This paper appears in *Attribute Grammars, Applications and Systems*, ed. by H. Alblas and B. Melichar, Prague (CZ), June 4–13, 1991, pp. 234–255, vol. 545 of *Lect. Notes in Comp. Sci.*, Springer-Verlag. Copyright © 1991 by Springer-Verlag, New-York, Heidelberg, Berlin.

A Survey of Parallel Attribute Evaluation Methods

Martin JOURDAN

INRIA*

Abstract

Exploiting parallelism in attribute evaluation is of potentially high interest because of both its applications (e.g. in speeding up heavily-used programs such as compilers) and its feasibility (i.e. most practical attribute grammars exhibit much parallelism). In this paper we review and compare 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. We pay particular attention to a simple but effective method for constructing efficient visit-sequence-based evaluators that run on tightly-coupled multi-processor machines by giving an account of how we implemented this method in practice and reporting the results of preliminary but realistic experiments; these results are highly encouraging.

1 Introduction and Motivation

Since compilers are heavily-used programs, it is of high interest to make them as fast as possible. One possible way of achieving this is to run them on parallel computers, provided they can exploit the available parallelism. Hand-made construction of parallel compilers is possible and has actually been done [Fra83, GZZ89, Lip79, SWJ88, Van88], but it is an error-prone process because it is hard to organize the accesses to central data structures, e.g. the program tree and the symbol table, to achieve both correctness and efficiency (cf. the "don't know yet" problem in Seshadri's works). Furthermore, methods that have been used to parallelize a compiler for a given language are not always immediately applicable to a compiler for another language. Lastly, "thinking parallel" adds up to the compiler development costs, which are already great when no good specification method is used.

In addition to reducing development costs, using a specification method allows to clearly separate the "intellectual" issues of the development, i.e. what is actually in the specification, from the more mundane implementation techniques. In particular, implementation techniques can be devised for a specification method independently from a particular application and reused for all of them. Each improvement in the implementation will then automatically benefit to all of its uses. Attribute grammars (AGs) [Alb91a,

*Author's address: INRIA, Projet ChLoÉ, Bât. 13, Domaine de Voluceau, Rocquencourt, BP 105, F-78153 LE CHESNAY Cedex, France. E-mail: `Martin.Jourdan@inria.fr`.

DJL88, Knu68] are a specification method that has proved quite useful for compiler construction [Kas91a]. It is clear that devising effective parallel attribute evaluation methods, i.e. implementations of AGs, is highly interesting since every compiler specified by an AG can then *automatically* and *safely* be turned into a parallel compiler.

Another highly interesting property of AGs is that they are indeed amenable to parallel implementations: the order in which to evaluate the attribute instances on a given tree is constrained only by the dependencies induced by the semantic rules, and in most practical AGs, if not all, this leads only to a (very) partial order, not a total one; then it is possible to evaluate concurrently instances that are not linked by a dependency path.

It is hence not surprising that parallel attribute evaluation has attracted a good deal of interest, especially since affordable, general-purpose multi-processor machines are relatively easily available. The aim of this paper is to review the various methods that have appeared in the literature. As in the sequential case, we'll have to distinguish exhaustive and incremental evaluation, but we'll also classify these methods according to the class of parallel machines—tightly-coupled (shared-memory) or loosely-coupled (distributed)—they target, because this leads to very different algorithms. As a case study, we'll describe with more details the implementation of a method sketched rather long ago by Schell [Sch79] and recently revamped [Mar90, Zar90]; this method is interesting because it aims at exhibiting and exploiting only the most "useful" parallelism, in order to reduce synchronization costs and gain speed even with a small number of processors. We'll give an account of the difficulties one encounters and the tradeoffs one should make when working on parallel machines.

This paper is organized as follows. First we briefly review the various parallel machine architectures and their programming model; in particular we discuss the cost of the various parallel primitive operations. Then we explain why parallel attribute evaluation is possible and attempt to devise a classification scheme for parallel evaluation methods. The next, main section reviews the various methods that have been proposed in the literature. Before concluding, we devote some space to the implementation of our preferred method and a brief assessment of it.

2 Parallel Architectures and Programming

The machines we are interested in here all have more than one processor. The first distinction we have to make is whether all the processors in a given machine operate in a synchronous way, i.e. execute the same instruction of the same code at the same moment (SIMD machines), or in an asynchronous way (MIMD machines). The first ones, which include in particular vector machines and such "massively parallel" machines as the Connection Machine, are ill-suited to parallel attribute evaluation because an attributed tree has a much less "regular" structure than arrays.

Another important distinction is between tightly-coupled (shared-memory) and loosely-coupled (distributed) machines. In the former, all processors can access with the same speed the same global memory (in addition to optional private memory). They can thus communicate easily through shared objects that can be structured at will. However the communication channels between the processors and the global memory can be overflowed by simultaneous accesses by the processors; hence this kind of machines does not scale

up to a massively number of processors. In a distributed architecture the processors can access only their private memory. Inter-processor communication has to be performed explicitly by sending messages through dedicated channels, either to a limited number of neighbors (hypercube architectures) or to all other processors (network); differences in the communication architectures result in different tradeoffs between speed and ease of access to all processors. Both kinds of architecture are amenable to parallel attribute evaluation but with rather different methods.

Programming distributed applications "reduces" to using **send** and **receive** primitives, which generally exist in both synchronous (blocking) and asynchronous versions. On tightly-coupled machines, applications are composed of a set of processes. In the synchronous model, process management and synchronization operations are hidden in the simple **cobegin/coend** or **fork/join** construct; the term "synchronous" does not imply that all the processes execute the same code but merely that they synchronize at these points (and only there). The asynchronous model uses lower-level process management operations: **spawn** for creation, **halt** for termination and **wait** for synchronization. To allow processes to wait for the availability of data computed by other processes, it is common to use *semaphores*. In any case, modifications to shared data structures must be protected by *locks*.

3 Generalities on Parallel Attribute Evaluation

In the classical theory of attribute grammars [Alb91a, Knu68], the order in which to evaluate the attribute instances on a given tree is constrained only by the dependencies induced by the semantic rules and represented in the (global) dependency graph for this tree. In most practical AGs, if not all, this leads only to a (very) partial order, not a total one. In a sequential evaluator this partial order is completely serialized, and various methods have been devised to do this either at run-time or at construction-time [Alb91b].

This serialization is however not imposed by the theory and is "only" an implementation technique. Instead, on a multi-processor machine, attribute computations can be distributed over several evaluation *processes* or *tasks* that run concurrently. It is still necessary to comply with the partial evaluation order, of course, but it is quite possible to evaluate concurrently instances that are not linked by a dependency path; such instances are called *independent*.

In most practical AGs, if not all, independent attribute instances are quite common. For instance, in an AG describing the static semantics of a block-structured language such as Pascal, all the attribute instances in the bodies of two distinct procedures are independent; in Pascal again, the attribute instances dealing with label declarations and uses are independent from the instances dealing with other kinds of declarations. Parallel attribute evaluation is hence potentially feasible and profitable.

Kuiper [Kui89, KuS90] contributes to the study of parallel attribute evaluation in two areas: he defines the concept of *distributor* as a model of (static) allocation of attribute instances to evaluation processes and he gives an algorithm to statically detect all pairs of independent attribute instances in a given AG.

