The Design of an Abstract Machine for Subset Equational Languages

Giancarlo Succi & Giuseppe A. Marino

DIST-Università di Genova
via Opera Pia 11a
I-16145 Genova, Italia

Abstract

Many researchers have been trying to use the implicit parallelism of logic languages parallelizing the execution of independent clauses. However, this approach has the disadvantage of requiring a heavy overhead for processes scheduling and synchronizing, for data migration and for collecting the results. In this paper it is proposed a different approach, the data parallel one. The focus is on large collections of data and the core idea is to parallelize the execution of element-wise operations. The target language is SEL, a Subset Equational Language. An abstract machine for it, the SAM (Subset Abstract Machine), is outlined, which, under certain points of views, belongs to the WAM family. The data parallel structure of the SAM and of its instruction set is here explained and some examples of how it works are given. Eventually some conclusions are drawn and it is presented the plan for the future research.

1 Parallel Implementations of Logic Languages

One of the biggest appeals of logic programming languages is what is commonly referred to as “implicit Parallelism”; i.e., the parallelism a compiler can automatically and easily identify and exploit, since no explicit constraint on the execution order is posed in the abstract logic evaluation scheme. Several attempts have been made for using this property in order to design and implement a language which can fully exploit the (parallel) architecture of its target machine, possibly without resorting to ad-hoc constructs and/or annotations.

Various forms of parallelism have been evidenced, such as and-, or- and stream-parallelism (and, of course, their combination), but nevertheless the results that have been achieved are not as expected. Probably the initial goals were set too high, but anyway we claim that too much attention has been devoted to the most “expensive” form of parallelism, expensive in terms of time and space overhead required for communications, synchronizations and processes management.

On the other side, very little attention has been given other simple forms of intrinsic parallelism, which could have clearer and simpler definitions, and that can due to more straightforward implementations.

This paper analyzes the data parallel approach for this problem. Section 2 introduces the concept of data parallelism, section 3 gives a general overview of the language used
in this approach, section 4 describes the SAM, the abstract machine under development, aimed to exploit the data parallelism, section 5 presents the data parallel instructions and section 8 draws the conclusions together with some plans for future research.

2 Data Parallel Declarative Programming

We can divide the intrinsic parallelism of logic programs in two classes:

- process parallelism
- data parallelism.

Process parallelism is the form of parallelism which parallelizes the execution of independent parts of a program, usually with the following approach:

1. identify closures which can be executed independently;
2. decide whether to execute them or not;
3. schedule the ones we have decided to execute on the processors;
4. collect the results.

Using this classification, and-or, or-and streams parallelism fall in this class, despite at a first sight they seem to act quite differently one from another.

It is evident that this approach has some intrinsic limitations since it requires quite a lot of communications between processes for exchanging data and synchronizing, a lot of time for forking and joining processes and a lot of bookkeeping to have a consistent execution. Furthermore there is the undesirable problem of how to choose what processes should run on what processors.

A different approach can be devised and a good candidate for it seems data parallelism, which is almost unexplored within the field of logic programming. It can be described by the following approach:

1. identify the collections of objects globally handled by our program;
2. decide how to spawn them onto the available processors;
3. manipulate them applying, as much in parallel as possible:
   a. "element-wise" operators,
   b. filters,
   c. "folding" operators;
   in order to obtain either new collections or just scalar elements.

This approach seems quite appropriate for an implementation on both MIMD parallel architectures and the SIMD ones, like the Connection Machine and it overcomes most of the limitations of process parallelism.

This design requires a language suitable for the representation of collections of objects and an effort of programmers to adapt their mind to this new paradigm. The question is then which kind of languages and of collections we should use and how we should spawn the collections onto the available processors.

3 SEL and Sets

The target language for our data parallel approach is SEL, the Subset Equational Language developed by Jayaraman et al. [1N88] at UNC/Chapel Hill and at SUNY/Buffalo. This language handles sets in a clean, neat and simple way. Using sets as the core collection has also the advantage that lots of people have experienced from many different fields in representing problems as relations between sets.

A SEL program is a sequence of two kinds of assertions:

equational assertions of the kind \( f(\text{terms}) = \text{expression} \)

subset assertions of the kind \( f(\text{terms}) \)

The meaning of this assertions is:

equational assertion: the function \( f \) applied to the ground equal to the corresponding ground instances of expression.

subset assertion: the function \( f \) applied to the ground instances the corresponding ground instances of expression.

The language incorporates the collect-all assumption for subset assertions, which states that the result of a function application to ground terms is the union of all the subsets obtained by all the subset assertions matching the ground terms with all the possible matching. We do not go into details here, and for a complete description of the language [Succi] can be consulted.

