Towards a complete framework for parallel implementation of logic languages: The data parallel implementation of SEL

GIANCARLO SUCCI AND CARL UHRIK
LII/DIVA, Università di Trento
Via Zeni 8, I-38068 Rovereto, Italy

SUMMARY
Although logic languages, due to their non-declarative nature, are widely proclaimed to be conducive in theory to parallel implementation, in fact there appears to be insufficient practical evidence to stimulate further developments in this field. The paper puts forward various complications which arise in assuming a solely process parallel approach as a possible explanation for this situation.

As an alternative, data parallelism is posited as an underutilized forte of logic programming. The paper illustrates a data parallel implementation of a particular language called SEL which is based on sets. Thus, SEL (set equational language) is introduced as an example of logic language which lends itself to an efficient data parallel implementation.

The strategy of this implementation assumes an abstract machine called SAM (set-oriented abstract machine) which is based on the WAM (Warren abstract machine). SAM serves as an intermediary between the SEL language and the target machine for the implementation, the Connection Machine. Finally, some preliminary benchmarks are presented.

1. INTRODUCTION
The Kowalski equation[1], Algorithm = Logic + Control, is a milestone towards a full understanding of logic languages. Logic languages are those languages which describe the ‘logic’ of algorithms and leave the control implicit, as opposed to imperative languages which explicitly state the flow of the control of an algorithm but not its logic. It is a common view that since logic languages do not limit the control of algorithms, they offer many opportunities for exploiting the inherent parallelism of algorithms. However, despite ongoing discussions, results towards efficient parallel implementations of logic languages are relatively scarce, centering mainly around Prolog[2].

Part of the problem is that the Kowalski equation does not lend itself to being interpreted as a constructive theorem. Rather, it should be regarded as a general guideline for researchers, quite like the D’Alambert theorem, which, while it states that the number of solutions of an nth degree equation is n, it does not provide any means for computing them.

This paper explores the parallel implementation of logic languages by proposing that the main form of parallelism to be exploited is data parallelism rather than process parallelism as will be described shortly. To illustrate this thesis, the authors examine the implementation of a logic language called SEL (set equational language) which has already proven itself useful in the context of more traditional implementations on sequential machines[3].

Firstly, some brief details of parallelism which regard SEL’s principle data structures, sets, are introduced in Section 2. Then, Section 3 describes with more precision how data parallelism is reflected in SEL itself. Then Section 4 discusses an implementation of SEL.
on an abstract machine which can be realized easily on any SIMD (single instruction, multiple data) machine, but specifically the Connection Machine will be taken as the target machine. Finally, in Section 5, some results of preliminary experiments are described, and Section 6 gives a brief discussion and draws some conclusions.

2. PARALLELISM AND SETS

Firstly, we consider generally the types of parallelism which might be possible in declarative languages. Subsequently, we explore data parallelism in more more detail as it applies to sets in SEL, the targeted declarative language of the parallel implementation to be described later.

2.1. Data parallelism, process parallelism and some problems

In considering an algorithm, one usually looks for two kinds of parallelism: process parallelism and data parallelism. With respect to logic languages, process parallelism identifies clauses (i.e. 'statements' in a logic language) that can be executed in parallel. On the other hand, data parallelism identifies data clusters or data structure elements over which a particular operation can be executed in parallel.

Suppose that, given a set \( s_1 \), a second set \( s_2 \) has to be computed,

\[
 s_2 = \{ x^3 : x \in s_1 \} \cup \{ 3x : x \in s_1 \}
\]

Consider first that a pure process parallel approach (depicted in Figure 1) executes the computation of the result set as the computation of two of its subsets \( \{ x^3 : x \in s_1 \} \) and \( \{ 3x : x \in s_1 \} \) in parallel, with the execution inside either subset proceeding sequentially, in the sense that the elements of the two sets are computed one after another. However, a pure data parallel approach (shown in Figure 2) computes first \( \{ x^3 : x \in s_1 \} \) and then \( \{ 3x : x \in s_1 \} \), but the computation of each is effected virtually in one shot, applying the \( (\cdot)^3 \) and \( 3 \times (\cdot) \) operators to all the elements of \( s_1 \) simultaneously. Here, no attention has been given to the problem of duplicates between the subsets in order to avoid any merging operation that would defeat the parallelism. For now, it is sufficient to say that preconditions can be applied to exclude such conflicts. E.g., \( x^3 = 3x \) has 3 cases to test for: \( x = 0, x = +\sqrt{3}, x = -\sqrt{3} \).

