A Bibliography on Parallel Parsing

Henk Alblas, Rieks op den Akker,
Paul Oude Luttighuis, and Klaas Sikkel
Department of Computer Science, University of Twente,
P.O. Box 217, 7500 AE Enschede, The Netherlands
{alblas,infrieks,oudelutt,sikkel}@cs.utwente.nl

Abstract

This bibliography on parallel parsing and recognition of context-free languages covers most of the important publications in this area of computer science and natural language processing. The list of publications is preluded by an introduction which presents an overview of the field.

1 Introduction

This bibliography on parallel parsing is a modified and compacted version of the annotated bibliography of Op den Akker et al.[1992], from which all annotations have been removed, as well as pre- and re-publications and papers that were not available to us. Readers interested in the complete annotated version are invited to contact the authors. During the last few years we have seen a tremendous number of papers related to parsing and recognition of (mostly very restricted) context-free grammars in the area of connectionist or neural networks. These networks do not only parse or recognition but simulate recognition of semantical and pragmatically aspects of words, or phrases in natural language utterances as well. The results obtained in this area are not covered.

If we look at the publications in the area of parallel parsing we may distinguish four different approaches or interests:

- There are papers which are rather theoretically concerned with language recognition and sometimes parsing. These papers introduce several models of parallel computation and try to relate these models with other models or with well-known classes of languages. One of the questions considered in these papers is for which subclasses of the class of context-free languages recognition can be optimally parallelized.

- Several papers concern parallelization of general context-free language parsing methods. Here we find methods for parallel CYK and Earley parsing or variants of these and proposals for systolic array or VLSI implementations of these parsing methods. Especially in the field of natural language processing emphasis is on these general parsing methods. Some papers focus attention to methods for language specification suitable for parallel parsing.

- This bibliography covers also papers which consider parallel parsing in a broader context of complete parallel compiler systems. In the following we will give a short overview of these different fields of interest.

2 Parallel Parsing of Programming Languages

This section describes work done in the area of parallel lexing and parsing of programming languages. Papers in this area can be categorized on the basis of several criteria:

1. compilation phase: lexing, parsing or both;
2. language class: arithmetic expressions, bracket matching, more general language classes;
3. parallel machine model: vector computer, shared memory model (SIMD, P-RAM), hypercube, mesh-, shuffle-, cube-connected and cube-connected cycles;
4. starting-point: sequential algorithms, or not;
5. number of processors: fixed, or depending on grammar or string size;
6. decomposed domain: grammar or string decomposition.

We will not make a full 6-dimensional split-up of this area according to these six criteria. Instead, our division of the subject obeys a tree-like structure, so that some of the sub-areas may be scattered over this tree. Our main split-up concerns the compilation phase under consideration.

Lexing


Lexing and parsing

Pronina and Chudin[1975] use a pipeline between a lexing and a parsing processor.

Parsing

In parallel parsing, two approaches may be distinguished. The first approach uses a decomposition of the grammar and lets processors be responsible for parts of the grammar, that is, for parsing as far as productions in its own part of the grammar are concerned. The other approach just distributes the string over processors, which all parse their part according to the whole grammar. An advantage of the first method is, that actual parsing is facilitated, because the string can be cut at convenient
places. Advantages of the second method are that distribution of the string is easy and that the distribution can freely be done in such a way that processors get more or less equally long parts of the string to handle.

Grammar decomposition.

Many papers using the grammar-decomposition approach originate from France. In Baccelli and Fleury[1982], Fleury[1983] and Baccelli and Musy[1986], the string is partitioned according to some decomposition of the grammar. Partitioning is performed (sequentially) by a host process. After that, different processes perform local parsing (Baccelli and Fleury[1982]) or interpreting (Baccelli and Musy[1986]). A similar idea appears in Khanna and Ghafoor[1990].

String decomposition.

The string decomposition approach is used by the vast majority of papers on this topic. Probably, this is because it allows for using arbitrary numbers of processors, that is, a number not fixed (by the algorithm or the grammar). Again, we split up these papers in several groups.

1. papers that use bracket matching as (part of) their algorithm;
2. papers that discuss parallel bottom-up parsing;
3. papers that use the parallel prefix sum algorithm;
4. other papers

Bracket matching. In the early '80's, work was done on parallel parsing of arithmetic expressions. In Dekel and Sahni[1983], an algorithm for this problem was presented. The algorithm runs on an EREW P-RAM. Then, Bar-On and Vishkin[1985] described a faster, even optimally parallel, algorithm, yet running on a less restricted machine model, the CREW P-RAM. By way of the most important part of the CREW P-RAM algorithm is the bracket matching part. It was shown that this matching can be done in O(\log n) time using O(n/\log n) processors.