Kuiper's distributors are functions that map each node in some dependency graph to one of n processes. To compute the value of an attribute instance, the values of all

its direct predecessors must be available, which requires some form of communication between evaluation processes. Thus a distributor must find a good tradeoff between the following conflicting goals:

- independent attribute instances must, as much as possible, be allocated to separate processes, so as to maximize the level of concurrency;
- dependent attribute instances must, as much as possible, be allocated to the same process, so as to minimize the level of inter-process communication.

Kuiper defines tree-based and attribute-based distributors. A tree-based distributor allocates all attribute instances of a same node to the same process. The most useful tree-based distributors are those that split up trees in connected regions; each evaluation process then computes all the attributes in a given region. Because attribute dependencies flow along the tree only, this reduces inter-process communication. This also simplifies the distributors: each region containing a distinguished node that is the ancestor of all other nodes in the region, a region-based distributor is uniquely defined by giving the set of nodes that are the root of the various regions (Kuiper calls them *selected* nodes). The most natural static criteria for selecting nodes are based on the production applied at the nodes or on the non-terminal labeling them; in addition, one must distinguish *nested* and *non-nested* distributors. The example of Pascal procedure bodies given above is an instance of tree-based distributor.

An attribute-based distributor allocates all instances of a same attribute or of a same attribute occurrence to the same process. For instance, the example of labels-related attributes given above is an example of attribute-based distributor. Since attribute-based distributors do not distinguish between different instances of the same attribute, they cannot exploit any independence between these instances and this generally limits the amount of parallelism.

A *combined* distributor consists in a tree-based distributor followed by an attribute-based distributor: the former splits up the trees into regions and the latter allocates the attribute instances attached to each region to a number of separate processes.

In our opinion, Kuiper's work is interesting because it lays down the foundations for a scheme to classify parallel attribute evaluation, but unfortunately it is incomplete: one should add the notion of dependency-based distributor. Such a distributor allocates to a given process all the attribute instances belonging to some connected region in the dependency graph. The reason why Kuiper did not address this notion in his work is that there generally exist no static criterion to define these connected regions. However, as will be described below, many parallel attribute evaluation methods are based on this notion of dependency-based distributor, and Kuiper is unable to make them fit in his classification scheme.

Another important part of Kuiper's work is an algorithm to statically compute all pairs of independent attribute instances in a given AG. Actually, and to put it briefly, his algorithm computes:

1. all the possible (synthesized) dependencies between the attributes of a given non-terminal, using Knuth's circularity test;
2. then, all the possible dependencies between the attributes of any pair of non-terminals $[X, U]$, where X is a possible ancestor of U;

3. and finally, all the possible dependencies between the attributes of U and V in any triple $[X, U, V]$, where there exist some tree in which U and V label cousin nodes and X labels one of their common ancestors.

Then, it is possible to prove whether all instances of attribute $U.a$ are independent of all instances of attribute $V.b$ in every tree by examining all the sets of the dependencies associated with all triples $[X, U, V]$. This algorithm is triply exponential but should benefit from the optimizations in the resolution of Grammar Flow Analysis [MöW91]. Since this work is rather recent, it has not received any application yet, as far as we know.

4 Parallel Evaluation Methods

4.1 Exhaustive evaluation on shared-memory machines

4.1.1 Dynamic methods

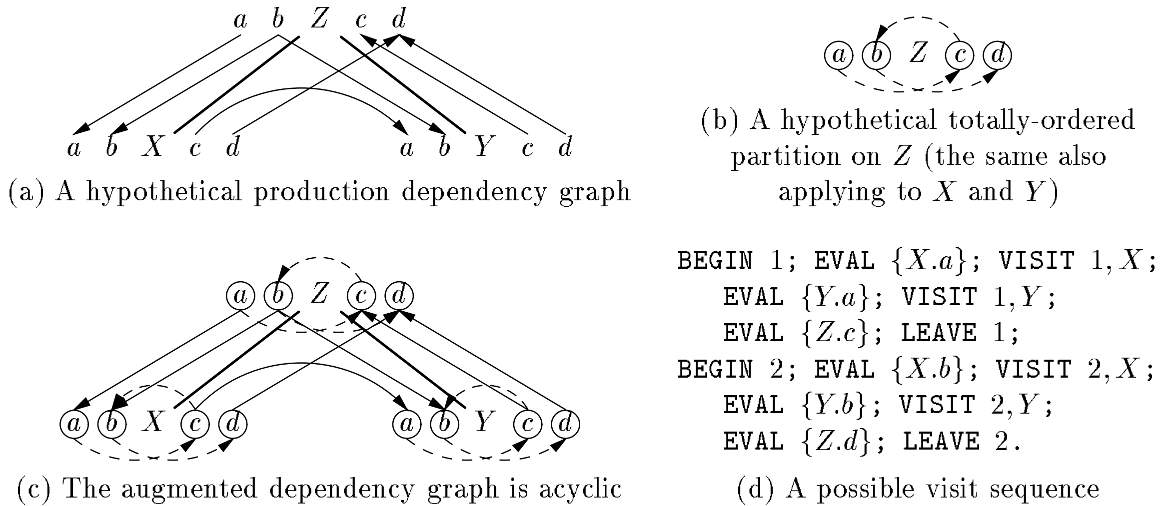
The possibility of evaluating attributes concurrently was recognized very early since it is the basis of the first historical implementation of AGs, the FOLDS system [Fan72], designed and implemented by Fang (his thesis advisor was Knuth himself). A simple, although not quite accurate, explanation of Fang's method is as follows: for each attribute instance[1] in the tree a separate process is spawned that waits until all the instances it needs (depends on) are available, computes the relevant value and then signals to all the processes needing it that it's available. This can be implemented by associating a semaphore with each attribute instance.

This method is inherently inefficient because each process only executes a few useful instructions and the overhead for managing and synchronizing the processes is much greater that what is gained through their parallel execution. Space inefficiency is even greater since, in addition to the parse tree and all the attribute instances it bears, space is needed for all the semaphores and the process descriptors. To compound Fang's problems, he had to simulate his parallel machine on a single sequential processor!

4.1.2 Static methods

These problems stated above drove other authors use a static analysis of the AG, based on the dependencies, to statically detect *useful* parallelism. Indeed, the main problem with Fang's dynamic approach is that the first thing most processes have to do when they start is wait for data computed by other processes, whereas a simple analysis of the dependencies in the AG could have shown that this and that process would only encumber the process queues without being able to do useful work. In fact, this is the same problem as for dynamic sequential evaluation methods that have to determine the evaluation order at run-time, which is equivalent to the process scheduling and synchronization problem in Fang's method.

[1] Actually the notion of process was explicit in SPINDLE, the language of FOLDS, as one construct to define statements. There was in principle no mandatory mapping between semantic rules (attribute instances) and processes but the usual programming style favored this situation.



(a) A hypothetical production dependency graph

(b) A hypothetical totally-ordered partition on Z (the same also applying to X and Y)

(c) The augmented dependency graph is acyclic

(d) A possible visit sequence

Figure 1: l-orderedness test and construction of visit sequences

Visit-sequence-based approach It is hence not surprising that the parallel evaluation problem was addressed with the same static analysis techniques as for sequential evaluation. Indeed, finding a total order for evaluating the attributes of each node is the only way to avoid expensive locking at the attribute-instance level and inactive processes waiting for not-yet-ready data. So much work was done to adapt the well-known visit-sequence evaluation paradigm to run on parallel machines [Alb91d, KlK89, KlK90, Mar90, Sch79, Zar90].

