

Log Time Parsing on the MasPar MP-1*

Randall A. Helzerman and Mary P. Harper

helz@ecn.purdue.edu, harper@ecn.purdue.edu

School of Electrical Engineering

Purdue University

West Lafayette, IN 47907

Abstract

This paper describes the parallelization of Constraint Dependency Grammar (CDG) parsing. Though CDG provides a flexible framework for text-based and spoken language parsing and has an expressivity strictly greater than context-free grammars (CFGs), it also has a relatively slow serial running time (i.e., $O(n^4)$). However, a parallelization for this algorithm is derived which uses $O(n^4)$ processors to parse in $O(k)$ time for a CRCW P-RAM, where n is the number of words in the sentence and k , the number of constraints, is a grammatical constant. Additionally, the paper describes an implementation of the algorithm on the MasPar MP-1, which uses the special features of the machine (particularly the global router) and $O(n^4)$ processors to obtain an $O(k + \log(n))$ running time. Because the average length of an English sentence is on the order of 10 words, the MasPar MP-1 has sufficient processors (i.e., 16,000) for parsing a typical sentence.

Previous work in parallel parsing has focused on speeding up the parsing of context-free grammars (CFGs). For example, Kosaraju's method [6] using cellular automata can parse CFGs in $O(n)$ time using $O(n^2)$ processors. However, achieving CFG parsing times of less than $O(n)$ has required more powerful, less easily implementable models of parallel computation and significantly more processors than previously required. For example, Ruzzo's method [12] has a running time of $O(\log^2(n))$ using a CREW P-RAM (Parallel Random Access Machine) model, but requires $O(n^6)$ processors to achieve that time bound.

In this paper, a parallel parsing algorithm is developed based on a new grammatical formalism, Constraint Dependency Grammar (CDG), introduced by Maruyama [9; 7; 8]. CDG has an expressivity strictly greater than context-free grammars (CFGs). This formalism also provides an extremely flexible framework for text-based and spoken language understanding (See Allen [1] for a review of natural language parsers). Though CDG has a relatively slow serial running time (i.e., $O(n^4)$), the parsing algorithm is much more parallelizable than CFG parsing. In section 1, CDG is introduced and its advantages described. Section 2 describes the parallelization of the CDG parsing algorithm and its implementation on the MasPar MP-1. Because the MasPar architecture is so well suited to the algorithm, it leads to a simple and elegant implementation.

*This work is supported in part by Purdue Research Foundation and NSF grant number IRI-9011179.

1 CDG Parsing

1.1 Elements of a CDG Grammar

A CDG grammar consists of a 5-tuple, (Σ, L, R, T, C) , where:

Σ = set of terminal symbols

L = set of labels = $\{l_1, \dots, l_p\}$

R = set of roles = $\{r_1, \dots, r_q\}$

T = a table indicating which labels in L are legal for the role values in R

C = a set of k unary and binary constraints

To illustrate what Σ , L , R , T , and C mean, a simple CDG grammar is developed which accepts the sentence *The program runs*. We choose a sentence with only three words to simplify the figures and discussion.

Terminals (elements of Σ) are the parts of speech or categories of the words in a sentence. The *program* sentence contains a noun (*program*), a verb (*runs*), and a determiner (*a*). Therefore, the elements of Σ are: noun, verb, or det.

Labels (members of L) are representative of the functions that words play in a sentence. In our grammar, a noun receives the label SUBJ or NP, a main verb receives the label ROOT or S, and a determiner receives the label DET or BLANK. Therefore, elements of L are: SUBJ, NP, ROOT, S, DET, or BLANK.

To develop a syntactic analysis for the sentence using CDG, a **constraint network** (or CN) of words is created (see figure 1). Associated with each word node is a set of roles. **Roles** (members of R), stand for various syntactic functions of a word. At least two roles per word are required to parse a sentence, though more can be used as needed. We use two roles called **governor** and **needs**. The governor role (shown as **G** in figure 1) indicates the function a word fills when it is governed by a particular head word (e.g., a subject (SUBJ) is governed by the main verb (ROOT) in a sentence). The needs role (shown as **N** in figure 1) indicates what things a word needs to be complete (e.g., a singular count noun always needs a determiner). Initially, each word's roles take on a number of different **role values** corresponding to the functions the word's role can legally fill in the sentence.

A **role value** is a tuple consisting of a **label** and a **modifier** (see figure 1 for the sets of role values initially assigned to the governor and needs roles for each word). The label and modifier describe the func-

tion the word fills. For example, if SUBJ-3 is a role value in the governor role for the word *program*, then *program* has the label SUBJ and modifies *runs*, the third word in the sentence. The modifiee can also be a special symbol *nil*, which means that this role value modifies no word in the sentence. Typically, the main verb is not governed by any other word; this would be represented by the role value ROOT-nil.

The **table**, T , restricts the possible labels for each role¹. Though T is not a necessary component of the grammar, it does make the analysis of a sentence more efficient. In our example, the table (not pictured) would show that the allowable governor role labels are SUBJ, ROOT, and DET, and the allowable needs role labels are NP, S, and BLANK.

