Yale University
Department of Computer Science

Efficient Breadth-First Expansion on the Connection Machine or: Parallel Processing of L-Systems

Ron Y. Pinter and Shlomit S. Pinter

YALEU/DCS/TR-719
July 1989

Efficient Breadth-First Expansion on the Connection Machine or: Parallel Processing of L-Systems

Ron Y. Pinter[1,2]

Shlomit S. Pinter[3,4]

Dept. of Computer Science
Yale University
New Haven, CT 06520

July 1989

Abstract

We present efficient algorithms for a breadth-first expansion kernel on the Connection Machine[5]. We use the formalism of L-systems to model the highly dynamic nature of such computations, especially with regard to the spatial distribution of the data (unlike numeric or other regular computations). We claim that an L-system, being defined as a set of production rules that are to be processed in breadth-first order, indeed captures the inherent parallelism present in this computational structure.

Our algorithms vary in complexity as we move on from simple (deterministic, context-free, flat) L-systems to more elaborate ones. They make intensive use of the data parallel operations that are offered by the target machine, especially the **scan** operator which is realized by a parallel-prefix implementation. The algorithms are presented here using a new, simple abstract programming model of the machine, but they have also been programmed and executed on the CM-2 at Yale; preliminary timing results are given herein.

Keywords: algorithms, breadth-first expansion, data parallel programming.

[1]On sabbatical leave from the IBM Israel Scientific Center.
[2]Work supported in part by ONR grant number N00014-89-J-1906.
[3]On sabbatical leave from the Dept. of Electrical Engineering, Technion — Israel Institute of Technology.
[4]Work supported in part by NSF grant number DCR-8405478.
[5]Connection Machine is a trademark of Thinking Machines Corporation.

1 Introduction

The process of breadth-first expansion (BFE) represents a commonly occurring schedule of computations arising in many different situations, such as searching a large solution space, **and**-parallelism in executing Prolog programs, event-driven simulation of discrete processes (*e.g.* synchronous circuits), simulation of biological growth [7], generation of graphics images (*e.g.* Fractals), and more. We have observed that **L-systems** [6] are particularly suitable for modelling the basic phenomenon which characterizes this computational process, namely the generation of frontiers (or layers) of events that are scheduled for handling. This formulations allowed us to study the potential parallelism that is inherent to this process and come up with a number of algorithms that can drive a BFE engine effectively on machines that strongly support data parallelism, such as the Connection Machine.

An L-system is a grammar, comprising a number of production rules over a finite alphabet of symbols. Unlike grammars in the Chomsky hierarchy (see, for example, [3]), the firing rule for a derivation is a parallel one, as follows: *all* symbols in the current word are replaced simultaneously, each with the right-hand-side of some rule (as defined by the grammar and its type). Thus, the semantics of this process is already defined in terms of a parallel computation; the main difficulty in realizing this computational structure is the continuous expansion and contraction of data.

L-systems encapsulate the gist of the problem, and allow us to concentrate on these spatial irregularities rather than dealing with other, irrelevant (to this issue) details. Specifically, the result of each single production may occupy much more space than the current symbol, and this expansion is compounded in subsequent steps (which may also involve some shrinkage). We address the question of how to handle this evolution by an efficient algorithm that can be hosted on the target machine architecture, particularly utilizing the special operations that offer massive data parallelism, such as parallel prefix [1, 5].

Using a dynamic data structure (such as a linked list), as is commonly done in a sequential implementation, is not good enough since the general storage allocation schemes proposed in [2] for the Connection Machine are too inefficient. Fortunately, L-systems (and the problems they model) have enough structure to allow a fast, space efficient family of algorithms that do take advantage of the data parallelism.

The basic algorithm that we present in this paper deals with deterministic, context-free, and flat (*i.e.* non-bracketed) L-systems. It shows how to overcome the data expansion (and contraction) problem, and the same essential idea is then used in the more complicated algorithms. Then we show how to deal with bracketing, which involves maintaining adjacencies between symbols (or events). The most complex variant includes, in addition, context recognition and resolving non-determinism.

In order to present our algorithms, we have developed a simple programming model for the Connection Machine which abstracts away many of the irrelevant details. We believe that this new model is, at least for the purpose of expressing and developing algorithms of the kind presented here, better than current existing models in the way it supports the rather different way of thinking which is required compared to sequential programming. It can be used rather conveniently without compromising the quality of the resulting code and the way it takes advantage of the CM architectural features.

