete Filter List after Operation Extraction inside a filter operation 23

3.10 Operation List 24

3.11 Filter structure after operation conversion 24

3.12 Different operation conversion in Tyro 25

3.13 Map in PySpark 28

3.14 Reduce in PySpark 29

3.15 Filter in PySpark 31

3.16 Filter Conversion Example 32

3.17 Filter Transformation in Tyro 32

3.18 Example of user defined function (UDF) in PySpark 33

3.19 Transformation in Tyro 34

3.20 Simple Join Operation Conversion by Tyro 35

3.21 Example of Tyro' Static Code Generation 37

3.22 Example of Dynamic Code Generation in Tyro 38
3.23 Input Test Cases 39
3.24 Failed Verification 40
3.25 Successful verification 40

4.1 Input Python program to find maximum 43
4.2 Generated PySpark code with Map Operation by Tyro 44
4.3 Generated PySpark code for Maximum Number by Tyro 44
4.4 Python program to find the average of number from a list 45
4.5 Generated PySpark Average 46
4.6 kNN Algorithm in Python 47
4.7 Generated PySpark KNN Code 48
4.8 Python program to find average using UDF 48
4.9 Final Generated PySpark Code in Reduce Stage 49

LIST OF TABLES

3.1 Map Conversion 28

3.2 Reduce Conversion 30

3.3 Pattern matching for aggregate functions 30

3.4 Concatenated filter operations after Filter Transformation in the gradual synthesis of Tyro 33

3.5 Global synthesis counter and its stages 41

4.1 Summary of Tyro's Translation 50

4.2 Speed up comparison of generated programs 51

ACKNOWLEDGMENTS

Through my Master program at Baylor, many people played important roles. I would like to express my sincere gratitude and appreciation to them. My advisor Dr. Gregory D. Speegle, is the most important person who helped me shape this research idea. He is very helpful, patient and tolerant of many mistakes I have made. He sets a high standard for all of the work with great enthusiasm for research. I greatly appreciate everything he has done for me. I would also like to thank other committee members, Dr. Pablo Rivas and Dr. Enrique Blair for their perspective on my work which helped me improve the work presented in this thesis. I am also extremely thankful to the Department of Computer Science at Baylor University for providing me with the opportunity to conduct this research. At last, I want to thank my family and friends who have provided invaluable support not just for research but everyday life. Without them the path to this thesis would have been quite difficult. Being far from home was tough but with constant love and motivation from my parents, my sister and friends in Nepal have made the journey less painful. I thank you all for the great encouragement and support.

To my parents and my late grandmother

CHAPTER ONE

Introduction

Parallelism has been a long pursued goal in the world of programming (Boyer and Moore 1984)(Gilles 1974). This programming paradigm has dynamically grown in the field of computer science with the rapid development of modern computer hardware (Ksiazek, Marszalek, Capizzi, Napoli, Połapl, and Woźniak 2018). Parallel computation is considered as high-end computation and involves breaking up of problems into smaller parts and solving them concurrently (Culler, Singh, and Gupta 1999). In the present context, the need to analyze huge amounts of data has led to modern architectures that support parallelism along with high level programming abstraction to take advantage of the underlying architecture (Nayak, Wang, Ioannidis, Weinsberg, Taft, and Shi 2015). So, parallel programming can result in huge performance gains compared to sequential programming (Ahmad and Cheung 2018; Fedyukovich, Ahmad, and Bodik 2017).

Popular parallel frameworks like MapReduce (Dean and Ghemawat 2004), Apache Flink (Katsifodimos and Schelter 2016), Spark (Zaharia et al. 2010), Hadoop (Shvachko, Kuang, Radia, and Chansler 2010), and Hive (Thusoo et al. 2009) are used in developing data-intensive applications and have varied and highly efficient implementations. However, generally, parallel programming is quite complex and difficult to learn (McKenney 2017). The complex problem of developing correct and efficient parallel programs can only be managed if the user has a deep and proper understanding of the paradigm. Modern parallel frameworks have made parallel programming a bit less painful to use(da Silva Morais 2015; Dobre and Xhafa 2014; Akil, Zhou, and Röhm 2017); however, to leverage these frameworks, the programmer still has to be very familiar with their APIs and the parallelism is limited by the framework..

An alternative approach is to generate a parallel program from a sequential one. Although such conversion of sequential programs into parallel paradigms is not easy or even always possible, as a developer must first understand the existing code and then rewrite the same functionality using various APIs calls. This requires familiarity with the frameworks and the API calls, which is made more challenging by the limited documentations of the constantly evolving frameworks (Nasehi, Sillito, Maurer, and Burns 2012). The whole process of rewriting the code into a parallel frameworks requires a lot of time and expertise.

Using computers to provide rare expertise is highly valuable. A compiler-based tool can convert sequential programs into parallel programs. The compiler must know where the parallellism is possible and convert the existing code into new parallel code that can be executed in the target framework.

Tyro is a new tool for translating a sequential Python program into a semantically equivalent PySpark (Nandi 2015) program. Tyro uses techniques from classical compiler like pattern matching rules (Pierre-Etienne, Ringeissen, and Vittek 2003) where compilers apply rules that match different input code patterns. With pattern matching, Tyro can translate code fragment, into the target framework. As far as we are aware, there is no such tool that converts a sequential Python program to a PySpark program. However, there exists compilers for translating Java programs into the MapReduce paradigm (Ahmad and Cheung 2018).

Tyro identifies parallelizable code segments and then converts each code pattern into equivalent PySpark operations. For this translation to work, we start by exploring the existing program. Using the Python AST package, we search for key code fragments that can be translated to the parallel framework. Tyro utilize the Gradual Synthesis for Static Parallelization (GRASSP) (Fedyukovich, Ahmad, and Bodik 2017) approach to convert the selected fragment from the existing code. The key idea behind GRASSP is to gradually grow the translation process i.e move from

simple conversions to complex conversions on subsequent iteration. It is a staging solution where each stage adds a layer of complexity to the solution.

Once the code generation is complete, we verify that the existing and generated codes are semantically equivalent. In general, to determine if two functions are operationally equivalent is undecidable (Boyer and Moore 1984). However, Tyro simplifies the verification process by using the testing feature of software engineering (Dasso 2006; Jiang and Su 2009). Tyro tests generated parallel program (fragments) using multiple test cases provided by the user and subsequently compares the results. The verification process is successful if the parallel program passes all of the test cases. If any of the test cases fails, the whole verification fails, and Tyro moves to the next stage of the gradual synthesis.

Tyro is implemented in Python and performs experiments on different sequential programs. For each sequential program, operation translation and gradual synthesis is evaluated by translating various sequential fragments of the program into the parallel framework. Eventually, the approach of operation translation can be used in creating compilers that can go beyond MapReduce for parallelization.

The remainder of this thesis is structured as follows: Chapter 2 gives some necessary background and reviews the related work. Chapter 3 presents the overall system architecture with explanation of each component. Chapter 4 confirms the results of Tyro and discussion about them. Chapter 5 summarizes and discusses ideas for future work.

CHAPTER TWO

Background

2.1 Program Synthesis

Program synthesis is the process of producing an executable program from a given specification (Manna and Waldinger 1980). The algorithmic synthesis produces the program automatically, without intervention from an expert. A program itself can be a program specification, so synthesis is typically concerned with the transformation of the program. Tyro uses algorithmic program synthesis. In the paper (Bodik and Jobstmann 2013), the authors talk about various type of program synthesis; however they focus on a reactive synthesis which aims to automatically construct a reactive system from the formal specification. Along with synthesis, the authors also talk about formal verification. After proper program synthesis, a formal verification (Keller 1976) is required to check the correctness of the program. Developers usually rely on simulation to check if the constructed system meets their intent. In Tyro, the formal verification is done using the given test cases along with the input Python code. Program synthesis can also be a good method to transform a program using methods like enumeration, deductive searching, and constraint solving (Gulwani, Polozov, Singh, et al. 2017) into different forms like converting application logic (Java) into SQL queries (Cheung, Solar-Lezama, and Madden 2012). Likewise program synthesis is a good tool for debugging and can be used as a basis for manual transformation.

GRASSP (Fedyukovich, Ahmad, and Bodik 2017) is a program synthesizer in which a program is partitioned into a sequence of segments, and each segment is processed separately. The partial output for the segments are merged into the final program. It has been implemented in an SMT-based programming language Rossette

(Torlak and Bodik 2013) and evaluates looping programs in C++. A GRASSP solution was able improve the performance by factor of five relative to the serial code on an 8 thread machine. The concept of gradual synthesis is used in this project. In general, gradual synthesis is a process of finding the optimal solution for a function f from a list of solutions, starting from simple and gradually increasing in complexity.

Tyro also uses a similar concept of gradual synthesis, where it starts from the simplest parallel operations (Map like expressions) and gradually increases to other operations until it finds the solution. Currently, Tyro works on six different search spaces but the future work looks to extend these spaces (Section 5.1.4).

2.2 Similar Tools

2.2.1 Casper

Casper (Ahmad and Cheung 2018) is a new tool that automatically translates sequential Java code into the MapReduce paradigm. The authors of the paper discuss how Casper identifies potential code fragments and translates them. It has two major steps for this whole process: *Program Synthesis* where it searches for a program summary of each code fragment and *Code Generation* which generates executable code from the program summary. Comparing Casper's generated code with the original code, the generated code performed up to 30x faster on an AWS cluster of 10 instances.

Program synthesis is a key component of Casper, and it searches for MapReduce programs which it can rewrite to match a given input sequential code. Casper uses program synthesis to reduce the search space. During this process, Ahmand and Cheung designed their own high-level intermediate language (IR) that lets them describe their program summaries. The IR is designed in such a way that it is easy to transform the IR into various programming language. Figures 2.1 shows snippet of IR language with input sequential code.

```
@Summary(
  m = map(reduce(map(mat, λm1), λr), λm2)
  λm1 : (i, j, v) → {(i, v)}
  λr : (v1, v2) → v1 + v2
  λm2 : (k, v) → {(k, v/cols)}
)
int[] rwm(int[][] mat, int rows, int cols) {
    int[] m = new int[rows];
    for (int i = 0; i < rows; i++) {
        int sum = 0;
        for (int j = 0; j < cols; j++) {
            sum += mat[i][j];
        }
        m[i] = sum / cols;
    }
    return m;
}
```

(a) Input: Sequential Java Program

```
RDD rwm(RDD mat, int rows, int cols) {
  Spark.broadcast(cols);
  RDD m = mat.mapToPair(e -> Tuple(e.i, e.v));
  m = m.reduceByKey((v1, v2) -> (v1 + v2));
  m = m.mapValues(v -> (v / cols));
  return m;
}
```

(b) Output: Apache Spark Program

Figure 2.1. Translation of a sequential Java codes into Apache Spark codes (Ahmad and Cheung 2018)

Casper has three major components in its system architecture: a *Program Analyzer*, a *Summary Generator*, a *Code Generator*. Program Analyzer consists of a Java parser, a code fragment identifier,a static code analyzer, and verification conditions. Next, the summary generator uses all the information provided by the program analyzer to synthesize and verify the program summary. To speed the search, it uses various techniques such as defining search spaces and incremental grammars with its search algorithms. It enumerates program summaries within the search space and

verifies it against the verification condition. A verified summary denotes that a effective translation is found. The code generator translates the selected summary into executable spark codes. Figure 2.1a shows an input sequential Java program with its generated summary. The summary generated here is the intermediate language (IR) designed by the authors.

Tyro shares major aspects with Casper. The initial static code analysis of Casper is similar to code analysis of Tyro. Both tool use ASTs for code analysis. The IR of a parsed code is good for code generations. However, Tyro doesn't use IR or any intermediate languages. Rather than focusing on IR languages, Tyro focuses on operation translation by using meta-information (Section 3.2). The meta-information is a data structure which acts as a central repository to each component of Tyro. It holds all the information of the input program which includes storing all of the functions in the program for each operation where parallelism is possible. The Intermediate Representation of Casper is used for generating summaries where as the meta-information is used by each component from analysis to code generation in Tyro. The operation translation will eventually lead us to our goal of going beyond MapReduce. Casper uses double verification with Sketch (Solar-Lezama 2008) and Dafny (Leino 2010), while Tyro uses unit testing where the test cases are provided by the user.

The whole process of translating Java code into MapReduce code is quite slow in Casper as its needs to search for correct summaries. With our method of operation translation, the whole process of generating summaries can be skipped making the conversion of parallel code much faster, although currently more limited.

2.2.2 Parsl

Another tool in the realm of parallel programming in Python is Parsl (Babuji et al. 2019). It is a parallel scripting library that supports the development and execution of asynchronous and implicitly parallel data-oriented workflows. The authors

have developed a new architecture where selected Python functions and external applications are connected by shared input/output data objects. Parsl is designed to work on both traditional and new analysis models. The Parsl architecture consists of Parsl scripts, a data flow kernel, and executors. Parsl scripts are decomposed into a simple dependency graph by the data flow kernel (DFK). The DFK also manages the Parsl Apps on a variety of sites. Parsl scripts are comprised of standard python code with an added number of "apps". The authors explained these "apps" as annotated Python functions. Executors are provided by the developer as specifications.