The bracket matching problem has appeared to be a key problem in parallel parsing. This is because it is the bracket-like construction (embedding) that distinguishes context-free languages from regular ones. A vast range of papers is dedicated to this sole problem of parallel bracket matching. Sarkar and Deo[1988] discuss another optimally parallel CREW P-RAM bracket matching algorithm, which is essentially the same as the one in Matthesius and Fiduccia[1982]. However, in this last paper, it was not recognized that only O(n/\log n) instead of \( n \) processors were needed. See also Ryter and Giancarlo[1987]. The first paper to present an optimally parallel EREW P-RAM algorithm for this problem was Tsang et al.[1989]. Later papers, Deo et al.[1990], Prasad and Deo[1991], Chen and Das[1990], and Das et al.[1991] present EREW P-RAM algorithms that are claimed to be simpler, but still optimally parallel. Pitsch and Schömer[1991] present optimal (average case) parallel recognition of bracket languages on hypercubes.

By the existence of optimal EREW P-RAM bracket matching algorithms, it can be shown that parsing arithmetic expressions can be done optimally in parallel on an EREW P-RAM.

Other papers to study parallel parsing of arithmetic expressions are Srikant and Shankar[1985,1987b] and Srikant[1990]. In the first two of those three papers, an algorithm that takes O(\log^2 n) time using n processors is presented. The other one contains an algorithm for mesh, cube and cube-connected cycles parallel computers, resulting in different time bounds.

Bracket matching is achieved by sorting. Srikant and Shankar[1987a] present a new way of specifying the syntax of programming languages, called Hierarchical Language Specification. Basically, the regular constructs are separated from the bracket parts and treated differently. Bracket matching can be used to parse much larger classes of context-free grammars. In Sarkar and Deo[1987], brackets are inserted in the strings of what they call restricted block-structured context-free languages.

Optimally parallel bracket matching algorithms are also used in Oude Luttikhuis[1989,1990a,1991]. The first two of these papers deal with parsing regular right-part grammars. The algorithm works for regular right-part grammars that generate a subclass of the LL(1) languages. It constructs the depth-first left-to-right traversal of the parse tree before building the parse tree first. Then, matching the steps down with the steps up in this traversal (using a bracket matching algorithm) yields the parse tree. The third paper uses context-free grammars that satisfy the LE(p, q) constraint. They constitute almost the class of all LL(k) grammars. The algorithm extracts pieces of the parse tree from a precomputed table, taking into consideration a look-back and look-ahead string. One piece corresponds to a series of production applications followed by one shift in the LL algorithm. The LE(p, q) classes constitute the largest class of CFLs, yet known to be parseable in optimally parallel time.

Bottom-up parsing. The first substantial work to have appeared on parallel parsing is found in Fischer[1975b]. One of the algorithms in this Ph.D. thesis is a parallel parsing method based on LR parsing. It uses a number of (sequential) parsers, all starting at some point in the string. A problem here is the state in which each of the parsers should start. The proposed solution is to maintain a parsing stack for every possible state of the (sequential) parser starts in, considering the point at which it starts. When the stack is too small to perform some reduction, the parser merges his stack with the stack of a neighboring parser and dies. Similar ideas appear in Pronina and Trakhtengerts[1974], Mickunas and Schell[1978], Schell[1979] and (very briefly) in Carlisle and Friesen[1985].

In his Ph.D. thesis, Gafter[1990b], the author describes parallel algorithms for incremental lexical, syntactical and semantic analysis. A usual text editor is stipulated to yield a log of edit actions. This log is used to change the old parse tree. For parsing, an adaptation of Schell[1979]'s LR parallel parsing algorithm is used. It is recognized that the depth of the parse tree is a bottleneck in speeding up parallel parsing. This depth is especially increased by numerous applications of list productions. The proposed solution to this is to use, instead of left or right recursive productions, rules of the form A → AA, so that logarithmic depth is achieved. To overcome the extreme ambiguity caused by these rules, the LR algorithm and items are adjusted. A formalism to express the parsing structure of programming languages, that allows for parallelization, called upward remote aggregates, is introduced. Gafter[1987,1990a] can be found in this thesis. Parallel incremental LR parsing is also discussed in Viswanathan and Srikant[1991].

Parallel prefix sum. The parallel prefix sum algorithm can also be used for parsing context-free grammars, as shown by Skillcorn and Barnard[1989]. This paper uses the parallel prefix sum algorithm to combine sets of stack transitions. These stack transitions are pairs of stack modifications and indicate how the stack changes (during LL(1) parsing) during processing of a
substrings. Stack transition sets corresponding to substrings are pairwise combined to yield the stack transition set corresponding to the concatenation of the substrings. The claim of the authors that the algorithm runs in $O(\log n)$ time using $n$ processors is incorrect however. Even when the used grammar is LL(1), the sets that occur during combination might show exponential explosion. See Oude Luttikhuis [1990b] and Hill [1992]. The latter of these discusses an implementation of the parsing algorithm on a distributed processor array with a two-dimensional topology.

