1988

Parallel Parsing in a Multiprocessor Environment

Dilip Sarkar
University of Central Florida

PARALLEL PARSING
IN A MULTIPROCESSOR ENVIRONMENT

by

DILIP SARKAR

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
the Department of Computer Science at
the University of Central Florida
Orlando, Florida

May 1988

Major Professor: Narsingh Deo

UNIVERSITY OF CENTRAL FLORIDA

OFFICE OF GRADUATE STUDIES

DISSERTATION APPROVAL

DATE: April 4, 1988

BASED ON THE CANDIDATE'S SUCCESSFUL ORAL DEFENSE, IT IS RECOMMENDED THAT THE DISSERTATION PREPARED BY Dilip Sarkar

ENTITLED Parallel Parsing in a Multiprocessor Environment

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGHREE OF Doctor of Philosophy

FROM THE DEPARTMENT OF Computer Science

IN THE COLLEGE OF Arts and Sciences

Narsingh Deo

Narsingh Deo, Major Professor
Department of Computer Science

Larry K. Cotrell

Larry K. Cotrell, Graduate Coordinator
Department of Computer Science

Jack B. Rollins

Jack B. Rollins, Dean
College of Arts and Sciences

Louis M. Trefonas

Louis M. Trefonas
Dean of Graduate Studies

PARALLEL PARSING

IN A MULTIPROCESSOR ENVIRONMENT

Dilip Sarkar
University of Central Florida
Orlando, FL 32816, May 1988

Major Professor: Narsingh Deo

ABSTRACT

Parsing in a multiprocessor environment is considered. Two models for asynchronous bottom-up parallel parsing are presented. A method for estimating speedup in asynchronous bottom-up parallel parsing is developed, and it is used to estimate speedup obtainable by bottom-up parallel parsing of Pascal-like languages. It is found that bottom-up parallel parsing algorithms can attain a maximum speedup of $O(L^{1/2})$ with $(L^{1/2})$ processors, where L is the number of tokens in the string being parsed. Hence, bottom-up parallel parsing technique does not yield good speedup.

A new parsing technique is proposed for parsing a class of block-structured languages. The novelty of the technique is that it is inherently parallel. By applying this new technique, a string of L tokens can be parsed in $O(\log L)$ time with $(L/\log L)$ processors. The parsing algorithm uses a parenthesis-matching algorithm developed here. The parenthesis-matching algorithm can find matching of a sequence of parentheses in $O(\log L)$ time with $(L/\log L)$ processors. Thus, the new parsing algorithm is cost optimal.

In memory of my father, and to my brother and mother.

ACKNOWLEDGEMENTS

I extend my sincere gratitude to my thesis advisor, Dr. Narsingh Deo, for his advice and support during my dissertation research and graduate studies at the Washington State University and at the University of Central Florida. I greatly appreciate his friendship, help and cooperation.

I am also grateful to Drs. Amar Mukherjee, Ronald D. Dutton, Brian E. Petrasko and Ratan K. Guha for their guidance during my stay at the University of Central Florida. Special thanks are also reserved for Sajal K. Das, N. Ranganathan and Sushil K. Prasad.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

A	The set of arcs in a digraph
A_i	An arc in a digraph
B	(With or without an index) a nonterminal symbol
CFG	Context-free grammar
D	A digraph
D_G	The digraph of a grammar G
DSP_A	The denominator of $SP_A(L, q)$
G	A grammar
G_B	The W-subgrammar for a nonterminal B and a subset W of V_N
H	The height of a parse tree
L	The length of the string of tokens being parsed
N	The total number of nodes in a parse tree
N_C	The number of nodes in the levels 1 through h in a parse tree
P	The set of production rules
P_i	A processor whose index is i
Q	The set of states
R	The start symbol
SE	A sequence of balanced parentheses

$SP(L, q)$	The speedup obtained by parsing a string of length L using q processors
$SP_A(L, q)$	Speedup obtained on the Model A
$SP_B(L, q)$	Speedup obtained on the Model B
$SP(q)$	Speedup obtained by using q processors
STK_i	The stack of the processor P_i
T_{cq}	The total parallel time for processor coordination and communication
T_{pq}	The total parallel time for parsing
T_{rq}	The total parallel time for reduce operations
T_{sq}	The total parallel time for shift operations
V	The set of all symbols in a grammar
V^*	The set of all strings from V
V_N	The set of nonterminals
V_T	The set of terminals
W	A subset of V_N
W_a	A subset of the set of nodes of a digraph that makes it acyclic
W_L	Set of all minimal subset of M that makes D acyclic
W_l	A minimal subset of the set of nodes of a digraph that makes it acyclic
$W_{\min}$	A minimum cardinality set in W_L
W_r	The set of recursive nonterminals
W_t	The set of recursive terminals

X	(With or without an index) a nonterminal symbol
Y	(With or without an index) a nonterminal symbol
a	(With or without an index) a terminal symbol
c_1	An integer
d	Average degree of the nodes in a parse tree
h	Critical height of a parse tree
i	Index
j	Index
k	Index
l	Index
$l(i)$	Left-hand side of the production rule i
$l[i, j]$	An array to store ul
m	An integer
n	Number of parentheses in a legal sequence of parentheses
n_{a_i}	Number of times terminal a_i occurs
q	Number of processors
q_0	Number of processors that gives maximum speedup
r_i	Number of times production rule i used to derive a string
$r(i)$	Right-hand side of the production i
$r[i, j]$	An array to store ur

s	A nonterminal symbol
se	(With an index) a subsequence of parentheses
t	(With or without an index) a terminal symbol
t_r	Time required for a reduce operation
t_s	Time required for a shift operation
u	Number of nonterminals in a grammar
ul	Number of unbalanced left parentheses
ur	Number of unbalanced right parentheses
v	Number of nodes in a digraph
w	Number of arcs in a digraph
z	A terminal symbol
Γ	The set of nodes in a digraph
α	(With or without an index) a string from V^*
β	(With or without an index) a string from V^*
γ	(With or without an index) a string from V^*
δ	(With or without an index) a string from V^*
μ_i	The probability that STK_i has i tokens
ρ	A state from Q
τ_i	A node in a digraph
ω	(With or without an index) a string from V^*

CHAPTER 1
INTRODUCTION

The demand for fast computation and the limitations on the speed of computation with a single processor have motivated researchers to investigate parallel computation. Today parallelism is one of the most salient themes in computer science. Multiple Instruction Multiple Data (MIMD) machines have been in existence for some time (Hwang and Briggs 1984, Kuck 1977). Two primary types of such machines seem to be emerging: 1) the fixed-connection model, such as Intel's iPSC family, and 2) the shared-memory model, such as the HEP computer. When a large number of processors are to be connected together, the former has an advantage from the hardware point of view, but the latter is more convenient to construct an algorithm on.

In the last ten years, a good deal of work has been reported on parallel algorithms in various application areas. Also, several parallel programming languages have been proposed to represent parallelism. However, the amount of work that has been reported on the problem of developing an efficient compiler to run on parallel machines is relatively meager. If parallel compilation techniques are not developed as parallel machines are installed, compilation will be done either on separate,

sequential machines or with inefficiency on parallel machines themselves. Clearly, neither is as attractive as a parallel compiler.

1.1 Compilation

The process of translating a program written in a high-level language, such as Pascal, Fortran, etc., into a machine language is called *compilation.* It is a complex process and is completed in several phases. In this dissertation, compilation is considered as a three-phase process -- *lexical analysis, parsing,* and *code generation.* During lexical analysis characters of the source language that belong together are grouped and each group is called a *token.* The output stream of tokens from the lexical analyzer is input to the parser that performs syntax analysis and constructs syntax tree-structures. A code generator uses the tree-structure to generate code for the target machine. Lexical analysis, parsing, and code generation take approximately 10, 40, and 50 percent of the total compilation time, respectively. Details on compilation can be found in Aho, Sethi, and Ullman (1986).

1.2 Parallel Compilation

Two approaches can be taken in exploiting parallelism in compilation. One approach is pipelining of different phases on a linearly connected array of processors. This approach can provide only a limited amount of speedup because of the limited

number of distinct phases in a compiler. The other approach is employing many processors in each phases of compilation. It is obvious that the two approaches are complementary. As in sequential compilation, the output of one phase is the input to the next phase. In the beginning of each phase, input for that phase is partitioned and one processor works on each partition. For example, the output token-string from the lexical analyzer is partitioned into substrings, and one processor parses one substring.

1.3 Related Research

Lexical analysis in logarithmic time can be done cost efficiently by the existing parallel algorithm of Mickunas and Schell (1978). However, cost-efficient parallel parsing algorithms have to be designed. Existing parallel bottom-up parsing algorithms of Mickunas and Schell (1978), and Fischer (1975) are direct extensions of existing sequential bottom-up parsing algorithms. In these algorithms, the input token-string is partitioned at regular intervals, and each processor is assigned to parse one partition. Occasionally processors exchange information. The main drawback of these algorithms is that the minimum parsing time is proportional to the height of the parse tree (Cohen, Hickey, and Katcoff 1982). Thus, the speedup dependents on the structure of the parse tree of the input string. For example, when the parse tree is very skewed, very little speedup is obtained. Performance evaluation of these algo-

rithms by simulation has hardly explained the question of speedup. An analytical method to determine speedup would be more desirable.

A cost-effective parallel parsing algorithm with logarithmic-time complexity has to be designed. If syntax-directed partitioning of input token-string is done, parallel parsing can be performed very fast. However, a parallel algorithm for syntax-directed partitioning of input string is not known. This dissertation provides some solutions to the problems pointed out in this section.

1.4 Overview of Dissertation

Chapter 2 presents a survey of previous work related to parallel compilation. Chapter 3 is devoted to parallelizing existing sequential parsing algorithms. Two models for asynchronous bottom-up parallel parsing are presented. These models are adaptations to existing sequential bottom-up parsing algorithms for a linearly connected array of processors and a completely connected network of processors. Also, it is illustrated that the speedup obtained by parallel parsing on these models dependents on the structure of the parse tree of the input string.

A method for estimating speedup in asynchronous bottom-up parallel parsing is presented in Chapter 4. To estimate the speedup, the probability of occurrence of each terminal in a language is used. The method developed is used to estimate

speedup for Pascal-like languages. It is observed that the bottom-up parallel parsing techniques do not achieve "good" speedup for Pascal-like languages.

In Chapters 5 and 6, a new parsing technique is developed. The novelty of this technique is that it is inherently parallel. Chapter 6 presents a parallel parenthesis-matching algorithm and a parallel parsing algorithm. The parallel parsing algorithm uses the parenthesis-matching algorithm and a parenthesis insertion table.

A method for constructing a parenthesis insertion table is described in Chapter 5. Strings generated by a finite set of finite-subgrammars of a given context-free grammar are used to construct the parenthesis insertion table. To define a finite set of finite subgrammars for a grammar the digraph of the grammar is used.

CHAPTER 2
PREVIOUS RESEARCH

This chapter is devoted to a brief survey of previous work related to parallel compilation and separate compilation. The work on lexical analysis, parsing, code generation, and separate and pipelined compilation is reviewed. Before presenting the review, a description of the most commonly used *Parallel Random Access Memory (PRAM)* model of parallel computation will be described.

In PRAM, many processors can access a shared memory. Restrictions on type of simultaneous read from and write into a memory cell by more than one processor divide PRAMs into three classes: 1) *Concurrent Read Concurrent Write (CRCW) PRAM:* simultaneous reads from and writes into a memory cell by more than one processor are allowed, 2) *Concurrent Read Exclusive Write (CREW) PRAM:* simultaneous reads into a memory cell by many processors are allowed but not simultaneous writes, 3) *Exclusive Read Exclusive Write (EREW) PRAM:* neither simultaneous reads nor simultaneous writes are allowed.

2.1 Lexical Analysis

Soon after the advent of vector computers, Lincoln (1970) proposed techniques for performing lexical analysis of Fortran-like languages using vector operations.

Subsequently, Zosel (1973) also used the same vector-operation method for lexical analysis. Donegon and Katzke (1975) presented an algorithm for lexical analysis utilizing the vector instruction set of the CDC STAR-100. Mickunas and Schell (1978) proposed a two-pass parallel lexical analyzer for regular lexical languages. In their algorithm, the input string is divided into substrings of approximately equal lengths and lexical analysis is performed on each substring in parallel. The algorithm achieves linear speedup if the number of processors is less than or equal to $(L \ / \log L)$.

2.2 Parsing

Very little literature exists pertaining to parallel parsing on either vector or on MIMD machines. Lincoln (1970) realized the need for parallel parsing and proposed some techniques for parsing Fortran-like languages on vector machines. Subsequent work extending Lincoln's idea is reported by Zosel (1973). Ellis (1971) presented two algorithms for parallel parsing of Fortran-like languages. He considered two data organizations - vertical and horizontal. In a vertical data organization each processor processes one statement and obtains speedup via inter-statement parallelism, whereas in a horizontal data organization many processors operate on all tokens of a statement simultaneously to take the advantage of intra-statement parallelism.

Donegon and Katzke (1975) described parsing techniques that can exploit the vector instruction set of the CDC STAR-100 machine.

Fischer (1975) laid the foundation for non-serial bottom-up parsing. In his method, the input string is divided into segments and each processor parses one segment from left to right. The parsing is necessarily nondeterministic in the sense that several stacks may have to be kept by each processor. This is because a processor (with exception to the left-most one) does not know the state of its left neighbor when that neighbor finishes scanning its segment. The grammar of the language being parsed, however, is deterministic.

Fischer's algorithm is synchronous. This means that at each point in the parsing, each processor tries to perform the same operation. Only after all the processors have finished this operation can they proceed to the next one. A parser may perform three operations:

1. *Shift*: Push an input symbol onto a stack.
2. *Reduce*: Replace a right-hand side of a production rule on the top of the stack by its left-hand side.
3. *Merge*: Combine the stacks of two neighboring processors and let the left processor proceed with parsing while the right processor becomes inactive.

Fischer's main concern was to prove the correctness of a variety of bottom-up parsing techniques using this synchronous model of parallel parsing. The results of

simulating the model by Fischer indicate that substantial gain in speedup could be attained when several processors are used. However, the speedup is input dependent.

The work of Cohen, Hickey and Katcoff (1982) was of theoretical interest. They determined upper bounds for the speedup attainable by bottom-up synchronous parsing as suggested by Fischer. The two basic operations each processor performs are shift and reduce. The time spent for merge operations is neglected in determining the upper bounds. Cohen and Kolodner (1985) have proposed a model for bottom-up parallel parsing using asynchronous processors. The model is based on an extension of shift-reduce parsers which are able to merge the information they keep on their stacks. The main objective of their work was to provide estimates of the speedup attainable when using the proposed model. Their simulation results, applicable to the parallel parsing of programs written in Pascal-like languages, show how speedup varies with the number of processors for different ratios of the times to shift, reduce, and merge.

Mickunas and Schell (1978) extended the LR parsing technique (Aho, Denning, and Ullman 1972; Aho, Sethi, and Ullman 1986; Knuth 1965) for multiple processors environment. In their method, processors can start parsing at different arbitrary places in the input string. The algorithm is very similar to the error-recovery algorithm of Mickunas and Modry (1978). The approach is based on two simple tactics:

1. Whenever a shift-reduce or reduce-reduce conflict is encountered, the parser transmits its stack symbols to its left neighbor, alters its parsing state, and resumes parsing.
2. If a reduction is indicated but the stack lacks the required information, the parser performs as much reduction as possible and transmits to its left neighbor the information enabling it to complete the reduction.

