Survey:Parsing and Parallelization

Greg Sandstrom

November 2, 2004

1 Introduction

My Senior Seminar project will explore the parallelization of parsing algorithms. This paper generally describes the processes involved in both parsing and parallelization, and provides brief introduction to several previous works in this field.

2 Parsing

Parsing is a process of building structure onto a sentence or other string of characters so that the meaning of the sentence or string can be derived.

Consider a simple English sentence: ”the boy hit the ball.” Thinking back to elementary school English class, the sentence can be broken into parts of speech as per Figure 1: an article followed by a noun, then a verb, then an article, then a noun. This string of parts of speech can then be grouped into phrases; the article noun pairs grouping to form noun phrases. Additionally the verb and the second noun phrase can be combined to form a verb phrase. Building farther, a sentence is formed from a noun phrase (the subject), and the verb phrase. Then and only then is the string of words recognized as a complete and valid English sentence. Figure 1 displays these steps in a graphical form known as a parse tree.

Approaching the same sentence from the other direction, the parse can be started with the desired result, a sentence. A simple valid sentence is made of two parts, a noun phrase (the subject) and a verb phrase. a verb phrase is made up in tern of a verb, and another noun phrase (the predicate). Each simple noun phrase can be further broken down into an article and a noun. Finally, the parts of speech are rewritten with the actual words of the sentence and the parse is completed.

The set of rules that define what structures can be derived is called a grammar. These grammars can take many forms, depending on how they are formalized, and how much information is need to be stored in the structure. The type of grammars

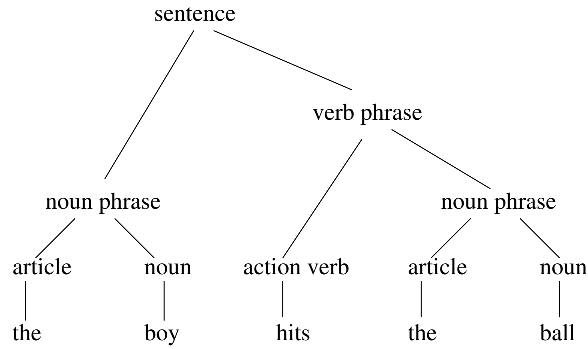

Figure 1: Parse tree for simple English sentence.

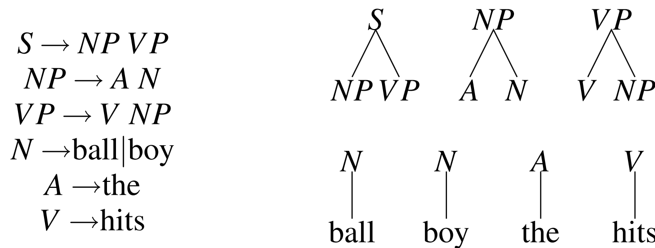

Figure 2: Simple English grammar as both rewriting rules and local trees.

that I will be investigating are known as Context Free Grammars (CFGs). Recalling back to the simple English example sentence from above, Figure 2 shows one simple CFG that can license it. In the figure, the grammar is represented in two ways. A more typical flat form, and a less common local tree form. The local tree form, while not as common, is useful in that it helps to visualize the possible structures within a graphical parse tree.

2.1 Context Free Grammars

Context Free Grammars are formalized as a four-tuple, containing an alphabet of non-terminals, an alphabet of terminals, a set of production rules, and a start state. In our initial example, the terminals are the actual words of the sentence, and the non-terminals are the parts of the sentence, i.e. Article, Noun, Verb, Verb Phrase, Noun Phrase, and Sentence. In our example, the start state is the non-terminal Sentence.

Context Free Grammars have several features that make them useful. First of all they are minimal in what is stored within the structure. In tree form, the structure stored is simply the rules of the grammar that have been applied. A second feature

is that there are restrictions on the context in which a grammar rule can be applied. As a consequence of this behavior, in tree form, any two subtrees with the same root can be swapped, to produce another parse tree recognized by the grammar. Using the simple English sentence from before as an example, because ’ball’ and ’boy’ are both nouns, wherever ’boy’ can be hung in the tree, ’ball’ can also be hung, and they both produce valid sentences.

2.2 Parsing Algorithms

Parsing algorithms are typically grouped according to which direction the parse tree is built. the most common types are top-down and bottom-up though there exist parsing algorithms that are neither and some that are a combination of the two.

The method used in the example would be one of bottom-up, where the terminals were mapped to non-terminals and then the non-terminals were combined according to the grammatical rules eventually yielding a single non-terminal root. An example of this occurs in Figure 1, where a parse tree is generated starting at the terminals of the simple string, and ending with the single non-terminal ”Sentence”.

