Stochastic Parsing and Parallelism*

Francisco-Mario Barcala, Oscar Sacristán, and Jorge Graña

Departamento de Computación
Facultad de Informática
Universidad de La Coruña
Campus de Elviña s/n
15071 - La Coruña
Spain
{barcala, agullo, grana}@dc.fi.udc.es

Abstract. Parsing CYK-like algorithms are inherently parallel: there are a lot of cells in the chart that can be calculated simultaneously. In this work, we present a study on the appropriate techniques of parallelism to obtain an optimal performance of the *extended CYK algorithm*, a stochastic parsing algorithm that preserves the same level of expressiveness as the one in the original grammar, and improves further tasks of robust parsing. We consider two methods of parallelization: distributed memory and shared memory. The excellent performance obtained with the second one turns this algorithm into an alternative that could compete with other parsing techniques more efficient *a priori*.

1 Introduction

The CYK algorithm [6, 9] can be extended to work with any kind of context-free grammars (CFGs), not only with grammars under Chomsky's normal form (CNFGs). Of course, one way to do this is by turning the original CFG into a CNFG. There are a lot of well-known algorithms for performing this transformation. However, this process produces a high number of new auxiliary non-terminal symbols in the grammar. These symbols make interpretation of the chart difficult, and generate non-intuitive parsing trees that do not match the corresponding trees from the original grammar. Following [4] and [2], we consider a more interesting alternative: to adapt the algorithm to work directly with any CFG.

The new bottom-up parsing algorithm, which we call *extended CYK algorithm*, can also handle stochastic context-free grammars (SCFGs). Therefore, for a given sentence of length n words, it is possible to know whether it belongs to the grammar language or not, to calculate its total probability (not only the

* This work was partially supported by the European Union under FEDER project 1FD97-0047-C04-02, and by the Autonomous Government of Galicia under project PGIDT99XI10502B.

probability of each individual analysis), to obtain the N-most probable analyses of any portion of the input sequence in $\mathcal{O}(n^3)$ time complexity, whilst preserving the original structure of the grammar. Explicitly extracting all the parse trees associated with the input sentence or any substring of the input sentence is also possible. The cost of this step depends on the complexity of the grammar, but even in the case where the number of parse trees is exponential in n, the chart used by the algorithm for their representation remains of $\mathcal{O}(n^2)$ space complexity.

However, $\mathcal{O}(n^3)$ time complexity could be too high in some applications. This fact leads us to reflect on the building of a parallel version of the extended CYK algorithm. Section 2 gives the details of the standard steps of the algorithm, section 3 discusses the different alternatives of parallelization, section 4 shows the experimental results, and section 5 presents the conclusion after the analysis of the data obtained.

2 Extended CYK algorithm

In this section, we introduce the basic extended CYK algorithm, postponing stochastic considerations. Furthermore, although the algorithm is able to work with arbitrary CFGs, we restrict our representation to a subclass of CFGs consisting of non-partially-lexicalized rules. In such grammars, terminals can only appear in "lexical" rules of the type $A \rightarrow w_1\, w_2\, \ldots\, w_k$, where A is a non-terminal and w_i are terminals, i.e. words. Most often k is equal to 1, $k > 1$ corresponding to compound words or lexicalized expressions. This restriction is not however critical for the algorithm, and it is inserted solely to isolate lexical rules in an initial step, which in practice is performed by using a lexicon.

2.1 Non-stochastic approach

The basic data structure used by the algorithm is a CYK-like table containing generalization of Earley-like items, instead of the non-terminals used by the original algorithm. The parsing table or chart is a lower triangular matrix with $\frac{n(n+1)}{2}$ cells, where n is the number of words of the input sequence to be parsed. Each cell N_{ij} contains two list of items:

- Items of the first list, called type-1 list, represent the non-terminals that parse the subsequence $w_i, w_{i+1}, \ldots, w_{i+j-1}$, that is, if A is such a non-terminal, then $A \stackrel{*}{\Rightarrow} w_i w_{i+1} \ldots w_{i+j-1}$, and the corresponding item is denoted by $[A; i, j]$.
- Items of the second list, called type-2 list, represent partial parses α of the subsequence $w_i, w_{i+1}, \ldots, w_{i+j-1}$, i.e. sequences α of non-terminals such that $\alpha \stackrel{*}{\Rightarrow} w_i w_{i+1} \ldots w_{i+j-1}$ for which there exists at least one rule in the grammar of the form $A \rightarrow \alpha\beta$, where β is a non-empty sequence of non-terminal symbols ($\beta = \lambda$ is precisely the case taken into account by the items in the type-1 list). Such items are denoted by $[\alpha \bullet \ldots; i, j]$.

Notice that the type-2 items represent a generalization of dotted rules used in Earley-like parsers, where only the first part of the rule is represented (i.e. the part already parsed), independently both of the left-hand side (which has not yet been rewritten) and of the end (which has not yet been parsed). This provides a much more compact representation of dotted rules (allowed by the bottom-up nature of the parsing).

In addition, each item in any of the two lists is associated with the list of all its possible productions. This allows a further factorization that avoids the repetition of the item itself in the representation of all these possible productions, and obtains items only once, even if they are produced several times in the same cell, thus speeding up the parsing process. For parse tree extraction purposes, each production contains an explicit reference to the two items that were used to create it.

The extended CYK algorithm to determine whether a sentence $s = w_1, w_2, \ldots, w_n$ belongs to the language of a CFG, with no lexicalized rules, $G = (N, T, P, S)$, consists of the following steps:

1. **Initialization step**. This step consists of filling all the cells of the chart for which there exists a lexical rule associated with the corresponding word or sequences of words in the input sentence. That is, the type-1 lists of the first rows are filled. More precisely, if a rule $A \to w_i \ldots w_{i+j-1} \in P$, item $[A; i, j]$ is added to the type-1 list of cell N_{ij}.
   To be complete, the initialization step also needs a "self-filling step" updating type-2 lists of these cells. This phase is also used in the parsing step that we explain next.
2. **Parsing step**. This step consists of applying two phases: the standard filling phase (over all the cells except the ones involved in the initialization step), and the self-filling phase (over all cells N_{ij} in the chart). These phases are applied bottom-up row by row.
   - **Standard filling phase**. An item $[\alpha \bullet \ldots; i, k]$ in the list 2 of a cell is combined with an item $[B; i+k, j-k]$ in the list 1 of another cell, if there is a rule of the form $A \to \alpha B \beta$ in the grammar. If $\beta = \lambda$, item $[A; i, j]$ is added to the list 1 of cell N_{ij}. Otherwise, the item $[\alpha B \bullet \ldots; i, j]$ is added to the list 2 of cell N_{ij}.
     Notice that the item is added when it does not exist. If the item already exists, only the new production is added to the sublist of productions of the item.
   - **Self-filling phase**. In this procedure, for each item $[B; i, j]$ in the list 1 of the cell, and for each rule of the form $A \to B\beta$, we do the following:
     - If $\beta = \lambda$, item $[A; i, j]$ is added to the list 1 of the cell. Notice that the self-filling phase must be applied again over this new item.
     - If $\beta \neq \lambda$, item $[B \bullet \ldots; i, j]$ is added to the list 2 of the cell.

     This second step is necessary both to deal with chaining of unitary rules and to keep the type-2 lists up to date after the standard filling procedure has been performed.

Therefore, $s \in L(G)$ when item $[S; 1, n]$ is in cell N_{1n}.

2.2 λ-rules

If the grammar contains so-called λ-rules, i.e. rules of the form $A \to \lambda$, our basic algorithm needs to be completed as follows:

- Whenever a type-2 item $[\alpha \bullet \ldots; i, j]$ is produced, for each non-terminal B producing λ ($B \to \lambda \in P$) that could complete that item ($A \to \alpha B \beta \in P$), we do the following:
  - If $\beta = \lambda$, item $[A; i, j]$ is added to list 1 of the same cell.
  - If $\beta \neq \lambda$, item $[\alpha B \bullet \ldots; i, j]$ is added to list 2 of the same cell.