A pure process parallel approach is advantageous because implicit process parallelism is easy to understand, easy to detect and widely present. However, since there are usually overly abundant opportunities for process parallelism, there should be a systematic means for limiting its application, recognizing the fact that the number of processes in real implementations is always limited! Thus in the process parallel approach arises a need to simulate a virtual machine which emulates a very high number of processes through an allocation and scheduling of the limited processor resources, but with the result that performance deteriorates drastically due to lack of complete knowledge of the sensitivities of the scheduling or virtualization to the nature of the processes and overheads of interprocess communication. The extent of the problems thereby imposed may even exceed the benefits gained from the parallel execution itself.

---

1 It can be 10, 100, 1000, ..., but it is usually an order of magnitude lower than what is needed with an unrestricted scheduling of parallel tasks in any moderately large application program.
Input and Output Sets and Operations to be Performed:

Step 1:  \[ 3x(\bullet) \]

Step 2:  \[ 3x(\bullet) \]

Step 3:  \[ 3x(\bullet) \]

Step 4:  \[ 3x(\bullet) \]

Step 5:  \[ 3x(\bullet) \]

Figure 1. The Process Parallel Approach

Figure 2. The Data Parallel Approach
The obvious approach of executing in parallel those tasks which are most critically needed for subsequent tasks and require the highest computational overheads is infeasible since evaluating precisely either the criticality or the execution times of every task is tantamount to executing the tasks themselves. A more practical approach defines different models of process parallelism in order to quasi-optimize performance on different cases: and-parallelism, or-parallelism, stream-parallelism and various hybridizations[4]. Moreover, after scheduling has resolved what to run in parallel, the load balancing question, which must resolve where to run the various processes, arises. Although numerous approaches were proposed, it still remains to be proven if any of them are satisfactory[5].

Assuming a resolution to the scheduling and load balancing issues, there is still the consideration that there is a high time and space cost just to set up and execute the process forking, communications, synchronizations and bookkeeping that occur in the simulation of the virtual machine’s computations on a real parallel machine.

2.2. Data parallelism in set-based languages and its advantages

Although data parallelism (like process parallelism) has the advantage of being easy to understand, it is not so easy to detect, and it may seem not to be widely present in algorithms expressed in declarative languages. However, the language SEL, the subset equational language, developed by Jayaraman[6], represents a remarkable exception.

SEL is based on the set data structure and has expressiveness similar to more common declarative languages such as Prolog and Lisp[7]. Furthermore, it is quite easy to learn as it is based on constructs widely used by mathematicians. The main drawback is that some learning time needs to be invested by programmers who are already expert in logic programming to slightly reshape their programming style to adapt to the SEL paradigm, but the advantages gained greatly outweigh this drawback. A complete description of SEL can be found in [8]. For present purposes it is sufficient to observe that since sets provide an aggregation of data, one naturally sees them as a primary focus for data parallelism.

Now, it is worth observing that taking a purely data parallel approach will overcome the scheduling issues, load balancing problems and execution overheads of the purely process parallel approach since all of the data parallelism to be found can be implemented effectively in the framework of a SIMD architecture. This is because process parallelism is usually effected under a MIMD (multiple instruction, multiple data) architecture with the resource limit of processors being saturated quickly, whereas in SIMD architectures the memory resource limit is not as strict (there is less real-estate per memory cell because the instruction set per processor is more limited; therefore more cells are available).

Thus, since SEL is set-oriented, it is intrinsically data parallel and thereby offers sufficient possibilities for parallelism apart from the process parallel considerations of the language (which, as shown in Section 2.1, may turn out to be problematic in any case). So, here the hypothesis to be tested is that a completely data parallel implementation will be more effective than a completely process parallel implementation of the language. The availability of such massive data parallelism in SEL and the possibility for a parallel implementation to take advantage of it is in part supported by earlier experiments effected on cellular automata[9].
3. DATA PARALLELISM IN SEL

Now, it is worth first considering the details of the data parallelism found in the set operations of SEL. There are three main patterns which seem quite suited for data parallelism in any set-based framework:

- map
- filter
- fold

All three are easily expressed in SEL, and each of these is considered in turn.

