Parsing gigabytes of JSON per second with parallel bit streams

by

Luiz Fernando Peres de Oliveira

B.Sc., Universidade Anhanguera, 2012

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Spring 2023

Declaration of Committee

Name:	**Luiz Fernando Peres de Oliveira**
Degree:	**Master of Science**
Thesis title:	**Parsing gigabytes of JSON per second with parallel bit streams**
Committee:	**Chair:** Arrvindh Shiraraman Associate Professor, Computing Science

Robert Cameron
Supervisor
Professor, Computing Science

Yuepeng Wang
Committee Member
Assistant Professor, Computing Science

Keval Vora
Committee Member
Assistant Professor, Computing Science

Tianzheng Wang
Committee Member
Assistant Professor, Computing Science

Anders Miltner
Examiner
Assistant Professor, Computing Science

Abstract

Studies have shown that it is possible to boost the efficiency of text processing by carefully eliminating branches as well as reducing branch mispredictions and cache misses, which can be achieved with a few techniques, such as the use of Boolean algebra to reduce pointer-chasing in data structures and to abstract branching. With current advances in technology, vector extensions (SIMD) have been added to commodity processors and have allowed the creation of new algorithms that are able to accomplish the non-trivial task of parallelly processing streams in Gigabytes per second. The Parabix framework exploits the concept of parallel bit streams to take even more advantage of SIMD instructions by transposing and processing streams in batches. This study focuses on using Parabix to boost the efficiency of JSON parsing for Big Data.

Keywords: JSON parsing; parallel parsing; Parabix; stream processing; high-performance text processing; SIMD

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the advances in technology and Big Data, most out-of-factory commodity processors (Intel, AMD, ARM, POWER) include vector extensions that can speed up data processing. Many popular data processing algorithms have become obsolete because they do not scale well as data grows bigger and, because of that, leveraging vector instructions is becoming a more active trend [11, 16, 23, 30].

Considering stream processing tasks such as parsing, querying, and data analysis for Big Data, it is essential that algorithms keep up with the input size as it increases, however many out-of-the-box algorithms that are in charge of said tasks often fail, for example, built-in parsers for streams in frameworks, system libraries [6] and data analysis tools for relational and non-relational DBMSs [13]. This happens because most of these algorithms are of sequential nature [28, 3, 22] and need to sacrifice efficiency by either (i) processing one item at a time in a sequential manner or (ii) processing multiple items at a time and getting penalized with synchronization in a multithreaded manner, for example, querying for multiple names in a database can easily be done in parallel, however, parsing a JSON document can be extremely difficult because of its sequential nature.

JSON (JavaScript Object Notation) is a lightweight text format that is able to represent data that can be interchanged in the Web with a language-independent syntax [7]. It was first created to represent objects in JavaScript (ECMAScript programming language) but it has largely been used in the tech industry because of its simplicity. JSON has a very straightforward grammar and can easily represent structured data as it only has 4 terminal types $\{string, number, boolean, null\}$ and 2 composed types $\{array, object\}$, where an array starts with a '[', ends with a ']' and can have any number of both terminal and composed symbols separated by ',', and an object starts with '{', ends with a '}', but differs from array because they are key-value properties with the key always being a symbol of type *string* and the value being any of the other valid symbols (terminal or composed types) separated by ',' and having its keys separated from their respective values with a ' : '. That is to say, JSON's simplicity makes structured data very easy to be abstracted, however, because its grammar is context-free §2.2, parsing documents written in this format has sequential

nature and can make the task of parallelizing instructions very difficult, for example finding an object depth. Despite that, we show in later chapters that we can still parse most of it in parallel by finding and exploiting regular sub-grammars inside its grammar.

This is where SIMD comes along. SIMD stands for Single Instruction Multiple Data and, as its name suggests, a SIMD instruction is capable of processing multiple items of data with a single instruction rather than one individual item per instruction. In this thesis, whenever not specified otherwise, item means the smallest unit that can be computed by a processor (a.k.a byte or $i8$). What this means is that on a 64-bit architecture that includes SSE2 (128-bit vector extension), we can process 16 items with a single instruction ($16 \times i8$). On the same machine, we can only process one item at a time using traditional instructions ($1 \times i8$, $1 \times i32$, etc). If we were parsing a text byte-by-byte, this is sixteen times more data processing with a single instruction and, if we were talking about AVX512 (512-bit vector extension), that would mean sixty-four times ($64 \times i8$) more data processing with a single instruction. The current trend in hardware is to increase the SIMD vector width, which can provide immediate benefits to vectorized programs that traditional byte-at-a-time programs cannot leverage. Needless to say, there seems to have a tendency where the best text processing programs [16, 30, 21, 9, 23] are the ones that are able to leverage both vector extension instructions and multithreading without adding too much complexity and synchronization time, and although, solving this efficiency-and-size problem such that algorithms are scalable can be fairly straightforward to be implemented when these algorithms are of parallel nature, they can be extremely difficult to be implemented when these algorithms are of sequential nature. In consequence of that, research has been done around frameworks, such as Parabix [25], that can be used as transparent as possible by programmers without jeopardizing the program's maintenance and parallelism.

Parabix (Parallel Bit Streams) is a framework and toolkit that allows applications to benefit from modern SIMD instructions. Previous works [1, 3, 4, 6] show that Parabix is scalable and works well for large-sized data. Its key is based on the transposition of byte-oriented data into parallel bit streams. This way, in a 64-bit machine, Parabix is able to process 64 bytes at a time rather than 64 bits (per processor), and as the architecture block width increases, so does the number of processed bytes at a time, in a linear fashion. Most existing applications of Parabix are related to text processing, such as XML and CSV parsing, data compressing and hashing at rates greater than 1 Gigabyte per second. Outside Parabix, Lemire and Sautot (2018) show that it is also possible to SIMDize operations in algorithms for database systems and applications for Business Intelligence and Big Data [8], proposing a new design of an algorithm that maximizes compressed indexes using SIMD.

This research involves the identification of regular sub-grammars within the JSON grammar (§2.1) that can leverage Parabix's multi-core scaling and full-SIMD-vectorization support for bitwise data parallelism to improve JSON overall parsing performance. At the end, we compare and contrast the performance of the solution here proposed against algorithms

for parsing, such as simdjson, yyjson and RapidJSON, which are, to our knowledge, the top-3 best parsers that currently exist. Chapter 2 explains the background of this research and related work, as well as details how we identify regular languages within the JSON grammar. Chapter 3 goes in depth into the design and implementation of the work here presented. Chapter 4 evaluates the experiments that were performed and compares their results. Chapter 5 presents applications that could benefit from this work, and finally, concludes and discusses future works.

Chapter 2

Background

2.1 Grammars

In language theory [4], a grammar G is defined to be a set of production rules that describe how to form syntactically valid strings in G with a given alphabet A. To illustrate that, let grammar G be defined by the set of production rules: $G \rightarrow cG'$ and $G \rightarrow b$ or simply $G \rightarrow cG' \mid b$, where G' is a recursion of G. The alphabet A in this example consists of only b and c, such that valid strings in G are $cccb$, cb, b, ..., $ccccccb$, etc. Figure 2.1 shows what a parse tree representation of G looks like for the string $cccb$. In G, b and c are said to be terminal symbols, G' is said to be a non-terminal symbol, since it recurses and, all strings that are formed with G are valid sentences that define a language L.

According to Chomsky (1959), a grammar can be compared to a device that is able to enumerate sentences of a language L, where L is a collection of sentences of finite length constructed from a finite alphabet A of symbols. Chomsky mentions [3, 4] that the first step in the linguistic analysis of any language L is to define a finite system that gives the representation for its sentences and divides grammars in 4 types: $type\ 0 \supseteq type\ 1 \supseteq type\ 2 \supseteq type\ 3$ — figure 2.2, where *type* 0 are essentially Turing machines, both *type* 1 and *type* 2 are systems that can describe phrase structure and *type* 3 can describe finite automata. In his famous Theorem 2 (1959), he defines every recursively enumerable set of strings to be a *type* 0 language.

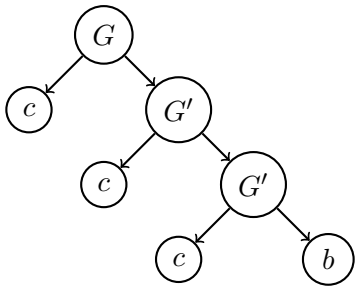

Figure 2.1: Parse tree representation of string *cccb* for grammar G

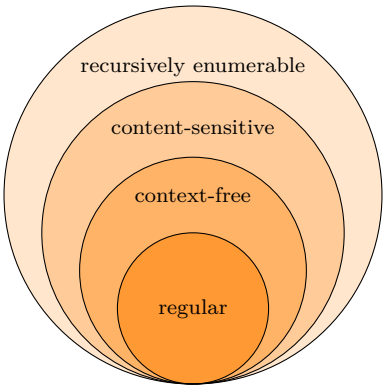

Figure 2.2: Chomsky hierarchy

In other words, *type* 0 is a superset that includes all recursively enumerable grammars $G \in type\ 0$ that can use a Turing machine [27] to describe and enumerate sentences of any language L_{type0}, *type* 1 represent the set of all context-sensitive grammars $G \in type\ 1$ that can use a linear bounded automaton [19, 14] to describe and enumerate sentences of any language L_{type1}, *type* 2 is set of all context-free grammars $G \in type\ 2$, such that a non-deterministic pushdown automaton [10] can be used to describe and enumerate sentences of any language L_{type2}, and finally *type* 3, which is the set of all regular grammars $G \in type\ 3$ that a finite state automaton [24] can describe and enumerate sentences of any language L_{type3}.

That is to say, this research benefits from the fact that every context-free grammar is a superset of a regular grammar, since $type\ 2 \supseteq type\ 3$, and further sections explain why this is relevant to the work here described.

2.1.1 Regular Grammars

Kleene's Theorem [12] defines regular grammars to be grammars that describe languages that can be decided by finite state automata and that can be expressed by regular expressions, which is an alternative way to express such regular grammars.

Regular grammars are either right-regular or left-regular. While right-regular grammars have production rules in which all non-terminal symbols are on the right-hand side of each of its productions, left-regular grammars have production rules in which all non-terminal symbols are on the left-hand side of each of its productions, and in case a grammar G has, at the same time, right-regular and left-regular rules, then this grammar is no longer regular [4]. For this reason, regular grammar restriction rules require that (1) all production rules have at most one non-terminal symbol and (2) that this non-terminal symbol is always the leftmost or the rightmost symbol. $G \rightarrow aB \mid b$ and $F \rightarrow Ba \mid b$ are examples of right-regular and left-regular grammars, respectively.

Regular expressions

Regular expressions are a sequence of characters that are able to represent regular languages. They were first described by Kleene (1951) when he created a mathematical notation called regular events that was used to express regular languages using only mathematical symbols. Nowadays, most string-searching algorithms use regular expressions as a mean to find patterns in text streams with the POSIX notation, where:

- Individual characters represent themselves unless they are one of the special characters `*+-?[]{}\()|^$.` which have to be escaped with a backslash symbol in order to be treated as literal characters. For example, the regular expression `\.` represents the literal character dot whereas the regular expression `.` represents the wildcard pattern which matches any single character.
- Character class operators `[]` can define matching pattern classes that allow any character of a given class to be matched, for example, class `[axZ]` matches strings `a`, `x` and `Z`. Consecutive ranges can use the character hyphen as a shorthand for a class range, for example `[abcdefg]` can be written as `[a-g]`.
- Multiple subpatterns concatenate, for example `[ab][cd]e` matches strings `ace`, `bce`, `ade` and `bde`.
- Alternation operator `|` allows patterns to have alternative forms, for example `ab|cd` matches strings `ab` or `cd`.
- Repetitions operators `*+{}` can define the number of occurrences of the previous pattern, where the operator `*` represents zero-or-more occurrences, the operator `+` represents one-or-more occurrences and `{}` specify a variable number of occurrences, for example `a{3}` matches string `aaa`, `a{2,4}` matches strings `aa`, `aaa`, `aaaa` and `a{2,}` matches strings `aa`, `aaa`, `aaaa`, ...
- Optional operator `?` represents zero-or-one occurrence of the previous pattern, for example `a?b` matches strings `ab` and `b`.
- Parentheses `()` may be used to change the precedence of elements and to create subpatterns.
- Characters `^$` may be used to search for patterns in the beginning or end of a line, respectively, in line-based searches.

With Thompson's algorithm [26] it is possible to convert a regular expression of length m into a nondeterministic finite automaton (NFA) with $O(m)$ states making it possible to search a text of length n in $O(mn)$ time. Converting a nondeterministic finite automaton (NFA) to a deterministic finite automaton (DFA) can be more efficient for some cases, as

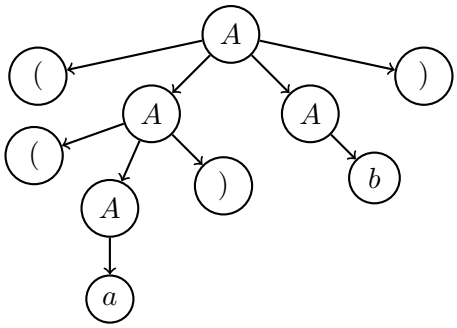

Figure 2.3: Parse tree representation of string $((a)b)$ for grammar A

DFAs have only one active state at any time and therefore searching a text of length n only takes $O(n)$ time, however, this conversion can result in state explosion as the DFA states may increase exponentially. Because of that, much work has been done around improving NFAs algorithms, such as Shift-Or algorithm [20] and its variations, which use the bitwise operations to simulate an NFA and are able to search a text of length n in $O(\frac{mn}{w})$ time, given a w-bit word. Another example is the algorithm described by Cameron et al. [11], which uses a data-parallel approach to simultaneously process data streams that are viewed as large integers with bitwise logic, stream shifting and addition, making it more efficient than the other approaches as they are performed byte-at-a-time.

2.1.2 Context-free grammars

Context-free grammars describe context-free languages. While every regular grammar is context-free, not every context-free grammar is regular. A simple way of defining context-free grammars is to think of the concept of regular grammars and add the fact that in context-free grammars, we can have non-terminal symbols anywhere in the production rules of a grammar G. A classic problem that is context-free but not regular is the bracket matching problem, for example $A \rightarrow (A) \mid AA \mid b \mid a$. See that the production rules (A) and AA are neither right-regular nor left-regular, but they are still valid CFG production rules. Figure 2.3 illustrates the tree parse representation of string $((a)b)$ that can be validated by grammar A.

The main difference between context-free grammars and regular grammars is that regular grammars only have access to its current state, and therefore, when it chooses a new state, all information about past states is lost, however, because CFGs can be recognized by pushdown automata, a stack may be used to manipulate and save information about previous states and decide which direction to take. That is to say, regular grammars have a more straightforward way of validating their production rules state-by-state in parallel, whereas CFGs are much harder to do the same as they usually compute non-terminal states depth-by-depth partially by keeping fragments of information in the stack until a terminal symbol is found and once it does, it recurses so that the partial computation is complete.

2.2 JSON grammar

JSON [7] stands for JavaScript Object Notation and although it was first intended to be used as a way to create and represent objects in JavaScript, its popularity has grown quite a lot as a text format to interchange data in the Web because of how easy it is to represent structured data with it. Nowadays, if two applications communicate with each other through the Web, chances are they are communicating through JSON.

JSON has a very simple context-free grammar and can easily represent structured data as it only has 4 terminal types $\{string, number, boolean, null\}$, and 2 composed types $\{array, object\}$.

Below is the context-free grammar for JSON:

Json → Value
Object → { } | { Members }
Members → Pair | Pair , Members
Pair → String : Value
Array → [ ] | [Elements]
Elements → Value | Value , Value
Value → Object | Array | `string` | `number` | `boolean` | `null`

JSON specification requires *string* values to be defaulted with UTF-8 format, *number* values to be any valid long integer or double precision, including scientific notation, *boolean* to be represented by keywords {**true**, **false**}. Special keyword **null** can be used to represent the non-existence of values.

2.2.1 Parallel approach for JSON tokenization

Because JSON's grammar is simple, its tokenization process can be totally done in parallel because only a few rules must be followed: (1) we must find *strings* before any other tokens as they can be escaped, (2) we must find valid *token breaks* such as terminal symbols and whitespace characters so we can find token boundaries, (3) we must find *keywords* and *numbers*, and finally, (4) we must *validate* if all tokens are valid tokens.