Parsl supports various executioner providers like AWS, Azure, and so on. With the help of the dependency graph, Parsl executes its workflow. The tasks are enqueued in these graphs and a queue is maintained for execution. With the selected executors, Parsl executes these tasks. The authors state that it is an easy to use model and can easily be integrated into existing environments. It includes various features like automated elasticity, multi-site execution, fault tolerance, automated direct and wide data management.

Parsl is developed for workflow management rather than converting sequential code. Considering the major trend of using high level languages for programming and the growing need of parallel computing, Parsl combined these two trends to create a workflow management tool i.e, it allows parallelism to be expressed using decorators in existing Python code. Workflow management and easy integration are good features of Parsl. However, Tyro has a completely different view than Parsl. Parsl provides various infrastructure and architecture in executing a parallel programs with extra information. This again reverts back to the problem of learning new frameworks and APIs. The user has to know about decorators, executors and other features to leverage Parsl. Tyro seeks to avoid this learning curve. Tyro focuses on converting sequential programs into PySpark programs without any additional information apart from tests which should exists for the sequential program anyway. The test cases

are also written in Python so the user doesn't have to know any other languages. Eventually, frameworks like Parsl might serve as the target for Tyro

2.2.3 Emma

In search of going beyond MapReduce, we found Emma (Alexandrov, Katsifodimos, Krastev, and Markl 2016) - a language deeply embedded in Scala, that provides implicit parallelism through declarative data flows. The authors talk about MapReduce as a good fit to work on generalized processing and aggregation of a single collection of a complex object. However, when expressing more complex programs, MapReduce reveals many limitations that hinder the programmer's productivity. A well know example is Join in a MapReduce framework like Hadoop. Emma provides parallelism transparency, advance optimizations with declarative data flows and a transparent execution engine. Emma uses an intermediate language representations to represent various operations. Instead of using an AST, Emma uses monad comprehension which is a layered intermediate representation where dataflow expressions found in the original AST are converted into a declarative, calculus like representation.

However, in the end, Emma is another novel language designed to have better performance utilizing the existing data-parallel execution engines, while Tyro converts a sequential program to a parallel program whenever possible. Likewise Tyro simply uses the AST to convert such operations instead of using complex monad comprehensions.

2.3 AST

The syntax tree of any program is a data structure which shows how different segments of the program are to be viewed as a grammar. The process of converting programs into a syntax tree is called parsing. The exact form of these trees are not convenient, so a modified form of syntax tree is used called Abstract Syntax Trees

(Grune, Van Reeuwijk, Bal, Jacobs, and Langendoen 2012). The Abstract Syntax Trees are used as intermediary representation in many of the compilers (Grosch and Emmelmann 1990; Sarcar and Cheon 2010; Nystrom, Clarkson, and Myers 2003). A compiler generally parses the different source files of a program and then generates an AST. Once the AST is produced from the source code, various static analysis can be performed. Traditionally, most types of static analysis depends on visitor pattern style traversals (Palsberg and Jay 1998) where separate methods are created to handle each type of node in the AST (Aho, Lamb, Sethi, and Ullman 2007). Tyro extensively uses the AST for translation. Specifically, Tyro uses the Python AST standard library (Van Rossum and Drake 2009) to parse and manipulate the syntax tree and its nodes.

CHAPTER THREE

Methodology

3.1 Overview

The whole process of generating parallel programs from sequential programs involves multiple stages. Figure 3.1 illustrates the overall design of Tyro. Tyro has five major components: a *Program Analyzer*, a *Feature Extractor*, a *Component Transformer*, a *Code Generator*, and a *Verifier*.

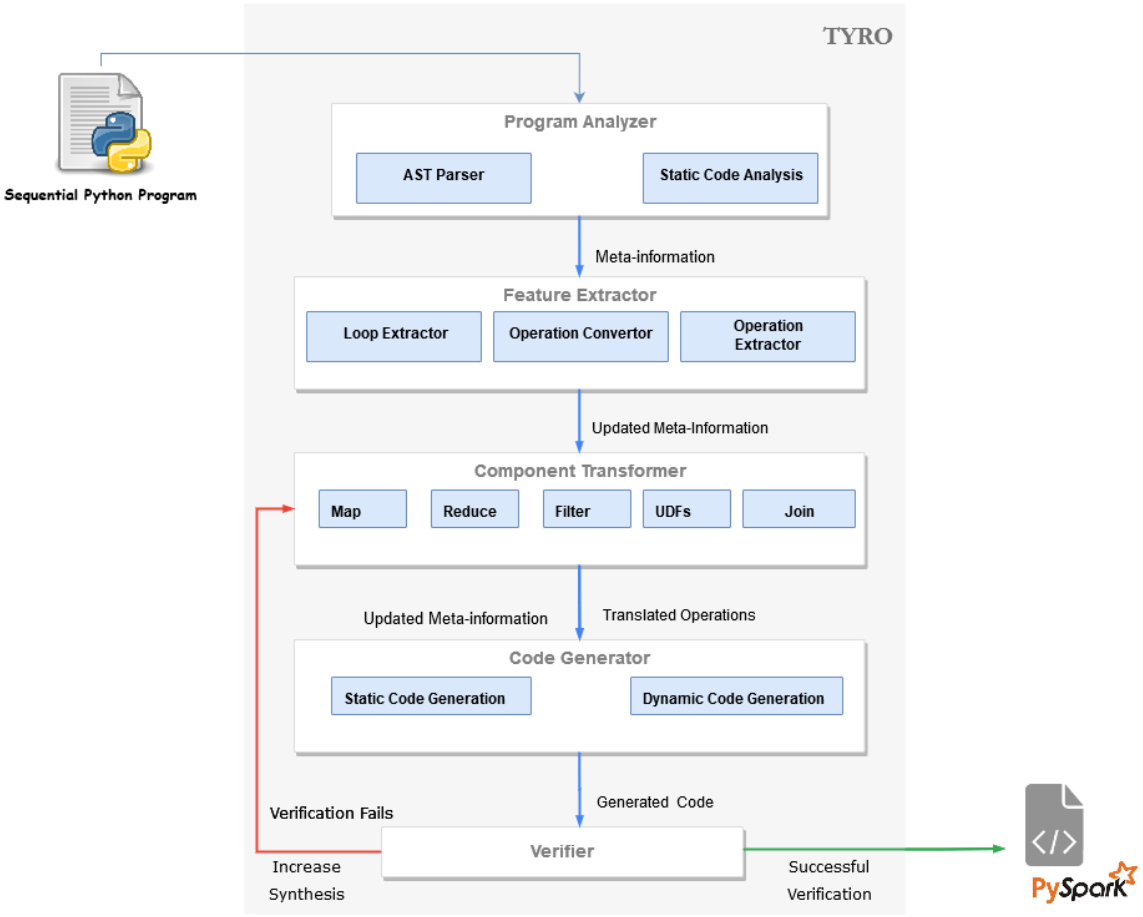


Figure 3.1. Tyro's System Architecture

As with other systems (Ahmad and Cheung 2018; Dean and Ghemawat 2004), Tyro works on a sequential program that iterates over large data sets. Clearly, such programs are the prime candidates for obvious parallelism, such as the Map function in MapReduce and the creation of RDDs in PySpark. Tyro takes a Python program with loops that sequentially iterate over data and translates those loops into a semantically equivalent PySpark program.

Figure 3.2a is an example of a sequential program that iterates over a list of numbers and generates the sum of the numbers. Tyro accepts this code as input and parses the source code into an Abstract Syntax Tree (AST) via the *Program Analyzer* (Section 3.2). With help of the AST module provided by Python, the Program Analyzer extracts all of the information on the program. The output of the Program Analyzer is a JSON like structure called *Meta-information*. The meta-information is key for all of the stages of Tyro, as it stores all of the nodes along with additional information of the node. It acts as a central repository in which each component access the required data.

The meta-information is passed to the *Feature Extractor*, where all the functions, loops, and operation inside the loop nodes are extracted. The goal of Tyro at this stage is to identify possible parallelism. Once the code fragments are detected, they are passed to the *Component Transformer* as updated meta-information. The updated meta-information already has all the extracted AST nodes and their information. The *Component Transformer* module uses meta-information to translate Python operations and loops into PySpark operations. These translated operations are stored in a list along with their input data sets. Gradual program synthesis that generally starts from Map operation occurs in this stage. The synthesis stage is represented by a global variable.

According to the synthesis stage, the *Component Transformer* translates the identified Python operation to its respective PySpark Operation. After the translation of all of the identified operations, the *Component Transformer* sends the meta-information and the translated operations list to the *Code Generator*. The *Code Generator* glues everything together by replacing the identified loops with the translated operations. It also adds basic PySpark commands to initialize the execution environment. Once the code generation is complete, the generated code must be verified by the *Verifier*.

The *Verifier* module verifies the generated code with the test cases provided as input as shown on *Line 9-26* in Figure 3.2a. In this case, the verification fails in the Map stage of the program synthesis. If the verification fails, the global variable is increased and Tyro starts again from the *Component Transformer*. As the synthesis stage increases, Tyro repeats the same process as explained above. In this stage, Tyro uses pattern matching to use the builtin features of PySpark.

For example, on *Line 7* in Figure 3.2b, Tyro matches the reduce operation to a builtin function called *sum()*. After generated the new code, the *Verifier* verifies the generated program. If the verification process is successful, Tyro outputs the generated code. For the input program in Figure 3.2a, the output is shown in figure 3.2b.

3.2 Program Analyzer

In order to begin the translation process, some information about the input program is needed. So for this process, Tyro uses the Program Analyzer. The Program Analyzer has two major tasks: (1) Parsing the source code into an AST, and (2) Generate meta-information about the input program. The first task is done by the AST parser and the second is done by the Static Code Analysis.

```python

def sum_array(numbers):
    sum_all = 0
    for n in numbers:
        sum_all = sum_all + n
    return sum_all

def test1():
    sum  = 1
    numbers  = [1]
    assert  sum_array(numbers) == sum

def test2():
    sum  = 55
    numbers  = [1,2,3,4,5,6,7,8,9,10]
    assert  sum_array(numbers) == sum

def test3():
    sum  = 30
    numbers  = [10,10,10,-10,9,1]
    assert  sum_array(numbers) == sum

def test4():
    sum  = -1
    numbers  = [1,2,3,4]
    assert  sum_array(numbers) != sum

```

(a) Tyro's Input: Sequential Python Program

```python
  import pyspark as ps

def sum_array(numbers):
    sum_all = 0
    sc = ps.SparkContext()
    sum_all_RDD = sc.parallelize(numbers)
    sum_all=sum_all_RDD.sum()
    sc.stop()
    return sum_all

def test1():
    sum  = 1
    numbers  = [1]
    assert  sum_array(numbers) == sum

def test2():
    sum  = 55
    numbers  = [1,2,3,4,5,6,7,8,9,10]
    assert  sum_array(numbers) == sum

def test3():
    sum  = 30
    numbers  = [10,10,10,-10,9,1]
    assert  sum_array(numbers) == sum

def test4():
    sum  = -1
    numbers  = [1,2,3,4]
    assert  sum_array(numbers) != sum

```

(b) Tyro's Output: PySpark Program

Figure 3.2. Translation of a sequential Python program into a PySpark program by Tyro

3.2.1 AST Parser

Tyro uses the standard Python library called Python AST(Van Rossum and Drake 2009) for parsing the input program into ASTs. The Python AST module is a standard library provided by Python for parsing Python programs into trees based on the Python abstract syntax grammar. The abstract syntax itself might change with each Python release; this module helps to find out programmatically what the current grammar looks like. For every version change or update in the Python language, the AST of that program also changes. Developers usually add/remove features or even change the structure of the syntax tree. So, this library enables parsing the ASTs without worrying about the version changes. In Figure 3.3, a simple visualization of the function *sum_array* shown in Figure 3.2a is converted into a AST using the

Python AST library provided by Python. This syntax tree is passed to the *Static Code Analysis* for further processing.