Other papers. In Lampe [1990b], parallelizing parsing is restricted to grammars that allow to make parsing decisions on the basis of very local context information. Within the grammar, local grammars are distinguished, which have to have this local parsability property (within the overall grammar).

In Loka [1984] and Tseytlin and Yushchenko [1977], one processor starts parsing from left to right at the beginning of the string, the other starts at the other end, parsing in the opposite direction. Somewhere in the middle, they meet and their parses are combined.

Fischer [1975b] also contains a vector precedence parsing algorithm and a vector algorithm for parsing (and compiling) arithmetic expressions. This last algorithm also appears in Fischer [1975a, 1980].

3 Parallel Context-Free Parsing in Natural Language Processing

In natural language processing circles there is a wide-spread interest in parallel processing techniques for language analysis. Unlike the compiler construction environment with its generally accepted theories, in natural language processing no generally advocated — and accepted — theory of natural language analysis and understanding is available. Therefore it is not only the desire to exploit parallelism for the improvement of speed, but it is also the assumption that human sentence processing is of an inherently parallel nature which makes computer linguists and cognitive scientists turn to parallel approaches for their problems. For these reasons, in natural language processing many kinds of parallel approaches can be distinguished. While some researchers aim at cognitive simulation, others are satisfied with high performance language systems. The former researchers may ultimately ask for numbers of processors and connections between processors that approximate the number of neurons and their connections in the human brain (that is, an order of $10^3$ neurons with $10^5$-$10^6$ connections each). They model human language processing with connectionist models and therefore they are interested in massive parallelism, distributed representation of knowledge and low degradation of overall behavior in the face of local errors. We will not discuss such approaches to language processing, because they are of a too different nature. Connectionist implementations of conventional approaches to parsing have been proposed. These can be seen as boolean circuit implementations of parallel parsing algorithms. For sentences up to some maximum length, these algorithms can be coded directly into hardware. See, e.g., Fanty [1985], Nijholt [1990], Sikkel and Nijholt [1991].

Any system used for understanding natural language sentences needs to distinguish different levels of analysis, e.g. analysis at the morphological, the lexical, the syntactic, the semantic and the referential level. For each level a different kind of knowledge has to be invoked. Therefore different tasks can be distinguished: the application of morphological knowledge, the application of lexical knowledge, etc. It is not necessarily the case that application of one type of knowledge is under control of one of the other types of knowledge. The tasks may interact and at times they can be performed simultaneously. Therefore processors which can work in parallel and which can communicate with each other may be assigned to these tasks in order to perform this interplay of multiple sources of knowledge. Hence, this task-oriented parallelism requires smart processors performing specialized functions.

Finally, and independent of a parallel nature that can be recognized in the domain of language processing, since operating in parallel with a collection of processors can achieve substantial speed-ups, designers and those who implement natural language processing systems will consider the application of available parallel processing power for any task or subtask which allows that application. We distinguish two views on this, not necessarily linguistically motivated parallelism.

The first view is the programming language point of view, the second is the processor or process point of view. In the first view the starting point is the programming language and the problem of language analysis is programmed using the parallel concepts that are offered by the programming language. For instance, parallel logic programming languages such as GHC (Guarded Horn Clauses), Parlog and Concurrent Prolog invite us to approach the analysis problem from a parallel point of view. Many natural language processing systems based on these parallel logic programming languages have been built. In Matusov [1987] and Matusov and Sugimura [1987] examples of this approach, using a left-corner method, can be found. In Tanaka and Numazaki [1989] GHC is used to implement a parallel version of Tomita's generalized LR parsing algorithm. In the papers of Matusov the terminal and nonterminal symbols are defined as parallel processes. In the paper of Tanaka and Numazaki each LR-table entry is defined as a process. Similarly, a parallel object-oriented point of view can be advocated, as is done by Yonezawa and Ohnawa [1989]. They use the parallel object-oriented programming language ABCL to implement a parsing system which is obtained by translating a collection of context-free grammar rules into a configuration of message passing, cooperating units (agents). Each occurrence of a terminal or nonterminal symbol in a grammar rule is represented as an agent in the system, the messages that are sent consist of control data or partial parse trees. Designing parsing methods from the point of view of available or desired processor configurations has led to a variety of methods. Attacking the context-free parsing problem with more than one processor almost always means using identical processors, that is, processors that run the same software and compute the same function. In some cases this can amount to having several asynchronous parsers working on the same input string (e.g., Fischer [1975], Loka [1984], Liskoviskii and Nirenburg [1986], Sikkel and Lankhorst [1992]).