Visit-sequence-based evaluation is described at length elsewhere in this volume [Alb91b, Kas91b]. Let's only recall here that such an evaluator can be produced only if the AG at hand is l-ordered, i.e. if it is possible to find, for each non-terminal, a totally-ordered partition of its attributes such that, when augmenting each production dependency graph with the corresponding edges between the attributes of the non-terminals in the RHS and LHS of the production, the graph remains acyclic. This is exemplified in Fig. 1(a, b, c). The corresponding (sequential) visit-sequence is then produced during a topological sort of the graph (Fig. 1(d)).

The basic idea of all parallel evaluation methods based on the visit-sequence paradigm is to keep a total order for evaluating the attributes of each non-terminal (node) but allow to evaluate the attributes of different nodes, and visit those nodes, in parallel. The only difference with the construction of the sequential visit sequences is as follows: when topologically ordering an augmented dependency graph, use sequentiality when it is mandatory, but use parallelism when you come to a node with more than one successor. For instance, using the synchronous model of parallelism given in section 2, the second part of the visit sequence in Fig. 1(d) can be rewritten as:

```
BEGIN 2; FORK
   EVAL {X.b}; VISIT 2,X ||
   EVAL {Y.b}; VISIT 2,Y
JOIN; EVAL {Z.d}; LEAVE 2.
```

The first visit cannot be parallelized because the edge $X.c \rightarrow Y.a$ mandates sequential execution of the visits to X and Y.

This method has the very important advantage that synchronization, i.e. the overhead induced by parallelism, which is the possible cause of inefficiency, is reduced to a minimum since, because it is driven by the total evaluation order for the attributes of each non-terminal, the construction guarantees that:

1. all data needed by a process are computed before the process is spawned, and

2. no two processes running concurrently access the same part of the tree and, in particular, write to the same locations.

In other words, all the overhead is concentrated in the (unavoidable) **fork** and **join** instructions. So each process will run as fast as the corresponding part of the sequential evaluator. In fact, the l-ordered AG class, with the associated total evaluation order, is the largest class for which it is possible to use synchronous parallelism. A possible implementation is detailed below.

There exist several possible variations of this method. Schell [Sch79, sec. 5.5] was the first to publish it in a form quite close to the one presented above; he presents modifications to the evaluator-construction algorithms by Kennedy and Warren [KeW76], Kastens [Kas80], and Bochmann [Boc76], all based on the idea to perform visits to subtrees in parallel whenever possible. However he does not detail the run-time operation of the evaluator further than **fork/join**. This method was rediscovered independently by the author during the summer of 1988 and actually implemented (see [Mar90] and below).

Zaring [Zar90] gives a quite thorough analysis of the construction and implementation of parallel exhaustive and incremental visit-sequence-based evaluators. Incremental evaluation is discussed below. Zaring distinguishes asynchronous and synchronous parallelism, as presented in section 2. Asynchronous evaluation is able to exploit more parallelism than synchronous evaluation but must resort to attribute-instance locking to achieve correctness. Zaring presents a lot of algorithms differing by the kind of parallel model they use (asynchronous or synchronous) and the kind of visit-sequence instructions they can perform in parallel (all of `EVAL`, `VISIT` and `LEAVE` instructions are considered). As can easily be guessed, the `LEAVE` instructions cause the most problems. In the asynchronous model these problems are pushed to run-time and solved by attribute-instance locking. In the synchronous model `LEAVE` instructions are harder to deal with because they cause a mismatch between the **fork/join** process control structure and that of the tree data structure; indeed, the above-stated property that no two processes running concurrently access the same part of the tree no longer holds. Zaring gives a rather convoluted solution to this problem involving additional synchronization. As with most of the works surveyed in this paper, practical experiments allowing to assess the relative merits of the various methods are missing.

Alblas [Alb91d] discusses asynchronous concurrent evaluation in which all instructions, including `LEAVE`, can be performed in parallel. He shows, as Zaring had pointed out but, in our opinion, somewhat overlooked, that this leads to much more synchronization operations than synchronous `VISIT`s only. To overcome this he tries to refine the static analysis of the AG by splitting the subsets in totally-ordered partitions into smaller independent subsets. According to Alblas himself, and in the absence of practical experiments, this complicates the analysis quite a lot without much effect on the behavior of the parallel evaluator.

Since we believe that synchronous visit-sequence-based parallel evaluation is, until now, the most efficient parallel evaluation technique, we'd like to explain with rather great detail how it can be efficiently implemented. Indeed, bad implementation choices have much greater consequences on the behavior of the system than in the sequential case. The detailed algorithm we present below, and which clearly exposes all low-level synchronization, is basically an interpreter for visit-sequences enriched by parallel constructs. It has appeared independently in slightly different forms in [Mar90] and [Zar90]. To avoid Zaring's problems, we assume that parallel blocks contain `EVAL` and `VISIT` instructions only (or other nested parallel blocks).

Traditionally [Kas91b], visit-sequences are represented and actually implemented as linear sequences of instructions. Here we need to be able to represent DAG structures embedded in linear sequences. We hence introduce the following instructions:

- `FORK` p marks the beginning of a block of p parallel branches; the next p instructions must be `BR_START` instructions.
- `BR_START` n marks the beginning of a parallel branch, the next instruction of which is located n places further.
- `BR_END` m marks the end of a parallel branch and states that the instruction following the whole parallel block is located m places further.

For instance, the parallel visit-sequence presented above would be implemented as:

```
BEGIN 2; FORK 2; BR_START 2; BR_START 4;
   EVAL {X.b}; VISIT 2,X; BR_END 4;
   EVAL {Y.b}; VISIT 2,Y; BR_END 1;
EVAL {Z.d}; LEAVE 2.
```

The parallel evaluator is made of one (big) process per available processor. Each processor runs the same visit-sequence interpreter presented in Fig. 2 and 3. The various concurrent activities are (small) *tasks*; at a given time there may exist many more tasks than processors: each processor executes one task while the others wait in a task queue.

The algorithm is designed so that each tree node contains no information in addition to the classical fields (pointers to the sons, production number, attribute values); bookkeeping information for the visits and tasks is stored in separate, explicitly-managed structures. This helps keeping space consumption to what is strictly necessary. In particular we do not require that each node contains a pointer to its father.

A visit descriptor contains a pointer to some node, a field for holding the return address into the visit sequence of the father, for `VISIT` and `LEAVE` instructions, and a pointer to the parent visit descriptor. In a sequential evaluator a stack is sufficient to store visit descriptors but in a parallel evaluator the control structure needs to be a "cactus stack," i.e. a tree. Note that, apart from that, `EVAL`, `VISIT` and `LEAVE` instructions are implemented as in a sequential evaluator. Choose-VSC(p,i) returns the index of the first instruction of the i-th visit in the visit-sequence for production p. The s parameter to an `EVAL` instruction is a set of attribute occurrences in the current production; we assume that, for each such set, a corresponding evaluation procedure has been generated.

