A Transputer Implementation of SEL

G. Succo, G. Marino, R. Cantuccio, M. Facciolo, Vernazza

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

Abstract

The main goal of this project is the implementation of a declarative language, SEL, on a MIMD parallel architecture. SEL is the acronymous of Subset Equational Language. It has been designed to handle sets in a clear and simple way, since, in many different fields, the problems are represented as relations between sets.

The target machine is a Transputer based multiprocessor system, which has been chosen to exploit the process parallelism intrinsic of the language. Indeed, SEL allows to perform computations building a single set independently and concurrently, making different subsets. Since each of these subsets is a necessary part of the set under construction, the process parallelism which arises from subset parallel computations is strictly conservative. This ensures that no useless and wasteful computation is tried. In addition, SEL is cheaper than Prolog, because its operational semantics is based on set matching and internal rewriting of expressions rather than term unification. It seems possible that its implementation on a parallel architecture can take advantage from these language characteristics, giving good results in terms of performance for general applications based on set handling.

1 Parallelism in SEL

SEL has been developed by Jayaraman et al. [JN88] at UNC/Chapel Hill and at SUNY/Buffalo. It is composed by two kinds of assertions:

- equational assertions like \( f(\text{terms}) = \text{expr} \)
- subset assertions like \( f(\text{terms}) \text{ contains } \text{expr} \)

While the first is well-known, the second is less trivial. The use of sets inside subset assertions is the core of the parallel implementation. The following example shows how sets can be defined by subset assertions, all matching the same head:

\[
\text{Set}_{\text{Inc}}(\{x1\}) \text{ contains } \{x+1\};
\text{Set}_{\text{Inc}}(\{x2\}) \text{ contains } \{x+2\};
\text{Set}_{\text{Inc}}(\{x3\}) \text{ contains } \{x+3\}.
\]

If a set-valued expression \( f(\text{terms}) \) has the form:

\[
f(\text{terms}) \text{ contains } S1
f(\text{terms}) \text{ contains } S2
\ldots
f(\text{terms}) \text{ contains } Sn
\]

and there are no other known subsets, in the program, for \( f(\text{terms}) \), the collect-all assumption allows to infer that the set \( f(\text{terms}) \) is given by the union of all subsets \( S_1, \ldots, S_n \). Here is a remarkable feature of SEL: multiple matching. Since no order is imposed over the elements of a set, a matching of the kind:

\[
\{x1\}
\]

produces the matching of \( x \) with all the elements of the argument set. By the collect-all assumption, the result is the set containing the elements of all elements. Multiple matches effectively serve to iterate over the elements of sets, thus permitting many useful set operations to be stated concisely.

Besides this, the impossibility-as-failure assumption allows to infer that

\[
f(\text{terms}) = f
\]

if all reductions from \( f(\text{terms}) \) terminate in a failure.

These two assumptions are consequences of the closed world one. They underly the meaning of subset-equational programs and suggest the computational independence of the subsets \( S_1, \ldots, S_n \). So, they may be computed by different processes, where each process operates on the same world that defines \( f(\text{terms}) \). Then, to obtain the whole \( f(\text{terms}) \) set, a main process has to do the union of all subsets computed by other processes.

The parallel computation of subsets \( S_1, \ldots, S_n \) is a form of OR-process parallelism, since it may be considered as the computation of multiple independent solutions to a sub-query or query like:

\[
? f(\text{terms}).
\]

In the case of the above example, this may be:

\[
? \text{Set}_{\text{Inc}}(\{4,5,6,7\})
\]

Note that because one-way matching is used instead of unification, arguments allowed to functions are only ground terms, and then, the query also uses such a kind of terms.
2 The Subset Abstract Machine (SAM)

The implementation of SEL has been realized developing a compiler targeted
to an abstract machine and then implementing the abstract machine on a real
architecture [Na88]. This approach is quite common for the implementation
of declarative languages and it ensures a partial independence from any target
architecture, besides many advantages in writing the compiler. So, a good
starting point is represented by the UNIX sequential implementation of SAM
[SM91] [GC92], which has been extended to exploit process parallelism, as
detailed in the next paragraphs.

