POLITECNICO DI MILANO

Facoltà di Ingegneria Industriale e dell' Informazione

Corso di Laurea Specialistica in
Ingegneria Informatica

A SYNTAX DIRECTED APPROACH TO EXPLOIT PARALLELISM IN XML-QUERY LANGUAGES

Relatore: Prof. Matteo Pradella
Correlatori: Ing. Alessandro Barenghi

Tesi di Laurea di:
Paolo Bassi
Matricola n. 801039

Anno Accademico 2015-2016

Paolo Bassi: *A syntax directed approach to exploit parallelism in XML-query languages*
Tesi di laurea specialistica, © Dicembre 2015.

"Computer science is no more about computers than astronomy is about telescopes."
Edsger Dijkstra

Sommario

L'analisi sintattica è un processo di analisi fondamentale per svolgere operazioni di riconoscimento e traduzione di un testo scritto, determinandone la struttura tramite la definizione di una grammatica formale e la generazione di un albero sintattico, il quale mostra le regole utilizzate durante il riconoscimento dell'input. Il parser, programma atto ad eseguire questo compito, può svolgere altre azioni più o meno complesse oltre alla generazione dell'albero sintattico, tramite azioni semantiche, a seconda del tipo di grammatica che descrive il linguaggio in esame, come ad esempio recuperare delle informazioni precise da un testo scritto a seconda di come esso è strutturato. PAPAGENO è un generatore di parser paralleli scritto in Python che è in grado di produrre il codice C necessario per eseguire parallelamente l'operazione di parsing su un testo la cui grammatica soddisfi le regole della Grammatica a Precedenza di Operatori. Quest'ultima permette la separazione del testo in parti uguali e l'esecuzione dell'analisi semantica in parallelo, affidando ad ogni processore una parte del testo, senza per questo alterarne il risultato finale, anche con un'esecuzione fuori ordine. L'utilizzo di più processori in contemporanea permette quindi di ricavare uno speedup più o meno elevato a seconda delle azioni semantiche da eseguire. La tesi in esame descrive l'estensione di PAPAGENO tramite l'implementazione di azioni semantiche che permettano il riconoscimento del linguaggio XML mentre se ne esegue un'analisi alla ricerca di informazioni richieste da una query descritta tramite il linguaggio XPath, il tutto sfruttando il parallelismo fornito dalla Grammatica a Precedenza di Operatori. In particolare, la tesi descrive in dettaglio l'algoritmo utilizzato per poter ritrovare le informazioni richieste e come gestire tutti i possibili casi particolari, oltre che il sottoinsieme della grammatica XML modificata per poter soddisfare le regole della grammatica. Inoltre, segue una descrizione dettagliata dell'implementazione del programma per l'esecuzione parallela dell'analisi lessicale (lexer) e delle strutture dati che contengono tutte le informazioni raccolte durante l'analisi semantica, la cui progettazione accurata permette di sfruttare al meglio la potenza di calcolo a disposizione. Si mostrano inoltre i risultati ottenuti con applicazioni concrete attinenti alla realtà, e di come si possano mantenere prestazioni ragguardevoli seppur mantenendo un approccio generico.

Abstract

The syntax analysis is a fundamental process to perform translation and parsing of a written text, determining the structure by the definition of a formal grammar and the generation of a syntax tree, which exhibits the rules used during the recognition of the input. The parser can perform other tasks (more or less complicated) other than the generation of the syntax tree by the use of semantic action, depending on the grammar that describes the target language under analysis, for example retrieving precise informations from a written text, taking into account its structure. PAPAGENO is a parallel parser generator written in Python that produces the C code necessary to execute parallel parsing on a text whose grammar satisfies the Operator Precedence Grammar (OPG) rules, also known as Floyd's Grammar. This grammar allows the split of a text in same dimension chunks and the parallel execution of the semantic analysis, granting each processor one of the chunks, without altering the final results, even with an out of order execution. The simultaneous use of more than one processor allows a speedup gain depending on the kind of semantic actions that has to be executed. The purpose of this thesis is to describe an extension of PAPAGENO through the implementation of all the semantic actions that allows the identification of the XML language while at the same moment some kind of information are retrieved from the input file, by means of a query described using the XPath language. All of this is done by exploiting the parallelism resulting from describing the input XML file using the OPG. Moreover, it follows a detailed description of how the program is implemented, the algorithm that handles all the special cases, the design and the implementation of a custom parallel lexer to further enhance the application and to exploit parallelism and the available computing power. Also, are shown the results obtained by intensive and close to reality test cases, and how performance can be improved albeit maintaining a general purpose approach against special purpose examples.

Contents

Index I

List of Figures III

List of Tables V

Introduzione 1

1 Introduction 1

2 State of the Art 3

2.1 On Parallel Lexing and Parsing 3
2.2 Operator Precedence Grammars and PAPAGENO 10

3 The XML Parallel Parser 16

3.1 The XML Grammar 16
3.2 The XPath Query Language 20
3.2.1 Data Model 20
3.2.2 Lexical and Syntactical Structure 21
3.3 Lexing and Parsing 25
3.4 Semantic Actions 34
3.5 XPath Query Algorithm 40
3.6 Data Structures and Optimization 44
3.6.1 Data Structures 44
3.6.2 The Memory Allocators 47
3.6.3 Techniques To Avoid Serializations 51

4 Configuration and Experimental Results **56**
4.1 Lexing 58
4.2 Parsing 64
4.3 Comparison With Other Solutions 64

5 Conclusions and Future Work **69**

A C code Data Structure Definition **70**

Bibliography **72**

Ringraziamenti **75**

List of Figures

2.1 Example of OPG for arithmetic expressions. 11
2.2 Typical usage of the PAPAGENO toolchain. The human operator stages are marked in green, while the PAPAGENO automated staged are marked in blue. 12

3.1 Example of OPG for XML Grammar 23
3.2 Description of a Lex Token structure 29
3.3 Description of a Node Info Structure 44
3.4 Description of a Leaf List Structure 47
3.5 How the two preallocation pools are used to store Node Infos and Leaf Lists 50
3.6 Argument Structure containing all the pointers that will be freed at the end of the parsing phase 54

4.1 Speedup and Parallel Code Portion using Flex as parallel lexer on three different machines. 60
4.2 Speedup and Parallel Code Portion using a Custom Lexer as parallel lexer on three different machines. 63
4.3 Speedup and Parallel Code Portion regarding the parsing phase on three different machines, missing the global variable and function arguments optimizations 65
4.4 Speedup and Parallel Code Portion regarding the parsing phase on three different machines, optimized with respect to global variable references and function arguments 66

4.5 Total Lexing and Parsing elapsed time employed by the Notebook and the PP Transducer running the TreeBank files. Dark lines show bigger files, respectively 85 MB, 172 MB, 258 MB, 344 MB, 516 MB, 860 MB and 1 GB 67

List of Tables

3.1 Token and their corresponding Regular Expression 27
3.2 Terminal description using the Regular Expressions of 3.1 . 28

4.1 Total text analysis times of the XML test-bench files, sequential execution, with 0.5% of standard deviation 58

1 Introduction

Language Parsing (or syntactic analysis) is the process of analyzing a string of symbols conforming to the rules of a formal grammar, and it occurs in many situations: compilation, natural language processing, document browsing, genome sequencing, program analysis targeted at detecting malicious behaviours, and others. It is frequently applied to very large data sets in contexts where speedups and related energy saving are often important. The common linear time left to right LR(1) and LL(1) algorithms used for deterministic context free (or BNF) languages are an important milestone of algorithmic research. Due to their ability to recognize a wider class of formal languages, they superseded earlier algorithms such as the ones employing Floyd's Operator Precedence Grammars (OPG), which rely on only local information to decide parsing steps (for an introduction see e.g. [1]).

Recently research on formal methods has renewed the interest for OPGs, and thanks specifically to their nice closure under the substring extraction and decidability properties [2], they're used to enable independent parallel parsing of substrings.

Parallel parsing requires an algorithm that, unlike a classical parser, is able to process substrings which are not syntactically legal, although they occur in legal strings. The few people who have performed some limited experimentation on algorithms that relies on splitting a long text into chunks have generally found that performances critically depend on the cut points between them: starting a chunk on an identifier opens too many syntactic alternatives. As a consequence such parsers have been typically combined with language-dependent heuristics for splitting the source text into chunks that start on keywords announcing a splitting friendly construct. This approach creates too much computational overhead since it requires a specific implementation depending on the language that is going to be analyzed:

moreover, in order to extract parallelism from the parsing process, some published works, e.g., [3] and [4], rely on the assumption that the parsed language is a subset of the original and other ad-hoc strategies that cannot be applied in a general purpose environment.

The need of such parallel parsers comes from the great amount of data that modern applications produce and use (a common example could be Twitter, that generates terabytes of data every day) coupled to the fact that data centers have great hardware capabilities (hundreds of cores and RAM space) but not general purpose software that exploits this features.

A concrete example of a general purpose parallel parser generator that exploits the FG properties and do not suffer from arbitrary text splitting is PAPAGENO [5]: measurements on different languages like JSON and Lua files indicate good scalability on different multi-core architectures, leading to significant reductions of parsing time with respect to state-of-the-art sequential parsers.

This thesis aims to use PAPAGENO and extend the parser with a subset of the XML language and enrich it by adding the capability to execute XPath queries on the XML files while doing the parallel parsing operation. XML has been chosen as the target language since it is possible to adapt its grammar to satisfy the FG properties, allowing the parallel parsing process; moreover, it's used in a lot of different environments, from documentation to databases, and in all cases the files have dimensions that could take advantage of it. XPath is a query language for selecting nodes from an XML document: it navigates the XML tree depending on the query and returns the corresponding results. There are a lot of applications where selecting a particular information in the tree is required, like data processing for information analysis on social networks (e.g. Twitter) or query on semi-structured databases, and doing that while parsing leads to a great time saving. The thesis explains the steps required to build and write the XML grammar, the semantic actions and how to extract the query required information from the file by exploiting multi-core architecture and the parallel parser generated using PAPAGENO.

The thesis is organized as follows: Chapter 2 provides information about the parallel parsing, the PAPAGENO parser generator and the other solutions provided by the state of the art, Chapter 3 proposes the implementation of the XML semantic actions, the data structures and how to integrate all with the XPath query, Chapter 4 presents all the experimental results on different kind of data structure and a comparison with an already non generic tested parallel parser. Finally, Chapter 5 draws the final conclusions and future works.

2
State of the Art

2.1 On Parallel Lexing and Parsing

The literature on parallel parsing (and lexing) is vast and extends over half a century. The valuable survey and bibliography [6] lists some two hundred publications in the period 1970 - 1994, and research has continued since, though perhaps less intensively. We omit, as less relevant to our objectives, some categories: the work on grammar types not belonging to the context-free family, the studies based on connectionistic or neural models of parallel computations, and the large amount of work on natural language processing. We are left, roughly speaking, with the following categories:

- Theoretical analysis of algorithmic complexity of parallel context-free language recognition and parsing, in the setting of abstract models of parallel computation, such as P-RAM.

- Parallel-parser design and performance analysis for specific programming/web languages, sometimes combined with experimentation, or, more often, simply with demonstration, on real parallel machines.

Category 1 is mainly concerned with the asymptotic complexity of recognition/parsing algorithms on abstract parallel machines. The algorithms proposed for unrestricted CF grammars require an unrealistic number of processors: for instance Rytter's [7] recognizer has asymptotic worst-case time complexity O(log n), with n the input length, and requires $O(n^6)$ processors; the numbers of processors grows to $O(n^8)$ if parsing, i.e., syntax-tree construction, is required. Several researchers have shown that such complexity bounds can be lowered, by restricting the language class, sometimes so much that it loses practical interest.

Such idealized results are, of course, not really comparable with experimental findings, as already asserted by [6], yet they offer some interesting indications. In particular, all the subfamilies of deterministic CF languages for which the theoretical complexity analysis reports a close to linear use of processors, are included in the family of OP languages used by PAPAGENO [5].

Such abstract complexity studies had little or no impact on practical developments, for several reasons. Firstly, it is known that the abstract parallel machines, such as P-RAM, poorly represent the features of real computers, which are responsible for performance improvements or losses. Secondly, asymptotic algorithmic time complexities disregard constant factors and mainly focus on worst cases, with the consequence that they are poorly correlated with experimental rankings of different algorithms. Lastly, most theoretical papers do not address the whole parsing task but just string recognition.

In the following years (1995-2013) the interest for research on the abstract complexity of parsing algorithms has diminished, with research taking more practical directions.

The classical tabular recognition algorithms (CKY, Earley) for unrestricted CF languages have attracted much attention, and a number of papers address their parallelization. It is known that such parsers use a table of configurations instead of a pushdown stack, and that their time complexity is related to the one for matrix multiplication, for which parallel algorithms have been developed in many settings. Parallel algorithms derived from CKY or from Earley sequential parsers (**sequential parsers** for brevity) may be pertinent to natural language processing, but have little promise for programming/data description languages. As tabular sequential parsers are significantly slower than LR or OPG sequential parsers (up to some orders of magnitude), it is extremely unlikely that the parallelization of such a heavy computational load would result in an implementation faster than a deterministic parallel parser. Moreover confirmation by experiment is not possible at present. The comparison with previous work in category 2. is more relevant and reveals the precursors of several ideas we use in our generator. We only report on work dealing with deterministic CF languages.

Bottom-up parsing. Some early influential efforts, in particular [8] (described in [9]) and [10], introduced data-parallelism for LR parsers, according to the following scheme: a number of LR sequential parsers are run on different text segments. Clearly, each sequential parser (except the leftmost one) does not know in which parser state to start, and the algorithm must spawn

as many deterministic LR sequential parsers as the potential states for the given grammar; each parser works on a private stack. When a parser terminates, either because it has completed the syntax tree of the text segment or because the lack of information on the neighboring segments blocks further processing, the stack is merged with the neighboring left or right stack, and the sequential parser process terminates. However, the idea of activating multiple deterministic bottom-up sequential parsers is often counterproductive: the processes, associated to the numerous parser states of a typical LR grammar, proliferate and reduce or nullify the speedup over sequential parsing.

Two ways of reducing process proliferation have been proposed: controlling the points of segmentation and restricting the family of languages considered.

An example of the first is in [11], so explained: "The given input string is divided into approximately q equal parts. The i-th processor starting at token scans to the right for the next synchronizing token (e.g. semicolon, end, etc.) and initiates parsing from the next token". If synchronizing tokens are cleverly chosen, the number of unsuccessful parsing attempts is reduced, but there are drawbacks to this approach: the parser is not just driven by the language grammar, but needs other language- specific indications, to be provided by the parser designer; thus, [11] chooses the synchronizing tokens for a Pascal-like language. Furthermore, to implement this technique, the lexer too must be customized, to recognize the synchronizing tokens.

Similar language-dependent text segmentation policies have been later adopted by other projects, notably by several developments for XML parsers; such projects have the important practical goal to speed-up web page browsing, and investigate the special complexities associated to parallel HTML parsing. Although they do not qualify as general purpose parsers, their practical importance deserves some words. The recent [12] paper surveys previous related research, and describes an efficient parallel parser, *Hpar*, for web pages encoded in HTML5. HTML5 has a poorly formalized BNF grammar and tolerates many syntax errors. A HTML5 source file may include a script (in JavaScript), which in turn can modify the source file; this feature would require costly synchronization between lexing and parsing threads, which make a pipelining scheme inconvenient. Hpar splits the source file into units of comparable length, taking care not to cut an XML tag. Each unit is parsed by an independent thread, producing a partial DOM tree; at last, the DOM trees are merged. A complication comes from the impossibility to know whether a unit, obtained by splitting, is part of a script, a DATA section, or a comment. The parser uses heuristics to speculates that

the unit is, say, part of a DATA section, and rolls-back if the speculation fails. More speculation is needed for another reason: when a unit parser finds a closing tag, say *</Table>*, it does not know if the corresponding opening tag occurred before in the preceding unit, or if it was missing by error. The best speedup achieved (2.5× using five threads) does not scale for the current web page sizes.

Returning to parsers purely driven by the grammar, in view of the popularity of (sequential) LR parser generation tools like Bison, before the implementation of [5], the fact that no parallel-parser generators existed was perhaps an argument against the feasibility of efficient parallel parsers for LR grammars. This opinion was strengthened by the fact that several authors have developed parallel parsers for language families smaller than the deterministic CF one, of which one example suffices. The grammars that are LR and RL (right to left) deterministic enjoy some (not quantified) reduction in the number of initial parser states to be considered by each unit parser. Such grammars are symmetrical with respect to scanning direction: rightwards/leftwards processing, respectively, uses look-ahead/look-back into the text to choose legal moves. By combining the two types of move into a bidirectional algorithm, dead-ended choices are detected at an earlier time. It was observed that Floyd's OPGs too have the property of reversibility with respect to the parsing direction and benefit from it for making local parsing decisions, which are unique and guaranteed to succeed if the input text is grammatically legal. Indeed, thanks to the local parsability property, OP languages do not incur in the penalties that affect LR parsers; the latter, as said, need to activate multiple computations for each deterministic unit parser, since many starting states are possible. For OP parsers, in fact, all the actions can be deterministically taken by inspecting a bounded context (two lexemes) around the current position, and do not depend on information coming from the neighboring unit parsers: thus, each text unit can be processed by an OP parser instance along a single computation, without incurring on the risk of backtrack.

Top-down parsing. Less effort has been spent on top-down deterministic LL parsers, possibly because, at first glance, their being goal-oriented makes them less suitable for parsing arbitrarily segmented text. The article [13] surveys the state of the art for such parsers and reports in detail a parallel (non-experimented) algorithm that works for a subclass of LL grammars, named LLP. Imagine that the text is segmented into substrings and on each segment a classical LL(k) s- parser is applied. Similar to the LR case, each sequential parser does not know the result (i.e., a stack representing the prefix of a leftmost derivation) for parsing the substring to its left: there-

fore each sequential parser has to spawn as many sequential parsers working on the same segment, as there are possible initial stacks, too many to be practical. Therefore it is proposed to limit the number of possible initial stacks by imposing a restrictive condition on LL(k) grammars. The subfamily thus obtained is named LLP (q, k) and is based on the idea of inspecting a look-back of length q tokens as well as the classical look-ahead of k tokens. Although not compared in this thesis, LLP (q, k) grammars look quite similar to the already mentioned bounded-context grammars. This and earlier studies on parallel LL parsers may be theoretically interesting but do not offer any hint on practical usability and performances.

