12-16-2020 10:30 AM

Applying Front End Compiler Process to Parse Polynomials in Parallel

Amha W. Tsegaye, *The University of Western Ontario*

Supervisor: Dr. Marc Moreno Maza, *The University of Western Ontario*

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

Recommended Citation

Tsegaye, Amha W., "Applying Front End Compiler Process to Parse Polynomials in Parallel" (2020). *Electronic Thesis and Dissertation Repository*. 7592.
https://ir.lib.uwo.ca/etd/7592

Abstract

Parsing large expressions, in particular large polynomial expressions, is an important task for computer algebra systems. Despite of the apparent simplicity of the problem, its efficient software implementation brings various challenges. Among them is the fact that this is a memory bound application for which a multi-threaded implementation is necessarily limited by the characteristics of the memory organization of supporting hardware.

In this thesis, we design, implement and experiment with a multi-threaded parser for large polynomial expressions. We extract parallelism by splitting the input character string, into meaningful sub-strings that can be parsed concurrently before being merged into a single polynomial. Our implementation targeting multi-core processors is realized with the Basic Polynomial Algebra Subprograms (BPAS). Experimental results show that the approach is promising both in terms of speedup factors and memory consumption.

Keywords: BPAS, Sparse Polynomials, Polynomial Arithmetic, Data Structures, High-Performance, Merge Sort, Parallel Parsing, Polynomial Parsing, Polynomial Multiplication

Lay Summary

Parsing large polynomial datasets is a time-consuming and computationally expensive process. In this thesis, we present a high-performance parallel parser that utilizes the front end compiler phase as its core operation. We will use lexical and semantic analysis programs, flex and bison, to automate the process. We apply data parallelism to concurrently parse split chunks of input sub-string in a multi-core processor. In conclusion, we gained significant speedup with lower memory footprints in parsing large polynomial datasets.

Contents

List of Algorithms

List of Source Code Listing

List of Figures

List of Tables

Chapter 1

Introduction

Parsing large expressions, in particular large polynomial expressions, is an important task for computer algebra systems. While parsing may not be regarded as an algebraic operation, it participates to the process of solving hard algebraic problems. Indeed, solving such problems symbolically often generate large intermediate expressions that one may want to write to a file before reloading them and parsing them.

The problem of parsing large polynomial expressions can become challenging because reading from disks or other storage devices can be significantly slower than producing data at the top of the memory hierarchy, say in registers or in cache memories. Therefore, for hard algebraic problems requiring to parse large intermediate expressions, parsing can consume a substantial amount of computing resources and can even be a bottleneck in some applications.

In the context of modern computer architectures, it is desirable to design multi-threaded algorithms for parsing and implement those algorithms on multi-core processors, which brings a third challenge. Indeed, parsing the algebraic expressions commonly manipulated by a computer algebra system can be done by deterministic context-free grammars (DCFGs) thus in time essentially proportional to the size of the input. Therefore, parsing such expressions is a memory bound application, that is, where the amount of work is proportional to the amount of data which is being read from and written to RAM memory. Such applications are difficult to parallelize on multi-core processors, as speedup factors are limited by the architecture (in particular the numbers of memory controllers and channels, which is typical less than the number of cores).

Another challenge in parsing algebraic expressions is the variety of input format. Focusing on polynomials, the topic of this dissertation, input polynomials may be expanded, that is, formatted as a sum of terms, or factored, that is, "smaller" formatted as a product of polynomials. Since polynomials in a computer algebra system often use canonical representations[1] for computational efficiency, parsing different input formats necessarily require data conversions and algebraic calculations. Popular data-structures for polynomials include:

1. linked lists (where each cell stores one non-zero term or a fixed number of such

[1]A canonical representation is a data-structure so that two polynomials that are mathematically different are encoded differently with that data-structure; in other words mathematical equality between two polynomials can be decided by comparing the data-structures representing those polynomials.

terms; this is an effective representation for expanded sparse polynomials)
2. multidimensional arrays (where each array element contains a coefficient, zero or not, and is indexed by an exponent vector; this is an effective representation for dense polynomials)
3. alternating arrays (where coefficients and exponent vectors of non-zero coefficients are alternated in a compact format to minimize costs related to memory accesses, in particular cache; this is an effective representation for sparse polynomials).

Hence a polynomial parser must dynamically choose an appropriate data-structure, or an appropriate combination of data-structures, in order to optimize memory usage. We note that the computer algebra system Maple and the Basic Polynomial Algebra Subprograms (BPAS), both used in this study, are examples of software where the above data-structures are available and where the implementation of polynomial arithmetic is highly optimized.

1.1 Contribution

Our goal is to design and implement a parser for polynomial expressions, which can take advantage of multi-core processors. It is natural to consider a solution where the input is split into several segments, which are then parsed concurrently, before merging the parsed polynomials into a single polynomial. As mentioned above, since the parsing algorithms that we shall use run in linear time, parallelizing overheads (due to scheduling and synchronization) may offset the benefits of concurrent execution. Moreover, for a data-intensive application like parsing, speedup factors on multicore processors are limited by hardware characteristics like the number of controllers.

In addition, polynomials being structured objects (similarly to well-parenthesized expressions) splitting the input cannot be done without a preliminary pass through the data in order to determine positions where to split. While this preliminary pass could theoretically be done in a parallel fashion, the resulting tasks would too fine-grained (similarly to the parallelization of vector addition) for multicore processors.

Despite of the challenges paused by this approach, this is the path that we have chosen to follow. This thesis dissertation reports on the algorithms that we have designed, implemented and experimented. We have gained speedup factors that can reach 4x for large enough data on a 12-core processor. Because of the hardware limitations discussed above (there are, indeed, three memory channels on the processor that we were using for our experimentation) we believe that this is a promising result. Nevertheless, improvements could still be made in the future, including:

1. parallelizing the algebraic operations involved in the parser, in particular polynomial multiplication,
2. parallelizing the splitting of the input, using a Graphics Processing Unit (GPU) on which the parallelization could bring benefits, in the same way that vector addition can be optimized on such devices.

1.1.1 Details of our approach

As mentioned above, the first step of our approach is to determine how to split the input polynomial into segments that can be parsed concurrently. Since this splitting procedure is executed serially, its code is highly optimized so as to minimize its contribution to the whole application. The segments are organized in a way that they can easily be processed by a multithreaded algorithm. This latter works in a divide and conquer manner. The combine phasis requires arithmetic operations (addition, multiplication). The input polynomial is assumed to be sparse. A representation based on linked lists is used until switching to an alternating array representation is detected to be a better choice. The multithreaded algorithm is implemented with CilkPlus [18].

1.2 Related works

Parsing is an important problem with application in every scientific discipline such as compiler design, natural language processing, parsing markup languages, etc. Algorithms that adopt front end compiler design architecture for solving parsing problems typically rely on a few core techniques aiming at maximizing performance. These core techniques include lexical analysis to prepare the input for parsing, syntactic, and semantic analysis to make sure the input data-set is valid and error-free. These core techniques are provided by software tools like `Flex` and `Bison` [13].

The problem of utilizing a multicore processor during parsing is not new. The Project presented here [15] applies to processing large input files that range up to hundreds of gigabytes. These large input files are broken into chunks before being processed. Those chunks of datasets are stored back as multiple files. Our approach avoids creating files. Moreover, in our case, the input data is structured, making the problem of splitting more difficult.

Another study describes a general-purpose parser generator (PA-PAGENO), which produces an efficient deterministic parallel parser exhibiting significant speedup when parsing large text on modern multi-core machines [3]. These large texts include XML files and real JSON documents. The researchers noted that the generated parser relies on the properties of Floyd's operator precedence grammars to provide a naturally parallel implementation of the parsing process. Hence, parallelism is extracted at the level of the parsing algorithm, not at the level of data. This approach is orthogonal to our approach. Combining them would be an interesting future work.

The study [7] discusses parsing XML-based documents, based on data-parallel processing. The input XML document is partitioned into substrings, and a DFA-based lexical analysis tool runs on the substring concurrently. The authors apply automated analysis tools such as Flex. The paper does not clearly describe the tools used to parallelize the process. Although the research geared towards parsing large XML documents, the steps described correlates with our research. However, this work is restricted to parsing regular languages and no experimentation is reported. Recall that our parser handles context-free grammars.

In a survey of parallel parsing strategies for natural language processing, the problem of parsing large input using a context-free parser in parallel is described. In the discussion, a scheme where more than one traditional parser is used, and this parser is assigned to split on non-deterministic choices during parsing [17]. One scheme where the number of processors used depends on the length of the sentence being parsed and other on the grammar size. Although it is a survey, the idea of a data-parallel approach parsing is described but not implemented.

Chapter 2

Background

In compilers design, parsing is a crucial step in the compilation process when translating one language to a different language. These parsing techniques used in compilers can be applied in computer algebera to parse large polynomials. In computer algebra, the parsing process plays an important role to generate an efficient data structure from large input polynomials. In this background section, we introduce topics necessary to understand the parsing process in compilers and the algorithm used in a parser generator program called Bison.
We will briefly discuss the theory of computation, and grammars and their limitations in Section 2.1.1. Then we will touch upon context-free grammars (CFG) in Section 2.1.2. In Section 2.2.1, we will have an in-depth look at the parse tree and some of its limitations. Then in Section 2.2.4, we will cover the different types of parsers and their corresponding algorithm. Our focus is to understand how the LALR(1) parsing algorithm works. To understand this algorithm, we have to understand all the other parsing algorithms because they are an improved version of one another. Lastly, we will talk about the anatomy of compilers, how we apply the front end compiler technique to implement our parser in Section 2.3 and Section 2.7, respectively.

2.1 Parsing

Parsing is one of the fundamental problems in computer algebra; a parsed input is structured to linear-representation[1] in accordance with a given grammar [11]. Also, it is the problem of taking a string of terminals and figuring out how to drive it from the root symbol of the grammar [1]. During parsing, we can report syntax errors early in the parsing process when the input is not conforming with the grammar structure.

[1]An input polynomial in the form of a character array or a string.

2.1.1 Theory of computation

One of the important questions that can be answered by the theory of computation is the limitation of computers, the ability to compute certain problems concerning speed and memory usage. The theory of computation can help us to develop a mathematical model of computation that reflects the real-world computers to determine computability, i.e., a way to classify problems as solvable or unsolvable [19]. A concept that deals with the definitions and properties of different computational models are called *automata theory*. Some of these models include finite automata and context-free grammar, etc. Finite automata are a model for computers with a limited amount of memory. Finite automata (FA) otherwise called a finite state machine (FSA), can be divided into two broad categories: FSA with output and FSA without output. The following are some concepts to be familiar with before we formally define finite automaton: *symbols* are any characters that are elements of an *alphabet*; $\sum$ is a collection of symbols; and *strings* are a sequence of symbols.

So, a finite automaton is a 5-tuple ($\mathbf{Q}$, $\sum$, σ, $\mathbf{q_0}$, $\mathbf{F}$), where [19]
1. $\mathbf{Q}$ is a finite set called the states,
2. $\sum$ is a finite set called the alphabet; the elements of $\sum$ are called symbols,
3. σ : is the transition function,
4. $\mathbf{q_0} \epsilon \mathbf{Q}$ is the start state, and
5. $\mathbf{F} \subseteq \mathbf{Q}$ is the set of accepted states.

The mathematical theory of finite automata represented using a state diagram. More information regarding state diagrams can be referenced here [19]. Every state in deterministic finite automata (DFA) has a unique state. Given a current state, it is easy to determine the next state of a DFA. In contrast, non-deterministic (NDFA) finite automata are the opposite of DFA. In addition, the DFA transition function, σ, produces $\mathbf{Q} \times \sum \rightarrow \mathbf{Q}$ states. Whereas, NDFA transition function, σ, produces $\mathbf{Q} \times \sum \rightarrow \mathbf{2^Q}$ states. A language can be accepted or rejected by the Finite state machine; any language that is accepted by FSA is called a regular language. In computer science, a language (formal language) L is a finite or infinite set of strings over a finite alphabet $\sum$—symbols or characters.

There are four Machine-Based hierarchy of language classes [19, p.32]:
- Regular languages, which can be accepted by some finite state machines.
- Context-free languages, which can be accepted by some pushdown automaton.
- Decidable languages,
- Semidecidable languages,

One method of describing a regular language is as *regular expression*. Regular expressions can be considered as the algebraic description of a regular language. Regular expressions can be defined by the following rules:
- ϵ and $\emptyset$ regular expressions
- For each $a \in \sum$, a is a regular expression

- If R_1 and R_2 are regular expressions ,then $R_1 \cup R_2$, R_1R_2, and R^* are also regular expressions.

Therefore, if L is a language and we can only say L is regular if and only if there exists a regular expression that describes it [19].

A grammar G is a set of production rules for a string in a language L. A formal grammar is a way to characterize a language L or list which string is in the language L or not. Simply, for a given grammar G, *L(G)* represents the set of all strings generated from grammar G. A grammar can have different categories: ambiguous, and unambiguous (Section 2.2.3); deterministic, and non-deterministic. Ambiguity addresses the actual syntactical structure of the grammar but determinism relates to the properties of the language. In addition to this, grammar can also be categorized based on performance and efficiency. For each language class, Noam Chomsky gave a mathematical model of a grammar which is effective for writing computer languages called the Chomsky hierarchy [19, p.539]. The basis for the Chomsky hierarchy is the amount of memory required and its organization to process the languages at each level.

Chomsky used the terms type 0, type 1, type 2, and type 3 to describe the four levels in his model:

- type 0 (Unrestricted grammars): no memory constraint and accepted by Recursively enumerable language
- type 1 (Context-sensitive grammars): memory limited by the length of the input string
- type 2 (Context-free grammars): unlimited memory but accessible only in a stack
- type 3 (Regular grammars): finite memory

2.1.2 Context-free grammars

In this thesis, we will only elaborate on type 2 grammars or Context-free grammars (CFG). As mentioned in the previous section, grammars describe the hierarchical structure of a language. CFG is more powerful than finite automata or regular expression, it cannot define all the possible languages but it is useful for hierarchical or nested structures. In CFG, the basic idea is to use variables to stand for sets of strings. These variables are defined recursively in terms of one another. In recursive rules or productions, only concatenation is involved, whereas alternative rules for variables allow unions. For example, an if-else statement in *C* programming language can have the following form:

if (expression) statement else statement

The above statement has if and else as keywords. To complete the if-else statement, it is concatenated with an expression surrounded by parenthesis and two statements. To create a production for the above statement, consider the statement below; *stmt* and *expr* are a more general terms or variables for expression and statement, respectively:

$$\textit{stmt} \rightarrow \textit{if (expr) stmt else stmt}$$

In general, a CFG is just like a regular grammar in which every rule must have a left-hand side that is a single nonterminal and a right-hand side that is ϵ or a single terminal or a single terminal followed by a single nonterminal. The only difference is that CFG must have a left-hand side that is a single nonterminal and a right-hand side [19]. If we consider the above if-else production statement, we can see that it is a context-free grammars that produces a string through concatenation, choice, and identification mechanisms to link the name of a nonterminal to the right-hand side of the rule. So, production is for a nonterminal if the nonterminal is the head of the production. A string of terminals are a sequence of zero or more terminals.

To summarize the above paragraph, context-free grammar has four components:

1. A set of terminal symbols, i.e., elementary symbols of the language defined by the grammar.
2. A set of nonterminals that represents a set of strings of terminals.
3. A set of productions where each production consists of a nonterminal called the head or left side of the production; an arrow; and a sequence of terminals and/or nonterminals called the body or right side of the production.
4. One of the nonterminals as the start symbol.

Deterministic context-free grammar

The deterministic context-free grammar (DCFG) is a subclass of the context-free grammars. DCFG have the property that they can be recognised by a deterministic pushdown automata [5, 9]. A pushdown automaton, or PDA, is a finite state machine that has been augmented by a single stack [19]. DCFGs are always unambiguous, and are an important subclass of unambiguous CFGs. DCFG parsers can run in linear time [19].

Context-free grammar notations

There are different styles of notations computer scientists use to represent context-free grammars. Backus-Naur Form (BNF) is the first CFG used to define ALGOL 60 (computer programming language for publishing algorithms). Any grammar expressed in BNF is a context-free grammar. BNF uses angle brackets to enclose non-terminals, semicolons to terminate or derive rules, and vertical bar to also denote the derive rules. For example, Figure 2.1 shows the classic expression grammar in BNF. Another notation used to represent context-free grammar is the Extended Backus-Naur Form (EBNF). EBNF is a collection of extensions to Backus-Naur form, for more reference [11, p.28].

2.2 Parsing Algorithms

2.2.1 Parse tree

In the previous section, we have described the role of a grammar as a precise rule that a language prescribes to generate a well-formed syntactic structure. Grammars offer more

$$\begin{aligned}\langle expr \rangle &: \langle term \rangle \texttt{ + } \langle expr \rangle \\ &\mid \ \langle term \rangle\end{aligned}$$

$$\begin{aligned}\langle term \rangle &: \langle factor \rangle * \langle term \rangle \\ &\mid \ \langle factor \rangle\end{aligned}$$

$$\begin{aligned}\langle factor \rangle &: \texttt{(} \ \langle expr \rangle \ \texttt{)} \\ &\mid \ \langle const \rangle\end{aligned}$$

$$\langle const \rangle : \text{integer}$$

Figure 2.1: Expression grammar in Backus-Naur Form (BNF).

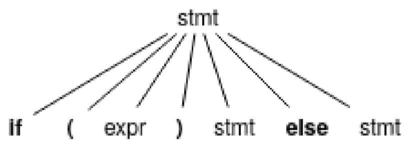


Figure 2.2: A parse tree for the production *stmt* $\rightarrow$ *if (expr) stmt else stmt.*

than rules; they can reveal syntactic ambiguities, design issues, and with the right parsing algorithm can detect syntax errors. To parse a string according to a grammar means to reconstruct the production tree. The production tree shows how the given string can be produced from the given grammar. Constructing a production tree based on Type 2 or context-free grammar has another name known as parser tree. It is important to note that Type 0 and Type 1 grammars are only capable of producing production graphs and their consequent parsing yields parsing graphs.

In this section we will introduce a parse tree. A parse tree determines the specific semantic attached to a specific rule [11, p.62]. A parse tree pictorially shows how the start symbol of a grammar derives a string in the language. If the nonterminal *stmt* has a production *stmt* $\rightarrow$ *if (expr) stmt else stmt*, then a parse tree may have an interior node labeled *stmt* with seven children labeled as *if*, *(*, *expr*, *)*, *stmt*, *else*, and *stmt*, from left to right as shown in Figure 2.2. So, in a CFG, a parse tree holds the following properties: the root is labeled by the start symbol, each leaf is labeled by a terminal or ϵ, and each interior node is labeled by a nonterminal.

Size of a parse tree

The size of a parse tree is directly proportional to the number of tokens generated. For an input string of n tokens (Section 2.2), a parse tree consists of n nodes belonging to a terminal, plus several nodes belonging to non-terminals. Let C_G be a constant that depends on the grammar, provided the grammar has no loops. Therefore, there cannot

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow id$$

Figure 2.3: A simple ambiguous grammar rule.

be more than $C_G n$ nodes belonging to non-terminals in a parse tree with n token nodes. This means that the size of any parse tree is linear in the length of the input [11, p.62].

2.2.2 Determinism

When an action is automatically performed in response to a stimuli it is an automaton. A non-deterministic automaton has several possible moves and the particular choice is not predetermined. By restricting the number of possible moves of a non-deterministic parsing automaton, we can achieve determinism. Since deterministic automaton involves no choice, the moves are determined unambiguously by an input stream. So, if a grammar is an unambiguous deterministic automaton, it can produce only one parsing tree. A deterministic automaton control mechanism is lookaheads. A deterministic automaton with a k lookahead match predicts and decides unambiguously what to do next [11].

2.2.3 Ambiguity

A grammar G is ambiguous if there is at least one string in $L(G)$ for which, G produces more than one parse tree. To show that a grammar is ambiguous, all we need to do is find a terminal string that yields more than one parse tree. Such a string with more than one parse tree usually has more than one meaning. Therefore, we need to design unambiguous grammars. Otherwise, we need to add an additional rules to resolve the ambiguities. On the other hand, if a language is regular, there is no reason to be concerned with ambiguity; a regular language does not care about assign internal structure to a string. In a context-free language, assigning internal structure to a string is crucial. Given a string w, if we want to assign meaning to w, it has to be unique. It is impossible to assign unique meaning to w if the parse tree is not unique. So, an ambiguous grammar which fails to produce a unique parse tree is a problem to a context-free language. The solution to this problem is to convert ambiguous grammar to unambiguous grammar, also known as disambiguity rule. Converting ambiguous grammar to unambiguous grammar can be done by applying recursion—a variable calling itself. There are two types of recursions: left recursion and right recursion. The question arises when and how to apply these recursions when parsing an ambiguous grammar. For example, we would like to generate a parse tree for the string $id + id * id$ using the grammar rule in Figure 2.3. This ambiguous grammar rule is capable of generating two different parse trees for the same input strings. To turn this grammar to unambiguous grammar, we need to use recursion. A grammar is left recursive if the leftmost symbol of the right-hand side (RHS) in the production is the same as the left-hand side (LHS). Since our input has two different operators, we need to consider operator precedence when converting our grammar to a left recursive unambiguous grammar. The grammar in Figure 2.4 is a left

$$\begin{aligned} E &\rightarrow E + G \mid \epsilon \\ G &\rightarrow G * H \mid \epsilon \\ H &\rightarrow id \end{aligned}$$

Figure 2.4: A simple unambiguous grammar rule.