We first define L-systems and some related terminology, and then present our programming model for the CM in Section 2. Section 3 describes the basic algorithm for deterministic, context free, and flat L-systems. In Section 4 we explain how to deal with bracketing, context, and non-determinism, followed by analysis and results in Section 5. We conclude with a discussion on how this research can apply to other machines and computational models.

2 Preliminaries

An L-system is a set of *rewriting rules* (or *productions*). Each rule is a pair of words over a given alphabet Σ, which are customarily separated by an arrow ($\rightarrow$); the two words are called the left hand side (or predecessor) and right hand side (or successor) of the rule, respectively. The L-system is applied to a word over Σ^* as follows: at every round, each single character in the word is replaced (independently) by the right hand side of one of the rules that apply to it. If no such rule is found, the symbol[1] is left intact; if more than one left hand side matches the application conditions, then one rule is picked non-deterministically. The system is applied to an initial (non-empty) word, called an *axiom*, but there is nothing special about the axiom in terms of the semantics of the system.

The simplest case is that of 0L-systems (or context-free systems), in which every left hand side is a single character, and then a rule applies if its left hand side is equal to the symbol in the word. For example, consider the following D0L-system (D for deterministic, since each character appears as a left hand side exactly once) over the alphabet $\{a, b, c, d\}$:

$$\begin{array}{lcl} a & \rightarrow & bc \\ b & \rightarrow & da \\ c & \rightarrow & b \\ d & \rightarrow & a \end{array}$$

Starting from the axiom a, the words that are generated by subsequent applications of the system are bc, dab, $abcda$, $bcdababc$, and so on. Notice that the rules are applied to all symbols simultaneously, and the new word is the result of concatenating the appropriate right hand sides in a row.

Context can be used to select the rule that applies to a given symbol. To denote context, the left hand side is written in the form $\alpha < \sigma > \beta$, where σ is the letter to be replaced and α and β are strings (over Σ^*) that must be present to the immediate left and right of σ in the current word. Notice, that unlike context-sensitive languages of the Chomsky hierarchy, only the single symbol is being replaced (when the context is found), **not** the whole left hand side. In general, a (k, l)-system is one in which the longest left context is of length k, and the right one — of maximal length l.

[1]Notice that we use the terms *character* and *symbol* interchangeably; usually, symbol refers to the occurrence of a character in a word that is being processed.

One further feature of L-systems is *bracketing,* which facilitates a deviation from the linear structure implied by words. Brackets (denoted by [and]) can appear on the right hand side of any rule, and they may be arbitrarily nested in words, thus allowing the dynamic creation of tree-like structures. Brackets delineate the context of rules, namely the (right and left) context of each symbol is determined by symbols that appear to its right and left *at the same nesting level* (regardless of the length of intervening symbols at lower levels).

We now turn to the target machine. The Connection Machine (both the idealized version described in [2] and the working models as documented in [8]) has a distributed memory, SIMD architecture. Data entities are organized in *xectors*, or `p-vars`; each such value is best thought of as a (one dimensional) vector whose elements reside one per processor, with the processor id being the index of the element. Instructions are either scalar, *i.e.* they apply to every element independently (*e.g.* add one to every entry, subtract the ith element of a from the corresponding element of b), or they apply to the xector as one entity, *e.g.* perform a parallel prefix add (or `scan-add`) on a.

In addition, xector elements can be copied (or sent) to other xectors. Sometimes, the elements are to be arranged in the same order, and then the copy operations can be performed locally in each processor (that is — element by element, in parallel), but often a different ordering of elements is desired, *e.g.* each processor wishes to send its value to the processor whose id is 1 more than its own. To support this latter type of copying, which includes not only permutations but also any random send-receive pattern, a routing network is used whose internal behavior is beyond the control of the programmer. One can only give it hints, saying whether there are no *collisions* (*i.e.* fan-in or fan-out of elements), and whether there are few or "many" such collisions to be expected.