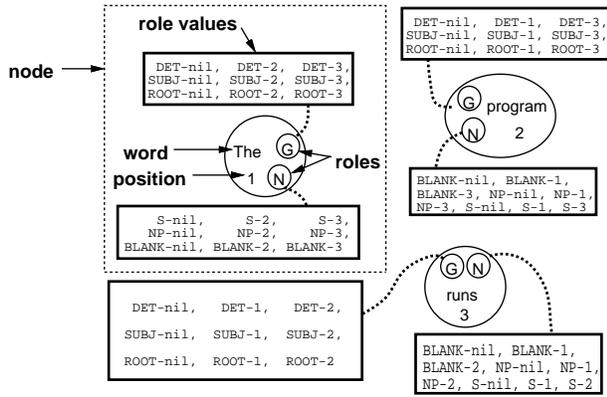


Figure 1. Nodes in a CN before the propagation of unary constraints.

1.2 CN Construction

Constraint Dependency Grammar provides a method of parsing which systematically determines which role value belongs in each role for each word in a sentence. Each of the n words in a sentence is represented as a node in a CN. Figure 1 illustrates the initial configuration of nodes in the CN for the *program* example. Each node contains the following information: the word with which it is associated, the possible parts of speech for that word (not shown in figure 1), the word's position in the sentence, and all of the roles in the set R . Initially, each role for every word contains an exhaustive list of all possible role values given the table T and the fact that no word ever modifies itself. There are $p * q * n = O(n)$ possible role values (where q , the number of roles per word, and p , the number of different labels, are grammatical constants and n is the number of modifiees or words) for each of the n words in the sentence. Therefore, there are $O(n * n) = O(n^2)$ role values altogether, which require $O(n^2)$ time to generate.

¹In our implementation, we also restrict labels by using word category information.

1.3 Constraints

To parse a sentence using CDG, **constraints** (members of C) are applied to the CN, eliminating ungrammatical *role values* from the *roles* of each word. A sentence is accepted by a CDG parser if and only if each role in each word node contains at least one role value which satisfies all of the constraints. Ideally, the process of applying constraints continues until each role for every word contains a single role value.

Constraints are of the form: (if *antecedent consequent*), where *antecedent* and *consequent* are either single predicates or a conjunction or disjunction of predicates. The variables in constraints stand for role values. A **unary constraint** contains only one variable, while a **binary constraint** contains two. One and two variable constraints allow for sufficient expressivity [7; 8] and more than two would unreasonably increase the running time of the parsing algorithm. Our constraints use the **access functions** and **predicates** defined below. Constraints may contain any access function or predicate, provided that it can be evaluated in constant time.

Access Functions:

(lab x) label for role value x
(mod x) modifiee for role value x
(role x) role for role value x
(pos x) word position for role value x
(word p) word at sentence position p
(cat w) part of speech for word w

Predicates:

(and p q) true if p and q are true, false otherwise
(or p q) true if p or q is true, false otherwise
(not p) true if p is false, false otherwise
(eq x y) true if $x = y$, false otherwise
(gt x y) true if $x > y$ and $x, y \in \text{Integers}$, false otherwise
(lt x y) true if $x < y$ and $x, y \in \text{Integers}$, false otherwise

Since all of the above functions either access local information, test for the truth of one or two predicates, or compare the ordinal relationship between two items, they can each be evaluated in constant time. Since all constraints contain a bounded number access functions and predicates, each constraint can also be evaluated in constant time.

Below are the syntactic unary and binary constraints required for the *program* example.

Unary Constraints:

Verbs have the label ROOT and are ungoverned.

```
(if (and (eq (cat (word (pos x))) verb)
         (eq (role x) governor))
    (and (eq (lab x) ROOT)
         (eq (mod x) nil)))
```

Verbs have the label S for the needs role and must modify something.

```
(if (and (eq (cat (word (pos x))) verb)
         (eq (role x) needs))
    (and (eq (lab x) S)
         (not (eq (mod x) nil))))
```

Nouns receive the label SUBJ for the governor role and must modify something.
 (if (and (eq (cat (word (pos x))) noun)
 (eq (role x) governor))
 (and (eq (lab x) SUBJ)
 (not (eq (mod x) nil))))

Nouns receive the label NP for the needs role and must modify something.
 (if (and (eq (cat (word (pos x))) noun)
 (eq (role x) needs))
 (and (eq (lab x) NP)
 (not (eq (mod x) nil))))

Determiners receive the label DET for the governor role and must modify something.
 (if (and (eq (cat (word (pos x))) det)
 (eq (role x) governor))
 (and (eq (lab x) DET)
 (not (eq (mod x) nil))))

Determiners receive the label BLANK for the needs role and modify nothing.
 (if (and (eq (cat (word (pos x))) det)
 (eq (role x) needs))
 (and (eq (lab x) BLANK)
 (eq (mod x) nil))))

Binary Constraints:

A SUBJ is governed by a ROOT to its right.
 (if (and (eq (lab x) SUBJ)
 (eq (lab y) ROOT))
 (and (eq (mod x) (pos y))
 (lt (pos x) (pos y))))

A verb with label S needs a SUBJ to its left.
 (if (and (eq (lab x) S)
 (eq (lab y) SUBJ))
 (and (eq (mod x) (pos y))
 (gt (pos x) (pos y))))

A DET must be governed by a noun to its right.
 (if (and (eq (lab x) DET)
 (eq (cat (word (pos y))) noun))
 (and (eq (mod x) (pos y))
 (lt (pos x) (pos y))))

A noun with label NP needs a DET to its left.
 (if (and (eq (lab x) NP)
 (eq (lab x) DET))
 (and (eq (mod x) (pos y))
 (gt (pos x) (pos y))))

1.4 Constraint Propagation Parsing

CDG parsing consists of the following steps: unary constraint propagation, binary constraint propagation, consistency maintenance, and filtering. We describe each step in detail below.