With this strategy a reduction may ripple through a number of parsing processes before it is completed. The main focus of Mickunas and Schell's work is to show how the shift-reduce table can be computed when the parsers operate in parallel. The paper is not concerned with the analysis of the speedup gains obtained by using parallel parsers. The authors, however, made a brief reference to the fact that parsing may require time proportional to the height of the derivation tree. This corresponds to one of the coarse estimates in the already mentioned work of Cohen, Hickey and Katcoff (1982).

Ligett, McCluskey and McKeeman (1982) also extended LR parsing algorithms and measured their performances experimentally. The speedup they obtained is linearly proportional to the number of processors if the size of the input to a processor is not "too small." Loka (1984) proposed a two-processor parallel-parser, one processor starting at the left end of the input string proceeds to the right as in LR parsing, while the other processor starting at the right end of the string moves towards

the left. However, more work has to be done before the idea can be implemented.

2.3 Code Generation

In compilation parsing is not an end unto itself. Rather it is closely connected to another phase of compilation - code generation. Code generation in a multiprocessing environment is almost an unexplored area. Ellis (1971) introduced an elementary idea of code generation in parallel for arithmetic expressions. Fischer (1975) developed a parallel algorithm to generate three-address intermediate codes from a given infix arithmetic expression. Krohn (1975) showed how code can be generated for Fortran-like languages utilizing the vector instruction set of the CDC STAR-100 machine. In his method object code is generated in parallel for three classes of statements: those statements not containing arithmetic expressions (such as DO), arithmetic expressions, and statements containing arithmetic expressions (such as IF). Each class of statements is processed using a sequence of vector transformations. Several passes through the same set of vector instructions are required, as the syntactical tree is built for an arithmetic expression. At each level, registers are assigned and the generated code is merged into the output stream. Schell (1979) extended the parallel parsing technique of Mickunas and Schell (1978) by using attribute grammar for code generation in a multiprocessor environment. However, they did not address the question of speedup and efficiency.

Dekel and Sahni (1983) considered the translation of infix arithmetic expressions

into their postfix or syntax-tree forms on EREW PRAMs. Their algorithm is a parallel version of the classical method which uses an explicit stack and operator weights to perform the translation. They have shown how to translate an infix string of length L using L processors in $O(\log^2 L)$ time. Bar-On and Vishkin (1985) reduced the time complexity for this problem using a more powerful model of computation, CREW-PRAM. They have shown that the computation tree form of the arithmetic expression of length L can be generated in $O(\log L)$ time using $(L/\log L)$ processors.

2.4 Separate and Pipelined Compilation

Baer and Ellis (1977) have shown that by modeling an existing sequential compiler we gain an understanding of modifications necessary to transform the sequential structure into a pipeline of processes. They have evaluated a pipelined compiler through simulation. Lipkie (1979) considered the compilation of Pascal-like programs using multiple independent processors. He dealt with two kinds of concurrency: one in which processors separately compile procedures of comparable size, the other in which processors simultaneously execute the various passes of a multiple-pass compiler (e.g., lexical analysis, parsing, etc.). However, the speedup dependents on the number of procedures the program has and size of the procedures.

CHAPTER 3
BOTTOM-UP PARALLEL PARSING

Two methods for designing parallel algorithms for a problem are: 1) transforming an existing sequential algorithm into a parallel algorithm keeping the basic strategy of the sequential algorithm unaltered, and 2) finding a new technique that is inherently parallel in nature. It is obvious that the latter method produces a new sequential algorithm. In the past, first method has been used for designing parallel bottom-up parsing algorithms.

In Section 3.1, definitions are presented and notations are introduced. Some sequential algorithms amenable to parallelization are also surveyed. Section 3.2 introduces the sequential shift-reduce parsing and presents two models for parallel bottom-up shift-reduce parsing. Section 3.3 illustrates the parallel parsing technique on a linear array of three processors (Model A). Section 3.4 concludes the chapter, discussing the best-case and the worst-case performances of the algorithms.

3.1 Preliminaries

In this section, we recall some standard definitions, introduce some notations, and present some sequential parsing strategies that might be considered for adapta-

tion to parallel parsing.

3.1.1 Definitions and Notations

The definitions and notations used in this chapter are conventional and are similar to those found in Aho, Denning, and Ullman (1972), Aho and Ullman (1972), and Gray and Harrison (1972). A *context-free grammar* (CFG) is a quadruple, $G = < V_N, V_T, P, R >$ where,

V_N is a finite nonempty set of nonterminal symbols,

V_T is a finite set of terminal symbols, and $V_N \cap V_T = \varnothing$, an empty set,

P is a finite set of production rules of the form $s := \alpha$, such that $s \in V_N$ and $\alpha \in V^*$, V^* is the set of all strings from $V = V_N \cup V_T$; s is called the *left part* and α is called the *right part* of the production,

R is a special symbol, $R \in V_N$ and is called the start symbol.

The set of strings generated by a grammar G is called the *language* of G and is denoted by $L(G)$. The language generated by a context-free grammar is called a *context-free language*. For $\omega_1, \omega_2, \gamma, \delta \in V^*$ we say that ω_2 is "directly derived from" ω_1, denoted by "$\omega_1 \rightarrow \omega_2$" if and only if there exist $(s := \alpha \in P)$, $(\omega_1 = \gamma s \delta)$, and $(\omega_2 = \gamma\alpha\delta)$. It is said that "$\omega_1 \rightarrow \omega_2$" is a *direct derivation.* If $\delta \in V^*_T$, then the direct derivation $\omega_1 \rightarrow \omega_2$ is called *rightmost.* If $\alpha_0 \rightarrow \alpha_1 \rightarrow \ldots \alpha_{r-1} \rightarrow \alpha_r$ then it is said that α_0 derives α_r. The sequence of

derivations is called a derivation of α_r from α_0. If all derivations used in a derivation are rightmost, then the derivation is called *right derivation* or *canonical derivation.* A *right parse* is the reverse of a sequence of productions applied in a rightmost derivation. Similarly, *left derivation* and *left parse* are defined. Left parses are produced by top-down parsers, and right parses are produced by bottom-up parsers.

A *sentential form,* ω , is a string in V^* such that R derives ω using zero or more derivation(s). The set of sentential forms for a grammar G is denoted by $SF(G)$. If R derives ω using zero or more rightmost derivation(s) only, then ω is a *canonical sentential form* (CSF). If ω is in V^*_T, then it is called a *sentence.* If R derives $\gamma s \delta$ and $\gamma s \delta \rightarrow \gamma\alpha\delta = \omega$ then α is called a *reducible phrase* of the sentential form ω. The leftmost reducible phrase in a *CSF* is also called a *handle* for the sentential form.

Every sentence generated by an *unambiguous* context-free grammar, G, has exactly one rightmost (or leftmost) derivation from the start symbol. Otherwise the grammar is ambiguous. A grammar is *reduced* if every nonterminal other than the start symbol 1) derives at least one terminal string, and 2) appears in at least one sentential form. We will consider only unambiguous and reduced context-free grammars.

In the subsequent sections, much attention will be given to *LR* grammars (Knuth 1965) and related grammars. Intuitively, a grammar is $LR(k)$ if, using full left-context and k symbols of lookahead, a handle can be located in a *CSF*. In the

following subsection different parsing methods are discussed.

3.1.2 Parsing Strategies

A brief survey of parsing techniques that may be considered for adaptation to parallel parsing is presented in this section. Two natural classes of parsing strategies are top-down and bottom-up. Top-down methods synthesize derivation trees from their root nodes, starting with the start symbol and constructing the tree from the top down as the input is scanned from left to right. Recursive descent parsers are widely known representatives of this class of parsers. The *LL* parsing of Knuth (1971) is also a top-down method. Bottom-up parsers, as the name implies, construct derivation trees from the bottom up, connecting subtrees to form new subtrees as the input is consumed. Precedence parsing, bounded context parsing, and *LR* parsing belong to this class. Top-down strategies are eliminated as they appear to be inherently unsuitable for adaptation to parallel parsing (Schell 1979).

Several sequential bottom-up parsing methods have been developed. A variety of methods, namely the *precedence* methods, are based on the use of relationships between symbols. Among the precedence methods are: *operator precedence* (Floyd 1963), *simple precedence* (Wirth and Weber 1966), *weak precedence* (Ichbiah and Morse 1970), and *total precedence* (Colmerauer 1970). Precedence parsing technique appears to provide a natural basis for adaptation to parallel parsing. Fischer (1975)

presented several variations on the precedence theme for parallel parsers. The major drawback in using precedence technique is the inconvenience to the compiler writer in producing grammars for them. Precedence technique requires *uniquely invertible grammars* in which the right part of each production must be distinct from the right part of any other production.

The unique invertibility requirement can be eliminated if context information is used to determine which production to apply when right parts are identical. This approach characterizes the *mixed strategy precedence* family of parsers (McKeeman, Horning, and Wortman 1970; Aho, Denning, and Ullman 1972). Even this class of parsing algorithms restricts the set of acceptable grammars. The *bounded context* parsing technique (Eickel et al. 1963; Floyd 1964) also exhibits this problem.

LR parsing algorithms work by constructing the parse tree from the bottom up. At every step, subtrees are connected by constructing a new node to form a bigger subtree until the complete parse tree is obtained. One way of parallelization of these sequential algorithms is by constructing more than one node (if exists) at each level of the parse tree simultaneously. Because of this simplicity, the *LR* family of parsing algorithms is an obvious candidate for adaptation for parallelization. Other good properties of LR parsers are: The class of languages recognized by *any LR* parser is exactly the class of deterministic context-free languages. *LR* parsers deterministically parse in linear time. Moreover, *LR* parsers are efficient in practice with respect to

both time and space. An added benefit in using *LR* parsing is the so-called *proper prefix property,* which allows *LR* parsers to detect errors at the earliest possible point before shifting the erroneous symbol.

3.2 Bottom-Up Parsing Algorithms

The preceding discussions show that *LR* family of parsing algorithms is amenable to parallelization. In this section, we briefly present the sequential shift-reduce parsing technique and present two parallel-parsing models based on the sequential shift-reduce parsing.

3.2.1 Sequential Shift-Reduce Parsing

Shift-reduce parsing, a technique of which *LR* parsing is a special case, uses a *pushdown stack* -- the *parse stack,* a *finite state control,* and an *input cursor,* or *read head.* During its operation, a shift-reduce parser can perform any of the four actions: *shift, reduce, accept,* or *error.* For every CFG, two functions, ACTION and NEXT, are defined to implement the finite state control. Both are defined on the domain $Q \times V$, where Q is the set of states of the finite state control and V is the set of symbols of the grammar. ACTION determines the parse action to be applied for state-symbol pair; NEXT supplies the parser's next state. Given a parser in state ρ with its input curser at symbol a, there are four possibilities:

1. ACTION(ρ, a) = shift. The symbol a is shifted (pushed) onto the stack, as is the new current state, ρ' = NEXT (ρ, a). The read head is moved to the next token to the right side of a.

2. ACTION(ρ, a) = reduce i. The i-th production is applied to reduce the stack. If i-th production is $s \rightarrow \alpha$, then $| \alpha |$ symbol-state pairs are removed from the stack, uncovering some state ρ''. The nonterminal α is placed onto the stack, followed by the new state, ρ' = NEXT(ρ, a).

3. ACTION(ρ, a) = accept. The parser halts and accepts the input.

4. ACTION(ρ, a) = error. In this simple model, the parser halts and rejects the input. In practice, error recovery might be attempted.

At the beginning, the initial state is placed onto the top of the stack and the read head is placed to the leftmost token of the input string. The action of a shift-reduce parser is uniquely determined by its current state, which is on the top of the stack, and by the next input symbol to be examined.

The sequential shift-reduce parsers are *canonical* in that the sequence of productions that they apply in deriving their operations is exactly the sequence applied in a right (or canonical) parse. Further, the configurations of the parser are directly related to the canonical sentential forms produced by the right parse. Parallel shift-reduce parsers presented in this section are non-canonical, and the shift-reduce algorithm is modified accordingly.

3.2.2 Parallel Shift-Reduce Parsing -- Two Models

In this section, we introduce two parallel parsing models based on the following assumptions:

1. A fixed number of processors are assigned at the start of parsing.
2. Parsers are started at arbitrary syntactic elements in the input string.
3. All parsers behave like classical shift-reduce parsers in that they consume inputs from left to right and do not backtrack on the input.
4. Each parser has its own pushdown stack, a finite state control, and an input cursor.
5. All parsers execute asynchronously.
6. There exist some direct or indirect channels between the pairs of parsers via which information can be passed.

The parsing models have a simple basis: each parser shifts and reduces using its own input segment, occasionally transmitting symbols to some other parser. Any symbols coming from other parsers are treated as extensions to the parser's input. The transmission mechanism will be discussed in the next subsections. As our algorithms cannot be implemented on presently available machines, implementation details of the algorithms will be avoided and the basic models only will be presented.

Model A. Let there be q processors $P_1, P_2, \ldots, P_q$ arranged in a linear array, such that processor P_i is directly connected with processor P_{i+1} and P_{i-1} (see Figure 3.1). A processor P_j is called a predecessor of the processor P_k if $j < k$ and processor P_k is referred to as a successor of P_j. Thus, processor P_i has $(i - 1)$ predecessors and $(q - i)$ successors. Processor P_1 has no predecessor and processor P_q has no successor. Every processor P_i has a stack, which is referred to as STK_i.

The given input string of length L is divided into approximately q equal parts. The ith processor P_i starting at token $\lfloor (i - 1)L / q \rfloor$ scans to the right for the next synchronizing token (e.g., semi-colon, end, etc.) and initiates parsing from the next token. A processor can be in one of the four states -- *active, wait, merge-only,* and *inactive*.

A processor remains in the *active* state if it is able to perform either of the two parse steps, namely shift or reduce, or if it is performing the *stack-merge* operation. *Stack-merge* is the process in which a processor P_i transfers the contents of its stack from bottom to the stack of another processor P_j until P_i encounters a *stack-separator* or its stack becomes empty. (*Stack-separator* is a special symbol used as a marker to separate the content of the stack of a processor.) When a processor cannot reduce due to insufficient information in its stack, but has received the next synchronizing token, it places a stack separator on the top of the stack and continues parsing. By placing a stack separator, a new stack is simulated.

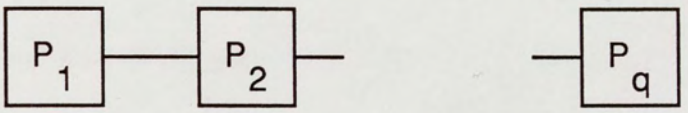


Figure 3.1. A Linearly Connected Arraay of Processors.

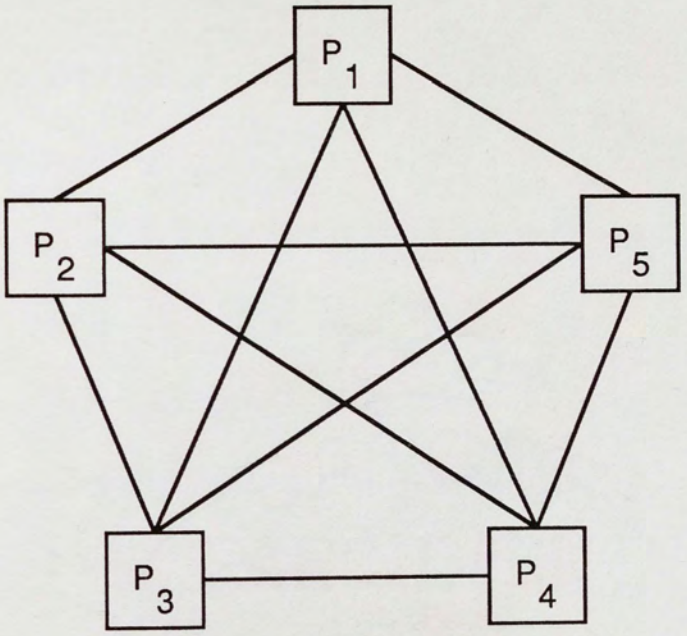


Figure 3.2. Five Completely Connected Processors.

