Transputer and occam Research: New Directions
J. Kerridge, Ed.
IOS Press, 1993

OR-Parallel Execution of Prolog on a Transputer-Based System *

Lourdes Araujo
lurdes@dia.ucm.es

Jose J. Ruz
jjruz@dia.ucm.es

Dpto. Informática y Automática
Universidad Complutense de Madrid
Madrid 28040, Spain

Abstract

In this paper we present a distributed multiprocessor model for OR parallel execution of Prolog. It is based on multiple sequential Prolog engines connected by real or virtual channels. In order to minimize the messages traffic, each processor has its own environment. Two approaches to construct the parent processor environment on an idle processor have been presented: stack copying and recomputation. It also presents and compares performance results of both approaches, implemented on a transputer-based system.

1 Introduction

Prolog is an appropriate tool for exploiting the implicit parallelism of applications due to its declarative semantics. A Prolog program presents two main kinds of parallelism: AND parallelism and OR parallelism. The former is the result of selecting more than one atom of a clause body to be solved simultaneously. The latter tries to unify a selected atom with the head of several clauses. For example, consider the program:

```
mother(jane,john).
mother(jane,alice).
father(james,john).
father(james,peter).
brother(X,Y) :- mother(Z,X),mother(Z,Y).
brother(X,Y) :- father(Z,X),father(Z,Y).
relative(X,Y) :- mother(X,Y).
relative(X,Y) :- father(X,Y).
relative(X,Y) :- brother(X,Y).
```

*Supported by the Prontic project TIC92-0793-C02-01

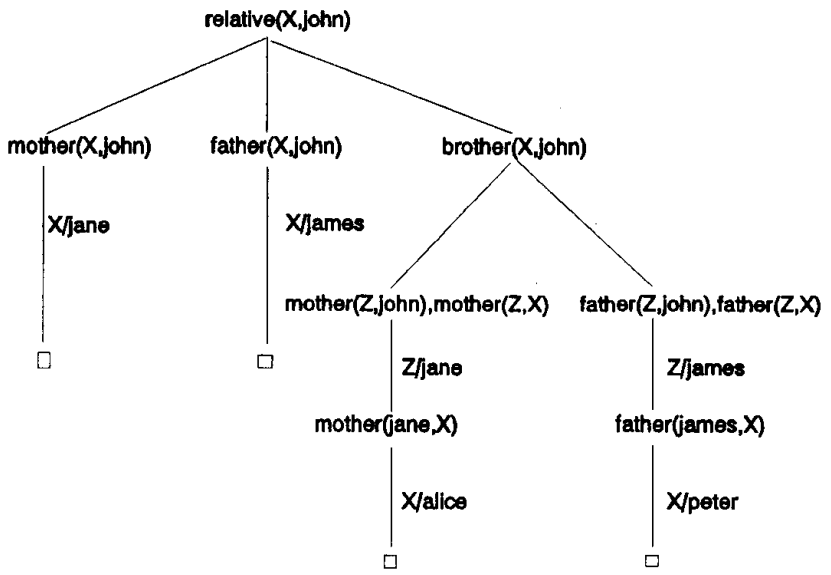

Figure 1: Search tree

To know John's relatives we ask:

```
:- relative(X,john).
```

For this program and this goal, the search tree, that represents the execution, has the form shown in figure 1.

If the goal is executed sequentially, the different solutions are only reached by backtracking. If OR parallelism is exploited every branch is explored in parallel. In general, the variables that are shared by several branches can be bound to different values, for example X is bound to jane, alice, james and peter.

The Warren Abstract Machine (WAM), developed by D. Warren [4], has become the standard sequential implementation technique for Prolog. It is an efficient execution model and compilation techniques that lead to one of the most efficient implementation of Prolog today. This model has incorporated several optimizations that increase notably its performance and can be kept in a multisequential parallel system.

To implement OR parallelism it is necessary to represent different bindings of the same variable corresponding to different branches of the search tree. A number of models have been proposed to deal with this issue: the SRI model [5] keeps the multiple variable binding in a binding array, the Argonne [3] [6] model uses a hash array, etc.