In addition to the tree and attributes, the global (shared) variables are the visit sequences themselves, with `FORK` and `BR_START` and `BR_END` instructions as above, and the

```
type visit-descr = record
        node: ↑tree;
        VSC: VS-index;
        parent-visit: ↑visit-descr;
end record;

type task-descr = record
        visit: ↑visit-descr;
        VSC: VS-index;
        join-counter: integer;
        join-lock: lock-type;
        parent-task: ↑ task-descr;
end record;

globalvar VS: VS-array { the visit sequences themselves };
    tq: queue of task-descr { with lock };
    root: ↑tree { root of global tree };
localvar visit, tmpv: ↑visit-descr;
    task, tmpt: ↑task-descr;
    VSC: VS-index;
    node: ↑tree;
    i: integer;
    done: boolean;

doforever
    task ← dequeue(tq) { waits until tq becomes non-empty };
    visit ← task↑.visit;
    VSC ← task↑.VSC;
    node ← visit↑.node;
    done ← false;
    repeat
        case VS[VSC] is
        EVAL s:      call procedure eval-s, with node as unique parameter;
                     VSC ← VSC + 1;
        VISIT i, j:  tmpv ← alloc-visit-descr();
                     tmpv↑.parent-visit ← visit;
                     visit ← tmpv;
                     visit↑.VSC ← VSC + 1 { return address };
                     node ← node↑.son[j];
                     visit↑.node ← node;
                     VSC ← choose-VSC(node↑.prod, i);
        LEAVE i:     if node = root then halt endif;
                     VSC ← visit↑.VSC;
                     tmpv ← visit;
                     visit ← visit↑.parent-visit;
                     free-visit-descr(tmpv);
                     node ← visit↑.node;

    { to be continued }
```

Figure 2: Algorithm for the parallel visit sequence interpreter

```
        FORK p:     task↑.join-counter ← p;
                    unlock(task↑.join-lock) { to make sure };
                    for i ← 1,...,p do
                        tmpt ← alloc-task-descr();
                        tmpt↑.visit ← visit;
                        tmpt↑.parent-task ← task;
                        tmpt↑.VSC ← VSC + i;
                        enqueue(tmpt)
                    endfor;
                    done ← true; { fall back to the scheduler }
        BR_START n:
                    VSC ← VSC + n;
        BR_END m:   tmpt ← task;
                    task ← task↑.parent-task;
                    free-task-descr(tmpt);
                    lock(task↑.join-lock);
                    decrement(task↑.join-counter);
                    i ← task↑.join-counter;
                    unlock(task↑.join-lock);
                    if i = 0 then { block is complete, continue with the visit sequence }
                        VSC ← VSC + m
                    else { fall back to the scheduler }
                        done ← true
                    endif
        endcase
    until done
enddo
```

Figure 3: Algorithm for the parallel visit sequence interpreter (continued)

task queue with associated lock variable (managed by procedures "enqueue" and "dequeue"). Note that the queue is a simple FIFO; tasks have no priorities. A task descriptor in this queue contains a pointer to some visit descriptor, itself containing a pointer to the relevant node, a visit sequence counter pointing to some instruction in the visit sequence for this node, a "join counter" and associated lock variable, and a pointer to the descriptor of the calling task. The (private) state variables of each process are a pointer to the current node, a visit sequence counter pointing to the current instruction in the visit sequence and pointers to the current task and visit descriptors. We omitted the details of process initialization and termination. Explicit chaining of task and visit descriptors allows nested parallel blocks. The implementation of FORK and BR_END instructions should now be easy to understand.

A possible bottleneck lies in the access to the task queue through "enqueue" and "dequeue," which must be protected by a lock. It is possible to alleviate this by having one queue per processor instead of a single queue: each processor enqueues in its own queue only while it can dequeue from any queue through polling; this does not eliminate locking but strongly increases concurrency. The only other synchronization points are in BR_END instructions; they cause no problem because the number of branches in each

parallel block is generally small. In addition, allocation and deallocation of visit and task descriptors should use one free list per process(or). Lastly, an obvious optimization is, on `FORK`s, to enqueue task descriptors for all the branches but the last one and then proceed to execute the latter on the same processor.

The results of section 5.2 will show that this technique allows to create enough parallel tasks to efficiently use a multi-processor machine.

Segment-based approach Klein [Kle91] describes another approach based on the notion of *segments*. The local dependency graph of each production is (statically) partitioned into a number of segments, with the condition that each two attributes of a non-terminal are always in the same segment in every production or in distinct segments. At run-time the various local 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 to obtain a computation order but the evaluation of the instances in distinct segments can proceed concurrently, with synchronization points defined by cross-segment dependencies. This general model is highly dynamic, so Klein proceeds with giving local-segments-choosing criteria that allow to statically precompute a lot of information, including the evaluation order in the (global) segments by means of visit-sequence-like code. His criteria thus apply to *parallel ordered attribute grammars*, a new class that includes the OAG class but which Klein otherwise fails to characterize. Once the local segments are given, the POAG test and the construction of parallel visit-sequences are performed by a polynomial algorithm quite similar to Kastens' OAG test. Klein also describes several methods for computing local segments that meet his POAG criteria and lead to different grains of parallelism. This work is attractive but unfortunately Klein gives little material for comparing it with other approaches; in particular, since processes are synchronized by explicit rendez-vous, the worst-case performance is potentially very bad (although simulations performed by Klein on realistic AGs exhibit good speedups).[2]

Klein and Koskimies [KlK89, KlK90] give a variation of this method restricted to one-pass evaluators (1L-AGs). This is a bit surprising because the main advantage of one-pass (sequential) evaluation, namely that evaluation can be performed during parsing without constructing a physical tree, is lost in parallel evaluation, while the strong restrictions on the expressive power remain. They nevertheless bring in an interesting result by describing how parallel one-pass evaluation can proceed concurrently with parsing and tree construction.

4.2 Incremental evaluation on shared-memory machines

4.2.1 Dynamic methods

Kaplan and Kaiser [KaK86] propose a simple modification of Reps' dynamic propagation algorithm [Rep84] for incremental attribute evaluation that is well-suited for tightly-coupled multiprocessor machines. Their algorithm is given in Fig. 4. The aim is to perform as many attribute (re)evaluations in parallel as possible, by choosing at a given

[2] *Note from the author*: since my German is a little weak, I may have missed something in Edi Klein's thesis. If so, I hope he will forgive me.

```
proc IncEval(t, r);
decl t: fully attributed tree;
    r: node in t with inconsistent attributes;
    s: set of attribute instances;
    a: attribute instance;
begin
    SetUp(t, r, s);
    foreach a ∈ s do
        spawn Propagate(a)
    endfor;
    halt
end IncEval;

proc Propagate(a);
decl a: attribute instance;
    s: set of attribute instances;
begin
    Evaluate(a);
    if a has changed then
        Expand(a)
    endif;
    Remove(a, s);
    foreach a ∈ s do
        spawn Propagate(a)
    endfor;
    halt
end Propagate;
```

Figure 4: Dynamic concurrent incremental evaluation algorithm

time the complete set of independent instances and spawning a separate process to work on each of them.

The synchronization between processes is hidden in the procedures Expand, Remove and, to a lesser extent, SetUp, which maintain the (incremental) dependency graph (the "model," to use Reps' terminology); this graph is shared between all the processes. SetUp(t, r, s) takes a tree t and a node r in t that has inconsistent attributes, constructs the (initial) model and returns in s the attribute instances that are ready for evaluation because they have in-degree 0 in the model. Expand(a) augments the model to include all instances that depend on a (see [Alb91c, Rep84] for more details). Remove(a, s) removes instance a and its associated edges from the model and returns in s the attribute instances that are ready for evaluation. Great care must be taken in the implementation of these procedures to ensure the consistency of the model and the correctness of the "ready for evaluation" sets that are returned to calling processes. This requires locking at the attribute instance level.