The strength of Bottom-up parsers lie in their data driven nature, in that the parser only stores structures that derive a substring of the string instance. One consequense of this nature is that with each application of the grammar rules, the string is distilled down to a smaller size. One negative consequence of bottom-up nature is that parses may be generated that cover the whole of the tree, but are still not licensed by the grammar because the ultimate root node is not the start symbol.

Top-down parsing is the reverse of the bottom-up process, as given in the second example, where the parse starts at a single start non-terminal, and applies the grammatical rules to build the structure of the sentence and after applying the terminal rules, yields a string of terminals. This process can also be though of as string generation. Another example of a string generation can be seen in Figure 3.

The strength of top-down parsers lie in their root driven nature, maintaining only structures that are derivable from the start symbol. One consequence of this nature is that without any guidence the algorithm will generate every possible string generated by the language, not just the instance in question. This however is very costly both in terms of time and space.

2.3 CKY Algorithm

The Cocke-Kasami-Younger (CKY) algorithm is a chart based parsing strategy for parsing context free grammars introduced in 1963[BP94]. CKY is a purely bottom up algorithm, where each entry into the table is based on previous entries.

S	$(S \to NP\ VP)$
$NP\ VP$	$(NP \to A\ N)$
$A\ N\ VP$	($A \to$'the')
'the' $N\ VP$	$(VP \to V\ NP)$
'the' $N\ VNP$	($V \to$'hits')
'the' N 'hits' NP	$(NP \to A\ N)$
'the' N 'hits' $A\ N$	($A \to$'the')
'the' N 'hits' 'the' N	($N \to$'boy')
'the' N 'hits' 'the' 'boy'	($N \to$'ball')
'the' 'ball' 'hits' 'the' 'boy'	

Figure 3: Top-down parsing as string generation.

CKY's main advantage is that it is easy to conceptualize and has complexity of $O(n^3)$, (where n is the number of elements in the string being parsed). CKY has a disadvantage in that in its base state, it requires the grammar to be in strictly 2-branching form, also known as Chomsky Normal Form (CNF). All CFGs can be represented in CNF, but the conversion is typically costly. Ultimately because of it's simplicity CKY has been extended from the CFGs it was designed to parse to parsing many different types of grammars such as Tree Adjoining Grammars (TAG) and others.

The CKY algorithm operates under the CFG property that if there exists A, B, and C such that $B \to w_i \dots w_{k-1}$ and $C \to w_k \dots w_j$ are true for some $i \leq k \leq j$ and $A \to B\ C$ exists in the grammar, then $A \to w_i \dots w_j$ is also true.

The CKY algorithm starts out listing the productions that yield every substring of length 1. The algorithm then gradually increases the length of the substring being parsed, building upon previously recorded entries in the table. If a production is legal, that production is recorded in the table. Eventually the substring being parsed is actually the whole of the string, at which point the algorithm is complete, and the table is the parse forest, representing all possible parse sub-trees.

Shieber, Schabes, and Pereira approach CKY as a set of axioms and inference rules. The algorithm stores items of form $[A,i,j]$ where A is a non-terminal and A derives a the substring from position i to $j-1$ of the string being operated on. Axioms use the terminal derivation rules of the grammar to build items of length 1. Inference rules apply the non-terminal derivation rules of the grammar, taking two items $[A,i,j]$ and $[B,j,k]$ and producing $[C,i,k]$ if $C \to A\ B$ is a valid rule. The algorithm completes when $[S,0,n]$ is computed. This process is shown in Figure 4 [SSP95]

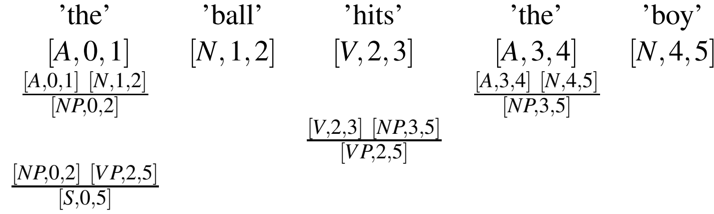

Figure 4: CKY as applied inference rules

2.4 Earley's Algorithm

A second parsing algorithm is Earley's algorithm, which is another chart parser introduced in 1970[BP94]. Depending on what source you consult, Earley's algorithm is either a bottom-up algorithm, that incorporates some top-down predictive elements, or a top-down predictive algorithm that has bottom-up subsumption checking. Either way, Earley's algorithm is seen as a marriage between a bottom-up and top-down approach.