3.2.2 Static Code Analysis

The syntax tree parsed by the AST Parser is traversed using functions within the tree with help of the same Python AST library. The parsed AST is now analyzed to gather information about the input program. With the walk feature of the AST library, the Static Code Analysis generates all of the information of the various functions, and input parameters. The information generated at this stage is stored in a JSON like structure called *meta-information* (Section 3.2.3). Figure 3.4 shows the basic structure of the meta-information with all its substructure. In the Static

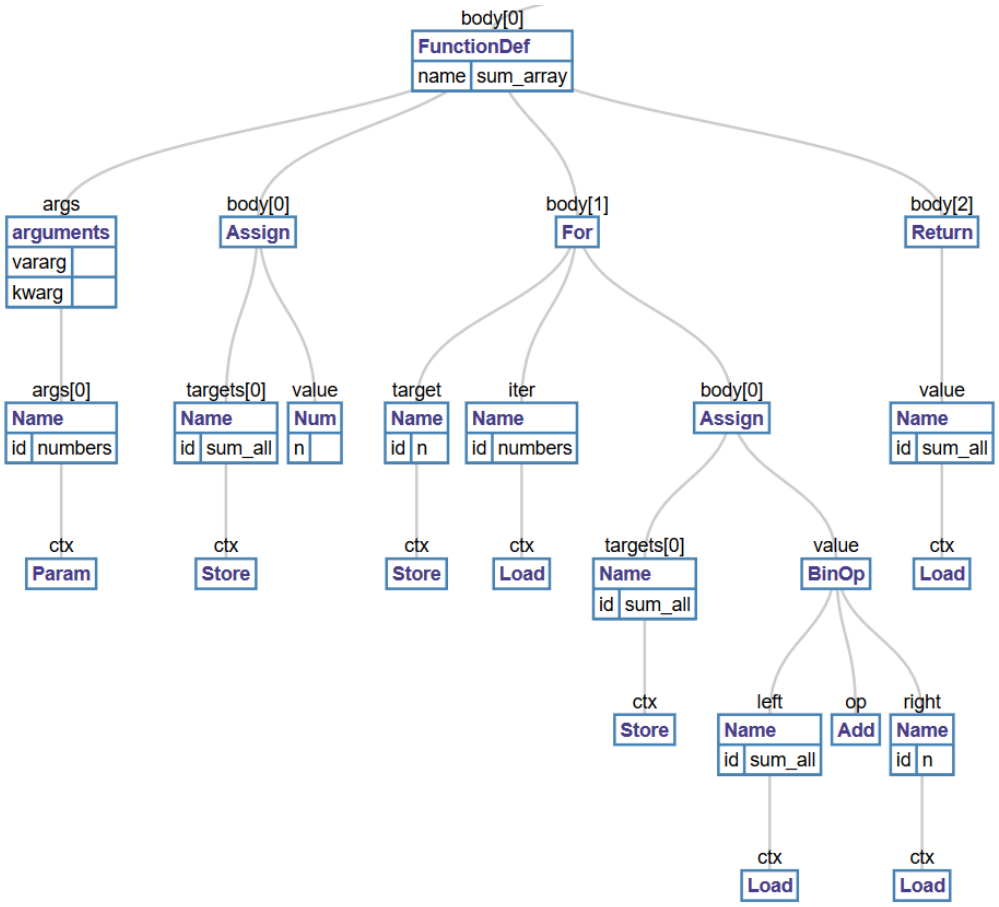


Figure 3.3. Conversion of a Python function into Abstract Syntax Tree (AST)

Code Analysis, Tyro only accesses information like the filename, the lines of code, the imports, and the datasets. The parameter filename which is the filename of the input program is used when generating the output and is a piece of additional information not available in the parsed syntax tree. The Python file read operation provides the filename for Tyro. Further, it also stores all the functions found in the input program using the Python AST library. In Figure 3.3, we can see that the root body[0] consists of an AST instance of *FunctionDef*. The Static Code Analysis traverses over all of the nodes provided by the AST parser and checks for instances of *FunctionDef*. With each detection, Tyro adds the AST node of the function and its function name to the substructure function information in the meta-information.

Tyro only scans for function nodes since Python is more of a functional programming language. Tyro assumes that each function has an independent task to complete. More complex function interaction is a future work (Section 5.1).

3.2.3 Meta-information

The meta-information structure has four different substructure: Program Information, Function Information, Loop Information, and Operation Information. Figure 3.4 shows meta-information and its substructure.

The Program Information stores all of the program information like Filename, line of codes, data sets, and imports used in the original program. All of the functions present in the program are stored as an array of function information. The substructure Function Information stores the initial and the final line numbers, the parameters of the function, and so on. The AST node of the function itself is stored in the function information because if the nodes are not stored, Tyro has to repeatedly search the whole tree every time for the information. The function information also stores the information of all loop fragments present in the function. The Loop Information substructure stores all of the information regarding a loop including the AST node for that loop as well. It holds flags for a filter, a User Defined Functions (UDFs) called

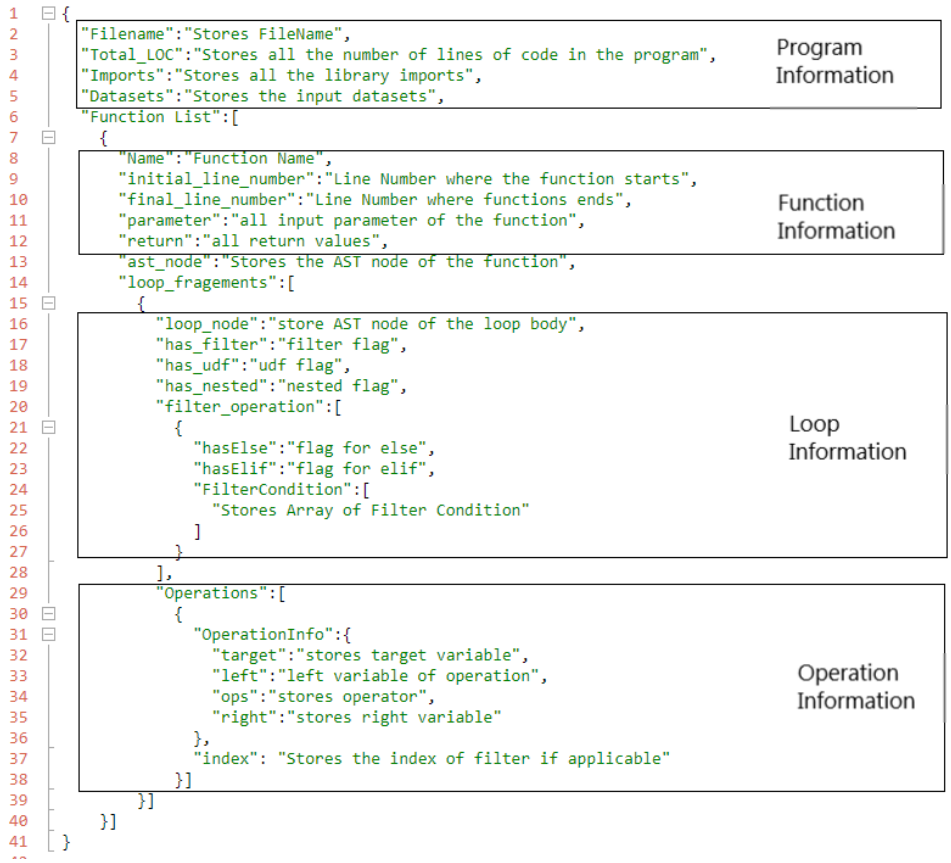


Figure 3.4. Meta-information structure used in Tyro

in the loop, and if the loop is nested. These components are later used in the gradual program synthesis by the Component Transformer. Loop Information also stores all of the operations inside that loop fragments as an array of Operation Information. The operation information stores the information about operations along with the filter index if applicable. It stores the transformed operation of the extracted AST node.

The meta-information in Tyro is a Python class, however for simplicity the structure is expressed in JSON as shown in Figure 3.4. For example, the meta-information of the input program in Figure 3.2a is represented in Figure 3.5 after completion of all required stages by Tyro. The extraction and conversion of the meta-information is discussed further in Section 3.3 and Section 3.4.

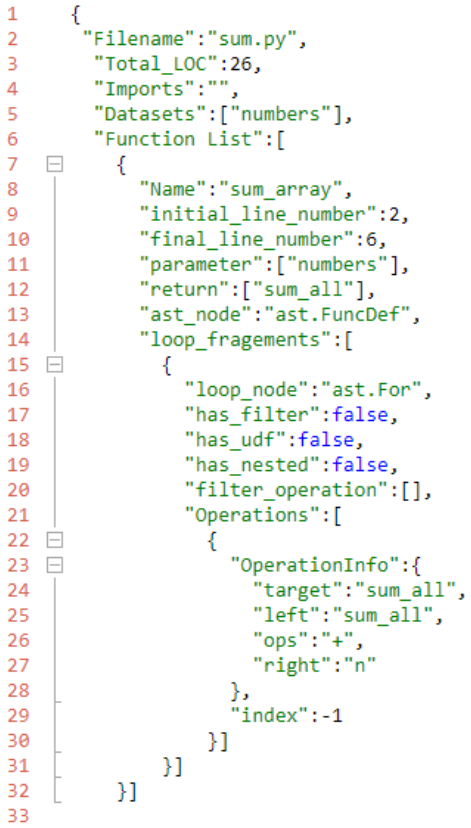


Figure 3.5. Meta-information after the completion of all steps in Tyro

3.3 Feature Extractor

The extraction process is a crucial phase of Tyro. Here, operations that can be translated are extracted and passed on to the next stage. Since Tyro is converting sequential operations into PySpark operations, it must figure out where and how these conversions can be done. It again uses the Python AST library and the meta-information to extract the required operations, loops, and function bodies and converts them to PySpark operations, if required.

The Feature Extractor has three major components: a *Loop Extractor* (Section 3.3.1), an *Operation Extractor* (Section 3.3.2), and an *Operation Converter* (Section 3.3.3). Each component modifies the meta-information to include more accurate representations.

First, the Loop Extractor iterates over the list of the AST functions nodes in the meta-information. These functions nodes are already stored in the meta-information by the Static Code Analysis (Section 3.2.2). The Loop Extractor scans each function node using the walk feature of AST. Each loop fragment detected in each function updates the substructure loop information in the meta-information.

The Operation Extractor then iterates through each loop node in the meta-information and captures all of the operations present inside these loop fragments. The captured operation fragments/nodes are passed to the Operation Converter.

Finally, the Operation Converter converts all of the arithmetic and logical operations previously extracted to an intermediate form so that conversion from Python Operation to PySpark Operation becomes easy. All of the components of the extraction stage use the Python AST library to walk over the different extracted AST nodes.

3.3.1 Loop Extractor

Each function's AST node has already been stored in the meta-information by the Program Analyzer. The Loop Extractor searches for loop fragments within each function's AST node stored in the meta-information. With the help of the Python AST library, it processes each node recursively. From the documentation of Python AST, there is no guarantee about the order of the node. But it guarantees that it will scan through every node present in the tree (Van Rossum and Drake 2009).

First, the Loop Extractor analyzes each loop body for static information like line numbers and the input data set of the loop body. The line numbers help us to replace the loop body with a new PySpark operation in the Code Generator phase. Likewise, flags like filter checks, UDFs checks, and nested loop are set in this stage. These flags denote whether the loop fragment has control statements, user-defined function calls, or a nested loop within the loop body.

The filter flag is set to true if the loop body has instances of some control structure like *ast.If*. If there is another user-defined function call within the loop fragment then the UDF flag is set to true. If the loop body has another instance of a for loop then Tyro assumes it as a nested loop and sets the nested loop flag. Currently, Tyro only scans instances of *ast.For* (for loop fragments). Other loop instances such as *ast.While* are future work discussed in Section 5.1.1.

The Loop Extractor only updates the Loop Information of the meta-information. Figure 3.6 is the updated substructure (Loop Information) of the meta-information for input program in Figure 3.2a . The flags *has_filter*, *has_udf*, *has_nested* have been set false as they do not pass the flag checks of the Loop Extractor. The filter condition in the substructure is empty because it will be processed late by the Operation Extractor (Section 3.3.2) to extract all of the filter condition if there exists. Likewise, the operation list is also empty in this stage because the Operation Extractor will extract operations after the loop extraction is complete.

```
{
  "loop_fragments":[
    {
      "loop_node":"body[1] ast.For",
      "has_filter":"false",
      "has_udf":"false",
      "has_nested":"false",
      "filter_condition":"",
      "operation_list":[]
    }
  ]
}
```

Figure 3.6. Updated Loop Information substructure of the meta-information after Loop Extractor with all flags

3.3.2 Operation Extractor

After the Loop extractor is complete, Tyro moves to a lower level detail operations extraction where operations such as arithmetic operations and logical operations within a loop fragments are processed. The Operation Extractor has two phases: a *Arithmetic Operation Extraction*, and a *Logical Operation Extraction*. The Logical Operation Extraction executes only if the Loop Information in the meta-information has the filter flag set to true.

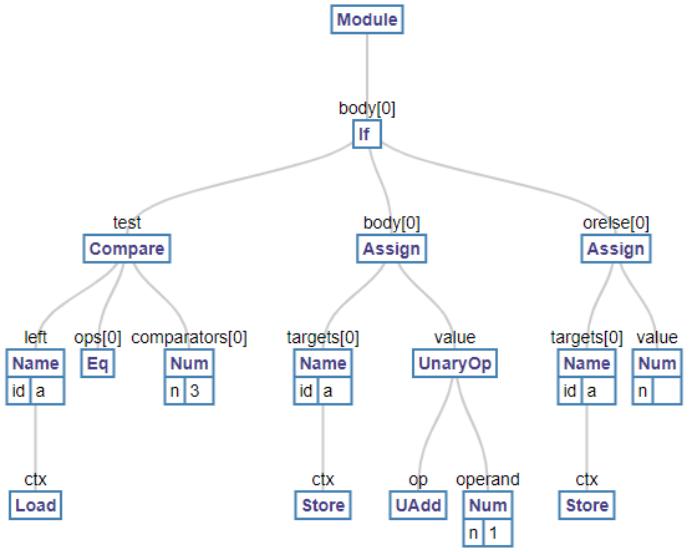
3.3.2.1 Logical operation extraction. If the meta-information states that there are logical operations within a loop body, then the logical extraction process is triggered. The nodes in the Loop Information substructure are searched for instances of *ast.If, ast.elif, ast.else* and appends AST nodes to the filter list. The filter list is a temporary structure that stores all of the filter operations before it gets converted by the Operation Converter (Section 3.3.3). These nodes are still stored in the structure of *ast.Node* as shown in Figure 3.7. For example, a simple Python code with if statements have the following AST structure as shown in Figure 3.7.