Unary constraints (i.e., they only access one role value) are applied to all role values for each word in a sentence to eliminate impossible role values. Consider the constraint network in figure 1. Because impossible labels for the roles are eliminated by using the table T, the primary function of the unary constraints is to eliminate role values which are incompatible with the category of a word or have an incompatible modifiee. The first unary constraint ensures the governor role of a verb in the sentence contains only the role value ROOT-nil. To apply this constraint to the network in figure 1, each role value for every role is examined to ensure that it obeys the constraint. A role value violates a constraint if and only if it causes the an-

tecedent of the constraint to evaluate to TRUE and the consequent to evaluate to FALSE. A role value which violates a unary constraint is eliminated from its role.

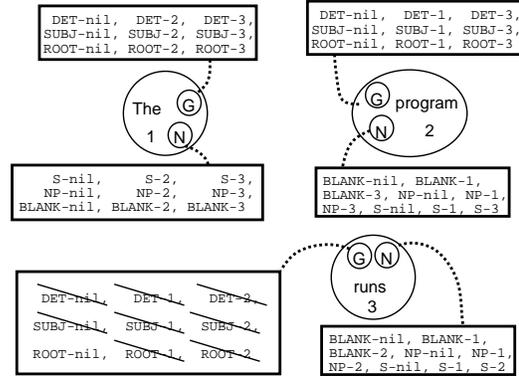


Figure 2. The CN after the propagation of the first unary constraint.

Figure 2 shows the remaining role values after the first unary constraint is applied. Note that the label ROOT-nil is the only remaining label for the governor role of *runs*. Because a unary constraint can be tested against one role value in constant time and there are $O(n^2)$ role values to check, the time to apply (i.e., propagate) a single unary constraint is $O(n^2)$. Initially, many unary constraints are applied to the CN to reduce the number of legal role values, requiring $O(k_u * n^2)$ time, where k_u is a constant for the number of unary constraints. Figure 3 depicts the CN after the remaining unary constraints are propagated.

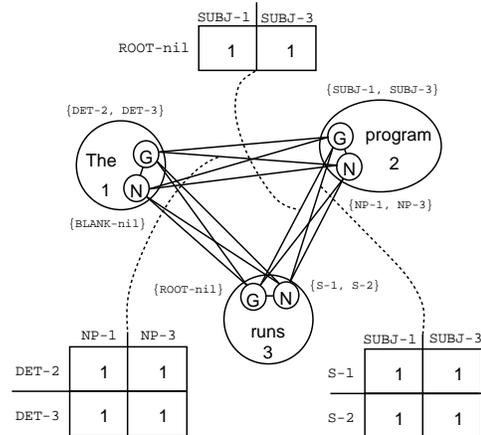


Figure 3. The CN after unary constraint propagation and before binary constraint propagation.

The binary constraints determine for pairs of roles which role values can legally coexist. To keep track of which pairs of role values can coexist given the binary constraints, **arcs** connect the roles associated with each node to all other roles in the network. Each of the arcs has associated with it an **arc matrix**, whose row and column indices are the role values associated with the two roles. The elements of the arc matrix

ces can hold either a **1** (indicating that the two role values which index it can legally coexist) or a **0** (indicating that either one or the other role value can exist, but not simultaneously). Initially, all entries in the matrices are set to **1**, indicating that there is nothing about one word's function which prohibits another word's right to have a certain function in the sentence. An illegal combination of role values is indicated by setting the entry in the matrix indexed by the corresponding role values to **0**. Since there are $\binom{q*n}{2} = O(n^2)$ arcs required in the CN and each arc contains a matrix with $O(n^2)$ elements, the time to construct the arcs and initialize the matrices is $O(n^4)$. Figure 3 shows several of the matrices associated with the arcs before any binary constraints are propagated.

The first binary constraint in the grammar ensures that a **SUBJ** is governed by a **ROOT**. After the application of this constraint to the network, the matrix on the arc connecting the governor roles for *program* and *runs* will have zero in the element indexed by the role values **ROOT-nil** and **SUBJ-1**. This is because *program* must be governed by the verb *runs*. Since it must be applied to $O(n^4)$ pairs of role values, the time to apply the constraint is $O(n^4)$. Figure 4 shows the arc matrices after the binary constraint is propagated.

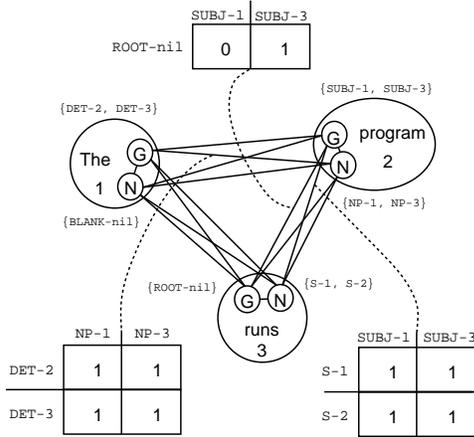


Figure 4. The CN after the first binary constraint is propagated.