Earley seems to come away from this marriage with the benefits of both top-down and bottom-up. From its bottom-up roots, Earley keeps a worst case runtime of $O(n^3)$, but because of the strength of the top-down predictive elements, it typically has a runtime of $O(n)$.

Shieber, Schabes, and Pereira approach Earley (as they did CKY), as a set of axioms and inference rules. The algorithm stores items of form $[i, A \rightarrow \alpha \cdot B\beta, j]$ where $A \rightarrow \alpha B\beta$ is a production rule within the grammar, and A has derived already derived a (possibly null) substring α, from from position i to $j-1$ of the string being operated on.

They describe three base operations scan, predict, and complete. Scan is akin to the axioms of CKY in that they recognise the pre-terminals that derive elements of the string, and produce items of the form $[i, A \rightarrow w_i \cdot, i+1]$. Predict is the top-down mechanism of the Earley algorith, and given an item in the form of $[i, A \rightarrow \alpha \cdot C\beta, j]$ will add an item $[j, C \rightarrow \cdot\gamma, j]$ for every rule $C \rightarrow \gamma$ in the grammar. The bottom-up mechanism is the complete operation, which takes two items, an item that needs to be further recognized $[i, A \rightarrow \alpha \cdot C\beta, k]$ and an item which has been fully recognised and can be predicted from the first unrecognized element of the first $[k, C \rightarrow \gamma \cdot, j]$ and produces an item much like the first, only with the next symbol recognized $[i, A \rightarrow \alpha C \cdot \beta, j]$.

The Earley algorith begins by first adding items for all the productions of the start symbol. The algorithm then predicts the items that are producible from those

0	$[0, S \rightarrow \cdot NP\ VP, 0]$	AXIOM
1	$[0, NP \rightarrow \cdot A\ N, 0]$	PREDICT from 0
2	$[0, A \rightarrow \cdot the, 0]$	PREDICT from 1
3	$[0, A \rightarrow the \cdot, 1]$	SCAN from 2
4	$[0, NP \rightarrow A \cdot N, 1]$	COMPLETE from 1 and 3
5	$[1, N \rightarrow \cdot ball, 1]$	PREDICT from 4
6	$[1, N \rightarrow \cdot boy, 1]$	PREDICT from 4
7	$[1, N \rightarrow ball \cdot, 2]$	SCAN from 5
8	$[0, NP \rightarrow A\ N \cdot, 2]$	COMPLETE from 4 and 7
9	$[0, S \rightarrow NP \cdot VP, 2]$	COMPLETE from 0 and 8
10	$[2, VP \rightarrow \cdot V\ NP, 2]$	PREDICT from 9
11	$[2, V \rightarrow \cdot hit, 2]$	PREDICT from 10
12	$[2, V \rightarrow hit \cdot, 3]$	SCAN from 11
13	$[2, VP \rightarrow V \cdot NP, 3]$	COMPLETE from 10 and 12
14	$[3, NP \rightarrow \cdot A\ N, 3]$	PREDICT from 13
15	$[3, A \rightarrow \cdot the, 3]$	PREDICT from 14
16	$[3, A \rightarrow the \cdot, 4]$	SCAN from 15
17	$[3, NP \rightarrow A \cdot N, 4]$	COMPLETE from 14 and 16
18	$[4, N \rightarrow \cdot ball, 4]$	PREDICT from 17
19	$[4, N \rightarrow \cdot boy, 4]$	PREDICT from 17
20	$[4, N \rightarrow boy \cdot, 5]$	SCAN from 19
21	$[3, NP \rightarrow A\ N \cdot, 5]$	COMPLETE from 17 and 20
22	$[2, VP \rightarrow V\ NP \cdot, 5]$	COMPLETE from 13 and 21
23	$[0, S \rightarrow NP\ VP \cdot, 5]$	COMPLETE from 9 and 22

Figure 5: Earley as applied inference rules

items, until all possiblities have been predicted. The algorithm then scans the first element of the string input, adding items as necessary. The algorithm then goes and attempts to complete all items that can. From those completed items, new items are predicted, the next element of the input string is scanned, and what items can, are completed. This process is repeated until the entire string has been scanned. An example parse of the simple english sentence using the Earley inference rules is shown in Figure 5. [SSP95]

3 Parallel Computing

As the size of the grammars and parse structures grows, there comes the need for more computing power to handle them. One way to achieve this extra computing power is to distribute the computation among several processors in a parallel environment.

