OR-Parallel Execution of Prolog on a Multi-Sequential Machine

by

Khayri A M Ali

SICS R86006B
Research Report
ISSN 0283-3638

OR-Parallel Execution of Prolog on a Multi-Sequential Machine

Khayri A M Ali
Logic Programming Systems
SICS
P.O. Box 1263
S-163 13 Spånga, Sweden

Revised Version
November 1986

"Accepted for Publication in IJPP"

ABSTRACT

Based on extending the sequential execution model of Prolog to include parallel execution, we present a method for OR-parallel execution of Prolog on a multiprocessor system. The method reduces the overhead incurred by parallel processing. It allows many processing elements (PEs) to process simultaneously a common branch of a search tree, and each of these PEs creates its local environment and selects a subtree for processing without communication. The run-time overhead is small: simple and efficient operations for selecting the proper subtree. Communication is necessary only when some PEs have exhausted their search spaces and there are others still searching for solutions. The method is able to utilise most of the technology devised for sequential implementation of Prolog. It is optimised for an architecture which supports broadcast copying.

Keywords: Broadcast copying, logic programming, OR-parallelism, parallel Prolog machine, distributed split algorithms.

1. Introduction

OR-parallelism in logic programming attempts to achieve increased speed by allowing many possible solutions to a goal to be tried in parallel. It is one of the different sources of parallelism in logic programs [1].

We conduct research on fast parallel execution of Prolog on medium size multiprocessor systems. We are now concentrating on OR-parallel execution. We have selected the approach of extending the sequential execution model of Prolog to include OR-parallel execution [2, 3]. This approach is selected for two main reasons. The first one is that the other OR-parallel execution models [4, 5, 6, 7], which are based on the global address space, require high run-time overhead in maintenance of bindings, variable access, load distribution, and in process scheduling, specially when implementing such models on non-shared memory multiprocessor architectures. The second reason is that we are able to use most of the technology devised for the sequential implementation of Prolog.

The main problem of the approaches based on extending the sequential execution model of Prolog to include OR-parallel processing is the high overhead of copying environments from one PE to another in order to split a job [2, 3]. In such approaches, each PE has a complete physical copy of the binding environment. We have already proposed a broadcast architecture that alleviates this problem [8, 9]. It has a special network that allows broadcasting write operations and copying an environment from one PE to a number of PEs in parallel. Broadcast of write operations allows several copies of an active environment to exist in different memories. This reduces the required copying because copies of active environments are available to idle processors when they are allocated a job, which in turn reduces the time required to split a job between PEs.

In this paper we present another method to reduce copying of an environment and the required time to split a job. The method requires less capacity and complexity of the network. The network allows any PE to copy a part of the environment to a number of PEs in parallel.

2. Basic Idea

The basic idea of our method described here is that instead of having only one PE that processes the top-level query (main goal), all PEs in the system will simultaneously process the same query. At each choice point (alternative clauses whose functor and arity match the current goal), which can be processed by more than one PE, each PE selects one or more branches (alternative clauses) for processing without communication with the

other PEs. The above step is repeated until each PE works alone on a separate subtree. Each PE searches its subtree by normal depth-first search with backtracking. Only when a number of PEs terminate (exhaust their search spaces) and there are other PEs still searching for solutions, one busy PE is selected and its environment is copied to all the idle PEs. These PEs select again subtrees for processing without communication, exactly as before.

In the rest of the paper we are going to answer to the following questions among others:

- What architecture do support this method?
- How does each PE select a subtree without communication?
- How is it guaranteed that no branch is ignored in any split operation?
- How each solution of the problem is reported only once.
- How the sequential-OR is treated.
- How Cuts, Bagof, and Negation as failure are implemented.
- How does this solution compare with the previous ones?

3. System Architecture

Figure 1 shows a system organisation of a multi-sequential machine that supports the method. The system consists of the following components:

1- n Processing Elements (PEs)
Each PE has a local memory containing a copy of the program code and an area for the dynamic information needed by a Prolog system.

2- Manager Processor (MP)
It keeps the PEs' status information. For example, which PE is busy and which is not. It selects one of the busy PEs for copying when the number of idle PEs reaches a specific limit.

3- Broadcast Copying Network (BCN)
It allows each PE to copy parts of its environment into a number of PEs' memories in parallel.

4- Interprocessor Network (IPN)
It allows each PE to communicate with all PEs and with MP.

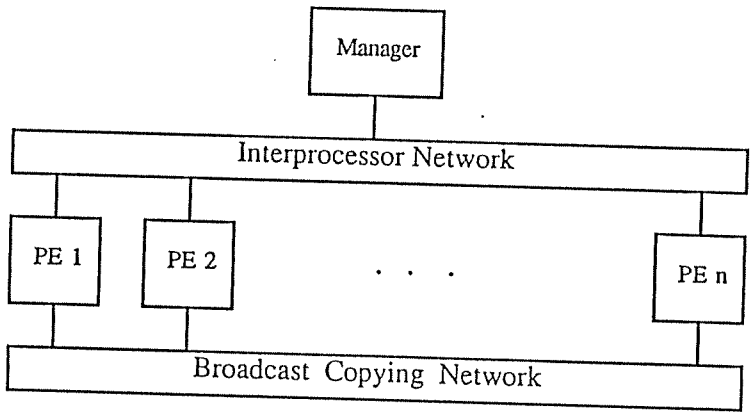


Figure 1: System Configuration of a Multi-Sequential Machine.

4. Execution Model

In this section an overview of the parallel execution model is presented for OR-parallel execution of pure Prolog on the architecture described above. Each PE maintains a run-time environment exactly the same as the run-time environment of Warren's virtual machine [10] - a few number of local stacks. The model is outlined as follows.

- When a user query is read, all PEs in the system start processing the query in parallel.

- Each PE creates its binding environment in its local memory without communication with other PEs during unifications.

- Whenever a PE comes to a choice point that is processed by more than one PE, it takes one or more branches according to a specific split scheme.

- When a PE comes to a situation at which the PE works alone on a part of the tree (subtree), it does not perform further split until a number of PEs become idle. The PE searches its subtree by depth-first with normal backtracking.

- When a PE terminates (exhausts its search space), it reports itself to MP as an idle PE.

- When the number of idle PEs becomes larger than a certain value k, one of the busy PEs is interrupted and (a part of) its current environment is copied to all the idle PEs in parallel via BCN. If the interrupted PE has a job that can be split, the job is partitioned into parts

and each PE takes one part. If the busy PE has not yet created a job that can be split, all these PEs work in parallel on the same branch until the next choice point. When any of these PEs encounters a choice point, it will perform splitting according to the split scheme.

- When more than one PE have got a solution of the same leaf, only one of them will report it.

5. Split Strategies

Partitioning of the search tree into parts and assigning these parts to the PEs is guided by an algorithm that satisfies a specific strategy. Some such strategies and their respective algorithms are described below. Any algorithm must guarantee that no part of the search tree is ignored in any split operation. Some of the possible strategies are:

1. *Balanced Strategy* in which the branches at each choice point are almost evenly assigned to the PEs processing that choice point. As an example, given a search tree with the top-level choice point of 3 branches and 7 PEs processing the top-level query in parallel. One possible assignment of branches to these PEs is as follows: the first branch to 2 PEs, the second branch to 2 PEs, and the third branch to 3 PEs.

