Data Parallelism in Logic Programming

Giancarlo Succi & Giuseppe 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 parallelism. 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 view, belongs to the WAM family. The data parallel structure of the SAM is here explained and some examples of how it works are given. Eventually it is explained the role of abstract analyzers in this framework 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 has 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, in section 6 some sample compilations for the SAM are presented, section 7 is devoted to some comments on abstract analysis for the SAM 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 decide to execute on the processors,
4. collect the results.

Using this classification, and, or- and stream-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 have 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 undecidable problem of how to choose what processes should run on what processors.

A different design 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:
   * "element-wise" operators,
   * filters,
   * "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 minds 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. [JN88] at UNC/Chapel Hill and at SUNY/Buffalo. This language handles sets in a clean, neat and simple way. Choosing sets as the core collection has also the advantage that lots of people have experience from many different fields in representing problems as relations between sets.

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

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

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

The meaning of this assertions is:

equational assertion: the function \( f \) applied to the ground instances of \( \text{terms} \) is equal to the corresponding ground instances of \( \text{expression} \); subset assertion: the function \( f \) applied to the ground instances of \( \text{terms} \) contains the corresponding ground instances of \( \text{expression} \).

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 [Suc91] 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.

\[
\text{squareSet}(\{x1, x2, \ldots, xn\}) \text{ contains } \{x1^2, x2^2, \ldots, xn^2\}.
\]

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 \( \{x1, x2, \ldots, xn\} \) 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:

\[ \text{cartProd}(\{x_1\},\{y_1\}) \text{ contains } \{x,y\}. \]

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:

\[ \text{perms}([]) = \{\}. \]
\[ \text{perms}([x_1]) \text{ contains } \text{distr}(x, \text{perms}(t)). \]
\[ \text{distr}(x, [t_1]) \text{ contains } \{x|t_2\}. \]

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:

\[ \text{filter}(\{x_1\}) \text{ contains if } p(x) \text{ then } [x] \text{ else } [\]. \]

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 that we have enough processors.

4 The SAM

We take the quite usual approach of dividing the implementation in two phases [Jay91]:

- the development of a compiler for the language targeted to an abstract machine,
- the implementation of the abstract machine to our real architecture.

The abstract machine we are developing is called SAM, Subset Abstract Machine. Its general philosophy comes from the WAM [AK90] and it is quite closely related to the SEL-WAM[Nai88]. We do not give a detailed description of the SAM here, our emphasis is on its data parallel structure; for a complete survey see [SM91b].

Like the SEL-WAM, the SAM treats an n arguments assertion like a \((n+1)\) arguments clause whose first \(n\) are input arguments and the \((n+1)\)-th is the output. Like the SEL-WAM, also, there is no need of unification capabilities, which has the consequences that:

- there is no need of "trail",
- faster store and match instructions replace the unify ones, since in the SEL framework it is decidable at compile time where we need to match and where to store.

Differently from the SEL-WAM, the SAM has the capability of handling functions, it uses a table of constants and it performs the environment trimming optimization. The usage of functions does not require any big complication neither in the original language -despite not being implemented, they were always present its formal description- nor in the compilation or in the execution. However, to simplify the compilation, we decided to require the user to write annotations of the kind:

\[ \text{functor f, g, h.} \]

specifying that \(f \), \(g\) and \(h\) are functions and not function, therefore they must not be reduced. The table of constants and environment trimming are aimed mostly to space optimization, nevertheless they allow also some time saving.

Quite new is the compilation of assertions: equational assertions are speeded up by means of an enhanced clause indexing strategy and the management of sets is entirely revised. The SAM includes a processing memory, that is a memory whose cells both store results and perform computations. Sets are stored here instead of on the heap and some "set-oriented" operations are executed virtually in parallel on it.

Dealing with sets, it is sometimes convenient to use basing techniques. When there are some sets differing one-another only for few elements, it may be useful to store in the processing memory a superset of all of them, the base, and to represent each of the original sets (called remote sets) with a bitvector -a 1 in its \(i\)-th position means that the \(i\)-th element of the base belongs to the remote set, and a 0 that it doesn't.

Figure 2 describes the processing memory: it can be viewed as an array of cells. The cells are outlined in figure 3. Each one contains a processor, a memory and some registers.
The local memory is divided in three parts: the first for storing the base element (in the example the number 45); the second for the bits of the remote sets (here the first remote set contains the number 45 while the second doesn’t). The last portion is devoted to a stack for local computations.

For this reason the adj-like instructions are replaced by map instructions, which generate a new set in terms of another one, and there are folding instructions for computing element oriented set properties and filtering instructions aimed to the definition of subsets of a given set.

We try to minimize the usage of the processing memory has to be parsimonious, since it is a scarce resource than “standard” memories. Moreover, architectural constraints may impose to allocate the processing memory cells at loading time. Hence abstract analyzers play a relevant role in the design of the SAM.

We can devise various implementations of the processing memory: presently it just coincides with the standard one; a single processor approach using hashing tables has been described in [Suc91] and an implementation on the Connection Machine is under development.

5 Data Parallel Instructions

As it is said before, ad hoc instructions have been designed to properly handle sets. They can be divided in three groups:

- mapping instructions,
- filtering instructions,
- folding instructions.