We present an OR parallel implementation of Prolog on a distributed memory system. This kind of systems has a good scalability since information exchanges are made by means of messages sent by real or virtual communication channels and no bandwidth saturation appears as number of processors in the system increases. The system consists of a pool of processors connected through an interconnection network. Some of them support the control functions to perform the distribution of pending goals among idle processors. The remaining processors work like machines able to execute Prolog

programs and exploit OR parallelism. The execution model is an extension of the WAM that keeps its optimizations.

The Prolog programs are compiled to an intermediate code where the parallelism is annotated. These annotations can be done by the compiler or by the user. The efficient exploitation of OR parallelism is constrained to a restriction: only the alternative clauses with enough granularity must be annotated.

The rest of the paper proceed as follow. In section 2 we shortly describe the standard WAM in order to create the frame of reference and be able to show the relative cost of the following modifications. Sections 3 discusses two approaches, stack coping and recomputation, to exploit OR parallelism on a multisequential system. Section 4 deals with the architecture of the abstract machine to exploit OR parallelism by both approaches. Section 5 is dedicated to present some implementation details and finally the results and conclusions are discusse.

2 The sequential execution model

The WAM consists of several memory zones and an instructions set to perform the unification and to consider automatically the different alternatives of a procedure with multiple clauses. The WAM memory zones (figure 2) include a code zone, which contains the program in intermediate code, and three data stacks:

- the *control* Stack, containing two kinds of structures: *choice points*, to consider automatically the clauses of the procedures, and *environments* to store the binding of the variables
- the *heap* stack containing the structured data
- the *trail* stack containing the data to restore if a fail occurs

A *choice point* stores part of the WAM state upon entry to a procedure, that is, holds the goal arguments and several control registers. A pointer to the next clause in the procedure is also kept.

There are also a number of argument registers, $(r_1, ..., r_n)$, used to pass call parameters and other registers used to control the memory zones. One of these registers is the instruction counter IC that points at the currently executed instruction.

The WAM instructions set can be classified into *get* and *unify* instructions (to perform unification), *put* instructions (responsible for loading argument registers with the parameters of the next goal), *procedural* instructions (call, alloc dealloc, proceed) to control transfer and environment allocation associated with procedure calling, and the *procedure control* instructions (tryMeElse, retryMeElse, trutMeElseFail, ...), to manage the non determinism of the program and link together the different clauses that make up a procedure. The compiling code corresponding to each clause of a multiple clauses procedure, is preceded by *try* instructions: *tryMeElse* for the first one, *trusMeElseFail* for the last one and *retryMeElse* for the others.

The WAM code corresponding to the goal *relative(X,john)* is:

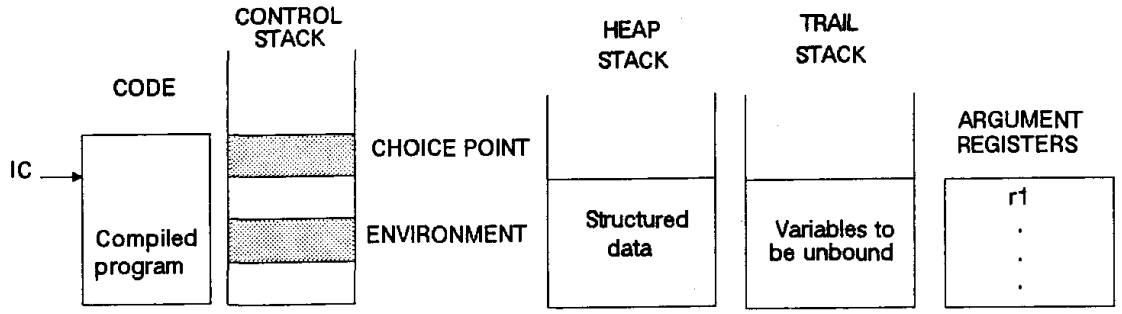

Figure 2: WAM data areas

```
                putVar r1, 1
                putCon r2, john
                call relative/2_0:, 1
```