3.1. Map

A mapping occurs when a new set \( m \) is defined as the collection of the results of applying the same function \( f \) to all of the elements of a given set \( s \), as shown in Figure 3 and denoted in mathematical notation as

\[
m = \{ f(x) : x \in s \}
\]

where \( f \) is any unary function. In SEL, mappings are written by means of symbolic matching between sets. Thus, for instance, the above mapping is effected in SEL as

\[
aMap(\{x\}_.) \text{ contains } f(x).
\]

which defines the aMap operation, which considers exhaustively all pattern matches of its given argument against a set of elements, one of which is to be stored in a variable \( x \) and the remainder matched to the void variable; since they are not relevant to the function value, nothing further happens to them. aMap's result contains collectively the values of the function evaluated for all such values of variable \( x \) from the pattern being matched.

![Figure 3. Mapping a set](image)

3.2. Filter

Filters occur when a new set \( n \) contains those elements of a given set \( s \) satisfying a given condition corresponding to a predicate \( p \) (i.e. a function returning a Boolean truth value), as shown in Figure 4 and denoted in mathematical notation as

\[
n = \{ x : x \in s, p(x) \}
\]
To properly express filters in SEL, multiple set matching is coupled with the if construct. Thus, the equivalent of the above filter would be

\[
\text{aFilter}(\{x\mid \}) \text{ contains if } p(x) \text{ then } \{x\} \text{ else } \{
\]

which repetitively matches elements of the input set, applies the predicate and inserts the result in the returned set when satisfied.

Clearly, mappings and filters can be combined. For example, the mathematical definition

\[
mn(s) = \{f(x) : x \in s, p(x)\}
\]

would correspond to the SEL definition below:

\[
\text{MN}(\{x\mid \}) \text{ contains if } p(x) \text{ then } \{f(x)\} \text{ else } \{
\]

3.3. Fold

Folding consists of reducing a set \(s\) into a single element \(el\) by means of the recursive application of a binary function \(f\) to pairs of its elements to create a new set using a constant \(z\) as the result in case the argument set is empty. In other words, folding cause an entire set to be ‘folded’ into a single value by means of repeated simple element-wise operations. The process of a generic (abstract) folding is shown in Figure 5.

For instance, in mathematical notation, one might write the following definition:

\[
\begin{align*}
\text{Fold}(S) &= f(\text{Fold}(A), \text{Fold}(B)) & A \cup B = S, A \cap B = \{\}, A \neq \{} \text{, } B \neq \{} \\
\text{Fold}(\{el_1, el_2\}) &= f(el_1, el_2) \\
\text{Fold}(\{el\}) &= el \\
\text{Fold}(\{\}) &= z
\end{align*}
\]

which has an implementation in SEL as
The folding function

\[ f(a,b) = g \]
\[ f(d,c) = h \]
\[ f(h,g) = 1 \]
\[ f(e,l) = r \]

\[ \text{fold}(\{x\}) = f(x, \text{fold}(r)). \]
\[ \text{fold}(\{e_1, e_2\}) = f(e_1, e_2). \]
\[ \text{fold}(\{e\}) = e_1 \]
\[ \text{fold}(\{\}) = z \]

A simple concrete example of a folding is the function `sumset` that computes the sum of the elements of a given set. A possible data parallel execution of `sumset(\{1,2,3,4,5\})` is shown in Figure 6.

Folding raises an issue with respect to non-deterministic execution which is particularly relevant to a parallel implementation of a logic language. Consider a proposed folding `minset` that ‘subtracts pairs of elements of a set’. Due to the non-commutative and non-associative nature of subtraction, the result of executing `minset(\{1,2,3\})` depends on the order in which elements are taken:

\[ (1-2)-3 = -4, \quad (2-1)-3 = -2, \quad (3-2)-1 = 0, \quad 1-(2-3) = 2, \quad 3-(2-3)=4 \]

so this proposed `minset` folding would be problematic. In essence, it is ill-defined. However, if the binary fold function is commutative and associative, the result is always unique[10].

