Bidirectional Attribute Evaluation

Ştefan ANDREI [*], Manfred KUDLEK [†], Cristian MASALAGIU [‡]

Abstract

Our intention is to describe a parallel algorithm (using two processors) for evaluating the attribute instances of an attributed derivation tree.

In the first section, we present some basic notions (graphs, trees, context free and attribute grammars).

The second section emphasizes two ways for representing the ordered oriented trees and the bidirectional traversal is also pointed out.

In the third section, we present a new approach for evaluating the attribute instances of an attributed derivation tree. We have called this strategy *the bidirectional attribute evaluation.*

In the last section, we formulate some conclusions and open problems.

Keywords: attribute grammars and evaluation, parallel algorithms

Mathematics Subject Classification: 68N20, 68P05, 68Q22, 68Q50, 68R10.

1 Basic Notions

An alphabet is a finite set V. V^* is the set of all words over V. The empty word is denoted by λ.

Definition 1.1 *We say that $\mathcal{G} = (\mathcal{V}, E, s, d)$ is an* **oriented graph** *if $\mathcal{V}$ is the set of vertices, E is the set of edges, $s : E \to \mathcal{V}$ (***the source function***), $d : E \to \mathcal{V}$ (***the destination***). If $s(e) = v$ and $d(e) = v'$, then the arc e will be denoted $e : v \to v'$ or $v \stackrel{e}{\to} v'$, or simply (v, v') if the name is not important (i.e. there exists - at least - one arc from v to v').*

For any $v \in \mathcal{V}$, let $E(v) = \{e \mid s(e) = v\}$ and $S(v) = \{v' \mid \exists\ e \in E$ such that $v \stackrel{e}{\to} v'\}$.

[*] Faculty of Informatics, "Al.I.Cuza" University, Str. Berthelot, nr. 16, 6600, Iaşi, România. E-mail: stefan@infoiasi.ro. This work was supported by The World Bank/Joint Japan Graduate Scholarship Program.

[†] Fachbereich Informatik, Universität Hamburg, Vogt-Kölln-Straße 30, D-22527 Hamburg, Germany. E-mail: kudlek@informatik.uni-hamburg.de

[‡] Faculty of Informatics, "Al.I.Cuza" University, Str. Berthelot, nr. 16, 6600, Iaşi, România. E-mail: mcristy@infoiasi.ro.

Definition 1.2 *For an oriented graph $\mathcal{G} = (\mathcal{V}, E, s, d)$, if $\mathcal{V}$ and E are finite, then $\mathcal{G}$ is a finite oriented graph. If for any $v \in \mathcal{V}$, $E(v)$ is a finite set, then G is called a* **locally oriented finite graph**.

For any $\mathcal{G} = (\mathcal{V}, E, s, d)$ a locally oriented finite graph, $E = \bigcup_{v \in \mathcal{V}} E(v)$ can be denoted in the following way:

$$E(v) = \{< v, 1 >, < v, 2 >, ..., < v, k_v >\}$$

That is, the set of all arcs having source v (consequently the set of sons of v, i.e. $S(v)$) can be viewed as an ordered set. Such a locally oriented graph is called **ordered oriented graph**. From now on, we shall work only with finite ordered oriented graphs. The local ordering induces a total *left-to-right* order on leaves.

Other natural concept notions on graphs can be easily translated to an easier notation for finite ordered oriented graphs. For example, a **path** of length n in G from v to v' is a word $p \in E^*$, $p = e_1 e_2 ... e_n$, $n \geq 1$, where:

$$v = v_0 \overset{e_1}{\rightarrow} v_1 \overset{e_2}{\rightarrow} v_2 ... v_{n-1} \overset{e_n}{\rightarrow} v_n = v'$$

Note. We shall suppose that for any node there exists a path of length 0 denoted by λ. Other notions (*connectivity, circuits, trees*, etc.) can be defined by translation from their classical definitions. Furthermore, we suppose that the reader is familiar with notions concerning *preorder traversal, depth first search, breadth search* corresponding to trees and graphs ([CLR91]).

Example 1.1 *Let us consider the following ordered oriented tree*
$T = (\{1, 2, ..., 11\}, \{< 1, 1 >, < 1, 2 >, < 1, 3 >, < 3, 1 >, < 3, 2 >, < 4, 1 >, < 4, 2 >, < 4, 3 >, < 5, 1 >, < 5, 2 >\}, s, d)$, *where*

$$\begin{array}{llll}
s(< 1, 1 >) = 1 & s(< 1, 2 >) = 1 & s(< 1, 3 >) = 1 & s(< 3, 1 >) = 3 \\
s(< 3, 2 >) = 3 & s(< 4, 1 >) = 4 & s(< 4, 2 >) = 4 & s(< 4, 3 >) = 4 \\
s(< 5, 1 >) = 5 & s(< 5, 2 >) = 5 & d(< 1, 1 >) = 2 & d(< 1, 2 >) = 3 \\
d(< 1, 3 >) = 4 & d(< 3, 1 >) = 5 & d(< 3, 2 >) = 6 & d(< 4, 1 >) = 7 \\
d(< 4, 2 >) = 8 & d(< 4, 3 >) = 9 & d(< 5, 1 >) = 10 & d(< 5, 2 >) = 11
\end{array}$$

T has the following graphic representation

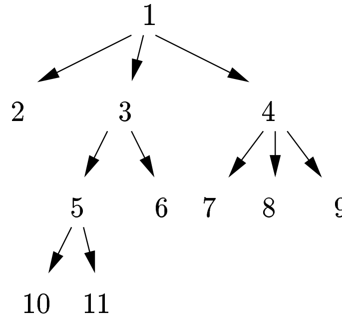

Figure 1.

and it has the preorder (depth first) traversal: 1,2,3,5,10,11,6,4,7,8,9.