The *put* instructions place the X and *john* parameters in the argument registers before calling to the *relative* procedure. The code corresponding to the procedure *relative* is :

```
relative/2_0:   tryMeElse 2, relative/2_1:
                alloc
                getVar r1, 1
                getVar r2, 2
                putVal r1, 1
                putVal r2, 2
                call mother/2_0:, 2
                dealloc
                proceed
relative/2_1:   retryMeElse relative/2_2:
                alloc
                getVar r1, 1
                getVar r2, 2
                putVal r1, 1
                putVal r2, 2
                call father/2_0:, 2
                dealloc
                proceed
relative/2_2:   trustMeElseFail
                alloc
                getVar r1, 1
                getVar r2, 2
                putVal r1, 1
                putVal r2, 2
                call brother/2_0:, 2
                dealloc
                proceed
```

The first clause in the procedure begins with a *tryMeElse* instruction, indicating that if a fail occurs, the next clause to try is *relative/2_1.* Then space is allocated and unification is performed by means of the *get* instructions. If the unification success, the parameters for the call to the *mother* procedure are place in the argument registers by the *put* instructions and the *call* is executed. The same stands for the next two clauses in the procedure.

3 Exploiting OR Parallelism

Because of our distributed architecture, we have designed the model to exploit OR parallelism in order to minimize the information interchanges, that is, each processor has its own environment and works independently. Therefore, to share a job it is necessary to construct the parent processor state in the new processor. Two approaches have been tried to get a computation state: stack copying and recomputing. The first one reconstructs the state making an explicit copy, and the second computes the initial goal again. When the state has been constructed, the processor works following the sequential model and no multiple variable bindings appear since each alternative is explored in a different environment.

3.1 The stack copying approach

This technique has been successfully implemented on a shared memory system [1]. In our system, the stack information is sent by a message. When an instruction marking the parallelism are reached, the address of the alternative clause is written down. While no request is received from idle processors, the pending alternatives are tried by backtracking. When a request is received, an explicit copy of the stacks, as they were when the parallel alternative appeared, is sent to the requesting processor. Pending alternatives do not care for the parent processor since they have been sent to the offspring one. A processor receiving a new job, places it properly in its memory and explores the pending alternatives.

3.2 The recomputation approach

In order to build the father state following this approach, the processor recomputes the initial goal without backtracking, that is, following the success path got from the parent processor. Therefore, the success path must be recorded by each processor when exploring the search tree, in order to send it to another processor, as it is depicted in figure 3.

The success path, (C2, C5, C10), specifies the current branch being processed. Since the search tree is explored in depth-first and left-to-right manner, the figure indicates that the C4 and C9 alternatives have been already explored and the C11, C3, C6, C7 and C8 alternatives have not been tried yet. If OR parallelism is exploited in this moment, the sent success path would be: C2, C5, C11, that is, the success path with the last alternative changed, in order to explore a new branch of the search tree.

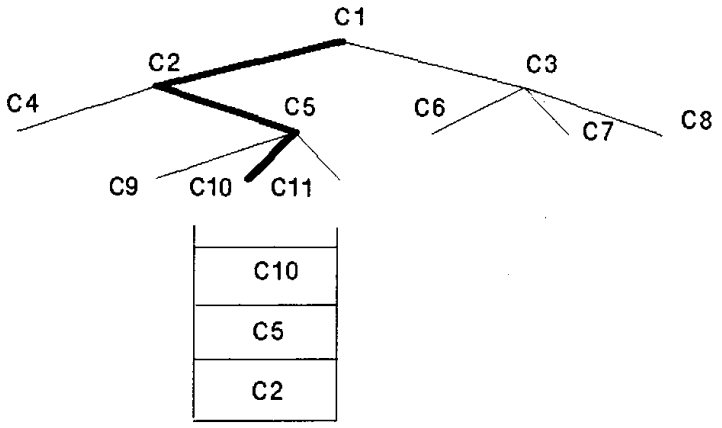

Figure 3: Success path

4 The Abstract Machine for OR Parallel Execution

The system consists of a pull of processors connected hierarchically (figure 4). To minimize the traffic in the network we have chosen a hierarchic control [2] to distribute the pending goals among the idle processors. There is a number of special processors, *controllers*, for this function. Each of them is assigned to a *workers* group, having information about idle processors and pending work.

Each worker supports an extended WAM, capable of exploiting OR parallelism. The workers can exploit parallelism supporting one of the described techniques to construct the parent state. In the next sections, we will describe our design for both techniques in terms of how it differs from the WAM.