The general structure of SEL abstract machine and its instruction set [SM-
C91] resemble those of the Prolog WAM [AK00], from many points of view.
Their main differences are mostly due to the different operational semantics of
the two languages. Indeed, SEL uses matching instead of the full unification
of Prolog. In addition, some optimizations allow time and memory savings.

As shown in figure 1, the structure of SAM is characterized by the following
parts:

- the main processor and a series of registers;
- the heap, a global stack area for data permanent storage, where tags
  are used for fast typing, because it contains a variety of data types;
- the stack, where a number of registers and memory addresses are
  saved to keep track of the computation;
- the push-down list, used to perform pattern matching instead of
  unification, like in the WAM (where it is also called Unification Area);
- the code area, to store the program and query code;
- additional tables, for constants and symbols storage.

But the most relevant aspect of the language implementation, for this paper,
is how multiple subset assertions are translated into SAM code.

In the Prolog WAM, multiple clauses, matching the same head, are trans-
lated with the instructions: try_me_else, retry_me_else and trust_me_else fail. In
the same manner, in SAM, multiple subset assertions are translated by mean-
s of try_sub and instructions, while after each subset computation, proceed_sub
instructions invoke the program continuation.

So, multiple subset assertions like the following:

\[
f(\text{terms}) \text{ contains } S_1
\]
\[
f(\text{terms}) \text{ contains } S_2
\]
\[
f(\text{terms}) \text{ contains } S_{n-1}
\]
\[
f(\text{terms}) \text{ contains } S_n
\]

are translated with a sequence of instructions like:

\[
\text{try\_sub} \text{ and } \text{label } 1 >
\]
\[
\text{code for subset } S_1 \text{ calculus } >
\]
\[
\text{proceed\_sub}
\]
\[
\text{label } 1 >
\]
\[
\text{try\_sub} \text{ and } \text{label } 2 >
\]
\[
\text{code for subset } S_2 \text{ calculus } >
\]
\[
\text{proceed\_sub}
\]
\[
\text{label } n >
\]
\[
\text{try\_sub} \text{ and } 1 >
\]
\[
\text{code for subset } S_n \text{ calculus } >
\]
\[
\text{proceed\_sub}
\]
\[
\]

Then, try_sub and instructions are used to introduce the computation of each
subset, which is part of the set f(terms). Their semantics states that the SAM
Program Counter receives the next label address, after every subset calculus,
when the proceed_sub instruction is executed. If this is the last subset, the
program execution proceeds with further computations, if there are any.

In case of single subset assertions like:
in the try_sub and proceed_sub instructions, their role is fundamental for the management of process parallelism, as follows.

3 SEL implementation on a MIMD architecture

A MIMD architecture seems well suited for a language implementation which exploits the process parallelism. The processes can be assigned to the available processor elements. In addition, in a distributed memory model, there are no bottlenecks due to structure sharing. So, no limit to the project scalability is imposed by bus saturation, when the number of processes becomes great.

On the other hand, the cost of a distributed computational environment is mainly due to the process scheduling, synchronization and communications. These activities, which are fundamental for the process management, may affect the computation, giving bad results. Besides this, when the program nature doesn't allow to take advantage from any form of parallelism, the language implementation must maintain an execution speed comparable with that of the sequential implementation.

So, the approach that seems more convenient is to assign most of the process management operations to server processes. The base idea is that a main process can perform the program computation, while a number of auxiliary processes only execute the parts of the program which are introduced by try_sub and instructions. If their results are available when the main process executes those instructions, it gets the results and jumps to the next try_sub and instruction of the program. Otherwise, the main process executes that code as in the sequential computational model.

![Figure 3: The main actors of SEL parallel implementation.](image)

The main actors of SEL parallel implementation.

The functions of process scheduling and balance of work are delegated to a
Distributor process. It also sends to the auxiliary processes the necessary code and data for their computations. Analogously, the functions of collecting results and storing them in the main process memory are assigned to a Collector process.

These two servers share some of the main process memory regions because they make an heavy use of their information. Code Area and Heap are examples of these shared memory regions. So, an expensive data replication is avoided.

Figure 3 shows the processes which are used to exploit the language parallelism and the flow of information exchanged.