We say that $G = (V_N, V_T, Z, P)$ is a *context free grammar*, if V_N is the alphabet of nonterminal symbols, V_T the alphabet of terminal symbols, and $V = V_N \cup V_T$ the set of symbols of G, $Z \in V_N$ the start symbol, $P \subseteq V_N \times V^*$ the set of productions. A pair $(A, \beta) \in P$ is called an $A-$production and it is denoted by $A \to \beta$. The productions $A \to \beta_1,\ A \to \beta_2, \ldots,\ A \to \beta_k$ will be denoted by $A \to \beta_1 \,|\, \beta_2 \,|\, \ldots \,|\, \beta_k$ (sometimes). The *empty word* will be denoted λ (the word of length 0). A *derivation in* G is denoted by $\alpha \underset{G}{\Longrightarrow} \beta$ if $\exists\ A \in \alpha$ and $A \to r \in P$ such that $\alpha = \alpha_1\, A\, \alpha_2,\ \beta = \beta_1\, r\, \beta_2$; the transitive (reflexive) closure of the relation $\underset{G}{\Longrightarrow}$ is denoted by $\overset{+}{\underset{G}{\Longrightarrow}}$ ($\overset{*}{\underset{G}{\Longrightarrow}}$).

Definition 1.3 *A* **derivation tree** $T = (\mathcal{V}, E, s, d)$ *for a context free grammar* $G = (V_N, V_T, Z, P)$ *is a node labelled finite ordered oriented tree. The labels of the nodes are given by a function* $f : \mathcal{V} \to V_N \cup V_T \cup \{\lambda\}$ *:*

For any $v \in \mathcal{V}$, *with* $S(v) = \{v_1, v_2, \ldots, v_k\}$, *if* $f(v) = X$, $f(v_1) = Y_1$, $f(v_2) = Y_2$, ..., $f(v_k) = Y_k$, *then* G *contains the production* $X \to Y_1 Y_2 \ldots Y_k$.

If $f(v) = X$ *(v being the root) and the word* $w = v_1 v_2 \ldots v_n$ *being the (ordered) labels of the leaves, we say that* T **describes the word** w **generated from** X**.** *If* $X = Z$ *and* $w \in V_T^*$, *then* T *describes a word from* $L(G)$.

Definition 1.4 *An* **attribute grammar** *([Alb91a]) is a five-tuple*

$$AG = (G, SD, AD, R, C),$$

defined as follows:

(1) $G = (V_N, V_T, Z, P)$ *is a (the underlying) context-free grammar. The grammar* G *is assumed to be reduced in the sense that every nonterminal symbol is accessible from the start symbol and can generate a string which contains no nonterminal symbols.*

(1.1) V_N *and* V_T *denote the alphabets of nonterminal and terminal symbols, respectively, and form the vocabulary* $V = V_N \cup V_T$, $V_N \cap V_T = \emptyset$;

(1.2) P *is the finite set of productions; a production* $p \in P$ *will be denoted as* $p : X_{p0} \to X_{p1} \ldots X_{pn_p}$, *where* $n_p \geq 0$, $X_{p0} \in V_N$ *and* $X_{pk} \in V$ *for* $1 \leq k \leq n_p$;

(1.3) $Z \in V_N$ *is the start symbol, which does not appear on the right side of any production.*

(2) $SD = (TYPE-SET, FUNC-SET)$ *is a semantic domain.*

(2.1) $TYPE-SET$ *is a finite set of sets;*

(2.2) $FUNC-SET$ *is a finite set of total functions of type* $type_1 \times \ldots \times type_n \to type_0$, *where* $n \geq 0$ *and* $type_i \in TYPE-SET$ $(0 \leq i \leq n)$.

(3) $AD = (A, I, S, TYPE)$ *is a description of attributes.*

(3.1) *For each symbol* $X \in V$ *there exists a set* $A(X)$ *of attributes which can be partitioned into two disjoint subsets* $I(X)$ *and* $S(X)$ *of* **inherited** *and* **synthesized** *attributes, respectively;*

(3.2) *The set of all attributes will be denoted by* A*, i.e.* $A = \bigcup_{X \in V} A(X)$.

(3.3) *Attributes associated with different symbols are considered as different, i.e.* $A(X) \cap A(Y) = \emptyset$ *if* $X \neq Y$*. If necessary an attribute* a *of symbol* X *will be denoted by* $X.a$*;*

(3.4) *For* $a \in A$, $TYPE(a) \in TYPE-SET$ *is the set of possible values of* a.

(4) $R(p)$ *is a finite set of attribute evaluation rules (semantic rules) associated with the production* $p \in P$.