Strings

Strings start and end with double quotation marks (") and accept any valid printable and non-printable characters, however in some exceptions those characters must be escaped, for example, if your string content contains a symbol `"`, you must use a backslash character preceding it `\"`. Backslash characters can escape themselves with the use of `\\`. *Strings* are by far the most complex tokens in the JSON grammar because, as mentioned,

they can include any number of escaped double quotation marks as well as valid symbols `{ } [ ] : ,` numbers, whitespace and unicode characters.

Although not very intuitive at first, the presence (or absence) of backlash characters define string delimitations as they may escape double quotes. As well explained by Lemire (2019), only an odd sequence of backlash characters define an escaped quotation mark:

- `"seq 1 \\\" seq 2 \""` is a valid string because all backslash sequences are odd and therefore all `"` are escaped properly inside the double quotation delimiters.
- `"\\""` is not a valid string because the backslash sequence is an even number – please remember that backslash characters escape themselves – and therefore this example contains an extra `"`.

After we eliminate all escaped quotes, the remaining quotes are all delimiters and we can easily mark all string spans in parallel as shown in §3.3.2. Finally, we validate all strings to make sure they are valid UTF-8 strings and filter them out from the initial input so that we can find the remaining tokens.

Symbols

Tokenizing symbols is a trivial task as we only need to create one marker for every individual character `{ } [ ] : , -` that is not contained in a string span. The same happens for whitespace characters (space, tab and line break). These characters are used later in the process of tokenizing the remaining tokens.

Keywords

We only have three keywords in JSON: **null**, **true** and **false**. Because the length of these keywords does not vary, we can create individual markers for every single character k_i that is contained in them, that is, characters `truefalsn`. We realize that these keywords are also a concatenation of valid sub-regular expressions formed by some of characters $k_1, k_2, ..., k_n$. As described by Cameron et al. (2014), in this case we create a marker stream m_i, for each subexpression k_i, where $i \in 1...n$. Set bits in marker m_i represent matching positions for subexpression k_i in input stream s. Once every subexpression k_i is processed, we apply a concatenation step in marker streams from left to right (starting from m_1), where we shift stream m_i forward by one bit and perform a bitwise-and with stream m_{i+1}. This sub-result is then used to check marker m_{i+2} and so on until we check all markers up to m_n, thus, we repeat this process $n - 1$ times in order to create a final result stream with matching positions for keyword k. At every step, we also create an error stream $e_{m_i(m_{i+1})}$ to help us identify steps where something may go wrong by applying a bitwise-xor between advanced stream m_i and m_{i+1}. At the end of this step, we join (bitwise-or) all these error streams as a final error stream. In case of no errors, the current marker stream is shifted forward

to represent a valid expression, which is an end position of a keyword. Finally, as we know the length of each of our keywords $k \in \{$ **true**, **false**, **null** $\}$, we can easily find the begin position (based on the end position) and create full spans for these tokens.

Numbers

Tokenization for numbers is slightly more complex than the keywords one because the regular expression for JSON numbers is $-?(0|[1-9][0-9]*)(.[0-9]+)?([Ee][+-]?[0-9]+)?$. Just like tokenizing keywords, this expression heavily relies on concatenation of subexpressions, however, now we have other subexpressions with repetitions given by symbols ***** and $+$ as well as optional subexpression given by **?**. Borrowing Cameron's algorithm (2014) one more time, we have:

- Concatenation of subexpressions and generation of error streams for them is done as described in the previous subsection (**Keywords**).
- If the expression r is a repetition of a character class expression of form $C*$, then we find the first occurrences of character class C in input stream s and current marker m (if any) by applying `C' = Advance(~C, 1) & C & m`. Finally, we use the *ScanThru* operation [11] to find the immediate valid position that marks the following bit after the end of pattern $C*$ if that pattern exists, or zeroes otherwise, that is, no occurrences of C found. The new marker m' is defined by `m' = ScanThru(C',C) | (C'⊕m)` and represents a new valid marking position for r. Note that because the Kleene star $(*)$ operator can have zero or more occurrences, no error streams need to be created for this case.
- If the expression is a repetition expression of form $R+$, then it gets converted into the concatenation of form $RR*$. In this case, an error stream is created to make sure the first R always exists.
- If the expression r is an optional expression of form $R?$, the output marker stream is the bitwise-or of the stream with zero occurrences of R matched and the output stream produced by compiling R with one occurrence of R matched and shifted forward. Note that we do not create any error streams here as optional expressions always match the input because they can have zero or one occurrences.
- At the end, we join all error streams that are given by the concatenation of subexpressions and $+$ operator. If the final stream has any of its bits set, then s is not a valid string for G, otherwise, if all bits are zeroes, s is a valid string for G.

It is important to notice that the original algorithm [11] is not intended for tokenization – it only marks any valid occurrences for a given regular expression, so we had to modify

it to get number spans: we find all occurrences of a number after a dot `.` and after one of `eE` characters, remove them, and finally, find the first digit **b** of the number token that should always be a `-` or a number `[0-9]`. We always make sure that the first digit **b** is after a token break character. Likewise, we had to mark the last digit **e** of the number so we could create a span from begin to end. This was done similarly to the way we find **b**, only now we find the last digit **e** that is followed by a non-number that is not a `.`, `eE` or `+-`. In the end, we create a span from **b** to **e** and any possible errors are marked in error streams that were described in the steps above.

Token breaks and validation

After excluding string spans from our input, token breaks are generated by the rules described below for a pair of consecutive characters:

- Whenever a white space character is followed by non-whitespace or vice versa.
- Any occurrence of the individual tokens `{ } [ ] : ,` after any other character.

Previous subsections explain how each type of token is validated and §3.3.3 gives an overview on how we keep only valid bytes before the tokenization process.

2.2.2 JSON syntactic parsing

Creating parsing algorithms for a context-free grammar G can be done by finding regular sub-grammars in G that can be SIMDized in parallel (similar to what is described in §2.2.1). Once we have marked the positions of tokens in a given grammar G, we can either (1) process the CFG parts sequentially by using a stack, or (2) we can find the total depth d^{MAX} of G and its regular sub-grammars and, for each depth d_i, $i \in (0, d^{MAX})$, run the parsing rules for G in a parallel manner. In the end, the algorithm returns *true* if every depth d_i is valid, and $false$ otherwise. Approach (2) is what we implement in this thesis. The complete details for this algorithm can be found in §3.4.

2.3 Parabix — Parallel bit streams

Parabix is a highly scalable framework that works very well for text/stream processing [25]. Lin et al. (2012) mention that the fundamental difference between the Parabix framework and traditional text processing models is how Parabix represents the source data: it transposes k-bit streams into k-separate bit streams. This way, for example, in an SSE machine, where the block width is equal to 128 bits, Parabix is able to process 128 bytes at a time; in an AVX2 machine, Parabix is able to process 256 bytes at a time and in an AVX-512 machine, Parabix processes 512 bytes at a time. The reason why Parabix scales well is because as the block width increases, the more bytes it can process at a time. Table 2.1 shows

Table 2.1: Tradition models for text/stream processing

`hello`	`.11.1...`	`.11..1.1`	`.11.11..`	`.11.11..`	`.11.1111`

Table 2.2: Parabix's model for text/stream processing

	`h`	`e`	`l`	`l`	`o`
b_7	.	.	.	.	.
b_6	1	1	1	1	1
b_5	1	1	1	1	1
b_4	.	.	.	.	.
b_3	1	.	1	1	1
b_2	.	1	1	1	1
b_1	.	.	.	.	1
b_0	.	1	.	.	1

the representation of the string `hello` in traditional text processing models and table 2.2 shows the same representation using parallel bit streams in Parabix.

On table 2.2, we see that the byte stream for the source `hello` was transposed, and instead of using the traditional one-byte-array model for ASCII characters, the source is transposed into 8-completely-separate bit streams b_0 through b_7, representing whether or not that bit is set for each one of the bytes of the source. This characteristic makes the Parabix framework very unique if compared with other stream representation models. In Parabix, for the most part, branching is abstracted using Boolean operations, and because it handles bits rather than bytes, it uses a large number of Boolean operations which cost only a few cycles to fully run in parallel in SIMD architectures. It is also built as an LLVM frontend so the final assembly generated is as fast as the current state-of-the-art techniques for compilers.

2.3.1 SIMD — Single Instruction Multiple Data

Parabix relies a lot on SIMD instructions and vector extensions as a mean to parallelize stream processing. SIMD stands for Single Instruction Multiple Data [8] and is a type of instruction that has been added to various architectures (Intel, AMD, ARM, POWER) in order to perform parallel processing on multiple data points simultaneously, that is, SIMD instructions allow processors to process more data at once. While traditional instructions may not completely use a register, SIMD instructions do not waste space. For example, when we load 8 bits in a 64-bit register with a traditional instruction, 56 bits of that register are unutilized, but the same is not true for SIMD. To better picture this, imagine we have an array with $8 \times i8$ integers `arr = [0,10,20,30,40,50,60,70]` and we want to add 2 to each of these integers. Below is how we would do it using traditional versus SIMD instructions (64-bit architecture):

// Traditional

1: **for each** c **in** arr **do**
2: $\quad arr \leftarrow arr + 2$
3: **end for**
4: *// result is* $arr = [2, 12, 22, 32, 42, 52, 62, 72]$

// SIMD
1: $arr \leftarrow$ **simd_add_i8**(arr, 2) *// every architecture has their own instrisics*
2: *// result is* $arr = [2, 12, 22, 32, 42, 52, 62, 72]$

As you can see above, while traditional algorithms need eight instructions to add a value to 8 integers of type $i8$ in an array, SIMD only uses one instruction as it is able to process multiple data with a single instruction.

2.3.2 StreamSet

Inside the framework, a StreamSet is a memory buffer that represents a set of bit streams that can be read/written by kernels for means of data transfer and are abstracted to be of infinite length. Tables 2.1 and 2.2 have two examples of StreamSets. Table 2.1 can be defined as a stream set consisting of a single stream of $i8$ values ($1 \times i8$) and the second StreamSet, in table 2.2, can be defined as a stream set of eight streams of $i1$ values ($8 \times i1$).

2.3.3 Marker Streams

Marker streams are StreamSets §2.3.2 that mark positions where their bits are set, that is, a stream that has no matches is comprised of all zeroes (all bits are unset) and a stream that matches all bytes is comprised of all ones (all bits are set). For example, `1..11..` is the marker stream for any letter 'a' in input `abcaaef`.

2.3.4 Kernel

A Parabix kernel can be abstracted as a black box (or function) that can be either stateful or state-free and that has StreamSets as inputs and outputs and can communicate with one another throughout the Pipeline with the use of StreamSets, but they cannot change the behavior of external kernels. The example in §2.4 is a Parabix kernel that finds digits $[0-9]$ for a given StreamSet.

2.3.5 Pipeline

In Parabix, a pipeline is what defines the data flow among kernels in a multi-threaded linear pipeline manner [17], which guarantees that the flow of data goes from prior to subsequent kernels as data is available, and divides every kernel's lifecycle into three stages: initialization, segment-processing and finalization. *Initialization* is the stage where kernels

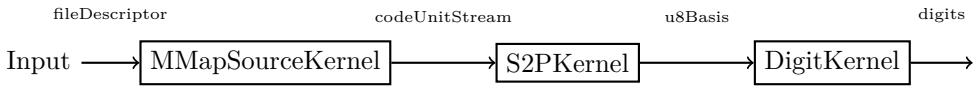

Figure 2.4: Pipeline for finding digits $[0-9]$ in source stream

are allowed to make internal configurations or allocation for the work that will be done, *segment-processing* is the stage where the actual work is done, for example finding digits $[0-9]$ in input StreamSet, and *finalization* is the stage where kernels can release any of the memory that was allocated during initialization [18]. Figure 2.4 represents a pipeline for §2.4, where *MMapSourceKernel* takes a $fileDescriptor$ scalar as input and returns a stream of code units and passes it as input to kernel *S2PKernel*, SP = serial to parallel, that returns the representation of the input as 8-completely-separate bit streams, just like the representation on table 2.2. Finally, *DigitKernel* takes an *u8basis* stream and marks positions in the stream where the ASCII character is one of the digits $[0-9]$.

2.4 Parabix character class compiler

The Parabix framework includes a character class compiler that optimizes Boolean operations for regular expressions. It uses a large number of binary Boolean operations **and**, **or**, **xor**, and **not** (which can be expressed by XORing the input expression with ones) as described in previous works [25, 11, 5].

Within the framework, it is possible to get the least amount of Boolean instructions necessary for computing a character class. The character class compiler is used to automatically produce bitstream logic for all the individual characters (such as delimiters) and character classes (digits, letters, etc...) used in a particular application. To illustrate that, imagine we want to compute the digits character class $[0-9]$. With only a few Boolean operations, it is possible to determine whether or not (ASCII) digits are contained within a stream:

Input: $[0-9]$, binary form $[00110000-00111001]$

Output:
$temp_1 = b_0 \mid b_1$
$temp_2 = b_2 \;\&\; b_3$
$temp_3 = temp_2 \;\&\; \sim temp_1$
$temp_4 = b_5 \mid b_6$
$temp_5 = b_4 \;\&\; temp_4$
$is\_digit = temp_3 \;\&\; \sim temp_5$

The code above computes the bit logic necessary to identify whether an ASCII character is part of the digits character class $[0-9]$. In general, we see that the approach used in Parabix is much faster than conventional ones. Counting all Boolean operations that were outputted (including two **not**s), Parabix **only** needs 8 instructions per block to mark all the positions of all the digits in the source stream with 1 bits. Note that Parabix processes block-width-bytes at a time, as mentioned previously. Therefore, in an AVX-512 machine, with only 8 instructions, Parabix is able to process 512 bytes at a time and output the marker stream for digits. That is much faster than traditional algorithms that are likely to only process one byte at a time with the comparison `if` $(c \geq$ `'0' and` $c \leq$ `'9')`. In better algorithms that use SIMD parallel power, in an AVX-512 machine, you can compare 64 bytes at a time, which is still 8 times less than Parabix at a higher cycle rate and branch mispredictions. The digit character class is a relatively simple case, but for more complex cases of character classes, Parabix usually still does much better than traditional approaches.

2.5 Related works

2.5.1 Parabix icgrep and XML parser

Cameron et al. (2008) prove that using the bit-stream model on SIMD can considerably improve the parsing of regular language bits within context-free grammars [9]. In their work, they present an XML parser as a case study, which is prior to the Parabix framework, that is able to leverage from SIMD capabilities and dramatically improve XML parsing time. Later in 2015, Cameron et al. write a case study in the Parabix framework named **icgrep** [5], which is a tool for searching plain-text data that was created to compete with the standard Unix grep. **icgrep** is shown to have a very good performance for regex matching. It does so by optimizing regular expressions using Parabix's character class compiler, which is explained in §2.4.

2.5.2 simdjson

Langdale et al. (2019) created a JSON parser named **simdjson** that achieves impressive parsing rates [15]. **simdjson** uses optimized bit logic that reduces the total number of cycles and avoids branching mispredictions and cache misses while parsing a JSON file. In their approach, they are able to improve JSON parsing time by breaking down parsing into two passes (most traditional algorithms only do one pass). In the first pass, they validate character encoding and find the starting location of all JSON nodes and, in the second pass, they process all nodes and structural characters in parallel with the use of SIMD instructions.

2.5.3 yyjson

Yyjson is an extremely performative parser for JSON compatible with C89. Lemire (2020) mentions that yyjson could be used in the future as a fallback for simdjson, since it works well on operating systems that have access to SIMD instructions as well as on operating systems that do not support SIMD, and for a few cases, yyjson can be faster than simdjson. Yyjson relies on a 128-bit tape to keep information of each JSON node in a document, where the higher 64-bits represent metadata related to the current node and the lower 64-bits keep the address to the next node. All values for nodes in yyjson are kept in another contiguous memory area.