An important notion is that of *processor selection.* At any given point in time, only a designated subset of processors is marked *selected*, meaning that the current instruction applies only to the processors in this set. Processors can be selected and de-selected dynamically based on their addresses, values of xector entries that they hold, or any other dynamically expressible condition. For certain operations (such as a scan) the processor array can also be broken into *segments*, by means of setting up *boundaries,* which delineate the effect of the operation and make it appear as if it were applied to each segment separately (in parallel).

For convenience, one can also store data on the host (which holds and issues the instruction stream). This facility is reserved for scalars or small structures, and is usually used to keep initialization data which is loaded into xectors at the beginning of a CM computation. Data interaction with the host is time consuming and should be avoided if a short elapse time is desired.

From a programming point of view, one can write C* or *Lisp [8] programs in a style not unlike *APL* [4]. Xectors are used as one-dimensional *APL* arrays, element-by-element operators are the same as the scalar operators when applied to arrays, parallel prefix operations are like scans, and sends perform assignments with indexing[2]. For example, $y[p] \leftarrow x$ means that the elements of xector y at processor id's denoted by xector p are assigned the

[2]We could also express selection as reduction with a bit-vector, but processor selection is usually applied to whole program fragments, not to short expressions, thus we prefer to denote it as a separate construct, using a keyword

appropriate elements of xector x in the same order; x and p had better be of the exact same length, and the assignment affects only those portions of y which reside in selected processors. Likewise, $y \leftarrow x[p]$ means that the elements of x whose indices are denoted by p are to be assigned to y (which then has the same length as p). Notice that while repetitions are allowed in the latter case (meaning that a particular element is broadcast to several locations), ambiguity arises in the former case; we assume that such conflicts are resolved at random, *i.e.* one of the possible values gets written.

In what follows, we shall use a pseudo-*APL* notation, using key-words rather than Greek letters to denote operators. We feel that this language allows us to focus on the algorithmic rather than the low-level issues. When presented along with a set of declarations for the variables' types, such program can be compiled into (say) *Lisp code relatively straightforwardly[3].

Finally, there is one additional complication concerning the mapping of xectors to the actual machine. The way things are set up, all xectors are of the same length, which is the number of available processors, and length adjustments are all effected through process selection. Sometimes, however, one would like to have xectors longer than the number of physically available processors. This is facilitated through the definition of *virtual processors,* of which there could be many more than physical ones. In reality, the xectors are folded onto themselves, so that each physical processor holds more than one element. The cost of doing so, both in terms of the additional time necessary to complete an operation as well as the extra drain on the routing network, is non negligible and must be taken into account even though it is (in some sense) hidden from the programmer.

3 Executing D0L-systems

We first describe the data structures that we use when processing a D0L-system. The current word w is stored as a xector, one symbol per processor (consecutively, starting from the left, using zero indexing), and only the first $|w|$ processors are selected. The symbols themselves are encoded into the numerals $0, \ldots, |\Sigma| - 1$. Since each symbol appears only once on the left hand side of a rule, the rules are ordered by the numeral assigned to their left hand sides, thus the rule number is equal to the encoding of its left hand side.

We also maintain three additional supporting data structures, whose utility will become clear when we present the algorithm itself. Each can be implemented as a xector, as an array on the host, or fully replicated in each processor (in which case an array of xectors is used), depending on efficiency considerations[4]. They are:

- `lgt` — the length of the right hand side of each rule (indexed by rule number),

[3]We have done this by hand, but an effort at automating the process is well under way, as refelcted by the companion report, "Efficient Compilation of Array Expressions for the Connection Machine", by Luis F. Ortiz and Ron Y. Pinter, YALEU/DCS/TR-720.

[4]The implementation may change in different variations of the algorithm, and it may even include some further variations on the three basic alternatives, as discussed in Section 5.

- `flat-rhs` — a concatenation (into one sequence) of the right hand sides of all rules (in order), and
- `start-rule` — the starting position of each right hand side in `flat-rhs` (again, indexed by rule number).

These entities can be easily compiled up-front (from the rule base), and their computation cost (which is minimal anyway) does not affect the running time of the algorithm.

The problem that arises when executing the productions of a D0L-system, is that the length of the current word keeps changing (mostly growing, unless there are λ-rules). Not only that, but if we want to maintain the above, simple structure for storing words, room needs to be made for the resulting right hand sides, thereby "pushing" the processors that are assigned to symbols up and up. The key observation that leads to an efficient algorithm for the above process is that we can predict precisely where each right hand side is going to be mapped based solely on the rule number and the information in the current word, without need to sequentialize the computation by processing one symbol at a time. This is done by using the scan-add operation, which is considered an efficient operator on the CM, and then using the result as the target for storing the appropriate right hand sides all at once, as we shall describe in more detail now.