(4.1) *Production* $p : X_{p0} \rightarrow X_{p1} \ldots X_{pn_p}$ *is said to have the* **attribute occurrence** (a, p, k) *if* $a \in A(X_{pk})$*;*

(4.2) *The set of all attribute occurrences of production* p *will be denoted by* $AO(p)$*;*

(4.3) *The set* $AO(p)$ *can be partitioned into two disjoint subsets of* **defined occurrences** *and* **used occurrences** *denoted by* $DO(p)$ *and* $UO(p)$*, respectively:*

$$DO(p) = \{(s, p, 0) \mid s \in S(X_{p0})\} \cup \{(i, p, k) \mid i \in I(X_{pk}) \wedge 1 \leq k \leq n_p\}$$

$$UO(p) = \{(i, p, 0) \mid s \in I(X_{p0})\} \cup \{(s, p, k) \mid i \in S(X_{pk}) \wedge 1 \leq k \leq n_p\}$$

The attribute evaluation rules of $R(p)$ *specify how to compute the values of the attribute occurrences in* $DO(p)$ *as a function of the values of certain other attribute occurrences in* $AO(p)$*. The evaluation rule defining the attribute occurrence* (a, p, k) *has the form*

$$(a, p, k) := f((a_1, p, k_1), \ldots, (a_m, p, k_m))$$

$(a, p, k) \in DO(p)$, $f : TYPE(a_1) \times \ldots \times TYPE(a_m) \rightarrow TYPE(a)$, $f \in FUNC-SET$ *and* $(a_i, p, k_i) \in AO(p)$ *for* $1 \leq k \leq m$*. We say that* (a, p, k) **depends on** (a_i, p, k_i)*, for* $1 \leq i \leq m$.

(5) $C(p)$ *is a finite set of semantic conditions associated with the production* p*. These conditions are predicates of the form*

$$\pi((a_1, p, k_1), \ldots, (a_m, p, k_m))$$

$\pi : TYPE(a_1) \times \ldots \times TYPE(a_m) \rightarrow \{true, false\}$, $\pi \in FUNC-SET$, *and* $(a_i, p, k_i) \in AO(p)$ *for* $1 \leq i \leq m$.

Semantic conditions allow the specification of a subset of the language defined by the underlying context-free grammar. A sentence that is generated by G is a sentence of the language specified by AG if the semantic conditions yield true. Traditionally, the definitions of attribute grammars require that both the start symbol and the terminal symbols to have no inherited attributes. We do not assume this restriction.

We have been so far concerned with the syntax of attribute grammars. Now let us discuss their semantics.

An unambiguos context-free grammar assigns a single derivation tree to each of its sentences. The nodes of a derivation tree are labelled with symbols from V. For each interior node there is a production $X_{p0} \to X_{p1} \ldots X_{pn_p}$, such that the node is labelled with X_{p0} and its n_p sons are labelled with X_{p1}, ..., X_{pn_p}, respectively. We say that p is the production *(applied) at* that node.

Definition 1.5 *A derivation tree is* **complete** *if it has only terminal symbols (or the empty string) as labels of its leaves and the start symbol as the label of its root.*

Unless stated otherwise our derivation trees are assumed to be complete.

Definition 1.6 *Given a derivation tree in an attribute grammar $AG = (G, SD, AD, R, C)$, instances of attributes are attached to the nodes in the following way: if node N is labelled with grammar symbol X, then for each attribute "a" $\in A(X)$ an instance of "a" is attached to node N. We say that the derivation tree has the attribute instance $N.a$. Let N_0 be a node, p a production at N_0 and N_1, ..., N_{n_p} the sons of N_0 in the given order (Definition 1.3). An* **attribute evaluation instruction**

$$N_k.a := f(N_{k_1}.a_1,\ \ldots,\ N_{k_m}.a_m)$$

is associated with attribute instance $N_k.a$ if the attribute evaluation rule

$$(a, p, k) := f((a_1, p, k_1),\ \ldots,\ (a_m, p, k_m))$$

is associated with production p. We say that attribute instance $N_k.a$ depends on *attribute instance $N_{k_i}.a_i$ for $1 \le i \le m$. If all the values are known and satisfy all attribute evaluation rules then we say that the attributed derivation tree is* **consistent.**

Definition 1.7 *A* **decorated** *(or* **attributed***)* **derivation tree** *is a derivation tree in which all attribute instances have a value (which is not necessarily consistent). A* **consistently decorated** *(***attributed***)* **derivation tree** *is a derivation tree in which all attribute instances are defined according to their associated attribute evaluation instructions, i.e. the execution of any evaluation instruction does not change the values of the attribute associated with a tree node (as described below).*

In this way, an attribute grammar assigns a (consistenly) decorated derivation tree to each of its sentences. Some applications concentrate on the result of the semantic conditions which conclusively decide whether a sentence is semantically correct or not. Other applications are only interesting in a decorated derivation tree as an intermediate result in the compilation process. For these applications semantic conditions are not used.