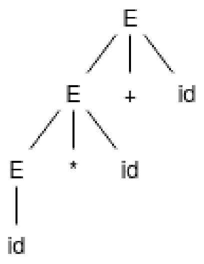


Figure 2.5: Restricting parser tree growth using associativity.

recursive grammar that obeys operator precedence. Recursion restricts the parse tree to grow on one side only. Also, to define precedence in the grammar we used levels within productions, so productions with the highest precedence have the least level (farther away from the start symbol) as shown in Figure 2.5. Similarly, a grammar is right recursive, if the right-most symbol of the right-hand side (RHS) in production is the same as the left-hand side (LHS).

Left factoring

Most parsers, especially top-down parser (Section 2.2.4) do not work properly with left recursive and non-deterministic grammars. This is because most left recursive parsers could get trapped into a never-ending loop by recursively calling the rightmost symbol of the RHS before checking other productions [1]. Consider the following left recursive grammar $A \rightarrow A\alpha/\beta$, the language generated by this grammar is $\beta\alpha*$. We see that A is calling itself recursively without doing anything, as shown in Figure 2.6. One would think converting left recursive grammar to right recursive grammar might resolve the issue. But it is not always guaranteed, it might even generate a different grammar. The solution is to use a grammar transformation process called left refactoring [1, p.214]. If the language $A \rightarrow \beta\alpha*$ has to be generated, we have to use the right recursive equivalent grammar of $A \rightarrow A\alpha/\beta$ as shown in Figure 2.7. It is a general left recursive rule that can be applied to any production. So, left factoring produces a right recursive equivalent grammar without

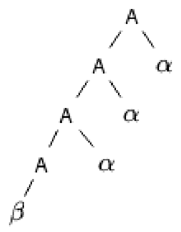


Figure 2.6: Left restricted parse tree calling A with a never ending loop.

$$A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \epsilon$$

Figure 2.7: A left factored grammar of an original left recursive grammar.

$$E \rightarrow E' \\ E' \rightarrow \epsilon \mid +GE' \\ G \rightarrow G' \\ G' \rightarrow \epsilon \mid *HG' \\ H \rightarrow id$$

Figure 2.8: A simple example of left factored grammar rule.

altering the original grammar. Figure 2.8 shows right recursive equivalent grammar for the grammar in Figure 2.4. We can also apply left factoring to convert non-deterministic grammars to deterministic grammars in the same manner.

2.2.4 Types of parsers

There are two most commonly used general types of parsers, especially in compiler design: top-down and bottom-up parsers. Both methods scan the input from left to right; top-down parser build the parse tree from the top (root) to the bottom (leaves), i.e., it imitates the original production process by rederiving the input from the start symbol [1]. Bottom-up parser start from the leaves and work its way up to the root, i.e., it rolls back the production process to reduce the input back to the start symbol [1]. There are different categories of top-down and bottom-up parsers, but in this thesis we will only elaborate on deterministic parsers; detailed information about the other methods can be found in [11].

FIRST and FOLLOW functions

When constructing both top-down and bottom-up parsers two functions are used to construct the table, FIRST and FOLLOW. Given an input symbol, these functions allow us to choose which production to apply. FIRST is a function given a sequence of grammar symbols, returns the set of symbols with which the strings derived from that sequence can begin. Consider the symbol c and $FIRST(\alpha)$ returned a production, the production could be $\alpha \rightarrow c\beta/\epsilon$. In a similar way, $FOLLOW(B)$ returns the production $A \rightarrow cB/\epsilon$ if c may follow B at some point in a derivation. To handle end-of-string conditions in the FOLLOW function, we add an extra production to a grammar called augmented production. For $A \rightarrow cB/\epsilon$, the augmented production is $A' \rightarrow A\$/\epsilon$; therefore, A' is a nonterminal that replaces A as start symbol and $ is a terminal symbol representing the end of input [1, 220].

Deterministic top-down parser

Left-to-right leftmost production (LL) parser is the only deterministic top-down parsing method. This top-down parser constructs a parse tree for an input string starting from the root by creating the nodes of the parse tree in preorder or depth-first order. It can also be viewed as finding the leftmost derivation for an input string. At each step of an LL parsing, the key problem is to determine the production to apply for a nonterminal. Once a production is chosen, the rest of the parsing process consists of matching the terminal symbols in the production body with the input string. LL parser is also called LL(k) parser because it is one of the parser that can perform predictive parsing with k symbols lookahead in an input. In this thesis we will only elaborate on k = *1*: LL(1) parser.

Both deterministic top-down and bottom-up parsers have four general components: the input stream, prediction stack (stack memory), parsing algorithm, and transition tables. The input stream is the actual input to be parsed and the prediction stack is the data structure used by the parsing algorithm to stack input tokens. The transition table is used to create a derivation rule, it is constructed using the parsing grammar.

To generate an LL(*1*) parsing tree, we need four general components as stated in the previous paragraph. One of the crusial component in this list is the transition table; to generate the transition table we first need to build the *FIRST* and *FOLLOW* table from a left factored grammar. For example, consider the following left factored grammar in Figure 2.9; Table 2.1 is the *FIRST* and *FOLLOW* table constructed according to thier definition in the previous section. Once this *FIRST* and *FOLLOW* table is constructed, the next step is to generate the LL(*1*) transition table. Table 2.2 show the transition table constructed using *FIRST* and *FOLLOW* table for the example grammar in Figure 2.9. Using the four components of an LL(*1*) parser, we can create an LL(*1*) parse tree. Depending on what is present in the prediction stack, there are two actions performed by the LL(*1*) parser: prediction and matching. The function *Predict(E,t)* performs prediction if the top of the prediction stack is the non-terminal *E*. In order to do this, the non-terminal E is popped from the top of the prediction stack and a look up in the transition table using the current token reads column E and row t will be performed: *cell[E, t]*. The result will be pushed onto the stack and the process will continue to consume the input string from left to right till the end of the input stream is reached. If the resulting return value from *cell[E, t]* in the transition table is empty, there is a syntax error in the input string.

Similarly, The function *Match(e, t)* performs matching if the top of the prediction stack is the terminal *e*. If the token t matches with the token read from the input e, t will be consumed and the input stream will be advanced by one to the right. Otherwise, if no match is found, there is a syntax error in the input string. When the prediction stack is empty, the *LL(1)* parser will be terminated and the input is accepted if and only if it is completely consumed without any error.

Deterministic bottom-up parser

Bottom-up parsing is the process of reducing an input string w to the start symbol of the grammar [1]. One form of bottom-up parsing is Shift-reduce; here, tokens are processed

$$\begin{gathered} E \rightarrow \textit{TE'} \\ E' \rightarrow +\textit{TE'} \mid \epsilon \\ T \rightarrow \textit{FT'} \\ T' \rightarrow *\textit{FT'} \mid \epsilon \\ T \rightarrow \textit{id}/(E) \end{gathered}$$

Figure 2.9: An example grammar to demonstrate the construction of LL(1) parser.

Table 2.1: *FIRST* and *FOLLOW* table for grammar in Figure 2.9

	FIRST()	**FOLLOW()**
$E \rightarrow TE'$	{id, (}	{\$,)}
$E' \rightarrow +TE'/\epsilon$	$\{+, \epsilon\}$	{\$,)}
$T \rightarrow FT'$	{id, (}	{+,), \$}
$T' \rightarrow *FT'/\epsilon$	$\{*, \epsilon\}$	{+,), \$}
$T \rightarrow id/(E)$	{id, (}	{*, +,), \$}

Table 2.2: LL(1) parse table generated for grammar in Figure 2.9

	id	+	*	(	)	**$**
E	$E \rightarrow TE'$			$E' \rightarrow +TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow /\epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow /\epsilon$	$T' \rightarrow /\epsilon$
T	$T \rightarrow id$			$T \rightarrow (E)$		

from an input stream by pushing them on the stack. When shifting a token by pushing it atop, the stack reduces some sequence of terminals and nonterminals atop the stack back to some nonterminal symbol [1]. Thus, one of the following four actions is carried out during Shift-reduce parsing: shift the next input symbol to the top of the stack; reduce; accept; and error detction [1]. At each reduction step, a matching body of the production is replaced by a nonterminal at the head of that production. Some of the decisions made during bottom-up parsing includes, when to decide to reduce and what production to apply.

Any grammar can be parsed bottom-up using a shift-reduce parser, but the parser might not be deterministic. This could result in guessing whether to apply shift-reduce and if the choice is wrong whether to backtrack the steps. So, the deterministic shift-reduce parser can not parse all grammars no matter how good the grammar is constructed. When a deterministic shift-reduce parser encounters a conflict, it can not parser a grammar or it enters a state where it cannot tell what action to take. There are two types of conflicts, *shift-reduce conflicts* and *reduce-reduce conflicts*. In a *shift-reduce conflict*, the parser cannot tell if it needs to perform a reduction or add the symbol to the stack. In a *reduce-reduce conflict*, the parser knows that it needs to replace the top symbols on the stack with some nonterminal, but it cannot tell what reduction to use.

The left-to-right rightmost (LR) production method is the most powerful type of deterministic bottom-up parsing method that we will elaborate on next. Just like the LL method, the LR method performs predictive parsing; LR(k) parsing uses k for the number of input symbols of lookaheads that are used in making parsing decisions. In this case $k = 0$ and $k = 1$ are of practical interest. LR(k) parsing is more powerful than LL(k) parsing, but it is the most complicated and difficult to understand. There are three classes of LR parser: *LR(0)*; *simple LR(1)* or *SLR(1)*; and *lookahead LR(1)* or *LALR(1).* These different classes of *LR* parser differ on their *action* and *goto* table organization (Section 2.2) and the size of the transition table. LR(0) is theoretically important but too weak to be useful; LR(0) parsers do not use lookahead: this is because of k=*0.* SLR(1) is an upgraded version of LR(0) parser, but weaker than LR(1) parser—a very powerful and memory consuming parser. Unlike LR(1), LALR(1) is both powerful and practically applicable in most bottom-up parsers [12].

Constructing the different classes of *LR(k)* parsers follows the same process as *LL(1)* parser; *LR(k)* parser uses prediction stack (stack memory), parsing algorithm, and transition tables. The difference between *LL(1)* and *LR(k)* parsers is when generating or constructing the transition table. An *LR* parser makes a shift-reduce decision by maintaining the state of the parser position [1, p.242]. To construct the parsing table for LR parsers, we will use an *LR(0)* item called *the canonical LR(0) collection*; both *LR(0)* and Simple *LR(1)* classes of parsers use the canonical *LR(0)* collection to construct their parsing table. For lookahead *LR(1)* parser we will use an *LR(1)* item called *the canonical LR(1) collection* [1, p.243]. An *item* is a production in the grammar with a dot at some position of the production body: $S \rightarrow \textbf{.}AA$ is an item. This dot in the production signifies anything to the right of the dot is not consumed or reached; once the dot is in the far right-hand side, every terminal or nonterminal is consumed and the production is ready to be reduced.

The canonical collection of LR(0) items is the starting point to create deterministic finite automata that are used to make parsing decisions. To briefly describe the major steps involved in constructing canonical collection of *LR(0)* for a grammar, we will use the following example in Figure 2.10. The first step is to redefine the grammar by adding augmented production as shown in Figure 2.11. The augmented production is used to indicate the parser when to stop and accept parsing the input. When constructing the canonical *LR(k)* collection for a grammar, we consider two functions; *CLOSURE* and *GOTO.* Given a grammar and a set of an item, *CLOSURE(I)* will use the following two rules:

- At the beginning we add every item of I to *CLOSURE(I)*
- given a production $A \rightarrow a.B$ is in *CLOSURE(I)*, if $B \rightarrow c$ then $B \rightarrow .c$ will be added to *CLOSURE(I).*

Similarly, for the function *GOTO(I, A)*, where I is the sets of item and A is a grammar symbol, defines the consumption of A in the grammar symbol—calling *GOTO(I, A)* in the production $A \rightarrow a.B$ produces $A \rightarrow aB.$ [1, p.245].

Constructing *LR(0)* parsing table

We will start by constructing the parsing table for an *LR(0)* parser. First, we need to create the goto-graph or transition diagram using the two functions described in the previous paragraph: *CLOSURE* and *GOTO*. Figure 2.12 shows the transition diagram or goto-graph for *LR(0)* parser using the grammar in Figure 2.11. Each node or state of the goto-graph is labeled with $I_{0..n}$. Once the goto-graph is constructed the next step is to create the *LR(0)* parsing table. The parsing table consists of two parts, the results from the *ACTION* function labeled as *action* and the result from the *GOTO* function labeled as *Goto*. Furthermore, the *action* section consists of all the terminals, whereas the *goto* section consists of all the nonterminals. In our example, *a*, *b*, and *$* are terminals that reside in the action sections of the parsing table. Similarly, *A* and *S* are nonterminals that occupy the goto sections of the parsing table. The *ACTION* function takes two arguments, *a state* and *a terminal*; the resulting outcome from the function could trigger either one of the following actions accompanied by a state number: shift, reduce, or accept. If any of the mentioned actions are not triggered, it means an error has occurred; an empty cell without any state can only be reached when there is an error. Each cell in the table with a transition state describes the action and production number or the actual production to transition to. In the case of *SHIFT*, the table cell will contain I_n, where n is the production number. In the case of *REDUCE*, it is described by the word *REDUCE* and a production value. The *GOTO* function, just like in the goto-graph, maps a state and a nonterminal to a new state. The *Goto* column in the table has a value that is neither shift nor reduce, but it provides the production number to reduce to. For example, the grammar in Figure 2.11 labelled each production with numbers. When the parsing table calls for reduce, the number in the subscript should match with the production number.

There are rules to follow when constructing an *LR(0)* parsing table. For the grammar in Figure 2.11, the parsing table is shown in TABLE 2.3. By following the rules below we can construct an LR(0) parsing table for any grammar:

1. Initial state is $A' \to .A$
2. If $A \to aB.$ is in I_i set ACTION(i, a) to reduce to $A \to aB$
3. If $A' \to A.$ is in I_i set ACTION(i, $) to accpet
4. If $A \to a.B$ is in I_i set successor(I_i, a) = I_j and then set ACTION(i, a) to shift
5. Any entries that is not defined by the above rule is an error
6. If a nonterminal in the successor(I_i, A) = I_j then $GOTO(i, A) = i$

Once the parsing table is constructed, we can parse an input string using the *LR(0)* parser by following the same steps introduced during parsing an *LL(0)* parser. The only difference is an *LR(0)* parser uses a different algorithm and parsing table. An LR-parsing program algorithm and a detailed step by step guide on how to use an *LR(0)* parser can be referenced in [1, p.251].

$$S \rightarrow AA$$
$$A \rightarrow aA/b$$

Figure 2.10: An example grammar to demonstrate the construction of LR parser.

$$S' \rightarrow S$$
$$S \rightarrow AA_{1}$$
$$A \rightarrow aA_{2}$$
$$A \rightarrow b_{3}$$

Figure 2.11: An example of augmented grammar to demonstrate the construction of LR(1) parser.

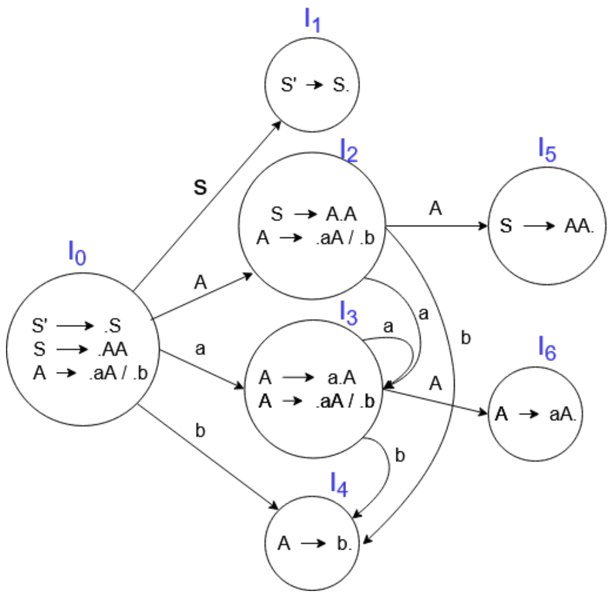


Figure 2.12: A transition diagram or goto-graph for an *LR(0)* parser.

Table 2.3: Parsing table for *LR(0)* parser. Where I_n is transitioning to n state according to the transition diagram in Figure 2.12.

	Action			Goto	
	a	**b**	**$**	**A**	**S**
0	I_3	I_4		I_2	I_1
1			$Accept_{S' \to S}$		
2	I_3	I_4		I_5	
3	I_3	I_4		I_6	
4	$Reduce_{A \to b}$	$Reduce_{A \to b}$	$Reduce_{A \to b}$		
5	$Reduce_{S \to AA}$	$Reduce_{S \to AA}$	$Reduce_{S \to AA}$		
6	$Reduce_{A \to aA}$	$Reduce_{A \to aA}$	$Reduce_{A \to aA}$		

Constructing ***SLR(1)*** parsing table

We have stated that both *LR(0)* and *SLR(1)* use canonical LR(0) collection items to construct their parsing tables. Also, we stated that all *LR(k)* parsers use the same parsing components, such as memory stack and *LR(k)* parsing algorithm. The question to ask is what makes an *LR(0)* parser different from *SLR(1)* parsers? One obvious difference is, an *LR(0)* parser uses zero tokens of lookahead to determine the next action to take. Therefore, the parser must have an unambiguous action to choose—either it shifts a specific symbol or applies a specific reduction. If there is more than one choice to make, the parser fails and it is no longer an *LR(0)* parser. Furthermore, an *LR(0)* parser cannot tell whether it encountered *shift-reduce conflicts* or *reduce-reduce conflicts* because it is not allowed to peek at the next token of an input. In contrast, *SLR(1)* performs a lookahead to decide whether it should reduce or shift. So, the difference between *LR(0)* and *SLR(1)* parser is this extra ability to help decide what action to take or makes a smart decision when there is a conflict. For this reason, any grammar that can be parsed by *LR(0)* parser can be parsed by *SLR(1)* parser. However, not all grammars that are parsed by *SLR(1)* cannot be parsed by *LR(0)* parser.

Table 2.4 shows the parsing table for *SLR(1)* parser. The only difference between *SLR(1)* and *LR(0)* table is that when a state is a final item in *LR(0)* parser, reduction is eminent to all the terminals. Where as, in *SLR(1)* parser it is not the case; for example, state I_5 has a final item $S \to AA.$ as shown in Figure 2.13; since the character after the "." is \$, we can call *FOLLOW* on the left hand side. So, *FOLLOW(S)* returns *\$*, and we can only place a reduce on the *\$* column of the *SLR(1)* parsing table. Therefore, *SLR(1)* parser says first call *FOLLOW* function on the left hand side of final item, then apply reduction to terminals returned by the function.

Table 2.4: Parsing table for *SLR(0)* parser. Where I_n is transitioning to n state according to the transition diagram in Figure 2.12.

	Action			Goto	
	a	**b**	**$**	**A**	**S**
0	I_3	I_4		I_2	I_1
1			$Accept_{S' \to S}$		
2	I_3	I_4		I_5	
3	I_3	I_4		I_6	
4	$Reduce_{A \to b}$	$Reduce_{A \to b}$	$Reduce_{A \to b}$		
5			$Reduce_{S \to AA}$		
6	$Reduce_{A \to aA}$	$Reduce_{A \to aA}$	$Reduce_{A \to aA}$		

Constructing ***LALR(1)*** parsing table

Lookahead LR(1) or *LALR(1)* is the most powerful and practically applicable bottom-up *LR(k)* class of parser. In the worst case, an LALR(1) parser has a time complexity of O(n). Comparing the parser size, both *SLR(1)* and *LALR(1)* parser tables are the same in size because they have the same number of states. Also, it is expected that sets of shift and accept transitions in these two parsers will be identical; whereas, sets of reduction actions in the *LALR(1)* parser is a subset of sets of reduction actions in *SLR(1)* parser. So, the difference between *SLR(1)* and *LALR(1)* parser becomes indisputable when applying the reduction action for a reducible item. We said *SLR(1)* parser applies reduction to any lookahead in the following sets of the LHS of the reducible item. In contrast, *LALR(1)* parser will only include the lookaheads which are feasible in the context of the state. This means *LALR(1)* uses canonical *LR(1)* collection—the combination of canonical *LR(0)* collection and a lookahead. For example, given the canonical *LR(0)* collection $S \to .aA,[a,b]$, a and b are the lookaheads for canonical *LR(0)* collection. The benefit of this lookahead comes when we reach the final item—$S \to aA.$ is the final item.