2. *Right-Biased Strategy* in which branches, at each choice point, are assigned to the available PEs as follows. From left to right, one branch is assigned to only one PE. When the number of PEs is greater than the number of branches, the right most branch is assigned to the remaining PEs. When the number of PEs is less than or equal to the number of branches, the right most branches are assigned to one PE. The assignment of the 3 branches to the 7 PEs in the previous example is as follows: the first branch to one PE, the second to one PE, and the third to 5 PEs.

3. *Left-Biased Strategy* is a mirror of the right-biased strategy.

6. An algorithm for the Right-Biased Strategy

The right-biased strategy, perhaps, is suitable for search trees that have long branches at the right edge. The following algorithm satisfies the properties of this strategy as specified above. It is based on a simple naming scheme. The naming scheme guarantees that each PE knows locally (without any communication), how many PEs are processing the same subtree, and what unique identification it has. With these two types of information alone we can implement many split strategies.

When the system starts up or when one PE copies its environment into a number of PEs' memories, each PE gets two values: *vn and np*. The system manager or the sender PE can distribute such values. The vn is a virtual name of the PE, and np is the number of PEs that process the same subtree in parallel. Each PE uses the local values of vn and np to select a subtree without any communication. The values of vn and np for each PE are updated according to the following scheme:

- When m PEs are processing in parallel a search tree starting from a common root, each PE gets a unique value of vn ranging from 1 to m, and all PEs get m as the value of np, i.e. vn = i, and np = m, where $1 \leq i \leq m$.

- When any PE of these m comes to a choice point of say b branches, one of the following three situations may occur:

(A) $b > m$

Assume that branches at each node are numbered from left to right starting by one. In this case, each PE takes the branch with the number equal to vn and the PE that has vn = m takes the remaining right most branches. The splitting is completely performed and each PE gets a separate part of the search tree. The values of vn and np are reset to 1 *indicating that the current PE works alone on a part of the tree.*

(B) $b = m$

Each PE takes one branch according to its vn. Here also the splitting is completely performed among the all m PEs. The values of vn and np are reset to 1.

(C) $b < m$

Each one of the m PEs that has $vn < b$ will take one branch according to its vn. The values of vn and np for each of these (b-1) PEs are reset to 1.

The PEs having vn that satisfy the condition, $m \geq vn \geq b$, take the right most branch. These PEs can do further split when a new choice point is encountered.

How do these PEs split at the new choice point without communication?

Each PE updates its vn and np after taking the right most branch as follows.

new vn = old vn - b + 1

new np = old np - b + 1, where old np = m.

That is, the (m - b +1) PEs that process the right most branch will get new values of vn and np as follows.

$vn = i$, where $1 \leq i \leq (m - b + 1)$
$np = m - b + 1$

Each of these PEs can do further split when it encounters a new choice point exactly as described above.

From any PE point of view, at any instance of time, the following invariant is always satisfied: *If np > 1, there are no choice points.*

Example:
A system of 7 PEs process a search tree shown in Figure 2 (a). When the system starts up the initial values of vn and np for each PE are shown in Table I.

Table I

PE#	1	2	3	4	5	6	7
vn	1	2	3	4	5	6	7
np	7	7	7	7	7	7	7

After processing the first split, the values of vn and np for each PE are in Table II (Figure 2 (b)).

Table II

PE#	1	2	3	4	5	6	7
vn	1	1	1	2	3	4	5
np	1	1	5	5	5	5	5
Selected branch#	1	2	3	3	3	3	3

The values of vn and np, after PEs "3 - 7" have completed the second split, are in Table III (Figure 2 (c)).

Table III

PE#	3	4	5	6	7
vn	1	1	1	1	2
np	1	1	1	2	2
Selected branch#	1	2	3	4	4

The values of vn and np, after PEs 6 and 7 have completed the third split, are in Table IV

(Figure 2 (d)).

Table IV

PE#	6	7
vn	1	1
np	1	1
Selected branch#	1	2,3,4

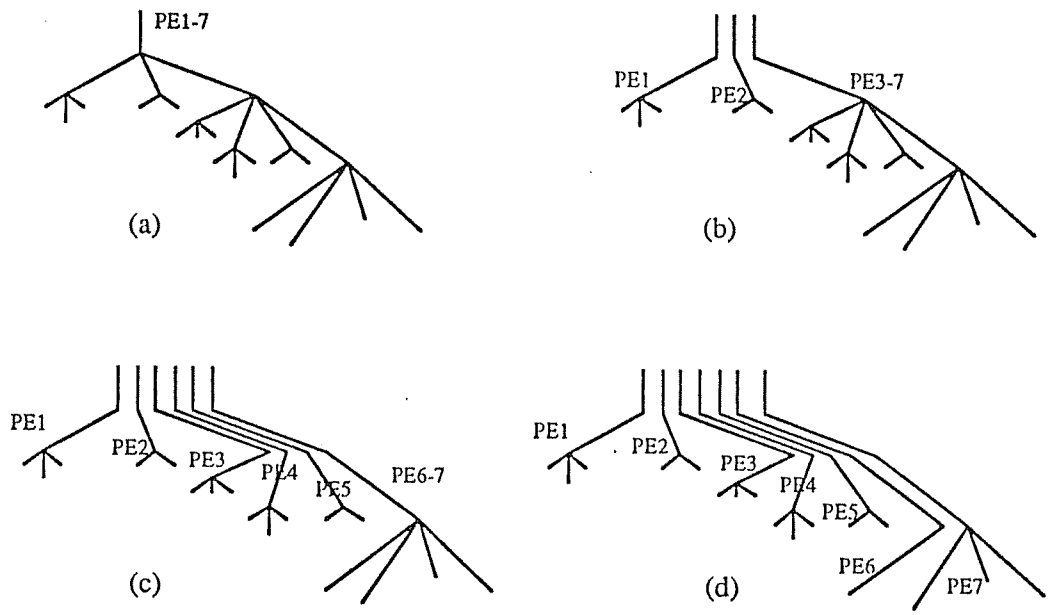


Figure 2: Illustration example of the Right-Biased scheme.

6.1. How Does the Scheme Work on Copying?

When the number of terminated PEs reaches some value, say j, the manager processor selects a busy PE for requesting a part of its environment to be copied into these j PEs' memories. The busy PE can be in one of the following two situations:

1. It works alone on a subtree.
2. It works with other PEs in parallel on a subtree.

Case 1:

In the first case, when the busy PE q copies its environment to the j PEs, q can be in one of the following situations:

(a) has untried branches (i.e. choice points),
(b) has no untried branches so far.

In case (a), the j PEs get unique values for vn from 2 to j+1, and the same value of np which is equal to j + 1. The deepest part of the environment (the initial part of run-time stack and the heap as defined by the earliest choice point), the earliest choice-point frame, along with the whole trail stack are copied. (The trail information is needed by each of the j PEs in order to undo the respective variables in the received environment.) The PE q removes the earliest choice-point frame (i.e. cuts the right branches of the current subtree at the point closest to the root). (The number of untried branches can be kept in one additional field in the respective choice-point frame). Each one of the j PEs backtracks to the earliest choice point. Each PE performs its split according to the number of untried branches in the earliest choice point and the local values of vn and np. Later on (in Section 8), we return to discuss different split strategies that can be used in this case.