Parallel lexing. The problem of breaking up a long string into lexemes is a classical one for data parallel algorithms, well described in [14]. They assume, as such studies invariably do, that each lexeme class is a regular language, therefore the sequential lexer is a deterministic finite automaton (DFA) that makes a state transition reading a character. For a string x, the chain of state transitions define a lexing function that maps a state p to another state q; moreover the function for the string $x \cdot y$ obtained by concatenation is obtained by function composition. The data-parallel algorithm is conceptually similar to the one for computing all partial sums of a sequence of numbers, also known in computer arithmetic as the parallel sum prefix algorithm.

In essence, the source text is split into pieces, and the DFA transition function is applied to each piece, taking each DFA state as a possible starting state. Then the functions obtained for neighboring pieces are composed and the cases of mismatch are discarded. Such processing can be formulated by means of associative matrix operations. This parallel algorithm is reported to be optimal from a purely theoretical viewpoint, but early simulation on fine-grained architectures with very many processing units is not conclusive. More recently, various experiments of similar algorithms on GPGPU and on multi-core architectures have been reported. A criticism is that such algorithms are very speculative, performing a significant amount of computation which may be later on discarded, thus yielding fairly poor energy efficiency. Some authors have considered the regular expression matching problem, instead of the lexing problem, and, although regular expressions and DFA models are equivalent, the parameters that dominate the experiments may widely differ in the two cases. An example suffices: [15] presents a notable new version of the mentioned [14] approach. They claim that for certain practical regular expressions that are used in network intrusion/detection systems, the size of the parallel lexer remains manageable and not bigger than the square of the minimal DFA. Then, they are able to construct the

parallel scanner on-the-fly, i.e. delaying as much as possible the construction of the states. Clearly, algorithm [15] is not intended as a lexer to be invoked by a parallel parser, but as a self-standing processor for matching regular expressions – yet partially so, since it does not address the central issue of ambiguous regular expression parsing, which fortunately does not concern our intended applications.

Recently, [16] has experimented on the Cell Processor a parallel version of the Aho-Korasick string matching algorithm. This work was motivated by the good performance of that algorithm on multi-core machines for string search against large dictionaries. But a downside of that approach is that it apparently assumes that the input file can be unambiguously divided into text segments; therefore it does not apply to the case of general programming- or data-representation languages, since, for such languages, scanning cannot avoid an initial degree of nondeterminism caused by the absence of a separator between tokens (as a newline) that could be identified by inspecting a bounded portion of the segments.

Compared with the mentioned studies, the PAPAGENO approach to parallel lexing addresses further critical issues. First, the approach is suitable for more general lexical grammars that involve pushdown stacks and cannot be recognized by a DFA (as the lexical grammar of Lua). Second, the approach integrates some pre-processing steps that enhance the performance of the following parsing stage. Furthermore, it addresses a complexity that made previous approaches such as [14] [10] unpractical: splitting the input file into segments may cause ambiguity, in the sense that the lexing function associated to a segment may return multiple values (states), depending on the assumed input state. To compute such function, several workers are needed, but in the PAPAGENO design their number does not equal the number of states of the automaton, but is limited to two or three, and does not critically affects performance, as attested by the experimental results.

Parallel Parsing This thesis follows the solution proposed by PAPAGENO (PArallel PArser GENeratOr) [17]. PAPAGENO is an open source project available under GNU General Public License and it is written in ANSI/ISO C and Python: the codebase can be downloaded at [18]. Its toolchain provides an automatic parallel parser generator that converts a specification of a syntactic grammar into an implementation of the operator precedence parallel parsing algorithm described in the following section. The generated parallel parsers can be complemented with parallel scanners, hence obtaining a complete parallel lexer and parser library. For the library, C has been chosen as implementation language, because it permits strict control over the computation process and memory management. For the sake of porta-

bility, all the C code generated by PAPAGENO employs fixed-size types standardized in the C99 standard; furthermore, it relies exclusively on the standard C runtime and a POSIX-compliant thread library, thus avoiding any architecture-specific optimization. The generated lexers and parsers run successfully on x86, x86 64, ARMv5 and ARMv7 based-platforms with no code modifications.

Some Notable Examples Some other studies, in particular [19], attempt to provide a concrete solution to parallel parsing, by inspecting the problem of executing an XPath query during the parsing process. The approach used is based on pushdown transducers, allowing the split of the input file in arbitrary sized chunks. The major problem to address is the fact that parsing is inherently a sequential process and the split needs to be done in a way to put the parser in a well-defined state at the start of each block of the stream. Some proposed solutions, such as [3] and [20] attempt to solve the empasse either through pre-processing the input file, splitting it into well-formed fragments, or by making speculative executions based on heuristics to guess the initial state of the parser.

The PP transducer described in [19] attempts to solve this problem mapping all possible starting states to corresponding final state, and, as the processing progresses, join all the converged solution in a sequential operation. The subset of XPath supported natively by PP Transducers is limited to child and descendant queries with no support for predicates and splitting the XML file does not produce well-formed fragments. Building the mappings requires the transducers to be executed multiple times over each chunk, once for each possible starting state, but it can be done in parallel since these mappings do not depend on the previous state of the transducer. After that, the join phase combines all the results with the initial state: the effectiveness of this approach depends on the amount of work needed to construct the mapping compared to sequential execution. Tree data structures are implemented in order to overcome redundant computations.

Compared with other known solution for parallel XML parsing, like PugiXML [21] and Expat [22]. The first is a light-weight C++ XML processing library featuring, among other things, a DOM-like interface with rich traversal/modification capabilities and an extremely fast non-validating XML parser which constructs the DOM tree from an XML file/buffer. Expat is a stream-oriented parser library written in C in which an application registers handlers for things the parser might find in the XML document (like start tags). The PP Transducer get some notable achievements like a 2.5 GB/s throughput (due mostly to the available computational power), but also some flaws: up to 25 cores, PugiXML outperforms PP Transducer

because the overhead of managing state mappings is greater than that of constructing DOM trees. The query structure is also an important factor, since the ones that uses descendant axis (i.e. //) are less efficient with PP Transducers since they add more transitions to it, reducing the convergence of the states. The tests highlight the need of building a separate DFA to execute the XPath query and the possibility to help the parser by providing all the tags inside the input files, saving some time by sacrificing the generality of the solution.

2.2 Operator Precedence Grammars and PAPAGENO

Operator Precedence Grammars Since Operator Precedence Grammars (OPG) and parsers are a classical technique for syntax definition and analysis, it suffices to recall the main relevant concepts from e.g. [1].

Let Σ denote the terminal alphabet of the language. A BNF grammar in operator form consists of a set of productions P of the form A $\rightarrow \alpha$ where A is a nonterminal symbol and α, called the right-hand side (rhs) of the production, is a nonempty string made of terminal and nonterminal symbols, such that if nonterminals occur in α, they are separated by at least one terminal symbol. The set of nonterminals is denoted by V_N .

It is well known that any BNF grammar can be recast into operator form. To qualify as OPG, an operator grammar has to satisfy a condition, known as absence of precedence conflicts. We will now introduce informally the concept of precedence relation, a partial binary relation over the terminal alphabet, which can take one of three values: $\lessdot$ (yields precedence) , $\gtrdot$ (takes precedence) , $\doteq$ (equal in precedence).

For a given OPG, the precedence relations are easily computed and represented in the operator precedence matrix (OPM). A grammar for simple arithmetic expressions and the corresponding OPM are in Figure 2.1. Entries such as $+ \lessdot a$ and $a \gtrdot +$ indicate that, when

Grammar G consists of Σ = { a, +, ×, (,) }, V_N = { E, T, F } , axiom = E and

$$P = \{\ E \rightarrow E + T \mid T,\ T \rightarrow T \times F \mid F,\ F \rightarrow (E) \mid a\ \}$$

Operator precedence matrix:

	a	$+$	$\times$	(	)
a		$\gtrdot$	$\gtrdot$		
$+$	$\lessdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$
$\times$	$\lessdot$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$
)		$\gtrdot$	$\gtrdot$		$\gtrdot$

Figure 2.1: Example of OPG for arithmetic expressions.

parsing a string containing the pattern ... + a + ..., the rhs a of rule F → a has to be reduced to the nonterminal F. Similarly the pattern ... + (E) × ... is reduced by rule F → (E) to ... + F × ... since the relations are ... + $\lessdot$ (E) $\gtrdot$ × ... . There is no relation between terminals a and (because they never occur as adjacent or separated by a nonterminal. A grammar is OPG if for any two terminals, at most one precedence relation holds. In sequential parsers it is customary to enclose the input string between two special characters ⊥, such that ⊥ yields precedence to any other character and any character takes precedence over ⊥. Precedence relations precisely determine if a substring matching a rhs should be reduced to a nonterminal. This test is very efficient, based on local properties of the text, and does not need long distance information (unlike the tests performed by LR(1) parsers). In case the grammar includes two productions such as A → x and B → x with the same rhs, the reduction of string x leaves the choice between A and B open. The uncertainty could be propagated until just one choice remains open, but, to avoid this minor complication, we assume without loss of generality, that the grammar does not have repeated right hand side's [2] [23]. The mentioned local properties suggest that OPG are an attractive choice for data-parallel parsing, but even for sequential parsing, they are very efficient [1] “Operator-precedence parsers are very easy to construct and very efficient to use, operator-precedence is the method of choice for all parsing problems that are simple enough to allow it”. In practice, even when the language reference grammar is not an OPG, small changes permit to obtain an equivalent OPG, except for languages of utmost syntactic complexity.

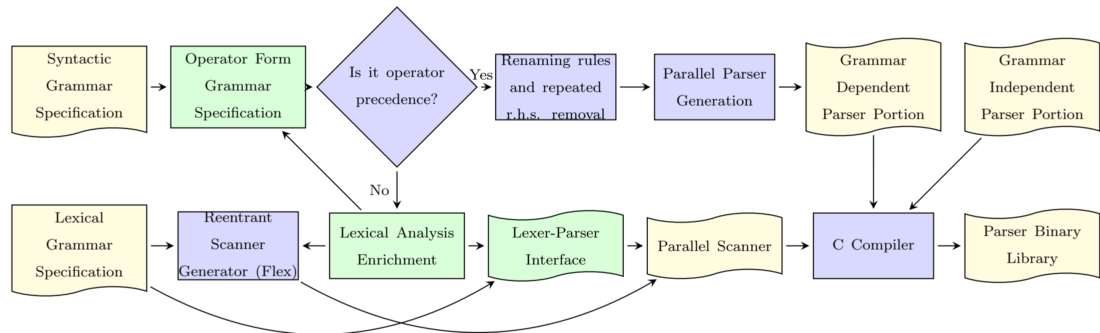


Figure 2.2: Typical usage of the PAPAGENO toolchain. The human operator stages are marked in green, while the PAPAGENO automated staged are marked in blue.

Architecture of PAPAGENO toolchain The architecture is depicted in Figure 2.2. The input of the process contains the specifications of the lexical and syntactic grammars of the target language. If the syntactic grammar of the language is not in operator precedence (OP) form, the tool notifies the inconsistency in the input specification and the user is given proper diagnostics pointing out the rules where precedence conflicts or adjacent nonterminals occur. The user has thus to modify the grammar: a convenient approach to eliminate precedence conflicts consists in enriching the lexical analysis stage with proper transformations, as insertions or renaming of tokens.

Then PAPAGENO automatically eliminates from the OP grammar both the repeated rhs rules and the renaming rules. At last, the C code of a parallel parser is generated. The parallel parser generator in PAPAGENO has been designed as a replacement for the classical GNU Bison generator and adopts the same basic syntax conventions, allowing an easy porting of the grammar descriptions available in Bison-compliant format. The generated parallel parser is logically split into two parts, as shown in Figure 2.2 a language independent support library, and a language dependent parser code portion. This choice was made to allow for easy extensions and possibly further architecture dependent optimization of the language independent portion, while retaining the automated code generation feature.

The parsing process is invoked by means of a function call, where the developer may specify at runtime the input stream to be analyzed and the number of workers to be employed to perform the analysis. Each worker is mapped to a single POSIX thread, belonging to a thread pool initialized at the beginning of the parsing process. The developer can choose between two parallel parsing strategies in the generated code. In the first strategy,

after a first execution of the parallel parsing algorithm, the recombination of the partial stacks is assigned to a single worker which operates in sequential mode. In the second strategy, instead, the first parallel pass of the parsing algorithm is followed by parallel recombination of the partially parsed substrings along the lines: the number of initial workers is reduced by at least two, and each of the remaining workers has to recombine two partial parsing stacks generated in the first pass. This recombination process is iterated until a single thread is left to complete the parsing. The second strategy aims at exploiting the parallelism offered by particularly deep parsing trees.

The PAPAGENO generated parsers can be naturally combined with either a sequential Flex generated scanner or a parallel scanner. Unlike the generation of a parallel parser, which is fully automatic, the phase of parallel lexer generation requires some interaction with the user. In particular, the programmer is expected to provide the specification of the grammar in the Flex input format for reentrant lexers, write the code managing the input character stream splitting, and the one handling the token list recombination. The input splitting code performs the actual chunking, possibly employing a fixed-width search window, and inputs the data into the Flex-generated scanners. The multiple working states of the scanner are mapped onto the multi-state lexer features offered by Flex, requiring from the programmer the definition of the language-specific transitions from one state to the other. At the end of the parallel lexing process, the information on the multiple lexer is exploited by the code written by the programmer to perform the constant-time recombination of the token lists produced by the parallel lexers.

Finally, once the parallel scanner is obtained, as a combination of the output of Flex and the user's lexer-parser interface, it is possible to compile all the sources generated by the toolchain, resulting in a complete binary lexing and parsing library.

Optimization Techniques The internal architecture of PAPAGENO relies on carefully designed implementation strategies and data structures, which play a fundamental role to obtain high performances of parallel lexers and parsers. From the paper, we recall the well-known bottlenecks preventing efficient parallelization and the solutions adopted in our tool to cope with them.

Two commonplace issues in achieving practical parallelism are 1) the data representation and handling geared towards efficient memory use, and 2) a proper management of the synchronization issues, typically minimizing the use of locks. Thanks to the computationally lightweight parsing algorithm devised for OP grammars, and the minimal requirement for synchronization

actions, issue 2) is less important, and memory management and memory allocation locality was found to be the crucial issue, as further explained in Section 3. We describe several simple yet effective memory optimization. First, terminal and nonterminal symbols are encoded as word-sized integers, taking care of employing one bit of the encoding to distinguish terminal from non-. By default, the most significant bit is used; however PAPAGENO allows to choose its position at parser generation time to allow room for further information packing. Such information packing does not prevent the definition of large target languages, as the architecture word length in modern devices is at least 32 bit, and 64 bit for most of them. Adopting this technique, it can be done without a look-up table to check whether a symbol on the parsing stack is a terminal or non-.

A second optimization towards improved data locality comes from the observation that the precedence relation between may take one out of four values ($\lessdot$, $\doteq$, $\gtrdot$, $\perp$). Using a bit-packed representation of the precedence matrix, it has been obtained a significant savings for large matrices (which occur in large languages), and, moreover, it manages to fit entirely the matrix in the highest level caches, thus significantly improving the average memory access latency.

Furthermore, in order to avoid serialization among the workers upon the system calls for dynamic memory allocation, it has been adopted a memory pooling strategy for each thread, wrapping every call to the malloc function. This strategy has also the advantage of reducing memory fragmentation, since the memory allocation is done in large contiguous segments. To evaluate the memory needed for pre-allocation during parsing, it is estimated the number of nodes of the parsing tree by computing the average branching factor of the AST as the average length of the rhs of the productions. Then, the parallel parser generator initially pre-allocates half of the guessed size of the AST and augments the memory pool of a worker by one fifth of this quantity, every time the thread requires more memory. A similar memory pooling strategy is employed in the lexing phase, in order to avoid serialization among the lexing threads in need for memory to allocate the token lists.

One of the most computationally intensive parts of OP parsers is the matching of a production right-hand-side (rhs) against the ones present in the grammar. The solution found is representing the rhs's as a prefix tree (trie), so that it becomes possible to find the corresponding left hand side in linear time with respect to the length of the longest rhs of the grammar. Furthermore, to optimize the size and the access time to the trie, it has been followed the technique described in [24], that represents the structure as an array, storing the pointers to the elements of the trie within the same vector.

The vectorized trie is fully pre-computed by PAPAGENO, and is included in the generated parser as a constant vector.

For the synchronization and locking issues in OP-based parallel parsing, the techniques used are rather straightforward. Since each parallel worker performs the parsing action on separate tokenized input chunks, it is completely independent from the other workers, and there is no need for any synchronization or communication between them. This in turn allows PAPAGENO's strategy to scale easily even in the cases where the inter-worker communication has a high cost, e.g. whenever the input is so large that they have to be run on different hosts. Similarly, all the lexers act independently on the input, without need of communication or synchronization while performing the lexing actions. The requirement for enforced synchronizations is only present in the following two cases: i) a single barrier-type synchronization point is required between the end of the lexing phase, and the beginning of the parsing one whenever the lexical grammar requires a constant-time chunk combination action to be performed by the lexer; ii) synchronizations are required to enforce data consistency if the user desires to perform multiple parallel parsing recombination passes, instead of a single one. While the first barrier synchronization cannot be subject to optimization, the synchronizations between multiple parallel parsing recombination passes can be fruitfully organized hierarchically. In particular, a parsing worker from the n-th pass will only need to wait for the completion of the n - 1 pass workers producing its own input, effectively avoiding the need of a global barrier synchronization between passes. Such a strategy allows to effectively exploit the advantages of multiple parallel passes whenever the parse tree is very high.

3 The XML Parallel Parser

This chapter presents the main work of this thesis. Section 1 describes the XML grammar used in the developed examples, the rules, the terminal and nonterminal tokens involved. Section 2 recaps the XPath query language, its data model, the lexical and syntactical structure used to extract information from the XML files. Section 3 describes the techniques employed to create a custom lexer and how to parallelize it together with the parser. Section 4 presents all the semantic actions involved in the process of parsing and their algorithm described in details. Section 5 shows the main algorithm used to retrieve the information requested by the query while parsing the XML input file. Finally, Section 6 describes the data structure used to store all the relevant data, illustrates ways to improve the memory allocation phase and describes various techniques employed to avoid unwanted serialization during the lexing and parsing parallel code portions.
While describing the algorithm and the grammar, we use different fonts to evidence and recall easily the arguments of the discourse:

- Portions in *Italics* highlight the topic of the speech
- Portions in `Teletype` font point out code portion and nonterminal tokens in the grammar description

In the Appendix A at the end of Chapter 5 there are the C code representation of all the figures that describes the data structure presented in this Chapter.