5.1 Mapping Instructions

A clause of the kind:

\[ \text{squareSet}([x]) \text{ contains } \{sqr(x)\} \]

Figures 3: Structure of a Cell

is quite common in SEL: it computes a new set as a mapping of an already existent set through a mapping function (here square). This situation is suited for a data parallel approach since all the mappings can be executed in (data) parallel over the elements of the set distributed on the active memory. The class of the map instructions is used here, which determine a new set in terms of an already existing one and a mapping function. The SAM code for this situation is:

```plaintext
squareSet/2:
  ... ... ...
  map_over <argument.set> <resulting.set>
  <compute square of the element>
  end_map_over <storing the result>
```

Figure 4 represents this situation: set2 is obtained mapping set1 through f. Note that if a potentially unlimited number of processors is available, this operation can be performed in constant time.

5.2 Filtering Instructions

As it was previously mentioned, certain classes of subset assertions can be defined as filters, e.g. they select which elements of a given set belong to another one. Their general form is:

\[ \text{filter([x]) contains if } p(x) \text{ then } [x] \text{ else } \{\}. \]

Here the resulting set contains those elements of the argument set satisfying the predicate p. The assertion `selectEven':

\[ \text{selectEven([x]) contains if even}(x) \text{ then } [x] \text{ else } \{\}. \]

is an operation of this kind. For this purpose the SAM uses the instructions of the class filter. They have the format:

\[ \text{selectEven/2} \]
filter_over <argument_set> <resulting_set>
<compute even on the element>
end.filter_over <storing the result if true>

\[ \begin{array}{c|c|c|c|c|}
| x | y | z | w | result \\\n\hline
| 1 | 1 | 0 | 1 | \text{remote1} \\\n| 0 | 1 | 0 | 0 | \text{remote2} \\\n\end{array} \]

Figure 5: remote2 is computed filtering remotel with p

Figure 5 evidences how this approach can be performed taking a data parallel approach. The predicate is computed over the elements of remotel; for those satisfying it (only b) a 1 is stored in result2 while for those not satisfying it (a and c) a 0 is stored.

5.3 Folding Instructions

The last group of operations targeted by this approach are those of the kind

\[
\begin{align*}
f([]) &= k, \\
f([x \mid t]) &= z(x, f(t)).
\end{align*}
\]

fold <argument set> <result> <zero> <folding function>

where <result> is the result of the computation (res in figure 6), <zero> is the value for the empty set (k in the example) and <folding function> is the function to be applied in parallel (here z/3).

Note that this class of operations is not deterministic, since no order is imposed on the sets, e.g., the (pseudo) function nonDet:

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

applied to the set \{1, 2, 3\} can give 0, 2 or 4 as result, depending on which matching we choose. However [MS89] demonstrated that if the folding function is commutative and associative the result is the same no matter of the matching; so since plus is both associative and commutative, the result of applying det:

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

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

6 Compilation Examples

Sample compilations can help understanding the design of the SAM; we present the SAM code for the squareSet program we explained in the previous section.

\[
squareSet/2:
\begin{align*}
\text{allocate} \\
\text{get_set} A1 Y1 \\
\text{get_variable} A2 Y2 \\
\text{map_over} Y1 Y3 Y4 \text{ end} \\
\text{begin} \\
\text{put_value} Y4 A1 \\
\text{put_value} Y4 A2 \\
\text{put_value} Y4 A3 \\
\text{call mult/3} \\
\text{end.map_over} Y2 Y3 Y4 A3 \text{ begin} \\
\text{end} \\
\text{deallocate}
\end{align*}
\]

Here we perform a mapping of the original set-pointed by Y1- on the new one-pointed by Y2. The instructions between map_over and end.map_over determine the operations that have to be done on all the elements of the set, which are pointed by Y3, and therefore these instructions can be executed in parallel on all processors of a parallel architecture.

A little bit more involved is the case for the clause intersect:
intersect({x1},{x1}) contains {x}.

which can be handled like filters, since the second matching poses a logic (and implicit) constraint on the fact that the element of the first set belongs to the result; the compilation of this assertion is therefore:

```
intersect/3:
get_set A1 X3
get_set A2 X4
get_variable A3 X5
filter_over X3 X5 X7 end
begin
match_set_element X7 X4 X5
end.filter_over X4 X6 X7 X8 begin
end.proceed
```

7 Abstract Analysis

Abstract analyzers play an important role in the process of building an efficient implementation of a logic language: their aim is to infer important properties of programs analyzing them without executing. Then ad hoc compilers can take advantage of these informations to produce a faster code. There is already a wide usage of abstract analyzers for computing the “mode” of a variable, for determining whether the same object can be shared or need to be copied, and so on. All these techniques can be extended to this framework. However there are more reasons for partial evaluators here.

It has already noted that the usage of the processing memory has to be minimized, since its availability is much lower than that of “standard” memories and that architectural constraints may impose to allocate the processing memory cells needed at loading time. Hence it is strictly necessary to design abstract analyzers to determine at compile time good approximations of the objects involved in the executions and of their sizes. In this design they are divided in two classes:

- **object size analyzers**, aimed to determine the sizes of the objects,
- **persistence analyzers**, targeted to compute the lifetime of objects.

A full description of them can be found in [SM91a].

8 Conclusions and Further Research

In this paper it is presented a new model for exploiting the intrinsic parallelism of logic programming, the data parallel one. A suitable language is overviewed and a suitable abstract machine is described quite in detail. Examples are given on how the data parallelism works on this machine and on its pseudo assembler.

At the present time we have almost completed an implementation of the SAM on a RISC Sun4 and we have started its Connection Machine porting which should be ready by October.

Obviously there are still lots of open questions about this project. It would be interesting to see how much we could integrate this two forms of parallelism: we could allow some very limited and conservative forms of process parallelism coupled with data parallelism which should be able to exploit a high degree of parallelism. The point is then what kind of architecture we should use for this approach.

9 Acknowledgments

The first author thanks prof. Jayaraman for being introduced by him to this field and for his precious comments.

References


