# Parallel Parsing of Dense Probabilistic Context-Free Grammars for Natural Language Processing

Omar Rahman
15-418 Final Project
Spring 2015

## Summary

I implemented a parallelized version of the CKY algorithm using OpenMP and compared its results to two versions of a sequential implementation.

## Background

Parsing grammars is a common task in natural language processing. A grammar is a list of production rules that describe how words can be syntactically related. The CKY algorithm is a dynamic programming algorithm that parses sentence in a worst case O(|G|n^3) complexity, where n is the length of the input sentence and |G| is the size of the grammar. The algorithm requires the grammar to be in Chomsky normal form (CNF), meaning that it consists of only unary (A → B) and binary (A → B C) production rules. Every context-free grammar can be transformed into an equivalent one which is in Chomsky normal form. In addition, each rule is paired with the probability of occurring.

## The Algorithm

The algorithm uses the upper triangular portion of a 2-dimensional chart (Fig. 1) and considers every possible subsequence of words of the input sentence. Each cell is filled with all the possible symbols that the subsequence parses to completely along with their probabilities of occurring. The cells are filled by considering every possible partition of the subsequence into two further subsequences and checks the grammar for the rule derived from the two subsequences. The cells that form these partitions are exactly those to the left and below the cell being considered. The sentence has a valid parse if and only if the start symbol (usually denoted "S" for "sentence") occurs with non-zero probability in the top-right cell of the chart, i.e. the cell that represents the entire input sentence.

The primary data structure of the algorithm is the upper triangular portion of a 2-dimensional chart (Fig. 1), where each cell of the chart contains a mapping of valid rules to their probabilities. Each cell is dependent on the cells to the left and below it; thus, the optimal way to fill in the chart is by diagonals computed in parallel.

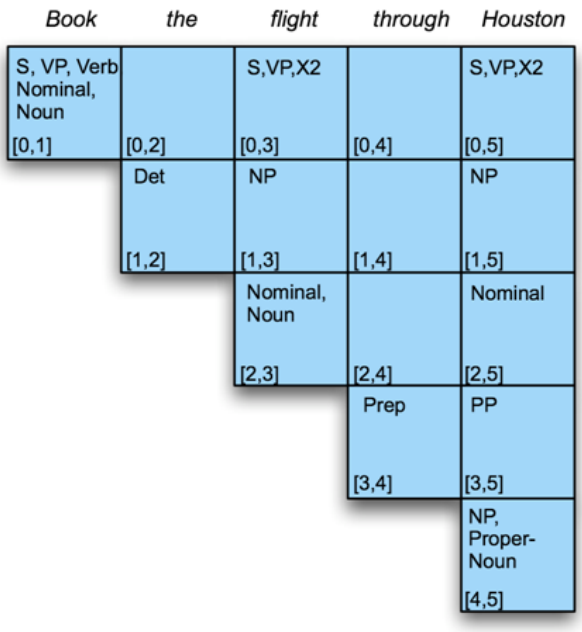


**Figure 1: A CKY chart. Note that probabilities are omitted**

## Approach

I first implemented a sequential version of the algorithm in C++ on the GHC machines. I represented the chart as a 2-dimensional array of maps of strings to doubles. I then attempted to parallelize the algorithm using CUDA; however, I achieved very poor results and decided to scrap it. The problem was that parsing a single sentence did not provide enough arithmetic intensity to warrant the overhead incurred in the memory copying required in GPU programming. Since my goal was to have a low latency solution for user-facing applications, I decided that a GPU approach would not be appropriate for my application.

I decided that parallelizing the sequential implementation directly using OpenMP would be a better approach. I parallelized the outside loop of the algorithm and synchronized accesses to each cell by maintaining private per-thread probabilities and then performed a parallel reduction to combine the results of individual threads. This achieved a modest 2x speedup. I

```
for i in 0, . . ., n−1:
  for a in 0, . . ., m−1:
    C[i,i+1,a] = T[W[i],a]

for gap in 2, . . ., n:
  for i in 0,..,n−gap:
    k = i+gap
    BC = Zero
    for j in i+1, . . ., k−1:
      for b in 0, . . ., m−1:
        for c in 0, . . ., m−1:
          BC[b,c] += C[i,j,b]∗C[j,k,c]
    for a in 0, . . ., m−1:
      C[i,k,a] = 0
      for b in 0, . . ., m−1:
        for c in 0, . . ., m−1:
          C[i,k,a] += R[a,b,c]∗BC[b,c]
```

where:

| | |
|---|---|
| $m$ | number of nonterminals in grammar |
| $n$ | length of input string |
| $W[i]$ | word in input string at position $i$ |
| $T[w, A]$ | probability of $A \to w$ |
| $R[A, B, C]$ | probability of $A \to B\,C$ |
| $C[i, k, A]$ | inside probability of $A$ spanning $(i, k)$ |
| $BC$ | an $m \times m$ scratch array |
| $Zero$ | an $m \times m$ array of zeros. |