When the end processor P_q has completed parsing its part of input and is left with a nonempty stack, it enters into *merge-only* state. When any other processor P_i, $1 \leq i < q$, has completed parsing its part and is left with a nonempty stack, it requests a merge to its successor P_{i+1} and enters into wait state. In the wait state a processor P_i may be acknowledged by the processor P_{i+1} or may get a request from processor P_{i-1}. In the former case the state of P_i is changed to active and P_i receives tokens from P_{i+1}, while in the latter case if processor P_i has a separated stack then it cancels its merge request to the processor P_{i+1}. If a processor P_i is not in wait state and receives a merge request from P_{i-1}, and its stack is separated, then P_i sends acknowledgments to P_{i-1}. After sending an acknowledgment, processor P_i starts stack-merge with P_{i-1}. Processor P_i, $1 < i < q$, with nonempty stack goes to merge-only state when its successor processor P_{i+1} is inactive. In merge-only state a processor P_i waits for a merge request from its predecessor P_{i-1}. Processor P_i becomes inactive if its stack is empty and P_{i-1} is inactive. A processor P_i in wait state with empty stack does not acknowledge a merge request immediately but waits for the contents of the stack of P_{i+1} and transfers them to P_{i-1}. In this model each processor executes the following algorithm.

```
while not (end of input) do
    shift;
    if a reduction is indicated then
        if sufficient information is in the stack then
            reduce
        else
            place a stack separator on the top of the stack;
        endif
    else
        if a merge request from the left neighbor and the stack is separated then
            acknowledge the merge request;
            transfer tokens from the bottom of the stack until a stack separator is
            found;
        endif;
    endif;
endwhile;
if i = q then
    enter into merge-only state
else
    send a merge request to the right neighbor;
    enter into wait state;
endif;
while not (in inactive state ) do
    case state of
    merge-only: if a merge request from the left neighbor then
                    acknowledge the request;
                    transfer content of the stack from the bottom
                    to the left neighbor until stack is empty
                    or a stack separator is found;
                    if the stack is empty then
                        enter into inactive state;
                    endif;
                endif;

    wait: if a merge request from the left neighbor and the stack is separated then
            cancel the merge requested to the right neighbor;
            acknowledge the merge requested by the left neighbor;
            transfer content of the stack from the bottom to the left
            neighbor until a stack separator is found;
            send a merge request to the right neighbor;
```

```
            endif;
            if the merge request is acknowledged by the right neighbor then
                enter into active state;
                accept information from the right neighbor;
                if a reduction is indicated then
                    if sufficient information is in the stack then
                        reduce
                    else
                        place a stack separator on the top of the stack;
                    endif
                endif;
                if the right neighbor is in inactive state then
                    enter into merge-only state
                else
                    send a merge request to the right neighbor;
                endif
            endif;
        endcase
endwhile;
```

Model B. In this model every processor can communicate directly with every other processor (see Figure 3.2). As expected, the extra cost of interprocessor interconnection provides an enhancement in parsing speed by reducing the interprocessor coordination and communication time. In such a completely connected system, although there is no predecessors and successors of a processor in strict sense, to identify each processor and its substring we number them as in Model A and use the same terms predecessor and successor. The processors of this model also have four states (as in Model A), but the state transition is different in a few cases, as discussion follows.

Processor P_{i+1} is called the immediate successor of P_i. As soon as the stack of

a processor becomes empty, it enters into inactive state irrespective of the state of its immediate successor. A processor P_i knows which processor P_k is its immediate active successor. Before a processor P_i enters into inactive state it informs its immediate active predecessor P_j the index of its immediate active successor P_k. For example, let the immediate active successor of P_3 be P_5 and that of P_5 be P_{10}. Consider the situation in which P_5 becomes inactive before P_3 and P_{10}. In this situation, before P_5 becomes inactive it informs P_3 that P_{10} is henceforth the immediate active successor of P_3. Each processor executes an algorithm very similar to that for each processor in Model A.

3.3 An Example

Parallel parsing on Model A is illustrated in this section. Consider the following set of productions for simple arithmetic expressions with only addition and multiplication.

$E := E + T$

$E := T$

$T := T * F$

$T := F$

$F :=$ **id**

$F := (E)$

In these productions, boldfaced symbols are terminals. Let us illustrate the parsing of the following expression on a linear array of three processors.

id * id + id * (id + id * id)

Since the input string (i.e., the expression) has 13 tokens, the first four tokens from the left of the string are assigned to processor P_1; the next four tokens are assigned to processor P_2, and the rightmost five tokens are assigned to P_3. After consuming the input, each processor builds subtree(s) (shown in Figure 3.3) and is left with some tokens in the stack of each processor as shown in Figure 3.4 . At this point no processor has enough information to continue parsing, and hence P_1 requests P_2 for information and P_2 requests P_3 for information. The processor P_3, being the boundary processor, enters into merge-only state, while P_1 and P_2 enter into wait state after passing merge requests to P_2 and P_3, respectively. After receiving the merge request from P_2, P_3 transfers information from its stack to P_2 starting at the bottom of its stack. On receiving this additional information, P_2 continues parsing and builds the parse subtree as shown in Figure 3.5. At the end of possible last reduction, it enters into merge-only state and transfers the contents of its stack to P_1. After transferring information to P_1, processor P_2 becomes inactive. Finally, P_1 completes parsing, building the subtree shown in Figure 3.6.

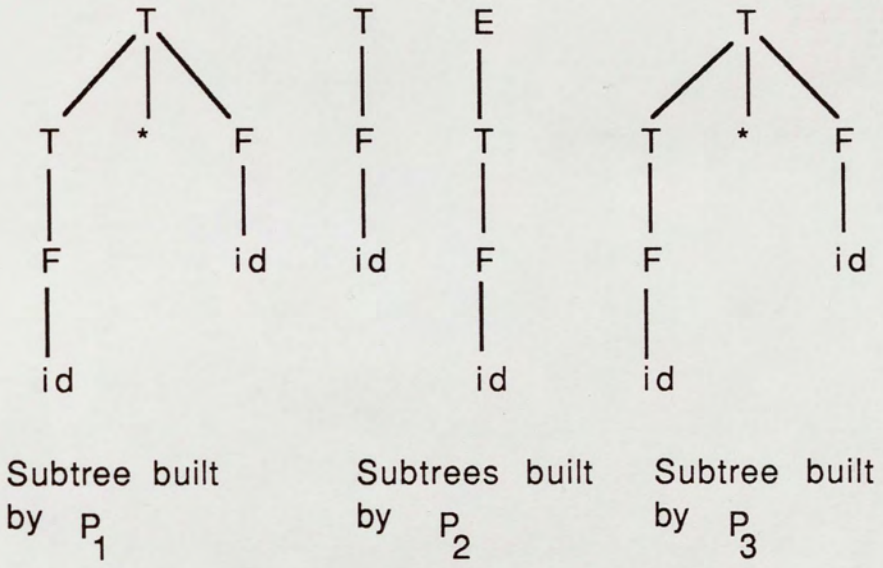


Figure 3.3. Subtrees Built by the Processors.

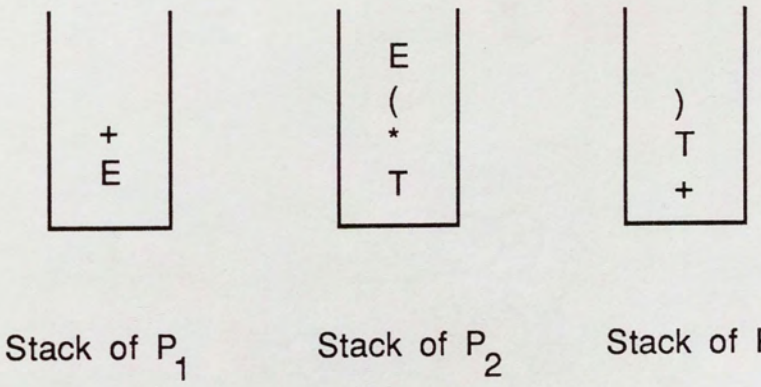


Figure 3.4. Contents of Stacks of the Processors.

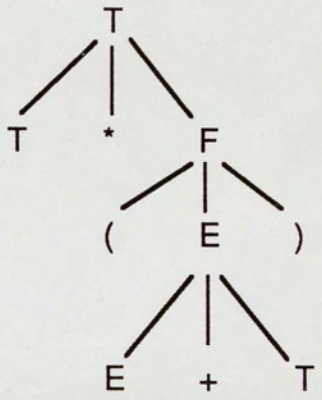


Figure 3.5 Subtree Built by P_2 After Receiving Tokens from P_3 .

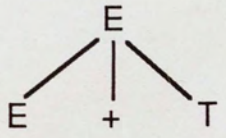


Figure 3.6. Subtree Built by P_1 After Receiving Tokens from P_2 .

3.4 Discussion

In this chapter two models for parallel parsing on multiprocessor systems have been proposed. One parser runs on each processor. One model is for an array of linearly connected processors. The other model is for a completely connected network of processors. It is expected that parallel parsing on these models will reduce parsing time. However, parsing on these models will take time at least equal to the height of the parse tree. The following two examples illustrate that 1) in the worst case, a parse tree might have a height proportional to the length of input and 2) in the best case, a parse tree might have a height proportional to the logarithm of the length of input.

Let us consider the following grammar:

E := E + T | **id**

T := T * T | **id**

The length of the string corresponding to the parse tree shown in Figure 3.7 can be expressed in terms of the height of the parse tree as:

$$L = 2(H - 2) + 1 = 2H - 3$$

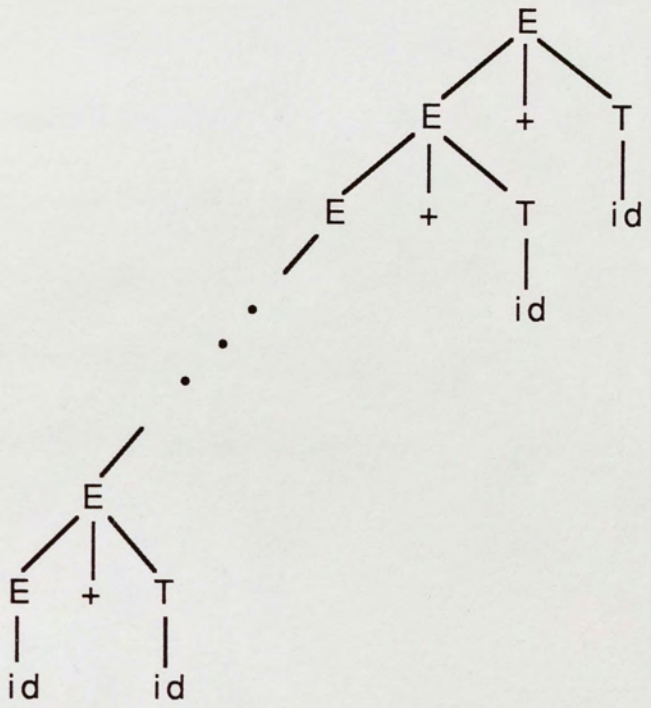


Figure 3.7. A Parse Tree Whose Height is Proportional to the Length of the String Parsed.

where L is the length of the string and H is the height of the parse tree. Thus, minimum parsing time on any of the models is $O(L)$. On the other hand, a parse tree may have a height proportional to the logarithm of the length of the string it represents. For example, the length of the string corresponding to the parse tree shown in Figure 3.8 is $O(\log L)$. Thus, minimum parsing time is $O(\log L)$. Therefore, average parsing time on the proposed parsing models would be a better measure of performance for them. In the next chapter, a method for estimating speedup obtainable by bottom-up parallel parsing is developed. It is also shown that neither of the models yields a "good" speedup. Since, even the completely connected network of processors fails to yield "good" performance, no other models are considered.

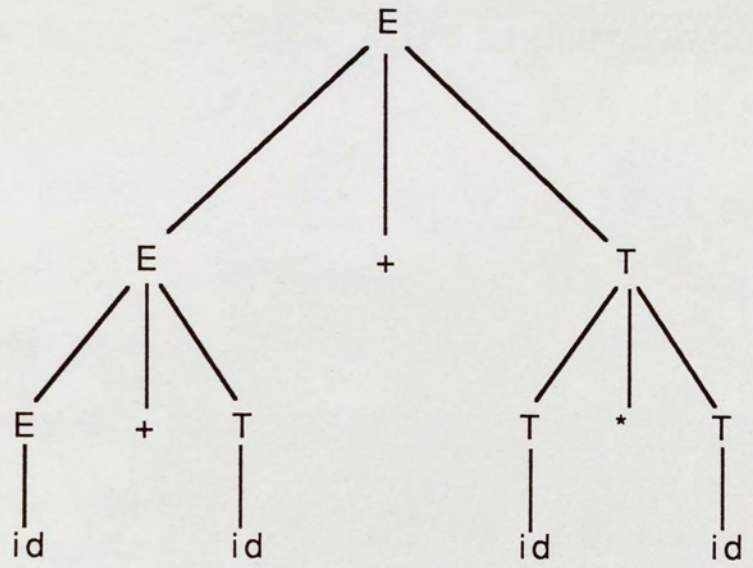


Figure 3.8. A Parse Tree Whose Height is Proportional to the Logarithm of the Length of the String Being Parsed.

CHAPTER 4
ESTIMATING THE SPEEDUP IN PARALLEL PARSING

One measure of performance for a parallel algorithm is the *speedup* that is obtained by using the parallel algorithm. The speedup is defined as the ratio of the execution time of the best known sequential algorithm to the execution time of the parallel algorithm. The performances of the two parsing models presented in the previous chapter are input dependent. There is a wide gap between the worst-case performance and the best-case performance. Schell (1979), and Cohen, Hickey, and Katcoff (1982) also observed that speedups obtained by parallel bottom-up parsing algorithms are input dependent. Thus, to get a better understanding of the performances of these algorithms, Cohen and Kolodner (1985) simulated a parallel bottom-up parsing model. Their simulation results, applicable for Pascal-like languages, show that estimated speedup depends on the number of processors used and the length of the input string. However, this simulation results do not provide any functional relation among the number of processors used, the length of the input string, and the estimated speedup.

In this chapter a method for estimating the speedup obtainable by parallel bottom-up parsing algorithms is developed. In order to develop this method, the total

parallel parsing time is divided into three constituent parts: parallel shift time, parallel reduce time, and parallel coordination and communication time. Using the set of production rules and probabilities of occurrences of terminal symbols of the grammar of a language, parallel shift and reduce times are expressed as a function of the number of processors used and the length of the input string. Parallel coordination and communication time is also expressed as a function of the number of processors and the length of the input string. Finally, the total parallel parsing time obtained by adding these three times is used to get an expression for the estimated speedup.

In Section 4.1, different notations are introduced and necessary definitions are presented. Section 4.2 estimates parallel shift and reduce time. Section 4.3 is devoted to estimating coordination and communication time for two parsing models presented in Chapter 3. Section 4.4 investigates the nature of speedup curves for parallel parsing on these two models. Section 4.5 illustrates the use of the method by estimating the speedup obtained by parallel bottom-up parsing of Pascal-like languages. Section 4.6 discusses the results.