Furthermore, if any arithmetic operation is present inside the logical operation, it appends the filter index that denotes which operations are inside the block of the filter operation. In Figure 3.7, the filter module has two branches: *test* and *body*. The Logical Operation Extraction after detecting the *ast.If* instance extracts the node and appends into a temporary structure called Filter List. Figure 3.8 is the updated structure after the logical operation extraction is complete for the node shown in Figure 3.7. The variable *FilterOperations* in the structure is empty at the moment because the Arithmetic Operation Extraction extracts all of the arithmetic operations to the next step.

3.3.2.2 Arithmetic operation extraction. Tyro discovers the operation performed inside the loop fragments. If the filter flag in the meta-information is true then

```
1  if a == 3:
2      a += 1
3  else:
4      a = 0
5
```

(a) Filter Operation in Python



(b) Converted Abstract Syntax Tree of operation on (a)

Figure 3.7. AST structure of a filter operation

the extraction process begins by processing the filter list passed by the logical operation extraction. Figure 3.8 on *Line 2* shows all of the filter operation nodes stored. The arithmetic operation extraction processes each node present in the filter list and searches for instances such as *ast.Assign*, *ast.AugAssign*, *ast.BinOp*, *ast.UnaryOp*. After each such instance is found, the arithmetic extractor appends the filter operation with its filter index as shown in Figure 3.9.

After the extraction of arithmetic operations from the filter node, the arithmetic operation extraction processes the node apart from filter nodes. In this next step, the Arithmetic Extractor processes the remaining nodes in the Loop Information substructure. The operation list updates with similar instances found by the Arithmetic Extractor. The operation list is another temporary structure that stores all operation nodes in a loop fragment. Figure 3.10 shows a temporary structure that stores all the remaining arithmetic operations in the loop fragments. The operation list and filter list are passed to the Operation Converter (Section 3.3.3) to convert these complex node structures into an intermediate representation.

```
{
    "FilterList": ["ast.IfCompare","ast.OrElse"],
    "FilterOperations":[]
}
```

Figure 3.8. Temporary structure of Filter List

```
{
    "FilterList":["ast.If, Compare","ast.OrElse"],
    "FilterOperations":[["ast.AugAssign",0],["ast.Assgin",1]]
}

```

Figure 3.9. Complete Filter List after Operation Extraction inside a filter operation

3.3.3 Operation Converter

After the extraction of all of the logical and arithmetic operations, Tyro converts these complex nodes into an intermediate form using Operation Converter. It starts the conversion by converting each operations from the *FilterList* shown in Figure 3.9. While AST nodes can be very complex, Tyro only requires a few pieces of information from the node itself. To filter data from the given dataset, we only require *left, ops,* and *comparator* as shown in Figure 3.3. The *ops* in the node is stored as an instance of the Python AST like *ast.EQ* is "==". These ops instances are converted into their logical forms such as $<, >, <=$, and so on. Figure 3.11 shows the result of the filter conversion of the AST node shown in Figure 3.7. The *'0'* in *Line 9* of the Figure 3.11 is the filter index number. The index number is later used to identify the correct operation inside its body. The filter flags *hasElse and hasElif* is later used by Code Generator and Component Transformer discussed in Section 3.5 and Section 3.4 respectively.

The operation converter has another task to convert the arithmetic operations to an intermediate representation such that complex AST nodes are easier to process.

```
{
    "OperationList": ['ast.AugAssign', 'ast.BinOp']
}
```

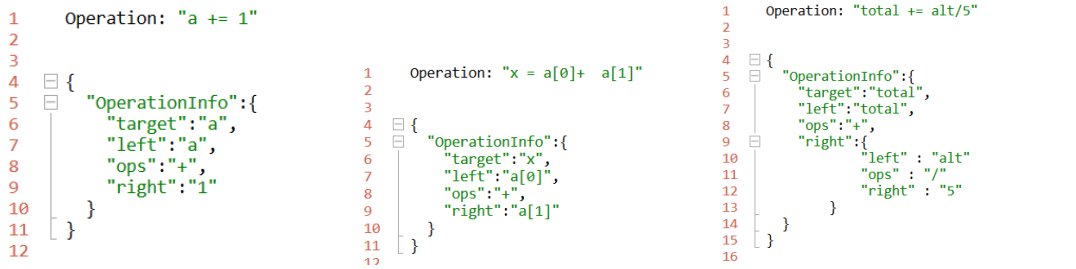
Figure 3.10. Temporary structure operation list after extracting all arithmetic node in a loop fragment

The intermediate representation converts an AST node structure shown in Figure 3.7 into a binary operation like $a = a + 1$.

In Figure 3.3, the *body* node has the arithmetic operation that needs to be converted into an intermediate representation. The binary structure of the operation should of the form *target = left 'op' right* where op denotes the operator. Also, in Figure 3.7b, we can see the instance of the body is *ast.AugAssign* a unary operation. The unary operation is converted by simply making the 'left' operation as the target and the value as 'right' as shown in Figure 3.12. The meta-information gets updated with the intermediate form. Again, the Python AST library helps in all these conversions.

```
{
  "FilterOperation":{
    "hasElse":"False",
    "hasElif":"False",
    "FilterCondition":[
      [
        "==",
        "3",
        "0"
      ]
    ]
  }
}
```

Figure 3.11. Filter structure after operation conversion



(a) Unary Operation (b) Binary Operation (c) Complex Binary Operation

Figure 3.12. Different operation conversion in Tyro

Figure 3.12c shows the nested structure of complex operations that are created by recursively parsing the AST. Once the operation conversion process is complete, the meta-information is updated with converted operations and passed forward to the Component Transformer stage.

3.4 Component Transformer

Once the feature extractor phase is complete, the component transformer phase begins. In this phase, gradual synthesis modifies the results. Initially, the components are transformed into Map operations, but if the result fails the Verifier, the synthesis stage increases. Tyro then moves to more complex operations like Reduce. Currently, Tyro works only on lists of numbers i.e on large-scale arrays of numbers. More complex data types and file handling is discussed as future work in Section 5.1.2.

Tyro has already converted the extracted operation into an intermediate form. These converted or intermediate structures are stored in the meta-information. Tyro utilizes gradual synthesis for transforming these Python operations to PySpark operations. The different stages in Tyro's gradual synthesis are: *Map Transformer* (Section 3.4.1), *Reduce Transformer* (Section 3.4.2), *Filter Transformer* (Section 3.4.3), *User Defined Function (UDF) Transformer* (Section 3.4.4) and *Join Transformer* (Section 3.4.5). Tyro generally begins with the Transformer phase from Map Transformer.

However, if its nested loop or UDF flag is true, then the transformation stars with the Join transformer or the UDF transformer. The Map transformer converts the extracted operation into a Map operation of PySpark. The Code Generator then uses the transformed operations which generate the PySpark codes. After that, the verifier uses unit tests provided by the user. If the verification is successful, the synthesis is complete. However, if the verification fails, the gradual synthesis moves into another stage by increasing the global synthesis variable.

Currently, our domain of sequential program analysis is covered by these stages. These stages can also be interrelated with each other as the Reduce transformer stage can use the Filter Transformer to get all the filter components of the current loop fragments. Similarly, a UDF Transformer can use Map or Reduce Transformer to convert an operation into Map or Reduce operation in PySpark. Before explaining Tyro's Component Transformer, let us look at PySpark operations. There are two types of PySpark operations: *Transformations* and *Actions*.

The transformation is a function that produces new *RDDs* from the existing RDDs. *RDD* stands for Resilient Distributed Dataset and it is a fundamental structure of PySpark. An RDD is an immutable collection of objects. Actions are PySpark functions that return a value to the driver program after running a computation on the dataset. For example, a map is a transformation that passes each dataset element through a function and returns a new RDD representing the results. On the other hand, a reduce is an action that aggregates all of the elements of the RDD using some function and returns the final result to the driver program. There are more than 80 different PySpark operations. Currently, Tyro supports the map, filter, reduce, join, and zip operations of PySpark. Furthermore, Tyro utilizes built-in function like Sum(), Count(), Min(), Max(). These built-in functions are also the action operations of PySpark so they generate a value rather than an RDD.

3.4.1 Map Transformer

In this step, an operation is converted into a Map operation in PySpark. A map in PySpark is a transformation that returns a new RDD by applying a function to each element of the current RDD. Figure 3.13 shows a simple PySpark map where an input RDD at *Line 1* is transformed using a map function as *Line 2* and the output is a new RDD shown at *Line 5*. For simplicity, we have shown the output in a list, however, we need to used another Action operation called *collect* to actually convert an RDD to an list.

The meta-information makes this transformation easy as Tyro can wrap the converted operation with a lambda function passed to the map operation. For example, a converted operation like *a = a + 1* is wrapped with a lambda function as *lambda a: a+1*. Likewise, the lambda function is then wrapped by a map operation as *map(lambda a: a+1)*. In this way, a converted operation is transformed into the Map operation of PySpark. However, if there are two or more operations in the loop fragment, then each operation is converted into a separate map operation in PySpark. The conversion process is the same for each operation. If there is a function call in the input program, a for example function called test() is present inside the loop fragment. It is then the map operation changes to *(map(test))* to leverage PySpark optimization. Figure 3.18 shows the efficient use of a UDF call in PySpark.

Map Transformer adds each transformed map operation to a list called the Mapper List. The Code Generator later uses this mapper list as discussed in Section 3.5. If the filter flag in the meta-information is true then the Filter transformation (Section 3.4.3) converts Python operation into filter operation. Table 3.1 below shows some map transformations. In the third transformation, the result is produced by the Reducer Transformer after it creates a map. The Map lambda function in the third transformation uses the dimension of each row in the list to determine the lambda value.

```
1   rdd = sc.parallelize([1, 2, 3, 4])
2   rdd_new =  rdd.map(lambda  a: a - 1)
3   rdd_new.show()
4
5   [0,1,2,3]
```

Figure 3.13. PySpark Map operation where an RDD is transformed to another RDD with a lambda function (*sc* is the SparkContext)

Table 3.1. Map Operation Transformation in Tyro

Python Operation	PySpark Operation
n1 = n1 + 2	map(lambda n1: n1+2)
total *= 10	map(lambda x: x*10)
sumall = sumall + (n[0] + n[1])	map(lambda x:(x[0],x[1]))

3.4.2 Reduce Transformer

After the failed verification of the Map Transformer, the global synthesis counter is increased by 1 which means the gradual synthesis moves to the Reduce Transformer. The Reduce Transformer converts operations to the reduce operation of PySpark. A reduce is a PySpark action that aggregates a data set (RDD) element using a function. The function can be a lambda function or a named function. Figure 3.14 shows a reduce operation in PySpark. We can see that reduce returns values instead of an RDD. At *Line 3*, a reduced operation of PySpark calls for another function. The optimization of PySpark helps in calling a reducer without a map function. The Reduce transformation is similar to Map transformation in that Tyro wraps the converted operation into the reduce syntax. However, there are few more details in a reduce operation. Reduce Transformation uses a new variable called an accumulator in the reduce stage. An accumulator is a shared variable that has a commutative and associative operation applied to it. For example, you can use an accumulator for a Sum or Count. So to convert an operation in reduction, Tyro adds an accumulator. For example, in Figure 3.12c, the operation $total+ = alt/5$ is already converted into an intermediate representation that is then converted into a lambda operation where

```
1   from operator import add
2   rdd = sc.parallelize([1, 2, 3, 4])
3   total = rdd.reduce(add)
4   print(total)
5
6   >>10
```

Figure 3.14. Reduce operation in PySpark where sum of the values of the RDD is returned back to driver program (*sc* is the SparkContext)

Tyro introduces the accumulator. Tyro now identifies that the variable *total* is the accumulator in this case. Tyro transforms the structure in the following way ($lambda\ total, alt : total + (alt/5)$). This lambda function is then wrapped by reduce syntax as $reduce\ (\ lambda\ (total, alt : total + (alt/5)))$. The key difference in the Map and Reduce transformation is the accumulator is not used in the Map stage. Similar to the map operation, the reduce operation can also call a function. The process of converting multiple operations within a loop fragment into a separate operation is similar to Map transformation. Table 3.2 shows a few reduce transformation for given Python operations.