To determine whether a role value is still supported for a role after a binary constraint is propagated, each of the matrices on the arcs incident to the role must be checked to ensure that the row (or column) indexed by the role value contains at least one **1**. If any arc matrix contains a row (column) of all **0**s for the role value, then that role value cannot coexist with any of the role values for the other roles and so must be removed from the list of legal role values for the role. Additionally, the rows (columns) associated with the eliminated role value must be removed from the arc matrices attached to the role. This step, called **consistency maintenance**, ensures that the CN is locally consistent following constraint propagation. The matrix associated with the governor roles of *program* and *runs* in figure 4

contains a column with all zeros since the label **SUBJ-1** cannot coexist with the label **ROOT-nil** in the governor role. Since **ROOT-nil** is the only possible role value for the governor role associated with *runs*, that means that **SUBJ-1** should be removed from the set of possible role values for *program*. Furthermore, it is also removed from the matrices associated with all arcs emanating from *program*'s governor roles. The CN after consistency maintenance is shown in figure 5. The time to perform consistency maintenance for a single unsupported role value is $O(n^2)$ (because there are $O(n)$ elements in a row (column) of a matrix to zero in $O(n)$ arc matrices when a role value is disallowed). After a binary constraint is propagated, each of the $O(n^2)$ role values may require one consistency maintenance step, requiring $O(n^4)$ time.

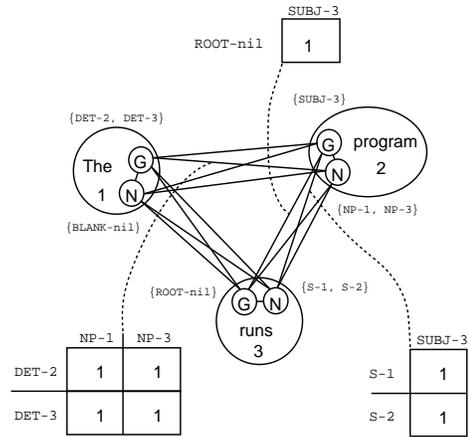


Figure 5. The CN after the first binary constraint and consistency maintenance.

After applying the binary constraint and making the network locally consistent, the CN in figure 5 is still ambiguous (with multiple role values for the governor role of *the* and for the needs roles of *program* and *runs*). Additional binary constraints can be applied to eliminate this ambiguity. The time required to propagate a set of binary constraints and make the network locally consistent is $O(k_b * n^4)$, where k_b is a grammatical constant for the number of binary constraints.

Figure 6 depicts the constraint network following the propagation of the remaining binary constraints and consistency maintenance. The modifiers of the remaining role values (which point to the words they modify) form the edges of the parse trees for the sentence. The parse trees in CDG are **precedence graphs**. The precedence graph for the *program* example is shown in figure 7. When a CN is syntactically ambiguous, there will be more than one precedence graph for the sentence. In the case of ambiguity, the precedence graphs are extracted by selecting a single role value for each role, all of which must be consistent given the arc matrices. Syntactic ambiguity is easy to spot in CDG; some of the roles in an ambigu-

ous sentence will contain more than one role value. In the case of an ambiguous CN, we can choose to propagate additional constraints or use a backtracking search to enumerate the parse graphs for the CN. Ideally, precedence graphs should only be extracted when the parse of a sentence is unambiguous. Since CNs compactly store multiple parses and such ambiguity is easy to detect, additional constraints can be applied as needed to further refine the analysis of an ambiguous sentence.

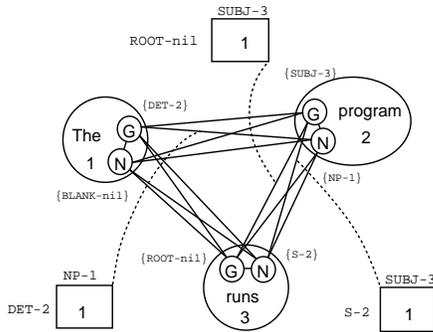


Figure 6. The CN after applying the remaining binary constraints and consistency maintenance.

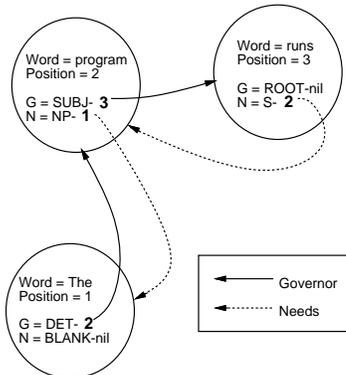


Figure 7. The precedence graph for *The program runs*.

Following the propagation of constraints and consistency maintenance, the CN could contain role values that would never be a member of a parse graph for the sentence. These role values can be eliminated by iteratively applying consistency maintenance until there are no role values whose matrices contain rows or columns with all zeros; a process called **filtering**. A single application of consistency maintenance may be insufficient to eliminate illegal role values since the elimination of a role value from one role could lead to the elimination of a role value from another role. Filtering continues until there are no role values indexing matrix rows or columns containing only zeros, which can take up to $O(n^4)$ time (see [7]). Though filtering is an optional part of the parsing algorithm, there is good reason to filter in the sequential model. Filtering has the potential to reduce the search time for parse graphs without increasing the asymptotic sequential

running time of the algorithm.

In summary, CDG parsing requires $O(k*n^4)$ time to parse a sentence with $k = k_u + k_b$ constraints. Though k is a factor in our time bound, the parse for a sentence can often be determined after only a portion of the constraints have been propagated.