**Figure 2: Factored CKY algorithm due to Johnson [2]**

experimented with different loop reorderings, which didn't really improve performance much at all. I then decided to optimize my serial implementation using the approach described in [2], in which the algorithm (Fig. 2) is factored into two stages: compute the intermediate production probabilities and write them to a scratch area and then multiply those values with those of the cell we're writing to. In other words, for a rule A → B C, I first multiply the probabilities of the parses of B and C into the scratch space in the first stage, and then multiply by the probability of the rule itself in the second stage. This approach provided a tremendous speedup to the baseline due to good cache locality, but the parallelized version still only achieved a 2x speedup from this factored version, which is not too surprising since the factored mainly exploited caching.

Finally, I decided to slightly modify my chart data structure as a 3-dimensional array of doubles, and indexed into the nonterminals vector to access a symbol. This optimization provided a 2x speedup to the sequential factored version and a 4-5x speedup to the parallel factored version. The amenability to parallelization of this version is likely due to much easier synchronization of standard array elements as opposed to the more complex concurrent map data structure.

**Results**

The main focus of my application was to optimize for latency, so I measured my results in terms of the time to parse a single sentence compared to both the baseline and factored implementations. These measurements were taken after the grammar and lexicon generation and word and rule chart initialization. I wanted to measure different aspects of my work: (1) how well I optimized the baseline sequential implementation using a factored algorithm and smarter data structures and more importantly, (2) how much parallelism I was able to achieve from the best sequential implementation.

To evaluate my system, I implemented a test harness to run and time all the various implementations three times each. The harness uses a seeded random grammar and lexicon generator with parameters for number of production rules, number of lexical rules, sentence length, and token length. A dense PCFG is constructed from the nonterminal symbols and, along with the lexicon, is also assigned random probabilities. The algorithm's input consists of the grammar, lexicon, and sentences to be parsed.

Since the size of the chart is directly related to the length of the input sentence, I decided to measure the parse time and speedup as sentence length varies (Appendix A, Figs. 3, 4). From the graph in Figure 3, which compares sentence length to parse time, we can see that the parallel version of the factored algorithm scales well to the problem size. Going past 60-word sentences is not necessary, since most real-world sentences average around 15-20 words in length. A 20-word sentence is parsed in about 0.05 seconds, which is certainly sufficient for use in user-facing applications. The graph in Figure 4 shows the speedup (the legend is interpreted as <1>_speedup_<2> means the speedup of <1> compared to <2>). The important result is the red line, representing the speedup of the parallel factored version compared to the serial factored version, which is the fairest comparison to make. It levels off at around a 5x speedup for large sentence lengths.

I also wanted to see how the algorithm scales with respect to thread count. From Figures 5 and 6, we see that performance scales well to about 6 cores and then trends downwards. This is likely due to the fact that more threads involve more contention for critical regions of code, which causes it to perform slightly worse at higher thread counts as opposed to smaller thread counts.

One interesting item to note is that before I moved from a 2D array of maps to a 3D array, my parallel implementation only achieved a 2x speedup; however, after the move, I achieved around a 4-5x speedup from the optimized sequential implementation using the same parallel approach. This result implies that the way I laid out my data made it more parallelizable; this was likely a result of the much simpler synchronization required to protect array elements as opposed to a map.