4.1 Notations and Definitions

Let T_{pq} be the total parallel time for parsing a string of length L using q processors. Let the total parallel parsing time T_{pq} be divided into three constituent parts - parallel shift time, parallel reduce time, and parallel coordination and communica-

tion time. (Pushing an input token onto a stack is called a shift operation. Replacing the right-hand side of a production rule on the top of a stack by its left-hand side is called a reduce operation.) Let T_{sq}, T_{rq}, and T_{cq} denote parallel shift time, parallel reduce time, and parallel coordination and communication time, respectively.

Thus, the total time to parse in parallel, T_{pq} can be expressed as

$$T_{pq} = T_{sq} + T_{rq} + T_{cq}$$

Since the coordination and communication time is zero in case of a single processor, the total parse time T_{p1} with a single processor is

$$T_{p1} = T_{s1} + T_{r1}$$

The speedup obtained with q processors is defined as:

$$SP(q) = \frac{T_{p1}}{T_{pq}} \tag{4.1}$$

4.2 Parallel Shift and Reduce Times

It is assumed that the parse tree has a *critical* level h such that for levels 1 (root) to h the number of nodes at each level is smaller than the number of processors, and all processors are not utilized while the parse tree is being constructed in these levels. The number of nodes at levels $(h + 1)$ and higher are such that q

processors work simultaneously and independently with negligible coordination and communication. From level h to level one, processor coordination and communication time is significant. Thus, we consider coordination and communication time, T_{cq}, for this part only.

Let the estimated number of nodes in an average parse tree of a string of length L be N and the estimated number of internal nodes in critical levels 1 through h of the parse tree be N_c. The number of shift operations is the length L of the input string, which is also the number of leaves in the parse tree. The number of reduce operations is $(N - L)$, the number of internal nodes in the parse tree. If t_r and t_s be the average reduce and shift times, respectively, (for one operation), then we can express the total parse time T_{p1} with a single processor as:

$$T_{p1} = (N - L)\, t_r + L\, t_s$$

In parallel parsing shift operations are executed in parallel and independently. Thus, $T_{sq} = \frac{L\, t_s}{q}$. The reduce operations corresponding to the internal nodes below level h are executed almost independently and in parallel (as the number of nodes at each level exceeds the number of processors) and requires $\frac{(N - L - N_c)\, t_r}{q}$ units of time. The internal nodes at levels 1 through h require h units of reduction time (Cohen, Hickey, and Katcoff 1982) in addition to the processor coordination and

communication time T_{cq}. Therefore,

$$T_{rq} = \frac{(N - L - N_c)\, t_r}{q} + h\ t_r$$

and

$$T_{pq} = \frac{(N - L - N_c)\, t_r}{q} + \frac{L\ t_s}{q} + h\ t_r + T_{cq}$$

Substituting T_{p1} and T_{pq} into equation (4.1) we get the expression for the speedup with q processors as:

$$SP(q) = \frac{(N - L)\, t_r + L\ t_s}{((N - L - N_c)\, t_r + L\ t_s)/q + h\ t_r + T_{cq}} \tag{4.2}$$

The number of nodes, N, depends on the length of the string and the grammar. The level h depends on the number of processors and the garmmar. The processor coordination and communication time, T_{cq}, depends on the number of processors as well as on the processor interconnection topology. First we estimate N, N_c and h. Then, T_{cq} will be estimated for both the models introduced in Chapter 3.

Consider a deterministic, context-free language with m production rules and u nonterminals. Let the production rules be numbered as 1, 2, . . . , m. In derivation of a string of length L, let i-th production rule be used r_i times. Then we can express the number of internal nodes in the parse tree as:

$$N - L = \sum_{i=1}^{m} r_i \tag{4.3}$$

Cohen and Roth (1978) developed a method for evaluating r_i in terms of occurrences of $(m - u)$ terminals $a_1, a_2, \ldots, a_{m-u}$. Using this method, the number of uses of each production in terms of the occurrences of the terminals *if, else, case, while, repeat, ;, +, *, >* and *()* for a Pascal-like language is shown in Table 4.1, where n_{a_i} is the number of times terminal a_i occurs. If the frequency of occurrence of terminals is known, we can estimate $N - L$. Two methods were described by Cohen and Roth (1978) to determine the average frequency of terminals. Using these techniques, we can approximate the number of internal nodes as a function of L, as follows:

$$N - L = k' L$$

where k' depends on the grammar. Therefore, we can write

$$N = k\ L \tag{4.4}$$

where $k = k' + 1$. Hence, the average number of sons of an internal node is

$$d = \frac{N - 1}{N - L} = \frac{k - 1/L}{k - 1}$$

TABLE 4.1

SYNTAX OF PASCAL-LIKE LANGUAGE AND COUNT RELATIONS BETWEEN TERMINALS

RULE NO. i	RULE	NUMBER OF USES IN A SUCCESSFUL PARSE r_i
1	P := S	1
2	S := S ; I	$n_{;}$
3	S := I	$n_{\text{if}} + n_{else} + n_{while} + n_{repeat} + n_{case} + 1$
4	I := id ← E	$n_{;} + n_{else} + 1$
5	I := *if* B *then* S *fi*	$n_{\text{if}} - n_{else}$
6	I := *if* B *then* S *else* S *fi*	n_{else}
7	I := *while* B *do* S *od*	n_{while}
8	I := *repeat* S *until* B	n_{repeat}
9	I := *case* E *of* S *end*	n_{case}
10	E := E + T	n_{+}
11	E := T	$n_{;} + n_{else} + n_{case} + n_{()} + 2n_{b_{op}} + 1$
12	T := T * F	n_{*}
13	T := F	$n_{+} + n_{()} + n_{;} + n_{else} + n_{b_{op}} + 1$
14	F := *id*	$n_{*} + n_{+} + n_{;} + n_{else} + n_{case} + 2n_{b_{op}} + 1$
15	F := (E)	$n_{()}$
16	B := E b_{op} E	$n_{b_{op}}$

The average number of sons for the internal nodes at levels 1 through h is also assumed to be d. Assuming that level h has exactly q internal nodes, we can write

$$q = d^{h-1}$$

from which we get

$$h = \log_d(q) + 1 \tag{4.5}$$

Similarly, N_c, the number of nodes at levels 1 through h, can be expressed in terms of q and d as follows:

$$N_c = \frac{d^h - 1}{d - 1}$$

$$\text{that is,} \qquad N_c = \frac{d\ q - 1}{d - 1} \tag{4.6}$$

Since now it is evident that speedup is also a function of L, the length of the input string, we will write $SP(L, q)$ instead of $SP(q)$. Substituting from (4.4), (4.5), and (4.6) into (4.2) we get expression for the speedup as:

$$SP(L, q) = \frac{k' L\ t_r + L\ t_s}{\dfrac{(k' L - \dfrac{(d\ q - 1)}{(d - 1)})\ t_r + L\ t_s}{q} + (\log_d(q) + 1)\ t_r + T_{cq}} \tag{4.7}$$

In the next section, we estimate the coordination and communication time T_{cq} for Model A and Model B and derive expressions for the speedup for Pascal-like languages.

4.3 Processor Coordination and Communication Time

Cohen, Hickey, and Katcoff (1982) showed that the upper bound for speedup for parallel bottom-up parsing increases monotonically with the number of processors and reaches a limiting value. Beyond this critical number of processors no further speedup is obtained. In obtaining these results, Cohen, Hickey, and Katcoff neglected the processor coordination and communication time. Furthermore, they conjectured that the average speedup curve for strings of a given length would be of the same shape as their maximum speedup curves.

We show that the Cohen-Hickey-Katcoff conjecture holds for Model B; but for Model A we get an expression for speedup which is close to the simulation result obtained by Cohen and Kolodner (1985), but quite different from the speedup conjectured by Cohen, Hickey, and Katcoff (1982).

The processor coordination and communication time depends on processor interconnection topology of the model used for parallel parsing. We determine the value of T_{cq} for each of the two models and then substitute them to get the expression for the speedups.

4.3.1 Coordination and Communication Time for Model A

In Model A the average coordination and communication time T_{cq} is determined by the average number of tokens left in STK_q and the number of processors, q. For a merge request to travel from P_1 to P_q, it takes $(q - 1)$ units of time, and for the first token to reach P_1 it takes $(q - 1)$ units of time, where the unit of time is the period required by two adjacent processors to exchange a message or datum. If k_1 is the average number of tokens in processor P_q's stack (when it enters into merge-only state) then the next $(k_1 - 1)$ tokens can be passed to P_1 in the next $(k_1 - 1)$ units of time using pipelining. This gives:

$$T_{cq} = 2(q - 1) + (k_1 - 1)$$

4.3.2 Coordination and Communication Time for Model B

In this model every processor can communicate with every other processor directly. Hence, to collect all irreducible tokens from P_q in h reduction steps, P_1 may need $(h - 1)$ requests and k_1 data transfers. Hence,

$$T_{cq} = (h - 1) + k_1$$

or, $$T_{cq} = \log_d(q) + k_1$$

Next we estimate k_1, the average number of tokens left on a stack.

4.3.3 Estimating the Average Number of Tokens

Let μ_i be the number of tokens left on STK_i when P_i completes parsing its part of the input. We define,

$$\mu = Max(\mu_i, i \in \{1, 2, \ldots, q\})$$

$$\text{Then,} \qquad k_1 = \sum_{i=1}^{\mu} i \, \text{pr}(i) \tag{4.8}$$

where $\text{pr}(i)$ is the probability that at least one stack has i tokens and no stack has more than i tokens.

Assuming that a processor may have any number of tokens between one and μ with equal probability of $\frac{1}{\mu}$, we derive an expression for $\text{pr}(i)$. The probability that STK_1 has μ tokens when P_1 completes parsing the input to it is $\frac{1}{\mu}$. The probability that STK_1 has fewer than μ tokens but STK_2 has μ tokens is $(1 - \frac{1}{\mu})\frac{1}{\mu}$. Similarly, the probability that $(q - 1)$ stacks have fewer than μ tokens and STK_q has μ tokens is given by:

$$(1 - \frac{1}{\mu})^{q-1} \frac{1}{\mu}$$

The probability that at least one stack has μ tokens is:

$$pr(\mu) = \frac{1}{\mu} + (1 - \frac{1}{\mu})\frac{1}{\mu} + \ldots + (1 - \frac{1}{\mu})^{q-1}\frac{1}{\mu}$$

$$\text{or,} \qquad pr(\mu) = 1 - (1 - \frac{1}{\mu})^{q}$$

Similarly we get,

$$pr(\mu - 1) = (1 - \frac{1}{\mu})^{q}(1 - (1 - \frac{1}{\mu})^{q}),$$

and in general,

$$pr(\mu - i) = (1 - \frac{1}{\mu})^{iq}(1 - (1 - \frac{1}{\mu})^{q}),$$
$$\text{for } i \in \{1, 2, \ldots, \mu - 1\}$$

and

$$pr(1) = 1 - \sum_{i=2}^{\mu} pr(i) = (1 - \frac{1}{\mu})^{(\mu-1)q}$$

Substitution of $pr(i)$ in equation (4.8) gives

$$k_1 = \sum_{i=0}^{\mu-2}(\mu - i)(1 - \frac{1}{\mu})^{iq}(1 - (1 - \frac{1}{\mu})^{q}) + (1 - \frac{1}{\mu})^{(\mu-1)q}$$

$$k_1 = (1 - (1 - \frac{1}{\mu})^{q})\sum_{i=0}^{\mu-2}(\mu - i)(1 - \frac{1}{\mu})^{iq} + (1 - \frac{1}{\mu})^{(\mu-1)q}$$

$$= \frac{\mu - (\mu + 1)(1 - \frac{1}{\mu})^{\mu})^{q} \; - \; 2(\mu - 2)(1 - \frac{1}{\mu})^{(\mu-1)q} + (2\mu - 3)(1 - \frac{1}{\mu})^{\mu q}}{1 - (1 - \frac{1}{\mu})^{q}}$$

In practical situations $\mu \geq 2$, and if the number of processors is large, then we can approximate k_1 by

$$k_1 = \mu - \frac{(1 - \frac{1}{\mu})^{q}}{1 - (1 - \frac{1}{\mu})^{q}}$$

When $\mu = \frac{L}{q}$, then we get $k_1 \cong \frac{L}{q} - \frac{L}{q^2}$.

In the rest of the chapter this expression for k_1 is used.

4.4 Nature of Speedup Function

The expression for average speedup for Model A is

$$SP_A(L, q) =$$

$$\frac{k' L \; t_r + L \; t_s}{\frac{(k' L - \frac{(d\; q \; - 1)}{(d \; - 1)}) \; t_r + L \; t_s}{q} + (\log_d(q) + 1) \; t_r + 2(q \; - 1) + \frac{L}{q} - \frac{L}{q^2} - 1} \qquad (4.9)$$

The general shape of the speedup curve for a given length L of the string with varying number of processors can be obtained as follows.

The numerator in $SP_A(L, q)$ does not depend on the number of processors. Hence, we consider only the denominator. Let the denominator be denoted by DSP_A. Taking the first derivative of DSP_A with respect to q and equating this to zero (after removing those terms that asymptotically go to zero), the expression for q is

$$q = ((k' t_r + t_s + 1)L/2)^{1/2} = q_0 \quad \text{(say)}$$

Substituting q_0 in the second derivative of DSP_A (with respect to q), an expression with a positive value is obtained. Therefore, q_0 is the number of processors which parse a string of length L in a minimum time. The speedup increases with the number of processors to a maximum and then decreases.

Similarly, the expression for speedup for Model B is given by

$$SP_B(L, q) = \frac{k' L\ t_r + L\ t_s}{\frac{(k' L - \frac{(d\ q - 1)}{(d - 1)})\ t_r + L\ t_s}{q} + (\log_d(q) + 1)\ t_r + \frac{L}{q} - \frac{L}{q^2} - 1} \tag{4.10}$$

It can be shown that $SP_B(L, q)$ increases to a maximum value monotonically, then it remains constant. Unlike $SP_A(L, q)$, $SP_B(L, q)$ does not decrease as the number of processors is increased beyond the critical number.

4.5 *Speedup for Pascal-like Languages*

To find the speedup for Pascal-like languages, we calculate N, estimated number of nodes, and d, estimated degree in the parse tree, from Table 4.1 as follows:

$$N - L = \sum_{i=1}^{16} r_i$$

$$= 5n_{;} + 2n_{\text{if}} + 5n_{else} + 2n_{while} + 5n_{case} + 3n + 4n_{()} + 4n_{>} + 2n_{*} + 6 \quad (4.11)$$

Using this expression and the frequency of occurrence of each terminal given in Cohen and Kolodner (1985) and shown in Table 4.2, we get

$$N = 2.4175\ L$$

$$d = 1.70547$$

Substituting these values of N and d in equations (4.9) and (4.10), we get

$$SP_A(L, q) =$$

$$\frac{1.4175\ L\ t_r + L\ t_s}{\dfrac{(1.4175\ L - \dfrac{(1.7047\ q - 1)}{(0.7047)})\ t_r + L\ t_s}{q} + (\log_{1.7047}(q) + 1)\ t_r + 2(q - 1) + \dfrac{L}{q} - \dfrac{L}{q^2} - 1}$$

and,

TABLE 4.2

AVERAGE OCCURRENCE OF SOME TERMINALS
IN PASCAL-LIKE LANGUAGES (COHEN AND KOLODNER 1985)

TERMINALS	FREQUENCY OF OCCURRENCE (EVERY 100 TERMINALS)
id	60
←	6
if	2
else	0.9
while	0.1
repeat	0.05
case	0.15
()	6.6
;	12
+	4.6
*	4.6
b_{op}	4.6

$$SP_B(L, q) =$$

$$\frac{1.4175\, L\, t_r + L\, t_s}{\frac{(1.4175\, L - \frac{(1.7047\, q - 1)}{(0.7047)})\, t_r + L\, t_s}{q} + (\log_{1.7047}(q) + 1)\, t_r + \frac{L}{q} - \frac{L}{q^2} - 1}$$

Figure 4.1 shows the speedup curves with $t_r = t_s = 1$ and $L = 1000$. The dotted curve presents the speedup obtained by simulation in Cohen and Kolodner (1985).