2.5.4 Mison

The Mison parser [29] is based on a parallel algorithm using SIMD that leverages the fact that Data Analytics applications typically only use only a certain number of fields in a JSON document, removing the need to validate the full document. Mison first builds a structural index and then speculates on the schema to directly jump to the position of the requested field, avoiding parsing irrelevant fields. Mison's design is based on the assumption that JSON data has limited structural variants and the position of a field can be quickly determined with SIMD instructions, since finding the exact position of a JSON field in a JSON record can be done by looking at only a few delimiter characters such as `:`.

2.5.5 Current research on other data format types

Although the use of parallel algorithms with SIMD in commodity processors has shown great results in parsing data formats such as JSON and CSV, there seems to exist a lack of research related to other less popular data formats, such as YAML. It is worth noting that the creators of simdjson, Geoff Langdale and Daniel Lemire, have also researched a CSV parser that follows the principles of simdjson. Likewise, Parabix has conducted research on parallel algorithms for parsing XML and CSV parsers. Aside from these, most papers on this topic are related to Data Analytics [2, 23, 29]. One reason for the small amount of research on parsing other data formats is that JSON and CSV parsing are currently hot topics because of their popularity, however, this presents an opportunity for researchers to develop and improve parsers for other data formats such as YAML and XML, which could benefit a range of applications in the industry.

Chapter 3

Design and Implementation

The JSON parser here presented heavily relies on the use of Parabix's SIMD extensions and multithreading support. In this parser, we divide the parsing process into four different stages: bitwise transposition of input data, tokenization, syntactical analysis and error processing:

- In stage one, we prepare and process our input in parallel by receiving a (scalar) file descriptor and transposing the basis stream, that is, the source input, into 8 separate bit streams that represent the bits 0 through 7 of every corresponding byte in the input, as described in §2.3, and feed that to our next group of kernels that are in charge of validating and classifying bytes.

- Stage two is in charge of validating and tokenizing bytes from the output stream of stage one. This stage can be broken down into two phases, where phase (1) is responsible for finding string spans starting and ending in double quotes (") and checking whether the source stream is contained of only valid UTF-8 characters and phase (2) is responsible for classifying all other (valid and invalid) symbols and structural characters that are **not** inside string spans from phase (1). The output of stage two is a stream that marks all combined lexemes in the source stream and their respective spans.

- In stage three, given marking positions of structural characters {, [,], } and a max depth constant d^{MAX}, we find the nesting depth of the JSON document and parse both terminal symbols, objects and arrays in parallel, as shown in §3.4 and §3.6.

- Finally, stage four merges all error streams and checks if the final error stream contains any errors. If any of the bits in the final error stream is set, we have a parsing error in the document, for instance, an invalid character, otherwise, the document is a valid JSON document.

Figure 3.1 shows what the Parabix pipeline for this solution looks like, where the pipeline P is a DAG and every kernel k is a node with incoming edges that represent its input streams and with outgoing edges that represent the output streams it produces.

Further sections explain every kernel's responsibility and how the pipeline processes their work in parallel.

3.1 Data parallelism among kernels

As discussed by Medforth (2022) and described in §2.3.5, Parabix's kernels are encapsulated in a multi-threaded linear pipeline — figure 3.1 shows the complete pipeline for the JSON parser —, which means that the data flows from prior to subsequent kernels once per segment such that the data is processed in parallel and the number of L2 cache misses is minimized, which means that the fewer data dependencies, the faster it gets.

This characteristic in Parabix differs greatly from other solutions that parse JSON in parallel, such as simdjson [15], which parses the JSON input in two separate passes, one after the other. In other words, because the work in Parabix is done segment by segment, we are able to process kernels in a multi-threaded environment and parallelize data as much as possible, whereas the same is very hard to do in a solution where data partition is not well-defined and the pass inputs depend on another pass' outputs.

3.2 Bitwise transposition of input data

3.2.1 MMapSourceKernel and S2PKernel

Most tools in Parabix need these two kernels as they are in charge to transpose the input from serial to parallel bitstreams and serve as base input to all other kernels [9]. For a brief explanation on how that works, check §2.3.

3.3 Input validation and classification

3.3.1 UTF-8 validation

Cameron (2008) shows that with about 0.5 cycles per byte, parallel bit streams instructions can be used to identify characters that are invalid in UTF-8 [1, 9], so we use that work to verify whether the input contains only valid UTF-8 characters. In figure 3.1, the kernel in charge of UTF-8 verification is *ValidateUTF8* kernel.

3.3.2 Finding string spans

Because all keys in JSON objects are strings, the majority of nodes in JSON documents are comprised of strings (both keys and values), thus, finding string spans before processing other symbols and parsing the JSON document is of extreme importance as we are able

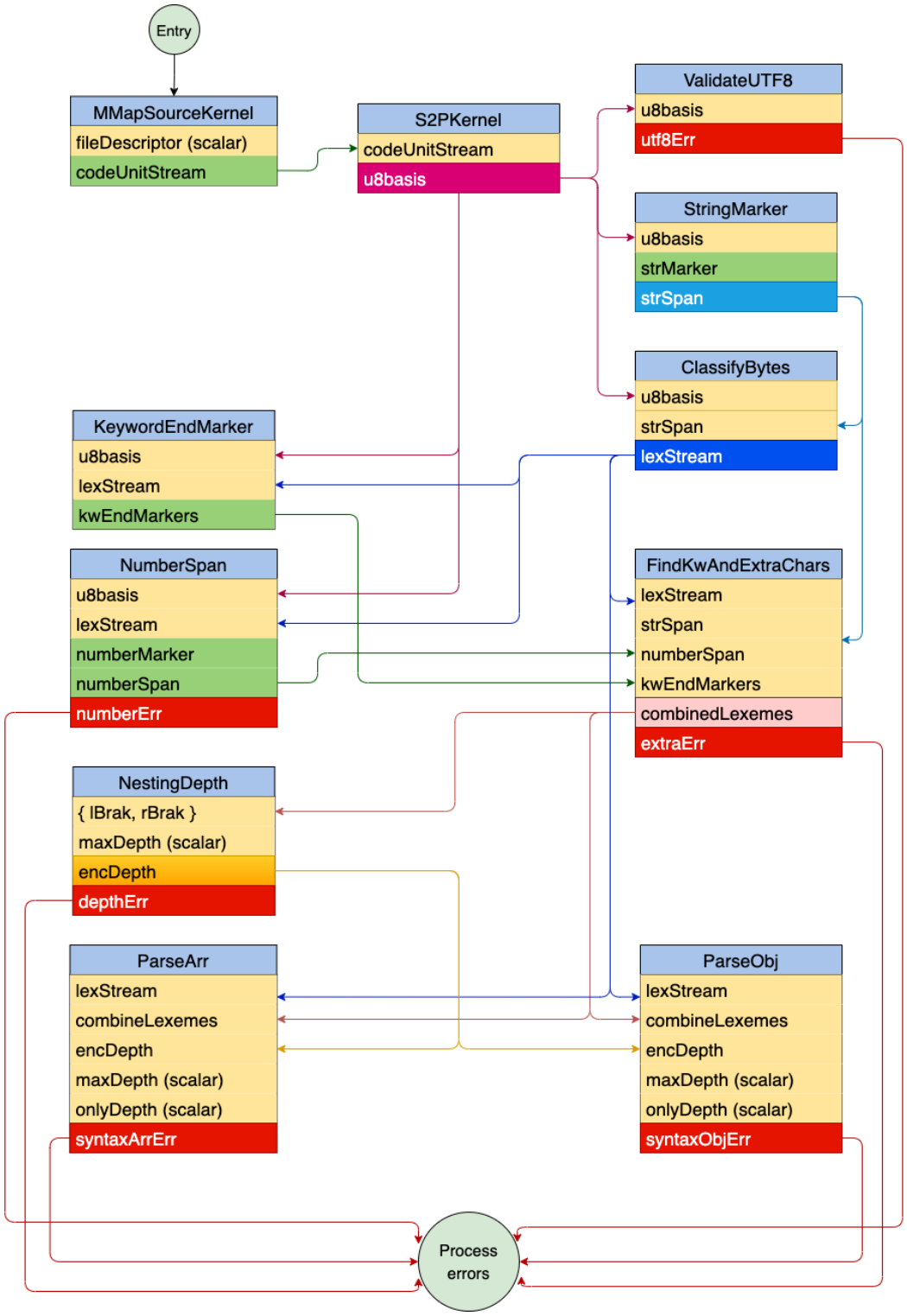

Figure 3.1: JSON parsing with Parabix

to filter out most of the document nodes and parse the rest (non-string nodes) as fast as possible.

In fact, with only a few operations in Parabix, it is possible to create a kernel that finds all valid string spans in a JSON document in parallel. Take a look at the algorithm below:

Input: basis stream
Output: string markers and spans

1: $dQuote \leftarrow$ MakeByte(")
2: $backslash \leftarrow$ MakeByte(\ )
3: $even \leftarrow$ Repeat(0xAA)
4: $odd \leftarrow$ Repeat(0x55)
5: $backslash^{1st} \leftarrow$ Advance($\sim backslash, 1$) & $backslash$
6: $backslash^{even} \leftarrow backslash^{1st}$ & $even$
7: $backslash^{odd} \leftarrow backslash^{1st}$ & odd
8: $escaped^{even} \leftarrow$ ScanThru($backslash^{even}$, $backslash$) & odd
9: $escaped^{odd} \leftarrow$ ScanThru($backslash^{odd}$, $backslash$) & $even$
10: $str^{marker} \leftarrow dQuote$ & $\sim$($escaped^{even}$ | $escaped^{odd}$)
11: $str^{span} \leftarrow$ InclusiveSpan(str^{marker})

To find all string spans and markers, first we tell the framework to mark all positions of valid double quotes (") and backslashes (\) and to store the results in streams *dQuote* and *backslash*, lines 1 and 2, respectively. After that, we need to find all quotes that are escaped by backslashes (\"), which can be done by creating even and odd constant streams (0xAA = $0b10101010$ and 0x55 = $0b01010101$) that are repeated to have the same length as the initial basis input. Considering a sequence of backslashes, if an odd-bit starting sequence ends on an even bit position or if an even-bit starting sequence ends on an odd bit position (lines $3 - 9$), where the next character is a double quote, that double quote is escaped and therefore we filter it out, creating a final stream of only valid double quotes (line 10), that is double quotes that are not escaped. Finally, with string markers representing only valid quotes, we are able to create string spans with the inclusive span operator (line 11). Example: *inclusiveSpan*(.1.1...1...1) = .111...11111. Figure 3.2 shows an example of execution of the *StringMarker* kernel.

3.3.3 Marking valid bytes

Once we validate our input and have string span positions, with the output of *StringMarker* kernel, we are able to filter out our streams, described in §3.3.2, and check for other valid symbols outside the string spans.

basis	{ " \ " e x a m p l e \ \ \ " " : " x D " }
dQuote	. 1 . 1 . . . . 1 1 . 1 . . 1 .
backslash	. . 1 . . . . 1 1 1 . . . .
even	. 1 . 1 . 1 . 1 . 1 . 1 . 1 . 1 . 1 . 1 . 1
odd	1 . 1 . 1 . 1 . 1 . 1 . 1 . 1 . 1 . 1 . 1 .
$backslash^{1st}$	. . 1 . . . . 1 . . . .
$backslash^{even}$	. . . . 1 . . . .
$backslash^{odd}$	. . 1 . . . .
$escaped^{even}$	. . . . 1 . . . .
$escaped^{odd}$	. . . 1 . . . .
str^{marker}	. 1 . . . . 1 . 1 . . 1 .
str^{span}	. 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 . 1 1 1 1 .

Figure 3.2: Creating string spans with *StringMarker* kernel

As mentioned before, after filtering out string spans, only a few characters are valid outside these spans, where they can only be either structural characters (curly braces, square brackets, colon and comma) or part of valid lexemes of keywords and numbers (§3.3.4), for example, 't', 'r', 'u', 'e', that together represent the keyword **true**. Thus, this kernel (*ClassifyBytes*) is a very simple kernel that uses Parabix to mark the positions of single characters by using built-in function *makeByte* as shown on §3.3.2 and has only two tasks: (1) to mark all valid characters that are not inside string spans and (2) to find any characters that are neither structural characters nor part of valid lexemes.

3.3.4 Finding keyword and number spans

Immediately after we classify the rest of the bytes (§3.3.3) that are outside the string spans, we are able to mark the span positions of keywords **true**, **false** and **null**, as well as to find numbers that are matched with regex $-?(0|[1-9][0-9]*)(.[0-9]+)?([Ee][+-]?[0-9]+)?$ (§2.2.1). At the end of this step (kernels *keywordEndMarker*, *NumberSpan* and *FindKwAndExtraChars*), we have all stream markers and spans that are needed to represent JSON symbols (structural characters, strings, numbers and keywords) and we are able to proceed to the next stage, syntactical analysis, which is discussed in §3.4.

3.4 Syntactical analysis

As mentioned in §2.2, JSON has a very straightforward grammar, and in fact, depending on the language, a byte-by-byte parser for it can be easily done with around 100 lines of code, however, when done in parallel, it can be quite challenging. That is to say, we break down the syntactical analysis in two steps: (1) finding the nesting depth of the JSON document with a given max depth d^{MAX} and (2) parsing JSON nodes given the nesting depth of characters in the document. Further subsections demonstrate how the kernels are divided for both steps (1) and (2).

Bracket Stream	`. . . [ . . { . } . . { . . . [ . [ . ] . . ] . . } . . . ] . . .`
Nesting Depth	`0 0 0 1 1 1 2 2 2 1 1 2 2 2 2 3 3 4 4 4 3 3 3 2 2 2 1 1 1 1 0 0 0`
L	`. . . 1 . . 1 . . . . 1 . . . 1 . 1 . . . . . . . . . . . . . . .`
R	`. . . . . . . . 1 . . . . . . . . . . 1 . . 1 . . 1 . . . 1 . . .`
ND BixNum[2]	`. . . . . . . . . . . . . . . . . 1 1 1 . . . . . . . . . . . . .`
ND BixNum[1]	`. . . . . . 1 1 1 . . 1 1 1 1 1 1 . . . 1 1 1 1 1 1 . . . . . . .`
ND BixNum[0]	`. . . 1 1 1 . . . 1 1 . . . . 1 1 . . . 1 1 1 . . . 1 1 1 1 . . .`

Figure 3.3: *NestingDepth* kernel example

3.4.1 Finding nesting depth

The *NestingDepth* kernel is a Parabix kernel, explained in §3.6, that can compute the nesting depth of characters in an input stream, given two character class bit streams *L* identifying all the left or opening delimiters and *R* identifying all the right or closing delimiters and a parameter d^{MAX} identifying the maximum nesting depth expected for an input source. The nesting depth is encoded with *BixNum*, where the number of streams for *NestingDepth BixNum* must be equal to $ceil(log_2(d^{MAX}+1))$. Figure 3.3 is an example of what the output of *NestingDepth* looks like for considering JSON delimiters {, }, [and] for json bracket stream `...[..{.}..{...[.[.]..]..}...]...`.

3.4.2 Syntax validation using nesting depth

Consider the problem of validating JSON array syntax, where whitespace characters (space, tab, new line) may be freely included between tokens:

array ::= '[' [value {',' value}] ']'
object ::= '{' string : [value {',' value}] '}'
value ::= object | array | number | string | 'true' | 'false' | 'null'

Validating non-nested JSON

A JSON grammar is given by JSON ::= value, meaning that not all JSON documents have nesting if they are of type `number | string | 'true' | 'false' | 'null'`, for example string `"This is valid JSON"` is a valid JSON by itself if it is the only node in the JSON document. What this means is, if a document does not start with a opening curly brace { or an opening square bracket [, that document **must** have depth zero, thus when we apply the *NestingDepth* kernel, the result returned must be `00000000000000000000`.

With this in mind, assume that *EOFbit* is the last bit of the source input (EOF), *ND* is a stream set that represents the nesting depths of nodes of a given JSON document and *valueToken* is a bit stream that marks the end position of any legal JSON numeral, string or keyword. The example below shows what this would look like if the JSON document was the string "This is valid JSON".

