

# A Parallel Extension of Earley's Parsing Algorithm

Greg Sandstrom

December 4, 2004

## 1 Introduction

Parsing is the process of deriving structure from a string, and can be used to describe the meaning of the string, and the relationships between its elements.

This paper describes two popular parsing algorithms, CKY and Earley. This paper also discusses attempts others have made to distribute the processing workload of the CKY algorithm in a parallel environment. The paper then describes how I hope to apply similar concepts to the parallelization of the Earley algorithm.

## 2 CKY Algorithm

The Cocke-Kasami-Younger (CKY) algorithm, first introduced in 1963[BP94], began as a dynamic programming algorithm, but can also be interpreted as a chart based parsing strategy for parsing context free grammars (CFG). CKY is strictly a bottom up algorithm, where each entry into the table is based on previous entries. 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 simplest form, it requires the grammar to be in a strictly 2-branching form, also known as Chomsky Normal Form (CNF). All CFGs can be represented in CNF, but the conversion is typically costly.

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

```

For( $0 \leq i \leq n$ ){
  Scan for  $w_i$ 
  producing  $[A, i, j]$  if  $A \rightarrow w_i \in \mathcal{G}$ 
}
For( $2 \leq l \leq n$ ){           #  $l$  is length of substring
  For( $0 \leq i \leq n-l$ ){     #  $i$  is start position of substring
     $j := i+l$                  #  $j$  is end position of substring
    For( $i < k < j$ ){         #  $k$  is split position within substring
      Complete for  $[B, i, k]$  and  $[C, k, j]$ 
      producing  $[A, i, j]$  if  $A \rightarrow B C \in \mathcal{G}$ 
    }
  }
}

```

Figure 1: Pseudo code for simple CKY algorithm

Shieber, Schabes, and Pereira[SSP95] 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 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  $C \rightarrow A B$  is a valid rule. The algorithm completes when  $[S, 0, n]$  is computed.

As a consequence of simple CKY algorithm being a dynamic programming algorithm, the algorithm can vary based on the control structures of the particular implementation. Figure 1 gives pseudo-code for a basic control structure. Using this control structure, it is useful to divide up the items of the CKY table into cells according to the start position and length of the substring that item dominates. In this structure the table becomes a triangular matrix, where all items in the  $i^{th}$  column of the  $l^{th}$  row, are the root of a structure that derives the substring from  $i$  to  $i + l$ . When filling in an cell of the matrix, the algorithm need only look down it's column and it's diagonal.

When parallelizing CKY it is often natural to decompose the problem in term of the major loops of the general framework of figure 1. The granularity of the parallelization depends on which of the major loops the implementation parallelizes.

## 2.1 Hill and Wayne's Parallelization of CKY

In their paper, Hill and Wayne[HW91] 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 columns in the CKY table. In terms of the loops of figure 1, they parallelize the second loop, assigning a processor to each  $i$ . As results were generated by the processors, they would be communicated to the processor to the left, as though moving on a diagonal within 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 fewer processors than elements of the string. In this revised approach, along with the string and the grammar, a distribution series is communicated at the beginning to the processors. This distribution series tells each processor how many columns it is responsible for computing, thus allowing for better user-specified distribution of work. In addition to the better distribution of work among the processors, having a single processor in charge of several adjacent columns allows 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 efficiency.

## 2.2 Ninomiya, Torisawa, Taura, and Tsujii's Parallelization of CKY

Ninomiya, Torisawa, Taura, and Tsujii[NTTT97] developed a parallel CKY-style algorithm for efficiency on a massively parallel computer. They approach the parallelization in a different manner. Their parallel environment 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 available. 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).

### 3 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[Erb94]. Either way, Earley's algorithm is seen as a marriage between a bottom-up and top-down approach[SSP95].

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, in many cases it has a runtime of  $O(n)$ .

Shieber, Schabes, and Pereira[SSP95] 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  $\alpha$ , 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 recognize 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 algorithm, 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 recognized 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 algorithm begins by first adding items for all the productions of the start symbol. The algorithm then predicts the items that are producible from those items, until all possibilities 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

```

Add items  $[0, S \rightarrow \cdot \gamma, 0]$  for start symbol  $S$ .
For( $0 \leq x \leq n$ ) {
    #  $w_0 \dots w_{x-1}$  has been parsed
    For( $0 \leq i \leq x$ ) {
        #  $i$  is the start position
        Predict  $[x, B \rightarrow \cdot \gamma, x]$  for all items  $[i, A \rightarrow \alpha \cdot B \beta, x]$ 
    }
    Scan  $[x, A \rightarrow w_x \cdot, x + 1]$  for items  $[x, A \rightarrow \cdot w_x, x]$ 
    For( $x > i \geq 0$ ) {
        #  $i$  is the start position
        For( $x > k \geq j$ ) {
            #  $k$  is the split position
            Complete  $[i, A \rightarrow \alpha B \cdot \beta, x]$ 
            for items  $[i, A \rightarrow \alpha \cdot B \beta, k]$  and  $[k, B \rightarrow \gamma \cdot, x]$ 
        }
    }
}
}

```

Figure 2: Pseudo code for simple Earley algorithm

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.

A simple pseudo-code segment that describes one possible control structure that could be built upon the Earley parser is given in figure 2.

## 4 Parallelization of Earley's Algorithm

There are many similarities between the structure of CKY and Earley parsing algorithms. Therefore techniques that may work well for CKY may be able to be applied to Earley as well. In this frame of mind, I hope to take many of the same techniques that Hill and Wayne applied to CKY, and apply them to Earley. The reason I chose to follow the example of Hill and Wayne is that they also use a message passing parallel environment.

### 4.1 Tasks At Hand

I have already been working on a sequential Earley algorithm. The first step for future work, is to further refine this sequential algorithm to divide the Earley table into cells akin to the cells of the CKY algorithm.

The next step involves parallelizing this improved Earley algorithm using the parallelization style that Hill and Wayne employed in their first attempt at parallelizing CKY. As such, I won't be looking for optimal speedup, but i will be interested in gaining a better understanding of the communication costs and other factors.

Figure 3: Schedule for future work

Following the same plan as Hill and Wayne, I then hope to refine the first parallel algorithm to distribute multiple columns of work to each processor, through a distribution series. Also I hope to find some implementation tricks (akin to Hill and Wayne computing the diagonals at a time) that will also help to improve the speedup.

When the second improved parallelized algorithm is completed, I hope to be able to refine the distribution series to achieve an "optimal" speedup. However in the spirit of Hill and Wayne, I may have to leave this as a future pedagogical exercise.

## 4.2 Timing

Figure 3 is a gant chart that shows a tentative schedule for when the tasks listed above will be completed within.

## References

- [BP94] Danilo Bruschi and Giovanni Pighizzini. A Parallel Version of Earley's Algorithm, 1994.
- [Erb94] Gregor Erbach. Bottom-up earley deduction. In *Proceedings of the 15th International Conference on Computational Linguistics COLING-94*, Kyoto, Japan, 1994.

- [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.
- [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.