3.1 The XML Grammar

XML has a well defined grammar [25], but for the purpose of this thesis, we use a subset of it, without excluding the most fundamental part. Since

we need a grammar that satisfies the Operator Precedence rules, a new one has been created in order to let PAPAGENO analyze and create the corresponding C code of the semantic actions, as described in the following sections.

DTD A Document Type Definition describes the structure of a class of XML documents and is composed by a list of declarations of the elements and attributes that are allowed within the documents. An element type declaration defines an element and its content. An element content can be:

- **Empty** the element has no content, i.e., it cannot have children elements nor text elements;
- **Any** the element has any content, i.e., that it may have any number (even none) and type of children elements (including text elements). As a simplifying restriction, we forbid this type of content.
- **Expressions** which specifies the elements allowed as direct children. They can be:
  - an *element content*, meaning that there cannot be text elements as children. The element content consists of a content particle, which can be either the name of an element declared in the DTD, or a sequence list or choice list.
    - a *sequence list* is an ordered list (specified between parentheses and separated by a "," character) of one or more content particles, which appear successively as direct children in the content of the defined element;
    - a *choice list* is a mutually exclusive list (specified between parentheses and separated by a "|" character) of two or more content particles.

    The element content may be followed by a quantifier (+ , * or ?), which specifies the number of successive occurrences of an item at the given position in the content of the element (one or more, zero or more, zero or one, respectively).
  - a *mixed content*, meaning that the content may include at least one text element and zero or more declared elements; differently than in an element content, their order and number of occurrences are not restricted:
    - (Parser Character Data): the content consists exactly of one text element (often called PC- DATA);

- (#PCDATA | element name | ...) *: the content consists of a choice (in an exclusive list between parentheses and separated by "|" characters and terminated by the "*" quantifier) of two or more child elements (including only text elements or also the specified declared elements).

Grammar Rules and Description The grammar syntax used to represent XML is not the official one described in [25], but it has to be refactored to generate a subset of it that satisfies the OPG rules of having each nonterminal in the rhs of each rule separated by at least one terminal, or no nonterminal at all.

First of all, let's start describing the set of terminal, nonterminal and axiom involved:

Nonterminals There is only one nonterminal in this grammar and it's the ELEM token. It serves different purposes depending on the rules where it is involved: it starts containing the minimum amount of information retrieved from a leaf and it ends collecting all the nodes information that satisfies the query statement.

Terminals Since there cannot be two close ELEM nonterminals, the purpose of the terminals is to separate them and the information they contains. They basically represents all the tokens that are not useful to the query statement, like angular parenthesis, tag names, spaces and newlines. The complete list follows:

- `Openbracket` represents the string inside the opening tag and the two angular parenthesis, e.g. $<a>$.
- `CloseBracket` represent the string inside the closing tag, the two angular parenthesis and the backslash, e.g. $$
- `OpenParams` represents the string inside the opening tag, the two angular parenthesis and all the parameters that follows the tag name, e.g. *<a param1="value1">*
- `OpenCloseInfo` represents a `OpenBracket` followed by a `CloseBracket`, without any string between them, e.g. $< tag >< /tag >$
- `OpenCloseParams` represents represents a `OpenBracket` with parameters followed by a `CloseBracket`, e.g. $< tagparam1 = "value1" >< /tag >$

- `AlternativeClose` is a shorten representation of the `OpenCloseInfo`, e.g. $< \backslash tag >$
- `Infos` represents the strings between tags, e.g. *word* in $<a>word$, possibly empty

Rules Since the grammar used is a subset of the original XML Grammar, there are few rules and no special cases that requires particular attention. The grammar is build taking into account that an XML document is structured like a tree and that the information can assume different meanings depending on their position inside the structure.

$ELEM \rightarrow ELEM$ `OpenBracket` $ELEM$ `CloseBracket`

$ELEM \rightarrow ELEM$ `OpenParams` $ELEM$ `CloseBracket`

$ELEM \rightarrow ELEM$ `OpenCloseInfo`

$ELEM \rightarrow ELEM$ `OpenCloseParam`

$ELEM \rightarrow ELEM$ `AlternativeClose`

$ELEM \rightarrow$ `OpenBracket` $ELEM$ `CloseBracket`

$ELEM \rightarrow$ `OpenParams` $ELEM$ `CloseBracket`

$ELEM \rightarrow$ `OpenCloseInfo`

$ELEM \rightarrow$ `OpenCloseParam`

$ELEM \rightarrow$ `AlternativeClose`

$ELEM \rightarrow$ `Infos`

With the exception of the rule recognizing the `Infos` terminal, all the others are coupled with a rule that places before the ELEM nonterminal. The rules without the starting ELEM nonterminal in the rhs are the first called when parsing a subtree with at least one element; after that, each other element in the same tree level is recognized, depending on their type, by one of the other rules that starts with the ELEM nonterminal, that contains all the previous results packed in a single element. This helps the rules on the same level to maintain the order of the elements in the case of a subtree request.

The Axiom There is no single rule that serve as axiom, since the only nonterminal used is in common among all of them. However, it is obvious that a common XML document has a single couple of tag elements called *root* that contains the entire tree. This root element can contain some parameters too, so the two rules that we expect to see as axioms can be $ELEM \rightarrow$ `OpenBracket` $ELEM$ `CloseBracket` and the corresponding one with the parameters. Nevertheless, the grammar is able to recognize XML documents that has multiple root elements align at the same tree level, so that the axiom can even be $ELEM \rightarrow ELEM$ `OpenBracket` $ELEM$ `CloseBracket` together with the rules that identifies the parameters instead of a single tag descriptor. So, the axiom varies depending on the structure of the XML document.

3.2 The XPath Query Language

XPath [26] is a language for addressing parts of an XML document. XPath operates on the abstract syntax tree of an XML document, rather than its surface syntax: it models an XML document as a tree of nodes. There are different types of nodes, including element nodes, attribute nodes and text nodes. XPath defines a way to compute a string-value for each type of node.

3.2.1 Data Model

XPath operates on an XML document as a tree: this model is conceptual only and does not mandate any particular implementation. There are seven type of nodes, but we use only the first four in our model representation:

Root Node The root node is the root of the tree, and it occurs only one time in the entire structure. Every other element node is a child of the root node.

Element Node There is an element node for each element in the document. The children of an element node are the element nodes, comment nodes, processing instruction nodes and text nodes for its content. Entity references to both internal and external entities are expanded.

Text Node Character data is grouped into text nodes. As much character data as possible is grouped into each text node: a text node never has an

immediately following or preceding sibling that is a text node. The string-value of a text node is the character data. A text node always has at least one character of data.

Attribute Node Each element node has an associated set of attribute nodes; the element is the parent of each of these attribute nodes; however, an attribute node is not a child of its parent element. Elements never share attribute nodes: if one element node is not the same node as another element node, then none of the attribute nodes of the one element node will be the same node as the attribute nodes of another element node.

Namespace Node Each element can have an associated set of namespace nodes, one for each distinct namespace prefix that is in scope for the element (including the XML prefix, which is implicitly declared by the XML Namespaces Recommendation). The document used in the following examples and tests do not use this notation, since we don't need to avoid name collisions or facilitate name recognition.

Processing Instruction Nodes This nodes generates instructions to create tags in XML. We choose not to deal with them since are seldom used with a few common exceptions, that are not included the case in exams in this thesis.

Comment Nodes Each comment in XML has a corresponding comment node. Comments are not common in social network data structures and semi-structured databases, so we decided to leave them out.

3.2.2 Lexical and Syntactical Structure

The primary syntactic construct in XPath is the *expression*, which is evaluated to yield an object which has one of the following four basic types:

- *node-set*, an unordered collection of nodes without duplicates
- *boolean*, true or false
- *number*, in floating point notation
- *string*, a sequence of characters

Each expression evaluation occurs with respect to a *context*, which consist in several components: the most important are a node (called *context node*)

and a pair of non-zero positive integers called *context position* and *context size*.

A particular kind of expression is the *location path* or *location step*, that selects a set of nodes relative to the context node. The result of evaluating an expression that is a location path is the node-set containing the nodes selected by the location path.

A location step is composed by three parts:

1. an axis, which specifies the tree relationship between the nodes selected by the location step and the context node

2. a node test, which specifies the node type and expanded-name of the nodes selected by the location step

3. zero or more predicates, which use arbitrary expressions to further refine the set of nodes selected by the location step

An XPath expression is evaluated with respect to a context node. An axis specifier (such as "child" or "descendant") denotes the direction along which the tree must be traversed from the context node. The node test and the predicate filter the nodes denoted by the axis specifier: the node test prescribes which is the label that all nodes navigated to must have, while the predicate consists of XPath expressions themselves that state some properties on these nodes.

Formally, we consider XPath expressions, P, specified by the nonterminals P,E,A and N. P is the axiom, and it generates all the child and descendant relationship between all the nodes. E stands for Elements, and it resolves into the character string representation of a tag name, while A is used for the attributes with the same purpose. N is used to produce wild card predicates (* and @*, that matches respectively any elements and any attribute node) together with the expression used to tell the query to search for the string contained between the tag that comes before the `text()` (or `text(s)`) terminal. In the latter, the string must match `s` in order to be considered.

The terminals are `s`, `text()`, `text(s)` and the wild cards `*` and `@*`. All of them targets a particular information contained inside the tree structure or, in the case of the wild cards, subtrees and list of attributes respectively. A special mention is required for the `/` and `//` terminal symbols, that are an alternative representation of their corresponding value "child" and "descendant"; this two predicates indicates how to search the element node that follows their preceding one. In the first case, the tag must be included inside the previous tag, more precisely being in the next tree level only; the descendant value instead forces the search for each element that is son of

the preceding tag, no matter how deep the tree develops.

It follows the grammar definition:

$P \rightarrow$ `/`N | `//`N | PP

$N \rightarrow E$ | A | `*` | `@*` | `text()` | `text(s)`

$A \rightarrow$ `/@`

$E \rightarrow$ `s`

Every location path can be expressed using the syntax derived from the previous grammar, from which we extracted a subset that we will use in the tests: although a subset, it will not inficiate the diversity of the executable queries, but rather eliminates the ones that are too specific for the case in analysis. The reasons why we use a subset and not the complete grammar lies in the fact that, first of all, we decided to exclude three type of nodes from the seven available, thus all the related rules that deal with them are not useful and can be removed. Secondly, XPath has rules that exploits relationships between nodes (like siblings or namespace related) that are not the main concern of the application since the input XML file is not structured in a way to consider them. Despite this exclusions, the XML grammar is sufficiently descriptive of the related language and the XPath grammar is able to generate a query that is able to satisfy the majority of the common request for which it is commonly used.

The following is the precedence table that describes the precedence relationships between all the terminals and nonterminals, to proof the OPness of the XML grammar explained before.

	OB	CB	OP	OCI	OCP	AC	$Infos$	$TERM$
$OpenBracket$	$\lessdot$	$\doteq$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$
$CloseBracket$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	ND	$\gtrdot$
$OpenParams$	$\lessdot$	$\doteq$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$
$OpenCloseInfo$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	ND	$\gtrdot$
$OpenCloseParam$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	ND	$\gtrdot$
$AlternativeClose$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	ND	$\gtrdot$
$Infos$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	ND	$\gtrdot$
$TERM$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$

Figure 3.1: Example of OPG for XML Grammar

XML Grammar and XPath Example *We show some query examples to better understand how this language retrieves information from an XML document together with an example of how the rules of the XML grammar are applied to recognize a document. Let's consider the following extract from an XML document:*

```
1 <breakfast_menu>
2     <food>
3         <name>Belgian Waffles</name>
4         <price>$5.95</price>
5         <description>
6             Two of our famous
7             Belgian Waffles with plenty of
8             real maple syrup
9         </description>
10        <calories>650</calories>
11    </food>
12 <breakfast_menu>
```

It consist in a root element called breakfast_menu that has a single child element called food. Food has 4 direct tag children, each one with a string between with some content related to the name of the tag. There are no attribute node in this example. During the parsing phase, the parser scans all the tokens that represents the elements of the document until one of the rhs of the grammar is recognized and can be reduced. Since there are no tags with empty information in this specific example, the algorithm scans the first two open tags that do not match any rule until, after the the third opening one, the rule that is recognized is ELEM $\rightarrow$ `Infos`*, that saves the "Belgian Waffles" string inside the proper data structure and pass it to the ELEM nonterminal in the lhs. Since nothing has been saved yet (thus there is no ELEM recognized and saved before the call of this rule), the ELEM* $\rightarrow$ `OpenBracket` *ELEM* `Closebracket` *rule is the one that matches the string. Depending on the query request, the proper data are saved and passed to the lhs nonterminal, that from now on will serve as element to join possible future other solution. Since there are two more strings surrounded by open tags that follows the one that has just been recognized, the ELEM* $\rightarrow$ *ELEM* `OpenBracket` *ELEM* `Closebracket` *rule is called two times, with the first ELEM containing the previous results (if any). The "food" and "breakfast_menu" tags contains no other tag and so no more information to be scanned, so again the ELEM* $\rightarrow$ `OpenBracket` *ELEM* `Closebracket` *is called two times, with the first ELEM token passing every time to the lhs the results from the tree tags analyzed before. It now follows some queries description and the results expected from their execution on the XML chunk.*

```
1 /breakfast_menu/food/name/text()
```

The first thing to notice is that the query ends with the `text()` *string, which means that the information we want to extract from the text is a string placed between two tags. Starting from the left of the query and the root of the XML document, we search for the breakfast_menu tag, a direct child named food and its direct child named name. After that, the string character, if any, inside the latter is the result we're searching for (in this case, the string "Belgian Waffles").*

Another example query could be the following:

```
1 /breakfast_menu/food
```

In this case, the query does not ask for a specific string but for the subtree structure that lies under the food tag, respecting the writing appearance. This is the result:

```
1 <food>
2     <name>Belgian Waffles</name>
3     <price>$5.95</price>
4     <description>
5         Two of our famous Belgian Waffles with plenty of real
              maple syrup
6     </description>
7     <calories>650</calories>
8 </food>
```

Another possible example is the following:

```
1 /breakfast_menu//text()
```

This case exploits the XPath characteristic of navigating the tree searching information based on relationship between elements with an height difference of more than one step. The request is again the string text between two tags, but the // token asks XPath to retrieve all the information that are descendant of the breakfast_menu tag, no matter how much deep in the tree they are. Thus, the results are the following strings:

```
1 Belgian Waffles
2 $5.95
3 Two of our famous Belgian Waffles with plenty of real maple
    syrup
4 650
```

3.3 Lexing and Parsing

Recap on Parallel Lexing Lexical analysis takes place before parsing and translation, and even if it is a common belief that it is a fairly easy and

less time consuming job compared with the following phases, in the case of operator precedence languages it often requires comparable effort. It is important to notice then that the gain in performing parallel parsing alone would be small without coupling it with parallel lexical analysis and pre-processing. Furthermore, lexical analysis is even better suited for parallel execution. However, to achieve this goal a few non-trivial technical difficulties must be taken into considerations and solved. In this section we present a the schema for parallelizing lexical analysis used by PAPAGENO, which can be applied to most programming languages. A distinguishing feature of this particular lexical analysis is that it produces a stream of tokens which, rather than being compatible with the original BNF of the source language, is ready to be parsed according to an "OP version" of the official grammar, thus yielding an advantage from both a performance, and an adaptation to OP parsing point of view.

The goal of the lexer is to recognize the lexemes in the source character stream and generate a sequence of tokens, removing the comments. It can happen that the lexical grammar may be not locally parsable in its immediate form and in most cases is ambiguous. Nevertheless, lexical analysis can be adapted for parallel execution and is a more natural candidate for efficient parallelization than parsing, which has to deal with the nesting of syntactic structures, as in fact it happened in practice. To achieve this goal, however, there is an issue that must be addressed.

Fortunately, it has already been solved: splitting the source text randomly into chunks to be processed by parallel workers may split a lexeme across different segments. and the results produced by lexers working on adjacent chunks will have to be reconciled to cope with this issue. The XML grammar we use and the way we split the file between threads allow the algorithm to find out the best way to positioning in the correct point before starting to lex. In fact, the starting point is chosen at random dividing the file size by the number of threads: all the threads except the first may start in a position that is not the start of a tag (the only one that is acceptable XML). To solve this problem, we simply start searching, for each thread, the next tag token that follows the random start position: whenever it is found, the algorithm marks it as the new starting. It doesn't matter what tag is chosen, since the objective of the lexing part is to recognize the tokens, no matter what kind.

We now illustrate the approach to parallel lexical analysis using the PAPAGENO code: we will show how a parallel lexer splits the source code into chunks, assigns them to different workers, and reassembles their partial outputs. We will also make use of a running example to better illustrate the various steps of the algorithms.

Table 3.1: Token and their corresponding Regular Expression

Token	Regular Expression
Lbracket	$<$
Rbracket	$>$
Lslash	$</$
Rslash	$/>$
Equals	$=$
Value	$''.*''$
Number	[0-9][0-9]*
Ident	$[a-zA-Z0-9_-.''=]+$
Infos	$[a-zA-Z0-9_:\#/,\&;.=-()@?[]+'\$''!]+$
Space	$(['' ''] \vert \backslash t)+$
Newline	$\backslash n$

The Parallel Lexing Algorithm The input file has to be analyzed by a lexer in order to retrieve all the tokens and pass them to the parser. The initial idea was to use the Flex program [27], already used by PAPAGENO to lex different kind of languages. Following the already developed parallel lexer for the JSON language, we found out that the overhead of the re-entrant scanners created some unexpected bottlenecks with respect to the simplicity of the XML grammar we were analyzing that suggested us to build our own version for this specific example. In order to be clear, we present in detail the lexical grammar used by the lexer to recognize the token: excluding the `Infos` token, all the other terminals are defined using a composition of different terms defined in turn by using regular expressions. Table 3.1 lists the name associated with their values: this terms are used then in Table 3.2 in order to define all the terminal of the grammar.

In order to avoid switching context problems that happen frequently in a multithreaded environment, we use a technique called *CPU pinning*, or *processor affinity*, that enables the binding and unbinding of a process or a thread to a CPU or a range of CPUs. In this way, the process or thread has a pool of preferred CPU from which the Operating System can choose to assign it, rather than choosing at random, depending on the state of the system. Processor affinity takes advantage of the fact that some remnants of a process that was run on a given processor may remain in that processor's memory state (i.e., data in the CPU cache) after another process is run on that CPU. Scheduling that process to execute on the same processor could exploit it by reducing performance-degrading situations such as cache misses.