Definition 1.8 *For each derivation tree T a* **dependency graph** $D(T)$ *can be defined by taking the attribute instances of T as its vertices. The directed arc $(N_i.a, N_j.b)$ is contained in the graph if and only if attribute instance $N_j.b$ depends on attribute instance $N_i.a$. A path in a dependency graph will be called a* **dependency path.** *For $n > 0$, $dp[N_1.a_1, N_2.a_2, ..., N_n.a_n]$ stands for a path with arcs $(N_1.a_1, N_2.a_2)$, $(N_2.a_2, N_3.a_3)$, ..., $(N_{n-1}.a_{n-1}, N_n.a_n)$. A path $dp[N_1.a_1, N_2.a_2, ..., N_n.a_n, N_1.a_1]$ will be called a* **circular** *dependency path. An attribute grammar is* **circular** *if it includes a derivation tree whose dependency graph contains a circular dependency graph. An attribute grammar is called* **non-circular (well defined)** *if it is not circular. The class of all well defined grammars is denoted by WAG.*

The task of an *attribute evaluator* is to compute the values of all attribute instances attached to the derivation tree, by executing the attribute evaluation instructions associated with these attribute instances. Generally, the order of the evaluation is free, with the only restriction that an attribute evaluation instruction cannot be executed before its arguments are available. An attribute instance is *available* if its value is defined, otherwise it is *unavailable*. Initially all attribute instances attached to the derivation tree are unavailable, with the exception of the inherited attribute instances attached to the root (containing information concerning the environment of the program) and the synthesized attribute instances attached to the leaves (determined by the parser). At each step an attribute instance whose value can be computed is chosen. The evaluation process continues until all attribute instances in the tree are defined or until none of the remaining attribute instances can be evaluated.

For a traditional attribute evaluator, as described above, it is impossible to evaluate attribute instances involved in a circular dependency path.

Example 1.2 *Let $AG_1 = (G_1, SD_1, AD_1, R_1, C_1)$ be the following attribute grammar:*

(1) $G_1 = (\{Z, A\}, \{a, b\}, P_1, Z_1)$ *the underlying context-free grammar and* $P1$ *given below;*

(2) $SD_1 = (\{integer\}, FUNC - SET_1)$, *where* $FUNC - SET_1$ *is described below (i.e. identity function, constant function, add function, etc.);*

(3) $AD_1 = (A_1, I_1, S_1, TYPE_1)$, *where*

- *(3.1)* $A_1 = \{i, s\}$;
- *(3.2)* $I_1(Z) = I_1(A) = \{i\}$;

(3.3) $S_1(Z) = S_1(A) = \{s\}$;

(3.4) $TYPE_1(i) = TYPE_1(s) = \{integer\}$;

(4) the set R_1 *of attribute evaluation rules is described below both with the productions of* G_1*;*

(5) the set C_1 *is also presented below both with* P_1 *and* R_1*.*

Because a production might contain an occurrence of the same nonterminal symbol X*, in the attribute evaluation rule* X*, will have an index (starting from* 1 *to the last occurrence). The second production of* G_1 *is such a case.*

Production 1:

$Z \rightarrow A$

Attribute evaluation rules:

$Z.i := 1$; $Z.s := A.s$ $A.i := Z.i$

Production 2:

$A \rightarrow a\, A$

Attribute evaluation rules:

$A_2.i := A_1.i + 1$; $A_1.s := A_2.s + 1$;

Production 3:

$A \rightarrow b$

Attribute evaluation rule:

if $A.i > 10$ **then** $A.s := 0$ **else** $A.s := 1$

Let us consider the word $w = aab$*. Figure 2 presents the corresponding derivation tree* T *and the dependency graph* $D(T)$*.*

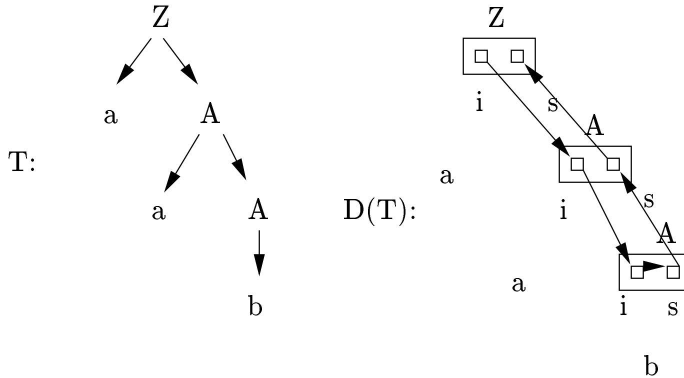

Figure 2.

In [Knu68] has been defined an important subclass of atrribute grammars, the so called *purely synthesized AG's* (SAG). In the same paper, was proven that SAG's have the same power as Turing machines. We can conclude that the power of attribute grammars is the same as Turing machines.

2 Data Representations of the Trees and their Bidirectional Traversal

In this section, we shall present two methods for representing the ordered oriented trees. For the second one, a bidirectional traversal of an ordered oriented tree is also presented.

First method of representation of ordered oriented trees.