In case (b), the PE q assigns 1 to vn and j+1 to np, the j PEs get unique values for vn from 2 to j+1, and all these PEs get the same value of np which is equal to j + 1. The whole environment of q is copied to all the j PEs. All the j+1 PEs process the same branch until a new choice point is encountered.

Case 2:

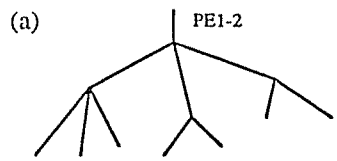
In the second case, there are two possible solutions. The first one requires each PE to know the names of the other PEs that are processing the same subtree. And each time a PE is about to copy to a new group, it communicates with members of the previous group to update the values of vn and np, and to add the members of the new group. This solution requires a centralised mechanism for updating the name of members of each group. The

other drawback of this first solution is the required communication overhead.

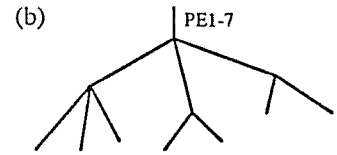
The second solution does not require extra communication overhead, but requires an extra local space. The idea of the second solution is that, when a PE q processes with other PEs a subtree and q is going to copy to a group of j PEs, the j PEs follow q until they arrive to a situation where q and the j PEs work alone on a subtree. In that situation, q can split with the j PEs that subtree. In order to make the j PEs to follow q, each one of the j PEs should get exactly the same state as q (current environment and current values of vn and np). And in order to make q to split with the j PEs, q and the j PEs get values of vn and np exactly as in the case (b). These values are usable only when it comes the situation in which q and the j PEs work alone on a subtree. Therefore, when q copies its current state to the j PEs, each of these PEs gets also new values of vn and np that should be saved somewhere in its local memory. It can happen that q has already copied its state to the j PEs and is going to copy to another group of say i PEs. That is, q copies to a number of groups one after the other. In this case, the i PEs should follow q all the way until it comes the the situation in which q and the i PEs work alone in a subtree. Since q has saved values to vn and np when it has copied to the j PEs, it has to save also a new pair corresponding to the i PEs. The first saved pair should be used before the second one. That is, saved pairs should be used in a first-in-first-out manner.

Therefore, we assume that each PE has a FIFO buffer to save such pairs. The buffer is empty when the current PE works with at most one group of PEs on the same branch (subtree). Generally, if the depth of a FIFO buffer is d (where, $d > 0$), the current PE works with d+1 groups on the same branch.

When a PE q is going to copy its environment into a group of j PEs and q works with other PEs on the same branch, q copies also the contents of the FIFO buffer into the j PEs. Only the last added entry of the FIFO buffer will be different in the j PEs and q. Each time the PE that is going to copy (source) buffers a new pair with $vn = 1$. An example in Figure 3 illustrates how the naming scheme works in the second case using the second solution.

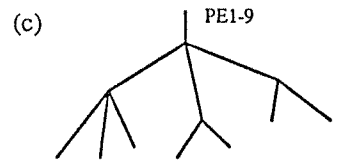


PE#	1	2
vn	1	2
np	2	2
FIFO:	< >	< >



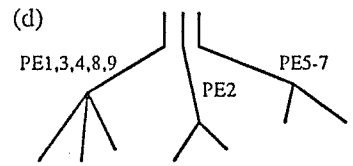
PE1 --> PE3-4
PE2 --> PE5-7

PE#	1	2	3	4	5	6	7
vn	1	2	1	1	2	2	2
np	2	2	2	2	2	2	2
FIFO:	<(1,3)>	<(1,4)>	<(2,3)>	<(3,3)>	<(2,4)>	<(3,4)>	<(4,4)>

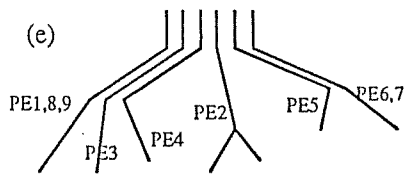


PE1 --> PE8-9

PE#	1	2	3	4	5	6	7	8	9
vn	1	2	1	1	2	2	2	1	1
np	2	2	2	2	2	2	2	2	2
FIFO:	<(1,3), (1,3)>	<(1,4)>	<(2,3)>	<(3,3)>	<(2,4)>	<(3,4)>	<(4,4)>	<(1,3), (2,3)>	<(1,3), (3,3)>



PE#	1	2	3	4	5	6	7	8	9
vn	1	1	2	3	1	2	3	1	1
np	3	1	3	3	3	3	3	3	3
FIFO:	<(1,3)>	< >	< >	< >	< >	< >	< >	<(2,3)>	<(3,3)>



PE#	1	2	3	4	5	6	7	8	9
vn	1	1	1	1	1	1	2	3	3
np	3	1	1	1	1	2	2	3	3
FIFO:	< >	< >	< >	< >	< >	< >	< >	< >	< >

Figure 3: Illustration example for the case in which a PE works with other PEs on a subtree and copies its environment into different groups of PEs at different instances of time.

(a) PE1 and PE2 work in parallel on a tree.

(b) PE1 copies into PE3 and PE4, and PE2 copies into PE5, PE6, and PE7.

(c) PE1 copies into PE8 and PE9.

(d) After PE1-9 have performed the first split.

(e) After PE1 and PE3-9 have performed the second split.

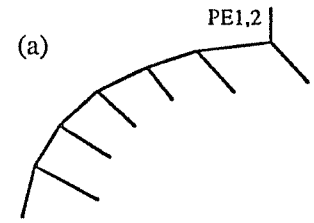
6.2. Problems in the Second Solution

There are two problems due to allowing any PE to copy to any number of groups of PEs and using a FIFO buffer:

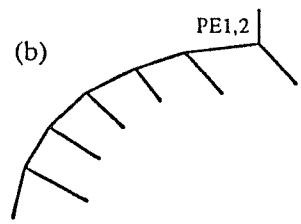
- The size of the buffer can continuously grow.
- The busy PE that is interrupted all the time can not do useful work.

To illustrate this problem we take an example which is shown in Figure 4. To solve the above two problems, we can allow copying only when the depth of the FIFO buffer is less than or equal to a specific value L. By this way, the FIFO buffer has a fixed size L, and each busy PE can proceed and do useful work.

But it is much more acceptable to select for copying a busy PE that has untried branches first, and then a busy PE that has not untried branches so far. And never select a PE that works with other PEs on the same subtree. The reader, who wishes to include the second case or who has a similar problem that requires including the second case, now knows about the problems and the solution. But, we are going to exclude this case in the rest of the paper.