Some examples of SEL programs can help understanding our approach. The first program we examine is aimed to compute the sets of the squares of a given set:

\[
\text{squareSet}(\{x\}) \rightarrow \{ \{x\} \}
\]

Here it is present a remarkable feature of SEL, i.e., the multiple matching: since no order is imposed over the elements of a set, a matching of the kind \( \{x\} \) produces the matching of \( x \) with all the elements of the argument set; therefore, by the collect all assumption, the result is the set containing the squares of all the elements.

A data parallel implementation on a SIMD architecture, for instance, can perform this operation in just one shot: if the argument set is already distributed among the processors what it is needed is just to ask each processor to square the element stored on it (and this can be done in parallel) plus some extra (constant time) bookkeeping.

In the same way it behaves the cartesian product of two sets:

\[
\text{cartProd}(\{x\}, \{y\}) \rightarrow \{ \{x, y\} \}
\]

Here we have two nested set mappings, but the general philosophy is the same, and so can be the implementation.

Also more complicate patterns can be handled in this way, like:

\[
\text{perms}() = \{()\}.
\]

\[
\text{perms}([x\downarrow t]) \rightarrow \text{distr}(x, \text{perms}(t)).
\]

\[
\text{str}(x, [t]) \rightarrow \{ [x, t] \}
\]
which determines all the possible permutations of the elements of a set. In this case
the computation proceeds first generating all the sets matching the pattern in linear
time (assuming to have enough available processors, otherwise we need some sort of
virtualization) and then applying to all the sets distr.
Filters can be implemented with this approach too:

\[
\text{filter}([x_1, \ldots]) \text{ contains } \text{if } p(x) \text{ then } \{x\} \text{ else } \emptyset
\]
The function filter selects the elements of its argument set that satisfy the predicate
\(p\); again we can have a data parallel implementation in just one shot, provided
we have enough processors.

4 The SAM

The implementation of \(\mathbb{SE}\) divided in two phases: NS:

- the developer of a target machine
- the implementation of the abstract machine on the real architecture.

The abstract machine is called SAM, Subset Abstract Machine. It belongs to the
WAM family since its general structure resembles quite a lot that of the WAM; however
it does not need full unification capabilities, therefore there is no need of the "trail"
and faster store and match instructions replace the unify ones. The SAM can be viewed
as a sister of the SEL-WAM [Naiss] from which it inherits most of the implementation
strategies, which are extended with environment trimming, table of constants and the
ability of handling function.

Figure 1 outlines the general structure of the SAM: in addition to the standard 4
components - heap, stack, push-down list and processor- there is the Active Memory,\(^1\)
- a memory whose cells both store data and perform computations. Its aim is to
- hold the sets so that data parallel operation can be executed on them.

Figures 2 and 3 detail the structure of the AM: it is a multidimensional array of
cells where each cell is composed by three elements: a processor, a set of registers
and memory. The memory is organized in two parts: a stack, for performing local
computations, and a region for keeping set elements; when there are some sets differing

\(^1\text{Form here it will be referred to as AM.}\)

\[\text{Figure 1: set2 is obtained mapping set1 through } f\]

The SAM takes advantage of three main situations to exploit the data parallelism
of problems, viz. when there are mappings of one set into another, when there are filters
applied to a set, and when there are foldings of a set in a single element. When there
is a map of one set into another, for instance in the set theoretic definition:

\[\text{set2} = \{ f(x) : x \in \text{set1} \}\]

it is possible to compute set2 in one shot applying \(f\) to all the elements of set1
parallel. Likewise if there is a filter like:

\[\text{remote2} = \{ x : x \in \text{remote1}, p(x) \}\]

remote2 can be computed in constant time. Foldings are definitions of the kind:

\[f(\{\}) = k,\]

\[f(x \cup t) = x, f(t)\].
Here it is possible to perform a tree-like computation in order to determine the result, as it is shown in figure 6, where z is applied to a and b, to c and d, to e and f, to g and h and to i and j at once, and then to each pair of results, and so on, until there is only one element left. Note that this last class of operations is not deterministic, since no order is imposed on the elements of sets. e.g., the (pseudo) function nonDet:

\[
\begin{align*}
\text{nonDet}() &= 0, \\
\text{nonDet}(\{x|t\}) &= \text{minus}(x, \text{nonDet}(t)).
\end{align*}
\]

applied to the set \{1,2,3\} can give 0, 2 or 4 as result, depending on which matching we choose. However, if the folding function is commutative and associative the result is the same no matter of the matching [Nis99], hence since plus is both associative and commutative, the result of applying det:

\[
\begin{align*}
\text{det}() &= 0, \\
\text{det}(\{x|t\}) &= \text{plus}(x, \text{det}(t)).
\end{align*}
\]

to \{1,2,3\} is always 6.