For both techniques, the parallelism is marked on the program by means of the instructions:

- *try_par*
- *retry_par*

These instructions take the place of *TryMeElse* and *RetryMeElse* instructions respectively and nothing else in the program structure is changed. When one of them is reached, in addition to perform the actions corresponding to the instruction, a new pending job is annotated and a warning is sent to the controller. In this way, if parallelism is exploited in the *relative* procedure presented in the program of section 1, the WAM code would be:

```
relative/2_0:   try_par 2, relative/2_1:
                alloc
                getVar r1, 1
                    .
                    .
                    .
```

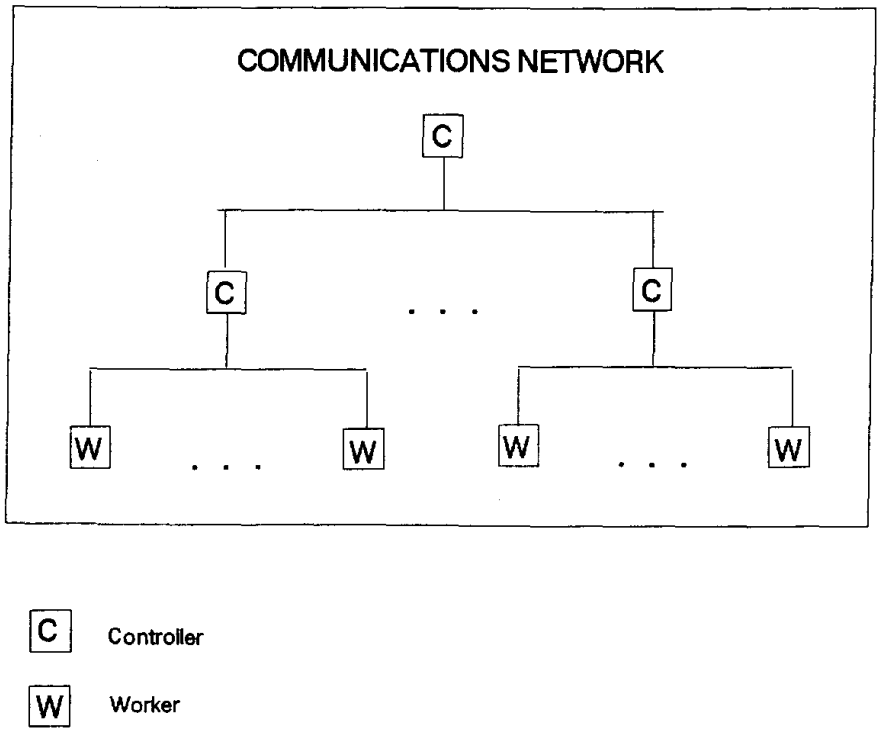

Figure 4: System organization

4.1 Copying Implementation

The implementation using the copying approach, does not need any change in the WAM architecture. We only have to append the data structures to partition the pending work and to support the communications. Every processor in the system must have identical memory address space in order to have coherency in the copied information. When a processor with pending job receives a request, sends a copy of its memory areas (figure 5) and marks the pending alternatives that have been sent to avoid performing backtracking over those alternatives that are going to be explored in other processor. To do this, a new register is introduced, the *back_point* register, that stores the first belonging to the processor alternative. When a new job is received, it is placed properly in the memory and the processor explores the following pending alternative.

The extension of the WAM to perform this approach is the following:

- Addition of the instructions

  try_par
  retry_par

  for alternatives to be exploited in parallel.

- Addition of the *Back_point* register that points to the first pending alternative that can be exploited in the processor

- Modification of the *fail* procedure to consult the back_point register when backtracking is performed