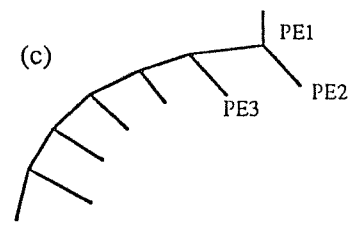


PE#	1	2
vn	1	2
np	2	2
FIFO:	< >	< >



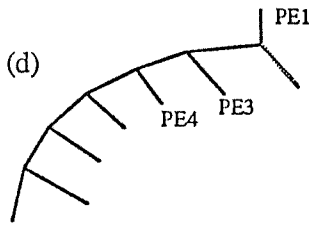
PE1 --> PE3

PE#:	1	2	3
vn:	1	2	1
np:	2	2	2
FIFO:	<(1,2)>	< >	<(2,2)>



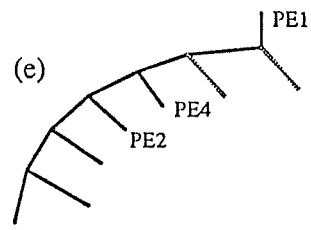
PE1 --> PE4

PE#:	1	2	3	4
vn:	1	1	1	1
np:	2	1	1	2
FIFO:	<(1,2), (1,2)>	< >	< >	<(1,2), (2,2)>



PE1 --> PE2

PE#	1	2	3	4
vn	1	1	1	1
np	2	2	1	1
FIFO:	<(1,2), (1,2), (1,2)>	<(1,2), (1,2), (2,2)>	< >	< >



PE1 --> PE3

PE#:	1	2	3	4
vn:	1	1	1	1
np:	2	2	1	1
FIFO:	<(1,2), (1,2), (1,2), (1,2)>	< >	<(1,2), (1,2), (1,2), (2,2)>	< >

Figure 4: Illustrates the possibility of one PE (PE1) may always copy to the other PEs if we allow a PE that works with other PEs on a subtree to copy its environment. The FIFO buffer increases all the time and PE1 can not process its subtree.

(a) PE1 and PE2 work in parallel on a tree.

(b) PE1 copies its environment into PE3.

(c) PE2 and PE3 process the respective branches, and PE1 copies to PE4.

(d) PE2 is terminated, and PE1 copies to PE2. PE3 and PE4 process the respective branches.

(e) PE3 is terminated, and PE1 copies to PE3. PE2 and PE4 process the respective branches.

6.3. The Right-Biased Algorithm

In this section, the above algorithm is summed up and more formally specified. The algorithm assumes that each PE has the following local information:

1- *Physical-ID*

It contains a number representing the name of the current PE. Each PE has a unique Physical-ID from 1 to n, where n is the number of PEs in the system.

2-*Virtual-Name (vn)*

It contains a number that can be used for selecting a proper branch from the next choice point. This value of vn can be changed after each split.

3- *Number-of-PEs (np)*

It contains the number of PEs that are currently processing the current branch. The value of np can be changed after each split.

Initially,

vn = Physical-ID

np = n

On splitting

The following algorithm is performed by each PE when a choice point is encountered. In the following specification, "b" stands for the number of branches at the current choice point, and "CP frame" stands for Choice-Point frame.

```
BEGIN-SPLIT-ALGORITHM
  IF np > 1                          {Do other PEs work on the current tree?}
  THEN                               {Yes, splitting will be performed}
    IF b > np                        {Is #branches > #PEs}
    THEN IF vn < np                  {Yes, Is it possible to take only one branch?}
         THEN                        {Yes, one branch is taken, np and vn are reset to 1}
           BEGIN
             "Take branch number vn without creating a CP frame"
             np := vn := 1
           END
         ELSE                        {PE of vn = np takes the remaining branches}
           BEGIN
             "Take the right most "b - np + 1" branches and create a CP frame"
             np := vn := 1
           END
    ELSE IF  b = np                  {Is #branches = #PEs}
         THEN                        {Yes, each PE takes only one branch}
           BEGIN
             "Take branch number vn without creating a CP frame"
             np :=  vn := 1
           END
         ELSE                        {b < np}
           IF  vn < b                {Is it possible to take alone one branch?}
           THEN                      {Yes, one branch is taken, np and vn are reset to 1}
             BEGIN
               "Take branch number vn without creating a CP frame"
               np := vn := 1
             END
           ELSE                      {take the left most branch and update np and vn}
             BEGIN
               "Take branch number "b" without creating a CP frame"
               np := np - (b -1)
               vn := vn - (b -1)
             END
END-SPLIT-ALGORITHM
```

On Termination

When a PE has exhausted its search space, it reports itself to the MP as an idle. The MP waits until the number of idle PEs reaches certain limit j, then it selects a busy PE that works alone on a subtree for copying.

On Copying

When one of the busy PEs that works alone on a subtree is interrupted for copying, it either has or has not untried branches.

If the busy PE has not any untried branch, the whole current environment is copied to all the j PEs. The j + 1 PEs get unique values for vn ranging from 1 to j + 1, and all the j + 1 PEs get np equal to j + 1. The situation now for these j + 1 PEs is similar to the situation when the system starts up.

If the busy PE q has untried branches (choice points), the deepest part of the environment as defined by the earliest choice point, say c, the choice point c, along with the whole trail stack are copied to all the j PEs. Each of the j PEs gets a unique value of vn ranging from 2 to j + 1, and all the j PEs get np equal to j + 1. The q takes the left most branch of c. The q has vn = 1 and np = 1. Each of the j PEs backtracks and selects a branch that corresponds to its vn. The values of vn and np will be updated exactly as described for the splitting case.

An algorithm for the balanced strategy is given in Appendix A.

7. Split Operation

Our split schemes guarantee that *no branch can be ignored in any split operation.* The reason is that each split operation satisfies the following invariant:

When a choice point of "b" branches is processed by more than one PE, each of the "b" branches is assigned to at least one of those PEs.

In both schemes, when one branch is assigned to r PEs, where r > 1, each of the r PEs gets a unique (vn, np) pair as follows: (1, r), (2, r), ... , (r, r). This makes each PE know locally that r PEs are going to process the next choice point. And each one has a unique vn making different branches at the next choice point be assigned to different PEs, one branch to a number of PEs, or a number of branches to one PE according to the specified strategy.

The schemes for selection of the proper branch(es) are simple. They require only two local variables for each PE, and simple tests and arithmetic operations.

8. On Copying Revisited

In this section we discuss improvements to the split schemes presented above. The improvement concerns the case in which a source PE q has a number of untried branches and q is going to copy a part of its environment into j idle PEs (i.e., case 1 (a) in Section 6.1). The details of these improvements are found in [9].

In the above schemes the splitting was done at the node which is closest to the root of the current subtree. That is, if q has a number of choice points s, only one out of s is assigned to the j PEs. Here we make q distribute the s choice points to the j PEs and itself. By this

way, each of the j PEs can get more branches than in the above schemes. In what follows we discuss two possible ways to distribute the s choice points to the j PEs and q.

In the first one, the j PEs are partitioned into groups and each group takes one choice point. And q takes the left most branch of the recent choice point. In this case, members of a group of say k PEs, get values of (vn, np) pairs as follows: (1, k), (2, k), ...(k,k). Figure 5 shows how to assign each node to a group of the j PEs. The PE q copies the whole environment to all the j PEs in parallel. Then it sends to each PE of the j PEs the address of the respective choice-point frame and new values of vn and np. Each PE backtracks to the specified choice-point frame, removes the others, and performs split according to the received values of vn and np.