1.5 Advantages of CDG

A CDG parser has many advantages over traditional CFG parsers. The set of languages which can be expressed by CDG is a superset of the set of languages which can be parsed by CFGs. In fact, Maruyama [7; 8] is able to construct a CDG grammar with two roles and two variable constraints which accepts the same language as any arbitrary CFG. Also, CDG can accept languages that CFGs cannot, for example, **ww** (where **w** is some string of terminal symbols).

In CDG parsing, any type of rule that can be formulated as an if-then rule containing one or two role value variables can be used to constrain a CN. Hence, rather than coordinating lexical, prosodic, syntactic, semantic, and contextual modules to develop the meaning of a sentence, the rules normally contained in such modules can be applied to the CN. This property allows decisions about structural ambiguities to be postponed until the constraints settle on a single structure, eliminating the need for backtracking. We have used this capability to incorporate prosodic information during parsing in a first step towards developing a spoken language understanding system (see [4]).

In CDG parsing, if a constraint applies to a word, it does not matter where in the sentence the word is (unless the constraint needs to relate the order of two words in the sentence). Hence, there is no notion of left-to-right parsing, which could be useful for building a spoken language understanding system, which must be able to tolerate repeated and aborted phrases. By using CDG's flexibility, combined with prosodic clues to spot errors, we should be able to develop a model which tolerates the typical grammatical errors of spoken English better than other approaches.

CDG allows us to easily add a contextual dimension by applying different sets of constraints in different situations. We are currently developing a core set of constraints (i.e., they apply in all situations), which are the first constraints to propagate, followed by other contextually-determined constraint sets. This flexibility is an advantage of CDG over traditional CFG parsers, which use a single set of rules to parse all sentences.

Though CDG parsing on a sequential machine is less tractable than CFG bottom-up parsers, which require $O(n^3)$ time, it can be efficiently parallelized. This parallelization will enable us to use CDG parsing in applications such as natural language data base interfaces or speech understanding systems for which fast response times are critical.

2 Parallelization of CDG Parsing

In this section, we describe PARSEC², a parallelization of the CDG constraint parser. The CDG algorithm is first adapted to the CRCW P-RAM model and then is simulated on a MasPar MP-1.

2.1 CDG Parsing on a CRCW P-RAM

For deriving absolute minimum time complexities, we first use the CRCW P-RAM (Common Read, Common Write Parallel Random Access Machine) model of parallel computation [3; 13; 2], which allows any number of processors to read from or write to any memory location. We assume that if more than one processor tries to write to the same location in memory, then a single random processor will succeed. This allows us to OR or AND any number of bits in constant time with a large enough number of processors [3]. To determine the time and processor complexity of the parallel algorithm, we consider the generation of role values, the propagation of constraints, consistency maintenance, and filtering.

Each role value, being nothing more than a label-modifier pair, can be generated independently of all others. The number of role values generated for each word's roles will be $n * p$ (where p is a grammatical constant for the number of labels) because each of the n words in a sentence can be modified by role values with each of the p labels. Hence, all the role values can be generated in constant time with $O(n^2)$ processors.

The propagation of constraints is a very local computation. To apply unary constraints, a processor requires access to information from one role value only, and for the binary constraints, only two role values need to be checked. Because of the shared-memory feature of the CRCW P-RAM, this information is immediately available to any processor that needs it. Furthermore, the checking of one role value or pair of role values is independent of the checking of other role values or pairs of role values, and hence all the checking can go on in parallel. Because there are $O(n^2)$ role values to test against the unary constraints, they can be propagated in constant time with $O(n^2)$ processors. Since there are $\binom{q*n}{2} = O(n^2)$ arcs in the constraint graph, each of which keeps track of the validity of $O(n^2)$ role values, a binary constraint must be checked against $O(n^2 * n^2) = O(n^4)$ pairs of role values. Given that k , the number of unary and binary constraints, is a grammatical constant and that each pair of role values can be checked concurrently, all of the constraints can be propagated in constant time using $O(n^4)$ processors.

Recall that one step of consistency maintenance involves removing unsupported role values from their

roles. A role value is still supported by the arc matrices associated with the role if each of the rows (or columns) indexed by the role value contain at least one **1**. We can determine this by logically ORing the elements contained in all the rows and columns indexed by the role value and then logically ANDing the results of those operations. If the result is **1**, then the role value is still supported. Since logical AND and OR operations on a CRCW P-RAM can be done in constant time [3], this requires only constant time. After removing a role value from a role, the algorithm must zero out all of the entries in the row (or column) which the role value indexes. Because there are $O(n)$ arcs connected to each role, and a column contains $O(n)$ entries, this step requires $O(n^2)$ work and can be performed in constant time with $O(n^2)$ processors (all entries can be zeroed simultaneously). Since there are $O(n^2)$ role values to check and each is checked independently of the others, with $O(n^4)$ processors, consistency maintenance can be completed in constant time.

We can only have limited success parallelizing filtering because one deleted role value can enable the deletion of other role values, resulting in a cascade of role value elimination. In the worst case, $O(n^2)$ role values would have to be sequentially eliminated, resulting in a running time of $O(n^2)$ for filtering³. However, this worst case is virtually eliminated by a number of factors. First, by lexically restricting role values when the CN is created and applying unary constraints, most of the role values associated with a role are directly eliminated. Second, though the remaining role values must be eliminated by using binary constraints, consistency maintenance, and filtering, filtering is postponed until after binary constraint propagation and consistency maintenance have further reduced the number of role values. Third, we have developed a variety of grammars for English, and have found that very few filtering steps (typically fewer than 10) are required at the end of constraint propagation. Given these factors, we can reasonably limit the number of iterations of filtering to a constant and because the other steps in the algorithm can be performed in constant time, the total running time of the algorithm is $O(k)$ with $O(n^4)$ processors, where k is the number of constraints propagated. However, if full filtering is required, then the running time can be $O(n^2)$.