When the main process executes the first try.sub and instruction of a sequence, it makes a request of parallel computation. Then, Distributor sends to the auxiliary processes an image of the actual content of registers, a copy of data structures pointed by them, and an initial Program Counter value. These processes start the computation from it. When their last proceed.sub instruction is executed, the elements computed are sent to Collector. This process picks up these subsets and store them in an Extended-Heap. It writes an identifier of each subset in a Result Area. Now, they are available to the main process.

The main process always executes the part of program introduced by the first try.sub and instruction, while the others are assigned to auxiliary processes, following a bottom-up order. This ensures the maximum distance from the main process computation and those performed by auxiliary processes.

When in the main process the next try.sub and instructions are executed, it makes a search in the Result Area. Both the Result Area and the Extended-Heap are shared by the main process and the Collector.

Figure 4 shows the most relevant data regions and their relations with the processes discussed.

4 Transputer and S.E.I.

The implementation on a massively parallel MIMD architecture also allows to reserve one processor element for each auxiliary process. The Transputer architecture seems well suited for the described competitive approach, for the simplicity of its general structure, its reconfigurability and the low cost of communications. An implementation on a Meiko Computing Surface has been realized using the CSTools facilities.

The mechanism which underlies all CSTools communications services is the CSN - the Computing Surface Network. Simple CSN interface routines provide general purpose functionality.

Figure 5 shows the physical layout of the parallel SAM implementation and its relations with the CSN.

![Figure 5: Physical layout of the parallel SAM implementation and CSN.](image)

Because of the different kind of the host and auxiliary processors (i.e. SPARC and Transputers), a preliminary conversion to a common data representation form is necessary. So, the Sun XDR conversion utilities are used for both transmitted and received messages. These conversions are executed by the two interface processes, Distributor and Collector, as well as by the auxiliary processes.
These data are sufficient to define its computational status to Distributor, which already shares with the main process the Code Area and the Additional Tables.

Figure 6 shows the content of the Data Exchange Area after an hypothetical first multiple try, sub, and execution. The main process writes such a kind of entries by handling the Data Exchange Area as a circular buffer. Distributor gets their contents and, after, invalidates them.

An additional control is necessary to ensure that Distributor doesn’t get any partially overwritten information.

Figure 7 details the content of headers: they univocally identify each entry and then any group of multiple try, sub, and instructions, by means of the pair of values PTC and PC: respectively the Parallel Try, sub, and Counter and their initial Program Counter value.

When the main process executes the first try, sub, and instruction of a group, it increments the PTC. So, if a computation is repeated many times, all these are univocally distinguished by its value. This happens, for example, with the execution of the following SEL assertion:

\[
\text{succ}(x) = [ f(x+1), f(x+2), f(x+3) ]
\]

where the function \( f() \) may be:

Since any multiple subset assertion is invoked by the instruction:

\[
\text{call <Assertion Name> / <Arity> (Num Arg is Arity)}
\]

the contents of the Argument Registers used by the call instruction always completely define the necessary information for the execution. Next instructions will probably modify these values: this is the why of their storage into the Data Exchange Area.

6 Process synchronization

The status of three shared flags is particularly important for the control of all system life. These flags are:

- the Help flag, which is shared by both Distributor and the main process. This last sets it when a new entry has been stored into the Data Exchange Area.

- the Work Reset flag, which is shared by the three host processes and which is set by the main process, when it completes the computation of the current multiple subset assertion.
- the End flag, which is shared by all the SAM host processes and which is set by the main process when it completes the query execution.

Figure 8 details their logical timings and represents the normal sequence of events which controls the various computational phases of the system.

```
Help flag

Work Read flag

End flag

End of program ...
```

Figure 8: The logical timings of the control flags.

An implementation based on an exclusive use of software interrupts is actually matter of study.

7 Collecting the results

Some details about this computational phase are very interesting because of the intrinsic difficulties to manage a lot of unexpected remote communication requests. In fact, the Collector doesn't know who will send it available results and the order in which these will come. A direct polling approach seems not to be the best one, since to establish a direct communication with each of the possible sources (i.e. auxiliary processes) is a very expensive operation and, probably, it may be an useless one.

