Abstract Machine for Parallel Rewriting : Design and Implementation of Parallel TRAM

Masaru Kondo

School of Information Science,
Japan Advanced Institute of Science and Technology

February 14, 1997

Keywords: parallel term rewriting, abstract machines, TRAM, parallel E-strategy, multiprocessors, CafeOBJ.

1 Introduction

Algebraic specification languages such as CafeOBJ have been widely attracting attention since they have clear semantics and the ability to write down lucid, non-ambiguous, and non-inconsistent specifications of software systems. Many algebraic specification languages use (order-sorted conditional) term rewriting systems as a general computaional model. This makes it possible to verify and prove various properties of algebraic specifications, and to execute the specifications as programs by using rewrite engines. For implementing the rewrite engines efficiently on conventional computers, abstract machines for term rewriting systems are designed. TRAM is one of these abstract machines.

On the other hand, with the rapid and steady advance of hardware technology, small-sized and low-priced multiprocessor computers have been developed. These machines are the next generation of uniprocessor workstations and personal computers and will undoubtedly become standard for the future computers.

Judging from these facts, it would be very important to develop a method of implementing rewrite engines efficiently on multiprocessors. In this study, we have parallelized TRAM so that the rewrite engines can be implemented efficiently on the multiprocessors. In addition, we have implemented the parallelized TRAM (Parallel TRAM) on the multiprocessor, and have assessed the performance of the implementation by executing several benchmark programs on Parallel TRAM.

2 Parallel E-strategy

Parallel E-strategy is an extension of OBJ's E-strategy, which allows us to specify the order of reductions explicitly including concurrent reductions. It is easy to adopt the strategy that reduces all of arguments automatically in parallel. But, we can not expect high performance because excessive number of parallel processes might be created in this strategy. If we wish to execute concurrent reductions efficiently, we should control the number of parallelisms in some way. Parallel E-strategy is able to control this problem. For example, `f/4 : (1 {2 3 4} 0)` specifies a strategy for an operator f of arity 4. Non-zero natural number n denotes the reduction of the nth argument of a term whose top operator is f, and zero denotes the reduction of the term. If we wish to reduce arguments in parallel, they are enclosed with brace brackets "`{`" and "`}`". The other elements in the strategy are reduced sequentialy from left. The above strategy indicates that firstly the 1st argument of the term is reduced, secondly the 2nd to 4th arguments are reduced in parallel, and lastly the term itself is reduced.

3 Parallel TRAM

Parallel TRAM carries out reductions based on Parallel E-strategy. The structure of Parallel TRAM is fundamentally the same as TRAM. However, the interpreters for abstract instractions and the mutable memory spaces whose contents may be changed during rewritings are replicated and given to each processor. (The marked units and spaces are replicated.)

- **Processing Units**

  Rule compiler : The rule compiler encodes the LHSs into a discrimination net and compiles the RHSs into pairs of matching program templates and strategy list templates.

  Term compiler : The term compiler compiles input terms into pairs of the matching programs and the strategy lists.

  Interpreter* : The interpreter interprets the matching program according to the strategy list.

- **Memory Spaces**

  DNET : This is the space for the discrimination net that is the tree structure for efficient search of matching rules.

  CODE* : This is the space for the matching programs that execute pattern-matching between a term and the discrimination net. Since matching programs are equivalent to the terms in Parallel TRAM, we can seek the applicable rules only executing the matching programs corresponding to the term.

SL* : This is the space for the strategy list that indicates the order of traversing the term. This strategy list is generated from the subject term according to each operation's strategy. Parallel TRAM interpreter rewrites the subject term according to the strategy list.

CR : This is the space for the matching program templates and the strategy list templates.

STACK* : This space is working place for pattern-matching.

VAR* : This space contains variable bindings for pattern-matching.

In Parallel TRAM, the rules and the terms are compiled by the main processor sequentially.

We mainly add following four abstract instractions to be able to reduce terms in parallel. If we specify the concurrent reduction, the labels for calling these instractions are put in the strategy list appropriately. The meanings of these instractions are as follows:

FORK : This instraction creates a new process for reducing a term independently and allocates it on an idle processor. If there are no such idle processors at the moment, this instraction is ignored and the processor calling this instraction executes the new process by itself.

JOIN : This instraction is used for synchronization. The processors calling this instraction suspend their executions until all of their child processes (that are allocated on the other processors by FORK) are finished. During they suspend, their states are changed into idle and they may execute some reductions from the other processors.

EXIT : This instraction tells the parent processor that the reduction allocated by FORK is finished. This instraction is allocated on child processors with processes for parallel reductions by FORK.

SLEEP : This instraction is called when a processor finishes all of its allocated reductions and changes its state into idle.

For example, given `f(A,B,C)` as an input term, and the strategy of `f` is specified as `f/3:({1 2 3} 0)` , then the generated strategy-list is as follow:

```
[<FORK>, <matching program for "A">, <FORK>, <matching program for "B">,
 <matching program for "C">, <JOIN>, <matching program for "f">,
 <return result>]
```

This strategy list are allocated on the main processor's SL, Parallel TRAM performs the reduction in the follwing way:

1. The main processor begins to reduce the input term by picking up an element in strategy list and executing it. At first, sub-processors 1 and 2 are suspended because of SLEEP.

```
SL main = [<FORK>, <m.p. for "A">, ···  , <return result>]
SL sub1 = [<SLEEP>]
SL sub2 = [<SLEEP>]
```

2. FORK creates the new process for the 1st argument `A` and allocates it on an idle processor (sub-processor 1) with EXIT. Similarly, the new process for the 2nd argument `B` and EXIT are allocated on sub-processor 2. The main processor picks up the next element (term `C`) in its strategy list.

```
SL main = [<m.p. for "C">, <JOIN>, <m.p. for "f">, <return result>]
SL sub1 = [<m.p. for "A">, <EXIT>, <SLEEP>]
SL sub2 = [<m.p. for "B">, <EXIT>, <SLEEP>]
```

3. Each processor executes each reduction in parallel.

4. The main processor may pause at JOIN until the sub-processors 1 and 2 finish their reductions. The two processors finish their reductions, then each EXIT tells the parent processor (the main processor) its termination.

5. The main processor executes the reduction for the whole term and returns the result. The sub-processor 1 and 2 are suspended again by SLEEP.

4 Conclusion

We considered the mechanism for reducing terms in parallel and designed Parallel TRAM which is an abstract machine for parallel term rewriting. We implemented this abstract machine on the multiprocessor workstation LUNA-88K2 that carries four MC88100 processors using C language and evaluated its performance. We got following results:

- We can restrain excessive number of the concurrent processes using Parallel E-strategy, consequently Parallel TRAM efficiently performs the reductions in parallel.
- We parallelize TRAM by integrating a concurrent mechanism in the strategy list, without losing the advantages of TRAM.
- Parallel TRAM was found to be 1.67 to 1.98 times faster than TRAM after executing some benchmark programs on the two versions of TRAM.