4.6 Discussion

A method for estimating the speedup for asynchronous, bottom-up, parallel parsing has been presented. To develop this method, a few assumptions about the nature of the parse tree were made. Thus, the expressions may not give exact speedup, but the closeness of the estimated speedup using the method developed here and the simulation result of Cohen and Kolodner (1985) indicates that the assumptions are realistic, and that the significant parameters have been taken into account.

Study of the nature of speedup curves has shown that the maximum speedup is obtained when $O(L^{1/2})$ processors are used. For $O(L^{1/2})$ processor a speedup of $O(L^{1/2})$ is obtained. Therefore, the parallel bottom-up parsing technique is not very parallel, i.e., it cannot produce even an average-case $O((log\ L)^k)$-time parallel algorithm for any constant k. Hence, a faster and better algorithm has to be designed from scratch. In the next two chapters, we develop an entirely new parallel parsing technique for a class of block-structured languages.

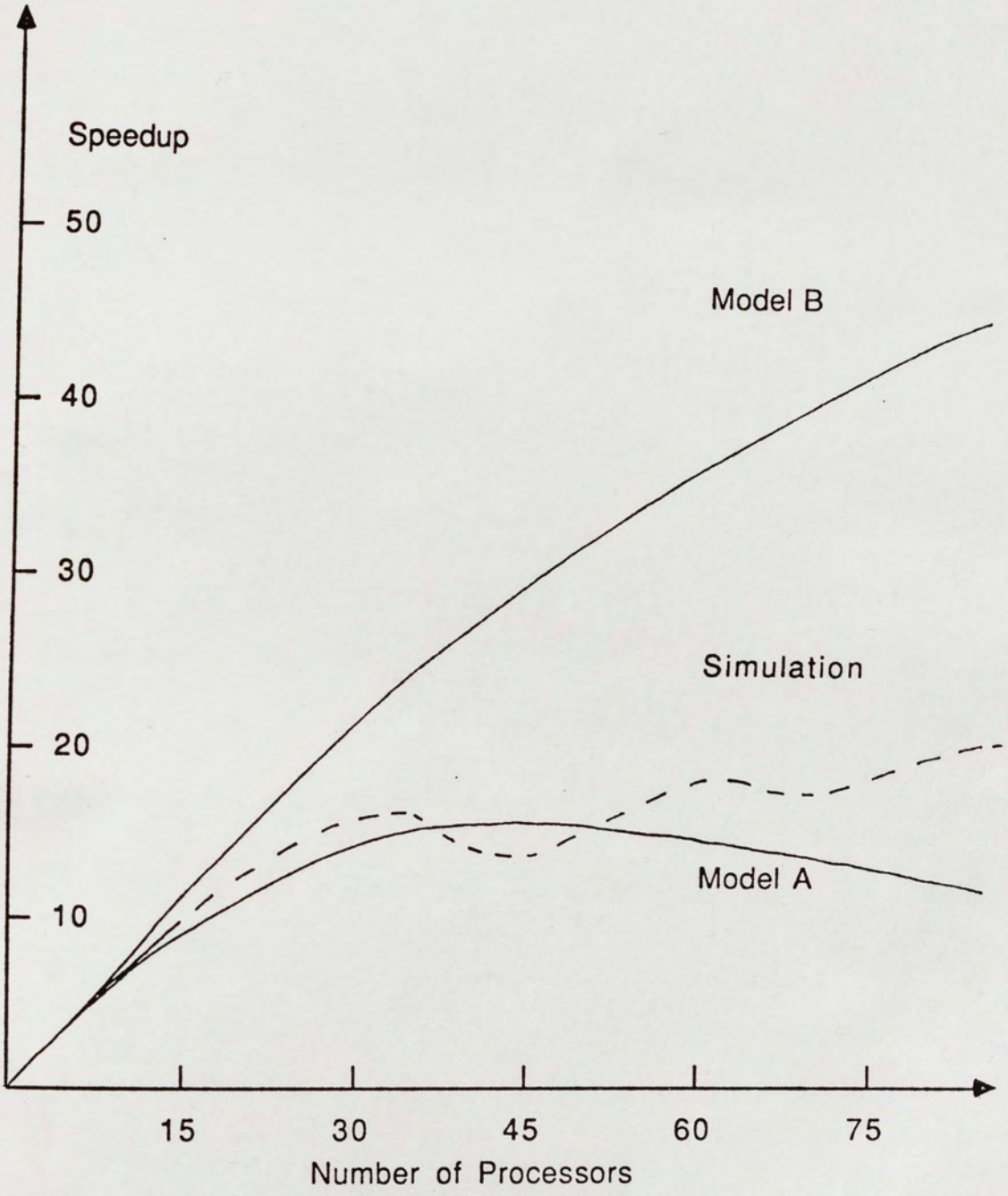


Figure 4.1 Number of Processors vs Speedup Curves for Pascal-Like Languages.

CHAPTER 5
SUBGRAMMARS AND PARENTHESIS INSERTION

In the last chapter it was established that parallelization of existing sequential bottom-up parsing algorithms fails to yield even a "good" estimated speedup. Thus, fast parallel parsing algorithms have to be designed from scratch. To this end, Baccelli and Fleury (1982) took a new and radically different approach. They proposed sequential lexical analysis and syntax-directed partitioning of the input string by a host processor, and concurrent parsing of these substrings by other processors. In their method, once the input has been partitioned, the processors can proceed with parsing without any further communication among themselves. However, a serious limitation of their algorithm is that the input string is partitioned sequentially, and thus partitioning requires a linear time. Also, in their method a substring may be very long, resulting in an unbalanced work load on different processors and a long concurrent parsing time.

In this chapter and in the following chapter, a new parallel parsing technique is presented. In this technique, parsing is completed in two phases. The first phase performs a syntax-directed partitioning of the input string in parallel. The second phase constructs the parse tree (also in parallel). Consequently, the syntax directed parti-

tioning technique of Baccelli and Fleury is different from ours in two ways: 1) our method is inherently parallel and 2) our method produces substrings each of whose length is bounded by a constant which is determined by the grammar. The present chapter develops the theoretical basis for the parallel parsing algorithm presented in the next chapter.

In Section 5.1 a method is proposed to define minimum subgrammars of a context-free grammar (with a finite or infinite language) such that the language generated by each of these subgrammars is finite and the length of every string is also finite. It may be worth mentioning that the union of languages of the subgrammars do not give the language of the original language. Section 5.2.1 identifies a class of block-structured languages that has fast parsing algorithms. Section 5.2.2 describes how to determine the number of parentheses to be inserted into the string to be parsed such that a syntax-directed partitioning of the input is achieved. Section 5.3 summarizes the results.

5.1 Minimum Number of Subgrammars

In this section, after recalling the definitions of subgrammars and digraphs, we develop a method to define minimum number of subgrammars of a CFG such that each of them generates a finite language. This set of subgrammars will be called minimum subgrammars.

5.1.1 Definitions

Following Baccelli and Fleury (1982), we define *subgrammars* of a context-free grammar $G = <V_N, V_T, P, R>$. For the i-th production ($s := \alpha \in P$), let the left side of i be defined as $l(i) = s$, and the right side of i be denoted as $r(i) = \alpha$; for $B \in V_N$, let $PR(B) = \{i \mid (i \in P) \text{ and } (l(i) = B)\}$.

Consider a subset $W \subseteq V_N$. For $B \in W$ the W-subgrammar G_B generated by B is defined as $G_B = <V_{NB}, V_{TB}, P_B, R_B>$, where R_B is a new symbol, $R_B \in V$, $V = (V_N \cup V_T)$; $P_B = (\bigcup_{i \in PR(B)} \{R_B := r(i)\}) \bigcup (\{PR(X) \mid X \in V_N - W$ and $B \Rightarrow \alpha X \beta$, such that $\alpha, \beta \in V^*\})$; $V_{NB} = \{l(i) \mid i \in P_B\}$, and $V_{TB} = W \bigcup \{a \mid (a \in V_T)$ and $B \Rightarrow \alpha a \beta$, such that $\alpha, \beta \in V^*\}$.

A *digraph* $D(\Gamma, A)$ consists of a set of nodes $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_v\}$ and a set of arcs $A = \{A_1, A_2, \ldots, A_w\}$, where $A_i \in \{(\Gamma \times \Gamma)\}$. If $A_i = <\tau_j, \tau_k> \in A$, we say that there is an arc from node τ_j to node τ_k. A *directed path* from a node τ_i to node τ_j is an alternating sequence of nodes and arcs, beginning with some node τ_i and ending with some node τ_j such that each arc is oriented from the node preceding it to the node following it. A directed path is called a *cycle* if it starts and ends in the same node. A digraph is *acyclic* if it has no cycle, otherwise it is called *cyclic*. For other graph theoretic terms the reader is referred to Deo (1974).

5.1.2 Digraph of a CFG and Minimum Subgrammars

Given a context-free grammar G, we can draw a digraph D_G with a node for each $s_i \in V_N$ and a directed arc from s_i to s_j if and only if there exists a production $s_i := \beta s_j \gamma$, $\beta, \gamma \in V^*$, and $s_j \in V_N$. Now the subgrammar G_B can be redefined as:

$$G_B = <V_{NB}, V_{TB}, P_B, R_B>$$

where,

R_B and V_{TB} are as defined earlier and

$V_{NB} = \{R_B\} \cup \{X \mid X \neq B$, and in D_G there exists a directed path from node B to node X without going through a node $Y \in W\}$, and

$$P_B = \bigcup_{q \in PR(B)} \{R_B := r(q)\} \cup \{PR(X) \mid X \in V_{NB}\}.$$

By this new definition of subgrammars, the digraph D_{G_B} of a subgrammar G_B is a subgraph of D_G which includes the node B and the nodes reachable from B, via a directed path in D_G, without going through any $X \in W$ and the directed arcs that connect these nodes.

From D_G if we choose a set of nodes W_a such that their removal makes the remaining digraph acyclic, then the digraph D_{G_B} of the subgrammar G_B, $B \in W_a$ is acyclic. Thus, the language generated by G_B is finite (Hopcroft and Ullman 1979). For a given context-free grammar G there exists a constant such that the length of

each of the strings generated by the subgrammars is bounded by the constant. Also, the number of strings generated by the subgrammars is bounded by a constant. Thus, The language generated by a subgrammar G_B is finite (and hence the length of every string in this language is finite), where $B \in W_a$ such that removal of the nodes in W_a makes D_G acyclic.

Let $W_l \subseteq V_N$ be the set of nodes in D_G of the context-free grammar G such that removal of the nodes in W_l from D_G makes it acyclic but removal of the nodes in any proper subset of W_l does not make it acyclic. Let W_L be the set of all such W_l, and $W_{\min}$ be a minimum cardinality set in W_L. The subgrammars corresponding to the set $W_{\min}$ will be called *minimum subgrammars* of G. Thus, the language generated by a grammar that belongs to the set of minimum subgrammars is finite (and hence the length of every string in this language is finite).

Determination of minimum subgrammars involves the computation of the set $W_{\min}$ from D_G which is known to be an NP-complete problem (Krishnamoorthy and Deo 1979). Therefore, computation of $W_{\min}$ may be very expensive. However, since for a given grammar the computation of $W_{\min}$ is done once for all, it may be worth spending a long time if it is practical. Otherwise, we can use a suboptimal set W_l computed using some heuristic.

5.1.3 An Example

We conclude this section with an example of a grammar G of a Pascal-like language for which we draw the digraph, define minimum subgrammars, and enumerate the strings of these subgrammars. G is defined as:

$G = <V_N, V_T, P, S>$,

where,

$V_N = \{S, I, B, E, T, F\}$,

V_T = { ; , ← , aend , if , then , else , fi , while , do , od , repeat , rend , until , case , of , end , + , * , (,) , b_{op} , id } (All terminals are underlined.)

S is the start symbol.

P :

1 $S := S$; I

2 $S := I$

3 $I :=$ id ← E aend

4 $I :=$ if B then S fi

5 $I :=$ if B then S else S fi

6 $I :=$ while B do S od

7 $I :=$ repeat S until B rend

8 I := case E of S end

9 $B := E \; b_{op} \; E$

10 $E := E + T$

11 $E := T$

12 $T := T * F$

13 $T := F$

14 $F :=$ id

15 $F := (E)$

The digraph corresponding to this grammar G is shown in Figure 5.1. Clearly, if the nodes $S, E,$ and T are removed, the digraph becomes acyclic. As each of these nodes has a self-loop, addition of any one of these nodes makes it cyclic. Thus, the set $W_{\min} = \{ S, E, T \}$. The minimum subgrammars, G_S, G_E, G_T, are as follows:

$G_S = <V_{NS}, V_{TS}, P_S, R_S>$,

where,

$V_{NS} = \{ R_S, I, B \}$

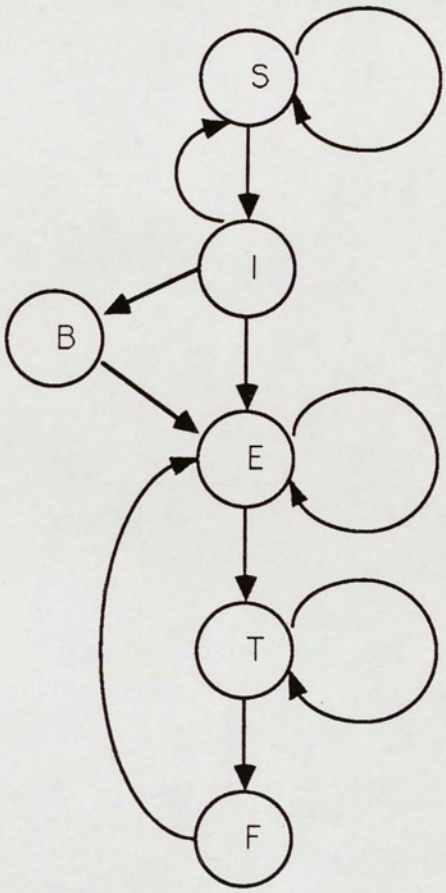


Figure 5.1. Digraph of the Context-Free Grammar in the Example 5.1.3.

V_{TS} = { ; , ← , aend , then , else , fi , while , do , od , repeat , until , rend , case , of , end , b_{op} , id , S , E }

P_S :

1 $R_S := S \ ; \ I$

2 $R_S := I$

3 $I :=$ id ← E aend

4 $I :=$ if B then S fi

5 $I :=$ if B then S else S fi

6 $I :=$ while B do S od

7 $I :=$ repeat S until B rend

8 $I :=$ case E of S end

9 $B := E \ b_{op} \ E$

$G_E = <V_{TE}, V_{NE}, P_E, R_E>$,

where,

V_{NE} = { R_E }

V_{TE} = { E , + , T }

P_E :

1 $R_E := \underline{E} \ \underline{+} \ \underline{T}$

2 $R_E := \underline{T}$

and $G_T = <V_{NT}, V_{TT}, P_T, R_T>$,

where,

$V_{NT} = \{ R_T, F \}$

$V_{TT} = \{ \underline{\text{id}}, \underline{(}, \underline{)}, \underline{E}, \underline{T}, \underline{*} \}$

P_T :

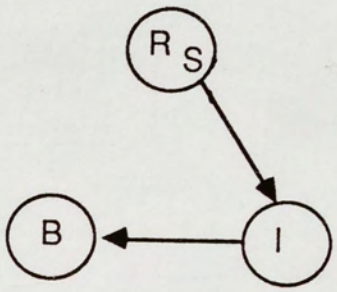
1 $R_T := \underline{T} \ \underline{*} \ F$

2 $R_T := F$

3 $F := \underline{\text{id}}$

4 $F := \underline{(} \ \underline{E} \ \underline{)}$

G_S, G_E, and G_T respectively generate twelve, two, and four strings with the longest string of length eleven, three, and five. The strings are shown in Tables 5.1 and 5.2. Digraphs corresponding to the subgrammars G_S, G_E, and G_T are shown in Figure 5.2.