The CRCW P-RAM, while useful in theoretical and conceptual domains, is a very unrealistic model, especially for a large number of processors. Fortunately, if we are willing to make minor concessions on time complexity, it is possible to implement the algorithm using more realistic architectures. Figure 8 compares the running times of CDG parsing with the running times of CFG parsing on a variety of architectures

²PARSEC stands for *Parallel ARchitecture SEntence Constrainer*.

³We have constructed an NC-reduction from the Monotone Circuit Value Problem to the filtering algorithm [5].

[4]. Note that k is the number of productions in CFG parsing and the number of constraints in CDG parsing.

In the following sections, we describe how the features of MasPar MP1 enable us to parse a sentence in $O(k + \log(n))$ time.

Architecture	CFG Parsing		CDG Parsing	
	# of PEs	Running Time	# of PEs	Running Time
Sequential Machine	1	$O(k^3 * n^3)$	1	$O(k * n^4)$
CRCW P-RAM	$O(n^6)$	$O(\log^2 n)$	$O(n^4)$	$O(k)$
2D Mesh	$O(n^2)$	$O(k * n)$	$O(n^2)$	$O(k + n^2)$
2D Cellular Automata	$O(n^2)$	$O(k * n)$	$O(n^2)$	$O(k + n^2)$
Tree and Hypercube	---	---	$O\left(\frac{n^4}{\log(n)}\right)$	$O(k + \log n)$

Figure 8. CDG and CFG parsing algorithms compared.

2.2 CDG Parsing on The MasPar

Because in CDG parsing the same constraint is applied to every role value or every pair of role values, the algorithm lends itself to implementation on a SIMD machine. Also, PE communication demands are small, so it can also be efficiently implemented on a distributed-memory system.

The MasPar MP-1 is such a system: a massively parallel SIMD computer, which supports up to 16K 4-bit processing elements (PEs), each with 16KB of local memory. Because typical English sentences only have on the order of 10 words, the MP-1 has a sufficient number of processors for this algorithm. The ACU (Array Control Unit) broadcasts instructions and data to the PE array. The MasPar also has a powerful global router which implements the scanAnd() and scanOr() primitives, which allow logarithmic-time ANDing and ORing of data values stored in the PEs [10].

The language in which our algorithm is implemented is MPL (*MasPar Language*) [11], an extension of C which supports the SIMD parallelism of the MasPar. The programming environment of the MasPar deserves special mention. The data visualization capabilities and the well integrated and extensive debugging support made the job of implementing the algorithm much easier.

MPL allows the programmer to view the PEs in two ways, either as a two dimensional grid of $128 * 128$ processors, or as a linear array of 16K processors. We choose to view the processors as a linear array. Because the MasPar has only 16K processors, one pro-

cessor may have to do the work of many to parse longer sentences. Unfortunately, MPL does not support transparent processor virtualization.

2.2.1 Design Decisions

We made six major design decisions in adapting our CRCW P-RAM algorithm for the MP-1. First, to simplify several computations in later stages of the algorithm, we construct the arc matrices before the propagation of the unary constraints. This implies that we need not propagate the unary constraints before the binary constraints. Figure 9 depicts the matrix which connects the governor role of *runs* with the governor role of *program*.

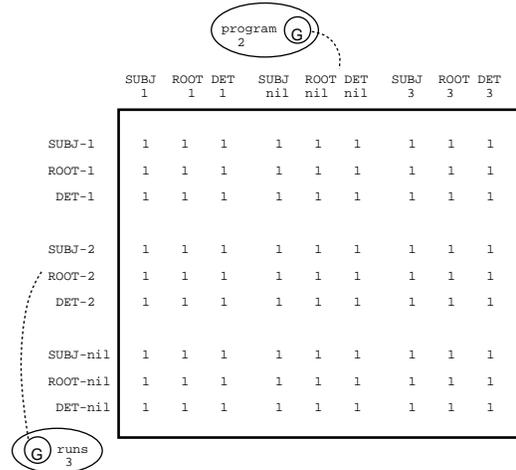


Figure 9. The arc connecting the governor role of *the* with the governor role of *program*.

Notice that **all** of the possible role values are present on the arc matrix in figure 9 because the unary constraints have not been propagated (in contrast to figure 3 after the propagation of the unary constraints). Second, we eliminate the need for shared memory: each PE gets the data it needs either by locally computing it, or by receiving it by global broadcast. Third, instead of ANDing and ORing in constant time, the scanAnd() and scanOr() primitives are used to do the global ANDs and ORs in logarithmic time. Fourth, for consistency maintenance, if a column or row of elements supporting a certain role value for a role contains only 0s, then the rows or columns indexed by that label are zeroed for all of the matrices associated with arcs emanating from that role rather than reducing their dimensions. Fifth, we allow only a constant number of iterations of consistency maintenance during filtering. Sixth, the PEs are virtualized such that each physical PE emulates a constant number of virtual PEs.