3.1 Parallel Environments

Parallel environments come in many different shapes and sizes. One of the main factors is the level at which the parallelism occurs. One of the lowest levels in which parallelism occurs is at the instruction level, where within a single processor there can be multiple execution units that can operate simultaneously. A similar concept is to have several processors accessing a single local memory.

A slightly higher level of parallelism involves multiple processors and multiple chunks of memory all connected together through some interconnection medium. Each processor has access to all the memory, meaning all the memory is addressed within a single address-space and shared between the processors. This is commonly called shared-memory multiprocessor architectures.

Another common type of parallelism involves connecting together many individual machines, each consisting of a processor and local memory, through some interconnection medium. In this architecture, each processor only has access to its own local memory. the interconnection medium provides a way for the various processors to communicate and exchange information. This architecture is know as a message-passing architecture.

A variation of the message-passing architecture, where all the local memories are grouped into one address space, and are accessible to all processors, are known as distributed shared memory systems. In these systems processors can access any of the memories that are local to another processor by sending a message to that particular processor. Through this process a large quantity of memory can be shared, however the access time for a particular segment of memory is un-uniform depending on what processor it is local to.

3.2 Granularity

To obtain any form of speedup in a parallel environment, the overall task must be divided into smaller processes of which several can be executed simultaneously. The size to which the task can be subdivided into can be described as the granularity of the task. If the individual processes are large and take a substantial time to compute,the granularity is coarse. Conversely, if the the individual processes are small

and quick to execute, the granularity is fine. With coarse granularity, there is less overhead to the distribution of tasks, as there are fewer of them to distribute. At the same time however, there can be less of them being operated on concurrently. With fine granularity, there are many processes that can be executed simultaneously, but they must be distributed more often, increasing overhead. When decomposing a problem for parallel execution, a balance must be struck between too fine and too coarse of granularity.

3.3 Models of Parallelism

This balance of granularity is closely linked to the particular parallel architecture on which the processes are being run. In a Symmetric Multiprocessing (SMP) environment where the parallelism takes place at a very low level, a fine granularity makes sense, because the communication overhead is quite low. However on the other end of the scale, a fine granularity would be completely useless in a massively parallel architecture such as Folding@Home, where the huge cost of the communication makes having very coarse granularity the best approach. The ratio of communication (and other overhead) time and computation time can be used to measure whether the granularity of an algorithm is appropriate.

3.4 Speedup

Ideally the introduction of additional processors that can operate concurrently would produce a speedup linear to the number of processors. This is however not typically the case, as there is typically overhead involved with distributing tasks among the processors. The speedup factor is the ratio between the time it takes one processor in a sequential implementation and n processors in a parallel implementation (it can also be measured by the number of computations as opposed the time it takes to execute them).

There are many overhead factors that cause the speedup to be sub-linear. One factor that causes overhead, are non-parallelizable sections of an algorithm, i.e. sections that must be computed serially before other processors can get involved. Coupled with that, is the fact that not all the processors will be doing useful work all the time. There will be periods where they will be idle, waiting for the next instruction. Another factor is that there typically are extra computations that must occur for the problem to be divided up. Another factor is the delay that is inherent in communication. The influence of these factors vary based on the problem and the architecture. [WA99]

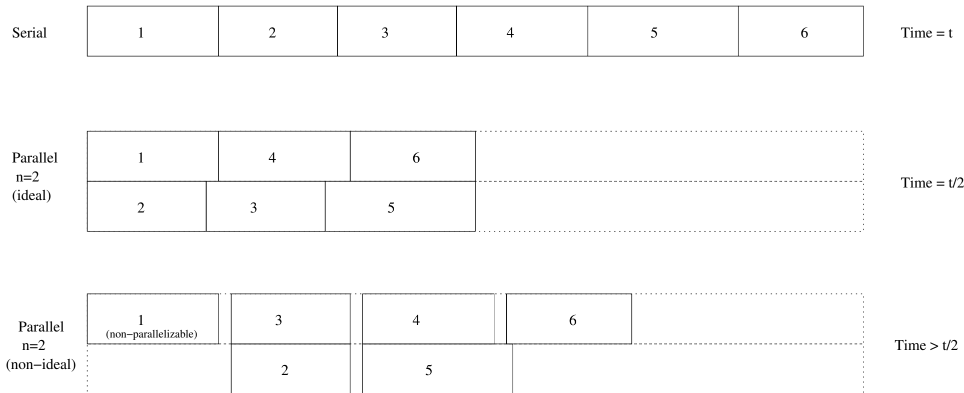

Figure 6: Time diagram of Serial code versus Parallel code on two processors.