Digraph of G_S

Digraph Of G_E

Digraph of G_T

Figure 5.2. Digraph of G_S, E_G, and G_T.

TABLE 5.1

ALL STRINGS GENERATED BY SUBGRAMMAR G_S

SUBGRAMMAR	STRINGS
G_S	S ; id $\leftarrow$ E aend
	S ; if E b_{op} E then S fi
	S ; if E b_{op} E then S else S fi
	S ; while E b_{op} E do S od
	S ; repeat S until E b_{op} E rend
	S ; case E of S end
	id $\leftarrow$ E aend
	if E b_{op} E then S fi
	if E b_{op} E then S else S fi
	while E b_{op} E do S od
	repeat S until E b_{op} E rend
	case E of S end

TABLE 5.2

ALL STRINGS GENERATED BY SUBGRAMMARS G_E AND G_T

SUBGRAMMAR	STRINGS
G_E	T
	$E + T$
G_T	id
	(E)
	T * id
	T * (E)

5.2 *Parenthesis Insertion*

The motivation for defining minimum subgrammars in the previous section is to use it for developing a method for syntax-directed partitioning of strings to be parsed. The string will be partitioned by inserting parentheses in it. First we describe a class of block-structured languages whose sentences can be partitioned by inserting parentheses in the string to be partitioned. Then we describe how to determine the

number of parentheses to be inserted in the string to be parsed such that every matched pair of parentheses has a syntax-directed partition of the input string. The strings generated by the minimum subgrammars are used to find the number of left (right) parentheses to be inserted to the right (left) of a terminal. The technique is illustrated with the example used in Section 5.1.3. In the following discussions, by *parentheses* we mean those parentheses which are inserted in the string in order to partition it.

5.2.1 Restricted Block-Structured CFGs (RBSCFGs)

In the rest of the dissertation we consider those CFGs that satisfy the following four constraints and show that the languages they generate can be parsed in $O(\log L)$ time.

1. For every production $s := \alpha$, α has no two or more consecutive nonterminals in it.
2. A terminal t_i is on the immediate left (right) of one and only one nonterminal s_i, but a nonterminal s_i may be on the immediate left (right) of more than one terminal in the same production and / or different productions.
3. The set of productions contains one or more of the following five types of productions:

Let $W_r = \{ s_i \mid s_i := s_i \ \underline{t_i} \ s_j \in P \text{ or } s_i := s_j \ \underline{t_i} \ s_i \in P \}$ and

$W_t = \{ \underline{t_i} \mid s_i := s_i \ \underline{t_i} \ s_j \in P \text{ or } s_i := s_j \ \underline{t_i} \ s_i \in P \}$.

Clearly, $W_r \subseteq W_{\min}$ and $W_t \subseteq V_T$.

a. Recursive Productions

(i) $s_i := s_i \ \underline{t_i} \ s_j$; (left recursive) and

(ii) $s_i := s_j \ \underline{t_i} \ s_i$; (right recursive)

for $s_i \in W_r$; $s_j \in V_N$; $i \neq j$; and $\underline{t_i} \in V_T$.

b. Unit Production

$s_i := s_j$; for $s_i \in W_r$; $s_j \in V_N$; and $i \neq j$.

c. Block Structured Production

$s_i :=$ <u>begin</u> α <u>end</u> ; for $s_i \in (V_N - W_r)$; $\alpha \in V^*$ and satisfy the Constraint 1; and <u>begin</u> , <u>end</u> $\in V_T$.

d. Nonrecursive Production

$s_i := s_j \ \underline{t_i} \ s_k$; for $i \neq j \neq k \neq i$; $s_j, s_k \in V_N$; $s_i \in (V_N - W_r)$ and s_i is on the right of a block-structured production; and $\underline{t_i} \in V_T$.

e. Terminal Production

$s_i :=$ $\underline{t_i}$; for $\underline{t_i} \in V_T$ - W_t and $s_i \in (V_N - W_r)$.

4. Every cycle in the digraph of a grammar has at least one self-loop-free node.

This property can be verified by showing that the digraph becomes acyclic when all the self-loop-free nodes and the self-loops are removed from it.

We denote this class of CFGs by RBSCFGs and the class of languages they generate by RBSCFLs. The languages of the minimum subgrammars of a RBSCFG have the property that two nonterminals (in a RBSCFG) are always separated by one or more terminals, and that on the immediate left (right) of every terminal there is a unique nonterminal. Exploiting these properties, a method to determine the number of parentheses to be inserted on the left (right) of each terminal is described in the next subsection. The grammar in Example 5.1.3 belongs to RBSCFG. It is clear from the grammar in Example 5.1.3 that a language in RBSCFL can include all essential constructs (e.g., assignment, loop, and conditional statements) for describing any computation (Kernighan and Plauger 74).

5.2.2 Number of Parentheses to Be Inserted

In the following discussions, we assume that all recursive productions are left recursive. This assumption simplifies the presentation without loss of generality, as the right recursive productions can be treated almost identically.

Let $SUBL(G)$ be the set of strings generated by a set of minimum subgrammars of G. For a string $\alpha \in SUBL(G)$ and a symbol z in α, let $right(z, \alpha)$ ($left(z, \alpha)$) be the symbol immediately right (left) of z. If z is the right (left) most symbol in

α, $right(z, \alpha)$ ($left(z, \alpha)$) is the null symbol.

We state two rules, one for $\underline{t_i} \in W_t$ and the other for $\underline{t_j} \in (V_T - W_t)$, to determine (from $SUBL(G)$) the number of parentheses to be inserted to the left and right of every terminal for a correct syntax-directed partitioning.

Rule 1 (for recursive terminals). If the string $\alpha_i \in SUBL(G)$ and α_i has $\underline{t_i} \in W_t$, then $left(\underline{t_i}, \alpha_i) \in W_r$; and $right(\underline{t_i}, \alpha_i)$ satisfies one of the following three:

(i) $right(\underline{t_i}, \alpha_i) \in V_T$,

(ii) $right(\underline{t_i}, \alpha_i) \in (W_{\min} - W_r)$, or

(iii) $right(\underline{t_i}, \alpha_i) \in W_r$.

For Cases (i) and (ii) we insert one left parenthesis to the right and one right parenthesis to the left of $\underline{t_i}$; and enclose α_i within one pair of parentheses.

Remark 1: This insertion of parentheses separates two matched pairs of parentheses by $\underline{t_i}$. The pair on the left of $\underline{t_i}$ encloses s_i, $s_i \in W_r$, and the pair on the right of $\underline{t_i}$ encloses a string which is either a single terminal or a block-structured string that does not contain any $\underline{t_i} \in W_t$. Because in the production $s_i := s_i \; \underline{t_i} \; s_j$, if $s_j \in W_r$ then $s_j \in V_{Ts_i}$, and $right(\underline{t_i}, \alpha_i) \in W_r$ which is a contradiction that either $right(\underline{t_i}, \alpha_i) \in V_T$ or $right(\underline{t_i}, \alpha_i) \in (W_{\min} - W_r)$. Thus, the string α_i for Cases (i) and (ii) has to be enclosed in one pair of parentheses for correct partitioning.

For Case (*iii*) $\alpha_i = s_i\ \underline{t_i}\ s_j$ and we insert $(m + 1)$ left (right) parentheses to the right (left) of $\underline{t_i}$ when strings generated by G_{s_j} have to be enclosed in m pairs of parentheses for correct syntax-directed partitioning. We also enclose α_i in $(m + 1)$ pairs of parentheses.

Remark 2: We have a string $\alpha_i = s_i\ \underline{t_i}\ s_j$ such that $s_i, s_j \in W_r$; $\underline{t_i} \in W_t$; and hence (in D_G) s_i and s_j have self-loops, and there is a directed arc from s_i to s_j; but there is no directed arc from s_j to s_i (because of Constraint 4), although there may be a longer directed path from s_j to s_i. Thus, for a correct syntax-directed partitioning, the string generated by G_{s_i} must be enclosed in $(m + 1)$ pairs of parentheses.

Rule 2 (for nonrecursive terminals). We consider $\alpha_i \in SUBL(G)$ such that α_i does not contain $\underline{t_i} \in W_t$. In this case we enclose every $s_i \in W_r$ in $(m + 1)$ pairs of parentheses if the string generated by G_{s_i} must be enclosed in m pairs of parentheses for correct syntax-directed partitioning. We also enclose every $s_i \in (W_{\min} - W_r)$ in one pair of parentheses.

Using these two rules we insert parentheses into every string $\alpha_i \in SUBL(G)$ and count the number of left (right) parentheses to the right (left) of every terminal in V_T. These are the number of parentheses to be inserted to the left (right) of the terminals. For example, when we consider the minimum subgrammars in the illustra-

tion of Section 5.1.3, we get the values in Table 5.3. For every grammar in RBSCFG, we can compute a similar table, which will be called the *insertion table.*

Let $M_i = \{ \underline{t_j} \mid \underline{t_j} \in W_t$, and there are i right (left) parentheses on the left (right) of $\underline{t_j}$ in the insertion table$\}$. If $R \in W_r$, $R := R\ \underline{t_i}\ s_k$, and $\underline{t_i} \in M_i$, then we enclose the string to be parsed in $(i + 1)$ pairs of parentheses. Otherwise, we enclose the string in one pair of parentheses. In the following discussions, when we refer to the insertion of parentheses, it is implied that the string is enclosed in an appropriate pair of parentheses.

Theorem 1. If parentheses are inserted using the insertion table (determined by the preceding two rules), then every string, α_i, in a matched pair of parentheses either belongs to $SUBL(G)$ such that α_i does not contain any symbol $\underline{t_i}$, $\underline{t_i} \in W_t$, or is of the form $s_j\ \underline{t_i}\ s_j\ \underline{t_i}\ \ldots\ s_j\ \underline{t_i}\ s_j$ such that $s_i := s_i\ \underline{t_i}\ s_j$ is a production in the grammar.

Proof. We prove the theorem by showing that:

(1) Rule 1 determines the correct number of parentheses for every $\underline{t_i} \in W_t$, and

(2) If Rule 1 determines the correct number of parentheses for every $\underline{t_i} \in W_t$, then Rule 2 does so for every $\underline{t_i} \in (V_T - W_t)$.

TABLE 5.3

INSERTION TABLE FOR THE GRAMMAR IN SECTION 5.1.3

TERMINAL	NUMBER OF RIGHT PARENTHESES TO BE INSERTED ON THE LEFT	NUMBER OF LEFT PARENTHESES TO BE INSERTED ON THE RIGHT
;	1	1
id	0	0
←	0	3
aend	3	0
if	0	3
b_{op}	3	3
then	3	2
fi	2	0
else	2	2
while	0	3
do	3	2
od	2	0
repeat	0	2
until	2	3
rend	3	0
case	0	3
of	3	2
end	2	0
+	2	2
*	1	1
(	0	3
)	3	0

(1) For every $\underline{t_i}$, $\underline{t_i} \in M_1$, Rule 1 determines the correct number of parentheses (see Remark 1). It is important to note that $M_1 \neq \varnothing$ if and only if $W_r \neq \varnothing$ (by Constraint 4).

Induction Hypothesis. Assume that for every $\underline{t_j} \in M_i$, $1 \leq i \leq k$, Rule 1 determines the correct number of parentheses.

Induction Step. To show that Rule 1 determines the correct number of parentheses for every $\underline{t_{i+1}} \in M_{i+1}$, $1 \leq i \leq k$.

Let s_i, $s_{i+1} \in W_r$ and $\underline{t_{i+1}} \in M_{i+1}$ such that $s_{i+1} := s_{i+1}\ \underline{t_{i+1}}\ s_i$ is a production in the grammar. The string generated from s_{i+1} is of the form $s_{i_1}\ \underline{t_{i+1}}\ s_{i_2}\ \underline{t_{i+1}}\ \ldots\ s_{i_m}$, where s_{i_j}, $j > 1$, are the strings generated from s_i.

By Remark 2 $s_i := s_i\ \underline{t_i}\ s_{i-1}$, $\underline{t_i} \in M_i$, is a production in the grammar and the string generated from s_i is correctly nested if it is enclosed in i pairs of parentheses. When we insert $(i + 1)$ left (right) parentheses to the right (left) of $\underline{t_{i+1}}$, the string generated from s_i is enclosed in $(i + 1)$ pairs of parentheses. Thus, by the induction hypothesis, the outermost pair of parentheses encloses an alternating sequence of s_i and $\underline{t_i}$ (when the contents of the next higher level of nested pairs are replaced by s_i). Therefore, Rule 1 determines the correct number of parentheses for every $\underline{t_{i+1}} \in M_{i+1}$, $1 \leq i \leq k$.

(2) Let $\alpha \in SUBL(G)$ and α has no $\underline{t_i} \in W_t$ in it. Every nonterminal s_i in α is enclosed by a distinct pair of terminals (by Constraints 1 and 2), say $\underline{t_l}$ and $\underline{t_r}$. If $s_i \in W_r$ and $s_i := s_i\ \underline{t_i}\ s_j$, $\underline{t_i} \in M_i$, is a production in the grammar then we have enclosed s_i in $(i + 1)$ pairs of parentheses. Thus, the string generated by s_i is properly nested. If $s_i \in (W_{\min} - W_r)$, then in the insertion table we have one left (right) parenthesis to the right (left) of $\underline{t_{l(r)}}$. Thus, the string generated from s_i is enclosed in one pair of parentheses which is the correct nesting for the string. Hence the theorem.

5.3 Discussion

We have developed a method to define subgrammars of a CFG such that each subgrammar generates a finite language. A class of block-structured, context-free languages, which allows a fast syntax-directed partitioning of any sentence, has been identified. Syntax-directed partitioning is done by inserting parentheses from a table into the sentence to be partitioned. A method to construct a parenthesis insertion table has been developed. In the next chapter the syntax-directed partitioning technique will be used to develop a $O(\log L)$-time parallel parsing algorithm for the class of languages described in this chapter.

CHAPTER 6
PARENTHESIS-MATCHING AND PARSING ALGORITHMS

A method for constructing a parenthesis insertion table has been developed in the previous chapter. When parentheses are inserted into the string to be parsed using this parenthesis insertion table, each matched pair of the parentheses contains a substring that represents a subtree of the parse tree, that is, each matched pair of parentheses contains a syntax-directed partition of the input string. Also, these parentheses can be inserted efficiently in parallel. However, for finding the syntax directed partitions, the matched pairs of parentheses are to be known. In this chapter we present a parallel parenthesis matching algorithm. The algorithm can find the matching for all parentheses of a sequence of n parentheses in $O(log\ n)$ time using $O(n/\log n)$-processor CREW PRAM. Finally, we present the new parallel parsing algorithm. The parsing algorithm uses our parallel parenthesis-matching algorithm as a subroutine. Section 6.1 presents a parallel parenthesis-matching algorithm. In Section 6.2, a parallel parsing algorithm is presented.

