Exploring The Effectiveness of Parallel Computation on CKY

Mahmoud Zeidan
mahmoud.zeidan@mail.utoronto.ca

Bocheng Zhang
bocheng.zhang@mail.utoronto.ca

Ioan Sahlas
ioannes.sahlas@mail.utoronto.ca

Abstract

The Cocke-Kasami-Younger (CKY) parsing algorithm has widely been used in natural language processing (NLP) tasks due to its simplicity and efficiency. However, its heavy reliance on its memoization data structure has restricted its implementations to be single-threaded. In this paper we develop a parallelized version of the CKY parsing algorithm and compare it with the traditional single-threaded version. We show that computational strategies can be employed to avoid communication, synchronization, and load imbalance related slow-downs due to its memoization data structure. We also show that parallelized CKY performs better than single-threaded CKY in parse time as the number of ambiguities in the input increases, although performs worse as input length increases, but the number of ambiguities stay the same.

1 Introduction

CKY is a bottom-up parsing algorithm for context-free grammars (CFG). It uses dynamic programming techniques to store previously computed parse trees in a memoization data structure for future computational steps. Almost every step of CKY relies on looking up computation saved in its memoization structure. This, along with the sequence the table is filled with, makes it very difficult to parallelize the algorithm without causing thre- ads to block, waiting for other threads to finish filling their cells. In this paper we develop a parallelized CKY algorithm that uses computational strategies to avoid as much parallelization overhead as possible, and present a detailed performance comparison between its traditional single threaded counterpart.

2 The Single-Threaded Algorithm

It is important to understand how the traditional single-threaded algorithm works in order to understand why it is difficult to parallelize it.

2.1 Explanation of Single-Threaded CKY

To begin, CKY takes an input sentence "x y z … " and represents it with numbered posts between each token: "0 x 1 y 2 z 3 …". Then, CKY fills a 3D matrix called a parse table from left to right, bottom to top. Each cell (i, j) contains every grammatical constituent that covers the span of the input from post i to post j. For example, let A → B C be a production in the CFG. If span (i, j) from the input sentence can be split such that there exists some k between i and j where B covers span (i, k) and C covers span (k + 1, j), then A should be added to cell (i, j). This process is done by looping through the cells under (i, j) starting from the one right under it and going down. For every cell b that is x cells away from the bottom, we check every label in b with every label in cell l, where l is x cells away to the left of cell (i, j).

If there exists any production in the CFG that produces one of the label combinations, then we add the label on the left hand side of the production to cell (i, j). The final cell that is filled is the top right cell.

	1	2	3	4	5
0	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
1		(1,2)	(1,3)	(1,4)	(1,5)
2			(2,3)	(2,4)	(2,5)
3				(3,4)	(3,5)
4					(4,5)

Figure 1.

2.2 Why It's Hard to Parallelize

As explained above, every cell (i, j) is dependent on all the cells to its left and all the cells beneath it. This means if we want to fill a cell, we need to wait until the entire left side of that row has already been filled. This places difficulty on how computation can be synchronized. Computational resources should be allocated such that only cells that can currently be filled should be processed.

3 The Parallelized Algorithm

There are not many ways to approach parallelizing the algorithm due to the computational dependency described in section 2.2. Here, we will describe the approaches we considered.

3.1 Diagonal Approach

Given that each cell relies only on all the cells to its left and beneath it, every cell on a diagonal can be independently computed from the others. In the visualization above (Figure 1), each color indicates that those cells are independent from one another, and hence can be computed in parallel. The problem with this approach is that an entire diagonal must be filled before moving to the one above it. It is possible for a thread to finish filling the cells allocated to it, but be unable to move on until all the other threads are done, even though it would be possible in some cases to move to the next diagonal. For example, consider a situation where (2, 3) and (3, 4) are the only cells that have been filled at a certain moment. It is possible to now fill cell (2, 4), since it relies only on (2, 3) and (3, 4) which have just been filled, but instead we must wait on all the other cells on the main diagonal to be filled before moving to cell (2, 4).