### 3.4. Reduced recursion in the SEL computational paradigm

The subset equational paradigm has an intrinsic advantage with respect to typical declarative languages. It does not require extensive use of recursion. Whereas the occurrence of multiple matching clauses in other languages usually implies recursion, in SEL they do not.

Consider, for instance, the clause that, given a collection, generates the collection of the squares of the elements of the original collection. Its PROLOG specification uses lists:
squareList([],[])  
squareList([X|T],[SQX|W]) :- squareList(T,W), SQX is X*X

whereas SEL uses sets:

squareSet([x|_]) contains x*x

In PROLOG, recursion is needed, while in SEL only multiple set matching occurs, which amounts to iterating over the elements of the set.

In some cases, an intelligent compiler can systematically eliminate a certain type of recursion called \textit{tail-recursion} by simply replacing it with iteration. For example, by making the 'squareList(T,W)' call follow 'SQX is X*X' above, recursion is no longer necessary. However this technique amounts to taking advantage of 'tricks' tied to the non-declarative nature of a supposedly declarative language, and it cannot be used to eliminate more innate cases of recursion. For example, computing the Cartesian product of two lists in PROLOG,

\begin{verbatim}
listCartProd([],[],[])  
listCartProd([X|T],Ty,W) :- mapList(X,Ty,W1), listCartProd(Tx,Ty,W2), append(W1,W2,W)
\end{verbatim}

\begin{verbatim}
mapList(X, [], [],[])  
mapList(X,Y,Ty,[(X,Y)|W]) :- mapList(X,Ty,W)
\end{verbatim}

offers no way to avoid recursion, but the SEL equivalent

\begin{verbatim}
setCartProd([x|_],[y|_]) contains {(x,y)}
\end{verbatim}

does not imply any recursion at all, simply a matching of elements in sets.

4. THE PARALLEL IMPLEMENTATION OF SEL ON THE CONNECTION MACHINE

For a data parallel implementation of SEL, the strategy used here is to put forward an intermediate structure: an abstract machine inserted between the SEL language and a real architecture. This abstract machine is called SAM, which stands for subset abstract machine. SAM serves both as a common denominator among a number of real machines' target architectures and as a way of introducing abstract entities useful to handling SEL structures. Here, only a brief characterization of SAM sufficient to understanding an implementation of SEL on the Connection Machine will be given. A more complete description of SAM can be found in [11] and [12].

\footnote{These target architectures may also include sequential machines (Suns, HPs, IBM's, etc.), for which it has already been implemented, as well as parallel ones (Connection Machines, Alliants, etc.).}
4.1. SAM, an abstract machine for SEL

SAM belongs to the WAM (Warren abstract machine) family[13], from which it inherits most of its structure (heap, stack, push-down list and basic instruction set). SAM, like the WAM, has both a stack and a heap. In both machines, these structures are regarded as infinite. However, any 'real' implementations will face the problem of the 'physical' limits of the target architecture, so that, regardless of the availability of such aids as an appropriate garbage collector, there is always the risk of running out of heap or stack space even for theoretically terminating computations.

SAM is, however, distinguished from the WAM since it possesses a special component called the Active Memory which enables data parallelism. The Active Memory corresponds to an array of cells, each of which contains local memory, a processor and some registers. The role of Active Memory is to hold sets that are used during computations. Each cell contains a single set element; therefore, a set is identified in the Active Memory by a cluster of cells. Such a structure is suited for data parallelism since mappings as well as filters can be implemented in just one shot, computing locally in each cell the functions or the predicates involved. Furthermore, foldings can be performed in logarithmic time by organizing the computation as a tree.

4.2. The CM2 implementation of SAM

SAM is implemented on the Connection Machine 2 (CM2), a SIMD machine with a variable number of nodes (up to 64K). The main idea in the CM2 implementation is to map the Active Memory into the CM2 hypercube and to place everything else in the workstation that acts as the front-end to the CM2. However, this mapping passes through an intermediate structure called the CM Data Array for reasons to be described below.