Given the ordered oriented tree $T = (\mathcal{V}, E, s, d)$, let m be the maximal number of sons (counted for all vertices). For representing the sons $v_1, v_2, ..., v_n$ of father v, we use exactly m locations (even if $n < m$). Consider now the array $t : \{1, 2, ..., p\} \to \mathcal{V} \cup \{null\}$, where p is a natural number. The array t (denoted by `t[]`) is defined in the following way:

$$\texttt{t[]} = root\, v_1 ... v_2\, v_{11} ... v_{1m}\, v_{21} ... v_{22}\, ...\, v_{1m} ... v_{mm} ... v_{\underbrace{mm...m}_{m\ times}}$$

constructed by structural induction:

- if the root of T has the sons $v_1, ..., v_n$ then `t[1]`$= root$, `t[2]`$= v_1$, ..., `t[n]`$= v_n$, `t[n+1]`$= null$, ..., `t[m]`$= null$, where $null$ is a special symbol (character);
- let $v_{wk} \in V(T)$ have the sons v_{wk1}, v_{wk2}, ..., v_{wkn} and `t[s]`$= v_{wk}$, where $w \in \{1, 2, ..., m\}^*$, $k \in \{1, 2, ..., m\}$, then `t[s+(m-k)(m+1)+1]`$= v_{wk1}$, `t[s+(m-k)(m+1)+2]`$= v_{wk2}$, ..., `t[s+(m-k)(m+1)+n]`$= v_{wkn}$, and the "null" elements `t[s+(m-k)(m+1)+n+1]`$= null$, ..., `t[s+(m-k)(m+1)+m]`$= null$.

This representation of ordered oriented trees is useful for *breadth first search* visits. We don't present in detail this method of traversing ordered oriented tree, because this representation has a disadvantage related to its size. That is, the number of elements of the array `t[]` is exponential in m. In fact $p = 1 + m + m^2 + ... + m^m = \frac{m^{m+1}-1}{m-1}$ $(m \neq 1)$. Of course, the number of vertices of T could be "very" smaller than this number.

The second method to represent ordered oriented trees.

Let $T = (\mathcal{V}, E, s, d)$ be the ordered oriented tree. For representing the sons v_1, v_2, ..., v_n of father v, we use n locations. Now, the number of locations of the corresponding vector will be the same as the cardinality of $\mathcal{V}$. Let m be the maximum number of sons (for all vertices of T). Consider now the array $t : \{1, 2, ..., s\} \to (\mathcal{V}, \{1, 2, ..., m\})$, where $s = |\mathcal{V}|$. The informations contained in t have the following meaning: `t[i]`$= (v, d)$ iff $v \in \mathcal{V}$ and d is the number of sons of v.

Example 2.1 *For the tree presented in Example 1.1, we have:*

```
t[]=(1,3)(2,0)(3,2)(5,2)(10,0)(11,0)(6,0)(4,3)(7,0)(8,0)(9,0)
```

According to the (possible) huge number of "free cells" from the array `t`, the first method for implementing ordered oriented trees, is not convenient for

deriving parallel algorithms. The second representation method (although it provides no direct access) allows us to use a bidirectional parsing according to the placement of leaves.

This representation of ordered oriented trees is useful for *depth first search* visits. As the input of the following bidirectional traversal is the array `t[]` which is the corresponding preorder representation of T (i.e. the second method for representing ordered oriented trees). In fact, we present a parallel combination of the two sequential strategies of traversal *up* and *down* for the tree. Furthermore, the down traversal coincides with the depth first search strategy. We consider two processors `P1`, `P2` and two global variables `i1`, `i2` (i.e. both processors can read these variables). We suppose that we have a procedure "`halt(P)`", which stops the running of the processor `P`. We shall call this algorithm (BT) (i.e. bidirectional traversal).