ND	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
valueToken	. . . . 1 .
EOFbit	 1

Now that we know what the streams look like, it is possible to check whether or not we have valid non-nesting values by running the algorithm below:

Input: *valueToken*, nesting depths *ND*, BixNum compiler *bnc*
Output: error stream with set bits in case of parsing failure

1: *otherND* ← bnc.UGT(*ND*, 0)
2: *zeroND* ← bnc.EQ(*ND*, 0)
3: *begin* ← ∼Advance(<1>, 1)
4: *valueAtZero* ← *valueToken* & *zeroND*
5: *stopAtEOF* ← ∼*EOFbit*
6: *firstValue* ← ScanTo(*begin*, *valueAtZero*)
7: *nonNestedValue* ← ScanTo(Advance(*firstValue*, 1), *anyToken*)
8: *errValue* ← ScanThru(Advance(*nonNestedValue*, 1), *stopAtEOF*)
9: *firstSymbol* ← ScanTo(*begin*, *symbols*)
10: *valueAtZeroAfterSymbol* ← ScanTo(Advance(*firstSymbol*, 1), *valueAtZero*)
11: *errSymbol* ← ScanThru(Advance(*valueAtZeroAfterSymbol*, 1), *stopAtEOF*)
12: *errEOF* ← *EOFbit* & *otherND*
13: *errSimpleValue* ← *errValue* | *errSymbol* | *errEOF*

To validate whether or not valid the JSON document has non-nesting valid values, we have to verify three rules:

- If we have a non-nested value at depth zero, we can only have one (terminal) node in the whole document, lines $6 - 8$. This happens because according to the JSON grammar, only one value can be the root element, thus, if a root element is also a terminal node, only that node can exist.

- If we have any structural character, we cannot have any terminal node at depth zero, lines $9 - 11$. This is because in that case, a JSON document must have started in either an object or array and all nodes would have to be in depth $d > 0$. By definition, arrays and objects make the JSON structure a nested structure.

- EOF bit is **always** at depth zero, otherwise we have unmatched parentheses, line 12, for example, a valid stream `.{..}.` would have *ND* as `011110` (EOF bit is zero), whereas an invalid stream `.{....` would have *ND* as `011111` (EOF bit is one).

Validating arrays

Assume we have two bitstreams for tokens: *valueToken*, a stream that marks the positions of all values we may have, and anyToken, a bitstream marking the end position of any legal or illegal token. Assume also that *LBrak*, *RBrak*, *RBrace*, and *Comma* are streams marking the position of JSON [,], }, and ',' tokens respectively. Suppose that a BixNum ND has been computed as the nesting depth involving the left and right delimiter sets consisting of square brackets and braces. Then the validation of array syntax at each nesting depth d can be determined as follows: each array will be a span from an opening [to its corresponding closing]. Between these brackets will be other tokens as well as nested arrays and objects. However, nested arrays and objects can be easily computed as those spans of elements having a nesting depth greater than d. Using a BixNum compiler *bnc*, the spans may be computed as bnc.UGT(*ND*, d). Tokens at depth d can be identified as those at positions identified by the stream bnc.EQ(*ND*, d). The following code can be used to validate all arrays at depth d, determining whether there are any errors after the opening *LBrak* or after any value or *Comma*.

Input: *LBrak*, *RBrak*, *RBrace*, *Comma*, nesting depths *ND*, BixNum compiler *bnc*
Output: error stream with set bits in case of parsing failure

```
1:  zeroND ← bnc.EQ( ND, 0 )
2:  validEndValues ← ( valueToken | RBrak | RBrace ) & ∼zeroND
3:  afterToken ← Advance( validEndValues, 1 )
4:  tokenNext ← ScanThru( afterToken, whitespace )
5:  errAfterValue ← tokenNext & ∼( Comma | RBrak | RBrace | zeroND )
6:  advComma ← Advance( Comma, 1 )
7:  validBeginValues ← ( valueToken | LBrak | LBrace ) & ∼zeroND
8:  errAfterComma ← ScanTo( advComma, anyToken ) & ∼validBeginValues )
9:  for every d in parallel, d ∈ 1...d^MAX do
10:     atDepth ← bnc.EQ( ND, d )
11:     nested ← bnc.UGT( ND, d )
12:     arrayStart ← atDepth & LBrak
13:     arrayEnd ← ScanThru( arrayStart, nested | ( atDepth & ∼( RBrak | RBrace )))
14:     errorAtEnd ← arrayEnd & RBrace
15:     sArrAnyToken ← ScanTo( Advance( arrayStart, 1 ), anyToken )
16:     errAfterLBrak ← sArrAnyToken & ∼( nested | ( valueToken & atDepth ) | RBrak )
17: end for
```

To check whether the arrays in a JSON document are valid, we need to validate the following rules:

- Every value in an array must be followed by either a comma, *RBrak*, lines $2-5$. The sub-production rule of arrays in the JSON grammar (§2.2) defines every value in an array to be followed by a *comma* or *RBrak*, but the reason why we allow the value to be followed by a *RBrace* in the code above is because, at this point, this step is done completely in parallel and that way we do not have to check whether that value belongs to an array or an object, thus, in practice this rule allows us to partially validate values in objects as well.

- Every comma must be followed by a value, lines $6-8$. This rule is **always** valid for any value in a JSON document, whether that value is inside an array or object. This is because arrays have the form `[value, value, ...]` and objects have the form `{ string : value, string : value, ... }` and because a *string* is a value, this rule holds true for all values, and again, just like the previous rule, we do not have to check whether a particular value belongs to an array or object to apply this rule.

- Every *LBrak* must have a matching *RBrak*, lines $10-14$. Every array in a given depth d must have an opening bracket at the beginning of its span with a closing bracket at the end of this span. §3.6 explains how that is done in parallel.

- Every *LBrak* must be followed by a value or *RBrak*, lines $15-16$. Here we validate whether, within its span, an opening bracket is followed by a value or a *RBrak*, in case it is empty. This rule requires us to know which nesting depth d that array is because it could be that the next value would be nested (therefore not at the same depth). In case the array is empty, this rule overlaps with the brackets matching rule.

- As a final and important rule, all values must be in nesting depth $d > 0$, lines 2, 5 and 7. This happens because we already validated $d = 0$ when we validate if the JSON document has non-nested nodes.

Validating objects

Assume we have two bitstreams for tokens: *valueToken*, a stream that marks the positions of all values we may have, and *anyToken*, a bit stream marking the end position of any legal or illegal token. Assume also that *LBrace*, *RBrace*, *DQuote*, *Colon* and *Comma* are streams marking the position of JSON {, }, ", : and ',' tokens respectively. Just like we did when validating arrays, assume that a *BixNum ND* has been computed as the nesting depth involving the left and right delimiter sets consisting of square brackets and braces. Validating the object syntax at each nesting depth d can be determined as follows: each object will be a span from an opening { to its corresponding closing }. Between these braces will be other tokens as well as nested arrays and objects. As before, nested arrays and objects can be computed as those spans of elements having a nesting depth greater than d with the built-in function bnc.UGT(*ND*, d) and tokens at depth d can be computed with built-in

function bnc.EQ(*ND*, *d*) included in the BixNum compiler. The following code can be used to validate all objects at depth *d*, determining whether there are any errors after the opening *LBrace* or after any value or *Comma*.

Input: *LBrace*, *RBrace*, *DQuote*, *Colon* and *Comma*, nesting depths *ND*, BixNum compiler *bnc*

Output: error stream with set bits in case of parsing failure

1: *str* ← *valueToken* & *DQuote*
2: *zeroND* ← bnc.EQ(*ND*, 0)
3: *validStr* ← *str* & ∼*zeroND*
4: *afterTokenStr* ← Advance(*validStr*, 1)
5: *tokenNextStr* ← ScanThru(*afterTokenStr*, *whitespace*)
6: *errAfterValueStr* ← *tokenNextStr* & ∼(*Comma* | *Colon* | *RBrace* | *RBrak* | *zeroND*)
7: *validBeginValues* ← (*valueToken* | *LBrak* | *LBrace*) & ∼*zeroND*
8: *advColon* ← Advance(*Colon*, 1)
9: *errAfterColon* ← ScanTo(*advColon*, *anyToken*) & ∼*validBeginValues*)
10: **for every** *d* **in parallel**, $d \in 1...d^{MAX}$ **do**
11: *atDepth* ← bnc.EQ(*ND*, *d*)
12: *nested* ← bnc.UGT(*ND*, *d*)
13: *objStart* ← *atDepth* & *LBrace*
14: *objEnd* ← ScanThru(*objStart*, *nested* | (*atDepth* & ∼(*RBrak* | *RBrace*)))
15: *objAtEnd* ← *objEnd* & *RBrak*
16: *objSpan* ← ExclusiveSpan(*objStart*, *objEnd*)
17: *errColonAtDepth* ← (*Colon* & *atDepth*) & ∼*objSpan*
18: *strAtDepth* ← *str* & *atDepth*
19: *advComma* ← Advance(*Comma* & *atDepth* & *objSpan*, 1)
20: *errAfterComma* ← ScanTo(*advComma*, *anyToken*) & ∼*strAtDepth*
21: *errorAtEnd* ← *objEnd* & *RBrak*
22: *sObjStartAnyToken* ← ScanTo(Advance(*objStart*, 1), *anyToken*)
23: *errAfterLBrace* ← *sObjStartAnyToken* & ∼(*nested* | *valueToken* | *RBrace*)
24: **end for**

To check whether the objects in a JSON document are valid, we need to validate the following rules:

- Every string in an object must be followed by either a comma, colon or *RBrace*, lines 3 – 6. This rule is similar to the first rule of array validation, however, here we only validate strings (both as keys and values) because all other types of values have been already processed when we validated arrays, as explained in previous subsection.

- Every colon must be followed by a value, lines $7-9$. This rule can be applied to the whole JSON document without knowledge of the nesting depth d because colons are only valid for objects, and therefore, a colon must be inside an object as guaranteed by further rule.

- Every colon in nesting depth d must be inside an object span, line 17. This rule is simple, if a colon is not within an object span, then it is outside of it and therefore it does not belong to an object, and if that is the case, that colon is invalid.

- Every comma at nesting depth d inside an object span must be followed by a key string, lines $18-20$. This rule is straightforward as a JSON key is always a string in the form `{ string : value, string : value, ... }`, thus, if any other value is after a comma at depth d, then that value is invalid.

- Every *LBrace* must have a matching *RBrace*, line 21. Every object in a given depth d must have an opening brace at the beginning of its span with a closing brace at the end of this span. §3.6 explains how that is done in parallel.

- Every *LBrace* must be followed by a value or *RBrak*, lines $15-16$. This rule requires us to know which nesting depth d that object is because the next value could be in a different depth. In case the object has no properties, this rule overlaps with the braces matching rule.

- Lastly, all values must be in nesting depth $d > 0$, lines 3, 6 and 7. This happens because we already validated $d = 0$ when we validate if the JSON document has non-nested nodes.

3.5 Error processing

Processing errors can be done by merging all error streams into one final error stream and checking whether or not any of the bits are set. If there is a bit set in position i of the final error stream, one or more kernels found an error in character of position i while processing the source input and an error is returned to the user. Otherwise, if the final error stream contains only zeroes, then the JSON document is valid and no errors are displayed.

3.6 More on solving bracket matching problem in parallel

The most important contribution of this research is how we solve the bracket-matching problem in parallel for every nesting depth d with a given max depth d^{MAX}.

The bracket-matching problem deals with parentheses (), square brackets [], braces {} and other syntactic elements that provide for nested syntactic structures with balanced sets of delimiters that traditionally can be solved with pushdown automata, as explained

in §2.1.2. In Parabix, a key concept that is used in the *NestingDepth* kernel to tackle the bracket-matching problem is BixNum, which is a stream set with $ceil(log_2(d^{MAX} + 1))$ streams, that are used to represent the nesting depth of syntactic elements at each position in a source stream. Other than d^{MAX}, the *NestingDepth* kernel also takes two input streams *LBrak* and *RBrak*, where *LBrak* represents all opening delimiters in the source stream and *RBrak* represents all closing delimiters in the source stream. After processing the input streams, *NestingDepth* kernel outputs two stream sets *encDepth* and *depthErr*, where *encDepth* is a BixNum that encodes the depth of each individual character in the source and *depthErr* marks the positions of characters that were not valid during encoding, which means that we are able to identify exactly where errors occurred by checking the bits that are set in stream *depthErr*, if a file has an invalid nesting depth. The algorithm we use to find the nesting depth of a source stream is as follows:

Input: *LBrak*, *RBrak*, max depth d^{MAX}, BixNum compiler *bnc*
Output: nesting depths *encDepth* and error stream *depthErr* with set bits in case of failure

1: *encDepth* ← create $ceil(log_2(d^{MAX} + 1))$ streams
2: *all_brackets* ← *LBrak* | *RBrak*
3: *bscan* ← AdvanceThenScanTo(*LBrak*, *all_brackets*)
4: *closed* ← *bscan* & *RBrak*
5: *errs* ← *bscan* & atEOF
6: *pendingL* ← *bscan* & LBrak
7: *span* ← InclusiveSpan(*LBrak*, *bscan*)
8: *encDepth*[0] ← *BixNum(span)*
9: **while** *pendingL* **do**
10: *unmatchedR* ← *RBrak* & ∼*closed*
11: *inPlay* ← *pendingL* | *unmatchedR*
12: *bscan* ← AdvanceThenScanTo(*pendingL*, *inPlay*)
13: *span* ← InclusiveSpan(*pendingL*, *bscan*)
14: *encDepth* ← bnc.AddModular(*encDepth*, BixNum(*span*))
15: *atMaxDepth* ← bnc.EQ(*encDepth*, BixNum(d^{MAX}))
16: *closed* ← *closed* | (*bscan* | *RBrak*)
17: *nextPending* ← *bscan* & *LBrak*
18: *tooDeep* ← *nextPending* & *atMaxDepth*
19: *errs* ← *errs* | *tooDeep* | (*bscan* & *atEOF*)
20: *pendingL* ← *nextPending* & ∼*tooDeep*
21: **end while**
22: *errs* ← *errs* | (*RBrak* & ∼*closed*)

The way the algorithm above works is by initially (lines $1-8$) finding the outermost brackets and setting the initial value of the stream $encDepth[0]$, where the bit is set if nesting depth is the outermost node and unset otherwise. After that, on lines $9-21$, the algorithm iterates and repeats the process depth-by-depth, and while doing so, it increments the inner elements that are matched in every iteration through the $pendingL$ stream. The process is repeated until all depths are satisfied or the depth is greater than d^{MAX}. Please note that both bnc.AddModular and bnc.EQ are built-in functions inside the BixNum compiler that allow us to perform modular addition (the results have the same length as the first argument) and check for equality on bit streams, respectively.

For illustration purposes, same as figure 3.3, imagine we have bracket stream S where `S = ...[..{.}..{...[.[.]..]..}...]..`. Because the max depth of this particular case is $d^{MAX} = 4$, we need $encDepth$ stream set to have $ceil(log_2(d^{MAX} + 1)) = 3$ streams to represent all respective bracket depths, as per line 1 in the algorithm. It is important to notice that in a real scenario we only have an estimation of what the max depth is, which can be done by analyzing various files or by setting an unrealistic number for it, $d^{MAX} > 15$ for JSON, for example. Because we want to keep this example simple, let $d^{MAX} = 4$ and our bracket stream be S. In that case, the algorithm starts on the depth $d = 1$, identifying the first span positions (up to second to last brace) and have `pendingL = ......1........1.1.............`, which will allow the loop to re-iterate and increment the inner depths that are remaining. As per the encoding of $encDepth$, figure 3.3 shows a set of 3 parallel bit streams (giving a 3-bit number at each character position) for S with the corresponding nesting depth of each element in the data stream and the nesting depth BixNum (labeled as ND).

3.6.1 Solving nested structures with nesting depths

Once we have computed the nesting depth for every character in the source stream, we can process every depth in parallel, since now we know where the nesting begins and ends. For example, the nesting for characters of figure 3.3 is `0001112221122223344433322211110 00` and the source input is `S = ...[..{.}..{...[.[.]..]..}...]..`. If we are trying to validate all objects (curly braces only) for this simple example in parallel, all we need to do is:

- Find if there are any opening braces at current depth d.
- Find all nesting depths within d. The span of 0011221100 is ..111111.. if we are in depth $d = 1$ and ....11.... if we are in depth $d = 2$.