If the dataset is a multidimensional list, we need to call the Map transformation to provide a map operation for the list. In Table 3.1 the 3rd transformation shows the map operation returned by the Map Transformation. The returned map operation is concatenated with reduce operation and this concatenated operation is added in the Reduce list. The Reduce List is similar to the Mapper list that holds transformed operations. The reduce list is passed to Code Generator (Section 3.5).

Likewise, the Reduce Transformer uses the filter flag to determine the Filter Transformer (Section 3.4.3). If the flag is true, the transformation moves to the Filter Transformer and repeats the process of Code Generator and Verifier (Section 3.6). Tyro can also use some of the builtin functions of PySpark by calling aggregate function like *Sum(), Count(),* and *Max()*. In PySpark, the aggregate function is a

Table 3.2. Reduce Transformation in Tyro

Python Operation	PySpark Operation
for n in numbers:	
sum = sum + n	*reduce*(lambda ac, n: ac+n)
return sum	
c *= 10	*reduce*(lambda ac, n: ac * 10)

reduce operations as well. These functions return values for a given RDD to the driver program. So the Reduce Transformer does one extra step of converting a reduce operation to the aggregate function. It is done by pattern matching in a compiler. If the current operation matches with a collection of predefined objects shown in Table 3.3, then the reduce operation replaces the operation with a builtin PySpark function. For example, if the lambda function adds 1 to a variable per element the reduce operation will match with Count and replaces it by the Count PySpark operation. These aggregate functions are an additional feature of PySpark which Tyro is leveraging.

3.4.3 Filter Transformer

The filter flags in the meta-information triggers the filter transformer which does not work by itself and is called via the Map or the Reduce transformer. The goal of this transformation is to convert the filter structure in the meta-information and convert them to the filter operation of PySpark.

A filter in PySpark is another transformation operations as it results in another RDD after executing a function for each element in the current loop. A new RDD

Table 3.3. Conversion of the matched reduce operation into aggregate functions by Tyro using pattern matching

Transformed Operations	Matched Operation
reduce(lambda ac, n: ac + n)	sum()
reduce(lambda ac, n: ac + 1)	count()
filter(lambda n: n <min).reduce(lambda n : n)	min()
filter(lambda n: n >max).reduce(lambda n : n)	max()

is returned containing the elements, which satisfy the function inside the filter. Note that though the filter transformation is not involved directly in the gradual synthesis, it is an important step within the other transformations. All of the filter criteria is in the meta-information.

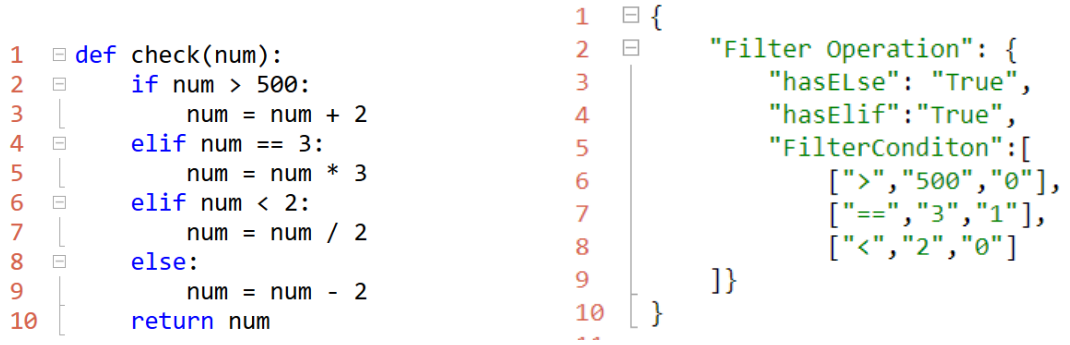
The filter transformer has three different sub phases that converts the filter structure into a filter operation in PySpark: *if transformation*, *elif transformation*, and *else transformation*. To show how this transformation is done, we will look at the example code in Figure 3.16a. Figure 3.16b shows the converted structure. The structure has already been added to the meta-information.

The *if transformation* works on the first condition of the filter condition as shown in Line 6 in Figure 3.16b. The first condition is denoted by the index *0* in the filter condition. The transformation algorithm is similar to the map and reduce transformation. For example, the first condition is transformed by adding the lambda function to the condition i.e *(lambda a: a > 500)*. The lambda function is wrapped with filter syntax. The *elif transformation* executes if there is more than one condition in the filter condition and also if the hasElif flag is set to true as shown in Figure 3.16b. The transformation algorithm is same as *if transformation*.

However, the *else transformation* is very different from if and elif transformations. In this case, the filter conditions are not present for else statement (See line 8 Figure 3.16a). Tyro generates a filter by negating all of the know conditions and

```
1  rdd = sc.parallelize([1, 2, 3, 4, 7, 9])
2  filterRDD = rdd.filter(lambda x : x  > 5)
3  print(filterRDD.collect())
4
5  >> [7, 9]
```

Figure 3.15. Filter in PySpark



```
def check(num):
    if num > 500:
        num = num + 2
    elif num == 3:
        num = num * 3
    elif num < 2:
        num = num / 2
    else:
        num = num - 2
    return num
```

(a) Input Filter Code Fragment

```
{
    "Filter Operation": {
        "hasELse": "True",
        "hasElif":"True",
        "FilterConditon":[
            [">","500","0"],
            ["==","3","1"],
            ["<","2","0"]
    ]}
}
```

(b) Converted Filter Structure

Figure 3.16. Conversion of if/else statements (a) to filter structure (b) in Tyro

creating a conjunct from them. For example, the else transformation of 3.16a would be (lambda a: not(a $>$ 500) and not (a $==$ 3) and not (a $<$ 2)).

Figure 3.17 shows the final result from filter transformations of the input code shown in Figure 3.16a. Tyro converts each filter condition from the meta-information to a lambda function then wraps it with filter operation as discussed above.

The Filter Transformation can also be executed after Map and Reduce Transformations. Both of these transformations are executed if the filter flag in the meta-information is set True. The Map or Reduce Transformer executes the Filter Transformer when the filter operations are converted to a PySpark filter operation applied to an already converted map or reduce operation. The new operation is then transferred to Code Generation and further to the Verifier. Table 3.4 shows examples of filter operations with map/reduce operations.

```
filter(lambda a:a>500)
filter(lambda a:a==3)
filter(lambda a:a<2))
filter(lambda a : not (a>500) and not (a==3) and not (a<2))
```

Figure 3.17. Filter Transformation in Tyro

Table 3.4. Concatenated filter operations after Filter Transformation in the gradual synthesis of Tyro

Python Operations	PySpark Opertation
if total == 123: total *= 10	filter(lambda x: x==123).map(lambda x: x*10)
if h <= 2: total = total + (alt/5)	filter (lambda h: h<=2).reduce(lambda total, alt: total + (alt/5))
if n >2: c += 1	filter(lambda n: n >2).count()

3.4.4 UDFs Transformer

User-defined functions are a key feature in PySpark. UDFs allow programmers to go beyond the builtin function available and process data in different ways. In this way, the programmer can utilize the benefits of parallel computing. For this stage, the UDF flag is set to true and the synthesis process starts from the UDF transformation instead of the Map transformer. The UDF flags are only set if there is another function call inside the loop fragments. Starting from the UDF Transformer stage does not mean Tyro does not execute the Map or the Reduce Transformers, it just starts with UDFs Transformer to make sure the Map and the Reducer Transformers converts their operation as a user-defined function call. The gradual synthesis is the same as explained above in the Map and the Reduce Transformer. From Figure 3.18, we can see that PySpark handles UDF similar to how lambdas are handled in Python or Java.

```
1  def add_one(value):
2      return value + 1
3
4  collection_rdd = sc.parallelize([1,2,3,4])
5  collection_rdd = collection_rdd.map(add_one)
6
7  >> [2,3,4,5]
```

Figure 3.18. Example of user defined function (UDF) in PySpark

Using this key feature of PySpark, Tyro generates a function call from the map or reduce stages according to the synthesis. However, for the reduce function, an additional parameter like the accumulator variable might be needed for the function to still work. The whole process inside the UDFs transformation starts from the Map transformation and proceeds to further stages. If the verification stage is successful, the synthesis is complete. However, if the verification fails, Tyro moves to the reduce transformation without any changes and repeat the same process. Again, if it fails, we move to change the function parameters i.e adding the accumulator which is done using the global synthesis counter. Figure 3.19b shows UDF transformation in Tyro after successful verification (discussed in Section 3.6).

3.4.5 Join Transformer

Tyro uses the Join transformer to utilize the join feature of PySpark. For this process in the synthesis, Tyro has a few assumptions about the input program. Tyro assumes that operations are only inside the inner loop. The operations inside the outer loop are not currently handled. Also, as assumed by the join operation in PySpark, Tyro assumes that the join key is the first field in each row. Further work to relax such assumption in this stage is discussed in Section 5.1.3.

```python
def check(a, n1):
    return a + n1

def sum_array(numbers):
    num = 0
    for n in numbers:
        num = check(num, n)
    return num
```

(a) Input Python Code with UDF

```python
import pyspark as ps

def check(a, n1):
    return a + n1

def sum_array(numbers):
    num = 0
    sc = ps.SparkContext()
    num_RDD_0 = sc.parallelize(numbers)
    num = num_RDD_0.reduce(check)
    sc.stop()
    return num
```

(b) Converted UDF

Figure 3.19. Transformation in Tyro

Similar to the UDFs transformation, the Join transformer also has a flag in the meta-information. So, Tyro starts from this stage if the nested loop flag is true regardless of other flags including the UDF flag. However, the transformation is a bit different. First, Tyro checks if there are operations other than assignment or appending of data inside the loop. Assignment and appending of data indicates there are no instances of *ast.BinOP* and *ast.AugAssign*. If there is none, we simply use the join operation of PySpark to join two RDDs. In PySpark, join returns a RDD with a pair of elements with the matching keys and all the values from both rows. Figure 3.20 shows how the nested loop is joined together. The initial two lines of the output are generated by the Static Code Generation (Section 3.5.1) phase where it converts the input dataset into a RDD.

Once the join has been identified, other operations can be handled as before starting with the UDF Transformer (Section 3.4.4) if the UDF flag is set to true and if the flag is false, then it starts with the Map Transformer and continuing with gradual synthesis. The process is similar as described in Section 3.4.1 and Section 3.4.2 except the input data set is the RDD created after the join.

```
for i in data_1:
    for j in data_2:
        if i[0] == j[0]:
            data_3.append((i[0] , i[1],j[1])) # key, tuple
```

```
data_1_RDD = sc.parallelize(data_1)
data_2_RDD = sc.parallelize(data_2)
data_1_RDD_combine =data_1_RDD.join(data_2_RDD)
```

Figure 3.20. Simple Join Operation Conversion by Tyro

3.5 Code Generator

After all of the operations are transformed from Python into PySpark, Tyro needs to generate a PySpark program. The Code Generator has two different phases: *Static Code Generation* and *Dynamic Code Generation*. Tyro starts the code generation phase with the original Python program. In addition to the original program, the code generation uses two pieces of information (a) the updated meta-information and (b) the transformed operations list. Combing these three inputs, the code generator generates a PySpark code version of a given Python program.

3.5.1 Static Code Generation

PySpark programs need a specific environment to execute. This requires loading the PySpark modules and initializing a SparkContext. A SparkContext represents the connection to a Spark cluster and can be used to create RDDs and broadcast variables on that cluster. Figure 3.21 is the generated code after static code generation of the input code shown in Figure 3.19a. The generator adds the commands at *Line 1*, where it imports the PySpark libraries. Likewise, at *Line 8* it starts the SparkContext using ps.SparkContext(). Once a SparkContext is initialized a flag is set so that it is not initialized again. For each function, Tyro generates a single SparkContext. Each SparkContext initialized in the function is stopped in the same function by the Dynamic Code Generation (Section 3.5.2). This is done for making the unit testing execute properly. Also, in this phase, all of the input datasets are converted into RDDs. Currently, Tyro works only with a list of numbers so it is straightforward to convert all of the input datasets for each block as RDDs created by the SparkContext (Line 9 in Figure 3.21). After the addition of each line in the Static Code Generation, the line numbers in the meta-information is also increased by 1.

```
1    import pyspark as ps
2
3    def check(a, n1):
4        return a + n1
5
6    def sum_array(numbers):
7        num = 0
8        sc = ps.SparkContext()
9        num_RDD_0 = sc.parallelize(numbers)
10       for n in numbers:
11           num = check(num, n)
12       return num
```

Figure 3.21. Static Code Generation in Tyro for input program in Figure 3.19a

3.5.2 Dynamic Code Generation

After static code generation is complete, the Dynamic Code Generation (DCG) uses the meta-information and converted operations list to generate PySpark code. The DCG edits the file generated by the static code generation. The meta-information provides the updated line numbers where each loop fragment is present. With that information, DCG replaces the loop fragments with converted operations. The operation list given by the Component Transformer will have operations according to the synthesis stage. For example, it be a might map operation if the synthesis stage is Map Transformation or an aggergate function from the Reduce Transformation. If the synthesis stage is UDF with the addition of an accumulator, then the changed function is also received as a parameter for DCG. The functions replace the existing UDF. An example of such a conversion is discussed in Section 4.4. This phase is called the Dynamic Code Generation because the work of this phase changes according to the synthesis stage. At the end of the function and before the return statement, DCG stops the SparkContext by adding *sc.stop()*. If Tyro does not add this step, the connection won't close and multiple test cases in the same program cannot be executed for the same function. Figure 3.22 shows the results after dynamic code generation

```
1   import pyspark as ps
2
3   def check(a, n1):
4       return a + n1
5
6   def sum_array(numbers):
7       num = 0
8       sc = ps.SparkContext()
9       num_RDD_0 = sc.parallelize(numbers)
10      num = num_RDD_0.reduce(check)
11      sc.stop()
12      return num
```

Figure 3.22. Generated PySpark code by the DCG for code in Figure 3.21

for the code in Figure 3.21. Here, the loop fragments on *line 10,11* in Figure 3.21 is replaced by the PySpark operation reduce calling a UDF. Note that Figure 3.22 is generated after the first synthesis step of Map Transformer fails. The failed synthesis is discussed in Section 3.6.