The above algorithm is easily extended to handle multiple asynchronous subtree replacements, i.e. replacement that occur at separate locations in the tree concurrently with each other and with attribute (re)evaluation. This situation naturally occurs in parallel implementations of attributed tree transformation systems [Alb91c]. Assuming that each subtree replacement is an atomic operation, modifications to the above reevaluation algorithm are as follows. SetUp is turned into an atomic operation that *merges* the model it creates with the "global" model; this merging is a *union* operation that equates identical edges and vertices in both models.[3] Then, each time a subtree replacement is performed, the IncEval procedure is executed and will *add* to the current model the initial local model at the replacement point. From this point it makes no difference if the new model for the new subtree overlaps with others or not; Remove and Expand will return the correct results regardless. Kaplan and Kaiser prove that, given a tree in which k subtrees are replaced asynchronously, any attribute instance is (re)evaluated at most k times.

Although, as far as we know, no real implementation of this algorithm has been built,

[3]This can be achieved, for instance, by attaching the edges in the model to the attribute instances that are themselves attached to the tree nodes.

we believe that it should perform rather well in practice because, on one hand, every process does not have to wait for results of other processes and can perform some useful work as soon as it is started (which is not the case for e.g. Fang's scheme described in section 4.1.1) and, on the other hand, each process accesses the model with a rather high degree of "locality" (one instance and its immediate successors), which means that it does not have to synchronize with too many other processes. In fact, it is easy to combine the above algorithm with the notion of tree-based distribution by partitioning the model into separate regions that are each managed by a distinct, much coarser-grained process, thus considerably reducing the amount of synchronization and inter-process communication. Kaplan and Kaiser have used this idea to adapt their algorithm to distributed architectures, see section 4.4 below.

4.2.2 Static methods

Zaring uses as the basis for his parallel incremental evaluation algorithms the simple sequential visit-sequence-based method described by Engelfriet [Eng84] and derived from one devised by Reps [Rep84, Ch. 9]. In this method the same visit-sequences as for exhaustive evaluation are used to drive a slightly modified interpreter: `VISIT` and `LEAVE` instructions are treated as no-ops if they would visit an *inactive* node; initially only the tree-modification point and its parent are active; a node is made active whenever one of its attribute instances is reevaluated and its new value is different from the old one. Evaluation is started at the first instruction of the visit-sequence of the tree-modification node. Engelfriet proves that this method is optimal in the usual sense [Rep84], even if it unnecessarily evaluates some attributes; it is obvious that it can be implemented very efficiently.

The problem with extending this scheme with asynchronous execution of `EVAL` instructions is that the decision to visit or not to visit some node through `VISIT` or `LEAVE` depends on whether the node is active or not, and this status may not be completely determined at the time when it is checked if there exist `EVAL` processes that may set the node active and have not yet finished to run; these `EVAL` instructions are called *(potential) activators* of the node. One way to overcome this problem is to require potential activators to be executed non-asynchronously, but this incurs a loss of concurrency. An alternative is to maintain at each node a counter of still-running potential activators. Zaring shows that asynchronous execution of `VISIT` instructions only or `LEAVE` instructions only is impractical because it would require a complicated attribute-instance locking protocol that would probably incur much run-time overhead. Asynchronous execution of both `VISIT` and `LEAVE` instructions (`EVAL` instructions being executed asynchronously or not) is easily achieved by the same simple attribute-instance locking scheme as for exhaustive evaluation; Zaring gives several algorithms to implement this.

Synchronous parallel incremental reevaluation is much easier: it's a quite simple, although not obvious, combination of the above (sequential) reevaluation algorithm based on the notion of active nodes and the **fork/join** model of evaluation presented in section 4.1.2, easily implemented with "atomic counters" as in the detailed algorithm presented in the same section. Zaring [Zar90] gives all the details and proofs.

Results obtained by simulation show that all these types of parallel incremental reevaluation methods perform roughly similarly and rather well, but only actual implementa-

tions and extensive experiments will allow to correctly assess them.

4.3 Exhaustive evaluation on distributed architectures

When working on a distributed architecture it is necessary to make data as local as possible to each process, in order to reduce synchronization and inter-process communication, which are much more expensive than on a tightly-coupled architecture. Global data structures that are heavily used in methods for shared-memory machines must instead be split into pieces that will be "owned" by the various processes.

Böhm and Zwanepoel [BöZ87] describe a concurrent attribute evaluation method that is well suited to run on, and has actually been implemented on, a distributed architecture, namely a network of workstations connected by a high-speed network. In this scheme, the parser builds the parse tree and splits it at nodes chosen by the AG author, so that the resulting tree fragments have a size greater than some specified minimum (fragments can be nested). It then sends these fragments to separate sites that perform attribute evaluation on each of them. The evaluator that runs on each site uses a hybrid static/dynamic method: dynamic evaluation is used only for the attributes belonging to tree nodes that lie on the path from the root of the local subtree to (the root of) a remotely-evaluated subtree. During the reconstruction of the subtree from the linearized form received over the network, it is determined for each node n whether it lies on such a path. If not, attributes at n and below will be evaluated by a static (visit-sequence-based) evaluator and no dependency information is computed. Otherwise, n's attributes are entered in the global (to the process) dependency graph that will be used for dynamic evaluation [Alb91b], together with the edges induced by the semantic rules. Attributes of the children of n that are to be evaluated by the static sub-evaluator are also added to this graph, however they are connected by the transitive static dependencies computed at generation time rather than by the true dynamic dependencies. When tree construction is complete, evaluation starts in topological order of the dependency graph, as is classical for dynamic evaluation. When all predecessors for a statically evaluated attribute become available, the appropriate static visit procedure is invoked. When all predecessors for a remotely evaluated attribute become available, a packet of information is sent to the remote site and evaluation continues with other ready vertices in the graph. Attributes that depend on remotely evaluated attributes are made ready for evaluation when the values of those remote attributes are received over the network. The efficiency of this scheme is highly dependent on the choice of the boundary nodes, which should minimize transfer of (generally bulky) information over the network. In addition, certain attributes can be specified as holding a priority, which directs the dynamic evaluator to try to compute them as soon as possible; attributes that must be transmitted to remote sites should be specified as holding a priority.

Böhm and Zwanepoel actually implemented their method on a collection of Sun-2 workstations running the message-based V-System and connected by a 10-megabit/s network. Experiments were made on an AG describing a compiler for a sizable subset of Pascal to VAX assembly language. Trees can be split at statements, statement lists, procedure declarations and lists of these. Measurements on typical 1000-lines-or-so programs exhibit a maximum speedup of about 3 when 5 machines are used, while the evaluator running on a single machine has performance equivalent to the vendor-supplied Pascal

compiler. Böhm and Zwanepoel describe various implementation techniques, e.g. applicative implementations of symbol tables and code strings as trees, that help make efficient use of parallelism. Detailed analysis of the behavior of the combined evaluator shows that less than 0.1% of the attributes are evaluated dynamically. It also shows that, because inter-process communication is so expensive, this method does not scale up very well to a large number of processors. Furthermore, since processes are so coarse-grained, the amount of concurrency is very sensitive to the way the tree is split into separate fragments: the best results are achieved when fragments are of approximately equal size but they plummet down quickly with more uneven decompositions.