- Find where the object ends and check if the object ends on a brace. If so, this object is valid, otherwise we have a parsing error.

See algorithm below:

Input: S, $LBrace$, $RBrace$, $RBrak$
Output: error stream with set bits in case of failure

1: **for every** d **in parallel**, $d \in 1...d^{MAX}$ **do**
2: $atDepth \leftarrow$ bnc.EQ(ND, d)
3: $nested \leftarrow$ bnc.UGT(ND, d)
4: $objStart \leftarrow atDepth$ & $LBrace$
5: $objEnd \leftarrow$ ScanThru($objStart$, $nested$ | ($atDepth$ & $\sim$($RBrak$ | $RBrace$)))
6: $errorAtEnd \leftarrow objEnd$ & $RBrak$
7: **end for**

Claim

Every depth d is independent and therefore can be fully parallelized.

Proof

By means of contradiction, assume that depth d_i has a dependency on depth d_j, where $i \neq j$. We have two cases:

- $i > j$: If i is greater than j, it means that d_i is at least one level deeper than d_j, thus, since every nested element is expected to be of valid syntactic structure, d_i does not depend on d_j because it is possible to process all nested elements in d_i without processing all nested elements in d_j. Therefore, assuming that d_i is dependent on d_j contradicts the fact that d_i can be processed independently from d_j as an individual syntactic structure.

- $i < j$: If j is greater than i, then it is possible to find and isolate d_i span with the output from *NestingDepth* kernel, for example by creating a stream that takes the output of bnc.UGT(ND, d_i). This way, at all times, depth d_j only needs to validate itself and expect that all other inner depths validate themselves on separate passes, but if that is ever the case, then d_j does not depend on d_i and, on its turn, d_i does not depend on d_j, as first assumed, which proves this to be a contradiction. □

How the actual implementation looks like?

Because we know beforehand the max depth d^{MAX} that is expected, we implement the actual code in a very simple manner: we copy the code for every separate depth d so that

1: $atDepth_1 \leftarrow$ bnc.EQ(ND, 1)
2: $nested_1 \leftarrow$ bnc.UGT(ND, 1)
3: $objStart_1 \leftarrow atDepth_1$ & $LBrace$
4: $objEnd_1 \leftarrow$ ScanThru($objStart_1$, $nested_2$ | ($atDepth_2$ & ∼($RBrak$ | $RBrace$)))
5: $errorAtEnd_1 \leftarrow objEnd_1$ & $RBrak$

6: $atDepth_2 \leftarrow$ bnc.EQ(ND, 2)
7: $nested_2 \leftarrow$ bnc.UGT(ND, 2)
8: $objStart_2 \leftarrow atDepth_2$ & $LBrace$
9: $objEnd_2 \leftarrow$ ScanThru($objStart_2$, $nested_2$ | ($atDepth_2$ & ∼($RBrak$ | $RBrace$)))
10: $errorAtEnd_2 \leftarrow objEnd_2$ & $RBrak$

11: $finalError \leftarrow errorAtEnd_1$ | $errorAtEnd_2$

Figure 3.4: Transformation of algorithm in §3.6 without branching for $d^{MAX} = 2$.

branching and data dependency is avoided, and let both Parabix and LLVM parallelize the instructions as needed. For $d^{MAX} = 2$, the algorithm above would be transformed into two blocks of code, as shown in figure 3.4, one for depth d_1 and one for depth d_2. At the end, the errors are merged and if no bits are set, then no parsing errors were found, otherwise, something is wrong in one of the depths.

Chapter 4

Evaluation

4.1 Experiments

This thesis separates its evaluation into three experiments: two experiments (§4.2 and §4.3) comparing our algorithm with the various parsing tools described in §2.5 and how they scale as file size and complexity increase and one experiment (§4.4) that shows how well our algorithm performs using different SIMD instructions with different block widths. We ran these experiments in two different SIMD machines (§4.1.1): one SSE4.2 with a 128-bit block width and one AVX-512, where we tried block widths of 256 bits with AVX-2 instructions and 512 bits with AVX-512 instructions. We compiled Parabix including our JSON parser as one of its tools with LLVM version 12 and the third-party parsers (simdjson, RapidJSON and yyjson) with their most up-to-date versions. In our own experiment — Parabix with different instruction sets, we run tests with one, two, three and four threads. In all other experiments, we only run tests on Parabix with 4 threads, which is the default number for that tool, and with 1 thread so we have a baseline comparison against the third-party parsers. It is important to notice that all third-party tools used in this evaluation only use one thread and do not add support for AVX-512 instructions; they use AVX and AVX-2 intrinsics in their code.

In this evaluation, we did not use the JSON test files that are used in previous works [15] because Parabix is focused on Big Data and those files did not exceed 3 megabytes, thus we would not be able to properly compare how the parsers scale as JSON file size and complexity increases, as explained in §4.5. Instead of that, we use a large range of randomly generated files, where the largest file has a size of 96MB and the smallest file has a size of 1.7MB. The contents of these files are a mix of randomized data with composed types {*array*, *object*} and terminal types {*string*, *number*, *boolean*, *null*}. In those randomized JSON files, we believe that we cover all cases for JSON parsing: small versus large files, dense versus sparse files and shallow versus deep nesting depths.

To accomplish this, we added three constraints p, d, and s to the script `RandomJson(p, d, s)` that generates random JSON objects, where (1) a JSON object file must have exactly p

properties at depth 1, (2) a JSON nesting must not exceed depth d at any point, and (3) a JSON file must have a size of at least $s \pm (s * 0.15)$ megabytes. For example, calling `RandomJson(10, 2, 6)` would create a random JSON object with 10 key-value properties, where none of these values would have nesting depth greater than one and the final size of the file, would be between 5.1 and 6.9 megabytes. These constraints allowed us to keep only files with similar characteristics for a fair comparison among the different tools used in this evaluation. The details on these randomly generated files and experiments can be found as appendices at the end of this thesis.

Further sections give a detailed explanation of the results obtained in this experiment.

4.1.1 Hardware Configuration

Two machines were used to evaluate and compare our work against the third-party tools': one SSE4.2 machine, where we evaluated AVX instructions generated by the tools presented using a 128-bit block width and one AVX-512 machine, which was used to test block widths of 256 and 512 bits using AVX-2 and AVX-512 instructions, respectively. The details and hardware specifications of these machines are listed in the table below:

Machine Name	Ubuntu_AVX512	Ubuntu_SSE4.2
Operating System	Ubuntu 18.04.4 LTS	Ubuntu 20.04.3 LTS
Architecture	X86_64	X86_64
CPU Model	Intel Xeon W-2102	Intel Core i7-3770
CPU MHz	2900	3400
CPU Max MHz	4000	3900
CPU Cores	4	4
L1 Cache	32KB	128KB
L2 Cache	1MB	1MB
L3 Cache	8MB	8MB
Memory	8GB 2666MHZ	4GB 1600MHZ
SIMD	AVX-512	SSE4.2

4.2 Performance as JSON file size increases

The intuition behind this experiment is to test how well our algorithm performs against simdjson, RapidJSON and yyjson as the file size increases with a fixed maximum depth. For that, we created JSON files with an average size of $\{2.3, 7, 16.3, 25.6, 40.2, 61.4, 83.6\}$ megabytes with maximum depth $d = 13$, that is, no file could have a nesting depth greater than 13. Figures 4.1, 4.2 and 4.3 represent the comparison for the results from tables in appendices A, B and C, which represent the tests that were performed using SSE4.2, AVX-

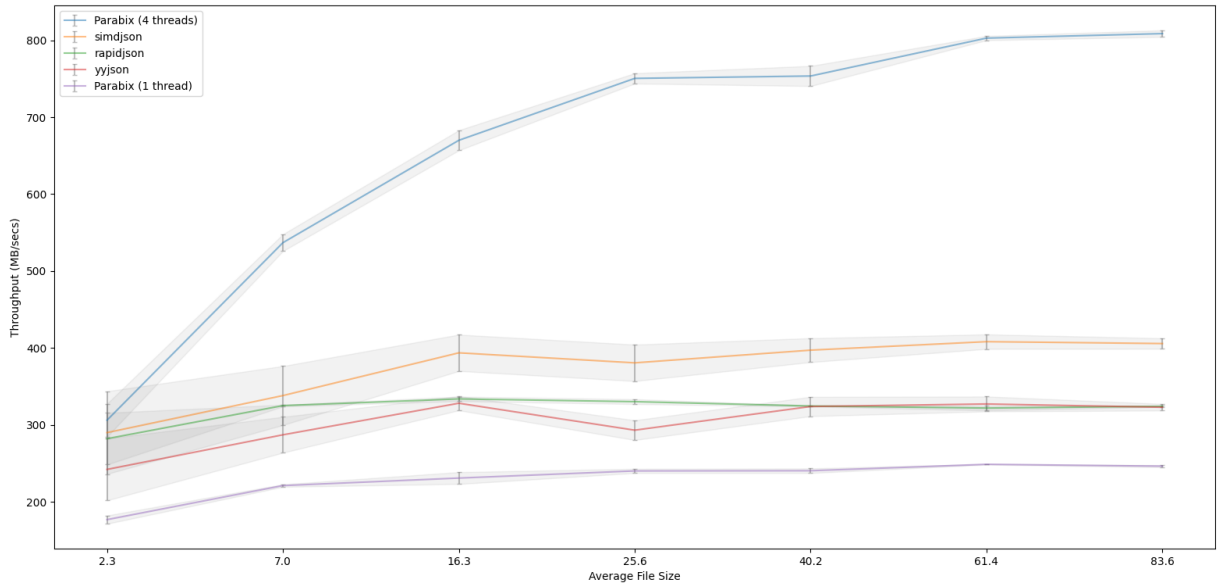

Figure 4.1: JSON parsing on SSE4.2 as file size increases

2 and AVX-512 instructions, respectively. Below is the formula for the contents of the randomly generated JSON files that were used in this experiment:

- $5 \times$ `RandomJson`$(p = 5,\ d = 13,\ s = 2.5)$
- $5 \times$ `RandomJson`$(p = 10,\ d = 13,\ s = 7)$
- $5 \times$ `RandomJson`$(p = 25,\ d = 13,\ s = 16)$
- $5 \times$ `RandomJson`$(p = 50,\ d = 13,\ s = 26)$
- $5 \times$ `RandomJson`$(p = 100,\ d = 13,\ s = 39)$
- $5 \times$ `RandomJson`$(p = 150,\ d = 13,\ s = 70)$
- $5 \times$ `RandomJson`$(p = 200,\ d = 13,\ s = 85)$

4.2.1 SSE4.2 performance as file size increases

In figure 4.1, the third-party tools as well as Parabix with 1 thread did not improve past the sixteen-megabytes-in-average mark, while Parabix with 4 threads only reaches a plateau at around the eighty-three-megabytes-in-average mark. One important thing to notice is that all tools included in this experiment reach a plateau at some point, however,

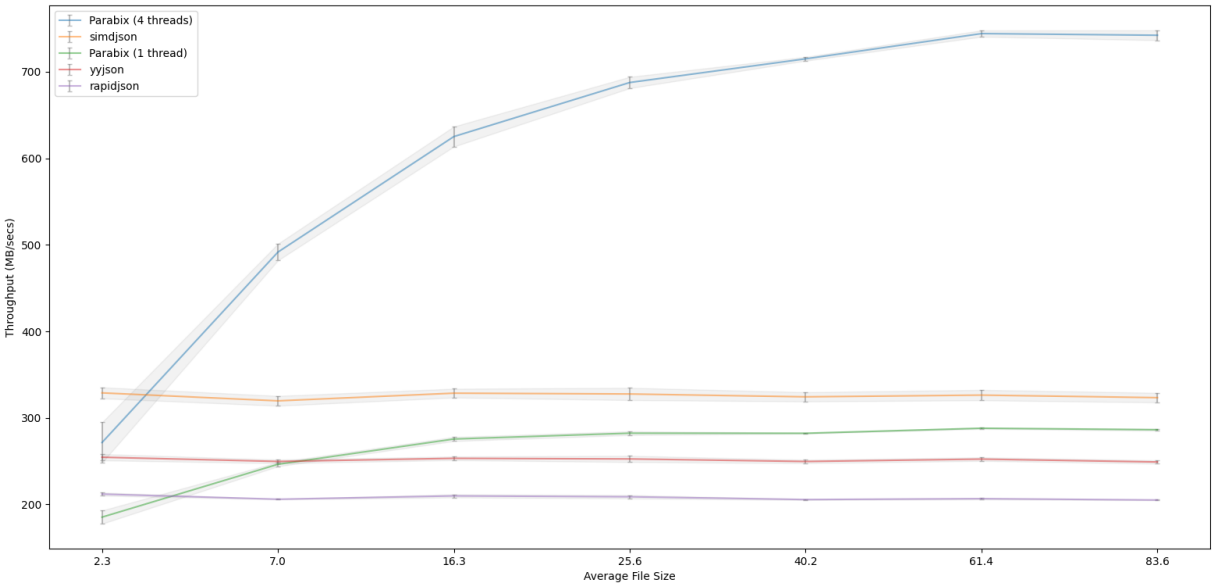

Figure 4.2: JSON parsing on AVX-2 as file size increases

Parabix takes longer to get there as the number of threads increase. Although taking too long to reach a maximum throughput is generally taken as a downside, Parabix with 4 threads performs much better than the other tools even for the smallest file with a size of 1.7 MB with a throughput of 252.68 MB/s against yyjson with 193.25 MB/s, simdjson with 177.81 MB/s and rapidjson 158.17 MB/s. This is more than a thirty percent speed up for a relatively small file, and if we compare its throughput against simdjson for the largest file (96 MB), we get a ninety-five percent speed up making Parabix almost twice as fast as simdjson on that case. Other than that, only simdjson reached a throughput of over 400 MB/s. In other words, although Parabix with 4 threads performed well, Parabix with 1 thread could not achieve a great performance as its throughput is extremely dependent on the depth of a JSON structure as shown in figures 4.4 and 4.7. What that means is that although Parabix running on a single thread did not perform very well for depth $d = 13$ on this case, it would generally not be the case had the maximum depth d been smaller, since our algorithm improves as d gets smaller but simdjson always performs the same for any depth d.

4.2.2 AVX-2/AVX-512 performance as file size increases

Checking figures 4.2 and 4.3, we realize that the performance for the third-party tools here compared do not improve as they do not support AVX-512 instructions, what results

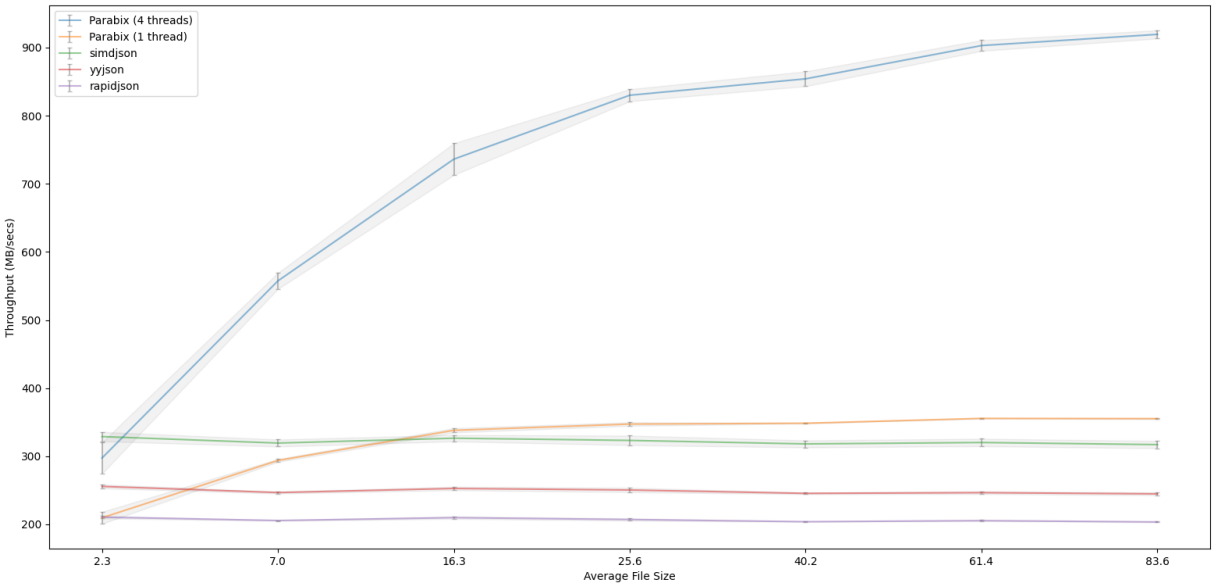

Figure 4.3: JSON parsing on AVX-512 as file size increases

in them using AVX-2 instructions for both cases. This happens because their code has been manually written and optimized with AVX-2 intrinsics, thus, it is non-trivial to add AVX-512 intrinsics to their codebase. The same is not true for Parabix as explained in §2.3, since as the block width increases, so does the capacity of processing more data at once in Parabix. Moreover, the final optimized code is done by the Parabix framework and LLVM, making it easier to add support to new SIMD architectures as needed. In other words, simdjson, RapidJSON and yyjson have no benefits from AVX-512, however, Parabix has almost a twenty-five percent speed improvement only by the fact that the block width increased from 256 bits on AVX-2 to 512 bits on AVX-512.