  This procedure must be performed recursively over the new items up to the point where no new item can be added.
- The self-filling phase also needs to be updated. For each type-1 item $[B; i, j]$, if there is a rule of the form $A \to CB\beta$, where $C \to \lambda$, then:
  - If $\beta = \lambda$, item $[A; i, j]$ is added to list 1 of the same cell.
  - If $\beta \neq \lambda$, item $[B \bullet \ldots; i, j]$ is added to list 2 of the same cell.

  That is, the self-filling step is performed as if the rule $A \to B\beta$ were in the grammar.

2.3 Extracting parse trees

Extraction of parse trees from the chart is performed simply by following the productions of those type-1 items of the cell in the top that contain the initial non-terminal, if such items exist. This procedure can also be applied to any other cell to extract subtrees.

Each production in these items is taken as a parse tree, and for each of these trees or productions of the item chosen as root, the following recursive procedure is performed: (1) The root node, labeled with the non-terminal in the item, is created. (2) For each reached node, a child node is created. If the node is a type-1 item, the node is labeled with the symbol in the item. Otherwise, the node has no label. (3) Nodes without labels are removed. The new father of its child nodes will be the father of this removed node. (4) Words in the input sentence are added.

2.4 Dealing with probabilities

For SCFGs, the probability of a parse tree is the product of the probability of the rule used to generate the subtrees of the root node, and the probabilities of each of these subtrees[1].

In the case of the extended CYK algorithm, we define the probability of a production p of a type-1 item $[A; i, j]$ as the probability of the subtree corresponding to the interpretation A of the sequence $w_i, \ldots, w_{i+j-1}$ for this production p, i.e.

[1] By using logarithmic probabilities, products can be replaced by sums. This speeds up the calculations and avoids problems of precision that arise in products with factors less than 1.

as the product of the probabilities of the items from which it is produced and the probability of the rule involved in this production.

In the same way, for type-2 items we define the probability of a production p of an item $[\alpha \bullet \ldots; i, j]$ as the product of the items which produce it. In this case, there is no rule probability, because type-2 items are partial interpretations.

It is then possible to define the maximum probability of an item as the maximum of the probabilities of all its productions (this avoids storing all those probabilities), taking into account that a production of an item is obtained from the combination of only two previously generated items (one of type-2 with another of type-1). This allows us to keep the N most probable parses during the parsing process, since the N biggest values for an item are included in the N^2 products obtained from the N-best probabilities of each of the constituting items. In cases where the N-best approach is not sufficient, it is of course always possible to recursively extract all the (sub)parses of the input (sub)sequence, the associated probabilities being calculated during extraction by the product of the probabilities of their constituents.

Moreover the probability of any (sub)sequence, defined as the sum of the probabilities of all its possible parses, is also computed during the parsing process simply by adding all the probabilities of each new production of an item, even if this probability is not kept in the N-best list.

3 Reflections on parallelism

Time complexity of the extended CYK algorithm, measured in terms of the size n of the input sentence, is clearly $\mathcal{O}(n^3)$. However, the multiplicative constant associated with n^3 in the complexity plays an important role. A more detailed complexity could be calculated for the worst case, in terms of the cardinals of non-terminal and rule sets, and we would find that it is lower than the one in other similar parsing algorithms. We will not detail this calculation here (it can be seen in [2]), but the intuitive explanation is that the gain compared with other Earley-like or CYK-like algorithms is because the extended CYK algorithm performs a better factorization of dotted rules (in type-2 list items), and processes fewer elements (items appear only once and there is no prediction).

Even with this, the complexity can be reduced as a consequence of the parallel nature of CYK-like algorithms: there are a lot of cells in the chart that can be calculated simultaneously. To parallelize the extended CYK algorithm [1], two alternatives have been considered: (1) distributed memory by passing messages between computers, and (2) shared memory, i.e. execution of several processes that concurrently access to the same area of memory. We explain both techniques in the following sections.