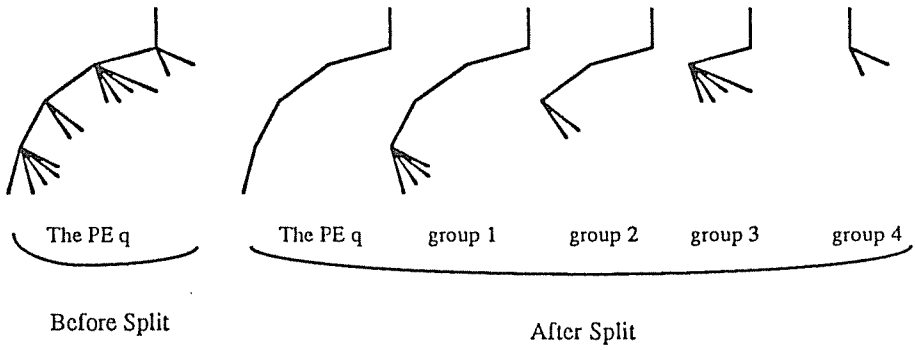


Figure 5: Assigning the available nodes in q to PEs' groups.

The other way of improvement assumes a shared global memory in the system. In this case, when the number of PEs in a group is less than the number of untried branches of the assigned choice point, one branch is taken by each PE and the rest of the branches are accessible to all PEs in the group. A global frame g is created in the global memory containing the number of untried branches and a pointer to the next untried branch. When a PE has finished processing its branch, it backtracks to that choice point and gets the next untried branch, if there is any. The choice point frame has a pointer that points to the frame g. The frame g mainly consists of two basic fields: *#UntriedBranches* and *NextUntriedBranch*. The #UntriedBranches field contains the number of untried branches at that choice point. The NextUntriedBranch contains a pointer to the next untried branch. Figure 6 shows an illustration example. It shows four choice points that are assigned to four groups of PEs. The node b is assigned to a group which consists of two PEs: q and another PE p. The PE q takes the left most (first) branch and p takes the second. When any

of q or p finishes processing its branch, it backtracks to b and from b it can access g and takes the next untried branch. It decrements the number of untried branches by one and updates the contents of the field NextUntriedBranch to point to the next untried branch. Whenever the number of untried branches at g > 0, either p or q can backtrack to b and select the next untried branch.

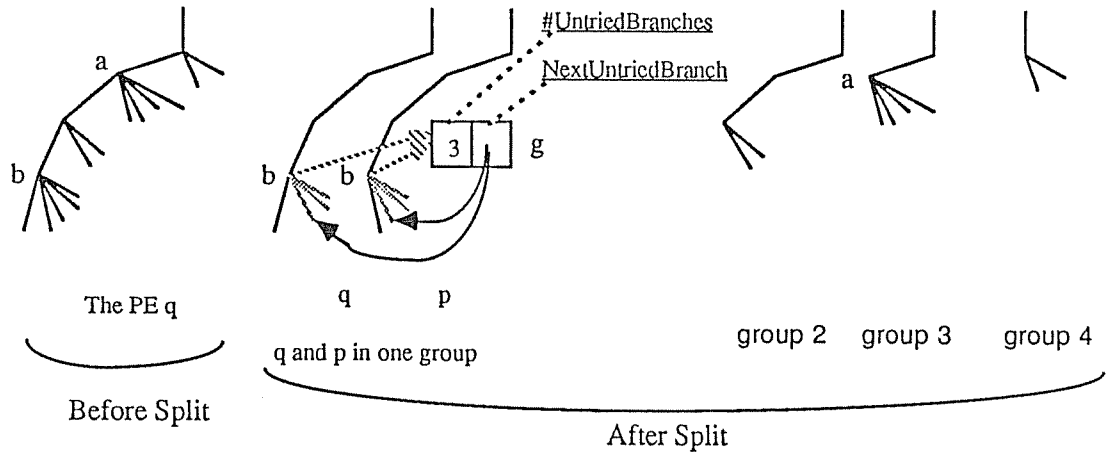


Figure 6: Assigning the available nodes in q to PEs' groups. The node b is processed by q and another PE p. The three untried branches of the node b are accessible to q and p. The same can be done for the node a.

The same can be done to the node "a" if it will be processed by a group of less than 4 PEs and more than one.

If a node of x branches is assigned to a group of y PEs and if $x \leq y$ or $y = 1$, we can use the first improvement described above. The other two nodes of Figure 6 belong to this category.

In the second improvement, before the PE q (i.e. the source) copies its environment to the j idle PE, it creates the required global frames in the global memory and saves their pointers in the respective choice-point frames. The required space for global frames is small: each frame consists of two full words, and the number of frames that exist at the same time is less than the number of PEs in the system [9].

Since the time required by a busy PE to copy is the only overhead in this approach, we made a busy PE copy to a group of idle PEs in parallel, instead of copying to one PE at a time. This reduces the copying time overhead.

9. PE's States

Each PE can be in one of the following states:

1. Idle

It does not process any subtree. Each PE will be in this state when it finishes processing its subtree.

2. Busy

It processes a subtree. The busy PE can be in one of the following substates:

(a) Work alone on a subtree and no choice point is created.
(b) Work alone on a subtree and choice points are created.
(c) Work with other PEs on the same subtree.

(We can have other substates that correspond to substate (c) with FIFO of size 1, 2, ..., etc. But as mentioned before, we will exclude such substates.)

The manager has one entry for each PE. The entry may contain information about the respective PE's status. For example, 2-bit for each entry is enough for coding the above four states: 1, 2(a), 2(b), and 2(c).

One additional entry for each PE can be used to indicate how much load in the respective PE. The load, for instance, can be measured by the number of untried branches or the number of choice points. It is not necessarily for each PE in the state 2(b) to report every change in its load to MP; may be after a reasonable change of its load. (Another additional entry for each PE may be used to indicate the size of information to be copied in the next split operation.)

When the system starts up, all PEs will be in state 2(c). When a PE moves from one state to another, it sends the new state to the manager. The system terminates when all PEs reach the idle state. It should be noticed that each PE communicates with MP only when its state is changed. We expect that for a medium size multiprocessor system (100 PEs) the MP will not be a bottleneck. Later on, decentralisation of the manager's function is discussed for a large system.

10. Reporting Solutions

When only one PE works alone on a subtree and gets a solution, it reports the solution to a specific I/O PE without any problem. But when more than one PE process the same leaf branch producing the same solution, only one of these solutions should be reported. One way to do that is by making each PE report its solution to the I/O PE and the I/O PE takes only the one that comes first and rejects the others. This solution requires a mechanism for making the I/O PE identify each solution.

Another way is to make only one of these PEs report the solution; the one with minimum value of vn (i.e. with vn = 1). We prefer this solution because it is much easier and more efficient to implement than the first one. According to our naming scheme each PE can locally decide (by using the current values of vn and np) if it is going to report the solution or not. For example, if three PEs process in parallel the same branch and having the following values of (vn, np): (1,3), (2,3), and (3,3), the one with vn = 1 will report the solution if that branch has a solution. The other two PEs will not report their solutions, each of them has the value of vn > 1.

11. How to Treat Sequential-OR

Every node mentioned before is assumed a parallel-OR. Branches of a parallel-OR node can be processed by different PEs, while branches of a sequential-OR must be processed by only one PE. We assume the user can annotate short branches in order to be processed sequentially or the compiler can detect such branches and generate a code that makes the run-time system distinguish between sequential-OR and parallel-OR nodes.

Assume a tree with a mixture of parallel-/sequential-OR nodes as shown in Figure 7 (a). We are going to answer to the following questions:

- How does one PE process all branches of a sequential-OR node?
- How does each of the other PEs remove such a node from its list of choice-point frames?
- How each solution is reported only once.