For large files on AVX-2, Parabix with 4 threads is 2.4 times faster than simdjson, 3.6 times faster than RapidJSON and 3 times faster than yyjson, and on AVX-512, it is 3 times faster than simdjson, 4.6 times faster than RapidJSON and 3.8 times faster than yyjson, which represents a trend in domination against its competitors.

In fact, Parabix is never worse than RapidJSON and yyjson when the average file size is greater than seven megabytes and it is only worse than simdjson when it is running on AVX-2 with one thread. The same is not true on AVX-512 because simdjson **always** uses AVX-2 intrinsics in both AVX-2 and AVX-512 cases. Had simdjson used AVX-512 intrinsics in its code, it could be that figure 4.3 would depict an improvement against Parabix on 1 thread, however, it is likely that Parabix with 4 threads would still dominate simdjson as

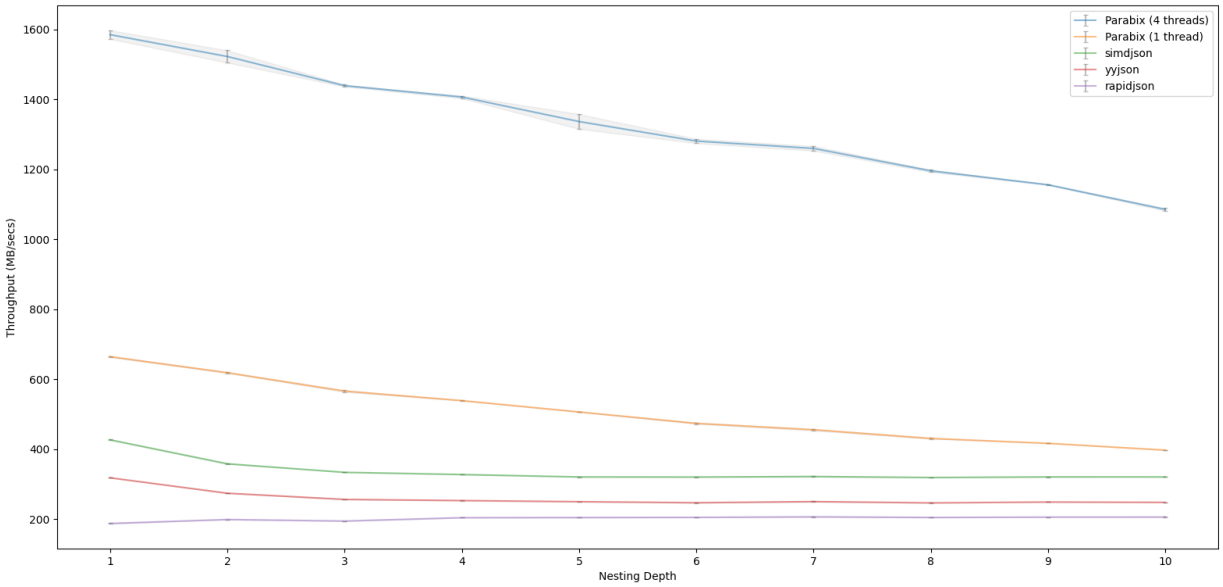

Figure 4.4: JSON parsing on AVX-512 as nesting depth increases

its parallel nature simply makes it able to process more data at once than its competitor, especially when depth d is not a very large number.

4.3 Performance as JSON nesting depth increases

This experiment focuses on how well the tools simdjson, yyjson and RapidJSON perform against our approach as the JSON file gets more complex in respect of its maximum depth. For this, we fix the randomly generated files to have a size of 75 ± 11.25 MB and define the constraint d of each one of these files to have a value in the inclusive range 1...10. Moreover, this experiment was only executed on the AVX-512 machine because we believe that running it in different architectures would not change how the parsers here compared behave, thus, only figure 4.4 is taken into consideration here. The details about these randomly generated files can be found in appendix D. Below is how we generated them:

- $3 \times$ `RandomJson(`$p = 3 \times 10^6,\ d = 1,\ s = 75$`)`
- $4 \times$ `RandomJson(`$p = 1.2^6,\ d = 2,\ s = 75$`)`
- $3 \times$ `RandomJson(`$p = 5.5 \times 10^5,\ d = 3,\ s = 75$`)`
- $3 \times$ `RandomJson(`$p = 2 \times 10^5,\ d = 4,\ s = 75$`)`

- $3 \times$ `RandomJson(`$p = 10^5,\ d = 5,\ s = 75$`)`
- $3 \times$ `RandomJson(`$p = 5 \times 10^4,\ d = 6,\ s = 75$`)`
- $3 \times$ `RandomJson(`$p = 2.5 \times 10^4,\ d = 7,\ s = 75$`)`
- $3 \times$ `RandomJson(`$p = 10^4,\ d = 8,\ s = 75$`)`
- $3 \times$ `RandomJson(`$p = 5 \times 10^3,\ d = 9,\ s = 75$`)`
- $3 \times$ `RandomJson(`$p = 10^3,\ d = 10,\ s = 75$`)`

Considering figure 4.4, we can easily identify that the third-party tools only have some variation between depths 1 and 2. This happens because their algorithm is of sequential nature and therefore, a JSON file with depth 1 can be parallelized using their approach by excluding first and last characters of the stream and checking that they are an open bracket and its corresponding closing bracket, respectively. After that, the remaining characters can be totally parallelized as discussed in §3.4. However, after depth 2, these third-party parsers plunge and reach throughputs that do not change as the depth increases; this is expected in algorithms that solve the bracket-matching problem sequentially.

On the other hand, the algorithm described in our approach in §3.6 has a direct impact on the depth of a JSON file representation. This occurs because that algorithm (1) checks whether a nesting depth exists and (2) only executes that nesting depth in parallel in case it exists. Clearly then, the higher the number of depths, the more data our algorithm needs to process and synchronize and the slower it becomes, however, this also means that it can leverage more than the sequential algorithms when we consider shallow structures and it is important to point out that most JSON datasets are shallow, for example, a list of objects is around no more than depth $d = 3$. Furthermore, JSON files do not usually exceed nesting depth $d = 15$, although there is no rule on how deep a JSON file can be.

Finally, when we compare our parallel approach with the other parsers here presented, we notice that Parabix can be up to 3.76 times faster than simdjson, 5 times faster than yyjson and 8.59 times faster than RapidJSON. Figure 4.4 shows that Parabix is faster than the other JSON parsers even when it only uses 1 thread and the nesting depth d is in inclusive range 1...10. For larger d, for example $d \geq 13$, we expect it to be at least as fast as simdjson when single-threaded, but still always faster than simdjson with four threads. Please note that we did not try $d > 15$ for any cases as this is not realistic for JSON datasets.

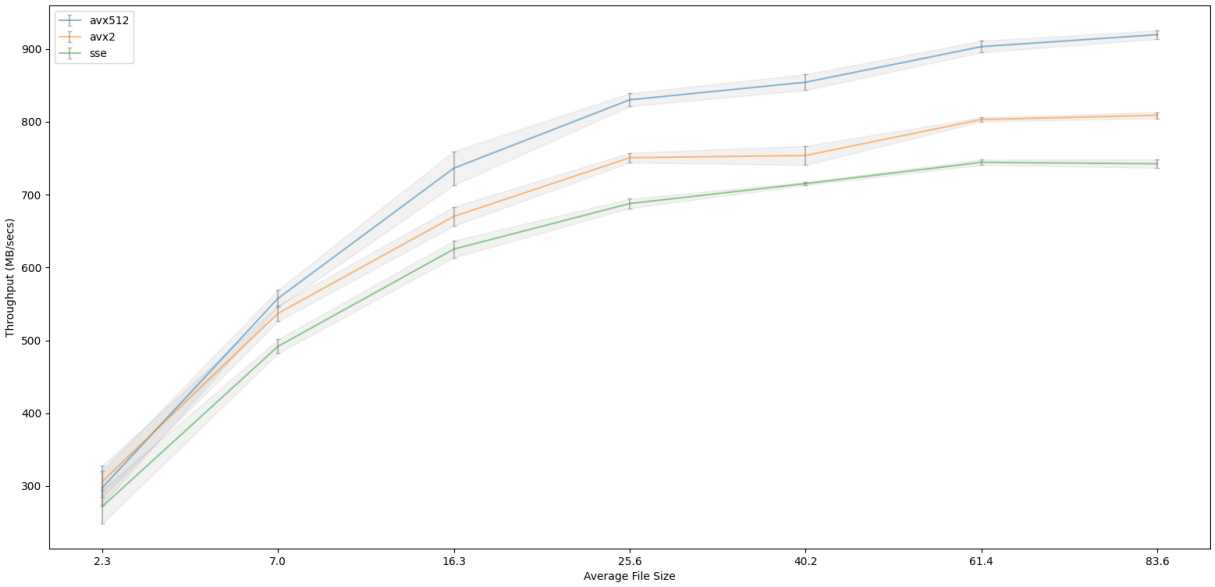

Figure 4.5: Parabix's performance on different architectures as file size increases

4.4 Parabix's performance

Previous experiments in §4.2 and §4.3 indicate that Parabix performs better than the top-performance parsers here presented. Nevertheless, this experiment focuses on how well Parabix scale on different architectures with different block widths.

That is to say, in this section we compare and contrast the values displayed in appendices E, F and G for figures 4.5, 4.6 and 4.7, respectively.

4.4.1 Parabix's performance on different architectures

Figure 4.5 illustrates how Parabix scales when we run our algorithm in different architectures using the randomly generated files described in §4.2. Our SSE4.2 machine has a block width of 128 bits and, by default, our AVX-512 machine has a block width of 512 bits, however in compile-time we can reduce its block width to 256 bits by passing compiler's flag `-BlockSize` that allows us to generate AVX-2 code.

Although two different machines were used in this evaluation, we notice that both SSE4.2 and AVX-512 machines have similar performance for files with sizes up to 8MB but as file size increases, the throughput of the SSE4.2 reaches a plateau 750MB/s against 819MB/s on AVX2 and 928MB/s on AVX-512. It is easy to notice that as we increase the block width, so does Parabix's throughput, but as shown in §4.4.3, the throughput also depends

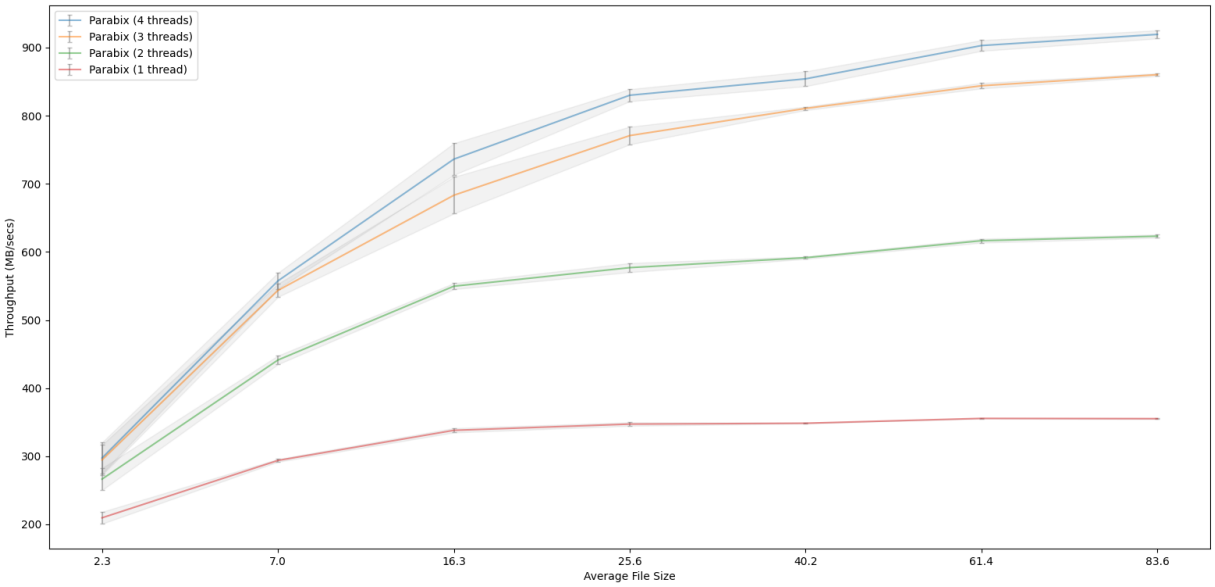

Figure 4.6: Parabix's performance on AVX-512 as file size increases

on the nesting depth of the JSON file. In fact, the files that were used in figure 4.5 have a maximum nesting depth $d = 13$, making the throughput slower than it should be in the average case because JSON files are usually not very deep, thus, considering figure 4.7, we could easily assume that in the average case, our throughput is around 1GB/s on SSE4.2, 1.2GB/s on AVX-2 and 1.4GB/s on AVX-512.

A major benefit displayed in figure 4.5 is that as computer architectures continuously improve and as their block width increase, so will Parabix's ability to process more and more data at once.

4.4.2 Parabix's performance on AVX-512 as JSON file size increases

This section discusses figure 4.6 and how the number of threads used in Parabix affects its overall performance as JSON file size increases – these files are described in §4.2. Here we are only interested in the results obtained from the AVX-512 machine because the throughput curve on different architectures is very similar as inferred from figure 4.5, thus, showing for one architecture should suffice.

It is to be noted that Parabix running on a single thread is usually at least as performant as the state-of-the-art parsers compared in this thesis (§4.2 and §4.3). However, things get more interesting as we increase the number of threads used by our algorithm, for example, when we consider very large files, running Parabix with one versus two threads gives a

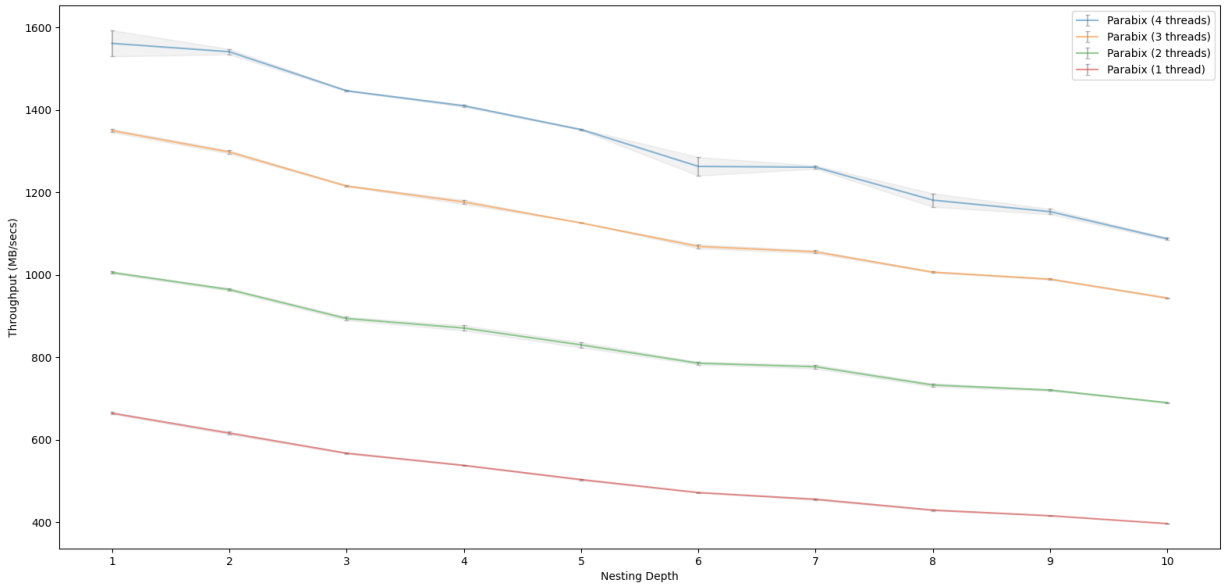

Figure 4.7: Parabix's performance on AVX-512 as nesting depth increases

huge difference in the performance, whereas we seem to have no much improvement when we go from three to four threads. That is, on AVX-512, Parabix on one thread only has a throughput of 357MB/s on a file with an average size of eighty-three megabytes and a maximum nesting depth of $d = 13$ against a throughput of 632MB/s on two threads, 867MB/s on three threads and 928MB/s on four threads. What that means is that our algorithm performs better as the number of threads increase, however, because of thread synchronization, only a little can be leveraged when the thread count gets high. There seems to exist a trend in Parabix's performance as the number of threads increase. It is possible to notice that when we run Parabix with 4 threads, we generally have an improvement of $2.5 \sim 3$ times its own throughput on a single thread.