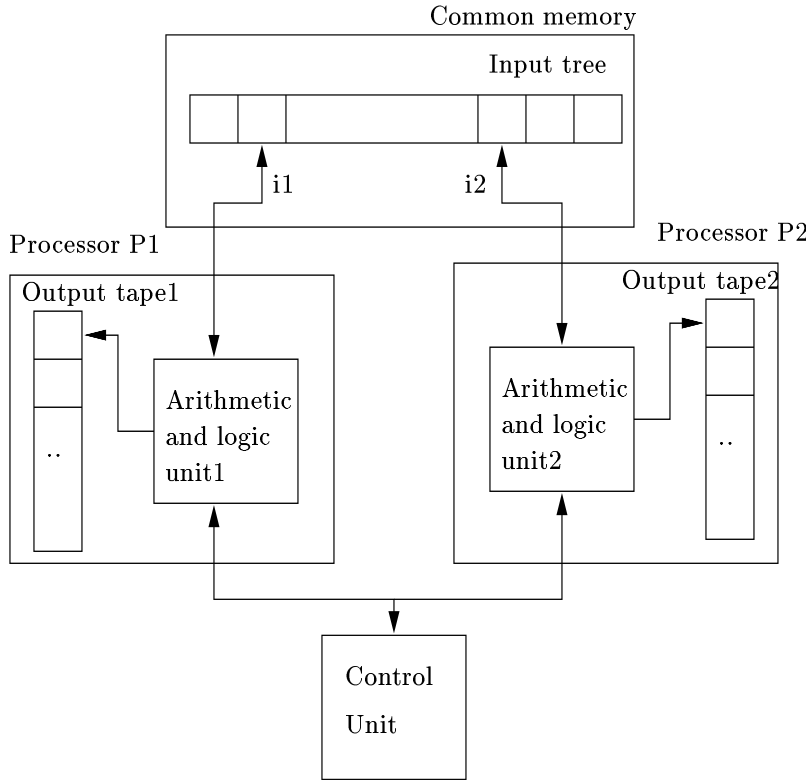

Figure 3. General SIMD Model for Bidirectional Traversal

We can say that our model is a SIMD (simple instruction stream and multiple data stream) computer ([Akl97]). This means that these two processors `P1` and `P2` operate synchronously. Furthermore, we can say that our model uses *multiprocessors* because the processors `P1`, `P2` share a common memory.

```
procedure visit_down(P1);
begin
  if i1 <= i2 then begin
    (v1,d) := t[i1];
    {visit the vertex v1}
    write("we have visited ", v1);
    write(" and it has ", d, " sons");
    i1 := i1 + 1;
    visit_down(P1)
  end
  else halt(P1);
end;
```

The up traversal of the tree is quite similar.

```
procedure visit_up(P2);
begin
  if i2 > i1 then begin
    (v2,d) := t[i2];
    {visit the vertex v2}
    write("we have visited ", v2);
    write(" and it has ", d, " sons");
    i2 := i2 - 1;
    visit_up(P2)
  end
  else halt(P2);
end;
```

The main program is

```
begin
  read(t[]); {read the tree}
  i1 := 1; i2 := n; {n being the number of vertices of T}
  repeat in parallel
    visit_down(P1);
    visit_up(P2)
  until (i1>i2);
end.
```

Example 2.2 *Let us consider the tree from Example 1.1 (and 2.1). We shall simulate the "parallel running" of Algorithm (BT).*

Initial: `i1 = 1` *and* `i2 = 2`.
Step 1: `P1` → *"we have visited 1 and it has 3 sons"*
`P2` → *"we have visited 9 and it has 0 sons"*
Step 2: `P1` → *"we have visited 2 and it has 0 sons"*

`P2` → *"we have visited 8 and it has 0 sons"*
Step 3: `P1` → *"we have visited 3 and it has 2 sons"*
`P2` → *"we have visited 7 and it has 0 sons"*
Step 4: `P1` → *"we have visited 5 and it has 2 sons"*
`P2` → *"we have visited 4 and it has 3 sons"*
Step 5: `P1` → *"we have visited 10 and it has 0 sons"*
`P2` → *"we have visited 6 and it has 0 sons"*
Step 6: `P1` → *"we have visited 11 and it has 0 sons"*
`P2` → *halts*

Compared to the classical preorder visit (which needs 11 steps), our bidirectional traversal needs only 6 steps.

We saw in Example 2.2 that the processor `P1` visits the tree in depth first search manner (the order of visiting the vertices is $\{1, 2, 3, 5, 10, 11, 6, 4, 7, 8, 9\}$) and `P2` in the opposited manner to `P1` (i.e. the order of visiting the vertices is $\{9, 8, 7, 4, 6, 11, 10, 5, 3, 2, 1\}$). In fact, `P2` visits all the sons from right to left and finally their root.

Theorem 2.1 *(correctness and completeness) Let $T = (\mathcal{V}, E, s, d)$ be a ordered oriented tree represented by an array which contains its preorder representation as the input of Algorithm (BT). Then:*

a) *After the execution of Algorithm (BT), all the vertices of T have been visited.*

b) *Let us denote with $T_1(n)$, $T_2(n)$ the running time of the procedures* `visit_down` *and* `visit_up`*, where $n = |\mathcal{V}|$. Then the parallel running time $t(n)$ satisfies the relation (we suppose that the routing time is zero):*

$$\frac{\min\{T_1(n), T_2(n)\}}{2} \leq t(n) \leq \max\{T_1(n), T_2(n)\}$$

Proof

a) If $i1 < i2$ then each call of procedures `visit_down` and `visit_up` implies the visit of two new vertices (i.e. which has not yet been visited). We know that the array `t` contains in fact the preorder representation of T. The procedures `visit_down` and `visit_up` read the array `t` cell by cell (because of the statements `i1 := i1 + 1` and `i2 := i2 - 1`). The cells `t[i1]` and `t[i2]` contain informations about the current vertices `v1` and `v2`, respectively. So, `v1`, `v2` have been visited at this parallel step.

b) The inequality $t(n) \leq \max\{T_1(n), T_2(n)\}$ can be obtaining by supposing that one processor stays. For instance, if `P1` stays, then $t(n) = T_2(n)$ (time routing is zero). The other inequality can be obtained by supposing that both processors work until $i1 = i2$. This implies a running time of $\frac{\min\{T_1(n), T_2(n)\}}{2}$.

■

3 Bidirectional Attribute Evaluation

In [Alb91b], the *flexible* and the *rigid tree-walking strategies* for traversing the attributed decorated tree have been presented. A *flexible strategy* is completely determined by the attribute dependencies of the grammar concerned. Typical example of attribute grammar classes with a flexible tree traversal strategy are the *absolute non-circular* (ANC) and the *ordered attribute grammars.*

A *rigid strategy* is independent of the attributed dependencies. A typical example of a rigid strategy is to make a number of passes over the derivation tree, where a *pass* is defined to be a depth-first left-to-right or right-to-left traversal of the derivation tree. An example of a strategy somewhere in between the flexible and the rigid strategies is the performance of attribute evaluation during a sequence of *sweeps* over the derivation tree. A *sweep*, as defined in [EnF82], is a depth-first traversal of the derivation tree, without the restriction of a left-to-right or a right-to-left order of succesion, i.e. the visiting order of the nodes is free with the only restriction that every tree node is visited exactly once.