The scan-add operation is defined[5] as follows: if y = scan-add(x), then $y[i] = \sum_{j=0}^{i-1} x[j]$ (and $y[0] = 0$). Recall that when segment boundaries are being used, then the scan is applied to each segment separately. Now, at each processor, we first look up the length of the right hand side of the rule that is going to be applied according to the rule number that is already stored in it (which is equal to the current symbol of w); these lengths are (locally) assigned to the xector l. Next perform a scan-add on l which results in each processor obtaining the sum of the lengths of the right hand sides of all the productions that will be applied to the symbols that appear to the left of its own symbol. Each entry in this value (stored in p) is exactly the processor id to which the first (leftmost) character of the symbol's right hand side should be assigned (assuming 0 indexing).

Since the right hand sides are of varying lengths, and there is no atomic assignment operation for strings on the CM, we are not quite done yet. The next step is to prepare address templates for moving the individual characters in each right hand side to the right place. We first change the processors selection to include as many processors as the length of the new word; this length is simply the sum of all elements in l, and it is obtained during the calculation of the scan-add. Then we use the result of the scan-add to set up boundaries between the processors, establishing the target region for each production. In each such segment, we generate (again, in parallel, using one scan-add and one scan-copy operation, the latter simply propagating a value from left to right) consecutive integers starting from the appropriate value of `start-rule`. Finally, these numbers are used to index `flat-rhs`, and in one operation we distribute all characters (from `flat-rhs`) to all the selected processes. Notice that each processor already contains the appropriate index, so the referencing can be performed in one step if `flat-rhs` is replicated, or using one send operation (using the routing network) in case it is a simple xector.

[5]We use the version in which the sum does not include the current element.

Now we are ready to present the D0L processing algorithm. We assume all the necessary supporting data, as well as the current word (be it the initial axiom or a word that has been derived by a number of steps) and its length, are loaded into the machine. Following is the pseudo-code for one derivation step; this is the body of the main loop, which should be iterated indefinitely or a prespecified number of times. Some brief comments appear (in curly braces) along with the code; we also use the keyword **using** to denote the usage of segments. Additional explanations and a clarification on temporary variables appear right after the code.

```
select the first length-of-current-word processors
l ← lgt[w] { l now contains the lengths of the right hand sides of the rules that
    need to be applied}
p ← scan-add(l) { p points to the target address of the right hand sides, as
    explained in the text}
length-of-new-word ← p[length-of-current-word]+l[length-of-current-word]

select the first length-of-new-word processors
sgt-bdry[p] ← all-1 { mark segment boundaries at processors corresponding to
    the beginning of right hand sides of rules that constitute the new word}
rhs-ind[p] ← start-rule[w] { this records rule starting indices in flat-rhs at the
    appropriate positions}
rhs-ind ← scan-copy(rhs-ind) using boundaries marked in sgt-bdry
ind ← scan-add(all-1) using boundaries marked in sgt-bdry
w ← flat-rhs[rhs-ind + ind] { this writes the new version of the word using one
    operation. The correct address is readily available.}
{ *** at this point, one can insert whatever processing ***}
{ *** that needs to be performed with the generated word ***}
length-of-current-word ← length-of-new-word { this up-date is necessary for the
    next iteration}
```

In this code, we have used a number of intermediate variables, whose purpose is as follows:

- `length-of-current-word, length-of-new-word` — two scalars, whose values are self explanatory

- `sgt-bdry` — a Boolean xector, used to divide the processor domain into segments each of length corresponding to the right hand side of the rule which will occupy that segment (according to the value of w).

- `rhs-ind, ind` — these two xectors are addressing templates, partitioned into segments per sgt-bdry, whose sum is the index into flat-rhs indicating the character to be used in the actual production. The first, rhs-ind, generates the base index, namely the starting address for each rule in flat-rhs, whereas the second generates sequences of successive integers, from 0 up to each corresponding length of a right hand side.