3.2 Block Partitioning

Another way to approach the problem is to allocate to each thread a contiguous set of columns. Each thread is responsible for filling the cells of those columns bottom up. This approach is also problematic. What it does is partition the parse table into non-concurrent sub-tables. Consider a parse table with 6 columns and 2 threads. Thread T1 is responsible for columns 1 to 3, and thread T2 is responsible for columns 4 to 6. For any cell in columns 4 to 6, T2 has to wait for the entire left side of the row that cell is on to be filled. However, those cells are being filled non-concurrently by T1. This means that while T1 is busy filling column 1, then 2, then 3 in that order T2 is blocked until T1 has gotten to column 3 and filled the last cell on that row. The problem stems from the fact that the more rightwards you proceed on the table, the more those cells rely upon the left side of the table. It doesn't make sense to split the table in contiguous blocks as the work starts exclusively on the left side of the table, and makes its way to the right. This problem is addressed effectively by strided partitioning.

3.3 Strided Partitioning

The approach we ended up implementing is strided partitioning. Instead of allocating to each thread a block of contiguous columns, each thread is allocated columns spaced out by a stride greater than 0. For example, in a 6-column parse table and 2 threads, thread T1 would be responsible for filling columns 1, 3, and 5, while T2 would fill 2, 4, and 6. This solves the problem in the following way: When T1 is filling column 1, T2 is filling column 2. T2 only requires a single cell to be filled by T1 in order to fill one of its cells. Both threads almost build upwards together, with T2 relying directly on T1. When T1 and T2 finish, they move on to column 3 and 4, respectively. Again, T2 only has to wait on T1 to finish its adjacent cell, since the cells on column 1 and 2 have already been filled. This example generalizes to more than 2 threads.

4 Implementation

4.1 Language

Given that CKY heavily relies on memory, we wanted to implement our algorithms in a language that has the least amount of memory read and write overhead as possible, but still provides tools to help us in our implementation. We elected to go with C++ as it allows us to modify memory directly through pointers, while providing enough libraries to keep our implementation readable.

4.2 Libraries

Both the single and parallelized versions of CKY rely on standard library containers commonly used in practice, such as std::vector, std::unordered_map, and std::string. Multithreading was done using the thread and mutex libraries. Performance timing was done using the chrono library.

5 Results

CKY's performance relies on two factors: the size of the parse table, and the amount of grammatical ambiguities existing in the input sentence. In order to thoroughly test our algorithm, we need to benchmark these factors separately and together.

5.1 Context Free Grammar

For the purpose of simplifying testing, we constructed a custom CFG that would allow us to manually generate sentences with specific lengths and specific amounts of ambiguities. The CFG we created is:

$$S \rightarrow S\ S \mid \text{`a'} \mid \text{`('}\ S\ \text{`)'}$$

The language modelled by the grammar is the language of all strings containing just 'a's, '('s, and ')'s, where a string can be of any length, and the bracketing is well formed. For example, "aaaaa", "(aaa(a))", and "a(aa(aa)a)aa" are all strings of the language. This grammar allows us to create arbitrarily large sentences simply by appending 'a's, and controlling ambiguity by place brackets. CKY only works with grammars in Chomsky Normal Form (CNF). This is the corresponding CNF grammar:

$$S \rightarrow S\ S$$

$$S \rightarrow a$$

$$S \rightarrow \text{LEFT-S RIGHT}$$

$$\text{LEFT-S} \rightarrow \text{LEFT S}$$

$$\text{LEFT} \rightarrow \text{`('}$$

$$\text{RIGHT} \rightarrow \text{`)'}$$

5.2 Sentence Length – 0 Ambiguities

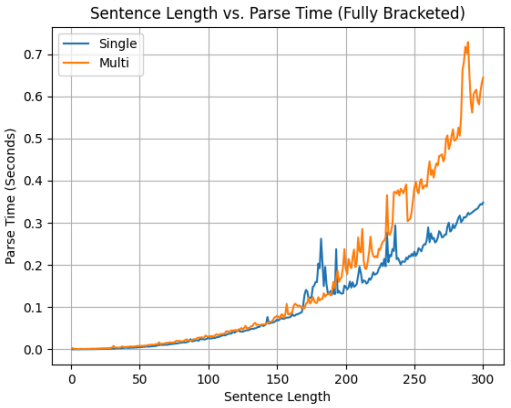

Figure 2.

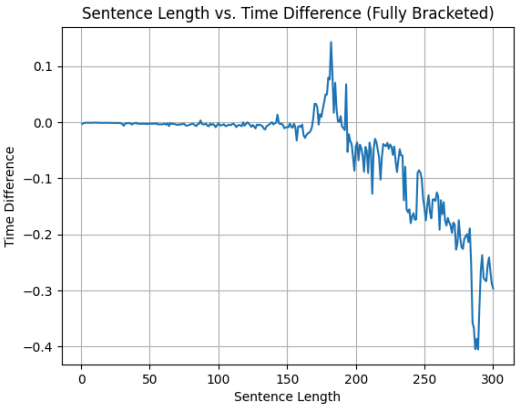

Figure 3.