When a number of PEs process in parallel a part of the tree that starts with sequential-OR nodes (Figure 7 (a)), all these PEs search the same short branches until reaching a parallel-OR node. We make only one of these PEs, for instance, the one of vn = 1, be responsible for the sequential-OR nodes that have been created so far. It reports solutions due to branches of these sequential-OR nodes. The other PEs try to find the next

parallel-OR node to select the respective subtree. At that node, each one of these PEs (except the one with vn = 1) removes all choice point frames that are created so far for these sequential-OR nodes. Figure 7 shows an illustration example.

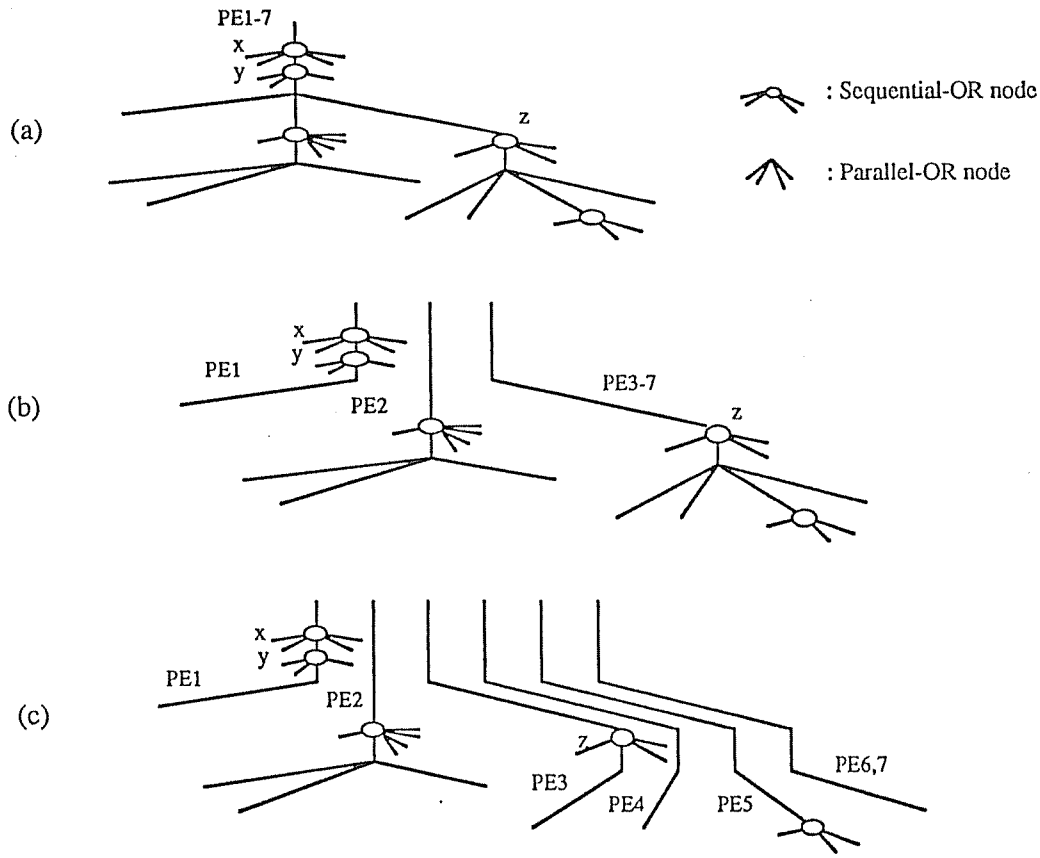


Figure 7: Illustrates how to treat sequential-OR nodes.

(a) Before any split, PE1 has vn = 1, and the other PEs have vn > 1.

(b) After the first split, the first two sequential-OR nodes, x and y, are kept in PE1 and removed from the other PEs. PE3 resets vn to 1, and each of PE4-7 resets its vn to a value greater than 1.

(c) After the second split, s is kept in PE3 and removed from PE4-7.

12. Cuts, Bagof, and Negation as Failure

These three features can be implemented in our method as follows.

(1) Cuts

Suppose that the clauses in a given predicate p contain at least one Cut. These clauses are partitioned according to the textual order into groups G1, G2, ... Gn according to the following rule:

"each group is either a set of clauses in which none of the clauses contains Cut, or a set of clauses in which each clause contains at least one Cut".

Let us first consider the case in which G1, G3, ... have the clauses which contain Cuts, and G2, G4, ... have the clauses which do not contain Cut.

i) When a call is made to p, no split is performed until either the last Cut in any clause of G1 is executed, or the first clause of G2 is about to be executed.

ii) In the latter case, the clauses of G2 can be executed in parallel while the remaining clauses (in G3, G4, ... Gn) will be executed sequentially until either the last Cut in any clause of G3 is executed, or the first clause of G4 is about to be executed, and so on.

The other case, in which G1, G3, ... have clauses containing no Cut. When a call is made to p, we get a situation similar to ii) where G2 is replaced by G1, G3 by G2, and G4 by G3.

That is, a subtree may be processed in sequential mode or in parallel mode depending on if its current left most branch contains Cut or not respectively.

(2) Bagof

Two possible ways to implement bagof in our method are as follows.

a) PEs executing bagof will not perform split until the bagof is done.

b) Only when a PE q is processing a bagof and is going to copy to a group of j PEs, q assigns a part of bagof job to each of the j PEs. When a PE out of the j PEs finishes processing its job, it sends to q the solutions, restores the state to that valid after the bagof call but before producing the results of bagof call, and waits for q. When q collects all solutions from the j PEs, it sends these solutions to the j PEs in parallel. At this point q and

the j PEs have the same environment. They proceed again according to our method.

Variants of solution b) are possible. For instance, instead of making q copy to only one group, we can make it more general by allowing members of the j PEs copy to other groups and members of the new group copy to others and so on. In this case q represents the higher master of these groups. When a member of any group finishes its job, it sends the solutions to its master, restores the state, and waits. When q collects all solutions of bagof, it copies the new solutions to members of its group. Each member that receives solutions and is a group master, forwards these solutions to its members.

(3) Negation as Failure

The same two ways for implementing bagof are also applicable here. That is either restricting the parallelism or allowing parallelism only until copying. The former is straight forward. In the the latter case, when one PE succeeds, not(X) fails and so all PEs involving in execution of not(X). When all fail, not(X) succeeds so all PEs involving in the execution of not(X) succeed.

13. A Large System

For a system having a large number of PEs (a few thousand PEs), the manager and the other global resources are a bottleneck. Figure 8 shows a system organisation for a large machine. The machine consists of a number of clusters. Each cluster has its local manager, networks, and possibly a global memory. Each cluster is managed by its local manager. The main manager manages the local managers.

Each local manager keeps the status information about its PEs only. When PEs in a cluster become idle, the local manager allocates them to one of the busy PEs in the same cluster. When all PEs in the same cluster become idle, the respective manager reports that to the main manager. The main manager selects a local manager that needs help, if there is any. When a local manager is selected, one of the busy PEs in that cluster is selected and its environment is copied to all PEs in the other cluster via the inter cluster copying network. When the main manager detects the situation in which no local manager has a busy PE, the system is terminated.

The function of components in each cluster is the same as described above (in Section 3). The main manager keeps information about each local manager. For instance, which cluster is busy and which is not, and how many busy or idle PEs there are in each cluster. If a

global memory is used (see Section 8), this memory is accessible only to PEs in the same cluster.