Traditional parsing methods have been re-investigated in order to see whether they can be adapted to a parallel processing view. For the Earley and the Cocke-Younger-Kasami (CYK) parsing methods implementations have been designed for multi-processor shared-memory computers, for pipelines of processors and for arrays of processors. A survey and an explanation of these methods is given in Nijholt [1989, 1990b].

4 Complexity of Parallel Context-Free Language Recognition and Parsing

The main practical reason for introducing parallel algorithms for problems which are not intrinsically parallel is to decrease the processing time needed by sequential algorithms. There is, besides
this practical motivation, a more theoretical question which motivates investigations after parallel solutions for particular problems. Is it possible to find a parallel solution which is essentially better, as far as processing time is concerned, than sequential solutions for the same problem? Or, stated in a more general way, what kind of operations or problems are intrinsically sequential and which are not? The specific question here is then, can we reduce time complexity by using parallelism in (context-free) language recognition and parsing?

In parallel complexity theory processors are considered as a resource just like time and space in sequential complexity theory. Trends in hardware technology suggest that it becomes more and more reasonable to regard the number of simple processors available for a job as unbounded. As an example of this technology, may serve the Connection Machine which has ten thousands of processors.

For a good parallel algorithm the product of (parallel) time and the number of processors used, will be of the same order of magnitude as the time complexity of a good sequential solution. An algorithm satisfying this is often called optimally parallel. The theory of context-free parsing has emerged from the practice of making automatic parsers and translators for programming or natural languages. In the area of compiler writing systems, it is sometimes said that if someone writes a grammar for a programming language it is almost LR(1) and with some minor changes it can be made to be LALR(1). This means that such a language can be parsed in linear time by a rather space efficient sequential parsing method. Since the existence of Yacc and other compiler generating tools there is hardly any practical motivation within this area of computer science for using parallelism. The situation seems to be different in the area of natural language processing, where traditionally one is more interested in grammars which are not restricted by some tools for automatic translators. Earley’s and Cocke-Younger-Kasami’s methods for recognition and parsing of general context-free languages are often used and studied in this field. Valiant’s variant of the CYK-method which runs in time $O(n^{3.5})$ is the fastest sequential parsing algorithm for general CFLs. Earley’s method takes time $O(n^2)$, but it has the advantage that it does not need a grammar in Chomsky Normal Form. Particularly, if one is dealing with parsing and translating general CFLs one may have a practical interest for speeding up processing time by using parallel algorithms and parallel architectures. About the same time as Valiant came up with his sequential algorithm, Kosaraju[1975] showed that the CYK-method can be used to recognize context-free languages in $O(n^2)$ time on a two-dimensional array of processors. A lot of papers are devoted to parallel versions and implementations of Earley’s method. This shows that this can be done in linear time by using $O(n^2)$ processors in $O(n^2)$ of total space. Recently, de Vreught and Honig[1989] came up with a general recognition method which resembles the tabular Earley algorithm. Instead of one, as usual, items have two dots, as in Bossi et al.[1983]. The method works bottom-up and doesn’t use the predictor operation. A parallel algorithm for a P-RAM shows that this method has a parallel solution with complexity properties like those for Earley’s method and the CYK-method.

Confronted with several parallel algorithms for different particular architectures the problem arises how we can compare them. Suppose we have a polynomial parallel time solution on a particular model of parallel computations. Does this mean that we also have a polynomial solution on any other parallel machine model? How are different models related to each other? There is a large variety of models for sequential computation, all of which can be used as defining the class of computable functions. Best known are the Turing machines (TM) and the random access machines (RAM). A lot of these models have a measure for computation resources (like time and space) such that whenever a machine $M$ in class $A$ runs in time $T$ and space $S$, then there is a simulating machine $M'$ in class $B$, such that the time used for the simulation of $M$ is polynomially related to $T$ and the space used by $M'$ is of the same order of magnitude as $S$.

For a "reasonable" model of sequentially computation there are some complexity classes, the definition of which is invariant with respect to the specific model used. These classes form the following hierarchy of sequential complexity classes.

\[
\text{LOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} = \text{NPSPACE} \subseteq \text{EXPSPACE} \subseteq \text{NEXPTIME}
\]

It is open whether the inclusions are really proper or not. The equality in this sequence is the well-known theorem of Savitch.