The key concept is that logic languages are referentially transparent, i.e. their objects never change their values. Despite this property, the most efficient implementations of PROLOG are based on structure copying rather than structure sharing because both the number of structural links to follow (e.g., in lists or sets) and the number of environments to maintain (e.g., in function calls) in structure sharing become quite large (see [14] and [15]). However, in SEL, referential transparency can be realized quite painlessly due to SEL's reduced requirements for recursion and structural data links (sets are essentially unstructured). Moreover, by introducing a structure sharing mechanism immediately between the front-end host and CM2, advantage is gained by heavily limiting the operations executed on the CM2 to those that truly require parallelism.

Note that these issues arise quite independently from the issue of data parallelism in SEL. To understand this better, consider, for instance, the identity operator working on sets:

$$id(S) = S.$$  

This operation takes a set and produces an identical set as a result. In this case, one can operate in two ways:

1. apply a data parallel copy on all the elements of the set, generating a new set identical to the previous one
2. create a new reference to the old set that will never change because of referential transparency.
Both operations can be performed, at least virtually, in a data parallel mode, but while the first one requires some interprocess communication between the host and the CM2 nodes with extra time overhead, the second one is entirely local to the front-end processor. Furthermore, the availability of Active Memory is fairly scarce, and while the second approach performs a cheap copy, the first one requires a space usage linear in the size of the argument. Even though this particular operation may seem overly trivial, situations analogous to the application of \( id/2 \) are very common, and each mapping using such a set would require at least one copy under method 1 above.

From these considerations, the design of the CM2 implementation uses the standard structure copying techniques for any data structure apart from sets, for the traditional reasons hinted at above; however, precisely because of the advantages to be gained, sets are handled with structure sharing. The intermediate structure that allows a clean interface between the CM2 and the front-end host is the CM Data Array mentioned earlier.

The CM Data Array can be regarded as an array of pointers to the data which is distributed within the CM2. The CM Data Array is an array of entries with each entry specifying a set with a tag value (an integer which identifies the set, which of course can be a set of sets), a type field plus some extra information which will be explained later.

The CM2 used for this project employs 16K processors, but nevertheless there is the need of providing some kind of virtualization of the processors, since it may happen that there is not enough space to store all of the desired sets within it. Note that something similar happens in standard sequential implementations using virtual memory; however, there are two significant differences:

1. available space is more limited here
2. cells to be virtualized are active, i.e. they not only store data, but they also perform computations.

The CM2 already provides some native means of virtualization and, despite the fact that it is suboptimal, the present design takes advantage of this feature in certain cases. Here two cases are distinguished: temporary sets (i.e. sets generated as temporary results of a computation) and permanent sets (i.e. sets that are the final result of an assertion). Temporary sets are virtualized explicitly while the management of permanent sets is left entirely to the CM2 Operating System.

A brief sample execution aids understanding of this approach. Consider the following assertion, which applies the function \( g/2 \) twice to all of the elements of the argument set:

\[
\begin{align*}
\text{applyTwice}(S) &= \{ g(g(X)) : X \in S \} \\
g(X) &= X + 1
\end{align*}
\]

The SAM code for this assertion has the following form\(^4\)

\(^3\) The notation name/number identifies the name of the predicate (here \( id \)) and its total number of parameters (here, two parameters are needed, one for the argument plus one for the result).

\(^4\) In the interest of simplicity, the following pieces of SAM code are only schemas, not actual SAM instructions, since some instructions and all of the arguments have been omitted.
[1] map_over
[2] put_value
[3] put_variable
[4] call g/2
[5] put_value
[6] put_variable
[7] call g/2
[8] insert
[9] end_map_over

start the mapping process
put the actual parameter of g in the argument register
put a reference in the result register to a free location for holding the result
make the call to g
as above
as above
make another call to g
create the permanent set and store the final result in it
end of the mapping

As can be seen from the comments, the flow of execution is the following: line 1 prepares the environment for a mapping. Lines 2 and 3 set up the argument register and the result register, respectively for the call to g/2 performed in line 4. The result of this call is a new set whose values are the original values incremented by one twice.

This new set is temporary since it is used only here on the way to producing the final result; therefore it is handled explicitly by the SAM: its values are stored in the same CM cells that are used for the original set, clearly in some other part of its internal memory, i.e. the original content is preserved.