When we are constructing *LR(0)* parsing table, if the final item is reached, we placed the reduced action for every terminal at its corresponding state number. Similarly, for *SLR(1)* parser, we placed the reduced action in the follow of LHS. However, in *LALR(1)* parser, reduce action is placed on the lookahead symbols. So, to determine the lookahead symbols for every canonical *LR(0)* collection and to construct the parsing table for *LALR(1)* parser, first we need to generate a goto-graph or transition table. For this we can follow this simple rule:

- Start with the augmented production, this production lookahead is always a *$*.
- If $A \to a.Bb$, is not a final item, production for B should be added as part of the current state. It should also use the *FIRST* function to determine the lookaheads.

Figure 2.13 shows the transition table or goto-graph for *LALR(1)* parser. Once the goto-graph is generated, we can follow the same steps as *SLR(1)* parser to construct the parsing table for *LALR(1)* parser. Table 2.6 show the parsing table constructed according to goto-graph for *LALR(1)* parser. The table size for *LALR(1)* parser is almost twice

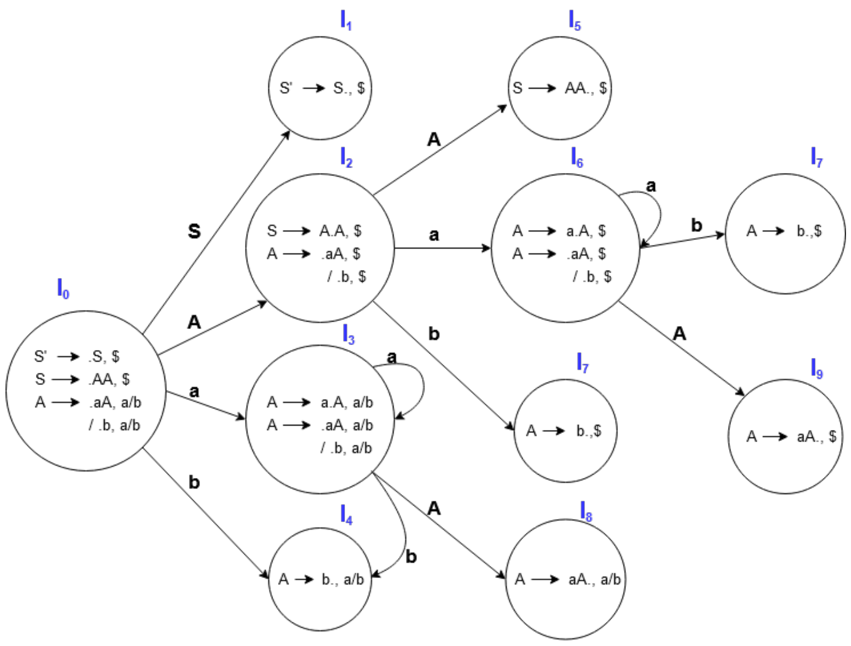


Figure 2.13: A transition diagram or goto-graph for *LALR(1)* parser.

the size of *SLR(1)* parser because of lookaheads. In order to reduce the table size we will combine the states with similar production but different lookaheads to the same row as shown in Table 2.6.

Table 2.5: Parsing table for *LALR(1)* parser. Where I_n is transitioning to n state according to the transition diagram in Figure 2.12.

	Action			Goto	
	a	**b**	**$**	**A**	**S**
0	I_3	I_4		I_2	
1					
2	I_6	I_7		I_5	
3	I_3	I_4		I_8	
4	$Reduce_{A \to b}$	$Reduce_{A \to b}$	$Reduce_{A \to b}$		
5			$Reduce_{S \to AA}$		
6	I_6	I_7	I_9		
7			$Reduce_{A \to b}$		
8	$Reduce_{A \to aA}$	$Reduce_{A \to aA}$			
9			$Reduce_{A \to aA}$		

2.2.5 CYK parsing algorithm

Cocke-Younger-Kasami algorithm (CYK) is a parsing algorithm that operates on context-free grammars. CYK is based on dynamic programming and can recognize CFG lan-

Table 2.6: Revised parsing table for *LALR(1)* parser. Where I_n is transitioning to n state according to the transition diagram in Figure 2.12.

	Action			Goto	
	a	**b**	**$**	**A**	**S**
0	$I_{\{3,6\}}$	$I_{\{4,7\}}$		I_2	
1					
2	$I_{\{3,6\}}$	$I_{\{4,7\}}$		I_5	
{3,6}	$I_{\{3,6\}}$	$I_{\{4,7\}}$		I_8	
{4,7}	$Reduce_{A \to b}$	$Reduce_{A \to b}$	$Reduce_{A \to b}$		
5			$Reduce_{S \to AA}$		
6	I_6	I_7	I_9		
{8,9}	$Reduce_{A \to aA}$	$Reduce_{A \to aA}$	$Reduce_{A \to aA}$		

guages in at most O(n^3) time for strings with length n [1]. A Graphics Processor Units (GPUs) implementation is reported in [22] with 26x speedup w.r.t. a sequential C implementation

2.3 Anatomy of a compiler

A program that can read a source language and translate it to a target language is known as a *compiler*. There are different types of compilers; the two major ones are the target program generator and an interpreter [1]. In this thesis we are interested in utilizing certain steps of the compiler process to implement our parser; for this, we need to understand the structures of a compiler. For an average user, a compile is one big box that takes in an input and produces an output. This box can be divided into two major parts: *analysis* or *front-end* and *synthesis* or *back-end*. During the compilation process, the analysis part takes in the source program or inputs and breaks them down into the constituent pieces and introduces a grammatical structure on them. In this step, important information about the input source is collected and stored in a *symbol-table* data structure. Then the synthesis part constructs the desired output using the information passed to it from the analysis part [1, p.5]. Figure 2.14 depicts conceptual structure of a compiler.

When both the analysis and synthesis part of a compilation process are examined in detail, we discover that they operate in sequences of *phases*. Figure 2.15 is the decomposition of the front-end and back-end process of a compiler into phases. To construct our parser, we will use the analysis phases of the compiler process—lexical analysis, syntax analysis, and semantic analysis—with the help of front-end analysis tools: flex and bison.

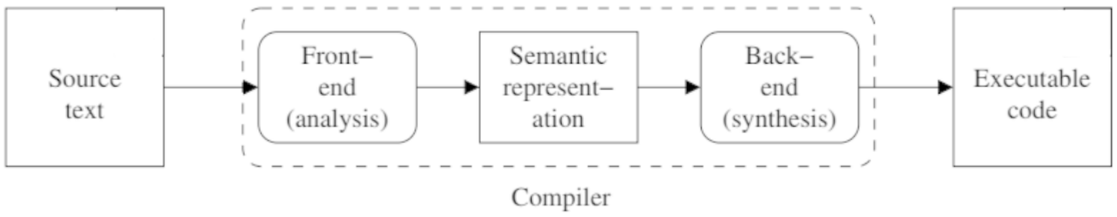


Figure 2.14: Conceptual structure of a compiler.

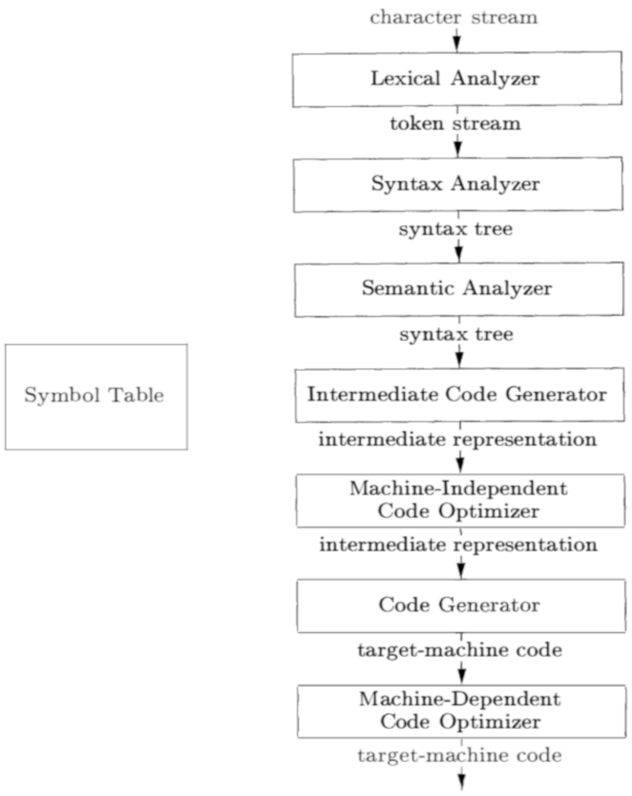


Figure 2.15: Front-end and back-end compiler process decompose into phases.

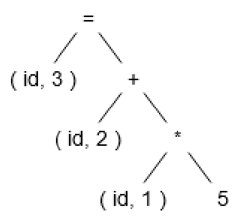


Figure 2.16: A syntax tree for the token of stream generated from the example *x = y + z * 5*.

2.3.1 Lexical analysis

In the first phase of the compilation process, the lexical analyzer is also known as a scanner that reads the input or source program one character at a time. During this process stream of characters making up the input or source program are grouped into meaningful sequences called *lexemes*. These lexemes are associated with an attribute value and a token name to produce what is called a token. A *token* is the smallest element or an atomic unit of an input or source program that is meaningful to a compiler. Most lexers use white space as a separation point for tokens.
There are many reasons why lexical analysis is separate from syntax analysis, the two major grounds are efficiency and modularity. In general, analysis is faster if lexical analysis is separated from syntax analysis because it is the simplest part of the parsing process. Similarly, if the analysis process is modular, the syntactical description of the language will not be cluttered with lexical details and will be easier to read and to understand.

It is easy to write a simple lexer by hand, though as the input program gets larger and complex, it is difficult to maintain these lexers. Hence, we can use *lexer-generator* to generate human-readable specifications of tokens including whitespace, into tokens. Lexer specification is written using regular expressions: algebraic notation for describing sets of strings Section 2.1.1. This is so the lexer can distinguish between several different types of tokens that are described by a regular expression. For our parser we will be using a program called *flex* to generate lexer for syntax analysis; this program takes in sets of token definitions and generates output to feed another program called *bison*.

2.3.2 Syntax analysis

The second phase of the process is syntax analysis, it is also known as parsing; it uses the result from lexical analysis to create a tree-like intermediate representation that depicts the grammatical structure of the token stream [1, p.8]. The structure of a *Syntax tree* is an interior node representing an operation and the children of the node represent the arguments of the operation. So if the leaves are read from left to right, the sequence is the same as the input text. In addition, the syntax analysis conducts error detection; it rejects invalid inputs and reports them as *syntax error*. For example, *x = y + z * 5* is an operation in which the assignment has to be performed; Figure 2.16 shows its syntax tree. Tree nodes with variables are represented with a token name and value; the syntax table matches the corresponding token-value with the variable.

2.3.3 Semantic analysis

The semantic analysis phase uses the syntax tree and the syntax table to check the input text or source program for semantic consistency. A context-free grammar or the human-readable format *BNF* grammar imposes a structure on the input string that is used to check this consistency. Besides, *type checking* is done so each operation has a matching operand. If type checking fails, a syntax error will be reported. Another important step in semantic analysis is *coercions*; it applies the correct type to a declared variables; for instance take $x = y + z * 5$, if y and z have floating-point type, the assigned value to x will be floating-point, despite the integer lexeme 5.

2.4 Polynomials

We recall the definition of a polynomial in an unformal way. A monomial is a product of powers of variables. A term is a product of a monomial and a coefficient. A polynomial is a sum of terms. For example, the polynomial $p(x,y) = 8x^5y^4 + 6xy + 3y + 2$. It has 4 terms, namely $8x^5y^4$, $6xy$, $3y$ and 2. We can see there are essentially two parts which make up each *term* of a polynomial, the numerical coefficient and the multiplicative combination of the variables. This multiplicative combination is called a *monomial*. We say that the coefficients belong to some ring, R, and that the polynomial is formed over that base ring [4]. In other words, *polynomials over R* to mean a ring of polynomials whose coefficients belong to R. In the above example, the polynomial $p(x,y)$ is a polynomial over $\mathbb{Z}$ with variables x and y. The ring formed by such polynomials is denoted by $\mathbb{Z}$[x,y].

In general, polynomials can be *univariate* or *multivariate* polynomials over R. A polynomial ring in the variables $x_1,\ldots,x_n$ over the base ring R is represented by $R[x_1,\ldots,x_n]$ [4]. We say it is *univariate* if variables $v = 1$ and *multivariate* if $v > 1$. In addition, specific polynomials have some aspects of internal structures. For example, a non-zero polynomial $p \in R[x_1,\ldots,x_n]$ have the following properties: *leading terms*, the first non-zero terms of p; *leading coefficient*, the coefficient of leading term; *total degree*, the maximum sum of exponent of a single non-zero term, etc. Detail reference regarding polynomials rings, definitions, notations, and internal structures can be found in [21] [4]. Some aspects of these internal structures can be used to perform polynomial ordering (*term ordering*). The two common polynomial orderings are *lexicographical* and *degree lexicographical*.

Given a bivariate monomial lexicographical ordering looks like [4]:

$$x^ny^n > x^{n-1}y^n > \ldots > xy^n > \ldots > x > y^n > y^{n-1} > \ldots > y > 1.$$

Given a bivariate monomial degree lexicographical ordering looks like [4]:

$$x^ny^n > x^ny^{n-1} > x^{n-1}y^n > \ldots > x^2y > xy^2 > x^2 > xy > y^2 > x > y > 1.$$

Sorting the parsed polynomial is important to define not only *leading* or *first* but also *canonical representation* of a polynomial. This representation is computationally important to efficiently perform operations such as degree, leading term, and equality testing. For our parsing library, we use lexicographical ordering.

There are dense and sparse polynomial representations. Dense polynomials include terms whose coefficients are zero, whereas sparse polynomial terms always include non-zero coefficients values. Simply put, a sparse polynomial is one whose zero coefficients are not explicitly stored. A natural sparse representation of a polynomial is a list of nonzero coefficient-exponent tuples. The parser uses sparse data-structures (linked-lists and alternating arrays) thus assuming by default that the input polynomial is sparse.

2.5 Basic polynomial algebra subprograms

Basic Polynomial Algebra Subprograms (BPAS) is an open-source library for high-performance polynomial operations; it is mainly written in C for performance and C++ to take advantage of object-oriented programming, interface portability, and end-user usability [4]. These high-performance polynomial operations include arithmetic, real root isolation, and polynomial system solving. BPAS makes use of the CilkPlus extension tool for parallelization and improved performance on multi-core processes [2]. It is within this library that we include our parallel polynomial parsing library that generates BPAS compatible polynomial data structure. Our parsing library is highly optimized to parse both dense and sparse polynomials; to accomplish this task, during the parsing process, we take advantage of the highly optimized functions available in the BPAS library for some arithmetic operations.

2.6 Alternating array

The BPAS library uses an alternating array data structure to effectively and efficiently represent a polynomial. An alternating array polynomial representation stores both coefficients and monomials side-by-side in the same array. When coefficient and monomials are side-by-side, it increases the locality of a coefficient with respect to its monomials [4]. On the other hand, storing the coefficients directly in the array increases the distance between adjacent monomials or coefficients. This decreases the locality of monomials with respect to its adjacent monomials and vice versa.

A single polynomial term is a combination of a coefficient (integer or rational number) and monomial. Since the BPAS library requires multi-precision coefficients, it uses GMP (GNU Multi-Precision arithmetic)—a highly optimized library for operating arbitrary precision arithmetic on integers and rational numbers—to encode coefficients [10]. Similarly, monomials need optimal representation. Monomials contain an unsigned integer vector (array) of exponents and their associated variables; we only need to encode the exponent vector because we imposed lexicographical ordering on the variables. Although encoding exponent vectors improves memory used by an alternating array, the BPAS library uses an exponent packing strategy to further improve memory footprint. Exponent packing is a method of encoding multiple integers into a single (64-bit) machine word; this can be accomplished using bit-masks and shifts [4, 36]. For more about how exponent vectors are packed into a single (64-bit) machine word, refer to [4].

Exponent packing has a huge advantage in BPAS alternating array polynomial representation. It saves memory and has an important computational advantage since it uses a single machine word for each monomial [4]. Using a single machine word monomial improves the overheads associated with Arithmetic algorithms memory usage and machine instruction size [4].

2.7 Bison and flex

Flex and Bison are open source tools used to build programs that handle structured input [13]. There are many reasons why we chose to use Flex and Bison to build our parser; in general, these tools together generate a parser that is faster for any complex grammar in a short time; also, updating and fixing Flex and Bison source code is easier than debugging custom designed parsers source code. In addition, parsing specific types of polynomial has its challenges; manually writing a parser that handles nested polynomials is impossible and difficult but Flex and Bison can easily resolve this challenge.

2.7.1 Flex

Scanning is the process of looking for a pattern in an input. For example, in a binary arithmetic operation, there are two operands and an operator. The operands can be variables assigned to a number or constant values. To describe these patterns we will use regular expressions. A Flex program consists of a list of regular expressions with instructions on what to do when the input matches called *actions*. Therefore, a flex generated scanner reads the input and matches the input against the regular expression and performs the appropriate actions [13]. As we stated earlier, a flex scanner uses a rich regular expression language. This pattern description uses a metalanguage, a language used to describe what we want the pattern to match. Although flex can generate a scanner for many programming languages, by default flex generates a *C* programming source code. The output generated by flex is fed to Bison as an input to create a parser.

2.7.2 Bison

Once the input streams are divided into pieces or tokens, we use the Bison program to group them logically. To do this first we specify a grammar so bison can recognize it. Bison takes a grammar that we specified and writes a parser that recognizes a syntactically valid sentence.

When running a Bison generated parser, the parser works by looking for rules that might match the tokens seen so far, then it creates a set of states with the possible positions in a partially parsed rules [13]. As a token is read, if it doesn't complete a rule, it is pushed to an internal stack and switches to a new state that reflects the token it read. This step is known as *shift*. Similarly, if a right-hand side symbol from the grammar that matches the sets of tokens in the stack is found, they are popped from the stack and replaced by the right-hand side symbol. This process is known as *reduction*. Few parsing algorithms follow the above steps. Bison generates an *LALR(1)* parser by default [13]. In addition to this, bison is capable of recognizing ambiguous grammars, and shift-shift or shift-reduce conflicts.

Chapter 3

Serial parser

In this chapter, we will discuss our serial parser and its implementation. We will start with a brief introduction on why we chose linked-list over an alternating array as our default data structure in Section 3.1. Scanning and parsing requires defining a regular expression to tokenize an input polynomial and using grammar to validate its structure. In Section 3.2, we will discuss regular expressions and grammars used in Flex and Bison generated codes, and in Section 3.2.3, we will explore all the grammar productions and their action codes. Finally, we conclude the chapter with a conclusion, Section 3.3.

3.1 Disadvantage of packed alternating array in parsing

In general, parsing an input polynomial has two main steps. Initially, we run it through a *lexical analyzer* or a *scanner* to divide the input into meaningful chunks or tokens based on a predefined regular expression. Then a *syntax analyzer* or *parser* takes these tokens to group them logically [13]. During these two steps, tokens passed from the scanner to the parser are assembled into *terms* and afterward to *polynomials.* We can accomplish this using the *C* code defined in the action section of the bison grammar as described in Chapter 2.

The *actions* taken throughout each reduction step during syntax analysis—executing codes associated with the rule—determines the time efficiency of our parser. It implies each time tokens are grouped into a *term* production, the time to reallocate contiguous memory location for an alternating array will significantly affect run time. While packed alternating array polynomial is an effective and efficient data structure to perform polynomial operations, it is not an efficient data structure to use for our parser.

To level with this difficulty, we used a linked-list as our parser's main data structure. It is faster to operate our parser using a linked-list data structure and later convert the result to an alternating array. An additional reason to choose a linked-list is that it is a dynamic data structure, its size can grow and shrink at runtime by allocating and deallocating memory. Also, insertion and deletion of nodes can be done at constant time, i.e., saving time wasted to re-allocating memory for fixed-size data structures like an alternating array. During the parsing process, every node in the linked-list is a polynomial *term*. Whenever tokens are grouped into a *term*, the reduction action will create a new linked-list for the first time or perform an update on an existing linked-list with constant time, i.e., $O(1)$.

3.2 Scanning and parsing

In the previous section, we stated the two main steps involved in parsing an input polynomial: *scanning* and *parsing*. In this section, we will see how these steps are combined to create a serial parser. As described in Chapter 2, we will be using flex and bison to generate a scanner and parser library.