4.4 Incremental evaluation on distributed architectures

Kaplan and Kaiser adapted their own work on concurrent incremental evaluation (see section 4.2.1 above) to work on loosely-coupled (distributed) architectures and in a different context described below. Their concurrent incremental reevaluation algorithm presented in Fig. 4 is amenable to "data splitting" as discussed at the beginning of previous section. The idea is, as in Böhm and Zwanepoel's scheme, to partition the tree into connected regions that are owned by separate processes that maintain them in a consistently-attributed state.

When a subtree replacement occurs in some region (Kaplan and Kaiser did not discuss inter-region tree modifications), attribute reevaluation begins as usual. Each time the model is expanded, a check is performed to see if the expansion would cause a flow of attribute propagation onto a remote process, or become dependent on an attribute "owned" by a remote process. For the former case, the model is built as normal and then cut so as to separate all remote attributes belonging to a given remote process from the rest of the model. A special vertex, called *remote*, is inserted into the model so that all dependencies flowing over the cut out of the local region now flow via *remote*. Attribute values flowing *into* the local region are handled by giving them the previous values they had when propagating into the local region (see below) and making them immediately ready for evaluation. The model is then altered so that anything flowing out of *remote* is discarded, all attributes flowing into the local region are made ready for evaluation, and evaluation continues as normal.

Propagate is modified so that, when the special vertex *remote* becomes ready for evaluation, local attribute information is packed into a message that is sent to the remote process. In the remote region attribute reevaluation will be triggered as if there had been a subtree replacement at the node that forms the border between the two regions. The model for that remote region is now built in dual to the approach just described for the local region: attributes flowing into the (new) local region are considered as ready for evaluation and attributes that flow out are assigned a *remote* vertex as described above.

It must be noted that this distributed algorithm is not optimal in the usual sense, because some attribute instances may be reevaluated more than once, even when a single local tree modification and no remote tree modification occurs. Furthermore, the global efficiency of this scheme is, as Böhm and Zwanepoel's one, highly dependent on the choice of region-separating nodes: they must be chosen so that dependencies flowing through them are reduced to a minimum. For instance, procedures in a block-structured language are good candidates to become regions. It is not clear however how such "good" nodes

can be automatically determined by static analysis of the AG.

As described above, the algorithm is suitable for all kinds of distributed architectures and all kinds of tree-modifying events. However Kaplan and Kaiser devised their method for a more specific application. They envisage a model of program development in which several programmers each use a workstation to develop a module, with the workstations connected by a high-speed network (this is a quite common setting nowadays). Each programmer runs a copy of an AG-based programming environment that performs incremental attribute reevaluation according to the above algorithm. The regions are the different modules that form the complete program at hand. To each programmer, and to the attribute reevaluation process that she runs, the module she is currently editing is the local region; all other modules are remote regions. The above algorithm works well in this setting, except that a few problems require some adaptation: a remote module may be "dormant" because it is not being edited or it is inaccessible due to network or machine failure, or some user may be performing some editing operation that temporarily makes the tree unsuitable for receiving attribute information. The problems are handled by the introduction of the general concept of *firewalls*. A firewall acts as a barrier behind which a region/process can shelter if it is not ready to accept change propagations from other regions. When an attribute propagation reaches a firewall that is in place, it queues until the firewall goes down, at which point the change is propagated to the region as if a tree modification had taken place at the firewall. See [KaK86] for more details. Not waiting for remote modules to propagate back changed values achieves a tradeoff between accurateness of semantic information and undue delays that is quite satisfying for the envisaged application context.

Kaplan and Kaiser's work was pursued by them and Kaiser's group at Columbia University, mainly by implementing it into the Mercury system [KKM87] and refining the engineering details; a very thorough survey of all this work appears in [KaK90], together with references to other more specific publications. Recently, Kaplan and Kaiser [KaK90] proposed a variation of the priority-based incremental reevaluation algorithm presented in [ACR87] that accommodates multiple asynchronous tree modifications and multiple execution threads.

Shinoda and Katayama [ShK90] propose a variation of AGs in which every node in an attributed tree is defined and represented as a persistent *object* in the object-oriented terminology. An object definition involves the specification of its static semantics (an AG in the usual sense) and its dynamic semantics (collection of messages it is able to respond to). According to Shinoda and Katayama, OOAGs are useful to specify and implement large and complex persistent data structures such as databases or programs in a software development system. Among the (dynamic) messages an object is able to respond are tree modification messages. These modifications trigger incremental reevaluation of the static attributes. Shinoda and Katayama's method is called "locally controlled distributed incremental attribute evaluation." It is a rather simple variation of Kaplan and Kaiser's distributed algorithm described previously, in which each node is a complete region. Propagation of the "changed" or "unchanged" status of an attribute instance is performed by sending special kinds of messages, called signals, that do not require an answer; Shinoda and Katayama define the Propagate and Relax signals. Signals sent to different objects can be handled concurrently by these objects and in turn generate other signals to other objects. Shinoda and Katayama claim that their scheme eliminates the

concurrency bottleneck represented by multiple atomic accesses to the (shared) global model. In our opinion this is hard to verify, given that their experimental implementation of this scheme is done on a sequential machine.

Alblas [Alb90] proposes a combination of Böhm and Zwanepoel's scheme with that of Kaplan and Kaiser, suitable for incremental reevaluation on distributed architectures. As with Kaplan and Kaiser's method, the tree is split in a "top" fragment and a collection of non-nested "bottom" subtrees, each of them being owned by a separate process. As with Böhm and Zwanepoel's scheme, both static and dynamic incremental reevaluation methods are used. Bottom subtrees are reevaluated by a static algorithm similar to the one of Reps and Engelfriet presented at the beginning of section 4.2.2, except that the decision to visit or not to visit a parent or child node is reexamined at each `VISIT` and `LEAVE` instruction by determining whether any of the "input attributes" to this visit has changed instead of relying on the coarser nodewise (in)active status bit. The top fragment is processed by an entirely dynamic algorithm similar to the one by Kaplan and Kaiser as described at the beginning of this section, or to the dynamic part of Böhm and Zwanepoel's algorithm. Concurrency is achieved by processing distinct bottom subtrees in parallel, since the dynamic algorithm is able to manage this. Alblas [Alb90] also describes how to delay reevaluation until after several tree modifications (he places his work in the context of attributed tree transformation systems) by introducing the notion of "safe approximations" of a consistently attributed tree. 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 a reevaluation phase or in a tree transformation phase, and that it will accept attribute change propagation only when in a reevaluation phase.

Feng [Fen91] improves Kaplan and Kaiser's distributed incremental reevaluation algorithm presented at the beginning of this section by applying the rather well-known idea of bypassing copy-rules chains in each sub-model. He proves that his algorithm is better in both time and communication costs.

5 Experiments with a Practical Implementation

For a long time we have thought of experimenting with parallel attribute evaluation. The availability of a general-purpose, shared-memory multi-processor machine at INRIA made this possible. Through the development and use of the FNC-2 AG-processing system [JoP91] and its ancestors, we already had great experience with both static and dynamic (sequential) attribute evaluation methods, and we knew that the former were much more efficient than the latter. We also knew about Fang's ideas and were wary that indiscriminate use of parallelism could do more harm than good, so we were interested in a method that would exploit "useful" parallelism only. This led us to rediscover Schell and Zaring's ideas presented in section 4.1.2. In addition we felt that only an actual implementation on a real machine would allow to correctly assess the value of parallel attribute evaluation.