Despite failing to achieve even close to theoretical maximum speedup, I believe OpenMP was a good choice for this project considering that latency was the primary constraint of my application. Had throughput been preferred, then I would probably have taken advantage of a GPU. The overhead of memory transfer is too large for a single sentence

# Appendix A – Graphs

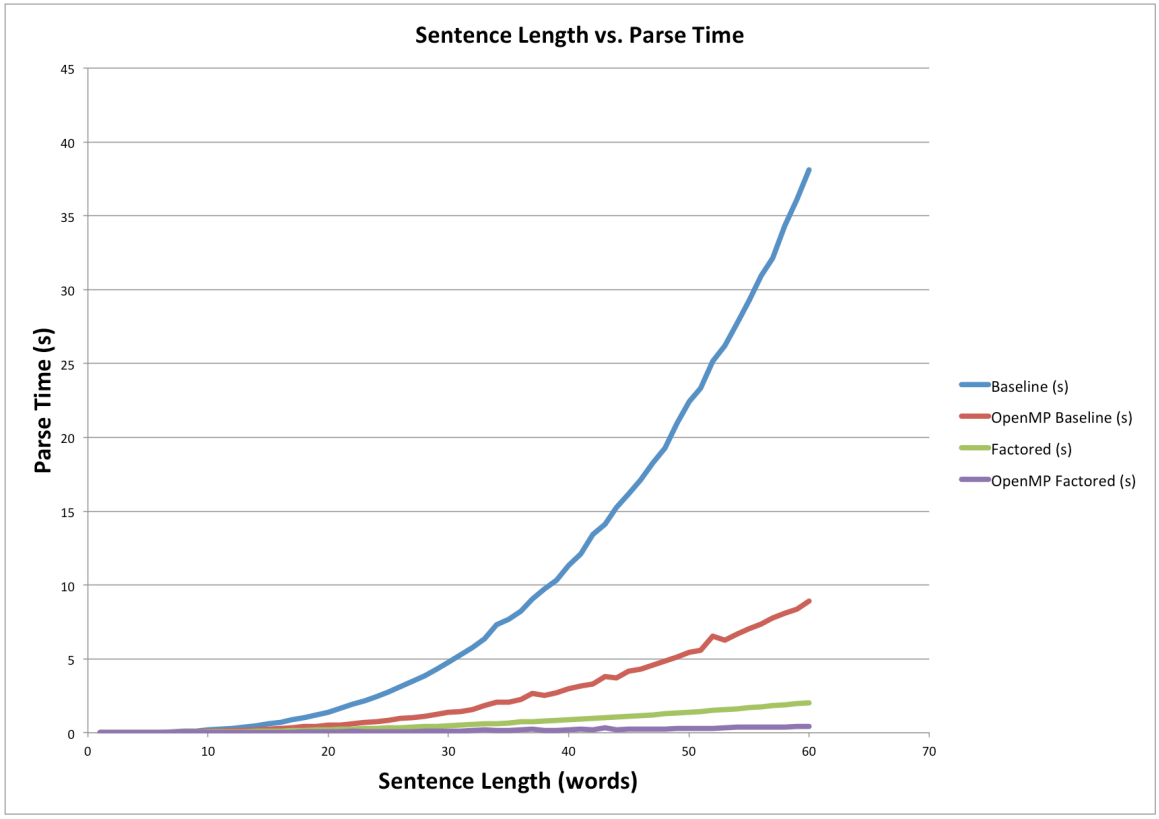


**Figure 3: Parse time as a function of sentence length**


**Figure 4: Speedup as a function of sentence length**

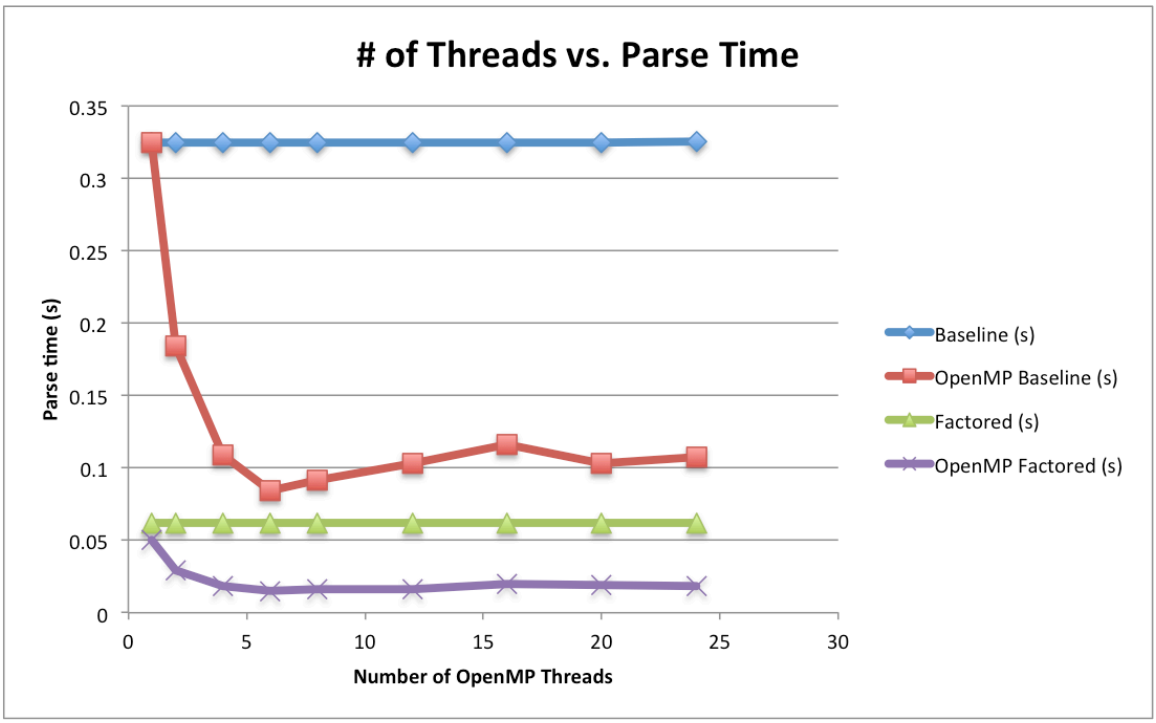