3.6 Verifier

In this step, Tyro decides whether a generated PySpark program is equivalent to an input Python program or not. In general, program equivalence is undecidable, so Tyro uses the concept of unit testing (IEEE 1990) from software engineering. A unit test is a level of software testing where individual units/components of a software are tested. The purpose is to validate that each unit of the program performs as designed. For the input program, the user needs to provide the unit tests. Tyro assumes that the input program passes the tests provided. These unit test cases are executed by Tyro to check whether the generated code is equivalent or not. The test cases are embedded in the input sequential program as shown in Figure 3.23.

With the help of a Python package called *pytest-spark*, Tyro can test the generated program without any modification. Since Tyro starts and ends its SparkContext within each function the problem of multiple connections does not affect the testing.

```
1  def test_1():
2      expected = 6
3      result = sum_array([1,2,3])
4      assert result == expected
5  def test_wrong():
6      expected = 8
7      result = sum_array([1,2,3,4])
8      assert result != expected
9  def test_2():
10     expected = -45
11     result = sum_array([10,-50,2,3,-10,7,8,-15])
12     assert result == expected
```

Figure 3.23. Test cases provided by the user for the input program in Figure 3.19a

Also, Tyro does not need to worry about importing any libraries while testing because the pytest can be executed by the *subprocess* module available in Python. The pytest module checks for functions with an assertion statement in it.

To successfully verify the generated program, all the unit tests should pass. For this process, Tyro uses another Python module called *subprocess*. This module allows Tyro to execute a shell or a windows command from a Python program. Further, the subprocess also checks different output pipelines like standard output and standard error. For successful verification, the standard error pipeline returned by the subprocess module should be empty. For Figure 3.19a, the first transformation stage after the UDF is a map stage. However, the map verification fails against the given test case in Figure 3.23 which can be seen in Figure 3.24.

Two test cases fail for the generated code. After the map stage, the output generated is a list of numbers where the test cases require a value. Thus, the assertion in these cases fails. However, one of the test cases is checking if the result is not equal to the expected so the test case "test_wrong" passes the verification. The gradual synthesis proceeds to another stage by increasing the global synthesis counter by one. This starts the next stage of the gradual synthesis. After repeating the whole process

```
platform linux -- Python 3.7.9, pytest-6.0.2, py-1.9.0, pluggy-0.13.1
spark version -- Spark 2.4.2 built for Hadoop 2.7.3 | Build flags: -B
e -Phadoop-2.7 -Phive -Phive-thriftserver -DzincPort=3037
rootdir: /home/hduser/PycharmProjects/ParallelPy/result
plugins: spark-0.6.0
collected 3 items

gen.py::test_1 FAILED
gen.py::test_wrong PASSED
gen.py::test_2 FAILED

============================================================ FAILURES =
```

Figure 3.24. Failed Verification

in the next stage, the code in Figure 3.19b passes the verification as shown in Figure 3.25.

```
platform linux -- Python 3.7.9, pytest-6.0.2, py-1.9.0, pluggy
spark version -- Spark 2.4.2 built for Hadoop 2.7.3 | Build fl
e -Phadoop-2.7 -Phive -Phive-thriftserver -DzincPort=3037
rootdir: /home/hduser/PycharmProjects/ParallelPy/result
plugins: spark-0.6.0
collected 3 items

gen.py::test_1 PASSED
gen.py::test_wrong PASSED
gen.py::test_2 PASSED

============================================================ 3 passed
```

Figure 3.25. Successful verification

Table 3.5 shows all the stages used by Tyro. Note that the nested loop is not represented in the synthesis counter because Tyro uses a nested loop only if the nested loop flag is set to true. The gradual synthesis does not use this step in any other step. Note that if all synthesis stages fails, Tyro cannot convert the program.

Table 3.5. Global synthesis counter with their respective stages used by Tyro

Global Counter	Synthesis Stage
1	Map
2	Reduce with no operation change
3	Reduce with added accumulator
4	UDF with Map
5	UDF with Reduce without accumulator
6	UDF with Reduce with accumulator

Each global synthesis counter represents a separate synthesis process. Tyro uses the global counter to determine its synthesis stage and work accordingly. The dynamic code generator also works according to the synthesis stage.

CHAPTER FOUR

Results and Analysis

In this chapter, we present a comprehensive analysis of Tyro by evaluating its ability to handle the different diverse workloads, finding an efficient translation of code fragments, and performance gain compared to their sequential implementations.

4.1 Evaluation

Tyro was evaluated by translating several different sequential Python programs. Tyro translated 13 different Python programs from 4 different test suites. For all of our translation experiments, we executed Tyro on a Windows 10 Machine with Intel Core i5 1.8 GHz CPU, 8 GBs of RAM and 256 GB of SSD storage with the latest stable version of all frameworks used i.e Spark, Hadoop and Python.

4.1.1 Test Suites

To evaluate Tyro, I created four different test suites that cover different domains.

- *Aggregate Functions* is a test suite that includes the common aggregate SQL functions SUM(), COUNT(), AVG(), MIN(), MAX(). The test suites have five different files with each file performing a single SQL aggregation function.
- *Nested Operations* consists of two different programs. The first program is a simple join operation between two different datasets, while the second is a simple join with further manipulation of the results.
- *Multidimensional Dataset* consists of three different programs which evaluate the translation of multidimensional data in Tyro. The test suite consists of k-Nearest Neighbors (kNN), a simple arithmetic operation and a conditional operation.

- *User Defined Functions* is a test suite consisting of three different user defined functions: Conditional Sum, Conditional Count and User Defined Average. This test suite is used to evaluate the UDF stage of Tyro.

4.2 Aggregate Functions

The aggregate functions suite verifies that Tyro can go beyond simple Map and Reduce operations and utilize the other operations of PySpark. Figure 4.1 is an example of an input program, in this case MAX().The input function iterates over sequential data on *Lines 3 - 4*. Tyro starts with the Map Transformer which also

```
1  def max_array(numbers):
2      max = 0
3      for n in numbers:
4          if max < n:
5              max = n
6      return max
7
8
9  def test1():
10     assert (max_array([1,2,3,4,5,-10,10,2])) == 10
11
12 def test2():
13     assert (max_array([100,2,3,4,5,-10,10,2])) == 100
14
15 def test3():
16     assert (max_array([1,1,1,1,1,1,1])) == 1
17
18 def test4():
19     assert (max_array([1,2,3,4,5,-10,10,2])) != -10
20
```

Figure 4.1. Python program to find the maximum number from a list *(Line 1 -6)* with its test cases *(Line 9 - 20)*

triggers the Filter Transformer since there is a if statement in the loop. The initial program synthesis from Tyro generates a program with PySpark Map and Filter operation as shown in Figure 4.2. The program doesn't pass the verification process, so Tyro starts over with the Reduce transformation in the gradual synthesis.

```
def max_array(numbers):
    max = 0
    sc = ps.SparkContext()
    max_RDD = sc.parallelize(numbers)
    max = max_RDD.filter(lambda n: n > max).map(lambda n: n).collect()
    sc.stop()
    return max
```

Figure 4.2. Generated PySpark code with Map Operation by Tyro

In the reduce transformation, the pattern of operation matched with one of the built in the function *MAX()*. Instead of using the reduce operation, Tyro uses this built in function. The code in Figure 4.3 is successfully verified. Note that Tyro identifies the loop fragments and the dataset and converts it to PySpark code that can be executed in a parallel environment.

The next experiment performed in Tyro was calculating the average from a list of numbers. Figure 4.4 is the input program for Tyro. The translation process

```
import pyspark as ps
def max_array(numbers):
    max = 0
    sc = ps.SparkContext()
    max_RDD = sc.parallelize(numbers)
    max=max_RDD.max()
    sc.stop()
    return max


def test1():
    assert (max_array([1,2,3,4,5,-10,10,2])) == 10

def test2():
    assert (max_array([100,2,3,4,5,-10,10,2])) == 100

def test3():
    assert (max_array([1,1,1,1,1,1,1])) == 1

def test4():
    assert (max_array([1,2,3,4,5,-10,10,2])) != -10
```

Figure 4.3. Generated PySpark code for Maximum Number by Tyro

is similar to the Max function but Tyro identifies 2 operations inside the loop - *sum* and *count*. Similar to the Max() operation, the Map transformation for average also fails and moves to the Reduce transformation. Figure 4.5 shows the output code after successful verification.

However, after manual inspection of the generated code, we can deduce a few key points. Tyro converts the same input dataset into two separate RDDs rather using the initial RDD for both *sum* and *count*. In the Figure 4.5 on *Line 7* and *Line 9*, Tyro is repeating the same conversion instead of using the already converted RDD. This optimization is future work for this project discussed in Section 5.1.

Another key deduction from this experiment, Tyro did not use the builtin function called *mean*. During pattern matching, Tyro doesn't replace one operation for two different python operations. Instead of going for *mean*, Tyro matched with *count* and *sum* as each individual operation. The optimization of such operations would be an improvement as Tyro can detect two operations working as a single unit operation. However, both programs: average with sum and count operation and average with mean operation when tested with a synthetic dataset of size 8 GB did not have a significant performance difference. This means optimization of such two operation into single operation would be a plus point but may not have a performance gain. Extending search space to detect such operation is the future work of Tyro (Section 5.1.4).

```
1  def avg(numbers):
2      a = 0
3      sum_all = 0
4      for n in numbers:
5          a = a + 1
6          sum_all = n + sum_all
7      average = sum_all/a
8      return average
```

Figure 4.4. Python program to find the average of number from a list

```
1   import pyspark as ps
2
3   def avg(numbers):
4       a = 0
5       sum_all = 0
6       sc = ps.SparkContext()
7       a_RDD = sc.parallelize(numbers)
8       a=a_RDD.count()
9       sum_all_RDD = sc.parallelize(numbers)
10      sum_all=sum_all_RDD.sum()
11      sc.stop()
12      average = sum_all/a
13      return average
14
```

Figure 4.5. Generated PySpark code of the Python program in Figure 4.4

4.3 Multidimensional Dataset

This test suite was used to see if Tyro can handle multi dimensional data. The kNN (k-Nearest Neighbour) algorithm is a simple, easy-to-implement supervised machine learning algorithm that can be used to solve both classification and regression problems. The kNN algorithm can benefit hugely from executing in a parallel environment (Maillo, Ramírez, Triguero, and Herrera 2017; Anchalia and Roy 2014). The kNN was successfully translated by Tyro. Figure 4.6 is kNN algorithm implemented in Python without classification or regression.

Figure 4.7 shows the translated code from Tyro.The kNN dataset in Figure 4.6 is a multidimensional dataset. The input program is using a user defined function to calculate the distance. The experiment shows that Tyro can handle both multidimensional data and user defined functions. The parsed AST results in only one loop fragment to parallelize. Tyro starts with the UDF transformation and moves to the Map transformation. After successful verification in the Map transformation, the synthesis stops. Figure 4.7 is the output generated by Tyro. Again, after manual inspection of the generated code, the function *sort* on Line *17* in Figure 4.6 can also be translated into PySpark operation. It can be transformed into PySpark operation

```
from math import sqrt
# calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
    distance = (row1[0] - row2[0]) ** 2 + (row1[1] - row2[1])**2
    return sqrt(distance)

# Get distance of points in the dataset
def get_neighbors(train, test_row):
    distances = list()
    for train_row in enumerate(train):
        dist = euclidean_distance(test_row, train_row)
        distances.append((dist,train_row))
    return distances
# Return Only K nearest neighbours, no any classification or regression
def KNN(data, query, k):
    data_list = get_neighbors(data,query)
    sortedData = sorted(data_list)
    return  sortedData[:k]


dataset = [[2.7, 2.5],
           [1.4, 2.3],
           [3.3, 4.4],
           [1.3, 1.8],
           [3.0, 3.0],
           [7.6, 2.7],
           [5.3, 2.08],
           [6.9, 1.7],
           [8.6, -0.2],
           [7.6, 3.5]]
```

Figure 4.6. kNN Algorithm implemented in Python

sortByKey(). Currently, such builtin Python functions are not translated by Tyro which is discussed in Section 5.1.1.