For parallel computation there is even a greater variety of "reasonable" machine models. A model is called "reasonable" with respect to parallel computation if it satisfies the Parallel Complexity Thesis. This thesis states that parallel time is polynomially related on all reasonable models. Moreover time on such a model should be polynomially equivalent to (log-cost) space on a sequential model. This means that the following equalities should hold:

\[
\text{par-LOGSPACE} = \text{P} \\
\text{par-POLYLOGTIME} = \text{POLYLOGSPACE} \\
\text{par-PTIME} = \text{PSPACE} \\
\text{par-NPTIME} = \text{NPSPACE}
\]

and so as a consequence of Savitch theorem $\text{par-PTIME} = \text{par-NPTIME}$. Thus deterministic parallelism can give more than polynomial speed-up if $\text{PTIME} = \text{PSPACE}$. An often used model of parallelism is the CREW or P-RAM introduced in Fortune and Wyllie[1978] or in Larmore and Wyllie[1983] like the SIMDAG model of Goldschlager or the CREW or W-RAM in which multiple wires into the same global memory location are allowed. Other "reasonable" models are combinatorial circuits (cf. Borodin[1977] and Ruzzo[1981]) and alternating Turing machines (cf. Ruzzo[1979] and Ibarra et al.[1988]).

As far as the complexity of context-free recognition and parsing is concerned the situation is as follows. An obvious lower-bound for sequential time is linear. The best known algorithms have more than quadratic time complexity (Earley’s algorithm, CYK-algorithm). Each CFL can be recognized using polynomial time and $n^2$ space. Some subclasses of DCFL (the class of deterministic context-free languages) are known to be in DSPACE($\log^n n$), bracket languages (Melhorn, Information Processing Letters 5(6), 1976) and input-driven languages (Van Braunmühl and Verbeek, Proceedings Fundamentals of Computer Theory 1983, 40–51). See also Ryter[1986a]. For deterministic CFLs, Cook[1979] was the first in which is presented a polynomial time and simultaneous $\log^2 n$ space recognition algorithm. It is not known whether all DCFLs are recognizable in $\log n$ space. It is also not known whether all CFLs are in the class SC (simultaneously polynomial time and poly-log space). Ruzzo[1979, 1981] has shown that they are in NC$^2$. In Larmore and Ryter[1992] sublinear parallel time algorithms are presented for the recognition of linear, unambiguous, and deterministic context-free languages are presented. These results imply an improvement of the known NC algorithms for dynamic programming.

NC$^2$ is the class of languages recognizable in $O(\log^4 n)$ time by a polynomial number of processors (see Pippenger[1979] and Ruzzo[1981]).
There are several methods for proving upper bounds for language recognition. One method is simulation of a deterministic PDA, Cook [1979] gives a simulation of a DPDA on a DTM within space \(O(\log^2 n)\) and time \(O(n^4 \log^2 n)\). Von Braunmühl and Verbeek [1980] improved Cook's result by giving an algorithm that works on a DTM in time \(O(n^2 \log^2 n)\) and space \(O(n \log^2 n)\). In Von Braunmühl, Cook, Mehlhorn and Verbeek [1983] a new algorithm is presented for DPDA simulation on a DTM within the same time and space bounds. The idea is to simulate a divide-and-conquer algorithm for checking whether \(C_{initial} \overset{+}{\to} C_{accept}\) on the DT.

A parallel simulation of a DPDA is also presented in Reif [1982] which results in an \(O(\log n)\) algorithm for recognition of deterministic CFLs. This method works only very practical. Optimal parallel recognition methods are known for bracket languages and input-driven languages (Gibbons and Ryter [1989]) and for Dyck languages (Matthesyes and Fiduccia [1982]) and regular languages. In Ibarra [1988] it is shown that the classes of one-sided Dyck languages, bracket languages and a subclass of the linear languages are in \(NC^1\).

Optimal parallel parsing methods are known for bracket languages (Ryter and Giancarlo [1987]), Dyck languages (Matthesyes and Fiduccia [1982]) and for DFLR(1) and LER(p, q) languages, both subclasses of the class of LL(1) languages (Oude Luttighuijs [1989, 1990a]). In all these papers an optimal parallel method for matching of parentheses like that presented in Bar-On and Vishkin [1985] is essential.