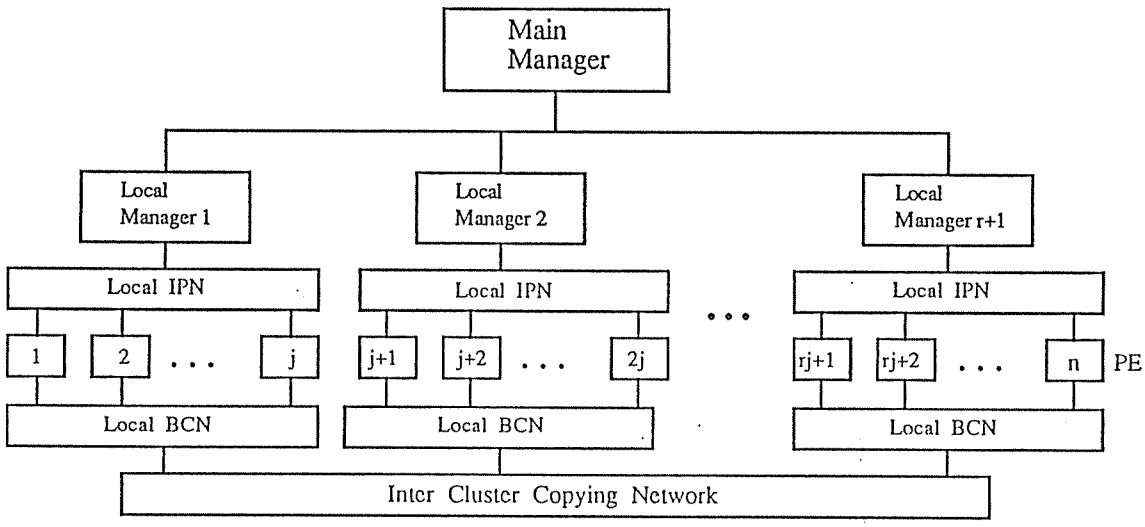


Figure 8: System configuration of a large machine.

Here also all PEs in the system start processing the top-level query in parallel. Each PE selects a subtree for processing as described above. Each PE reports its status information to its local manager. Terminated PEs in a cluster are allocated to help busy PEs in the same cluster until all PEs in the same cluster become idle. Then this cluster can get a new job from a busy PE in another cluster.

14. Conclusion and Discussion

We have presented a method for OR-parallel execution of Prolog on a multiprocessor system. The method can utilise most of the technology devised for the sequential implementation of Prolog. The complex part of the proposed architecture is the broadcast copying network(s). This network can be a simplified version of the broadcast network designed for our BC-machine [11]. It is a circuit-switched crossbar network that can support a sustained transfer rate between PEs excess of 10 Mbytes/sec. on each channel, which is the maximum speed a powerful processor, like Motorola's 68020, can move information from its local memory to the network.

Our method gives also performance improvement to each PE. Even in the optimised sequential implementation of Prolog [10], when execution comes to a choice point, a choice-point frame is created and the state of the computation is saved in the frame. Furthermore, modifications to the previous environment have to be saved (in the trail stack). In our method, some such overhead is reduced as follows. When a subtree is processed by j PEs and j is greater than or equal to the number of branches at the next choice point, each PE will take one branch. In this situation, no choice-point frame is created and the corresponding trail information will not be saved. That is, whenever we have this situation, such overhead is reduced to zero.

A simple implementation of our method is to exclude copying. In this case, the method can be implemented on a system having a very large number of PEs with neither copying network(s) nor manager(s). All PEs start processing the top-level query in parallel. Each PE selects a subtree for processing according to the selected split scheme (one of the presented schemes or a new one that works better for a specific class of programs). When a PE finishes processing the subtree, it never gets a new job. Since there is no communication overhead and each PE has the same performance as the optimised sequential machine, the speedup of the system over the sequential machine depends only on the degree of parallelism in the program.

In comparison with the work reported in [2, 3], which is directly related to our work presented here, their approach allows only one PE to process the top-level query and copy to one PE at a time at each split. While in our approach, all PEs in the system start in parallel to process the top-level query and each PE selects a proper branch without any copying or communication. Our approach allows one PE to copy (a part of) its environment to a group of idle PEs in parallel in order to split with those PEs in short time. In our approach some choice points may not need to be created at all, and handling sequential execution is simple. Our approach is also capable of utilising split strategies that reduce

copying.

In our method, the Manager Processor waits until some number k of PEs are idle and there is a busy PE that processes alone a subtree before migrating branches of the subtree to them. This could lead to some processors being idle most of the time. A study of the frequency/cost of copying and correct value of k for typical programs needs to be done. Also, the issue of which choice points are selected for OR-parallel execution needs to be investigated.

It is planned to evaluate the method and the ideas presented here on our BC-machine which is currently being built in the laboratory of logic programming systems at SICS [9, 11].

Acknowledgements

The author would like to thank Seif Haridi, Lars-Erik Thorelli, Roland Karlsson, and Lennart E. Fahlén for their valuable comments and observations. Also thanks to the referees for their interesting suggestions for improving the paper.

Appendix A. An Algorithm for the Balanced Strategy

The same naming scheme described in the right-biased algorithm (Section 6.3) is used here. The values of vn and np for each PE are updated according to the following scheme:

- When m PEs are processing in parallel a subtree starting from a common root, each PE of these m get a unique pair (vn, np) as follows.

$vn = i$, where $1 \leq i \leq m$
$np = m$

- When any PE of these m comes to a choice point of say b branches, one of the following situations may occur:

(A) $b > m$,
(B) $b = m$, and
(C) $b < m$.

The cases (A) and (B) can be exactly the same as in the right-biased algorithm. In case (C), the first branch is assigned to PEs that have vn of value: 1, b+1, 2b+1, 3b+1, etc. The second branch is assigned to PEs that have vn of value: 2, b+2, 2b+2, 3b+2, etc. And so on for the rest of these b branches.

When a branch is selected by only one PE, the values of vn and np of this PE become 1. When a branch can be selected by v PEs, the new values of (vn, np) of these PEs will be: (1, v), (2, v), ..., (v-1, v), (v, v). These v PEs can do further split when another new choice point on the selected is encountered.

Formally we define the selected branch number and values of vn and np for each PE as follows. At a choice point s of b branches, the values of vn and np can be expressed in terms of b by the following two equations:

$np = y * b + x$ where, $y \geq 1, 1 \leq x \leq b$ ..... (1)
$vn = z * b + w$ where, $z \geq 0, 1 \leq w \leq b$ ..... (2)

All PEs that process the choice point s have the same value of np, but different values of vn ranging from 1 to np.

Selected Branch

Each PE selects only one branch of the b branches at point s. According to equation number (2), the branch number w is selected by the current PE. It should be noted that different PEs have different values of vn at point s. But different PEs with different values of vn can have the same value of w.

Updating vn and np

Referring to the equations (1) and (2), the new values of vn and np are updated as follows.

$$\text{new vn} = z+1$$

$$\text{new np} = \begin{cases} y & \text{when } x < w \\ y+1 & \text{otherwise} \end{cases}$$

Example

Figure 9 shows an illustration example of the Balanced scheme. Table V shows values of vn and np before and after the first split, and selected branch by each PE (see Figure 9 (b)).