The first factor we tested is how performance compares when sentence length increases, but ambiguities stay the same. We generated sentences of the form "(a(a(a(…)))", which all have no ambiguities since they are fully bracketed. We tested sentences ranging from 1-100 'a's. We found that as the length of the sentence increases, both CKY and parallelized CKY parse times get longer with an almost quadratic relation. This makes sense since the parse table has dimensions $n \times n$ where n is the length of the sentence. We also found that the parse time difference between single and parallelized CKY becomes greater as the sentence size increases, with parallelized CKY being generally slower except in some occasions when the sentence length is sub 100 and around 175. We propose that parallelized CKY is slower due to communication overhead being much greater than the computation time spent on a single cell. Since there are no amb-

iguities in the input, each cell only gets filled with one label, meaning a very small amount of time is spent on each cell. Conversely, parallelized CKY must acquire and release a lock for every cell to the left of the one it's filling, and once for the cell its filling. This communication overhead probably causes parallelized CKY to be slower than the single-threaded version.

5.3 Sentence Length – Max Ambiguities

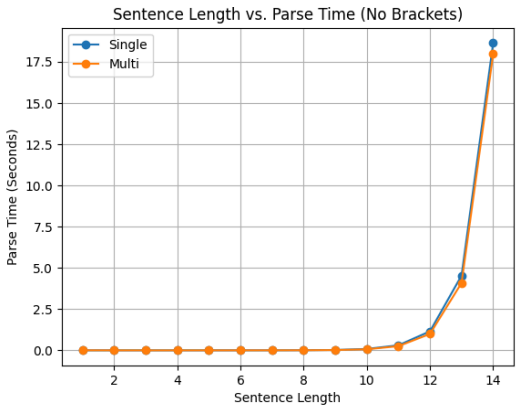

Figure 4.

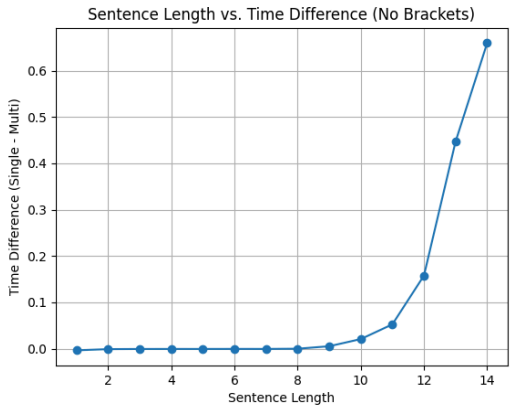

Figure 5.

Our next benchmark was to test how performance is affected when the sentence length increases and the number of ambiguities increases. To do this, we generated sentences of increasing length that contain no brackets. For example, "aa", "aaa", "aaaa". Since there are no brackets to reduce the ambiguities, as the sentence length increases, the amount of ambiguities also increases. We tested both the single and multithreaded CKY on a range of sentences from 1-14. Our computational resources did not allow us to test sentence lengths greater than 14. This is because the amount of ambiguities is proportional to the n'th catalan number, $\frac{(2n)!}{(n+1)!n!}$. This means a sentence of length 15 would have 9,694,845 possible ambiguities. This was not computationally possible for us, given our hardware. Regardless, our results show that parallelized CKY becomes faster and faster as we approach a sentence length of 14. The trend appears to be factorial, and we expect the trend to keep growing as the sentence length goes beyond 14. The explanation for this is simple. As the amount of ambiguities increases, the amount of computation per cell increases proportionally to the Catalan number. As a result, the communication overhead resulting from acquiring and releasing locks becomes negligible. In other words, the amount of computation per cell takes the lead in terms of parse time, and hence the ability for parallelized CKY to concurrently fill cells becomes apparent.