It's important to point out that each thread does not open and read the input file directly but instead, a *memory mapping* is used: the input file is written in the system memory region using a POSIX function called `mmap`. It creates a new mapping in the virtual address space of the calling process, specifying the starting address and the length of the mapping: after it is created, it can be accessed just like an array in the program. Each thread maps the entire file in an array and not its relative substring since the it cannot be done. This however is more efficient than reading or writing a file by means of a stream, as only the regions of the file that a program actually accesses are loaded directly into memory space.

Once done, each thread prepares the arguments that will use during the lexing process: a unique *thread identifier*, used to associate each CPU with a specific thread (CPU pinning), the file name to open, a preallocated memory space where to write all the read characters, the lexing stack and finally the cut points where the file is supposed to start and end its lexing. The latter are two precomputed integers that show to each thread where in the mapped array, representing the input file, they have to start lexing and where they have to stop. This two values vary depending on the number of employed threads, but there is a warning mechanism that stops the lexing process if the dimension of the file is too small with respect to the amount of thread that are employed. This because it is required a minimum amount of characters per thread to be analyzed, otherwise the thread creation and initialization with the sole purpose to handle a small amount of character could easily become the bottleneck.

The Custom Lexer Reached this point, each thread starts its own task: it starts by opening the file, mapping it into the memory and it reads the array from the beginning cut point. As stated before, a custom lexer has been written to ensure that only the fundamental operations (reading characters

Table 3.2: Terminal description using the Regular Expressions of 3.1

Terminal	Regular Expression
`OpenBracket`	Lbracket Ident Rbracket
`CloseBracket`	Lslash Ident Rbracket
`AlternativeClose`	Lbracket Ident Rslash
`OpenParams`	Lbracket Ident Space Ident Equals Value Rbracket
`OpenCloseInfo`	OpenBracket CloseBracket
`OpenCloseParam`	OpenParams CloseBracket

and recognizing tokens) are carried out and nothing more.

The token recognized must be saved in a dedicated structure that the parser will use after to retrieve the information read: the token value, that is assigned by PAPAGENO to each terminal when it generates the C code related to the grammar, and the semantic value that is the string representation of the information.

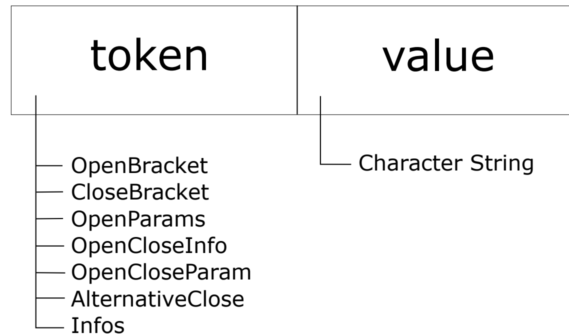


Figure 3.2: Description of a Lex Token structure

Flex operates by writing the regular expression that are used to find the tokens inside the input file. This tokens can be part of a larger regular expression and composed to form other tokens: together with them there are the lexing rules, that can represent a single or a composition of regular expression. Once recognized, the C code assigned to the rule is executed, the information and the token are saved inside the `lex token` structure and in the array that saves each character.

Algorithm 1 Custom Lexing Algorithm

```
 1: Input: map, current position, flex token, array memory pointer
 2: Output: lexing result
 3: start ← array memory pointer
 4: read ← map[current pos]
 5: if read not NULL then
 6:     Switch read do
 7:     case '<':
 8:     read ← next
 9:     if read = '/' then
10:         flex token ← CloseBracket token + semantic value
11:     else
12:         if read = letter or number then
13:             flex token ← OpenBracket or OpenParams token + sem. value
14:         else
15:             flex token ← AlternativeClose token + semantic value
16:         end if
17:     end if
18:     case space:
19:     while read = space do
20:     end while
21:          return NEWLINE OR SPACE
22:     case newline:
23:     while read = newline do
24:     end while
25:          return NEWLINE OR SPACE
26:     case any other symbol:
27:          while read not '<':
28:               array memory pointer ← read
29:               current pos ← current pos + 1
30:               read ← map[current pos]
31:          flex token ← start
32:          return LEX CORRECT
33: else
34:     end of the array, return LEX CORRECT
35: end if
```

Since we use a custom lexer, we have to write the code part necessary to recognize a regular expression, which can be more tightly specialized with respect to the general purpose recognizer emitted by Flex. The lexer receives some input variables prepared by each thread: *map* is the array

representation of the portion of the input file to scan, *current position* is the integer pointing the map array where the current character read is located, *flex token* is the structure in charge of saving the token and its semantic value and finally the *array memory pointer* saves the address of the map before any scan starts, in order to save it inside the flex token in the case of successful token recognition. The local variable are *read*, which saves the current read character and *next* that represents the character following the current one. The only one output variable is the result of the lexing phase for the current read token, and their value are further explained next.

The lexer works as follows: when it is called, the current position in the map array is saved in memory, to return it as the address pointing to the beginning of the token and the first character is then read (line 2-3). Then, a sequence of switch cases finds out which of the terminal the token belongs to: in case of a space (18-21) or a newline (22-25) character, the algorithm scans the input until a different one is found, and then returns the NEWLINE or SPACE integer to the lexer. Lines 27-33 shows the case of string character recognition between two tags: the scan continues until a '<' token is found (i.e. the closing tag is starting), then the starting pointer is written in the semantic value field of the flex token structure, together with the *Infos* token name. Lines 5-17 regard about the recognition of different kind of tags: the starting '<' is mandatory (7-8) followed by some if statement to discern the type of tag. (9-10) regards the close tag, easily recognized by the immediate '/' that follows the opening character. Letters and numbers instead identify an opening tag (12-14): the presence of parameters is related to the appearance of at least one space that separates the characters inside the tag name, plus a series of one or more *parameter="value"* lexical structure to indicate the parameter name and its corresponding value. Excluding all this cases, the remaining one (14-16) is the Alternative Close, that relies on the presence of a '/' before the '>' token.

No copying method is applied when saving onto the flex token structure in order to avoid too many reading and writing memory accesses: instead, the starting address is pointed to retrieve the different strings inside the array of tokens. Because of that, an end-of-string delimiter character (i.e. the one with the ASCII encoding 0) has to be inserted between the end and the start of a new token in the lexing stack array, to avoid reading over the expected string end when it will be used in the parsing phase. This is true for each token (i.e. every token has a separator with respect to the one that precedes and that follows it).

Depending on the result, some predefined integers are return to let the thread know what happened during the lexing of the mapped array:

- *LEX CORRECT*: the token read has been recognized and saved cor-

rectly in the lex token structure

- *ERROR*: the token read did not fall into any of the cases, resulting into an unrecognized token. The lexing is stopped and the program aborts, pointing out where the problem has occurred.

- *NEWLINE OR SPACE*: spaces and newlines are not considered as part of the XML grammar and can be simply skipped. Every time one of the two character is found, the lexer starts skipping similar characters until a different one is found, in order to avoid counting them, and then finally returns. Thus, we are sure that the next character will be a non-space, non-newline one.

Whenever a LEX CORRECT value is returned, the `lex token` structure is saved inside the lexing stack provided to the thread with a push function. The lexing stack is further analyzed in section 3.6.1.

After the portion of input file assigned to each thread is finished, the thread task saves in the struct passed as argument the address of the beginning and the end of the token list; this, together with the other results coming from the other threads, helps the recombination of all the results in a single array, that will be used by the different threads to parse the content of the input file.

Example *Let's consider a small input file as the one following.*

```
1 <a>
2     <b>
3         <c>text</c>
4     </b>
5 </a>
```

Let's suppose a 2 thread lexing phase. First we have to find the starting point for each thread that is not the first one (that starts at the beginning of the file). Indentations are considered as 4 spaces characters and newlines as 1, so in this example there are 45 characters to be lexed. The half split occurs after the last 't' character of the 'text' word, so the thread 2 will start by reading the next character, that fortunately in this case is the open tag. If that wasn't the case, it would have scanned the mapped array until an '<' character would have been found. Following the first thread execution, the first character scanned is '<': now the lexer reads all the following characters until a closing tag is found, paying attention to actually recognize the right kind of tag. Since the next char is a letter, it's not a closing tag as so it cannot be a CloseBracket token, nor can be an AlternativeClose one because / is missing : the next thing to check is if there are some spaces somewhere after

this character during the scan that leads to the closing tag. Because it's not the case, and the closing tag is the following character, it's a case of OpenBracket without parameters. The 0 *character is inserted after the closing tag, the semantic value points to the start address of the string (that would eventually move towards the next free character in the memory allocation array), and the token field is filled with the OpenBracket integer representation. Now the LEX CORRECT result is returned, the thread pushes onto its stack the recognized token and moves towards the next available character.*

Preparing the threads The threads must be prepared as the lexer thread did. First of all, since the different arrays produced by the lexing threads have been merged together in a single one, a new bound computing operations must be made in order to assign to each thread its amount of tokens to be parsed, without caring where the cut is made, since the structure of the OP grammar is such that the recombination is possible without any sort of problems. Thus, a simple division of the total number of tokens by the number of threads is sufficient.

Then it follows the memory allocation: all the data structure necessary to save the relevant information (see section 3.6.1 to know their functionality and reference) are preallocated here before calling the threads. The `perform parser memory allocation` function computes the dimension of the file and declares the required local variables to pass as arguments to the threads to avoid cross references between them and force serializations. This variables references are pointed by a global structure that takes care of freeing them at the end of each task; there is also the *starting* data structure each leaf node is compared against and global integer flags to help the parser to distinguish between different kind of query without checking them every time. The sections 3.6.1 and 3.6.2 explains in details every field of the previous discussed data structures.

The recombination of the results coming from different threads is performed by extra threads that are created depending on the recombination strategy chosen when PAPAGENO is launched the first time:

- *Single*: a single extra thread is created, that joins all the results of the others
- *Log*: instead of a single thread, a log_2 number of threads are generated. Whenever the original threads finish their first pass, another series of passes is done, each time halving the number of thread used.

At the end, the bounds, together with the general parsing context structure and the argument structure that contains all the pointers related to the preallocated memory space are passed as arguments to each thread.

The Parsing Algorithm Each thread initialize its own stacks where to save the partial results of reductions and where the final results will be found. A look-ahead pointer is maintained since at each step a `get precedence` function is called to know which precedence relation is held between the current token and the following one. Based on the results, two things can happen:

- EQ or LT precedence token: the action to perform is a *shift*. The token is pushed into the final parsing stack and the yield precedence stack if the precedence is LT.
- GT precedence token: the action to perform is *reduce*. With this token, it has been found the end of a substring that could be parsed by a still to be known rule and reduced to its lhs.

During the reduction phase, the reverse procedure takes place: the parsing stack containing all the tokens that has been shifted are now popped out until the starting one is found, that is the one pushed into the yield precedence stack first. With the starting point retrieved, the reduction phase can decide which semantic rule fits most and has to be called on it; starting from the yielded token, the algorithm scans the candidate rhs and matches it against the reduction trie.

With the reduction list ready, the `call semantics` function is now able to select the correct grammar rule. The semantic function and some of its actions are partially generated by PAPAGENO, in particular the declaration of a variable for each terminal and nonterminal in the rhs and lhs, the push of the lhs into the stack and finally, from the starting point of the reduction obtained in the yield precedence stack, the attachment of all the rhs token respecting the relationship of parent and child between the tokens. The lhs became the parent of all the rhs tokens, the following token of the lhs will be the one located next to the rightmost of the rhs list, and the first token on the rhs became the direct child of the lhs.

After that, it comes the semantic action, as explained in the following section.

3.4 Semantic Actions

As explained before, PAPAGENO generates some common actions for each rule, that can be enriched by adding some semantic action, that composes the main part of the algorithm used to extract the query results from the input file. In fact, the grammar used is an *attribute grammar*, where all

the semantic actions involved are done whenever a grammar rule is recognized in a bottom-up technique, starting from the leaves (in this case study, the information contained in the XML tags) and continuing by passing all the computed information to the parents until the root is reached, giving the final results.

Thanks specifically to their nice closure under the substring extraction, the XML OPG grammar allows to compute the semantic actions separately and join the results in the same way it's done with the parsing process. Each rules have its specific semantic action that extracts the values from the leaves of the tree XML structure and checks whether it is query compliant as well as parsing it.

Algorithm 2 General Semantic Action Algorithm

```
1: if ELEM in rhs not NULL then
2:     path matching()
3:     pass to lhs()
4: else
5:     pass NULL
6: end if
```

The idea behind this algorithm is simple: during the OP parsing of the document, each time a reduction is performed it checks whether the tokens that have been reduced correspond to nodes that might satisfy the pattern of the XPath expression. Since the semantic processing is paired with the bottom-up OP parsing algorithm, the pattern matching against the expression is performed bottom-up too; thus initially it checks whether the reduced tokens correspond to the location step at the end of the path of the expression. If this is the case, it computes which is the rest of the path that the upper part of the parsing tree must satisfy so that the reduced tokens are actually a correct match for the query. This information is propagated to the nonterminal that is the lhs of the rule used in the reduction, which will replace the rhs on the stack of the parser, and the pattern matching continues thereon.

In this way, while parsing the XML document, the information are checked against the query: the one that can reach the root of the tree and the start of the query path at the same time are recognized as solution to the query statement and saved in an array for further use.

The General Semantic Action Algorithm [2] shows a really simplified version of a semantic action: in case of a failed match, the NULL token is passed to the lhs, so that the subtree parts to let future rules know that from that specific subtree there are no relevant information to keep track of. In

case of not null information, they have to pass through a fine grained filtering function called *path matching* that covers all the possible combination that the current information and query may ask to check.

The focus in this algorithm is the **path matching** function, that is further explained in section 3.5: all the data structures referenced inside the description of each of the following rules are in section 3.6.1.

Semantic Rules Descriptions Each rule implements a specific version of this algorithm, depending on the rhs tokens positioning. It now follows some further explanation of all the semantic actions.

ELEM → Infos

Algorithm 3 Retrieving tag information Rule

1: **Input**: token semantic value
2: **Output**: data structures passed to lhs
3: **if** Infos is a query end predicate **then**
4: extract new path()
5: extract information()
6: set flags()
7: **else**
8: **if** Infos is an attribute **then**
9: extract new path()
10: extract attribute()
11: set flags()
12: **else**
13: extract new path()
14: extract information()
15: set flags()
16: **end if**
17: **end if**

The Info Rule [3] is the most important of all, since it is one of the first that is called and that pass to the lhs all the necessary information in order to further analyze its content in the following rules. As explained in section 3.2, all the query statements ends in ways, that creates different cases inside the semantic action, here analyzed in details:

- *text()*: a *destination* structure is retrieved from the memory pool to create the node structure representing the leaf. The second step is to recognize a child or descendant relationship between the query end

and the preceding string (e.g. the amount of '/' token that separates the two); at last, the information in the rhs is saved, together with the corresponding flags, and passed to the lhs (3-7).

- *text(S)*: like the previous case explained, with an additional check on whether the information in the rhs is equal or not to the S string.

- *@id*: the query end targets an attribute. Since the attribute is found inside the tags, this rule is not able to retrieve the data, that is surely located in the next rule that would be called. Since the name of the attribute to find is located in this particular query end and it couldn't be passed (if the rule of the separators preceding the query end says so), it's safer to save it in the info field of the structure and pass it to the lhs: the flag variable would inform the next rule that the data in the field is not a true information but an attribute index, to be used to check if the real information is present or not (8-11)

- *tag name*: the query ends with the request of a tag, thus all the subtree structure that lies under it. It differs from the query end predicate in the flag to be set, that would inform the next rule to append all the leaves in a single structure in order to recreate the subtree, paying attention to maintain the order of the element in the list (13-15).

There cannot be other possibilities apart from the four mentioned, making the rule consistent as it always returns a coherent result to the following rules (NULL is the value that tells that nothing has been passed).

Algorithm 4 First Rule

```
1: Input: ELEM data structure, tag tokens
2: Output: data structure composed and passed to lhs
3: if both ELEM in rhs not NULL then
4:     path matching()
5:     if infos has to be concatenated then
6:         concatenate()
7:     end if
8:     remove unused elements()
9:     append ELEM infos()
10:    pass to lhs()
11: else
12:    if Second ELEM is NULL then
13:        pass to lhs()
14:    else
15:        if First ELEM is NULL then
16:            call OpenBracket ELEM CloseBracket rule
17:        else
18:            both NULL, pass NULL
19:        end if
20:    end if
21: end if
```

ELEM → ELEM OpenBracket ELEM Closebracket The rhs of this rule contains two ELEM nonterminal tokens, that refers to specific part of the text: the first one always contains the already processed and checked information coming from previous reduction done on the tree before this rule is called, and for this reason, this rule would not be called as first. The second ELEM token is by grammar construction (3.1) one among *Infos*, *AlternativeClose*, *OpenCloseInfo* and *OpenCloseParameter*. The most important part to notice however is the fact that this second nonterminal is surrounded by a pair of terminal tokens that contains the opening tag and its corresponding closing one. As shown in section 3.5, the path matching algorithm would check if the *ELEM* token contains an element that satisfies the surrounding tags, and if it is the case, passing to the lhs, together with the first *ELEM* token, the necessary information.

The first branch of the algorithm (1-8) is called when both ELEM tokens contains something (i.e. a data structure that has information inside them). After the path matching, in case of subtree requests, all the information that passed the test against the query end contained in each of the node info structure of the second ELEM token are extracted and concatenated into a

single one, then surrounded with the tag terminal tokens (usually, the first). The other ones are flagged as removable, in order to reuse them as soon as possible and avoid passing to the lhs a useless heavy data structure. In the second case of specific string information requests, all the nodes that passes the test (and that are not removed by the corresponding removing function) are linked together by the appending function, and passed to the lhs as a linked list, together with the ones inside the first ELEM terminal token.

To be more precise:

- the *remove elements* function scans the list contained inside the ELEM token, checking whether each node info has the REMOVE ELEMENTS flag set to 1. If it's the case, the data structure is removed from the list and inserted into the corresponding pool, waiting to be reused, otherwise it is appended to the linked list of the first ELEM token (that can't be NULL, otherwise it wouldn't be in this if statement section). Even if scanning the list seems computationally heavy, the fact that this rule is one of the most called implies that the length of the linked list inside the ELEM tokens consist on average of only a single element. In fact, if the query targets a single string information inside a subtree, whenever an incorrect one is saved inside a structure and reaches this rule, it's immediately flagged as removable by the path matching function and it's pooled, never joining the other when passing to the lhs.

