Exploiting the Data Parallelism of Subset Equational Languages

Giancarlo Succi

DST
Università di Genova
I-16145 Genova, Italy

Abstract

In this paper we propose an alternative approach to the parallel implementation of declarative languages, 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 view, 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.

1 Introduction: Models of Parallelism

We can divide the intrinsic parallelism of declarative programs into two classes:

* process parallelism,
* data parallelism.

Process parallelism is the form of parallelism which parallelizes the execution of independent clauses of a program. Using this classification, 

* and-parallelism fall in this class, despite at 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 communication 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 declarative 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 as possible:
   * element-wise operators,
   * filters,
   * folding operators;

in order to obtain either new collections or just scan 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 position is that which kinds of languages and of collections we should use and how we should spawn the collections onto the available processors.

2 SEL and Sets

The target language for our data parallel approach is SEL, the Subset Equational Language developed by Jayaraman et al. [JN85] 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 experienced from many different fields in representing problems as relations between sets. We do not go into details here, and for a complete description of the language [Succ77] 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:

\[ \text{sqrtSet}\{x,y\} \times \{x^2\} \]

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 \( \{x\} \) produces the matching of 1 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 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:

\[ \text{cartProd}\{x,y\} \times \{x,y\} \]

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

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

\[ \text{perm}(\{x\}) = \{\} \]

\[ \text{perm}(\{x,y\}) \times \text{distr}(x,\text{perm}(\{y\})) \]

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.

3 The Abstract Machine

The implementation of SEL is divided in two phases [JN86]:

* the development of a compiler targeted to an abstract 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 capability, therefore there is no need of the 'trivial' and faster store and match instructions replacing the unify ones. The SAM can be viewed as a member of the SEL-WAM [Nau88] from which it inherits most of the implementation strategies, which are extended with environment trimming, table of constants and the capability of handling functions.

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, 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.

Figure 2: The Active Memory

Figure 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, which are stored

Footnote 1: Form here will be referred to as AM.
4 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 architecture. While the instructions devoted to sequential execution has changed only slightly from the one of the SEL-WAM [27,9], the one concerning the sub-set assertions has been entirely revised.

Most of the operations which involve the elements of a set have the shape of generating a new set whose elements are functions of the elements of the original one: this is the standard definition of map-pings which was introduced in section 3. Those mappings can be divided in three categories, depending on the space they need to perform the matching process (which is a multiple matching).

Constant space: this is the case of patterns of the kind:

\[ f(x_1, x_2) \] contains \( g(x) \),

in which we obviously need only constant space since we just scan the set; for this case we use the machine instructions:

```
map_over 2a 2b 2c and
end_map_over 21 2a start
```

which have the following behavior:

- **map_over 2a 2b 2c end**: produces the (possibly parallel) analysis of the set pointed by 2b using 2a as index, storing in 2c the value pointed by 2b and jumping to end if the set is empty;
- **end_map_over 21 2a start**: increments the scanner 21 and updates 2a consistently; if the increment succeeds (i.e., we have not yet examined all the set) the execution jumps back to start else it goes to the next instruction.

Quadratic space: when we have patterns of the kind:

\[ f([x_1]) \] contains \( g(x, t) \),

in general we need quadratic space, since we need to build n copies of the set, being n the cardinality of the set. The virtual machine instructions for handling this situation are:

```
map_generating.copy 2a 21 2m 2c end
end_map_generating 2a 21 2m 2c start
```

with the same behavior as above.

Linear space: patterns of this latter case need only linear space for matching when the "remainder" of the set \( x \) is not used in the answer since in this situation we can constructively update it. The instructions are:

```
map_overriding.copy 2a 21 2m 2c end
end_map_overriding 2a 21 2m 2c start
```

again with the same behavior as above. Simple abstract analyzers can be used to determine when this coup of 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:

```
squareset([x_1]) contains [x_1^2]
```

the corresponding SAM instructions are:

```
squareset/2:
allocate
get_set A1 Y1
get_variable A2 Y2
map_over Y1 Y2 Y1 Y4 end
begin:
push_value Y4 A1
push_value Y4 A2
put_variable Y6 A3
call mult/3
insert Y2 Y6
end_map_over Y3 Y4 begin:
deallocate
```

5 Distributed Pattern Matching

The purpose of pattern matching is to identify positions of a structure companding 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 the 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 the local stack element and on the local template and at the end of it either a local flag is set or the register contains the desired values.