- `all-1` — the constant xector 1, namely a copy of 1 at every processor

To illustrate this algorithm, here is an example of one derivation step using the D0L-system of Section 2. Let us assume that the current word is *bcdababc*, whose length is 8. It is encoded as 1 2 3 0 1 0 1 2 and is stored in processors 0 through 7 which are currently selected. `lgt` is 2 2 1 1, `flat-rhs` is 1 2 3 0 1 0, and `start-rule` is 0 2 4 5. We now display the values computed in each row of the algorithm, one by one:

```
l ← 2 1 1 2 2 2 2 1
p ← 0 2 3 4 6 8 10 12
length-of-new-word ← 13

select the first 13 processors
sgt-bdry[p] ← 1 1 1 1 1 1 1 1
rhs-ind[p] ← 2 4 5 0 2 0 2 4
rhs-ind ← 2 2 4 5 0 0 2 2 0 0 2 2 4
ind ← 0 1 0 0 0 1 0 1 0 1 0 1 0
w ← 3 0 1 0 1 2 3 0 1 2 3 0 1
length-of-current-word ← 13
```

4 Handling General L-systems

In this section we show how the algorithm of the previous section, which processes D0L-systems, can be extended to dealing with L-systems that contain bracketing, context, and non-determinism. Bracketing by itself (regardless of context determination) is easy to handle: once generated (by the right hand side of some rule), brackets are retained as is (either implicitly or by adding the productions [→ [and] →] to the rule base)[6].

Context affects only rule selection. This needs to be done right at the beginning of the loop-body presented in Section 3, and once we know which rule is to be applied to each symbol we can use the rest of the algorithm as is. For sake of presentation, we consider (1,1)-systems, namely grammars in which each left hand side can have up to three characters: a left context, the symbol to be replaced, and a right context (such systems are often called 2L-systems). The generalization to (k, l)-systems is straightforward.

Without bracketing, rule selection amounts to a pattern matching phase. In the current setting, the best way to parallelize this process is as follows: Send each symbol in w once to its left neighbour and once to its right neighbour; this can be implemented as two collision free sends[7] or by using the CM's grid (NEWS) operations. Then, at each processor, compare the three characters to all left hand sides of rules, and if the system is deterministic there will be at most one match (if there is none then use the default rule). This can be implemented either by running a fast pattern matcher on all processors simultaneously, which looks for

[6]In case the subword between brackets is eliminated, *i.e.* it becomes the empty word, the enclosing brackets can be cancelled using context sensitive rules.

[7]Since this process involves both a two-fold fan-out as well as a two-fold fan-in, we cannot use just one send or one receive; these operations require that the mapping is either one-to-many or many-to-one.

the context of each symbol as a pattern in one text comprising all the left hand sides of rules concatenated together[8], or by using some other table look-up method.

When more than one rule can be applied to a given character depending on context[9], the correspondence between character numbers and rule numbers breaks down. This is easily remedied by keeping another xector alongside with w which records the rule numbers (and gets updated once per round); whenever w was used to identify a rule rather than a symbol, it should be replaced by this new xector.

Recall that in a system with bracketing, a neighbour is defined as the closest symbol at the same level of nesting (both on the right and on the left). Thus, a neighbour is not necessarily the closest symbol in the string. We employ the following scheme in order to maintain neighbourhood in a deterministic 2L-system: with each symbol (processor) in the word, we continuously keep the two processor id's holding its neighbours, as well as the neighbours' symbols (this, by the way, eliminates the need for the two sends mentioned above and rule selection becomes entirely local). The processor id's are stored in the xectors `left` and `right`, and the symbols to the left and the right are kept in `left-symbol` and `right-symbol`.

In order to keep maintaining these structures for future rounds, each processor sends the projected (new) id's of the prospective neighbouring symbols of the right hand side that is going to be used in its expansion immediately before the rules are fired. These new processor id's are calculated using the values of l and p (as well as the static information that has been compiled for the rule base), and are temporarily stored in `next-left` and `next-right`. The new symbols are then updated after the firing, using `next-left` and `next-right`, which subsequently replace `left` and `right` as well.