- the *append elements* function, as the name suggests, simply appends two linked list, maintaining the order of the various elements. As shown in section 3.6.1, the lists are optimized regarding the tail inserting, to avoid scanning long element list only to reach the last element.

The other branches of the if statement and are easier to manage and represent a special case of the one already explained, since they imply a NULL ELEM terminal token, either the first (13-15) or the second (10-12). In the first case, the same algorithm is applied except the appending function; in the second case nothing has to be done and the first ELEM token is passed to the lhs. The last possibility is that both ELEM nonterminals are empty (line 16): in this case, a NULL pointer is passed to the lhs.

ELEM → ELEM OpenParams ELEM Closebracket This rule has a small addition with respect to 4, to handle the parameters that follow the opening tag token: initially, a list containing all the attribute values, either correct or incorrect, is created. Then, the same procedure is applied as before. The list of arguments is then used by the path matching function in the case of attribute request by the query.

ELEM → ELEM OpenCloseInfo, ELEM → ELEM OpenCloseParameter and ELEM → ELEM AlternativeClose This three rules has somehow the same algorithm structure, that requires building the information starting from the terminal tokens, heavily depending on the query request. There is no information to extract between the tags, so in the case of a query end text predicate (i.e. the string information that sometimes can be found between tags), this rules basically do nothing but passing the elements inside the ELEM token to the lhs, ignoring the tags.

The case is different if a subtree it requested:

- OpenCloseInfo: the procedure is the same as the *ELEM OpenBracket ELEM CloseBracket* rule with both the ELEM token NULL. The opening and closing tags must be concatenated before everything else starts and after that, the same procedure is applied.

- OpenCloseParameter: the concatenation is performed in the same way as explained for OpenCloseInfo, with the addition of separating and saving all the parameters inside the open tag token as already explained.

- AlternativeClose: this rule differs from the other two because there is not a couple of opening/closing tag, but a single one, used to simplify and shorten the notation for particularly long tag entries. In this case, there is no need to concatenate any string, but as the other cases, the node info structure with the tag must be saved and passed to the lhs, if requested.

Other rules All the other rules are derived from the ones described before, with a missing *ELEM* token in front of them. This token gathers all the information on the same tree level that has been found and saved by the algorithm: every time a new subtree scan begins, there are no preceding elements to append information to, so one of this grammar rules has to be called to start. The following will have the prepending ELEM token until the entire subtree is scanned or another one is found during the analysis. It's important to notice that there are no semantic action differences between this rules and the one with the first ELEM token as NULL, but they're fundamental grammar-wise to let the parser recognize the language correctly.

3.5 XPath Query Algorithm

The *path matching* algorithm decides whether an information inside a token has the required properties, with respect to the current query path,

in order to be passed to the lhs of the rule and to proceed further up in the tree structure.

Algorithm 5 Path Matching Algorithm

```
1:  Input: input list, open/close tag, attribute list, args_t structure
2:  Output: input list modified
3:  for each node info in destination list do
4:      if node is a solution then
5:          flag solution()
6:      else
7:          if Query ends without a string tag id then
8:              analyze data()
9:          else
10:             if attribute requested then
11:                 Extract attribute()
12:             else
13:                 Extract data or subtree()
14:             end if
15:         end if
16:     end if
17: end for
```

Now it follows the complete description of algorithm 5.

The argument list The function needs several arguments in order to manage different possible situations:

- *input list*: a list containing all the current node info that has to be analyzed, corresponding to the ELEM nonterminal located in the rhs of the rule that calls the path matching function.
- *open/close tags and parameter list*: string representations of the tags content, included the list of parameters
- *argument list structure*: data structure with all the pool memory references, as described in section 3.6.

The output of the function consists in the same input list passed as argument, with each element suitably modified depending on the algorithm rules. It's important to notice that the list of node infos that requires a deep analysis of their content is always located inside the ELEM nonterminal that stands between two terminal tags; the other ELEM terminal that do not satisfy this

rule are simply a container of already analyzed node info lists that simply need to be passed to the lhs and their elements are not the target of this algorithm.

Checking the solution The first thing to verify is whether inside the current node there is valid solution to the current query path (4-6). If the path inside the node targeting the query end is empty or *//* (it can be verified using checking the corresponding flags value together with the corresponding field), the information are indeed part of the solution and the pointer to the entire structure will be saved in the array in the proper following function. The structure remains in the list but it's flagged in order to be ignored in the following scans, to avoid counting it more than one time. In case of attribute request, it is necessary to extract the value corresponding to the id found in the query and overwriting the info field of the structure before saving the pointer.

Analyzing the current information If the node info do not contain a solution, it's time to check for different possibilities, based on how the current query path ends. The first case (7-9) consists in a query that does not terminate with a character string identifying a tag (i.e. there is a separator token). It's important to distinguish two cases, based on what kind of separator terminates the query end string. In case it is *//*, it means that are required information that are currently inside the nodes under analysis that are descendant in any number of levels with the tags preceding the current query end. Now, the separator preceding the string that comes before the *//* characters (e.g. */q/b//* or */q//b//*) is the discriminating factor in deciding what to do with the current node information.

- single slash character (*/*): If the tags strings surrounding the node info corresponds to the one required by the current state of the query, then it must be passed to the lhs both the path *q* and the path *q/b//*. In this case the structure must be duplicated, by picking a pointer inside the memory pool and copying all the information besides the path, that is cut as previously described. Both the new and the old one are appended one after the other before passing to the lhs.

- double slash character (*//*): the path is *q//b//*. In this case there is no need of duplication, the path is simply cut (*q//*) and the node is passed to the lhs with the other fields untouched.

Query ends with no slashes, Data Extraction The query ends with no slashes (line 9), so a tag check is required against the information inside

the current node. Even here there are two different cases depending on what kind of / token precedes the query end string (line 13).

- single slash character (/): If the tag matches the terminal surrounding the ELEM nonterminal inside of which the current node info is located (e.g. */q/b or /q//b*), then the path has to be modified in *q* (or *q//*, resp.). The current information is between legal tags and it is recognized as possible solution (the corresponding flag is set too). The path and the current query end can be recomputed to prepare the node to match against the upper tree level tags, if any. If it's the case of subtree request, this branch handles the concatenation of the tags surrounding the information.

- double slash character (//): the same procedure as single slash character is applied, taking care of inserting a // character to the end of the path.

In case of attribute request (line 10), the same algorithm is applied, paying attention to extract and overwrite the attribute in the info field of the node info structure.

The Data Extraction function (line 11-13) handles the cases of wrong matches between tags and path too: the flag that recognized a node as a possible solution must be checked to decide what to do with the current information. If it was not set, it tells the algorithm that the current node could have possibly satisfied precedent reductions and it cannot be removed since the query requests data that are descendant (i.e. there is at least a // somewhere in the query path) and until this node doesn't reach the tags from which the descendant scan starts, nothing can be decided. Thus, even if wrong, the node info is passed unmodified to the lhs (but it will be checked as soon as it finds the descendant starting point, or discarded if not).

Instead, if the flag is set, the current node contains surely wrong data with respect to the query, and the removable flag will tell the `remove elements` function to discard it.

Attributes and Wild Cards The wild cards and attributes are handled inside all the function explained, since a simple check on the corresponding flag will tell the algorithm if the information to save is between the two tags or inside the opening one: in the latter case, there is no need to extract the string since they're passed as arguments in an array format to the path matching function.

3.6 Data Structures and Optimization

3.6.1 Data Structures

One of the most important part regarding building the lexer/parser algorithm resides in how the data are managed while walking the tree and how each thread handles the common data. The input file information must be saved in order to be processed by the parser and as the file grows bigger, so could the problems related on how to avoid slowdowns related to reading and writing on disk, or on inefficient data structures that do not exploits the low level cache hardware appropriately.

We're going to further analyze and enrich the algorithm part by explaining how all the data retrieved in the input file are saved and passed through the rules, what kind of data are used to minimize the allocation space in order to handle all the cases presented by the Path Matching Algorithm 5 and how to avoid wasting space by saving useless information that are known to be wrong during the parsing process.

The Node Information This structure represent the core of the algorithm data part: whenever a terminal token string information is reached, a **node info** structure is allocated and filled with the necessary data to let the following parsing rule know what to do with the newly received one.

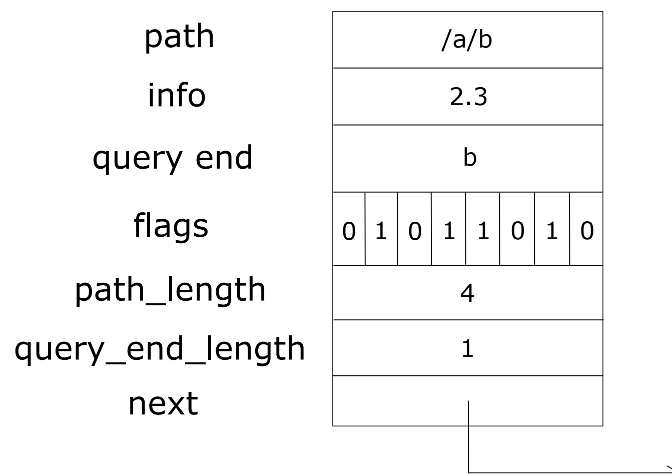


Figure 3.3: Description of a Node Info Structure

Referring Figure 3.3:

- *path* contains the current path of the query string. It is updated following the rules of Algorithm 5 and it is used as current reference by

the *path matching* function.

- *info* saves the string representation of the information. If the query asks for a subtree, it is built and saved here.

- *query end* saves the last part of the query path, since it would be computationally intensive to recompute it every time it is needed.

- *path length and query end length*, same as *query end*, saves time and speed up the parsing computation.

- *flags*: in this bit-packed variable are saved all the binary values that helps to identify which case of the Algorithm 5 has to be taken. All the semantic actions decide what to do based on the values contained inside the flags. To reduce the size of the entire structure, this variable is handled using bit-wise operations.

  The available flags are the followings:

  - *Already Associated*: if this bit is set to 1, the current structure has already saved its information inside the solutions array and must not be processed again.
  - *Is Attribute*: if this bit is 1, tells to the following rule calls that the information required is the attribute of a tag and not the string inside them.
  - *Tag Append*: if this bit is 1, a subtree request has been made and the information passed to the lhs must be concatenated with any couple of tags that contains it.
  - *Already Recognized*: this bit started with a 0 value. In the case of a query asking for a subtree structure, it can happen that the required subtree is not starting near the leaves. Since the query end string do not match the bottom of the tree, the information has to be discarded. However in this particular case, the subtree must be built anyway and passed to the lhs until a match between the query end and the current path of the structure is not found: this because a subtree request do not always involve the leaves and can start in the middle of the structure, and discarding it can be an error. In the positive case, we can start from there the comparison between the two strings, and we set this flag to 1 to let the algorithm know it; in the case of wrong match from here on, the bit is reset to 0. In the negative case, the structure is discarded when it reaches the root. If the flags arrives at the

root with this flag set to 1, the subtree is saved into the array of solutions.

This kind of computation is really intensive depending on the distance between the query and the tree depth. In general, requesting a subtree is computationally intensive and has to be done only for tags close to the leaves than to the root.

e.g. Suppose to have an XML tree with depth 4, and a query that asks the all the subtrees of level 2 in the form of /a/b. Since the algorithm starts building the node info structure bottom up, it checks first whether the information inside the leaves are surrounded by the query end tag, (b in this case). It is obviously false, but the structure must not be discarded until it reaches a tag that is equal to the query end. If it doesn't it is discarded. If it finds it, the already recognized flag is set and from there on the match can start until the root is find.

- *To Be Removed*: this flag is set to 1 during the *path matching* function call in case the information contained in the *ELEM* token is not correct with respect to the query end and the tag surrounding it. The *remove element* function is called afterwards so the structure has to be flagged in order to be recognized as removable. The bit can still be set for other reasons outside the one explained if there are some situations where the information must be removed.
- *Stop Concatenating*: if set to 1, the information contained in the structure doesn't concatenate any other tags in the path to the root. It is the case of queries that targets subtrees without a path that starts with the root element: since the parsing process continues in any case until the root element, setting this bit avoids concatenating further information.
- *Sep After and Sep Before*: this couple of bits tells the algorithm which and how many / character comes before and after the current query end. Since there are different paths that leads to different kind of solutions based on this properties, included the function that computes the query end string, it is fundamental to keep track of this characters during the parsing process. Since there are 3 kinds of combination possible for each property (/, // or nothing), four bits are required, two for each one.

This flag variable is 32 bit long and each of the previous bit has a macro function related that executes the bit-wise operation require to set and reset it.

- *next*: the structure is a common linked list, so that new information could be appended and could reach the root as a single unit. The linked list is preferred against an array since in case of subtree request it is really important to maintain the order of appearance of the elements, not mentioning that an array structure implies knowledge about how much space the solutions requires, that is not known until the end of the parsing process. Moreover, removing an element from an array do not reduce its size, so the risk is to scan a lot of empty values in search for the valid ones.

The Leaf List Since all the information passed through the rules are saved inside a list, it is really useful to wrap them inside a **leaf list** structure that keeps track of the last element of the list, to enhance the appending operations without scanning it every time. It is really time saving when the algorithm comes to and end and all the results coming from different subtrees must be merged in a single structure.

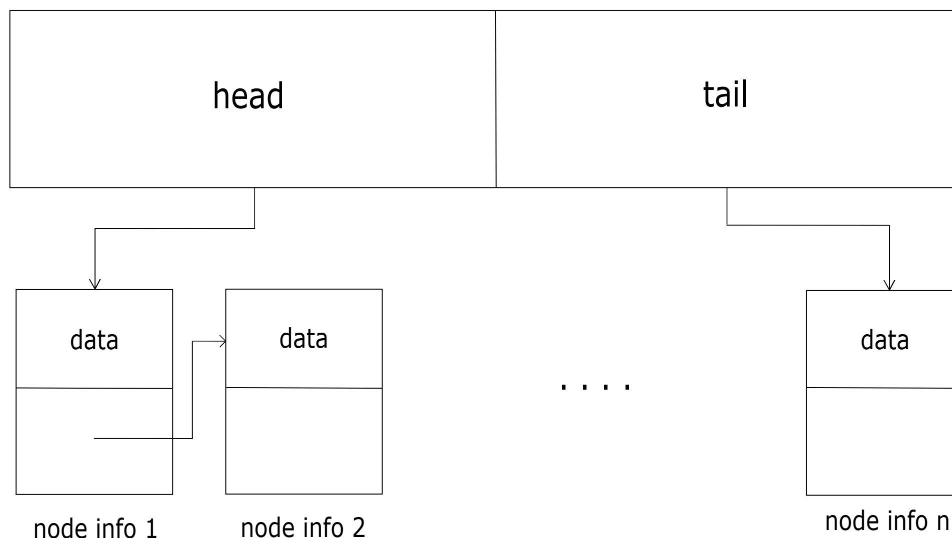


Figure 3.4: Description of a Leaf List Structure

3.6.2 The Memory Allocators

Lexing and parsing are not computationally intensive operations, and the semantic actions related to the latter are basically string manipulation that, even if the C language is not the best one to perform this kind of operations, are really basic and smooth. The real problem resides in how the information are managed in memory.

Since both the parsing and the lexing actions are executed by concurrent threads, it is fundamental to avoid sharing any kind of variable and/or environment information between them. At the same time, it's important to allocate the necessary memory in a way that it is not sparse, and exploits the L1 cache of each processor as much as possible.

Memory pooling The solution adopted is to preallocate the estimated space needed by the structures hosting all the input data. In fact, each information residing in the leaves of the input tree must be saved inside its proper *node info* structure before being analyzed as either a correct or wrong one. Thus, each information is stored at least once in a *node info* structure during the parsing process: needless to say, the amount of space required to save all this data is at least as much as the input file dimension. For bigger files this could be a problem, since the preallocation of a huge amount of memory could fill up the RAM space, leaving nothing for the computation part.

With the *memory pooling* technique, we decide to preallocate a certain amount of space, depending on the query request (i.e., for subtree request the amount is greater, since the solutions are subtrees structures) and to manage them in order to avoid filling them up and freeing them as quick as possible.