4.4 User Defined Function

In general, translating User Defined functions is the goal of program synthesis. In the Section 4.3, we have already shown a successful translation of UDF by Tyro. The input kNN program uses a user defined function *euclidean distance* to calculate distance between data points.

```
from math import sqrt
import pyspark as ps

# calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
    distance = (row1[0] - row2[0]) ** 2 + (row1[1] - row2[1])**2
    return sqrt(distance)

# Get distance of points in the dataset
def get_neighbors(train, test_row):
    distances = []
    sc = ps.SparkContext()
    train_RDD = sc.parallelize(train)
    distances = train_RDD.map(lambda train_row: (euclidean_distance(train_row,test_row),train_row)).collect()
    sc.stop()
    return distances

# Return Only K nearest neighbours, no any classification or regression done right now
def KNN(data, query, k):
    data_list = get_neighbors(data, query)
    sortedData = sorted(data_list)
    return sortedData[:k]

```

Figure 4.7. Generated PySpark KNN Code

Figure 4.8 is a complex arithmetic operation and UDF as an example input for Tyro. This program is actually finding the average of a list of numbers. Unfortunately, this translation by Tyro was unsuccessful.

```
def check(num, i, n):
    return ((num * i) + n) / (i + 1)


def avg(numbers):
    num = 0
    b = list(range(len(numbers)))
    for i in b:
        num = check(num, i, numbers[i])
    return num

```

Figure 4.8. Python program to find average using UDF

```
1   import pyspark as ps
2   def udf(accum, num):
3       return((accum[0] * num[1]) + num[0]) / (num[1] +1)
4
5   def avg(numbers):
6       num = 0
7       b = list(range(len(numbers)))
8       sc = ps.SparkContext(numbers)
9       num_RDD_0 = sc.parallelize(b)
10      num_RDD_1 = sc.parallelize(numbers)
11      num_RDD_2 = num_RDD_1.zip(num_RDD_0)
12      num = num_RDD_2.map(lambda x: (x[0],x[1])).reduce(lambda a, num: udf(a,num))
13      sc.stop()
14      return num
```

Figure 4.9. Final Generated PySpark Code in Reduce Stage

In the Figure 4.9 on Line *3* Tyro converts the complex arithmetic operation and adds the accumulator. The accumulator is only updated by the driver program as the value in the function doesn't gives the correct answer, resulting in a wrong result. This makes the program fail the verification process. Presently, Tyro can only use operation translation so it does not recognize the function is only calculating the average. The search space for program synthesis must be modified in order to give correct result in this case. Extending the search space for equivalent programs is future work discussed in Section 5.1.4.

Table 4.1 provides a quick summary of the Tyro's translations along with the number of loop fragments in the input. The optimization column denotes whether or not the future work intends to improve the result. The optimization might be using builtin functions, detecting more loop fragments or better operation translation.

4.5 Speed Up

For testing performance between the sequential and the parallel programs, I conducted experiments on an AWS cluster of 9 m5a.xlarge instances (1 master node, 8 core nodes), where each node contains an Intel Xeon 2.5 Ghz processor with 4 vCPUs, 16GB of memory, and 64 GB of SSD storage.

Table 4.1. Summary of Tyro's Translation

Test Suite	**Program Name**	**# of Loop Fragments**	**Optimization Required?**
Aggregate Function	Sum	1	No
	Count	1	No
	Min and Max	2	No
	Average	1	Yes
Nested Loop	Join - No Operations	2	No
	Join - With Operation	2	No
Multidimensional	KNN	2 (one only translated)	Yes
	Sum of all	1	No
User Defined Function	Conditional Sum	1	No
	Conditional Count	1	No
	User Defined Average	1 (Not translated)	Yes

Creating parallel programs from sequential ones is only valuable if the performance of parallel programs is significantly better. The translated PySpark code along with the original Python were executed on a synthetic dataset of 8 GB. These datasets were divided into 8 different files (1 GB each). The file was divided in 8 parts because the sequential program couldn't handle such huge file at once in the master node. For PySpark, the files were stored in Hadoop File System (HDFS) with replication factor 3. For Python, the files were stored in the master node .

Currently, Tyro does not support files so both the sequential and the generated PySpark programs were modified in order to handle files. In the case of sequential program, a function was added that converts the file into list of numbers and passed to the test function. The time required to convert these files to list were not recorded. Using the Python's Time library, only the time required for the test function was recorded.

On the other hand, a function that converts files into a RDD was added into the generated parallel program. The RDD was then collected as list and passed over to the testing function. Again, the time was not recorded for these additional changes. In order make the environment and hardware configuration similar, the sequential program was executed in the master node of the cluster.

Overall, the PySpark implementation generated by Tyro had a mean speed up of 6.2X compare to their sequential counterparts. Table 4.2 shows the mean and max speedup observed for different test suites. These results shows that Tyro can effectively improve the performance of these sequential programs by retargeting the critical code fragments for executing in a disturbed environment in this case PySpark.

Table 4.2. Translated Code Fragments and their mean and max speedups compared to the sequential implementations

Translated Codes	Mean Speedup	Max Speedup
Sum	8.5x	9x
Count	7.4x	7.8x
Max	5.7x	6x
Min	5.7x	6x
Average	4.6x	5
Conditional Sum	6.1x	6.7x
Conditional Count	5.9x	6.3x

CHAPTER FIVE

Future Work and Conclusion

5.1 Future Work

Tyro as the name suggests, is just a beginning in the exploration of automatic parallel programming. While successfully translating different Python programs to PySpark programs, there are limitations in Tyro that provide opportunities for improvement.

5.1.1 Increased Loop Detection

Currently, Tyro only detects For loops. It is a straight forward process to parallelize all kind of loops. Additionally, Tyro needs to check for the builtin functions like sort or sum, which are loops in disguise. These builtin functions can be translated by creating a catalog where all of them are stored. Other Python modules can also be added to this catalog using annotation or configuration files.

5.1.2 Complex Data Types and File Handling

Tyro presently handles a list of numbers as an input datasets. However, it can be extended to handle complex data types and files. In order to handle files, Tyro can use the method *textFile* in PySpark which converts a file into RDDs instead of using the method *parallelize*. Handling other data types is straight forward if the input datasets is uniform i.e the input dataset has one data type.

However, if the data types are non-uniform, Tyro must be able to use the additional feature of PySpark like DataFrames. DataFrames is a distributed collections in PySpark that acts like a SQL table. It can be manipulated using the various domain-specific-language (DSL) functions. The program synthesis must be changed in order to handle such functions, so further research is required.

5.1.3 Optimized Nested Loop Handling

Tyro can handle nested loop but with a lot of assumptions. The assumption of no operations inside outer loop can be omitted if Tyro treats each loop separately i.e., the outer loop is handled like a regular loop and the inner loop is handled as a nested loop. This modification will be able to handle any operation inside the outer loop of a nested loop.

Another assumption is that Tyro only checks for nested loops of size two. However, if we can expand the search space to go deeper in the nested loop, Tyro can search for multi level nested loops and handle them as explained above. The search space of Tyro is extended in this way.

5.1.4 Extending The Search Space

Instead of using a few PySpark operations, Tyro can extend its search space by adding multiple operations such FlatMap, GroupBy, and so on. Another addition that Tyro can make is checking for aggregate functions without any program synthesis i.e., test all the aggregate functions before starting any stage in the gradual synthesis. Such additions will make the search space larger than the present one. Likewise, the failed test discussed in Section 4.4 would have a successful translation after extending the search space.

5.1.5 Verification

Tyro uses the unit testing as a verification tool. The unit tests currently are sufficient for verification. However, Tyro can extend its verification process by using verifiers like Sketch (Solar-Lezama 2008) and Dafny (Leino 2010). These verifiers do not required test cases but rather a program specification. The specification can be generated using the input program.

5.1.6 *Beyond MapReduce*

Tyro currently translates Python operations into PySpark operations which are still a based on MapReduce. However, Tyro can go beyond this framework by exploring different parallel frameworks like Parsl and Dask. These architectures allow different features compared to existing MapReduce frameworks. For example, Dask allows parallel computations on single machines by leveraging their multi-core CPUs and streaming data efficiently between the cores. Not only that, it efficiently transfers to a distributed cluster with ease. However, writing Dask programs is challenging because it has its own data structures, interfaces and dataframes which are complex to understand and implement. Thus, generating Dask programs from sequential program would allow greater parallelism.

5.1.7 *Partitionable Functions*

After running different experiments, not all functions can be execute in parallel environment. The functions like sum and count can easily work in sequential and parallel world. Non-determinism is what makes a function unfit to translate to parallel programs. A non-determinism might be working on a shared mutable variable, which might return invalid or incorrect results.

Tyro can be applied to specific functions, called *partitionable* in (Sanjel and Speegle 2020). Let $f(X, Y) = Z$ be an iterating function over X, such that each $x \in X$ is distinguishable, Y be a dictionary of parameters appropriate to f and let Z be a set of data items such that each $z \in Z$ is distinguishable. Two distinguishable elements may have the same value, for example, two integers with the value 7, but are distinguished via an external mechanism, such as the location in a file or the index of an array.

Let $\hat{X}$ be a partition of X, meaning that every $\hat{x} \in \hat{X}$ is a subset of X, is non-empty and pair-wise disjoint with every other element in $\hat{X}$. Furthermore, $\bigcup_{\hat{x}} = X$. If $\hat{x}$ is in the domain of f, then $f(\hat{x}, Y) = z$. Assuming all partitions are in the

domain of f, denote the result of applying f to each element in the partition $\hat{X}$ as $f(\hat{X}, Y) = \bigcup_z = \hat{Z}$.

A function $f(X, Y)$ is *partitionable* if there exists a function g such that $g(f(\hat{X}, Y), Y) = g(\hat{Z}, Y) \equiv f(X, Y)$.

We further define two different partitionable function called Identity Partitionable (IP) and Self Partitionable (SP).

- **IP Functions** are the class of partitionable functions applied in the map phase of map-reduce programs. In general, a function is an IP function if the state is the same whenever data item x is processed
- **SP Functions** are the class of partitionable functions in which f and g are the same function. For example, some functions related to relational database operations are known to be SP functions. Consider the aggregate operators SUM, MIN and MAX. For each of these functions, f serves the role of g, so $f(f(\hat{X}, Y), Y) = f(\hat{Z}, Y) \equiv f(X, Y)$. While Tyro currently handles for-loops implementing these aggregates, if a function can be discovered to be self-partitionable, then Tyro could generate either a map or reduce operation for that function.

5.2 Conclusion

Tyro is a new compiler that identifies and converts sequential Python code fragments into PySpark code. With the help of pattern matching, gradual synthesis and operation translation, Tyro generates parallel code equivalent to the original sequential program. Tyro parses the sequential input program into an AST, then extracts operations from the tree and finally translates the extracted operation into a PySpark operation. Tyro uses gradual synthesis to grow the solution space. At each stage of gradual synthesis, a unit test is performed with the given test cases. Tyro uses a verifier that executes these test cases against the generated code. If all tests pass, then the verification is completed and the current generated code is sent

as an output. After five expansions of the solution space, if Tyro cannot generate equivalent code, then it exits.

Tyro's operation translation is evaluated by running the experiments as shown in Table 4.1. Specifically, we determined whether: a) Tyro can detect loop fragments and b) convert these detected loop fragment into PySpark operations. From the test suites and different test cases, Tyro was able to translate 13 of 15 loop fragments. Out of 13 translations, 3 of them were map operations, 10 of them were reduce operations. The two fragments that Tyro couldn't translate was because: 1) Tyro could not detect a fragment where parallelization was possible and 2) the accumulator in PySpark cannot be manipulated by a worker node which made the generated program to produce incorrect result.The generated PySpark code preforms up to 9x faster compared to the original sequential programs in a 9 node cluster.