5.1 Implementation

Our experiments were conducted on a Sequent Balance machine whose features seemed, at first sight, to perfectly match our needs: multi-processor (10 processors) with shared memory; **fork/join** model of parallel computation; C language and Unix interface; local availability.

At first, we thought of directly using the `fork` and `wait` system calls of the DYNIX operating system running on the Sequent; this fitted well in our sequential implementation of visit sequences, which FNC-2 translates into recursive procedures. After modifying the evaluator generator to have it produce parallel visit sequences, the second step was thus to create a new back-end which translated them into recursive procedures with calls to `fork` and `wait`; the translation of individual semantic rules was unmodified, except for the insertion of `private` or `shared` storage specifiers. Space optimizations are not possible as for the sequential case [Kas91b]; in consequence, we use the classical solution of storing all the attributes at tree nodes.

We quite early aborted the implementation of the **fork/wait** model when we read in the manual that DYNIX implemented only heavyweight UNIX processes, in which the **fork** system call involves a complete copy of the non-shared portion of the memory image of the parent process; the corresponding time penalty (of the order of 50 ms) was intolerable, given that the processes we generate are quite small. Our solution is hence to replace the recursive procedure with **fork/wait** model by the purely interpretive algorithm presented at the end of section 4.1.2.

We have actually made several simplifications to this algorithm. First, each branch in a parallel block is restricted to be a single `VISIT` instruction. Secondly, space for visit descriptors (pointer to father and return address) and for part of the task descriptors (join counter and lock) is pre-allocated in the tree nodes rather than allocated on demand in separate descriptors. This simplifies the implementation (visit descriptors disappear entirely) and probably gains some speed, at the expense of space consumption. However we have very early come to have one task queue per processor, because we believed that access to the single queue was a bottleneck before discovering that our first disappointing results were the fault of the shared-memory allocator (see below).

5.2 Preliminary experiments

Parallel evaluators were constructed for two AGs that are in our opinion typical of, although much simpler than, those that could be written for compiler construction applications. They respectively describe the contextual constraints checking and translation to an intermediate form of the block-structured toy language *simproc* [JoP89, App. D]. These AGs were not modified at all for the experiment. The generated visit-sequences exhibit 9 parallel blocks out of 17 total blocks (`BEGIN–LEAVE` pairs), with 2.55 branches per parallel block, for the first one, and 9 parallel blocks out of 11 total blocks, with 2.44 branches per parallel block, for the second one. This already shows that even simple AGs offer a lot of parallelism, and using larger AGs would lead to even more exploitable opportunities for parallelism and hence better results than those presented below.

These AGs were run on a number of typical source texts. We give here measures for only one of them (Fig. 5). The figures were computed by measuring the real (wall-clock)

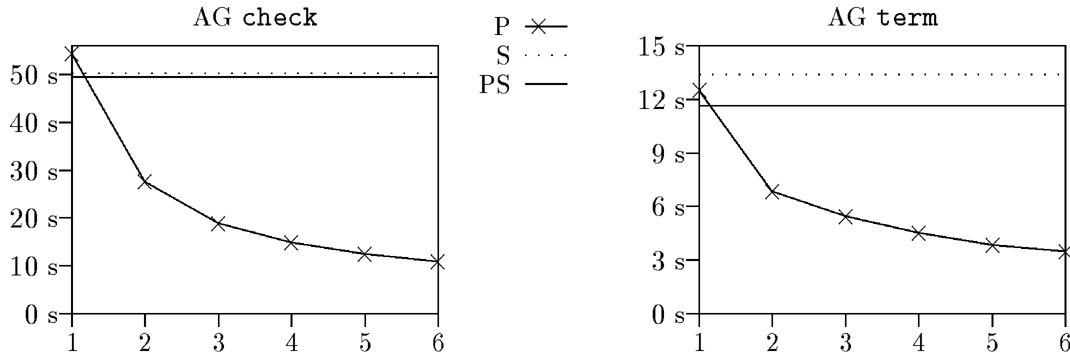


Figure 5: Typical practical results. Horizontal axis is the number of processors, vertical axis is (real) running time. P is the parallel interpreter, S is a sequential interpreter, PS is the parallel interpreter driven by sequential visit sequences.

running time of the evaluators on a varying number of processors. It must be noted that this time does not include parsing and tree construction but only pure attribute evaluation. We believe that these figures are quite satisfactory, given the simplicity of our evaluation method and implementation. Furthermore, as said above, they are sort of a lower bound: we expect measures on more realistic examples to be even better.

The performance of the parallel evaluator (P) on the `check` AG is a perfect example of successful parallelization: on one processor it is comparable (only 8% slower) to a sequential evaluator (S), which means that the overhead caused by the parallelization is quite small, and adding more processors leads to quasi-linear speedup. The efficacy, i.e. the quotient of the speedup by the number of processors, smoothly decreases from 98% with 2 processors to 83% with 6 of them; these are very good figures. The parallel evaluator running on only two processors is already 82% faster than the sequential one.

The performance on the `term` AG is less good (but still reasonable, given the small size of the examples). This is because `check` performs much more data traversal, i.e. pure computation, than data creation (allocation), whereas it is the opposite for `term`, which merely creates a new representation of the input program that is quasi-isomorphic to the input tree. The shape of the curve for `term` (efficacy of 91% with 2 processors and only 60% with 6 of them) clearly shows that the bottleneck is here the shared-memory allocator, which is out of our control. The fact that the sequential evaluator (S) runs a bit slower on `term` than the parallel evaluator driven by sequential visit sequences (i.e. with no `FORK`, etc.) can be explained by the fact that, on `VISIT` instructions, the latter stores the return address in the (pre-allocated) tree while the former stores it and the pointer to the father in a separate stack that it must explicitly manage; this "overhead" is less visible in more computation-bound AGs such as `check`.

These experiments, although certainly not intensive enough, already show that synchronous parallel attribute evaluation is able to fully exploit the power of the machine it runs on, and that it holds its promises of making best use of cheap parallelism and reducing synchronization costs to a minimum. Its implementation on shared-memory multi-processor machines is easy. We have thus proved that parallel attribute evaluation is quite possible and effective.

In addition to performing more experiments, we intend to pursue this work in two

directions:

- restrict parallelism by a "less blind" static analysis of the AG, i.e. parallelize only at well-chosen nodes (as Böhm and Zwanepoel and others do, but more automatically); this would lead to bigger tasks and smaller overhead;
- devise specific space optimization techniques for parallel evaluation methods.

6 Conclusion

In this paper we have explained why parallel attribute evaluation is both valuable and feasible, and reviewed the various methods therefore that have appeared in the literature. We have also shown on a prototype implementation that preliminary results are quite encouraging. We believe that this topic will increasingly attract both theoretical and practical research.

References

[Alb91a] H. Alblas, "Introduction to Attribute Grammars," in this volume, 1991.

[Alb91b] ———, "Attribute Evaluation Methods," in this volume, 1991.

[Alb91c] ———, "Incremental Attribute Evaluation," in this volume, 1991.

[Alb91d] ———, "The Trouble with Parallel Attribute Evaluation: an Explanation by Example," oral presentation at IFIP WG 2.4 meeting, Grassau, Jan. 1991.

[Alb90] ———, "Concurrent Incremental Attribute Evaluation," in *Attribute Grammars and their Applications (WAGA)*, Pierre Deransart & Martin Jourdan, eds., Lect. Notes in Comp. Sci. **#461**, Springer-Verlag, New York–Heidelberg–Berlin, Sept. 1990, 343–358.