Algorithm 6 Memory pool Algorithm

```
1: Input: string information, pointer structure
2: Output: address of a free memory space
3: if memory pool is empty then
4:     Retrieve pointer from slots()
5: else
6:     Retrieve pointer from pool()
7: end if
8:
9: function RETRIEVE POINTER FROM SLOTS
10:     if Retrieve Leaf List pointer then
11:         save pointer()
12:         index ← index + 1
13:         ceil ← ceil - leaf list dimension
14:     else
15:         save pointer()
16:         ceil ← ceil - leaf list dimension
17:         info ← get pointer from slab()
18:         flags ← 0
19:         next ← NULL
20:         index ← index + 1
21:     end if
22:     return pointer
23: end function
24:
25: function RETRIEVE POINTER FROM POOLS
26:     if Retrieve Leaf List pointer then
27:         save pointer()
28:         index ← index - 1
29:     else
30:         save pointer()
31:         index ← index - 1
32:     end if
33:     return pointer
34: end function
```

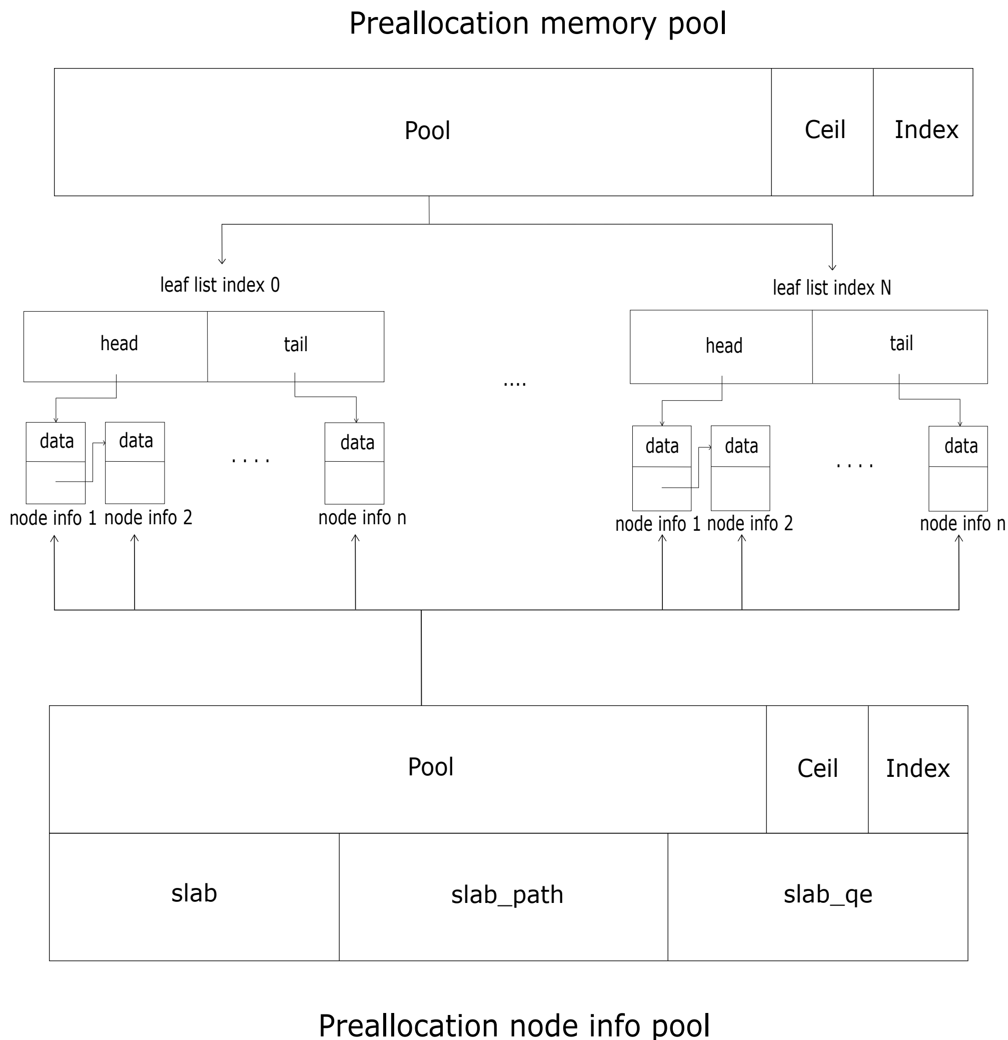


Figure 3.5: How the two preallocation pools are used to store Node Infos and Leaf Lists

As depicted, there are two memory allocation pools, one for the *node info* and one for the *leaf list* structure. Regarding the *leaf list* pool structure, a pool variable containing a single pool for each thread is created, with the index variable used to target the different ones and ceil to assure that the space allocated for each pool is never exceeded. In the second case, the *node info* pool structure, the idea is the same, with an addition: the space required by each single node info pool is preallocated too and assigned to three *slab* variables. Every time a new node info structure is required, the amount of memory where to save the information read are not requested by allocation functions in the thread (i.e. `malloc`), but are obtained by pointing the correct slab variable, that was allocated before launching the threads.

Algorithm 6 describes how memory is managed: every time it is called, both a leaf list and a node pool memory slots are granted to the caller. Lines (3-6) show how the algorithm search for available space: depending on the state of the pools, empty or not, different choices are made. Retrieving memory pointers from slots means that there are no already used and ready to be rewritten memory spaces in the pool, so the memory address must be retrieved from the preallocation slots. In case of leaf list, the address of the entire memory is granted to the caller and both the index identifying the memory zone and the ceil, that checks whether there is still available space, are updated (10-13). In case of a node info structure (15-20) the things are a little bit different, since it requires other preallocated space to fill up the *info* field (i.e. the slab, as depicted). Retrieving memory from the pools (25-33) is basically the same task both for leaf list and node info structures: the recycled memory pools are linked lists targeting the reusable locations inside the preallocated pool memory. The name of this recycling structure is **memory leaf pool**, defined in Appendix A: the main difference with the corresponding preallocated version is its purpose: the preallocation pools are used to allocate space that is used to contain useful information (i.e. the ones that would results in final solutions) and temporary ones, thus data that are not yet recognized as solutions or not. The memory leaf pool instead is a temporary space for wrong/discarded structures. As further explained in the following sections, allocating and freeing data are actions that require synchronization and they force a sequential execution: whereas we need to avoid this kind of situations inside the threads executions, freeing a structure is something that we don't want to do at all. This memory leaf pool is simply a preallocated memory structure that saves the pointer to the data structure that are recognized as removable; instead of allocating other memory and force the synchronization, the algorithm searches first if there is some pointer that could be reused and returns it to whomever it calls.

3.6.3 Techniques To Avoid Serializations

One of the problems that we faced during the implementation of the algorithm was to enhance parallelism by removing as far as possible all the serialization from the source code. The memory allocation was first implemented using memory allocation functions (*malloc*) wherever they were useful inside the thread task: this was immediately spotted as the bottleneck of the entire procedure, since allocating memory requires a system call that is wrapped around a locking system that denies at all any kind of parallelism. In this section we analyze all the enhancements that lead the algorithm to

be parallel efficient at its best.

Preallocation Firstly, the memory allocation functions must be extracted from the thread tasks: a single call force the operating system to lock the context of the function call and to execute it in a sequential way. So memory pools are required, as explained in section 3.6.2.

Each thread, both for lexing and parsing executions, prepares all of its content before starting, and it is required that the structure and data passed are not global and/or shared among other threads. This because the threads must believe they're working on their unique memory pools without depending on external data, that would require a change of context or a cache miss since surely the data required are not loaded in the nearest CPU memory.

The preallocation phase is done both in lexing and parsing, and it consists in multiply calls of memory allocation functions like `posix memalign` (preferred to the malloc since it forces the alignment in memory), to fill all the pools depending on the query request and the dimension of the input file. When all the structures are ready, each thread receives its own allocation space and it uses it for its entire duration. Since the preallocation pools save only the pointers, their dimension is less than the preallocated memory required to store the field of the node info structure. We choose 10 MiB of space for the pointers, that is divided between the number of threads: experimental results show that the amount of space required depends a lot on the type of query request and the dimension of the file. Regarding the slab memory space that saves the node info structure fields, the amount of memory changes depending on the query request: 200 characters of space is required for each node info in the case of a query end text predicate, while 65000 instead for a subtree. Thus, 100 MiB of space, to be divided between threads, is the amount of memory available to store all the possible results from the computation. There are some assertions in the code to help the user to know if the query requested requires an amount of memory that exceeds the available one.

e.g. *Let's suppose to have an ELEM $\rightarrow$ Infos semantic action call. The first thing to do is to allocate the memory for the structure that would host the information found inside the leaf. Instead of locally allocating memory, the algorithm searches in the memory leaf pool if there are some free pointers to use: if there are, the old and useless data inside is freed and it is assigned to the caller. If the pool is empty, a new structure is required, but instead of allocating him (and forcing the serialization), the address of the first preallocated leaf list inside the memory pool is assigned to the caller. Once done, the algorithm could start checking the current string information against the*

query requirements and fill the data space.

It's useful to notice that with this kind of data structures, the common pattern in case of query end text predicate is designed to reduce the space required to handle the tree structure. In this case, each subtree contains only one information that has to be retrieved while all the others can be discarded. The common pattern is to allocate the space (thus, retrieving the first free address in the pool), filling it with data and then check against the query end; if it's not the correct one, that memory space is returned into the pool and reused in the future allocations.

This continues until the desired data is found: at that point, a new address is used multiple times until another correct data is found and so on. Even if the amount of memory is sufficient to write the entire tree, the pooling algorithm favours the reuse of the same memory location to exploit the locality of reference. Furthermore, each memory space occupied by correct information is written subsequently one after the other, helping the linked list structure, whenever it's the case of scanning it entirely, to exploit the the same principle of locality.

Global Variables The use of assignable global variables is not recommended in multithreaded applications, since their values are not reliable across all the threads, even more if they're going to be edited. However, there must be some kind of global reference since all the memory allocated space must be freed at the end of the parsing process to avoid huge memory leaks. Since each thread receives its own sub-pool space, it is not able to free the array of pointers where its sub-pool is located, and passing as argument the complete array implies that each thread would have a reference to a variable that is shared among all of them, with the risk of serialization and cache missing. The action of freeing all the memory initially allocated is done at the end of the parsing step, and it requires a minimum amount of global reference to reduce the passing of useless pointers between all the function during the parsing execution: the idea is to give to a particular structure the task of saving all the pointers that points to the all the memory allocations spaces, without let any other part of the code modifying them. With this in mind, at the end of the parsing phase, all the frees are called as once in a single point, removing all the possible leaks.

lexing stack	parsing stack
leaf pool	node pool
prealloc memory pool	prealloc node pool
read token	slab pool pointer

Figure 3.6: Argument Structure containing all the pointers that will be freed at the end of the parsing phase

It follows the description of the fields inside the structure, depicted in 3.6:

- *lexing and parsing stacks*: the lexing array saves the tokens lexed by each thread; the final array is then formed by appending all this arrays into one (there is a reference to the tail of each, so it's not a heavy computational process). The parsing array save the stack of each thread during the parsing process; the partial results, depending on the recombination technique used, are joined by the last threads and the final result is computed.

- *leaf and node pool*: this is the reference to all the memory pools containing effectively the solution of the parsing phase.

- *prealloc memory and node pool*: this is the reference to all the pools used to save the partial results of all the computation of all the information of the input file.

- *read token*: this array contains all the tokens read from the input file. It is the join result from all the lexing stacks.

- *slab pool pointer*: it is the array of pointer where all the memory pools are allocated to. The difference with the leaf and node pool pointer is that the latter can be modified and the starting address could be lost in the computational process.

This structure saves all the listed structure starting addresses as soon as the main procedure compute them, before calling each thread: the latter receives a copy of the pointers contained here, in order to let each thread modify them to insert and remove elements, without losing the starting address. When each thread ends the lexing and parsing execution, the main thread can start freeing all the data structures initialized at the beginning : using the addresses computed by each single thread would lead to errors since

they point to the first free space available, while the starting address point is needed in order to free them correctly. At this point, *args_t* is convenient since a series of cycle can be called to free each structure, using the total number of threads as an index to know when to finish. This procedure is done after computing the total amount of time employed by lexing and parsing, since freeing such amount of structure could inficiate in a relevant way the final results.

Besides, there are other global variables that act as immutable flags for each thread:

- the number of threads that are actually executed
- the allocation size of a single node info structure (it depends on the kind of information requested)
- a binary value to help the algorithm to know if the query is searching an element starting from the root or the tree or not

A special mention goes to a particular node info structure that represents the starting point to each leaf. In fact, each node info leaf is created from this *starting* structure, that contains nothing but the path of the complete query. From that, each leaf information is compared using this path and crafted accordingly to the results and the tag surrounding it. *Starting* is never modified since each leaf data is written directly into the memory pool assigned, with the exception of path that is computed cutting the *starting* path according to the query request.

CPU pinning As explained is the 3.3 section, CPU pinning is useful to let each thread believe that it has a single CPU to work on, avoiding the others to accidentally load all the context information in order to continue the suspended work.

4 Configuration and Experimental Results

In this section, we present and discuss the experimental results of parallel lexing and parsing system on XML language.

The Benchmarks The input file chosen for the benchmark comes from the Twitter Logs, but in recent years the company decided to switch from XML to JSON representation, so for this case we had to build manually the input file. Starting from an XML representation of some tweets in XML format, we concatenate to create file of different dimensions, to simulate different loads. Unlike in PAPAGENO, as explained in the relative paper, where the only action to be performed in the parsing phase is the construction of the Abstract Syntax Tree (AST), the semantic actions are now performed during the building of the tree and have a significantly impact. Moreover, as we developed a custom XML parallel lexer, we show the achievements gained with respect to the reentrant Flex scanners. To conclude, both lexing and parsing time are considered in the evaluation of a complete run.

We tested our parallel parser and lexer against files of different dimension, respectively 1 MB, 3 MB, 10 MB, 30 MB, 100 MB, 300 MB and 1 GB. Moreover, the Treebank dataset [28] is considered to reflect a typical XML schema and as a comparison with other practical solution that can be found in literature. Treebank is an XML Data Repository that contains different kind of documents with variable dimensions: some of them can be used as real life examples of how an XML document could be structured or to test different tree structures without caring too much about the content. It is used by the PP Transducer as test example, by downloading and concatenating an 83 MB file multiple times in order to obtain different size comparison examples, and so we do in order to have a fair comparison. However, the difference between our examples and the Treebank ones resides in the content:

the former describes a real Twitter XML log file, with strings information and tag name coherent with the English language and a tree structure with a limited depth (i.e. no more than 3 levels). Treebank benchmarks instead, combines different characters, even illegal ones for the XML language, to describe the tags, and the content of the majority of them is empty. The purpose of the latter for PP Transducer is to provide a tree structure with a very long depth and variable structure that does not focus on the content.

Hardware Platforms To evaluate the practical speedups obtained, we use three different platforms:

1. A quad-Opteron 8378 host (named opteron-server), thus amounting to 16 physical cores (4 cores per socket): the Opteron 8378 CPUs are endowed with independent, per CPU, L1 and L2 caches, and a chip-wide shared L3 cache. The host runs Ubuntu Linux 14.04 (x86 64 architecture) server and is endowed with enough RAM to contain the whole AST generated during the parsing process and token list. The purpose of the evaluation on this platform is to highlight the scalability of our approach, even in the context of a multi-socket system with a non uniform memory access, together with the power7-server.

2. A quad-core Asus K550J notebook, thus amounting to 4 i7 4710HQ physical cores (4 cores per socket): the CPU's are endowed with independent, per CPU, L1 and L2 caches and a chip-wide shared L3 cache. The hosts runs Ubuntu Linux 14.10 (x86 64 architecture), with 8 GB DDR3 RAM (able to contain all the data structure used in the examples). The cores use the hyper-threading technology, thus 8 logical cores are at disposal.

3. A PowerLinux 7R2 server (named power7-server), a Linux only 2U rack-mount server with two processor sockets offering 16-core 3.6 GHz and 4.2 GHz POWER7+ configurations. It supports a maximum of 16 DDR3 DIMM slots, with four DIMM slots included in the base configuration and 12 DIMM slots available with three optional memory riser cards, allowing for a maximum system memory of 512 GB and it is specifically designed for emerging workloads that are proved ideal for a virtualized scale-out Linux server environment. More details are shown at [29]

All the executable binaries have been produced through clang 3.5.0, based on LLVM 3.5.0, employing standard release grade optimization to obtain an efficient binary (-O3 -march=native optimization options). All

Table 4.1: Total text analysis times of the XML test-bench files, sequential execution, with 0.5% of standard deviation

Input Size	Elapsed Time [ms]					
	16cores server			4cores notebook		
	Lexing	Parsing	Total	Lexing	Parsing	Total
1 MB	33.7	19.6	53.4	10.8	4.7	15.6
3 MB	60.2	41.6	101.8	025.5	13.1	38.7
10 MB	159.9	116.3	276.3	087.5	41.9	129.5
30 MB	445.6	333.8	779.5	253.8	122.1	375.9
100 MB	1391.9	1038.8	2430.8	792.2	389.4	1181.6
300 MB	4164.5	3240.8	7405.4	2414.6	1166.7	3581.3
1000 MB	14303.8	12511.6	26815.4	8935.0	4691.3	13626.4

the timing results presented have been collected employing Linux real-time clock primitives, and are the average of 30 runs to reduce measurement noise.

Sequential Execution Table 4.1 reports the absolute processing times obtained on all the XML input files, using a purely sequential PAPAGENO lexer-parser pair: we use them as a practical baseline for comparison. A point worth noting is that the time spent in the lexical analysis of the input is non negligible: more specifically, it is around 70% for 16 cores machine and 80% for the notebook. This result substantiates the previous claim that, for OP-based parsing, the lexical analysis accounts for a non trivial amount of the text processing time. This tests are made using Flex as a scanner, and the results suggests us to develop something customized for this examples, since the scalability after the 300MB file was incoherent with respect to the expected theoretical results. Moreover, the lexing phase results incredibly time consuming with respect to the parsing phase, that contains semantic actions but nonetheless exhibits a faster behaviour.

Obviously, since the lexing phase is not executed in parallel, the total amount of parallel portion code and speedup of the combined phases are not a relevant index for now.

4.1 Lexing

Parallel Execution With Flex The next move was to extend the lexer with a parallel execution algorithm, using the reentrant scanners offered

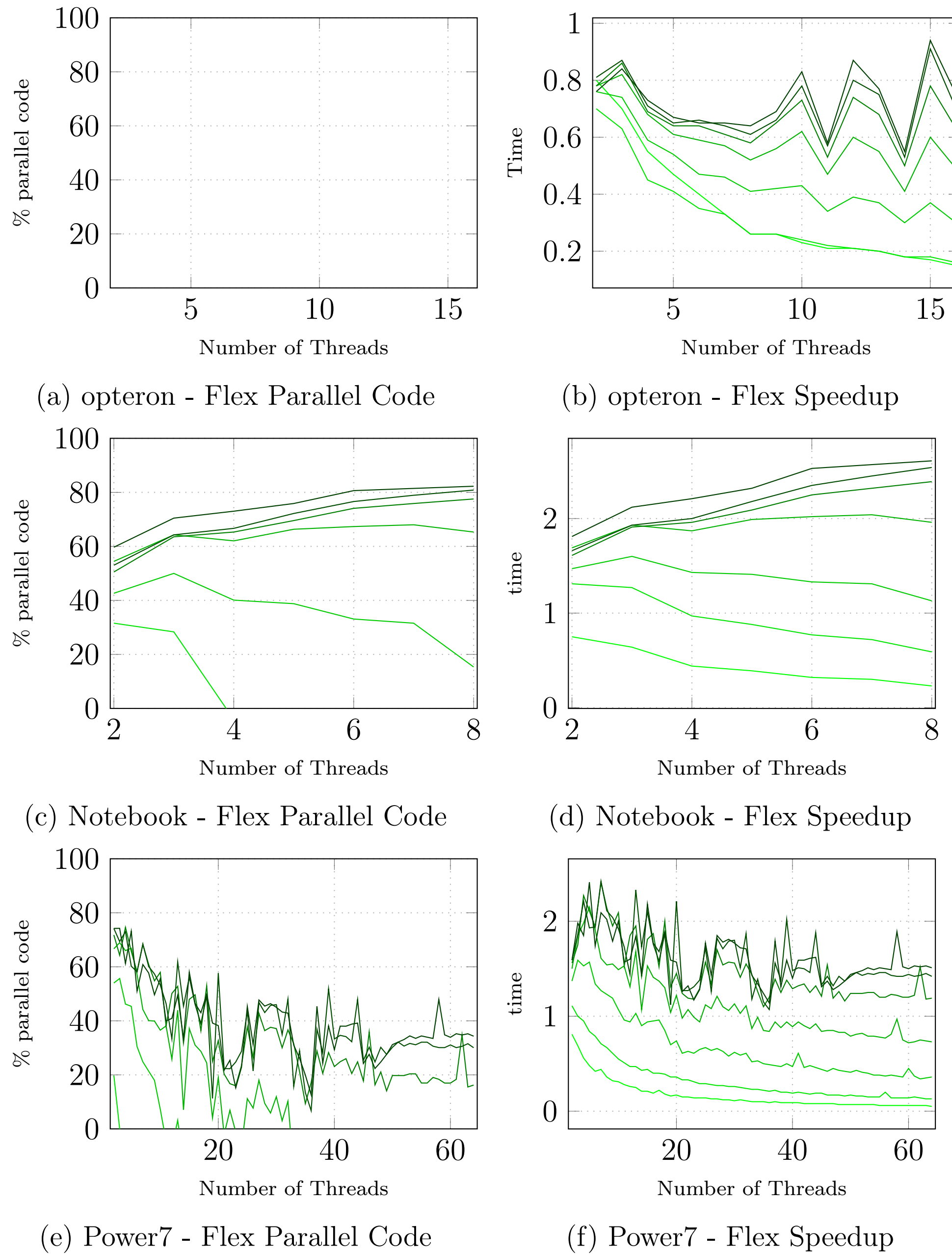
by Flex. We introduce the Power7 machine as a third comparison. As said before, Flex do not improve as expected the lexing phase, exhibiting strange fluctuating and surely not scalable behaviour. This reflects on the two indexes that we use to analyze the performance, the speedup respect to a single core execution and the amount of parallel code, as show in Figure 4.1

Parallel Code Percentage Let's start with the analysis of the percentage of parallel code in the lexing. We use the Karp-Flatt metric [30] as PAPAGENO does in the relative paper to keep the same performance evaluation indexes in order to have more significant results. This metric is a counterpart to Amdahl's law that states the maximum achievable speedup, while this metric extracts the exactly amount of parallel code executed given an amount p of processors and s as the achieved speedup.