2.2.2 Mapping Arc Elements to PEs

The arc elements must be allocated to the PEs such that the scanOr() and scanAnd() functions can perform one iteration of consistency maintenance in logarithmic time. Figure 10 shows the arc matrices which

are connected to the governor role of the word *program* in the example sentence and reflects the same state in the parse of the sentence as in Figure 4. This figure illustrates which bits must be Ored and which must be ANDed to determine whether the role value SUBJ-1 is grammatical for the sentence.

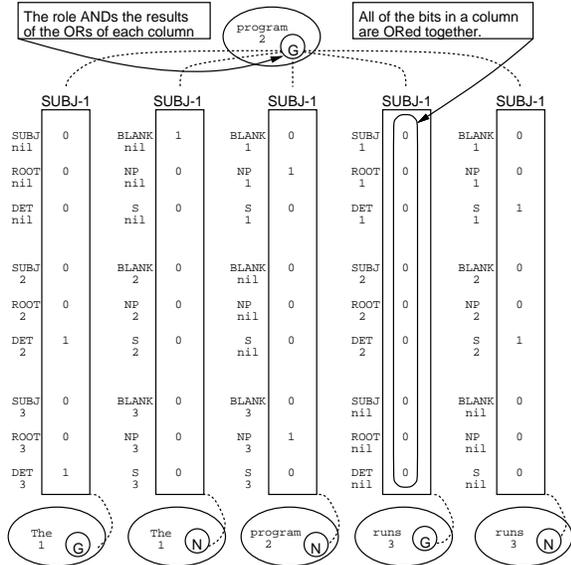


Figure 10. How arc elements are Ored and the results ANDed by the role.

All of arc elements in each of the columns of the arc matrices indexed by SUBJ-1 must be Ored together to form a result. Then, all of those results are ANDed. If the result of the final AND is 1, then SUBJ-1 is legal, but if the result is 0, then SUBJ-1 is eliminated as a role value for the role. Because all of the arc elements in a row of the matrix must be Ored with scanOr(), the processors for those arc elements must be placed in the same segment. There is no need to broadcast to each PE which arc elements it should process, because each PE can calculate that from its processor ID number. Similarly, the results of those scanOr(s) must end up in the same segment so that they can be ANDed with scanAnd(). One way of laying out the processors consistent with the above criteria is shown in figure 11.

Consider processor number 9 in figure 11. It is responsible for processing several elements in an arc matrix. Recall that each element in an arc matrix is indexed by 2 role values, one for its row, and one for its column. The column role values for processor 9 belong to the word *the* (because its processor ID is less than 107), the role for the column role values is governor, and their modifiee value is nil. The row role values' word is *program* and their role is needs. Notice in the figure that processors 0, 1, and 2 are disabled. This is because they represent an arc from a role to itself.

The ACU broadcasts to the PEs the instructions

necessary to evaluate the constraints, thus the total amount of time to propagate the k constraints is $O(k)$. After the constraints have been propagated, the ORing and ANDing operations, described earlier, are carried out, as illustrated in figure 12. This demonstrates how the consistency maintenance is performed on the MasPar (compare with the consistency maintenance step transforming the CN in figure 4 into the CN in figure 5). The results of the OR are stored in the highlighted PEs in figure 12 and then those values are ANDed and deposited in the highlighted processor at the bottom of the figure. If the result of the AND operation is 1, then SUBJ-1 is allowed, otherwise it is disallowed as a role value for the role.

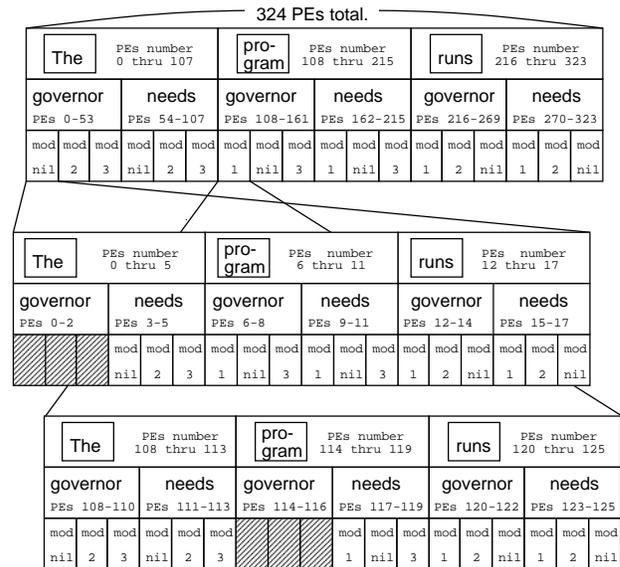


Figure 11. The PE allocation for the sentence *The program runs*.

2.2.3 Processor Virtualization

We choose to virtualize the PEs as depicted in figure 13, which shows the arc elements processed by each PE. Notice that each of the labels that the PE is responsible for has the same word, role, and modifiee for both its row and column role values. This lets us use the scanOr() and scanAnd() functions as previously described. However, the functions must be repeated 3 times, once for each of the labels allowed in the role. Because l (i.e., the number of labels in the grammar) is a constant, each processor will always simulate a constant number of processors. Therefore, a more complete upper bound for the running time would be $O(l^2 * k + l * \log(n))$, which simplifies to $O(k + \log(n))$, as l is a small grammatical constant.