3.2.1 Scanning with flex

Tokenizing input data using a lexical analyzer such as flex to produce tokens requires rich regular expression language. Flex uses regular expression—pattern description using metalanguages—to describe what we want the pattern to match [13]. This regular expression language used in flex is a POSIX-extended regular expression. It means the metalanguages are standard text characters to represent themselves and their patterns. Luckily, generating a pattern description or regular expression for an input polynomial is not as complicated as it sounds. Table 3.1 shows all patterns used to tokenize a string polynomial using our scanner, and their return types. The *Token* column shows the definition of the tokens, whereas the *Token Values* column shows the actual regular expressions. For example, when tokenizing the polynomial term *8∗x^5*, the scanner recognizes the following patterns: [*rational numbers*],[*multiply*], [*variables*], [*power*], and [*digit*]. Once a token is recognized by one of the regular expressions in Table 3.1, it is returned by the function *yylex*—a entry point *C* function name that flex program gives to the scanner. This return value passed to the parser generator (Bison) as a *string* type and grouped logically based on a pre-defined grammar.

3.2.2 Parsing with bison explain

As described in the Bison introduction paragraph in Chapter 2, Bison logically groups sets of tokens using a well-defined grammar. Since we are using the default *LALR(1)* algorithm, bison takes a pre-defined grammar and writes a parser that recognizes valid polynomial syntax. Grammar 3.1 shows all the production steps involved to generate a parser that returns a single linked-list polynomial data structure before converting it into an alternating array.

In the previous paragraph, we stated that Grammar 3.1 generates a single linked-list polynomial data structure from an input polynomial. Depending upon the input type, it is not always the case. When an un-expanded input polynomial is used, i.e., one that requires arithmetic operation such as multiplication, the semantic value assigned to the production will change from a linked-list to an alternating array. It is because our parser depends on the BPAS arithmetic operation helper library—*SMQP_Support*—for arithmetic operations. These functions take in an alternating array as input and return an alternating array. Some of these arithmetic functions used from the library include: *addPolynomials_AA*, to add two alternating arrays; *exponentiatePoly_AA*, to exponentiate an alternating array; and *multiplyPolynomials_AA*, to multiply two alternating arrays. Although we stated that linked-list to be our parser main data structure, if an arithmetic operation is required in the parsing process, the linked-list data structure will be converted to an alternating array. Section 3.2.3 will clarify the previous statement when we talk about the action codes triggered by *poly* production rules.

Tokens	Token Values	Return type Enum
digit	$[0-9]+$	NUM
variables	$[a-zA-Z\_]\,[\_a-zA-Z0-9]*$	VAR
whitespace	[\t]$*$	
newlines	[\n]	\n
period	”.”	exit
rational numbers	$[0-9]+\backslash/[0-9]+$	RATNUM
negative rational numbers	$[-][0-9]+\backslash/[0-9]+$	MINUS RATNUM
minus	” − ”	MINUS
plus	” + ”	PLUS
divide	”/”	DIVIDE
left bracket	”(”	L_BRACE
right bracket	”)”	R_BRACE
multply	” ∗ ”	MULTIPLY
comma	”, ”	COMMA
left square bracket	”[”	LS_BRACE
right square bracket	”]”	RS_BRACE
power	”^”	POWER

Table 3.1: Alternating Array definition and helpers

To understand how our Bison's parser generates an alternating array, we will look into each production and their corresponding action code. Our Bison generated parser uses eight productions with multiple rules to parse a flat[1] or un-expanded[2] input polynomial to an alternating array as shown in Grammar 3.1. Depending on the input type (flat or un-expanded), it will return an alternating array or a linked-list that requires an additional step to convert it to an alternating array. Once converted, the alternating array monomials are simplified to contain only the exponent vector. For this reason, to keep track of the exponent vectors and their size, our parser uses two helper properties

[1]plain simple polynomial without any nested components

[2]nested polynomial

$$
\begin{array}{lcl}
\langle \mathit{polynomial} \rangle & ::= & \langle \mathit{poly} \rangle \\
 & | & \langle \mathit{polynomial} \rangle\ \langle \mathit{other} \rangle \\
 & | & \langle \mathit{empty} \rangle \\[1ex]
\langle \mathit{poly} \rangle & ::= & \langle \mathit{poly} \rangle\ \text{`}\texttt{+}\text{'}\ \langle \mathit{poly} \rangle \\
 & | & \langle \mathit{poly} \rangle\ \text{`}\texttt{-}\text{'}\ \langle \mathit{poly} \rangle \\
 & | & \text{`}\texttt{(}\text{'}\ \langle \mathit{poly} \rangle\ \text{`}\texttt{)}\text{'`}\texttt{\^{}}\text{'}\ \langle \mathit{exponent} \rangle \\
 & | & \langle \mathit{MINUS} \rangle\ \text{`}\texttt{(}\text{'}\ \langle \mathit{poly} \rangle\ \text{`}\texttt{)}\text{'`}\texttt{\^{}}\text{'}\ \langle \mathit{exponent} \rangle \\
 & | & \langle \mathit{poly} \rangle\ \text{`}\texttt{*}\text{'}\ \text{`}\texttt{(}\text{'}\ \langle \mathit{poly} \rangle\ \text{`}\texttt{)}\text{'} \\
 & | & \langle \mathit{poly} \rangle\ \text{`}\texttt{*}\text{'}\ \langle \mathit{term} \rangle \\
 & | & \langle \mathit{term} \rangle\ \text{`}\texttt{*}\text{'}\ \text{`}\texttt{(}\text{'}\ \langle \mathit{poly} \rangle\ \text{`}\texttt{)}\text{'} \\
 & | & \langle \mathit{coef} \rangle\ \text{`}\texttt{*}\text{'}\ \text{`}\texttt{(}\text{'}\ \langle \mathit{poly} \rangle\ \text{`}\texttt{)}\text{'} \\
 & | & \langle \mathit{poly} \rangle\ \text{`}\texttt{+}\text{'}\ \langle \mathit{term} \rangle \\
 & | & \langle \mathit{poly} \rangle\ \text{`}\texttt{-}\text{'}\ \langle \mathit{term} \rangle \\
 & | & \text{`}\texttt{(}\text{'}\ \langle \mathit{poly} \rangle\ \text{`}\texttt{)}\text{'} \\
 & | & \langle \mathit{MINUS} \rangle\ \text{`}\texttt{(}\text{'}\ \langle \mathit{poly} \rangle\ \text{`}\texttt{)}\text{'} \\
 & | & \langle \mathit{term} \rangle \\[1ex]
\langle \mathit{term} \rangle & ::= & \langle \mathit{coef} \rangle \\
 & | & \langle \mathit{powerVariable} \rangle \\
 & | & \langle \mathit{coef} \rangle\ \text{`}\texttt{*}\text{'}\ \langle \mathit{term} \rangle \\
 & | & \langle \mathit{powerVariable} \rangle\ \text{`}\texttt{*}\text{'}\ \langle \mathit{term} \rangle \\[1ex]
\langle \mathit{powerVariable} \rangle & ::= & \langle \mathit{variable} \rangle\ \text{`}\texttt{\^{}}\text{'}\ \langle \mathit{exponent} \rangle \\
 & | & \langle \mathit{variable} \rangle \\[1ex]
\langle \mathit{variable} \rangle & ::= & \langle \mathit{VAR} \rangle \\
 & | & \langle \mathit{MINUS} \rangle\ \langle \mathit{VAR} \rangle \\[1ex]
\langle \mathit{exponent} \rangle & ::= & \langle \mathit{NUM} \rangle \\[1ex]
\langle \mathit{coef} \rangle & ::= & \langle \mathit{NUM} \rangle \\
 & | & \langle \mathit{MINUS} \rangle\ \langle \mathit{NUM} \rangle \\
 & | & \langle \mathit{MINUS} \rangle\ \langle \mathit{RATNUM} \rangle \\
 & | & \langle \mathit{RATNUM} \rangle \\[1ex]
\langle \mathit{other} \rangle & ::= & \langle \mathit{variable} \rangle \\
 & | & \text{`}\texttt{[}\text{'}\ \langle \mathit{other} \rangle \\
 & | & \langle \mathit{other} \rangle\ \text{`}\texttt{]}\text{'} \\
 & | & \langle \mathit{other} \rangle\ \text{`}\texttt{,}\text{'}\ \langle \mathit{variable} \rangle
\end{array}
$$

Grammar 3.1: Bison grammar that reduces non-terminal *term* into non-terminal *poly*.

of type string array and an integer. These properties are defined as *g_variables* and *g_num_variables*. These properties are returned with the parsed polynomial.

3.2.3 Bison grammar rules and action codes

Since Bison is a bottom up parser, we will start explaining Grammar 3.1 production rules starting from the last production. The bottom production, *coef*, has six rules: *NUM*, *MINUS NUM*, *RATNUM*, and *MINUS RATNUM*. These rules are matching terminal symbols used in the regular expression Table 3.1 column *Return type Enum*. When one of these rules are a match, the token returned is a coefficient. All six rules have simple action, i.e., to convert the token value to a string. Listing 3.1 shows rule *MINUS NUM* action code; the symbol *$2* and *$$* are reference to *NUM* token and *coef*, respectively.

Listing 3.1: Bison grammar production *coef* action codes for rule *MINUS NUM*

```
char *negative_num = (char*)calloc(strlen($2)+2, sizeof(char));
strncat(negative_num, "-", 1);
strncat(negative_num, $2, strlen($2));
$$ = negative_num;
```

The *exponent* production, as the name suggests identifies exponents; this production is triggered if *NUM* token is followed by a *POWER* token. The production has one rule that converts token *NUM* to a string as shown inListing 3.2.

Listing 3.2: Bison grammar production *exponent* action codes for rule *NUM*

```
$$ = strdup($1);
free($1);
```

The *variable* production has two rules: *VAR*, a rule that matches variable token; and *MINUS VAR*, a rule that matches a negative variable. All variables without coefficient are considered to have a *1* as their coefficient. Similarly, a negative variable has *-1* as its coefficient. Therefore, similar to the *expoenent* production, the action taken by the *variable* production converts the token value to a string.

The *powerVariable* production is matched when a monomial with a single variable introduced. This production has two rules: *variable POWER exponent*, and *variable*. As discussed in Chapter 2, Bison uses stack memory as part of the parsing process. If stack memory matches one of these rules, it will reduce to a monomial with a single variable. As a result, a special struct type called *powervar* is returned by the action code. To create this type, we used a helper function called *create_power_var* that combines a variable and a power exponent. Listing 3.3 shows action code for the first rule (*variable POWER exponent*). Similar action can be applied to the second rule with an exponent value of *1*.

Listing 3.3: Bison grammar production *powerVariable* action codes for rule *NUM*

```
powervar *temp_power_var;
temp_power_var = create_power_var($1, strtoul($3, NULL, 10));
$$ = temp_power_var;
```

After creating one or more *powervar* struct types in the *powerVariable* production, the next step is to generate a polynomial term. The *term* production action code generates a term; it is a combination of one or more *powervar* preceded with or without a coefficient. We used a special *C* struct type, *term_q* to represent a polynomial term in our parser. To match the *term* production, the matching token on top of the memory stack must be preceded and followed by either a *PLUS* or *MINUS* token. The following five rules define the structure of a polynomial term:

1. Rule *coef*: a number or coefficient,
2. Rule *powerVariable*: a combination of a variable followed by an exponent or *powervar* type,
3. Rule *coef MULTIPLY term*: the product of a number or coefficient and a *term_q* type,
4. Rule *powerVariable Multiply term*: the product of multiple *powervar* type,
5. Rule *powerVariable MULTIPLY L_BRACE term R_BRACE*: any *term_q* type in parenthesis muliplied by a *powervar* type

To generate a *term_q* type we used two helper functions: *create_term_empty_q*, instantiate *term_q* structure; and *linked_list_fill_term_exponent*, initialize the structure components. Listing 3.4 shows the action code used for rule *powerVariable* in *term* production. The action code initially creates an empty *term_q* type in **Line 2**. Since the coefficient value is *1*, **Line 3** checks if it is preceded by a *MINUS* token to apply the correct sign. The function in **Line 7** initializes the empty *term_q* components instantiated in **Line 2**. All the other rules from the *term* production share similar action code as Listing 3.4 with a few exceptions (refer the full source code).

Listing 3.4: Bison grammar production *powerVariable* action codes for rule *NUM*

```
 1  term_q* local_term;
 2  local_term = create_term_empty_q(pParserEntry->g_num_variables);
 3  if(pParserEntry->is_negative_var){
 4          mpq_set_str(local_term->coef, "-1", 10);
 5          pParserEntry->is_negative_var = 0;
 6  }
 7  linked_list_fill_term_exponent(pParserEntry->g_variables,
 8                                  local_term->deg,
 9                                  $1->var,
10                                  $1->exp,
11                                  &(pParserEntry->g_num_variables)
12  );
13  $$ = local_term;
```

Finally, the last production that will generate a linked-list or alternating array is the *poly* production. This production has multiple rules that are carefully crafted to reduce any parsing conflicts. To reduce tokens residing in the stack memory into a *poly* production type, i.e., linked-list or alternating array, it must contain one or more *term_q* types. One or more terms can make up a polynomial. When *term_q* types that make up a single polynomial are shifted (pushed) into the stack, they are preceded and followed by a *PLUS* or *MINUS* tokens. Below is a list of all the possible rules and their descriptions to create of a *poly* type:

1. Rule *term*: reduces one or more *terms* to *poly*,
2. Rule *MINUS L_BRACE poly R_BRACE*: negates a *poly* surrounded by a parentheses and reduced to a *poly*,
3. Rule *L_BRACE poly R_BRACE*: a *poly* surrounded by a parentheses is reduced to a *poly*,
4. Rule *poly MINUS term*: a *term* subtracted from a *poly* is reduced to a *poly*,
5. Rule *poly PLUS term*: a *term* added to a *poly* is reduced to a *poly*,
6. Rule *coef MULTIPLY L_BRACE poly R_BRACE*: a coefficient multiplied by a *poly* surrounded by parentheses is reduced to a *poly*,
7. Rule *term MULTIPLY L_BRACE poly R_BRACE*: a *term* multiplied by a *poly* surrounded by parentheses is reduced to a *poly*,
8. Rule *poly MULTIPLY term*: a *poly* multiplied by a *term* is reduced to a *poly*,
9. Rule *poly MULTIPLY L_BRACE poly R_BRACE*: a *poly* multiplied by a *poly* surrounded by parentheses is reduced to a *poly*,
10. Rule *MINUS L_BRACE poly R_BRACE POWER exponent*: a *poly* surrounded by parentheses is negated and exponentiated then reduced to a *poly*,
11. Rule *poly MINUS poly*: a *poly* subtracted from a *poly* is reduced to a *poly*,
12. Rule *poly PLUS poly*: a *poly* added to a *poly* is reduced to a *poly*,

Most *poly* production rules return a linked-list semantic[3] value; the following are rules that return an alternating array: *MINUS L_BRACE poly R_BRACE POWER exponent*; *poly MULTIPLY L_BRACE poly R_BRACE*; *poly PLUS poly*; and *poly MINUS poly*. If the input polynomial type, i.e., flat or un-expanded, affect the semantic type (linked-list or alternating array) of the parsed polynomial to change from a linked-list to an alternating array, some of the *poly* production rules action codes have a conditional statement that dynamically guide to the correct action code. These conditional statements rely on a boolean variable called *ret_type*, i.e., one of the many *C* properties defined to assist the Bison code. *ret_type* is a *return_type* enum that keeps track of the current state of *poly* semantic value pushed into the stack memory. It means, if a *poly* production semantic value is linked-list, *ret_type* is assigned *LINKEDLIST_TYPE*, otherwise it is an *ALTARRAY_TYPE*. For example, if *poly PLUS poly* rule is invoked, and if the first *poly* semantic value pushed to the stack as a linked-list and the second *poly* semantic value is an alternating array, a type mis-match error will occur.

Most *poly* production rules have similar initiation action code or algorithm before

[3]a production return value; bisons $$ contains the return value from its corresponding action code

generating a linked-list or an alternating array. Listing 3.5 shows the action code for the *term* rule. If this rule is called for the first time, *ret_type* will be *LINKEDLIST_TYPE*. Then an empty linked-list created using a helper function *create_linked_list_q*. The value associated with the rule is referenced using *$1*. This value has a *C* struct type called *term_q*. The value is inserted into the empty linked-list created in **Line 11** using the helper function called *linked_list_insert_q* in **Line 12**.

Listing 3.5: Bison grammar production *poly* action codes for rule *term*

```
1  if(pParserEntry->ret_type == return_type::ATLARRY_TYPE){
2          AltArr_t *temp_aa = makePolynomial_AA(DEFAULT_AA_SIZE,
3                                  pParserEntry->g_num_variables);
4          parser::add_packed_degree_term_to_smqp_aa(temp_aa,
5                                  ((term*)$1)->exp,
6                                  ((term*)$1)->coef,
7                                  pParserEntry->g_num_variables);
8          $$.list_ret = temp_aa;
9          $$.cur_num_vars = pParserEntry->g_num_variables;
10 }else{
11         linked_list_q *temp_ll = create_linked_list_q();
12         linked_list_insert_q(temp_ll, NULL, (term_q*)$1);
13         $$.list_ret = temp_ll;
14         $$.cur_num_vars = pParserEntry->g_num_variables;
15 }
```

When the *poly* production rule *MINUS L_BRACE poly R_BRACE* match the stack memory for reduction, the action code invoked depends on the *ret_type* value. If the value is an alternating array, a BPAS helper function is invoked to negate the alternating array. On the contrary, for a linked-list, a similar helper function is used to negate the polynomial referenced by *$3.list_ret*. Listing 3.6 shows the action code for this rule.

Listing 3.6: Bison grammar production *poly* action codes for rule *term*

```
if(pParserEntry->ret_type == return_type::ATLARRY_TYPE){
        negatePolynomial_AA((AltArr_t*)$3.list_ret);
        $$.list_ret = $3.list_ret;
        $$.cur_num_vars = pParserEntry->g_num_variables;
}else{
        linked_list_negate((linked_list_q*)$3.list_ret);
        $$.list_ret = $3.list_ret;
        $$.cur_num_vars = pParserEntry->g_num_variables;
}
```

One of the simplest action codes in the *poly* production is for the rule *L_BRACE poly R_BRACE* as shown in Listing 3.7. It extracts the value of a production that is surrounded by parentheses, and returns a linked-list.

Listing 3.7: Bison grammar production *poly* action codes for rule *term*

```
$$.list_ret = $2.list_ret;
$$.cur_num_vars = pParserEntry->g_num_variables;
```

The rules *poly PLUS term* and *poly MINUS term* use arithmetic operations to add or subtract a term to or from a polynomial. The rule with the *PLUS* token uses *addPolynomial_AA*. It is BPAS helper function that adds a *term* to an alternating array if *ret_type* is *ALTARRAY_TYPE* as shown in **Line 8**; the alternating array is created in **Line 2** using *makePolynomial_AA* helper function. On the other hand, if *ret_type* a is *LINKEDLIST_TYPE*, the *term* in (*$3*) is inserted into an existing linked-list in (*$1*). The same algorithm is applied to the rule with the *MINUS* token except it negates the *term* to reflect the correct *term_q* coefficient. Listing 3.8 shows the action code to add a negated *term_q* type to an alternating array or linked-list depending upon *ret_type* value.

Listing 3.8: Bison grammar production *poly* action codes for rule *poly MINUS term*

```
1  if(pParserEntry->ret_type == return_type::ATLARRY_TYPE){
2          AltArr_t *temp_aa = makePolynomial_AA(DEFAULT_AA_SIZE,
3                                  pParserEntry->g_num_variables);
4          parser::add_packed_degree_term_to_smqp_aa(temp_aa,
5                                  ((term*)$3)->exp, ((term*)$3)->coef,
6                                  pParserEntry->g_num_variables);
7                                  negatePolynomial_AA(temp_aa);
8          $$.list_ret = addPolynomials_AA((AltArr_t*)$1.list_ret,
9                                  temp_aa,
10                                 pParserEntry->g_num_variables);
11         $$.cur_num_vars = pParserEntry->g_num_variables;
12         freePolynomial_AA(temp_aa);
13         free_linked_list_term_q((term_q*)$3);
14 }else{
15         mpq_t temp_coef;
16         mpq_init(temp_coef);
17         mpq_set_str(temp_coef, "-1", 10);
18         mpq_mul(((term_q*)$3)->coef,
19                                 ((term_q*)$3)->coef,
20                                 temp_coef);
21         linked_list_insert_q((linked_list_q*)$1.list_ret,
22                                 NULL,
23                                 (term_q*)$3);
24         $$= $1;
25         mpq_clear(temp_coef);
26 }
```

When the stack memory matches a rule that multiplies a coefficient with a linked-list in a parenthesis, the reduction rule *coef MULTIPLY L_BRACE poly R_BRACE* is invoked. The helper function *linked_list_multiply_constant_coef* is used to evaluate their products as shown in Listing 3.9. The coefficient and linked-list values are referenced using the symbol *$1* and *$4*, respectively.