- Addition of the pending alternatives table and counter

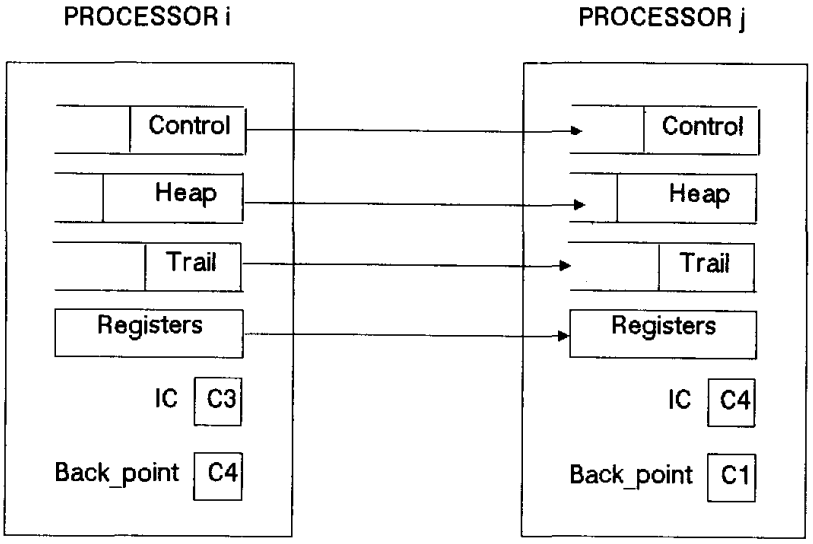

Figure 5: The stack copying approach

4.2 Recomputation Implementation

For the recomputation approach some new data structures are need to store the *success path*. To write down the success path, the *success stack* and the *success register* pointing to its top, are introduced. A new field is also added to the choice point containing the success register value in order to retrieve the success stack automatically during backtracking. When a processor is requested for a job, the processor replies sending the success path. When an idle processor receives the success path it becomes "*guided*", changing a switch GUIDED/FREE. In GUIDED mode the processor reexecutes the initial goal without backtracking, that is, following the addresses in the success path. When recomputation is completed the switch is changed to "*free*" and the machine operates in FREE mode, that is, creating choice points and writing down the success path again. For a program like this:

```
G:   :- p.

P1:  p :- q, r, s.
P2:  p :- t, v.

Q1:  q :- ...
Q2:  q :- ...
Q3:  q :- ...

R1:  r :- ...
R2:  r :- ...
```

We can find the machine in the state depicted in figure 6. The stacks content indicates that the first clause of the P procedure, the last one of the Q procedure (so the associated choice point has disappeared) and the first one of the R procedure are

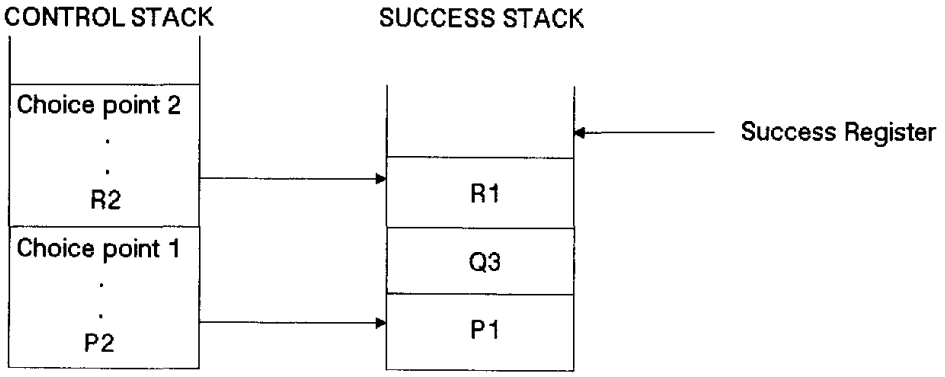

Figure 6: Success path computation

being explored. If a request is received in this time, the processor would send the following success path: P1,Q3,R2, and the choice point 2 would be updated with the alternative R3, to indicate that R2 is not a pending alternative any more.

The extension of the WAM to perform this approach is the following:

- Addition of the instructions
  try_par
  retry_par
  for the alternatives to be exploited in parallel.
- Addition of the *success stack* and the *success register* to contain the success path
- Addition of a new field in the choice point to contain the success register value
- Addition of the pending alternatives table and counter
- Modification of the instructions
  TryMeElse
  RetryMeElse
  TrustMeElseFail
  to write down and update the success path

A possible optimization is to take advantage of the state common part of two processors sharing a job, since only the different part need to be copied or recomputed. Because our system has distributed memory, knowing the common part of two processors can represent a high cost in communications. We reduce the recomputation time comparing the received success path and the previous success path and avoiding the recomputation of the common part. The controller takes into account this optimization, ordering to share work to processors that have share it previously, since they must have some common part.