Some methods for general CFL recognition which lead to \(n\) parallel algorithms are based on the following idea. Given a string \(a_0 \cdots a_n\) and a context free grammar \(G\). The problem of recognition of \(L(G)\) can be formulated as deriving a statement in a theory. There are \(n \) axioms \(a_0\), denoted by \(r = (i, a_i, i + 1)\) and the rules are given by the production rules of \(G\). If \(A \overset{+}{\to} BCD\) is a production then, we have \((B, j), (C, k), (k, D, p)\) \(\overset{+}{\to} (i, A, p)\) as a derivation rule in our theory. Since we only have to look at rules with \(i \leq j \leq k \leq p\) we have a polynomial number of rules. We have to prove that \(\overset{+}{\to} (0, S, n)\) which denotes that \(a_0 \cdots a_n\) can be derived from \(S\), the start symbol of \(G\). For grammars in CNF, this view is worked out in Brent and Goldschlager [1985] and Ryter [1987b]. Ryter shows by using this method that unambiguous CFLs can be recognized in \(O(\log n)\) time on a \(P\)-RAM time by using \(O(\log n)\) processors. Each triple \((i, A, j)\) is assigned a processor. Brent and Goldschlager use a similar method for the SIMDAG model and show that any CFL can be recognized and parsed within the same complexity bounds, i.e. by \(O(\log n)\) processors in \(O(\log n)\) time.

The amount of processors seems to be too large to be practical. Some authors do not carefully distinguish recognition and parsing. In Ryter [1987a] it is shown that when we consider parallel time then parsing is indeed not harder than recognition, contrary to the sequential case. See also Oude Luttighuijs [1991] in which it is shown that on a \(P\)-RAM there is no essential difference in complexities between parallel parsers yielding derivations and those yielding parse trees. A quit assumption made in all the literature on parallel parsing.

Another bibliography on the theory of parallel recognition and parsing is de Vreught [1991]. Here you can also find a short overview of work in this area.

5 Parallel Compilation

Although this bibliography mainly concentrates on parallel parsing, it also includes papers on parallel compilation in general. Compilers are among the tools most heavily used by programmers. Therefore, the speed of compilation is directly related to programmer productivity. One way to speed up compilation is to exploit parallelism. Other techniques to reduce compile time are incremental compilation and separate compilation of program modules. Parallel compilation can make both of these techniques faster. As parallel processing technology advances, concurrency becomes an attractive vehicle by which compilation speed may be increased.

Theoretical research in the field of parallel compilation has dealt mainly with its best understood area — parsing regular and context-free grammars (e.g., Mckunis and Schell [1978]). The first significant theoretical investigation into other phases of compilation has been the work of Schell [1979]. In addition to lexical and syntactic analysis, his thesis examined semantic analysis based on parallel attribute evaluation. Any practical investigations in concurrent compilation have naturally been constrained to the multiprocessing hardware available at the time. Early efforts were aimed at vector processors, while more recent research has been directed towards loosely coupled distributed systems and more tightly coupled MIMD multiprocessors.

The first practical attempts to parallelize compilation date back to the early 1970s. Most of these efforts were restricted to making certain phases of compilation concurrent for a specific language (i.e., FORTRAN, APL) on vector processors (e.g., CDCSTAR-100), which were the only multiprocessors available at the time. Lincoln [1970] showed how parallel techniques for lexical analysis and parsing of FORTRAN programs may be expressed by APL vector operations.

Donegan and Katzke [1975] elaborated Lincoln's theme and Krohn [1975] extended it to code generation. Similar attempts to apply vector processing techniques to compilation were reported by Ellis [1971] and Zosel [1973]. See also House [1981].

Local area network technology later in the 1970s, offered prospects for more coarsely grained parallel compiler designs. The earlier efforts which utilized this kind of distributed processing concentrated on a pipelined approach to concurrent compilation. Pipelining is a common technique to exploit concurrency in tasks that can be divided into a series of stages. The stages are cascaded so that the output of one stage is the input for the next. Some form of buffering is needed between each pair of stages to facilitate the transfer of information from one stage to the next. If each stage runs concurrently, the maximum speedup is equal to the number of stages. The easiest way in which to transform a sequential compiler into a concurrent compiler is to run the compiler phases as separate processes in a pipeline, and to have them communicate through shared data structures. This approach has two limitations. First, the degree of concurrency is limited by the number of stages in the pipeline, which places a hard limit on the number of processors, thus potentially leaving available processors idle. Second, the overall speed of compilation is constrained by the speed of the slowest pipeline stage. Many compilers spend a large fraction of their execution time in one or two phases. Much of the effort to implement an efficient pipeline may go to balancing the different stages. Consequently, pipelining the phases on a multi-processor will generally yield a less than ideal speedup.

Designs for pipelined compilers have been presented in several papers. Baer and Ellis [1977] discussed a graph model based on a generalized Petri Net to describe the control and data flow of an existing compiler. The modelled version of the compiler was modified so that its sequential structure could be easily transformed into a pipeline of processes. Miller and LeBlanc [1982] compared for a subset of Pascal a sequential single-pass compiler with a distributed version. The stages in their pipeline consisted of a lexical analyzer, a syntactic analyzer and a semantic analyzer/code generator which emitted intermediate code. Huen et al. [1977] described the design of a pipelined DYNAMO compiler which ran on a network computer. The compiler produced parallel code segments for the same network computer. The object code was automatically partitioned into
clusters so that the clusters could execute in parallel on the individual computers in the network computer. See also Christopher et al [1981].