As it can be seen in Figure 4.1a, 4.1c and 4.1e, the results are different as the machines shows different behaviour with respect to the same code portion. We decide to show the percentage of parallel code between 0 and 100 since there are no other possible results achievable, unless the relative speedup is less than 1. During the run though, we verify that the speedup of opteron-server grows as the file dimension does, but not over the threshold. As a consequence, all data computed for the parallel execution are below the 0% and do not appear in the graph. This does not happen for the notebook and the power7 cases (Figure 4.1d and Figure 4.1f) as their corresponding parallel code graphs have values.

The reason of this difference is that inside the code of the Flex scanners there are a lot of `strncpy` functions, used to save the results inside the respective arrays. This because the variable containing the current scanned variable is always rewritten at each scan and thus it cannot be referenced by different variables since the memory is always the same. Moreover, the instruction set used by opteron-server is the SSE2, while the notebook and power7 use the SSE3 set, that improves the operations like `strncpy`.

Thread migration It is a common pattern that the use of multiple cores for small files, together with the possible thread migration from one CPU to the other, negatively affect the performances. More in detail, a thread migration from one CPU to another implies a significant drop in the effectiveness of the caches, as the computation is moved to a processor where the working-set is not pre-heated in cache. By contrast, a higher load on all the available CPUs will prevent the scheduler from moving the tasks in an attempt to equalize the load. This issue is partially solved by pinning the threads to a specific CPU through processor affinity settings.



(a) opteron - Flex Parallel Code

(b) opteron - Flex Speedup

(c) Notebook - Flex Parallel Code

(d) Notebook - Flex Speedup

(e) Power7 - Flex Parallel Code

(f) Power7 - Flex Speedup

Figure 4.1: Speedup and Parallel Code Portion using Flex as parallel lexer on three different machines.

Conclusions The discussed problems leads to a degradation of the performance for all the platforms: opteron-server is not able to pass the 1× speedup, basically worsen the sequential one core run, while the other

two machines reaches the 2× speedup, but only considering the biggest files. This results leads us to think that a custom lexer could be relieved from the overhead that Flex has, since the grammar is really small and we can take full control of the multithreading environment. Also, by rewriting the lexer we get rid of the `strncpy` functions, since all the information are now pointed by an array without the need of moving them in different memory areas during the lexing process.

Parallel Execution With Custom Lexer Using a custom lexer improves a lot the performance of the lexing phase by at least 1.5 order of magnitude with respect to Flex; the latter can be run in a mode that supports multithreading, but even after accurate scans of the related assembly code, nothing shows up that could suggest a fix inside the user written code but instead we believe that Flex do not correctly separate the memory space between threads, thus resulting into a general slowdown of the performance, unacceptable since even if, as already said, the time spent on lexing of operator precedence grammar is comparable to the parsing, an 1.5 order of magnitude is a too heavy slowdown with respect to normal expectations.

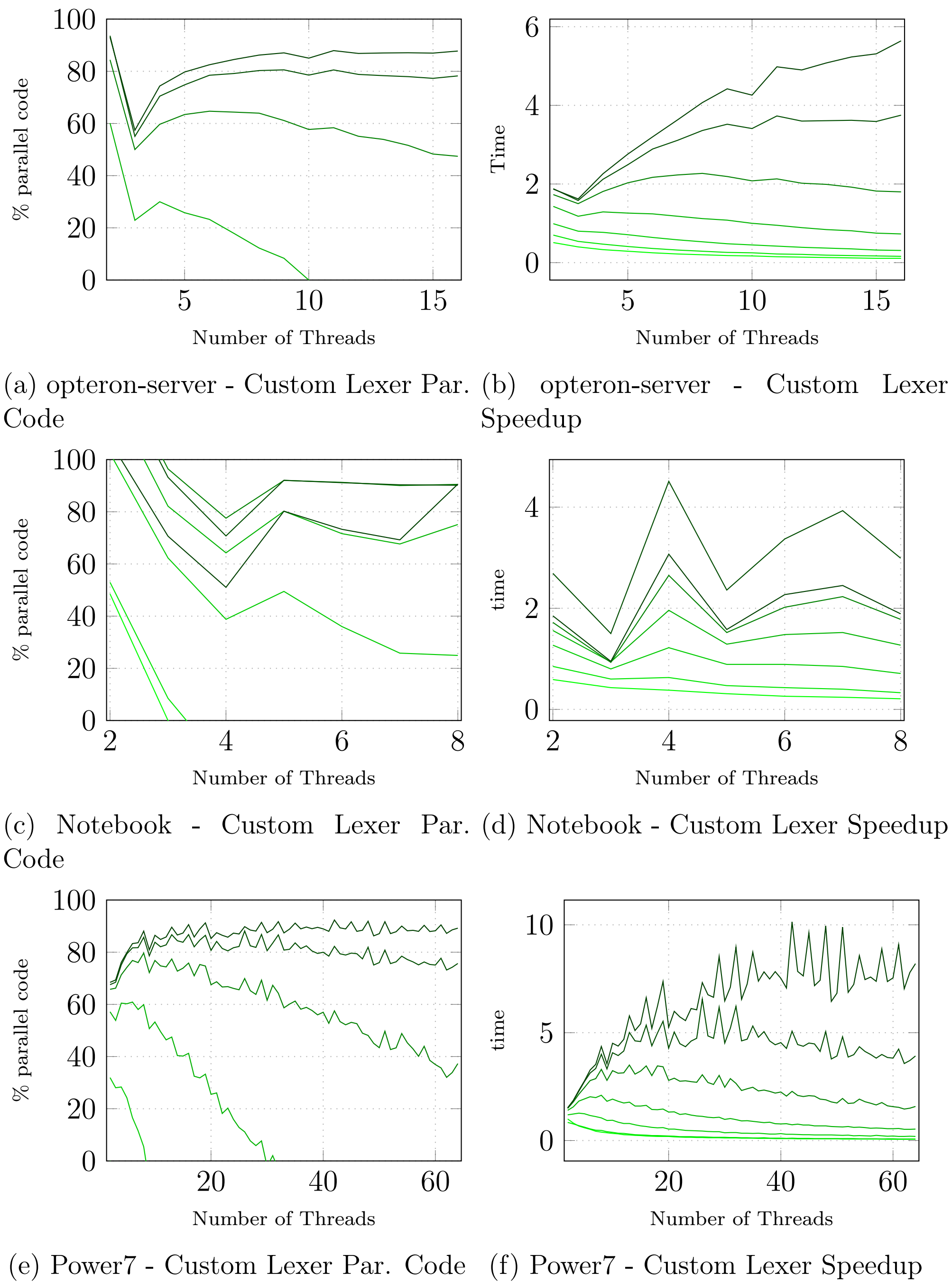
Figure 4.2 show the parallel code and speedup results using the custom lexer explained in Chapter 3. Looking at Figure 4.2a and 4.2b, we can see that opteron-server has improved with the removal of the *strncpy* and the Flex overhead, being close to 90% of parallel code and a speedup of 6× for the biggest files; same behaviour for the smaller files, that suffers from the communications between the cores more than gaining from their parallel work. The throughput has grown bigger, going from 60 MB/s to 850 MB/s with the 1 GB file, an increase of 14×; the related elapsed time for a complete lex run through the same file reveals that even the single core run with the custom lexer outperforms the full core one executed with the Flex generated scanner by 2.5×, with a maximum of 13× when fully operational.

The notebook gets an improvement too regarding both speedup and parallel code portion: the first always stand over 2×, reaching 4.5× with 4 cores, even if there is problem due to some sharing variables that slows down the process, even if it doesn't prevent an global improvement of the statistic. The parallel code portion is stable around 90%, following the opteron-server values, confirming the improvement that a custom lexer can provide to this kind of applications. As like as the opteron-server, the throughput of the application reaches a 1100MB/s with a 1GB input file, more than 18× better than the Flex parallel lexer, also because the architecture employed is newer with respect to the opteron-server.

Regarding power7, Figure 4.2f shows a 10× improvement with a correlated stable 90% of parallel code in Figure 4.2e. This results shows the

exploitation of the many cores of the machine and confirm once again that a custom lexer for a specific application is better than the general purpose available in literature. It is worth noticing that with this configuration the lexing time is always less than its correspondent parsing time and as the input files grows bigger, the parallel portion of the code remains stable as the number of cores employed grow. The throughput shows impressive enhancements, going from 150 MB/s to 1500 MB/s (a 10× improvement), just as it has the elapsed time, going from an average of 10 seconds against the new 1.5 seconds (a 6.6× improvement).

It has been shown with all the test that a custom lexer clearly enhances the overall performance of the lexing phase, mainly due to an easier grammar and syntax with respect to other language (e.g. JSON and Lua, used as examples on the PAPAGENO paper) that do not require special positioning or have different configurations depending on the cut points chosen at the beginning of the scan phase. Moreover it is clear that Flex has some problems handling multithreaded applications that shares different data between them even if the data structure are built in order to avoid any kind of reference between threads.



(a) opteron-server - Custom Lexer Par. Code

(b) opteron-server - Custom Lexer Speedup

(c) Notebook - Custom Lexer Par. Code

(d) Notebook - Custom Lexer Speedup

(e) Power7 - Custom Lexer Par. Code

(f) Power7 - Custom Lexer Speedup

Figure 4.2: Speedup and Parallel Code Portion using a Custom Lexer as parallel lexer on three different machines.

4.2 Parsing

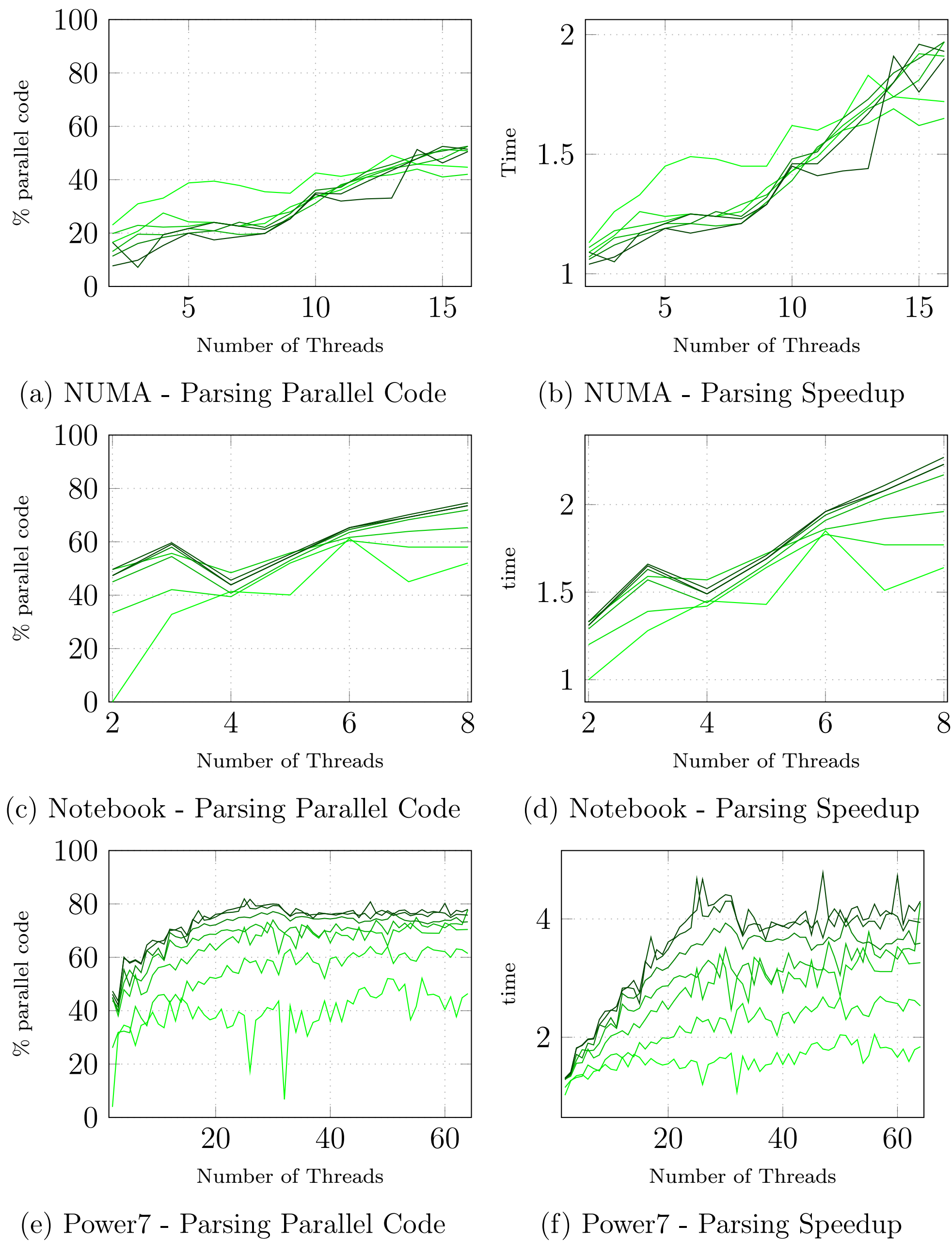
Parallel Parsing without Optimization Regarding the parsing phase, Figure 4.3 depicts the results obtained before the global variable optimization as a comparison to show the concrete improvements that such a implementation leads to. Graphs 4.3a and 4.3b show only a 2× speedup at most and less than 60% of parallel code, while things get better with the notebook tests that manage to extend the percentage by 20. This leads us to think that something other than architecture specific problems were involved, since we expected at least an 80% of parallel parsing code . Power7-server instead shows good results even before the optimization, and also shows different values depending on the file dimensions as the number of cores employed grow, that is not true for the first two examples, that exhibits common patterns for all the input files.

Parallel Parsing after Optimization The optimization, as explained in Chapter 4, aim to remove any kind of variable relationship between threads, in order to let each one of them handle its own fixed memory space and exploits as much as possible the L1 and L2 caches. The results of this can be seen in Figure 4.4. Each one of the three runs shows improved statistics: opteron-server gets 20% more parallel code and a 50% more speedup, going from 2 to 3. The notebook has more solid values among all the files and a slightly improvement in both parallel code and speedup. Power7-server instead shows the most relevant improvement regarding the speedup phase, going from 4× to 6×, while the speedup is stable over 80%, just a little bit better than without the optimization.

As a consequence of this little improvements, the overall parsing throughput is slightly better than before (power7-server has the most relevant one, with a gain of 200MB/s, going from 600 to 800 on average) and the elapsed time are also improved, but not as much as the lexer one does (0.5 seconds on average on all the examples).