To make sure that the neighbourhood information is properly maintained inside each right hand side (and not just at the borders between them), we form a neighbourhood template for each rule which tells us who the left and right neighbours of each symbol are relative to the beginning of the right hand side itself (using 0 indexing). This is what actually sets the whole structure up when brackets appear on the right hand side of a production. For example, for $a[ca]b$ the left neighbour pointers are 0 0 0 2 0 0, and the right ones are 5 0 3 0 0 0 (notice we use 0's both to denote that the information is irrelevant, such as for the bracketing symbols themselves or when the neighbour is undefined, as well as for the leftmost and rightmost characters in the right hand side, which are taken care of as above). The concatenation of these templates is stored in `flat-ln` and `flat-rn` (similarly to `flat-rhs`) at compile time.

Thus, the following code needs to be added to the body of the loop, replacing the assignment to w (which remains intact in there):

```
next-left[p] ← p [left] + (lgt[w[left]] – all-1)
next-left ← flat-ln[rhs-ind + ind] + scan-copy(next-left) using boundaries
    marked in sgt-bdry
```

[8]At the expense of using up more processors, the pattern matcher can be sped up considerably, using CM specific instructions such as scan-and.

[9]This is not to be confused with non-determinism.

```
next-right[p] ← p [right]
next-right ← flat-rn[rhs-ind + ind] + **scan-copy**(next-right) **using** boundaries
    marked in sgt-bdry
w ← flat-rhs[rhs-ind + ind]
left-symbol ← w[next-left]
right-symbol ← w[next-right]
left ← next-left
right ← next-right
```

In the above code there are some fence post problems arising from the fact that the leftmost and rightmost symbols in w do not have a left and right neighbour, respectively. These are easily dealt with using special values and some scalar operations which are not detailed here. More importantly, we assume that there are no λ-rules, and also that none of the right hand sides is of the form $[\alpha]$. Such cases do not occur in most applications we have seen; nevertheless, in order to treat them, a somewhat more complicated scheme is necessary, whose time complexity depends on the nesting depth. Also, a slightly different definition of context is sometimes used (see [7]), which our algorithm does handle (but the details are too long for this extended abstract).

Finally, the issue of non-determinism. First we have to aggregate in each processor the numbers of all rules that could be applied to the symbol it holds. In case the system is context free, this can be done by preparing the list for each character at compile time (in the same way we compute `lgt` etc.), and then store the appropriate list at each processor according to its current symbol. In the context sensitive case, we simply record the prospective rules during the matching process. In either case, once the lists are there, all we have to do is make an independent random selection in parallel (as supported on the CM) of which rule is chosen.

5 Analysis and Results

As we have mentioned in Section 3, there are several ways for realizing the assisting structures `lgt`, `flat-rhs`, and `start-rule`. If time is our prime minimization criterion, then it is best to replicate all three values and maintain each as an array at every processor (making it into an array of xectors). This way, xector indexed references to these entities (*e.g.* lgt[w]) are *local*, and the algorithm is then dominated by two factors: the two array assignments using p as their index, and the three scan operations.

First, in reality, the two array assignments are done with one send (`*pset`) operation on the CM-2 (using the `:notify` option); moreover, the send is collision free, so it takes only one time unit. As for the scans — these operations are advertized as being particularly efficient on the CM, and the whole point is to use them (rather than other constructs) whenever possible so as to utilize the machine's potential parallelism. Scans are implemented as parallel prefix computations [5], whose time complexity is logarithmic in the number of processors participating in the operation. This is the case even when segment boundaries are being used [1]. So, all in all, in this fastest of all realizations, each iteration of the main algorithm should take time that is logarithmic in the length of the word w being produced; in practice (on the CM-2), the time is the logarithm of the total number of processors on the machine (which is a constant).

If the rule base is large, and `flat-rhs` does not fit into each processor, it can be implemented as a xector (residing in, say, the first r processors, where r is the number of elements in `flat-rhs`). This makes the assignment to w (as well as those to `next-left` and `next-right`, for the bracketing case) somewhat expensive, and it becomes the critical part of the whole computation, since it involves massive fan-out. Another feasible alternative — in case the alphabet is small — is to write one character at a time, using $|\Sigma|$ separate one-to-many sends (the code to do so can be easily generated at "compile" time), thereby somewhat reducing the routing congestion. Better yet, we could replicate the xector as many times as it fits, *i.e.* storing one copy of it (one element per processor) in the first r processors, then storing a second copy in the following r processors, and so on (up to the total number of processors available). Then the assignments to w need to be carefully orchestrated so as to achieve as many disjoint sends as possible so as to reduce fan-out; this can be done either deterministically or by randomization.