The same situation recurs in lines 5 through 7. Then, the final set is produced. Such a set must be saved in some global area before the termination of applyTwice/2, so that it can be accessed from outside.

This happens when the SAM insert instruction executes: new global space is reserved in the Active Memory and the new set is placed there.

Now consider what happens when the CM2 runs out of free cells? in the case of temporary sets, the problem is solved explicitly by the SAM virtualizing over the Active Memory cells. Regarding permanent sets, everything is left to the CM O/S: the insert instruction reclaims Active Memory space, and the CM2 takes care of supplying virtual space according to some internal policy.

5. EXPERIMENTAL RESULTS

Currently, the system outlined in the preceding Sections has been tested on four simple cases:

- incrmap—a simple map function increments every elements of a set by one
- evenFilter—a simple filter operation takes only the even members from a given set
- sumFold—the sumFold operation takes the sum of all elements in a given set
- N-queens—the outcome for a chess end game with a board configuration consisting of N-queens is computed (given that both players have finite look-ahead)

In running these test cases, the problem of duplicate checks on the resulting sets from the data parallel computations are assumed to be handled by specialized abstract analyzers which identify the possibilities for conflict a-priori and thus largely limit such contingencies.

Three different implementation approaches are compared. Firstly, a Prolog version, written in SICStus Prolog, using lists as the basic data structure, is the base implementation

---

5 Here, the case of reaching the limits of the memory in the process of requesting additional cells is being considered. The case of exhausting all memory in the process of building a single cell has yet to be addressed.
Table 1. Experimental results for SICStus, SAM and CM-SAM on four simple test cases

<table>
<thead>
<tr>
<th>Test Program</th>
<th>Size</th>
<th>SICStus</th>
<th>SAM</th>
<th>CM-SAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>incrMap</td>
<td>10</td>
<td>10</td>
<td>12</td>
<td>0.22</td>
</tr>
<tr>
<td>incrMap</td>
<td>100</td>
<td>180</td>
<td>180</td>
<td>0.22</td>
</tr>
<tr>
<td>incrMap</td>
<td>1000</td>
<td>1620</td>
<td>3300</td>
<td>0.22</td>
</tr>
<tr>
<td>incrMap</td>
<td>10000</td>
<td>15000</td>
<td>405017</td>
<td>0.22</td>
</tr>
<tr>
<td>evenFilter</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>0.27</td>
</tr>
<tr>
<td>evenFilter</td>
<td>100</td>
<td>169</td>
<td>175</td>
<td>0.27</td>
</tr>
<tr>
<td>evenFilter</td>
<td>1000</td>
<td>1740</td>
<td>3280</td>
<td>0.27</td>
</tr>
<tr>
<td>evenFilter</td>
<td>10000</td>
<td>17300</td>
<td>406034</td>
<td>0.27</td>
</tr>
<tr>
<td>sumFold</td>
<td>10</td>
<td>21</td>
<td>14</td>
<td>1.22</td>
</tr>
<tr>
<td>sumFold</td>
<td>100</td>
<td>131</td>
<td>40</td>
<td>1.22</td>
</tr>
<tr>
<td>sumFold</td>
<td>1000</td>
<td>1010</td>
<td>450</td>
<td>1.22</td>
</tr>
<tr>
<td>sumFold</td>
<td>10000</td>
<td>11360</td>
<td>3900</td>
<td>1.22</td>
</tr>
<tr>
<td>4-queens</td>
<td>4</td>
<td>600</td>
<td>540</td>
<td>190</td>
</tr>
<tr>
<td>8-queens</td>
<td>8</td>
<td>15400</td>
<td>6400</td>
<td>520</td>
</tr>
</tbody>
</table>

Note: All run times above (SICStus, SAM, CM-SAM) are given in CPU-ms.

on a Sun4 under SunOS 4.1.1. In addition, a sequential SAM version (i.e. SAM implemented on a conventional sequential machine in C++) was achieved also on a Sun4 under SunOS 4.1.1. Finally, the CM-SAM implementation on a Connection Machine 2 with 8K processors accessed through a Sun4 front-end has been effected.