The engaged auxiliary processes perform their computation and only at last they need to send results to the Collector. They have neither time nor necessity of any communication with it, before. So, the Collector doesn't know the status of them until the completion of their work has come.

Among the possible solutions to this problem, two seem to be, at the moment, well suited:

1. A group of light-weight collector processes are allocated, each one to serve only those communications that will come from a specific auxiliary process; so, many communication requests can be served virtually in parallel. But all these light-weight collector processes share the same host resources and they need to write on the same Result Area and Extended Heap, that is more critical.

2. The Collector allocates a number of different ports. They are as many as the auxiliary processes. Then, the Collector alternately waits for any asynchronous message (i.e. communication request) from them. It continuously tests the ports status, to detect as soon as possible the presence of messages. Then it knows who has sent it (by the number of the port used) and establishes a synchronous communication with that remote source, to serve its call. There are no possibilities of receiving results asynchronously, since there are no limits to their size. So, new messages might overwrite those already received but not yet served.

The second solution seems more promising and so the actual parallel SAM implementation adopts it.

Figure 9 shows the scheme of the port array which is used by Collector to accepts asynchronous messages from any remote process. The source process must know the name of the port and specifies it to the CSN.

```
Figure 9: The Collector ports array.
```

There is a character buffer associated with each port. Its size is very small because it is not relevant what it contains but the action of receiving something into it. Other buffers are dinamically allocated to support the next synchronous transmissions of results.
8 Critical issues

Designing parallel implementations for logic programming languages is nowadays greatly the art of minimizing the cost of process communications, synchronization and scheduling. Since these are the most critical operations needed for the soundness of the whole system, their implementation must agree to the following requirements:

- the amount of time due to data exchange between processes has to be only a minimum percentage of the execution time in any application.

- if all processes have the same importance, no one of them must be stopped to wait for any other. That is, they have to work the most independently is possible, avoiding delays due to close interactions.

- recomputations must be avoided. So, the assignment of work to the available processors must be very accurate.

- the politics of work scheduling must ensure an equilibrium load balancing, a fast allocation of the system resources and that obsolete computations neither are kept in consideration nor take place.

- the communications must be reliable, so that there is no need of any complex communication protocol, heavy error checking and message re-transmissions too.

The Transputer SAM implementation meets almost all the above criteria, while, on the other hand, their complete satisfaction strongly depends by the nature of the application.

The first two requirements manifest a fundamental character for any parallel implementation: the locality of data and then, of their computation. This is an aspect which underlies the possibility of making advantageous computations in a distributed environment, within the same application. In general, the cost of message passing between processes takes $O(n)$, both in time and space, where $n$ represents the amount of data transmitted. This may be intrinsically very hard to handle.

In the SAM parallel implementation there are three phases which involve indispensable message passing. This happens at first from Distributor to the auxiliary processes and at last from them to the Collector. The three phases are:

- the Boot phase, that is when Distributor reads the Code Area and Additional Tables to send their content to the necessary auxiliary processes, establishing direct synchronous communications with them (note that if a multiple subset assertion has only, e.g., three branches, there is no need of using a greater number of auxiliary processes).

- the Start phase, that is when Distributor reads the content of the Data Exchange Area to send it to the used auxiliary processes.

- the Collect phase, that is when Collector tests the contents of its ports to detect the presence of communication requests and then, it establishes a synchronous communication with the source, to get available results.

All these moments are extremely significant for the system life, but their cost increases with the amount of data involved. There is no way to put some limit to it, because all the information exchanged are essentials.

On the other hand, the scheduling algorithm assigns independent computations to the available auxiliary processes and no explicit recomputation takes place. At last, if the main process completes the execution of the multiple subset assertion, it sets the Work Reset flag. Distributor tests it before any allocation of new computations; so, no obsolete assertions are assigned.

Besides this, the CSN utilities ensure reliable communications and the choice of making the major part of them synchronously reinforces their reliability.

Since both Distributor and Collector are processes which serve the main process and because it doesn’t wait for them, the use of synchronous communications doesn’t directly influence its performances. At most, if these operations cause a significative delay there aren’t ready results to help its work.

![Figure 10: The application area of Transputer SRL implementation.](image-url)