**Figure 5: Parse time as a function of OpenMP thread count**

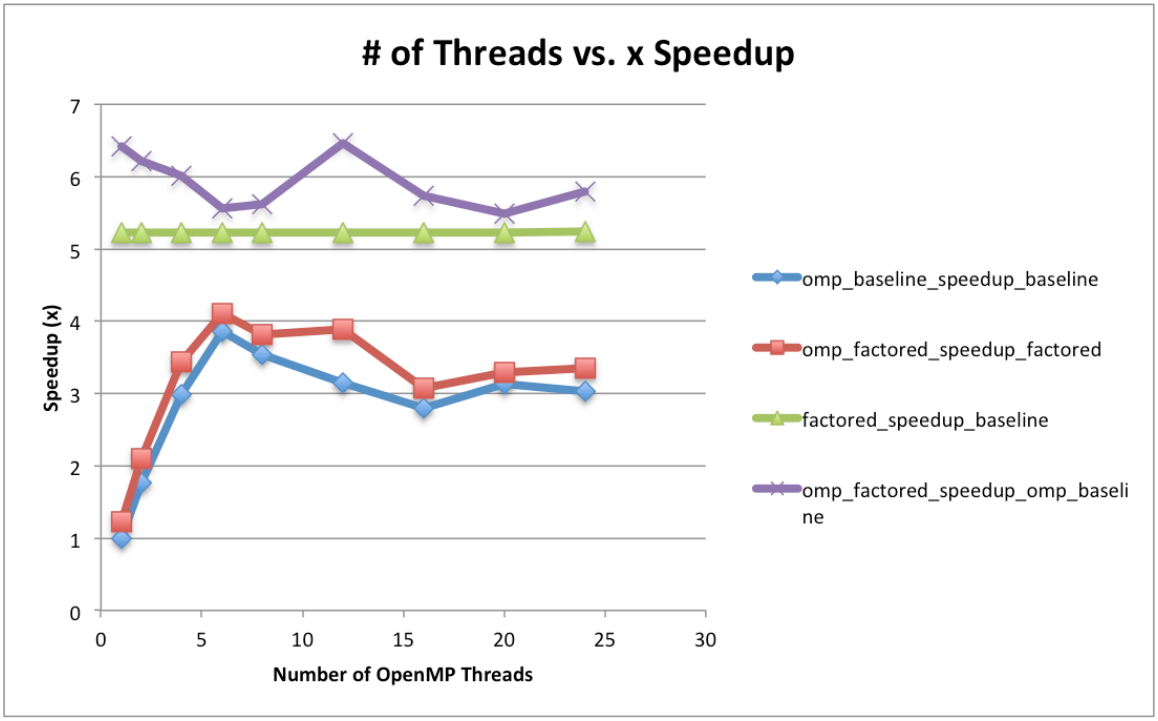


**Figure 6: Speedup as a function of OpenMP thread count**

## References

[1] Wikipedia, 'CYK algorithm', 2015. [Online]. Available: http://en.wikipedia.org/wiki/CYK_algorithm. [Accessed: 09- May- 2015].

[2] M. Johnson, 'Parsing in Parallel on Multiple Cores and GPUs', *Proceedings of Australasian Language Technology Association Workshop*, pp. 29–37, 2011.