In this determination of which phases of the compilation process can be carried out concurrently, one may construct a compiler which splits the source program into segments which are then compiled concurrently. These segments would typically be well-defined syntactic constructs, such as procedure bodies or statements lists. From the point of view of semantic analysis, the source code must be partitioned in such a way as to minimize the amount of inter-process communication. Unlike the pipelined scheme, this approach does not inherently limit the amount of concurrency which can be exploited using the underlying hardware — that is, the amount of concurrency which exists in the compiler is determined to a greater extent by the characteristics of the source program being compiled and the underlying hardware, rather than by the structure of the compiler.

These efforts have been targeted at both loosely coupled uniprocessor networks and more recently developed shared MIMD machines.

Lipkie[1979] was probably the first to suggest a combination of pipelining and concurrent processing of program segments. Lipkie proposed the design of a parallel compiler which broke the source code at procedure boundaries. The source code division would occur during parsing, after a sequential prescanning phase. Each procedure would be processed concurrently with the others, with the resultant code merged at the end. For each procedure definition a pipeline would be instantiated. Lipkie’s thesis did not report on any implementation.

A related, but more general approach, can be found in the thesis of Messerer[1982]. The source program is subdivided into fragments. During the combined lexical and structural analysis of any fragment a partial structure graph is built whose nodes represent blocks, procedures and control structures. From the partial graphs a total graph is composed which reflects the structure of the entire program. This graph indicates how the program can be subdivided into blocks and control structures which can be analyzed in parallel. Experimental results are not reported.

Frankel[1983] extended a one-pass recursive descent Pascal compiler to translate procedures concurrently. Whenever a compiler instance encounters a child scope while compiling a parent scope, it creates a new instance to compile the child scope and skips to the end of the child scope by matching delimiters.

Vandeloeve[C1988a,b] constructed a concurrent C compiler that runs on a tightly coupled multiprocessor. The compiler consists of a two-stage pipeline. The first stage performs extended lexical analysis for the second stage, which does the parsing, semantic analysis, and code generation. The second stage processes units of the source program concurrently. The division of the source code is accomplished in a manner similar to that of Frankel’s compiler. However, Vandeloeve’s compiler incorporates a finer grain of parallelism. Large statements are broken into smaller units which are processed concurrently through semantic analysis and code generation. Vandeloeve’s compiler differs from previous concurrent compilers in several ways. First, it extends the role of the scanner to include matching delimiters for the parser. This allows the parser to skip to the end of a child scope quickly. Second, Vandeloeve’s compiler uses finer grains of parallelism than have been used on distributed systems. Previous compilers have processed only procedures in parallel. Vandeloeve’s compiler processes units as small as a single statement concurrently. Finally, it runs on a tightly coupled multi-processor. Since such computers have only recently become widely available, most previous parallel compilers have been built on top of distributed workstations.

Seshadri et al[1988a,b,c,1991], Junkin and Wortman[1990], Wortman[1990], and Wortman and Junkin[1992] used a similar approach to build parallel Modula–2+ compilers to run on a distributed system. They do not restrict themselves to the parallel compilation of statements, but they investigate the concurrent processing of declarations as well. The later introduces what they call the “Doesn’t Know Yet” problem: the compiler may attempt to access information introduced by a declaration before it has been processed. Other implementations of parallel compilers are reported by Brezany[1986] and Gross et al[1989].