Listing 3.9: Bison grammar production *poly* action codes for rule *poly MINUS term*

```
linked_list_multiply_constant_coef(
                        (linked_list_q*)$4.list_ret,
                        $1
                );
$$.list_ret = (linked_list_q*)$4.list_ret;
$$.cur_num_vars = pParserEntry->g_num_variables;
```

Similarly, when we multiply a *term* with a polynomial, the stack memory is reduced to the following rules: *term MULTIPLY L_BRACE poly R_BRACE*, and *poly MULTIPLY term*. Listing 3.10 shows the action code for the production rule *term MULTIPLY L_-BRACE poly R_BRACE*. In **Line 1**, helper functions *linked_list_multiply_constant_-term_q* are invoked. It multiplies a *term* referenced by *$1* and a polynomial linked-list referenced by *$4*.

Listing 3.10: Bison grammar production *poly* action codes for rule *poly MINUS term*

```
1 linked_list_multiply_constant_term_q(
2                 (linked_list_q*)$4.list_ret,
3                 (const term_q*)$1,
4                 pParserEntry->g_variables,
5                 pParserEntry->g_num_variables
6         );
7 $$.list_ret = $4.list_ret;
8 $$.cur_num_vars = pParserEntry->g_num_variables;
```

When two polynomials are multiplied, the rule *poly MULTIPLY L_BRACE poly R_-BRACE* is matched. The resulting polynomial is an alternating array. This is because we are utilizing the BPAS helper function called *multiplyPolynomials_AA*. The function takes in the polynomials as a sorted alternating array arguments. This means the input arguments are alternating arrays that require to be sorted before invoking the function. Even worse, if one or both the input arguments are linked-list, they have to be sorted and converted to an alternating array. Listing 3.11 shows the action code applied to reduce a stack memory that matches the product of two polynomials.

Listing 3.11: Bison grammar production *poly* action codes for rule *poly MULTIPLY L_BRACE poly R_BRACE*

```
if(pParserEntry->ret_type == return_type::ATLARRY_TYPE){
        AltArr_t *temp_aa_result = makePolynomial_AA(
                                        DEFAULT_AA_SIZE,
                                        pParserEntry->
                                           g_num_variables
                                );
        mergeSortPolynomial_AA((AltArr_t*)$1.list_ret);
        mergeSortPolynomial_AA((AltArr_t*)$4.list_ret);
        temp_aa_result = multiplyPolynomials_AA(
                                        (AltArr_t*)$1.list_ret,
                                        (AltArr_t*)$4.list_ret,
                                        pParserEntry->
                                           g_num_variables
                                );
        $$.list_ret = temp_aa_result;
        $$.cur_num_vars = pParserEntry->g_num_variables;
}else{
        AltArr_t *temp_aa_result = makePolynomial_AA(
                                        DEFAULT_AA_SIZE,
                                        pParserEntry->
                                           g_num_variables
                                );
        AltArr_t *first = convert_linkedlist_to_altarr_q(
                                        (linked_list_q*)$1.list_ret,
                                        pParserEntry->
                                           g_num_variables
                                );
        mergeSortPolynomial_AA(first);
        linked_list_destroy_q((linked_list_q*)$1.list_ret);
        AltArr_t *second = convert_linkedlist_to_altarr_q(
                                        (linked_list_q*)$4.list_ret,
                                        pParserEntry->
                                           g_num_variables
                                );
        mergeSortPolynomial_AA(second);
        linked_list_destroy_q((linked_list_q*)$4.list_ret);
        temp_aa_result = multiplyPolynomials_AA(first,
                                        second,
                                        pParserEntry->
                                           g_num_variables
                                );
        $$.list_ret = temp_aa_result;
        $$.cur_num_vars = pParserEntry->g_num_variables;
        pParserEntry->ret_type = return_type::ATLARRY_TYPE;
}
```

When exponentiating a polynomial, the rules *L_BRACE poly R_BRACE POWER exponent* and *MINUS L_BRACE poly R_BRACE POWER exponent* matches during reduction. Also, the action code uses the BPAS helper function, *exponentiatedPoly_AA*. Both rules have similar action code that produces an alternating array. The only difference is that the second rule has a *MINUS* token in front of the polynomial, i.e., *poly ($2)* must be negated before exponentiation. Listing 3.12 shows the action code for the rule *L_BRACE poly R_BRACE POWER exponent*. Initially, the linked-list polynomial to exponentiate is sorted in **Line 1**. Then it is converted to an alternating array in **Line 6**. **Line 14** exponentiates the polynomial to return an alternating array.

Listing 3.12: Bison grammar production *poly* action codes for rule *L_BRACE poly R_BRACE POWER exponent*

```
$2.list_ret = serial_list_sort(
        (linked_list_q*)$2.list_ret,
        pParserEntry->g_num_variables,
        CACHE, MAX_SORT
);
AltArr_t *arr = convert_linkedlist_to_altarr_q(
        (linked_list_q*)$2.list_ret,
        pParserEntry->g_num_variables
);
linked_list_destroy_q((linked_list_q*)$2.list_ret);
AltArr_t *ret = makePolynomial_AA(DEFAULT_AA_SIZE,
        pParserEntry->g_num_variables
);
ret = exponentiatePoly_AA(arr, atoi($5),
        pParserEntry->g_num_variables
        );
$$.list_ret = ret;
$$.cur_num_vars = pParserEntry->g_num_variables;
pParserEntry->ret_type = return_type::ATLARRY_TYPE;
```

The last two rules of *poly* production are *poly PLUS poly* and *poly MINUS poly*; these rules add and subtract two polynomials respectively. If the semantic value of these rules is an alternating array, we can utilize the addition function *addPolynomial_AA*. Otherwise, we will use *linked_list_cat_q* to concatenate two linked-list. Both rules have similar action code, except during subtraction, the polynomial to subtract is negated. Listing 3.13 shows the action code for adding two polynomials.

Listing 3.13: Bison grammar production *poly* action codes for rule *poly PLUS poly*

```
if(pParserEntry->ret_type == return_type::ATLARRY_TYPE){
        AltArr_t *temp_aa_result = makePolynomial_AA(DEFAULT_AA_SIZE
            ,
                        pParserEntry->g_num_variables
        );
        mergeSortPolynomial_AA((AltArr_t*)$1.list_ret );
        mergeSortPolynomial_AA((AltArr_t*)$3.list_ret );
        temp_aa_result = addPolynomials_AA((AltArr_t*)$1.list_ret,
                        (AltArr_t*)$3.list_ret,
                        pParserEntry->g_num_variables
        );
        $$.list_ret = temp_aa_result;
        $$.cur_num_vars = pParserEntry->g_num_variables;
}else{
        linked_list_cat_q((linked_list_q*)$1.list_ret ,
                        (linked_list_q*)$3.list_ret
        );
        $1.list_ret  = serial_list_sort((linked_list_q*)$1.list_ret,
                                pParserEntry->g_num_variables , CACHE ,
                                    MAX_SORT
            );
            $$.list_ret   = $1.list_ret  ;
            $$.cur_num_vars = pParserEntry->g_num_variables;
}
```

Our Bison generated parser is a reentrant parser, i.e., a thread-safe parser that is written in a *C++* programming language (Chapter 2). The practical way to create a parallel parser is to use a reentrant parser. We adapted this reentrant parser without compromising efficiency so that we can organize all the parse-time variables used by bison grammar in a *C++* class. There are four *C++* fields used as a parse-time variables[4] in our Bison grammar. These parse-time variables, as defined in the source code are, *g_variables*, *g_num_variables*, *llist_q_data*, *altarr_data*, and *ret_type*. The first two parse-time variables are introduced in the last section; *g_variables* is an array that stores all the variables used in the input polynomial, and *g_num_variables* tracks the array size. After the input polynomial is parsed and returned as a linked-list, it is stored in *llist_q_data*. Otherwise, if it is an alternating array it is stored in *altarr_data*. Finally, the property *ret_type* is an enum type; it is used to track the currently half way parsed input type or the fully parsed polynomial type. This property is directly accessible in the grammar action code through a special Bison keyword. This keyword is called *%parse-param*, it is used to access user defined parse-time variables by passing them as argument [6].

Algorithm 1 shows a simple entry point class method to our serial parser. The algorithm takes in two arguments. A string polynomial to parse, and a scanner variable that is used by Flex program. The scanner variable of type *yysacn_t* is initialized using the function *yy_init*. Once the *scanner* is initialized, the input string is streamed to an internal buffer(default scanned size limitation is *16Kb*) using the function *yy_scan_string*. In **Line 3**, *yyparse* function, i.e., a parser function generated by the Bison program, uses the scanner variable and parses the input. If the parser is successful, an alternating array or a linked-list will be returned. The variable *result* will return 1 if successful. The next step is to generate the final output. The first step is to determine the parsed output result; the variable *ret_type* of type *return_type* is used to track the return type of the output. If it is an alternating array, the array is sorted using a merge sort algorithm and returned. Otherwise, the linked-list is converted to an alternating array before sorting and returning.

In the grammar, the production *other* scans additional information provided with the input polynomial, i.e., variables used in the polynomial. This information helps the parser to perform a single pass execution. Otherwise, we need to scan the input polynomial twice, once to discover the variable, then to parse. This is an efficient step that will speed up the serial parser; it does not need to update the array (linked-list or alternating)

[4]Parse-time variables are common values that are modified by the parser throughout the parsing process

variable count during parsing.

Algorithm 1: Bison parser entry point algorithm

```
   Input: An Expanded or Nested String Polynomial input
   Input: A flex scanner yysacn_t type scanner
   Output: An Alternating Array altarr_data
 1 yylex_init(scanner)
 2 yy_scan_string(input, scanner)
 3 result ←yyparse(scanner)
 4 if result then
 5     if ret_type == Alternating_Array then
         mergeSortPolynomial_AA(altarr_data)
         return altarr_data
 6
 7     else
         altarr_data = convert_linkedlist_to_altarr_q(llist_q_data,
         g_num_variables)
         mergeSortPolynomial_AA(altarr_data)
         return altarr_data
 8
 9 else
10     return NULL
```

3.3 Conclusion

Parsing a string input to a BPAS compatible polynomial using Grammar 3.1 is a gradual process. Tokens are reduced from the stack if there is only a matching production rule. If reduced, *poly* production holds a partially or fully parsed polynomial until the entire input string is scanned. During the parsing process, if any action code in the *poly* production rules uses functions from the BPAS helper library, the fully parsed input polynomial data structure is returned as an alternating array. Usually, this happens when the input string is a nested polynomial. Otherwise, a linked-list is generated; a linked-list is an efficient data structure to our parser. Since the BPAS library operates on an alternating array, we convert this linked-list to an alternating array for a final output. It is more efficient and faster to process the input using a linked-list data structure during parsing and convert it to an alternating array for final ourput.

Chapter 4

Parallel parser

In this chapter, we will introduce our parallel parsing algorithm. In Section 4.1, we will have a background introduction to parallel parsing; Section 4.1.1, fork-join control flows data management; and Section 4.1.3, CilkPlus, the extension tool used to parallelize our parser. Also, we will do an in-depth explanation on how the CilkPlus tool works and how it utilizes the fork-join design pattern. In Section 4.2, we will lay out our parallel parser design architecture and explain the difference between our serial and parallel implementations. Our parallel parser is divided into four stages: each stage has tasks executed with a specific algorithm. Stages one and two, Section 4.2.1 and Section 4.2.2, are where we split and organize the input. In stage three, Section 4.2.3, we parse the split input in parallel. Finally, stage four, Section 4.2.4, is where we convert the input polynomial to a BPAS compatible data structure if needed. Each stage has a specific algorithm, we will examine each algorithm in-depth.

4.1 Introduction to parallel parsing

The opportunity for parallel execution of computations strongly depends on the architecture of the execution platform [20, p. 9]. All modern computers support hardware parallelism through at least one of the many parallel features such as multithreaded cores, multicore processors, vector instruction, and graphic engines, etc. [16, p. 1]. In this chapter, our focus is on multicore processors. Our parser will be utilizing all the available resources within a multicore processor at the instruction level to gain significant performance increase [20, p. 9]. This is possible because we can approach practical parallel programming problems with common parallel algorithm design patterns. To parallelize our parser, we will be using the fork-join [16, p. 20] design pattern with the help of a high-level programming extension tool called CilkPlus.

4.1.1 Fork-Join control flow data managment

An algorithm has two fundamental components, tasks, and data. A task operates on data, this creates data dependency. When a data dependency occurs, a task cannot execute before the data it needs is generated by another task [16, p. 39]. Control dependency is another kind of dependency which occurs when an event due to I/O operations results in ordering [16, p. 39]. These control flow dependencies can be managed using task managements such as fork-join design patterns. To characterize the parallelism available in processor types, we have to understand how they combine control flow and data management. Flynn's characterization is a classical categorization that divides parallel processors whether they have multiple flows of control, multiple streams of data, or both. These categories include: Single Instruction Single Data (SISD), a single processor element that has access to a single program and data storage; Single Instruction Multiple Data (SIMD), a single operation or task executes simultaneously on multiple elements of data; and Multiple Instruction Multiple Data (MIMD), where separate stream with own flow control operating on separate data [16, p. 52]. Our parallel parser is categorized under SIMD, where we use divide-and-conquer strategy to parse multiple data inputs on multiple cores and generate a single output as an alternating array data structure.

4.1.2 CilkPlus language extension tool

CilkPlus is a language extension programming tool for C and C++ languages [18]. It is used to express tasks and data parallelism. A divide-and-conquer problem is a befitting candidate for CilkPlus extension; the problem can be divided into parallel independent tasks and the result can be combined afterward. When utilizing the CilkPlus extension to a C code, it is the responsibility of the programmer to structure the program to expose its inherent parallelism [8]. On the other hand, the computational tasks on the parallel processor are scheduled by the CilkPlus runtime system [8].

4.1.3 Dependency graph and fork-join pattern

Data dependency in an algorithm can be represented graphically; dependency graph (DG) is used to illustrate data dependency in an algorithm task. A dependence graph is a set of nodes and edge where the node represents the tasks, and the edge represents the data used by the tasks. When a DG has no cycle, it is called *direct acyclic graph* (DAG) [8]. When representing a parallel computation as a DAG, each node represents the execution of an instruction. Therefore, the DAG of a computation represents a partial ordering of dependencies between instructions in the computation [8].

The fork-join pattern forks control flow into multiple parallel flows that rejoin later [16]. The CilkPlus extension uses the fork-join pattern by generalizing serial calls to a parallel call DAG. It can achieve this by letting code spawn a function instead of calling it [16]. A spawn call is a normal call, except the caller continues calling without waiting for the callee to return. Therefore, forking control flow between caller and callee. Later, to merge the control flow, the caller executes a join operation, i.e., sync, to wait for the callee to return [16].

CilkPlus, therefore, includes the fork-join model task parallelism features, spawn, and sync. *cilk_spawn* is the CilkPlus extension keyword that specifies functions to be executed in parallel with a reminder of the calling function. Similarly, *cilk_sync* keyword specifies all spawned calls to complete before execution continues. These two keywords, *cilk_spawn* and *cilk_sync*, express opportunities (not guaranteed) for parallelism. When a function is *cilk_spawned*, part of the application that runs in parallel is determined by the CilkPlus runtime. Once determined, an efficient work-steal scheduler implements task parallelism [16].

CilkPlus extension uses a work-stealing scheduler that automatically balances the fork-join load. In a work-steal strategy, spawned tasks maintain a double-ended queue. These tasks are added to the back of a queue. When a processor has no work, it steals a task from the front of some random victim processor's queue that is a continuation of the function that spawned a function call [16, p. 218].

4.2 Parallel parsing algorithms explained

When tasks are performed in sequence, one after the other due to data dependency, it is called a serial algorithm [8, p. 7]. On the other hand, when there is data independence, we can use tools such as CilkPlus to create a parallel algorithm that can execute tasks in parallel. When combining these two classifications of an algorithm, we can create a Serial-Parallel Algorithm (SPA). An SPA is one where tasks are grouped in stages such that one or more tasks are executed in parallel and the stages are executed sequentially [8, p. 8]. Our parallel parsing algorithm can be described as a Serial-Parallel Algorithm.

Our serial parser in Chapter 3, generates an alternating array from an input polynomial in two stages. These include parsing, the first major stage with the task to create BPAS compatible data structure; and the second major stage is tasked to sort and convert the result obtained from the first stage. Both stages execute tasks in serial[1]. When parsing in parallel, the number of stages are doubled. The first stage is tasked to discover a splitting position within the input polynomial. In the second stage, these splitting positions are used to build sub-polynomials and the results are stored in a single array where these sub-polynomials are separated by a null terminator. Afterward, in the third stage, the sub-polynomials are parsed in parallel to generate a single linked-list (this depends on the input type as discussed in Section 4.2.3). Finally, the generated linked-list is converted to an alternating array. Algorithm 2 is a top-level parallel parsing function that executes the above four stages mentioned. Initially, in **Line 1** we read the user defined split size from file input using the function getSPlitSize. The split size value defines how many polynomial terms to count from the input file before considering the next character as a split position. It is used to divide the input polynomial into pieces. **Line 2** of Algorithm 2 call combines the first and the second stages of our parallel parser architecture. The SplitInputAndPosition function takes in the input polynomial to be parsed as a file descriptor and the split size result from **Line 1**; the results are a set

[1]serial or sequential are used interchangeably for the same meaning

of data as described in the algorithm description that are required by ParallelMergeSort function. **Line 3** is stage three where we call the ParallelMergeSort function, in this stage we parse the input in parallel. In **Line 4**, depending on the result returned from **Line 3**, if it is a linked-list, it will be converted to an alternating array before the function returns. The function convert is used to convert the linked-list output to an alternating array. The only stage that performs its tasks in parallel is stage three where we invoke the ParallelMergeSort function; in the parsing stage, the parsing function (yyparser from Chapter 3) is spawned on each splitted sub-polynomials in parallel. All the other stages have a simple serial algorithm.

```
Algorithm 2: ParseParallel(f)
f, file descriptor reference; splitSize, split size; identifiers, variables;
identifiercount, number of variables; A, array of split positions;
Altarr, resulting alternating array or null; Linklist, resulting linked-list or null;
P, list of input in a single array separated by a null terminator; s, file size;
si, ei, start and ending index position for A; type, define the return type;
return (Altarr, identifiers, identifierCount)
-----
1 splitsize←getSplitSize(SPLIST_SIZE.txt)
2 A, P, identifiers, identifierCount, s, A.length←SplitInputAndPosition(f, splitSize)
3 (Altarr, Linklist, identifiers, si, ei, type)←ParallelMergeSort(A, P, Altarr, Linklist, si, ei)
4 if type == LINKEDLIST_TYPE then
5     altarr←convert(Linklist, idetnifiers.length)
6     type ← ALTARR_TYPE
7 return (Altarr, identifiers, identifierCount)
```

```
Algorithm 3: SplitInputAndPosition(f, splitSize)
f, file descriptor reference; splitSize, split size; L, linked-list of split positions;
identifiers, variables; identifiercount, number of variables; A, array of split positions; A.length, Array A length
P, list of input in a single array separated by a null terminator; s, file size;
return (A, P, identifiers, identifierCount, s, A.length)
-----
1 s ←getFileSize(f)
2 L, identifiers, identifierscount←GetVariableAndSplitPosition(f, s, splitSize)
3 A ←PositionLinkedListToArray(L)
4 P ←SingleArrayOfNullSeparatedSubPoly(f, s, L)
5 return (A, P, identifiers, identifierCount, s, A.length)
```

Figure 4.1 and Figure 4.2 depicts a general architecture of our serial and parallel parsers. These diagrams resemble a DAG without the box. In these diagrams, a single box depicts a stage where a specific task is executed sequentially or in parallel. Figure 4.1 shows two stages that execute tasks sequentially, tasks depicted with a circle. On the other hand, Figure 4.2 shows multiple stages where tasks are executed in sequence, except in stage three, where tasks forked into multiple nodes for parallel execution. When these forked tasks complete execution, they are then joined to a single node before passing to the next stage.

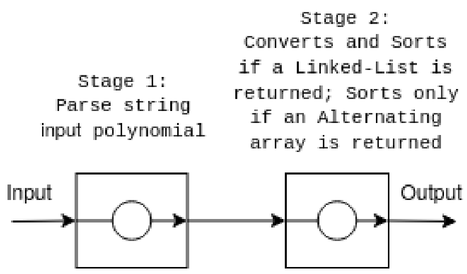


Figure 4.1: Input Polynomials is parsed in serial. Each stage has a single task that is executed sequentially.