5 Split strategies

The search tree distribution between workers sharing a pending job can be guided by different strategies. All of them must guarantee that no part of the search tree is ignored. Some of the possible strategies are:

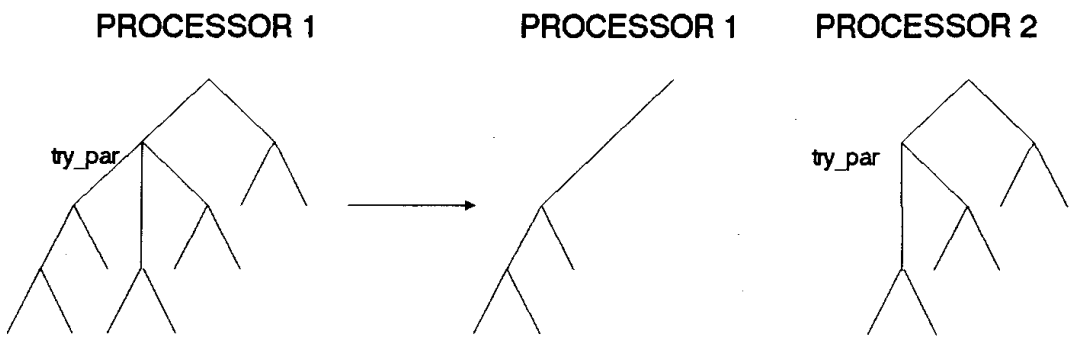

Figure 7: Split strategy 1

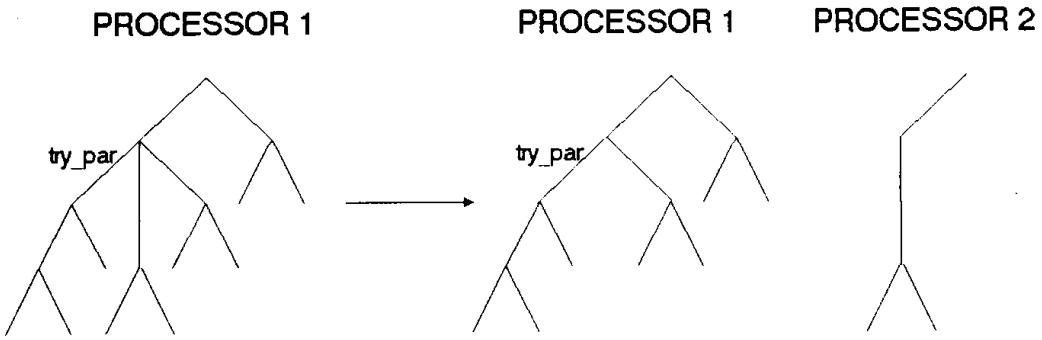

Figure 8: Split strategy 2

1. The parent processor gives all alternatives except one (figure 7)
2. The parent processor keeps all alternatives except one(figure 8)
3. The alternatives are split between the parent and the offspring processors in a balanced way

So far, we have considered the two first ones alternatives, since the last one requires grouping parallel alternatives and therefore, to change the program structure. The first one seems to be better since the only modification need to erase the pending alternatives that have been sent to other processor, is to update the back_point register with the address where the next pending alternative will be created. However, for the recomputing approach the second strategy could be more indicated since choice points does not need to be created during recomputation because an only alternative is sent. Therefore, both strategies have been tried in both approaches resulting that there is not any appreciable difference for the copying approach with any strategy whereas the recomputing approach works quite better with the second one.

6 Implementation on transputers

Our current implementation has been made in Parallel ANSI C with five T800 transputers. For more transputers, we will incorporate a new process to provide virtual links between any pair of transputers. Nevertheless this question will be solved with the appearance of the T9000 transputer that incorporates automatic routing of messages.