[ACR87] B. Alpern, A. Carle, B. Rosen, P. Sweeney & F. K. Zadeck, "Incremental Evaluation of Attributed Graphs," IBM T.J. Watson Research Center, research report RC 13205, Yorktown Heights, NY, Oct. 1987, Also published as Tech. Rep. CS-87-29, Dept. of Comp. Sc., Brown Univ., Providence, RI.

[Boc76] G. V. Bochmann, "Semantic Evaluation from Left to Right," *Comm. ACM* **19** (Feb. 1976), 55–62.

[BöZ87] H-J. Böhm & W. Zwanepoel, "Parallel Attribute Grammar Evaluation," in *7th Int. Conf. on Distributed Computing Systems*, Radin Popescu-Zeletin, Gérard Le Lann & Kane H. Kim, eds., Sept. 1987, 347–354.

[DJL88] P. Deransart, M. Jourdan & B. Lorho, *Attribute Grammars: Definitions, Systems and Bibliography*, Lect. Notes in Comp. Sci. **#323**, Springer-Verlag, New York–Heidelberg–Berlin, Aug. 1988.

[Eng84] J. Engelfriet, "Attribute Grammars: Attribute Evaluation Methods," in *Methods and Tools for Compiler Construction*, Bernard Lorho, ed., Cambridge Univ. Press, New York, NY, 1984, 103–138.

[Fan72] I. Fang, "FOLDS, a Declarative Formal Language Definition System," Comp. Sc. Dept., Stanford Univ., PhD thesis, report STAN-CS-72-329, Dec. 1972.

[Fen91] A. Feng, "Efficient Incremental Attribute Evaluation in Distributed Structure-Oriented Software Environments," Dept. of Inf. and Comp. Sc., Faculty of Eng. Sc., Univ. of Osaka, PhD thesis, Jan. 1991.

[Fra83] J. L. Frankel, "The Architecture of Closely-coupled Distributed Computers and their Language Processors," Dept. of Applied Maths., Harvard Univ., PhD thesis, 1983.

[GZZ89] T. Gross, A. Zobel & M. Zolg, "Parallel Compilation for a Parallel Machine," *ACM SIGPLAN Notices* **24** (July 1989), 91–100.

[JoP91] M. Jourdan & D. Parigot, "Internals and Externals of the FNC-2 Attribute Grammar System," in this volume, 1991.

[JoP89] ———, *The FNC-2 System User's Guide and Reference Manual* release 0.4, INRIA, Rocquencourt, Feb. 1989, This manual is periodically updated.

[KKM87] G. E. Kaiser, S. M. Kaplan & J. Micallef, "Multiuser, Distributed Language-Based Environments," *IEEE Software* (Nov. 1987).

[KaK90] S. M. Kaplan & G. E. Kaiser, "Parallel and Distributed Incremental Attribute Evaluation Algorithms for Multi-User Software Development Environments," Dept. of Comp. Sc., Columbia Univ., report CUCS–019–90, New York, Apr. 1990.

[KaK86] ———, "Incremental Attribute Evaluation in Distributed Language-based Environments," in *5th ACM Symp. on Principles of Distributed Computing*, Aug. 1986, 121–130.

[KaK90] ———, "An Incremental Priority-Based Solution to the Multiple Asynchronous Edit Problem," Dept. of Comp. Sc., Columbia Univ., report CUCS–010–90, New York, Nov. 1990.

[Kas80] U. Kastens, "Ordered Attribute Grammars," *Acta Inform.* **13** (1980), 229–256.

[Kas91a] ———, "An Attribute Grammar System in a Compiler Construction Environment," in this volume, 1991.

[Kas91b] ———, "Implementation of Visit-Oriented Attribute Evaluators," in this volume, 1991.

[KeW76] K. Kennedy & S. K. Warren, "Automatic Generation of Efficient Evaluators for Attribute Grammars," in *3rd ACM Symp. on Principles of Progr. Languages*, Jan. 1976, 32–49.

[Kle91] E. Klein, "Ein Modell zur Generierung paralleler Attributauswerter," Fakultät für Informatik, Univ. Karlsruhe, Dissertation, 1991.

[KlK89] E. Klein & K. Koskimies, "The Parallelization of One-Pass Compilers," GMD, Arbeitspapier 416, Karlsruhe, Nov. 1989.

[KlK90] ———, "Parallel One-pass Compilation," in *Attribute Grammars and their Applications (WAGA)*, Pierre Deransart & Martin Jourdan, eds., Lect. Notes in Comp. Sci. #**461**, Springer-Verlag, New York–Heidelberg–Berlin, Sept. 1990, 76–90.

[Knu68] D. E. Knuth, "Semantics of Context-free Languages," *Math. Systems Theory* **2** (June 1968), 127–145.

[Kui89] M. F. Kuiper, "Parallel Attribute Evaluation," Faculteit Wiskunde en Informatica, Rijksuniv. Utrecht, PhD thesis, Nov. 1989.

[KuS90] M. F. Kuiper & S. D. Swierstra, "Parallel Attribute Evaluation: Structure of Evaluators and Detection of Parallelism," in *Attribute Grammars and their Applications (WAGA)*, Pierre Deransart & Martin Jourdan, eds., Lect. Notes in Comp. Sci. #**461**, Springer-Verlag, New York–Heidelberg–Berlin, Sept. 1990, 61–75.

[**Lip79**] D. E. Lipkie, "A Compiler Design for Multiple Independent Processor Computers," Dept. of Comp. Sc., Univ. of Washington, PhD thesis, Seattle, WA, 1979.

[**Mar90**] B. Marmol, "Évaluateurs d'attributs parallèles sur multi-processeurs à mémoire partagée," Univ. d'Orléans, rapport de DEA, Sept. 1990.

[**MöW91**] U. Möncke & R. Wilhelm, "Grammar Flow Analysis," in this volume, 1991.

[**Rep84**] T. Reps, *Generating Language-based Environments*, MIT Press, Cambridge, MA, 1984.

[**Sch79**] R. M. Schell, "Methods for Constructing Parallel Compilers for Use in a Multiprocessor Environment," Dept. of Comp. Sc., Univ. of Illinois at Urbana-Champaign, PhD thesis, report UIUCDCS-R-79-0958, Feb. 1979.

[**SWJ88**] V. Seshadri, D. B. Wortman, M. D. Junkin, S. Weber, C. P. Yu & I. Small, "Semantic Analysis in a Concurrent Compiler," *ACM SIGPLAN Notices* **23** (July 1988), 233–239.

[**ShK90**] Y. Shinoda & T. Katayama, "Object-oriented Extension of Attribute Grammars and its Implementation Using Distributed Attribute Evaluation Algorithm," in *Attribute Grammars and their Applications (WAGA)*, Pierre Deransart & Martin Jourdan, eds., Lect. Notes in Comp. Sci. #**461**, Springer-Verlag, New York–Heidelberg–Berlin, Sept. 1990, 177–191.

[**Van88**] M. T. Vandevoorde, "Parallel Compilation on a Tightly-coupled Multiprocessor," DEC Systems Research Center, SRC report 26, Palo Alto, CA, Mar. 1988.

[**Zar90**] A. K. Zaring, "Parallel Evaluation in Attribute Grammar-based Systems," Dept. of Comp. Sc., Cornell Univ., PhD thesis, Tech. Rep. 90-1149, Aug. 1990.