6.1 Parenthesis Matching Algorithm

In this section, we consider the parenthesis-matching problem. The problem is to find the matching parenthesis for each parenthesis in a given "legal" sequence of n parentheses. By "legal" it is meant that every parenthesis has its matching parenthesis in the sequence. It is easy to construct an $O(n)$-time sequential algorithm for the problem. Bar-On and Vishkin (1985) have proposed an $O(\log n)$-time parallel parenthesis-matching algorithm and used it to design an optimal parallel algorithm for generation of computation tree forms. To design the algorithm for parallel generation of computation tree form, they have adopted the parenthesis-insertion technique of Knuth (1962). Their $O(\log n)$-time parallel algorithm on a $(n/\log n)$-processor CREW-PRAM is an improvement over Dekel and Sahni's (1983) algorithm for construction of computation tree forms of arithmetic expressions on an EREW-PRAM. The motivation behind their construction of computation tree forms in logarithmic time is the logarithmic-time arithmetic expression evaluation algorithms of Brent (1975) and Miller and Reif (1985). They have shown that if the computation tree of an arithmetic expression is given, the expression can be evaluated in $O(\log n)$-time using n processors, even if the height of the computation tree is greater than $\log n$.

6.1.1 An Outline

To construct an $O(\log n)$-time parallel parenthesis-matching algorithm, Bar-On and Vishkin observed that: (1) each of the $(n / \log n)$ processors assigned to a substring of length $\log n$ can find the pairs of matching parentheses in its subsequence in $O(\log n)$ time; (2) after this local parenthesis matching, each processor is left with an unmatched sequence of parentheses of the form)) . . .)((. . .(; finally, (3) from the remaining sequence of parentheses, if matching for the leftmost left and rightmost right parentheses can be found in $O(\log n)$ time, the matchings for all other parentheses can also be found in $O(\log n)$ time. Then, they proposed an algorithm to find the matching of a parenthesis in $O(\log n)$ time. They used a binary tree to compute the nesting level of each parenthesis and used these nesting levels to construct a variant of a balanced binary search tree. Finally, they proposed a search procedure to find the matching of a given parenthesis in $O(\log n)$ time.

In the following subsection, we propose a simple and elegant optimal algorithm to find the matching of a parenthesis in $O(\log n)$ time. We also use a variant of a binary search tree. However, we do not compute the nesting level of each parenthesis. We observe that if the number of unmatched left parentheses and the number of unmatched right parentheses in each of the two adjacent substrings are known, we can compute the number of unmatched left parentheses and unmatched right parentheses in the string obtained by concatenating these two strings. Using the

technique of concatenating two strings recursively, we build a binary tree. This binary tree is used as the search tree to find the match for each parenthesis.

In Section 6.1.2, we present a procedure to construct the search tree and a procedure to find the matching of a parenthesis. Section 6.1.3 illustrates the algorithm with an example.

6.1.2 Parenthesis Matching Algorithm

Let SE be a legal sequence of balanced parentheses stored in a linear array. Let se_1 and se_2 be two arbitrary consecutive subsequences in SE. A right parenthesis in se_2 without a matching parenthesis in se_2 must have its matching parenthesis in the substring to the left of the substring se_2. Similarly, a left parenthesis in se_1 without a matching parenthesis in se_1 must have its matching parenthesis in the substring to the right of substring se_1. Let ur_1 (respectively ur_2) be the number of right parentheses in se_1 (respectively in se_2) that do not have their matching parentheses in se_1 (respectively in se_2), and let ul_1 (respectively ul_2) be the number of left parentheses in e_1 (respectively in se_2) that do not have their matching parentheses in se_1 (respectively in se_2). In the concatenated string $se = se_1 se_2$, the number of right parenthesis ur is given by

$$ur = ur_1 \qquad \text{if } ur_2 \leq ul_1$$

$= ur_1 + (ur_2 - ul_1)$ if $ur_2 > ul_1$

Thus, the concatenated string se must have $ur_1 + ur_2$ - $\min(ul_1, ur_2)$ right parentheses whose matching parentheses are in the substring to the left of se. Similarly, se must have $ul_1 + ul_2 -$ $\min(ul_1, ur_2)$ right parentheses whose matching parentheses are in the string to the left of se.

We utilize this observation to construct the i-th level of a balanced binary search tree in a bottom-up fashion using values of ur_x and ul_x at every node x at $(i - 1)$th level of the tree (the leaf nodes are at the zero level of the tree). To find the match of a parenthesis, we search on this binary search tree. Each parenthesis in the sequence SE is a leaf-node in the tree. We label a node of the binary tree by an ordered pair $<i, j>$ of nonnegative integers, where the first integer represents the level of the node (from the bottom) and the second represents its position from the left in that level. Nodes $<i - 1, 2j - 1>$ and $<i - 1, 2j>$ are then the left and the right children of the node $<i, j>$. We also use two arrays, $r[i, j]$ and $l[i, j]$, to store the values at nodes $<i, j>$, $0 \leq i < \lceil \log_2 n \rceil$ and $1 \leq j \leq 2^i$. If the parenthesis at position j (in string SE) is a right (respectively left) parenthesis, then $l[0, j] = 0$ (respectively $l[0, j] = 1$) and $r[0, j] = 1$ (respectively $r[0, j] = 0$). The search tree for a given balanced sequence of parentheses is shown in Figure 6.1. The algorithm for construction of the binary search tree is as follows.

CONSTRUCTION OF THE BINARY TREE:

Input: A legal sequence of n parentheses stored in a linear array.

Output: A binary search tree of height $\lceil \log_2 n \rceil$.

Step 1: {Initialization -- for all j **do in parallel** }
if there is a right parenthesis at position j **then**
$r[0, j] := 1$
else
$r[0, j] := 0$;
endif
if there is a left parenthesis at position j **then**
$l[0, j] := 1$
else
$l[0, j] := 0$;
endif

Step 2:
for $i := 1$ **to** $\lceil \log_2 n \rceil$ **do**
for $j := 1$ **to** $2^{\lceil \log_2 n \rceil - i}$ **do** {in parallel}
$r[i, j] := r[i - 1, 2j - 1] + r[i - 1, 2j] -$
$\min (l[i - 1, 2j - 1], r[i - 1, 2j])$;
$l[i, j] := l[i - 1, 2j - 1] + l[i - 1, 2j] -$
$\min (l[i - 1, 2j - 1], r[i - 1, 2j])$;
endfor ;
endfor

In the following discussion, two descendants of a node in the search tree are identified as its left child and right child; the left (respectively right) child is called the left (respectively right) brother of the right (respectively left) child. Let us consider the search procedure to find the matching right parenthesis of a left parenthesis at position x of the input string (for convenience we shall call the parenthesis at position x as parenthesis x or simply x). We search on the search tree constructed by the

previous procedure. Obviously, the matching right parenthesis of x is in the substring to the right of the x. Suppose in the searching process we have arrived at a node of the search tree such that there are c_1 unmatched left parentheses to its right in the substring corresponding to this node. If the present node is a right child, its string concatenates with a substring on the left side of its substring; and hence, no left parenthesis comes to the right of x in the concatenated string, and the value of c_1 does not change. If the present node is a left child, we determine how many right parentheses in the substring corresponding to its right brother do not have a match, i.e., find the value of ur corresponding to its right brother. If $ur \leq c_1$, then the match for x is not in the string corresponding to the right brother and we climb up to the father of the present node. The number of left parentheses to the right of x in the concatenated string is given by $c_1 - ur + ul$. Thus, c_1 is assigned $c_1 - ur + ul$ (ul is the number of unmatched left parentheses of its right brother). We continue to climb towards the root of the search tree by these rules until we reach a node whose right brother has ur unbalanced right parentheses such that $ur > c_1$. At this point we know that the matching parenthesis for x is in the substring corresponding to the right brother, and we move to the right brother and continue to climb down towards the leaves until we reach a leaf-node which has the match for x. While we are climbing down towards the leaf-nodes, we test the number of unbalanced right parentheses in the string corresponding to the left child. If ur-value of the left child is greater

than c_1 then move to the left subtree, otherwise $c_1 := c_1 - ur + ul$ and move to the right child. We continue this process until a leaf-node is reached. The formal description of the procedure is as follows.

SEARCHING FOR THE RIGHT MATCHING PARENTHESIS;

Input: Search tree and the position of a left parenthesis in the sequence of the parentheses.

Output: Position of the matching right parenthesis.

```
{for the matching of a left parenthesis at position x}
    count := 0;   i := 0;   j := x;
    {present node is <i, j>}
    if j is odd {i.e., present node is a left-child} then
        if (count - r[i, j]) < 0 then {end of climbing towards the root}
            j := j + 1; {move to the right brother}
            while i ≠ 0 do
                if count - r[i - 1, 2j - 1] ≥ 0 then
                    count := count - r[i - 1, 2j - 1] + l[i - 1, 2j - 1];
                    i := i - 1;   j := 2j;
                else
                    i := i - 1;   j := 2j -1;
                endif
            endwhile {right parenthesis at location j is the match for left
            parenthesis at location x}
        else {update count and climb towards the root}
            count := count + l[i, j + 1] - r[i, j + 1];
            i := i + 1;   j := (j + 1) / 2 ;
        endif
    else { present node is a right child - climb towards the root}
        i := i + 1;   j := j / 2;
    endif
```

A similar procedure for searching the match of a right parenthesis can be constructed. It is not difficult to implement these two procedures in $O(\log n)$ time using $(n / \log n)$ processors along the line shown in Chin and Chen (1982), Vishkin (1984), and Wyllie (1979).

6.1.3 An Example

A legal sequence of parentheses and its search tree are shown in Figure 6.1. Construction of the search tree using the given procedure is easy. Let us illustrate the searching procedure by searching the matching right parenthesis of the left parenthesis at position five of the input sequence. Initially, we start at node <0, 5> with the count value $c_1 = 0$. The node <0, 5> is a left node; and hence, we compare the *ur*-value of its right brother, $r[0, 6]$ with c_1 and find that c_1 is not smaller than $r[0, 6]$. Therefore, c_1 is assigned $c_1 - r[0, 6] + l[0, 6]$ (= 1) and we climb to the node <1, 3>. We repeat similar steps and climb to the node <2, 2> with c_1-value one. Node <2, 2> is a right child; hence we move to its father, node <3, 1>, without changing the value of c_1. From the left child node <3, 1>, we compare the c_1 value with the *ur*-value, $r[3, 2]$ of its right brother, and find that c_1 is smaller. Therefore, we move to right brother <3, 2>. The left child of the node <3, 2> has an *ur*-value greater than c_1. Hence, we move to left child node <2, 3> and then, for similar

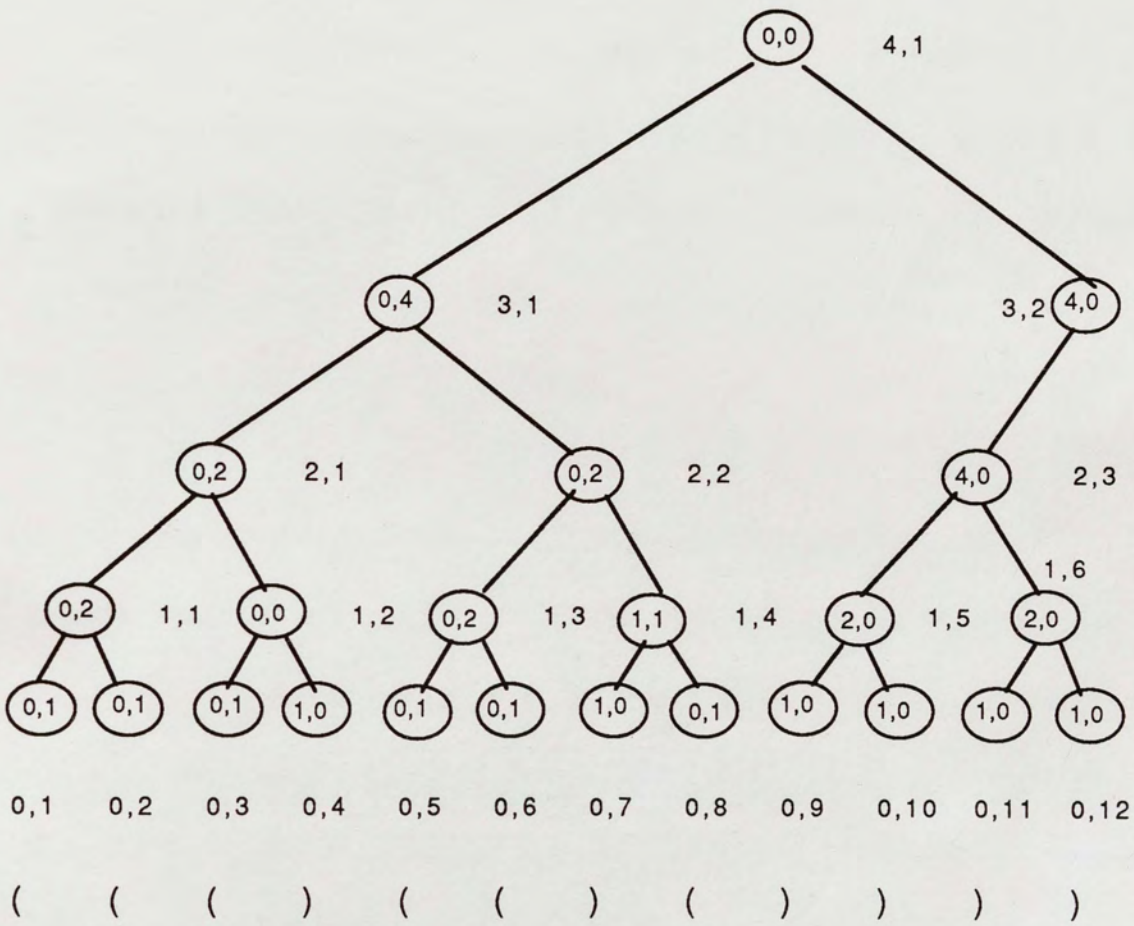


Figure 6.1. A Sequence of Parentheses and Its Search Tree. Node Labels Are Shown Next to the Nodes, and the (r, l) Values Are Shown Within the Circles.

reasons, to node <1, 5>. At this point the c_1 value is one and comparing this value with the *ur*-value of its left child, we find that c_1 is not smaller than the *ur*-value of its left child. Therefore, we move to the right child node <0, 10> assigning c_1 to $c_1 - r[0, 9] + l[0, 9]$ where $r[0, 9]$ and $l[0, 9]$ are the *ur*-value and *ul*-value of the left child of the node <1, 5>. The node <0, 10> is a leaf node, and hence the parenthesis at position ten is the matching parenthesis for the left parenthesis at position five.

6.2 Parsing Algorithm

In this section a two-phase parallel parsing algorithm for a class of block-structured Pascal-like languages is developed. We insert parentheses in the string to be parsed and find the match for each left (right) parenthesis using the parenthesis-matching algorithm presented earlier. One processor is assigned to constructing a parse subtree for the string in each matched pair of parentheses, where the contents of its next higher level of matched pairs are replaced by nonterminal symbols that must be produced in a successful parsing. We assume that L processors are available. To make the presentation concise, let us define the following three types of left parentheses:

(*i*) A type-1 left parenthesis has a nonterminal immediately right to it.

(*ii*) A left parenthesis is type-2 if immediately right to it is a left parenthesis whose right match is located immediately left to the right match of the former left parenthesis (see Figure 6.2.a).

(*iii*) A type-3 left parenthesis has immediately right to it a left parenthesis whose right match is not located immediately left to the right match of the former left parenthesis (see Figure 6.2.b).

The predecessor of a type-2 left parenthesis is defined as follows:

(*i*) If two type-2 left parentheses are adjacent, the one on the right is the predecessor of the other.