5.4 Number of Ambiguities, Fixed Length

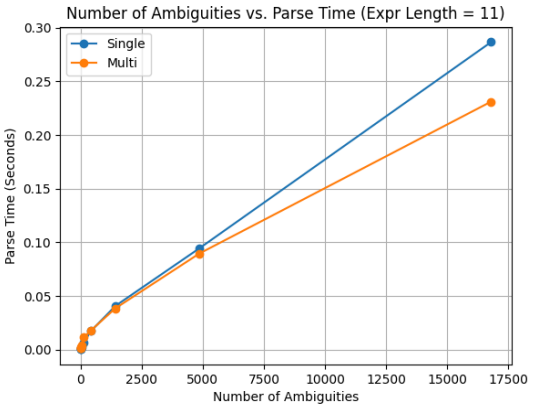

Figure 6.

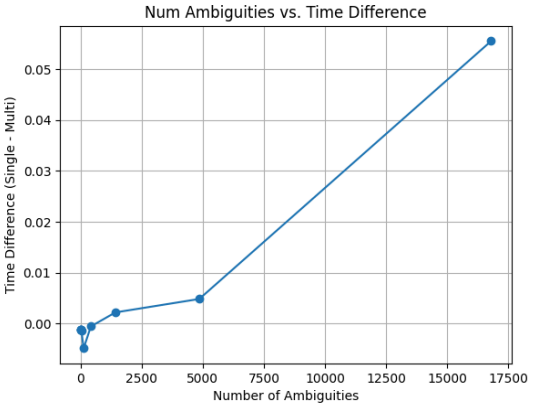

Figure 7.

Given the results from our test in section 5.3, we wanted to see how the performance compares when the sentence length is fixed, but the amount of ambiguities increases. We fixed the sentence length to 11, and for each Catalan number c from 1 to 11, we generated 10 sentences of length 11 and c ambiguities. We ran single and multithreaded CKY on each sentence and took the average. We could not increase the sentence length to more than 11 because our ambiguous sentence generator would take too long to generate 10 sentences. Our results are similar to section 5.3. Parallelized CKY performs better and better as the number of ambiguities increases. We expect that the trend is also proportional to the Catalan number, but our hardware limitations did not allow us to test with bigger Catalan numbers. The explanation for the trend is the same as in section 5.3.

5.5 Summary of Tests

We found that parallelized CKY is faster than single-threaded CKY as the number of ambiguities in the input sentence increases. This is because as the number of ambiguities increases, the amount of computation done to fill a cell increases proportionally. Parallelized CKY is able to compute cells in parallel, while single-threaded CKY can only compute one cell at a time. On the other hand, parallelized CKY does not benefit from the sentence length being large. The bigger the parse table, the more communication overhead exists between threads. In such cases, single-threaded CKY out-performs. It is important to note that throughout all our tests, the parse time difference between CKY and parallelized CKY was never more than a second.

6 Conclusion

6.1 Concluding Summary

Using strided partitioning, we were able to develop a parallelized version of the CKY parsing algorithm that outperforms single-threaded CKY on an atleast one metric: *grammatical ambiguity*. Single-threaded CKY outperformed our parallel algorithm when solely considering input length. However, the results were negligible, with the parse time difference never exceeding one second in any test. While parallelized CKY showed improvement over single-threaded CKY proportional to factorial, our tests already exceeded realworld NLP contexts. It is highly unlikely that anyone would need to parse a natural language sentence with more than 10000 ambiguities, or more than 300 words. The results of our study demonstrate that it is possible to create a parallelized version of CKY that performs as well as single-threaded CKY, but not sufficiently better to serve as a replacement.

6.2 Future Work

To simplify testing, we created a CFG that generated simple sentences allowing us to control length and the number of ambiguities. Our language was not representative of actual human natural language. It would be interesting to see how parallelized CKY would perform in real world NLP contexts with human natural language.

5 References

J. W. BACKUS. "*THE SYNTAX AND SEMANTICS OF THE PROPOSED INTERNATIONAL ALGEBRAIC LANGUAGE OF THE ZURICH ACMGAMM CONFERENCE*." International Business Machines Corp., https://www.software-preservation.org/projects/ALGOL/paper/Backus-Syntax_and_Semantics_of_Proposed_IAL.pdf (shows the re-invention of the concept of a context-free grammar to describe the syntax of the programming language ALGOL. Our parsers will be using context-free grammar in order to parse sentences).

Groucho Marx, Animal Crackers. "*Context-Free Grammars and Constituency Parsing*." Speech and Language Processing. 1930, https://web.stanford.edu/~jurafsky/slp3/18.pdf (the textbook chapter on constituency parsing. The CKY algorithm that we will parallelize is based on the one described in this chapter).