Attribute grammars have proved to be a useful formalism for specifying the syntax and the semantics of programming languages, as well as for implementing editors, compilers, translator writing systems and compiler generators. For a given syntax tree the evaluation order of its attributes is only restricted by the partial order induced by its dependency graph. A sequential evaluator completely serializes this order. Parallel evaluators are based on partial orders only. The thesis of Schell[1979] includes a chapter devoted to parallel tree-walk evaluation. His work is of a design nature, and did not result in any implementation. Practical experiments were reported by Jourdan and Marmol, Lampe, Kaplan and Kaiser, Boehm and Zwanepoel, and Kuiper and Dijkstra. Jourdan[1991] and Marmol[1990] independently rediscovered the method of Schell and reported a simple and efficient implementation. Lampe[1988,1990b] reported concurrent attribute evaluation based on an event-driven control scheme. Kaplan and Kaiser[1986] presented a model for distributed program editing. Their algorithm for incremental attribute evaluation allows multiple asynchronous edits on program modules that are distributed across a number of workstations connected by a high-speed network. Boehm and Zwanepoel[1987] developed a compiler-generator which produces a sequential parser and a number of parallel attribute evaluators from a single attribute grammar specification. The parser builds the syntax tree, divides it into subtrees, and sends them to the attribute evaluators. The attribute evaluators then proceed with the actual translation by evaluating attributes belonging to the symbols in their subtree, transmitting values of shared attributes as necessary. For efficiency, both static and dynamic attribute evaluators are used. Static evaluation is used to evaluate all attributes local to a subtree. Dynamic evaluation is applied for attributes which are shared between subtrees. Bohem and Zwanepoel used this method to generate a compiler for a large Pascal subset, which runs on a collection of up to 8 SUN–2 workstations connected by an Ethernet. Altbas[1990] proposes a combination of Boehm and Zwanepoel’s scheme with that of Kaplan and Kaiser for the incremental reevaluation of attributes after a tree transformation. He also formulates criteria based on safe approximations of a consistently attributed tree which permit a delay in calling the reevaluator until after several tree modifications. Thus, different reevaluations and different tree transformations may occur asynchronously and concurrently in different regions. The only constraint is that each region is either in its transformation phase at the cost of adding extra attributes (see also Kuiper[1988a]). Experiments with attribute evaluation on a network of transputers are reported by Kuiper and Dijkstra[1989]. Klein[1990,1991,1992] describes an approach based on the notion of segments. The dependency graph of each production is statically partitioned into segments, with the condition that each two attributes of a nonterminal are always in the same segment in every production or in distinct
segments. At run-time the various segments are vertically melted whenever they have at least one common attribute instance. This leads to a partition of the global dependency graph into a number of segments. In each segment the dependencies are linearized, but the evaluation of the instances in different segments can proceed concurrently, with synchronization points defined by cross-segment dependencies. Criteria are presented that allow to statically precompute the evaluation order in the segments. These criteria define a new class of parallel ordered attribute grammars. The construction of parallel visits sequences is performed by a polynomial-time algorithm.

Klein and Koskimies[1989,1990,1991] present a syntax-directed compilation scheme in which a parser runs in parallel with a set of parallel processes that carry out semantic computation. To establish a pipeline-type connection between the sequential parser and the set of parallel semantic processes, a tree-structured pipeline buffer is developed, allowing several waiting points for any number of semantic processes. The method can be applied to a variety of parallel semantic evaluation strategies. The only restriction is that semantic analysis works in a syntax-directed way. Parallel one-pass evaluation was the original motivation for this work.


To avoid processes with low load, weights are assigned to attributes, thus providing a means by which it can be decided at evaluation time whether or not to fork a sub-process.

Zaring[1990] gives a quite thorough analysis of the construction and implementation of parallel and incremental visit-sequence-based evaluators. He considers both asynchronous and synchronous parallelism. Asynchronous evaluation is able to exploit more parallelism than synchronous parallelism but requires lack of attribute instances to achieve correctness. Zaring presents several algorithms for the different parallel evaluation models. A survey of parallel attribute evaluation methods can be found in Jourdan[1991]. This paper reviews and compares the various methods that have appeared in the literature for both exhaustive and incremental attribute evaluation on both tightly-coupled (shared-memory) and loosely-coupled (distributed) architectures.

Many compilers spend a large fraction of their execution time in back-end phases such as instruction selection, register allocation, and optimization. This means that the parallelization of these phases is imperative to fast compiler operation. Parallelization of these phases should therefore be explored in the near future. Zobal[1990] states that the parallelization of machine-independent global optimization should be the core area of investigation. A basic component of optimization, which may be carried out many times during the compilation of a program, is data-flow analysis. Results on the parallelization of data-flow analysis are reported by Gupta et al.[1990] and Lee et al.[1990,1991].

Efforts in concurrent compiling that have thus far not been mentioned are the work of Banâtre et al., Deth, Dowson and Sanderson, Katself, and Alblas. Banâtre et al.[1990] consider an event-driven technique for semantic analysis. Deth[1984] describes an analyzer which maps a program onto a graph of semantic routines. It is shown how this graph can be evaluated in parallel, and also how the sequential evaluation of the semantic functions can be parallelized. Dowson and Sanderson[1986] investigate the use of Path Pascal for implementing an assembler on a distributed computer system. They report experiments using simulated concurrency on a single processor computer. Katself[1988] applies the technique of data partitioning to implement a parallel assembler. Alblas[1992] investigates the parallelization of a parser generator.

In the literature a large number of papers are devoted to the parallel execution of programs, and in particular to the parallel evaluation of arithmetic expressions. These papers are not included in this bibliography. Also not included in this bibliography are papers on compiler optimizations for supercomputers.

Another survey of parallel compilation can be found in Skillcorn and Barnard[1990a]. See also Skillcorn and Barnard[1990b].

Bibliography