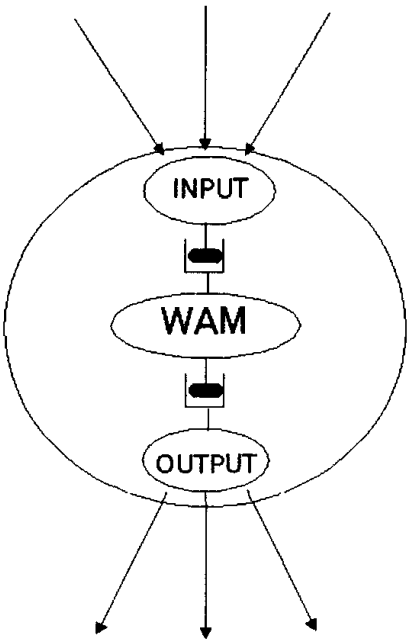

Figure 9: Processes scheme

In each processor the computation and communication functions have been split. There are three processes controlling the input, output and computation respectively as it is depicted in the figure 9.

The input process is suspended until a message arrives by any of the input channel to the processor. The messages are stored in a global memory area (int_message) from that are taken to be analyzed. The access to this area is controlled by a semaphore (int_sema).

```
{{{  InputProcess }}}
static void InputProcess (Process *InputP, Channel *Input[])
{
  signal(SIGUSR1, SIG_IGN);
  while (1)
  {
    Index = ProcAltList(Input);
    SemWait(int_sema);
    message_reception(Input[Index], int_message);
    SemSignal(int_sema);
  }
}
```

The process begins with a *signal* instruction to ignore the signal SIGUSR1 and avoid to go through the default signal handling system. The process consists of an infinite loop of messages reading. The *ProcAltList* instruction suspends the process until a message is received by any of its channels. When it happens, an index into the channel array is returned for the ready channel. The semaphore to have access to the memory area for input, is acquired with the *SemWait* instruction, and released with *SemSignal*.

The process dedicated to interpret the WAM code executes instructions until being warned of an arriving message.

```
{{{  WamProcess  }}}
static void WamProcess (Process *WamP)
{
  signal(SIGUSR1, SIG_IGN);

  while(1)
  {
    if thereis_message
      treat_message();

    if(state == ACTIVE)
      execute_instruction(opcode);
  }
}
```

After the protection of the SIGUSR1 signal, the process performs a loop handling of messages and WAM instructions execution. If it is need to send a message in an instruction execution, the signal SIGUSR1 is activated by means of the *raise* instruction.

The output process is asleep until a software interruption (signal) is raised if a message need to be sent.

```
{{{  OutputProcess  }}}
static void OutputProcess (Process *OutputP, Channel *Output[])
{
  signal(SIGUSR1, sig_cap);
  while(1)
  {
    ProcReschedule();
    SemWait(sal_sema);
    sent_message(out_message);
    SemSignal(sal_sema);
  }
}
sig_cap()
{
  signal(SIGUSR1, SIG_IGN);
}
```

The process gets ready to receive the SIGUSR1 signal. The *ProcReschedule* instruction places the process at the end of the active processes queue and only will become active after receiving the SIGUSR1 signal.

Program	sequential	OR parallelism by copying	OR parallelism by recomp.
qsort	14.9	15.9	16.0
matriz	28.4	29.4	29.5
merge	80.9	83.9	84.4
reverse	263.1	273.9	275.0

Table 1

Queens n.	sequential	OR parallelism by coping	OR parallelism by recomp.
4	165	108	109
5	105	173	146
6	1711	403	410
7	441	485	505
8	12265	2775	2780
9	5190	2540	2549
10	16867	16824	16840

Table 2

7 Results

We have investigated speedups obtained on both OR parallel machines using stacks copying and recomputation techniques. Some sequential programs have been tried in order to evaluate the overhead due to the parallel mechanism. Results are shown in table 1. The experiments demostrate that the overhead when a sequential program is run over our parallel machines is less than 1%. The comparison between the measurements in both parallel machines shown that the overhead due to write down the success path (it is the only difference for sequential programs) is less than 1%.

Another observation is the important overhead due to the information interchange with the controller. In our first implementation, each worker sent a message every time it encounters an OR parallel clause, to notify the pending work. The results (time in ms) obtained for the first solution of the queen problem benchmark (how to place N queen in a NxN chess board) with different values of N, are shown in table 2.