Our approach refers to a rigid strategy which works for general non-circular attributed grammars. We called it *bidirectional attribute grammars.* Next, we present the attribute evaluation algorithm which works for a well-defined attribute grammar (WAG). We call it *bidirectional attribute evaluation algorithm*, and it will be denoted by (BAE). Each vertex v of the input attributed derivation tree has three components. The first component is the label, i.e. the terminal or non-terminal symbol X, the other components being the set of instances of inherited and synthesized attributes of X, respectively.

The Algorithm (BAE):

Input: A well-defined attribute grammar AG and an attributed derivation tree $T=(\mathcal{V},E)$ where only the inherited attribute instances of the start symbol and the synthesized attribute instances of the terminal symbols are defined.
Output: An attributed derivation tree where all attribute instances are defined.
Method: Like in Algorithm (BT), we shall present two procedures for each of the processors and a main program. Because each element of the array `t[]` stores two informations, the current vertex and the number of sons of the current vertex (i.e. the pair `(v,d)`), we denote `t[].one=v` and `t[].two=d`. Each of these procedures contain two calls of procedure `evaluate(X.a)`. This procedure is in fact the corresponding attribute evaluation instruction to the current production according to Definition 1.6.

```
procedure visit_down(P1);
begin
   if i1 <=|V| then begin
     (v1,d) := t[i1];
     (Xp0,I(Xp0),S(Xp0)):=v;
     for k := 1 to d do
       (Xpk,I(Xpk),S(Xpk)) := t[i1+k].one;
```

```
    for k := 1 to d do
      for (all a∈I(Xpk)) do
        if (Xpk.a is undefined) and
          (all argument instances of the evaluation instruction
           for Xpk.a are defined)
        then evaluate(Xpk.a);
    for (all a∈S(Xp0)) do
      if (Xp0.a is undefined) and
        (all argument instances of the evaluation instruction
         for Xp0.a are defined)
      then evaluate(Xp0.a);
    i1 := i1 + 1;
    visit_down(P1)
  end
  else halt(P1);
end;
```

The up traversal of the tree is quite similar.

```
procedure visit_up(P2);
begin
  if i2 >= 1 then begin
    (v2,d) := t[i2];
    (Xp0,I(Xp0),S(Xp0)):=v;
    for k := 1 to d do
      (Xpk,I(Xpk),S(Xpk)) := t[i2+k].one;
    for k := 1 to d do
      for (all a∈I(Xpk)) do
        if (Xpk.a is undefined) and
          (all argument instances of the evaluation instruction
           for Xpk.a are defined)
        then evaluate(Xpk.a);
    for (all a∈S(Xp0)) do
      if (Xp0.a is undefined) and
        (all argument instances of the evaluation instruction
         for Xp0.a are defined)
      then evaluate(Xp0.a);
    i2 := i2 - 1;
    visit_up(P2)
  end
  else halt(P2);
end;
```

The main program is

```
begin
  read(t[]); {read the attributed derivation tree}
  i1 := 1; i2 := n;
```

```
  {n being the number of vertices of the input tree}
  repeat in parallel
    visit_down(P1);
    visit_up(P2)
  until (all attribute instances are evaluated);
  write(t[]);
  {the attributed derivation tree is consistently decorated}
end.
```

Remark 3.1 *Every attribute instance is evaluated during the earliest possible pass and the different instances of the same attribute may be evaluated during different passes. In the classical model of evaluation the attributed instances of the attributed derivation tree depend in fact on the number of levels (depth) of the input derivation tree. For instance, the synthesized attributed instances of the root of* T *will be defined after at least* m *visits of the attributed derivation tree, where* m *is the depth of it.*

Using our bidirectional attribute evaluation algorithm, the synthesized attribute instances of the root of T *will be defined in one visit of the attributed derivation tree, for the synthesized attributed instances of the terminal symbols.*

Compared to [Alb91b], our bidirectional attribute evaluation algorithm has "half" time running than the classical attribute evaluation algorithm.

Example 3.1 *Let us consider the attribute grammar (Example 2.1). It is easy to see that for* $w = aab$*, according to Algorithm (BAE), we need only* **one** *down visit and* **one** *up visit for evaluating all the attribute instances of the correspoding attributed derivation tree. On the other hand, using classical methods (down visits), for evaluating the attribute instances, we need to visit the attributed derivation tree* **three** *times (i.e. three down visits).*

We can immediately generalize to the input word $w = a^m b$*. In that case, Algorithm (BAE) needs the same* **one** *down visit and* **one** *up visit, instead of* **m+1** *down visits in the classical method.*

Theorem 3.1 *(correctness and complexity) If the input attributed grammar AG is non-circular, then Algorithm (BAE) will compute the attribute instances of the input attributed derivation tree in a finite number of steps. That is, the output of the Algorithm (BAE) will be a consistently decorated attibuted derivation tree.*

Proof Let AG be a non-circular (well defined) attribute grammar. This means that for any derivation tree T, the dependency graph $D(T)$ has no cycles. In $D(T)$, the directed arc $(N_i.a, N)j.b)$ is an arc iff attribute instance $N_j.b$ depends on attribute instance $N_i.a$. Because $D(T)$ is acyclic, its arcs specify a partial ordering of the attribute instances. The existence of arc $(N_i.a, N_j.b)$ indicates that the value of attribute instance $N_i.a$ must be defined before attribute instance $N_j.b$ can be computed.