APPENDIX

APPENDIX A

Additional Results

A.1 Count

```python
## For AST analysis for ParallelPy
def count_array(numbers):
    cnt = 0
    for n in numbers:
        cnt = cnt + 1
    return cnt

def test1():
    cnt  = 1
    numbers  = [1]
    assert  count_array(numbers) == cnt

def test2():
    cnt  = 10
    numbers  = [1,2,3,4,5,6,7,8,9,10]
    assert  count_array(numbers) == cnt

def test3():
    cnt  = 6
    numbers  = [10,10,10,-10,9,1]
    assert  count_array(numbers) == cnt

def test4():
    cnt  = -1
    numbers  = [1,2,3,4]
    assert  count_array(numbers) != cnt
```

(a) Input Sequential Code

```python
import pyspark as ps
## For AST analysis for ParallelPy

def count_array(numbers):
    cnt = 0
    sc = ps.SparkContext()
    cnt_RDD = sc.parallelize(numbers)
    cnt=cnt_RDD.count()
    sc.stop()
    return cnt

def test1():
    cnt  = 1
    numbers  = [1]
    assert  count_array(numbers) == cnt

def test2():
    cnt  = 10
    numbers  = [1,2,3,4,5,6,7,8,9,10]
    assert  count_array(numbers) == cnt

def test3():
    cnt  = 6
    numbers  = [10,10,10,-10,9,1]
    assert  count_array(numbers) == cnt

def test4():
    cnt  = -1
    numbers  = [1,2,3,4]
    assert  count_array(numbers) != cnt
```

(b) Output PySpark Code

Figure A.1. Tyro's translation for Aggregate Function Count

```python

def min_array(numbers):
    min = float('inf')
    for n in numbers:
        if min > n:
            min = n
    return min


def test1():
    assert (min_array([1, 2, 3, 4, 5, -10, 10, 2])) == -10

def test2():
    assert (min_array([100, 2, 3, 4, 5, -100, 10, 2])) == -100

def test3():
    assert (min_array([1, 1, 1, 1, 1, 1, 1])) == 1

def test4():
    assert (min_array([1, 2, 3, 4, 5, -10, 10, 2])) != 10

```

(a) Input Sequential Code

```python
import pyspark as ps
def min_array(numbers):
    min = float('inf')
    sc = ps.SparkContext()
    min_RDD = sc.parallelize(numbers)
    min = min_RDD.min()
    sc.stop()
    return min


def test1():
    assert (min_array([1, 2, 3, 4, 5, -10, 10, 2])) == -10

def test2():
    assert (min_array([100, 2, 3, 4, 5, -100, 10, 2])) == -100

def test3():
    assert (min_array([1, 1, 1, 1, 1, 1, 1])) == 1

def test4():
    assert (min_array([1, 2, 3, 4, 5, -10, 10, 2])) != 10

```

(b) Output PySpark Code

Figure A.2. Tyro's translation for Aggregate Function Min

A.3 Join with no operation

```python
def alter_values(data_1, data_2):
    data_3 = []
    for i in data_1:
        for j in data_2:
            if i[0] == j[0]:
                data_3.append((i[0] , i[1],j[1])) # key, tuple
    return data_3


def test1():
    data_1 = [('a', 100), ('b', 3), ('c', 2), ('d', -203)]
    data_2 = [('a', 1), ('b', 103), ('c', 2), ('d', 3)]
    expected = ('a',100,1)
    result = alter_values(data_1,data_2)
    assert result[0] == expected

def test2():
    data_1 = [('a', 100), ('b', 3), ('c', 2), ('d', -203)]
    data_2 = [('a', 1), ('b', 103), ('c', 2), ('d', 3)]
    expected = ('a',101)
    result = alter_values(data_1,data_2)
    assert result[1] != expected

def test3():
    data_1 = [('a', 100), ('b', 3), ('c', 2), ('d', -203)]
    data_2 = [('a', 1), ('b', 103), ('c', 2), ('d', 3)]
    expected = ('c',2,2)
    result = alter_values(data_1,data_2)
    assert result[2] == expected

```

(a) Input Sequential Code

```python
import pyspark as ps
def alter_values(data_1, data_2):
    sc = ps.SparkContext()
    data_1_RDD = sc.parallelize(data_1)
    data_2_RDD = sc.parallelize(data_2)
    data_1_RDD_combine =data_1_RDD.join(data_2_RDD)
    return data_1_RDD_combine.collect()


def test1():
    data_1 = [('a', 100), ('b', 3), ('c', 2), ('d', -203)]
    data_2 = [('a', 1), ('b', 103), ('c', 2), ('d', 3)]
    expected = ('a',100,1)
    result = alter_values(data_1,data_2)
    assert result[0] == expected

def test2():
    data_1 = [('a', 100), ('b', 3), ('c', 2), ('d', -203)]
    data_2 = [('a', 1), ('b', 103), ('c', 2), ('d', 3)]
    expected = ('a',101)
    result = alter_values(data_1,data_2)
    assert result[1] != expected

def test3():
    data_1 = [('a', 100), ('b', 3), ('c', 2), ('d', -203)]
    data_2 = [('a', 1), ('b', 103), ('c', 2), ('d', 3)]
    expected = ('c',2,2)
    result = alter_values(data_1,data_2)
    assert result[2] == expected

```

(b) Output PySpark Code

Figure A.3. Tyro's translation for Join with no operation

```python
def alter_values(data_1, data_2):
    data_3 = []
    for i in data_1:
        for j in data_2:
            if i[0] == j[0]:
                sum_1 = i[1] + j[1]
                data_3.append((i[0] , sum_1)) # key, tuple
    return data_3

def test1():
    data_1 = [('a', 100), ('b', 3), ('c', 2), ('d', -203)]
    data_2 = [('a', 1), ('b', 103), ('c', 2), ('d', 3)]
    expected = ('a',101)
    result = alter_values(data_1,data_2)
    assert result[0] == expected

def test2():
    data_1 = [('a', 100), ('b', 3), ('c', 2), ('d', -203)]
    data_2 = [('a', 1), ('b', 103), ('c', 2), ('d', 3)]
    expected = ('a',101)
    result = alter_values(data_1,data_2)
    assert result[1] != expected

def test3():
    data_1 = [('a', 100), ('b', 3), ('c', 2), ('d', -203)]
    data_2 = [('a', 1), ('b', 103), ('c', 2), ('d', 3)]
    expected = ('c',4)
    result = alter_values(data_1,data_2)
    assert result[2] == expected

```

(a) Input Sequential Code

```python
import pyspark as ps
def alter_values(data_1, data_2):
    sc = ps.SparkContext()
    data_1_RDD = sc.parallelize(data_1)
    data_2_RDD = sc.parallelize(data_2)
    data_1_RDD_combine =data_1_RDD.join(data_2_RDD)
    data_1_RDD_combine =data_1_RDD_combine.map(lambda x: (x[0],x[1][0]+x[1][1])).collect()
    sc.stop()
    return data_1_RDD_combine

def test1():
    data_1 = [('a', 100), ('b', 3), ('c', 2), ('d', -203)]
    data_2 = [('a', 1), ('b', 103), ('c', 2), ('d', 3)]
    expected = ('a',101)
    result = alter_values(data_1,data_2)
    assert result[0] == expected

def test2():
    data_1 = [('a', 100), ('b', 3), ('c', 2), ('d', -203)]
    data_2 = [('a', 1), ('b', 103), ('c', 2), ('d', 3)]
    expected = ('a',101)
    result = alter_values(data_1,data_2)
    assert result[1] != expected

def test3():
    data_1 = [('a', 100), ('b', 3), ('c', 2), ('d', -203)]
    data_2 = [('a', 1), ('b', 103), ('c', 2), ('d', 3)]
    expected = ('c',4)
    result = alter_values(data_1,data_2)
    assert result[2] == expected

```

(b) Output PySpark Code

Figure A.4. Tyro's translation for Join with operation inside the loop body

BIBLIOGRAPHY

Ahmad, M. B. S. and A. Cheung (2018). Automatically leveraging mapreduce frameworks for data-intensive applications. SIGMOD '18, New York, NY, USA, pp. 1205–1220. Association for Computing Machinery.

Aho, A. V., M. S. Lamb, R. Sethi, and J. D. Ullman (2007). *Compilers, principles, techniques, tools*. Pearson Addison Wesley.

Akil, B., Y. Zhou, and U. Röhm (2017). On the usability of hadoop mapreduce, apache spark apache flink for data science. In *2017 IEEE International Conference on Big Data (Big Data)*, pp. 303–310.

Alexandrov, A., A. Katsifodimos, G. Krastev, and V. Markl (2016, June). Implicit parallelism through deep language embedding. *SIGMOD Rec. 45*(1), 51–58.

Anchalia, P. P. and K. Roy (2014). The k-nearest neighbor algorithm using mapreduce paradigm. In *2014 5th International Conference on Intelligent Systems, Modelling and Simulation*, pp. 513–518.

Babuji, Y., A. Woodard, Z. Li, D. S. Katz, B. Clifford, I. Foster, M. Wilde, and K. Chard (2019). Scalable parallel programming in python with parsl. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, PEARC '19, pp. 22:1–22:8. ACM. babuji19scalable.pdf.

Bodik, R. and B. Jobstmann (2013, October). Algorithmic program synthesis: Introduction. *Int. J. Softw. Tools Technol. Transf. 15*(5–6), 397–411.

Boyer, R. S. and J. S. Moore (1984). A mechanical proof of the unsolvability of the halting problem. *Journal of the ACM (JACM) 31*(3), 441–458.

Cheung, A., A. Solar-Lezama, and S. Madden (2012). Inferring sql queries using program synthesis. *arXiv preprint arXiv:1208.2013*.

Culler, D., J. P. Singh, and A. Gupta (1999). *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing.

da Silva Morais, T. (2015). Survey on frameworks for distributed computing: Hadoop, spark and storm. In *Proceedings of the 10th Doctoral Symposium in Informatics Engineering-DSIE*, Volume 15.

Dasso, A. (2006). *Verification, validation and testing in software engineering*. IGI Global.

Dean, J. and S. Ghemawat (2004). Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, pp. 137–150.

Dobre, C. and F. Xhafa (2014). Parallel programming paradigms and frameworks in big data era. *International Journal of Parallel Programming 42*(5), 710–738.

Fedyukovich, G., M. B. S. Ahmad, and R. Bodik (2017, June). Gradual synthesis for static parallelization of single-pass array-processing programs. *SIGPLAN Not. 52*(6), 572–585.

Gilles, K. (1974). The semantics of a simple language for parallel programming. *Information processing 74*, 471–475.

Grosch, J. and H. Emmelmann (1990). A tool box for compiler construction. In *International Workshop on Compiler Construction*, pp. 106–116. Springer.

Grune, D., K. Van Reeuwijk, H. E. Bal, C. J. Jacobs, and K. Langendoen (2012). *Modern compiler design*. Springer Science & Business Media.

Gulwani, S., O. Polozov, R. Singh, et al. (2017). Program synthesis. *Foundations and Trends® in Programming Languages 4*(1-2), 1–119.

IEEE (1990). IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, 1–84.

Jiang, L. and Z. Su (2009). Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, New York, NY, USA, pp. 81–92. Association for Computing Machinery.

Katsifodimos, A. and S. Schelter (2016). Apache flink: Stream analytics at scale. In *2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)*, pp. 193–193.

Keller, R. M. (1976, July). Formal verification of parallel programs. *Commun. ACM 19*(7), 371–384.

Ksiazek, K., Z. Marszalek, G. Capizzi, C. Napoli, D. Połapl, and M. Woźniak (2018). The impact of parallel programming on faster image filtering. In *2018 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pp. 545–550.

Leino, K. R. M. (2010). Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pp. 348–370. Springer.

Maillo, J., S. Ramírez, I. Triguero, and F. Herrera (2017). knn-is: An iterative spark-based design of the k-nearest neighbors classifier for big data. *Knowledge-Based Systems 117*, 3–15.

Manna, Z. and R. Waldinger (1980, January). A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst. 2*(1), 90–121.

McKenney, P. E. (2017). Is parallel programming hard, and, if so, what can you do about it? (v2017.01.02a).

Nandi, A. (2015). *Spark for Python Developers*. Packt Publishing.

Nasehi, S. M., J. Sillito, F. Maurer, and C. Burns (2012). What makes a good code example?: A study of programming q a in stackoverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 25–34.

Nayak, K., X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi (2015). Graphsc: Parallel secure computation made easy. In *2015 IEEE Symposium on Security and Privacy*, pp. 377–394.

Nystrom, N., M. R. Clarkson, and A. C. Myers (2003). Polyglot: An extensible compiler framework for java. In *International Conference on Compiler Construction*, pp. 138–152. Springer.

Palsberg, J. and C. B. Jay (1998). The essence of the visitor pattern. In *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac '98) (Cat. No.98CB 36241)*, pp. 9–15.

Pierre-Etienne, M., C. Ringeissen, and M. Vittek (2003). A pattern matching compiler for multiple target languages. In G. Hedin (Ed.), *Compiler Construction*, Berlin, Heidelberg, pp. 61–76. Springer Berlin Heidelberg.

Sanjel, A. and G. Speegle (2020). Tyro: A system for automatically parallelizing partitionable functions. In preparation.

Sarcar, A. and Y. Cheon (2010). A new eclipse-based jml compiler built using ast merging. In *2010 Second World Congress on Software Engineering*, Volume 2, pp. 287–292.

Shvachko, K., H. Kuang, S. Radia, and R. Chansler (2010). The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pp. 1–10. IEEE.

Solar-Lezama, A. (2008). *Program synthesis by sketching*. University of California, Berkeley.

Thusoo, A., J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy (2009). Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment 2*(2), 1626–1629.

Torlak, E. and R. Bodik (2013). Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, New York, NY, USA, pp. 135–152. Association for Computing Machinery.

Van Rossum, G. and F. L. Drake (2009). *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace.

Zaharia, M., M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, et al. (2010). Spark: Cluster computing with working sets. *HotCloud 10*(10-10), 95.