It may be observed from the table that the performance has not been improved in all the cases: only if the first solution is not the solution computed by the original processor. It is also shown that times are slightly better in the copying approach. These results are improved if a worker only can emit a warning of pending work at each time. It is say, until the pending work disappears, the new pending jobs are not notified. Results are shown in table 3.

Despite this constraint means a delay in the work interchange, the results have improved because of the time saved in the communications of the messages to the controller.

Queens n.	sequential	OR parallelism by coping	OR parallelism by recomp.
4	165	84	81
5	104	118	117
6	1711	397	395
7	441	453	452
8	12265	2766	2762
9	5190	1906	1916
10	16867	16786	16770

Table 3

queens n.	sequential	OR parallelism by copying	OR parallelism by recomp.
4	486	110	124
5	4274	1391	1401
6	8576	3468	5346
7	58665	17889	18763
8	248614	75148	75879

Table 4

The table 4 shows the results of running the programs finding all the solutions, in which case the improvement of the parallelism exploitation is more important.

These results show significant speedups for all the programs. Though the achieved speedup using stacks copying and recomputation is quite similar, it is slightly better for the copying approach. This is because using recomputation approach, the offspring processor spends some time constructing the state after receiving the success path, while in the copying approach this processor begins to work immediately after receiving the stack copy. It is possible to argue that the amount of information to send in the copying approach is clearly bigger that in the recomputation approach. But, it has been found through measurements that the overhead of sending several little messages is greater than the overhead due to send a big message.

8 Conclusions

We have presented two approaches to exploit OR parallelism of Prolog programs on a distributed memory system: one based on copying and the other on recomputation. The extension of the sequential execution model, WAM, have been presented for both approaches. The performance results of both methods have been compared, resulting a significant speedup. To validate these results, it is necessary to extend the system to a larger number of processors, where the messages have to cross a number of processors before arriving to its destination. If they are confirmed, a system will be implemented

combining both techniques to rebuilt the state, along with exploitation of independent AND parallelism. When independent AND parallelism is exploited following the model presented in [2], it is necessary to store the parent processor state in order to be able to transmit it when OR parallelism appears. Since the recomputation approach allows a compact representation of the state, it can be used to exploit OR parallelism after AND parallelism, while the copying approach is used to exploit pure OR parallelism.

Referencias

[1] Khayri A. M. Ali and Roland Karlsson. The Muse Approach to Or-Parallel Prolog. International Journal of Parallel Programming Vol. 19 No. 2 April 1990.

[2] Araujo, L. and Ruz, J.J. Combining Restricted AND and OR Parallelism on a distributed Memory System. Technical Report DIA 92/12. Computer Science Department. Complutense University of Madrid (1992).

[3] K. Shen and D. H. D. Warren. A simulation study of the Argonne model for Or-parallel execution of Prolog. SLP, 1987.

[4] Warren, D.H.D., An Abstract Prolog Instruction Set. Technical Note 309, SRI International, (1983).

[5] Warren. D.H.D., Or-Parallel Execution Models of Prolog. DATSOFT'87.

[6] Westphal, H, Robert, P, Chassin, J, Syre, J. The PEPsys model: Combining Backtracking, AND- and OR-parallelism. SLP 1987.

Transputer and occam Research: New Directions

edited by

Jon Kerridge
(University of Sheffield)

WoTUG-16

Proceedings of the 16th World occam* and Transputer User Group Technical Meeting
28th-31st March 1993
Sheffield, UK

IOS Press
1993
Amsterdam • Oxford • Washington, DC • Tokyo

ISBN 90 5199 121 5
ISSN 0925-4986

Publisher:

IOS Press
Van Diemenstraat 94
1013 CN Amsterdam
Netherlands

Sole distributor in the UK and Ireland:

IOS Press/Lavis Marketing
73 Lime Walk
Headington
Oxford OX3 7AD
England

Distributor in the USA and Canada:

IOS Press, Inc.
P.O. Box 10558
Burke, VA 22009-0558
USA

Distributor in Japan:

Kaigai Publications, Ltd.
21 Kanda Tsukasa-Cho 2-Chome
Chiyoda-Ku
Tokyo 101
Japan

Printed in the United Kingdom by Bell and Bain Ltd., Glasgow