5 Data Parallel Instruction Set

The instruction set of the SAM has been designed along two main guidelines: obviously the first is exploiting the data parallel construct of the language and the second is trying to maintain the global design as simple as possible in order to ease the implementation of the abstract machine on different kinds of architectures. While the instructions devoted to equational assertions has changed only slightly from the one of the SEI-WAM [Jay91], the one concerning subset assertions has been entirely revised.

5.1 Instruction Set for Equational Assertions

As we said little has changed for equational assertions from the SEI-WAM; a minor change has been introduced to speed up the operation of choosing the (equational) clause to apply: it has been devised a kind of clause indexing strategy: based on the analysis of not only the structure of the first argument but also of all the others and therefore we designed the instruction:

\[
\text{switch.on.ground.} \text{expr} <\text{expr} > \text{var} \text{st} \text{set} \text{functor} \text{constant}
\]

which on the basis of the kind of <expr> jumps to the proper label or fails.

A remarkable modification has been performed for equational assertions dealing with sets, trying to identify and to take advantage of the implicit data parallelism. In order to clarify the analysis, assertions can be divided in two classes:

- those generating a set smaller

\[
f(\{x|t\}) = \{h(x, t) \mid m(x, t)\}
\]

- those producing a single element

\[
f(\{x|t\}) = h(p(x, t), m(x, t))
\]

Under certain circumstances simple analysis can produce an optimal both classes. Consider, for instance, when the function has the shape:

\[
f(\{x|t\}) = h(x) f(t)
\]

we can rewrite this pattern as

\[
f(\{x|t\}) \text{ contains } \{h(x)\}
\]

which can be efficiently implemented as it is explained in the subset assertion section of this paper, since it is a mapping between two sets through the function b.

On the other side when the latter is of the kind:

\[
f(\{x|t\}) = z, \\
f(\{x|t\}) = h(p(x), f(t))
\]

there is the opportunity for folding, as it was explained in section 3. For this purpose it has been designed the fold-like SAM instructions:

\[
\text{fold.} \text{f} \text{Zs Zs Zr h/3 f/2} \\
\text{fold Zs Zs Zr h/3}
\]

which perform the computation on the set Zs storing the result in Zr using as zero Zr; the difference between the two is that in fold f/2 is the identity function.

\[2\]Adopting the SAM convention of calling x1 the argument registers, xi the temporary ones and yi the permanent ones; furthermore when it is not known whether the registers are temporary or permanent they are called zi.
5.2 Instruction Set for Subset Assertions

The compilation of subset assertions for the SAM differs from the one targeted to the SEL-WAM since the SAM is aimed at set up a "nice" environment for "simple" sorting on (data) parallel machines.

The first difference is in union, fuse, and insert: the what they are quite different:

- $2a + 2b$ takes in

- $Za := 2a \cup 2b$

and performs the

- $Za \cup 2b Zf$ is like union apart from the:

- $Zf$ in true, i.e.

- if $Zf$ then $Za := 2a \cup 2b$

- insert $2a \cup 2b$ inserts the element $2b$ in the

- $Za := 2a \cup 2b$

insert $2a \cup 2b$: if insert into the an element is

- $Za := 2a \cup 2b$

$Za$ of $Zf$.

These four new instructions have a critical role for performing mappings and filter function: they can act in parallel on all the AM cells of a set.

Regarding pattern matching, two can be its purposes inside subset assertions:

(i) to reduce an element of a given set.

(ii) to iterate over the elements of a set.

It is possible to distinguish at compile time these two situations because in the first case we deal with a ground element, like in:

lookForS(\{x\}) = true

or an already matched variable, like:

$h(x, [x])$ contains $f(x)$.

while in the latter there is a free variable. For the first case we can use the instructions:

- match.set.element $Za \leftarrow 2b$
- match.set.elementRemaining $Za \leftarrow 2b$

which match the element $Za$ against the set $Zb$ storing in $Zb$ the boolean result of the matching and in $Za$ the remainder of the set (only for match.set.elementRemaining).

5.3 Mapping Instructions

Most of the operations which iterate generative new set whose elements are in the standard definition of map, mappings can be divided in three to perform the matching process (which is a multiple matching).

Constant space: this is the case of:

$$f([x]) \text{ contains } g(x)$$

in which we obviously use the machine

map.over $Za \leftarrow [Za \ U [Zb]$ end

end.map.over $Zi \leftarrow [Zi \ U [Zb]$ start

which has the following behavior:

- map.generate.copy $Za \leftarrow [Za \ U [Zb]$ end: map.generate.copy $Za \leftarrow [Za \ U [Zb]$ end: here $Za$ is the argument.set $\{x\}$.

- $Zi \leftarrow x$ and $Zc$ to $t$; $Za$ contains what is pointed by $Zi$. If the argument set is empty the execution jumps to the label end, else a copy of it is generated without an element and the execution goes to the next instruction.