3 Results and Conclusions

Time trials indicate that it takes less than 10 milliseconds to propagate a constraint in a network of one to seven words. The execution time of the algorithm on the MasPar as a function of the number of words in the

sentence is $\lceil \frac{4n^4}{2^{14}} \rceil * .15 \text{ sec}$. Hence, the total time for the MasPar to parse the example sentence is approximately 0.15 seconds, but the processing time required for a sentence of 10 words (because of processor virtualization) is .45 seconds. The graph of the parsing time as a function of the number of words in the sentence would look like a discrete step function which grows as n^4 . The corresponding times for our serial implementation (running on a Sun Sparcstation 1) is 15 seconds to apply a single constraint and 3 minutes to parse a sentence of 7 words.

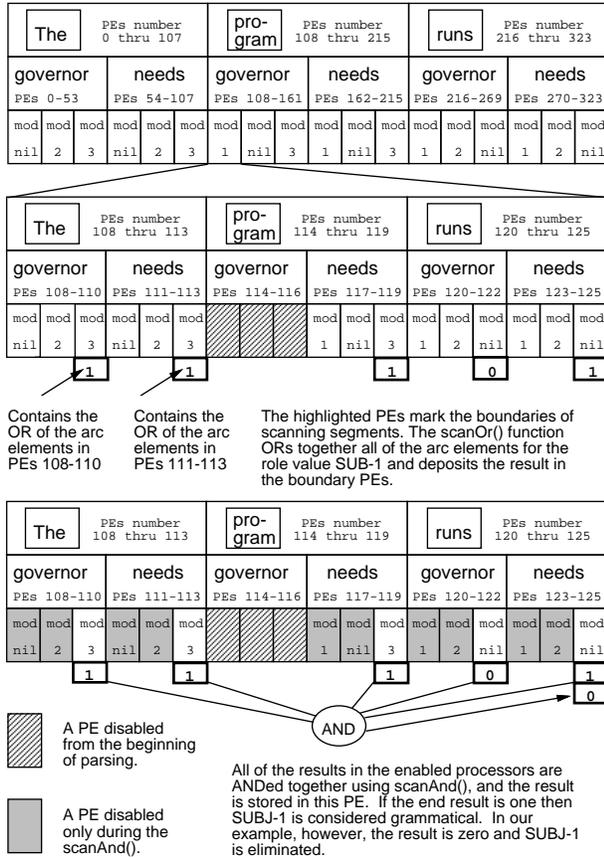


Figure 12. How scanOr and scanAnd cooperate to check the role value SUBJ-1.

The MP-1 provides an almost ideal test bed for the development and testing of CDG grammars and parsers. The scanOr() and scanAnd() functions, as well as the number of processors, allow us to reach a $O(\log(n))$ parsing time. Because of the power of the MasPar and the parallel nature of the algorithm, we will not have to sacrifice the flexibility and expressivity of CDG grammars when doing natural language parsing in a speech understanding system. Because natural language parsing can be done quickly and efficiently on commercially available parallel machines, it will not be a bottleneck for real-time systems.

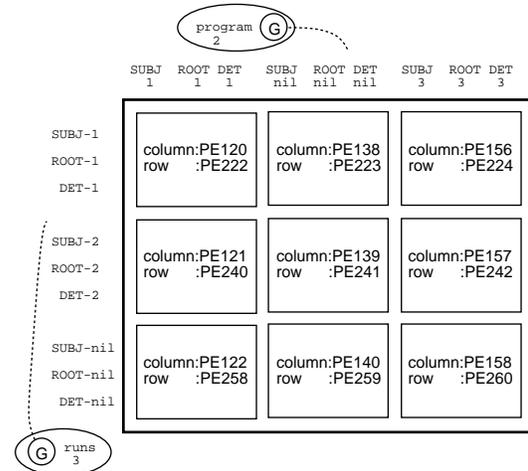


Figure 13. Each PE processes a 3 * 3 element submatrix.

References

- [1] J. Allen. *Natural Language Understanding*. The Benjamin/Cummings Publishing Company, Menlo Park, CA, 1987.
- [2] S. Fortune and J. Wyllie. Parallelism in random access machines. *Symposium on the Theory of Computation*, pages 114-118, 1978.
- [3] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, Cambridge, England, 1988.
- [4] M.P. Harper, R.A. Helzerman, and C.B. Zoltowski. Constraint parsing: A powerful framework for text-based and spoken language processing. Technical Report EE-91-34, Purdue University, School of Electrical Engineering, West Lafayette, IN, 1991.
- [5] R. Helzerman and M.P. Harper. PARSEC: Algorithms + Architectures = Parallel Parsing. June 1991.
- [6] S.R. Kosaraju. Speed of recognition of context-free languages by array automata. *SIAM Journal of Computing*, 4(3):331-340, September 1975.
- [7] H. Maruyama. Constraint dependency grammar. Technical Report #RT0044, IBM, Tokyo, Japan, 1990.
- [8] H. Maruyama. Constraint dependency grammar and its weak generative capacity. *Computer Software*, 1990.
- [9] H. Maruyama. Structural disambiguation with constraint propagation. In *The Proceedings of the Annual Meeting of ACL*, 1990.
- [10] MasPar Corp. *MasPar System Overview*, July, 1990. PN 9300-0100-2790.
- [11] MasPar Corp. *MasPar Parallel Application Language MPL Reference Manual*, March, 1991. PN 9302-0000.
- [12] W. Ruzzo. Tree-size bounded alternation. *Journal of Computers and System Sciences*, 21:218-235, 1980.
- [13] J. von zur Gathen. Parallel arithmetic computations: A survey. *Proceedings of the 12th Symposium on Mathematical Foundations of Computer Science*, 1986.