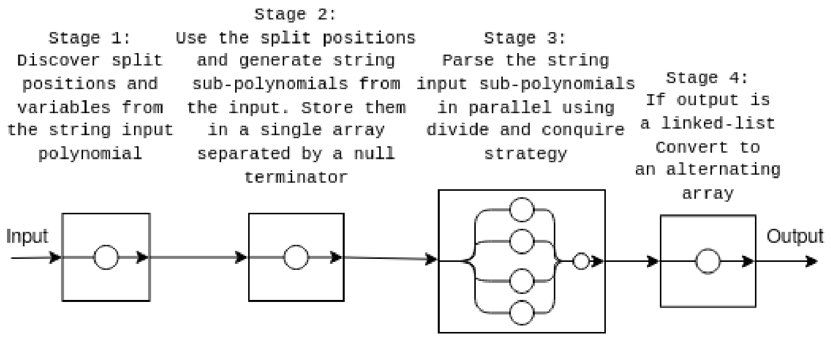


Figure 4.2: Input polynomial is parsed in Serial-Parallel algorithm. Only on stage has its tasks forked to be executed in parallel.

4.2.1 Determining split positions

Our parser accepts input polynomial in two formats, as a single null-terminated character array or as a text file. In Figure 4.2, stage 1, Algorithm 6 performs the task of splitting the input polynomial into smaller pieces with few limitations. The algorithm always expects the input polynomial to contain ASCII (American Standard Code for Information Interchange) characters. When non-ASCII characters are discovered, the input polynomial will be an invalid data, and the parser will exit with an error.

Before explaining our splitting algorithm, it is important to understand the structure of an input polynomial. In Chapter 2, we examined the structure of a polynomial; we said a polynomial is a mathematical function in some variables that is a linear combination of multiplicative combinations of those variables. Also, a polynomial can be considered flat when the polynomials' terms are separated only by + or - characters. On the other hand, it is a nested polynomial when the polynomial is not expanded into its simplest form, i.e., it has one or multiple terms or polynomials that require expanding. Equation 4.1 and 4.2 show an example of a flat and a nested polynomial respectively.

$$5{*}x{*}y\hat{\ }4{+}x\hat{\ }3{+}y\hat{\ }6{*}x\hat{\ }7{+}y\hat{\ }2{*}x\hat{\ }3{*}z\hat{\ }5y{*}x\hat{\ }3{*}z\hat{\ }6{*}w\hat{\ }4{-}7{*}x{*}y{+}.....{+}x{+}y{+}5 \tag{4.1}$$

$$(x{*}(5{*}x{*}y\hat{\ }4{+}x\hat{\ }6{+}y\hat{\ }9{*}x\hat{\ }3{-}7{*}x{*}y)){*}(x\hat{\ }2{+}y\hat{\ }8{*}x\hat{\ }5{-}7{*}x{*}y){+}y\hat{\ }7{*}x\hat{\ }2{-}7{*}x{*}y \tag{4.2}$$

Our strategy to split a flat polynomial is a simple process. Given a *splitSize* n (number of possible polynomial terms to count), we scan through the input polynomial character by character and count the number of + or - character symbols discovered. Every time we discover these character symbols, a possible split position is encountered. The number of characters scanned and the possible split position are counted and saved to variables *count* and d respectively as show in Algorithm 6. When d counts is bigger than *splitSize*, a split position is discovered. *count* tells us the scanner head position, i.e., the character count value used to determine the split positions. When a split position is discovered, the current value of *count*, i.e., the index position of the character, is inserted into a linked-list. The process will continue until all the characters in the input polynomial are scanned and *splitCount*, Equation 4.3, numbers of split index positions are discovered.

$$splitCount = \frac{Number\ of\ terms\ in\ an\ input\ polynomials}{n} \tag{4.3}$$

Algorithm 4: PositionLinkedListToArray(L)
L, linked-list of split positions; A, array of split positions
return A

```
1 A←array(L.length)           // allocate memory for an array of linked-list size
2 n←L.head                     // linked-list head is stored in a variable
  // L.next check if the next node is empty
3 while L.next do
4     A[i]←n.positionStart     // n.positionStart is the data stored in the node
5     n←n.next
6     i++                      // progress the array index position
7 return A
```

Algorithm 5: SingleArrayOfNullSeparatedSubPoly(f, fsize, L)
f, open input file reference; fsize, file size; L, linked-list of split positions;
P, single array of null separated sub-inputs or sub-polynomials;
return P

```
1 P←array(fsize)               // allocate memory for an array of linked-list size
2 n←L.head
3 while L.next do
4     p←read(f, n.positionStart, n.positionEnd)      // read input file from give positions
5     memcpy(P+n.positionStart, p, p.length+1) // memory copy the read data to the exact position
       into array P
6     n←n.next
7 return P
```

On the other hand, if the input polynomial is nested, priority is given to split the nested region. It means, when the input polynomial is nested, part of the polynomial is surrounded by an opening and closing parenthesis. During the scanning process, the first time a left parenthesis is discovered, the numbers of left parenthesis count is incremented and stored into a variable. When a matching right parenthesis discovered, the count value decremented. If the parenthesis count is zero, left and right parenthesis are matched, and a possible split position is gained. When a valid split position is found, its character count position is inserted into a linked-list that collects all the split positions.

So Algorithm 6 carries out two tasks. It computes the split positions and determines all the identifiers (variables) used in the input polynomial. In the algorithm, the top-level ***while*** loop reads a character from the input polynomial and stores it to *curChar* variable for further processing. Also, a character count variable called *count* is incremented every time a new character is read. This two informations, *curChar* and *count*, are dynamically updated and used to determine split positions and identifiers.

Algorithm 6 identify identifiers if they start with non-numeric characters or an underscore followed by one or more characters, underscores, or numbers. The pseudo-code from **Line 4** to **Line 20** discovers a single identifier (variable). When an identifier has more than one character, *curChar* value is bound between the ASCII character values 46 and 57. In addition, the boolean variable *readIdentifer* is set to true if we are scanning part of an identifier. If non-numeric characters are scanned for the first time and *readIdentifer* is false as shown in **Line 8** to **Line 14**, part of an identifier are temporarily gathered in an array called *temporaryVAR*. *temporaryVar* array reserves index 0 to save identifier length assuming that the variable-length does not exceed 255 characters. If numeric characters are scanned as part of an identifier, **Line 4** to **Line 7** are executed, this means *readIdentifier* is set to true. When all the above conditions are not satisfied, **Line 16** to **Line 20** checks the variable in *temporaryVar* for duplication against the other list of identifiers in an *identifiers* array. *identifiers* contain a unique list of variables discovered in the input polynomial ordered as first come first serve.

When left or right parenthesis is scanned by the scanner, part of the pseudo-code in **Line 21** to **Line 58** of Algorithm 6 are executed. Specifically, if *curChar* contains a left parenthesis, **Line 21** is satisfied. This prompts the *countBracket* (parenthesis count variable) to increment by one. *countBracket* is used to keep track of the number of left parentheses discovered; *countBracket* is decremented when previously discovered left parenthesis match with the right parenthesis. When the value of *countBracket* is zero, all the left parenthesis have a matching right parenthesis, and if the next character scanned is not an asterix (multiplication symbol), a possible splitting position is discovered. Algorithm 6 stores split position into the linked-list L. It is because the linked-list is an efficient data structure compared to an array, i.e., unlike arrays, linked-list can insert elements dynamically without modifying its size.

When the input is a flat polynomial candidate, the pseudo-code in **Line 59** to **Line 71** of Algorithm 6 are executed. In a flat polynomial, when + or - character symbols are scanned, it means a polynomial term is discovered. We keep track of input polynomial term counts in variable d; if the value in d exceeds the *splitSize* value, it is inserted into linked-list L as a split position.

After the input polynomial is fully scanned and the split positions, and its identifiers are discovered, the algorithm returns the following sets of values: a linked-list of split positions, L; a unique sets of variables used in the input polynomial, *identifiers*; and the length of *identifiers* array, *identifierCount*.

4.2.2 Organizing input polynomial for parsing

In Stage 2, we invoke the functions *PositionLinkedListToArray* and *SingleArrayOfNullSeparatedSubPoly* as shown in Algorithm 3 **Line 3** to **Line 4**. These two functions use data passed from *GetVariableAndSplitPosition* in **Line 2** and efficiently structure them before adapting it for our parallel merge sort algorithm. Algorithm 4, *PositionLinkedListToArray*, is a simple algorithm that converts a linked-list of split positions in L to an array. Initially, the Split positions are stored in a linked-list, each node containing the start and end index position of a sub-polynomial. The function *PositionLinkedListToArray* converts this linked-list L to an array with each element only containing the start splitting index position. It makes sense because the ending splitting index position will be obsolete when the input polynomial is divided into smaller chunks, with each sub-polynomials stored in a single array separated by a null terminator. This means, anytime we want to access a specific sub-polynomial we position the reader head to the start index position and read until the null terminator is reached. In addition, navigating to a specific split position in an array data structure is more efficient than a linked-list. In stage 1, *GetVariableAndSplitPosition* benefitted from the dynamic nature of a linked-list data structure to efficiently insert elements without updating the list size. In stage 2, *PositionLinkedListToArray* convert this list to an array so that we can utilize this split positions data efficiently in our parallel *MergeSort* algorithm.

As stated in the previous paragraph, Algorithm 5, *SingleArrayOfNullSeparatedSubPoly*, structures and stores the input polynomial into a single array. If the input polynomial is a text file, accessing sub-polynomials requires moving around a file cursor to the right position before reading. It is an inefficient process. Once the input polynomial splitted into smaller chunks, we can efficiently access the sub-polynomials when they are organized in a single array that are separated by a null terminator. For example, if we want to access split position m where the sub-polynomial starts, we can change the array A reference position by add m to it and point text reader to read until the first null terminator is encountered. It significantly cut down the access time when extracting sub-polynomial from an input polynomial during parsing.

4.2.3 Parsing input polynomial in parallel

Stage 3 is where we parse the input polynomial in parallel. Algorithm 7, *ParallelMergeSort*, resolve the task using a merge sort algorithm to parse them in parallel with the help of the CilkPlus extension tool. Once the pieces of polynomials are parsed, Algorithm 8 will merge them.

Algorithm 6: GetVariablesAndSplitPosition(f, s, splitSize)
f, open input file reference, s, file size, splitSize, split size
L, linked-list of split positions, identifiers, polynomial variables, identifierCount, number of variables
return (L, identifiers, identifierCount)

```
While not f do {
    curChar←readChar(f)
    count++
    if curChar > 47 and curChar < 58 then
        if readingIdentifier then
            temporaryVar[0]←temporaryVar[0] + 1
            temporaryVar[temporaryVar[0] - 48]←curChar
    else if (curChar > 64 and curChar < 91)or curChar == 95 or (curChar >
     95 and curChar < 123) then
        if not readingIdentifier then
            temporaryVar←allocateMemory()
            temporaryVar[0]←'0'
            readingIdentifier←1
        temporaryVar[0]←temporaryVar[0] + 1
        temporaryVar[temporaryVar[0] - 48]←curChar
    else
        if readingIdentifier then
            if not identifierMatch(identifiers, temporaryVar) then
                identifierPosition++
                identifierCount++
                identifiers[identifierPosition]←temporaryVar
    if curChar == '(' then
        countBrackets++
        if readingIdentifier then
            if not identifierMatch(identifiers, temporaryVar) then
                identifierPosition++
                identifierCount++
                identifiers[identifierPosition]←temporaryVar
```

```
if curChar←')' then
    countBrackets–;
    if readingIdentifier then
        if not identifierMatch(identifiers, temporaryVar) then
            identifierPosition++;
            identifierCount++;
            identifiers[identifierPosition]←temporaryVar;
        readingIdentifier←0;
    if countBrackets == 0 then
        curChar←readChar(f);
        count++;
        while not f and curChar == ' ' and curChar != '(' do
            curChar←readChar(f);
            count++;
            if readingIdentifier then
                if not identifierMatch(identifiers, temporaryVar) then
                    identifierPosition++;
                    identifierCount++;
                    identifiers[identifierPosition]←temporaryVar;
                readingIdentifier←0;
        if curChar != '*' then
            if readingIdentifier then
                if not identifierMatch(identifiers, temporaryVar) then
                    identifierPosition++;
                    identifierCount++;
                    identifiers[identifierPosition]←temporaryVar;
            if L.tail == NULL then
                n←createNode(positionStart, positionEnd);
            if n != NULL then
                insertNode(L, n);
            d←0;
    if (curChar == '-' or curChar == '+')and (curChar == 0 or curChar ==
     s) then
        d++;
        if readingIdentifier then
            if not identifierMatch(identifiers, temporaryVar) then
                identifierPosition++;
                identifierCount++;
                identifiers[identifierPosition]←temporaryVar;
        if (d > splitSize or count == s) then
            if L.tail == NULL then
                n←createNode(positionStart, positionEnd);
            if n != NULL then
                insertNode(L, n);
            d←0;
}                                                    ▷ End of while loop
for i←0 to identifierCount do
    if L.tail == NULL then
        variableSize←GETINT(identifier[k][0]);
        memcpy(temporary, identifiers[k]+1, variableSize)
         identifiers[k]←temporary
return L, identifiers, identifierCount;
```

The base case in *ParallelMergeSort*, **Line 2** to **Line 7** is where the actual parsing takes effect. To satisfy the condition in the base case, the list of split position A is divided into smaller units so that when index e and s, i.e., start and end index position of A, point to the same element. This index element is used to access the split position from array A and retrieve the sub-polynomial from P for parsing. The sub-polynomial is then parsed using the same serial parsing algorithm described in Chapter 3. Once parsed, the conditional pseudo-code in the base case checks the return type of the parsed sub-polynomial. The return type can be *ALTARRAY_TYPE* or *LINKEDLIST_TYPE* depending upon the input polynomial. The variable *type* holds a return type, which is part of the set of values returned by this algorithm. On the other hand, if the split position array A has more than one element, the function *ParallelMergeSortit* is called recursively until the base case is satisfied. During the recursive call, the CilkPlus extension tool used to spawn multiple calls of *ParallelMergeSort* function in parallel to take advantage of multicore processors. Once the spawned calls are returned, Algorithm 8, *Merge*, merges the final output.

One of the return values from *ParallelMergeSort*, *type*, is used to identify the types of the left and right sub-polynomial being merged to avoid type conflict. The conditional statements in **Line 1** to **Line 14** in Algorithm 8, confirms if the sub-polynomials have matching types. If both the left and right spawned function in *ParallelMergeSort* return an alternating array, their type is a match, and *addPolynomials_AA* function is used to merge the two polynomials. If one output is a linked-list and the other one is an alternating array, the linked-list output is converted to an alternating array before merging. For this reason, the return type of *Merge* depends upon the sub-polynomials to merge, i.e., it could be an alternating array or a linked-list. Eventually, the function returns with five sets of values: an alternating array, *Altarr*; a linked-list, *Linkedlist*; variables used in the parsed polynomial, *identifiers*; and the return type, *type*. Depending on the return type, *Altarr* or *Linkedlist* is set to null.

4.2.4 Converting output

Stage 4 of the parallel parsing process has a simple task. Depending upon the final result of the *ParallelMergeSort* algorithm, if a linked-list is returned, it converts it to an alternating array.

Algorithm 7: ParallelMergeSort(A, P, Altarr, Linklist, s, e)
A, array of split positions; P, null separated sub-input; Altarr, generated an alternating array;
Linkedlist, generated a linked-list; identifiers, variables used in the input polynomials to be parsed;
s, e, start and ending index position for A; type, define the return type;
return (Altarr, Linklist, identifiers, s, e, type)

```
1  if e == s then
2      tempInput = P + A[s]
3      returnType←parse(tempInput)          ▷ Chapter 2 parsing algorithm is used
4      if returnType == ALTARR_TYPE then
5          type←ALTARR_TYPE
6      else
7          type←LINKEDLIST_TYPE
8      return (Altarr, Linkedlist, s, e, type)
9  mid←floor(s + e)/2
10 ls←s, le←mid, rs←mid+1, re←e
11 (lAltarr, lLinkedlist, ls, le, ltype) ← cilk_spawn ParallelMergeSort(A, P, lAltarr, lLinkedlist, ls, le)
12 (rAltarr, rLinkedlist, rs, re, rtype) ← ParallelMergeSort(A, P, rAltarr, rLinkedlist, rs, re)
13 cilk_sync
14 return merge(A, P, identifiers, (lAltarr, lLinkedlist, ls, le, ltype), (rAltarr, rLinkedlist, rs, re, rtype))
```

Algorithm 8: Merge(A, P, identifiers, (lAltarr, lLinkedlist, ls, le, ltype), (rAltarr, rLinkedlist, rs, re, rtype))
(lAltarr, lLinkedlist, ls, le, ltype), left half of the list to be merged
(rAltarr, rLinkedlist, rs, re, rtype), right half of the list to be merged
A, array of split positions; P, null separated sub-input; Altarr, generated an alternating array;
Linkedlist, generated a linked-list; identifiers, variables used in the input polynomials to be parsed;
s, e, start and ending index position for A; type, define the return type;
return (Altarr, Linklist, identifiers, s, e, type)

```
1  if ltype == ALTARRY_TYPE and rtype == ALTARR_TYPE then
2      Altarr ← addPolynomials_AA(lAltarr, rAltarr, identifiers.length)
3      type ← ALTARR_TYPE
4  else if ltype == LINKEDLIST_TYPE and rtype == LINKEDLIST_TYPE then
5      Linkedlist ← linked_list_cat_q(lLinkedList, rLinkedlist)
6      Linkedlist ← LinkedListMergeSort(Linkedlist, identifiers.length)
7      type ← LINKEDLIST_TYPE
8  else if ltype == ALTARRY_TYPE and rtype == LINKEDLIST_TYPE then
9      altarr ← convert(rLinkedlist, idetnifiers.length)
10     Altarr ← addPolynomials_AA(lAltarr, altarr, identifiers.length)
11     type ← ALTARR_TYPE
12 else if ltype == LINKEDLIST_TYPE and rtype == ALTARR_TYPE then
13     altarr ← convert(lLinkedlist, idetnifiers.length)
14     Altarr ← addPolynomials_AA(altarr, rAltarr, identifiers.length)
15     type ← ALTARR_TYPE
16 return (Altarr, Linklist, identifiers, s, e, type)
```

4.3 Conclusion

When parsing in parallel, the first two stages of the parsing process includes: splitting the input polynomial into smaller peices and organizing them to an efficient data structure.

Split positions are determined by scanning input polynomials character by character until the index count value matches the predefined split size. Then these index positions are collected to an array for further processing. The split size counts can be determined by averaging input polynomial character term count with user defined split size input. After the split positions are determined, the input polynomial is divided into sub-polynomials and stored into a single array separated by a null terminator. Organizing the input polynomial becomes important when accessing sub-polynomial during parallel parsing. These sub-polynomials parsed using a merge sort algorithm, and the final result converted to an alternating array.

Chapter 5

Experimentation

In this chapter, we present experimental data for our parser. We will compare the performance of our parser over a rational number polynomials using the Maple program as a comparison point. For these experiments, we will use a randomly generated, flat[1] (expanded) sparse polynomials and nested dense polynomials. The nested dense polynomial is computation-intensive; it will utilize BPAS polynomial arithmetic functions for a nested polynomial multiplication operation. The flat sparse polynomial does not require additional resources from the BPAS library; our parsing library will independently generate the final output, i.e., alternating array, without invoking any external auxiliary functions. The experimental data collected includes memory consumption; and running time for splitting, parsing, and converting. The experimentation is conducted both in our serial and parallel algorithms. All the experimental results are verified using the Maple program.

We will discuss single-pass and multi-pass parsing (Section 5.1); multi-pass parsing understanding is crucial before discussing the experimental data. Next, we will hav an in-depth discussion on the experimental results for expanded sparse multivariate polynomials (Section 5.2); we will analyze the data gathered for serial and parallel implementations. Lastly, we will analyze experimental data for nested dense multivariate polynomials (Section 5.3); like the previous section, we compare and contrast data gathered for serial and parallel implementations and draw our final conclusion.

5.1 Overview

The original goal of our research was to create a parser that parses polynomials in a single pass—passing through the input polynomial only once to generate the output data structure, i.e., alternating array. With the use of front end compiler tools such as Bison and Flex, it is possible to create a single-pass parser for a univariate polynomial. In a multivariate sparse polynomial, it is a different story. When parsing a multivariate sparse polynomial, to maintain the single-pass process, we need information about exponent

[1]flat and expanded terms will be used interchangeably in this chapter to refer to expanded sparse polynomial

variables (exponent vectors and their size). The information becomes vital to allocate the correct memory space for our output data structure. It means, during parsing, when new variables are discovered, we have to visit previously generated terms to introduce the new variable by reallocating additional memory space. To accomplish this, we have to access and modify each polynomial term that resides in the memory stack. For this reason, with multivariate polynomial inputs, backtracking the stack memory to update with additional information is a must.

When part of the input polynomial is parsed and in the porcess if a new variable is discovered, the vector exponents in the previously parsed section become incomplete. Therefore, each polynomial term is revisited, and a bigger memory space is allocated to accommodate the new variable. As the polynomial gets larger, this step significantly slows down the parsing process. To alleviate this problem, we introduced an additional production to our bison grammar. Each input polynomial needs to provide information about variables; this new input polynomial structure includes all variables used as a tuple followed by the actual polynomial. Though the suggested step only resolves the issue for the short term, it introduces a new question; what will happen when the input does not provide this variables information? For this reason, we concluded that single-pass parsing works if we have enough information about the polynomial, otherwise, to generate an error-free output, we have to revisit a previously parsed portion of the polynomial terms multiple times and update the exponent vectors.