3.1 Distributed memory

With this technique, each computer calculates a fragment of the chart. That is, the chart is distributed among the memories of the computers. Every computer could calculate one or more chart columns, approaching parallelism at column level. The advantage of this distribution is that every processor always possesses one of the cells that it needs in order to calculate the next cell. The disadvantage is that a given processor does not help the others when it finishes with its columns. On the other hand, if we have more processors than words in the sentence, the remaining processors are idle.

Under this approach, the only thing to solve is how to distribute the chart columns among computers and how to interchange messages between processors. With regard to the distribution, the most reasonable solution is to perform a reverse cyclic distribution. If we have for instance 10 columns and 3 processors, columns 1, 6 and 7 are assigned to processor 1, columns 2, 5 and 8 are assigned to processor 2, and columns 3, 4, 9 and 10 are assigned to processor 3, as we can see in figure 1. In this way, we obtain a balanced overload for all the computers. The right-hand side of the figure shows the subcharts stored in the memory of each processor.

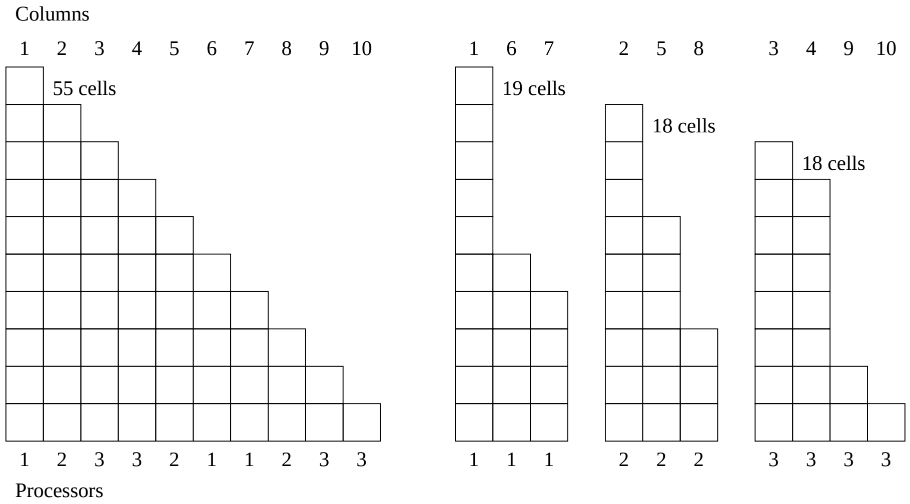

Fig. 1. Chart columns distribution in a parallelization of the extended CYK algorithm by distributed memory

With regard to the messages, if every processor sends every cell that has already been calculated to the processors that need the cell in the future, we eliminate bidirectional communications. There will be no request for cells among processors, only cell deliveries. Furthermore, cell deliveries are performed only once to each processor, so the target must determine whether the received cell will be necessary later, and send it back to itself in that case.

However, this approach gives priority to speed, not to reliability. By removing bidirectional communications, we increase the performance. Each processor looks for the cells it needs in its input queue. If it does not find a given cell it is because the processor responsible for that calculation is still working and has not yet sent the cell, causing a temporary delay. But processors could exhaust storage capacity. This problem is particularly serious when we use few processors and could lead to a deadlock in the system.

Even with this optimization in the interchange of messages, the main problem of the distributed memory approach is the granularity of the process that we are trying to parallelize. In our case, even without deadlocks and long delays, processors spend most of their time on sending and receiving data, not on the calculus of that data. The gain in performance may not be significant compared to the computational resources used, or the same process may even be executed faster in a sequential computer with only one processor. All these reflections lead us to consider the shared memory approach.

3.2 Shared memory

In this approach, all processors share the same chart and fill the cells simultaneously. Unlike the preceding case, cells are calculated from left to right and bottom-up, as the processors become free. In this way, no processor is prematurely free and their capabilities are better exploited. This approach is not viable with distributed memory due to the high cost of the synchronization of all processors.