Table 1 summarizes the results of these experiments. The first two cases (incrMap & evenFilter) are somewhat trivial. The Prolog version (results in the SICStus column) outperforms its SAM counterpart (results in the SAM column). This is not very surprising since one would expect a performance-oriented, more conventional tool to outperform a rough prototype, and these cases (incrementing and testing a condition over a series of numbers) do not really represent operations that can be argued to be very unique to the set-oriented declarative programming paradigm. However, in the sumFold case, the SAM implementation outperforms the Prolog implementation by as much as a factor of 2 or 3. Moreover, this performance continues in the case of the $N$-queens problem, which is an even more realistic test case for the declarative set programming paradigm.

Naturally, in all cases, the CM-SAM implementation vastly outperforms the sequential SAM and Prolog versions. The more interesting observation is in the comparison of relative performance as the complexity (size column) scales up within each case. For example, for incrMap, evenFilter and sumFold, the run-time performance of CM-SAM within each case is invariant with problem complexity, while in both the Prolog and sequential SAM implementations the run-times scale up according to problem size. In the $N$-queens problem, the rate of increase in run-time with problem complexity for CM-SAM is significantly less than for the Prolog and sequential SAM implementations.

6. CONCLUSIONS

One paradoxical result of using SEL is that it can also result in better efficiency (e.g. through reduced recursion) even on a conventional sequential implementation[3]. This explains
why in the sumFold and \( N \)-queens cases of Table 1, the sequential SAM implementation outperforms SICStus, one of the best available Prologs, even before having any form of \textit{ad hoc} optimizations considered for the sequential SAM. This seems to support the original thesis that the set-oriented, data parallel approach to logic programming has some intrinsic advantages as a computational paradigm. The experiments presented here seem also to support the observation that some of this efficiency is carried over into the parallel implementation and multiplied.

The large difference in run-times between the sequential SAM implementation and the parallel implementation on the CM2 is quite predictable. However, for the incrMap, evenFilter and sumFold problems, the run-time performance of CM-SAM within each case is invariant with problem complexity, while in both the Prolog and sequential SAM implementations the run-times scale up according to problem size. For the \( N \)-queens problem, the magnitude of the difference between the sequential SAM and the CM-SAM is also quite pronounced: for the 4-queens the difference is 4:1 while for the 8-queens it is more than 12:1, which seems to indicate that the relative advantage of a data parallel approach on such a massively parallel architecture for such set-oriented problems grows with complexity.

Further discussion of the benefits of adapting the approach of parallel implementations of logic languages in terms of abstract entities as well as a discussion of further experiments on real parallel machines is given in [16].

This paper began by observing that logic languages are abstractly argued to be inherently conducive to parallel implementation, but insufficient practical evidence seemed to be available to substantiate the claim. The actual development of massively parallel architectures such as the Connection Machine have not been enough to prompt much experimentation in this field. This situation may be due to the practical problems of dealing with the process parallel view of implementing logic languages.

Data parallelism is posited as an underutilized forte of logic programming, and SEL is introduced as an example of a logic language which lends itself well to an efficient data parallel implementation. The data parallel approach effectively sidesteps many practical complications previously standing in the way of process parallel implementation of logic programming languages.

Thus, the road to a more global approach to parallelizing logic languages is opened by exploring data parallel approaches and their advantages. The implementation of SEL described here assumes an abstract machine which represents an intermediary structure between SEL and the target parallel architecture. Implementation of the subset abstract machine (SAM) on the Connection Machine is outlined, and some preliminary benchmarks are presented. The few experiments presented here indicate that potential performance benefits may be achieved from a data parallel approach on a massively parallel architecture.

This is just the beginning of the research. There are many open issues. For example, there are means for avoiding the quadratic duplicate check which should be compared (abstract analyzers for identifying cases when the duplicates will occur), and ways to limit the use of the Active Memory space should also be explored. Ultimately, the goal is to integrate process parallelism with data parallelism, for which a multiSIMD machine should be most appropriate. The opportunity to integrate set abstraction within a more standard language, such as Prolog, could also be an agenda item to address in the long run.
ACKNOWLEDGEMENTS

The authors wish to thank Giuseppe Marino for ideas contributed at an earlier stage in the writing of this paper. Also, thanks go to Mark Foy for reading the paper and making useful editing suggestions.

REFERENCES