At the extreme, assuming processors with very little memory, all these entities could be implemented as arrays on the host, and then each array access operation amounts to distributing the array entries one at a time in a loop (with as many iterations as the length of the array, but each value is broadcast in one parallel step selecting the appropriate processors).

Another factor affecting the performance is the use of virtual processors (VPs). This becomes necessary when the length of the derived word, w, exceeds the number of physically available processors. If we know (or can safely estimate) the size of the longest word that will be encountered, one could naively allocate as many processors up-front. With the current implementation of VPs, this slows matters down considerably even when only a small number of processors is selected (and they reside one per physical processor). The alternative is to dynamically allocate VPs as we go, but this incurs a different kind of overhead. We are still experimenting with this issue, but one observation is that it would have been nice if one could have explicit control over the association between VPs and actual processors (without having to resort to embeddings in higher-dimensional grids, as can indeed be done).

The algorithm of Section 3 was implemented in *Lisp Release 5.0 on a CM-2 with 8K processors (used with only 4K at a time). We measured 6msec per production (the raw clock speed being 6.7MHz) with all assisting structures as xectors (not xector arrays) and without VPs; this went up to about 20msec per production with a fixed VP-ratio of 16. When all the assisting structures are fully replicated (as xector arrays), these figures go down to 4msec and 13msec, respectively.

6 Conclusions

Both the algorithms that we have described in this paper and the programming model in which they are presented apply to a variety of physical architectures. The requirements from the target architecture are essentially three:

- one-dimensional arrays should be easily embeddable,
- simultaneous assignments among processors (à la send) are efficient, and
- parallel prefix operations with segmentation (à la scan) are supported.

Several networks, such as the hypercube (which has a natural Hamiltonian cycle), satisfy these requirements, and our algorithms can be easily mapped to them without loss of efficiency. For other networks, which may display a weakness in one of the above points, certain adjustments will have to be made in the algorithms, and we plan to pursue this further.

On the negative side, even the CM did not prove to be as fast as we had expected. For one, it would be nice if the complexity of the parallel prefix computation would depend on the number of currently selected processors. Second, the handling of collisions when the fan-out is large is less than wonderful, as far as we as users can tell. Third, it would be nice to have a primitive for many-to-many sends (and have the machine find the best way to realize it on a dynamic basis). We expect that some of these problems will be addressed by future versions of both the software and the hardware.

Another item of interest for further research is to build applications on top of the breadth-first-order driver that was developed. Expressing the algorithms in the applications' domains may take advantage of the L-systems formalism, and our algorithms supply the computational feasibility for developing such tools. Any situation in which events are being scheduled in a layered fashion, namely a number of independent tasks are spawned at each stage, can be handled in essentially the same manner, and the algorithms described herein would apply (with appropriate adjustments).

All in all, we have presented a comprehensive solution to an interesting and useful class of problems, which is geared towards the target parallel architecture. As we all know, in developing these algorithms a radically different way of thinking is required compared to sequential programming, and we think this work contributes a new angle to this issue.

Acknowledgements. We would like to thank Przemyslaw Prusinkiewicz, of the University of Regina, Canada, for introducing us to L-systems and for many helpful discussions. We would also like to thank Michel Jacquemin of Yale University, and Paul Oppenheimer and Jim Salem of Thinking Machines Corporation, for help in programming the Connection Machine and for comments on an earlier draft of this paper.

References

[1] G. Blelloch. Scans as primitive parallel operations. In *International Conference on Parallel Processing*, pages 355–362, 1987.

[2] W. D. Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[3] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979.

[4] K. Iverson. *A Programming Language.* John Wiley, New York, 1962.

[5] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *JACM*, 27(4):831–838, October 1980.

[6] A. Lindenmayer. Mathematical models for cellular interaction in development, Parts I and II. *Journal of Theoretical Biology*, 18:280–315, 1968.

[7] P. W. Prusinkiewicz. Graphical applications of L-systems. In *Graphics Interface '86 — Vision Interface '86*, pages 247–253, May 1986.

[8] Thinking Machines Corporation, Technical Report HA87-4. *Connection Machine Model CM-2 Technical Summary*, April 1987.