As mentioned in the previous paragraph, reallocating memory for the previously parsed portion of the input polynomial consumes time and CPU resources (reallocating a memory takes CPU time). Therefore, there are efficiency concerns when considering a single-pass parser with a multivariate polynomial input. One way to resolve this issue is to perform multi-pass parsing; the first pass runs through the input polynomial to discover all the variables, and the second pass does the actual parsing to generate the data structure (alternating array). It is an efficient way of running a multi-pass parser because the algorithm we used to discover all the variables in the input polynomial, Algorithm 3, is significantly faster than the single-pass parsing process, a process mentioned in the previous paragraph (revisiting and updating exponent vectors).

In this experiment, we collected data only for the algorithms that utilize a multi-pass parsing strategy.

5.2 Experimentation for expanded sparse multivariate polynomial

Throughout all benchmarks presented in this section, our benchmarks were collected on a machine with an Intel Xeon X560 processor at 2.67 GHz, 32KB L1 data cache, 256KB L2 cache, 12288KB L3 cache, and 48GB of RAM. The total number of hardware threads available in this machine is 24. As mentioned in the introduction paragraph, our experimentation used nested dense and sparse multivariate polynomials. Nested dense polynomials are generated using the following parameters: number of variables, v; degree's, d; number of nested polynomial arithmetic operations, m (Equation (5.2)); and randomly generated rational number coefficient with a maximum of five-digit integers for both numerator and denominator. These nested dense polynomial data are generated using Maple script; the source code is available in Appendix A Listing A.1. In this Maple script, the function *PolyC* in Listing A.1 invokes *PolyA_No_terms* m numbers of iterations. On the other hand, *PolyA_No_terms* returns a randomly generated polynomial that is converted to a string using the maple commands ***randpoly*** and ***convert***, respectively. Expanded sparse polynomial testing data are generated using the function in Listing A.2, Appendix A. In this function the following parameters are used: number of variables, *num_vars*; list of variables, *variables*; number of terms to generate, *num_terms*; maximum exponent value, *max_degs_range*; and value used in *GNU GMP* random class to randomize the coefficient values, *bits*. Finally, benchmarks are collected for both serial and parallel versions of the algorithms.

5.2.1 Serial and parallel implementations

We begin by comparing the running time and memory usage of pure serial parser against a single-threaded parallel implementation. As described in Chapter 3, our serial parser accepts a list of identifiers (variables) used as part of the input polynomial. If these lists of identifiers are defined early, our serial parser can perform a single pass parsing. Table 5.1 shows run time and memory usage comparison of an input polynomial with or without identifiers defined. There is an eightfold decrease in run time when parsing a randomly generated sparse polynomial with identifiers defined. It is because the pure serial parser spends less time backtracking for an update. Similarly, memory usage goes down by half when identifiers are defined; with the identifiers' information, a pure serial parser allocates the exact memory required before it scans the next polynomial term. It is not always the case though users can provide input without overhead information about identifiers. The most efficient way to overcome this issue is by utilizing multiple passes on the input polynomial, i.e., first, discover identifiers, and then apply the parsing algorithm. The splitting algorithm is responsible for discovering identifiers as discussed in Chapter 4. Comparing run times of pure serial implementation against single-threaded parallel implementation, when identifiers are defined, the data is almost comparable, as shown in Table 5.1 and Table 5.2. It is because the split algorithm used in our parallel implementation discovers all the variables faster than the pure serial parser that requires backtracking for a variable update when input is provided without identifiers.

To further our experimentation, the effect of splitting and parallelizing expanded sparse polynomials have on our parsing process we compare the running time for serial and parallel parsing algorithms using Maple as a comparison point. Table 5.2 shows the running time and memory usage data for our parallel parser and Maple program. Our single-threaded parallel parser run time is significantly faster than the Maple program

		Time(s)		Memory(MB)	
n	Size(MB)	Defined	Not Defined	Defined	Not Defined
1000	0.021	0.001	0.013	3	7
5000	0.101	0.015	0.136	4	8
10000	0.225	0.031	0.285	6	13
100000	2.3	0.338	3.04	45	91
500000	13	1.34	12.0	223	447
1000000	26	2.80	25.2	440	880
1500000	39	4.18	37.6	601	1202
2000000	53	5.47	49.3	726	1452
2500000	75	6.49	58.4	1014	2028
3000000	90	8.69	78.2	1196	2392
3500000	105	10.2	91.9	1311	2622
4000000	147	11.8	107	1825	3650
4500000	206	13.9	125	2124	4249

Table 5.1: Comparing running time and memory usage for randomly generated sparse polynomial on a pure serial implementation with or without identifiers/varialbles defined.

because input data is organized into smaller pieces and passed to a parser that uses a divide and conquers strategy. Table 5.3, data gathered for parallel parser with two threads, breakdown the running time spent at each stage. Run time significantly change during the parsing process as shown in column Parse(s). In this particular case, total run time in Table 5.3 and serial column in Table 5.2 are almost identical. This is because to see a speed-up in the parsing process, first, more worker threads are needed, and second, the input polynomial must split at least twice the number of worker threads, i.e., the *splitCount*, Chapter 4. This conclusion is reached after carefully optimizing the split size (*splitSize*) input value provided by the user. Simply, the number of ways the input polynomial split needs to be greater than the selected hardware thread value. When there is more work for the worker threads, the CilkPlus runtime library work-steal scheduler efficiently steals work for idle worker threads.

Returning to Table 5.2, comparing the running time for a flat sparse polynomial input in a single-threaded parallel parser with Maple program, there is a significant running time improvement with our parallel parser when the number of terms n increases. The data shows when tasks are organized into separate stages and executed sequentially instead of being promoted as part one large task, the running time of the overall process improves. Figure 5.1 shows the effect on running time as the number worker threads increase when parsing flat sparse polynomials between our serial and parallel implementation and Maple program. When the hardware threads used are fixed between 4 and 8, we observe a significant speedup. Since the experimentation was conducted on a 12 core CPU, the threshold to achieve genuine parallelization is under eight hardware threads. Over this threshold, the scale of the plot shows a slight speedup because it utilizes Intel's hyperthreading technology. Besides, looking at the plot for serial and Maple data, as the input size increases (number of terms n), Maple program run time shows a steep inclination. On the other hand, our serial implementation shows a gradual increase. It means the Maple program does not utilize multi-core programming when parsing.

		Time(s)		Memory(MB)	
n	Size(MB)	Maple	Serial(t=1)	Maple	Serial
1000	0.021	0.018	0.008	1	4
5000	0.101	0.056	0.022	8	5
10000	0.225	0.103	0.041	16	8
100000	2.3	5.41	0.438	160	49
500000	13	18.5	2.20	800	226
1000000	26	34.0	4.20	1600	449
1500000	39	46.7	6.02	2400	672
2000000	53	59.6	7.99	3200	894
2500000	75	75.7	10.69	4000	1135
3000000	90	83.8	12.6	4800	1360
3500000	105	108	14.8	5600	1587
4000000	147	145	19.5	7200	2064
4500000	206	155	21.8	8000	2291

Table 5.2: Comparing running time and memory usage for randomly generated sparse polynomials on Maple program and single threaded parallel implementation.

n	Size(MB)	Total(s)	Split(s)	Parse(s)	Convert(s)	Memory(MB)	Threads(t)
1000	0.021	0.016	0.001	0.014	0.0005	4	2
5000	0.101	0.023	0.005	0.015	0.002	5	2
10000	0.225	0.043	0.008	0.030	0.004	8	2
100000	2.3	0.427	0.087	0.291	0.048	47	2
500000	13	2.20	0.463	1.458	0.287	227	2
1000000	26	4.31	0.939	2.870	0.509	450	2
1500000	39	6.67	1.56	4.275	0.840	671	2
2000000	53	8.78	1.90	5.747	1.141	896	2
2500000	75	10.48	2.49	6.528	1.467	1135	2
3000000	90	13.4	2.98	8.885	1.578	1361	2
3500000	105	15.7	3.38	10.256	2.096	1586	2
4000000	147	19.9	4.74	12.699	2.530	2063	2
4500000	206	22.0	5.06	14.026	2.984	2293	2

Table 5.3: Running time for Randomly generated sparse polynomial on our parallel algorithm.

Hardware Threads				
Speedup($\frac{T_1}{T_p}$)	t=2	t=4	t=8	t=16
	1.09	1.43	2	2.2

Table 5.4: Comparing Figure 5.1 speedup, a randomly generated expanded sparse polynomial.

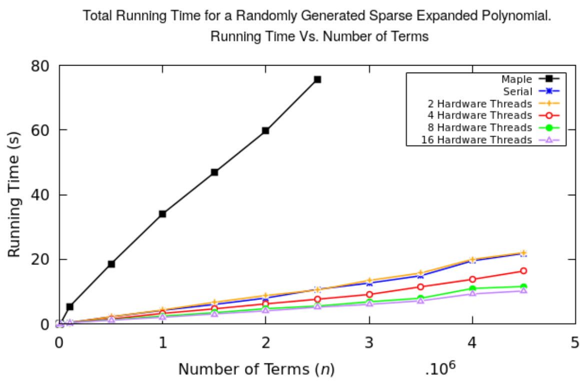


Figure 5.1: Comparing run time of expanded sparse polynomial parsed in serial and parallel implementation against Maple program. The number of terms varies on the x-axis, while the algorithm and hardware threads vary as noted in the legend.

5.2.2 Experimental data for splitting, parsing, and converting

When parsing a flat sparse multivariate polynomial using our parallel algorithm, four main steps are involved: splitting, organizing, parsing, and converting, as discussed in Chapter 4. These are the main steps performed to efficiently generate the BPAS data structure (alternating array) in parallel. In Figure 5.2, we compare run time against a number of terms n changes at each step using a flat sparse rational multivariate polynomial at different hardware threads. The splitting process performed over n number of terms; Figure 5.2 (a) shows that run time is the same across different hardware threads because splitting is performed in serial. Depending upon the final output, i.e., linked-list or alternating array, converting is also a serial process. A detailed description of why we need converting on the final output polynomial is available in Chapter 4. Run time performance in this final converting process again is the same across different hardware threads, Figure 5.2 (c). On the contrary, during the parsing step, we see a gradual incline in run time performance across different hardware threads shown in Figure 5.2 (b). The parsing step is where the majority of the work is done in our parallel algorithm; the *MergeSort* function, Chapter 4, invoked to divide the work among different threads. As the number of worker threads increases, the run time of our parser decreases. Therefore, the splitting and sorting steps are considered as pre and post parsing steps, respectively; this is because they always maintain the same run time across different hardware threads. The total time is the run time accumulation of all these three steps; choosing the right split size and hardware threads affect the run time performance in parallel parsing.

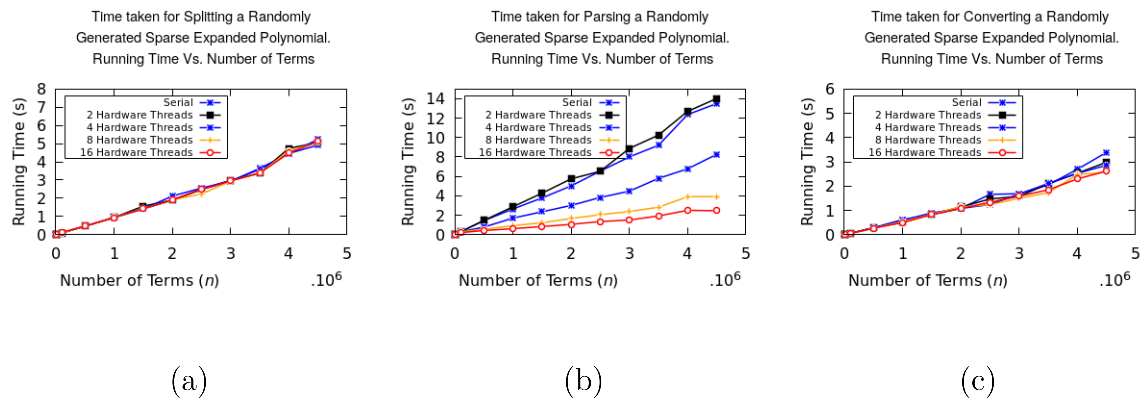


Figure 5.2: Comparing three major parsing steps involved in our parallel algorithm.The randomly generated sparse multivariate polynomial result is compared on a serial or single threaded and multithreaded algorithm. Table (a) and Table (c) show the pre and post parsing steps, splitting and converting respectively, of the parsing process compared across multiple threads. Table (b) shows the parsing process across multiple threads.

5.2.3 Memory

There is a remarkable difference in memory usage between the Maple program and our parser shown in Table 5.5. When parsing a randomly generated flat sparse polynomial, our algorithm implementation uses five times less memory to perform the same operation. It is because the linked-list data structure is very efficient to manage our partially parsed input during parsing, i.e., the Maple program is using a simple array or an alternating array. Similarly, Across the various fixed hardware threads used, memory consumption stayed consistent.

5.2.4 Experimentation with large-scale expanded polynomial datasets

Based on our experience running a small-scale operation, we conducted a run time analysis on large-scale datasets. These datasets, as shown in Table 5.6, range from half a gigabyte to two gigabytes. Run time analysis shows our implementation is scalable and sustains speedup with multicore processors and large-scale datasets.

		Memory(MB)					
n	Size(MB)	MAPLE	Serial	t=2	t=4	t=8	t=16
1000	0.021	1.71	4.30	4.49	4.34	4.28	4.62
5000	0.101	8.12	5.98	5.86	6.20	6.17	6.44
10000	0.225	16.1	8.24	8.16	8.49	8.44	8.49
100000	2.3	160	49.2	47.9	49.4	48.2	50.5
500000	13	800	226	227	229	228	226
1000000	26	1600	449	450	453	458	454
1500000	39	2400	672	671	673	679	687
2000000	53	3200	894	896	898	904	913
2500000	75	4000	1135	1135	1137	1143	1156
3000000	90	4800	1360	1361	1364	1370	1381
3500000	105	5600	1587	1586	1589	1595	1606
4000000	147	7200	2064	2063	2067	2073	2086
4500000	206	8000	2291	2293	2296	2302	2315

Table 5.5: Comparing randomly generated expanded sparse polynomial memory usage in our parallel parser implementation across different threads using Maple program as a reference point.

		Run Time(sec)					
n	Size(MB)	MAPLE	t=1	t=2	t=4	t=8	t=16
10000000	667	3175	152	106	88.8	79.2	85.7
20000000	1400	11874	364	258	193	182	182
30000000	2000	14470	609	409	346	313	296

Table 5.6: Comparing randomly generated expanded sparse polynomial runtime in our parallel parser implementation across different threads using Maple program as a reference point. These expanded sparse polynomial file size ranges in gigabytes.

Hardware Threads				
Speedup($\frac{T_1}{T_p}$)	t=2	t=4	t=8	t=16
	1.5	1.8	2	2.1

Table 5.7: Comparing Table 5.6 speedup a randomly generated expanded sparse polynomial.

5.3 Experimentation for nested dense multivariate polynomials

Since Maple has become the leader in a polynomial arithmetic operation, it is important to assume that they are accompanied by efficient auxiliary functions for parsing. In the previous section, we compared experimental results for a flat sparse polynomial on our serial and parallel algorithm implementations using Maple as a comparison point. During this experimentation, we used a Maple command called ***parse*** to parse flat sparse polynomials. It is clear from our experimental result, Maple program does not utilize parallelization during parsing. Again, we will use the same Maple command, ***parse*** to experiment on nested dense multivariate polynomials. To generate this experimental nested dense multivariate polynomials, two input polynomials f and g are randomly generated with the following parameters: degree's, d; number of polynomial terms, n; and number of nested arithmetic operation (multiplication), m. These parameters are used to create un-expanded nested dense multivariate polynomials as shown in Equation (5.2). Since these polynomials are generated using Maple script, shown in Appendix A Listing A.1, the number of polynomial terms, n changes depending on the number of variable v used.

$$f_{\mathrm{i}}, g_{\mathrm{j}} \in \mathbf{Q}[x_1, \ldots, x_{\mathrm{v}}] \tag{5.1}$$

.

$$h = (f_1 * g_1) + \cdots + (f_{\mathrm{m}} * g_{\mathrm{m}}) \tag{5.2}$$

When testing these experimental data in the Maple program, in addition to the ***parse*** command, to expand this nested polynomial, we used another Maple command called ***expand***. As the name suggests, the expand command distributes products over sums [14]. Expanding Equation (5.2) is a computationally-intensive process. It requires resolving the product of polynomials *(f)* and *(g)* before invoking the ***parse*** command. Looking at the data in Table 5.9, Maple's ***expand*** command runs slower. As the nested dense multivariate polynomial degree d increases, Maple program run time performance will increase exponentially i.e., as degree d increases the size of the dense polynomials f and g grows. Eventually, it exhausts memory and crashes. For this reason and to have a concret data for comparison, we kept our degree d value constant. Our parsing algorithm implementation, on the other hand, handles this efficiently by distributing this nested operation across multiple hardware threads. It is important to note that during the splitting step, our split algorithm prioritizes to split the nested region over user-defined split size as discussed in Chapter 4. Once these nested polynomial arithmetic operations are split, they are distributed across multiple threads for computation. It gives our parallel parser algorithm a significant speedup to parse large input polynomials compared to Maple program.

5.3.1 Run time analysis for maple and BPAS implementations

Just like our flat sparse polynomial experimentation, we performed benchmarking for nested dense multivariate polynomials on our parallel algorithm implementation using Maple as a comparison point. In this experiment, when generating the nested dense multivariate polynomials, we applied a fixed number of degree d=4, and varying numbers of variables *(v)* and nested arithmetic operation *(m)*.

Table 5.9 shows the total run time taken in seconds when parsing a randomly generated nested dense polynomial between our parallel algorithm implementation and Maple program. If the variable count in the nested polynomials *(f)* and *(g)* from Equation (5.2) increases, the polynomial grows in size. When these nested polynomials size grows, the number of nested arithmetic operations *(m)* performed, i.e., multiplication operations, as shown in Equation (5.2), become computationally intensive and memory hungry. In Maple program, though the ***expand*** command performs parallel operation, the ***parse*** commands face a big challenge. Therefore, when analyzing the run time data from Table 5.9, as the variable counts(v) increase, the run time performance of the Maple program drops. Whereas for our serial and parallel algorithm implementation, the performance improves both with increased variable count and worker threads. As illustrated in Figure 5.3, when the number of variables *(v)* increases, Maple program run time shows a steep incline. This shows that as the polynomial becomes larger and nested, arithmetic operation m increases and the resulting polynomials become larger. As a result the time taken to perform these operations combined with the time it takes to ***parse*** causes the Maple program total run time to grow exponentially. On the other hand, in our parallel algorithm implementation, there is a significant speedup as the number of hardware threads increases. It is because as the input is divided across this nested arithmetic operation, multiple worker threads take the task of parsing and solving arithmetic operations in parallel. As shown in Table 5.9, when the hardware thread *(t)*=8, it has become the threshold for genuine parallelization; at this threshold point speedup doubles.

The running time taken by our pure serial parser shows poor performance as the variable count and number of iteration increases. Table 5.8 shows a total run time comparison between input with or without variable information. It shows that the Maple program runs time performance in Table 5.9, outperforming our pure serial parser when input data does not include variable information. This is because our parser relies on BPAS Auxilary functions for arithmetic operations, this function is slightly slower than Maple command ***expand***. Besides, our serial parser performs multiple passes on the parsed data structure to update newly discovered variables.

5.3.2 Memory consumption on maple and BPAS

Memory usage between the Maple program and our parallel algorithm implementation when parsing a nested dense multivariate polynomial is not as promising as our first experimentations shown in Table 5.10. When parsing a nested dense multivariate polynomial,

			Time(s)		Memory(MB)	
v	m	Size(MB)	Not Defined	Defined	Not Defined	Defined
4	16	0.038	3.12	1.99	38.3	23.028
	32	0.078	7.14	4.57	61.2	36.7
	64	0.156	15.6	10.0	106	63.8
	128	0.621	35.3	22.6	203	122
6	16	0.128	27.5	17.6	123	61.8
	32	0.256	61.1	39.1	254	127
	64	0.511	136	87.3	530	265
	128	1.2	294	188	833	555
8	16	0.321	144	92.2	415	277
	32	0.641	323	206	902	601
	64	1.3	2186	932	1956	1304
	128	2.6	4922	2100	4156	2771

Table 5.8: Run time dataset and memory useage for parsing Nested Dense Polynomial in pure serial parser with/without variable information defined followed by the input polynomial.

expansion is done by calling BPAS auxiliary function ***multiplyPolynomials_AA***. In Chapter 3, we have seen how our parser alternates between linked-list and alternating array during parsing. When two or more polynomial expansion is required, and this helper function invoked, the result is an alternating array. Once our parser semantic value is an alternating array, the remainder of the data structure used during the parsing process will be an alternating array. During this time, our parser memory efficiency is highly dependent on this helper function. As we can see in Table 5.10, across different hardware threads, memory consumption is the same. When comparing these columns against the Maple program, our implementation memory usage was cut by one-third. It shows that an alternating array data structure is not as efficient as linked-list when it comes to parsing.