The procedures `visit_down(P1)` and `visit_up(P2)` use the procedure `evaluate(X.a)`, which is in fact an attribute evaluation instruction. Because of the loop `repeat ... until`, from the main program, it follows that all attributed instances of T will be computed. Because $D(T)$ has no cycles, it follows that the number of iterations of the above loop is finite. ■

4 Conclusions

The complexity of Algorithm (BAE) applied to an attributed derivation tree is related to the complexity of Algorithm (BT) applied to an ordinary tree. It is obvious that the number of visits for evaluating all the attribute instances of the attributed derivation tree made by Algorithm (BAE) is less than the classical algorithm ([Alb91b]). As we saw in Example 3.1, there exist situations for which our bidirectional evaluating strategy is independent of the size of the input attributed derivation tree (only two visits).

But, sometimes, due to the dependency graph, the processors have to wait one each other and only one processor works. In that case, the parallel attribute evaluation coincides with the classical one. One open problem could be: In which cases (for what kinds of attribute grammars) our bidirectional attribute evaluation is strictly (or two times) better than the classical one ?

If the underlying context free grammar of the corresponding attribute grammar is in *Chomsky normal form*, then any derivation tree will be binary. In that case, the first method of representing ordered oriented trees is a convenient data structure for Algorithm (BAE), too. Furthermore, any node of the tree could be directly accessed (i.e. not sequential, like in the second method of representing ordered oriented trees).

References

[Alb91a] Alblas, H.: *Introduction to Attribute Grammars.* Attribute Grammars, Applications and Systems, LNCS 545, Eds. Alblas, H., Melichar, B., pp. 1-15 (1991)

[Alb91b] Alblas, H.: *Attribute Evaluation Methods.* Attribute Grammars, Applications and Systems, LNCS 545, Eds. Alblas, H., Melichar, B., pp. 48-113 (1991)

[AhU72] Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation, and Compiling. Volume I, II, Prentice Hall, 1972

[Akl97] Akl, S.: Parallel Computation. Models and Methods. Prentice Hall, 1997

[AnG95] Andrei, Şt., Grigoraş, Gh.: Tehnici de compilare. Lucrări de laborator. Editura Universităţii "Al.I.Cuza", Iaşi, 1995

[AnK99] Andrei, Şt., Kudlek, M.: *Bidirectional Parsing for Context Free Languages.* B-219, Fachbereich Informatik, Universität Hamburg: pp. 1-56 (1999)

[ASU86] Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley Publishing Company, U.S.A., 1986

[CLR91] Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. The MIT Press, New York, 1991

[EnF82] Engelfriet, J., Filé, G.: *Simple multi-visit attribute grammars.* Journal of Computer and System Science 24, pp. 283-314 (1982)

[GiR89] Gibbons, A., Rytter, W.: Efficient Parallel Algorithms. Cambrigde University Press, 1989

[GTW78] Goguen, J. A., Thatcher, J. W., Wagner, E.G.: *An initial algebra approach to the specification, correctness, and implementation of abstract data types.* Current Trends in Programming Methodology, IV: Data Structuring, (R.T. Yeh Ed.) Prentice Hall, New Jersey, pp. 80-149 (1978)

[Gri86] Grigoraş, Gh.: Limbaje formale şi tehnici de compilare. Editura Universităţii "Al.I.Cuza", Iaşi, 1986

[Har78] Harrison, M. A.: Introduction to Formal Language Theory. Addison - Wesley Publishing Company, 1978

[Hay88] Hayes, J.P.: Computer Architecture and Organization, McGraw-Hill International Editions, 1988

[HoU79] Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison - Wesley Publishing Company, 1979

[JuA97] Jucan, T, Andrei, Şt.: Limbaje formale şi teoria automatelor. Culegere de probleme. Editura Universităţii "Al. I. Cuza", Iaşi, 1997

[Knu68] Knuth, D. E.: *Semantics of Context-Free Languages.* Mathematical Systems Theory, Vol. 2, No. 2 Springer Verlag, New York, pp. 127-145 (1968)

[Mas92] Masalagiu, C.: Tipuri abstracte de date. Editura Universităţii "Al. I. Cuza", Iaşi, 1992

[Sal73] Salomaa, A.: Formal Languages. Academic Press. New York, 1973

[Tha67] Thatcher, J. W.: *Characterizing derivation trees of context free grammars through a generalization of finite automata theory.* Journal of Computer and System Science 1, pp. 317-322 (1967)

[ThS92] Thulasiraman, K., Swamy, M.N.S.: Graphs: Theory and Algorithms. John Wiley & Sons, INC., New York, 1992