Constant space: this is the case of:

$$f([x]) \text{ contains } g(x, y)$$

in which we obviously use the machine

map.generate.copy $Za \leftarrow [Za \ U [Zb]$ end

end.map.generate.copy $Za \leftarrow [Za \ U [Zb]$ start

which have the following behavior:

- map.generate.copy $Za \leftarrow [Za \ U [Zb]$ end: here $Za$ is the argument.set $\{x\}$.

- $Zi \leftarrow x$ and $Zc$ to $t$; $Za$ contains what is pointed by $Zi$. If the argument set is empty the execution jumps to the label end, else a copy of it is generated without an element and the execution goes to the next instruction.

- end.map.generate.copy $Za \leftarrow [Za \ U [Zb]$ end: if we have not yet completed the analysis of the argument set, $Za$ and $Zc$ are differently matched against $\{x\}$. $Za$ is properly updated and the execution jumps to the label start, else we go to the next instruction; $Za$ is added as parameter to speed up the creation of the new pattern.
Linear space: patterns of this latter case need only linear space for matching when the "remainder" of the set \( G \) is not used in the answer since in this situation we can destructively update it. The instructions are:

```plaintext
map_overriding_copy Z; Zm Zc end
end_map_overriding Z; Zm Zc start
```

with the following behavior:

- \( map_{\text{overriding}} \) copy \( Z; Zm Zc \) end is almost identical to \( map_{\text{generating}} \) copy;
- \( end_map_{\text{overriding}} \) start differs from \( end_map_{\text{generating}} \) only in the destructively modification of the set pointed by \( Zc \).

Simple abstract analysis can be used to determine when these complex instructions can be used.

A sample compilation chunk can help understanding this new design; consider the following code for the set of the squares of a given set:

```plaintext
squareSet[{x1}] contains [x*x]
```

the corresponding SAM instructions are:

```plaintext
squareSet/2:
allocate
get_set A1 Y1
get_variable A2 "?"
multiplication Y1 Y2 Y4
begin:
put.value Y4 A1
put.value Y4 A2
put_variable Y5 A3
call mult/3
insert Y2 Y5
end_map_overriding Y3 Y4 begin
and:
decalllocate
```

At the end of this discussion we would like to stress that the \( end\_map\) instructions do not suggest that we are having a sequential execution, nor do it the indices of the map. cases: the fact is that we are acting in data parallel, therefore everything between \( map\) and \( end\_map\) is executed in data parallel on the active memory.

### 5.4 Distributed Pattern Matching

The purpose of pattern matching is to identify portion(s) of a structure comparing it against a template. The WAM uses the PD-list to accomplish this task, however this approach is intrinsically sequential, since the list is a FIFO structure. Had this approach been taken also for the SAM, there would have been the need of sequentializing the computation each time a pattern matching would have been required on the elements of a set, consequently creating a critical bottleneck. A completely different mechanism has been therefore devised.

Whenever there is a need of performing a pattern matching on the elements of a set, a template of the pattern is created in the local stack of each AM cell of the set. This template contains a reference to a free register of the cell for each of its free variables. The operation of matching can then be performed locally in the cell operating on the local set element and on the local template and at the end of it either a local fail flag is set or the registers contains the desired values.

![Figure 7: Local stack during the distributed pattern matching of the example.](image)

The operations for creating the template are:

```plaintext
start_match 2a
store.temp_variable 2a
store.temp.value 2a
store.temp.function 1/n
store.temp.set 2a
store.temp.list
store.temp.constant c
store.temp.unref
```

start_match 2a places in the register 2a the address of the top of the stack, which is the place where the matching process will start. store.temp_variable 2a stores in the stack a reference to the register 2a, where the result of the pattern matching will be placed. store.temp.value 2a store in the stack a reference to the register 2a which was already referred by another store.temp_variable 2a in order to handle nested matching (the situation is analogous to the one of [get/put/unify].value of the WAM); store.temp.function f/n stores in the stack the function f together with its arity n, store.temp.set 2a places a reference to the register 2a which holds a pointer to a set, store.temp.list sets the template to wait for a list while store.temp.constant c for the constant c and finally store.temp.unref is used to handle the "don't care"...

Taking this approach the clause select.father can be compiled as:

```plaintext
elect.father/3:
get_set A1 X3
get_variable A2 X4
store.match X5
store.temp.function family/2
store.variable X6
```
Machine implementation has started and it is in the early stage the mentioned project for the process parallelism integration. Many are the open problems, such as the ones about the best object allocation scheme for sets and the ones about abstract analyzers.

Acknowledgments

The first author thanks Prof. J man, since most of the ideas of thi to his outstanding supervision NY/Buffalo.

References