4.4.3 Parabix's performance on AVX-512 as JSON nesting depth increases

As mentioned in §4.3, Parabix does much better than the state-of-the-art parsers compared in this thesis when we consider the nesting depth of a JSON structure. This happens because while our algorithm uses the parallel approach described in §3.6 for the bracket matching problem, all the other ones (simdjson, RapidJSON and yyjson) solve it sequentially, which gives them no gain in performance in shallow structures. As a matter of fact,

even Parabix on a single thread is better than the other parsers when the maximum nesting depth $d \leq 5$.

Figure 4.7 shows that, as expected, the higher the thread count, the faster Parabix can process JSON files. On AVX-512, while Parabix on one thread for nesting depth $d = 1$ has a throughput of around 665MB/s, it reaches over 1GB/s on two threads, 1.3GB/s on three threads and 1.6GB/s on four threads, that is, the throughput on four threads is nearly 3 times its own throughput on a single thread.

Although the results of this research are exceptional, we need to draw attention to the fact that Parabix's throughput decreases as the maximum nesting depth increases, for example, when $d = 10$, its performance reduces to around 400MB/s on a single thread and 1GB/s on four threads. This could be a problem for its performance on deeply nested JSON files, however, because we do not expect JSON datasets to have a very high nesting depth d, it is safe to say that, in the average case, Parabix with 4 threads (default) always have a performance greater than 1GB/s on AVX-512, where the shallower a JSON file is, the faster Parabix can process it.

4.4.4 Parabix's microarchitecture performance

While investigating the performance of Parabix for parsing JSON structures of different sizes, we found out that the throughput rates for smaller files were inferior to the ones for larger files, which is demonstrated in figure 4.6. Upon closer examination of the data given in appendix I, we identified that the rate of cache misses for smaller files could surpass 50%; moreover, the rate of branch mispredictions for these files could reach 3%. However, for larger files, the performance of Parabix is less affected, with significantly lower percentages of cache misses and branch mispredictions. For example, a 246 MB file size showed only 13% cache misses and 0.53% branch mispredictions, resulting in a high throughput of 960 MB/s. In contrast, a much smaller file with a size of 1.7 MB had a lower throughput of 240 MB/s. Overall, this performance trend happens over and over as displayed in figures 4.3 and 4.6 and suggests that the early stages of our solution are causing these cache misses and branch mispredictions, which can impact the overall performance of smaller files. That is to say, it may exist an initial overhead in our parser such that smaller files may experience these performance issues, while larger files are not greatly impacted. Further investigation and improvements should be considered in future works.

4.4.5 Parabix's performance ceiling

As discussed in previous sections, Parabix's parsing performance can vary depending on the depth of the document. Nonetheless, there exists a trend, such as shown in figure 4.3, that indicates that Parabix's performance reaches a ceiling as the file size increases. With this intuition in mind, we performed experiments on files of size between 200 MB and 1.6 GB (appendix H), which helped us to verify that in fact these large files all had a similar

throughput confirming then that a plateau had been reached. For example, on SSE4.2, Parabix's throughput ceiling for shallow files is 1.46 GB/s, while for deeply-nested files, it is 780 MB/s. On AVX-2, Parabix's ceiling for shallow files is 1.6 GB/s, and for deep files it is 840 MB/s. Finally, on AVX-512, Parabix reaches impressive throughput rates of around 1.75 GB/s for shallow files and 966 MB/s for deep files. In these experiments, we only needed to check the throughput of JSON files with a nesting depth of 1 and 13 because we know that every depth d_i has a worse performance than depth d_{i-1}, for $i > 1$, as shown in figure 4.7, thus, we know that the ceiling performance for depths $d_i, i \in [2..12]$ is somewhere between the performance of parsing very shallow versus very deep files.

4.5 Simdjson's performance

Simdjson is reported to have an excellent performance on Skylake (Intel i7-6700) [15], and when we ran its built-in evaluation scripts on a subset of the files used by Langdale and Lemire (2019), its performance on our AVX-512 machine was only around fifteen percent slower than what was reported and thirty-five percent slower on our SSE4.2 machine (table 4.1). The reason why we achieved closer results on AVX-512 is because that machine has a similar hardware configuration to Skylake's, making us believe that the use of different machines did not have a negative impact on the evaluation described in previous sections.

Although we would need to run our solution in a machine with the same hardware configuration as Skylake to prove that Parabix is in fact faster than simdjson as the JSON complexity increases, we may still infer that this is true given the trend that can be identified in figures 4.1, 4.2, 4.3 and 4.4. Moreover, we ran the scripts for performance evaluation that were provided in their benchmarks folder against larger files and we had similar results to the ones shown in appendices A, B, C and D.

Still in table 4.1, we see that the rationale behind not including small files in the evaluation of this thesis is simply because that is not the main focus of Parabix. Parabix is meant to solve Big Data problems, for instance, while simdjson parsed a 1.7 MB file at a rate of 2.26 GB/s on AVX-512, Parabix parsed the same file at a rate of 280 MB/s, which is over eight times slower. In spite of that, as explained in previous sections, Parabix scales well as the file size and complexity increase and simdjson does not do very well in that matter, for example, if we compare the results for larger files in appendix B, we will see that simdjson only reached a maximum rate of 353 MB/s.

This difference in rates raised a few questions about simdjson and once we dove into the implementation of the benchmark files for simdjson, we discovered that the high performance reported, which is over 2 GB/s in some cases, is due to the fact that they leave memory/page allocation and OS-related tasks out during their evaluation. Nevertheless, the evaluation described in §4.2 and §4.3 does not ignore them; it only ignores the time taken for loading a file. That being said, to make sure the comparison among Parabix, simdjson and the other

tools was as fair as possible, we included memory/page allocation and OS-related tasks as part of our evaluation. That allowed us to focus on the real performance of simdjson and the other tools.

In conclusion, simdjson should only be chosen over Parabix when the file size is very small but since it does not scale very well, Parabix is still the best choice for most cases.

Table 4.1: Simdjson's performance vs Parabix's performance on small files

filename	filesize MB	simdjson Skylake [15] MB/s	simdjson SSE4.2 MB/s	simdjson AVX-512 MB/s	Parabix SSE4.2 MB/s	Parabix AVX-512 MB/S
apache_builds	0.125	2200	1521	2195	22	26
citm_catalog	1.7	2500	1882	2261	255	280
gsoc-2018	3.2	3000	2110	2561	427	476
instruments	0.216	2000	1478	1777	38	44
twitter	0.617	2200	1599	1993	101	118

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This thesis shows that there is still a lot that can be improved regarding the efficiency of JSON processing tools in commodity processors. Although some tools such as simdjson, yyjson and RapidJSON do an excellent job in parsing JSON files, we present here a new approach that is often better than these top-notch tools that are known for their formidable performance and have been largely used in the tech industry.

With the use of parallel bit streams, our approach can frequently be three times faster than the current state-of-the-art when JSON files are large enough. That is to say, our results are outstanding and should inspire new research around validating and parsing input for context-free grammars, since the major contribution in this research is solving the bracket matching problem in parallel (§3.6), which should not be taken as an easy task because multithreading and synchronization tend to be challenging when implemented as a mean to solve problems of sequential nature.

Despite our promising results, much still can be done around this solution alongside Parabix. Further sections outline what kind of applications could benefit from this research as well as what we would hope to accomplish in future works.

5.2 Applications

Below are a few examples of how related research could benefit from the concepts studied in this thesis:

- As mentioned in §2.2, JSON is one the most popular ways to interchange data through the Web, and, the more technology advances, the more and more data is transferred at once. Therefore, algorithms for encoding/decoding to/from JSON need to start taking advantage of parallel processing.

- The field of Big Data benefits from it especially because many datasets are stored in either JSON or CSV because of their simple and yet effective grammar, for example public government datasets and etc.
- NoSQL databases are non-relational databases that do not require a fixed schema and that are stored in JSON format. Because their data schema is flexible and is used mostly in scenarios where the data is semi-structured and unstructured, many times indexing does not work well or is not a viable option. This type of database could benefit from this research, that is, we could parallelize its parsing with rates of over 1 GB/s — querying JSON files is mentioned as a future research in §5.3.
- Converting JSON to other text formats, such as CSV or XML, could be even faster.
- Many CFG algorithms could benefit from solving the bracket-matching problem in parallel such as what is described in §3.6.
- simdjson's solution is very similar to ours. It would be beneficial to them to consider starting a research on multithreading (similar to what we do in this thesis).

5.3 Future Work

This work is to be considered only as a starting point for a future research since it is far from being done. For reference, below are a few suggestions for future work that could improve this project even further:

- This solution only checks if a JSON file is valid, thus, further work on this parser should be considered, for example adding support for querying and conversion to other formats, such as CSV, YAML, etc.
- At the moment our solution is part of the tools folder in the Parabix project. In the future, we could make it a standalone library that could be used by other programs.
- In our algorithm, we could research a better way to work in multiple nesting depths in parallel in order to improve its overall performance since at the moment our algorithm depends a lot on the maximum depth of a JSON structure, which is clearly our biggest performance bottleneck.
- We could research the other parsing tools that were mentioned in this project and implement AVX-512 instructions on them in order to have a better idea on how they scale. This would allow us to have a better performance overview of this solution.
- Currently in Parabix, any algorithms that are based in CFGs need to be implemented manually, so a Parabix kernel that takes a grammar G and generates a parser for it would be a great contribution.

Bibliography

[1] Robert D. Cameron. *A case study in SIMD text processing with parallel bit streams: UTF-8 to UTF-16 transcoding.* PPOPP 2008: 91-98, 2008.

[2] Eric Eilebrecht , Badrish Chandramouli, Donald Kossmann Chang Ge, Yinan Li. *Speculative Distributed CSV Data Parsing for Big Data Analytics.* 2019 ACM SIGMOD Conference, 2019.

[3] Noam Chomsky. *Three models for the description of language.* RE Transactions on Information Theory. 2 (3): 113-124. doi:10.1109/TIT.1956.1056813, 1956.

[4] Noam Chomsky. *On Certain Formal Properties of Grammars.* Information and Control Volume 2, Issue 2, Pages 137-167, 1959.

[5] Robert D. Cameron, Nigel Medforth, Dan Lin, Dale Denis and William N. Sumner. *Bitwise Data Parallelism with LLVM: The ICgrep Case Study.* ICA3PP (2) 2015: 373-387, 2015.

[6] Hardeep Kaur Dhalla. *A Performance Analysis of Native JSON Parsers in Java, Python, MS.NET Core, JavaScript, and PHP.* 16th International Conference on Network and Service Management (CNSM), pp. 1-5, 2020, 2020.

[7] Crockford, Douglas and Morningstar, Chip. *Standard ECMA-404 The JSON Data Interchange Syntax.* 10.13140/RG.2.2.28181.14560, 2017.

[8] Michael J. Flynn. *Some Computer Organizations and Their Effectiveness.* IEEE Transactions on Computers. C-21 (9): 948-960. doi:10.1109/TC.1972.5009071, 1972.

[9] Robert D. Cameron, Kenneth S. Herdy and Dan Lin. *High performance XML parsing using parallel bit stream technology.* CASCON 2008: 17, 2008.

[10] John E. Hopcroft and Jeffrey D. Ullman. *Nonerasing Stack Automata.* Journal of Computer and System Sciences. 1 (2): 166-186. doi:10.1016/s0022-0000(67)80013-8, 1967.

[11] Robert D. Cameron, Thomas C. Shermer, Arrvindh Shriraman, Kenneth S. Herdy, Dan Lin, Benjamin R. Hull and Meng Lin. *Bitwise data parallelism in regular expression matching.* PACT 2014: 139-150, 2014.

[12] S. C. Kleene. *Representation of events in nerve nets and finite automata.* Automata Studies, Princeton University Press, NJ, 3-41, 1951.

[13] Kamil Kolonko. *Performance comparison of the most popular relational and non-relational database management systems.* SE-371 79 Karlskrona, 2018.

[14] P.S. Landweber. *Three Theorems on Phrase Structure Grammars of Type 1.* nformation and Control. 6 (2): 131-136 doi:10.1016/s0019-9958(63)90169-4, 1963.

[15] Geoff Langdale and Daniel Lemire. *Parsing Gigabytes of JSON per Second.* The VLDB Journal, 28(6), 2019. 1902.08318, 2019.

[16] Daniel Lemire and Lucile Sautot. *Next Generation Indexes For Big Data Engineering.* RNTI B-15 , ISBN 979-10-96289-11, 2018.

[17] Dan Lin. *Multidimensional Parallelization for Streaming Text Processing Applications Based on Parabix Framework.* PhD thesis, Simon Frasier University, 2017.

[18] Nigel W. Medforth. *Literature Review on Parabix and Dataflow Systems.* PhD thesis, Simon Frasier University, 2022.

[19] John Myhill. *Linear Bounded Automata.* Wright Patterson AFB, Wright Air Development Division, 1960.

[20] G. Navarro. *Nr-grep: A fast and flexible pattern matching tool.* Software Practice and Experience SPE, 31:2001, 2000, 2000.

[21] Bin Ren, Gagan Agrawal, James R. Larus, Todd Mytkowicz, Tomi Poutanen and Wolfram Schulte. *SIMD Parallelization of Applications that Traverse Irregular Data Structures.* CGO '13. CGO.2013.6494989, 2013.

[22] G.K. Pullum and G. Gazdar. *Natural languages and context-free languages.* Linguist Philos 4, 471-504 (1982). https://doi.org/10.1007/BF00360802, 1982.

[23] Orestis Polychroniou, Arun Raghavan and Kenneth A. Ross. *Rethinking SIMD Vectorization for In-Memory Databases.* SIGMOD '15. 2723372.2747645, 2015.

[24] S. Seshu. *Introduction to the Theory of Finite-state Machines.* Proceedings of the IEEE, volume 51, number 9, pages: 1275-1275, doi:10.1109/PROC.1963.2548, 1967.

[25] Dan Lin, Nigel Medforth, Kenneth S. Herdy, Arrvindh Shriraman and Robert D. Cameron. *Parabix: Boosting the efficiency of text processing on commodity processors.* HPCA 2012: 373-384, 2012.

[26] K. Thompson. *Programming techniques: Regular expression search algorithm.* Communications of the ACM, 11(6):419-422, 1968, 1968.

[27] A. M. TUKING. *On Computable Numbers, with an Application to the Entschidungsproblem.* Proceedings of the London Mathematical Society, 2:242, p230-256, 1936.

[28] AhoA. V. and J. D.Ullmann. *The Theory of Parsing, Translation and Compiling.* Volume II: Translation and Compiling (Prentice-Hall, Englewood Cliffs, New Jersey), 1973.

[29] Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, Donald Kossmann Yinan Li. *Mison: A Fast JSON Parser for Data Analytics*. Proc VLDB Endow 10(10):1118-1129, DOI 10.14778/3115404.3115416, 2017.