4.3 Comparison With Other Solutions

We decided to compare the proposed solution with other examples in literature, in order to give to our results a broader value. We take the already cited solution, the PP Transducer [19], and run the same tests that we run on our own, taken from the Treebank dataset [28]. The output of the PP Transducer consists only in one value of time, so we deduce that it represents the sum of the lexing and the parsing phase; in Figure 4.5 appears



(a) NUMA - Parsing Parallel Code

(b) NUMA - Parsing Speedup

(c) Notebook - Parsing Parallel Code

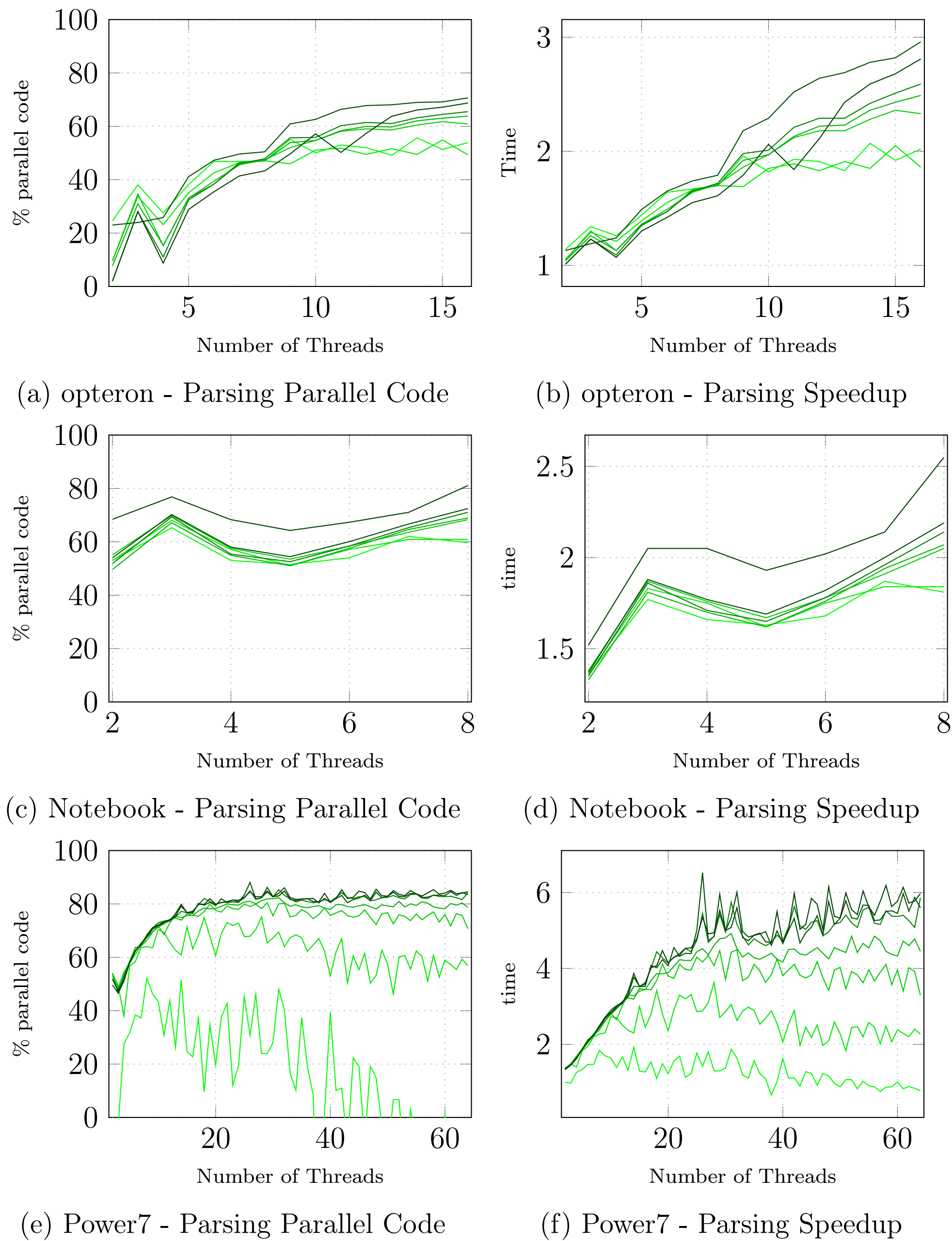
(d) Notebook - Parsing Speedup

(e) Power7 - Parsing Parallel Code

(f) Power7 - Parsing Speedup

Figure 4.3: Speedup and Parallel Code Portion regarding the parsing phase on three different machines, missing the global variable and function arguments optimizations

the only two statistic that we've been able to compare, the total amount of lexing and parsing time employed by their solution and the Parallel Parsing



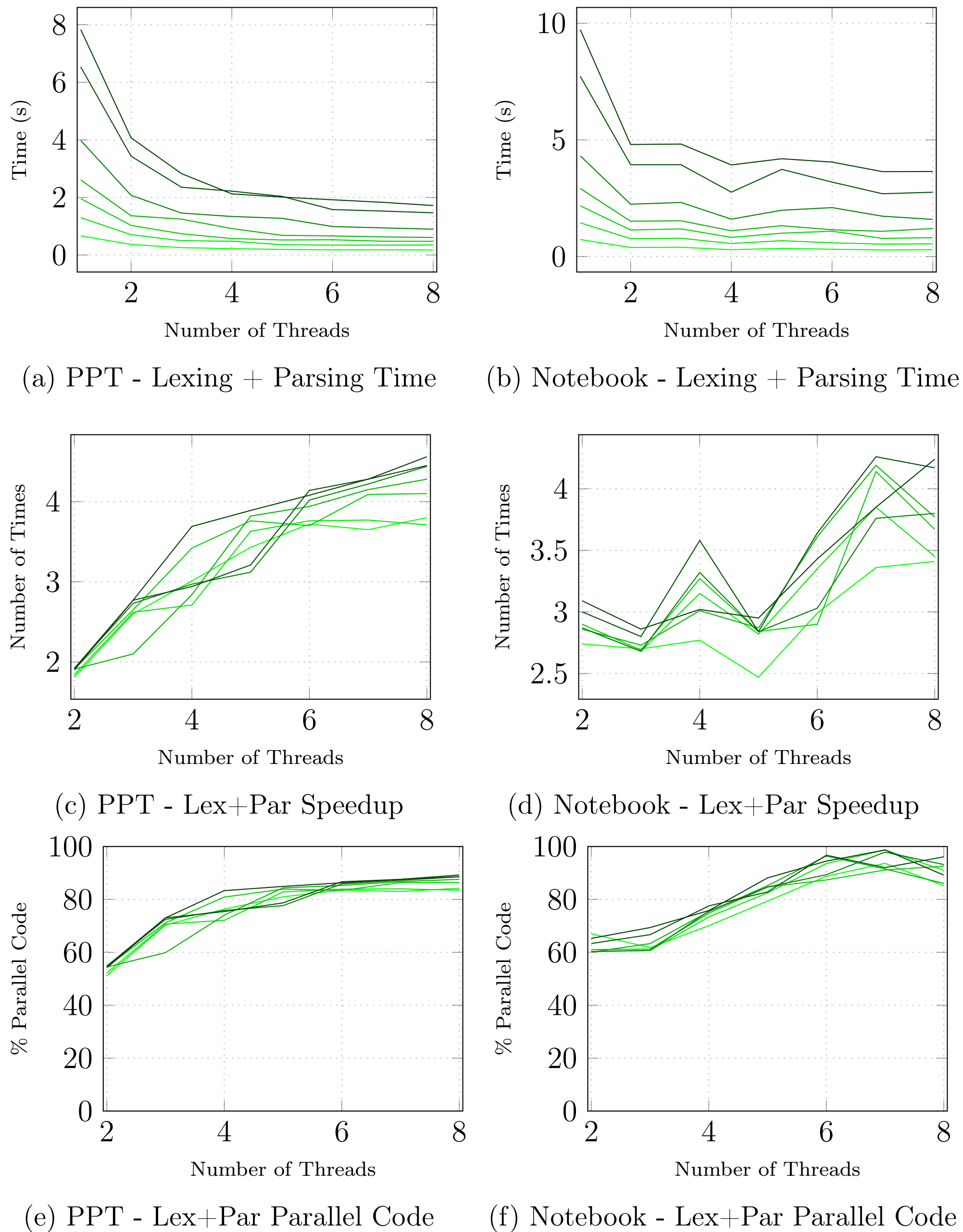
(a) opteron - Parsing Parallel Code (b) opteron - Parsing Speedup

(c) Notebook - Parsing Parallel Code (d) Notebook - Parsing Speedup

(e) Power7 - Parsing Parallel Code (f) Power7 - Parsing Speedup

Figure 4.4: Speedup and Parallel Code Portion regarding the parsing phase on three different machines, optimized with respect to global variable references and function arguments

Algorithm on the notebook, together with their parallel code portion and speedup.



(a) PPT - Lexing + Parsing Time
(b) Notebook - Lexing + Parsing Time
(c) PPT - Lex+Par Speedup
(d) Notebook - Lex+Par Speedup
(e) PPT - Lex+Par Parallel Code
(f) Notebook - Lex+Par Parallel Code

Figure 4.5: Total Lexing and Parsing elapsed time employed by the Notebook and the PP Transducer running the TreeBank files. Dark lines show bigger files, respectively 85 MB, 172 MB, 258 MB, 344 MB, 516 MB, 860 MB and 1 GB

The were some difficulty in running the examples on the other two platform, the opteron-server and power7-server, since the PPT has a lot and sparse dependencies that requires a non-trivial effort in setting up all the running requirements, that we spent in setting up the notebook. At the end, as shown by the final results, the PP Transducer has slightly better timings, but there are some differences in the way it computes the final results to consider. First of all, the PP Transducer needs the help of a DFA (Deterministic Finite Automaton) in order to solve the query, while our solution builds it up while scanning the tree, deriving the solution during while proceeding in the algorithm steps; secondly, the transducer has prior knowledge of the tag inside the input file, since the DFA needs to be built before the parsing phase, using this pre-known tags. This is not true in our case, since everything is discovered as the tree is scanned, leaving the parser unable to predict any kind of information regarding the tag string value. Thus, evidently, gives the PP Transducer an advantage, at the cost of prior knowledge, a solution that we want to avoid to maintain generalization and to avoid future users to provide additional data before the parsing.

Although all of this, the results are really close: in Figure 4.5d the notebook has some problems handling the data from Treebank passing from 4 to 5 cores, but the final results with 8 cores is identical to the one in Figure 4.5c. Both PPT and our parser shows great parallel coding portion, with the latter closer to 95% after 6 core and stable in that direction. The throughput of the notebook is always over 500MB/s, with a spike of 1000MB/s with 4 cores and over 800MB/s with 8 cores, while the PPT is stable around 500-600MB/s.

5 Conclusions and Future Work

We described a syntax directed approach to exploit parallelism in XML-query languages, employing and extending PAPAGENO. We started by listing the current state of the art of parallel parsing and lexing, recapping the main structure of the OPG grammar and how to use it to exploit parallelism. Then we elaborate a grammar describing a meaningful subset of XML together with the query language XPath: the latter has been described in details with a data model and the lexical and syntactical structure that has been exploited to create an efficient algorithm. Then, the lexing and parsing phase of the PAPAGENO algorithm has been analyzed with the creation of a custom lexer and some memory optimization techniques that leads to a relevant parsing time improvement. The XPath Query Algorithm with its semantic actions allow us to perform different kind of queries during the parsing process, full exploiting the available cores and hardware to full power, with a 90% of parallel code on the most performing one. The data structures play a fundamental part inside the algorithm, since their smart and efficient design allows a lightweight management of the resources, enabling even low powered architecture to perform stressful tasks with the minimum amount of effort. The final tests compare our results with other appearing in the state of the art, showing that it is possible to achieve significant results, sometimes better, even with a general purpose solution. Possible future works on this project may aim to extend the XML and XPath grammar, to cover all the possible queries and XML document. Another possible improvement regards how the threads handle the structure containing the variables concerning their state, trying to avoid sharing data between thread as much as possible in order to exploit the local caches of each processor.

C code Data Structure Definition

In this appendix can be found all the definition of the data structure used inside the algorithm described in this thesis. All the references are related to the images that depicts the corresponding data structure definition.

Lex Token 3.2

```
typedef struct lex_token {
  gr_token token;
  char *semantic_value;
} lex_token;
```

Node Info 3.3

```
typedef struct node_info {
  char path[PATH_ALLOCATION_SIZE];
  char *info;
  char *query_end;
  flags_t flags;

  int path_length;
  int query_end_length;

  struct node_info *next;
} node_info;
```

Leaf List 3.4

```
typedef struct leaf_list {
  node_info *head;
  node_info *tail;
} leaf_list;
```

Preallocation memory pools 3.5

```
typedef struct preallocation_memory_pool {
  leaf_list **pool;
  int ceil;
  int index;

  char* next;
} preallocation_memory_pool;


typedef struct preallocation_node_info_pool{
  char* slab;
  char* slab_path;
  char* slab_qe;
  node_info **pool;
  int ceil;
  int index;

  char* next;
} preallocation_node_info_pool;
```

Memory Leaf Pool 3.6.2

```
typedef struct memory_leaf_pool {
  leaf_list **pool;
  int index;
} memory_leaf_pool;
```

Argument Pointer Structure 3.6

```
typedef struct args_pointer_struct{
  token_node_stack **lexing_stack_to_free;
  token_node_stack **parsing_stack_to_free;
  memory_leaf_pool *leaf_pool;
  node_info *node_pool;
  preallocation_memory_pool *prealloc_mem_pool;
  preallocation_node_info_pool *prealloc_node_pool;
  void** read_token;
  void** slab_pool_pointer;
} args_pointer_struct;
```

Bibliography

[1] C.J.H. Grune D., Jacobs. *Parsing techniques a practical guide.* Ellis Horwood Limited, Chichester, England, 1990.

[2] Stefano Crespi-Reghizzi and Dino Mandrioli. Operator precedence and the visibly pushdown property. *J. Comput. Syst. Sci.*, 78(6):1837–1867, 2012.

[3] Wei Lu, Kenneth Chiu, and Yinfei Pan. A parallel approach to XML parsing. In *7th IEEE/ACM International Conference on Grid Computing (GRID 2006), September 28-29, 2006, Barcelona, Spain, Proceedings*, pages 223–230, 2006.

[4] Yinfei Pan, Ying Zhang, and Kenneth Chiu. Hybrid parallelism for XML SAX parsing. In *2008 IEEE International Conference on Web Services (ICWS 2008), September 23-26, 2008, Beijing, China*, pages 505–512, 2008.

[5] Alessandro Barenghi, Stefano Crespi-Reghizzi, Dino Mandrioli, Federica Panella, and Matteo Pradella. The PAPAGENO parallel-parser generator. In *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pages 192–196, 2014.

[6] Henk Alblas, Rieks op den Akker, Paul Oude Luttighuis, and Klaas Sikkel. A bibliography on parallel parsing. *SIGPLAN Notices*, 29(1):54–65, 1994.

[7] Wojciech Rytter. On the complexity of parallel parsing of general context-free languages. *Theor. Comput. Sci.*, 47(3):315–321, 1986.

[8] C. N. Fischer. *On parsing context free languages in parallel environments*. Cornell University, 1975.

[9] Jacques Cohen, Timothy J. Hickey, and Joel Katcoff. Upper bounds for speedup in parallel parsing. *J. ACM*, 29(2):408–428, 1982.

[10] M. Dennis Mickunas and Richard M. Schell. Parallel compilation in A multiprocessor environment (extended abstract). In *Proceedings 1978 ACM Annual Conference, Washington, DC, USA, December 4-6, 1978, Volume I*, pages 241–246, 1978.

[11] Dilip Sarkar and Narsingh Deo. Estimating the speedup in parallel parsing. *IEEE Trans. Software Eng.*, 16(7):677–683, 1990.

[12] Zhijia Zhao, Michael Bebenita, Dave Herman, Jianhua Sun, and Xipeng Shen. Hpar: A practical parallel parser for html-taming HTML complexities for parallel parsing. *TACO*, 10(4):44, 2013.

[13] Ladislav Vagner and Borivoj Melichar. Parallel LL parsing. *Acta Inf.*, 44(1):1–21, 2007.

[14] W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[15] Ryoma Sin'ya, Kiminori Matsuzaki, and Masataka Sassa. Simultaneous finite automata: An efficient data-parallel model for regular expression matching. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*, pages 220–229, 2013.

[16] G. Umarani Srikanth. Parallel lexical analyzer on the cell processor. In *Fourth International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2010, Singapore, June 9-11, 2010 - Companion Volume*, pages 28–29, 2010.

[17] Alessandro Barenghi, Stefano Crespi Reghizzi, Dino Mandrioli, Federica Panella, and Matteo Pradella. Parallel parsing made practical. *Science of Computer Programming*, 112, Part 3:195 – 226, 2015.

[18] PAPAGENO: the parallel parser generator for operator precedence grammars, `https://github.com/PAPAGENO-devels/papageno`.

[19] Peter Ogden, David B. Thomas, and Peter Pietzuch. Scalable XML query processing using parallel pushdown transducers. *PVLDB*, 6(14):1738–1749, 2013.

[20] Ying Zhang, Yinfei Pan, and Kenneth Chiu. Speculative p-dfas for parallel XML parsing. In *16th International Conference on High Performance Computing, HiPC 2009, December 16-19, 2009, Kochi, India, Proceedings*, pages 388–397, 2009.

[21] PugiXML, `http://pugixml.org/benchmark/`.

[22] Expat Parser, `http://expat.sourceforge.net`.

[23] Violetta Lonati, Dino Mandrioli, Federica Panella, and Matteo Pradella. Operator precedence languages: Their automata-theoretic and logic characterization. *SIAM J. Comput.*, 44(4):1026–1088, 2015.

[24] U. Germann, E. Joanis, and S. Larkin. Tightly packed tries: How to fit large models into memory and make them load fast too. In *In Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing*, pages 31–39, 2009.

[25] W3C XML grammar description, `http://cs.lmu.edu/~ray/notes/xmlgrammar/`.

[26] W3C XPath, `http://www.w3.org/TR/xpath/`.

[27] Flex, the Fast Lexical Analyzer, `http://flex.sourceforge.net/`.

[28] Treebank dataset, 2013, `http://goo.gl/c6O3X`.

[29] Power7 detailed Description, `http://www-01.ibm.com/common/ssi/cgi-bin/ssialias?infotype=an&subtype=ca&appname=gpateam&supplier=897&letternum=ENUS113-037`.

[30] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Commun. ACM*, 33(5):539–543, 1990.

Ringraziamenti

Desidero ringraziare prima di tutto il professor Matteo Pradella per avermi dato la possibilità di poter svolgere questo lavoro di tesi e anche per ogni suo corso che ho seguito, stimolando in me il desiderio di approfondire tematiche differenti e interessanti. Ringrazio anche il mio correlatore Alessandro Barenghi per avermi supportato nelle varie scelte da intraprendere durante la stesura del documento e della sua realizzazione, oltre che avermi accompagnato durante tutto il mio lavoro. Ho imparanto tante cose nuove e sempre con entusiasmo. Non avrei potuto desiderare due guide migliori.

Un ringraziamento particolare va ovviamente alla mia famiglia: mia madre, con cui ho condiviso passo passo tutta la mia carriera scolastica e da cui ho tratto ispirazione nel prendere le scelte che mi hanno portato qui oggi. Mio padre, il cui supporto non è mai mancato in tutta la mia vita e a cui devo la mia possibilità di festeggiare questo grande traguardo. Mio fratello, a cui, inutile negarlo, ho seguito come una guida e al quale mi ispiro ancora oggi in alcune delle mie scelte.

Ultimi ma non per questo tali, tutti gli amici che mi hanno accompagnato in questi anni durante il mio percorso. Da quando ho iniziato, sono certo di essere cresciuto e di essere una persona diversa (spero migliore) e sono assolutamente sicuro che ognuno vi abbia preso parte, e si meritano tutti i miei ringraziamenti. In particolare Alessandra e Isaia, grazie per avermi tenuto compagnia durante questo ultimo periodo, ne avevo veramente bisogno. Un ringraziamento particolare va anche a Stefano, l'unico amico che mi abbia accompagnato in tutto e per tutto, so che ci sarai sempre quando ne avrò bisogno.

Non preoccupatevi se non siete stati citati, le persone che veramente contano nella mia vita avranno sempre la mia gratitudine.