4 Past Approaches To Same Problem

In their paper, Hill and Wayne outline two different approaches to adapting the CKY algorithm to a message-passing based parallel environment. Their first approach was to first distribute the grammar and string to be parsed among the processors,using a separate processor for each of the collumns in the CKY table. As results were generated by the processors, they would be communicated to the processor to the left, as though moving on a diagonal with in the CKY table. This implementation had several flaws. Firstly it required a processor for each element of the string. Additionally they found however that the cost of inter-communication was too costly and resulted in very little speedup.

Hill and Wayne then revise their approach to deal with the situation where there are less processors than elements of the string. In this revised approach, along with the string and the grammar, a distribution series is communicated at the begining to the processors. This distribution series told each processor how many collumns it would be responsible for computing, and allowed for better user-specified distribution of work. In addition to the better distribution of work, having a single processor in charge of several adjacent collumns allowed them to better arrange the order in which the individual CKY cells were computed. They found that by calculating along the diagonals first they were able to compute results fastest. With the improved algorithm, they were able to get a speedup of over 2 times with 3 processors (70% efficiency), and a 3.5 times speedup with 7 (50% efficiency). They found that with more processors the communications overhead became greater, leading to the lesser effeciency. [HW91]

Ninomiya, Torisawa, Taura, and Tsujii developed a parallel CKY-style algo-

rithm for efficiency on a massively parallel computer. They approach the parallelization in a different manner. Their parallel envirionment allowed them to parallelize the two driving loops of the CKY algorithm, in effect making the algorithm into a bunch of semi-independent processes at the level of a cell in the CKY table. These independent cells would begin executing as soon as results from the cells they were dependent on became availible. As soon as a cell achieves a result, that result is made accessible to all other processes, allowing those that are dependent on it to proceed. In this way, the algorithm greatly resembles the original dynamic programming algorithm for CKY, where the only explicit process ordering is inherent to the bottom-up nature of CKY.

They were able to achieve this semi-independence through the use of a shared memory system, where all the processes could easily access the results of the others. The particular architecture they use consists of a 256 node parallel machine where each node contains a SuperSparc chip and memory. runs ABCL/f, a concurrent object-oriented language. They also compare it to a serial version running on a single processor. The single processor architecture they use is a Sun SPARC Station 10, which runs the same processor as the parallel architecture. The result of their tests showed that the influence of extra processors becomes greater as the length of the string being parsed grows and as the number of processors grows. They were able to obtain a speedup of 45 times using 250 processors, with the largest sentence length they tested (115-125 characters). [NTTT97]

Bruschi and Pighizzini first develop a parallelized Earley-style recognizer with time complexity $O(\log n)$, using $O(n^6)$ processors. From this recognizer they develop a parallelized Earley-style parser of the same complexity. The parallel architecture they are using is a system of several independent sequential processors using a shared memory system, with concurrent reads and writes.[BP94]

Cohen, Hickey and Katcoff do not really develop an concrete algorithm or even a pseudo-code framework for a parsing algorithm. instead they deal with the theoretical speedup, for any parallel parsing algorithm. They begin by developing several coarse upper bounds and then refining them [CHK82]

Janssen, Poel, Sikkel, and Zwiers first propose a general abstract framework of parallel parsing, that they call the primordial soup algorithm, which they then refine to obtain a CKY-style bottom-up algorithm. [JPSZ92]

References

[BP94] Danilo Bruschi and Giovanni Pighizzini. A parallel version of earley's algorithm, 1994.

[CHK82] Jacques Cohen, Timothy Hickey, and Joel Katcoff. Upper bounds for speedup in parallel parsing. *J. ACM*, 29(2):408–428, 1982.

[HW91] Jane C. Hill and Andrew Wayne. A cyk approach to parsing in parallel: a case study. In *Proceedings of the twenty-second SIGCSE technical symposium on Computer science education*, pages 240–245. ACM Press, 1991.

[JPSZ92] Wil Janssen, Mannes Poel, Klaas Sikkel, and Job Zwiers. The primordial soup algorithm: a systematic approach to the specification of parallel parsers. In *Proceedings of the 14th conference on Computational linguistics*, pages 373–379. Association for Computational Linguistics, 1992.

[NTTT97] Takashi Ninomiya, Kentaro Torisawa, Kenjiro Taura, and Jun'ichi Tsujii. A parallel cky parsing algorithm on large-scale distributed-memory parallel machines, September 1997.

[SSP95] Stuart Shieber, Yves Schabes, and Fernando Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 1995.

[WA99] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 1999.