A feature inherent to the shared memory paradigm is that there is no data transfer between processors. This is advantageous where the amount of information to share is great. However, we must take into account that processors have to access concurrently to the same areas of memory, so we must determine critical sections and give them adequate protection. Here, it is sufficient to define a variable associated with every cell indicating whether the cell is already calculated or not, and a semaphore associated with that variable.

The order of the loops that determine which cell is going to be processed is very important for the parser to perform properly, and must satisfy the requirements of this alternative. The external loop must run on the rows, and the internal one in the columns. In this way, deadlocks, i.e. situations where a processor needs a cell that is being calculated in another processor and vice-versa, do not occur. When a processor starts the calculation of a cell, the cells that it needs are already calculated or are being calculated in processors that are not going to need the cell of the first one, because it is overhead or at the same level. Therefore, this approach guarantees that no deadlock will be produced.

From the above, we can deduce that the extended CYK algorithm presents a natural scheme of parallelization. As we will see later, it is not suitable to be implemented on a set of computers with distributed memory because the penalization of communications damages its performance. However, the implementation on a multiprocessor with several threads provides excellent results.

4 Analysis of results

Experiments have been performed on a computer under the Linux operating system, with the following components:

- 1 main node with 2 Pentium II-350 MHz. processors, with 384 MB RAM (multiprocessor of shared memory).
- 23 additional AMD K6-300 MHz. nodes, with 96 MB RAM, forming a *switch fast ethernet* cluster (multicomputer of distributed memory).

Sequential and multithread versions of the algorithm have been executed in the main node. Distributed memory version has been executed in all nodes, even in the main one, and communication between processors has been implemented with MPI (Message Passing Interface) standard library.

Input data for evaluating the algorithm have been taken from SUSANNE treebank [7, 8]. This treebank was divided into two parts: the first part includes sentences without traces (4,292 sentences, 77,275 words), and the second one includes sentences with traces (2,188 sentences, 60,759 words). The first part has been used to extract the grammar (17,669 rules and 1,525 non-terminals) [5], and the second one to measure parsing times on the different versions of the algorithm.

The results in table 1 show times of sequential, 1 thread, 2 threads, 2 MPI, 4 MPI, etc. These descriptors correspond respectively to sequential version, multithread version with 1 thread (in order to infer the cost of initialization and management of semaphores), multithread version with 2 threads, and distributed memory versions with 2 processors, 4 processors, etc. Times appear in `mm:ss.mmm` format (minutes, seconds and milliseconds). User times for 2 threads are the sum of the execution time of both processors. Therefore, although real time is usually less reliable, it is more representative in this case.

Analysis of 2,188 sentences							
Times	Sequential	1 thread	2 thread	2 MPI	4 MPI	8 MPI	16 MPI
Real	03:45.259	03:52.648	02:24.311	10:47.108	09:28.020	07:57.424	07:40.391
User	03:44.810	03:50.580	03:53.040	05:08.900	03:55.730	02:36.370	02:07.190
System	00:00.036	00:00.530	00:01.370	05:16.200	05:26.370	05:17.350	05:28.990

Table 1. Execution times of extended CYK algorithm over the SUSANNE treebank

These data show that there is an additional cost due to the handling of semaphores, but in any case the multithread version with 2 threads is much faster than the sequential version. And what is more, the evaluated implementation considers parallelism at sentence level, and threads are created and destroyed for each sentence. An obvious optimization could be achieved simply by keeping the threads from sentence to sentence.

On the other hand, the distributed memory version seems to be the worst in any case, and this is due to three main reasons, some of them already mentioned. First, the data communication cost among processors is high when parsing short sentences. Second, when the sentence is long, distribution of data among the chart columns, and the corresponding overload in the processors, should be balanced, but this is not usual because charts are asymmetric and usually contain a high number of empty cells. Finally, when the sentence is complex, cells contain too much information, so communication between processors is very expensive (when we increase the number of processors, execution time decreases, but system time grows).