5.3.3 Run time analysis for splitting and parsing

Splitting a nested dense multivariate polynomial is different from the sparse multivariate polynomial. As discussed Chapter 4, when selecting split positions, our algorithm gives priority to opening and closing parenthesis. Specifically, the splitting process is not restricted by split size d when a left parenthesis is scanned. Therefore, just like our sparse mutlivariate polynomials experimentations, the splitting process run time show similarities across different columns in Table 5.12 is because it is executed in serial. In the parsing step, a fluctuating run time performance observed, as shown in Table 5.12. As the hardware threads increase, there is a speedup. A significant speedup gained when the number of worker threads matches the actual CPU cores. It means pure parallelization is obtained by dividing each nested arithmetic operation amongst the worker threads at the same time.

			Total Time(s)						
v	m	Size(MB)	MAPLE	t=1	t=2	t=4	t=8	t=16	t=24
	16	0.038	0.238	0.124	0.084	0.064	0.061	0.061	0.067
4	32	0.078	0.448	0.285	0.173	0.126	0.115	0.101	0.105
	64	0.156	1.06	0.627	0.374	0.252	0.211	0.209	0.213
	128	0.621	2.55	1.41	0.834	0.534	0.438	0.418	0.379
	16	0.128	2.952	1.1	0.662	0.480	0.447	0.379	0.373
6	32	0.256	8.6	2.44	1.46	1.05	0.86	0.784	0.74
	64	0.511	24.8	5.45	3.25	2.32	2.11	1.6	1.8
	128	1.2	74.9	11.7	7.31	4.85	4.07	3.58	3.37
	16	0.321	31.1	5.76	3.58	2.63	2.08	1.92	2.2
8	32	0.641	83.1	12.9	8.11	5.14	4.4	3.85	4.46
	64	1.3	821	29.1	17.7	12	9.77	8.67	8.25
	128	2.6	3063	65.6	40.5	27.9	21	18.6	18.4

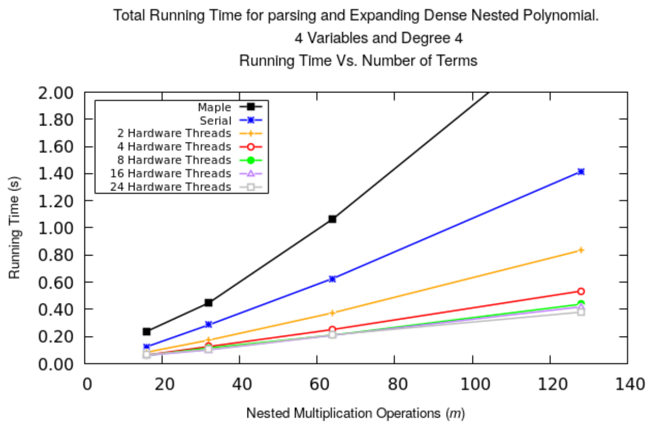
Table 5.9: Run time dataset for parsing Nested Dense Polynomial; it includes total run time data for BPAS serial and parallel implementation using Maple program as a reference point. We fixed the degree, d, of the polynomial to 4. The number of variables, v, and nested arithmetic operation (multiplication), m, of the polynomial vary as indicated.

			Memory(MB)						
v	m	Size(MB)	MAPLE	t=1	t=2	t=4	t=8	t=16	t=24
	16	0.038	2	7	7	7	7	7	7
4	32	0.078	5	12	12	12	11	11	11
	64	0.156	14	21	21	21	20	20	20
	128	0.621	42	40	40	40	40	40	39
	16	0.128	20	30	31	31	31	34	34
6	32	0.256	50	63	63	64	64	64	65
	64	0.511	139	132	132	132	133	133	134
	128	1.2	372	277	278	278	278	279	280
	16	0.321	145	138	138	139	139	140	140
8	32	0.641	338	300	301	301	302	302	303
	64	1.3	847	652	652	652	653	653	654
	128	2.6	2304	1385	1386	1386	1389	1388	1388

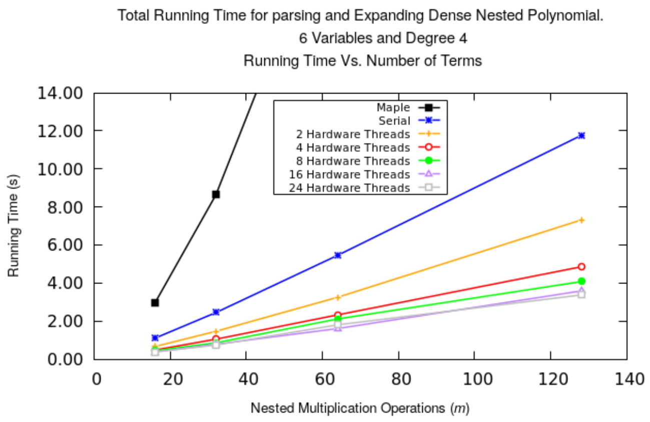
Table 5.10: Comparing the effect alternating array data structure has on memory during parsing a nested dense multivariate polynomial. We fixed the degree, d, of the polynomial to 4. The number of variables, v, and nested arithmetic operation (multiplication), m, of the polynomial vary as indicated.

		Memory(MB)				
Speedup($\frac{T_1}{T_p}$)	v	t=2	t=4	t=8	t=16	t=24
	4	1.7	2.52	3	3.1	3.1
	6	1.67	2.33	2.57	3.4	3.4
	8	1.65	2.43	2.85	3.43	3.45

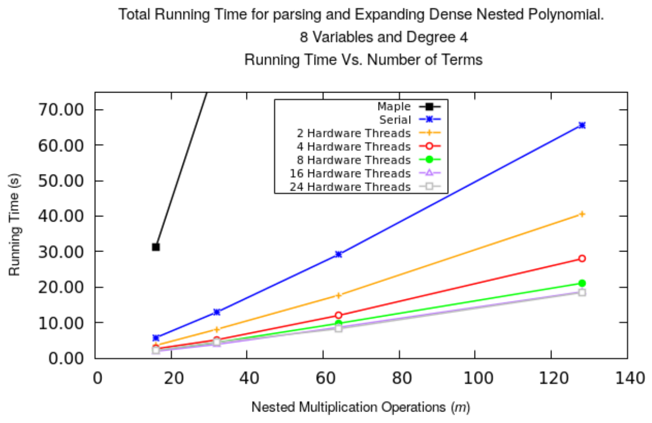
Table 5.11: Comparing the speedup of a nested dense multivariate polynomial across different variable vectors as shown in Figure 5.3.



(a)



(b)



(c)

Figure 5.3: Comparing runtime and number of multiplcation operations in the input over a Nested Dense Multivariate Polynomial. Figure (a), (b), and (c) have variable number 4, 6, and 8, respectively. As the number of variable vectors increases the runtime significantly increases.

			Time(s)											
			t=1		t=2		t=4		t=8		t=16		t=24	
v	m	Size(MB)	Split(sec)	Parse(sec)	Split(s)	Parse(s)	Split(s)	Parse(s)	Split(s)	Parse(s)	Split(s)	Parse(s)	Split(s)	Parse(s)
4	16	0.038	0.002	0.122	0.003	0.074	0.003	0.061	0.004	0.056	0.003	0.058	0.002	0.064
	32	0.078	0.005	0.280	0.004	0.168	0.005	0.120	0.006	0.108	0.005	0.096	0.005	0.100
	64	0.156	0.006	0.621	0.009	0.365	0.006	0.245	0.010	0.200	0.006	0.203	0.006	0.206
	128	0.621	0.016	1.398	0.022	0.812	0.017	0.517	0.013	0.424	0.017	0.401	0.014	0.365
6	16	0.128	0.005	1.094	0.013	0.648	0.005	0.474	0.006	0.440	0.005	0.373	0.005	0.367
	32	0.256	0.010	2.43	0.021	1.44	0.010	1.03	0.011	0.848	0.011	0.769	0.010	0.729
	64	0.511	0.022	5.34	0.031	3.22	0.025	2.30	0.025	2.09	0.021	1.58	0.022	1.78
	128	1.2	0.044	11.7	0.050	7.26	0.047	4.80	0.046	4.03	0.046	3.53	0.044	3.32
8	16	0.321	0.017	5.74	0.022	3.56	0.018	2.61	0.0179	2.06	0.015	1.91	0.019	2.18
	32	0.641	0.030	12.9	0.037	8.07	0.030	5.11	0.032	4.37	0.030	3.82	0.032	4.43
	64	1.3	0.057	29	0.066	17.6	0.061	11.3	0.06	9.71	0.046	8.62	0.058	8.19
	128	2.6	0.113	65.5	0.173	40.3	0.112	27.8	0.117	20.9	0.110	18.5	0.123	18.3

Table 5.12: Runtime dataset for parsing Nested Dense Polynomials; it includes splitting and parsing run time data for BPAS parallel implementation across different hardware threads. We fixed the degree, d, of the polynomial to 4. The number of variables, v, and nested arithmetic operation (multiplication), m, of the polynomial vary as indicated.

Chapter 6

Conclusion

Polynomial parsing is a fundamental operation in computer algebra. It allows the use of large polynomials for basic arithmetic operation. Our parser is designed to work with the BPAS library, and most BPAS functions require an alternating array to work with. Of course, it is possible to create an alternating array data structure from an input polynomial manually. The problem is, creating these input polynomials manually has a size limitation, or we have to rely on the in-efficient parser to generate this data structure. Applying large polynomials to these functions becomes impossible. Therefore, this is where our parallel parser makes it possible and open path to implementing a high-performance algorithm for computer algebra that can be challenged by large polynomials.

When parallelizing a parser, it is important to consider different input types, i.e., univariate or multivariate polynomials. A single-pass parser can efficiently run and generate the BPAS data structure, but can not take all input types. A univariate polynomial has a single variable. We can create a simple container that can store a polynomial term using *C* structs. When multivariate input polynomials are introduced, information about variables and their corresponding exponents varies; the polynomial term type, i.e., C structs, could have a varying size of exponent vectors dynamically allocated during parsing. Since an input polynomial does not display variables with zero exponents, the exact count of exponent vectors used is not clear until the input polynomial is fully scanned. For this reason, multivariate polynomials need to be scanned twice, first to discover variables and their corresponding exponent and then to parse. We took the approach of scanning it twice instead of backtracking for an update because it is efficient.

In Chapter 3, we presented our serial parser that parsers both dense or sparse; and expanded or nested polynomials. Our parser is architected using a front end compiler design tools flex and bison. We showed regular expressions applied in the flex program to generate tokens and grammars used to structure the input correctly to generate the polynomials. The serial parser uses a linked-list as its default data structure. If arithmetic functions are invoked from the BPAS library, the parser's data structure semantic changes to an alternating array. These BPAS auxiliary functions can only take an alternating array input. So, the result of our serial parser alternates between linked-list and alternating array, depending on the input polynomial. If the parser returns a linked-list, it is converted to an alternating array as a final result.

In Chapter 4, we presented our parallel parser architecture and its implementation. We presented the divide and conquer strategy to parallelize the parsing process using the CilkPlus tool [18]. Different steps are involved to parallelize the parser, i.e., splitting, organizing, parsing, and converting. Splitting the input into smaller pieces and organizing them into a single array separated by a null terminator makes it efficient to access the split input data. Parsing is done by recursively invoking the parsing function generated using bison on these multiple chunks of polynomials as discussed in Chapter 3. The parsing function algorithm utilizes the merge-sort strategy. Depending on the input type, if a linked-list semantic is used in the parser, the final result is converted to an alternating array.

Lastly, in Chapter 5, we conducted experiments to reveal the robustness of our parser using the Maple program as our comparison point. We chose our experimental input to be a randomly generated sparser polynomial, with varying numbers of polynomial terms and a nested dense polynomial with various parameters such as the number of nested inputs, number of polynomial terms in the nested inputs, etc. Across different threads, our flat polynomial experimental result shows significant speed-up. Similarly, our nested polynomial shows significant run time improvement, but as the input size grows, memory consumption increases because an additional arithmetic operation is required to expand the polynomial. Compared with the Maple Program, our parser utilizes multicore processors for parsing large input polynomials faster.

However, our parallel parser could be yet further improved across nested polynomial inputs. Currently, our parsers depend on the BPAS auxiliary function for arithmetic operations such as multiplication. Once these functions are invoked, the semantic of the parser switches from a linked-list data structure to an alternating array. Since an alternating array is a special type of array, it is an inefficient data structure to use during parsing. This could be further improved by implementing a stand alone linked list arithmetic operation functions for the parser. Doing so will force the parser to rely on the linked-list data structure. In this case, the input data will not swith its semantic to an alternating array when parsing nested input polynomials.

Furthermore, we can use the PAPAGENO paper [3] to combine parallelism at the level of parsing algorithm, not at the level of data. Using Graphics Processor Units (GPUs) with a Cocke-Younger-Kasami algorithm (CYK) and a parallelized polynomials multiplication function, we can double our speedup factor.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[2] Mohammadali Asadi, Alexander Brandt, Changbo Chen, Svyatoslav Covanov, Farnam Mansouri, Davood Mohajerani, Robert Moir, Marc Moreno Maza, Linxiao Wang, Ning Xie, and Yuzhen Xie. Basic Polynomial Algebra Subprograms (BPAS), 2019. `http://www.bpaslib.org`.

[3] Alessandro Barenghi, Ermes Viviani, Stefano Crespi-Reghizzi, Dino Mandrioli, and Matteo Pradella. PAPAGENO: A parallel parser generator for operator precedence grammars. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745 of *Lecture Notes in Computer Science*, pages 264–274. Springer, 2012.

[4] A. Brandt. High performance sparse multivariate polynomials: Fundamental data structures and algorithms. Master's thesis, University of Western Ontario, London, ON, Canada, 2018.

[5] Alexander Clark. Learning deterministic context free grammars: The omphalos competition. *Machine Learning*, 66:93–110, 01 2007.

[6] Charles Donnelly and Richard Stallman. Bison, the yacc-compatible parser generator, Jul 2020.

[7] Zacharia Fadika, Michael R. Head, and Madhusudhan Govindaraju. Parallel and distributed approach for processing large-scale XML datasets. In *Proceedings of the 2009 10th IEEE/ACM International Conference on Grid Computing, October 13-15, 2009, Banff, Alberta, Canada*, pages 105–112. IEEE Computer Society, 2009.

[8] Fayez Gebali. *Algorithms And Parallel Computing*. Wiley, 2014.

[9] Seymour Ginsburg and Sheila A. Greibach. Deterministic context free languages. *Inf. Control.*, 9(6):620–648, 1966.

[10] Torbjorn Granlund and the GMP development team. *GNU GMP: The GNU Multiple Precision Arithmetic Library*, 6.0.0 edition, 2019. `http://gmplib.org/`.

[11] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques - A Practical Guide.* Monographs in Computer Science. Springer, 2008.

[12] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs, and Koen Langendoen. *Modern Compiler Design.* Springer Publishing Company, Incorporated, 2nd edition, 2012.

[13] John R. Levine. *flex and bison - Unix text processing tools.* O'Reilly, 2009.

[14] Waterloo Maple Inc. Maplesoft. *Online Help*, 2020. `https://www.maplesoft.com/support/help/Maple/view.aspx?path=expand`.

[15] Alistair Martin. *Oxford Protein Informatics Group*, 2016. `https://www.blopig.com/blog/2016/08/processing-large-files-using-python/`.

[16] Michael D. McCool, Arch D. Robison, and James Reinders. *Structured parallel programming patterns for efficient computation.* Elsevier/Morgan Kaufmann, Waltham, MA, 2012.

[17] Anton Nijholt. Parallel parsing strategies in natural language processing. In Masaru Tomita, editor, *Proceedings of the First International Workshop on Parsing Technologies, IWPT 1989, Pittsburgh, Pennsylvania, USA, August 1989*, pages 240–253. Carnegy Mellon University / ACL, 1989.

[18] Intel Cilk Plus. *Introducing Intel Cilk Plus:Extensions to simplify task and data parallelism*, 2015. `https://www.cilkplus.org/cilk-plus-tutorial`.

[19] Elaine Rich. *Automata, computability and complexity : theory and applications.* Upper Saddle River, N.J. : Pearson Prentice Hall, 2008. Includes bibliographical references and index.

[20] Gudula Runger and Thomas Rauber. *Parallel Programming - for Multicore and Cluster Systems; 2nd Edition.* Springer, 2013.

[21] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra (3. ed.).* Cambridge University Press, 2013.

[22] Youngmin Yi, Chao-Yue Lai, Slav Petrov, and Kurt Keutzer. Efficient parallel CKY parsing on gpus. In *Proceedings of the 12th International Conference on Parsing Technologies, IWPT 2011, October 5-7, 2011, Dublin City University, Dubin, Ireland*, pages 175–185. The Association for Computational Linguistics, 2011.

Appendix A

Generate Experimental Data

Listing A.1: The maple source code *gen_nested_poly* generates a nested dense polynomial. The parameters include: *filename*, stores generated polynomial; variables used in the polynomial, *n*; maximum exponent value, *degrees*; number of nested regions, *iteration*, and randomly generated coefficients, *coef*.

```
#n = variables as [x, y, z]
#d = degree
#m = terms
#coef = coefficients random value

PolyA_No_terms := proc(n, d, coef)
        return convert(randpoly(n, coeffs=coef, degree=d, dense),
           rational)
end proc;

PolyC := proc(filename, n, d, m, coef)
        f := fopen(filename, APPEND, TEXT):
        for i from 1 to m do
                res := convert(PolyA_No_terms(n, d, coef)*
                   PolyA_No_terms(n, d, coef), string):

                if i=m then fprintf(f, "%s", res):
                else fprintf(f, "%s", res): fprintf(f, "%s", "+"):
                end if:

        end do;
        fclose(f):
        return res;
end proc;


gen_nested_poly := proc (filename, vars, degrees, iteration, coef)
        PolyC(filename, vars, degrees, iteration, coef):
end proc;
```

Listing A.2: The *C++* source code *random_polynomial_cpp_rand_file* generates flat (expanded) sparse polynomial. Generated polynomials is stored in *filename*. It takes *num_vars* number of variables; list of variables, *variables*; number of terms to generate, *num_terms*; maximum exponent value, *max_degs_range*; and value used in *GNU GMP* random class to randomize the coefficient values, *bits*.

```cpp
std::vector<unsigned int> degrees(int varSize, int range){
    std::vector<unsigned int> degs;
        unsigned int first;
        unsigned int sec;
    for(int i=0; i<varSize; i++){
        first = gen_random(range);
        sec = gen_random(first);
        degs.push_back(gen_random(65534)%sec);
    }
    return degs;
}


void random_polynomial_cpp_rand_file(std::string filename, int num_vars, std::vector<std::string> variables
   , int num_terms, unsigned int max_degs_range, int bits){
        std::ofstream f;
        f.open(filename);
        if(!f){
                std::cout << "Error creating file." << std::endl;
                exit(EXIT_FAILURE);
        }

        gmp_randclass rr (gmp_randinit_default);
        rr.seed(time(NULL));
        srand (time(NULL));
        int sign = 1;
        mpq_class e;
        long long int i;
        for (i = 0; i < num_terms; i++) {
                std::vector<unsigned int> ds = degrees(num_vars, max_degs_range);
                sign = 1;
                e = sign * mpq_class(rr.get_z_bits(bits)+1, rr.get_z_bits(bits)+1);
                if(i == 0 || sgn(e) == -1){
                        f << e;
                        for(int k=0; k<num_vars; k++){
                                if(ds[k] == 0)
                                        continue;
                                if(ds[k] == 0)
                                        continue;
                                if(ds[k] == 1){
                                        f << "*" << variables[k];
                                }else{
                                        f << "*" << variables[k] << "^" << ds[k];
                                }
                                f.flush();
                        }
                }else{
                        f << "+" << e;
                        for(int j=0; j<num_vars; j++){
                                if(ds[j] == 0)
                                        continue;
                                if(ds[j] == 0)
                                        continue;
                                if(ds[j] == 1){
                                        f << "*" << variables[j];
                                }else{
                                        f << "*" << variables[j] << "^" << ds[j];
                                }
                        }
                }
        }
        f.flush();
        f.close();
}
```

Curriculum Vitae

Name:	Amha Tsegaye
Post-Secondary Education and Degrees:	University of Western Ontario London, ON 2014 - 2017 Computer Science, B.Sc.
	University of Western Ontario London, ON 2018 - 2020 M.Sc.
Related Work Experience:	Teaching Assistant University of Western Ontario 2017 - 2020
	Research Assistant University of Western Ontario 2017 - 2018