(*ii*) If immediately right to a type-2 left parenthesis is a type-3 left parenthesis then go to the right match of the type-3 parenthesis, move to the right parenthesis immediately left to it, and find the left match of the last right parenthesis. The latter left parenthesis is the predecessor of the type-2 left parenthesis (see Figure 6.3).

(*iii*) If a type-1 parenthesis is immediately right to a type-2 parenthesis, the type-1 left parenthesis is the predecessor of the type-2 parenthesis.

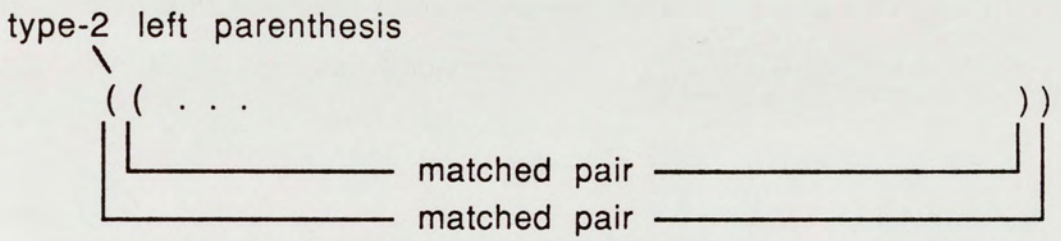


Figure 6.2.a. A Type-2 Left Parenthesis.

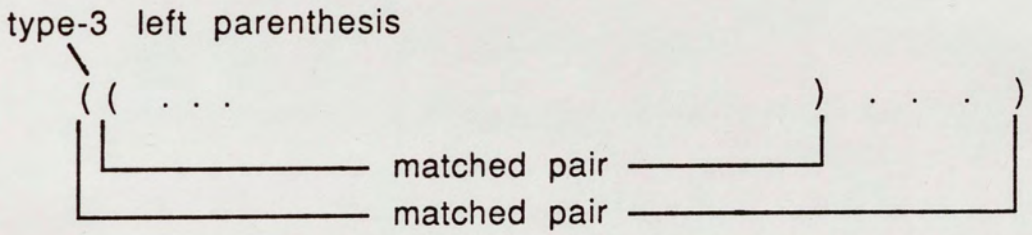


Figure 6.2.b. A Type-3 Left Parenthesis.

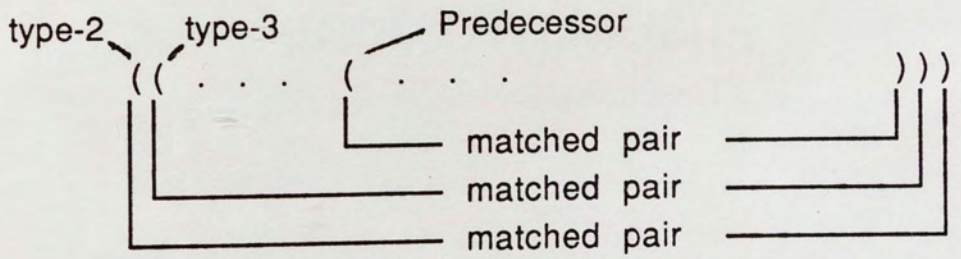


Figure 6.3. Predecessor of a Type-2 Parenthesis.

Now we present the parallel parsing algorithm.

Phase 1. String partitioning:

Step 1: Parenthesis insertion - Assign one processor to each terminal of the string. Every processor inserts parentheses to the left and right of the terminal (assigned to it) using the insertion table. This takes $O(1)$ time. The new string obtained after insertion of the parentheses is referred to as the string in the next steps.

Step 2: Parenthesis matching - Find the match for every parenthesis using the parenthesis matching algorithm described earlier. This step takes $O(\log L)$ time using $(L / \log L)$ processors.

Step 3: String partitioning - One processor is assigned to every type-1 left parenthesis. Every processor does the following:

(a) Collects nonterminals from the right until it finds either a left parenthesis or the right match of its own left parenthesis.

(b) If a processor finds a left parenthesis, it goes to the right match of the left parenthesis and inserts a nonterminal in its string according to the context and goes to Step 3(a).

Remark 3: Insertion of nonterminal is possible due to Constraint 2. In an implementation we can keep a reserved place for every processor. This

place is used to store the root of the subtree that the processor generates. A nonterminal insertion can be done by writing the nonterminal into the reserved place of the processor of the left parenthesis (for which nonterminal is inserted) and placing a pointer.

(c) If a processor finds the right match of its own left parenthesis, then it stops.

The length of the string collected by a processor is bounded by a constant.

Phase 2. Concurrent parsing:

Step 1: (*a*) One processor is assigned to every left parenthesis whose left terminal

$\underline{t_i} \in W_t$. It constructs a subtree taking the left son as the root of the subtree for the string enclosed in the matched pair on the left of $\underline{t_i}$ and the right son as the root of the subtree for the string enclosed in its own matched pair. The processor stores the root of the subtree in the reserved place for its root. In this step subtrees corresponding to all recursive productions are constructed.

(*b*) One processor is assigned to every left parenthesis that has a type-3 left parenthesis on its left. The processor inserts one left parenthesis to the right of its own left parenthesis and one right parenthesis to the left of the right match of its own left parenthesis. This pair of parentheses is inserted for a unit production from a recursive nonterminal to another nonterminal.

Step 2: One processor is assigned to every type-1 left parenthesis, and it constructs the subtree for the string it has collected in Step 3 of Phase 1. Subtrees corresponding to all block-structured, nonrecursive, and terminal productions are built here.

Step 3: One processor is assigned to every type-2 parenthesis. The processor waits for its predecessor to complete the construction of the subtree. When the predecessor completes the construction of the subtree, the root of the predecessor is the string for it. Every processor constructs a subtree corresponding to a unit production.

6.3 Discussion

We have presented an $O(\log n)$-time optimal parallel algorithm to find the match of a given parenthesis in a balanced sequence of parentheses. Our algorithm is neither a simulation nor an adaptation of any existing sequential algorithm. The algorithm has been designed from a simple observation.

Stack and queue lie at the heart of many sequential algorithms. These data structures appear to be the bottleneck in the parallelization of these algorithms. Thus, parallel algorithms for problems that have efficient sequential algorithms with stack and / or queue have to be designed from scratch in order to achieve appreciable speedup. Since most practical parsing algorithms use stacks, they are not suitable for

direct parallelization.

We have presented an entirely new parsing algorithm which is inherently parallel in nature. It is neither a simulation nor an adaptation of any existing sequential algorithm. To simplify the presentation of the algorithm, we assumed that L processors are available; then each step, except the parenthesis-matching step, requires $O(1)$ time. Thus, if we have only $(L / \log L)$ processors, these steps can be completed in $O(\log L)$ time. It has been shown that parenthesis-matching can be done in $O(\log L)$ time using $(L / \log L)$ processors. Therefore, the proposed algorithm can parse any string of length L of a class of block-structured languages in $O(\log L)$ time using $(L / \log L)$ processors. Hence, the algorithm is cost optimal within a multiplicative constant.

CHAPTER 7
CONCLUSIONS

The principal contributions of this work are a technique for estimating the speedup in parallel bottom-up parsing and a technique for parsing. The parsing technique permits the design of a cost-efficient polylogarithmic-time parallel algorithm.

The two models for parallel bottom-up parsing presented here are direct parallelization of existing sequential bottom-up parsing algorithms. The speedup obtained by these algorithms is input dependent. Hence, to measure the performance of these parallel bottom-up parsing algorithms, a method for estimating the speedup obtainable by bottom-up parallel parsing has been developed. To estimate speedup by this method, the probabilities of occurrences of different terminal symbols in the language and the production rules of the grammar are required. By using the developed technique, the speedup obtainable by parallel bottom-up parsing of Pascal-like languages has been studied. The study shows that maximum speedup of $O(L^{½})$ is obtained with $L^{½}$ processors.

It is suspected that use of stack as a data structure in sequential bottom-up parsing algorithms is the bottleneck in parallelization of these sequential algorithms. It is believed that sequential algorithms that use stack and / or queue as a data structure

are not parallelizable. For these problems parallel algorithms have to be designed from scratch.

A new technique has been developed for parsing a class of block-structured languages. The technique is inherently parallel and can parse a token-string of length L in $\log L$ time with $(L/\log L)$ processors. The string of tokens to be parsed is partitioned, in parallel, by inserting parentheses in the string and then by finding a match for each parenthesis. Each partition is parsed by one processor.

There are several directions for future work. Firstly, a technique should be developed for modifying a given language such that the modified language can be parsed using the technique developed here. Secondly, extension of the technique for generating codes using attribute trees could be considered, although, the extension may not be straightforward. Other phases of compilation, such as code improvement, should also be considered for parallelization. Future development in these areas will lay down a solid foundation for designing a fast parallel compiler.

LIST OF REFERENCES

Aho, A. V., P. J. Denning, and J. D. Ullman. "Weak and Mixed Strategy Precedence Parsing." *J. ACM* 19 (April 1972): 225-43.

Aho, A. V., and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling, Vol. 1: Parsing*. NJ: Prentice-Hall, 1972.

Aho, A. V., R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. MA: Addision-Wesley, 1986.

Baccelli, F., and T. Fleury. "On Parsing Arithmetic Expressions in a Multiprocessing Environment." *Acta Informatica* 17 (1982): 287-310.

Baer, J. -L., and C. S. Ellis. "Model, Design, and Evaluation of a Compiler for a Parallel Processing Environment." *IEEE Trans. Soft. Eng.* SE-3 (Nov. 1977): 394-405.

Bar-On, I., and U. Vishkin. "Optimal Parallel Generation of Computation Tree Form." *ACM Trans. Prog. Lang. and Syst.* 7 (April 1985): 348-57.

Brent, P. R. "The Parallel Evaluation of General Arithmetic Expressions." *J. ACM* 21 (April 1974): 201-6.

Chin, Y. F., and I. Chen. "Efficient Parallel Algorithms for Some Graph Problems." *Comm. ACM* 25 (Mar. 1982): 659-65.

Cohen, J., T. Hickey, and J. Katcoff. "Upper Bounds for Speedup in Parallel Parsing." *J.* ACM 29 (April 1982): 408-28.

Cohen, J., and S. Kolodner. "Estimating the Speedup in Parallel Parsing." *IEEE Trans. Soft. Eng.* SE-11 (Jan. 1985): 114-124.

Cohen, J., and M. S. Roth. "Analyses of Deterministic Parsing Algorithms." *Comm. ACM* 21 (June 1978): 448-58.

Colmerauer, A. "Total Precedence Relations." *J. ACM* 17 (Jan. 1970): 14-30.

Dekel, E., and S. Sahni. "Parallel Generation of Postfix and Tree Forms." *ACM Trans. Prog. Lang. Syst.* 5 (July 1983): 300-17.

Deo, N. *Graph Theory with Applications to Engineering and Computer Science.* NJ: Prentice-Hall, 1974.

Donegon, M. K., and S. W. Katzke. " Lexical Analysis and Parsing Techniques for Vector Machine," in *Proc. Conf. Prog. Lang. and Compilers for Parallel and Vector Machines* 10 (March 1975): 138-45.

Eickel, J., M. Paul, F. L. Bauer, and K. Samelson. "A Syntax-Controlled Generator of Formal Language Processors." *Comm. ACM* 6 (Aug. 1963): 451-55.

Ellis, C. A. "Parallel Compiling Techniques," in *Proc. ACM* (1971): 508-19.

Fischer, C. N. *On Parsing Context-Free Languages in Parallel Environments.* Ph. D. Dissertation, Cornell Univ., Ithaca, NY, April 1975.

Floyd, R. W. "Syntactic Analysis and Operator Precedence." *J. ACM* 10, (July 1963): 316-33.

Graham, S. L. "Extended Precedence Languages, Bounded Right Context Languages and Deterministic Languages," in *Proc. 11th Annual Symposium on Switching and Automata Theory* (1970): 175-80.

Gray, J. N., and M. A. Harrison. "On the Covering and Reduction Problems for Context-Free Grammars." *J. ACM* 19 (Aug. 1972): 675-98.

Hopcropt, J. E., and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* MA: Addison-Wesley, 1979.

Hwang, K., and F. A. Briggs. *Computer Architecture and Parallel Processing*. NY: McGraw-Hill, 1984.

Ichbiah, J., and S. Morse. "A Technique for Generating Almost Optimal Floyd-Evans Productions of Precedence Grammars." *Comm. ACM* 13 (1970): 501-08.

Kernighan, B. W., and P. J. Plauger. *The Elements of Programming Style*. NY: McGraw-Hill, 1974.

Knuth, D. E. "A History of Writing Compilers." *Comput. Autom.* (1962): 6-19.

Knuth, D. E. "On the Translation of Languages from Left to Right." *Information and Control* 8 (1965): 607-39.

Knuth, D. E. "Top-Down Syntax Analysis." *Acta Informatica* 1 (1971): 79-110.

Krishnamoorthy, M. S., and N. Deo. "Node-Deletion NP-Complete Problems." *SIAM J. COMPUT,* 8 (Nov. 1979): 619-26.

Krohn, H. E. "A Parallel Approach to Code Generation for Fortran-like Compilers," in *Proc. Conf. Prog. Lang. and Compilers for Parallel and Vector Machines,* 10 (March 1975): 146-52.

Kuck, D. J. "A Survey of Parallel Machine Organization and Programming." *ACM Comput. Surveys* 9 (March 1977): 29-59.

Ligett, D., G. McCluskey, and W. M. McKeeman. "Parallel LR Parsing." Wang Institute of Graduate Studies, School of Information Technology, Tech. Report TR-82-03, July 1982.

Lincoln, N. "Parallel Programming Techniques for Compilers." *SIGPLAN Notices* 5 (Oct. 1970): 18-31.

Lipkie, D. E. *A Compiler Design for Multiple Independent Processor Computer*. Ph. D. Dissertation, Univ. Washington, Seattle, WA, 1979.

Loka, R. R. "A Note on Parallel Parsing." *SIGPLAN Notices* 19 (Jan. 1984): 57-59.

McKeeman, W. M. "An Approach to Computer Language Design." Ph.D. Dissertation, Stanford Univ., Stanford, California, 1966.

McKeeman, W. M., J. J. Horning, and D. B. Wortman. *A Compiler Generator.* NJ: Prentice-Hall, 1970.

Mickunas, M. D., and J. A. Modry. "Automatic Error Recovery for LR-Parsers." *Comm. ACM* 21 (June 1978): 459-65.

Mickunas, M. D., and R. M. Schell. "Parallel Compilation in a Multiprocessor Environment," in *Proc. ACM* (1978): 241-46.

Miller, G. L., and J. H. Reif. "Parallel Tree Contraction and Its Application," in *Proc. Symposium on Foundations of Computer Science* (1985): 478-89.

Schell, R. M. *Methods for Constructing Parallel Compilers for Use in a Multiprocessor Environment.* Ph. D. Dissertation, Univ. Illinois, Urbana, IL, 1979.

Sarkar, D., and N. Deo. "Estimating the Speedup in Parallel Parsing," in *Proc. Int. Conf. Parallel Processing* (Aug. 1986): 157-63.

Vishkin, U. "An Optimal Parallel Connectivity Algorithm." *Discrete Appl. Math.* 9 (1984): 197-207.

Wirth, N., and H. Weber. "Eulier -- A Generalization of Algol and its Formal Definition, Parts 1 and 2." *Comm. ACM* 9 (Jan. & Feb. 1966): 13-23, 89-99.

Wyllie, J. C. "The Complexity of Parallel Computation." TR 79-387, Dept. Comput. Sci., Cornell Univ., Ithaca, NY, 1979.

Zosel, M. "A Parallel Approach to Compilation," in *Proc. ACM Symposium on the Principles of Programming Languages* (1973): 59-70.