There is an obvious gain in the 2 threads version: a parsing time of 2:24.311 over 2,188 sentences yields an average of 0.065 seconds per sentence. But this result is not representative either. To show it, it is sufficient to look at table 2, in which we study independent execution times on 5 sentences. There are sentences that can be parsed in times much higher than the average, between 318 milliseconds and almost 2 minutes. It is interesting to note the behavior on sentence 5, where the versions with threads do not produce a significant improvement in performance, probably because the chart of this sentence contains a lot of empty cells. This in fact is the case because parsing time is really low for a 214 word sentence (23,005 cells), and the low number of cells with items leads to a high frequency of waiting periods in the processors. In other words, parallelism is better exploited when the sentence to parse is complex.

	Sentence 1	Sentence 2	Sentence 3	Sentence 4	Sentence 5
Words	30	35	55	150	214
Sequential times					
Real	00:00.351	00:04.646	02:40.821	00:00.750	00:22.268
User	00:00.340	00:04.630	02:40.600	00:00.740	00:22.160
System	00:00.010	00:00.020	00:00.220	00:00.010	00:00.100
2 threads times					
Real	00:00.318	00:02.735	01:57.759	00:00.623	00:21.004
User	00:00.460	00:04.810	02:39.800	00:00.950	00:23.480
System	00:00.001	00:00.020	00:00.140	00:00.030	00:00.200

Table 2. Independent execution times of extended CYK algorithm on 5 sentences

It is also evident that the complexity of a sentence is not proportional to its number of words. This is clear if we compare parsing times of sentences 3 and 4. Nevertheless, the improvement of the 2 threads version respect to the sequential remains present.

5 Conclusion

Parallelization of the extended CYK algorithm by using shared memory turns this algorithm into an alternative that could compete with other more efficient parsing techniques. Although those techniques could also be parallelized, this parallelization, when possible, is not intrinsic to the algorithm and leads to higher design and implementation complexities.

Finally, we should point out the advantages when we combine the stochastic paradigm with robust parsing techniques. These two features are inherent to the extended CYK algorithm, and are currently required in many natural language processing applications. The availability of efficient tagging and parsing tools able to deal with incomplete dictionaries and grammars, with the help of the stochastic framework, provides good prospects of immediate application on high level information processing systems, such as information retrieval, information extraction or question answering systems.

References

1. Barcala Rodríguez, M. (1999). Diseño e implementación de una herramienta de análisis sintáctico estocástico para el procesamiento de textos en lenguaje natural. *Proyecto Fin de Carrera en Ingeniería Informática*, dirigido por J. Graña Gil en el Departamento de Computación de la Universidad de A Coruña.
2. Chappelier, J.-C.; Rajman, M. (1998). A practical bottom-up algorithm for on-line parsing with stochastic context-free grammars. *Rapport Technique 98/284*, Departement d'Informatique, Ecole Polytechnique Fédérale de Lausanne (Suisse).
3. Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, vol. 13(2), pp. 94-102.
4. Erbach, G. (1994). Bottom-up Earley deduction. In *Proceedings of the 14th International Conference on Computational Linguistics (COLING-94)*, Kyoto (Japan).
5. Graña Gil, J.; Chappelier, J.-C.; Rajman, M. (1999). Using syntactic constraints in natural language disambiguation. *Rapport Technique 99/315*, Departement d'Informatique, Ecole Polytechnique Fédérale de Lausanne (Suisse).
6. Kasami, T. (1965). An efficient recognition and syntax analysis algorithm for context-free languages. *Technical Report*, AF CRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA.
7. Sampson, G. (1994a). The SUSANNE corpus, release 3, 04/04/1994. School of Cognitive & Computing Sciences, University of Sussex, Falmer, Brighton, England.
8. Sampson, G. (1994b). English for the computer. *Oxford University Press*.
9. Younger, D.H. (1967). Recognition of context-free languages in time n^3. *Information and Control*, vol. 10(2), pp. 189-208.