Figure 7: Local stack during the distributed pattern matching of the example.

The operations for creating the template are:

```plaintext
start.match 2a 
store.tempvariable 2a 
store.tempvalue 2a 
store.tempfunctor f/n 
store.templist 
store.tempconstant c 
store.tempref
```

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.tempvariable 2a stores in the stack a reference to the register 2a, where the result of the pattern matching will be stored. store.tempvalue 2a stores in the stack a reference to the register 2a which was already referred to by another store.tempvariable 2a in order to handle nested matching. the situation is analogous to the one of get/put/unify. value of the WAM; store.tempfunctor f/n stores in the stack the functor f together with his arity n. store.tempvariable 2a places a reference to the register 2a which holds a pointer to a set. store.tempconstant c for the constant c and finally store.tempref is used to handle the "don't care".

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

```plaintext
select.father/3
get.set3 1 23
get.value3 42 34
store.match 15
store.tempfunctor family/2
store.variable 16
store.tempvalue
map_over.matching 33 37 end begin
insert 24 36
end_and.map_over.matching 33 37 begin
and_proceed
```

The store.temp prepare the pattern to be matched against inside the matching storing the template in the stack; the store.tempvariable 16 stores in the stack a reference to the register 16 so that at the end of the matching process this register will contain the result. Figure 7 shows how the pattern is built on the local stack of an AM cell.

6 Optimizations

It has already been 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 work they are divided in two classes:

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

The first analysis that is performed concerns estimating the sizes of the objects involved in the program. The goal would be to express those sizes as functions of the sizes of the input. As it has been explained above, this information would be used to allocate properly the sets at compile time. However this problem is in general undecidable, therefore the interest is in a way of statically analyzing a program for determining approximations of the sizes of the objects involved in the program. This approach is based on the work of Debray, Lin and Heremengildo [DLH90] for relational Prolog-like language.

Persistency analysis are aimed to determine the lifetime of objects at compile time. Also this problem is undecidable, therefore again the interest is in approximations. These approximations must be sound, viz. there must not be the case of underestimating the life time. This means that the system must never fail in looking for allocated objects. Since the main purpose of these analyzers is to save active memory space, as it was explained before, the approach is highly "set oriented". Three kind of persistency analyses are described here:

1. shared bases analysis, that is which sets can be expressed as remote sets of a common base;
2. shared objects analysis, viz. which objects can be shared among different structures;
3. destructive update analysis, viz. when it is possible to destructively update a set instead of building a new one.

A detailed analysis of the abstract analyzers goes beyond the purposes of this paper; a full description of them can be found in [SM90].

7 Integration with Process Parallelism

The fact that this approach is based on data parallelism and not process parallelism does not mean that this paper claims that process parallelism shouldn't be exploited at all. It would be very interesting to try to couple some limited forms of process parallelism with data parallelism in the framework of SEL. Furthermore SEL seems to be a good candidate for such an integration since it seems quite simple to have a conservative or parallelism, parallelizing the execution of multiple subset assertions matching the same head. The design of the SAM does not require big changes to handle this extended framework: the communications and synchronizations can be confined in the top level and in the union instructions. There is a starting project in this field and there is the hope of getting some early figures by the beginning of 1992.

8 Conclusion

In this paper it is described the instruction set of the abstract machine we are developing for the data parallel execution of declarative languages, the SAM. Presently it has been completed the implementation of the SAM on a Risc Sun machine and it is benchmarked. The very first figures seem promising. In the meanwhile the Connection 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. Jayaraman, since most of the ideas of this paper are due to his outstanding supervision at SUNY/Buffalo.

References