Table V

PE#:	1	2	3	4	5	6	7	8
current value of vn:	1	2	3	4	5	6	7	8
current value of np:	8	8	8	8	8	8	8	8
selected branch#:	1	2	3	1	2	3	1	2
new value of vn:	1	1	1	2	2	2	3	3
new value of np:	3	3	2	3	3	2	3	3

After the second split, the selected branches by each PE are shown in Figure 9 (c) and the new values of vn and np are shown in Table VI.

Table VI

PE#:	1	2	3	4	5	6	7	8
new value of vn :	1	1	1	1	1	1	1	1
new value of np :	1	1	1	1	1	1	1	1

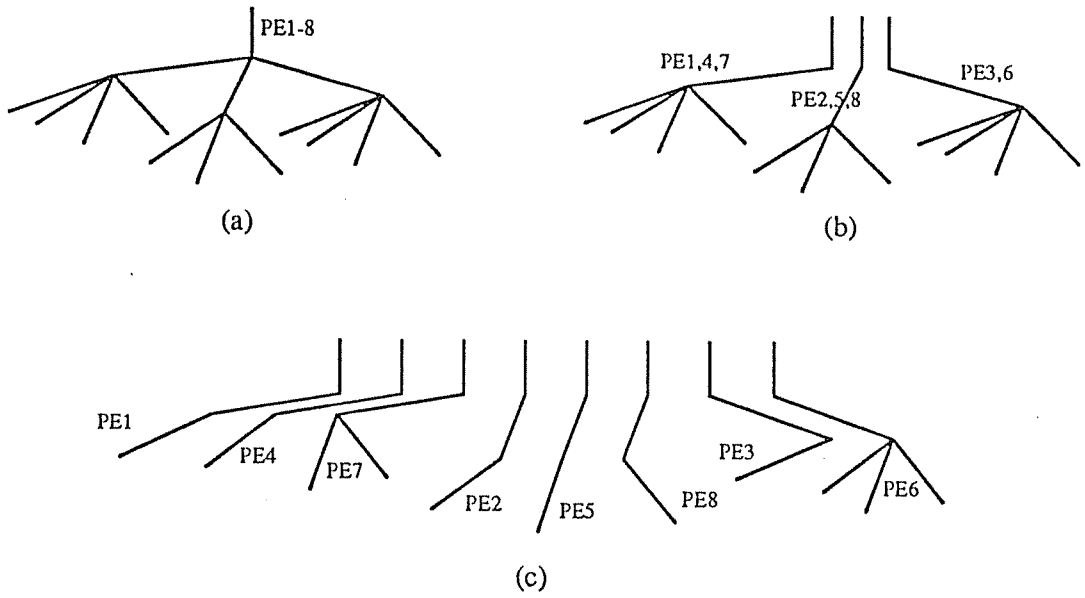


Figure 9: Illustration example of the Balanced scheme.

(a) PE1-8 work in parallel on a tree.

(b) After all the PEs have been performed the first split.

(c) After all the PEs have been performed the second split.

A.1. The Balanced Algorithm

In this algorithm, we assume the same local variables and the initial state as in the right-biased algorithm.

On splitting

The following algorithm is performed by each PE when a choice point is encountered.

```
BEGIN-SPLIT-ALGORITHM
  IF np > 1                          {Do other PEs work on the current tree?}
  THEN                               {Yes, splitting will be performed}
    IF b > np
    THEN IF vn < np
         THEN                        {Yes, one branch is taken, np and vn are reset to 1}
           BEGIN
             "Take branch number vn without creating a CP frame"
             np := vn := 1
           END
         ELSE                        {take the remaining branches}
           BEGIN
             "Take the right most "b - np + 1" branches and create a CP frame"
             np := vn := 1
           END
    ELSE IF  b = np
         THEN                        {Yes, one branch is taken, np and vn are reset to 1}
           BEGIN
             "Take branch number vn without creating a CP frame"
             np :=  vn := 1
           END
         ELSE                        {b < np}
           BEGIN
             "Take branch no. "(vn - 1) MOD b + 1" without creating a CP frame"
             y := np DIV b
             x := (np -1) MOD b + 1
             IF y = 1 AND x < w
                                     {Is there no another PE which will take the same branch?}
             THEN vn := np := 1
             ELSE
               BEGIN
                    z := vn DIV b
                 w := (vn -1) MOD b + 1
                   IF y > 1 AND x < w
                 THEN np := y
                 ELSE np := y + 1
                 vn := z + 1
               END
           END
END-SPLIT-ALGORITHM
```

On Termination

Exactly as in the right-biased algorithm.

On Copying

It can be exactly the same as the right-biased scheme except the case at which the source PE q has choice-point frames. In this case, q takes the left most branch of the earliest choice-point and the right branches to be assigned to the j PEs. The q decrements first b by one in the earliest choice point, and copies the deepest part of the environment as defined by the earliest choice point, the earliest choice point, along with the whole trail stack to all the j PEs. Each of the j PEs gets a unique value of vn ranging from 1 to j, and all the j PEs get np equal to j. The q removes the earliest choice point frame and resets vn and np to 1. Each of the j PEs backtracks and selects branches according to np, vn and the new b. The values of vn and np will be updated exactly as described above for the splitting case.

References

[1] J. S. Conery and D. F. Kibler, "Parallel Interpretation of Logic Programs", in Proc. of the ACM Conference on Functional Programming Languages and Computer Architecture, pp. 163-170, October 1981.

[2] H. Yasuhara and K. Nitadori, "ORBIT: A Parallel Computing Model of Prolog", New Generation Computing, Vol 2, pp. 277-288, 1984.

[3] Y. Sohma, K. Satoh, K. Kumon, H Masuzawa, A. Itashiki, "A New Parallel Inference Mechanism Based on Sequential Processing," Proc. of Working Conf. Fifth Generation Computer Architecture, Manchester, July 1985.

[4] A. Ciepielewski and S. Haridi, "A Formal Model for OR-Parallel Execution of Logic Programs", in Proc. Inform. Processing 83, pp. 299-305, 1983.

[5] P. Borgwardt, "Parallel Prolog Using Stack Segments on Shared-Memory Multiprocessors", in Proc. 1984 Int. Symp. Logic Programming, pp. 2-11, February 1984.

[6] G. Lindstrom, "Or-parallelism on Applicative Architectures", in Proc. Second Int. Logic Programming Conf., pp. 159-170, July 1984.

[7] J. Crammond, "A Comparative Study of Unification Algorithms for OR-Parallel Execution of Logic Programs", IEEE Trans. on Computers, Vol. c-34, no. 10, pp. 911-917, October 1985.

[8] K. A. M. Ali, "Pool Machine: A multiprocessor Architecture for OR-Parallel Execution of Logic Programs", Rep. TRITA-CS-8603, The Royal Institute of Technology, Stockholm, October 1985.

[9] K. A. M. Ali, "Architectures for OR-Parallel Execution of Prolog", SICS, Working Paper July 1986.

[10] D. H. D. Warren, "An Abstract Prolog Instruction Set", Technical Note 309, SRI International, Menlo Park, CA, October 1983.

[11] L. E. Fahlén, "The BC-Machine Prototype, Architecture and Interconnection Network", SICS, Working Paper July 1986.