[30] Jingren Zhou and Kenneth A. Ross. *Implementing database operations using SIMD instructions.* SIGMOD '02. Pages 145-156. 564691.564709, 2002.

Appendix A

Comparison on SSE4.2 - Properties

#properties	filesize MB	Parabix 1T MB/s	Parabix 4T MB/s	simdjson MB/s	rapidjson MB/s	yyjson MB/s
5	1.7	164.01	252.68	177.81	158.17	193.25
5	1.8	163.64	258.36	424.93	262.39	183.37
5	2.4	182.43	320.73	248.29	337.51	154.36
5	2.7	184.49	339.88	411.40	326.64	338.60
5	3	189.11	358.51	187.06	325.24	340.68
10	7.51	224.09	558.45	326.24	324.10	265.04
10	7.5	223.45	567.62	431.88	328.37	345.00
10	7.7	223.36	511.76	408.57	326.67	338.66
10	6.2	216.27	522.77	217.82	322.58	227.30
10	6.3	217.93	523.95	305.85	323.43	259.24
25	15	208.76	678.06	327.53	330.62	315.29
25	16.1	244.74	617.90	427.35	339.05	349.38
25	16.2	246.19	684.99	425.76	341.00	317.80
25	16	238.31	683.21	442.00	333.42	351.05
25	18	216.00	686.03	345.69	324.59	307.05
50	23	247.21	745.52	324.35	333.54	279.02
50	24	241.07	732.27	336.00	323.23	274.29
50	26	242.44	745.93	407.24	322.73	338.00
50	27	239.16	772.58	381.67	332.10	268.91
50	28	230.55	756.53	453.93	339.26	305.22
100	38	230.62	704.43	411.02	325.25	275.93
100	39.1	244.73	763.88	409.07	323.97	322.35
100	39	244.59	780.03	439.22	326.31	346.48
100	42	235.99	767.56	378.07	322.24	336.41
100	43	245.78	752.34	348.36	324.82	338.18
150	47	249.28	794.19	391.90	328.38	296.20
150	58	248.65	809.33	443.35	318.82	354.09
150	60	247.49	803.38	412.46	324.68	321.85
150	70	248.37	799.81	394.74	311.17	339.75
150	72	249.61	808.00	398.47	326.13	323.94
200	75.1	248.95	819.26	412.94	323.60	320.69
200	75	247.24	800.23	387.69	324.44	328.42
200	85	240.72	810.06	426.04	327.30	313.87
200	87	247.15	816.83	393.26	322.77	314.71
200	96	247.32	797.65	408.74	322.50	336.72

Appendix B

Comparison on AVX-2 - Properties

#properties	filesize MB	Parabix 1T MB/s	Parabix 4T MB/s	simdjson MB/s	rapidjson MB/s	yyjson MB/s
5	1.7	167.24	206.56	324.61	216.20	258.20
5	1.8	166.84	226.84	329.55	212.82	248.69
5	2.4	193.31	290.00	352.79	215.40	267.26
5	2.7	195.71	310.17	322.85	207.09	250.44
5	3	203.57	324.32	314.99	208.41	248.30
10	7.51	251.23	510.26	312.33	206.35	248.58
10	7.5	250.20	503.49	342.42	207.01	258.01
10	7.7	249.31	508.18	314.11	205.73	248.36
10	6.2	240.20	467.29	314.16	205.65	246.98
10	6.3	241.18	469.03	315.68	205.51	246.55
25	15	273.98	626.44	320.47	208.20	251.69
25	16.1	279.17	644.26	329.98	213.44	255.53
25	16.2	282.95	647.79	330.44	214.99	258.22
25	16	270.46	624.71	345.83	206.50	255.29
25	18	271.50	582.68	316.37	206.35	245.85
50	23	283.44	685.42	326.58	210.47	254.48
50	24	276.60	668.15	315.94	204.84	244.20
50	26	277.89	680.91	315.97	204.18	247.47
50	27	286.92	699.57	325.31	210.25	251.48
50	28	287.01	704.24	354.62	214.65	265.40
100	38	283.21	717.09	319.66	206.01	249.03
100	39.1	283.09	715.66	318.13	204.43	248.01
100	39	282.75	708.02	345.66	207.32	257.15
100	42	280.69	713.05	318.46	204.28	245.59
100	43	281.75	721.34	319.96	206.38	248.52
150	47	288.76	737.23	322.18	208.30	251.73
150	58	288.51	742.25	350.42	208.81	260.63
150	60	285.42	737.13	319.47	205.27	248.95
150	70	287.80	747.93	319.20	204.52	250.01
150	72	289.93	756.56	320.67	206.53	250.67
200	75.1	290.43	754.49	319.59	204.58	251.07
200	75	286.54	728.64	318.40	205.53	247.94
200	85	286.45	750.37	345.54	206.78	255.60
200	87	283.56	728.05	317.29	204.66	244.97
200	96	284.90	749.87	316.41	204.26	245.91

Appendix C

Comparison on AVX-512 - Properties

#properties	filesize MB	Parabix 1T MB/s	Parabix 4T MB/s	simdjson MB/s	rapidjson MB/s	yyjson MB/s
5	1.7	188.41	240.66	333.86	211.29	253.35
5	1.8	189.51	248.04	322.06	210.38	257.88
5	2.4	216.49	311.24	353.67	217.39	265.93
5	2.7	222.37	330.11	318.47	206.34	251.72
5	3	231.27	357.06	316.19	207.51	249.75
10	7.51	295.90	577.60	316.32	205.56	245.27
10	7.5	297.93	575.73	340.03	207.44	252.67
10	7.7	297.23	575.87	314.30	206.23	245.11
10	6.2	287.49	521.62	313.43	204.34	245.78
10	6.3	290.54	536.54	312.75	204.64	244.62
25	15	332.42	649.72	319.02	208.51	247.52
25	16.1	343.88	755.12	327.12	213.17	256.91
25	16.2	347.48	779.56	329.64	214.40	257.74
25	16	331.63	727.67	342.14	208.86	256.08
25	18	334.83	768.80	314.63	204.29	245.27
50	23	347.40	825.59	322.05	209.52	251.84
50	24	339.00	803.56	311.91	203.58	243.90
50	26	344.76	823.57	312.11	203.21	244.17
50	27	351.05	841.04	320.03	207.38	250.65
50	28	353.91	856.40	350.12	211.91	261.72
100	38	346.72	863.52	314.13	203.98	244.02
100	39.1	349.64	870.40	312.40	203.22	244.92
100	39	349.17	814.38	338.73	204.81	250.36
100	42	348.43	848.19	312.02	202.95	243.64
100	43	348.49	874.11	313.31	204.55	244.59
150	47	354.96	891.60	316.37	206.54	245.61
150	58	356.65	906.89	342.16	208.13	254.22
150	60	355.21	878.95	313.53	204.16	245.28
150	70	354.52	915.32	314.08	203.66	242.43
150	72	356.58	922.15	314.72	204.98	245.08
200	75.1	356.79	922.45	313.73	203.53	242.84
200	75	354.21	920.86	312.18	203.55	244.54
200	85	357.40	928.54	338.49	205.69	252.73
200	87	352.99	896.16	310.10	202.36	242.80
200	96	354.53	928.94	310.87	202.54	240.93

Appendix D

Comparison on AVX-512 - Depth

depth	filesize MB	Parabix 1T MB/s	Parabix 4T MB/s	simdjson MB/s	rapidjson MB/s	yyjson MB/s
1	66	661.02	1564.02	424.71	185.43	317.42
1	77.1	666.13	1585.51	427.96	188.11	318.37
1	77	665.68	1607.18	426.65	187.58	317.71
2	82.1	616.36	1470.64	355.38	197.75	272.76
2	82.2	621.18	1535.99	359.52	198.49	273.58
2	82.3	620.37	1546.00	359.83	199.17	275.15
2	82	615.88	1539.44	356.61	198.13	273.09
3	70	567.39	1438.20	333.38	194.65	256.99
3	70.1	568.96	1444.68	333.21	193.87	254.09
3	70.2	560.86	1435.76	333.54	193.51	256.82
4	73.1	537.78	1400.30	326.17	203.35	251.98
4	73.2	539.92	1413.29	328.48	203.93	253.22
4	73	538.10	1407.58	326.78	203.72	252.70
5	78.1	506.16	1294.48	319.41	204.65	249.63
5	78.2	507.22	1356.30	321.33	204.75	249.95
5	78	505.08	1360.00	319.94	203.05	248.74
6	72.1	470.49	1274.62	319.13	203.82	245.23
6	72.2	473.16	1275.26	319.25	204.38	246.11
6	73	476.71	1293.29	321.58	205.49	247.99
7	81.1	453.32	1255.05	320.30	205.62	249.08
7	81.2	460.04	1273.11	324.53	207.68	251.85
7	81	452.07	1251.76	319.52	204.99	248.08
8	77	432.50	1200.24	319.71	205.46	246.27
8	78	428.06	1189.04	317.30	203.14	245.65
8	79	430.42	1198.04	318.68	204.44	246.15
9	79	416.86	1154.50	319.30	205.05	248.42
9	82	415.78	1155.42	319.72	205.02	248.14
9	83	416.57	1158.13	321.99	205.65	248.75
10	80	397.30	1092.75	320.72	205.77	247.43
10	78	397.00	1082.27	319.59	204.78	247.10
10	74	396.56	1080.84	320.65	205.76	247.67

Appendix E

Parabix - Architecture comparison

#properties	filesize MB	SSE MB/s	AVX-2 MB/s	AVX-512 MB/s
5	1.7	206.56	252.68	240.66
5	1.8	226.84	258.36	248.04
5	2.4	290.00	320.73	311.24
5	2.7	310.17	339.88	330.11
5	3	324.32	358.51	357.06
10	6.2	467.29	522.77	521.62
10	6.3	469.03	523.95	536.54
10	7.51	510.26	558.45	577.60
10	7.5	503.49	567.62	575.73
10	7.7	508.18	511.76	575.87
25	15	626.44	678.06	649.72
25	16.1	644.26	617.90	755.12
25	16.2	647.79	684.99	779.56
25	16	624.71	683.21	727.67
25	18	582.68	686.03	768.80
50	23	685.42	745.52	825.59
50	24	668.15	732.27	803.56
50	26	680.91	745.93	823.57
50	27	699.57	772.58	841.04
50	28	704.24	756.53	856.40
100	38	717.09	704.43	863.52
100	39.1	715.66	763.88	870.40
100	39	708.02	780.03	814.38
100	42	713.05	767.56	848.19
100	43	721.34	752.34	874.11
150	47	737.23	794.19	891.60
150	58	742.25	809.33	906.89
150	60	737.13	803.38	878.95
150	70	747.93	799.81	915.32
150	72	756.56	808.00	922.15
200	75.1	754.49	819.26	922.45
200	75	728.64	800.23	920.86
200	85	750.37	810.06	928.54
200	87	728.05	816.83	896.16
200	96	749.87	797.65	928.94

Appendix F

Parabix - Comparison on AVX-512 - Properties

#properties	filesize MB	Parabix 1T MB/s	Parabix 2T MB/s	Parabix 3T MB/s	Parabix 4T MB/s
5	1.7	188.41	226.27	237.70	240.66
5	1.8	189.51	232.65	249.48	248.04
5	2.4	216.49	271.65	307.97	311.24
5	2.7	222.37	294.28	326.72	330.11
5	3	231.27	307.19	351.33	357.06
10	6.2	287.49	421.91	520.27	521.62
10	6.3	290.54	430.62	519.89	536.54
10	7.51	295.90	450.65	557.04	577.60
10	7.5	297.93	454.79	556.38	575.73
10	7.7	297.23	448.38	563.69	575.87
25	25	332.42	539.24	635.97	649.72
25	16.1	343.88	558.33	728.08	755.12
25	16.2	347.48	561.00	741.69	779.56
25	16	331.63	540.76	706.53	727.67
25	18	334.83	548.86	603.72	768.80
50	23	347.40	573.35	731.74	825.59
50	24	339.00	556.75	751.60	803.56
50	26	344.76	572.37	776.28	823.57
50	27	351.05	585.57	792.09	841.04
50	28	353.91	596.42	802.27	856.40
100	38	346.72	586.68	809.68	863.52
100	39.1	349.64	592.13	817.17	870.40
100	39	349.17	597.33	804.95	814.38
100	42	348.43	592.55	805.79	848.19
100	43	348.49	588.78	815.86	874.11
150	47	354.96	610.25	828.34	891.60
150	58	356.65	615.67	849.64	906.89
150	60	355.21	618.44	848.86	878.95
150	70	354.52	611.26	843.91	915.32
150	72	356.58	626.56	849.14	922.15
200	75.1	356.79	618.80	860.28	922.45
200	75	354.21	619.25	855.83	920.86
200	85	357.40	632.35	867.54	928.54
200	87	352.99	623.50	857.58	896.16
200	96	354.53	621.73	859.86	928.94

Appendix G

Parabix - Comparison on AVX-512 - Depth

depth	filesize MB	Parabix 1T MB/s	Parabix 2T MB/s	Parabix 3T MB/s	Parabix 3T MB/s
1	66	661.02	1011.20	1338.31	1564.02
1	77	665.68	1018.86	1354.08	1607.18
1	77.1	666.13	1014.13	1362.89	1585.51
2	82.1	616.36	938.17	1298.58	1470.64
2	82.2	621.18	963.99	1305.03	1535.99
2	82.3	620.37	983.90	1306.95	1546.00
2	82	615.88	974.51	1296.28	1539.44
3	70	567.39	922.13	1214.67	1438.20
3	70.1	568.96	899.94	1214.40	1444.68
3	70.2	560.86	912.70	1204.20	1435.76
4	73.1	537.78	873.38	1170.95	1400.30
4	73.2	539.92	863.23	1173.51	1413.29
4	73	538.10	850.68	1170.28	1407.58
5	78.1	506.16	844.49	1132.06	1294.48
5	78.2	507.22	827.36	1138.30	1356.30
5	78	505.08	825.01	1120.51	1360.00
6	72.1	470.49	773.61	1065.70	1274.62
6	72.2	473.16	778.45	1063.50	1275.26
6	73	476.71	789.27	1078.30	1293.29
7	81.1	453.32	765.08	1051.96	1255.05
7	81.2	460.04	788.95	1065.10	1273.11
7	81	452.07	775.04	1048.45	1251.76
8	77	432.50	732.36	1012.27	1200.24
8	78	428.06	729.93	1000.59	1189.04
8	79	430.42	738.88	1012.26	1198.04
9	79	416.86	722.81	985.62	1154.50
9	82	415.78	718.71	987.25	1155.42
9	83	416.57	724.84	988.42	1158.13
10	74	396.56	684.36	943.76	1080.84
10	80	397.30	690.55	947.43	1092.75
10	78	397.00	681.12	942.75	1082.27

Appendix H

Parabix - Performance ceiling

filesize MB	depth	SSE MB/s	AVX-2 MB/s	AVX-512 MB/s
218	1	1457.5992	1663.7030	1753.5110
435	1	1459.3936	1672.4625	1766.3683
870	1	1477.1268	1669.1319	1754.6761
246	13	771.8833	827.2160	960.1911
756	13	773.1310	835.5298	960.1267
1200	13	799.7787	850.6319	968.8812
1600	13	787.9355	851.4632	975.6471

Appendix I

Parabix - Microarchitecture performance

filesize MB	# cache references	% cache misses	# branches	% branch misses	AVX-512 MB/s
1.7	915,530	51.087	24,673,562	2.98	240.66
3	1,125,234	43.934	26,997,273	2.77	357.06
6.2	1,561,654	35.161	34,671,912	2.24	521.62
18	3,233,982	23.779	60,234,555	1.47	768.80
28	4,575,608	20.351	79,825,800	1.21	856.40
42	6,768,514	18.094	111,639,786	0.99	848.19
60	9,272,712	16.422	154,626,018	0.82	878.95
87	13,437,823	14.994	212,065,519	0.72	896.16
246	36,899,367	13.062	558,646,062	0.53	960.19
756	111,310,619	12.247	1,810,997,879	0.42	960.12
1200	170,510,176	12.165	2,666,138,896	0.43	968.88
1600	232,565,492	12.232	3,691,515,010	0.41	975.64